

目 录

译者序

前言

第一部分 Linux 内核源代码

arch/i386/kernel/entry.S	2	include/asm-i386/spinlock.h	133
arch/i386/kernel/init_task.c	8	include/asm-i386/system.h	137
arch/i386/kernel/irq.c	8	include/asm-i386/uaccess.h	139
arch/i386/kernel/irq.h	19	include/linux/binfmts.h	146
arch/i386/kernel/process.c	22	include/linux/capability.h	147
arch/i386/kernel/signal.c	30	include/linux/elf.h	150
arch/i386/kernel/smp.c	38	include/linux/elfcore.h	156
arch/i386/kernel/time.c	58	include/linux/interrupt.h	157
arch/i386/kernel/traps.c	65	include/linux/kernel.h	158
arch/i386/lib/delay.c	73	include/linux/kernel_stat.h	159
arch/i386/mm/fault.c	74	include/linux/limits.h	160
arch/i386/mm/init.c	76	include/linux/mm.h	160
fs/binfmt-elf.c	82	include/linux/module.h	164
fs/binfmt_java.c	96	include/linux/msg.h	168
fs/exec.c	98	include/linux/personality.h	169
include/asm-generic/smplock.h	107	include/linux/reboot.h	169
include/asm-i386/atomic.h	108	include/linux/resource.h	170
include/asm-i386/current.h	109	include/linux/sched.h	171
include/asm-i386/dma.h	109	include/linux/sem.h	179
include/asm-i386/elf.h	113	include/linux/shm.h	180
include/asm-i386/hardirq.h	114	include/linux/signal.h	181
include/asm-i386/page.h	114	include/linux/slab.h	184
include/asm-i386/pgtable.h	115	include/linux/smp.h	184
include/asm-i386/ptrace.h	122	include/linux/smp_lock.h	185
include/asm-i386/semaphore.h	123	include/linux/swap.h	185
include/asm-i386/shmparam.h	124	include/linux/swapctl.h	187
include/asm-i386/sigcontext.h	125	include/linux/sysctl.h	188
include/asm-i386/siginfo.h	125	include/linux/tasks.h	194
include/asm-i386/signal.h	127	include/linux/time.h	194
include/asm-i386/smp.h	130	include/linux/timer.h	195
include/asm-i386/softirq.h	132	include/linux/times.h	196
		include/linux/tqueue.h	196
		include/linux/wait.h	198
		init/main.c	198
		init/version.c	212

ipc/msg.c	213
ipc/sem.c	218
ipc/shm.c	227
ipc/util.c	236
kernel/capability.c	237
kernel/dma.c	240
kernel/exec_domain.c	241
kernel/exit.c	242
kernel/fork.c	248
kernel/info.c	255
kernel/itimer.c	255
kernel/kmod.c	257
kernel/module.c	259
kernel/panic.c	270
kernel/printk.c	271
kernel/sched.c	275
kernel/signal.c	295
kernel/softirq.c	307
kernel/sys.c	307
kernel/sysctl.c	318
kernel/time.c	330
mm/memory.c	335
mm/mlock.c	345
mm/mmap.c	348
mm/mprotect.c	358
mm/mremap.c	361
mm/page_alloc.c	363
mm/page_io.c	368
mm/slab.c	372
mm/swap.c	394
mm/swap_state.c	395
mm/swapfile.c	398
mm/vmalloc.c	406
mm/vmscan.c	409

第二部分 Linux 内核源代码分析

第1章 Linux简介	416
1.1 Linux和Unix的简明历史	416
1.2 GNU通用公共许可证	418
1.3 Linux开发过程	419

第2章 代码初识	421
2.1 Linux内核源程序的部分特点	421
2.1.1 gcc特性的使用	421
2.1.2 内核代码习惯用语	422
2.1.3 减少#ifdef和#endif的使用	423
2.2 代码样例	424
2.2.1 printk	424
2.2.2 等待队列	429
2.2.3 内核模块	432
2.3 配置与编译内核	434
2.3.1 配置内核	434
2.3.2 构建内核	436
2.3.3 备份的重要性	436
2.3.4 发布你的改进	437
第3章 内核体系结构概述	439
3.1 内核设计目标	439
3.1.1 清晰性	439
3.1.2 兼容性	439
3.1.3 可移植性	440
3.1.4 健壮性和安全性	440
3.1.5 速度	441
3.2 内核体系结构初识	441
3.3 内核体系结构的深入了解	442
3.4 Linux内核的类型	444
3.5 设计和实现的关系	446
3.5.1 内核源程序目录结构	446
3.5.2 体系结构相关和体系结构无关的代码	450
第4章 系统初始化	451
4.1 引导PC机	451
4.2 初始化Linux内核	452
4.2.1 BogoMIPS	455
4.2.2 分析内核选项	456
4.3 init	459
第5章 系统调用	462
5.1 什么是系统调用	462
5.2 如何激活系统调用	463
5.2.1 system_call	464
5.2.2 lcall7	468

5.3 系统调用样例	469	7.13.2 wait	532
第6章 信号、中断和时间	474	第8章 内存	535
6.1 锁的概述	474	8.1 虚拟内存	535
6.2 信号	474	8.1.1 交换和分页	536
6.2.1 数据结构	475	8.1.2 地址空间	537
6.2.2 应用函数	476	8.1.3 内存管理单元	537
6.2.3 传送信号	480	8.1.4 页目录和页表	538
6.2.4 其他有关信号的函数	489	8.1.5 转换后备缓存	540
6.2.5 内核如何区分实时信号和非 实时信号	491	8.1.6 段	540
6.3 中断	492	8.2 进程的内存组织	541
6.3.1 中断请求: IRQ	492	8.2.1 struct vm_area_struct	541
6.3.2 下半部分	493	8.2.2 struct vm_operations_struct	542
6.3.3 数据结构	493	8.2.3 struct mm_struct	542
6.3.4 操作和IRQ	496	8.2.4 VMA的操作	542
6.3.5 硬件中断处理程序和下半部分	499	8.3 分页	544
6.4 时间	502	8.3.1 页面保护详述	544
第7章 进程和线程	505	8.3.2 写拷贝	545
7.1 调度和时间片	505	8.3.3 页面错误	546
7.2 实时进程	506	8.3.4 页面调出	551
7.3 优先级	506	8.4 交换设备	552
7.4 进程ID: PID	506	8.5 内存映射mmap	556
7.5 引用计数	506	8.6 用户空间和内核空间的动态内存	560
7.6 权能	507	8.6.1 brk	561
7.7 进程在内核中是如何表示的	508	8.6.2 vmalloc和vfree	562
7.8 进程来源: fork和_clone	511	8.7 主存储器信息转储	565
7.9 运行新程序	514	第9章 System V IPC	568
7.10 可执行格式	517	9.1 消息队列	568
7.11 调度及它们是如何运行的	519	9.2 信号量	581
7.11.1 调度函数和调度策略	519	9.3 共享内存	590
7.11.2 计算goodness值	522	第10章 对称多处理	596
7.11.3 非实时优先级	523	10.1 并行程序设计概念及其原语	597
7.11.4 实时优先级	525	10.1.1 原子操作	597
7.12 遵守限制	526	10.1.2 test-and-set	599
7.12.1 权能	526	10.1.3 信号量	600
7.12.2 用户ID和组ID	529	10.1.4 自旋锁	604
7.12.3 资源限制	530	10.2 APIC和CPU-To-CPU通信	607
7.13 进程的结束	530	10.3 SMP支持如何影响内核	607
7.13.1 exit	530	10.3.1 对调度的影响	607
		10.3.2 smp_local_timer_interrupt	610

10.3.3 lock_kernel和unlock_kernel	611	11.1 /proc/sys 支持	616
10.3.4 softirq_trylock	612	11.2 sysctl系统调用	621
10.3.5 cli和sti	612	附录A Linux 2.4	627
10.3.6 irq_enter和irq_exit	613	附录B GNU通用公共许可证	629
第11章 可调内核参数	614	附录C 光盘上的内容及系统需求	634

第一部分 Linux 内核源代码

arch/i386/kernel/entry.S

```
1 /*
2 * linux/arch/i386/entry.S
3 *
4 * Copyright (C) 1991, 1992 Linus Torvalds
5 */
6
7 /*
8 * entry.S contains the system-call and fault low-level
9 * handling routines. This also contains the
10 * timer-interrupt handler, as well as all interrupts and
11 * faults that can result in a task-switch.
12 *
13 * NOTE: This code handles signal-recognition, which
14 * happens every time after a timer-interrupt and after
15 * each system call.
16 *
17 * I changed all the .align's to 4 (16 byte alignment),
18 * as that's faster on a 486.
19 *
20 * Stack layout in 'ret_from_system_call':
21 * ptrace needs to have all regs on the stack.
22 * if the order here is changed, it needs to be
23 * updated in fork.c:copy_process,
24 * signal.c:do_signal, ptrace.c and ptrace.h
25 *
26 * 0(%esp) - %ebx
27 * 4(%esp) - %ecx
28 * 8(%esp) - %edx
29 * C(%esp) - %esi
30 * 10(%esp) - %edi
31 * 14(%esp) - %ebp
32 * 18(%esp) - %eax
33 * 1C(%esp) - %ds
34 * 20(%esp) - %es
35 * 24(%esp) - orig_eax
36 * 28(%esp) - %eip
37 * 2C(%esp) - %cs
38 * 30(%esp) - %eflags
39 * 34(%esp) - %oldesp
40 * 38(%esp) - %oldss
41 *
42 * "current" is in register %ebx during any slow entries.
43 */
44
45 #include <linux/sys.h>
46 #include <linux/linkage.h>
47 #include <asm/segment.h>
```

```
48 #define ASSEMBLY
49 #include <asm/smp.h>
50
51 EBX           = 0x00
52 ECX           = 0x04
53 EDX           = 0x08
54 ESI           = 0x0C
55 EDI           = 0x10
56 EBP           = 0x14
57 EAX           = 0x18
58 DS           = 0x1C
59 ES           = 0x20
60 ORIG_EAX     = 0x24
61 EIP           = 0x28
62 CS           = 0x2C
63 EFLAGS       = 0x30
64 OLDESP       = 0x34
65 OLDSS        = 0x38
66
67 CF_MASK      = 0x00000001
68 IF_MASK      = 0x00000200
69 NT_MASK      = 0x00004000
70 VM_MASK      = 0x00020000
71
72 /*
73 * these are offsets into the task-struct.
74 */
75 state        = 0
76 flags        = 4
77 sigpending   = 8
78 addr_limit   = 12
79 exec_domain  = 16
80 need_resched = 20
81
82 ENOSYS = 38
83
84
85 #define SAVE_ALL
86 cld;
87 pushl %es;
88 pushl %ds;
89 pushl %eax;
90 pushl %ebp;
91 pushl %edi;
92 pushl %esi;
93 pushl %edx;
94 pushl %ecx;
95 pushl %ebx;
```

```

96 movl $(__KERNEL_DS),%edx; \
97 movl %dx,%ds; \
98 movl %dx,%es; \
99 \
100 #define RESTORE_ALL \
101 popl %ebx; \
102 popl %ecx; \
103 popl %edx; \
104 popl %esi; \
105 popl %edi; \
106 popl %ebp; \
107 popl %eax; \
108 1: popl %ds; \
109 2: popl %es; \
110 addl $4,%esp; \
111 3: iret; \
112 .section .fixup,"ax"; \
113 4: movl $0,(%esp); \
114 jmp 1b; \
115 5: movl $0,(%esp); \
116 jmp 2b; \
117 6: pushl %ss; \
118 popl %ds; \
119 pushl %ss; \
120 popl %es; \
121 pushl $11; \
122 call do_exit; \
123 .previous; \
124 .section __ex_table,"a"; \
125 .align 4; \
126 .long 1b,4b; \
127 .long 2b,5b; \
128 .long 3b,6b; \
129 .previous \
130 \
131 #define GET_CURRENT(reg) \
132 movl %esp, reg; \
133 andl $-8192, reg; \
134 \
135 ENTRY(lcall7) \
136 pushfl # We get a different stack layout with call \
137 pushl %eax # gates, which has to be cleaned up later.. \
138 SAVE_ALL \
139 movl EIP(%esp),%eax # this is eflags, not eip.. \
140 movl CS(%esp),%edx # this is eip.. \
141 movl EFLAGS(%esp),%ecx # and this is cs.. \
142 movl %eax,EFLAGS(%esp) # \
143 movl %edx,EIP(%esp) # move to their "normal" places

```

```

144 movl %ecx,CS(%esp) # \
145 movl %esp,%ebx \
146 pushl %ebx \
147 andl $-8192,%ebx # GET_CURRENT \
148 movl exec_domain(%ebx),%edx # Get the execution domain \
149 movl 4(%edx),%edx # Get lcall7 handler for domain \
150 call *%edx \
151 popl %eax \
152 jmp ret_from_sys_call \
153 \
154 \
155 ALIGN \
156 .globl ret_from_fork \
157 ret_from_fork: \
158 #ifdef __SMP__ \
159 call SYMBOL_NAME(schedule_tail) \
160 #endif /* __SMP__ */ \
161 GET_CURRENT(%ebx) \
162 jmp ret_from_sys_call \
163 \
164 /* \
165 * Return to user mode is not as complex as all this \
166 * looks, but we want the default path for a system call \
167 * return to go as quickly as possible which is why some \
168 * of this is less clear than it otherwise should be. \
169 */ \
170 \
171 ENTRY(system_call) \
172 pushl %eax # save orig_eax \
173 SAVE_ALL \
174 GET_CURRENT(%ebx) \
175 cmpl $(NR_syscalls),%eax \
176 jae badsys \
177 testb $0x20,flags(%ebx) # PF_TRACESYS \
178 jne tracesys \
179 call *SYMBOL_NAME(sys_call_table)(,%eax,4) \
180 movl %eax,EAX(%esp) # save the return value \
181 ALIGN \
182 .globl ret_from_sys_call \
183 .globl ret_from_intr \
184 ret_from_sys_call: \
185 movl SYMBOL_NAME(bh_mask),%eax \
186 andl SYMBOL_NAME(bh_active),%eax \
187 jne handle_bottom_half \
188 ret_with_reschedule: \
189 cmpl $0,need_resched(%ebx) \
190 jne reschedule \
191 cmpl $0,sigpending(%ebx)

```

```

192 jne signal_return
193 restore_all:
194 RESTORE_ALL
195
196 ALIGN
197 signal_return:
198 sti # we can get here from an interrupt handler
199 testl $(VM_MASK),EFLAGS(%esp)
200 movl %esp,%eax
201 jne v86_signal_return
202 xorl %edx,%edx
203 call SYMBOL_NAME(do_signal)
204 jmp restore_all
205
206 ALIGN
207 v86_signal_return:
208 call SYMBOL_NAME(save_v86_state)
209 movl %eax,%esp
210 xorl %edx,%edx
211 call SYMBOL_NAME(do_signal)
212 jmp restore_all
213
214 ALIGN
215 tracesys:
216 movl $-ENOSYS,EAX(%esp)
217 call SYMBOL_NAME(syscall_trace)
218 movl ORIG_EAX(%esp),%eax
219 call *SYMBOL_NAME(sys_call_table)(,%eax,4)
220 movl %eax,EAX(%esp) # save the return value
221 call SYMBOL_NAME(syscall_trace)
222 jmp ret_from_sys_call
223 badsys:
224 movl $-ENOSYS,EAX(%esp)
225 jmp ret_from_sys_call
226
227 ALIGN
228 ret_from_exception:
229 movl SYMBOL_NAME(bh_mask),%eax
230 andl SYMBOL_NAME(bh_active),%eax
231 jne handle_bottom_half
232 ALIGN
233 ret_from_intr:
234 GET_CURRENT(%ebx)
235 movl EFLAGS(%esp),%eax # mix EFLAGS and CS
236 movb CS(%esp),%al
237 testl $(VM_MASK | 3),%eax # rtn to VM86 mode|non-super?
238 jne ret_with_reschedule
239 jmp restore_all

```

```

240
241 ALIGN
242 handle_bottom_half:
243 call SYMBOL_NAME(do_bottom_half)
244 jmp ret_from_intr
245
246 ALIGN
247 reschedule:
248 call SYMBOL_NAME(schedule) # test
249 jmp ret_from_sys_call
250
251 ENTRY(divide_error)
252 pushl $0 # no error code
253 pushl $ SYMBOL_NAME(do_divide_error)
254 ALIGN
255 error_code:
256 pushl %ds
257 pushl %eax
258 xorl %eax,%eax
259 pushl %ebp
260 pushl %edi
261 pushl %esi
262 pushl %edx
263 decl %eax # eax = -1
264 pushl %ecx
265 pushl %ebx
266 cld
267 movl %es,%cx
268 xchgl %eax, ORIG_EAX(%esp) # orig_eax (get error code.)
269 movl %esp,%edx
270 xchgl %ecx, ES(%esp) # get the addr and save es.
271 pushl %eax # push the error code
272 pushl %edx
273 movl $(__KERNEL_DS),%edx
274 movl %dx,%ds
275 movl %dx,%es
276 GET_CURRENT(%ebx)
277 call *%ecx
278 addl $8,%esp
279 jmp ret_from_exception
280
281 ENTRY(coprocessor_error)
282 pushl $0
283 pushl $ SYMBOL_NAME(do_coprocessor_error)
284 jmp error_code
285
286 ENTRY(device_not_available)
287 pushl $-1 # mark this as an int

```



```

288 SAVE_ALL
289 GET_CURRENT(%ebx)
290 pushl $ret_from_exception
291 movl %cr0,%eax
292 testl $0x4,%eax      # EM (math emulation bit)
293 je SYMBOL_NAME(math_state_restore)
294 pushl $0              # temp storage for ORIG_EIP
295 call SYMBOL_NAME(math_emulate)
296 addl $4,%esp
297 ret
298
299 ENTRY(debug)
300 pushl $0
301 pushl $SYMBOL_NAME(do_debug)
302 jmp error_code
303
304 ENTRY(nmi)
305 pushl $0
306 pushl $SYMBOL_NAME(do_nmi)
307 jmp error_code
308
309 ENTRY(int3)
310 pushl $0
311 pushl $SYMBOL_NAME(do_int3)
312 jmp error_code
313
314 ENTRY(overflow)
315 pushl $0
316 pushl $SYMBOL_NAME(do_overflow)
317 jmp error_code
318
319 ENTRY(bounds)
320 pushl $0
321 pushl $SYMBOL_NAME(do_bounds)
322 jmp error_code
323
324 ENTRY(invalid_op)
325 pushl $0
326 pushl $SYMBOL_NAME(do_invalid_op)
327 jmp error_code
328
329 ENTRY(coprocessor_segment_overrun)
330 pushl $0
331 pushl $SYMBOL_NAME(do_coprocessor_segment_overrun)
332 jmp error_code
333
334 ENTRY(reserved)
335 pushl $0

```

```

336 pushl $SYMBOL_NAME(do_reserved)
337 jmp error_code
338
339 ENTRY(double_fault)
340 pushl $SYMBOL_NAME(do_double_fault)
341 jmp error_code
342
343 ENTRY(invalid_TSS)
344 pushl $SYMBOL_NAME(do_invalid_TSS)
345 jmp error_code
346
347 ENTRY(segment_not_present)
348 pushl $SYMBOL_NAME(do_segment_not_present)
349 jmp error_code
350
351 ENTRY(stack_segment)
352 pushl $SYMBOL_NAME(do_stack_segment)
353 jmp error_code
354
355 ENTRY(general_protection)
356 pushl $SYMBOL_NAME(do_general_protection)
357 jmp error_code
358
359 ENTRY(alignment_check)
360 pushl $SYMBOL_NAME(do_alignment_check)
361 jmp error_code
362
363 ENTRY(page_fault)
364 pushl $SYMBOL_NAME(do_page_fault)
365 jmp error_code
366
367 ENTRY(spurious_interrupt_bug)
368 pushl $0
369 pushl $SYMBOL_NAME(do_spurious_interrupt_bug)
370 jmp error_code
371
372 .data
373 ENTRY(sys_call_table)
374 .long SYMBOL_NAME(sys_ni_syscall)      /* 0 */
375 .long SYMBOL_NAME(sys_exit)
376 .long SYMBOL_NAME(sys_fork)
377 .long SYMBOL_NAME(sys_read)
378 .long SYMBOL_NAME(sys_write)
379 .long SYMBOL_NAME(sys_open)          /* 5 */
380 .long SYMBOL_NAME(sys_close)
381 .long SYMBOL_NAME(sys_waitpid)
382 .long SYMBOL_NAME(sys_creat)
383 .long SYMBOL_NAME(sys_link)

```

```

384 .long SYMBOL_NAME(sys_unlink)          /* 10 */
385 .long SYMBOL_NAME(sys_execve)
386 .long SYMBOL_NAME(sys_chdir)
387 .long SYMBOL_NAME(sys_time)
388 .long SYMBOL_NAME(sys_mknod)
389 .long SYMBOL_NAME(sys_chmod)          /* 15 */
390 .long SYMBOL_NAME(sys_lchown)
391 .long SYMBOL_NAME(sys_ni_syscall) /*old break holder*/
392 .long SYMBOL_NAME(sys_stat)
393 .long SYMBOL_NAME(sys_lseek)
394 .long SYMBOL_NAME(sys_getpid)        /* 20 */
395 .long SYMBOL_NAME(sys_mount)
396 .long SYMBOL_NAME(sys_oldumount)
397 .long SYMBOL_NAME(sys_setuid)
398 .long SYMBOL_NAME(sys_getuid)
399 .long SYMBOL_NAME(sys_stime)        /* 25 */
400 .long SYMBOL_NAME(sys_ptrace)
401 .long SYMBOL_NAME(sys_alarm)
402 .long SYMBOL_NAME(sys_fstat)
403 .long SYMBOL_NAME(sys_pause)
404 .long SYMBOL_NAME(sys_utime)        /* 30 */
405 .long SYMBOL_NAME(sys_ni_syscall) /* old stty holder */
406 .long SYMBOL_NAME(sys_ni_syscall) /* old gtty holder */
407 .long SYMBOL_NAME(sys_access)
408 .long SYMBOL_NAME(sys_nice) /*next: old ftime holder*/
409 .long SYMBOL_NAME(sys_ni_syscall)   /* 35 */
410 .long SYMBOL_NAME(sys_sync)
411 .long SYMBOL_NAME(sys_kill)
412 .long SYMBOL_NAME(sys_rename)
413 .long SYMBOL_NAME(sys_mkdtr)
414 .long SYMBOL_NAME(sys_rmdir)        /* 40 */
415 .long SYMBOL_NAME(sys_dup)
416 .long SYMBOL_NAME(sys_pipe)
417 .long SYMBOL_NAME(sys_times)
418 .long SYMBOL_NAME(sys_ni_syscall) /* old prof holder */
419 .long SYMBOL_NAME(sys_brk)          /* 45 */
420 .long SYMBOL_NAME(sys_setgid)
421 .long SYMBOL_NAME(sys_getgid)
422 .long SYMBOL_NAME(sys_signal)
423 .long SYMBOL_NAME(sys_geteuid)
424 .long SYMBOL_NAME(sys_getegid)     /* 50 */
425 .long SYMBOL_NAME(sys_acct)
426 .long SYMBOL_NAME(sys_umount) /*recyc never used phys*/
427 .long SYMBOL_NAME(sys_ni_syscall) /* old lock holder */
428 .long SYMBOL_NAME(sys_ioctl)
429 .long SYMBOL_NAME(sys_fcntl)       /* 55 */
430 .long SYMBOL_NAME(sys_ni_syscall) /* old mpx holder */
431 .long SYMBOL_NAME(sys_setpgid)

432 .long SYMBOL_NAME(sys_ni_syscall) /*old ulimit holder*/
433 .long SYMBOL_NAME(sys_olduname)
434 .long SYMBOL_NAME(sys_umask)       /* 60 */
435 .long SYMBOL_NAME(sys_chroot)
436 .long SYMBOL_NAME(sys_ustat)
437 .long SYMBOL_NAME(sys_dup2)
438 .long SYMBOL_NAME(sys_getppid)
439 .long SYMBOL_NAME(sys_getpgrp)     /* 65 */
440 .long SYMBOL_NAME(sys_setsid)
441 .long SYMBOL_NAME(sys_sigaction)
442 .long SYMBOL_NAME(sys_sgetmask)
443 .long SYMBOL_NAME(sys_ssetmask)
444 .long SYMBOL_NAME(sys_setreuid)    /* 70 */
445 .long SYMBOL_NAME(sys_setregid)
446 .long SYMBOL_NAME(sys_sigsuspend)
447 .long SYMBOL_NAME(sys_sigpending)
448 .long SYMBOL_NAME(sys_sethostname)
449 .long SYMBOL_NAME(sys_setrlimit)   /* 75 */
450 .long SYMBOL_NAME(sys_getrlimit)
451 .long SYMBOL_NAME(sys_getrusage)
452 .long SYMBOL_NAME(sys_gettimeofday)
453 .long SYMBOL_NAME(sys_settimeofday)
454 .long SYMBOL_NAME(sys_getgroups)   /* 80 */
455 .long SYMBOL_NAME(sys_setgroups)
456 .long SYMBOL_NAME(old_select)
457 .long SYMBOL_NAME(sys_symlink)
458 .long SYMBOL_NAME(sys_lstat)
459 .long SYMBOL_NAME(sys_readlink)    /* 85 */
460 .long SYMBOL_NAME(sys_uselib)
461 .long SYMBOL_NAME(sys_swapon)
462 .long SYMBOL_NAME(sys_reboot)
463 .long SYMBOL_NAME(old_readdir)
464 .long SYMBOL_NAME(old_mmap)        /* 90 */
465 .long SYMBOL_NAME(sys_munmap)
466 .long SYMBOL_NAME(sys_truncate)
467 .long SYMBOL_NAME(sys_ftruncate)
468 .long SYMBOL_NAME(sys_fchmod)
469 .long SYMBOL_NAME(sys_fchown)     /* 95 */
470 .long SYMBOL_NAME(sys_getpriority)
471 .long SYMBOL_NAME(sys_setpriority)
472 .long SYMBOL_NAME(sys_ni_syscall) /*old profil holder*/
473 .long SYMBOL_NAME(sys_statfs)
474 .long SYMBOL_NAME(sys_fstatfs)    /* 100 */
475 .long SYMBOL_NAME(sys_ioperm)
476 .long SYMBOL_NAME(sys_socketcall)
477 .long SYMBOL_NAME(sys_syslog)
478 .long SYMBOL_NAME(sys_setitimer)
479 .long SYMBOL_NAME(sys_getitimer)  /* 105 */

```

```

480 .long SYMBOL_NAME(sys_newstat)
481 .long SYMBOL_NAME(sys_newlstat)
482 .long SYMBOL_NAME(sys_newfstat)
483 .long SYMBOL_NAME(sys_uname)
484 .long SYMBOL_NAME(sys_iopl) /* 110 */
485 .long SYMBOL_NAME(sys_vhangup)
486 .long SYMBOL_NAME(sys_idle)
487 .long SYMBOL_NAME(sys_vm86old)
488 .long SYMBOL_NAME(sys_wait4)
489 .long SYMBOL_NAME(sys_swapoff) /* 115 */
490 .long SYMBOL_NAME(sys_sysinfo)
491 .long SYMBOL_NAME(sys_ipc)
492 .long SYMBOL_NAME(sys_fsync)
493 .long SYMBOL_NAME(sys_sigreturn)
494 .long SYMBOL_NAME(sys_clone) /* 120 */
495 .long SYMBOL_NAME(sys_setdomainname)
496 .long SYMBOL_NAME(sys_newuname)
497 .long SYMBOL_NAME(sys_modify_ldt)
498 .long SYMBOL_NAME(sys_adjtimex)
499 .long SYMBOL_NAME(sys_mprotect) /* 125 */
500 .long SYMBOL_NAME(sys_sigprocmask)
501 .long SYMBOL_NAME(sys_create_module)
502 .long SYMBOL_NAME(sys_init_module)
503 .long SYMBOL_NAME(sys_delete_module)
504 .long SYMBOL_NAME(sys_get_kernel_syms) /* 130 */
505 .long SYMBOL_NAME(sys_quotactl)
506 .long SYMBOL_NAME(sys_getpgid)
507 .long SYMBOL_NAME(sys_fchdir)
508 .long SYMBOL_NAME(sys_bdflush)
509 .long SYMBOL_NAME(sys_sysfs) /* 135 */
510 .long SYMBOL_NAME(sys_personality)
511 .long SYMBOL_NAME(sys_ni_syscall) /* for afs_syscall */
512 .long SYMBOL_NAME(sys_setfsuid)
513 .long SYMBOL_NAME(sys_setfsuid)
514 .long SYMBOL_NAME(sys_llseek) /* 140 */
515 .long SYMBOL_NAME(sys_getdents)
516 .long SYMBOL_NAME(sys_select)
517 .long SYMBOL_NAME(sys_flock)
518 .long SYMBOL_NAME(sys_msync)
519 .long SYMBOL_NAME(sys_readv) /* 145 */
520 .long SYMBOL_NAME(sys_writev)
521 .long SYMBOL_NAME(sys_getsid)
522 .long SYMBOL_NAME(sys_fdatasync)
523 .long SYMBOL_NAME(sys_sysctl)
524 .long SYMBOL_NAME(sys_mlock) /* 150 */
525 .long SYMBOL_NAME(sys_munlock)
526 .long SYMBOL_NAME(sys_mlockall)
527 .long SYMBOL_NAME(sys_munlockall)

```

```

528 .long SYMBOL_NAME(sys_sched_setparam)
529 .long SYMBOL_NAME(sys_sched_getparam) /* 155 */
530 .long SYMBOL_NAME(sys_sched_setscheduler)
531 .long SYMBOL_NAME(sys_sched_getscheduler)
532 .long SYMBOL_NAME(sys_sched_yield)
533 .long SYMBOL_NAME(sys_sched_get_priority_max)
534 .long SYMBOL_NAME(sys_sched_get_priority_min) /* 160 */
535 .long SYMBOL_NAME(sys_sched_rr_get_interval)
536 .long SYMBOL_NAME(sys_nanosleep)
537 .long SYMBOL_NAME(sys_mremap)
538 .long SYMBOL_NAME(sys_setresuid)
539 .long SYMBOL_NAME(sys_getresuid) /* 165 */
540 .long SYMBOL_NAME(sys_vm86)
541 .long SYMBOL_NAME(sys_query_module)
542 .long SYMBOL_NAME(sys_poll)
543 .long SYMBOL_NAME(sys_nfsservctl)
544 .long SYMBOL_NAME(sys_setresgid) /* 170 */
545 .long SYMBOL_NAME(sys_getresgid)
546 .long SYMBOL_NAME(sys_prctl)
547 .long SYMBOL_NAME(sys_rt_sigreturn)
548 .long SYMBOL_NAME(sys_rt_sigaction)
549 .long SYMBOL_NAME(sys_rt_sigprocmask) /* 175 */
550 .long SYMBOL_NAME(sys_rt_sigpending)
551 .long SYMBOL_NAME(sys_rt_sigtimedwait)
552 .long SYMBOL_NAME(sys_rt_sigqueueinfo)
553 .long SYMBOL_NAME(sys_rt_sigsuspend)
554 .long SYMBOL_NAME(sys_pread) /* 180 */
555 .long SYMBOL_NAME(sys_pwrite)
556 .long SYMBOL_NAME(sys_chown)
557 .long SYMBOL_NAME(sys_getcwd)
558 .long SYMBOL_NAME(sys_capget)
559 .long SYMBOL_NAME(sys_capset) /* 185 */
560 .long SYMBOL_NAME(sys_sigaltstack)
561 .long SYMBOL_NAME(sys_sendfile)
562 .long SYMBOL_NAME(sys_ni_syscall) /* streams1 */
563 .long SYMBOL_NAME(sys_ni_syscall) /* streams2 */
564 .long SYMBOL_NAME(sys_vfork) /* 190 */
565
566 /*
567 * NOTE!! This doesn't have to be exact - we just have
568 * to make sure we have _enough_ of the sys_ni_syscall
569 * entries. Don't panic if you notice that this hasn't
570 * been shrunk every time we add a new system call.
571 */
572 .rept NR_syscalls-190
573 .long SYMBOL_NAME(sys_ni_syscall)
574 .endr

```

arch/i386/kernel/init_task.c

```

575 #include <linux/mm.h>
576 #include <linux/sched.h>
577
578 #include <asm/uaccess.h>
579 #include <asm/pgtable.h>
580 #include <asm/desc.h>
581
582 static struct vm_area_struct init_mmap = INIT_MMAP;
583 static struct fs_struct init_fs = INIT_FS;
584 static struct file * init_fd_array[NR_OPEN] = { NULL, };
585 static struct files_struct init_files = INIT_FILES;
586 static struct signal_struct init_signals = INIT_SIGNALS;
587 struct mm_struct init_mm = INIT_MM;
588
589 /* Initial task structure.
590 * We need to make sure that this is 8192-byte aligned
591 * due to the way process stacks are handled. This is
592 * done by having a special "init_task" linker map
593 * entry.. */
594 union task_union init_task_union
595     __attribute__((__section__(".data.init_task"))) =
596     { INIT_TASK };
597

```

arch/i386/kernel/irq.c

```

598 /*
599 *      linux/arch/i386/kernel/irq.c
600 *
601 * Copyright (C) 1992, 1998 Linus Torvalds, Ingo Molnar
602 *
603 * This file contains the code used by various IRQ
604 * handling routines: asking for different IRQ's should
605 * be done through these routines instead of just
606 * grabbing them. Thus setups with different IRQ numbers
607 * shouldn't result in any weird surprises, and
608 * installing new handlers should be easier. */
609
610 /* IRQs are in fact implemented a bit like signal
611 * handlers for the kernel. Naturally it's not a 1:1
612 * relation, but there are similarities. */
613
614 #include <linux/config.h>
615 #include <linux/ptrace.h>
616 #include <linux/errno.h>
617 #include <linux/kernel_stat.h>
618 #include <linux/signal.h>

```

```

619 #include <linux/sched.h>
620 #include <linux/ioport.h>
621 #include <linux/interrupt.h>
622 #include <linux/timex.h>
623 #include <linux/malloc.h>
624 #include <linux/random.h>
625 #include <linux/smp.h>
626 #include <linux/tasks.h>
627 #include <linux/smp_lock.h>
628 #include <linux/init.h>
629
630 #include <asm/system.h>
631 #include <asm/io.h>
632 #include <asm/irq.h>
633 #include <asm/bitops.h>
634 #include <asm/smp.h>
635 #include <asm/pgtable.h>
636 #include <asm/delay.h>
637 #include <asm/desc.h>
638
639 #include "irq.h"
640
641 unsigned int local_bh_count[NR_CPUS];
642 unsigned int local_irq_count[NR_CPUS];
643
644 atomic_t nmi_counter;
645
646 /* Linux has a controller-independent x86 interrupt
647 * architecture. every controller has a
648 * 'controller-template', that is used by the main code
649 * to do the right thing. Each driver-visible interrupt
650 * source is transparently wired to the appropriate
651 * controller. Thus drivers need not be aware of the
652 * interrupt-controller.
653 *
654 * Various interrupt controllers we handle: 8259 PIC, SMP
655 * IO-APIC, PIIX4's internal 8259 PIC and SGI's Visual
656 * Workstation Cobalt (IO-)APIC. (IO-APICs assumed to be
657 * messaging to Pentium local-APICs)
658 *
659 * the code is designed to be easily extended with
660 * new/different interrupt controllers, without having to
661 * do assembly magic. */
662
663 /* Micro-access to controllers is serialized over the
664 * whole system. We never hold this lock when we call the
665 * actual IRQ handler. */
666 spinlock_t irq_controller_lock;

```

```

667
668 /* Dummy controller type for unused interrupts */
669 static void do_none(unsigned int irq,
670                    struct pt_regs * regs)
671 {
672     /* we are careful. While for ISA irqs it's common to
673      * happen outside of any driver (think autodetection),
674      * this is not at all nice for PCI interrupts. So we
675      * are stricter and print a warning when such spurious
676      * interrupts happen. Spurious interrupts can confuse
677      * other drivers if the PCI IRQ line is shared.
678      *
679      * Such spurious interrupts are either driver bugs, or
680      * sometimes hw (chipset) bugs. */
681     printk("unexpected IRQ vector %d on CPU#%d!\n",
682           irq, smp_processor_id());
683
684 #ifdef __SMP__
685     /* [currently unexpected vectors happen only on SMP and
686      * APIC. if we want to have non-APIC and non-8259A
687      * controllers in the future with unexpected vectors,
688      * this ack should probably be made
689      * controller-specific.] */
690     ack_APIC_irq();
691 #endif
692 }
693 static void enable_none(unsigned int irq) { }
694 static void disable_none(unsigned int irq) { }
695
696 /* startup is the same as "enable", shutdown is same as
697 * "disable" */
698 #define startup_none    enable_none
699 #define shutdown_none   disable_none
700
701 struct hw_interrupt_type no_irq_type = {
702     "none",
703     startup_none,
704     shutdown_none,
705     do_none,
706     enable_none,
707     disable_none
708 };
709
710 /* This is the 'legacy' 8259A Programmable Interrupt
711 * Controller, present in the majority of PC/AT boxes. */
712
713 static void do_8259A_IRQ(unsigned int irq,
714                        struct pt_regs * regs);

```

```

715 static void enable_8259A_irq(unsigned int irq);
716 void disable_8259A_irq(unsigned int irq);
717
718 /* startup is the same as "enable", shutdown is same as
719 * "disable" */
720 #define startup_8259A_irq    enable_8259A_irq
721 #define shutdown_8259A_irq   disable_8259A_irq
722
723 static struct hw_interrupt_type i8259A_irq_type = {
724     "XT-PIC",
725     startup_8259A_irq,
726     shutdown_8259A_irq,
727     do_8259A_IRQ,
728     enable_8259A_irq,
729     disable_8259A_irq
730 };
731
732 /* Controller mappings for all interrupt sources: */
733 irq_desc_t irq_desc[NR_IRQS] = { [0 ... NR_IRQS-1] =
734     { 0, &no_irq_type, } };
735
736
737 /* 8259A PIC functions to handle ISA devices: */
738
739 /* This contains the irq mask for both 8259A irq
740 * controllers, */
741 static unsigned int cached_irq_mask = 0xffff;
742
743 #define __byte(x,y) (((unsigned char *)&(y))[x])
744 #define __word(x,y) (((unsigned short *)&(y))[x])
745 #define __long(x,y) (((unsigned int *)&(y))[x])
746
747 #define cached_21      (__byte(0,cached_irq_mask))
748 #define cached_A1     (__byte(1,cached_irq_mask))
749
750 /* Not all IRQs can be routed through the IO-APIC, eg. on
751 * certain (older) boards the timer interrupt is not
752 * connected to any IO-APIC pin, it's fed to the CPU IRQ
753 * line directly.
754 *
755 * Any '1' bit in this mask means the IRQ is routed
756 * through the IO-APIC. this 'mixed mode' IRQ handling
757 * costs nothing because it's only used at IRQ setup
758 * time. */
759 unsigned long io_apic_irqs = 0;
760
761 /* These have to be protected by the irq controller
762 * spinlock before being called. */

```

```

763 void disable_8259A_irq(unsigned int irq)
764 {
765     unsigned int mask = 1 << irq;
766     cached_irq_mask |= mask;
767     if (irq & 8) {
768         outb(cached_A1,0xA1);
769     } else {
770         outb(cached_21,0x21);
771     }
772 }
773
774 static void enable_8259A_irq(unsigned int irq)
775 {
776     unsigned int mask = ~(1 << irq);
777     cached_irq_mask &= mask;
778     if (irq & 8) {
779         outb(cached_A1,0xA1);
780     } else {
781         outb(cached_21,0x21);
782     }
783 }
784
785 int i8259A_irq_pending(unsigned int irq)
786 {
787     unsigned int mask = 1<<irq;
788
789     if (irq < 8)
790         return (inb(0x20) & mask);
791     return (inb(0xA0) & (mask >> 8));
792 }
793
794 void make_8259A_irq(unsigned int irq)
795 {
796     disable_irq(irq);
797     __long(0,io_apic_irqs) &= ~(1<<irq);
798     irq_desc[irq].handler = &i8259A_irq_type;
799     enable_irq(irq);
800 }
801
802 /* Careful! The 8259A is a fragile beast, it pretty much
803 * _has_ to be done exactly like this (mask it first,
804 * _then_ send the EOI, and the order of EOI to the two
805 * 8259s is important! */
806 static inline void mask_and_ack_8259A(unsigned int irq)
807 {
808     cached_irq_mask |= 1 << irq;
809     if (irq & 8) {
810         inb(0xA1);        /* DUMMY */

```

```

811         outb(cached_A1,0xA1);
812         outb(0x62,0x20);    /* Specific EOI to cascade */
813         outb(0x20,0xA0);
814     } else {
815         inb(0x21);        /* DUMMY */
816         outb(cached_21,0x21);
817         outb(0x20,0x20);
818     }
819 }
820
821 static void do_8259A_IRQ(unsigned int irq,
822                          struct pt_regs * regs)
823 {
824     struct irqaction * action;
825     irq_desc_t *desc = irq_desc + irq;
826
827     spin_lock(&irq_controller_lock);
828     {
829         unsigned int status;
830         mask_and_ack_8259A(irq);
831         status = desc->status & ~IRQ_REPLAY;
832         action = NULL;
833         if (!(status & (IRQ_DISABLED | IRQ_INPROGRESS)))
834             action = desc->action;
835         desc->status = status | IRQ_INPROGRESS;
836     }
837     spin_unlock(&irq_controller_lock);
838
839     /* Exit early if we had no action or it was disabled */
840     if (!action)
841         return;
842
843     handle_IRQ_event(irq, regs, action);
844
845     spin_lock(&irq_controller_lock);
846     {
847         unsigned int status = desc->status & ~IRQ_INPROGRESS;
848         desc->status = status;
849         if (!(status & IRQ_DISABLED))
850             enable_8259A_irq(irq);
851     }
852     spin_unlock(&irq_controller_lock);
853 }
854
855 /* This builds up the IRQ handler stubs using some ugly
856 * macros in irq.h
857 *
858 * These macros create the low-level assembly IRQ

```

```

859 * routines that save register context and call do_IRQ().
860 * do_IRQ() then does all the operations that are needed
861 * to keep the AT (or SMP IOAPIC) interrupt-controller
862 * happy. */
863
864 BUILD_COMMON_IRQ()
865
866 #define BI(x,y) \
867     BUILD_IRQ(##x##y)
868
869 #define BUILD_16_IRQS(x) \
870     BI(x,0) BI(x,1) BI(x,2) BI(x,3) \
871     BI(x,4) BI(x,5) BI(x,6) BI(x,7) \
872     BI(x,8) BI(x,9) BI(x,a) BI(x,b) \
873     BI(x,c) BI(x,d) BI(x,e) BI(x,f)
874
875 /* ISA PIC or low IO-APIC triggered (INTA-cycle or APIC)
876 * interrupts: (these are usually mapped to vectors
877 * 0x20-0x30) */
878 BUILD_16_IRQS(0x0)
879
880 #ifdef CONFIG_X86_IO_APIC
881 /* The IO-APIC gives us many more interrupt sources. Most
882 * of these are unused but an SMP system is supposed to
883 * have enough memory ... sometimes (mostly wrt. hw
884 * bugs) we get corrupted vectors all across the
885 * spectrum, so we really want to be prepared to get all
886 * of these. Plus, more powerful systems might have more
887 * than 64 IO-APIC registers.
888 *
889 * (these are usually mapped into the 0x30-0xff vector
890 * range) */
891 BUILD_16_IRQS(0x1) BUILD_16_IRQS(0x2) BUILD_16_IRQS(0x3)
892 BUILD_16_IRQS(0x4) BUILD_16_IRQS(0x5) BUILD_16_IRQS(0x6)
893 BUILD_16_IRQS(0x7) BUILD_16_IRQS(0x8) BUILD_16_IRQS(0x9)
894 BUILD_16_IRQS(0xa) BUILD_16_IRQS(0xb) BUILD_16_IRQS(0xc)
895 BUILD_16_IRQS(0xd)
896 #endif
897
898 #undef BUILD_16_IRQS
899 #undef BI
900
901
902 #ifdef __SMP__
903 /* The following vectors are part of the Linux
904 * architecture, there is no hardware IRQ pin equivalent
905 * for them, they are triggered through the ICC by us
906 * (IPIs) */

```

```

907 BUILD_SMP_INTERRUPT(reschedule_interrupt)
908 BUILD_SMP_INTERRUPT(invalidate_interrupt)
909 BUILD_SMP_INTERRUPT(stop_cpu_interrupt)
910 BUILD_SMP_INTERRUPT(mtrr_interrupt)
911 BUILD_SMP_INTERRUPT(spurious_interrupt)
912
913 /* every pentium local APIC has two 'local interrupts',
914 * with a soft-definable vector attached to both
915 * interrupts, one of which is a timer interrupt, the
916 * other one is error counter overflow. Linux uses the
917 * local APIC timer interrupt to get a much simpler SMP
918 * time architecture: */
919 BUILD_SMP_TIMER_INTERRUPT(apic_timer_interrupt)
920
921 #endif
922
923 #define IRQ(x,y) \
924     IRQ##x##y##_interrupt
925
926 #define IRQLIST_16(x) \
927     IRQ(x,0), IRQ(x,1), IRQ(x,2), IRQ(x,3), \
928     IRQ(x,4), IRQ(x,5), IRQ(x,6), IRQ(x,7), \
929     IRQ(x,8), IRQ(x,9), IRQ(x,a), IRQ(x,b), \
930     IRQ(x,c), IRQ(x,d), IRQ(x,e), IRQ(x,f)
931
932 static void (*interrupt[NR_IRQS])(void) = {
933     IRQLIST_16(0x0).
934
935 #ifdef CONFIG_X86_IO_APIC
936     IRQLIST_16(0x1), IRQLIST_16(0x2), IRQLIST_16(0x3),
937     IRQLIST_16(0x4), IRQLIST_16(0x5), IRQLIST_16(0x6),
938     IRQLIST_16(0x7), IRQLIST_16(0x8), IRQLIST_16(0x9),
939     IRQLIST_16(0xa), IRQLIST_16(0xb), IRQLIST_16(0xc),
940     IRQLIST_16(0xd)
941 #endif
942 };
943
944 #undef IRQ
945 #undef IRQLIST_16
946
947
948 /* Special irq handlers. */
949
950 void no_action(int cpl, void *dev_id,
951                struct pt_regs *regs)
952 {}
953
954 #ifndef CONFIG_VISWS

```

```

955 /* Note that on a 486, we don't want to do a SIGFPE on an
956 * irq13 as the irq is unreliable, and exception 16 works
957 * correctly (ie as explained in the intel
958 * literature). On a 386, you can't use exception 16 due
959 * to bad IBM design, so we have to rely on the less
960 * exact irq13.
961 *
962 * Careful.. Not only is IRQ13 unreliable, but it is also
963 * leads to races. IBM designers who came up with it
964 * should be shot. */
965 static void math_error_irq(int cpl, void *dev_id,
966                          struct pt_regs *regs)
967 {
968     outb(0,0xF0);
969     if (ignore_irq13 || !boot_cpu_data.hard_math)
970         return;
971     math_error();
972 }
973
974 static struct irqaction irq13 =
975 { math_error_irq, 0, 0, "fpu", NULL, NULL };
976
977 /* IRQ2 is cascade interrupt to second interrupt
978 * controller */
979 static struct irqaction irq2 =
980 { no_action, 0, 0, "cascade", NULL, NULL};
981 #endif
982
983 /* Generic, controller-independent functions: */
984
985 int get_irq_list(char *buf)
986 {
987     int i, j;
988     struct irqaction * action;
989     char *p = buf;
990
991     p += sprintf(p, "          ");
992     for (j=0; j<smp_num_cpus; j++)
993         p += sprintf(p, "CPU%d      ",j);
994     *p++ = '\n';
995
996     for (i = 0 ; i < NR_IRQS ; i++) {
997         action = irq_desc[i].action;
998         if (!action)
999             continue;
1000         p += sprintf(p, "%3d: ",i);
1001 #ifndef __SMP__
1002         p += sprintf(p, "%10u ", kstat_irqs(i));

```

```

1003 #else
1004     for (j=0; j<smp_num_cpus; j++)
1005         p += sprintf(p, "%10u ",
1006                    kstat_irqs[cpu_logical_map(j)][i]);
1007 #endif
1008     p += sprintf(p, " %14s",
1009                irq_desc[i].handler->typename);
1010     p += sprintf(p, " %s", action->name);
1011
1012     for (action=action->next; action;
1013          action = action->next) {
1014         p += sprintf(p, " %s", action->name);
1015     }
1016     *p++ = '\n';
1017 }
1018 p += sprintf(p, "NMI: %10u\n",
1019             atomic_read(&nmi_counter));
1020 #ifdef __SMP__
1021 p += sprintf(p, "ERR: %10lu\n", ipi_count);
1022 #endif
1023 return p - buf;
1024 }
1025
1026 /* Global interrupt locks for SMP. Allow interrupts to
1027 * come in on any CPU, yet make cli/sti act globally to
1028 * protect critical regions.. */
1029 #ifdef __SMP__
1030 unsigned char global_irq_holder = NO_PROC_ID;
1031 unsigned volatile int global_irq_lock;
1032 atomic_t global_irq_count;
1033
1034 atomic_t global_bh_count;
1035 atomic_t global_bh_lock;
1036
1037 /* "global_cli()" is a special case, in that it can hold
1038 * the interrupts disabled for a longish time, and also
1039 * because we may be doing TLB invalidates when holding
1040 * the global IRQ lock for historical reasons. Thus we
1041 * may need to check SMP invalidate events specially by
1042 * hand here (but not in any normal spinlocks) */
1043 static inline void check_smp_invalidate(int cpu)
1044 {
1045     if (test_bit(cpu, &smp_invalidate_needed)) {
1046         clear_bit(cpu, &smp_invalidate_needed);
1047         local_flush_tlb();
1048     }
1049 }
1050

```



```

1051 static void show(char * str)
1052 {
1053     int i;
1054     unsigned long *stack;
1055     int cpu = smp_processor_id();
1056     extern char *get_options(char *str, int *ints);
1057
1058     printk("\n%s, CPU %d:\n", str, cpu);
1059     printk("irq: %d [%d %d]\n",
1060         atomic_read(&global_irq_count), local_irq_count[0],
1061         local_irq_count[1]);
1062     printk("bh: %d [%d %d]\n",
1063         atomic_read(&global_bh_count), local_bh_count[0],
1064         local_bh_count[1]);
1065     stack = (unsigned long *) &stack;
1066     for (i = 40; i ; i--) {
1067         unsigned long x = *++stack;
1068         if (x > (unsigned long) &get_options &&
1069             x < (unsigned long) &vsprintf) {
1070             printk("<[%08lx]> ", x);
1071         }
1072     }
1073 }
1074
1075 #define MAXCOUNT 100000000
1076
1077 static inline void wait_on_bh(void)
1078 {
1079     int count = MAXCOUNT;
1080     do {
1081         if (!--count) {
1082             show("wait_on_bh");
1083             count = ~0;
1084         }
1085         /* nothing .. wait for the other bh's to go away */
1086     } while (atomic_read(&global_bh_count) != 0);
1087 }
1088
1089 /* I had a lockup scenario where a tight loop doing
1090 * spin_unlock()/spin_lock() on CPU#1 was racing with
1091 * spin_lock() on CPU#0. CPU#0 should have noticed
1092 * spin_unlock(), but apparently the spin_unlock()
1093 * information did not make it through to CPU#0
1094 * ... nasty, is this by design, do we have to limit
1095 * 'memory update oscillation frequency' artificially
1096 * like here?
1097 *
1098 * Such 'high frequency update' races can be avoided by

```

```

1099 * careful design, but some of our major constructs like
1100 * spinlocks use similar techniques, it would be nice to
1101 * clarify this issue. Set this define to 0 if you want
1102 * to check whether your system freezes. I suspect the
1103 * delay done by SYNC_OTHER_CORES() is in correlation
1104 * with 'snooping latency', but i thought that such
1105 * things are guaranteed by design, since we use the
1106 * 'LOCK' prefix. */
1107 #define SUSPECTED_CPU_OR_CHIPSET_BUG_WORKAROUND 1
1108
1109 #if SUSPECTED_CPU_OR_CHIPSET_BUG_WORKAROUND
1110 # define SYNC_OTHER_CORES(x) udelay(x+1)
1111 #else
1112 /* We have to allow irqs to arrive between __sti and
1113 * __cli */
1114 # define SYNC_OTHER_CORES(x) __asm__ __volatile__ ("nop")
1115 #endif
1116
1117 static inline void wait_on_irq(int cpu)
1118 {
1119     int count = MAXCOUNT;
1120
1121     for (;;) {
1122
1123         /* Wait until all interrupts are gone. Wait for
1124         * bottom half handlers unless we're already
1125         * executing in one.. */
1126         if (!atomic_read(&global_irq_count)) {
1127             if (local_bh_count[cpu] ||
1128                 !atomic_read(&global_bh_count))
1129                 break;
1130         }
1131
1132         /* Duh, we have to loop. Release the lock to avoid
1133         * deadlocks */
1134         clear_bit(0, &global_irq_lock);
1135
1136         for (;;) {
1137             if (!--count) {
1138                 show("wait_on_irq");
1139                 count = ~0;
1140             }
1141             __sti();
1142             SYNC_OTHER_CORES(cpu);
1143             __cli();
1144             check_smp_invalidate(cpu);
1145             if (atomic_read(&global_irq_count))
1146                 continue;

```

```

1147     if (global_irq_lock)
1148         continue;
1149     if (!local_bh_count[cpu] &&
1150         atomic_read(&global_bh_count))
1151         continue;
1152     if (!test_and_set_bit(0,&global_irq_lock))
1153         break;
1154     }
1155 }
1156 }
1157
1158 /* This is called when we want to synchronize with bottom
1159 * half handlers. We need to wait until no other CPU is
1160 * executing any bottom half handler.
1161 *
1162 * Don't wait if we're already running in an interrupt
1163 * context or are inside a bh handler. */
1164 void synchronize_bh(void)
1165 {
1166     if (atomic_read(&global_bh_count) && !in_interrupt())
1167         wait_on_bh();
1168 }
1169
1170 /* This is called when we want to synchronize with
1171 * interrupts. We may for example tell a device to stop
1172 * sending interrupts: but to make sure there are no
1173 * interrupts that are executing on another CPU we need
1174 * to call this function. */
1175 void synchronize_irq(void)
1176 {
1177     if (atomic_read(&global_irq_count)) {
1178         /* Stupid approach */
1179         cli();
1180         sti();
1181     }
1182 }
1183
1184 static inline void get_irqlock(int cpu)
1185 {
1186     if (test_and_set_bit(0,&global_irq_lock)) {
1187         /* do we already hold the lock? */
1188         if ((unsigned char) cpu == global_irq_holder)
1189             return;
1190         /* Uhhuh.. Somebody else got it. Wait.. */
1191         do {
1192             do {
1193                 check_smp_invalidate(cpu);
1194             } while (test_bit(0,&global_irq_lock));

```

```

1195         } while (test_and_set_bit(0,&global_irq_lock));
1196     }
1197     /* We also to make sure that nobody else is running in
1198     * an interrupt context. */
1199     wait_on_irq(cpu);
1200
1201     /* Ok, finally.. */
1202     global_irq_holder = cpu;
1203 }
1204
1205 #define EFLAGS_IF_SHIFT 9
1206
1207 /* A global "cli()" while in an interrupt context turns
1208 * into just a local cli(). Interrupts should use
1209 * spinlocks for the (very unlikely) case that they ever
1210 * want to protect against each other.
1211 *
1212 * If we already have local interrupts disabled, this
1213 * will not turn a local disable into a global one
1214 * (problems with spinlocks: this makes
1215 * save_flags+cli+sti usable inside a spinlock). */
1216 void __global_cli(void)
1217 {
1218     unsigned int flags;
1219
1220     __save_flags(flags);
1221     if (flags & (1 << EFLAGS_IF_SHIFT)) {
1222         int cpu = smp_processor_id();
1223         __cli();
1224         if (!local_irq_count[cpu])
1225             get_irqlock(cpu);
1226     }
1227 }
1228
1229 void global_sti(void)
1230 {
1231     int cpu = smp_processor_id();
1232
1233     if (!local_irq_count[cpu])
1234         release_irqlock(cpu);
1235     __sti();
1236 }
1237
1238 /* SMP flags value to restore to:
1239 * 0 - global cli
1240 * 1 - global sti
1241 * 2 - local cli
1242 * 3 - local sti */

```

```

1243 unsigned long __global_save_flags(void)
1244 {
1245     int retval;
1246     int local_enabled;
1247     unsigned long flags;
1248
1249     __save_flags(flags);
1250     local_enabled = (flags >> EFLAGS_IF_SHIFT) & 1;
1251     /* default to local */
1252     retval = 2 + local_enabled;
1253
1254     /*check for global flags if we're not in an interrupt*/
1255     if (!local_irq_count[smp_processor_id()]) {
1256         if (local_enabled)
1257             retval = 1;
1258         if (global_irq_holder ==
1259             (unsigned char) smp_processor_id())
1260             retval = 0;
1261     }
1262     return retval;
1263 }
1264
1265 void __global_restore_flags(unsigned long flags)
1266 {
1267     switch (flags) {
1268     case 0:
1269         __global_cli();
1270         break;
1271     case 1:
1272         __global_sti();
1273         break;
1274     case 2:
1275         __cli();
1276         break;
1277     case 3:
1278         __sti();
1279         break;
1280     default:
1281         printk("global_restore_flags: %08lx (%08lx)\n",
1282             flags, (&flags)[-1]);
1283     }
1284 }
1285
1286 #endif
1287
1288 /* This should really return information about whether we
1289 * should do bottom half handling etc. Right now we end
1290 * up _always_ checking the bottom half, which is a waste

```

```

1291 * of time and is not what some drivers would prefer. */
1292 int handle_IRQ_event(unsigned int irq,
1293     struct pt_regs * regs, struct irqaction * action)
1294 {
1295     int status;
1296     int cpu = smp_processor_id();
1297
1298     irq_enter(cpu, irq);
1299
1300     status = 1;    /* Force the "do bottom halves" bit */
1301
1302     if (!(action->flags & SA_INTERRUPT))
1303         __sti();
1304
1305     do {
1306         status |= action->flags;
1307         action->handler(irq, action->dev_id, regs);
1308         action = action->next;
1309     } while (action);
1310     if (status & SA_SAMPLE_RANDOM)
1311         add_interrupt_randomness(irq);
1312     __cli();
1313
1314     irq_exit(cpu, irq);
1315
1316     return status;
1317 }
1318
1319 /* Generic enable/disable code: this just calls down into
1320 * the PIC-specific version for the actual hardware
1321 * disable after having gotten the irq controller lock.
1322 */
1323 void disable_irq(unsigned int irq)
1324 {
1325     unsigned long flags;
1326
1327     spin_lock_irqsave(&irq_controller_lock, flags);
1328     if (!irq_desc[irq].depth++) {
1329         irq_desc[irq].status |= IRQ_DISABLED;
1330         irq_desc[irq].handler->disable(irq);
1331     }
1332     spin_unlock_irqrestore(&irq_controller_lock, flags);
1333
1334     if (irq_desc[irq].status & IRQ_INPROGRESS)
1335         synchronize_irq();
1336 }
1337
1338 void enable_irq(unsigned int irq)

```

```

1339 {
1340     unsigned long flags;
1341
1342     spin_lock_irqsave(&irq_controller_lock, flags);
1343     switch (irq_desc[irq].depth) {
1344     case 1:
1345         irq_desc[irq].status &= ~(IRQ_DISABLED |
1346                                   IRQ_INPROGRESS);
1347         irq_desc[irq].handler->enable(irq);
1348         /* fall through */
1349     default:
1350         irq_desc[irq].depth--;
1351         break;
1352     case 0:
1353         printk("enable_irq() unbalanced from %p\n",
1354               __builtin_return_address(0));
1355     }
1356     spin_unlock_irqrestore(&irq_controller_lock, flags);
1357 }
1358
1359 /* do_IRQ handles all normal device IRQ's (the special
1360 * SMP cross-CPU interrupts have their own specific
1361 * handlers). */
1362 asmlinkage void do_IRQ(struct pt_regs regs)
1363 {
1364     /* We ack quickly, we don't want the irq controller
1365     * thinking we're snobs just because some other CPU has
1366     * disabled global interrupts (we have already done the
1367     * INT_ACK cycles, it's too late to try to pretend to
1368     * the controller that we aren't taking the interrupt).
1369     *
1370     * 0 return value means that this irq is already being
1371     * handled by some other CPU. (or is disabled) */
1372     int irq = regs.orig_eax & 0xff; /* subtle, see irq.h */
1373     int cpu = smp_processor_id();
1374
1375     kstat.irqs[cpu][irq]++;
1376     irq_desc[irq].handler->handle(irq, &regs);
1377
1378     /* This should be conditional: we should really get a
1379     * return code from the irq handler to tell us whether
1380     * the handler wants us to do software bottom half
1381     * handling or not.. */
1382     if (1) {
1383         if (bh_active & bh_mask)
1384             do_bottom_half();
1385     }
1386 }

```

```

1387
1388 int setup_x86_irq(unsigned int irq,
1389                  struct irqaction * new)
1390 {
1391     int shared = 0;
1392     struct irqaction *old, **p;
1393     unsigned long flags;
1394
1395     /* Some drivers like serial.c use request_irq()
1396     * heavily, so we have to be careful not to interfere
1397     * with a running system. */
1398     if (new->flags & SA_SAMPLE_RANDOM) {
1399         /* This function might sleep, we want to call it
1400         * first, outside of the atomic block. Yes, this
1401         * might clear the entropy pool if the wrong driver
1402         * is attempted to be loaded, without actually
1403         * installing a new handler, but is this really a
1404         * problem, only the sysadmin is able to do this. */
1405         rand_initialize_irq(irq);
1406     }
1407
1408     /* The following block of code has to be executed
1409     * atomically */
1410     spin_lock_irqsave(&irq_controller_lock, flags);
1411     p = &irq_desc[irq].action;
1412     if ((old = *p) != NULL) {
1413         /* Can't share interrupts unless both agree to */
1414         if (!(old->flags & new->flags & SA_SHIRQ)) {
1415             spin_unlock_irqrestore(&irq_controller_lock, flags);
1416             return -EBUSY;
1417         }
1418
1419         /* add new interrupt at end of irq queue */
1420         do {
1421             p = &old->next;
1422             old = *p;
1423         } while (old);
1424         shared = 1;
1425     }
1426
1427     *p = new;
1428
1429     if (!shared) {
1430         irq_desc[irq].depth = 0;
1431         irq_desc[irq].status &= ~(IRQ_DISABLED |
1432                                   IRQ_INPROGRESS);
1433         irq_desc[irq].handler->startup(irq);
1434     }

```

```

1435 spin_unlock_irqrestore(&irq_controller_lock, flags);
1436 return 0;
1437 }
1438
1439 int request_irq(unsigned int irq,
1440 void (*handler)(int, void *, struct pt_regs *),
1441 unsigned long irqflags,
1442 const char * devname,
1443 void *dev_id)
1444 {
1445 int retval;
1446 struct irqaction * action;
1447
1448 if (irq >= NR_IRQS)
1449 return -EINVAL;
1450 if (!handler)
1451 return -EINVAL;
1452
1453 action = (struct irqaction *)
1454 kmalloc(sizeof(struct irqaction), GFP_KERNEL);
1455 if (!action)
1456 return -ENOMEM;
1457
1458 action->handler = handler;
1459 action->flags = irqflags;
1460 action->mask = 0;
1461 action->name = devname;
1462 action->next = NULL;
1463 action->dev_id = dev_id;
1464
1465 retval = setup_x86_irq(irq, action);
1466
1467 if (retval)
1468 kfree(action);
1469 return retval;
1470 }
1471
1472 void free_irq(unsigned int irq, void *dev_id)
1473 {
1474 struct irqaction * action, **p;
1475 unsigned long flags;
1476
1477 if (irq >= NR_IRQS)
1478 return;
1479
1480 spin_lock_irqsave(&irq_controller_lock, flags);
1481 for (p = &irq_desc[irq].action;
1482 (action = *p) != NULL; p = &action->next) {

```

```

1483 if (action->dev_id != dev_id)
1484 continue;
1485
1486 /* Found it - now free it */
1487 *p = action->next;
1488 kfree(action);
1489 if (!irq_desc[irq].action) {
1490 irq_desc[irq].status |= IRQ_DISABLED;
1491 irq_desc[irq].handler->shutdown(irq);
1492 }
1493 goto out;
1494 }
1495 printk("Trying to free free IRQ%d\n", irq);
1496 out:
1497 spin_unlock_irqrestore(&irq_controller_lock, flags);
1498 }
1499
1500 /* IRQ autodetection code..
1501 *
1502 * This depends on the fact that any interrupt that comes
1503 * in on to an unassigned handler will get stuck with
1504 * "IRQ_INPROGRESS" asserted and the interrupt disabled.
1505 */
1506 unsigned long probe_irq_on(void)
1507 {
1508 unsigned int i;
1509 unsigned long delay;
1510
1511 /* first, enable any unassigned irqs */
1512 spin_lock_irq(&irq_controller_lock);
1513 for (i = NR_IRQS-1; i > 0; i--) {
1514 if (!irq_desc[i].action) {
1515 unsigned int status =
1516 irq_desc[i].status | IRQ_AUTODETECT;
1517 irq_desc[i].status = status & ~IRQ_INPROGRESS;
1518 irq_desc[i].handler->startup(i);
1519 }
1520 }
1521 spin_unlock_irq(&irq_controller_lock);
1522
1523 /* Wait for spurious interrupts to trigger */
1524 for (delay = jiffies + HZ/10;
1525 time_after(delay, jiffies); )
1526 /* about 100ms delay */ synchronize_irq();
1527
1528 /* Now filter out any obviously spurious interrupts */
1529 spin_lock_irq(&irq_controller_lock);
1530 for (i=0; i<NR_IRQS; i++) {

```

```

1531     unsigned int status = irq_desc[i].status;
1532
1533     if (!(status & IRQ_AUTODETECT))
1534         continue;
1535
1536     /* It triggered already - consider it spurious. */
1537     if (status & IRQ_INPROGRESS) {
1538         irq_desc[i].status = status & ~IRQ_AUTODETECT;
1539         irq_desc[i].handler->shutdown(i);
1540     }
1541 }
1542 spin_unlock_irq(&irq_controller_lock);
1543
1544 return 0x12345678;
1545 }
1546
1547 int probe_irq_off(unsigned long unused)
1548 {
1549     int i, irq_found, nr_irqs;
1550
1551     if (unused != 0x12345678)
1552         printk("Bad IRQ probe from %lx\n", (&unused)[-1]);
1553
1554     nr_irqs = 0;
1555     irq_found = 0;
1556     spin_lock_irq(&irq_controller_lock);
1557     for (i=0; i<NR_IRQS; i++) {
1558         unsigned int status = irq_desc[i].status;
1559
1560         if (!(status & IRQ_AUTODETECT))
1561             continue;
1562
1563         if (status & IRQ_INPROGRESS) {
1564             if (!nr_irqs)
1565                 irq_found = i;
1566             nr_irqs++;
1567         }
1568         irq_desc[i].status = status & ~IRQ_AUTODETECT;
1569         irq_desc[i].handler->shutdown(i);
1570     }
1571     spin_unlock_irq(&irq_controller_lock);
1572
1573     if (nr_irqs > 1)
1574         irq_found = -irq_found;
1575     return irq_found;
1576 }
1577
1578 void init_ISA_irqs (void)

```

```

1579 {
1580     int i;
1581
1582     for (i = 0; i < NR_IRQS; i++) {
1583         irq_desc[i].status = IRQ_DISABLED;
1584         irq_desc[i].action = 0;
1585         irq_desc[i].depth = 0;
1586
1587         if (i < 16) {
1588             /* 16 old-style INTA-cycle interrupts: */
1589             irq_desc[i].handler = &i8259A_irq_type;
1590         } else {
1591             /* 'high' PCI IRQs filled in on demand */
1592             irq_desc[i].handler = &no_irq_type;
1593         }
1594     }
1595 }
1596
1597 __initfunc(void init_IRQ(void))
1598 {
1599     int i;
1600
1601     #ifndef CONFIG_X86_VISWS_APIC
1602     init_ISA_irqs();
1603     #else
1604     init_VISWS_APIC_irqs();
1605     #endif
1606     /* Cover the whole vector space, no vector can escape
1607      * us. (some of these will be overridden and become
1608      * 'special' SMP interrupts) */
1609     for (i = 0; i < NR_IRQS; i++) {
1610         int vector = FIRST_EXTERNAL_VECTOR + i;
1611         if (vector != SYSCALL_VECTOR)
1612             set_intr_gate(vector, interrupt[i]);
1613     }
1614
1615     #ifdef __SMP__
1616
1617     /* IRQ0 must be given a fixed assignment and
1618      * initialized before init_IRQ_SMP. */
1619     set_intr_gate(IRQ0_TRAP_VECTOR, interrupt[0]);
1620
1621     /* The reschedule interrupt is a CPU-to-CPU
1622      * reschedule-helper IPI, driven by wakeup. */
1623     set_intr_gate(RESCCHEDULE_VECTOR, reschedule_interrupt);
1624
1625     /* IPI for invalidation */
1626     set_intr_gate(INVALIDATE_TLB_VECTOR,

```

```

1627         invalidate_interrupt);
1628
1629 /* IPI for CPU halt */
1630 set_intr_gate(STOP_CPU_VECTOR, stop_cpu_interrupt);
1631
1632 /* self generated IPI for local APIC timer */
1633 set_intr_gate(LOCAL_TIMER_VECTOR, apic_timer_interrupt);
1634
1635 /* IPI for MTRR control */
1636 set_intr_gate(MTRR_CHANGE_VECTOR, mtrr_interrupt);
1637
1638 /* IPI vector for APIC spurious interrupts */
1639 set_intr_gate(SPURIOUS_APIC_VECTOR, spurious_interrupt);
1640 #endif
1641 request_region(0x20, 0x20, "pic1");
1642 request_region(0xa0, 0x20, "pic2");
1643
1644 /* Set the clock to 100 Hz, we already have a valid
1645  * vector now: */
1646 outb_p(0x34, 0x43); /* binary, mode 2, LSB/MSB, ch 0 */
1647 outb_p(LATCH & 0xff, 0x40); /* LSB */
1648 outb(LATCH >> 8, 0x40); /* MSB */
1649
1650 #ifndef CONFIG_VISWS
1651 setup_x86_irq(2, &irq2);
1652 setup_x86_irq(13, &irq13);
1653 #endif
1654 }
1655
1656 #ifdef CONFIG_X86_IO_APIC
1657 __initfunc(void init_IRQ_SMP(void))
1658 {
1659     int i;
1660     for (i = 0; i < NR_IRQS; i++)
1661         if (IO_APIC_VECTOR(i) > 0)
1662             set_intr_gate(IO_APIC_VECTOR(i), interrupt[i]);
1663 }
1664 #endif
1665

```

arch/i386/kernel/irq.h

```

1666 #ifndef __irq_h
1667 #define __irq_h
1668
1669 #include <asm/irq.h>
1670
1671 /* Interrupt controller descriptor. This is all we need

```

```

1672  * to describe about the low-level hardware. */
1673 struct hw_interrupt_type {
1674     const char * typename;
1675     void (*startup)(unsigned int irq);
1676     void (*shutdown)(unsigned int irq);
1677     void (*handle)(unsigned int irq, struct pt_regs * regs);
1678     void (*enable)(unsigned int irq);
1679     void (*disable)(unsigned int irq);
1680 };
1681
1682 extern struct hw_interrupt_type no_irq_type;
1683
1684 /* IRQ line status. */
1685 #define IRQ_INPROGRESS 1 /* active - do not enter! */
1686 #define IRQ_DISABLED 2 /* disabled - do not enter! */
1687 #define IRQ_PENDING 4 /* pending, replay on enable */
1688 #define IRQ_REPLAY 8 /* replayed but not acked */
1689 #define IRQ_AUTODETECT 16 /* IRQ being autodetected */
1690
1691 /* This is the "IRQ descriptor", which contains various
1692  * information about the irq, including what kind of
1693  * hardware handling it has, whether it is disabled etc
1694  * etc.
1695  *
1696  * Pad this out to 32 bytes for cache and indexing
1697  * reasons. */
1698 typedef struct {
1699     /* IRQ status - IRQ_INPROGRESS, IRQ_DISABLED */
1700     unsigned int status;
1701     /* handle/enable/disable functions */
1702     struct hw_interrupt_type *handler;
1703     /* IRQ action list */
1704     struct irqaction *action;
1705     /* Disable depth for nested irq disables */
1706     unsigned int depth;
1707 } irq_desc_t;
1708
1709 /* IDT vectors usable for external interrupt sources
1710  * start at 0x20: */
1711 #define FIRST_EXTERNAL_VECTOR 0x20
1712
1713 #define SYSCALL_VECTOR 0x80
1714
1715 /* Vectors 0x20-0x2f are used for ISA interrupts. */
1716
1717 /* Special IRQ vectors used by the SMP architecture:
1718  *
1719  * (some of the following vectors are 'rare', they might

```

```

1720 * be merged into a single vector to save vector
1721 * space. TLB, reschedule and local APIC vectors are
1722 * performance-critical.) */
1723 #define RESCHEDULE_VECTOR      0x30
1724 #define INVALIDATE_TLB_VECTOR  0x31
1725 #define STOP_CPU_VECTOR        0x40
1726 #define LOCAL_TIMER_VECTOR     0x41
1727 #define MTRR_CHANGE_VECTOR     0x50
1728
1729 /* First APIC vector available to drivers: (vectors
1730 * 0x51-0xfe) */
1731 #define IRQ0_TRAP_VECTOR       0x51
1732
1733 /* This IRQ should never happen, but we print a message
1734 nevertheless. */
1735 #define SPURIOUS_APIC_VECTOR   0xff
1736
1737 extern irq_desc_t irq_desc[NR_IRQS];
1738 extern int irq_vector[NR_IRQS];
1739 #define IO_APIC_VECTOR(irq)    irq_vector[irq]
1740
1741 extern void init_IRQ_SMP(void);
1742 extern int handle_IRQ_event(unsigned int,
1743                             struct pt_regs *, struct irqaction *);
1744 extern int setup_x86_irq(unsigned int,
1745                          struct irqaction *);
1746
1747 /* Various low-level irq details needed by irq.c,
1748 * process.c, time.c, io_apic.c and smp.c
1749 *
1750 * Interrupt entry/exit code at both C and assembly level
1751 */
1752
1753 extern void no_action(int cpl, void *dev_id,
1754                      struct pt_regs *regs);
1755 extern void mask_irq(unsigned int irq);
1756 extern void unmask_irq(unsigned int irq);
1757 extern void disable_8259A_irq(unsigned int irq);
1758 extern int i8259A_irq_pending(unsigned int irq);
1759 extern void ack_APIC_irq(void);
1760 extern void FASTCALL(send_IPI_self(int vector));
1761 extern void smp_send_mtrr(void);
1762 extern void init_VISWS_APIC_irqs(void);
1763 extern void setup_IO_APIC(void);
1764 extern int IO_APIC_get_PCI_irq_vector(int bus, int slot,
1765                                       int fn);
1766 extern void make_8259A_irq(unsigned int irq);
1767 extern void send_IPI(int dest, int vector);
1768 extern void init_pic_mode(void);
1769 extern void print_IO_APIC(void);
1770
1771 extern unsigned long io_apic_irqs;
1772
1773 extern char _stext, _etext;
1774
1775 #define MAX_IRQ_SOURCES 128
1776 #define MAX_MP_BUSSES 32
1777 enum mp_bustype {
1778     MP_BUS_ISA,
1779     MP_BUS_PCI
1780 };
1781 extern int mp_bus_id_to_type [MAX_MP_BUSSES];
1782 extern int mp_bus_id_to_pci_bus [MAX_MP_BUSSES];
1783 extern char ioapic_OEM_ID [16];
1784 extern char ioapic_Product_ID [16];
1785
1786 extern spinlock_t irq_controller_lock;
1787
1788 #ifdef __SMP__
1789
1790 #include <asm/atomic.h>
1791
1792 static inline void irq_enter(int cpu, unsigned int irq)
1793 {
1794     hardirq_enter(cpu);
1795     while (test_bit(0,&global_irq_lock)) {
1796         /* nothing */;
1797     }
1798 }
1799
1800 static inline void irq_exit(int cpu, unsigned int irq)
1801 {
1802     hardirq_exit(cpu);
1803 }
1804
1805 #define IO_APIC_IRQ(x) (((x) >= 16) || \
1806                       ((1<<(x)) & io_apic_irqs))
1807
1808 #else
1809
1810 #define irq_enter(cpu, irq)    (++local_irq_count[cpu])
1811 #define irq_exit(cpu, irq)    (--local_irq_count[cpu])
1812
1813 #define IO_APIC_IRQ(x) (0)
1814
1815 #endif

```



```

1816
1817 #define __STR(x) #x
1818 #define STR(x) __STR(x)
1819
1820 #define SAVE_ALL \
1821 "cld\n\t" \
1822 "pushl %es\n\t" \
1823 "pushl %ds\n\t" \
1824 "pushl %eax\n\t" \
1825 "pushl %ebp\n\t" \
1826 "pushl %edi\n\t" \
1827 "pushl %esi\n\t" \
1828 "pushl %edx\n\t" \
1829 "pushl %ecx\n\t" \
1830 "pushl %ebx\n\t" \
1831 "movl $" STR(__KERNEL_DS) ",%edx\n\t" \
1832 "movl %dx,%ds\n\t" \
1833 "movl %dx,%es\n\t"
1834
1835 #define IRQ_NAME2(nr) nr##_interrupt(void)
1836 #define IRQ_NAME(nr) IRQ_NAME2(IRQ##nr)
1837
1838 #define GET_CURRENT \
1839 "movl %esp, %ebx\n\t" \
1840 "andl $-8192, %ebx\n\t"
1841
1842 #ifdef __Smp__
1843
1844 /* SMP has a few special interrupts for IPI messages */
1845
1846 #define BUILD_SMP_INTERRUPT(x) \
1847 asmlinkage void x(void); \
1848 __asm__( \
1849 "\n"__ALIGN_STR"\n" \
1850 SYMBOL_NAME_STR(x) ":\n\t" \
1851 "pushl $-1\n\t" \
1852 SAVE_ALL \
1853 "call "SYMBOL_NAME_STR(smp_##x)"\n\t" \
1854 "jmp ret_from_intr\n");
1855
1856 #define BUILD_SMP_TIMER_INTERRUPT(x) \
1857 asmlinkage void x(struct pt_regs * regs); \
1858 __asm__( \
1859 "\n"__ALIGN_STR"\n" \
1860 SYMBOL_NAME_STR(x) ":\n\t" \
1861 "pushl $1\n\t" \
1862 SAVE_ALL \
1863 "movl %eip,%eax\n\t" \

```

```

1864 "pushl %eax\n\t" \
1865 "call "SYMBOL_NAME_STR(smp_##x)"\n\t" \
1866 "addl $4,%esp\n\t" \
1867 "jmp ret_from_intr\n");
1868
1869 #endif /* __SMP__ */
1870
1871 #define BUILD_COMMON_IRQ() \
1872 __asm__( \
1873 "\n"__ALIGN_STR"\n" \
1874 "common_interrupt:\n\t" \
1875 SAVE_ALL \
1876 "pushl $ret_from_intr\n\t" \
1877 "jmp "SYMBOL_NAME_STR(do_IRQ));
1878
1879 /* subtle. orig_eax is used by the signal code to
1880 * distinct between system calls and interrupted 'random
1881 * user-space'. Thus we have to put a negative value into
1882 * orig_eax here. (the problem is that both system calls
1883 * and IRQs want to have small integer numbers in
1884 * orig_eax, and the syscall code has won the
1885 * optimization conflict ;) */
1886 #define BUILD_IRQ(nr) \
1887 asmlinkage void IRQ_NAME(nr); \
1888 __asm__( \
1889 "\n"__ALIGN_STR"\n" \
1890 SYMBOL_NAME_STR(IRQ) #nr "_interrupt:\n\t" \
1891 "pushl $"#nr"-256\n\t" \
1892 "jmp common_interrupt");
1893
1894 /* x86 profiling function, SMP safe. We might want to do
1895 * this in assembly totally? */
1896 static inline void x86_do_profile (unsigned long eip)
1897 {
1898     if (prof_buffer && current->pid) {
1899         eip -= (unsigned long) &_stext;
1900         eip >>= prof_shift;
1901         /* Don't ignore out-of-bounds EIP values silently,
1902          * put them into the last histogram slot, so if
1903          * present, they will show up as a sharp peak. */
1904         if (eip > prof_len-1)
1905             eip = prof_len-1;
1906         atomic_inc((atomic_t *)&prof_buffer[eip]);
1907     }
1908 }
1909
1910 #endif

```

arch/i386/kernel/process.c

```
1911 /*
1912 * linux/arch/i386/kernel/process.c
1913 *
1914 * Copyright (C) 1995 Linus Torvalds
1915 */
1916
1917 /* This file handles the architecture-dependent parts of
1918 * process handling.. */
1919
1920 #define __KERNEL_SYSCALLS__
1921 #include <stdarg.h>
1922
1923 #include <linux/errno.h>
1924 #include <linux/sched.h>
1925 #include <linux/kernel.h>
1926 #include <linux/mm.h>
1927 #include <linux/smp.h>
1928 #include <linux/smp_lock.h>
1929 #include <linux/stddef.h>
1930 #include <linux/unistd.h>
1931 #include <linux/ptrace.h>
1932 #include <linux/malloc.h>
1933 #include <linux/vmalloc.h>
1934 #include <linux/user.h>
1935 #include <linux/a.out.h>
1936 #include <linux/interrupt.h>
1937 #include <linux/config.h>
1938 #include <linux/unistd.h>
1939 #include <linux/delay.h>
1940 #include <linux/smp.h>
1941 #include <linux/reboot.h>
1942 #include <linux/init.h>
1943 #if defined(CONFIG_APM) && defined(CONFIG_APM_POWER_OFF)
1944 #include <linux/apm_bios.h>
1945 #endif
1946
1947 #include <asm/uaccess.h>
1948 #include <asm/pgtable.h>
1949 #include <asm/system.h>
1950 #include <asm/io.h>
1951 #include <asm/ldt.h>
1952 #include <asm/processor.h>
1953 #include <asm/desc.h>
1954 #ifdef CONFIG_MATH_EMULATION
1955 #include <asm/math_emu.h>
1956 #endif
1957
```

```
1958 #include "irq.h"
1959
1960 spinlock_t semaphore_wake_lock = SPIN_LOCK_UNLOCKED;
1961
1962 asmlinkage void ret_from_fork(void)
1963     __asm__("ret_from_fork");
1964
1965 #ifdef CONFIG_APM
1966 extern int apm_do_idle(void);
1967 extern void apm_do_busy(void);
1968 #endif
1969
1970 static int hlt_counter=0;
1971
1972 #define HARD_IDLE_TIMEOUT (HZ / 3)
1973
1974 void disable_hlt(void)
1975 {
1976     hlt_counter++;
1977 }
1978
1979 void enable_hlt(void)
1980 {
1981     hlt_counter--;
1982 }
1983
1984 #ifndef __SMP__
1985
1986 static void hard_idle(void)
1987 {
1988     while (!current->need_resched) {
1989         if (boot_cpu_data.hlt_works_ok && !hlt_counter) {
1990 #ifdef CONFIG_APM
1991             /* If the APM BIOS is not enabled, or there
1992              * is an error calling the idle routine, we
1993              * should hlt if possible. We need to check
1994              * need_resched again because an interrupt
1995              * may have occurred in apm_do_idle(). */
1996             start_bh_atomic();
1997             if (!apm_do_idle() && !current->need_resched)
1998                 __asm__("hlt");
1999             end_bh_atomic();
2000 #else
2001             __asm__("hlt");
2002 #endif
2003         }
2004         if (current->need_resched)
2005             break;

```

```

2006     schedule();
2007 }
2008 #ifdef CONFIG_APM
2009     apm_do_busy();
2010 #endif
2011 }
2012
2013 /* The idle loop on a uniprocessor i386.. */
2014 static int cpu_idle(void *unused)
2015 {
2016     int work = 1;
2017     unsigned long start_idle = 0;
2018
2019     /* endless idle loop with no priority at all */
2020     current->priority = 0;
2021     current->counter = -100;
2022     for (;;) {
2023         if (work)
2024             start_idle = jiffies;
2025
2026         if (jiffies - start_idle > HARD_IDLE_TIMEOUT)
2027             hard_idle();
2028         else {
2029             if (boot_cpu_data.hlt_works_ok &&
2030                 !hlt_counter && !current->need_resched)
2031                 __asm__("hlt");
2032         }
2033
2034         work = current->need_resched;
2035         schedule();
2036         check_pgt_cache();
2037     }
2038 }
2039
2040 #else
2041
2042 /* This is being executed in task 0 'user space'. */
2043
2044 int cpu_idle(void *unused)
2045 {
2046     /* endless idle loop with no priority at all */
2047     current->priority = 0;
2048     current->counter = -100;
2049     while(1) {
2050         if (current_cpu_data.hlt_works_ok && !hlt_counter &&
2051             !current->need_resched)
2052             __asm__("hlt");
2053         /* although we are an idle CPU, we do not want to get

```

```

2054         * into the scheduler unnecessarily. */
2055         if (current->need_resched) {
2056             schedule();
2057             check_pgt_cache();
2058         }
2059     }
2060 }
2061
2062 #endif
2063
2064 asmlinkage int sys_idle(void)
2065 {
2066     if (current->pid != 0)
2067         return -EPERM;
2068     cpu_idle(NULL);
2069     return 0;
2070 }
2071
2072 /* This routine reboots the machine by asking the
2073 * keyboard controller to pulse the reset-line low. We
2074 * try that for a while, and if it doesn't work, we do
2075 * some other stupid things. */
2076
2077 static long no_idt[2] = {0, 0};
2078 static int reboot_mode = 0;
2079 static int reboot_thru_bios = 0;
2080
2081 __initfunc(void reboot_setup(char *str, int *ints))
2082 {
2083     while(1) {
2084         switch (*str) {
2085             case 'w': /* "warm" reboot (no memory testing etc) */
2086                 reboot_mode = 0x1234;
2087                 break;
2088             case 'c': /* "cold" reboot (w/ memory testing etc) */
2089                 reboot_mode = 0x0;
2090                 break;
2091             case 'b': /* "bios" reboot by jumping thru the BIOS*/
2092                 reboot_thru_bios = 1;
2093                 break;
2094             case 'h':
2095                 /* "hard" reboot by toggling RESET and/or crashing
2096                  * the CPU */
2097                 reboot_thru_bios = 0;
2098                 break;
2099         }
2100         if((str = strchr(str, ',')) != NULL)
2101             str++;

```

```

2102     else
2103         break;
2104     }
2105 }
2106
2107 /* The following code and data reboots the machine by
2108 * switching to real mode and jumping to the BIOS reset
2109 * entry point, as if the CPU has really been reset. The
2110 * previous version asked the keyboard controller to
2111 * pulse the CPU reset line, which is more thorough, but
2112 * doesn't work with at least one type of 486
2113 * motherboard. It is easy to stop this code working;
2114 * hence the copious comments. */
2115 static unsigned long long
2116 real_mode_gdt_entries [3] =
2117 {
2118     0x0000000000000000ULL, /* Null descriptor */
2119     /* 16-bit real-mode 64k code at 0x00000000 */
2120     0x00009a000000ffffULL,
2121     /* 16-bit real-mode 64k data at 0x00000100 */
2122     0x000092000100ffffULL
2123 };
2124
2125 static struct
2126 {
2127     unsigned short    size __attribute__((packed));
2128     unsigned long long * base __attribute__((packed));
2129 }
2130 real_mode_gdt = { sizeof (real_mode_gdt_entries) - 1,
2131                 real_mode_gdt_entries },
2132 real_mode_idt = { 0x3ff, 0 };
2133
2134 /* This is 16-bit protected mode code to disable paging
2135    and the cache, switch to real mode and jump to the
2136    BIOS reset code.
2137
2138    The instruction that switches to real mode by writing
2139    to CRO must be followed immediately by a far jump
2140    instruction, which set CS to a valid value for real
2141    mode, and flushes the prefetch queue to avoid running
2142    instructions that have already been decoded in
2143    protected mode.
2144
2145    Clears all the flags except ET, especially PG
2146    (paging), PE (protected-mode enable) and TS (task
2147    switch for coprocessor state save). Flushes the TLB
2148    after paging has been disabled. Sets CD and NW, to
2149    disable the cache on a 486, and invalidates the cache.

```

```

2150     This is more like the state of a 486 after reset. I
2151     don't know if something else should be done for other
2152     chips.
2153
2154     More could be done here to set up the registers as if
2155     a CPU reset had occurred; hopefully real BIOSs don't
2156     assume much. */
2157
2158 static unsigned char real_mode_switch [] =
2159 {
2160     0x66, 0x0f, 0x20, 0xc0,      /*movl %cr0,%eax */
2161     0x66, 0x83, 0xe0, 0x11,      /*andl $0x00000011,%eax*/
2162     /*orl $0x60000000,%eax*/
2163     0x66, 0x0d, 0x00, 0x00, 0x00, 0x60,
2164     0x66, 0x0f, 0x22, 0xc0,      /*movl %eax,%cr0 */
2165     0x66, 0x0f, 0x22, 0xd8,      /*movl %eax,%cr3 */
2166     0x66, 0x0f, 0x20, 0xc3,      /*movl %cr0,%ebx */
2167     /*andl $0x60000000,%ebx*/
2168     0x66, 0x81, 0xe3, 0x00, 0x00, 0x00, 0x60,
2169     0x74, 0x02,                  /*jz f */
2170     0x0f, 0x08,                  /*invd */
2171     0x24, 0x10,                  /*f: andb $0x10,a1*/
2172     0x66, 0x0f, 0x22, 0xc0,      /*movl %eax,%cr0*/
2173     0xea, 0x00, 0x00, 0xff, 0xff /*ljmp $0xffff,$0x0000*/
2174 };
2175
2176 static inline void kb_wait(void)
2177 {
2178     int i;
2179
2180     for (i=0; i<0x10000; i++)
2181         if ((inb_p(0x64) & 0x02) == 0)
2182             break;
2183 }
2184
2185 void machine_restart(char * __unused)
2186 {
2187     #if __SMP__
2188     /* turn off the IO-APIC, so we can do a clean reboot */
2189     init_pic_mode();
2190     #endif
2191
2192     if(!reboot_thru_bios) {
2193         /* rebooting needs to touch the page at abs addr 0 */
2194         *((unsigned short *)__va(0x472)) = reboot_mode;
2195         for (;;) {
2196             int i;
2197             for (i=0; i<100; i++) {

```

```

2198     kb_wait();
2199     udelay(50);
2200     outb(0xfe,0x64);          /* pulse reset low */
2201     udelay(50);
2202 }
2203 /* That didn't work - force a triple fault.. */
2204 __asm__ __volatile__ ("lidt %0" : : "m" (no_idt));
2205 __asm__ __volatile__ ("int3");
2206 }
2207 }
2208
2209 cli();
2210
2211 /* Write zero to CMOS register number 0x0f, which the
2212 BIOS POST routine will recognize as telling it to do
2213 a proper reboot. (Well that's what this book in
2214 front of me says -- it may only apply to the Phoenix
2215 BIOS though, it's not clear). At the same time,
2216 disable NMIs by setting the top bit in the CMOS
2217 address register, as we're about to do peculiar
2218 things to the CPU. I'm not sure if 'outb_p' is
2219 needed instead of just 'outb'. Use it to be on the
2220 safe side. */
2221
2222 outb_p (0x8f, 0x70);
2223 outb_p (0x00, 0x71);
2224
2225 /* Remap the kernel at virtual address zero, as well as
2226 offset zero from the kernel segment. This assumes
2227 the kernel segment starts at virtual address
2228 PAGE_OFFSET. */
2229
2230 memcpy (swapper_pg_dir, swapper_pg_dir + USER_PGD_PTRS,
2231         sizeof (swapper_pg_dir [0]) * KERNEL_PGD_PTRS);
2232
2233 /* Make sure the first page is mapped to the start of
2234 physical memory. It is normally not mapped, to trap
2235 kernel NULL pointer dereferences. */
2236
2237 pg0[0] = _PAGE_RW | _PAGE_PRESENT;
2238
2239 /* Use 'swapper_pg_dir' as our page directory. We
2240 * bother with 'SET_PAGE_DIR' because although might be
2241 * rebooting, but if we change the way we set root page
2242 * dir in the future, then we wont break a seldom used
2243 * feature ;) */
2244
2245 SET_PAGE_DIR(current,swapper_pg_dir);
2246
2247 /* Write 0x1234 to absolute memory location 0x472. The
2248 BIOS reads this on booting to tell it to "Bypass
2249 memory test (also warm boot)". This seems like a
2250 fairly standard thing that gets set by REBOOT.COM
2251 programs, and the previous reset routine did this
2252 too. */
2253
2254 *((unsigned short *)0x472) = reboot_mode;
2255
2256 /* For the switch to real mode, copy some code to low
2257 memory. It has to be in the first 64k because it is
2258 running in 16-bit mode, and it has to have the same
2259 physical and virtual address, because it turns off
2260 paging. Copy it near the end of the first page, out
2261 of the way of BIOS variables. */
2262
2263 memcpy ((void *) (0x1000 - sizeof (real_mode_switch)),
2264         real_mode_switch, sizeof (real_mode_switch));
2265
2266 /* Set up the IDT for real mode. */
2267
2268 __asm__ __volatile__
2269 ("lidt %0" : : "m" (real_mode_idt));
2270
2271 /* Set up a GDT from which we can load segment
2272 descriptors for real mode. The GDT is not used in
2273 real mode; it is just needed here to prepare the
2274 descriptors. */
2275
2276 __asm__ __volatile__
2277 ("lgdt %0" : : "m" (real_mode_gdt));
2278
2279 /* Load the data segment registers, and thus the
2280 descriptors ready for real mode. The base address
2281 of each segment is 0x100, 16 times the selector
2282 value being loaded here. This is so that the
2283 segment registers don't have to be reloaded after
2284 switching to real mode: the values are consistent
2285 for real mode operation already. */
2286
2287 __asm__ __volatile__ ("movl $0x0010,%eax\n"
2288                      "\tmovl %%ax,%%ds\n"
2289                      "\tmovl %%ax,%%es\n"
2290                      "\tmovl %%ax,%%fs\n"
2291                      "\tmovl %%ax,%%gs\n"
2292                      "\tmovl %%ax,%%ss" : : "eax");
2293

```

```

2294 /* Jump to the 16-bit code that we copied earlier. It
2295     disables paging and the cache, switches to real
2296     mode, and jumps to the BIOS reset entry point. */
2297
2298 __asm__ __volatile__ ("ljmp $0x0008,%0"
2299     :
2300     : "i" ((void *) (0x1000 -
2301         sizeof (real_mode_switch))));
2302 }
2303
2304 void machine_halt(void)
2305 {}
2306
2307 void machine_power_off(void)
2308 {
2309 #if defined(CONFIG_APM) && defined(CONFIG_APM_POWER_OFF)
2310     apm_power_off();
2311 #endif
2312 }
2313
2314
2315 void show_regs(struct pt_regs * regs)
2316 {
2317     long cr0 = 0L, cr2 = 0L, cr3 = 0L;
2318
2319     printk("\n");
2320     printk("EIP: %04x:[<%08lx>]",
2321         0xffff & regs->xcs,regs->eip);
2322     if (regs->xcs & 3)
2323         printk(" ESP: %04x:%08lx",
2324             0xffff & regs->xss,regs->esp);
2325     printk(" EFLAGS: %08lx\n",regs->eflags);
2326     printk("EAX: %08lx EBX: %08lx ECX: %08lx EDX: %08lx\n",
2327         regs->eax,regs->ebx,regs->ecx,regs->edx);
2328     printk("ESI: %08lx EDI: %08lx EBP: %08lx",
2329         regs->esi, regs->edi, regs->ebp);
2330     printk(" DS: %04x ES: %04x\n",
2331         0xffff & regs->xds,0xffff & regs->xes);
2332     __asm__("movl %%cr0, %0": "=r" (cr0));
2333     __asm__("movl %%cr2, %0": "=r" (cr2));
2334     __asm__("movl %%cr3, %0": "=r" (cr3));
2335     printk("CR0: %08lx CR2: %08lx CR3: %08lx\n",
2336         cr0, cr2, cr3);
2337 }
2338
2339 /* Allocation and freeing of basic task resources.
2340  *
2341  * NOTE! The task struct and the stack go together

```

```

2342  *
2343  * The task structure is a two-page thing, and as such
2344  * not reliable to allocate using the basic page alloc
2345  * functions. We have a small cache of structures for
2346  * when the allocations fail..
2347  *
2348  * This extra buffer essentially acts to make for less
2349  * "jitter" in the allocations..
2350  *
2351  * On SMP we don't do this right now because:
2352  * - we aren't holding any locks when called, and we
2353  *   might as well just depend on the generic memory
2354  *   management to do proper locking for us instead of
2355  *   complicating it here.
2356  * - if you use SMP you have a beefy enough machine that
2357  *   this shouldn't matter.. */
2358 #ifndef __SMP__
2359 #define EXTRA_TASK_STRUCT      16
2360 static struct task_struct *
2361 task_struct_stack[EXTRA_TASK_STRUCT];
2362 static int task_struct_stack_ptr = -1;
2363 #endif
2364
2365 struct task_struct * alloc_task_struct(void)
2366 {
2367 #ifndef EXTRA_TASK_STRUCT
2368     return (struct task_struct *)
2369         __get_free_pages(GFP_KERNEL,1);
2370 #else
2371     int index;
2372     struct task_struct *ret;
2373
2374     index = task_struct_stack_ptr;
2375     if (index >= EXTRA_TASK_STRUCT/2)
2376         goto use_cache;
2377     ret = (struct task_struct *)
2378         __get_free_pages(GFP_KERNEL,1);
2379     if (!ret) {
2380         index = task_struct_stack_ptr;
2381         if (index >= 0) {
2382 use_cache:
2383             ret = task_struct_stack[index];
2384             task_struct_stack_ptr = index-1;
2385         }
2386     }
2387     return ret;
2388 #endif
2389 }

```

```

2390
2391 void free_task_struct(struct task_struct *p)
2392 {
2393 #ifdef EXTRA_TASK_STRUCT
2394     int index = task_struct_stack_ptr+1;
2395
2396     if (index < EXTRA_TASK_STRUCT) {
2397         task_struct_stack[index] = p;
2398         task_struct_stack_ptr = index;
2399     } else
2400 #endif
2401     free_pages((unsigned long) p, 1);
2402 }
2403
2404 void release_segments(struct mm_struct *mm)
2405 {
2406     if (mm->segments) {
2407         void * ldt = mm->segments;
2408         mm->segments = NULL;
2409         vfree(ldt);
2410     }
2411 }
2412
2413 void forget_segments(void)
2414 {
2415     /* forget local segments */
2416     __asm__ __volatile__ ("movl %w0,%%fs : movl %w0,%%gs"
2417         : /* no outputs */
2418         : "r" (0));
2419
2420     /* Get the LDT entry from init_task. */
2421     current->tss.ldt = _LDT(0);
2422     load_ldt(0);
2423 }
2424
2425 /* Create a kernel thread */
2426 int kernel_thread(int (*fn)(void *), void * arg,
2427     unsigned long flags)
2428 {
2429     long retval, d0;
2430
2431     __asm__ __volatile__(
2432         "movl %%esp,%%esi\n\t"
2433         "int $0x80\n\t"          /* Linux/i386 system call */
2434         "cml %%esp,%%esi\n\t"  /* child or parent? */
2435         "je lf\n\t"           /* parent - jump */
2436         /* Load the argument into eax, and push it. That
2437          * way, it does not matter whether the called

```

```

2438     * function is compiled with -mregparm or not. */
2439     "movl %4,%eax\n\t"
2440     "pushl %%eax\n\t"
2441     "call *%5\n\t"           /* call fn */
2442     "movl %3,%0\n\t"       /* exit */
2443     "int $0x80\n\t"
2444     "l:\t"
2445     :"%&a" (retval), "%&S" (d0)
2446     : "0" (__NR_clone), "i" (__NR_exit),
2447       "r" (arg), "r" (fn),
2448       "b" (flags | CLONE_VM)
2449     : "memory");
2450     return retval;
2451 }
2452
2453 /* Free current thread data structures etc.. */
2454 void exit_thread(void)
2455 {
2456     /* nothing to do ... */
2457 }
2458
2459 void flush_thread(void)
2460 {
2461     int i;
2462     struct task_struct *tsk = current;
2463
2464     for (i=0 ; i<8 ; i++)
2465         tsk->tss.debugreg[i] = 0;
2466
2467     /* Forget coprocessor state.. */
2468     clear_fpu(tsk);
2469     tsk->used_math = 0;
2470 }
2471
2472 void release_thread(struct task_struct *dead_task)
2473 {
2474 }
2475
2476 /* If new_mm is NULL, we're being called to set up the
2477 * LDT descriptor for a clone task. Each clone must have
2478 * a separate entry in the GDT. */
2479 void copy_segments(int nr, struct task_struct *p,
2480     struct mm_struct *new_mm)
2481 {
2482     struct mm_struct * old_mm = current->mm;
2483     void * old_ldt = old_mm->segments, * ldt = old_ldt;
2484
2485     /* default LDT - use the one from init_task */

```

```

2486 p->tss.ldt = _LDT(0);
2487 if (old_ldt) {
2488     if (new_mm) {
2489         ldt = vmalloc(LDT_ENTRIES*LDT_ENTRY_SIZE);
2490         new_mm->segments = ldt;
2491         if (!ldt) {
2492             printk(KERN_WARNING "ldt allocation failed\n");
2493             return;
2494         }
2495         memcpy(ldt, old_ldt, LDT_ENTRIES*LDT_ENTRY_SIZE);
2496     }
2497     p->tss.ldt = _LDT(nr);
2498     set_ldt_desc(nr, ldt, LDT_ENTRIES);
2499     return;
2500 }
2501 }
2502
2503 /* Save a segment. */
2504 #define savesegment(seg,value) \
2505     asm volatile("movl %%#seg ",%0":"-m" \
2506                 (*(int *)&(value)))
2507
2508 int copy_thread(int nr, unsigned long clone_flags,
2509                unsigned long esp,struct task_struct * p,
2510                struct pt_regs * regs)
2511 {
2512     struct pt_regs * childregs;
2513
2514     childregs = ((struct pt_regs *)
2515                 (2*PAGE_SIZE + (unsigned long)p)) - 1;
2516     *childregs = *regs;
2517     childregs->eax = 0;
2518     childregs->esp = esp;
2519
2520     p->tss.esp = (unsigned long) childregs;
2521     p->tss.esp0 = (unsigned long) (childregs+1);
2522     p->tss.ss0 = __KERNEL_DS;
2523
2524     p->tss.tr = _TSS(nr);
2525     set_tss_desc(nr,&(p->tss));
2526     p->tss.eip = (unsigned long) ret_from_fork;
2527
2528     savesegment(fs,p->tss.fs);
2529     savesegment(gs,p->tss.gs);
2530
2531     /* a bitmap offset pointing outside of the TSS limit
2532      * causes a nicely controllable SIGSEGV. The first
2533      * sys_ioperm() call sets up the bitmap properly. */

```

```

2534     p->tss.bitmap = sizeof(struct thread_struct);
2535
2536     unlazy_fpu(current);
2537     p->tss.i387 = current->tss.i387;
2538
2539     return 0;
2540 }
2541
2542 /* fill in the FPU structure for a core dump. */
2543 int dump_fpu(struct pt_regs * regs,
2544             struct user_i387_struct * fpu)
2545 {
2546     int fpvalid;
2547     struct task_struct *tsk = current;
2548
2549     fpvalid = tsk->used_math;
2550     if (fpvalid) {
2551         unlazy_fpu(tsk);
2552         memcpy(fpu,&tsk->tss.i387.hard,sizeof(*fpu));
2553     }
2554
2555     return fpvalid;
2556 }
2557
2558 /* fill in the user structure for a core dump.. */
2559 void dump_thread(struct pt_regs * regs,
2560                 struct user * dump)
2561 {
2562     int i;
2563
2564     /* changed the size calculations - should hopefully work
2565      better. lbt */
2566     dump->magic = CMAGIC;
2567     dump->start_code = 0;
2568     dump->start_stack = regs->esp & ~(PAGE_SIZE - 1);
2569     dump->u_tsize =
2570         ((unsigned long) current->mm->end_code)
2571         >> PAGE_SHIFT;
2572     dump->u_dsize =
2573         ((unsigned long) (current->mm->brk + (PAGE_SIZE-1)))
2574         >> PAGE_SHIFT;
2575     dump->u_dsize -= dump->u_tsize;
2576     dump->u_ssize = 0;
2577     for (i = 0; i < 8; i++)
2578         dump->u_debugreg[i] = current->tss.debugreg[i];
2579
2580     if (dump->start_stack < TASK_SIZE)
2581         dump->u_ssize =

```



```

2582     ((unsigned long) (TASK_SIZE - dump->start_stack))
2583     >> PAGE_SHIFT;
2584
2585 dump->regs.ebx = regs->ebx;
2586 dump->regs.ecx = regs->ecx;
2587 dump->regs.edx = regs->edx;
2588 dump->regs.esi = regs->esi;
2589 dump->regs.edi = regs->edi;
2590 dump->regs.ebp = regs->ebp;
2591 dump->regs.eax = regs->eax;
2592 dump->regs.ds = regs->xds;
2593 dump->regs.es = regs->xes;
2594 savesegment(fs,dump->regs.fs);
2595 savesegment(gs,dump->regs.gs);
2596 dump->regs.orig_eax = regs->orig_eax;
2597 dump->regs.eip = regs->eip;
2598 dump->regs.cs = regs->xcs;
2599 dump->regs.eflags = regs->eflags;
2600 dump->regs.esp = regs->esp;
2601 dump->regs.ss = regs->xss;
2602
2603 dump->u_fpvalid = dump_fpu (regs, &dump->i387);
2604 }
2605
2606 /* This special macro can be used to load a debugging
2607  * register */
2608 #define loaddebug(tsk,register) \
2609     __asm__ ("movl %0,%db" #register \
2610             : /* no output */ \
2611             : "r" (tsk->tss.debugreg[register]))
2612
2613
2614 /*      switch_to(x,yn) should switch tasks from x to y.
2615  *
2616  * We fsave/fwait so that an exception goes off at the
2617  * right time (as a call from the fsave or fwait in
2618  * effect) rather than to the wrong process. Lazy FP
2619  * saving no longer makes any sense with modern CPU's,
2620  * and this simplifies a lot of things (SMP and UP become
2621  * the same).
2622  *
2623  * NOTE! We used to use the x86 hardware context
2624  * switching. The reason for not using it any more
2625  * becomes apparent when you try to recover gracefully
2626  * from saved state that is no longer valid (stale
2627  * segment register values in particular). With the
2628  * hardware task-switch, there is no way to fix up bad
2629  * state in a reasonable manner.

```

```

2630  *
2631  * The fact that Intel documents the hardware
2632  * task-switching to be slow is a fairly red herring -
2633  * this code is not noticeably faster. However, there
2634  * _is_ some room for improvement here, so the
2635  * performance issues may eventually be a valid point.
2636  * More important, however, is the fact that this allows
2637  * us much more flexibility. */
2638 void __switch_to(struct task_struct *prev,
2639                 struct task_struct *next)
2640 {
2641     /* Save FPU and set TS if it wasn't set before.. */
2642     unlazy_fpu(prev);
2643
2644     /* Reload TR, LDT and the page table pointers..
2645      *
2646      * We need TR for the IO permission bitmask (and the
2647      * vm86 bitmaps in case we ever use enhanced v86 mode
2648      * properly).
2649      *
2650      * We may want to get rid of the TR register some day,
2651      * and copy the bitmaps around by hand. Oh, well. In
2652      * the meantime we have to clear the busy bit in the
2653      * TSS entry, ugh. */
2654     gdt_table[next->tss.tr >> 3].b &= 0xffffdfff;
2655     asm volatile("ltr %0": : "g"
2656                 (*(unsigned short *)&next->tss.tr));
2657
2658     /* Save away %fs and %gs. No need to save %es and %ds,
2659      * as those are always kernel segments while inside the
2660      * kernel. */
2661     asm volatile("movl %%fs,%0":"=m"
2662                 (*(int *)&prev->tss.fs));
2663     asm volatile("movl %%gs,%0":"=m"
2664                 (*(int *)&prev->tss.gs));
2665
2666     /* Re-load LDT if necessary */
2667     if (next->mm->segments != prev->mm->segments)
2668         asm volatile("lldt %0": : "g"
2669                     (*(unsigned short *)&next->tss.ldt));
2670
2671     /* Re-load page tables */
2672     {
2673         unsigned long new_cr3 = next->tss.cr3;
2674         if (new_cr3 != prev->tss.cr3)
2675             asm volatile("movl %0,%%cr3": : "r" (new_cr3));
2676     }
2677

```

```

2678 /* Restore %fs and %gs. */
2679 loadsegment(fs,next->tss.fs);
2680 loadsegment(gs,next->tss.gs);
2681
2682 /* Now maybe reload the debug registers */
2683 if (next->tss.debugreg[7]){
2684     loaddebug(next,0);
2685     loaddebug(next,1);
2686     loaddebug(next,2);
2687     loaddebug(next,3);
2688     loaddebug(next,6);
2689     loaddebug(next,7);
2690 }
2691 }
2692
2693 asmlinkage int sys_fork(struct pt_regs regs)
2694 {
2695     return do_fork(SIGCHLD, regs.esp, &regs);
2696 }
2697
2698 asmlinkage int sys_clone(struct pt_regs regs)
2699 {
2700     unsigned long clone_flags;
2701     unsigned long newsp;
2702
2703     clone_flags = regs.ebx;
2704     newsp = regs.ecx;
2705     if (!newsp)
2706         newsp = regs.esp;
2707     return do_fork(clone_flags, newsp, &regs);
2708 }
2709
2710 /* This is trivial, and on the face of it looks like it
2711 * could equally well be done in user mode.
2712 *
2713 * Not so, for quite unobvious reasons - register
2714 * pressure. In user mode vfork() cannot have a stack
2715 * frame, and if done by calling the "clone()" system
2716 * call directly, you do not have enough call-clobbered
2717 * registers to hold all the information you need. */
2718 asmlinkage int sys_vfork(struct pt_regs regs)
2719 {
2720     return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD,
2721                 regs.esp, &regs);
2722 }
2723
2724 /* sys_execve() executes a new program. */
2725 asmlinkage int sys_execve(struct pt_regs regs)

```

```

2726 {
2727     int error;
2728     char * filename;
2729
2730     lock_kernel();
2731     filename = getname((char *) regs.ebx);
2732     error = PTR_ERR(filename);
2733     if (IS_ERR(filename))
2734         goto out;
2735     error = do_execve(filename, (char **) regs.ecx,
2736                     (char **) regs.edx, &regs);
2737     if (error == 0)
2738         current->flags &= ~PF_DTRACE;
2739     putname(filename);
2740 out:
2741     unlock_kernel();
2742     return error;
2743 }

```

arch/i386/kernel/signal.c

```

2744 /*
2745 * linux/arch/i386/kernel/signal.c
2746 *
2747 * Copyright (C) 1991, 1992 Linus Torvalds
2748 * 1997-11-28 Modified for POSIX.1b signals by Richard
2749 * Henderson */
2750
2751 #include <linux/config.h>
2752
2753 #include <linux/sched.h>
2754 #include <linux/mm.h>
2755 #include <linux/smp.h>
2756 #include <linux/smp_lock.h>
2757 #include <linux/kernel.h>
2758 #include <linux/signal.h>
2759 #include <linux/errno.h>
2760 #include <linux/wait.h>
2761 #include <linux/ptrace.h>
2762 #include <linux/unistd.h>
2763 #include <linux/stddef.h>
2764 #include <asm/ucontext.h>
2765 #include <asm/uaccess.h>
2766
2767 #define DEBUG_SIG 0
2768
2769 #define _BLOCKABLE (~(sigmask(SIGKILL)|sigmask(SIGSTOP)))
2770

```

```

2771 asmlinkage int sys_wait4(pid_t pid,
2772             unsigned long *stat_addr,
2773             int options, unsigned long *ru);
2774 asmlinkage int FASTCALL(do_signal(struct pt_regs *regs,
2775             sigset_t *oldset));
2776
2777 /* Atomically swap in the new signal mask, and wait for a
2778 * signal. */
2779 asmlinkage int
2780 sys_sigsuspend(int history0, int history1,
2781             old_sigset_t mask)
2782 {
2783     struct pt_regs * regs = (struct pt_regs *) &history0;
2784     sigset_t saveset;
2785
2786     mask &= _BLOCKABLE;
2787     spin_lock_irq(&current->sigmask_lock);
2788     saveset = current->blocked;
2789     siginitset(&current->blocked, mask);
2790     recalc_sigpending(current);
2791     spin_unlock_irq(&current->sigmask_lock);
2792
2793     regs->eax = -EINTR;
2794     while (1) {
2795         current->state = TASK_INTERRUPTIBLE;
2796         schedule();
2797         if (do_signal(regs, &saveset))
2798             return -EINTR;
2799     }
2800 }
2801
2802 asmlinkage int
2803 sys_rt_sigsuspend(sigset_t *unewset, size_t sigsetsize)
2804 {
2805     struct pt_regs * regs = (struct pt_regs *) &unewset;
2806     sigset_t saveset, newset;
2807
2808     /* XXX: Don't preclude handling different sized
2809     * sigset_t's. */
2810     if (sigsetsize != sizeof(sigset_t))
2811         return -EINVAL;
2812
2813     if (copy_from_user(&newset, unewset, sizeof(newset)))
2814         return -EFAULT;
2815     sigdelsetmask(&newset, ~_BLOCKABLE);
2816
2817     spin_lock_irq(&current->sigmask_lock);
2818     saveset = current->blocked;

```

```

2819     current->blocked = newset;
2820     recalc_sigpending(current);
2821     spin_unlock_irq(&current->sigmask_lock);
2822
2823     regs->eax = -EINTR;
2824     while (1) {
2825         current->state = TASK_INTERRUPTIBLE;
2826         schedule();
2827         if (do_signal(regs, &saveset))
2828             return -EINTR;
2829     }
2830 }
2831
2832 asmlinkage int
2833 sys_sigaction(int sig, const struct old_sigaction *act,
2834             struct old_sigaction *oact)
2835 {
2836     struct k_sigaction new_ka, old_ka;
2837     int ret;
2838
2839     if (act) {
2840         old_sigset_t mask;
2841         if (verify_area(VERIFY_READ, act, sizeof(*act)) ||
2842             __get_user(new_ka.sa.sa_handler,
2843                 &act->sa_handler) ||
2844             __get_user(new_ka.sa.sa_restorer,
2845                 &act->sa_restorer))
2846             return -EFAULT;
2847         __get_user(new_ka.sa.sa_flags, &act->sa_flags);
2848         __get_user(mask, &act->sa_mask);
2849         siginitset(&new_ka.sa.sa_mask, mask);
2850     }
2851
2852     ret = do_sigaction(sig, act ? &new_ka : NULL,
2853                 oact ? &old_ka : NULL);
2854
2855     if (!ret && oact) {
2856         if (verify_area(VERIFY_WRITE, oact, sizeof(*oact)) ||
2857             __put_user(old_ka.sa.sa_handler,
2858                 &oact->sa_handler) ||
2859             __put_user(old_ka.sa.sa_restorer,
2860                 &oact->sa_restorer))
2861             return -EFAULT;
2862         __put_user(old_ka.sa.sa_flags, &oact->sa_flags);
2863         __put_user(old_ka.sa.sa_mask.sig[0], &oact->sa_mask);
2864     }
2865
2866     return ret;

```

```

2867 }
2868
2869 asmlinkage int
2870 sys_sigaltstack(const stack_t *uss, stack_t *uoss)
2871 {
2872     struct pt_regs *regs = (struct pt_regs *) &uss;
2873     return do_sigaltstack(uss, uoss, regs->esp);
2874 }
2875
2876
2877 /* Do a signal return; undo the signal stack. */
2878
2879 struct sigframe
2880 {
2881     char *precode;
2882     int sig;
2883     struct sigcontext sc;
2884     struct _fpstate fpstate;
2885     unsigned long extramask[_NSIG_WORDS-1];
2886     char retcode[8];
2887 };
2888
2889 struct rt_sigframe
2890 {
2891     char *precode;
2892     int sig;
2893     struct siginfo *pinfo;
2894     void *puc;
2895     struct siginfo info;
2896     struct ucontext uc;
2897     struct _fpstate fpstate;
2898     char retcode[8];
2899 };
2900
2901
2902 static inline int restore_i387_hard(struct _fpstate *buf)
2903 {
2904     struct task_struct *tsk = current;
2905     clear_fpu(tsk);
2906     return __copy_from_user(&tsk->tss.i387.hard, buf,
2907                             sizeof(*buf));
2908 }
2909
2910 static inline int restore_i387(struct _fpstate *buf)
2911 {
2912     int err;
2913     #ifndef CONFIG_MATH_EMULATION
2914     err = restore_i387_hard(buf);
2915     #else
2916     if (boot_cpu_data.hard_math)
2917         err = restore_i387_hard(buf);
2918     else
2919         err = restore_i387_soft(&current->tss.i387.soft, buf);
2920     #endif
2921     current->used_math = 1;
2922     return err;
2923 }
2924
2925 static int
2926 restore_sigcontext(struct pt_regs *regs,
2927                   struct sigcontext *sc, int *peax)
2928 {
2929     unsigned int err = 0;
2930
2931     #define COPY(x)          err |= __get_user(regs->x, &sc->x)
2932
2933     #define COPY_SEG(seg) \
2934     { unsigned short tmp; \
2935       err |= __get_user(tmp, &sc->seg); \
2936       regs->x##seg = tmp; }
2937
2938     #define COPY_SEG_STRICT(seg) \
2939     { unsigned short tmp; \
2940       err |= __get_user(tmp, &sc->seg); \
2941       regs->x##seg = tmp|3; }
2942
2943     #define GET_SEG(seg) \
2944     { unsigned short tmp; \
2945       err |= __get_user(tmp, &sc->seg); \
2946       loadsegment(seg, tmp); }
2947
2948     GET_SEG(gs);
2949     GET_SEG(fs);
2950     COPY_SEG(es);
2951     COPY_SEG(ds);
2952     COPY(edi);
2953     COPY(esi);
2954     COPY(ebp);
2955     COPY(esp);
2956     COPY(ebx);
2957     COPY(edx);
2958     COPY(ecx);
2959     COPY(eip);
2960     COPY_SEG_STRICT(cs);
2961     COPY_SEG_STRICT(ss);
2962

```

```

2963 {
2964     unsigned int tmpflags;
2965     err |= __get_user(tmpflags, &sc->eflags);
2966     regs->eflags = (regs->eflags & ~0x40DD5) |
2967                 (tmpflags & 0x40DD5);
2968     regs->orig_eax = -1; /* disable syscall checks */
2969 }
2970
2971 {
2972     struct _fpstate * buf;
2973     err |= __get_user(buf, &sc->fpstate);
2974     if (buf) {
2975         if (verify_area(VERIFY_READ, buf, sizeof(*buf)))
2976             goto badframe;
2977         err |= restore_i387(buf);
2978     }
2979 }
2980
2981 err |= __get_user(*peax, &sc->eax);
2982 return err;
2983
2984 badframe:
2985 return 1;
2986 }
2987
2988 asmlinkage int sys_sigreturn(unsigned long __unused)
2989 {
2990     struct pt_regs *regs = (struct pt_regs *) &__unused;
2991     struct sigframe *frame =
2992         (struct sigframe *) (regs->esp - 8);
2993     sigset_t set;
2994     int eax;
2995
2996     if (verify_area(VERIFY_READ, frame, sizeof(*frame)))
2997         goto badframe;
2998     if (__get_user(set.sig[0], &frame->sc.oldmask)
2999         || (_NSIG_WORDS > 1
3000         && __copy_from_user(&set.sig[1], &frame->extramask,
3001             sizeof(frame->extramask))))
3002         goto badframe;
3003
3004     sigdelsetmask(&set, ~_BLOCKABLE);
3005     spin_lock_irq(&current->sigmask_lock);
3006     current->blocked = set;
3007     recalc_sigpending(current);
3008     spin_unlock_irq(&current->sigmask_lock);
3009
3010     if (restore_sigcontext(regs, &frame->sc, &eax))

```

```

3011         goto badframe;
3012     return eax;
3013
3014 badframe:
3015     force_sig(SIGSEGV, current);
3016     return 0;
3017 }
3018
3019 asmlinkage int sys_rt_sigreturn(unsigned long __unused)
3020 {
3021     struct pt_regs *regs = (struct pt_regs *) &__unused;
3022     struct rt_sigframe *frame =
3023         (struct rt_sigframe *) (regs->esp - 4);
3024     sigset_t set;
3025     stack_t st;
3026     int eax;
3027
3028     if (verify_area(VERIFY_READ, frame, sizeof(*frame)))
3029         goto badframe;
3030     if (__copy_from_user(&set, &frame->uc.uc_sigmask,
3031         sizeof(set)))
3032         goto badframe;
3033
3034     sigdelsetmask(&set, ~_BLOCKABLE);
3035     spin_lock_irq(&current->sigmask_lock);
3036     current->blocked = set;
3037     recalc_sigpending(current);
3038     spin_unlock_irq(&current->sigmask_lock);
3039
3040     if (restore_sigcontext(regs, &frame->uc.uc_mcontext,
3041         &eax))
3042         goto badframe;
3043
3044     if (__copy_from_user(&st, &frame->uc.uc_stack,
3045         sizeof(st)))
3046         goto badframe;
3047     /* It is more difficult to avoid calling this function
3048      * than to call it and ignore errors. */
3049     do_sigaltstack(&st, NULL, regs->esp);
3050
3051     return eax;
3052
3053 badframe:
3054     force_sig(SIGSEGV, current);
3055     return 0;
3056 }
3057
3058 /* Set up a signal frame. */

```

```

3059
3060 static inline int save_i387_hard(struct _fpstate * buf)
3061 {
3062     struct task_struct *tsk = current;
3063
3064     unlazy_fpu(tsk);
3065     tsk->tss.i387.hard.status = tsk->tss.i387.hard.swd;
3066     if (__copy_to_user(buf, &tsk->tss.i387.hard,
3067                       sizeof(*buf)))
3068         return -1;
3069     return 1;
3070 }
3071
3072 static int save_i387(struct _fpstate *buf)
3073 {
3074     if (!current->used_math)
3075         return 0;
3076
3077     /* This will cause a "finit" to be triggered by the
3078      * next attempted FPU operation by the 'current'
3079      * process. */
3080     current->used_math = 0;
3081
3082     #ifndef CONFIG_MATH_EMULATION
3083     return save_i387_hard(buf);
3084     #else
3085     return boot_cpu_data.hard_math ? save_i387_hard(buf)
3086         : save_i387_soft(&current->tss.i387.soft, buf);
3087     #endif
3088 }
3089
3090 static int
3091 setup_sigcontext(struct sigcontext *sc,
3092                 struct _fpstate *fpstate,
3093                 struct pt_regs *regs, unsigned long mask)
3094 {
3095     int tmp, err = 0;
3096
3097     tmp = 0;
3098     __asm__("movl %%gs,%w0" : "=r"(tmp): "0"(tmp));
3099     err |= __put_user(tmp, (unsigned int *)&sc->gs);
3100     __asm__("movl %%fs,%w0" : "=r"(tmp): "0"(tmp));
3101     err |= __put_user(tmp, (unsigned int *)&sc->fs);
3102
3103     err |= __put_user(regs->xes, (unsigned int *)&sc->es);
3104     err |= __put_user(regs->xds, (unsigned int *)&sc->ds);
3105     err |= __put_user(regs->edi, &sc->edi);
3106     err |= __put_user(regs->esi, &sc->esi);

```

```

3107     err |= __put_user(regs->ebp, &sc->ebp);
3108     err |= __put_user(regs->esp, &sc->esp);
3109     err |= __put_user(regs->ebx, &sc->ebx);
3110     err |= __put_user(regs->edx, &sc->edx);
3111     err |= __put_user(regs->ecx, &sc->ecx);
3112     err |= __put_user(regs->eax, &sc->eax);
3113     err |= __put_user(current->tss.trap_no, &sc->trapno);
3114     err |= __put_user(current->tss.error_code, &sc->err);
3115     err |= __put_user(regs->eip, &sc->eip);
3116     err |= __put_user(regs->xcs, (unsigned int *)&sc->cs);
3117     err |= __put_user(regs->eflags, &sc->eflags);
3118     err |= __put_user(regs->esp, &sc->esp_at_signal);
3119     err |= __put_user(regs->xss, (unsigned int *)&sc->ss);
3120
3121     tmp = save_i387(fpstate);
3122     if (tmp < 0)
3123         err = 1;
3124     else
3125         err |= __put_user(tmp ? fpstate : NULL, &sc->fpstate);
3126
3127     /* non-iBCS2 extensions.. */
3128     err |= __put_user(mask, &sc->oldmask);
3129     err |= __put_user(current->tss.cr2, &sc->cr2);
3130
3131     return err;
3132 }
3133
3134 /* Determine which stack to use.. */
3135 static inline void *
3136 get_sigframe(struct k_sigaction *ka,
3137              struct pt_regs *regs, size_t frame_size)
3138 {
3139     unsigned long esp;
3140
3141     /* Default to using normal stack */
3142     esp = regs->esp;
3143
3144     /* This is the X/Open sanctioned signal stack
3145      * switching. */
3146     if (ka->sa.sa_flags & SA_ONSTACK) {
3147         if (!on_sig_stack(esp))
3148             esp = current->sas_ss_sp + current->sas_ss_size;
3149     }
3150
3151     /* This is the legacy signal stack switching. */
3152     else if ((regs->xss & 0xffff) != __USER_DS &&
3153             !(ka->sa.sa_flags & SA_RESTORER) &&
3154             ka->sa.sa_restorer) {

```

```

3155     esp = (unsigned long) ka->sa.sa_restorer;
3156 }
3157
3158 return (void *)((esp - frame_size) & -8ul);
3159 }
3160
3161 static void setup_frame(int sig, struct k_sigaction *ka,
3162     sigset_t *set, struct pt_regs * regs)
3163 {
3164     struct sigframe *frame;
3165     int err = 0;
3166
3167     frame = get_sigframe(ka, regs, sizeof(*frame));
3168
3169     if (!access_ok(VERIFY_WRITE, frame, sizeof(*frame)))
3170         goto give_sigsegv;
3171
3172     err |= __put_user((current->exec_domain
3173         && current->exec_domain->signal_invmmap
3174         && sig < 32
3175         ? current->exec_domain->signal_invmmap[sig]
3176         : sig),
3177         &frame->sig);
3178
3179     err |= setup_sigcontext(&frame->sc, &frame->fpstate,
3180         regs, set->sig[0]);
3181
3182     if (_NSIG_WORDS > 1) {
3183         err |= __copy_to_user(frame->extramask, &set->sig[1],
3184             sizeof(frame->extramask));
3185     }
3186
3187     /* Set up to return from userspace.  If provided, use a
3188      * stub already in userspace.  */
3189     if (ka->sa.sa_flags & SA_RESTORER) {
3190         err |= __put_user(ka->sa.sa_restorer,
3191             &frame->pretcode);
3192     } else {
3193         err |= __put_user(frame->retcode, &frame->pretcode);
3194         /* This is popl %eax ; movl $,%eax ; int $0x80 */
3195         err |= __put_user(0xb85b,
3196             (short *) (frame->retcode+0));
3197         err |= __put_user(__NR_sigreturn,
3198             (int *) (frame->retcode+2));
3199         err |= __put_user(0x80cd,
3200             (short *) (frame->retcode+6));
3201     }
3202 }

```

```

3203     if (err)
3204         goto give_sigsegv;
3205
3206     /* Set up registers for signal handler */
3207     regs->esp = (unsigned long) frame;
3208     regs->eip = (unsigned long) ka->sa.sa_handler;
3209
3210     set_fs(USER_DS);
3211     regs->xds = __USER_DS;
3212     regs->xes = __USER_DS;
3213     regs->xss = __USER_DS;
3214     regs->xcs = __USER_CS;
3215     regs->eflags &= ~TF_MASK;
3216
3217     #if DEBUG_SIG
3218     printk("SIG deliver (%s:%d): sp=%p pc=%p ra=%p\n",
3219         current->comm, current->pid, frame, regs->eip,
3220         frame->pretcode);
3221     #endif
3222
3223     return;
3224
3225 give_sigsegv:
3226     if (sig == SIGSEGV)
3227         ka->sa.sa_handler = SIG_OFL;
3228     force_sig(SIGSEGV, current);
3229 }
3230
3231 static void setup_rt_frame(int sig,
3232     struct k_sigaction *ka, siginfo_t *info,
3233     sigset_t *set, struct pt_regs * regs)
3234 {
3235     struct rt_sigframe *frame;
3236     int err = 0;
3237
3238     frame = get_sigframe(ka, regs, sizeof(*frame));
3239
3240     if (!access_ok(VERIFY_WRITE, frame, sizeof(*frame)))
3241         goto give_sigsegv;
3242
3243     err |= __put_user((current->exec_domain
3244         && current->exec_domain->signal_invmmap
3245         && sig < 32
3246         ? current->exec_domain->signal_invmmap[sig]
3247         : sig),
3248         &frame->sig);
3249     err |= __put_user(&frame->info, &frame->pinfo);
3250     err |= __put_user(&frame->uc, &frame->puc);

```

```

3251 err |= __copy_to_user(&frame->info, info, sizeof(*info));
3252
3253 /* Create the ucontext. */
3254 err |= __put_user(0, &frame->uc.uc_flags);
3255 err |= __put_user(0, &frame->uc.uc_link);
3256 err |= __put_user(current->sas_ss_sp,
3257                  &frame->uc.uc_stack.ss_sp);
3258 err |= __put_user(sas_ss_flags(regs->esp),
3259                  &frame->uc.uc_stack.ss_flags);
3260 err |= __put_user(current->sas_ss_size,
3261                  &frame->uc.uc_stack.ss_size);
3262 err |= setup_sigcontext(&frame->uc.uc_mcontext,
3263                        &frame->fpstate,
3264                        regs, set->sig[0]);
3265 err |= __copy_to_user(&frame->uc.uc_sigmask, set,
3266                      sizeof(*set));
3267
3268 /* Set up to return from userspace. If provided, use a
3269  * stub already in userspace. */
3270 if (ka->sa.sa_flags & SA_RESTORER) {
3271     err |= __put_user(ka->sa.sa_restorer,
3272                     &frame->precode);
3273 } else {
3274     err |= __put_user(frame->retcode, &frame->precode);
3275     /* This is movl $,%eax ; int $0x80 */
3276     err |= __put_user(0xb8, (char*)(frame->retcode+0));
3277     err |= __put_user(__NR_rt_sigreturn,
3278                     (int*)(frame->retcode+1));
3279     err |= __put_user(0x80cd,
3280                     (short*)(frame->retcode+5));
3281 }
3282
3283 if (err)
3284     goto give_sigsegv;
3285
3286 /* Set up registers for signal handler */
3287 regs->esp = (unsigned long) frame;
3288 regs->eip = (unsigned long) ka->sa.sa_handler;
3289
3290 set_fs(USER_DS);
3291 regs->xds = __USER_DS;
3292 regs->xes = __USER_DS;
3293 regs->xss = __USER_DS;
3294 regs->xcs = __USER_CS;
3295 regs->eflags &= ~TF_MASK;
3296
3297 #if DEBUG_SIG
3298 printk("SIG deliver (%s:%d): sp=%p pc=%p ra=%p\n",

```

```

3299         current->comm, current->pid, frame, regs->eip,
3300         frame->precode);
3301 #endif
3302
3303 return;
3304
3305 give_sigsegv:
3306 if (sig == SIGSEGV)
3307     ka->sa.sa_handler = SIG_DFL;
3308 force_sig(SIGSEGV, current);
3309 }
3310
3311 /* OK, we're invoking a handler */
3312
3313 static void
3314 handle_signal(unsigned long sig, struct k_sigaction *ka,
3315              siginfo_t *info, sigset_t *oldset, struct pt_regs * regs)
3316 {
3317     /* Are we from a system call? */
3318     if (regs->orig_eax >= 0) {
3319         /* If so, check system call restarting.. */
3320         switch (regs->eax) {
3321             case -ERESTARTNOHAND:
3322                 regs->eax = -EINTR;
3323                 break;
3324
3325             case -ERESTARTSYS:
3326                 if (!(ka->sa.sa_flags & SA_RESTART)) {
3327                     regs->eax = -EINTR;
3328                     break;
3329                 }
3330                 /* fallthrough */
3331             case -ERESTARTNOINTR:
3332                 regs->eax = regs->orig_eax;
3333                 regs->eip -= 2;
3334         }
3335     }
3336
3337     /* Set up the stack frame */
3338     if (ka->sa.sa_flags & SA_SIGINFO)
3339         setup_rt_frame(sig, ka, info, oldset, regs);
3340     else
3341         setup_frame(sig, ka, oldset, regs);
3342
3343     if (ka->sa.sa_flags & SA_ONESHOT)
3344         ka->sa.sa_handler = SIG_DFL;
3345
3346     if (!(ka->sa.sa_flags & SA_NODEFER)) {

```



```

3347 spin_lock_irq(&current->sigmask_lock);
3348 sigorsets(&current->blocked,&current->blocked,
3349           &ka->sa.sa_mask);
3350 sigaddset(&current->blocked,sig);
3351 recal_sigpending(current);
3352 spin_unlock_irq(&current->sigmask_lock);
3353 }
3354 }
3355
3356 /* Note that 'init' is a special process: it doesn't get
3357 * signals it doesn't want to handle. Thus you cannot
3358 * kill init even with a SIGKILL even by mistake.
3359 *
3360 * Note that we go through the signals twice: once to
3361 * check the signals that the kernel can handle, and then
3362 * we build all the user-level signal handling
3363 * stack-frames in one go after that. */
3364 int do_signal(struct pt_regs *regs, sigset_t *oldset)
3365 {
3366     siginfo_t info;
3367     struct k_sigaction *ka;
3368
3369     /* We want the common case to go fast, which is why we
3370     * may in certain cases get here from kernel mode. Just
3371     * return without doing anything if so. */
3372     if ((regs->xcs & 3) != 3)
3373         return 1;
3374
3375     if (!oldset)
3376         oldset = &current->blocked;
3377
3378     for (;;) {
3379         unsigned long signr;
3380
3381         spin_lock_irq(&current->sigmask_lock);
3382         signr = dequeue_signal(&current->blocked, &info);
3383         spin_unlock_irq(&current->sigmask_lock);
3384
3385         if (!signr)
3386             break;
3387
3388         if ((current->flags & PF_PTRACED) &&
3389             signr != SIGKILL) {
3390             /* Let the debugger run. */
3391             current->exit_code = signr;
3392             current->state = TASK_STOPPED;
3393             notify_parent(current, SIGCHLD);
3394             schedule();

```

```

3395
3396     /* We're back. Did the debugger cancel the sig? */
3397     if (!(signr = current->exit_code))
3398         continue;
3399     current->exit_code = 0;
3400
3401     /* The debugger continued. Ignore SIGSTOP. */
3402     if (signr == SIGSTOP)
3403         continue;
3404
3405     /* Update the siginfo structure. Is this good? */
3406     if (signr != info.si_signo) {
3407         info.si_signo = signr;
3408         info.si_errno = 0;
3409         info.si_code = SI_USER;
3410         info.si_pid = current->p_pptr->pid;
3411         info.si_uid = current->p_pptr->uid;
3412     }
3413
3414     /* If (new) signal is now blocked, requeue it. */
3415     if (sigismember(&current->blocked, signr)) {
3416         send_sig_info(signr, &info, current);
3417         continue;
3418     }
3419 }
3420
3421 ka = &current->sig->action[signr-1];
3422 if (ka->sa.sa_handler == SIG_IGN) {
3423     if (signr != SIGCHLD)
3424         continue;
3425     /* Check for SIGCHLD: it's special. */
3426     while (sys_wait4(-1, NULL, WNOHANG, NULL) > 0)
3427         /* nothing */;
3428     continue;
3429 }
3430
3431 if (ka->sa.sa_handler == SIG_DFL) {
3432     int exit_code = signr;
3433
3434     /* Init gets no signals it doesn't want. */
3435     if (current->pid == 1)
3436         continue;
3437
3438     switch (signr) {
3439     case SIGCONT: case SIGCHLD: case SIGWINCH:
3440         continue;
3441
3442     case SIGTSTP: case SIGTTIN: case SIGTTOU:

```

```

3443     if (is_orphaned_pgrp(current->pgrp))
3444         continue;
3445     /* FALLTHRU */
3446
3447 case SIGSTOP:
3448     current->state = TASK_STOPPED;
3449     current->exit_code = signr;
3450     if (!(current->p_pptr->sig->action[SIGCHLD-1].
3451         sa.sa_flags & SA_NOCLDSTOP))
3452         notify_parent(current, SIGCHLD);
3453     schedule();
3454     continue;
3455
3456 case SIGQUIT: case SIGILL: case SIGTRAP:
3457 case SIGABRT: case SIGFPE: case SIGSEGV:
3458     lock_kernel();
3459     if (current->binfmt
3460         && current->binfmt->core_dump
3461         && current->binfmt->core_dump(signr, regs))
3462         exit_code |= 0x80;
3463     unlock_kernel();
3464     /* FALLTHRU */
3465
3466 default:
3467     lock_kernel();
3468     sigaddset(&current->signal, signr);
3469     current->flags |= PF_SIGNALED;
3470     do_exit(exit_code);
3471     /* NOTREACHED */
3472 }
3473 }
3474
3475 /* Whee! Actually deliver the signal. */
3476 handle_signal(signr, ka, &info, oldset, regs);
3477 return 1;
3478 }
3479
3480 /* Did we come from a system call? */
3481 if (regs->orig_eax >= 0) {
3482     /* Restart the system call - no handlers present */
3483     if (regs->eax == -ERESTARTNOHAND ||
3484         regs->eax == -ERESTARTSYS ||
3485         regs->eax == -ERESTARTNOINTR) {
3486         regs->eax = regs->orig_eax;
3487         regs->eip -- 2;
3488     }
3489 }
3490 return 0;
3491 }

```

arch/i386/kernel/smp.c

```

3492 /*
3493 * Intel MP v1.1/v1.4 specification support routines
3494 * for multi-pentium hosts.
3495 *
3496 * (c) 1995 Alan Cox, CymruNET Ltd <alan@cymru.net>
3497 * (c) 1998 Ingo Molnar
3498 *
3499 * Supported by Caldera http://www.caldera.com.
3500 * Much of the core SMP work is based on previous
3501 * work by Thomas Radke, to whom a great many thanks
3502 * are extended.
3503 *
3504 * Thanks to Intel for making available several
3505 * different Pentium, Pentium Pro and
3506 * Pentium-II/Xeon MP machines.
3507 *
3508 * This code is released under the GNU public
3509 * license version 2 or later.
3510 *
3511 * Fixes
3512 * Felix Koop : NR_CPUS used properly
3513 * Jose Renau : Handle single CPU case.
3514 * Alan Cox : By repeated request 8) -
3515 * Total BogoMIP report.
3516 * Greg Wright : Fix for kernel stacks panic.
3517 * Erich Boleyn : MP v1.4 and additional changes.
3518 * Matthias Sattler : Changes for 2.1 kernel map.
3519 * Michel Lespinasse: Changes for 2.1 kernel map.
3520 * Michael Chastain : Change trampoline.S to gnu as.
3521 * Alan Cox : Dumb bug: 'B' step PPro's are fine
3522 * Ingo Molnar : Added APIC timers, based on code
3523 * from Jose Renau
3524 * Alan Cox : Added EBDA scanning
3525 * Ingo Molnar : various cleanups and rewrites */
3526
3527 #include <linux/config.h>
3528 #include <linux/mm.h>
3529 #include <linux/kernel_stat.h>
3530 #include <linux/delay.h>
3531 #include <linux/mc146818rtc.h>
3532 #include <linux/smp_lock.h>
3533 #include <linux/init.h>
3534 #include <asm/mtrr.h>
3535
3536 #include "irq.h"
3537
3538 extern unsigned long start_kernel;

```

```

3539 extern void update_one_process( struct task_struct *p,
3540     unsigned long ticks, unsigned long user,
3541     unsigned long system, int cpu);
3542 /*     Some notes on processor bugs:
3543 *
3544 *     Pentium and Pentium Pro (and all CPUs) have
3545 *     bugs. The Linux issues for SMP are handled as
3546 *     follows.
3547 *
3548 * Pentium Pro:
3549 * Occasional delivery of 'spurious interrupt' as trap
3550 * #16. This is very rare. The kernel logs the event and
3551 * recovers
3552 *
3553 * Pentium:
3554 * There is a marginal case where REP MOVS on 100MHz SMP
3555 * machines with B stepping processors can fail. XXX
3556 * should provide an L1cache=WriteThrough or L1cache=off
3557 * option.
3558 *
3559 * B stepping CPUs may hang. There are hardware work
3560 * arounds for this. We warn about it in case your board
3561 * doesnt have the work arounds. Basically thats so I can
3562 * tell anyone with a B stepping CPU and SMP problems
3563 * "tough".
3564 *
3565 *     Specific items [From Pentium Processor
3566 *     Specification Update]
3567 *
3568 *     1AP. Linux doesn't use remote read
3569 *     2AP. Linux doesn't trust APIC errors
3570 *     3AP. We work around this
3571 *     4AP. Linux never generated 3 interrupts of the
3572 *     same pri to cause a lost local interrupt.
3573 *     5AP. Remote read is never used
3574 *     9AP. XXX NEED TO CHECK WE HANDLE THIS XXX
3575 *     10AP. XXX NEED TO CHECK WE HANDLE THIS XXX
3576 *     11AP. Linux reads the APIC between writes to
3577 *     avoid this, as per the documentation. Make
3578 *     sure you preserve this as it affects the C
3579 *     stepping chips too.
3580 *
3581 *     If this sounds worrying believe me these bugs are
3582 *     ___RARE___ and there's about nothing of note with
3583 *     C stepping upwards. */
3584
3585
3586 /* Kernel spinlock */

```

```

3587 spinlock_t kernel_flag = SPIN_LOCK_UNLOCKED;
3588
3589 /* function prototypes: */
3590
3591 static void cache_APIC_registers (void);
3592 static void stop_this_cpu (void);
3593
3594 /* Set if we find a B stepping CPU */
3595 static int smp_b_stepping = 0;
3596
3597 /* Setup configured maximum number of CPUs to activate */
3598 static int max_cpus = -1;
3599 /* Have we found an SMP box */
3600 int smp_found_config=0;
3601
3602 /* Bitmask of physically existing CPUs */
3603 unsigned long cpu_present_map = 0;
3604 /* Bitmask of currently online CPUs */
3605 unsigned long cpu_online_map = 0;
3606 /* Total count of live CPUs */
3607 int smp_num_cpus = 1;
3608 /* Set when the idlers are all forked */
3609 int smp_threads_ready=0;
3610 /* which CPU maps to which logical number */
3611 volatile int cpu_number_map[NR_CPUS];
3612 /* which logical number maps to which CPU */
3613 volatile int __cpu_logical_map[NR_CPUS];
3614 /* We always use 0 the rest is ready for parallel
3615  * delivery */
3616 static volatile
3617 unsigned long cpu_callin_map[NR_CPUS] = {0,};
3618 /* We always use 0 the rest is ready for parallel
3619  * delivery */
3620 static volatile
3621 unsigned long cpu_callout_map[NR_CPUS] = {0,};
3622 /* Used for the invalidate map that's also checked in the
3623  * spinlock */
3624 volatile unsigned long smp_invalidate_needed;
3625 /* Stack vector for booting CPUs */
3626 volatile unsigned long kstack_ptr;
3627 /* Per CPU bogomips and other parameters */
3628 struct cpuinfo_x86 cpu_data[NR_CPUS];
3629 /* Internal processor count */
3630 static unsigned int num_processors = 1;
3631 /* Address of the I/O apic (not yet used) */
3632 unsigned long mp_ioapic_addr = 0xFEC00000;
3633 /* Processor that is doing the boot up */
3634 unsigned char boot_cpu_id = 0;

```

```

3635 /* Tripped once we need to start cross invalidating */
3636 static int smp_activated = 0;
3637 /* APIC version number */
3638 int apic_version[NR_CPUS];
3639 /* Just debugging the assembler.. */
3640 unsigned long apic_retval;
3641
3642 /* Number of times the processor holds the lock */
3643 volatile unsigned long kernel_counter=0;
3644 /* Number of times the processor holds the syscall lock*/
3645 volatile unsigned long syscall_count=0;
3646
3647 /* Number of IPIs delivered */
3648 volatile unsigned long ipi_count;
3649
3650 const char lk_lockmsg[] =
3651     "lock from interrupt context at %p\n";
3652
3653 int mp_bus_id_to_type [MAX_MP_BUSSES] = { -1, };
3654 extern int mp_irq_entries;
3655 extern struct mpc_config_intsrc mp_irqs[MAX_IRQ_SOURCES];
3656 extern int mpc_default_type;
3657 int mp_bus_id_to_pci_bus [MAX_MP_BUSSES] = { -1, };
3658 int mp_current_pci_id = 0;
3659 unsigned long mp_lapic_addr = 0;
3660 /* 1 if "noapic" boot option passed */
3661 int skip_ioapic_setup = 0;
3662
3663 /* #define SMP_DEBUG */
3664
3665 #ifdef SMP_DEBUG
3666 #define SMP_PRINTK(x)    printk x
3667 #else
3668 #define SMP_PRINTK(x)
3669 #endif
3670
3671 /* IA s/w dev Vol 3, Section 7.4 */
3672 #define APIC_DEFAULT_PHYS_BASE 0xf0000000
3673
3674 /* Reads and clears the Pentium Timestamp-Counter */
3675 #define READ_TSC(x)    __asm__ __volatile__ ( "rdtsc" \
3676     : "=a" (((unsigned long*)&(x))[0]), \
3677     : "=d" (((unsigned long*)&(x))[1])) \
3678     :
3679 #define CLEAR_TSC \
3680     __asm__ __volatile__ ("\t.byte 0x0f, 0x30;\n":: \
3681     : "a"(0x00001000), "d"(0x00001000), "c"(0x10):"memory")
3682

```

```

3683 /*      Setup routine for controlling SMP activation
3684 *
3685 *      Command-line option of "nosmp" or "maxcpus=0"
3686 *      will disable SMP activation entirely (the MPS
3687 *      table probe still happens, though).
3688 *
3689 *      Command-line option of "maxcpus=<NUM>", where
3690 *      <NUM> is an integer greater than 0, limits the
3691 *      maximum number of CPUs activated in SMP mode to
3692 *      <NUM>. */
3693
3694 void __init smp_setup(char *str, int *ints)
3695 {
3696     if (ints && ints[0] > 0)
3697         max_cpus = ints[1];
3698     else
3699         max_cpus = 0;
3700 }
3701
3702 void ack_APIC_irq(void)
3703 {
3704     /* Clear the IPI */
3705
3706     /* Dummy read */
3707     apic_read(APIC_SPIV);
3708
3709     /* Docs say use 0 for future compatibility */
3710     apic_write(APIC_EOI, 0);
3711 }
3712
3713 /* Intel MP BIOS table parsing routines: */
3714
3715 #ifndef CONFIG_X86_VISWS_APIC
3716 /* Checksum an MP configuration block. */
3717
3718 static int mpf_checksum(unsigned char *mp, int len)
3719 {
3720     int sum=0;
3721     while(len--)
3722         sum+=*mp++;
3723     return sum&0xFF;
3724 }
3725
3726 /* Processor encoding in an MP configuration block */
3727
3728 static char *mpc_family(int family,int model)
3729 {
3730     static char n[32];

```

```

3731 static char *model_defs[] =
3732 {
3733     "80486DX", "80486DX",
3734     "80486SX", "80486DX/2 or 80487",
3735     "80486SL", "Intel5X2(tm)",
3736     "Unknown", "Unknown",
3737     "80486DX/4"
3738 };
3739 if (family == 0x6)
3740     return("Pentium(tm) Pro");
3741 if (family == 0x5)
3742     return("Pentium(tm)");
3743 if (family == 0x0F && model == 0x0F)
3744     return("Special controller");
3745 if (family == 0x04 && model < 9)
3746     return model_defs[model];
3747 sprintf(n, "Unknown CPU [%d:%d]", family, model);
3748 return n;
3749 }
3750
3751 /* Read the MPC */
3752
3753 static int __init
3754 smp_read_mpc(struct mp_config_table *mpc)
3755 {
3756     char str[16];
3757     int count = sizeof(*mpc);
3758     int ioapics = 0;
3759     unsigned char *mpt = ((unsigned char *)mpc) + count;
3760
3761     if (memcmp(mpc->mpc_signature, MPC_SIGNATURE, 4))
3762     {
3763         panic("SMP mptable: bad signature [%c%c%c%c]!\n",
3764             mpc->mpc_signature[0],
3765             mpc->mpc_signature[1],
3766             mpc->mpc_signature[2],
3767             mpc->mpc_signature[3]);
3768         return 1;
3769     }
3770     if (mpf_checksum((unsigned char *)mpc, mpc->mpc_length))
3771     {
3772         panic("SMP mptable: checksum error!\n");
3773         return 1;
3774     }
3775     if (mpc->mpc_spec != 0x01 && mpc->mpc_spec != 0x04)
3776     {
3777         printk("Bad Config Table version (%d)!!\n",
3778             mpc->mpc_spec);

```

```

3779         return 1;
3780     }
3781     memcpy(str, mpc->mpc_oem, 8);
3782     str[8] = 0;
3783     memcpy(ioapic_OEM_ID, str, 9);
3784     printk("OEM ID: %s ", str);
3785
3786     memcpy(str, mpc->mpc_productid, 12);
3787     str[12] = 0;
3788     memcpy(ioapic_Product_ID, str, 13);
3789     printk("Product ID: %s ", str);
3790
3791     printk("APIC at: 0x%lx\n", mpc->mpc_lapic);
3792
3793     /* save the local APIC address, it might be
3794      * non-default */
3795     mp_lapic_addr = mpc->mpc_lapic;
3796
3797     /* Now process the configuration blocks. */
3798
3799     while(count < mpc->mpc_length)
3800     {
3801         switch(*mpt)
3802         {
3803             case MP_PROCESSOR:
3804             {
3805                 struct mpc_config_processor *m =
3806                     (struct mpc_config_processor *)mpt;
3807                 if (m->mpc_cpuflag & CPU_ENABLED)
3808                 {
3809                     printk("Processor # %d %s APIC version %d\n",
3810                         m->mpc_apicid,
3811                         mpc_family((m->mpc_cpufeature &
3812                             CPU_FAMILY_MASK) >> 8,
3813                             (m->mpc_cpufeature &
3814                             CPU_MODEL_MASK) >> 4),
3815                         m->mpc_apicver);
3816 #ifdef SMP_DEBUG
3817                     if (m->mpc_featureflag & (1 << 0))
3818                         printk("    Floating point unit present.\n");
3819                     if (m->mpc_featureflag & (1 << 7))
3820                         printk("    Machine Exception supported.\n");
3821                     if (m->mpc_featureflag & (1 << 8))
3822                         printk("    64 bit compare & exchange "
3823                             "supported.\n");
3824                     if (m->mpc_featureflag & (1 << 9))
3825                         printk("    Internal APIC present.\n");
3826 #endif

```

```

3827     if (m->mpc_cpuflag&CPU_BOOTPROCESSOR)
3828     {
3829         SMP_PRINTK(("  Bootup CPU\n"));
3830         boot_cpu_id=m->mpc_apicid;
3831     }
3832     else /* Boot CPU already counted */
3833         num_processors++;
3834
3835     if (m->mpc_apicid>NR_CPUS)
3836         printk("Processor %#d unused. (Max %d "
3837             "processors).\n",m->mpc_apicid, NR_CPUS);
3838     else
3839     {
3840         int ver = m->mpc_apicver;
3841
3842         cpu_present_map|=(1<<m->mpc_apicid);
3843         /* Validate version */
3844         if (ver == 0x0) {
3845             printk("BIOS bug, APIC version is 0 for "
3846                 "CPU#%d! fixing up to 0x10. (tell "
3847                 "your hw vendor)\n", m->mpc_apicid);
3848             ver = 0x10;
3849         }
3850         apic_version[m->mpc_apicid] = ver;
3851     }
3852 }
3853 mpt+=sizeof(*m);
3854 count+=sizeof(*m);
3855 break;
3856 }
3857 case MP_BUS:
3858 {
3859     struct mpc_config_bus *m=
3860         (struct mpc_config_bus *)mpt;
3861     memcpy(str,m->mpc_bustype,6);
3862     str[6]=0;
3863     SMP_PRINTK(("Bus %#d is %s\n",
3864         m->mpc_busid,
3865         str));
3866     if ((strcmp(m->mpc_bustype,"ISA",3) == 0) ||
3867         (strcmp(m->mpc_bustype,"EISA",4) == 0))
3868         mp_bus_id_to_type[m->mpc_busid] =
3869             MP_BUS_ISA;
3870     else
3871     if (strcmp(m->mpc_bustype,"PCI",3) == 0) {
3872         mp_bus_id_to_type[m->mpc_busid] =
3873             MP_BUS_PCI;
3874         mp_bus_id_to_pci_bus[m->mpc_busid] =
3875             mp_current_pci_id;
3876         mp_current_pci_id++;
3877     }
3878     mpt+=sizeof(*m);
3879     count+=sizeof(*m);
3880     break;
3881 }
3882 case MP_IOAPIC:
3883 {
3884     struct mpc_config_ioapic *m=
3885         (struct mpc_config_ioapic *)mpt;
3886     if (m->mpc_flags&MPC_APIC_USABLE)
3887     {
3888         ioapics++;
3889         printk("I/O APIC %#d Version %d at 0x%lx.\n",
3890             m->mpc_apicid,m->mpc_apicver,
3891             m->mpc_apicaddr);
3892         /* we use the first one only currently */
3893         if (ioapics == 1)
3894             mp_ioapic_addr = m->mpc_apicaddr;
3895     }
3896     mpt+=sizeof(*m);
3897     count+=sizeof(*m);
3898     break;
3899 }
3900 case MP_INTSRC:
3901 {
3902     struct mpc_config_intsrc *m=
3903         (struct mpc_config_intsrc *)mpt;
3904
3905     mp_irqs [mp_irq_entries] = *m;
3906     if (++mp_irq_entries == MAX_IRQ_SOURCES) {
3907         printk("Max irq sources exceeded!!\n");
3908         printk("Skipping remaining sources.\n");
3909         --mp_irq_entries;
3910     }
3911
3912     mpt+=sizeof(*m);
3913     count+=sizeof(*m);
3914     break;
3915 }
3916 case MP_LINTSRC:
3917 {
3918     struct mpc_config_intlocal *m=
3919         (struct mpc_config_intlocal *)mpt;
3920     mpt+=sizeof(*m);
3921     count+=sizeof(*m);
3922     break;

```

```

3923     )
3924     }
3925 }
3926 if (ioapics > 1)
3927 {
3928     printk("Warning: "
3929           "Multiple IO-APICs not yet supported.\n");
3930     printk("Warning: switching to non APIC mode.\n");
3931     skip_ioapic_setup=1;
3932 }
3933 return num_processors;
3934 }
3935
3936 /* Scan the memory blocks for an SMP configuration block.
3937 */
3938
3939 static int __init smp_scan_config(unsigned long base,
3940                                  unsigned long length)
3941 {
3942     unsigned long *bp=phys_to_virt(base);
3943     struct intel_mp_floating *mpf;
3944
3945     SMP_PRINTK(("Scan SMP from %p for %ld bytes.\n",
3946               bp,length));
3947     if (sizeof(*mpf)!=16)
3948         printk("Error: MPF size\n");
3949
3950     while (length>0)
3951     {
3952         if (*bp==SMP_MAGIC_IDENT)
3953         {
3954             mpf=(struct intel_mp_floating *)bp;
3955             if (mpf->mpf_length==1 &&
3956                 !mpf_checksum((unsigned char *)bp,16) &&
3957                 (mpf->mpf_specification == 1
3958                  || mpf->mpf_specification == 4) )
3959             {
3960                 printk("Intel MultiProcessor Specification "
3961                       "v1.%d\n", mpf->mpf_specification);
3962                 if (mpf->mpf_feature2&(1<<7))
3963                     printk("    IMCR and PIC "
3964                           "compatibility mode.\n");
3965                 else
3966                     printk("    Virtual Wire "
3967                           "compatibility mode.\n");
3968                 smp_found_config=1;
3969                 /* Now see if we need to read further. */
3970                 if (mpf->mpf_feature1!=0)

```

```

3971     {
3972         unsigned long cfg;
3973
3974         /* local APIC has default address */
3975         mp_lapic_addr = APIC_DEFAULT_PHYS_BASE;
3976         /* We need to know what the local APIC id of
3977            * the boot CPU is! */
3978
3979 /* HACK HACK HACK HACK HACK HACK HACK HACK HACK HACK HACK
3980  * It's not just a crazy hack. ;-) */
3981
3982         /* Standard page mapping functions don't work
3983            * yet. We know that page 0 is not used.
3984            * Steal it for now! */
3985
3986         cfg=pg0[0];
3987         pg0[0] = (mp_lapic_addr |
3988                 _PAGE_RW | _PAGE_PRESENT);
3989         local_flush_tlb();
3990
3991         boot_cpu_id =
3992             GET_APIC_ID(*((volatile unsigned long *)
3993                         APIC_ID));
3994
3995         /* Give it back */
3996         pg0[0]= cfg;
3997         local_flush_tlb();
3998
3999 /*
4000  * END OF HACK   END OF HACK   END OF HACK   END OF HACK
4001  */
4002         /* 2 CPUs, numbered 0 & 1. */
4003         cpu_present_map=3;
4004         num_processors=2;
4005         printk("I/O APIC at 0xFEC00000.\n");
4006
4007         /* Save the default type number, we need it
4008            * later to set the IO-APIC up properly: */
4009         mpc_default_type = mpf->mpf_feature1;
4010
4011         printk("Bus #0 is ");
4012     }
4013     switch(mpf->mpf_feature1)
4014     {
4015     case 1:
4016     case 5:
4017         printk("ISA\n");
4018         break;

```

```

4019     case 2:
4020         printk("EISA with no IRQ8 chaining\n");
4021         break;
4022     case 6:
4023     case 3:
4024         printk("EISA\n");
4025         break;
4026     case 4:
4027     case 7:
4028         printk("MCA\n");
4029         break;
4030     case 0:
4031         break;
4032     default:
4033         printk("????\nUnknown standard configuration "
4034             "%d\n", mpf->mpf_feature1);
4035         return 1;
4036     }
4037     if (mpf->mpf_feature1 > 4)
4038     {
4039         printk("Bus #1 is PCI\n");
4040
4041         /* Set local APIC version to the integrated
4042          * form. It's initialized to zero otherwise,
4043          * representing a discrete 82489DX. */
4044         apic_version[0] = 0x10;
4045         apic_version[1] = 0x10;
4046     }
4047     /* Read the physical hardware table. Anything
4048     * here will override the defaults. */
4049     if (mpf->mpf_physptr)
4050         smp_read_mpc((void *)mpf->mpf_physptr);
4051
4052     __cpu_logical_map[0] = boot_cpu_id;
4053     global_irq_holder = boot_cpu_id;
4054     current->processor = boot_cpu_id;
4055
4056     printk("Processors: %d\n", num_processors);
4057     /* Only use the first configuration found. */
4058     return 1;
4059 }
4060 }
4061 bp+=4;
4062 length-=16;
4063 }
4064
4065 return 0;
4066 }

```

```

4067
4068 void __init init_intel_smp (void)
4069 {
4070     /* FIXME: Linux assumes you have 640K of base ram..
4071     * this continues the error...
4072     *
4073     * 1) Scan the bottom 1K for a signature
4074     * 2) Scan the top 1K of base RAM
4075     * 3) Scan the 64K of bios */
4076     if (!smp_scan_config(0x0,0x400) &&
4077         !smp_scan_config(639*0x400,0x400) &&
4078         !smp_scan_config(0xF0000,0x10000)) {
4079         /* If it is an SMP machine we should know now, unless
4080         * the configuration is in an EISA/MCA bus machine
4081         * with an extended bios data area.
4082         *
4083         * there is a real-mode segmented pointer pointing to
4084         * the 4K EBDA area at 0x40E, calculate and scan it
4085         * here.
4086         *
4087         * NOTE! There are Linux loaders that will corrupt
4088         * the EBDA area, and as such this kind of SMP config
4089         * may be less trustworthy, simply because the SMP
4090         * table may have been stomped on during early
4091         * boot. These loaders are buggy and should be fixed.
4092         */
4093         unsigned int address;
4094
4095         address = *(unsigned short *)phys_to_virt(0x40E);
4096         address<<=4;
4097         smp_scan_config(address, 0x1000);
4098         if (smp_found_config)
4099             printk(KERN_WARNING "WARNING: MP table in the EBDA"
4100                 " can be UNSAFE, contact linux-smp@vger.rutgers."
4101                 "edu if you experience SMP problems!\n");
4102     }
4103 }
4104
4105 #else
4106
4107 /* The Visual Workstation is Intel MP compliant in the
4108 * hardware sense, but it doesnt have a
4109 * BIOS(-configuration table). No problem for Linux. */
4110 void __init init_visws_smp(void)
4111 {
4112     smp_found_config = 1;
4113
4114     cpu_present_map |= 2; /* or in id 1 */

```



```

4115 apic_version[1] |= 0x10; /* integrated APIC */
4116 apic_version[0] |= 0x10;
4117
4118 mp_lapic_addr = APIC_DEFAULT_PHYS_BASE;
4119 }
4120
4121 #endif
4122
4123 /* - Intel MP Configuration Table
4124 * - or SGI Visual Workstation configuration */
4125 void __init init_smp_config (void)
4126 {
4127 #ifndef CONFIG_VISWS
4128   init_intel_smp();
4129 #else
4130   init_visws_smp();
4131 #endif
4132 }
4133
4134 /* Trampoline 80x86 program as an array. */
4135
4136 extern unsigned char trampoline_data [];
4137 extern unsigned char trampoline_end [];
4138 static unsigned char *trampoline_base;
4139
4140 /* Currently trivial. Write the real->protected mode
4141 * bootstrap into the page concerned. The caller has made
4142 * sure it's suitably aligned. */
4143
4144 static unsigned long __init setup_trampoline(void)
4145 {
4146   memcpy(trampoline_base, trampoline_data,
4147          trampoline_end - trampoline_data);
4148   return virt_to_phys(trampoline_base);
4149 }
4150
4151 /* We are called very early to get the low memory for the
4152 * SMP bootup trampoline page. */
4153 unsigned long __init
4154 smp_alloc_memory(unsigned long mem_base)
4155 {
4156   if (virt_to_phys((void *)mem_base) >= 0x9F000)
4157     panic("smp_alloc_memory: Insufficient low memory for"
4158          " kernel trampoline 0x%x.", mem_base);
4159   trampoline_base = (void *)mem_base;
4160   return mem_base + PAGE_SIZE;
4161 }
4162

```

```

4163 /* The bootstrap kernel entry code has set these up. Save
4164 * them for a given CPU */
4165 void __init smp_store_cpu_info(int id)
4166 {
4167   struct cpuinfo_x86 *c=&cpu_data[id];
4168
4169   *c = boot_cpu_data;
4170   c->pte_quick = 0;
4171   c->pgd_quick = 0;
4172   c->pgtable_cache_sz = 0;
4173   identify_cpu(c);
4174   /* Mask B, Pentium, but not Pentium MMX */
4175   if (c->x86_vendor == X86_VENDOR_INTEL &&
4176       c->x86 == 5 &&
4177       c->x86_mask >= 1 && c->x86_mask <= 4 &&
4178       c->x86_model <= 3)
4179     /* Remember we have B step Pentia with bugs */
4180     smp_b_stepping=1;
4181 }
4182
4183 /* Architecture specific routine called by the kernel
4184 * just before init is fired off. This allows the BP to
4185 * have everything in order [we hope]. At the end of
4186 * this all the APs will hit the system scheduling and
4187 * off we go. Each AP will load the system gdt's and jump
4188 * through the kernel init into idle(). At this point the
4189 * scheduler will one day take over and give them jobs to
4190 * do. smp_callin is a standard routine we use to track
4191 * CPUs as they power up. */
4192
4193 static atomic_t smp_commenced = ATOMIC_INIT(0);
4194
4195 void __init smp_commence(void)
4196 {
4197   /* Lets the callins below out of their loop. */
4198   SMP_PRINTK(("Setting commenced=1, go go go\n"));
4199
4200   wmb();
4201   atomic_set(&smp_commenced,1);
4202 }
4203
4204 void __init enable_local_APIC(void)
4205 {
4206   unsigned long value;
4207
4208   value = apic_read(APIC_SPIV);
4209   value |= (1<<8); /* Enable APIC (bit==1) */
4210   value &= ~(1<<9); /* Enable focus processor (bit==0) */

```

```

4211 value |= 0xff; /* Set spurious IRQ vector to 0xff */
4212 apic_write(APIC_SPIV,value);
4213
4214 /* Set Task Priority to 'accept all' */
4215 value = apic_read(APIC_TASKPRI);
4216 value &= ~APIC_TPRI_MASK;
4217 apic_write(APIC_TASKPRI,value);
4218
4219 /* Clear the logical destination ID, just to be safe.
4220 * also, put the APIC into flat delivery mode. */
4221 value = apic_read(APIC_LDR);
4222 value &= ~APIC_LDR_MASK;
4223 apic_write(APIC_LDR,value);
4224
4225 value = apic_read(APIC_DFR);
4226 value |= SET_APIC_DFR(0xf);
4227 apic_write(APIC_DFR, value);
4228
4229 udelay(100); /* B safe */
4230 }
4231
4232 unsigned long __init
4233 init_smp_mappings(unsigned long memory_start)
4234 {
4235     unsigned long apic_phys;
4236
4237     memory_start = PAGE_ALIGN(memory_start);
4238     if (smp_found_config) {
4239         apic_phys = mp_lapic_addr;
4240     } else {
4241         /* set up a fake all zeroes page to simulate the
4242          * local APIC and another one for the IO-APIC. We
4243          * could use the real zero-page, but it's safer this
4244          * way if some buggy code writes to this page ... */
4245         apic_phys = __pa(memory_start);
4246         memset((void *)memory_start, 0, PAGE_SIZE);
4247         memory_start += PAGE_SIZE;
4248     }
4249     set_fixmap(FIX_APIC_BASE,apic_phys);
4250     printk("mapped APIC to %08lx (%08lx)\n",
4251           APIC_BASE, apic_phys);
4252
4253 #ifdef CONFIG_X86_IO_APIC
4254 {
4255     unsigned long ioapic_phys;
4256
4257     if (smp_found_config) {
4258         ioapic_phys = mp_ioapic_addr;

```

```

4259     } else {
4260         ioapic_phys = __pa(memory_start);
4261         memset((void *)memory_start, 0, PAGE_SIZE);
4262         memory_start += PAGE_SIZE;
4263     }
4264     set_fixmap(FIX_IO_APIC_BASE,ioapic_phys);
4265     printk("mapped IOAPIC to %08lx (%08lx)\n",
4266           fix_to_virt(FIX_IO_APIC_BASE), ioapic_phys);
4267 }
4268 #endif
4269
4270 return memory_start;
4271 }
4272
4273 extern void calibrate_delay(void);
4274
4275 void __init smp_callin(void)
4276 {
4277     int cpuid;
4278     unsigned long timeout;
4279
4280     /* (This works even if the APIC is not enabled.) */
4281     cpuid = GET_APIC_ID(apic_read(APIC_ID));
4282
4283     SMP_PRINTK(("CPU%d waiting for CALLOUT\n", cpuid));
4284
4285     /* STARTUP IPis are fragile beasts as they might
4286      * sometimes trigger some glue motherboard
4287      * logic. Complete APIC bus silence for 1 second, this
4288      * overestimates the time the boot CPU is spending to
4289      * send the up to 2 STARTUP IPis by a factor of
4290      * two. This should be enough. */
4291
4292     /* Waiting 2s total for startup (udelay is not yet
4293      * working) */
4294     timeout = jiffies + 2*HZ;
4295     while (time_before(jiffies,timeout))
4296     {
4297         /* Has the boot CPU finished its STARTUP sequence? */
4298         if (test_bit(cpuid,
4299                     (unsigned long *)&cpu_callout_map[0]))
4300             break;
4301     }
4302
4303     while (!time_before(jiffies,timeout)) {
4304         printk("BUG: CPU%d started up but did not get a "
4305              "callout!\n", cpuid);
4306         stop_this_cpu();

```

```

4307 }
4308
4309 /* the boot CPU has finished the init stage and is
4310  * spinning on callin_map until we finish. We are free
4311  * to set up this CPU, first the APIC. (this is
4312  * probably redundant on most boards) */
4313 SMP_PRINTK(("CALLIN, before enable_local_APIC().\n"));
4314 enable_local_APIC();
4315
4316 /* Set up our APIC timer. */
4317 setup_APIC_clock();
4318
4319 __sti();
4320
4321 #ifdef CONFIG_MTRR
4322 /* Must be done before calibration delay is computed */
4323 mtrr_init_secondary_cpu ();
4324 #endif
4325 /* Get our bogomips. */
4326 calibrate_delay();
4327 SMP_PRINTK(("Stack at about %p\n",&cpu_id));
4328
4329 /* Save our processor parameters */
4330 smp_store_cpu_info(cpu_id);
4331
4332 /* Allow the master to continue. */
4333 set_bit(cpu_id, (unsigned long *)&cpu_callin_map[0]);
4334 }
4335
4336 int cpucount = 0;
4337
4338 extern int cpu_idle(void * unused);
4339
4340 /* Activate a secondary processor. */
4341 int __init start_secondary(void *unused)
4342 {
4343 /* Don't put anything before smp_callin(), SMP booting
4344  * is too fragile that we want to limit the things done
4345  * here to the most necessary things. */
4346 smp_callin();
4347 while (!atomic_read(&smp_commenced))
4348 /* nothing */ ;
4349 return cpu_idle(NULL);
4350 }
4351
4352 /* Everything has been set up for the secondary CPUs
4353  * they just need to reload everything from the task
4354  * structure */

```

```

4355 void __init initialize_secondary(void)
4356 {
4357 struct thread_struct * p = &current->tss;
4358
4359 /* Load up the LDT and the task register. */
4360 asm volatile("ltd %%ax": : "a" (p->ldt));
4361 asm volatile("ltr %%ax": : "a" (p->tr));
4362 stts();
4363
4364 /* We don't actually need to load the full TSS,
4365  * basically just the stack pointer and the eip. */
4366
4367 asm volatile(
4368 "movl %0,%%esp\n\t"
4369 "jmp *%1"
4370 :
4371 : "r" (p->esp), "r" (p->eip));
4372 }
4373
4374 extern struct {
4375 void * esp;
4376 unsigned short ss;
4377 } stack_start;
4378
4379 static void __init do_boot_cpu(int i)
4380 {
4381 unsigned long cfg;
4382 pgd_t maincfg;
4383 struct task_struct *idle;
4384 unsigned long send_status, accept_status;
4385 int timeout, num_starts, j;
4386 unsigned long start_eip;
4387
4388 /* We need an idle process for each processor. */
4389
4390 kernel_thread(start_secondary, NULL, CLONE_PID);
4391 cpucount++;
4392
4393 idle = task[cpucount];
4394 if (!idle)
4395 panic("No idle process for CPU %d", i);
4396
4397 idle->processor = i;
4398 __cpu_logical_map[cpucount] = i;
4399 cpu_number_map[i] = cpucount;
4400
4401 /* start_eip had better be page-aligned! */
4402 start_eip = setup_trampoline();

```

```

4403
4404 /* So we see what's up */
4405 printk("Booting processor %d eip %lx\n", i, start_eip);
4406 stack_start.esp = (void *) (1024 + PAGE_SIZE +
4407                             (char *)idle);
4408
4409 /* This grunge runs the startup process for the
4410  * targeted processor. */
4411
4412 SMP_PRINTK(("Setting warm reset code and vector.\n"));
4413
4414 CMOS_WRITE(0xa, 0xf);
4415 local_flush_tlb();
4416 SMP_PRINTK(("1.\n"));
4417 *((volatile unsigned short *) phys_to_virt(0x469)) =
4418     start_eip >> 4;
4419 SMP_PRINTK(("2.\n"));
4420 *((volatile unsigned short *) phys_to_virt(0x467)) =
4421     start_eip & 0xf;
4422 SMP_PRINTK(("3.\n"));
4423
4424 maincfg=swapper_pg_dir[0];
4425 ((unsigned long *)swapper_pg_dir)[0]=0x102007;
4426
4427 /* Be paranoid about clearing APIC errors. */
4428
4429 if ( apic_version[i] & 0xF0 )
4430 {
4431     apic_write(APIC_ESR, 0);
4432     accept_status = (apic_read(APIC_ESR) & 0xEF);
4433 }
4434
4435 /* Status is now clean */
4436
4437 send_status = 0;
4438 accept_status = 0;
4439
4440 /* Starting actual IPI sequence... */
4441
4442 SMP_PRINTK(("Asserting INIT.\n"));
4443
4444 /* Turn INIT on */
4445
4446 cfg=apic_read(APIC_ICR2);
4447 cfg&=0x00FFFFFF;
4448 /* Target chip */
4449 apic_write(APIC_ICR2, cfg|SET_APIC_DEST_FIELD(i));
4450 cfg=apic_read(APIC_ICR);
4451 /* Clear bits */
4452 cfg&=0xC0FFFF;
4453 cfg |= (APIC_DEST_LEVELTRIG | APIC_DEST_ASSERT |
4454         APIC_DEST_DM_INIT);
4455 /* Send IPI */
4456 apic_write(APIC_ICR, cfg);
4457
4458 udelay(200);
4459 SMP_PRINTK(("Deasserting INIT.\n"));
4460
4461 cfg=apic_read(APIC_ICR2);
4462 cfg&=0x00FFFFFF;
4463 /* Target chip */
4464 apic_write(APIC_ICR2, cfg|SET_APIC_DEST_FIELD(i));
4465 cfg=apic_read(APIC_ICR);
4466 /* Clear bits */
4467 cfg&=0xC0FFFF;
4468 cfg |= (APIC_DEST_LEVELTRIG | APIC_DEST_DM_INIT);
4469 /* Send IPI */
4470 apic_write(APIC_ICR, cfg);
4471
4472 /* Should we send STARTUP IPIs?
4473  *
4474  * Determine this based on the APIC version. If we
4475  * don't have an integrated APIC, don't send the
4476  * STARTUP IPIs. */
4477
4478 if ( apic_version[i] & 0xF0 )
4479     num_starts = 2;
4480 else
4481     num_starts = 0;
4482
4483 /* Run STARTUP IPI loop. */
4484
4485 for (j = 1; !(send_status || accept_status)
4486      && (j <= num_starts); j++)
4487 {
4488     SMP_PRINTK(("Sending STARTUP #%d.\n",j));
4489     apic_write(APIC_ESR, 0);
4490     SMP_PRINTK(("After apic_write.\n"));
4491
4492     /* STARTUP IPI */
4493
4494     cfg=apic_read(APIC_ICR2);
4495     cfg&=0x00FFFFFF;
4496     /* Target chip */
4497     apic_write(APIC_ICR2, cfg|SET_APIC_DEST_FIELD(i));
4498     cfg=apic_read(APIC_ICR);

```

```

4499  /* Clear bits          */
4500  cfg&--0xCDFFF;
4501  /* Boot on the stack   */
4502  cfg |= (APIC_DEST_DM_STARTUP | (start_eip >> 12));
4503  SMP_PRINTK(("Before start apic_write.\n"));
4504  /* Kick the second     */
4505  apic_write(APIC_ICR, cfg);
4506
4507  SMP_PRINTK(("Startup point 1.\n"));
4508
4509  timeout = 0;
4510  SMP_PRINTK(("Waiting for send to finish...\n"));
4511  do {
4512      SMP_PRINTK(("+"));
4513      udelay(100);
4514      send_status = apic_read(APIC_ICR) & 0x1000;
4515      } while (send_status && (timeout++ < 1000));
4516
4517  /* Give the other CPU some time to accept the IPI. */
4518  udelay(200);
4519  accept_status = (apic_read(APIC_ESR) & 0xEF);
4520  }
4521  SMP_PRINTK(("After Startup.\n"));
4522
4523  if (send_status) /* APIC never delivered?? */
4524      printk("APIC never delivered???\n");
4525  if (accept_status) /* Send accept error */
4526      printk("APIC delivery error (%lx).\n", accept_status);
4527
4528  if ( !(send_status || accept_status) )
4529  {
4530      /* allow APs to start initializing. */
4531      SMP_PRINTK(("Before Callout %d.\n", i));
4532      set_bit(i, (unsigned long *)&cpu_callout_map[0]);
4533      SMP_PRINTK(("After Callout %d.\n", i));
4534
4535      for(timeout=0;timeout<50000;timeout++)
4536      {
4537          if (cpu_callin_map[0]&(1<<i))
4538              break; /* It has booted */
4539          udelay(100); /* Wait 5s total for a response */
4540      }
4541      if (cpu_callin_map[0]&(1<<i))
4542      {
4543          /* # CPUs logically, starting from 1 (BSP is 0) */
4544          #if 0
4545              cpu_number_map[i] = cpucount;
4546              __cpu_logical_map[cpucount] = i;

```

```

4547 #endif
4548     printk("OK.\n");
4549     printk("CPU%d: ", i);
4550     print_cpu_info(&cpu_data[i]);
4551     }
4552     else
4553     {
4554         if (*(volatile unsigned char *)phys_to_virt(8192))
4555             = 0xA5)
4556             printk("Stuck ??\n");
4557         else
4558             printk("Not responding.\n");
4559     }
4560 SMP_PRINTK(("CPU has booted.\n"));
4561 }
4562 else
4563 {
4564     __cpu_logical_map[cpucount] = -1;
4565     cpu_number_map[i] = -1;
4566     cpucount--;
4567 }
4568
4569 swapper_pg_dir[0]=maincfg;
4570 local_flush_tlb();
4571
4572 /* mark "stuck" area as not stuck */
4573 *((volatile unsigned long *)phys_to_virt(8192)) = 0;
4574 }
4575
4576 cycles_t cacheflush_time;
4577 extern unsigned long cpu_hz;
4578
4579 static void smp_tune_scheduling (void)
4580 {
4581     unsigned long cachesize;
4582     /* Rough estimation for SMP scheduling, this is the
4583      * number of cycles it takes for a fully memory-limited
4584      * process to flush the SMP-local cache.
4585      *
4586      * (For a P5 this pretty much means we will choose
4587      * another idle CPU almost always at wakeup time (this
4588      * is due to the small L1 cache), on PII's it's around
4589      * 50-100 usecs, depending on the cache size) */
4590
4591     if (!cpu_hz) {
4592         /* this basically disables processor-affinity
4593          * scheduling on SMP without a TSC. */
4594         cacheflush_time = 0;

```

```

4595     return;
4596 } else {
4597     cachesize = boot_cpu_data.x86_cache_size;
4598     if (cachesize == -1)
4599         cachesize = 8; /* Pentiums */
4600
4601     cacheflush_time = cpu_hz/1024*cachesize/5000;
4602 }
4603
4604 printk("per-CPU timeslice cutoff: %ld.%02ld usecs.\n",
4605        (long)cacheflush_time/(cpu_hz/1000000),
4606        ((long)cacheflush_time*100/(cpu_hz/1000000)) % 100);
4607 }
4608
4609 unsigned int prof_multiplier[NR_CPUS];
4610 unsigned int prof_counter[NR_CPUS];
4611
4612 /* Cycle through the processors, sending APIC IPIs to
4613  * boot each. */
4614 void __init smp_boot_cpus(void)
4615 {
4616     int i;
4617
4618 #ifdef CONFIG_MTRR
4619     /* Must be done before other processors booted */
4620     mtrr_init_boot_cpu ();
4621 #endif
4622     /* Initialize the logical to physical CPU number
4623     * mapping and the per-CPU profiling counter/multiplier
4624     */
4625
4626     for (i = 0; i < NR_CPUS; i++) {
4627         cpu_number_map[i] = -1;
4628         prof_counter[i] = 1;
4629         prof_multiplier[i] = 1;
4630     }
4631
4632     /* Setup boot CPU information */
4633
4634     /* Final full version of the data */
4635     smp_store_cpu_info(boot_cpu_id);
4636     smp_tune_scheduling();
4637     printk("CPU%d: ", boot_cpu_id);
4638     print_cpu_info(&cpu_data[boot_cpu_id]);
4639
4640     /* not necessary because the MP table should list the
4641     * boot CPU too, but we do it for the sake of
4642     * robustness anyway. (and for the case when a non-SMP

```

```

4643     * board boots an SMP kernel) */
4644     cpu_present_map |= (1 << hard_smp_processor_id());
4645
4646     cpu_number_map[boot_cpu_id] = 0;
4647
4648     /* If we couldnt find an SMP configuration at boot
4649     * time, get out of here now! */
4650     if (!smp_found_config)
4651     {
4652         printk(KERN_NOTICE "SMP motherboard not detected. "
4653                "Using dummy APIC emulation.\n");
4654 #ifndef CONFIG_VISWS
4655         io_apic_irqs = 0;
4656 #endif
4657         cpu_online_map = cpu_present_map;
4658         goto smp_done;
4659     }
4660
4661     /* If SMP should be disabled. really disable it! */
4662
4663     if (!max_cpus)
4664     {
4665         smp_found_config = 0;
4666         printk(KERN_INFO "SMP mode deactivated, forcing use "
4667                "of dummy APIC emulation.\n");
4668     }
4669
4670 #ifdef SMP_DEBUG
4671     {
4672         int reg;
4673
4674         /* This is to verify that we're looking at a real
4675         * local APIC. Check these against your board if the
4676         * CPUs aren't getting started for no apparent
4677         * reason. */
4678         reg = apic_read(APIC_VERSION);
4679         SMP_PRINTK(("Getting VERSION: %x\n", reg));
4680
4681         apic_write(APIC_VERSION, 0);
4682         reg = apic_read(APIC_VERSION);
4683         SMP_PRINTK(("Getting VERSION: %x\n", reg));
4684
4685         /* The two version reads above should print the same
4686         * NON-ZERO!!! numbers. If the second one is zero,
4687         * there is a problem with the APIC write/read
4688         * definitions.
4689         *
4690         * The next two are just to see if we have sane

```

```

4691     * values. They're only really relevant if we're in
4692     * Virtual Wire compatibility mode, but most boxes
4693     * are anymore. */
4694     reg = apic_read(APIC_LVTO);
4695     SMP_PRINTK(("Getting LVTO: %x\n", reg));
4696
4697     reg = apic_read(APIC_LVT1);
4698     SMP_PRINTK(("Getting LVT1: %x\n", reg));
4699 }
4700 #endif
4701
4702     enable_local_APIC();
4703
4704     /* Set up our local APIC timer: */
4705     setup_APIC_clock ();
4706
4707     /* Now scan the CPU present map and fire up the other
4708     * CPUs. */
4709
4710     /* Add all detected CPUs. (later on we can down
4711     * individual CPUs which will change cpu_online_map but
4712     * not necessarily cpu_present_map. We are pretty much
4713     * ready for hot-swap CPUs.) */
4714     cpu_online_map = cpu_present_map;
4715     mb();
4716
4717     SMP_PRINTK(("CPU map: %lx\n", cpu_present_map));
4718
4719     for(i=0;i<NR_CPUS;i++)
4720     {
4721         /* Don't even attempt to start the boot CPU! */
4722         if (i == boot_cpu_id)
4723             continue;
4724
4725         if ((cpu_online_map & (1 << i))
4726             && (max_cpus < 0 || max_cpus > cpucount+1))
4727         {
4728             do_boot_cpu(i);
4729         }
4730
4731         /* Make sure we unmap all failed CPUs */
4732
4733         if (cpu_number_map[i] == -1 &&
4734             (cpu_online_map & (1 << i))) {
4735             printk("CPU #%d not responding. "
4736                 "Removing from cpu_online_map.\n", i);
4737             cpu_online_map &= ~(1 << i);
4738         }

```

```

4739     }
4740
4741     /* Cleanup possible dangling ends... */
4742     #ifndef CONFIG_VISWS
4743     {
4744         unsigned long cfg;
4745
4746         /* Install writable page 0 entry. */
4747         cfg = pg0[0];
4748         /* writeable, present, addr 0 */
4749         pg0[0] = _PAGE_RW | _PAGE_PRESENT;
4750         local_flush_tlb();
4751
4752         /* Paranoid: Set warm reset code and vector here back
4753         * to default values. */
4754         CMOS_WRITE(0, 0xf);
4755
4756         *((volatile long *) phys_to_virt(0x467)) = 0;
4757
4758         /* Restore old page 0 entry. */
4759         pg0[0] = cfg;
4760         local_flush_tlb();
4761     }
4762     #endif
4763
4764     /* Allow the user to impress friends. */
4765     SMP_PRINTK(("Before bogomips.\n"));
4766     if (cpucount-->0)
4767     {
4768         printk(KERN_ERR
4769             "Error: only one processor found.\n");
4770         cpu_online_map = (1<<hard_smp_processor_id());
4771     }
4772     else
4773     {
4774         unsigned long bogosum=0;
4775         for(i=0;i<32;i++)
4776         {
4777             if (cpu_online_map&(1<<i))
4778                 bogosum+=cpu_data[i].loops_per_sec;
4779         }
4780         printk(KERN_INFO "Total of %d processors activated "
4781             "(%lu.%02lu BogoMIPS).\n",
4782             cpucount+1,
4783             (bogosum+2500)/500000,
4784             ((bogosum+2500)/5000)%100);
4785         SMP_PRINTK(("Before bogocount - "
4786             "setting activated=1.\n"));

```

```

4787     smp_activated=1;
4788     smp_num_cpus=cpucount+1;
4789 }
4790 if (smp_b_stepping)
4791     printk(KERN_WARNING "WARNING: SMP operation may be "
4792            "unreliable with B stepping processors.\n");
4793 SMP_PRINTK(("Boot done.\n"));
4794
4795     cache_APIC_registers();
4796 #ifndef CONFIG_VISWS
4797     /* Here we can be sure that there is an IO-APIC in the
4798      * system. Let's go and set it up: */
4799     if (!skip_ioapic_setup)
4800         setup_IO_APIC();
4801 #endif
4802
4803 smp_done:
4804 }
4805
4806
4807 /* the following functions deal with sending IPIs between
4808 * CPUs.
4809 *
4810 * We use 'broadcast', CPU->CPU IPIs and self-IPIs too.*/
4811
4812
4813 /* Silly serialization to work around CPU bug in P5s. We
4814 * can safely turn it off on a 686. */
4815 #ifdef CONFIG_X86_GOOD_APIC
4816 # define FORCE_APIC_SERIALIZATION 0
4817 #else
4818 # define FORCE_APIC_SERIALIZATION 1
4819 #endif
4820
4821 static unsigned int cached_APIC_ICR;
4822 static unsigned int cached_APIC_ICR2;
4823
4824 /* Caches reserved bits, APIC reads are (mildly)
4825 * expensive and force otherwise unnecessary CPU
4826 * synchronization. (We could cache other APIC registers
4827 * too, but these are the main ones used in RL.) */
4828 #define slow_ICR (apic_read(APIC_ICR) & ~0xFDFDF)
4829 #define slow_ICR2 (apic_read(APIC_ICR2) & 0x00FFFFFF)
4830
4831 void cache_APIC_registers (void)
4832 {
4833     cached_APIC_ICR = slow_ICR;
4834     cached_APIC_ICR2 = slow_ICR2;

```

```

4835     mb();
4836 }
4837
4838 static inline unsigned int __get_ICR (void)
4839 {
4840 #if FORCE_APIC_SERIALIZATION
4841     /* Wait for the APIC to become ready - this should
4842      * never occur. It's a debugging check really. */
4843     int count = 0;
4844     unsigned int cfg;
4845
4846     while (count < 1000)
4847     {
4848         cfg = slow_ICR;
4849         if (!(cfg & (1 << 12))) {
4850             if (count)
4851                 atomic_add(count, (atomic_t*)&ipi_count);
4852             return cfg;
4853         }
4854         count++;
4855         udelay(10);
4856     }
4857     printk("CPU #%d: previous IPI still not cleared "
4858            "after 10ms\n", smp_processor_id());
4859     return cfg;
4860 #else
4861     return cached_APIC_ICR;
4862 #endif
4863 }
4864
4865 static inline unsigned int __get_ICR2 (void)
4866 {
4867 #if FORCE_APIC_SERIALIZATION
4868     return slow_ICR2;
4869 #else
4870     return cached_APIC_ICR2;
4871 #endif
4872 }
4873
4874 static inline int __prepare_ICR (unsigned int shortcut,
4875                                 int vector)
4876 {
4877     unsigned int cfg;
4878
4879     cfg = __get_ICR();
4880     cfg |= APIC_DEST_DM_FIXED|shortcut|vector;
4881
4882     return cfg;

```



```

4883 }
4884
4885 static inline int __prepare_ICR2 (unsigned int dest)
4886 {
4887     unsigned int cfg;
4888
4889     cfg = __get_ICR2();
4890     cfg |= SET_APIC_DEST_FIELD(dest);
4891
4892     return cfg;
4893 }
4894
4895 static inline void
4896 __send_IPI_shortcut(unsigned int shortcut, int vector)
4897 {
4898     unsigned int cfg;
4899     /* Subtle. In the case of the 'never do double writes'
4900      * workaround we have to lock out interrupts to be
4901      * safe. Otherwise it's just one single atomic write to
4902      * the APIC, no need for cli/sti. */
4903     #if FORCE_APIC_SERIALIZATION
4904     unsigned long flags;
4905
4906     __save_flags(flags);
4907     __cli();
4908     #endif
4909
4910     /* No need to touch the target chip field */
4911
4912     cfg = __prepare_ICR(shortcut, vector);
4913
4914     /* Send the IPI. The write to APIC_ICR
4915      * fires this off. */
4916     apic_write(APIC_ICR, cfg);
4917     #if FORCE_APIC_SERIALIZATION
4918     __restore_flags(flags);
4919     #endif
4920 }
4921
4922 static inline void send_IPI_allbutself(int vector)
4923 {
4924     __send_IPI_shortcut(APIC_DEST_ALLBUT, vector);
4925 }
4926
4927 static inline void send_IPI_all(int vector)
4928 {
4929     __send_IPI_shortcut(APIC_DEST_ALLINC, vector);
4930 }

```

```

4931
4932 void send_IPI_self(int vector)
4933 {
4934     __send_IPI_shortcut(APIC_DEST_SELF, vector);
4935 }
4936
4937 static inline void send_IPI_single(int dest, int vector)
4938 {
4939     unsigned long cfg;
4940     #if FORCE_APIC_SERIALIZATION
4941     unsigned long flags;
4942
4943     __save_flags(flags);
4944     __cli();
4945     #endif
4946
4947     /* prepare target chip field */
4948
4949     cfg = __prepare_ICR2(dest);
4950     apic_write(APIC_ICR2, cfg);
4951
4952     /* program the ICR*/
4953     cfg = __prepare_ICR(0, vector);
4954
4955     /* Send the IPI. The write to APIC_ICR fires this off.
4956      */
4957     apic_write(APIC_ICR, cfg);
4958     #if FORCE_APIC_SERIALIZATION
4959     __restore_flags(flags);
4960     #endif
4961 }
4962
4963 /* This is fraught with deadlocks. Probably the situation
4964  * is not that bad as in the early days of SMP, so we
4965  * might ease some of the paranoia here. */
4966
4967 void smp_flush_tlb(void)
4968 {
4969     int cpu = smp_processor_id();
4970     int stuck;
4971     unsigned long flags;
4972
4973     /* it's important that we do not generate any APIC
4974      * traffic until the AP CPUs have booted up! */
4975     if (cpu_online_map) {
4976         /* The assignment is safe because it's volatile so
4977          * the compiler cannot reorder it, because the i586
4978          * has strict memory ordering and because only the

```

```

4979  * kernel lock holder may issue a tlb flush. If you
4980  * break any one of those three change this to an
4981  * atomic bus locked or. */
4982
4983  smp_invalidate_needed = cpu_online_map;
4984
4985  /* Processors spinning on some lock with IRQs
4986  * disabled will see this IRQ late. The
4987  * smp_invalidate_needed map will ensure they don't
4988  * do a spurious flush tlb or miss one. */
4989
4990  __save_flags(flags);
4991  __cli();
4992
4993  send_IPI_allbutself(INVALIDATE_TLB_VECTOR);
4994
4995  /* Spin waiting for completion */
4996  stuck = 50000000;
4997  while (smp_invalidate_needed) {
4998      /* Take care of "crossing" invalidates */
4999      if (test_bit(cpu, &smp_invalidate_needed))
5000          clear_bit(cpu, &smp_invalidate_needed);
5001      --stuck;
5002      if (!stuck) {
5003          printk("stuck on TLB IPI wait (CPU#%d)\n",cpu);
5004          break;
5005      }
5006  }
5007  __restore_flags(flags);
5008 }
5009
5010 /* Flush the local TLB */
5011 local_flush_tlb();
5012 }
5013
5014
5015 /* this function sends a 'reschedule' IPI to another CPU.
5016 * it goes straight through and wastes no time
5017 * serializing anything. Worst case is that we lose a
5018 * reschedule ... */
5019 void smp_send_reschedule(int cpu)
5020 {
5021     send_IPI_single(cpu, RESCHEDULE_VECTOR);
5022 }
5023
5024 /* this function sends a 'stop' IPI to all other CPUs in
5025 * the system. it goes straight through. */
5026 void smp_send_stop(void)

```

```

5027 {
5028     send_IPI_allbutself(STOP_CPU_VECTOR);
5029 }
5030
5031 /* this function sends an 'reload MTRR state' IPI to all
5032 * other CPUs in the system. it goes straight through,
5033 * completion processing is done on the mtrr.c level. */
5034 void smp_send_mtrr(void)
5035 {
5036     send_IPI_allbutself(MTRR_CHANGE_VECTOR);
5037 }
5038
5039 /* Local timer interrupt handler. It does both profiling
5040 * and process statistics/rescheduling.
5041 *
5042 * We do profiling in every local tick,
5043 * statistics/rescheduling happen only every 'profiling
5044 * multiplier' ticks. The default multiplier is 1 and it
5045 * can be changed by writing the new multiplier value
5046 * into /proc/profile. */
5047 void smp_local_timer_interrupt(struct pt_regs * regs)
5048 {
5049     int cpu = smp_processor_id();
5050
5051     /* The profiling function is SMP safe. (nothing can
5052     * mess around with "current", and the profiling
5053     * counters are updated with atomic operations). This
5054     * is especially useful with a profiling
5055     * multiplier != 1 */
5056     if (!user_mode(regs))
5057         x86_do_profile(regs->eip);
5058
5059     if (--prof_counter[cpu]) {
5060         int user=0,system=0;
5061         struct task_struct * p = current;
5062
5063         /* After doing the above, we need to make like a
5064         * normal interrupt - otherwise timer interrupts
5065         * ignore the global interrupt lock, which is the
5066         * WrongThing (tm) to do. */
5067
5068         if (user_mode(regs))
5069             user=1;
5070         else
5071             system=1;
5072
5073         irq_enter(cpu, 0);
5074         if (p > pid) {

```

```

5075 update_one_process(p, 1, user, system, cpu);
5076
5077 p->counter -- 1;
5078 if (p->counter < 0) {
5079     p->counter = 0;
5080     p->need_resched = 1;
5081 }
5082 if (p->priority < DEF_PRIORITY) {
5083     kstat.cpu_nice += user;
5084     kstat.per_cpu_nice[cpu] += user;
5085 } else {
5086     kstat.cpu_user += user;
5087     kstat.per_cpu_user[cpu] += user;
5088 }
5089
5090 kstat.cpu_system += system;
5091 kstat.per_cpu_system[cpu] += system;
5092
5093 }
5094 prof_counter[cpu]=prof_multiplier[cpu];
5095 irq_exit(cpu, 0);
5096 }
5097
5098 /* We take the 'long' return path, and there every
5099  * subsystem grabs the appropriate locks (kernel lock/
5100  * irq lock).
5101  *
5102  * we might want to decouple profiling from the 'long
5103  * path', and do the profiling totally in assembly.
5104  *
5105  * Currently this isn't too much of an issue
5106  * (performance wise), we can take more than 100K local
5107  * irqs per second on a 100 MHz P5. */
5108 }
5109
5110 /* Local APIC timer interrupt. This is the most natural
5111  * way for doing local interrupts, but local timer
5112  * interrupts can be emulated by broadcast interrupts
5113  * too. [in case the hw doesnt support APIC timers]
5114  *
5115  * [ if a single-CPU system runs an SMP kernel then we
5116  * call the local interrupt as well. Thus we cannot
5117  * inline the local irq ... ] */
5118 void smp_apic_timer_interrupt(struct pt_regs * regs)
5119 {
5120     /* NOTE! We'd better ACK the irq immediately, because
5121      * timer handling can be slow, and we want to be able
5122      * to accept NMI tlb invalidates during this time. */

```

```

5123     ack_APIC_irq();
5124     smp_local_timer_interrupt(regs);
5125 }
5126
5127 /* Reschedule call back. Nothing to do, all the work is
5128  * done automatically when we return from the interrupt.
5129  */
5130 asmlinkage void smp_reschedule_interrupt(void)
5131 {
5132     ack_APIC_irq();
5133 }
5134
5135 /* Invalidate call-back */
5136 asmlinkage void smp_invalidate_interrupt(void)
5137 {
5138     if (test_and_clear_bit(smp_processor_id(),
5139                             &smp_invalidate_needed))
5140         local_flush_tlb();
5141 }
5142 ack_APIC_irq();
5143 }
5144
5145 static void stop_this_cpu (void)
5146 {
5147     /* Remove this CPU: */
5148     clear_bit(smp_processor_id(), &cpu_online_map);
5149
5150     if (cpu_data[smp_processor_id()].hlt_works_ok)
5151         for(;;) __asm__("hlt");
5152     for (;;);
5153 }
5154
5155 /* CPU halt call-back */
5156 asmlinkage void smp_stop_cpu_interrupt(void)
5157 {
5158     stop_this_cpu();
5159 }
5160
5161 void (*mtrr_hook) (void) = NULL;
5162
5163 asmlinkage void smp_mtrr_interrupt(void)
5164 {
5165     ack_APIC_irq();
5166     if (mtrr_hook) (*mtrr_hook)();
5167 }
5168
5169 /* This interrupt should _never_ happen with our APIC/SMP
5170  * architecture */

```

```

5171 asmlinkage void smp_spurious_interrupt(void)
5172 {
5173     ack_APIC_irq();
5174     /* see sw-dev-man vol 3, chapter 7.4.13.5 */
5175     printk("spurious APIC interrupt on CPU#%d, "
5176           "should never happen.\n", smp_processor_id());
5177 }
5178
5179 /* This part sets up the APIC 32 bit clock in LVTT1, with
5180 * HZ interrupts per second. We assume that the caller
5181 * has already set up the local APIC.
5182 *
5183 * The APIC timer is not exactly sync with the external
5184 * timer chip, it closely follows bus clocks. */
5185
5186 /* The timer chip is already set up at HZ interrupts per
5187 * second here, but we do not accept timer interrupts
5188 * yet. We only allow the BP to calibrate. */
5189 static unsigned int __init get_8254_timer_count(void)
5190 {
5191     unsigned int count;
5192
5193     outb_p(0x00, 0x43);
5194     count = inb_p(0x40);
5195     count |= inb_p(0x40) << 8;
5196
5197     return count;
5198 }
5199
5200 /* This function sets up the local APIC timer, with a
5201 * timeout of 'clocks' APIC bus clock. During calibration
5202 * we actually call this function twice, once with a
5203 * bogus timeout value, second time for real. The other
5204 * (noncalibrating) CPUs call this function only once,
5205 * with the real value.
5206 *
5207 * We are strictly in irqs off mode here, as we do not
5208 * want to get an APIC interrupt go off accidentally.
5209 *
5210 * We do reads before writes even if unnecessary, to get
5211 * around the APIC double write bug. */
5212 #define APIC_DIVISOR 16
5213
5214 void setup_APIC_timer(unsigned int clocks)
5215 {
5216     unsigned long lvttl_value;
5217     unsigned int tmp_value;
5218

```

```

5219     /* Unfortunately the local APIC timer cannot be set up
5220     * into NMI mode. With the IO APIC we can re-route the
5221     * external timer interrupt and broadcast it as an NMI
5222     * to all CPUs, so no pain. */
5223     tmp_value = apic_read(APIC_LVTT);
5224     lvttl_value = APIC_LVT_TIMER_PERIODIC |
5225                 LOCAL_TIMER_VECTOR;
5226     apic_write(APIC_LVTT, lvttl_value);
5227
5228     /* Divide PCLK by 16 */
5229     tmp_value = apic_read(APIC_TDCR);
5230     apic_write(APIC_TDCR, (tmp_value & ~APIC_TDR_DIV_1 )
5231               | APIC_TDR_DIV_16);
5232
5233     tmp_value = apic_read(APIC_TMICT);
5234     apic_write(APIC_TMICT, clocks/APIC_DIVISOR);
5235 }
5236
5237 void __init wait_8254_wraparound(void)
5238 {
5239     unsigned int curr_count, prev_count=0;
5240     int delta;
5241
5242     curr_count = get_8254_timer_count();
5243
5244     do {
5245         prev_count = curr_count;
5246         curr_count = get_8254_timer_count();
5247         delta = curr_count - prev_count;
5248
5249         /* This limit for delta seems arbitrary, but it
5250          * isn't, it's slightly above the level of error a
5251          * buggy Mercury/Neptune chipset timer can cause. */
5252     } while (delta < 300);
5253 }
5254
5255 /* In this function we calibrate APIC bus clocks to the
5256 * external timer. Unfortunately we cannot use jiffies
5257 * and the timer irq to calibrate, since some later
5258 * bootup code depends on getting the first irq? Ugh.
5259 *
5260 * We want to do the calibration only once since we want
5261 * to have local timer irqs synchron. CPUs connected by
5262 * the same APIC bus have the very same bus frequency.
5263 * And we want to have irqs off anyways, no accidental
5264 * APIC irq that way. */
5265
5266 int __init calibrate_APIC_clock(void)

```

```

5267 {
5268     unsigned long long t1,t2;
5269     long tt1,tt2;
5270     long calibration_result;
5271     int i;
5272
5273     printk("calibrating APIC timer ... ");
5274
5275     /* Put whatever arbitrary (but long enough) timeout
5276      * value into the APIC clock, we just want to get the
5277      * counter running for calibration. */
5278     setup_APIC_timer(100000000);
5279
5280     /* The timer chip counts down to zero. Let's wait for a
5281      * wraparound to start exact measurement: (the current
5282      * tick might have been already half done) */
5283
5284     wait_8254_wraparound ();
5285
5286     /* We wrapped around just now. Let's start: */
5287     READ_TSC(tt1);
5288     tt1=apic_read(APIC_TMCCT);
5289
5290     #define LOOPS (HZ/10)
5291     /* Let's wait LOOPS wraparounds: */
5292     for (i=0; i<LOOPS; i++)
5293         wait_8254_wraparound ();
5294
5295     tt2=apic_read(APIC_TMCCT);
5296     READ_TSC(tt2);
5297
5298     /* The APIC bus clock counter is 32 bits only, it might
5299      * have overflowed, but note that we use signed longs,
5300      * thus no extra care needed.
5301      *
5302      * underflowed to be exact, as the timer counts down ;)
5303      */
5304
5305     calibration_result = (tt1-tt2)*APIC_DIVISOR/LOOPS;
5306
5307     SMP_PRINTK(("n.... %ld CPU clocks in 1 timer chip "
5308              "tick.", (unsigned long)(t2-t1)/LOOPS));
5309
5310     SMP_PRINTK(("n.... %ld APIC bus clocks in 1 timer "
5311              "chip tick.", calibration_result));
5312
5313     printk("n.... CPU clock speed is %ld.%04ld MHz.n",
5314           ((long)(t2-t1)/LOOPS)/(1000000/HZ),

```

```

5315           ((long)(t2-t1)/LOOPS)%(1000000/HZ));
5316     .
5317     printk(".... system bus clock speed is %ld.%04ld "
5318           "MHz.n",
5319           calibration_result/(1000000/HZ),
5320           calibration_result%(1000000/HZ) );
5321     #undef LOOPS
5322
5323     return calibration_result;
5324 }
5325
5326 static unsigned int calibration_result;
5327
5328 void __init setup_APIC_clock(void)
5329 {
5330     unsigned long flags;
5331
5332     static volatile int calibration_lock;
5333
5334     __save_flags(flags);
5335     __cli();
5336
5337     SMP_PRINTK(("setup_APIC_clock() called.n"));
5338
5339     /* [ setup_APIC_clock() is called from all CPUs, but we
5340      * want to do this part of the setup only once ... and
5341      * it fits here best ] */
5342     if (!test_and_set_bit(0,&calibration_lock)) {
5343
5344         calibration_result=calibrate_APIC_clock();
5345         /* Signal completion to the other CPU[s]: */
5346         calibration_lock = 3;
5347
5348     } else {
5349         /* Other CPU is calibrating, wait for finish: */
5350         SMP_PRINTK(("waiting for other CPU "
5351                  "calibrating APIC ... "));
5352         while (calibration_lock == 1);
5353         SMP_PRINTK(("done, continuing.n"));
5354     }
5355
5356     /* Now set up the timer for real. */
5357     setup_APIC_timer (calibration_result);
5358
5359     /* We ACK the APIC, just in case there is something
5360      * pending. */
5361     ack_APIC_irq ();
5362

```

```

5363 __restore_flags(flags);
5364 }
5365
5366 /* the frequency of the profiling timer can be changed by
5367 * writing a multiplier value into /proc/profile.
5368 *
5369 * usually you want to run this on all CPUs ;) */
5370 int setup_profiling_timer(unsigned int multiplier)
5371 {
5372     int cpu = smp_processor_id();
5373     unsigned long flags;
5374
5375     /* Sanity check. [at least 500 APIC cycles should be
5376     * between APIC interrupts as a rule of thumb, to avoid
5377     * irq flooding us] */
5378     if ( (!multiplier) ||
5379         (calibration_result/multiplier < 500))
5380         return -EINVAL;
5381
5382     save_flags(flags);
5383     cli();
5384     setup_APIC_timer(calibration_result/multiplier);
5385     prof_multiplier[cpu]=multiplier;
5386     restore_flags(flags);
5387
5388     return 0;
5389 }
5390
5391 #undef APIC_DIVISOR
5392

```

arch/i386/kernel/time.c

```

5393 /*
5394 * linux/arch/i386/kernel/time.c
5395 *
5396 * Copyright (C) 1991, 1992, 1995 Linus Torvalds
5397 *
5398 * This file contains the PC-specific time handling
5399 * details: reading the RTC at bootup, etc..
5400 * 1994-07-02 Alan Modra
5401 *     fixed set_rtc_mmss, fixed time.year for >= 2000,
5402 *     new mktime
5403 * 1995-03-26 Markus Kuhn
5404 *     fixed 500 ms bug at call to set_rtc_mmss, fixed
5405 *     DS12887 precision CMOS clock update
5406 * 1996-05-03 Ingo Molnar
5407 *     fixed time warps in

```

```

5408 *     do_[slow|fast]_gettimeoffset()
5409 * 1997-09-10 Updated NTP code according to technical
5410 *     memorandum Jan '96 "A Kernel Model for Precision
5411 *     Timekeeping" by Dave Mills
5412 * 1998-09-05 (Various) More robust
5413 *     do_fast_gettimeoffset() algorithm implemented
5414 *     (works with APM, Cyrix 6x86MX and Centaur C6),
5415 *     monotonic gettimeofday() with
5416 *     fast_get_timeoffset(), drift-proof precision TSC
5417 *     calibration on boot (C. Scott Ananian
5418 *     <cananian@alumni.princeton.edu>, Andrew D. Balsa
5419 *     <andreabalsa@altern.org>, Philip Gladstone
5420 *     <philip@raptor.com>; ported from 2.0.35 Jumbo-9
5421 *     by Michael Krause <m.krause@tu-harburg.de>).
5422 * 1998-12-16 Andrea Arcangeli
5423 *     Fixed Jumbo-9 code in 2.1.131: do_gettimeofday
5424 *     was missing 1 jiffy because was not accounting
5425 *     lost_ticks.
5426 * 1998-12-24 Copyright (C) 1998 Andrea Arcangeli
5427 *     Fixed a xtime SMP race (we need the xtime_lock rw
5428 *     spinlock to serialize accesses to
5429 *     xtime/lost_ticks).
5430 */
5431
5432 #include <linux/errno.h>
5433 #include <linux/sched.h>
5434 #include <linux/kernel.h>
5435 #include <linux/param.h>
5436 #include <linux/string.h>
5437 #include <linux/mm.h>
5438 #include <linux/interrupt.h>
5439 #include <linux/time.h>
5440 #include <linux/delay.h>
5441 #include <linux/init.h>
5442 #include <linux/smp.h>
5443
5444 #include <asm/processor.h>
5445 #include <asm/uaccess.h>
5446 #include <asm/io.h>
5447 #include <asm/irq.h>
5448 #include <asm/delay.h>
5449
5450 #include <linux/mc146818rtc.h>
5451 #include <linux/timex.h>
5452 #include <linux/config.h>
5453
5454 #include <asm/fixmap.h>
5455 #include <asm/cobalt.h>

```

```

5456
5457 /* for x86_do_profile() */
5458 #include "irq.h"
5459
5460
5461 /* Detected as we calibrate the TSC */
5462 unsigned long cpu_hz;
5463
5464 /* Number of usecs that the last interrupt was delayed */
5465 static int delay_at_last_interrupt;
5466
5467 /* lsb 32 bits of Time Stamp Counter */
5468 static unsigned long last_tsc_low;
5469
5470 /* Cached *multiplier* to convert TSC counts to
5471 * microseconds. (see the equation below). Equal to
5472 * 2^32 * (1 / (clocks per usec) ). Initialized in
5473 * time_init. */
5474 static unsigned long fast_gettimeoffset_quotient=0;
5475
5476 extern rwlock_t xtime_lock;
5477
5478 static inline unsigned long do_fast_gettimeoffset(void)
5479 {
5480     register unsigned long eax asm("ax");
5481     register unsigned long edx asm("dx");
5482
5483     /* Read the Time Stamp Counter */
5484     __asm__("rdtsc"
5485           : "=a" (eax), "=d" (edx));
5486
5487     /* .. relative to previous jiffy (32 bits is enough) */
5488     eax -= last_tsc_low; /* tsc_low delta */
5489
5490     /* Time offset
5491     * = (tsc_low delta) * fast_gettimeoffset_quotient
5492     * = (tsc_low delta) * (usecs_per_clock)
5493     * = (tsc_low delta) * (usecs_per_jiffy /
5494     *                       clocks_per_jiffy)
5495     * Using a mull instead of a divl saves up to 31 clock
5496     * cycles in the critical path. */
5497
5498     __asm__("mull %2"
5499           : "=a" (eax), "=d" (edx)
5500           : "g" (fast_gettimeoffset_quotient),
5501           "0" (eax));
5502
5503     /* our adjusted time offset in microseconds */

```

```

5504     return delay_at_last_interrupt + edx;
5505 }
5506
5507 #define TICK_SIZE tick
5508
5509 #ifndef CONFIG_X86_TSC
5510
5511 /* This function must be called with interrupts disabled
5512 * It was inspired by Steve McCanne's microtime-i386 for
5513 * BSD. -- jrs
5514 *
5515 * However, the pc-audio speaker driver changes the
5516 * divisor so that it gets interrupted rather more often
5517 * - it loads 64 into the counter rather than 11932! This
5518 * has an adverse impact on do_gettimeoffset() -- it
5519 * stops working! What is also not good is that the
5520 * interval that our timer function gets called is no
5521 * longer 10.0002 ms, but 9.9767 ms. To get around this
5522 * would require using a different timing source. Maybe
5523 * someone could use the RTC - I know that this can
5524 * interrupt at frequencies ranging from 8192Hz to
5525 * 2Hz. If I had the energy, I'd somehow fix it so that
5526 * at startup, the timer code in sched.c would select
5527 * using either the RTC or the 8253 timer. The decision
5528 * would be based on whether there was any other device
5529 * around that needed to trample on the 8253. I'd set up
5530 * the RTC to interrupt at 1024 Hz, and then do some
5531 * jiggery to have a version of do_timer that advanced
5532 * the clock by 1/1024 s. Every time that reached over
5533 * 1/100 of a second, then do all the old code. If the
5534 * time was kept correct then do_gettimeoffset could just
5535 * return 0 - there is no low order divider that can be
5536 * accessed.
5537 *
5538 * Ideally, you would be able to use the RTC for the
5539 * speaker driver, but it appears that the speaker driver
5540 * really needs interrupt more often than every 120 us or
5541 * so.
5542 *
5543 * Anyway, this needs more thought....  pjsg (1993-08 28)
5544 *
5545 * If you are really that interested, you should be
5546 * reading comp.protocols.time.ntp! */
5547 static unsigned long do_slow_gettimeoffset(void)
5548 {
5549     int count;
5550
5551     /* for the first call after boot */

```

```

5552 static int count_p = LATCH;
5553 static unsigned long jiffies_p = 0;
5554
5555 /* cache volatile jiffies temporarily; we have IRQs
5556  * turned off. */
5557 unsigned long jiffies_t;
5558
5559 /* timer count may underflow right here */
5560 outb_p(0x00, 0x43); /* latch the count ASAP */
5561
5562 count = inb_p(0x40); /* read the latched count */
5563
5564 /* We do this guaranteed double memory access instead
5565  * of a _p postfix in the previous port access. Wheee,
5566  * hackady hack */
5567 jiffies_t = jiffies;
5568
5569 count |= inb_p(0x40) << 8;
5570
5571 /* avoiding timer inconsistencies (they are rare, but
5572  * they happen)... there are two kinds of problems
5573  * that must be avoided here: 1. the timer counter
5574  * underflows 2. hardware problem with the timer, not
5575  * giving us continuous time, the counter does small
5576  * "jumps" upwards on some Pentium systems, (see c't
5577  * 95/10 page 335 for Neptune bug.) */
5578
5579 /* you can safely undefine this if you don't have the
5580  * Neptune chipset */
5581
5582 #define BUGGY_NEPTUN_TIMER
5583
5584 if( jiffies_t == jiffies_p ) {
5585     if( count > count_p ) {
5586         /* the nutcase */
5587
5588         outb_p(0x0A, 0x20);
5589
5590         /* assumption about timer being IRQ1 */
5591         if( inb(0x20) & 0x01 ) {
5592             /* We cannot detect lost timer interrupts ...
5593              * well, that's why we call them lost, don't we?
5594              * :) [hmm, on the Pentium and Alpha we can
5595              * ... sort of] */
5596             count -= LATCH;
5597         } else {
5598 #ifndef BUGGY_NEPTUN_TIMER
5599             /* for the Neptune bug we know that the 'latch'

```

```

5600         * command doesnt latch the high and low value of
5601         * the counter atomically. Thus we have to
5602         * subtract 256 from the counter ... funny,
5603         * isn't it? :) */
5604
5605         count -= 256;
5606 #else
5607         printk("do_slow_gettimeoffset(): "
5608             "hardware timer problem?\n");
5609 #endif
5610     }
5611 } else
5612     jiffies_p = jiffies_t;
5613
5614     count_p = count;
5615
5616     count = ((LATCH-1) - count) * TICK_SIZE;
5617     count = (count + LATCH/2) / LATCH;
5618
5619     return count;
5620 }
5621
5622 static unsigned long (*do_gettimeoffset)(void) =
5623     do_slow_gettimeoffset;
5624
5625 #else
5626 #define do_gettimeoffset() do_fast_gettimeoffset()
5627 #endif
5628
5629 /* This version of gettimeofday has microsecond
5630  * resolution and better than microsecond precision on
5631  * fast x86 machines with TSC. */
5632 void do_gettimeofday(struct timeval *tv)
5633 {
5634     extern volatile unsigned long lost_ticks;
5635     unsigned long flags;
5636     unsigned long usec, sec;
5637
5638     read_lock_irqsave(&time_lock, flags);
5639     usec = do_gettimeoffset();
5640     {
5641         unsigned long lost = lost_ticks;
5642         if (lost)
5643             usec += lost * (1000000 / HZ);
5644     }
5645 }

```



```

5648 sec = xtime.tv_sec;
5649 usec += xtime.tv_usec;
5650 read_unlock_irqrestore(&xtime_lock, flags);
5651
5652 while (usec >= 1000000) {
5653     usec -= 1000000;
5654     sec++;
5655 }
5656
5657 tv->tv_sec = sec;
5658 tv->tv_usec = usec;
5659 }
5660
5661 void do_settimeofday(struct timeval *tv)
5662 {
5663     write_lock_irq(&xtime_lock);
5664     /* This is revolting. We need to set the xtime.tv_usec
5665      * correctly. However, the value in this location is
5666      * is value at the last tick.
5667      * Discover what correction gettimeofday
5668      * would have done, and then undo it!
5669      */
5670     tv->tv_usec -= do_gettimeoffset();
5671
5672     while (tv->tv_usec < 0) {
5673         tv->tv_usec += 1000000;
5674         tv->tv_sec--;
5675     }
5676
5677     xtime = *tv;
5678     time_adjust = 0;          /* stop active adjtime() */
5679     time_status |= STA_UNSYNC;
5680     time_maxerror = NTP_PHASE_LIMIT;
5681     time_esterror = NTP_PHASE_LIMIT;
5682     write_unlock_irq(&xtime_lock);
5683 }
5684
5685 /* In order to set the CMOS clock precisely, set_rtc_mmss
5686  * has to be called 500 ms after the second nowtime has
5687  * started, because when nowtime is written into the
5688  * registers of the CMOS clock, it will jump to the next
5689  * second precisely 500 ms later. Check the Motorola
5690  * MC146818A or Dallas DS12887 data sheet for details.
5691  *
5692  * BUG: This routine does not handle hour overflow
5693  * properly; it just sets the minutes. Usually you'll
5694  * only notice that after reboot! */
5695 static int set_rtc_mmss(unsigned long nowtime)

```

```

5696 {
5697     int retval = 0;
5698     int real_seconds, real_minutes, cmos_minutes;
5699     unsigned char save_control, save_freq_select;
5700
5701     /* tell the clock it's being set */
5702     save_control = CMOS_READ(RTC_CONTROL);
5703     CMOS_WRITE((save_control|RTC_SET), RTC_CONTROL);
5704
5705     /* stop and reset prescaler */
5706     save_freq_select = CMOS_READ(RTC_FREQ_SELECT);
5707     CMOS_WRITE((save_freq_select|RTC_DIV_RESET2),
5708               RTC_FREQ_SELECT);
5709
5710     cmos_minutes = CMOS_READ(RTC_MINUTES);
5711     if (!(save_control & RTC_DM_BINARY) || RTC_ALWAYS_BCD)
5712         BCD_TO_BIN(cmos_minutes);
5713
5714     /* since we're only adjusting minutes and seconds,
5715      * don't interfere with hour overflow. This avoids
5716      * messing with unknown time zones but requires your
5717      * RTC not to be off by more than 15 minutes */
5718     real_seconds = nowtime % 60;
5719     real_minutes = nowtime / 60;
5720     if (((abs(real_minutes - cmos_minutes) + 15)/30) & 1)
5721         real_minutes += 30; /* correct for 1/2-hour tzone */
5722     real_minutes %= 60;
5723
5724     if (abs(real_minutes - cmos_minutes) < 30) {
5725         if (!(save_control & RTC_DM_BINARY) ||
5726             RTC_ALWAYS_BCD) {
5727             BIN_TO_BCD(real_seconds);
5728             BIN_TO_BCD(real_minutes);
5729         }
5730         CMOS_WRITE(real_seconds, RTC_SECONDS);
5731         CMOS_WRITE(real_minutes, RTC_MINUTES);
5732     } else {
5733         printk(KERN_WARNING
5734               "set_rtc_mmss: can't update from %d to %d\n",
5735               cmos_minutes, real_minutes);
5736         retval = -1;
5737     }
5738
5739     /* The following flags have to be released exactly in
5740      * this order, otherwise the DS12887 (popular MC146818A
5741      * clone with integrated battery and quartz) will not
5742      * reset the oscillator and will not update precisely
5743      * 500 ms later. You won't find this mentioned in the

```

```

5744 * Dallas Semiconductor data sheets, but who believes
5745 * data sheets anyway ... -- Markus Kuhn */
5746 CMOS_WRITE(save_control, RTC_CONTROL);
5747 CMOS_WRITE(save_freq_select, RTC_FREQ_SELECT);
5748
5749 return retval;
5750 }
5751
5752 /* last time the cmos clock got updated */
5753 static long last_rtc_update = 0;
5754
5755 /* timer_interrupt() needs to keep up the real-time
5756 * clock, as well as call the "do_timer()" routine every
5757 * clocktick */
5758 static inline void do_timer_interrupt(int irq,
5759 void *dev_id, struct pt_regs *regs)
5760 {
5761 #ifdef CONFIG_VISWS
5762 /* Clear the interrupt */
5763 co_cpu_write(CO_CPU_STAT,
5764 co_cpu_read(CO_CPU_STAT) & ~CO_STAT_TIMEINTR);
5765 #endif
5766 do_timer(regs);
5767 /* In the SMP case we use the local APIC timer interrupt
5768 * to do the profiling, except when we simulate SMP mode
5769 * on a uniprocessor system, in that case we have to call
5770 * the local interrupt handler. */
5771 #ifndef __SMP__
5772 if (!user_mode(regs))
5773 x86_do_profile(regs->eip);
5774 #else
5775 if (!smp_found_config)
5776 smp_local_timer_interrupt(regs);
5777 #endif
5778
5779 /* If we have an externally synchronized Linux clock,
5780 * then update CMOS clock accordingly every ~11
5781 * minutes. Set_rtc_mmss() has to be called as close as
5782 * possible to 500 ms before the new second starts. */
5783 if ((time_status & STA_UNSYNC) == 0 &&
5784 xtime.tv_sec > last_rtc_update + 660 &&
5785 xtime.tv_usec >= 500000 - ((unsigned) tick) / 2 &&
5786 xtime.tv_usec <= 500000 + ((unsigned) tick) / 2) {
5787 if (set_rtc_mmss(xtime.tv_sec) == 0)
5788 last_rtc_update = xtime.tv_sec;
5789 else /* do it again in 60 s */
5790 last_rtc_update = xtime.tv_sec - 600;
5791 }

```

```

5792
5793 #ifdef CONFIG_MCA
5794 if( MCA_bus ) {
5795 /* The PS/2 uses level-triggered interrupts. You *
5796 * can't turn them off, nor would you want to (any
5797 * attempt to enable edge-triggered interrupts
5798 * usually gets intercepted by a special hardware
5799 * circuit). Hence we have to acknowledge the timer
5800 * interrupt. Through some incredibly stupid design
5801 * idea, the reset for IRQ 0 is done by setting the
5802 * high bit of the PPI port B (0x61). Note that some
5803 * PS/2s, notably the 55SX, work fine if this is
5804 * removed. */
5805 irq = inb_p( 0x61 ); /* read the current state */
5806 outb_p( irq|0x80, 0x61 ); /* reset the IRQ */
5807 }
5808 #endif
5809 }
5810
5811 static int use_tsc = 0;
5812
5813 /* This is the same as the above, except we _also_ save
5814 * the current Time Stamp Counter value at the time of
5815 * the timer interrupt, so that we later on can estimate
5816 * the time of day more exactly. */
5817 static void timer_interrupt(int irq, void *dev_id,
5818 struct pt_regs *regs)
5819 {
5820 int count;
5821
5822 /* Here we are in the timer irq handler. We just have
5823 * irqs locally disabled but we don't know if the
5824 * timer_bh is running on the other CPU. We need to
5825 * avoid to SMP race with it. NOTE: we don't need the
5826 * irq version of write_lock because as just said we
5827 * have irq locally disabled. -arca */
5828 write_lock(&xtime_lock);
5829
5830 if (use_tsc)
5831 {
5832 /* It is important that these two operations happen
5833 * almost at the same time. We do the RDTSC stuff
5834 * first, since it's faster. To avoid any
5835 * inconsistencies, we need interrupts disabled
5836 * locally. */
5837
5838 /* Interrupts are just disabled locally since the
5839 * timer irq has the SA_INTERRUPT flag set. -arca */

```

```

5840
5841 /* read Pentium cycle counter */
5842 __asm__("rdtsc" : "=a" (last_tsc_low) : : "edx");
5843
5844 outb_p(0x00, 0x43); /* latch the count ASAP */
5845
5846 count = inb_p(0x40); /* read the latched count */
5847 count |= inb(0x40) << 8;
5848
5849 count = ((LATCH-1) - count) * TICK_SIZE;
5850 delay_at_last_interrupt = (count + LATCH/2) / LATCH;
5851 )
5852
5853 do_timer_interrupt(irq, NULL, regs);
5854
5855 write_unlock(&xtime_lock);
5856
5857 }
5858
5859 /* Converts Gregorian date to seconds since 1970-01-01
5860 * 00:00:00. Assumes input in normal date format,
5861 * i.e. 1980-12-31 23:59:59 => year=1980, mon=12, day=31,
5862 * hour=23, min=59, sec=59.
5863 *
5864 * [For the Julian calendar (which was used in Russia
5865 * before 1917, Britain & colonies before 1752, anywhere
5866 * else before 1582, and is still in use by some
5867 * communities) leave out the -year/100+year/400 terms,
5868 * and add 10.]
5869 *
5870 * This algorithm was first published by Gauss (I think).
5871 *
5872 * WARNING: this function will overflow on 2106-02-07
5873 * 06:28:16 on machines were long is 32-bit! (However, as
5874 * time_t is signed, we will already get problems at
5875 * other places on 2038-01-19 03:14:08) */
5876 static inline unsigned long mktime(
5877 unsigned int year, unsigned int mon,
5878 unsigned int day, unsigned int hour,
5879 unsigned int min, unsigned int sec)
5880 {
5881 if (0 >= (int) (mon -- 2)) { /* 1..12 -> 11,12,1..10 */
5882 mon += 12; /* Puts Feb last since it has leap day */
5883 year -= 1;
5884 }
5885 return (((
5886 (unsigned long)(year/4 - year/100 + year/400 +
5887 367*mon/12 + day) + year*365 - 719499

```

```

5888 )*24 + hour /* now have hours */
5889 )*60 + min /* now have minutes */
5890 )*60 + sec; /* finally seconds */
5891 }
5892
5893 /* not static: needed by APM */
5894 unsigned long get_cmos_time(void)
5895 {
5896 unsigned int year, mon, day, hour, min, sec;
5897 int i;
5898
5899 /* The Linux interpretation of the CMOS clock register
5900 * contents: When the Update-In-Progress (UIP) flag
5901 * goes from 1 to 0, the RTC registers show the second
5902 * which has precisely just started. Let's hope other
5903 * operating systems interpret the RTC the same way. */
5904 /* read RTC exactly on falling edge of update flag */
5905 /* may take up to 1 second... */
5906 for (i = 0 ; i < 1000000 ; i++)
5907 if (CMOS_READ(RTC_FREQ_SELECT) & RTC_UIP)
5908 break;
5909 /* must try at least 2.228 ms */
5910 for (i = 0 ; i < 1000000 ; i++)
5911 if (!(CMOS_READ(RTC_FREQ_SELECT) & RTC_UIP))
5912 break;
5913 /* Isn't this overkill?
5914 * UIP above should guarantee consistency */
5915 do {
5916 sec = CMOS_READ(RTC_SECONDS);
5917 min = CMOS_READ(RTC_MINUTES);
5918 hour = CMOS_READ(RTC_HOURS);
5919 day = CMOS_READ(RTC_DAY_OF_MONTH);
5920 mon = CMOS_READ(RTC_MONTH);
5921 year = CMOS_READ(RTC_YEAR);
5922 } while (sec != CMOS_READ(RTC_SECONDS));
5923 if (!(CMOS_READ(RTC_CONTROL) & RTC_DM_BINARY) ||
5924 RTC_ALWAYS_BCD) {
5925 BCD_TO_BIN(sec);
5926 BCD_TO_BIN(min);
5927 BCD_TO_BIN(hour);
5928 BCD_TO_BIN(day);
5929 BCD_TO_BIN(mon);
5930 BCD_TO_BIN(year);
5931 }
5932 if ((year += 1900) < 1970)
5933 year += 100;
5934 return mktime(year, mon, day, hour, min, sec);
5935 }

```

```

5936
5937 static struct irqaction irq0 =
5938 { timer_interrupt, SA_INTERRUPT, 0, "timer", NULL, NULL};
5939
5940 /* ----- Calibrate the TSC -----
5941 * Return 2^32 * (1 / (TSC clocks per usec)) for
5942 * do_fast_gettimeoffset(). Too much 64-bit arithmetic
5943 * here to do this cleanly in C, and for accuracy's sake
5944 * we want to keep the overhead on the CTC speaker
5945 * (channel 2) output busy loop as low as possible. We
5946 * avoid reading the CTC registers directly because of
5947 * the awkward 8 bit access mechanism of the 82C54
5948 * device. */
5949
5950 #define CALIBRATE_LATCH (5 * LATCH)
5951 #define CALIBRATE_TIME (5 * 1000020/HZ)
5952
5953 __initfunc(static unsigned long calibrate_tsc(void))
5954 {
5955     /* Set the Gate high, disable speaker */
5956     outb((inb(0x61) & ~0x02) | 0x01, 0x61);
5957
5958     /* Now let's take care of CTC channel 2
5959     *
5960     * Set the Gate high, program CTC channel 2 for mode 0,
5961     * (interrupt on terminal count mode), binary count,
5962     * load 5 * LATCH count, (LSB and MSB) to begin
5963     * countdown. */
5964     outb(0xb0, 0x43); /* binary, mode 0, LSB/MSB, Ch 2 */
5965     outb(CALIBRATE_LATCH & 0xff, 0x42); /* LSB of count */
5966     outb(CALIBRATE_LATCH >> 8, 0x42); /* MSB of count */
5967
5968     {
5969         unsigned long startlow, starthigh;
5970         unsigned long endlow, endhigh;
5971         unsigned long count;
5972
5973         __asm__ __volatile__ ("rdtsc":"=a" (startlow),"=d"
5974                               (starthigh));
5975         count = 0;
5976         do {
5977             count++;
5978         } while ((inb(0x61) & 0x20) == 0);
5979         __asm__ __volatile__ ("rdtsc":"=a" (endlow),"=d"
5980                               (endhigh));
5981
5982         last_tsc_low = endlow;
5983

```

```

5984     /* Error: ECTCNEVERSET */
5985     if (count <= 1)
5986         goto bad_ctc;
5987
5988     /* 64-bit subtract - gcc just messes up with long
5989     * longs */
5990     __asm__ ("subl %2,%0\n\t"
5991             "sbb1 %3,%1"
5992             : "=a" (endlow), "=d" (endhigh)
5993             : "g" (startlow), "g" (starthigh),
5994             "0" (endlow), "1" (endhigh));
5995
5996     /* Error: ECPUTOOFAST */
5997     if (endhigh)
5998         goto bad_ctc;
5999
6000     /* Error: ECPUTOOSLOW */
6001     if (endlow <= CALIBRATE_TIME)
6002         goto bad_ctc;
6003
6004     __asm__ ("divl %2"
6005             : "=a" (endlow), "=d" (endhigh)
6006             : "r" (endlow), "0" (0), "1" (CALIBRATE_TIME));
6007
6008     return endlow;
6009 }
6010
6011 /* The CTC wasn't reliable: we got a hit on the very
6012 * first read, or the CPU was so fast/slow that the
6013 * quotient wouldn't fit in 32 bits.. */
6014 bad_ctc:
6015     return 0;
6016 }
6017
6018 __initfunc(void time_init(void))
6019 {
6020     xtime.tv_sec = get_cmos_time();
6021     xtime.tv_usec = 0;
6022
6023     /* If we have APM enabled or the CPU clock speed is
6024     * variable (CPU stops clock on HLT or slows clock to
6025     * save power) then the TSC timestamps may diverge by up
6026     * to 1 jiffy from 'real time' but nothing will break.
6027     * The most frequent case is that the CPU is "woken" from
6028     * a halt state by the timer interrupt itself, so we get
6029     * 0 error. In the rare cases where a driver would "wake"
6030     * the CPU and request a timestamp, the maximum error is
6031     * < 1 jiffy. But timestamps are still perfectly ordered.

```

```

6032 * Note that the TSC counter will be reset if APM
6033 * suspends to disk; this won't break the kernel, though.
6034 * 'cuz we're smart. See arch/i386/kernel/apm.c. */
6035 /* Firstly we have to do a CPU check for chips with a
6036 * potentially buggy TSC. At this point we haven't run
6037 * the ident/bugs checks so we must run this hook as it
6038 * may turn off the TSC flag.
6039 *
6040 * NOTE: this doesn't yet handle SMP 486 machines where
6041 * only some CPU's have a TSC. Thats never worked and
6042 * nobody has moaned if you have the only one in the
6043 * world - you fix it! */
6044
6045 dodgy_tsc();
6046
6047 if (boot_cpu_data.x86_capability & X86_FEATURE_TSC) {
6048     unsigned long tsc_quotient = calibrate_tsc();
6049     if (tsc_quotient) {
6050         fast_gettimeoffset_quotient = tsc_quotient;
6051         use_tsc = 1;
6052 #ifndef do_gettimeoffset
6053         do_gettimeoffset = do_fast_gettimeoffset;
6054 #endif
6055         do_get_fast_time = do_gettimeofday;
6056
6057         /* report CPU clock rate in Hz. The formula is
6058          * (10^6 * 2^32) / (2^32 * 1 / (clocks/us)) =
6059          * clock/second. Our precision is about 100 ppm. */
6060         {
6061             unsigned long eax=0, edx=1000000;
6062             __asm__("divl %2"
6063                 : "=a" (cpu_hz), "=d" (edx)
6064                 : "r" (tsc_quotient),
6065                   "0" (eax), "1" (edx));
6066             printk("Detected %ld Hz processor.\n", cpu_hz);
6067         }
6068     }
6069
6070 #ifdef CONFIG_VISWS
6071     printk("Starting Cobalt Timer system clock\n");
6072
6073     /* Set the countdown value */
6074     co_cpu_write(CO_CPU_TIMEVAL, CO_TIME_HZ/HZ);
6075
6076     /* Start the timer */
6077     co_cpu_write(CO_CPU_CTRL,
6078                 co_cpu_read(CO_CPU_CTRL)|CO_CTRL_TIMERUN);
6079

```

```

6080 /* Enable (unmask) the timer interrupt */
6081 co_cpu_write(CO_CPU_CTRL,
6082             co_cpu_read(CO_CPU_CTRL) & ~CO_CTRL_TIMEMASK);
6083
6084 /* Wire cpu IDT entry to s/w handler (and Cobalt APIC
6085 * to IDT) */
6086 setup_x86_irq(CO_IRQ_TIMER, &irq0);
6087 #else
6088 setup_x86_irq(0, &irq0);
6089 #endif
6090 }

```

arch/i386/kernel/traps.c

```

6091 /*
6092 * linux/arch/i386/traps.c
6093 *
6094 * Copyright (C) 1991, 1992 Linus Torvalds
6095 */
6096
6097 /* 'Traps.c' handles hardware traps and faults after we
6098 * have saved some state in 'asm.s'. */
6099 #include <linux/config.h>
6100 #include <linux/sched.h>
6101 #include <linux/kernel.h>
6102 #include <linux/string.h>
6103 #include <linux/errno.h>
6104 #include <linux/ptrace.h>
6105 #include <linux/timer.h>
6106 #include <linux/mm.h>
6107 #include <linux/smp.h>
6108 #include <linux/smp_lock.h>
6109 #include <linux/init.h>
6110 #include <linux/delay.h>
6111
6112 #ifdef CONFIG_MCA
6113 #include <linux/mca.h>
6114 #include <asm/processor.h>
6115 #endif
6116
6117 #include <asm/system.h>
6118 #include <asm/uaccess.h>
6119 #include <asm/io.h>
6120 #include <asm/spinlock.h>
6121 #include <asm/atomic.h>
6122 #include <asm/debugreg.h>
6123 #include <asm/desc.h>
6124

```

```

6125 #include <asm/smp.h>
6126
6127 #ifdef CONFIG_X86_VISWS_APIC
6128 #include <asm/fixmap.h>
6129 #include <asm/cobalt.h>
6130 #include <asm/lithium.h>
6131 #endif
6132
6133 #include "irq.h"
6134
6135 asmlinkage int system_call(void);
6136 asmlinkage void lcall7(void);
6137
6138 struct desc_struct default_ldt = { 0, 0 };
6139
6140 /* The IDT has to be page-aligned to simplify the Pentium
6141 * F0 0F bug workaround.. We have a special link segment
6142 * for this. */
6143 struct desc_struct idt_table[256]
6144 __attribute__((__section__(".data.idt"))) = { {0, 0}, };
6145
6146 static inline void console_verbose(void)
6147 {
6148     extern int console_loglevel;
6149     console_loglevel = 15;
6150 }
6151
6152 #define DO_ERROR(trapnr, signr, str, name, tsk) \
6153 asmlinkage void do_##name(struct pt_regs * regs, \
6154                          long error_code) \
6155 { \
6156     tsk->tss.error_code = error_code; \
6157     tsk->tss.trap_no = trapnr; \
6158     force_sig(signr, tsk); \
6159     die_if_no_fixup(str,regs,error_code); \
6160 }
6161
6162 #define DO_VM86_ERROR(trapnr, signr, str, name, tsk) \
6163 asmlinkage void do_##name(struct pt_regs * regs, \
6164                          long error_code) \
6165 { \
6166     lock_kernel(); \
6167     if (regs->eflags & VM_MASK) { \
6168         if (!handle_vm86_trap((struct kernel_vm86_regs *) \
6169                             regs, error_code, trapnr)) \
6170             goto out; \
6171         /* else fall through */ \
6172     } \
6173     tsk->tss.error_code = error_code; \
6174     tsk->tss.trap_no = trapnr; \
6175     force_sig(signr, tsk); \
6176     die_if_kernel(str,regs,error_code); \
6177 out: \
6178     unlock_kernel(); \
6179 }
6180
6181 void page_exception(void);
6182
6183 asmlinkage void divide_error(void);
6184 asmlinkage void debug(void);
6185 asmlinkage void nmi(void);
6186 asmlinkage void int3(void);
6187 asmlinkage void overflow(void);
6188 asmlinkage void bounds(void);
6189 asmlinkage void invalid_op(void);
6190 asmlinkage void device_not_available(void);
6191 asmlinkage void double_fault(void);
6192 asmlinkage void coprocessor_segment_overrun(void);
6193 asmlinkage void invalid_TSS(void);
6194 asmlinkage void segment_not_present(void);
6195 asmlinkage void stack_segment(void);
6196 asmlinkage void general_protection(void);
6197 asmlinkage void page_fault(void);
6198 asmlinkage void coprocessor_error(void);
6199 asmlinkage void reserved(void);
6200 asmlinkage void alignment_check(void);
6201 asmlinkage void spurious_interrupt_bug(void);
6202
6203 int kstack_depth_to_print = 24;
6204
6205 /* These constants are for searching for possible module
6206 * text segments. VMALLOC_OFFSET comes from
6207 * mm/vmalloc.c; MODULE_RANGE is a guess of how much
6208 * space is likely to be vmalloced. */
6209 #define VMALLOC_OFFSET (8*1024*1024)
6210 #define MODULE_RANGE (8*1024*1024)
6211
6212 static void show_registers(struct pt_regs *regs)
6213 {
6214     int i;
6215     int in_kernel = 1;
6216     unsigned long esp;
6217     unsigned short ss;
6218     unsigned long *stack, addr, module_start, module_end;
6219
6220     esp = (unsigned long) (1+regs);

```

```

6221 ss = __KERNEL_DS;
6222 if (regs->xcs & 3) {
6223     in_kernel = 0;
6224     esp = regs->esp;
6225     ss = regs->xss & 0xffff;
6226 }
6227 printk("CPU:   %d\nEIP:   %04x:[<%08lx>]"
6228        "\nEFLAGS: %08lx\n", smp_processor_id(),
6229        0xffff & regs->xcs, regs->eip, regs->eflags);
6230 printk("eax: %08lx ebx: %08lx ecx: %08lx "
6231        "edx: %08lx\n",
6232        regs->eax, regs->ebx, regs->ecx, regs->edx);
6233 printk("esi: %08lx edi: %08lx ebp: %08lx "
6234        "esp: %08lx\n",
6235        regs->esi, regs->edi, regs->ebp, esp);
6236 printk("ds: %04x es: %04x ss: %04x\n",
6237        regs->xds & 0xffff, regs->xes & 0xffff, ss);
6238 store_TR(i);
6239 printk("Process %s (pid: %d, process nr: %d, "
6240        "stackpage=%08lx)", current->comm, current->pid,
6241        0xffff & i, 4096+(unsigned long)current);
6242
6243 /* When in-kernel, we also print out the stack and code
6244  * at the time of the fault.. */
6245 if (in_kernel) {
6246     printk("\nStack: ");
6247     stack = (unsigned long *) esp;
6248     for(i=0; i < kstack_depth_to_print; i++) {
6249         if (((long) stack & 4095) == 0)
6250             break;
6251         if (i && ((i % 8) == 0))
6252             printk("\n      ");
6253         printk("%08lx ", *stack++);
6254     }
6255     printk("\nCall Trace: ");
6256     stack = (unsigned long *) esp;
6257     i = 1;
6258     module_start = PAGE_OFFSET + (max_mapnr<<PAGE_SHIFT);
6259     module_start = ((module_start + VMALLOC_OFFSET) &
6260                    ~(VMALLOC_OFFSET-1));
6261     module_end = module_start + MODULE_RANGE;
6262     while (((long) stack & 4095) != 0) {
6263         addr = *stack++;
6264         /* If the address is either in the text segment of
6265          * the kernel, or in the region which contains
6266          * vmalloc'ed memory, it *may* be the address of a
6267          * calling routine; if so, print it so that someone
6268          * tracing down the cause of the crash will be able

```

```

6269         * to figure out the call path that was taken. */
6270         if (((addr >= (unsigned long) &_stext) &&
6271             (addr <= (unsigned long) &_etext)) ||
6272             ((addr >= module_start) &&
6273              (addr <= module_end))) {
6274             if (i && ((i % 8) == 0))
6275                 printk("\n      ");
6276             printk("[<%08lx>] ", addr);
6277             i++;
6278         }
6279     }
6280     printk("\nCode: ");
6281     for(i=0;i<20;i++)
6282         printk("%02x ", ((unsigned char *)regs->eip)[i]);
6283 }
6284 printk("\n");
6285 }
6286
6287 spinlock_t die_lock;
6288
6289 void die(const char * str, struct pt_regs * regs,
6290         long err)
6291 {
6292     console_verbose();
6293     spin_lock_irq(&die_lock);
6294     printk("%s: %04lx\n", str, err & 0xffff);
6295     show_registers(regs);
6296     spin_unlock_irq(&die_lock);
6297     do_exit(SIGSEGV);
6298 }
6299
6300 static inline void die_if_kernel(const char * str,
6301                                 struct pt_regs * regs, long err)
6302 {
6303     if (!(regs->eflags & VM_MASK) && !(3 & regs->xcs))
6304         die(str, regs, err);
6305 }
6306
6307 static void die_if_no_fixup(const char * str,
6308                             struct pt_regs * regs, long err)
6309 {
6310     if (!(regs->eflags & VM_MASK) && !(3 & regs->xcs))
6311     {
6312         unsigned long fixup;
6313         fixup = search_exception_table(regs->eip);
6314         if (fixup) {
6315             regs->eip = fixup;
6316             return;

```

```

6317     }
6318     die(str, regs, err);
6319 }
6320 }
6321
6322 DO_VM86_ERROR( 0, SIGFPE, "divide error", divide_error,
6323               current)
6324 DO_VM86_ERROR( 3, SIGTRAP, "int3", int3, current)
6325 DO_VM86_ERROR( 4, SIGSEGV, "overflow", overflow, current)
6326 DO_VM86_ERROR( 5, SIGSEGV, "bounds", bounds, current)
6327 DO_ERROR( 6, SIGILL, "invalid operand", invalid_op,
6328          current)
6329 DO_VM86_ERROR( 7, SIGSEGV, "device not available",
6330               device_not_available, current)
6331 DO_ERROR( 8, SIGSEGV, "double fault", double_fault,
6332          current)
6333 DO_ERROR( 9, SIGFPE, "coprocessor segment overrun",
6334          coprocessor_segment_overrun, current)
6335 DO_ERROR(10, SIGSEGV, "invalid TSS", invalid_TSS,
6336          current)
6337 DO_ERROR(11, SIGBUS, "segment not present",
6338          segment_not_present, current)
6339 DO_ERROR(12, SIGBUS, "stack segment", stack_segment,
6340          current)
6341 DO_ERROR(17, SIGSEGV, "alignment check", alignment_check,
6342          current)
6343 DO_ERROR(18, SIGSEGV, "reserved", reserved, current)
6344 /* I don't have documents for this but it does seem to
6345  * cover the cache flush from user space exception some
6346  * people get. */
6347 DO_ERROR(19, SIGSEGV, "cache flush denied",
6348          cache_flush_denied, current)
6349
6350 asmlinkage void cache_flush_denied(struct pt_regs * regs,
6351                                   long error_code)
6352 {
6353     if (regs->eflags & VM_MASK) {
6354         handle_vm86_fault((struct kernel_vm86_regs *) regs,
6355                          error_code);
6356         return;
6357     }
6358     die_if_kernel("cache flush denied",regs,error_code);
6359     current->tss.error_code = error_code;
6360     current->tss.trap_no = 19;
6361     force_sig(SIGSEGV, current);
6362 }
6363
6364 asmlinkage void do_general_protection(

```

```

6365     struct pt_regs * regs, long error_code)
6366 {
6367     if (regs->eflags & VM_MASK)
6368         goto gp_in_vm86;
6369
6370     if (!(regs->xcs & 3))
6371         goto gp_in_kernel;
6372
6373     current->tss.error_code = error_code;
6374     current->tss.trap_no = 13;
6375     force_sig(SIGSEGV, current);
6376     return;
6377
6378 gp_in_vm86:
6379     lock_kernel();
6380     handle_vm86_fault((struct kernel_vm86_regs *) regs,
6381                      error_code);
6382     unlock_kernel();
6383     return;
6384
6385 gp_in_kernel:
6386     {
6387         unsigned long fixup;
6388         fixup = search_exception_table(regs->eip);
6389         if (fixup) {
6390             regs->eip = fixup;
6391             return;
6392         }
6393         die("general protection fault", regs, error_code);
6394     }
6395 }
6396
6397 static void mem_parity_error(unsigned char reason,
6398                             struct pt_regs * regs)
6399 {
6400     printk("Uhhuh. NMI received. Dazed and confused, "
6401           "but trying to continue\n");
6402     printk("You probably have a hardware problem with "
6403           "your RAM chips\n");
6404 }
6405
6406 static void io_check_error(unsigned char reason,
6407                            struct pt_regs * regs)
6408 {
6409     unsigned long i;
6410
6411     printk("NMI: IOCK error (debug interrupt?)\n");
6412     show_registers(regs);

```



```

6413
6414 /* Re-enable the IOCK line, wait for a few seconds */
6415 reason |= 8;
6416 outb(reason, 0x61);
6417 i = 2000;
6418 while (--i) udelay(1000);
6419 reason &= ~8;
6420 outb(reason, 0x61);
6421 }
6422
6423 static void unknown_nmi_error(unsigned char reason,
6424                               struct pt_regs * regs)
6425 {
6426 #ifdef CONFIG_MCA
6427 /* Might actually be able to figure out what the guilty
6428  * party is. */
6429 if( MCA_bus ) {
6430     mca_handle_nmi();
6431     return;
6432 }
6433 #endif
6434 printk("Uhhuh. NMI received for unknown reason %02x.\n"
6435        , reason);
6436 printk("Dazed and confused, but trying to continue\n");
6437 printk("Do you have a strange power saving mode "
6438        "enabled?\n");
6439 }
6440
6441 asmlinkage void do_nmi(struct pt_regs * regs,
6442                       long error_code)
6443 {
6444     unsigned char reason = inb(0x61);
6445     extern atomic_t nmi_counter;
6446
6447     atomic_inc(&nmi_counter);
6448     if (reason & 0x80)
6449         mem_parity_error(reason, regs);
6450     if (reason & 0x40)
6451         io_check_error(reason, regs);
6452     if (!(reason & 0xc0))
6453         unknown_nmi_error(reason, regs);
6454 }
6455
6456 /* Careful - we must not do a lock-kernel until we have
6457  * checked that the debug fault happened in user
6458  * mode. Getting debug exceptions while in the kernel has
6459  * to be handled without locking, to avoid deadlocks..
6460  *

```

```

6461 * Being careful here means that we don't have to be as
6462 * careful in a lot of more complicated places (task
6463 * switching can be a bit lazy about restoring all the
6464 * debug state, and ptrace doesn't have to find every
6465 * occurrence of the TF bit that could be saved away even
6466 * by user code - and we don't have to be careful about
6467 * what values can be written to the debug registers
6468 * because there are no really bad cases). */
6469 asmlinkage void do_debug(struct pt_regs * regs,
6470                          long error_code)
6471 {
6472     unsigned int condition;
6473     struct task_struct *tsk = current;
6474
6475     if (regs >eflags & VM_MASK)
6476         goto debug_vm86;
6477
6478     __asm__ __volatile__ ("movl %%db6,%0" : "=r"
6479                          (condition));
6480
6481     /* Mask out spurious TF errors due to lazy TF
6482      * clearing */
6483     if (condition & DR_STEP) {
6484         /* The TF error should be masked out only if the
6485          * current process is not traced and if the TRAP flag
6486          * has been set previously by a tracing process
6487          * (condition detected by the PF_DTRACE flag);
6488          * remember that the i386 TRAP flag can be modified
6489          * by the process itself in user mode, allowing
6490          * programs to debug themselves without the ptrace()
6491          * interface. */
6492         if ((tsk->flags & (PF_DTRACE|PF_PTRACED)) ==
6493             PF_DTRACE)
6494             goto clear_TF;
6495     }
6496
6497     /* Mask out spurious debug traps due to lazy DR7
6498      * setting */
6499     if (condition & (DR_TRAP0|DR_TRAP1|DR_TRAP2|DR_TRAP3)){
6500         if (!tsk->tss.debugreg[7])
6501             goto clear_dr7;
6502     }
6503
6504     /* If this is a kernel mode trap, we need to reset db/
6505      * to allow us to continue sanely */
6506     if ((regs->xcs & 3) == 0)
6507         goto clear_dr7;
6508

```

```

6509  /* Ok, finally something we can handle */
6510  tsk->tss.trap_no = 1;
6511  tsk->tss.error_code = error_code;
6512  force_sig(SIGTRAP, tsk);
6513  return;
6514
6515  debug_vm86:
6516  lock_kernel();
6517  handle_vm86_trap((struct kernel_vm86_regs *) regs,
6518                  error_code, 1);
6519  unlock_kernel();
6520  return;
6521
6522  clear_dr7:
6523  __asm__("movl %0,%db7"
6524         : /* no output */
6525         : "r" (0));
6526  return;
6527
6528  clear_TF:
6529  regs->eflags &= ~TF_MASK;
6530  return;
6531 }
6532
6533 /* Note that we play around with the 'TS' bit in an
6534 * attempt to get the correct behaviour even in the
6535 * presence of the asynchronous IRQ13 behaviour */
6536 void math_error(void)
6537 {
6538  struct task_struct * task;
6539
6540  /* Save the info for the exception handler (this will
6541   * also clear the error) */
6542  task = current;
6543  save_fpu(task);
6544  task->tss.trap_no = 16;
6545  task->tss.error_code = 0;
6546  force_sig(SIGFPE, task);
6547 }
6548
6549 asmlinkage void do_coprocessor_error(
6550 struct pt_regs * regs, long error_code)
6551 {
6552  ignore_irq13 = 1;
6553  math_error();
6554 }
6555
6556 asmlinkage void do_spurious_interrupt_bug(

```

```

6557 struct pt_regs * regs, long error_code)
6558 {
6559  #if 0
6560  /* No need to warn about this any longer. */
6561  printk("Ignoring P6 Local APIC Spurious Interrupt "
6562         "Bug...\n");
6563  #endif
6564 }
6565
6566 /* 'math_state_restore()' saves the current math
6567 * information in the old math state array, and gets the
6568 * new ones from the current task
6569 *
6570 * Careful.. There are problems with IBM-designed IRQ13
6571 * behaviour. Don't touch unless you *really* know how
6572 * it works. */
6573 asmlinkage void math_state_restore(struct pt_regs regs)
6574 {
6575  /* Allow maths ops (or we recurse) */
6576  __asm__ __volatile__("clts");
6577  if(current->used_math)
6578  __asm__("frstor %0": : "m" (current->tss.i387));
6579  else
6580  {
6581    /* Our first FPU usage, clean the chip. */
6582    __asm__("fninit");
6583    current->used_math = 1;
6584  }
6585  /* So we fnsave on switch_to() */
6586  current->flags |= PF_USED_FPU;
6587 }
6588
6589 #ifndef CONFIG_MATH_EMULATION
6590
6591 asmlinkage void math_emulate(long arg)
6592 {
6593  lock_kernel();
6594  printk("math-emulation not enabled and no coprocessor "
6595         "found.\n");
6596  printk("killing %s.\n", current->comm);
6597  force_sig(SIGFPE, current);
6598  schedule();
6599  unlock_kernel();
6600 }
6601
6602 #endif /* CONFIG_MATH_EMULATION */
6603
6604 __initfunc(void trap_init_f00f_bug(void))

```

```

6605 {
6606     unsigned long page;
6607     pgd_t * pgd;
6608     pmd_t * pmd;
6609     pte_t * pte;
6610
6611     /* Allocate a new page in virtual address space, move
6612      * the IDT into it and write protect this page. */
6613     page = (unsigned long) vmalloc(PAGE_SIZE);
6614     pgd = pgd_offset(&init_mm, page);
6615     pmd = pmd_offset(pgd, page);
6616     pte = pte_offset(pmd, page);
6617     free_page(pte_page(*pte));
6618     *pte = mk_pte(&idt_table, PAGE_KERNEL_RO);
6619     local_flush_tlb();
6620
6621     /* "idt" is magic - it overlaps the idt_descr variable
6622      * so that updating idt will automatically update the
6623      * idt descriptor.. */
6624     idt = (struct desc_struct *)page;
6625     __asm__ __volatile__ ("lidt %0": "=m" (idt_descr));
6626 }
6627
6628 #define _set_gate(gate_addr, type, dpl, addr) \
6629 do { \
6630     int __d0, __d1; \
6631     __asm__ __volatile__ ("movw %%dx, %%ax\n\t" \
6632      "movw %4, %%dx\n\t" \
6633      "movl %%eax, %0\n\t" \
6634      "movl %%edx, %1" \
6635      : "=m" (*((long *) (gate_addr))), \
6636      "=m" (*(1+(long *) (gate_addr))), "=&a" (__d0), \
6637      "&d" (__d1) \
6638      : "i" ((short) (0x8000+(dpl<<13)+(type<<8))), \
6639      "3" ((char *) (addr)), "2" (__KERNEL_CS << 16)); \
6640 } while (0)
6641
6642
6643 /* This needs to use 'idt_table' rather than 'idt', and
6644 * thus use the _nonmapped_ version of the IDT, as the
6645 * Pentium F0 0F bugfix can have resulted in the mapped
6646 * IDT being write-protected. */
6647 void set_intr_gate(unsigned int n, void *addr)
6648 {
6649     _set_gate(idt_table+n, 14, 0, addr);
6650 }
6651
6652 static void __init set_trap_gate(unsigned int n,

```

```

void *addr)
6653
6654 {
6655     _set_gate(idt_table+n, 15, 0, addr);
6656 }
6657
6658 static void __init set_system_gate(unsigned int n,
6659     void *addr)
6660 {
6661     _set_gate(idt_table+n, 15, 3, addr);
6662 }
6663
6664 static void __init set_call_gate(void *a, void *addr)
6665 {
6666     _set_gate(a, 12, 3, addr);
6667 }
6668
6669 #define _set_seg_desc(gate_addr, type, dpl, base, limit) { \
6670     *((gate_addr)+1) = ((base) & 0xff000000) | \
6671     (((base) & 0x00ff0000)>>16) | \
6672     ((limit) & 0xf0000) | \
6673     ((dpl)<<13) | \
6674     (0x00408000) | \
6675     ((type)<<8); \
6676     *(gate_addr) = (((base) & 0x0000ffff)<<16) | \
6677     ((limit) & 0x0ffff); }
6678
6679 #define _set_tssldt_desc(n, addr, limit, type) \
6680 __asm__ __volatile__ ("movw %3.0(%2)\n\t" \
6681      "movw %%ax, 2(%2)\n\t" \
6682      "rorl $16, %%eax\n\t" \
6683      "movb %%al, 4(%2)\n\t" \
6684      "movb %4, 5(%2)\n\t" \
6685      "movb $0, 6(%2)\n\t" \
6686      "movb %%ah, 7(%2)\n\t" \
6687      "rorl $16, %%eax" \
6688      : "=m"(*n) : "a" (addr), "r"(n), "ir"(limit), \
6689      "i"(type))
6690
6691 void set_tss_desc(unsigned int n, void *addr)
6692 {
6693     _set_tssldt_desc(gdt_table+FIRST_TSS_ENTRY+(n<<1),
6694         (int)addr, 235, 0x89);
6695 }
6696
6697 void set_ldt_desc(unsigned int n, void *addr,
6698     unsigned int size)
6699 {
6700     _set_tssldt_desc(gdt_table+FIRST_LDT_ENTRY+(n<<1),

```

```

6701             (int)addr, ((size << 3) - 1), 0x82);
6702 }
6703
6704 #ifdef CONFIG_X86_VISWS_APIC
6705
6706 /* On Rev 005 motherboards legacy device interrupt lines
6707 * are wired directly to Lithium from the 307. But the
6708 * PROM leaves the interrupt type of each 307 logical
6709 * device set appropriate for the 8259. Later we'll
6710 * actually use the 8259, but for now we have to flip the
6711 * interrupt types to level triggered, active lo as
6712 * required by Lithium. */
6713 #define REG 0x2e /* The register to read/write */
6714 #define DEV 0x07 /* Register: Logical device select */
6715 #define VAL 0x2f /* The value to read/write */
6716
6717 static void
6718 superio_outb(int dev, int reg, int val)
6719 {
6720     outb(DEV, REG);
6721     outb(dev, VAL);
6722     outb(reg, REG);
6723     outb(val, VAL);
6724 }
6725
6726 static int __attribute__((unused))
6727 superio_inb(int dev, int reg)
6728 {
6729     outb(DEV, REG);
6730     outb(dev, VAL);
6731     outb(reg, REG);
6732     return inb(VAL);
6733 }
6734
6735 #define FLOP 3 /* floppy logical device */
6736 #define PPORT 4 /* parallel logical device */
6737 #define UART5 5 /* uart2 logical device (not wired up) */
6738 #define UART6 6 /* uart1 logical device
6739 * (THIS is the serial port!) */
6740 #define IDEST 0x70 /* int. destination
6741 * (which 307 IRQ line) reg. */
6742 #define ITYPE 0x71 /* interrupt type register */
6743
6744 /* interrupt type bits */
6745 #define LEVEL 0x01 /* bit 0, 0 == edge triggered */
6746 #define ACTHI 0x02 /* bit 1, 0 == active lo */
6747
6748 static void

```

```

6749 superio_init(void)
6750 {
6751     if (visws_board_type == VISWS_320 &&
6752         visws_board_rev == 5) {
6753         /* 0 means no intr propagated */
6754         superio_outb(UART6, IDEST, 0);
6755         printk("SGI 320 rev 5: "
6756             "disabling 307 uart1 interrupt\n");
6757     }
6758 }
6759
6760 static void
6761 lithium_init(void)
6762 {
6763     set_fixmap(FIX_LI_PCIA, LI_PCI_A_PHYS);
6764     printk("Lithium PCI Bridge A, Bus Number: %d\n",
6765         li_pcia_read16(LI_PCI_BUSNUM) & 0xff);
6766     set_fixmap(FIX_LI_PCIB, LI_PCI_B_PHYS);
6767     printk("Lithium PCI Bridge B (PIIX4), Bus Number: "
6768         "%d\n", li_pcib_read16(LI_PCI_BUSNUM) & 0xff);
6769
6770     /* XXX blindly enables all interrupts */
6771     li_pcia_writel6(LI_PCI_INTEN, 0xffff);
6772     li_pcib_writel6(LI_PCI_INTEN, 0xffff);
6773 }
6774
6775 static void
6776 cobalt_init(void)
6777 {
6778     /* On normal SMP PC this is used only with SMP, but we
6779     * have to use it and set it up here to start the
6780     * Cobalt clock */
6781     set_fixmap(FIX_APIC_BASE, APIC_PHYS_BASE);
6782     printk("Local APIC ID %1x\n", apic_read(APIC_ID));
6783     printk("Local APIC Version %1x\n",
6784         apic_read(APIC_VERSION));
6785
6786     set_fixmap(FIX_CO_CPU, CO_CPU_PHYS);
6787     printk("Cobalt Revision %1x\n",
6788         co_cpu_read(CO_CPU_REV));
6789
6790     set_fixmap(FIX_CO_APIC, CO_APIC_PHYS);
6791     printk("Cobalt APIC ID %1x\n",
6792         co_apic_read(CO_APIC_ID));
6793
6794     /* Enable Cobalt APIC being careful to NOT change the
6795     * ID! */
6796     co_apic_write(CO_APIC_ID,

```

```

6797             co_apic_read(CO_APIC_ID|CO_APIC_ENABLE);
6798
6799     printk("Cobalt APIC enabled: ID reg %1x\n",
6800            co_apic_read(CO_APIC_ID));
6801 }
6802 #endif
6803 void __init trap_init(void)
6804 {
6805     if (readl(0xOFFFD9) ==
6806         'E' + ('I' << 8) + ('S' << 16) + ('A' << 24))
6807         EISA_bus = 1;
6808     set_call_gate(&default_ldt, lcall7);
6809     set_trap_gate(0, &divide_error);
6810     set_trap_gate(1, &debug);
6811     set_trap_gate(2, &nmi);
6812     /* int3-5 can be called from all */
6813     set_system_gate(3, &int3);
6814     set_system_gate(4, &overflow);
6815     set_system_gate(5, &bounds);
6816     set_trap_gate(6, &invalid_op);
6817     set_trap_gate(7, &device_not_available);
6818     set_trap_gate(8, &double_fault);
6819     set_trap_gate(9, &coprocessor_segment_overnrun);
6820     set_trap_gate(10, &invalid_TSS);
6821     set_trap_gate(11, &segment_not_present);
6822     set_trap_gate(12, &stack_segment);
6823     set_trap_gate(13, &general_protection);
6824     set_trap_gate(14, &page_fault);
6825     set_trap_gate(15, &spurious_interrupt_bug);
6826     set_trap_gate(16, &coprocessor_error);
6827     set_trap_gate(17, &alignment_check);
6828     set_system_gate(SYSCALL_VECTOR, &system_call);
6829
6830     /* set up GDT task & ldt entries */
6831     set_tss_desc(0, &init_task.tss);
6832     set_ldt_desc(0, &default_ldt, 1);
6833
6834     /* Clear NT, so that we won't have troubles with that
6835      * later on */
6836     __asm__("pushfl ; andl $0xffffbfff, (%esp) ; popfl");
6837     load_TR(0);
6838     load_ldt(0);
6839 #ifdef CONFIG_X86_VISWS_APIC
6840     superio_init();
6841     lithium_init();
6842     cobalt_init();
6843 #endif
6844 }

```

arch/i386/lib/delay.c

```

6845 /*
6846  *   Precise Delay Loops for i386
6847  *
6848  *   Copyright (C) 1993 Linus Torvalds
6849  *   Copyright (C) 1997 Martin Mares
6850  *   <mj@atrey.karlin.mff.cuni.cz>
6851  *
6852  * The __delay function must _NOT_ be inlined as its
6853  * execution time depends wildly on alignment on many x86
6854  * processors. The additional jump magic is needed to get
6855  * the timing stable on all the CPU's we have to worry
6856  * about.
6857  */
6858
6859 #include <linux/sched.h>
6860 #include <linux/delay.h>
6861
6862 #ifdef __SMP__
6863 #include <asm/smp.h>
6864 #endif
6865
6866 void __delay(unsigned long loops)
6867 {
6868     int d0;
6869     __asm__ __volatile__(
6870         "\tjmp 1f\n"
6871         ".align 16\n"
6872         "1:\tjmp 2f\n"
6873         ".align 16\n"
6874         "2:\tdecl %0\n\tjns 2b"
6875         : "=a" (d0)
6876         : "0" (loops));
6877 }
6878
6879 inline void __const_udelay(unsigned long xloops)
6880 {
6881     int d0;
6882     __asm__("mull %0"
6883         : "=d" (xloops), "=a" (d0)
6884         : "1" (xloops), "0" (current_cpu_data.loops_per_sec));
6885     __delay(xloops);
6886 }
6887
6888 void __udelay(unsigned long usecs)
6889 {
6890     __const_udelay(usecs * 0x000010c6); /* 2**32/1000000 */
6891 }

```

arch/i386/mm/fault.c

```
6892 /*
6893  * linux/arch/i386/mm/fault.c
6894  *
6895  * Copyright (C) 1995 Linus Torvalds
6896  */
6897
6898 #include <linux/signal.h>
6899 #include <linux/sched.h>
6900 #include <linux/kernel.h>
6901 #include <linux/errno.h>
6902 #include <linux/string.h>
6903 #include <linux/types.h>
6904 #include <linux/ptrace.h>
6905 #include <linux/mman.h>
6906 #include <linux/mm.h>
6907 #include <linux/smp.h>
6908 #include <linux/smp_lock.h>
6909 #include <linux/interrupt.h>
6910
6911 #include <asm/system.h>
6912 #include <asm/uaccess.h>
6913 #include <asm/pgtable.h>
6914 #include <asm/hardirq.h>
6915
6916 extern void die(const char *,struct pt_regs *,long);
6917
6918 /* Ugly, ugly, but the goto's result in better assembly..
6919  */
6920 int __verify_write(const void * addr, unsigned long size)
6921 {
6922     struct vm_area_struct * vma;
6923     unsigned long start = (unsigned long) addr;
6924
6925     if (!size)
6926         return 1;
6927
6928     vma = find_vma(current->mm, start);
6929     if (!vma)
6930         goto bad_area;
6931     if (vma->vm_start > start)
6932         goto check_stack;
6933
6934 good_area:
6935     if (!(vma->vm_flags & VM_WRITE))
6936         goto bad_area;
6937     size--;
6938     size += start & ~PAGE_MASK;
```

```
6939     size >>= PAGE_SHIFT;
6940     start &= PAGE_MASK;
6941
6942     for (;;) {
6943         handle_mm_fault(current,vma, start, 1);
6944         if (!size)
6945             break;
6946         size--;
6947         start += PAGE_SIZE;
6948         if (start < vma->vm_end)
6949             continue;
6950         vma = vma->vm_next;
6951         if (!vma || vma->vm_start != start)
6952             goto bad_area;
6953         if (!(vma->vm_flags & VM_WRITE))
6954             goto bad_area;;
6955     }
6956     return 1;
6957
6958 check_stack:
6959     if (!(vma->vm_flags & VM_GROWSDOWN))
6960         goto bad_area;
6961     if (expand_stack(vma, start) == 0)
6962         goto good_area;
6963
6964 bad_area:
6965     return 0;
6966 }
6967
6968 asmlinkage void do_invalid_op(struct pt_regs *,
6969                               unsigned long);
6970 extern unsigned long idt;
6971
6972 /* This routine handles page faults. It determines the
6973  * address, and the problem, and then passes it off to
6974  * one of the appropriate routines.
6975  *
6976  * error_code:
6977  *     bit 0 == 0 means no page found, 1 means prot fault
6978  *     bit 1 == 0 means read, 1 means write
6979  *     bit 2 == 0 means kernel, 1 means user-mode */
6980 asmlinkage void do_page_fault(struct pt_regs *regs,
6981                               unsigned long error_code)
6982 {
6983     struct task_struct *tsk;
6984     struct mm_struct *mm;
6985     struct vm_area_struct * vma;
6986     unsigned long address;
```

```

6987 unsigned long page;
6988 unsigned long fixup;
6989 int write;
6990
6991 /* get the address */
6992 __asm__("movl %%cr2,%0":"=r" (address));
6993
6994 tsk = current;
6995 mm = tsk->mm;
6996
6997 /* If we're in an interrupt or have no user context, we
6998  * must not take the fault.. */
6999 if (in_interrupt() || mm == &init_mm)
7000     goto no_context;
7001
7002 down(&mm->mmap_sem);
7003
7004 vma = find_vma(mm, address);
7005 if (!vma)
7006     goto bad_area;
7007 if (vma->vm_start <= address)
7008     goto good_area;
7009 if (!(vma->vm_flags & VM_GROWSDOWN))
7010     goto bad_area;
7011 if (error_code & 4) {
7012     /* accessing the stack below %esp is always a bug.
7013     * The "+ 32" is there due to some instructions (like
7014     * pusha) doing post decrement on the stack and that
7015     * doesn't show up until later.. */
7016     if (address + 32 < regs->esp)
7017         goto bad_area;
7018 }
7019 if (expand_stack(vma, address))
7020     goto bad_area;
7021 /* Ok, we have a good vm_area for this memory access, so
7022  * we can handle it.. */
7023 good_area:
7024 write = 0;
7025 switch (error_code & 3) {
7026     default: /* 3: write, present */
7027 #ifdef TEST_VERIFY_AREA
7028     if (regs->cs == KERNEL_CS)
7029         printk("WP fault at %08lx\n", regs->eip);
7030 #endif
7031     /* fall through */
7032     case 2: /* write, not present */
7033     if (!(vma > vm flags & VM_WRITE))
7034         goto bad_area;

```

```

7035     write++;
7036     break;
7037     case 1: /* read, present */
7038     goto bad_area;
7039     case 0: /* read, not present */
7040     if (!(vma->vm_flags & (VM_READ | VM_EXEC)))
7041         goto bad_area;
7042 }
7043
7044 /* If for any reason at all we couldn't handle the
7045  * fault, make sure we exit gracefully rather than
7046  * endlessly redo the fault. */
7047 if (!handle_mm_fault(tsk, vma, address, write))
7048     goto do_sigbus;
7049
7050 /* Did it hit the DOS screen mem VA from vm86 mode? */
7051 if (regs->eflags & VM_MASK) {
7052     unsigned long bit = (address - 0xA0000) >> PAGE_SHIFT;
7053     if (bit < 32)
7054         tsk->tss.screen_bitmap |= 1 << bit;
7055 }
7056 up(&mm->mmap_sem);
7057 return;
7058
7059 /* Something tried to access memory that isn't in our
7060  * memory map.. Fix it, but check if it's kernel or user
7061  * first.. */
7062 bad_area:
7063 up(&mm->mmap_sem);
7064
7065 /* User mode accesses just cause a SIGSEGV */
7066 if (error_code & 4) {
7067     tsk->tss.cr2 = address;
7068     tsk->tss.error_code = error_code;
7069     tsk->tss.trap_no = 14;
7070     force_sig(SIGSEGV, tsk);
7071     return;
7072 }
7073
7074 /* Pentium F0 0F C7 C8 bug workaround. */
7075 if (boot_cpu_data.f00f_bug) {
7076     unsigned long nr;
7077
7078     nr = (address - idt) >> 3;
7079
7080     if (nr == 6) {
7081         do_invalid_op(regs, 0);
7082         return;

```

→ p.547

```

7083     }
7084 }
7085
7086 no_context:
7087 /* Are we prepared to handle this kernel fault? */
7088 if ((fixup = search_exception_table(regs->eip)) != 0) {
7089     regs->eip = fixup;
7090     return;
7091 }
7092
7093 /* Oops. The kernel tried to access some bad page. We'll
7094 * have to terminate things with extreme prejudice.
7095 * First we check if it was the bootup rw-test, though..
7096 */
7097 if (boot_cpu_data.wp_works_ok < 0 &&
7098     address == PAGE_OFFSET && (error_code & 1)) {
7099     boot_cpu_data.wp_works_ok = 1;
7100     pg0[0] = pte_val(mk_pte(PAGE_OFFSET, PAGE_KERNEL));
7101     local_flush_tlb();
7102     /* Beware: Black magic here. The printk is needed
7103     * here to flush CPU state on certain buggy
7104     * processors. */
7105     printk("Ok");
7106     return;
7107 }
7108
7109 if (address < PAGE_SIZE)
7110     printk(KERN_ALERT "Unable to handle kernel "
7111            "NULL pointer dereference");
7112 else
7113     printk(KERN_ALERT "Unable to handle kernel "
7114            "paging request");
7115 printk(" at virtual address %08lx\n", address);
7116 __asm__("movl %%cr3,%0" : "=r" (page));
7117 printk(KERN_ALERT "current->tss.cr3 = %08lx, "
7118        "%%cr3 = %08lx\n", tsk->tss.cr3, page);
7119 page = ((unsigned long *) __va(page))[address >> 22];
7120 printk(KERN_ALERT "*pde = %08lx\n", page);
7121 if (page & 1) {
7122     page &= PAGE_MASK;
7123     address &= 0x003ff000;
7124     page = ((unsigned long *)
7125            __va(page))[address >> PAGE_SHIFT];
7126     printk(KERN_ALERT "*pte = %08lx\n", page);
7127 }
7128 die("Oops", regs, error_code);
7129 do_exit(SIGKILL);
7130

```

```

7131 /* We ran out of memory, or some other thing happened to
7132 * us that made us unable to handle the page fault
7133 * gracefully. */
7134 do_sigbus:
7135     up(&mm->mmap_sem);
7136
7137     /* Send a sigbus, regardless of whether we were in
7138     * kernel or user mode. */
7139     tsk->tss.cr2 = address;
7140     tsk->tss.error_code = error_code;
7141     tsk->tss.trap_no = 14;
7142     force_sig(SIGBUS, tsk);
7143
7144     /* Kernel mode? Handle exceptions or die */
7145     if (!(error_code & 4))
7146         goto no_context;
7147 }

```

arch/i386/mm/init.c

```

7148 /*
7149 * linux/arch/i386/mm/init.c
7150 *
7151 * Copyright (C) 1995 Linus Torvalds
7152 */
7153
7154 #include <linux/config.h>
7155 #include <linux/signal.h>
7156 #include <linux/sched.h>
7157 #include <linux/kernel.h>
7158 #include <linux/errno.h>
7159 #include <linux/string.h>
7160 #include <linux/types.h>
7161 #include <linux/ptrace.h>
7162 #include <linux/mman.h>
7163 #include <linux/mm.h>
7164 #include <linux/swap.h>
7165 #include <linux/smp.h>
7166 #include <linux/init.h>
7167 #ifdef CONFIG_BLK_DEV_INITRD
7168 #include <linux/blk.h>
7169 #endif
7170
7171 #include <asm/processor.h>
7172 #include <asm/system.h>
7173 #include <asm/uaccess.h>
7174 #include <asm/pgtable.h>
7175 #include <asm/dma.h>

```



```

7176 #include <asm/fixmap.h>
7177
7178 extern void show_net_buffers(void);
7179 extern unsigned long init_smp_mappings(unsigned long);
7180
7181 void __bad_pte_kernel(pmd_t *pmd)
7182 {
7183     printk("Bad pmd in pte_alloc: %08lx\n", pmd_val(*pmd));
7184     pmd_val(*pmd) = _KERNPG_TABLE + __pa(BAD_PAGETABLE);
7185 }
7186
7187 void __bad_pte(pmd_t *pmd)
7188 {
7189     printk("Bad pmd in pte_alloc: %08lx\n", pmd_val(*pmd));
7190     pmd_val(*pmd) = _PAGE_TABLE + __pa(BAD_PAGETABLE);
7191 }
7192
7193 pte_t *get_pte_kernel_slow(pmd_t *pmd,
7194                             unsigned long offset)
7195 {
7196     pte_t *pte;
7197
7198     pte = (pte_t *) __get_free_page(GFP_KERNEL);
7199     if (pmd_none(*pmd)) {
7200         if (pte) {
7201             clear_page((unsigned long)pte);
7202             pmd_val(*pmd) = _KERNPG_TABLE + __pa(pte);
7203             return pte + offset;
7204         }
7205         pmd_val(*pmd) = _KERNPG_TABLE + __pa(BAD_PAGETABLE);
7206         return NULL;
7207     }
7208     free_page((unsigned long)pte);
7209     if (pmd_bad(*pmd)) {
7210         __bad_pte_kernel(pmd);
7211         return NULL;
7212     }
7213     return (pte_t *) pmd_page(*pmd) + offset;
7214 }
7215
7216 pte_t *get_pte_slow(pmd_t *pmd, unsigned long offset)
7217 {
7218     unsigned long pte;
7219
7220     pte = (unsigned long) __get_free_page(GFP_KERNEL);
7221     if (pmd_none(*pmd)) {
7222         if (pte) {
7223             clear_page(pte);

```

```

7224         pmd_val(*pmd) = _PAGE_TABLE + __pa(pte);
7225         return (pte_t *) (pte + offset);
7226     }
7227     pmd_val(*pmd) = _PAGE_TABLE + __pa(BAD_PAGETABLE);
7228     return NULL;
7229 }
7230 free_page(pte);
7231 if (pmd_bad(*pmd)) {
7232     __bad_pte(pmd);
7233     return NULL;
7234 }
7235 return (pte_t *) (pmd_page(*pmd) + offset);
7236 }
7237
7238 int do_check_pgt_cache(int low, int high)
7239 {
7240     int freed = 0;
7241     if (pgtable_cache_size > high) {
7242         do {
7243             if (pgd_quicklist)
7244                 free_pgd_slow(get_pgd_fast()), freed++;
7245             if (pmd_quicklist)
7246                 free_pmd_slow(get_pmd_fast()), freed++;
7247             if (pte_quicklist)
7248                 free_pte_slow(get_pte_fast()), freed++;
7249         } while (pgtable_cache_size > low);
7250     }
7251     return freed;
7252 }
7253
7254 /* BAD_PAGE is the page that is used for page faults when
7255 * linux is out-of-memory. Older versions of linux just
7256 * did a do_exit(), but using this instead means there is
7257 * less risk for a process dying in kernel mode, possibly
7258 * leaving an inode unused etc..
7259 *
7260 * BAD_PAGETABLE is the accompanying page-table: it is
7261 * initialized to point to BAD_PAGE entries.
7262 *
7263 * ZERO_PAGE is a special page that is used for
7264 * zero-initialized data and COW. */
7265 pte_t * bad_pagetable(void)
7266 {
7267     extern char empty_bad_page_table[PAGE_SIZE];
7268     int d0, d1;
7269
7270     asm    volatile__("cld ; rep ; stosl"
7271                      : "=&D" (d0), "~&c" (d1)

```

```

7272         : "a" (pte_val(BAD_PAGE)),
7273         "0" ((long) empty_bad_page_table),
7274         "1" (PAGE_SIZE/4)
7275         : "memory");
7276 return (pte_t *) empty_bad_page_table;
7277 }
7278
7279 pte_t __bad_page(void)
7280 {
7281     extern char empty_bad_page[PAGE_SIZE];
7282     int d0, d1;
7283
7284     __asm__ __volatile__("cld ; rep ; stosl"
7285         : "=&D" (d0), "=&c" (d1)
7286         : "a" (0),
7287         "0" ((long) empty_bad_page),
7288         "1" (PAGE_SIZE/4)
7289         : "memory");
7290     return pte_mkdirty(mk_pte((unsigned long)empty_bad_page
7291         PAGE_SHARED));
7292 }
7293
7294 void show_mem(void)
7295 {
7296     int i, free = 0, total = 0, reserved = 0;
7297     int shared = 0, cached = 0;
7298
7299     printk("Mem-info:\n");
7300     show_free_areas();
7301     printk("Free swap:      %6dkB\n",
7302         nr_swap_pages<<(PAGE_SHIFT-10));
7303     i = max_mapnr;
7304     while (i-- > 0) {
7305         total++;
7306         if (PageReserved(mem_map+i))
7307             reserved++;
7308         else if (PageSwapCache(mem_map+i))
7309             cached++;
7310         else if (!atomic_read(&mem_map[i].count))
7311             free++;
7312         else
7313             shared += atomic_read(&mem_map[i].count) - 1;
7314     }
7315     printk("%d pages of RAM\n", total);
7316     printk("%d reserved pages\n", reserved);
7317     printk("%d pages shared\n", shared);
7318     printk("%d pages swap cached\n", cached);
7319     printk("%ld pages in page table cache\n",

```

```

7320         pgtable_cache_size);
7321     show_buffers();
7322 #ifdef CONFIG_NET
7323     show_net_buffers();
7324 #endif
7325 }
7326
7327 extern unsigned long free_area_init(unsigned long,
7328                                     unsigned long);
7329
7330 /* References to section boundaries */
7331
7332 extern char _text, _etext, _edata, __bss_start, __end;
7333 extern char __init_begin, __init_end;
7334
7335 #define X86_CR4_VME 0x0001 /* enable vm86 extensions */
7336 #define X86_CR4_PVI 0x0002 /* virt intrs flag enable */
7337 #define X86_CR4_TSD 0x0004 /* disable tm stamp at ipl 3*/
7338 #define X86_CR4_DE 0x0008 /* enable debug extensions */
7339 #define X86_CR4_PSE 0x0010 /* enable pg size extensions*/
7340 #define X86_CR4_PAE 0x0020 /* enable phys addr extnsns */
7341 #define X86_CR4_MCE 0x0040 /* Machine check enable */
7342 #define X86_CR4_PGE 0x0080 /* enable global pages */
7343 #define X86_CR4_PCE 0x0100 /* enable performance counters
7344     * at ipl 3 */
7345
7346 /* Save the cr4 feature set we're using (ie Pentium 4MB
7347 * enable and PPro Global page enable), so that any CPU's
7348 * that boot up after us can get the correct flags. */
7349 unsigned long mmu_cr4_features __initdata = 0;
7350
7351 static inline void set_in_cr4(unsigned long mask)
7352 {
7353     mmu_cr4_features |= mask;
7354     __asm__("movl %%cr4,%%eax\n\t"
7355         "orl %0,%%eax\n\t"
7356         "movl %%eax,%%cr4\n\t"
7357         : : "irg" (mask)
7358         : "ax");
7359 }
7360
7361 /* allocate page table(s) for compile-time fixed
7362 * mappings */
7363 static unsigned long __init fixmap_init(
7364     unsigned long start_mem)
7365 {
7366     pgd_t * pg_dir;
7367     unsigned int idx;

```

```

7368 unsigned long address;
7369
7370 start_mem = PAGE_ALIGN(start_mem);
7371
7372 for (idx=1; idx <= __end_of_fixed_addresses;
7373      idx += PTRS_PER_PTE)
7374 {
7375     address = __fix_to_virt(__end_of_fixed_addresses-idx);
7376     pg_dir = swapper_pg_dir + (address >> PGDIR_SHIFT);
7377     memset((void *)start_mem, 0, PAGE_SIZE);
7378     pgd_val(*pg_dir) = _PAGE_TABLE | __pa(start_mem);
7379     start_mem += PAGE_SIZE;
7380 }
7381
7382 return start_mem;
7383 }
7384
7385 static void set_pte_phys (unsigned long vaddr,
7386                          unsigned long phys)
7387 {
7388     pgprot_t prot;
7389     pte_t * pte;
7390
7391     pte = pte_offset(pmd_offset(pgd_offset_k(vaddr), vaddr)
7392                    , vaddr);
7393     prot = PAGE_KERNEL;
7394     if (boot_cpu_data.x86_capability & X86_FEATURE_PGE)
7395         pgprot_val(prot) |= _PAGE_GLOBAL;
7396     set_pte(pte, mk_pte_phys(phys, prot));
7397
7398     local_flush_tlb();
7399 }
7400
7401 void set_fixmap (enum fixed_addresses idx,
7402                unsigned long phys)
7403 {
7404     unsigned long address = __fix_to_virt(idx);
7405
7406     if (idx >= __end_of_fixed_addresses) {
7407         printk("Invalid set_fixmap\n");
7408         return;
7409     }
7410     set_pte_phys (address, phys);
7411 }
7412
7413 /* paging_init() sets up the page tables - note that the
7414 * first 4MB are already mapped by head.S.
7415 *

```

```

7416 * This routines also unmaps the page at virtual kernel
7417 * address 0, so that we can trap those pesky
7418 * NULL-reference errors in the kernel. */
7419 __initfunc(unsigned long paging_init(
7420            unsigned long start_mem, unsigned long end_mem))
7421 {
7422     pgd_t * pg_dir;
7423     pte_t * pg_table;
7424     unsigned long tmp;
7425     unsigned long address;
7426
7427     /* Physical page 0 is special; it's not touched by Linux
7428     * since BIOS and SMM (for laptops with [34]86/SL chips)
7429     * may need it. It is read and write protected to detect
7430     * null pointer references in the kernel. It may also
7431     * hold the MP configuration table when we are booting
7432     * SMP. */
7433     start_mem = PAGE_ALIGN(start_mem);
7434     address = PAGE_OFFSET;
7435     pg_dir = swapper_pg_dir;
7436     /* unmap the original low memory mappings */
7437     pgd_val(pg_dir[0]) = 0;
7438
7439     /* Map whole memory from PAGE_OFFSET */
7440     pg_dir += USER_PGD_PTRS;
7441     while (address < end_mem) {
7442         /* If we're running on a Pentium CPU, we can use the
7443         * 4MB page tables.
7444         *
7445         * The page tables we create span up to the next 4MB
7446         * virtual memory boundary, but that's OK as we won't
7447         * use that memory anyway. */
7448         if (boot_cpu_data.x86_capability & X86_FEATURE_PSE) {
7449             unsigned long __pe;
7450
7451             set_in_cr4(X86_CR4_PSE);
7452             boot_cpu_data.wp_works_ok = 1;
7453             __pe = _KERNPG_TABLE + _PAGE_4M + __pa(address);
7454             /* Make it "global" too if supported */
7455             if (boot_cpu_data.x86_capability & X86_FEATURE_PGE) {
7456                 set_in_cr4(X86_CR4_PGE);
7457                 __pe += _PAGE_GLOBAL;
7458             }
7459             pgd_val(*pg_dir) = __pe;
7460             pg_dir++;
7461             address += 4*1024*1024;
7462             continue;
7463         }

```

```

7464
7465 /* We're on a [34]86, use normal page tables.
7466 * pg_table is physical at this point */
7467 pg_table = (pte_t *) (PAGE_MASK & pgd_val(*pg_dir));
7468 if (!pg_table) {
7469     pg_table = (pte_t *) __pa(start_mem);
7470     start_mem += PAGE_SIZE;
7471 }
7472
7473 pgd_val(*pg_dir) = _PAGE_TABLE |
7474     (unsigned long) pg_table;
7475 pg_dir++;
7476
7477 /* now change pg_table to kernel virtual addresses */
7478 pg_table = (pte_t *) __va(pg_table);
7479 for (tmp = 0; tmp < PTRS_PER_PTE; tmp++, pg_table++) {
7480     pte_t pte = mk_pte(address, PAGE_KERNEL);
7481     if (address >= end_mem)
7482         pte_val(pte) = 0;
7483     set_pte(pg_table, pte);
7484     address += PAGE_SIZE;
7485 }
7486 }
7487 start_mem = fixmap_init(start_mem);
7488 #ifdef __SMP__
7489 start_mem = init_smp_mappings(start_mem);
7490 #endif
7491 local_flush_tlb();
7492
7493 return free_area_init(start_mem, end_mem);
7494 }
7495
7496 /* Test if the WP bit works in supervisor mode. It isn't
7497 * supported on 386's and also on some strange 486's
7498 * (NexGen etc.). All 586+'s are OK. The jumps before and
7499 * after the test are here to work-around some nasty CPU
7500 * bugs. */
7501 __initfunc(void test_wp_bit(void))
7502 {
7503     unsigned char tmp_reg;
7504     unsigned long old = pg0[0];
7505
7506     printk("Checking if this processor honours the WP bit "
7507           "even in supervisor mode... ");
7508     pg0[0] = pte_val(mk_pte(PAGE_OFFSET, PAGE_READONLY));
7509     local_flush_tlb();
7510     current->mm->mmap->vm_start += PAGE_SIZE;
7511     __asm__ __volatile__(

```

```

7512     "jmp 1f; 1:\n"
7513     "movb %0,%1\n"
7514     "movb %1,%0\n"
7515     "jmp 1f; 1:\n"
7516     : "=m" (*(char *) __va(0)),
7517     "=q" (tmp_reg)
7518     : /* no inputs */
7519     : "memory");
7520     pg0[0] = old;
7521     local_flush_tlb();
7522     current->mm->mmap->vm_start -= PAGE_SIZE;
7523     if (boot_cpu_data.wp_works_ok < 0) {
7524         boot_cpu_data.wp_works_ok = 0;
7525         printk("No.\n");
7526     #ifdef CONFIG_X86_WP_WORKS_OK
7527         panic("This kernel doesn't support CPU's with broken"
7528             " WP. Recompile it for a 386!");
7529     #endif
7530     } else
7531         printk(".\n");
7532 }
7533
7534 __initfunc(void mem_init(unsigned long start_mem,
7535                          unsigned long end_mem))
7536 {
7537     unsigned long start_low_mem = PAGE_SIZE;
7538     int codepages = 0;
7539     int reservedpages = 0;
7540     int datapages = 0;
7541     int initpages = 0;
7542     unsigned long tmp;
7543
7544     end_mem &= PAGE_MASK;
7545     high_memory = (void *) end_mem;
7546     max_mapnr = num_physpages - MAP_NR(end_mem);
7547
7548     /* clear the zero-page */
7549     memset(empty_zero_page, 0, PAGE_SIZE);
7550
7551     /* mark usable pages in the mem_map[] */
7552     start_low_mem = PAGE_ALIGN(start_low_mem)+PAGE_OFFSET;
7553
7554     #ifdef __SMP__
7555     /* But first pinch a few for the stack/trampoline stuff
7556      * FIXME: Don't need the extra page at 4K, but need to
7557      * fix trampoline before removing it. (see the GDT
7558      * stuff) */
7559     start_low_mem += PAGE_SIZE; /* 32bit startup code */

```

```

7560 /* AP processor stacks */
7561 start_low_mem = smp_alloc_memory(start_low_mem);
7562 #endif
7563 start_mem = PAGE_ALIGN(start_mem);
7564
7565 /* IBM messed up *AGAIN* in their thinkpad: 0xA0000 ->
7566 * 0x9F000. They seem to have done something stupid
7567 * with the floppy controller as well. */
7568 while (start_low_mem < 0x9f000+PAGE_OFFSET) {
7569     clear_bit(PG_reserved,
7570             &mem_map[MAP_NR(start_low_mem)].flags);
7571     start_low_mem += PAGE_SIZE;
7572 }
7573
7574 while (start_mem < end_mem) {
7575     clear_bit(PG_reserved,
7576             &mem_map[MAP_NR(start_mem)].flags);
7577     start_mem += PAGE_SIZE;
7578 }
7579 for (tmp = PAGE_OFFSET; tmp < end_mem;
7580     tmp += PAGE_SIZE) {
7581     if (tmp >= MAX_DMA_ADDRESS)
7582         clear_bit(PG_DMA, &mem_map[MAP_NR(tmp)].flags);
7583     if (PageReserved(mem_map+MAP_NR(tmp))) {
7584         if (tmp >= (unsigned long) &_text &&
7585             tmp < (unsigned long) &_edata) {
7586             if (tmp < (unsigned long) &_etext)
7587                 codepages++;
7588             else
7589                 datapages++;
7590         } else if (tmp >= (unsigned long) &__init_begin
7591                 && tmp < (unsigned long) &__init_end)
7592             initpages++;
7593         else if (tmp >= (unsigned long) &_bss_start
7594                 && tmp < (unsigned long) start_mem)
7595             datapages++;
7596         else
7597             reservedpages++;
7598         continue;
7599     }
7600     atomic_set(&mem_map[MAP_NR(tmp)].count, 1);
7601 #ifdef CONFIG_BLK_DEV_INITRD
7602     if (!initrd_start || (tmp < initrd_start || tmp >=
7603         initrd_end))
7604 #endif
7605         free_page(tmp);
7606 }
7607 printk("Memory: %luk/%luk available (%dk kernel code, "

```

```

7608         "%dk reserved, %dk data, %dk init)\n",
7609         (unsigned long) nr_free_pages << (PAGE_SHIFT-10),
7610         max_mapnr << (PAGE_SHIFT-10),
7611         codepages << (PAGE_SHIFT-10),
7612         reservedpages << (PAGE_SHIFT-10),
7613         datapages << (PAGE_SHIFT-10),
7614         initpages << (PAGE_SHIFT-10));
7615
7616 if (boot_cpu_data.wp_works_ok < 0)
7617     test_wp_bit();
7618 }
7619
7620 void free_initmem(void)
7621 {
7622     unsigned long addr;
7623
7624     addr = (unsigned long)&__init_begin;
7625     for (; addr < (unsigned long)&__init_end;
7626         addr += PAGE_SIZE) {
7627         mem_map[MAP_NR(addr)].flags &= ~(1 << PG_reserved);
7628         atomic_set(&mem_map[MAP_NR(addr)].count, 1);
7629         free_page(addr);
7630     }
7631     printk("Freeing unused kernel memory: %dk freed\n",
7632           (&__init_end - &__init_begin) >> 10);
7633 }
7634
7635 void si_meminfo(struct sysinfo *val)
7636 {
7637     int i;
7638
7639     i = max_mapnr;
7640     val->totalram = 0;
7641     val->sharedram = 0;
7642     val->freeram = nr_free_pages << PAGE_SHIFT;
7643     val->bufferram = buffermem;
7644     while (i-- > 0) {
7645         if (PageReserved(mem_map+i))
7646             continue;
7647         val->totalram++;
7648         if (!atomic_read(&mem_map[i].count))
7649             continue;
7650         val->sharedram += atomic_read(&mem_map[i].count) - 1;
7651     }
7652     val->totalram <<= PAGE_SHIFT;
7653     val->sharedram <<= PAGE_SHIFT;
7654     return;
7655 }

```

fs/binfmt_elf.c

```

7656 /*
7657  * linux/fs/binfmt_elf.c
7658  *
7659  * These are the functions used to load ELF format
7660  * executables as used on SVr4 machines. Information on
7661  * the format may be found in the book "UNIX SYSTEM V
7662  * RELEASE 4 Programmers Guide: Ansi C and Programming
7663  * Support Tools".
7664  *
7665  * Copyright 1993, 1994: Eric Youngdale (ericy@cais.com).
7666  */
7667
7668 #include <linux/module.h>
7669
7670 #include <linux/fs.h>
7671 #include <linux/stat.h>
7672 #include <linux/sched.h>
7673 #include <linux/mm.h>
7674 #include <linux/mman.h>
7675 #include <linux/a.out.h>
7676 #include <linux/errno.h>
7677 #include <linux/signal.h>
7678 #include <linux/binfmts.h>
7679 #include <linux/string.h>
7680 #include <linux/file.h>
7681 #include <linux/fcntl.h>
7682 #include <linux/ptrace.h>
7683 #include <linux/malloc.h>
7684 #include <linux/shm.h>
7685 #include <linux/personality.h>
7686 #include <linux/elfcore.h>
7687 #include <linux/init.h>
7688
7689 #include <asm/uaccess.h>
7690 #include <asm/pgtable.h>
7691
7692 #include <linux/config.h>
7693
7694 #define DLINFO_ITEMS 13
7695
7696 #include <linux/elf.h>
7697
7698 static int load_elf_binary(struct linux_binprm * bprm,
7699                          struct pt_regs * regs);
7700 static int load_elf_library(int fd);
7701 extern int dump_fpu (struct pt_regs *, elf_fpregset_t *);
7702 extern void dump_thread(struct pt_regs *, struct user *);
7703
7704 #ifndef elf_addr_t
7705 #define elf_addr_t unsigned long
7706 #define elf_caddr_t char *
7707 #endif
7708
7709 /* If we don't support core dumping, then supply a NULL
7710  * so we don't even try. */
7711 #ifdef USE_ELF_CORE_DUMP
7712 static int elf_core_dump(long signr,
7713                          struct pt_regs * regs);
7714 #else
7715 #define elf_core_dump NULL
7716 #endif
7717
7718 #define ELF_PAGESTART(_v) \
7719 ((_v) & ~(unsigned long)(ELF_EXEC_PAGESIZE-1))
7720 #define ELF_PAGEOFFSET(_v) \
7721 ((_v) & (ELF_EXEC_PAGESIZE-1))
7722 #define ELF_PAGEALIGN(_v) \
7723 (((_v) + ELF_EXEC_PAGESIZE - 1) & \
7724 ~(ELF_EXEC_PAGESIZE - 1))
7725
7726 static struct linux_binfmt elf_format = {
7727 #ifndef MODULE
7728     NULL, NULL,
7729     load_elf_binary, load_elf_library, elf_core_dump
7730 #else
7731     NULL, &__this_module,
7732     load_elf_binary, load_elf_library, elf_core_dump
7733 #endif
7734 };
7735
7736 static void set_brk(unsigned long start,
7737                    unsigned long end)
7738 {
7739     start = ELF_PAGEALIGN(start);
7740     end = ELF_PAGEALIGN(end);
7741     if (end <= start)
7742         return;
7743     do_mmap(NULL, start, end - start,
7744             PROT_READ | PROT_WRITE | PROT_EXEC,
7745             MAP_FIXED | MAP_PRIVATE, 0);
7746 }
7747
7748
7749 /* We need to explicitly zero any fractional pages

```

```

7750 after the data section (i.e. bss). This would
7751 contain the junk from the file that should not
7752 be in memory */
7753
7754
7755 static void padzero(unsigned long elf_bss)
7756 {
7757     unsigned long nbyte;
7758
7759     nbyte = ELF_PAGEOFFSET(elf_bss);
7760     if (nbyte) {
7761         nbyte = ELF_EXEC_PAGESIZE - nbyte;
7762         clear_user((void *) elf_bss, nbyte);
7763     }
7764 }
7765
7766 static elf_addr_t *
7767 create_elf_tables(char *p, int argc, int envc,
7768                  struct elfhdr * exec,
7769                  unsigned long load_addr,
7770                  unsigned long load_bias,
7771                  unsigned long interp_load_addr, int ibcs)
7772 {
7773     elf_caddr_t *argv;
7774     elf_caddr_t *envp;
7775     elf_addr_t *sp, *csp;
7776     char *k_platform, *u_platform;
7777     long hwcap;
7778     size_t platform_len = 0;
7779
7780     /* Get hold of platform and hardware capabilities masks
7781      * for the machine we are running on. In some cases
7782      * (Sparc), this info is impossible to get, in others
7783      * (i386) it is merely difficult. */
7784     hwcap = ELF_HWCAP;
7785     k_platform = ELF_PLATFORM;
7786
7787     if (k_platform) {
7788         platform_len = strlen(k_platform) + 1;
7789         u_platform = p + platform_len;
7790         __copy_to_user(u_platform, k_platform, platform_len);
7791     } else
7792         u_platform = .p;
7793
7794     /* Force 16 byte _final_ alignment here for generality.
7795      * Leave an extra 16 bytes free so that on the PowerPC
7796      * we can move the aux table up to start on a 16-byte
7797      * boundary. */

```

```

7798     sp = (elf_addr_t *)
7799         ((~15UL & (unsigned long)(u_platform)) - 16UL);
7800     csp = sp;
7801     csp -= ((exec ? DLINFO_ITEMS*2 : 4) +
7802            (k_platform ? 2 : 0));
7803     csp -= envc+1;
7804     csp -= argc+1;
7805     csp -= (libcs ? 3 : 1); /* argc itself */
7806     if ((unsigned long)csp & 15UL)
7807         sp -= ((unsigned long)csp & 15UL) / sizeof(*sp);
7808
7809     /* Put the ELF interpreter info on the stack */
7810 #define NEW_AUX_ENT(nr, id, val)
7811     __put_user ((id), sp+(nr*2));
7812     __put_user ((val), sp+(nr*2+1));
7813
7814     sp -= 2;
7815     NEW_AUX_ENT(0, AT_NULL, 0);
7816     if (k_platform) {
7817         sp -= 2;
7818         NEW_AUX_ENT(0, AT_PLATFORM,
7819                    (elf_addr_t)(unsigned long) u_platform);
7820     }
7821     sp -= 2;
7822     NEW_AUX_ENT(0, AT_HWCAP, hwcap);
7823
7824     if (exec) {
7825         sp -= 11*2;
7826
7827         NEW_AUX_ENT(0, AT_PHDR, load_addr + exec->e_phoff);
7828         NEW_AUX_ENT(1, AT_PHENT, sizeof (struct elf_phdr));
7829         NEW_AUX_ENT(2, AT_PHNUM, exec->e_phnum);
7830         NEW_AUX_ENT(3, AT_PAGESZ, ELF_EXEC_PAGESIZE);
7831         NEW_AUX_ENT(4, AT_BASE, interp_load_addr);
7832         NEW_AUX_ENT(5, AT_FLAGS, 0);
7833         NEW_AUX_ENT(6, AT_ENTRY, load_bias + exec->e_entry);
7834         NEW_AUX_ENT(7, AT_UID, (elf_addr_t) current->uid);
7835         NEW_AUX_ENT(8, AT_EUID, (elf_addr_t) current->euid);
7836         NEW_AUX_ENT(9, AT_GID, (elf_addr_t) current->gid);
7837         NEW_AUX_ENT(10, AT_EGID, (elf_addr_t) current->egid);
7838     }
7839 #undef NEW_AUX_ENT
7840
7841     sp -= envc+1;
7842     envp = (elf_caddr_t *) sp;
7843     sp -= argc+1;
7844     argv = (elf_caddr_t *) sp;
7845     if (!libcs) {

```

```

7846     __put_user((elf_addr_t)(unsigned long) envp,--sp);
7847     __put_user((elf_addr_t)(unsigned long) argv,--sp);
7848 }
7849
7850 __put_user((elf_addr_t)argc,--sp);
7851 current->mm->arg_start = (unsigned long) p;
7852 while (argc-->0) {
7853     __put_user((elf_caddr_t)(unsigned long)p,argv++);
7854     p += strlen_user(p);
7855 }
7856 __put_user(NULL, argv);
7857 current->mm->arg_end =
7858     current->mm->env_start = (unsigned long) p;
7859 while (envc-->0) {
7860     __put_user((elf_caddr_t)(unsigned long)p,envp++);
7861     p += strlen_user(p);
7862 }
7863 __put_user(NULL, envp);
7864 current->mm->env_end = (unsigned long) p;
7865 return sp;
7866 }
7867
7868
7869 /* This is much more generalized than the library routine
7870    read function, so we keep this separate. Technically
7871    the library read function is only provided so that we
7872    can read a.out libraries that have an ELF header */
7873
7874 static unsigned long load_elf_interp(
7875     struct elfhdr * interp_elf_ex,
7876     struct dentry * interpreter_dentry,
7877     unsigned long *interp_load_addr)
7878 {
7879     struct file * file;
7880     struct elf_phdr *elf_phdata;
7881     struct elf_phdr *epnt;
7882     unsigned long load_addr = 0;
7883     int load_addr_set = 0;
7884     unsigned long last_bss = 0, elf_bss = 0;
7885     unsigned long error = ~0UL;
7886     int elf_exec_fileno;
7887     int retval, i, size;
7888
7889     /* First of all, some simple consistency checks */
7890     if (interp_elf_ex->e_type != ET_EXEC &&
7891         interp_elf_ex->e_type != ET_DYN)
7892         goto out;
7893     if (!elf_check_arch(interp_elf_ex->e_machine))

```

```

7894     goto out;
7895     if (!interpreter_dentry->d_inode->i_op ||
7896         !interpreter_dentry->d_inode->i_op >
7897             default_file_ops->mmap)
7898         goto out;
7899
7900     /* If the size of this structure has changed, then
7901     * punt, since we will be doing the wrong thing. */
7902     if (interp_elf_ex->e_phentsize !=
7903         sizeof(struct elf_phdr))
7904         goto out;
7905
7906     /* Now read in all of the header information */
7907
7908     size = sizeof(struct elf_phdr) *interp_elf_ex->e_phnum;
7909     if (size > ELF_EXEC_PAGESIZE)
7910         goto out;
7911     elf_phdata =
7912         (struct elf_phdr *) kcalloc(size, GFP_KERNEL);
7913     if (!elf_phdata)
7914         goto out;
7915
7916     retval = read_exec(interpreter_dentry,
7917                       interp_elf_ex->e_phoff,
7918                       (char *) elf_phdata, size, 1);
7919     error = retval;
7920     if (retval < 0)
7921         goto out_free;
7922
7923     error = ~0UL;
7924     elf_exec_fileno = open_dentry(interpreter_dentry,
7925                                  O_RDONLY);
7926     if (elf_exec_fileno < 0)
7927         goto out_free;
7928     file = fget(elf_exec_fileno);
7929
7930     epnt = elf_phdata;
7931     for (i=0; i<interp_elf_ex->e_phnum; i++, epnt++) {
7932         if (epnt->p_type == PT_LOAD) {
7933             int elf_type = MAP_PRIVATE | MAP_DENYWRITE;
7934             int elf_prot = 0;
7935             unsigned long vaddr = 0;
7936             unsigned long k, map_addr;
7937
7938             if (epnt->p_flags & PF_R) elf_prot = PROT_READ;
7939             if (epnt->p_flags & PF_W) elf_prot |= PROT_WRITE;
7940             if (epnt->p_flags & PF_X) elf_prot |= PROT_EXEC;
7941             vaddr = epnt->p_vaddr;

```



```

7942     if (interp_elf_ex->e_type == ET_EXEC ||
7943         load_addr_set) {
7944         elf_type |= MAP_FIXED;
7945 #ifdef __sparc__
7946     } else {
7947         load_addr = get_unmapped_area(0, eppnt->p_filesz +
7948             ELF_PAGEOFFSET(vaddr));
7949 #endif
7950     }
7951
7952     map_addr = do_mmap(file,
7953         load_addr + ELF_PAGESTART(vaddr),
7954         eppnt->p_filesz +
7955             ELF_PAGEOFFSET(eppnt->p_vaddr),
7956         elf_prot,
7957         elf_type,
7958         eppnt->p_offset -
7959             ELF_PAGEOFFSET(eppnt->p_vaddr));
7960     if (map_addr > -1024UL) /* Real error */
7961         goto out_close;
7962
7963     if (!load_addr_set &&
7964         interp_elf_ex->e_type == ET_DYN) {
7965         load_addr = map_addr - ELF_PAGESTART(vaddr);
7966         load_addr_set = 1;
7967     }
7968
7969     /* Find the end of the file mapping for this phdr,
7970     * and keep track of the largest address we see for
7971     * this. */
7972     k = load_addr + eppnt->p_vaddr + eppnt->p_filesz;
7973     if (k > elf_bss)
7974         elf_bss = k;
7975
7976     /* Do the same thing for the memory mapping -
7977     * between elf_bss and last_bss is the bss section.
7978     */
7979     k = load_addr + eppnt->p_memsz + eppnt->p_vaddr;
7980     if (k > last_bss)
7981         last_bss = k;
7982 }
7983 }
7984
7985 /* Now use mmap to map the library into memory. */
7986
7987 /* Now fill out the bss section. First pad the last
7988 * page up to the page boundary, and then perform a
7989 * mmap to make sure that there are zero-mapped pages

```

```

7990     * up to and including the last bss page. */
7991     padzero(elf_bss);
7992     /* What we have mapped so far */
7993     elf_bss = ELF_PAGESTART(elf_bss +
7994         ELF_EXEC_PAGESIZE - 1);
7995
7996     /* Map the last of the bss segment */
7997     if (last_bss > elf_bss)
7998         do_mmap(NULL, elf_bss, last_bss - elf_bss,
7999             PROT_READ|PROT_WRITE|PROT_EXEC,
8000             MAP_FIXED|MAP_PRIVATE, 0);
8001
8002     *interp_load_addr = load_addr;
8003     error =
8004         ((unsigned long) interp_elf_ex->e_entry) + load_addr;
8005
8006 out_close:
8007     fput(file);
8008     sys_close(elf_exec_fileno);
8009 out_free:
8010     kfree(elf_phdata);
8011 out:
8012     return error;
8013 }
8014
8015 static unsigned long load_aout_interp(
8016     struct exec * interp_ex,
8017     struct dentry * interpreter_dentry)
8018 {
8019     unsigned long text_data, offset, elf_entry = ~0UL;
8020     char * addr;
8021     int retval;
8022
8023     current->mm->end_code = interp_ex->a_text;
8024     text_data = interp_ex->a_text + interp_ex->a_data;
8025     current->mm->end_data = text_data;
8026     current->mm->brk = interp_ex->a_bss + text_data;
8027
8028     switch (N_MAGIC(*interp_ex)) {
8029     case OMAGIC:
8030         offset = 32;
8031         addr = (char *) 0;
8032         break;
8033     case ZMAGIC:
8034     case QMAGIC:
8035         offset = N_TXTOFF(*interp_ex);
8036         addr = (char *) N_TXTADDR(*interp_ex);
8037         break;

```

```

8038 default:
8039     goto out;
8040 }
8041
8042 do_mmap(NULL, 0, text_data,
8043         PROT_READ|PROT_WRITE|PROT_EXEC,
8044         MAP_FIXED|MAP_PRIVATE, 0);
8045 retval = read_exec(interpreter_dentry, offset, addr,
8046                  text_data, 0);
8047 if (retval < 0)
8048     goto out;
8049 flush_icache_range((unsigned long)addr,
8050                  (unsigned long)addr + text_data);
8051
8052 do_mmap(NULL,
8053         ELF_PAGESTART(text_data + ELF_EXEC_PAGESIZE-1),
8054         interp_ex->a_bss,
8055         PROT_READ|PROT_WRITE|PROT_EXEC,
8056         MAP_FIXED|MAP_PRIVATE, 0);
8057 elf_entry = interp_ex->a_entry;
8058
8059 out:
8060     return elf_entry;
8061 }
8062
8063 /* These are the functions used to load ELF style
8064  * executables and shared libraries. There is no binary
8065  * dependent code anywhere else. */
8066
8067 #define INTERPRETER_NONE 0
8068 #define INTERPRETER_AOUT 1
8069 #define INTERPRETER_ELF 2
8070
8071
8072 static inline int
8073 do_load_elf_binary(struct linux_binprm * bprm,
8074                  struct pt_regs * regs)
8075 {
8076     struct file * file;
8077     struct dentry *interpreter_dentry = NULL;
8078     unsigned long load_addr = 0, load_bias;
8079     int load_addr_set = 0;
8080     char * elf_interpreter = NULL;
8081     unsigned int interpreter_type = INTERPRETER_NONE;
8082     unsigned char ibcs2_interpreter = 0;
8083     mm_segment_t old_fs;
8084     unsigned long error;
8085     struct elf_phdr * elf_ppnt, *elf_phdata;

```

```

8086     unsigned long elf_bss, k, elf_brk;
8087     int elf_exec_fileno;
8088     int retval, size, i;
8089     unsigned long elf_entry, interp_load_addr = 0;
8090     unsigned long start_code, end_code, end_data;
8091     struct elfhdr elf_ex;
8092     struct elfhdr interp_elf_ex;
8093     struct exec interp_ex;
8094     char passed_fileno[6];
8095
8096     /* Get the exec-header */
8097     elf_ex = *((struct elfhdr *) bprm->buf);
8098
8099     retval = -ENOEXEC;
8100     /* First of all, some simple consistency checks */
8101     if (elf_ex.e_ident[0] != 0x7f ||
8102         strcmp(&elf_ex.e_ident[1], "ELF", 3) != 0)
8103         goto out;
8104
8105     if (elf_ex.e_type != ET_EXEC &&
8106         elf_ex.e_type != ET_DYN)
8107         goto out;
8108     if (!elf_check_arch(elf_ex.e_machine))
8109         goto out;
8110 #ifdef __mips__
8111     /* IRIX binaries handled elsewhere. */
8112     if (elf_ex.e_flags & EF_MIPS_ARCH) {
8113         retval = -ENOEXEC;
8114         goto out;
8115     }
8116 #endif
8117     if (!bprm->dentry->d_inode->i_op ||
8118         !bprm->dentry->d_inode->i_op->default_file_ops ||
8119         !bprm->dentry->d_inode->i_op->default_file_ops->
8120             mmap)
8121         goto out;
8122
8123     /* Now read in all of the header information */
8124
8125     retval = -ENOMEM;
8126     size = elf_ex.e_phentsize * elf_ex.e_phnum;
8127     elf_phdata =
8128         (struct elf_phdr *) kmalloc(size, GFP_KERNEL);
8129     if (!elf_phdata)
8130         goto out;
8131
8132     retval = read_exec(bprm->dentry, elf_ex.e_phoff,
8133                      (char *) elf_phdata, size, 1);

```

```

8134     if (retval < 0)
8135         goto out_free_ph;
8136
8137     retval = open_dentry(bprm->dentry, O_RDONLY);
8138     if (retval < 0)
8139         goto out_free_ph;
8140     elf_exec_fileno = retval;
8141     file = fget(elf_exec_fileno);
8142
8143     elf_ppnt = elf_phdata;
8144     elf_bss = 0;
8145     elf_brk = 0;
8146
8147     start_code = ~0UL;
8148     end_code = 0;
8149     end_data = 0;
8150
8151     for (i = 0; i < elf_ex.e_phnum; i++) {
8152         if (elf_ppnt->p_type == PT_INTERP) {
8153             retval = -EINVAL;
8154             if (elf_interpreter)
8155                 goto out_free_interp;
8156
8157             /* This is the program interpreter used for
8158              * shared libraries - for now assume that this
8159              * is an a.out format binary
8160              */
8161
8162             retval = -ENOMEM;
8163             elf_interpreter =
8164                 (char *) kmalloc(elf_ppnt->p_filesz, GFP_KERNEL);
8165             if (!elf_interpreter)
8166                 goto out_free_file;
8167
8168             retval = read_exec(bprm->dentry,
8169                             elf_ppnt->p_offset,
8170                             elf_interpreter,
8171                             elf_ppnt->p_filesz, 1);
8172             if (retval < 0)
8173                 goto out_free_interp;
8174             /* If the program interpreter is one of these two,
8175              * then assume an iBCS2 image. Otherwise assume
8176              * a native linux image.
8177              */
8178             if (!strcmp(elf_interpreter, "/usr/lib/libc.so.1")
8179                 || !strcmp(elf_interpreter, "/usr/lib/ld.so.1"))
8180                 ibcs2_interpreter = 1;
8181 #if 0

```

```

8182         printk("Using ELF interpreter %s\n",
8183                elf_interpreter);
8184 #endif
8185         old_fs = get_fs(); /* Could probably be optimized*/
8186         set_fs(get_ds());
8187 #ifdef __sparc__
8188         if (ibcs2_interpreter) {
8189             unsigned long old_pers = current->personality;
8190
8191             current->personality = PER_SVR4;
8192             interpreter_dentry = open_namei(elf_interpreter,
8193                                             0, 0);
8194             current->personality = old_pers;
8195         } else
8196 #endif
8197             interpreter_dentry = open_namei(elf_interpreter,
8198                                             0, 0);
8199         set_fs(old_fs);
8200         retval = PTR_ERR(interpreter_dentry);
8201         if (IS_ERR(interpreter_dentry))
8202             goto out_free_interp;
8203         retval = permission(interpreter_dentry->d_inode,
8204                             MAY_EXEC);
8205         if (retval < 0)
8206             goto out_free_dentry;
8207         retval = read_exec(interpreter_dentry, 0,
8208                             bprm->buf, 128, 1);
8209         if (retval < 0)
8210             goto out_free_dentry;
8211
8212         /* Get the exec headers */
8213         interp_ex = *((struct exec *) bprm->buf);
8214         interp_elf_ex = *((struct elfhdr *) bprm->buf);
8215     }
8216     elf_ppnt++;
8217 }
8218
8219 /* Some simple consistency checks for the interpreter*/
8220 if (elf_interpreter) {
8221     interpreter_type = INTERPRETER_ELF|INTERPRETER_AOUT;
8222
8223     /* Now figure out which format our binary is */
8224     if ((N_MAGIC(interp_ex) != OMAGIC) &&
8225         (N_MAGIC(interp_ex) != ZMAGIC) &&
8226         (N_MAGIC(interp_ex) != QMAGIC))
8227         interpreter_type = INTERPRETER_ELF;
8228
8229     if (interp_elf_ex.e_ident[0] != 0x7f ||

```

```

8230     strncpy(&interp_elf_ex.e_ident[1], "ELF", 3))
8231     interpreter_type &= ~INTERPRETER_ELF;
8232
8233     retval = -ELIBBAD;
8234     if (!interpreter_type)
8235         goto out_free_dentry;
8236
8237     /* Make sure only one type was selected */
8238     if ((interpreter_type & INTERPRETER_ELF) &&
8239         interpreter_type != INTERPRETER_ELF) {
8240         printk(KERN_WARNING
8241             "ELF: Ambiguous type, using ELF\n");
8242         interpreter_type = INTERPRETER_ELF;
8243     }
8244 }
8245
8246 /* OK, we are done with that, now set up the arg stuff,
8247    and then start this sucker up */
8248
8249 if (!bprm->sh_bang) {
8250     char * passed_p;
8251
8252     if (interpreter_type == INTERPRETER_AOUT) {
8253         sprintf(passed_fileno, "%d", elf_exec_fileno);
8254         passed_p = passed_fileno;
8255
8256         if (elf_interpreter) {
8257             bprm->p = copy_strings(1, &passed_p, bprm->page,
8258                                 bprm->p, 2);
8259             bprm->argc++;
8260         }
8261     }
8262     retval = -E2BIG;
8263     if (!bprm->p)
8264         goto out_free_dentry;
8265 }
8266
8267 /* Flush all traces of the currently running exe */
8268 retval = flush_old_exec(bprm);
8269 if (retval)
8270     goto out_free_dentry;
8271
8272 /* OK. This is the point of no return */
8273 current->mm->end_data = 0;
8274 current->mm->end_code = 0;
8275 current->mm->mmap = NULL;
8276 current->flags &= ~PF_FORKNOEXEC;
8277 elf_entry = (unsigned long) elf_ex.e_entry;

```

```

8278
8279 /* Do this immediately, since STACK_TOP as used in
8280    setup_arg_pages may depend on the personality. */
8281 SET_PERSONALITY(elf_ex, ibcs2_interpreter);
8282
8283 /* Do this so that we can load the interpreter, if need
8284    be. We will change some of these later */
8285 current->mm->rss = 0;
8286 bprm->p = setup_arg_pages(bprm->p, bprm);
8287 current->mm->start_stack = bprm->p;
8288
8289 /* Try and get dynamic programs out of the way of the
8290    default mmap base, as well as whatever program they
8291    might try to exec. This is because the brk will
8292    follow the loader, and is not movable. */
8293
8294 load_bias = ELF_PAGESTART(elf_ex.e_type == ET_DYN
8295                          ? ELF_ET_DYN_BASE : 0);
8296
8297 /* Now we do a little grungy work by mmaping the ELF
8298    image into the correct location in memory. At this
8299    point, we assume that the image should be loaded at
8300    fixed address, not at a variable address. */
8301
8302 old_fs = get_fs();
8303 set_fs(get_ds());
8304 for(i = 0, elf_ppnt = elf_phdata; i < elf_ex.e_phnum;
8305     i++, elf_ppnt++) {
8306     int elf_prot = 0, elf_flags;
8307     unsigned long vaddr;
8308
8309     if (elf_ppnt->p_type != PT_LOAD)
8310         continue;
8311
8312     if (elf_ppnt->p_flags & PF_R) elf_prot |= PROT_READ;
8313     if (elf_ppnt->p_flags & PF_W) elf_prot |= PROT_WRITE;
8314     if (elf_ppnt->p_flags & PF_X) elf_prot |= PROT_EXEC;
8315
8316     elf_flags = MAP_PRIVATE|MAP_DENYWRITE|MAP_EXECUTABLE;
8317
8318     vaddr = elf_ppnt->p_vaddr;
8319     if (elf_ex.e_type == ET_EXEC || load_addr_set) {
8320         elf_flags |= MAP_FIXED;
8321     }
8322
8323     error = do_mmap(file,
8324                    ELF_PAGESTART(load_bias + vaddr),
8325                    (elf_ppnt->p_filesz +

```

```

8326     ELF_PAGEOFFSET(elf_ppnt->p_vaddr)),
8327     elf_prot, elf_flags, (elf_ppnt->p_offset -
8328     ELF_PAGEOFFSET(elf_ppnt->p_vaddr)));
8329
8330     if (!load_addr_set) {
8331         load_addr_set = 1;
8332         load_addr =
8333             (elf_ppnt->p_vaddr - elf_ppnt->p_offset);
8334         if (elf_ex.e_type == ET_DYN) {
8335             load_bias += error -
8336                 ELF_PAGESTART(load_bias + vaddr);
8337             load_addr += error;
8338         }
8339     }
8340     k = elf_ppnt->p_vaddr;
8341     if (k < start_code) start_code = k;
8342     k = elf_ppnt->p_vaddr + elf_ppnt->p_filesz;
8343     if (k > elf_bss)
8344         elf_bss = k;
8345     if ((elf_ppnt->p_flags & PF_X) && end_code < k)
8346         end_code = k;
8347     if (end_data < k)
8348         end_data = k;
8349     k = elf_ppnt->p_vaddr + elf_ppnt->p_memsz;
8350     if (k > elf_brk)
8351         elf_brk = k;
8352 }
8353 set_fs(old_fs);
8354 fput(file); /* all done with the file */
8355
8356 elf_entry += load_bias;
8357 elf_bss += load_bias;
8358 elf_brk += load_bias;
8359 start_code += load_bias;
8360 end_code += load_bias;
8361 end_data += load_bias;
8362
8363 if (elf_interpreter) {
8364     if (interpreter_type == INTERPRETER_AOUT)
8365         elf_entry = load_aout_interp(&interp_ex,
8366             interpreter_dentry);
8367     else
8368         elf_entry = load_elf_interp(&interp_elf_ex,
8369             interpreter_dentry,
8370             &interp_load_addr);
8371
8372     dput(interpreter_dentry);
8373     kfree(elf_interpreter);

```

```

8374
8375     if (elf_entry == ~0UL) {
8376         printk(KERN_ERR "Unable to load interpreter\n");
8377         kfree(elf_phdata);
8378         send_sig(SIGSEGV, current, 0);
8379         return 0;
8380     }
8381 }
8382
8383 kfree(elf_phdata);
8384
8385 if (interpreter_type != INTERPRETER_AOUT)
8386     sys_close(elf_exec_fileno);
8387
8388 if (current->exec_domain &&
8389     current->exec_domain->module)
8390     __MOD_DEC_USE_COUNT(current->exec_domain->module);
8391 if (current->binfmt && current->binfmt->module)
8392     __MOD_DEC_USE_COUNT(current->binfmt->module);
8393 current->exec_domain =
8394     lookup_exec_domain(current->personality);
8395 current->binfmt = &elf_format;
8396 if (current->exec_domain &&
8397     current->exec_domain->module)
8398     __MOD_INC_USE_COUNT(current->exec_domain->module);
8399 if (current->binfmt && current->binfmt->module)
8400     __MOD_INC_USE_COUNT(current->binfmt->module);
8401
8402 #ifndef VM_STACK_FLAGS
8403     current->executable = dget(bprm->dentry);
8404 #endif
8405 compute_creds(bprm);
8406 current->flags &= ~PF_FORKNOEXEC;
8407 bprm->p = (unsigned long)
8408     create_elf_tables((char *)bprm->p,
8409         bprm->argc,
8410         bprm->envc,
8411         (interpreter_type == INTERPRETER_ELF
8412         ? &elf_ex : NULL),
8413         load_addr, load_bias,
8414         interp_load_addr,
8415         (interpreter_type == INTERPRETER_AOUT ? 0 : 1));
8416 /* N.B. passed_fileno might not be initialized? */
8417 if (interpreter_type == INTERPRETER_AOUT)
8418     current->mm->arg_start += strlen(passed_fileno) + 1;
8419 current->mm->start_brk = current->mm->brk = elf_brk;
8420 current->mm->end_code = end_code;
8421 current->mm->start_code = start_code;

```

```

8422 current->mm->end_data = end_data;
8423 current->mm->start_stack = bprm->p;
8424
8425 /* Calling set_brk effectively mmmaps the pages that we
8426  * need for the bss and break sections */
8427 set_brk(elf_bss, elf_brk);
8428
8429 padzero(elf_bss);
8430
8431 #if 0
8432 printk("(start_brk) %x\n" , current->mm->start_brk);
8433 printk("(end_code) %x\n" , current->mm->end_code);
8434 printk("(start_code) %x\n" , current->mm->start_code);
8435 printk("(end_data) %x\n" , current->mm->end_data);
8436 printk("(start_stack) %x\n", current->mm->start_stack);
8437 printk("(brk) %x\n" , current->mm->brk);
8438 #endif
8439
8440 if ( current->personality == PER_SVR4 )
8441 {
8442     /* Why this, you ask??? Well SVr4 maps page 0 as
8443     read-only, and some applications "depend" upon
8444     this behavior. Since we do not have the power to
8445     recompile these, we emulate the SVr4 behavior.
8446     Sigh. */
8447     /* N.B. Shouldn't the size here be PAGE_SIZE?? */
8448     error = do_mmap(NULL, 0, 4096, PROT_READ | PROT_EXEC,
8449     MAP_FIXED | MAP_PRIVATE, 0);
8450 }
8451
8452 #ifndef ELF_PLAT_INIT
8453 /* The ABI may specify that certain registers be set up
8454  * in special ways (on i386 %edx is the address of a
8455  * DT_FINI function, for example. This macro performs
8456  * whatever initialization to the regs structure is
8457  * required. */
8458 ELF_PLAT_INIT(regs);
8459 #endif
8460
8461 start_thread(regs, elf_entry, bprm->p);
8462 if (current->flags & PF_PTRACED)
8463     send_sig(SIGTRAP, current, 0);
8464 retval = 0;
8465 out:
8466 return retval;
8467
8468 /* error cleanup */
8469 out_free_dentry:

```

```

8470 dput(interpreter_dentry);
8471 out_free_interp:
8472 if (elf_interpreter)
8473     kfree(elf_interpreter);
8474 out_free_file:
8475 fput(file);
8476 sys_close(elf_exec_fileno);
8477 out_free_ph:
8478 kfree(elf_phdata);
8479 goto out;
8480 }
8481
8482 static int
8483 load_elf_binary(struct linux_binprm * bprm,
8484                struct pt_regs * regs)
8485 {
8486     int retval;
8487
8488     MOD_INC_USE_COUNT;
8489     retval = do_load_elf_binary(bprm, regs);
8490     MOD_DEC_USE_COUNT;
8491     return retval;
8492 }
8493
8494 /* This is really simpleminded and specialized - we are
8495    loading an a.out library that is given an ELF
8496    header. */
8497
8498 static inline int
8499 do_load_elf_library(int fd)
8500 {
8501     struct file * file;
8502     struct dentry * dentry;
8503     struct inode * inode;
8504     struct elf_phdr *elf_phdata;
8505     unsigned long elf_bss = 0, bss, len, k;
8506     int retval, error, i, j;
8507     struct elfhdr elf_ex;
8508     loff_t offset = 0;
8509
8510     error = -EACCES;
8511     file = fget(fd);
8512     if (!file || !file->f_op)
8513         goto out;
8514     dentry = file->f_dentry;
8515     inode = dentry->d_inode;
8516
8517     /* seek to the beginning of the file */

```

```

8518 error = -ENOEXEC;
8519
8520 /* N.B. save current DS?? */
8521 set_fs(KERNEL_DS);
8522 retval = file->f_op->read(file, (char *) &elf_ex,
8523                          sizeof(elf_ex), &offset);
8524 set_fs(USER_DS);
8525 if (retval != sizeof(elf_ex))
8526     goto out_putf;
8527
8528 if (elf_ex.e_ident[0] != 0x7f ||
8529     strcmp(&elf_ex.e_ident[1], "ELF", 3) != 0)
8530     goto out_putf;
8531
8532 /* First of all, some simple consistency checks */
8533 if (elf_ex.e_type != ET_EXEC || elf_ex.e_phnum > 2 ||
8534     !elf_check_arch(elf_ex.e_machine) ||
8535     (!inode->i_op ||
8536     !inode->i_op->default_file_ops->mmap))
8537     goto out_putf;
8538
8539 /* Now read in all of the header information */
8540
8541 j = sizeof(struct elf_phdr) * elf_ex.e_phnum;
8542 if (j > ELF_EXEC_PAGESIZE)
8543     goto out_putf;
8544
8545 error = -ENOMEM;
8546 elf_phdata = (struct elf_phdr *) kmalloc(j, GFP_KERNEL);
8547 if (!elf_phdata)
8548     goto out_putf;
8549
8550 /* N.B. check for error return?? */
8551 retval = read_exec(dentry, elf_ex.e_phoff,
8552                  (char *) elf_phdata,
8553                  sizeof(struct elf_phdr) * elf_ex.e_phnum, 1);
8554
8555 error = -ENOEXEC;
8556 for (j = 0, i = 0; i < elf_ex.e_phnum; i++)
8557     if ((elf_phdata + i)->p_type == PT_LOAD) j++;
8558 if (j != 1)
8559     goto out_free_ph;
8560
8561 while (elf_phdata->p_type != PT_LOAD) elf_phdata++;
8562
8563 /* Now use mmap to map the library into memory. */
8564 error = do_mmap(file,
8565                ELF_PAGESTART(elf_phdata->p_vaddr),

```

```

8566                (elf_phdata->p_filesz +
8567                 ELF_PAGEOFFSET(elf_phdata->p_vaddr)),
8568                PROT_READ | PROT_WRITE | PROT_EXEC,
8569                MAP_FIXED | MAP_PRIVATE | MAP_DENYWRITE,
8570                (elf_phdata->p_offset -
8571                 ELF_PAGEOFFSET(elf_phdata->p_vaddr)));
8572 if (error != ELF_PAGESTART(elf_phdata->p_vaddr))
8573     goto out_free_ph;
8574
8575 k = elf_phdata->p_vaddr + elf_phdata->p_filesz;
8576 if (k > elf_bss)
8577     elf_bss = k;
8578 padzero(elf_bss);
8579
8580 len = ELF_PAGESTART(elf_phdata->p_filesz +
8581                    elf_phdata->p_vaddr +
8582                    ELF_EXEC_PAGESIZE - 1);
8583 bss = elf_phdata->p_memsz + elf_phdata->p_vaddr;
8584 if (bss > len)
8585     do_mmap(NULL, len, bss - len,
8586            PROT_READ|PROT_WRITE|PROT_EXEC,
8587            MAP_FIXED|MAP_PRIVATE, 0);
8588 error = 0;
8589
8590 out_free_ph:
8591 kfree(elf_phdata);
8592 out_putf:
8593 fput(file);
8594 out:
8595 return error;
8596 }
8597
8598 static int load_elf_library(int fd)
8599 {
8600     int retval;
8601
8602     MOD_INC_USE_COUNT;
8603     retval = do_load_elf_library(fd);
8604     MOD_DEC_USE_COUNT;
8605     return retval;
8606 }
8607
8608 /* Note that some platforms still use traditional core
8609 * dumps and not the ELF core dump. Each platform can
8610 * select it as appropriate. */
8611 #ifdef USE_ELF_CORE_DUMP
8612
8613 /* ELF core dumper

```

```

8614 *
8615 * Modelled on fs/exec.c:aout_core_dump()
8616 * Jeremy Fitzhardinge <jeremy@sw.oz.au>
8617 */
8618 /* These are the only things you should do on a
8619 * core-file: use only these functions to write out all
8620 * the necessary info. */
8621 static int dump_write(struct file *file,
8622                     const void *addr, int nr)
8623 {
8624     return file->f_op->write(file, addr, nr, &file->f_pos)
8625         == nr;
8626 }
8627
8628 static int dump_seek(struct file *file, off_t off)
8629 {
8630     if (file->f_op->llseek) {
8631         if (file->f_op->llseek(file, off, 0) != off)
8632             return 0;
8633     } else
8634         file->f_pos = off;
8635     return 1;
8636 }
8637
8638 /* Decide whether a segment is worth dumping; default is
8639 * yes to be sure (missing info is worse than too much;
8640 * etc). Personally I'd include everything, and use the
8641 * coredump limit...
8642 *
8643 * I think we should skip something. But I am not sure
8644 * how. H.J. */
8645 static inline int maydump(struct vm_area_struct *vma)
8646 {
8647     if (!(vma->vm_flags & (VM_READ|VM_WRITE|VM_EXEC)))
8648         return 0;
8649
8650     /* Do not dump I/O mapped devices! -DaveM */
8651     if (vma->vm_flags & VM_IO)
8652         return 0;
8653     #if 1
8654     if (vma->vm_flags & (VM_WRITE|VM_GROWSUP|VM_GROWSDOWN))
8655         return 1;
8656     if (vma->vm_flags & (VM_READ|VM_EXEC|VM_EXECUTABLE|
8657                         VM_SHARED))
8658         return 0;
8659     #endif
8660     return 1;
8661 }

```

```

8662
8663 #define roundup(x, y) (((x)+(y)-1)/(y))*(y)
8664
8665 /* An ELF note in memory */
8666 struct memelfnote
8667 {
8668     const char *name;
8669     int type;
8670     unsigned int datasz;
8671     void *data;
8672 };
8673
8674 static int notesize(struct memelfnote *en)
8675 {
8676     int sz;
8677
8678     sz = sizeof(struct elf_note);
8679     sz += roundup(strlen(en->name), 4);
8680     sz += roundup(en->datasz, 4);
8681
8682     return sz;
8683 }
8684
8685 /* #define DEBUG */
8686
8687 #ifdef DEBUG
8688 static void dump_regs(const char *str, elf_greg_t *r)
8689 {
8690     int i;
8691     static const char *regs[] = {
8692         "ebx", "ecx", "edx", "esi", "edi", "ebp",
8693         "eax", "ds", "es", "fs", "gs",
8694         "orig_eax", "eip", "cs",
8695         "efl", "uesp", "ss"};
8696     printk("Registers: %s\n", str);
8697
8698     for(i = 0; i < ELF_NGREG; i++)
8699     {
8700         unsigned long val = r[i];
8701         printk("  %2d %-5s-%08lx %lu\n",
8702             i, regs[i], val, val);
8703     }
8704 }
8705 #endif
8706
8707 #define DUMP_WRITE(addr, nr) \
8708     do { if (!dump_write(file, (addr), (nr))) return 0; } \
8709     while(0)

```



```

8710 #define DUMP_SEEK(off)
8711 do { if (!dump_seek(file, (off))) return 0; } while(0)
8712
8713 static int writenote(struct memelfnote *men,
8714                    struct file *file)
8715 {
8716     struct elf_note en;
8717
8718     en.n_namesz = strlen(men->name);
8719     en.n_descsz = men->datasz;
8720     en.n_type = men->type;
8721
8722     DUMP_WRITE(&en, sizeof(en));
8723     DUMP_WRITE(men->name, en.n_namesz);
8724     /* XXX - cast from long long to long to avoid need for
8725      * libgcc.a */
8726     /* XXX */
8727     DUMP_SEEK(roundup((unsigned long)file->f_pos, 4));
8728     DUMP_WRITE(men->data, men->datasz);
8729     /* XXX */
8730     DUMP_SEEK(roundup((unsigned long)file->f_pos, 4));
8731
8732     return 1;
8733 }
8734 #undef DUMP_WRITE
8735 #undef DUMP_SEEK
8736
8737 #define DUMP_WRITE(addr, nr)
8738     if (!dump_write(&file, (addr), (nr)))
8739         goto close_coredump;
8740 #define DUMP_SEEK(off)
8741     if (!dump_seek(&file, (off)))
8742         goto close_coredump;
8743 /* Actual dumper
8744 *
8745 * This is a two-pass process; first we find the offsets
8746 * of the bits, and then they are actually written out.
8747 * If we run out of core limit we just truncate. */
8748 static int elf_core_dump(long signr,
8749                        struct pt_regs * regs)
8750 {
8751     int has_dumped = 0;
8752     struct file file;
8753     struct dentry *dentry;
8754     struct inode *inode;
8755     mm_segment_t fs;
8756     char corefile[6+sizeof(current->comm)];
8757     int segs;

```

```

8758     int i;
8759     size_t size;
8760     struct vm_area_struct *vma;
8761     struct elfhdr elf;
8762     off_t offset = 0, dataoff;
8763     unsigned long limit =
8764         current->rlim[RLIMIT_CORE].rlim_cur;
8765     int numnote = 4;
8766     struct memelfnote notes[4];
8767     struct elf_prstatus prstatus; /* NT_PRSTATUS */
8768     elf_fpregset_t fpu; /* NT_PRFPREG */
8769     struct elf_prpsinfo psinfo; /* NT_PRPSINFO */
8770
8771     if (!current->dumpable ||
8772         limit < ELF_EXEC_PAGESIZE ||
8773         atomic_read(&current->mm->count) != 1)
8774         return 0;
8775     current->dumpable = 0;
8776
8777 #ifndef CONFIG_BINFMT_ELF
8778     MOD_INC_USE_COUNT;
8779 #endif
8780
8781     /* Count what's needed to dump, up to the limit of
8782      * coredump size */
8783     segs = 0;
8784     size = 0;
8785     for(vma = current->mm->mmap; vma != NULL;
8786         vma = vma->vm_next) {
8787         if (maydump(vma))
8788             {
8789             unsigned long sz = vma->vm_end-vma->vm_start;
8790
8791             if (size+sz >= limit)
8792                 break;
8793             else
8794                 size += sz;
8795             }
8796
8797         segs++;
8798     }
8799 #ifdef DEBUG
8800     printk("elf core_dump: %d segs taking %d bytes\n",
8801           segs, size);
8802 #endif
8803
8804     /* Set up header */
8805     memcpy(elf.e_ident, ELFMAG, SELFMAG);

```

```

8806 elf.e_ident[EI_CLASS] = ELF_CLASS;
8807 elf.e_ident[EI_DATA] = ELF_DATA;
8808 elf.e_ident[EI_VERSION] = EV_CURRENT;
8809 memset(elf.e_ident+EI_PAD, 0, EI_NIDENT-EI_PAD);
8810
8811 elf.e_type = ET_CORE;
8812 elf.e_machine = ELF_ARCH;
8813 elf.e_version = EV_CURRENT;
8814 elf.e_entry = 0;
8815 elf.e_phoff = sizeof(elf);
8816 elf.e_shoff = 0;
8817 elf.e_flags = 0;
8818 elf.e_ehsize = sizeof(elf);
8819 elf.e_phentsize = sizeof(struct elf_phdr);
8820 elf.e_phnum = segs+1; /* Include notes */
8821 elf.e_shentsize = 0;
8822 elf.e_shnum = 0;
8823 elf.e_shstrndx = 0;
8824
8825 fs = get_fs();
8826 set_fs(KERNEL_DS);
8827 memcpy(corefile, "core.", 5);
8828 #if 0
8829 memcpy(corefile+5, current->comm, sizeof(current->comm));
8830 #else
8831 corefile[4] = '\0';
8832 #endif
8833 dentry = open_namei(corefile,
8834 O_CREAT | 2 | O_TRUNC | O_NOFOLLOW, 0600);
8835 if (IS_ERR(dentry)) {
8836 dentry = NULL;
8837 goto end_coredump;
8838 }
8839 inode = dentry->d_inode;
8840
8841 if(inode->i_nlink > 1)
8842 goto end_coredump; /* multiple links- don't dump */
8843
8844 if (!IS_REG(inode->i_mode))
8845 goto end_coredump;
8846 if (!inode->i_op || !inode->i_op->default_file_ops)
8847 goto end_coredump;
8848 if (init_private_file(&file, dentry, 3))
8849 goto end_coredump;
8850 if (!file.f_op->write)
8851 goto close_coredump;
8852 has_dumped = 1;
8853 current->flags |= PF_DUMPCORE;
8854
8855 DUMP_WRITE(&elf, sizeof(elf));
8856 offset += sizeof(elf); /* Elf header */
8857 /* Program headers */
8858 offset += (segs+1) * sizeof(struct elf_phdr);
8859
8860 /* Set up the notes in similar form to SVR4 core dumps
8861 * made with info from their /proc. */
8862 memset(&psinfo, 0, sizeof(psinfo));
8863 memset(&prstatus, 0, sizeof(prstatus));
8864
8865 notes[0].name = "CORE";
8866 notes[0].type = NT_PRSTATUS;
8867 notes[0].datasz = sizeof(prstatus);
8868 notes[0].data = &prstatus;
8869 prstatus.pr_info.si_signo = prstatus.pr_cursig = signr;
8870 prstatus.pr_sigpend = current->signal.sig[0];
8871 prstatus.pr_sighold = current->blocked.sig[0];
8872 psinfo.pr_pid = prstatus.pr_pid = current->pid;
8873 psinfo.pr_ppid = prstatus.pr_ppid =
8874 current->p_pptr->pid;
8875 psinfo.pr_pgrp = prstatus.pr_pgrp = current->pgrp;
8876 psinfo.pr_sid = prstatus.pr_sid = current->session;
8877 prstatus.pr_utime.tv_sec =
8878 CT_TO_SECS(current->times.tms_utime);
8879 prstatus.pr_utime.tv_usec =
8880 CT_TO_USECS(current->times.tms_utime);
8881 prstatus.pr_stime.tv_sec =
8882 CT_TO_SECS(current->times.tms_stime);
8883 prstatus.pr_stime.tv_usec =
8884 CT_TO_USECS(current->times.tms_stime);
8885 prstatus.pr_cutime.tv_sec =
8886 CT_TO_SECS(current->times.tms_cutime);
8887 prstatus.pr_cutime.tv_usec =
8888 CT_TO_USECS(current->times.tms_cutime);
8889 prstatus.pr_cstime.tv_sec =
8890 CT_TO_SECS(current->times.tms_cstime);
8891 prstatus.pr_cstime.tv_usec =
8892 CT_TO_USECS(current->times.tms_cstime);
8893
8894 /* This transfers the registers from regs into the
8895 * standard coredump arrangement, whatever that is. */
8896 #ifdef ELF_CORE_COPY_REGS
8897 ELF_CORE_COPY_REGS(prstatus.pr_reg, regs)
8898 #else
8899 if (sizeof(elf_gregset_t) != sizeof(struct pt_regs))
8900 {
8901 printk("sizeof(elf_gregset_t) (%ld) != "

```

```

8902     "sizeof(struct pt_regs) (%ld)\n",
8903     (long)sizeof(elf_gregset_t),
8904     (long)sizeof(struct pt_regs));
8905 }
8906 else
8907     *(struct pt_regs *)&prstatus.pr_reg = *regs;
8908 #endif
8909
8910 #ifdef DEBUG
8911     dump_regs("Passed in regs", (elf_greg_t *)regs);
8912     dump_regs("prstatus regs",
8913             (elf_greg_t *)&prstatus.pr_reg);
8914 #endif
8915
8916     notes[1].name = "CORE";
8917     notes[1].type = NT_PRPSINFO;
8918     notes[1].datasz = sizeof(psinfo);
8919     notes[1].data = &psinfo;
8920     i = current->state ? ffz(~current->state) + 1 : 0;
8921     psinfo.pr_state = i;
8922     psinfo.pr_sname = (i < 0 || i > 5) ? '.' : "RSDZTD"[i];
8923     psinfo.pr_zomb = psinfo.pr_sname == 'Z';
8924     psinfo.pr_nice = current->priority-15;
8925     psinfo.pr_flag = current->flags;
8926     psinfo.pr_uid = current->uid;
8927     psinfo.pr_gid = current->gid;
8928     {
8929         int i, len;
8930
8931         set_fs(fs);
8932
8933         len = current->mm->arg_end - current->mm->arg_start;
8934         if (len >= ELF_PRARGSZ)
8935             len = ELF_PRARGSZ-1;
8936         copy_from_user(&psinfo.pr_psargs,
8937             (const char *)current->mm->arg_start, len);
8938         for(i = 0; i < len; i++)
8939             if (psinfo.pr_psargs[i] == 0)
8940                 psinfo.pr_psargs[i] = ' ';
8941         psinfo.pr_psargs[len] = 0;
8942
8943         set_fs(KERNEL_DS);
8944     }
8945     strncpy(psinfo.pr_fname, current->comm,
8946             sizeof(psinfo.pr_fname));
8947
8948     notes[2].name = "CORE";
8949     notes[2].type = NT_TASKSTRUCT;

```

```

8950     notes[2].datasz = sizeof(*current);
8951     notes[2].data = current;
8952
8953     /* Try to dump the FPU. */
8954     prstatus.pr_fpvalid = dump_fpu (regs, &fpu);
8955     if (!prstatus.pr_fpvalid)
8956     {
8957         numnote--;
8958     }
8959     else
8960     {
8961         notes[3].name = "CORE";
8962         notes[3].type = NT_PRFPREG;
8963         notes[3].datasz = sizeof(fpu);
8964         notes[3].data = &fpu;
8965     }
8966
8967     /* Write notes phdr entry */
8968     {
8969         struct elf_phdr phdr;
8970         int sz = 0;
8971
8972         for(i = 0; i < numnote; i++)
8973             sz += notesize(&notes[i]);
8974
8975         phdr.p_type = PT_NOTE;
8976         phdr.p_offset = offset;
8977         phdr.p_vaddr = 0;
8978         phdr.p_paddr = 0;
8979         phdr.p_filesz = sz;
8980         phdr.p_memsz = 0;
8981         phdr.p_flags = 0;
8982         phdr.p_align = 0;
8983
8984         offset += phdr.p_filesz;
8985         DUMP_WRITE(&phdr, sizeof(phdr));
8986     }
8987
8988     /* Page-align dumped data */
8989     dataoff = offset = roundup(offset, ELF_EXEC_PAGESIZE);
8990
8991     /* Write program headers for segments dump */
8992     for(vma = current->mm->mmap, i = 0;
8993         i < segs && vma != NULL; vma = vma->vm.next) {
8994         struct elf_phdr phdr;
8995         size_t sz;
8996
8997         i++;

```

```

8998
8999     sz = vma->vm_end - vma->vm_start;
9000
9001     phdr.p_type = PT_LOAD;
9002     phdr.p_offset = offset;
9003     phdr.p_vaddr = vma->vm_start;
9004     phdr.p_paddr = 0;
9005     phdr.p_filesz = maydump(vma) ? sz : 0;
9006     phdr.p_memsz = sz;
9007     offset += phdr.p_filesz;
9008     phdr.p_flags = vma->vm_flags & VM_READ ? PF_R : 0;
9009     if (vma->vm_flags & VM_WRITE) phdr.p_flags |= PF_W;
9010     if (vma->vm_flags & VM_EXEC) phdr.p_flags |= PF_X;
9011     phdr.p_align = ELF_EXEC_PAGESIZE;
9012
9013     DUMP_WRITE(&phdr, sizeof(phdr));
9014 }
9015
9016 for(i = 0; i < numnote; i++)
9017     if (!writenote(&notes[i], &file))
9018         goto close_coredump;
9019
9020 set_fs(fs);
9021
9022 DUMP_SEEK(dataoff);
9023
9024 for(i = 0, vma = current->mm->mmap;
9025     i < segs && vma != NULL;
9026     vma = vma->vm_next) {
9027     unsigned long addr = vma->vm_start;
9028     unsigned long len = vma->vm_end - vma->vm_start;
9029
9030     i++;
9031     if (!maydump(vma))
9032         continue;
9033 #ifdef DEBUG
9034     printk("elf_core_dump: writing %08lx %1x\n",
9035           addr, len);
9036 #endif
9037     DUMP_WRITE((void *)addr, len);
9038 }
9039
9040 if ((off_t) file.f_pos != offset) {
9041     /* Sanity check */
9042     printk("elf_core_dump: file.f_pos (%ld) != "
9043           "offset (%ld)\n",
9044           (off_t) file.f_pos, offset);
9045 }

```

```

9046 close_coredump:
9047     if (file.f_op->release)
9048         file.f_op->release(inode,&file);
9049
9050
9051 end_coredump:
9052     set_fs(fs);
9053     dput(dentry);
9054 #ifndef CONFIG_BINFMT_ELF
9055     MOD_DEC_USE_COUNT;
9056 #endif
9057     return has_dumped;
9058 }
9059 #endif /* USE_ELF_CORE_DUMP */
9060
9061 int __init init_elf_binfmt(void)
9062 {
9063     return register_binfmt(&elf_format);
9064 }
9065
9066 #ifdef MODULE
9067
9068 int init_module(void)
9069 {
9070     /* Install the COFF, ELF and XOUT loaders. N.B. We
9071      * *rely* on the table being the right size with the
9072      * right number of free slots... */
9073     return init_elf_binfmt();
9074 }
9075
9076
9077 void cleanup_module( void)
9078 {
9079     /* Remove the COFF and ELF loaders. */
9080     unregister_binfmt(&elf_format);
9081 }
9082 #endif

```

fs/binfmt_java.c

```

9083 /*
9084  * linux/fs/binfmt_java.c
9085  *
9086  * Copyright (C) 1996 Brian A. Lantz
9087  * derived from binfmt_script.c
9088  *
9089  * Simplified and modified to support binary java
9090  * interpreters by Tom May <ftom@netcom.com>.

```

```

9091 */
9092
9093 #include <linux/module.h>
9094 #include <linux/string.h>
9095 #include <linux/stat.h>
9096 #include <linux/malloc.h>
9097 #include <linux/binfmts.h>
9098 #include <linux/init.h>
9099
9100 #define _PATH_JAVA      "/usr/bin/java"
9101 #define _PATH_APPLET   "/usr/bin/appletviewer"
9102
9103 /* These paths can be modified with sysctl(). */
9104
9105 char binfmt_java_interpreter[65] = _PATH_JAVA;
9106 char binfmt_java_appletviewer[65] = _PATH_APPLET;
9107
9108 static int do_load_java(struct linux_binprm *bprm,
9109                       struct pt_regs *regs)
9110 {
9111     char *i_name;
9112     int len;
9113     int retval;
9114     struct dentry *dentry;
9115     unsigned char *ucp = (unsigned char *) bprm->buf;
9116
9117     if ((ucp[0] != 0xca) || (ucp[1] != 0xfe) ||
9118         (ucp[2] != 0xba) || (ucp[3] != 0xbe))
9119         return -ENOEXEC;
9120
9121     /* Fail if we're called recursively, e.g., the Java
9122      * interpreter is a java binary. */
9123     if (bprm->java)
9124         return -ENOEXEC;
9125
9126     bprm->java = 1;
9127
9128     dput(bprm->dentry);
9129     bprm->dentry = NULL;
9130
9131     /* Set args: [0] the name of the java interpreter
9132      *             [1] name of java class to execute, which
9133      *             is the filename without the path and
9134      *             without trailing ".class". Note that the
9135      *             interpreter will use its own way to found
9136      *             the class file (typically using
9137      *             environment variable CLASSPATH), and may
9138      *             in fact execute a different file from the

```

```

9139      *             one we want.
9140      *
9141      * This is done in reverse order, because of how the
9142      * user environment and arguments are stored. */
9143     remove_arg_zero(bprm);
9144     len = strlen(bprm->filename);
9145     if (len >= 6 &&
9146         !strcmp(bprm->filename + len - 6, ".class"))
9147         bprm->filename[len - 6] = 0;
9148     if ((i_name = strrchr(bprm->filename, '/')) != NULL)
9149         i_name++;
9150     else
9151         i_name = bprm->filename;
9152     bprm->p = copy_strings(1, &i_name, bprm->page,
9153                          bprm->p, 2);
9154     bprm->argc++;
9155
9156     i_name = binfmt_java_interpreter;
9157     bprm->p = copy_strings(1, &i_name, bprm->page,
9158                          bprm->p, 2);
9159     bprm->argc++;
9160
9161     if (!bprm->p)
9162         return -E2BIG;
9163     /* OK, now restart the process with the interpreter's
9164      * dentry. */
9165     bprm->filename = binfmt_java_interpreter;
9166     dentry = open_namei(binfmt_java_interpreter, 0, 0);
9167     retval = PTR_ERR(dentry);
9168     if (IS_ERR(dentry))
9169         return retval;
9170
9171     bprm->dentry = dentry;
9172     retval = prepare_binprm(bprm);
9173     if (retval < 0)
9174         return retval;
9175
9176     return search_binary_handler(bprm, regs);
9177 }
9178
9179 static int do_load_applet(struct linux_binprm *bprm,
9180                          struct pt_regs *regs)
9181 {
9182     char *i_name;
9183     struct dentry *dentry;
9184     int retval;
9185
9186     if (strncmp(bprm->buf, "<!--applet", 10))

```

```

9187     return -ENOEXEC;
9188
9189     dput(bprm->dentry);
9190     bprm->dentry = NULL;
9191
9192     /* Set args: [0] the name of the appletviewer
9193      *           [1] filename of html file
9194      *
9195      * This is done in reverse order, because of how the
9196      * user environment and arguments are stored. */
9197     remove_arg_zero(bprm);
9198     i_name = bprm->filename;
9199     bprm->p = copy_strings(1, &i_name, bprm->page,
9200                          bprm->p, 2);
9201     bprm->argc++;
9202
9203     i_name = binfmt_java_appletviewer;
9204     bprm->p = copy_strings(1, &i_name, bprm->page,
9205                          bprm->p, 2);
9206     bprm->argc++;
9207
9208     if (!bprm->p)
9209         return -E2BIG;
9210     /* OK, now restart the process with the interpreter's
9211      * dentry. */
9212     bprm->filename = binfmt_java_appletviewer;
9213     dentry = open_namei(binfmt_java_appletviewer, 0, 0);
9214     retval = PTR_ERR(dentry);
9215     if (IS_ERR(dentry))
9216         return retval;
9217
9218     bprm->dentry = dentry;
9219     retval = prepare_binprm(bprm);
9220     if (retval < 0)
9221         return retval;
9222
9223     return search_binary_handler(bprm, regs);
9224 }
9225
9226 static int load_java(struct linux_binprm *bprm,
9227                    struct pt_regs *regs)
9228 {
9229     int retval;
9230     MOD_INC_USE_COUNT;
9231     retval = do_load_java(bprm, regs);
9232     MOD_DEC_USE_COUNT;
9233     return retval;
9234 }

```

```

9235
9236 static struct linux_binfmt java_format = {
9237     #ifndef MODULE
9238         NULL, 0, load_java, NULL, NULL
9239     #else
9240         NULL, &__this_module, load_java, NULL, NULL
9241     #endif
9242 };
9243
9244 static int load_applet(struct linux_binprm *bprm,
9245                      struct pt_regs *regs)
9246 {
9247     int retval;
9248     MOD_INC_USE_COUNT;
9249     retval = do_load_applet(bprm, regs);
9250     MOD_DEC_USE_COUNT;
9251     return retval;
9252 }
9253
9254 static struct linux_binfmt applet_format = {
9255     #ifndef MODULE
9256         NULL, 0, load_applet, NULL, NULL
9257     #else
9258         NULL, &__this_module, load_applet, NULL, NULL
9259     #endif
9260 };
9261
9262 int __init init_java_binfmt(void)
9263 {
9264     register_binfmt(&java_format);
9265     return register_binfmt(&applet_format);
9266 }
9267
9268 #ifdef MODULE
9269 int init_module(void)
9270 {
9271     return init_java_binfmt();
9272 }
9273
9274 void cleanup_module(void) {
9275     unregister_binfmt(&java_format);
9276     unregister_binfmt(&applet_format);
9277 }
9278 #endif

```

fs/exec.c

```

9279 /*
9280  * linux/fs/exec.c

```

```

9281 *
9282 * Copyright (C) 1991, 1992 Linus Torvalds
9283 */
9284
9285 /*
9286 * #-checking implemented by tytso.
9287 */
9288 /* Demand-loading implemented 01.12.91 - no need to read
9289 * anything but the header into memory. The inode of the
9290 * executable is put into "current->executable", and page
9291 * faults do the actual loading. Clean.
9292 *
9293 * Once more I can proudly say that linux stood up to
9294 * being changed: it was less than 2 hours work to get
9295 * demand-loading completely implemented.
9296 *
9297 * Demand loading changed July 1993 by Eric Youngdale.
9298 * Use mmap instead, current->executable is only used by
9299 * the procs. This allows a dispatch table to check for
9300 * several different types of binary formats. We keep
9301 * trying until we recognize the file or we run out of
9302 * supported binary formats. */
9303
9304 #include <linux/config.h>
9305 #include <linux/slab.h>
9306 #include <linux/file.h>
9307 #include <linux/mman.h>
9308 #include <linux/a.out.h>
9309 #include <linux/stat.h>
9310 #include <linux/fcntl.h>
9311 #include <linux/user.h>
9312 #include <linux/smp_lock.h>
9313 #include <linux/init.h>
9314
9315 #include <asm/uaccess.h>
9316 #include <asm/pgtable.h>
9317 #include <asm/mmu_context.h>
9318
9319 #ifdef CONFIG_KMOD
9320 #include <linux/kmod.h>
9321 #endif
9322
9323 /* Here are the actual binaries that will be accepted:
9324 * add more with "register_binfmt()" if using modules...
9325 *
9326 * These are defined again for the 'real' modules if you
9327 * are using a module definition for these routines. */
9328

```

```

9329 static struct linux_binfmt *formats =
9330         (struct linux_binfmt *) NULL;
9331
9332 void __init binfmt_setup(void)
9333 {
9334 #ifdef CONFIG_BINfmt_MISC
9335     init_misc_binfmt();
9336 #endif
9337
9338 #ifdef CONFIG_BINfmt_ELF
9339     init_elf_binfmt();
9340 #endif
9341
9342 #ifdef CONFIG_BINfmt_ELF32
9343     init_elf32_binfmt();
9344 #endif
9345
9346 #ifdef CONFIG_BINfmt_AOUT
9347     init_aout_binfmt();
9348 #endif
9349
9350 #ifdef CONFIG_BINfmt_AOUT32
9351     init_aout32_binfmt();
9352 #endif
9353
9354 #ifdef CONFIG_BINfmt_JAVA
9355     init_java_binfmt();
9356 #endif
9357
9358 #ifdef CONFIG_BINfmt_EM86
9359     init_em86_binfmt();
9360 #endif
9361
9362     /* This cannot be configured out of the kernel */
9363     init_script_binfmt();
9364 }
9365
9366 int register_binfmt(struct linux_binfmt * fmt)
9367 {
9368     struct linux_binfmt ** tmp = &formats;
9369
9370     if (!fmt)
9371         return -EINVAL;
9372     if (fmt->next)
9373         return -EBUSY;
9374     while (*tmp) {
9375         if (fmt == *tmp)
9376             return -EBUSY;

```

```

9377     tmp = &(*tmp)->next;
9378 }
9379 fmt->next = formats;
9380 formats = fmt;
9381 return 0;
9382 }
9383
9384 #ifdef CONFIG_MODULES
9385 int unregister_binfmt(struct linux_binfmt * fmt)
9386 {
9387     struct linux_binfmt ** tmp = &formats;
9388
9389     while (*tmp) {
9390         if (fmt == *tmp) {
9391             *tmp = fmt->next;
9392             return 0;
9393         }
9394         tmp = &(*tmp)->next;
9395     }
9396     return -EINVAL;
9397 }
9398 #endif /* CONFIG_MODULES */
9399
9400 /* N.B. Error returns must be < 0 */
9401 int open_dentry(struct dentry * dentry, int mode)
9402 {
9403     struct inode * inode = dentry->d_inode;
9404     struct file * f;
9405     int fd, error;
9406
9407     error = -EINVAL;
9408     if (!inode->i_op || !inode->i_op->default_file_ops)
9409         goto out;
9410     fd = get_unused_fd();
9411     if (fd >= 0) {
9412         error = -ENFILE;
9413         f = get_empty_filp();
9414         if (!f)
9415             goto out_fd;
9416         f->f_flags = mode;
9417         f->f_mode = (mode+1) & O_ACCMODE;
9418         f->f_dentry = dentry;
9419         f->f_pos = 0;
9420         f->f_reada = 0;
9421         f->f_op = inode->i_op->default_file_ops;
9422         if (f->f_op->open) {
9423             error = f->f_op->open(inode,f);
9424             if (error)

```

```

9425         goto out_filp;
9426     }
9427     fd_install(fd, f);
9428     dget(dentry);
9429 }
9430 return fd;
9431
9432 out_filp:
9433     if (error > 0)
9434         error = -EIO;
9435     put_filp(f);
9436 out_fd:
9437     put_unused_fd(fd);
9438 out:
9439     return error;
9440 }
9441
9442 /* Note that a shared library must be both readable and
9443 * executable due to security reasons.
9444 *
9445 * Also note that we take the address to load from from
9446 * the file itself. */
9447 asmlinkage int sys_uselib(const char * library)
9448 {
9449     int fd, retval;
9450     struct file * file;
9451     struct linux_binfmt * fmt;
9452
9453     lock_kernel();
9454     fd = sys_open(library, 0, 0);
9455     retval = fd;
9456     if (fd < 0)
9457         goto out;
9458     file = fget(fd);
9459     retval = -ENOEXEC;
9460     if (file && file->f_dentry &&
9461         file->f_op && file->f_op->read) {
9462         for (fmt = formats ; fmt ; fmt = fmt->next) {
9463             int (*fn)(int) = fmt->load_shlib;
9464             if (!fn)
9465                 continue;
9466             /* N.B. Should use file instead of fd */
9467             retval = fn(fd);
9468             if (retval != -ENOEXEC)
9469                 break;
9470         }
9471     }
9472     fput(file);

```



```

9473 sys_close(fd);
9474 out:
9475 unlock_kernel();
9476 return retval;
9477 }
9478
9479 /* count() counts the number of arguments/envelopes */
9480 static int count(char ** argv)
9481 {
9482     int i = 0;
9483
9484     if (argv != NULL) {
9485         for (;;) {
9486             char * p;
9487             int error;
9488
9489             error = get_user(p,argv);
9490             if (error)
9491                 return error;
9492             if (!p)
9493                 break;
9494             argv++;
9495             i++;
9496         }
9497     }
9498     return i;
9499 }
9500
9501 /* 'copy_string()' copies argument/envelope strings from
9502 * user memory to free pages in kernel mem. These are in
9503 * a format ready to be put directly into the top of new
9504 * user memory.
9505 *
9506 * Modified by TYT, 11/24/91 to add the from_kmem
9507 * argument, which specifies whether the string and the
9508 * string array are from user or kernel segments:
9509 *
9510 * from_kmem   argv *      argv **
9511 * 0           user space  user space
9512 * 1           kernel space user space
9513 * 2           kernel space kernel space
9514 *
9515 * We do this by playing games with the fs segment
9516 * register. Since it is expensive to load a segment
9517 * register, we try to avoid calling set_fs() unless we
9518 * absolutely have to. */
9519 unsigned long copy_strings(
9520     int argc, char ** argv,

```

```

9521     unsigned long *page, unsigned long p, int from_kmem)
9522 {
9523     char *str;
9524     mm_segment_t old_fs;
9525
9526     if (!p)
9527         return 0; /* bullet-proofing */
9528     old_fs = get_fs();
9529     if (from_kmem==2)
9530         set_fs(KERNEL_DS);
9531     while (argc-- > 0) {
9532         int len;
9533         unsigned long pos;
9534
9535         if (from_kmem == 1)
9536             set_fs(KERNEL_DS);
9537         get_user(str, argv+argc);
9538         if (!str)
9539             panic("VFS: argc is wrong");
9540         if (from_kmem == 1)
9541             set_fs(old_fs);
9542         len = strlen_user(str); /* includes the '\0' */
9543         if (p < len) { /* this shouldn't happen - 128kB */
9544             set_fs(old_fs);
9545             return 0;
9546         }
9547         p -= len;
9548         pos = p;
9549         while (len) {
9550             char *pag;
9551             int offset, bytes_to_copy;
9552
9553             offset = pos % PAGE_SIZE;
9554             if (!(pag = (char *) page[pos/PAGE_SIZE]) &&
9555                 !(pag - (char *) page[pos/PAGE_SIZE] -
9556                   (unsigned long *) get_free_page(GFP_USER))) {
9557                 if (from_kmem==2)
9558                     set_fs(old_fs);
9559                 return 0;
9560             }
9561             bytes_to_copy = PAGE_SIZE - offset;
9562             if (bytes_to_copy > len)
9563                 bytes_to_copy = len;
9564             copy_from_user(pag + offset, str, bytes_to_copy);
9565             pos += bytes_to_copy;
9566             str += bytes_to_copy;
9567             len -= bytes_to_copy;
9568         }

```

```

9569     }
9570     if (from_kmem==2)
9571         set_fs(old_fs);
9572     return p;
9573 }
9574
9575 unsigned long setup_arg_pages(unsigned long p,
9576                               struct linux_binprm * bprm)
9577 {
9578     unsigned long stack_base;
9579     struct vm_area_struct *mpnt;
9580     int i;
9581
9582     stack_base = STACK_TOP - MAX_ARG_PAGES*PAGE_SIZE;
9583
9584     p += stack_base;
9585     if (bprm->loader)
9586         bprm->loader += stack_base;
9587     bprm->exec += stack_base;
9588
9589     mpnt = kmem_cache_alloc(vm_area_cache, SLAB_KERNEL);
9590     if (mpnt) {
9591         mpnt->vm_mm = current->mm;
9592         mpnt->vm_start = PAGE_MASK & (unsigned long) p;
9593         mpnt->vm_end = STACK_TOP;
9594         mpnt->vm_page_prot = PAGE_COPY;
9595         mpnt->vm_flags = VM_STACK_FLAGS;
9596         mpnt->vm_ops = NULL;
9597         mpnt->vm_offset = 0;
9598         mpnt->vm_file = NULL;
9599         mpnt->vm_pte = 0;
9600         insert_vm_struct(current->mm, mpnt);
9601         current->mm->total_vm =
9602             (mpnt->vm_end - mpnt->vm_start) >> PAGE_SHIFT;
9603     }
9604
9605     for (i = 0 ; i < MAX_ARG_PAGES ; i++) {
9606         if (bprm->page[i]) {
9607             current->mm->rss++;
9608             put_dirty_page(current, bprm->page[i], stack_base);
9609         }
9610         stack_base += PAGE_SIZE;
9611     }
9612     return p;
9613 }
9614
9615 /* Read in the complete executable. This is used for "-N"
9616  * files that aren't on a block boundary, and for files

```

```

9617  * on filesystems without bmap support. */
9618 int read_exec(
9619     struct dentry *dentry, unsigned long offset,
9620     char * addr, unsigned long count, int to_kmem)
9621 {
9622     struct file file;
9623     struct inode * inode = dentry->d_inode;
9624     int result = -ENOEXEC;
9625
9626     if (!inode->i_op || !inode->i_op->default_file_ops)
9627         goto end_readexec;
9628     if (init_private_file(&file, dentry, 1))
9629         goto end_readexec;
9630     if (!file.f_op->read)
9631         goto close_readexec;
9632     if (file.f_op->llseek) {
9633         if (file.f_op->llseek(&file, offset, 0) != offset)
9634             goto close_readexec;
9635     } else
9636         file.f_pos = offset;
9637     if (to_kmem) {
9638         mm_segment_t old_fs = get_fs();
9639         set_fs(get_ds());
9640         result =
9641             file.f_op->read(&file, addr, count, &file.f_pos);
9642         set_fs(old_fs);
9643     } else {
9644         result = verify_area(VERIFY_WRITE, addr, count);
9645         if (result)
9646             goto close_readexec;
9647         result =
9648             file.f_op->read(&file, addr, count, &file.f_pos);
9649     }
9650     close_readexec;
9651     if (file.f_op->release)
9652         file.f_op->release(inode, &file);
9653 end_readexec:
9654     return result;
9655 }
9656
9657 static int exec_mmap(void)
9658 {
9659     struct mm_struct * mm, * old_mm;
9660     int retval, nr;
9661
9662     if (atomic_read(&current->mm->count) == 1) {
9663         flush_cache_mm(current->mm);
9664         mm_release();

```

```

9665     release_segments(current->mm);
9666     exit_mmap(current->mm);
9667     flush_tlb_mm(current->mm);
9668     return 0;
9669 }
9670
9671     retval = -ENOMEM;
9672     mm = mm_alloc();
9673     if (!mm)
9674         goto fail_nomem;
9675
9676     mm->cpu_vm_mask = (1UL << smp_processor_id());
9677     mm->total_vm = 0;
9678     mm->rss = 0;
9679     /* Make sure we have a private ldt if needed ... */
9680     nr = current->tarray_ptr - &task[0];
9681     copy_segments(nr, current, mm);
9682
9683     old_mm = current->mm;
9684     current->mm = mm;
9685     retval = new_page_tables(current);
9686     if (retval)
9687         goto fail_restore;
9688     activate_context(current);
9689     up(&mm->mmap_sem);
9690     mm_release();
9691     mmput(old_mm);
9692     return 0;
9693
9694     /* Failure ... restore the prior mm_struct. */
9695 fail_restore:
9696     /* The pgd belongs to the parent ... don't free it! */
9697     mm->pgd = NULL;
9698     current->mm = old_mm;
9699     /* restore the ldt for this task */
9700     copy_segments(nr, current, NULL);
9701     mmput(mm);
9702
9703 fail_nomem:
9704     return retval;
9705 }
9706
9707 /* This function makes sure the current process has its
9708 * own signal table, so that flush_signal_handlers can
9709 * later reset the handlers without disturbing other
9710 * processes. (Other processes might share the signal
9711 * table via the CLONE_SIGHAND option to clone().) */
9712

```

```

9713 static inline int make_private_signals(void)
9714 {
9715     struct signal_struct * newsig;
9716
9717     if (atomic_read(&current->sig->count) <= 1)
9718         return 0;
9719     newsig = kmalloc(sizeof(*newsig), GFP_KERNEL);
9720     if (newsig == NULL)
9721         return -ENOMEM;
9722     spin_lock_init(&newsig->siglock);
9723     atomic_set(&newsig->count, 1);
9724     memcpy(newsig->action, current->sig->action,
9725            sizeof(newsig->action));
9726     current->sig = newsig;
9727     return 0;
9728 }
9729
9730 /* If make_private_signals() made a copy of the signal
9731 * table, decrement the refcount of the original table,
9732 * and free it if necessary. We don't do that in
9733 * make_private_signals() so that we can back off in
9734 * flush_old_exec() if an error occurs after calling
9735 * make_private_signals(). */
9736 static inline void release_old_signals(
9737     struct signal_struct * oldsig)
9738 {
9739     if (current->sig == oldsig)
9740         return;
9741     if (atomic_dec_and_test(&oldsig->count))
9742         kfree(oldsig);
9743 }
9744
9745 /* These functions flushes out all traces of the
9746 * currently running executable so that a new one can be
9747 * started */
9748 static inline void flush_old_files(
9749     struct files_struct * files)
9750 {
9751     unsigned long j;
9752
9753     j = 0;
9754     for (;;) {
9755         unsigned long set, i;
9756
9757         i = j * __NFDBITS;
9758         if (i >= files->max_fds)
9759             break;
9760         set = files->close_on_exec.fds_bits[j];

```

```

9761     files->close_on_exec.fds_bits[j] = 0;
9762     j++;
9763     for ( ; set ; i++,set >>= 1) {
9764         if (set & 1)
9765             sys_close(i);
9766     }
9767 }
9768 }
9769
9770 int flush_old_exec(struct linux_binprm * bprm)
9771 {
9772     char * name;
9773     int i, ch, retval;
9774     struct signal_struct * oldsig;
9775
9776     /* Make sure we have a private signal table */
9777     oldsig = current->sig;
9778     retval = make_private_signals();
9779     if (retval) goto flush_failed;
9780
9781     /* Release all of the old mmap stuff */
9782     retval = exec_mmap();
9783     if (retval) goto mmap_failed;
9784
9785     /* This is the point of no return */
9786     release_old_signals(oldsig);
9787
9788     if (current->euid == current->uid &&
9789         current->egid == current->gid)
9790         current->dumpable = 1;
9791     name = bprm->filename;
9792     for (i=0; (ch = *(name++)) != '\0';) {
9793         if (ch == '/')
9794             i = 0;
9795         else
9796             if (i < 15)
9797                 current->comm[i++] = ch;
9798     }
9799     current->comm[i] = '\0';
9800
9801     flush_thread();
9802
9803     if (bprm->e_uid != current->euid ||
9804         bprm->e_gid != current->egid ||
9805         permission(bprm->dentry->d_inode,MAY_READ))
9806         current->dumpable = 0;
9807
9808     flush_signal_handlers(current);

```

```

9809     flush_old_files(current->files);
9810
9811     return 0;
9812
9813 mmap_failed:
9814     if (current->sig != oldsig)
9815         kfree(current->sig);
9816 flush_failed:
9817     current->sig = oldsig;
9818     return retval;
9819 }
9820
9821 /* We mustn't allow tracing of suid binaries, unless the
9822  * tracer has the capability to trace anything.. */
9823 static inline int
9824 must_not_trace_exec(struct task_struct * p)
9825 {
9826     return (p->flags & PF_PTRACED) &&
9827         !cap_raised(p->p_pptr->cap_effective, CAP_SYS_PTRACE);
9828 }
9829
9830 /* Fill the binprm structure from the inode. Check
9831  * permissions, then read the first 512 bytes */
9832 int prepare_binprm(struct linux_binprm *bprm)
9833 {
9834     int mode;
9835     int retval,id_change,cap_raised;
9836     struct inode * inode = bprm->dentry->d_inode;
9837
9838     mode = inode->i_mode;
9839     if (!S_ISREG(mode)) /* must be regular file */
9840         return -EACCES;
9841     if (!(mode & 0111)) /* with >= 1 execute bit set */
9842         return -EACCES;
9843     if (IS_NOEXEC(inode)) /* FS mustn't be mounted noexec*/
9844         return -EACCES;
9845     if (!inode->i_sb)
9846         return -EACCES;
9847     if ((retval = permission(inode, MAY_EXEC)) != 0)
9848         return retval;
9849     /* better not exec files that are being written to */
9850     if (inode->i_writcount > 0)
9851         return -ETXTBSY;
9852
9853     bprm->e_uid = current->euid;
9854     bprm->e_gid = current->egid;
9855     id_change = cap_raised = 0;
9856

```

```

9857 /* Set-uid? */
9858 if (mode & S_ISUID) {
9859     bprm->e_uid = inode->i_uid;
9860     if (bprm->e_uid != current->euid)
9861         id_change = 1;
9862 }
9863
9864 /* Set-gid? */
9865 /* If setgid is set but no group execute bit then this
9866  * is a candidate for mandatory locking, not a setgid
9867  * executable. */
9868 if ((mode & (S_ISGID | S_IXGRP)) ==
9869     (S_ISGID | S_IXGRP)) {
9870     bprm->e_gid = inode->i_gid;
9871     if (!in_group_p(bprm->e_gid))
9872         id_change = 1;
9873 }
9874
9875 /* We don't have VFS support for capabilities yet */
9876 cap_clear(bprm->cap_inheritable);
9877 cap_clear(bprm->cap_permitted);
9878 cap_clear(bprm->cap_effective);
9879
9880 /* To support inheritance of root-permissions and
9881  * suid-root executables under compatibility mode, we
9882  * raise the effective and inherited bitmasks of the
9883  * executable file (translation: we set the executable
9884  * "capability dumb" and set the allowed set to
9885  * maximum). We don't set any forced bits.
9886  *
9887  * If only the real uid is 0, we only raise the
9888  * inheritable bitmask of the executable file
9889  * (translation: we set the allowed set to maximum and
9890  * the application to "capability smart"). */
9891
9892 if (!issecure(SECURE_NOROOT)) {
9893     if (bprm->e_uid == 0 || current->uid == 0)
9894         cap_set_full(bprm->cap_inheritable);
9895     if (bprm->e_uid == 0)
9896         cap_set_full(bprm->cap_effective);
9897 }
9898
9899 /* Only if pP' is _not_ a subset of pP, do we consider
9900  * there has been a capability related "change of
9901  * capability". In such cases, we need to check that
9902  * the elevation of privilege does not go against other
9903  * system constraints. The new Permitted set is
9904  * defined below -- see (***). */

```

```

9905 {
9906     kernel_cap_t working =
9907         cap_combine(bprm->cap_permitted,
9908                   cap_intersect(bprm->cap_inheritable,
9909                                current->cap_inheritable));
9909     if (!cap_issubset(working, current->cap_permitted)) {
9910         cap_raised = 1;
9911     }
9912 }
9913
9914 if (id_change || cap_raised) {
9915     /* We can't suid-execute if we're sharing parts of
9916      * the executable or if we're being traced (or if
9917      * suid execs are not allowed) (current->mm->count
9918      * > 1 is ok, as we'll get a new mm anyway) */
9919     if (IS_NOSUID(inode)
9920         || must_not_trace_exec(current)
9921         || (atomic_read(&current->fs->count) > 1)
9922         || (atomic_read(&current->sig->count) > 1)
9923         || (atomic_read(&current->files->count) > 1)) {
9924         if (id_change && !capable(CAP_SETUID))
9925             return -EPERM;
9926         if (cap_raised && !capable(CAP_SETPCAP))
9927             return -EPERM;
9928     }
9929 }
9930
9931 memset(bprm->buf, 0, sizeof(bprm->buf));
9932 return read_exec(bprm->dentry, 0, bprm->buf, 128, 1);
9933 }
9934
9935 /* This function is used to produce the new IDs and
9936  * capabilities from the old ones and the file's
9937  * capabilities. The formula used for evolving
9938  * capabilities is:
9939  *
9940  *
9941  *      pI' = pI
9942  *      (***) pP' = fP | (fI & pI)
9943  *      pE' = pP' & fE          [NB. fE is 0 or ~0]
9944  *
9945  * I=Inheritable, P=Permitted, E=Effective // p=process,
9946  * // f=file; ' indicates post-exec(). */
9947
9948 void compute_creds(struct linux_binprm *bprm)
9949 {
9950     int new_permitted = cap_t(bprm->cap_permitted) |
9951         (cap_t(bprm->cap_inheritable) &
9952          cap_t(current->cap_inheritable));

```

```

9953
9954 /* For init, we want to retain the capabilities set
9955  * in the init_task struct. Thus we skip the usual
9956  * capability rules */
9957 if (current->pid != 1) {
9958     cap_t(current->cap_permitted) = new_permitted;
9959     cap_t(current->cap_effective) = new_permitted &
9960         cap_t(bprm->cap_effective);
9961 }
9962
9963 /* AUD: Audit candidate if current->cap_effective is
9964  * set */
9965
9966 current->suid =
9967     current->euid = current->fsuid = bprm->e_uid;
9968 current->sgid =
9969     current->egid = current->fsgid = bprm->e_gid;
9970 if (current->euid != current->uid ||
9971     current->egid != current->gid ||
9972     !cap_issubset(new_permitted,
9973         current->cap_permitted))
9974     current->dumpable = 0;
9975 }
9976
9977
9978 void remove_arg_zero(struct linux_binprm *bprm)
9979 {
9980     if (bprm->argc) {
9981         unsigned long offset;
9982         char * page;
9983         offset = bprm->p % PAGE_SIZE;
9984         page = (char*)bprm->page[bprm->p/PAGE_SIZE];
9985         while(bprm->p++,*(page+offset++))
9986             if(offset==PAGE_SIZE){
9987                 offset=0;
9988                 page = (char*)bprm->page[bprm->p/PAGE_SIZE];
9989             }
9990         bprm->argc--;
9991     }
9992 }
9993
9994 /* cycle the list of binary formats handler, until one
9995  * recognizes the image */
9996 int search_binary_handler(struct linux_binprm *bprm,
9997     struct pt_regs *regs)
9998 {
9999     int try,retval=0;
10000     struct linux_binfmt *fmt;

```

```

10001 #ifdef __alpha__
10002 /* handle /sbin/loader.. */
10003 {
10004     struct exec * eh = (struct exec *) bprm->buf;
10005     struct linux_binprm bprm_loader;
10006
10007     if (!bprm->loader && eh->fh.f_magic == 0x183 &&
10008         (eh->fh.f_flags & 0x3000) == 0x3000)
10009     {
10010         int i;
10011         char * dynloader[] = { "/sbin/loader" };
10012         struct dentry * dentry;
10013
10014         dput(bprm->dentry);
10015         bprm->dentry = NULL;
10016
10017         bprm_loader.p = PAGE_SIZE * MAX_ARG_PAGES
10018             - sizeof(void *);
10019         for (i=0 ; i<MAX_ARG_PAGES ; i++)/* clear pg-tbl */
10020             bprm_loader.page[i] = 0;
10021
10022         dentry = open_namei(dynloader[0], 0, 0);
10023         retval = PTR_ERR(dentry);
10024         if (IS_ERR(dentry))
10025             return retval;
10026         bprm->dentry = dentry;
10027         bprm->loader = bprm_loader.p;
10028         retval = prepare_binprm(bprm);
10029         if (retval<0)
10030             return retval;
10031         /* should call search_binary_handler recursively
10032          * here, but it does not matter */
10033     }
10034 }
10035 #endif
10036 for (try=0; try<2; try++) {
10037     for (fmt = formats ; fmt ; fmt = fmt->next) {
10038         int (*fn)(struct linux_binprm *, struct pt_regs *)
10039             = fmt->load_binary;
10040         if (!fn)
10041             continue;
10042         retval = fn(bprm, regs);
10043         if (retval >= 0) {
10044             if (bprm->dentry)
10045                 dput(bprm->dentry);
10046             bprm->dentry = NULL;
10047             current->did_exec = 1;
10048             return retval;

```

```

10049     }
10050     if (retval != -ENOEXEC)
10051         break;
10052     /* We don't have the dentry anymore */
10053     if (!bprm->dentry)
10054         return retval;
10055     }
10056     if (retval != -ENOEXEC) {
10057         break;
10058 #ifdef CONFIG_KMOD
10059     } else {
10060 #define printable(c) \
10061     (((c)=='\t') || ((c)=='\n') || (0x20<=(c) && (c)<=0x7e))
10062     char modname[20];
10063     if (printable(bprm->buf[0]) &&
10064         printable(bprm->buf[1]) &&
10065         printable(bprm->buf[2]) &&
10066         printable(bprm->buf[3]))
10067         break; /* -ENOEXEC */
10068     sprintf(modname, "binfmt-%04x",
10069             *(unsigned short *)&bprm->buf[2]);
10070     request_module(modname);
10071 #endif
10072     }
10073     }
10074     return retval;
10075 }
10076
10077
10078 /* sys_execve() executes a new program. */
10079 int do_execve(char * filename, char ** argv,
10080              char ** envp, struct pt_regs * regs)
10081 {
10082     struct linux_binprm bprm;
10083     struct dentry * dentry;
10084     int retval;
10085     int i;
10086
10087     bprm.p = PAGE_SIZE*MAX_ARG_PAGES-sizeof(void *);
10088     for (i=0 ; i<MAX_ARG_PAGES ; i++) /* clear pg_tbl */
10089         bprm.page[i] = 0;
10090
10091     dentry = open_namei(filename, 0, 0);
10092     retval = PTR_ERR(dentry);
10093     if (IS_ERR(dentry))
10094         return retval;
10095
10096     bprm.dentry = dentry;
10097     bprm.filename = filename;

```

```

10098     bprm.sh_bang = 0;
10099     bprm.java = 0;
10100     bprm.loader = 0;
10101     bprm.exec = 0;
10102     if ((bprm.argc = count(argv)) < 0) {
10103         dput(dentry);
10104         return bprm.argc;
10105     }
10106
10107     if ((bprm.envc = count(envp)) < 0) {
10108         dput(dentry);
10109         return bprm.envc;
10110     }
10111
10112     retval = prepare_binprm(&bprm);
10113
10114     if (retval >= 0) {
10115         bprm.p = copy_strings(1, &bprm.filename, bprm.page,
10116                             bprm.p, 2);
10117         bprm.exec = bprm.p;
10118         bprm.p = copy_strings(bprm.envc, envp, bprm.page,
10119                             bprm.p, 0);
10120         bprm.p = copy_strings(bprm.argc, argv, bprm.page,
10121                             bprm.p, 0);
10122         if (!bprm.p)
10123             retval = -E2BIG;
10124     }
10125
10126     if (retval >= 0)
10127         retval = search_binary_handler(&bprm, regs);
10128     if (retval >= 0)
10129         /* execve success */
10130         return retval;
10131
10132     /* Something went wrong, return the inode and free the
10133     * argument pages*/
10134     if (bprm.dentry)
10135         dput(bprm.dentry);
10136
10137     for (i=0 ; i<MAX_ARG_PAGES ; i++)
10138         free_page(bprm.page[i]);
10139
10140     return retval;
10141 }

```

include/asm-generic/smplock.h

```

10142 /*
10143  * <asm/smplock.h>

```

```

10144 *
10145 * Default SMP lock implementation
10146 */
10147 #include <linux/interrupt.h>
10148 #include <asm/spinlock.h>
10149
10150 extern spinlock_t kernel_flag;
10151
10152 /* Release global kernel lock and global interrupt lock
10153 */
10154 #define release_kernel_lock(task, cpu) \
10155 do { \
10156     if (task->lock_depth >= 0) \
10157         spin_unlock(&kernel_flag); \
10158     release_irqlock(cpu); \
10159     __sti(); \
10160 } while (0)
10161
10162 /* Re-acquire the kernel lock */
10163 #define reacquire_kernel_lock(task) \
10164 do { \
10165     if (task->lock_depth >= 0) \
10166         spin_lock(&kernel_flag); \
10167 } while (0)
10168
10169
10170 /* Getting the big kernel lock.
10171 *
10172 * This cannot happen asynchronously, so we only need to
10173 * worry about other CPUs. */
10174 extern __inline__ void lock_kernel(void)
10175 {
10176     if (!++current->lock_depth)
10177         spin_lock(&kernel_flag);
10178 }
10179
10180 extern __inline__ void unlock_kernel(void)
10181 {
10182     if (--current->lock_depth < 0)
10183         spin_unlock(&kernel_flag);
10184 }

```

include/asm-i386/atomic.h

```

10185 #ifndef __ARCH_I386_ATOMIC__
10186 #define __ARCH_I386_ATOMIC__
10187
10188 /* Atomic operations that C can't guarantee us. Useful

```

```

10189 * for resource counting etc.. */
10190
10191 #ifdef __SMP__
10192 #define LOCK "lock ; "
10193 #else
10194 #define LOCK ""
10195 #endif
10196
10197 /* Make sure gcc doesn't try to be clever and move things
10198 * around on us. We need to use __exactly__ the address the
10199 * user gave us, not some alias that contains the same
10200 * information. */
10201 #define __atomic_fool_gcc(x) \
10202 (*(volatile struct { int a[100]; } *)x)
10203
10204 #ifdef __SMP__
10205 typedef struct { volatile int counter; } atomic_t;
10206 #else
10207 typedef struct { int counter; } atomic_t;
10208 #endif
10209
10210 #define ATOMIC_INIT(i) { (i) }
10211
10212 #define atomic_read(v) ((v)->counter)
10213 #define atomic_set(v,i) ((v)->counter) = (i)
10214
10215 static __inline__ void atomic_add(int i,
10216                                   volatile atomic_t *v)
10217 {
10218     __asm__ __volatile__(
10219         LOCK "addl %1,%0"
10220         : "=m" (__atomic_fool_gcc(v))
10221         : "ir" (i), "m" (__atomic_fool_gcc(v)));
10222 }
10223
10224 static __inline__ void atomic_sub(int i,
10225                                   volatile atomic_t *v)
10226 {
10227     __asm__ __volatile__(
10228         LOCK "subl %1,%0"
10229         : "=m" (__atomic_fool_gcc(v))
10230         : "ir" (i), "m" (__atomic_fool_gcc(v)));
10231 }
10232
10233 static __inline__ void atomic_inc(volatile atomic_t *v)
10234 {
10235     __asm__ __volatile__(
10236         LOCK "incl %0"

```



```

10237     : "=m" (__atomic_fool_gcc(v))
10238     : "m" (__atomic_fool_gcc(v));
10239 }
10240
10241 static __inline__ void atomic_dec(volatile atomic_t *v)
10242 {
10243     __asm__ __volatile__(
10244         LOCK "decl %0"
10245         : "=m" (__atomic_fool_gcc(v))
10246         : "m" (__atomic_fool_gcc(v)));
10247 }
10248
10249 static __inline__ int atomic_dec_and_test(
10250     volatile atomic_t *v)
10251 {
10252     unsigned char c;
10253
10254     __asm__ __volatile__(
10255         LOCK "decl %0; sete %1"
10256         : "=m" (__atomic_fool_gcc(v)), "=qm" (c)
10257         : "m" (__atomic_fool_gcc(v)));
10258     return c != 0;
10259 }
10260
10261 /* These are x86-specific, used by some header files */
10262 #define atomic_clear_mask(mask, addr) \
10263     __asm__ __volatile__(LOCK "andl %0,%1" \
10264         : : "r" (~(mask)), \
10265         "m" (__atomic_fool_gcc(addr)) : "memory")
10266
10267 #define atomic_set_mask(mask, addr) \
10268     __asm__ __volatile__(LOCK "orl %0,%1" \
10269         : : "r" (mask), "m" (__atomic_fool_gcc(addr)) : "memory")
10270
10271 #endif

```

include/asm-i386/current.h

```

10272 #ifndef _I386_CURRENT_H
10273 #define _I386_CURRENT_H
10274
10275 struct task_struct;
10276
10277 static inline struct task_struct * get_current(void)
10278 {
10279     struct task_struct *current;
10280     __asm__ ("andl %%esp,%0; ; "=r" (current) :
10281         "0" (~8191UL));

```

```

10282     return current;
10283 }
10284
10285 #define current get_current()
10286
10287 #endif /* !(_I386_CURRENT_H) */

```

include/asm-i386/dma.h

```

10288 /* $Id: dma.h,v 1.7 1992/12/14 00:29:34 root Exp root $
10289 * linux/include/asm/dma.h: Defines for using and
10290 * allocating dma channels.
10291 * Written by Hennus Bergman, 1992.
10292 * High DMA channel support & info by Hannu Savolainen
10293 * and John Boyd, Nov. 1992. */
10294
10295 #ifndef _ASM_DMA_H
10296 #define _ASM_DMA_H
10297
10298 #include <linux/config.h>
10299 #include <asm/io.h> /* need byte IO */
10300 #include <asm/spinlock.h> /* And spinlocks */
10301 #include <linux/delay.h>
10302
10303
10304 #ifdef HAVE_REALLY_SLOW_DMA_CONTROLLER
10305 #define dma_outb     outb_p
10306 #else
10307 #define dma_outb     outb
10308 #endif
10309
10310 #define dma_inb     inb
10311
10312 /* NOTES about DMA transfers:
10313 *
10314 * controller 1: chans 0-3, byte operations, ports 00-1F
10315 * controller 2: chans 4-7, word operations, ports C0-DF
10316 *
10317 * - ALL registers are 8 bits only, regardless of
10318 *   transfer size
10319 * - channel 4 is not used - cascades 1 into 2.
10320 * - channels 0-3 are byte - addresses/counts are for
10321 *   physical bytes
10322 * - channels 5-7 are word - addresses/counts are for
10323 *   physical words
10324 * - transfers must not cross physical 64K (0-3) or 128K
10325 *   (5-7) boundaries
10326 * - transfer count loaded to registers is 1 less than

```



```

10423 #define DMA_PAGE_1      0x83
10424 #define DMA_PAGE_2      0x81
10425 #define DMA_PAGE_3      0x82
10426 #define DMA_PAGE_5      0x8B
10427 #define DMA_PAGE_6      0x89
10428 #define DMA_PAGE_7      0x8A
10429
10430 /* I/O to memory, no autoinit, increment, single mode */
10431 #define DMA_MODE_READ      0x44
10432 /* memory to I/O, no autoinit, increment, single mode */
10433 #define DMA_MODE_WRITE     0x48
10434 /* pass thru DREQ->HRQ, DACK<-HLDA only */
10435 #define DMA_MODE_CASCADE   0xC0
10436
10437 #define DMA_AUTOINIT       0x10
10438
10439
10440 extern spinlock_t dma_spin_lock;
10441
10442 static __inline__ unsigned long claim_dma_lock(void)
10443 {
10444     unsigned long flags;
10445     spin_lock_irqsave(&dma_spin_lock, flags);
10446     return flags;
10447 }
10448
10449 static __inline__ void release_dma_lock(
10450     unsigned long flags)
10451 {
10452     spin_unlock_irqrestore(&dma_spin_lock, flags);
10453 }
10454
10455 /* enable/disable a specific DMA channel */
10456 static __inline__ void enable_dma(unsigned int dmanr)
10457 {
10458     if (dmanr<=3)
10459         dma_outb(dmanr, DMA1_MASK_REG);
10460     else
10461         dma_outb(dmanr & 3, DMA2_MASK_REG);
10462 }
10463
10464 static __inline__ void disable_dma(unsigned int dmanr)
10465 {
10466     if (dmanr<=3)
10467         dma_outb(dmanr | 4, DMA1_MASK_REG);
10468     else
10469         dma_outb((dmanr & 3) | 4, DMA2_MASK_REG);
10470 }

```

```

10471
10472 /* Clear the 'DMA Pointer Flip Flop'.
10473 * Write 0 for LSB/MSB, 1 for MSB/LSB access.
10474 * Use this once to initialize the FF to a known state.
10475 * After that, keep track of it. :-)
10476 * -- In order to do that, the DMA routines below should
10477 * -- only be used while holding the DMA lock ! */
10478 static __inline__ void clear_dma_ff(unsigned int dmanr)
10479 {
10480     if (dmanr<=3)
10481         dma_outb(0, DMA1_CLEAR_FF_REG);
10482     else
10483         dma_outb(0, DMA2_CLEAR_FF_REG);
10484 }
10485
10486 /* set mode (above) for a specific DMA channel */
10487 static __inline__ void set_dma_mode(unsigned int dmanr,
10488     char mode)
10489 {
10490     if (dmanr<=3)
10491         dma_outb(mode | dmanr, DMA1_MODE_REG);
10492     else
10493         dma_outb(mode | (dmanr&3), DMA2_MODE_REG);
10494 }
10495
10496 /* Set only the page register bits of the transfer
10497 * address. This is used for successive transfers when
10498 * we know the contents of the lower 16 bits of the DMA
10499 * current address register, but a 64k boundary may have
10500 * been crossed. */
10501 static __inline__ void set_dma_page(unsigned int dmanr,
10502     char pagenr)
10503 {
10504     switch(dmanr) {
10505     case 0:
10506         dma_outb(pagenr, DMA_PAGE_0);
10507         break;
10508     case 1:
10509         dma_outb(pagenr, DMA_PAGE_1);
10510         break;
10511     case 2:
10512         dma_outb(pagenr, DMA_PAGE_2);
10513         break;
10514     case 3:
10515         dma_outb(pagenr, DMA_PAGE_3);
10516         break;
10517     case 5:
10518         dma_outb(pagenr & 0xfe, DMA_PAGE_5);

```

```

10519     break;
10520     case 6:
10521         dma_outb(pagenr & 0xfe, DMA_PAGE_6);
10522         break;
10523     case 7:
10524         dma_outb(pagenr & 0xfe, DMA_PAGE_7);
10525         break;
10526     }
10527 }
10528
10529
10530 /* Set transfer address & page bits for specific DMA
10531 * channel. Assumes dma flipflop is clear. */
10532 static __inline__ void set_dma_addr(unsigned int dmanr,
10533                                     unsigned int a)
10534 {
10535     set_dma_page(dmanr, a>>16);
10536     if (dmanr <= 3) {
10537         dma_outb(a & 0xff,
10538                 ((dmanr&3)<<1) + IO_DMA1_BASE );
10539         dma_outb((a>>8) & 0xff,
10540                 ((dmanr&3)<<1) + IO_DMA1_BASE);
10541     } else {
10542         dma_outb((a>>1) & 0xff,
10543                 ((dmanr&3)<<2) + IO_DMA2_BASE);
10544         dma_outb((a>>9) & 0xff,
10545                 ((dmanr&3)<<2) + IO_DMA2_BASE);
10546     }
10547 }
10548
10549
10550 /* Set transfer size (max 64k for DMA1..3, 128k for
10551 * DMA5..7) for a specific DMA channel. You must ensure
10552 * the parameters are valid. NOTE: from a manual: "the
10553 * number of transfers is one more than the initial word
10554 * count"! This is taken into account. Assumes dma
10555 * flip-flop is clear. NOTE 2: "count" represents
10556 * _bytes_ and must be even for channels 5-7. */
10557 static __inline__ void set_dma_count(unsigned int dmanr,
10558                                     unsigned int count)
10559 {
10560     count--;
10561     if (dmanr <= 3) {
10562         dma_outb(count & 0xff,
10563                 ((dmanr&3)<<1) + 1 + IO_DMA1_BASE);
10564         dma_outb((count>>8) & 0xff,
10565                 ((dmanr&3)<<1) + 1 + IO_DMA1_BASE);
10566     } else {

```

```

10567         dma_outb((count>>1) & 0xff,
10568                 ((dmanr&3)<<2) + 2 + IO_DMA2_BASE);
10569         dma_outb((count>>9) & 0xff,
10570                 ((dmanr&3)<<2) + 2 + IO_DMA2_BASE);
10571     }
10572 }
10573
10574
10575 /* Get DMA residue count. After a DMA transfer, this
10576 * should return zero. Reading this while a DMA transfer
10577 * is still in progress will return unpredictable
10578 * results. If called before the channel has been used,
10579 * it may return 1. Otherwise, it returns the number of
10580 * _bytes_ left to transfer.
10581 *
10582 * Assumes DMA flip-flop is clear. */
10583 static __inline__ int get_dma_residue(unsigned int dmanr)
10584 {
10585     unsigned int io_port = (dmanr <= 3)
10586         ? ((dmanr & 3) << 1) + 1 + IO_DMA1_BASE
10587         : ((dmanr & 3) << 2) + 2 + IO_DMA2_BASE;
10588
10589     /* using short to get 16-bit wrap around */
10590     unsigned short count;
10591
10592     count = 1 + dma_inb(io_port);
10593     count += dma_inb(io_port) << 8;
10594
10595     return (dmanr<=3)? count : (count<<1);
10596 }
10597
10598
10599 /* These are in kernel/dma.c: */
10600 /* reserve a DMA channel */
10601 extern int request_dma(unsigned int dmanr,
10602                       const char * device_id);
10603 /* release it again */
10604 extern void free_dma(unsigned int dmanr);
10605
10606 /* From PCI */
10607
10608 #ifdef CONFIG_PCI_QUIRKS
10609 extern int isa_dma_bridge_buggy;
10610 #else
10611 #define isa_dma_bridge_buggy (0)
10612 #endif
10613
10614 #endif /* _ASM_DMA_H */

```

include/asm-i386/elf.h

```
10615 #ifndef __ASM1386_ELF_H
10616 #define __ASM1386_ELF_H
10617
10618 /* ELF register definitions.. */
10619
10620 #include <asm/ptrace.h>
10621 #include <asm/user.h>
10622
10623 typedef unsigned long elf_greg_t;
10624
10625 #define ELF_NGREG
10626 (sizeof (struct user_regs_struct) / sizeof(elf_greg_t))
10627 typedef elf_greg_t elf_gregset_t[ELF_NGREG];
10628
10629 typedef struct user_i387_struct elf_fpregset_t;
10630
10631 /* This is used to ensure we don't load something for the
10632 * wrong architecture. */
10633 #define elf_check_arch(x)
10634 ((x) == EM_386) || ((x) == EM_486) )
10635
10636 /* These are used to set parameters in the core dumps. */
10637 #define ELF_CLASS ELFCLASS32
10638 #define ELF_DATA ELFDATA2LSB
10639 #define ELF_ARCH EM_386
10640
10641 /* SVR4/i386 ABI (pages 3-31, 3-32) says that when the
10642 program starts %edx contains a pointer to a function
10643 which might be registered using 'atexit'. This
10644 provides a mean for the dynamic linker to call DT_FINI
10645 functions for shared libraries that have been loaded
10646 before the code runs.
10647
10648 A value of 0 tells we have no such handler.
10649
10650 We might as well make sure everything else is cleared
10651 too (except for %esp), just to make things more
10652 deterministic. */
10653 #define ELF_PLAT_INIT(_r) do {
10654 _r->ebx = 0; _r->ecx = 0; _r->edx = 0;
10655 _r->esi = 0; _r->edi = 0; _r->ebp = 0;
10656 _r->eax = 0;
10657 } while (0)
10658
10659 #define USE_ELF_CORE_DUMP
10660 #define ELF_EXEC_PAGESIZE 4096
```

```
10661
10662 /* This is the location that an ET_DYN program is loaded
10663 if exec'ed. Typical use of this is to invoke "./ld.so
10664 someprog" to test out a new version of the loader. We
10665 need to make sure that it is out of the way of the
10666 program that it will "exec", and that there is
10667 sufficient room for the brk. */
10668
10669 #define ELF_ET_DYN_BASE (2 * TASK_SIZE / 3)
10670
10671 /* Wow, the "main" arch needs arch dependent functions
10672 * too.. :) */
10673
10674 /* regs is struct pt_regs, pr_reg is elf_gregset_t (which
10675 * is now struct_user_regs, they are different) */
10676
10677 #define ELF_CORE_COPY_REGS(pr_reg, regs)
10678 pr_reg[0] = regs->ebx;
10679 pr_reg[1] = regs->ecx;
10680 pr_reg[2] = regs->edx;
10681 pr_reg[3] = regs->esi;
10682 pr_reg[4] = regs->edi;
10683 pr_reg[5] = regs->ebp;
10684 pr_reg[6] = regs->eax;
10685 pr_reg[7] = regs->xds;
10686 pr_reg[8] = regs->xes;
10687 /* fake once used fs and gs selectors? */
10688 pr_reg[9] = regs->xds; /* was fs and __fs */
10689 pr_reg[10] = regs->xds; /* was gs and __gs */
10690 pr_reg[11] = regs->orig_eax;
10691 pr_reg[12] = regs->eip;
10692 pr_reg[13] = regs->xcs;
10693 pr_reg[14] = regs->eflags;
10694 pr_reg[15] = regs->esp;
10695 pr_reg[16] = regs->xss;
10696
10697 /* This yields a mask that user programs can use to
10698 figure out what instruction set this CPU supports.
10699 This could be done in user space, but it's not easy,
10700 and we've already done it here. */
10701
10702 #define ELF_HWCAP (boot_cpu_data.x86_capability)
10703
10704 /* This yields a string that ld.so will use to load
10705 implementation specific libraries for optimization.
10706 This is more specific in intent than poking at uname
10707 or /proc/cpuinfo.
10708
```

```

10709   For the moment, we have only optimizations for the
10710   Intel generations, but that could change... */
10711
10712 #define ELF_PLATFORM
10713   ("i386\0i486\0i586\0i686"+((boot_cpu_data.x86-3)*5))
10714
10715 #ifdef __KERNEL__
10716 #define SET_PERSONALITY(ex, ibcs2)
10717   current->personality = (ibcs2 ? PER_SVR4 : PER_LINUX)
10718 #endif
10719
10720 #endif

```

include/asm-i386/hardirq.h

```

10721 #ifndef __ASM_HARDIRQ_H
10722 #define __ASM_HARDIRQ_H
10723
10724 #include <linux/tasks.h>
10725
10726 extern unsigned int local_irq_count[NR_CPUS];
10727
10728 /* Are we in an interrupt context? Either doing bottom
10729  * half or hardware interrupt processing? */
10730 #define in_interrupt()
10731   (( int __cpu = smp_processor_id();
10732   (local_irq_count[__cpu] + local_bh_count[__cpu] != 0); })
10733
10734 #ifndef __SMP__
10735
10736 #define hardirq_trylock(cpu) (local_irq_count[cpu] == 0)
10737 #define hardirq_endlock(cpu) do { } while (0)
10738
10739 #define hardirq_enter(cpu) (local_irq_count[cpu]++)
10740 #define hardirq_exit(cpu) (local_irq_count[cpu]--)
10741
10742 #define synchronize_irq() barrier()
10743
10744 #else
10745
10746 #include <asm/atomic.h>
10747
10748 extern unsigned char global_irq_holder;
10749 extern unsigned volatile int global_irq_lock;
10750 extern atomic_t global_irq_count;
10751
10752 static inline void release_irqlock(int cpu)
10753 {

```

```

10754   /* if we didn't own the irq lock, just ignore.. */
10755   if (global_irq_holder == (unsigned char) cpu) {
10756       global_irq_holder = NO_PROC_ID;
10757       clear_bit(0,&global_irq_lock);
10758   }
10759 }
10760
10761 static inline void hardirq_enter(int cpu)
10762 {
10763   ++local_irq_count[cpu];
10764   atomic_inc(&global_irq_count);
10765 }
10766
10767 static inline void hardirq_exit(int cpu)
10768 {
10769   atomic_dec(&global_irq_count);
10770   --local_irq_count[cpu];
10771 }
10772
10773 static inline int hardirq_trylock(int cpu)
10774 {
10775   return !atomic_read(&global_irq_count) &&
10776   !test_bit(0,&global_irq_lock);
10777 }
10778
10779 #define hardirq_endlock(cpu) do { } while (0)
10780
10781 extern void synchronize_irq(void);
10782
10783 #endif /* __SMP__ */
10784
10785 #endif /* __ASM_HARDIRQ_H */

```

include/asm-i386/page.h

```

10786 #ifndef _I386_PAGE_H
10787 #define _I386_PAGE_H
10788
10789 /* PAGE_SHIFT determines the page size */
10790 #define PAGE_SHIFT 12
10791 #define PAGE_SIZE (1UL << PAGE_SHIFT)
10792 #define PAGE_MASK (~(PAGE_SIZE-1))
10793
10794 #ifdef __KERNEL__
10795 #ifndef __ASSEMBLY__
10796
10797 #define STRICT_MM_TYPECHECKS
10798

```

```

10799 #define clear_page(page)
10800 memset((void*)(page), 0, PAGE_SIZE)
10801 #define copy_page(to,from)
10802 memcpy((void*)(to), (void*)(from), PAGE_SIZE)
10803
10804 #ifdef STRICT_MM_TYPECHECKS
10805 /* These are used to make use of C type-checking.. */
10806 typedef struct { unsigned long pte; } pte_t;
10807 typedef struct { unsigned long pmd; } pmd_t;
10808 typedef struct { unsigned long pgd; } pgd_t;
10809 typedef struct { unsigned long pgprot; } pgprot_t;
10810
10811 #define pte_val(x) ((x).pte)
10812 #define pmd_val(x) ((x).pmd)
10813 #define pgd_val(x) ((x).pgd)
10814 #define pgprot_val(x) ((x).pgprot)
10815
10816 #define __pte(x) ((pte_t) { (x) })
10817 #define __pmd(x) ((pmd_t) { (x) })
10818 #define __pgd(x) ((pgd_t) { (x) })
10819 #define __pgprot(x) ((pgprot_t) { (x) })
10820
10821 #else
10822 /* .. while these make it easier on the compiler */
10823 typedef unsigned long pte_t;
10824 typedef unsigned long pmd_t;
10825 typedef unsigned long pgd_t;
10826 typedef unsigned long pgprot_t;
10827
10828 #define pte_val(x) (x)
10829 #define pmd_val(x) (x)
10830 #define pgd_val(x) (x)
10831 #define pgprot_val(x) (x)
10832
10833 #define __pte(x) (x)
10834 #define __pmd(x) (x)
10835 #define __pgd(x) (x)
10836 #define __pgprot(x) (x)
10837
10838 #endif
10839 #endif /* !__ASSEMBLY__ */
10840
10841 /* to align the pointer to the (next) page boundary */
10842 #define PAGE_ALIGN(addr) (((addr)+PAGE_SIZE-1)&PAGE_MASK)
10843
10844 /* This handles the memory map.. We could make this a
10845 * config option, but too many people screw it up, and
10846 * too few need it.

```

```

10847 *
10848 * A __PAGE_OFFSET of 0xC0000000 means that the kernel
10849 * has a virtual address space of one gigabyte, which
10850 * limits the amount of physical memory you can use to
10851 * about 950MB. If you want to use more physical memory,
10852 * change this define.
10853 *
10854 * For example, if you have 2GB worth of physical memory,
10855 * you could change this define to 0x80000000, which
10856 * gives the kernel 2GB of virtual memory (enough to most
10857 * of your physical memory as the kernel needs a bit
10858 * extra for various io-memory mappings)
10859 *
10860 * IF YOU CHANGE THIS, PLEASE ALSO CHANGE
10861 *
10862 *     arch/i386/vmlinux.lds
10863 *
10864 * which has the same constant encoded.. */
10865 #define __PAGE_OFFSET (0xC0000000)
10866
10867 #define PAGE_OFFSET ((unsigned long)__PAGE_OFFSET)
10868 #define __pa(x) ((unsigned long)(x)-PAGE_OFFSET)
10869 #define __va(x) ((void*)((unsigned long)(x) +
10870 PAGE_OFFSET))
10871 #define MAP_NR(addr) (__pa(addr) >> PAGE_SHIFT)
10872
10873 #endif /* __KERNEL__ */
10874
10875 #endif /* _I386_PAGE_H */

```

include/asm-i386/pgtable.h

```

10876 #ifndef _I386_PGTABLE_H
10877 #define _I386_PGTABLE_H
10878
10879 #include <linux/config.h>
10880
10881 /* The Linux memory management assumes a three-level page
10882 * table setup. On the i386, we use that, but "fold" the
10883 * mid level into the top-level page table, so that we
10884 * physically have the same two-level page table as the
10885 * i386 mmu expects.
10886 *
10887 * This file contains the functions and defines necessary
10888 * to modify and use the i386 page table tree. */
10889 #ifndef __ASSEMBLY__
10890 #include <asm/processor.h>
10891 #include <asm/fixmap.h>

```

```

10892 #include <linux/tasks.h>
10893
10894 /* Caches aren't brain-dead on the intel. */
10895 #define flush_cache_all() do { } while (0)
10896 #define flush_cache_mm(mm) do { } while (0)
10897 #define flush_cache_range(mm, start, end) \
10898 do { } while (0)
10899 #define flush_cache_page(vma, vmaddr) do { } while (0)
10900 #define flush_page_to_ram(page) do { } while (0)
10901 #define flush_icache_range(start, end) do { } while (0)
10902
10903 /* TLB flushing:
10904 *
10905 * - flush_tlb() flushes the current mm struct TLBs
10906 * - flush_tlb_all() flushes all processes TLBs
10907 * - flush_tlb_mm(mm) flushes the specified mm context
10908 *   TLB's
10909 * - flush_tlb_page(vma, vmaddr) flushes one page
10910 * - flush_tlb_range(mm, start, end) flushes a range of
10911 *   pages
10912 *
10913 * ..but the i386 has somewhat limited tlb flushing
10914 * capabilities, and page-granular flushes are available
10915 * only on i486 and up. */
10916
10917 #define __flush_tlb() \
10918 do { unsigned long tmpreg; \
10919 __asm__ __volatile__ ("movl %%cr3,%0\n\tmovl %0,%%cr3": \
10920 "r" (tmpreg) : : "memory"); } while (0)
10921
10922 #ifndef CONFIG_X86_INVLPG
10923 #define __flush_tlb_one(addr) flush_tlb()
10924 #else
10925 #define __flush_tlb_one(addr) \
10926 __asm__ __volatile__ ("invlpg %0": : "m" (*(char *) addr))
10927 #endif
10928
10929 #ifndef __SMP__
10930
10931 #define flush_tlb() __flush_tlb()
10932 #define flush_tlb_all() __flush_tlb()
10933 #define local_flush_tlb() __flush_tlb()
10934
10935 static inline void flush_tlb_mm(struct mm_struct *mm)
10936 {
10937     if (mm == current->mm)
10938         __flush_tlb();
10939 }

```

```

10940
10941 static inline void flush_tlb_page(
10942     struct vm_area_struct *vma, unsigned long addr)
10943 {
10944     if (vma->vm_mm == current->mm)
10945         __flush_tlb_one(addr);
10946 }
10947
10948 static inline void flush_tlb_range(struct mm_struct *mm,
10949     unsigned long start, unsigned long end)
10950 {
10951     if (mm == current->mm)
10952         __flush_tlb();
10953 }
10954
10955 #else
10956
10957 /* We aren't very clever about this yet - SMP could
10958 * certainly avoid some global flushes.. */
10959 #include <asm/smp.h>
10960
10961 #define local_flush_tlb()
10962 __flush_tlb()
10963
10964 #define CLEVER_SMP_INVALIDATE
10965 #ifdef CLEVER_SMP_INVALIDATE
10966
10967 /* Smarter SMP flushing macros. c/o Linus Torvalds.
10968 * These mean you can really definitely utterly forget
10969 * about writing to user space from interrupts. (It's not
10970 * allowed anyway). */
10971
10972 static inline void flush_tlb_current_task(void)
10973 {
10974     /* just one copy of this mm? */
10975     if (atomic_read(&current->mm->count) == 1)
10976         local_flush_tlb(); /* and that's us, so.. */
10977     else
10978         smp_flush_tlb();
10979 }
10980
10981 #define flush_tlb() flush_tlb_current_task()
10982 #define flush_tlb_all() smp_flush_tlb()
10983
10984 static inline void flush_tlb_mm(struct mm_struct * mm)
10985 {

```



```

10988 if (mm == current->mm && atomic_read(&mm->count) == 1)
10989     local_flush_tlb();
10990 else
10991     smp_flush_tlb();
10992 }
10993
10994 static inline void flush_tlb_page(
10995     struct vm_area_struct * vma, unsigned long va)
10996 {
10997     if (vma->vm_mm == current->mm &&
10998         atomic_read(&current->mm->count) == 1)
10999         __flush_tlb_one(va);
11000 else
11001     smp_flush_tlb();
11002 }
11003
11004 static inline void flush_tlb_range(struct mm_struct * mm,
11005     unsigned long start, unsigned long end)
11006 {
11007     flush_tlb_mm(mm);
11008 }
11009
11010
11011 #else
11012
11013 #define flush_tlb()
11014     smp_flush_tlb()
11015
11016 #define flush_tlb_all() flush_tlb()
11017
11018 static inline void flush_tlb_mm(struct mm_struct *mm)
11019 {
11020     flush_tlb();
11021 }
11022
11023 static inline void flush_tlb_page(
11024     struct vm_area_struct *vma, unsigned long addr)
11025 {
11026     flush_tlb();
11027 }
11028
11029 static inline void flush_tlb_range(struct mm_struct *mm,
11030     unsigned long start, unsigned long end)
11031 {
11032     flush_tlb();
11033 }
11034 #endif
11035 #endif

```

```

11036 #endif /* !__ASSEMBLY__ */
11037
11038
11039 /* Certain architectures need to do special things when
11040  * PTEs within a page table are directly modified. Thus,
11041  * the following hook is made available. */
11042 #define set_pte(pteptr, pteval) ((*pteptr) = (pteval))
11043
11044 /* PMD_SHIFT determines the size of the area a
11045  * second-level page table can map */
11046 #define PMD_SHIFT      22
11047 #define PMD_SIZE      (1UL << PMD_SHIFT)
11048 #define PMD_MASK      ~(PMD_SIZE-1)
11049
11050 /* PGDIR_SHIFT determines what a third-level page table
11051  * entry can map */
11052 #define PGDIR_SHIFT    22
11053 #define PGDIR_SIZE    (1UL << PGDIR_SHIFT)
11054 #define PGDIR_MASK    ~(PGDIR_SIZE-1)
11055
11056 /* entries per page directory level: the i386 is
11057  * two-level, so we don't really have any PMD directory
11058  * physically. */
11059 #define PTRS_PER_PTE   1024
11060 #define PTRS_PER_PMD   1
11061 #define PTRS_PER_PGD   1024
11062 #define USER_PTRS_PER_PGD (TASK_SIZE/PGDIR_SIZE)
11063
11064 /* pgd entries used up by user/kernel: */
11065
11066 #define USER_PGD_PTRS (PAGE_OFFSET >> PGDIR_SHIFT)
11067 #define KERNEL_PGD_PTRS (PTRS_PER_PGD-USER_PGD_PTRS)
11068 #define __USER_PGD_PTRS
11069     ((__PAGE_OFFSET >> PGDIR_SHIFT) & 0x3ff)
11070 #define __KERNEL_PGD_PTRS (PTRS_PER_PGD-__USER_PGD_PTRS)
11071
11072 #ifndef __ASSEMBLY__
11073 /* Just any arbitrary offset to the start of the vmalloc
11074  * VM area: the current 8MB value just means that there
11075  * will be a 8MB "hole" after the physical memory until
11076  * the kernel virtual memory starts. That means that any
11077  * out-of-bounds memory accesses will hopefully be
11078  * caught. The vmalloc() routines leaves a hole of 4kB
11079  * between each vmallocated area for the same reason. :) */
11080 #define VMALLOC_OFFSET (8*1024*1024)
11081 #define VMALLOC_START  (((unsigned long) high_memory + \
11082     VMALLOC_OFFSET) & ~(VMALLOC_OFFSET-1))
11083 #define VMALLOC_VMADDR(x) ((unsigned long)(x))

```

```

11084 #define VMALLOC_END      (FIXADDR_START)
11085
11086 /* The 4MB page is guessing.. Detailed in the infamous
11087 * "Chapter H" of the Pentium details, but assuming intel
11088 * did the straightforward thing, this bit set in the
11089 * page directory entry just means that the page
11090 * directory entry points directly to a 4MB-aligned block
11091 * of memory. */
11092 #define _PAGE_PRESENT      0x001
11093 #define _PAGE_RW           0x002
11094 #define _PAGE_USER         0x004
11095 #define _PAGE_WT           0x008
11096 #define _PAGE_PCD         0x010
11097 #define _PAGE_ACCESSED    0x020
11098 #define _PAGE_DIRTY       0x040
11099 /* 4 MB page. Pentium+, if present.. */
11100 #define _PAGE_4M          0x080
11101 #define _PAGE_GLOBAL      0x100 /* Global TLB entry PPro+*/
11102
11103 #define _PAGE_PROTNONE    0x080 /* If not present */
11104
11105 #define _PAGE_TABLE      (_PAGE_PRESENT | _PAGE_RW      | \
11106                          _PAGE_USER   | _PAGE_ACCESSED | \
11107                          _PAGE_DIRTY)
11108 #define _KERNPG_TABLE   (_PAGE_PRESENT | _PAGE_RW      | \
11109                          _PAGE_ACCESSED | _PAGE_DIRTY)
11110 #define _PAGE_CHG_MASK  (_PAGE_MASK   | _PAGE_ACCESSED | \
11111                          _PAGE_DIRTY)
11112
11113 #define PAGE_NONE       __pgprot(_PAGE_PROTNONE | \
11114                                _PAGE_ACCESSED)
11115 #define PAGE_SHARED     __pgprot(_PAGE_PRESENT | \
11116                                _PAGE_RW       | \
11117                                _PAGE_USER     | \
11118                                _PAGE_ACCESSED)
11119 #define PAGE_COPY       __pgprot(_PAGE_PRESENT | \
11120                                _PAGE_USER     | \
11121                                _PAGE_ACCESSED)
11122 #define PAGE_READONLY   __pgprot(_PAGE_PRESENT | \
11123                                _PAGE_USER     | \
11124                                _PAGE_ACCESSED)
11125 #define PAGE_KERNEL     __pgprot(_PAGE_PRESENT | \
11126                                _PAGE_RW       | \
11127                                _PAGE_DIRTY    | \
11128                                _PAGE_ACCESSED)
11129 #define PAGE_KERNEL_RO  __pgprot(_PAGE_PRESENT | \
11130                                _PAGE_DIRTY    | \
11131                                _PAGE_ACCESSED)

```

```

11132
11133 /* The i386 can't do page protection for execute, and
11134 * considers that the same are read. Also, write
11135 * permissions imply read permissions. This is the
11136 * closest we can get.. */
11137 #define __P000 PAGE_NONE
11138 #define __P001 PAGE_READONLY
11139 #define __P010 PAGE_COPY
11140 #define __P011 PAGE_COPY
11141 #define __P100 PAGE_READONLY
11142 #define __P101 PAGE_READONLY
11143 #define __P110 PAGE_COPY
11144 #define __P111 PAGE_COPY
11145
11146 #define __S000 PAGE_NONE
11147 #define __S001 PAGE_READONLY
11148 #define __S010 PAGE_SHARED
11149 #define __S011 PAGE_SHARED
11150 #define __S100 PAGE_READONLY
11151 #define __S101 PAGE_READONLY
11152 #define __S110 PAGE_SHARED
11153 #define __S111 PAGE_SHARED
11154
11155 /* Define this if things work differently on an i386 and
11156 * an i486: it will (on an i486) warn about kernel memory
11157 * accesses that are done without a
11158 * 'verify_area(VERIFY_WRITE,...)' */
11159 #undef TEST_VERIFY_AREA
11160
11161 /* page table for 0-4MB for everybody */
11162 extern unsigned long pg0[1024];
11163 /* zero page used for uninitialized stuff */
11164 extern unsigned long empty_zero_page[1024];
11165
11166 /* BAD_PAGETABLE is used when we need a bogus page-table.
11167 * while BAD_PAGE is used for a bogus page. ZERO_PAGE is
11168 * a global shared page that is always zero: used for
11169 * zero-mapped memory areas etc.. */
11170 extern pte_t __bad_page(void);
11171 extern pte_t * __bad_pagetable(void);
11172
11173 #define BAD_PAGETABLE __bad_pagetable()
11174 #define BAD_PAGE __bad_page()
11175 #define ZERO_PAGE ((unsigned long) empty_zero_page)
11176
11177 /* number of bits that fit into a memory pointer */
11178 #define BITS_PER_PTR (8*sizeof(unsigned long))
11179

```

```

11180 /* to align the pointer to a pointer address */
11181 #define PTR_MASK (~(sizeof(void*)-1))
11182
11183 /* sizeof(void*)==1<<SIZEOF_PTR_LOG2 */
11184 /* 64-bit machines, beware! SRB. */
11185 #define SIZEOF_PTR_LOG2 2
11186
11187 /* to find an entry in a page-table */
11188 #define PAGE_PTR(address) \
11189 ((unsigned long)(address)>> \
11190 (PAGE_SHIFT-SIZEOF_PTR_LOG2)&PTR_MASK&~PAGE_MASK)
11191
11192 /* to set the page-dir */
11193 #define SET_PAGE_DIR(tsk,pgdir) \
11194 do { \
11195     unsigned long __pgdir = __pa(pgdir); \
11196     (tsk)->tss.cr3 = __pgdir; \
11197     if ((tsk == current) \
11198         __asm__ \
11199         __volatile__("movl %0,%cr3": : "r" (__pgdir)); \
11200 } while (0)
11201
11202 #define pte_none(x) (!pte_val(x))
11203 #define pte_present(x) (pte_val(x) & \
11204 (_PAGE_PRESENT|_PAGE_PROTNONE))
11205 #define pte_clear(xp) do { pte_val(*(xp)) = 0; } while(0)
11206
11207 #define pmd_none(x) (!pmd_val(x))
11208 #define pmd_bad(x) \
11209 ((pmd_val(x) & (~PAGE_MASK & ~PAGE_USER)) != \
11210 _KERNPG_TABLE)
11211 #define pmd_present(x) (pmd_val(x) & _PAGE_PRESENT)
11212 #define pmd_clear(xp) do { pmd_val(*(xp)) = 0; } while(0)
11213
11214 /* The "pgd_xxx()" functions here are trivial for a
11215 * folded two-level setup; the pgd is never bad, and a
11216 * pmd always exists (as it's folded into the pgd entry)
11217 */
11218 extern inline int pgd_none(pgd_t pgd) { return 0; }
11219 extern inline int pgd_bad(pgd_t pgd) { return 0; }
11220 extern inline int pgd_present(pgd_t pgd) { return 1; }
11221 extern inline void pgd_clear(pgd_t * pgdp) { }
11222
11223 /* The following only work if pte_present() is true.
11224 * Undefined behaviour if not.. */
11225 extern inline int pte_read(pte_t pte) \
11226 { return pte_val(pte) & _PAGE_USER; }
11227 extern inline int pte_exec(pte_t pte) \
11228 { return pte_val(pte) & _PAGE_USER; }
11229 extern inline int pte_dirty(pte_t pte) \
11230 { return pte_val(pte) & _PAGE_DIRTY; }
11231 extern inline int pte_young(pte_t pte) \
11232 { return pte_val(pte) & _PAGE_ACCESSED; }
11233 extern inline int pte_write(pte_t pte) \
11234 { return pte_val(pte) & _PAGE_RW; }
11235
11236 extern inline pte_t pte_rdprotect(pte_t pte) \
11237 { pte_val(pte) &= ~_PAGE_USER; return pte; }
11238 extern inline pte_t pte_exprotect(pte_t pte) \
11239 { pte_val(pte) &= ~_PAGE_USER; return pte; }
11240 extern inline pte_t pte_mkclean(pte_t pte) \
11241 { pte_val(pte) &= ~_PAGE_DIRTY; return pte; }
11242 extern inline pte_t pte_mkold(pte_t pte) \
11243 { pte_val(pte) &= ~_PAGE_ACCESSED; return pte; }
11244 extern inline pte_t pte_wrprotect(pte_t pte) \
11245 { pte_val(pte) &= ~_PAGE_RW; return pte; }
11246 extern inline pte_t pte_mkread(pte_t pte) \
11247 { pte_val(pte) |= _PAGE_USER; return pte; }
11248 extern inline pte_t pte_mkexec(pte_t pte) \
11249 { pte_val(pte) |= _PAGE_USER; return pte; }
11250 extern inline pte_t pte_mkdirty(pte_t pte) \
11251 { pte_val(pte) |= _PAGE_DIRTY; return pte; }
11252 extern inline pte_t pte_mkyoung(pte_t pte) \
11253 { pte_val(pte) |= _PAGE_ACCESSED; return pte; }
11254 extern inline pte_t pte_mkwrite(pte_t pte) \
11255 { pte_val(pte) |= _PAGE_RW; return pte; }
11256
11257 /* Conversion functions: convert a page and protection to
11258 * a page entry, and a page entry and page directory to
11259 * the page they refer to. */
11260 #define mk_pte(page, pgprot) \
11261 ({ pte_t __pte; \
11262     pte_val(__pte) = __pa(page) + pgprot_val(pgprot); \
11263     __pte; })
11264
11265 /* This takes a physical page address that is used by the
11266 * remapping functions */
11267 #define mk_pte_phys(phypage, pgprot) \
11268 ({ pte_t __pte; \
11269     pte_val(__pte) = phypage + pgprot_val(pgprot); \
11270     __pte; })
11271
11272 extern inline pte_t pte_modify(pte_t pte, \
11273                               pgprot_t newprot)
11274 { pte_val(pte) = (pte_val(pte) & _PAGE_CHG_MASK) | \
11275     pgprot_val(newprot); return pte; }

```

```

11276
11277 #define pte_page(pte)
11278 ((unsigned long) __va(pte_val(pte) & PAGE_MASK))
11279
11280 #define pmd_page(pmd)
11281 ((unsigned long) __va(pmd_val(pmd) & PAGE_MASK))
11282
11283 /* to find an entry in a page-table-directory */
11284 #define pgd_offset(mm, address)
11285 ((mm)->pgd + ((address) >> PGDIR_SHIFT))
11286
11287 /* to find an entry in a kernel page-table-directory */
11288 #define pgd_offset_k(address)
11289 pgd_offset(&init_mm, address)
11290
11291 /* Find an entry in the second-level page table.. */
11292 extern inline pmd_t * pmd_offset(pgd_t * dir,
11293 unsigned long address)
11294 {
11295     return (pmd_t *) dir;
11296 }
11297
11298 /* Find an entry in the third-level page table.. */
11299 #define pte_offset(pmd, address)
11300 ((pte_t *) (pmd_page(*pmd) + ((address)>>10) &
11301 ((PTRS_PER_PTE-1)<<2)))
11302
11303 /* Allocate and free page tables. The xxx_kernel()
11304 * versions are used to allocate a kernel page table -
11305 * this turns on ASN bits if any. */
11306
11307 #define pgd_quicklist (current_cpu_data.pgd_quick)
11308 #define pmd_quicklist ((unsigned long *)0)
11309 #define pte_quicklist (current_cpu_data.pte_quick)
11310 #define pgtable_cache_size
11311 (current_cpu_data.pgtable_cache_sz)
11312
11313 extern __inline__ pgd_t *get_pgd_slow(void)
11314 {
11315     pgd_t *ret=(pgd_t *)__get_free_page(GFP_KERNEL), *init;
11316
11317     if (ret) {
11318         init = pgd_offset(&init_mm, 0);
11319         memset(ret, 0, USER_PTRS_PER_PGD * sizeof(pgd_t));
11320         memcpy(ret + USER_PTRS_PER_PGD,
11321             init + USER_PTRS_PER_PGD,
11322             (PTRS_PER_PGD - USER_PTRS_PER_PGD) * sizeof(pgd_t));
11323     }

```

```

11324     return ret;
11325 }
11326
11327 extern __inline__ pgd_t *get_pgd_fast(void)
11328 {
11329     unsigned long *ret;
11330
11331     if((ret = pgd_quicklist) != NULL) {
11332         pgd_quicklist = (unsigned long *)(*ret);
11333         ret[0] = ret[1];
11334         pgtable_cache_size--;
11335     } else
11336         ret = (unsigned long *)get_pgd_slow();
11337     return (pgd_t *)ret;
11338 }
11339
11340 extern __inline__ void free_pgd_fast(pgd_t *pgd)
11341 {
11342     *(unsigned long *)pgd - (unsigned long) pgd_quicklist;
11343     pgd_quicklist = (unsigned long *) pgd;
11344     pgtable_cache_size++;
11345 }
11346
11347 extern __inline__ void free_pgd_slow(pgd_t *pgd)
11348 {
11349     free_page((unsigned long)pgd);
11350 }
11351
11352 extern pte_t *get_pte_slow(pmd_t *pmd,
11353 unsigned long address_preadjusted);
11354 extern pte_t *get_pte_kernel_slow(pmd_t *pmd,
11355 unsigned long address_preadjusted);
11356
11357 extern __inline__ pte_t *get_pte_fast(void)
11358 {
11359     unsigned long *ret;
11360
11361     if((ret = (unsigned long *)pte_quicklist) != NULL) {
11362         pte_quicklist = (unsigned long *)(*ret);
11363         ret[0] = ret[1];
11364         pgtable_cache_size--;
11365     }
11366     return (pte_t *)ret;
11367 }
11368
11369 extern __inline__ void free_pte_fast(pte_t *pte)
11370 {
11371     *(unsigned long *)pte = (unsigned long) pte_quicklist;

```

```

11372 pte_quicklist = (unsigned long *) pte;
11373 pgtable_cache_size++;
11374 }
11375
11376 extern __inline__ void free_pte_slow(pte_t *pte)
11377 {
11378     free_page((unsigned long)pte);
11379 }
11380
11381 /* We don't use pmd cache, so these are dummy routines */
11382 extern __inline__ pmd_t *get_pmd_fast(void)
11383 {
11384     return (pmd_t *)0;
11385 }
11386
11387 extern __inline__ void free_pmd_fast(pmd_t *pmd)
11388 {
11389 }
11390
11391 extern __inline__ void free_pmd_slow(pmd_t *pmd)
11392 {
11393 }
11394
11395 extern void __bad_pte(pmd_t *pmd);
11396 extern void __bad_pte_kernel(pmd_t *pmd);
11397
11398 #define pte_free_kernel(pte)    free_pte_fast(pte)
11399 #define pte_free(pte)          free_pte_fast(pte)
11400 #define pgd_free(pgd)          free_pgd_fast(pgd)
11401 #define pgd_alloc()            get_pgd_fast()
11402
11403 extern inline pte_t * pte_alloc_kernel(
11404     pmd_t * pmd, unsigned long address)
11405 {
11406     address = (address >> PAGE_SHIFT) & (PTRS_PER_PTE - 1);
11407     if (pmd_none(*pmd)) {
11408         pte_t * page = (pte_t *) get_pte_fast();
11409
11410         if (!page)
11411             return get_pte_kernel_slow(pmd, address);
11412         pmd_val(*pmd) = _KERNPG_TABLE + __pa(page);
11413         return page + address;
11414     }
11415     if (pmd_bad(*pmd)) {
11416         __bad_pte_kernel(pmd);
11417         return NULL;
11418     }
11419     return (pte_t *) pmd_page(*pmd) + address;

```

```

11420 }
11421
p549 11422 extern inline pte_t * pte_alloc(pmd_t * pmd,
11423     unsigned long address)
11424 {
11425     address = (address >> (PAGE_SHIFT-2)) &
11426         4*(PTRS_PER_PTE - 1);
11427
p560 11428     if (pmd_none(*pmd))
11429         goto getnew;
11430     if (pmd_bad(*pmd))
11431         goto fix;
11432     return (pte_t *) (pmd_page(*pmd) + address);
11433 getnew:
11434 {
11435     unsigned long page = (unsigned long) get_pte_fast();
11436
11437     if (!page)
11438         return get_pte_slow(pmd, address);
11439     pmd_val(*pmd) = _PAGE_TABLE + __pa(page);
11440     return (pte_t *) (page + address);
11441 }
11442 fix:
11443     __bad_pte(pmd);
11444     return NULL;
11445 }
11446
11447 /* allocating and freeing a pmd is trivial: the 1-entry
11448 * pmd is inside the pgd, so has no extra memory
11449 * associated with it. */
11450 extern inline void pmd_free(pmd_t * pmd)
11451 {
11452 }
11453
p549 11454 extern inline pmd_t * pmd_alloc(pgd_t * pgd,
11455     unsigned long address)
11456 {
11457     return (pmd_t *) pgd;
11458 }
11459
11460 #define pmd_free_kernel        pmd_free
11461 #define pmd_alloc_kernel      pmd_alloc
11462
11463 extern int do_check_pgt_cache(int, int);
11464
11465 extern inline void set_pgdir(unsigned long address,
11466     pgd_t entry)
11467 {

```

```

11468 struct task_struct * p;
11469 pgd_t *pgd;
11470 #ifdef __SMP__
11471 int i;
11472 #endif
11473
11474 read_lock(&tasklist_lock);
11475 for_each_task(p) {
11476     if (!p->mm)
11477         continue;
11478     *pgd_offset(p->mm, address) = entry;
11479 }
11480 read_unlock(&tasklist_lock);
11481 #ifndef __SMP__
11482 for (pgd = (pgd_t *)pgd_quicklist; pgd;
11483      pgd = (pgd_t *)*(unsigned long *)pgd)
11484     pgd[address >> PGDIR_SHIFT] = entry;
11485 #else
11486 /* To pgd_alloc/pgd_free, one holds master kernel lock
11487  * and so does our callee, so we can modify pgd caches
11488  * of other CPUs as well. -jj */
11489 for (i = 0; i < NR_CPUS; i++)
11490     for (pgd = (pgd_t *)cpu_data[i].pgd_quick; pgd;
11491          pgd = (pgd_t *)*(unsigned long *)pgd)
11492         pgd[address >> PGDIR_SHIFT] = entry;
11493 #endif
11494 }
11495
11496 extern pgd_t swapper_pg_dir[1024];
11497
11498 /* The i386 doesn't have any external MMU info: the
11499  * kernel page tables contain all the necessary
11500  * information. */
11501 extern inline void update_mmu_cache(
11502     struct vm_area_struct * vma,
11503     unsigned long address, pte_t pte)
11504 {
11505 }
11506
11507 #define SWP_TYPE(entry) (((entry) >> 1) & 0x3f)
11508 #define SWP_OFFSET(entry) ((entry) >> 8)
11509 #define SWP_ENTRY(type, offset)
11510     (((type) << 1) | ((offset) << 8))
11511
11512 #define module_map      vmalloc
11513 #define module_unmap    vfree
11514
11515 #endif /* !_ASSEMBLY__ */

```

```

11516
11517 /* Needs to be defined here and not in linux/mm.h, as it
11518  * is arch dependent */
11519 #define PageSkip(page)      (0)
11520 #define kern_addr_valid(addr) (1)
11521
11522 #endif /* _I386_PAGE_H */

```

include/asm-i386/ptrace.h

```

11523 #ifndef _I386_PTRACE_H
11524 #define _I386_PTRACE_H
11525
11526 #define EBX 0
11527 #define ECX 1
11528 #define EDX 2
11529 #define ESI 3
11530 #define EDI 4
11531 #define EBP 5
11532 #define EAX 6
11533 #define DS 7
11534 #define ES 8
11535 #define FS 9
11536 #define GS 10
11537 #define ORIG_EAX 11
11538 #define EIP 12
11539 #define CS 13
11540 #define EFL 14
11541 #define UESP 15
11542 #define SS 16
11543
11544 /* this struct defines the way the registers are stored
11545  * on the stack during a system call. */
11546 struct pt_regs {
11547     long ebx;
11548     long ecx;
11549     long edx;
11550     long esi;
11551     long edi;
11552     long ebp;
11553     long eax;
11554     int xds;
11555     int xes;
11556     long orig_eax;
11557     long eip;
11558     int xcs;
11559     long eflags;
11560     long esp;

```

```

11561 int xss;
11562 };
11563
11564 /* Arbitrarily choose the same ptrace numbers as used by
11565  * the Sparc code. */
11566 #define PTRACE_GETREGS          12
11567 #define PTRACE_SETREGS          13
11568 #define PTRACE_GETFPREGS       14
11569 #define PTRACE_SETFPREGS       15
11570
11571 #ifdef __KERNEL__
11572 #define user_mode(regs)
11573     ((VM_MASK & (regs)->eflags) || (3 & (regs)->xcs))
11574 #define instruction_pointer(regs) ((regs)->eip)
11575 extern void show_regs(struct pt_regs *);
11576 #endif
11577
11578 #endif

```

include/asm-i386/semaphore.h

```

11579 #ifndef _I386_SEMAPHORE_H
11580 #define _I386_SEMAPHORE_H
11581
11582 #include <linux/linkage.h>
11583
11584 /*
11585  * SMP- and interrupt-safe semaphores..
11586  *
11587  * (C) Copyright 1996 Linus Torvalds
11588  *
11589  * Modified 1996-12-23 by Dave Grothe <dave@gcom.com> to
11590  * fix bugs in the original code and to make semaphore
11591  * waits interruptible so that processes waiting on
11592  * semaphores can be killed.
11593  * Modified 1999-02-14 by Andrea Arcangeli, split the
11594  * sched.c helper functions in asm/semaphore-helper.h
11595  * while fixing a potential and subtle race discovered by
11596  * Ulrich Schmid in down_interruptible(). Since I started
11597  * to play here I also implemented the 'trylock'
11598  * semaphore operation.
11599  *
11600  * If you would like to see an analysis of this
11601  * implementation, please ftp to gcom.com and download
11602  * the file
11603  * /pub/linux/src/semaphore/semaphore-2.0.24.tar.gz. */
11604
11605 #include <asm/system.h>

```

```

11606 #include <asm/atomic.h>
11607 #include <asm/spinlock.h>
11608
11609 struct semaphore {
11610     atomic_t count;
11611     int waking;
11612     struct wait_queue * wait;
11613 };
11614
11615 #define MUTEX
11616 ((struct semaphore) { ATOMIC_INIT(1), 0, NULL })
11617 #define MUTEX_LOCKED
11618 ((struct semaphore) { ATOMIC_INIT(0), 0, NULL })
11619
11620 asmlinkage void
11621 __down_failed(void /* special reg calling convention */);
11622 asmlinkage int
11623 ___down_failed_interruptible(void /* params in regs */);
11624 asmlinkage int
11625 __down_failed_trylock(void /* params in registers */);
11626 asmlinkage void
11627 __up_wakeup(void /* special reg calling convention */);
11628
11629 asmlinkage void __down(struct semaphore * sem);
11630 asmlinkage int __down_interruptible(
11631     struct semaphore * sem);
11632 asmlinkage int __down_trylock(struct semaphore * sem);
11633 asmlinkage void __up(struct semaphore * sem);
11634
11635 extern spinlock_t semaphore_wake_lock;
11636
11637 #define sema_init(sem, val)
11638     atomic_set(&((sem)->count), (val))
11639
11640 /* This is ugly, but we want the default case to fall
11641  * through. "down_failed" is a special asm handler that
11642  * calls the C routine that actually waits. See
11643  * arch/i386/lib/semaphore.S */
11644 extern inline void down(struct semaphore * sem)
11645 {
11646     __asm__ __volatile__(
11647         "# atomic down operation\n\t"
11648     #ifdef __SMP__
11649         "lock ; "
11650     #endif
11651         "decl 0(%%0)\n\t"
11652         "js 2f\n"
11653         "1:\n"

```

PG01

```

11654     ".section .text.lock,\"ax\"\n"
11655     "2:\tpushl $1b\n\t"
11656     "jmp __down_failed\n"
11657     ".previous"
11658     /* no outputs */
11659     : "c" (sem)
11660     : "memory");
11661 }
11662
11663 extern inline int
11664 down_interruptible(struct semaphore * sem)
11665 {
11666     int result;
11667
11668     __asm__ __volatile__(
11669         "# atomic interruptible down operation\n\t"
11670 #ifdef __SMP__
11671         "lock ; "
11672 #endif
11673         "decl 0(%1)\n\t"
11674         "js 2f\n\t"
11675         "xorl %0,%0\n"
11676         "1:\n"
11677         ".section .text.lock,\"ax\"\n"
11678         "2:\tpushl $1b\n\t"
11679         "jmp __down_failed_interruptible\n"
11680         ".previous"
11681         : "=a" (result)
11682         : "c" (sem)
11683         : "memory");
11684     return result;
11685 }
11686
11687 extern inline int down_trylock(struct semaphore * sem)
11688 {
11689     int result;
11690
11691     __asm__ __volatile__(
11692         "# atomic interruptible down operation\n\t"
11693 #ifdef __SMP__
11694         "lock ; "
11695 #endif
11696         "decl 0(%1)\n\t"
11697         "js 2f\n\t"
11698         "xorl %0,%0\n"
11699         "1:\n"
11700         ".section .text.lock,\"ax\"\n"
11701         "2:\tpushl $1b\n\t"

```

```

11702         "jmp __down_failed_trylock\n"
11703         ".previous"
11704         : "=a" (result)
11705         : "c" (sem)
11706         : "memory");
11707     return result;
11708 }
11709
11710 /* Note! This is subtle. We jump to wake people up only
11711 * if the semaphore was negative (== somebody was waiting
11712 * on it). The default case (no contention) will result
11713 * in NO jumps for both down() and up(). */
11714 extern inline void up(struct semaphore * sem)
11715 {
11716     __asm__ __volatile__(
11717         "# atomic up operation\n\t"
11718 #ifdef __SMP__
11719         "lock ; "
11720 #endif
11721         "incl 0(%0)\n\t"
11722         "jle 2f\n"
11723         "1:\n"
11724         ".section .text.lock,\"ax\"\n"
11725         "2:\tpushl $1b\n\t"
11726         "jmp __up_wakeup\n"
11727         ".previous"
11728         /* no outputs */
11729         : "c" (sem)
11730         : "memory");
11731 }
11732
11733 #endif

```

include/asm-i386/shmparam.h

```

11734 #ifndef _ASMI386_SHMPARAM_H
11735 #define _ASMI386_SHMPARAM_H
11736
11737 /* address range for shared memory attaches if no address
11738 * passed to shmat() */
11739 #define SHM_RANGE_START 0x50000000
11740 #define SHM_RANGE_END 0x60000000
11741
11742 /* Format of a swap-entry for shared memory pages
11743 * currently out in swap space (see also mm/swap.c).
11744 *
11745 * SWP_TYPE = SHM_SWP_TYPE
11746 * SWP_OFFSET is used as follows:

```



```

11747 * bits 0..6 : id of shared memory segment page belongs
11748 * to (SHM_ID)
11749 * bits 7..21: index of page within shared memory
11750 * segment (SHM_IDX) (actually fewer bits get used since
11751 * SHMMAX is so low) */
11752
11753 /* Keep _SHM_ID_BITS as low as possible since SHMMNI
11754 * depends on it and there is a static array of size
11755 * SHMMNI. */
11756 #define _SHM_ID_BITS 7
11757 #define SHM_ID_MASK ((1<<_SHM_ID_BITS)-1)
11758
11759 #define SHM_IDX_SHIFT (_SHM_ID_BITS)
11760 #define _SHM_IDX_BITS 15
11761 #define SHM_IDX_MASK ((1<<_SHM_IDX_BITS)-1)
11762
11763 /* _SHM_ID_BITS + _SHM_IDX_BITS must be <= 24 on the i386
11764 * and SHMMAX <= (PAGE_SIZE << _SHM_IDX_BITS). */
11765
11766 #define SHMMAX 0x2000000 /* max shared seg size (bytes)*/
11767 /* Try not to change the default shipped SHMMAX - people
11768 * rely on it */
11769
11770 /* min shared seg size (bytes) */
11771 #define SHMMIN 1 /* really PAGE_SIZE */
11772 /* max num of segs system wide */
11773 #define SHMMNI (1<<_SHM_ID_BITS)
11774 /* max shm system wide (pages) */
11775 #define SHMALL (1<<(_SHM_IDX_BITS+_SHM_ID_BITS))
11776 /* attach addr a multiple of this */
11777 #define SHMLBA PAGE_SIZE
11778 /* max shared segs per process */
11779 #define SHMSEG SHMMNI
11780
11781 #endif /* _ASMI386_SHMPARAM_H */

```

include/asm-i386/sigcontext.h

```

11782 #ifndef _ASMI386_SIGCONTEXT_H
11783 #define _ASMI386_SIGCONTEXT_H
11784
11785 /* As documented in the iBCS2 standard..
11786 *
11787 * The first part of "struct _fpstate" is just the normal
11788 * i387 hardware setup, the extra "status" word is used
11789 * to save the coprocessor status word before entering
11790 * the handler. */
11791 struct _fpreg {

```

```

11792 unsigned short significand[4];
11793 unsigned short exponent;
11794 };
11795
11796 struct _fpstate {
11797     unsigned long cw,
11798     sw,
11799     tag,
11800     ipoff,
11801     cssel,
11802     dataoff,
11803     datasel;
11804     struct _fpreg _st[8];
11805     unsigned long status;
11806 };
11807
11808 struct sigcontext {
11809     unsigned short gs, __gsh;
11810     unsigned short fs, __fsh;
11811     unsigned short es, __esh;
11812     unsigned short ds, __dsh;
11813     unsigned long edi;
11814     unsigned long esi;
11815     unsigned long ebp;
11816     unsigned long esp;
11817     unsigned long ebx;
11818     unsigned long edx;
11819     unsigned long ecx;
11820     unsigned long eax;
11821     unsigned long trapno;
11822     unsigned long err;
11823     unsigned long eip;
11824     unsigned short cs, __csh;
11825     unsigned long eflags;
11826     unsigned long esp_at_signal;
11827     unsigned short ss, __ssh;
11828     struct _fpstate * fpstate;
11829     unsigned long oldmask;
11830     unsigned long cr2;
11831 };
11832
11833
11834 #endif

```

include/asm-i386/siginfo.h

```

11835 #ifndef I386_SIGINFO_H
11836 #define _I386_SIGINFO_H

```

```

11837
11838 #include <linux/types.h>
11839
11840 /* XXX: This structure was copied from the Alpha; is
11841  * there an iBCS version? */
11842
11843 typedef union sigval {
11844     int sival_int;
11845     void *sival_ptr;
11846 } sigval_t;
11847
11848 #define SI_MAX_SIZE      128
11849 #define SI_PAD_SIZE     ((SI_MAX_SIZE/sizeof(int)) - 3)
11850
11851 typedef struct siginfo {
11852     int si_signo;
11853     int si_errno;
11854     int si_code;
11855
11856     union {
11857         int _pad[SI_PAD_SIZE];
11858
11859         /* kill() */
11860         struct {
11861             pid_t _pid;           /* sender's pid */
11862             uid_t _uid;          /* sender's uid */
11863         } _kill;
11864
11865         /* POSIX.1b timers */
11866         struct {
11867             unsigned int _timer1;
11868             unsigned int _timer2;
11869         } _timer;
11870
11871         /* POSIX.1b signals */
11872         struct {
11873             pid_t _pid;           /* sender's pid */
11874             uid_t _uid;          /* sender's uid */
11875             sigval_t _sigval;
11876         } _rt;
11877
11878         /* SIGCHLD */
11879         struct {
11880             pid_t _pid;           /* which child */
11881             uid_t _uid;          /* sender's uid */
11882             int _status;         /* exit code */
11883             clock_t _utime;
11884             clock_t _stime;

```

```

11885     } _sigchld;
11886
11887     /* SIGILL, SIGFPE, SIGSEGV, SIGBUS */
11888     struct {
11889         void *_addr; /* faulting insn/memory ref. */
11890     } _sigfault;
11891
11892     /* SIGPOLL */
11893     struct {
11894         int _band;           /* POLL_IN, POLL_OUT, POLL_MSG */
11895         int _fd;
11896     } _sigpoll;
11897     } _sifields;
11898 } siginfo_t;
11899
11900 /* How these fields are to be accessed. */
11901 #define si_pid      _sifields._kill._pid
11902 #define si_uid      _sifields._kill._uid
11903 #define si_status   _sifields._sigchld._status
11904 #define si_utime    _sifields._sigchld._utime
11905 #define si_stime    _sifields._sigchld._stime
11906 #define si_value    _sifields._rt._sigval
11907 #define si_int      _sifields._rt._sigval.sival_int
11908 #define si_ptr      _sifields._rt._sigval.sival_ptr
11909 #define si_addr     _sifields._sigfault._addr
11910 #define si_band     _sifields._sigpoll._band
11911 #define si_fd       _sifields._sigpoll._fd
11912
11913 /* si_code values Digital reserves positive values for
11914  * kernel-generated signals. */
11915 #define SI_USER      0 /* sent by kill/sigsend/raise */
11916 #define SI_KERNEL   0x80 /* sent by the kernel */
11917 #define SI_QUEUE    -1 /* sent by sigqueue */
11918 #define SI_TIMER    -2 /* sent by timer expiration */
11919 #define SI_MESGQ    -3 /* sent by RT mesq state chg */
11920 #define SI_ASYNCIO  -4 /* sent by AIO completion */
11921 #define SI_SIGIO    -5 /* sent by queued SIGIO */
11922
11923 #define SI_FROMUSER(siptr) ((siptr)->si_code <= 0)
11924 #define SI_FROMKERNEL(siptr) ((siptr)->si_code > 0)
11925
11926 /* SIGILL si_codes */
11927 #define ILL_ILLOPC  1 /* illegal opcode */
11928 #define ILL_ILLOPN  2 /* illegal operand */
11929 #define ILL_ILLADR  3 /* illegal addressing mode */
11930 #define ILL_ILLTRP  4 /* illegal trap */
11931 #define ILL_PRVOPC  5 /* privileged opcode */
11932 #define ILL_PRVREG  6 /* privileged register */

```

```

11933 #define ILL_COPROC      7 /* coprocessor error */
11934 #define ILL_BADSTK      8 /* internal stack error */
11935 #define NSIGILL          8
11936
11937 /* SIGFPE si_codes */
11938 #define FPE_INTDIV      1 /* integer divide by zero */
11939 #define FPE_INTOVF      2 /* integer overflow */
11940 #define FPE_FLTDIV      3 /* floating point divide by zero */
11941 #define FPE_FLTOVF      4 /* floating point overflow */
11942 #define FPE_FLTUND      5 /* floating point underflow */
11943 #define FPE_FLTRES      6 /* floating point inexact result */
11944 #define FPE_FLTINV      7 /* floating point invalid op */
11945 #define FPE_FLTSUB      8 /* subscript out of range */
11946 #define NSIGFPE         8
11947
11948 /* SIGSEGV si_codes */
11949 #define SEGV_MAPERR      1 /* address not mapped to object */
11950 #define SEGV_ACCERR      2 /* invalid perms for mapped obj */
11951 #define NSIGSEGV        2
11952
11953 /* SIGBUS si_codes */
11954 #define BUS_ADRALN      1 /* invalid address alignment */
11955 #define BUS_ADRERR      2 /* non-existent physical address */
11956 #define BUS_OBJERR      3 /* object specific hardware error */
11957 #define NSIGBUS         3
11958
11959 /* SIGTRAP si_codes */
11960 #define TRAP_BRKPT      1 /* process breakpoint */
11961 #define TRAP_TRACE      2 /* process trace trap */
11962 #define NSIGTRAP        2
11963
11964 /* SIGCHLD si_codes */
11965 #define CLD_EXITED      1 /* child has exited */
11966 #define CLD_KILLED      2 /* child was killed */
11967 #define CLD_DUMPED      3 /* child terminated abnormally */
11968 #define CLD_TRAPPED     4 /* traced child has trapped */
11969 #define CLD_STOPPED     5 /* child has stopped */
11970 #define CLD_CONTINUED   6 /* stopped child has continued */
11971 #define NSIGCHLD        6
11972
11973 /* SIGPOLL si_codes */
11974 #define POLL_IN          1 /* data input available */
11975 #define POLL_OUT        2 /* output buffers available */
11976 #define POLL_MSG        3 /* input message available */
11977 #define POLL_ERR        4 /* i/o error */
11978 #define POLL_PRI        5 /* high-pri input available */
11979 #define POLL_HUP        6 /* device disconnected */
11980 #define NSIGPOLL        6

```

```

11981
11982 /* sigevent definitions
11983 *
11984 * It seems likely that SIGEV_THREAD will have to be
11985 * handled from userspace, libpthread transmuted it to
11986 * SIGEV_SIGNAL, which the thread manager then catches
11987 * and does the appropriate nonsense. However,
11988 * everything is written out here so as to not get lost.
11989 */
11990 #define SIGEV_SIGNAL 0 /* notify via signal */
11991 #define SIGEV_NONE 1 /* other notif: meaningless */
11992 #define SIGEV_THREAD 2 /* deliver via thread creation */
11993
11994 #define SIGEV_MAX_SIZE 64
11995 #define SIGEV_PAD_SIZE ((SIGEV_MAX_SIZE/sizeof(int)) - 3)
11996
11997 typedef struct sigevent {
11998     sigval_t sigev_value;
11999     int sigev_signo;
12000     int sigev_notify;
12001     union {
12002         int _pad[SIGEV_PAD_SIZE];
12003
12004         struct {
12005             void (*function)(sigval_t);
12006             void *_attribute; /* really pthread_attr_t */
12007             pthread_t _sigev_thread;
12008         } _sigev_un;
12009     } sigevent_t;
12010
12011 #define sigev_notify_function \
12012     _sigev_un._sigev_thread._function \
12013 #define sigev_notify_attributes \
12014     _sigev_un._sigev_thread._attribute
12015
12016 #endif

```

include/asm-i386/signal.h

```

12017 #ifndef _ASMi386_SIGNAL_H
12018 #define _ASMi386_SIGNAL_H
12019
12020 #include <linux/types.h>
12021
12022 /* Avoid too many header ordering problems. */
12023 struct siginfo;
12024
12025 #ifdef _KERNEL_

```

```

12026 /* Most things should be clean enough to redefine this at
12027 * will, if care is taken to make libc match. */
12028
12029 #define _NSIG          64
12030 #define _NSIG_BPW      32
12031 #define _NSIG_WORDS    (_NSIG / _NSIG_BPW)
12032
12033 typedef unsigned long old_sigset_t; /* at least 32 bits*/
12034
12035 typedef struct {
12036     unsigned long sig[_NSIG_WORDS];
12037 } sigset_t;
12038
12039 #else
12040 /* Here we must cater to libcs that poke about in kernel
12041 * headers. */
12042
12043 #define NSIG            32
12044 typedef unsigned long sigset_t;
12045
12046 #endif /* __KERNEL__ */
12047
12048 #define SIGHUP          1
12049 #define SIGINT          2
12050 #define SIGQUIT        3
12051 #define SIGILL         4
12052 #define SIGTRAP        5
12053 #define SIGABRT        6
12054 #define SIGIOT         6
12055 #define SIGBUS         7
12056 #define SIGFPE         8
12057 #define SIGKILL        9
12058 #define SIGUSR1        10
12059 #define SIGSEGV        11
12060 #define SIGUSR2        12
12061 #define SIGPIPE        13
12062 #define SIGALRM        14
12063 #define SIGTERM        15
12064 #define SIGSTKFLT     16
12065 #define SIGCHLD        17
12066 #define SIGCONT        18
12067 #define SIGSTOP        19
12068 #define SIGTSTP        20
12069 #define SIGTTIN        21
12070 #define SIGTTOU        22
12071 #define SIGURG         23
12072 #define SIGXCPU        24
12073 #define SIGXFSZ        25

```

```

12074 #define SIGVTALRM      26
12075 #define SIGPROF        27
12076 #define SIGWINCH      28
12077 #define SIGIO          29
12078 #define SIGPOLL        SIGIO
12079 /*
12080 #define SIGLOST        29
12081 */
12082 #define SIGPWR         30
12083 #define SIGUNUSED     31
12084
12085 /* These should not be considered constants from
12086 * userland. */
12087 #define SIGRTMIN       32
12088 #define SIGRTMAX       (_NSIG-1)
12089
12090 /* SA_FLAGS values:
12091 *
12092 * SA_ONSTACK indicates that a registered stack_t will be
12093 * used.
12094 * SA_INTERRUPT is a no-op, but left due to historical
12095 * reasons. Use the
12096 * SA_RESTART flag to get restarting signals (which were
12097 * the default long ago)
12098 * SA_NOCLDSTOP flag to turn off SIGCHLD when children
12099 * stop.
12100 * SA_RESETHAND clears the handler when the signal is
12101 * delivered.
12102 * SA_NOCLDWAIT flag on SIGCHLD to inhibit zombies.
12103 * SA_NODEFER prevents the current signal from being
12104 * masked in the handler.
12105 * SA_ONESHOT and SA_NOMASK are the historical Linux
12106 * names for the Single Unix names RESETHAND and NODEFER
12107 * respectively. */
12108 #define SA_NOCLDSTOP   0x00000001
12109 #define SA_NOCLDWAIT   0x00000002 /* not supported yet*/
12110 #define SA_SIGINFO     0x00000004
12111 #define SA_ONSTACK     0x08000000
12112 #define SA_RESTART     0x10000000
12113 #define SA_NODEFER     0x40000000
12114 #define SA_RESETHAND   0x80000000
12115
12116 #define SA_NOMASK      SA_NODEFER
12117 #define SA_ONESHOT     SA_RESETHAND
12118 #define SA_INTERRUPT   0x20000000 /* dummy -- ignored */
12119
12120 #define SA_RESTORER    0x04000000
12121

```

```

12122 /* sigaltstack controls */
12123 #define SS_ONSTACK 1
12124 #define SS_DISABLE 2
12125
12126 #define MINSIGSTKSZ 2048
12127 #define SIGSTKSZ 8192
12128
12129 #ifdef __KERNEL__
12130
12131 /* These values of sa_flags are used only by the kernel
12132 * as part of the irq handling routines.
12133 *
12134 * SA_INTERRUPT is also used by the irq handling
12135 * routines.
12136 * SA_SHIRQ is for shared interrupt support on PCI and
12137 * EISA. */
12138 #define SA_PROBE SA_ONESHOT
12139 #define SA_SAMPLE_RANDOM SA_RESTART
12140 #define SA_SHIRQ 0x04000000
12141 #endif
12142
12143 #define SIG_BLOCK 0 /* for blocking signals */
12144 #define SIG_UNBLOCK 1 /* for unblocking signals */
12145 #define SIG_SETMASK 2 /* for setting the signal mask */
12146
12147 /* Type of a signal handler. */
12148 typedef void (*__sighandler_t)(int);
12149
12150 /* default signal handling */
12151 #define SIG_DFL ((__sighandler_t)0)
12152 /* ignore signal */
12153 #define SIG_IGN ((__sighandler_t)1)
12154 /* error return from signal */
12155 #define SIG_ERR ((__sighandler_t)-1)
12156
12157 #ifdef __KERNEL__
12158 struct old_sigaction {
12159     __sighandler_t sa_handler;
12160     old_sigset_t sa_mask;
12161     unsigned long sa_flags;
12162     void (*sa_restorer)(void);
12163 };
12164
12165 struct sigaction {
12166     __sighandler_t sa_handler;
12167     unsigned long sa_flags;
12168     void (*sa_restorer)(void);
12169     sigset_t sa_mask; /* mask last for extensibility */

```

```

12170 };
12171
12172 struct k_sigaction {
12173     struct sigaction sa;
12174 };
12175 #else
12176 /* Here we must cater to libcs that poke about in kernel
12177 * headers. */
12178 struct sigaction {
12179     union {
12180         __sighandler_t _sa_handler;
12181         void (*_sa_sigaction)(int, struct siginfo *, void *);
12182     } _u;
12183     sigset_t sa_mask;
12184     unsigned long sa_flags;
12185     void (*sa_restorer)(void);
12186 };
12187
12188 #define sa_handler _u._sa_handler
12189 #define sa_sigaction _u._sa_sigaction
12190
12191 #endif /* __KERNEL__ */
12192
12193 typedef struct sigaltstack {
12194     void *ss_sp;
12195     int ss_flags;
12196     size_t ss_size;
12197 } stack_t;
12198
12199 #ifdef __KERNEL__
12200 #include <asm/sigcontext.h>
12201
12202 #define __HAVE_ARCH_SIG_BITOPS
12203
12204 extern __inline__ void sigaddset(sigset_t *set, int _sig)
12205 {
12206     __asm__("btsl %1,%0" : "=m"(*set) : "ir"(_sig - 1)
12207           : "cc");
12208 }
12209
12210 extern __inline__ void sigdelset(sigset_t *set, int _sig)
12211 {
12212     __asm__("btrl %1,%0" : "=m"(*set) : "ir"(_sig - 1)
12213           : "cc");
12214 }
12215
12216 extern __inline__ int __const_sigismember(sigset_t *set,
12217     int _sig)

```

```

12218 {
12219     unsigned long sig = _sig - 1;
12220     return 1 & (set->sig[sig / _NSIG_BPW] >>
12221             (sig % _NSIG_BPW));
12222 }
12223
12224 extern __inline__ int __gen_sigismember(sigset_t *set,
12225                                         int _sig)
12226 {
12227     int ret;
12228     __asm__("btl %2,%1\n\t;sbbl %0,%0"
12229           : "=r"(ret) : "m"(*set), "ir"(_sig-1) : "cc");
12230     return ret;
12231 }
12232
12233 #define sigismember(set,sig)          \
12234     (__builtin_constant_p(sig) ?    \
12235      __const_sigismember((set),(sig)) : \
12236      __gen_sigismember((set),(sig)))
12237
12238 #define sigmask(sig)    (1UL << ((sig) - 1))
12239
12240 extern __inline__ int sigfindinword(unsigned long word)
12241 {
12242     __asm__("bsfl %1,%0" : "=r"(word) : "rm"(word) : "cc");
12243     return word;
12244 }
12245
12246 #endif /* __KERNEL__ */
12247
12248 #endif

include/asm-i386/smp.h
12249 #ifndef __ASM_SMP_H
12250 #define __ASM_SMP_H
12251
12252 /* We need the APIC definitions automatically as part of
12253  * 'smp.h' */
12254 #include <linux/config.h>
12255 #ifdef CONFIG_X86_LOCAL_APIC
12256 #ifndef ASSEMBLY
12257 #include <asm/fixmap.h>
12258 #include <asm/i82489.h>
12259 #include <asm/bitops.h>
12260 #endif
12261 #endif
12262

```

```

12263 #ifdef __SMP__
12264 #ifndef ASSEMBLY
12265
12266 #include <linux/tasks.h>
12267 #include <linux/ptrace.h>
12268
12269 /* Support definitions for SMP machines following
12270  * the intel multiprocessing specification */
12271
12272 /* This tag identifies where the SMP configuration
12273  * information is. */
12274
12275 #define SMP_MAGIC_IDENT
12276     (('_'<<24)|('P'<<16)|('M'<<8)|'_')
12277
12278 struct intel_mp_floating
12279 {
12280     char mpf_signature[4]; /* "_MP_" */
12281     unsigned long mpf_physptr; /* Config table address */
12282     unsigned char mpf_length; /* Our len (paragraphs) */
12283     unsigned char mpf_specification; /* Spec version */
12284     unsigned char mpf_checksum; /* Checksum (makes sum 0) */
12285     unsigned char mpf_feature1; /* Std or configuration? */
12286     unsigned char mpf_feature2; /* Bit7 set for IMCR|PIC */
12287     unsigned char mpf_feature3; /* Unused (0) */
12288     unsigned char mpf_feature4; /* Unused (0) */
12289     unsigned char mpf_feature5; /* Unused (0) */
12290 };
12291
12292 struct mp_config_table
12293 {
12294     char mpc_signature[4];
12295     #define MPC_SIGNATURE "PCMP"
12296     unsigned short mpc_length; /* Size of table */
12297     char mpc_spec; /* 0x01 */
12298     char mpc_checksum;
12299     char mpc_oem[8];
12300     char mpc_productid[12];
12301     unsigned long mpc_oemptr; /* 0 if not present */
12302     unsigned short mpc_oemsize; /* 0 if not present */
12303     unsigned short mpc_oemcount;
12304     unsigned long mpc_lapic; /* APIC address */
12305     unsigned long reserved;
12306 };
12307
12308 /* Followed by entries */
12309
12310 #define MP_PROCESSOR 0

```

```

12311 #define MP_BUS          1
12312 #define MP_IOAPIC      2
12313 #define MP_INTSRC      3
12314 #define MP_LINTSRC     4
12315
12316 struct mpc_config_processor
12317 {
12318     unsigned char mpc_type;
12319     unsigned char mpc_apicid;        /* Local APIC number */
12320     unsigned char mpc_apicver;      /* Its versions */
12321     unsigned char mpc_cpuflag;
12322 #define CPU_ENABLED      1 /* Processor is available */
12323 #define CPU_BOOTPROCESSOR 2 /* Processor is the BP */
12324     unsigned long mpc_cpufeature;
12325 #define CPU_STEPPING_MASK 0x0F
12326 #define CPU_MODEL_MASK 0xF0
12327 #define CPU_FAMILY_MASK 0xF00
12328     unsigned long mpc_featureflag; /* CPUID feature value*/
12329     unsigned long mpc_reserved[2];
12330 };
12331
12332 struct mpc_config_bus
12333 {
12334     unsigned char mpc_type;
12335     unsigned char mpc_busid;
12336     unsigned char mpc_bustype[6] __attribute__((packed));
12337 };
12338
12339 #define BUSTYPE_EISA      "EISA"
12340 #define BUSTYPE_ISA      "ISA"
12341 #define BUSTYPE_INTERN   "INTERN" /* Internal BUS */
12342 #define BUSTYPE_MCA      "MCA"
12343 #define BUSTYPE_VL       "VL"     /* Local bus */
12344 #define BUSTYPE_PCI      "PCI"
12345 #define BUSTYPE_PCMCIA   "PCMCIA"
12346
12347 /* We don't understand the others */
12348
12349 struct mpc_config_ioapic
12350 {
12351     unsigned char mpc_type;
12352     unsigned char mpc_apicid;
12353     unsigned char mpc_apicver;
12354     unsigned char mpc_flags;
12355 #define MPC_APIC_USABLE   0x01
12356     unsigned long mpc_apicaddr;
12357 };
12358

```

```

12359 struct mpc_config_intsrc
12360 {
12361     unsigned char mpc_type;
12362     unsigned char mpc_irqtype;
12363     unsigned short mpc_irqflag;
12364     unsigned char mpc_srcbus;
12365     unsigned char mpc_srcbusirq;
12366     unsigned char mpc_dstapic;
12367     unsigned char mpc_dstirq;
12368 };
12369
12370 #define MP_INT_VECTORED  0
12371 #define MP_INT_NMI       1
12372 #define MP_INT_SMI       2
12373 #define MP_INT_EXTINT    3
12374
12375 #define MP_IRODIR_DEFAULT 0
12376 #define MP_IRODIR_HIGH   1
12377 #define MP_IRODIR_LOW    3
12378
12379
12380 struct mpc_config_intlocal
12381 {
12382     unsigned char mpc_type;
12383     unsigned char mpc_irqtype;
12384     unsigned short mpc_irqflag;
12385     unsigned char mpc_srcbusid;
12386     unsigned char mpc_srcbusirq;
12387     unsigned char mpc_destapic;
12388 #define MP_APIC_ALL      0xFF
12389     unsigned char mpc_destapiclint;
12390 };
12391
12392
12393 /*      Default configurations
12394 *
12395 *      1   2 CPU ISA 82489DX
12396 *      2   2 CPU EISA 82489DX no IRQ 8 or timer chaining
12397 *      3   2 CPU EISA 82489DX
12398 *      4   2 CPU MCA 82489DX
12399 *      5   2 CPU ISA+PCI
12400 *      6   2 CPU EISA+PCI
12401 *      7   2 CPU MCA+PCI */
12402
12403 /* Private routines/data */
12404
12405 extern int smp_found config;
12406 extern void init smp config(void);

```

```

12407 extern unsigned long smp_alloc_memory(
12408     unsigned long mem_base);
12409 extern unsigned char boot_cpu_id;
12410 extern unsigned long cpu_present_map;
12411 extern unsigned long cpu_online_map;
12412 extern volatile int cpu_number_map[NR_CPUS];
12413 extern volatile unsigned long smp_invalidate_needed;
12414 extern void smp_flush_tlb(void);
12415
12416 extern volatile unsigned long cpu_callin_map[NR_CPUS];
12417 extern void smp_message_irq(int cpl, void *dev_id,
12418     struct pt_regs *regs);
12419 extern void smp_send_reschedule(int cpu);
12420 extern unsigned long ipi_count;
12421 extern void smp_invalidate_rcv(void); /* Process NMI */
12422 extern void smp_local_timer_interrupt(
12423     struct pt_regs * regs);
12424 extern void (*mtrr_hook) (void);
12425 extern void setup_APIC_clock (void);
12426 extern volatile int __cpu_logical_map[NR_CPUS];
12427 extern inline int cpu_logical_map(int cpu)
12428 {
12429     return __cpu_logical_map[cpu];
12430 }
12431
12432
12433 /* General fns that each host system must provide. */
12434
12435 extern void smp_callin(void);
12436 extern void smp_boot_cpus(void);
12437 /* Store per CPU info (like the initial udelay numbers */
12438 extern void smp_store_cpu_info(int id);
12439
12440 /* This function is needed by all SMP systems. It must
12441  * _always_ be valid from the initial startup. We map
12442  * APIC_BASE very early in page_setup(), so this is
12443  * correct in the x86 case. */
12444
12445 #define smp_processor_id() (current->processor)
12446
12447 extern __inline int hard_smp_processor_id(void)
12448 {
12449     /* we don't want to mark this access volatile - bad
12450      * code generation */
12451     return GET_APIC_ID(*(unsigned long *)
12452         (APIC_BASE+APIC_ID));
12453 }
12454

```

```

12455 #endif /* !ASSEMBLY */
12456
12457 #define NO_PROC_ID 0xFF /* No processor magic marker */
12458
12459 /* This magic constant controls our willingness to
12460  * transfer a process across CPUs. Such a transfer incurs
12461  * misses on the L1 cache, and on a P6 or P5 with
12462  * multiple L2 caches L2 hits. My gut feeling is this
12463  * will vary by board in value. For a board with separate
12464  * L2 cache it probably depends also on the RSS, and for
12465  * a board with shared L2 cache it ought to decay fast as
12466  * other processes are run. */
12467
12468 #define PROC_CHANGE_PENALTY    15 /* Schedule penalty */
12469
12470 #endif
12471 #endif

```

include/asm-i386/softirq.h

```

12472 #ifndef __ASM_SOFTIRQ_H
12473 #define __ASM_SOFTIRQ_H
12474
12475 #include <asm/atomic.h>
12476 #include <asm/hardirq.h>
12477
12478 extern unsigned int local_bh_count[NR_CPUS];
12479
12480 #define get_active_bhs()    (bh_mask & bh_active)
12481 #define clear_active_bhs(x) \
12482     atomic_clear_mask((x), &bh_active)
12483
12484 extern inline void init_bh(int nr, void (*routine)(void))
12485 {
12486     bh_base[nr] = routine;
12487     atomic_set(&bh_mask_count[nr], 0);
12488     bh_mask |= 1 << nr;
12489 }
12490
12491 extern inline void remove_bh(int nr)
12492 {
12493     bh_mask &= ~(1 << nr);
12494     mb();
12495     bh_base[nr] = NULL;
12496 }
12497
12498 extern inline void mark_bh(int nr)
12499 {

```



```

12500 set_bit(nr, &bh_active);
12501 }
12502
12503 #ifdef __SMP__
12504
12505 /* The locking mechanism for base handlers, to prevent
12506 * re-entrancy, is entirely private to an implementation,
12507 * it should not be referenced at all outside of this
12508 * file. */
12509 extern atomic_t global_bh_lock;
12510 extern atomic_t global_bh_count;
12511
12512 extern void synchronize_bh(void);
12513
12514 static inline void start_bh_atomic(void)
12515 {
12516     atomic_inc(&global_bh_lock);
12517     synchronize_bh();
12518 }
12519
12520 static inline void end_bh_atomic(void)
12521 {
12522     atomic_dec(&global_bh_lock);
12523 }
12524
12525 /* These are for the IRQs testing the lock */
12526 static inline int softirq_trylock(int cpu)
12527 {
12528     if (!test_and_set_bit(0, &global_bh_count)) {
12529         if (atomic_read(&global_bh_lock) == 0) {
12530             ++local_bh_count[cpu];
12531             return 1;
12532         }
12533         clear_bit(0, &global_bh_count);
12534     }
12535     return 0;
12536 }
12537
12538 static inline void softirq_endlock(int cpu)
12539 {
12540     local_bh_count[cpu]--;
12541     clear_bit(0, &global_bh_count);
12542 }
12543
12544 #else
12545
12546 extern inline void start_bh_atomic(void)
12547 {

```

0.012

```

12548     local_bh_count[smp_processor_id()]++;
12549     barrier();
12550 }
12551
12552 extern inline void end_bh_atomic(void)
12553 {
12554     barrier();
12555     local_bh_count[smp_processor_id()]--;
12556 }
12557
12558 /* These are for the irq's testing the lock */
12559 #define softirq_trylock(cpu) (local_bh_count[cpu] \
12560     ? 0 : (local_bh_count[cpu]=1))
12561 #define softirq_endlock(cpu) (local_bh_count[cpu] = 0)
12562 #define synchronize_bh() barrier()
12563
12564 #endif /* SMP */
12565
12566 /* These use a mask count to correctly handle nested
12567 * disable/enable calls */
12568 extern inline void disable_bh(int nr)
12569 {
12570     bh_mask &= ~(1 << nr);
12571     atomic_inc(&bh_mask_count[nr]);
12572     synchronize_bh();
12573 }
12574
12575 extern inline void enable_bh(int nr)
12576 {
12577     if (atomic_dec_and_test(&bh_mask_count[nr]))
12578         bh_mask |= 1 << nr;
12579 }
12580
12581 #endif /* __ASM_SOFTIRQ_H */

```

include/asm-i386/spinlock.h

```

12582 #ifndef __ASM_SPINLOCK_H
12583 #define __ASM_SPINLOCK_H
12584
12585 #ifdef __SMP__
12586
12587 /* 0 == no debugging, 1 == maintain lock state, 2 == full
12588 * debug */
12589 #define DEBUG_SPINLOCKS 0
12590
12591 #if (DEBUG_SPINLOCKS < 1)
12592

```

```

12593 /* Your basic spinlocks, allowing only a single CPU
12594  * anywhere
12595  *
12596  * Gcc-2.7.x has a nasty bug with empty initializers. */
12597 #if (__GNUC__ > 2) || \
12598     (__GNUC__ == 2 && __GNUC_MINOR__ >= 8)
12599     typedef struct { } spinlock_t;
12600     #define SPIN_LOCK_UNLOCKED (spinlock_t) { }
12601 #else
12602     typedef struct { int gcc_is_buggy; } spinlock_t;
12603     #define SPIN_LOCK_UNLOCKED (spinlock_t) { 0 }
12604 #endif
12605
12606 #define spin_lock_init(lock)    do { } while(0)
12607 #define spin_lock(lock)        do { } while(0)
12608 #define spin_trylock(lock)     (1)
12609 #define spin_unlock_wait(lock) do { } while(0)
12610 #define spin_unlock(lock)      do { } while(0)
12611 #define spin_lock_irq(lock)    cli()
12612 #define spin_unlock_irq(lock)  sti()
12613
12614 #define spin_lock_irqsave(lock, flags) \
12615     do { save_flags(flags); cli(); } while (0)
12616 #define spin_unlock_irqrestore(lock, flags) \
12617     restore_flags(flags)
12618
12619 #elif (DEBUG_SPINLOCKS < 2)
12620
12621 typedef struct {
12622     volatile unsigned int lock;
12623 } spinlock_t;
12624 #define SPIN_LOCK_UNLOCKED (spinlock_t) { 0 }
12625
12626 #define spin_lock_init(x) do { (x)->lock = 0; } while (0)
12627 #define spin_trylock(lock) (!test_and_set_bit(0,(lock)))
12628
12629 #define spin_lock(x)      do { (x)->lock = 1; } while (0)
12630 #define spin_unlock_wait(x) do { } while (0)
12631 #define spin_unlock(x)   do { (x)->lock = 0; } while (0)
12632 #define spin_lock_irq(x) \
12633     do { cli(); spin_lock(x); } while (0)
12634 #define spin_unlock_irq(x) \
12635     do { spin_unlock(x); sti(); } while (0)
12636
12637 #define spin_lock_irqsave(x, flags) \
12638     do { save_flags(flags); spin_lock_irq(x); } while (0)
12639 #define spin_unlock_irqrestore(x, flags) \
12640     do { spin_unlock(x); restore_flags(flags); } while (0)

```

```

12641
12642 #else /* (DEBUG_SPINLOCKS >= 2) */
12643
12644 typedef struct {
12645     volatile unsigned int lock;
12646     volatile unsigned int babble;
12647     const char *module;
12648 } spinlock_t;
12649 #define SPIN_LOCK_UNLOCKED \
12650     (spinlock_t) { 0, 25, __BASE_FILE__ }
12651
12652 #include <linux/kernel.h>
12653
12654 #define spin_lock_init(x) do { (x)->lock = 0; } while (0)
12655 #define spin_trylock(lock) (!test_and_set_bit(0,(lock)))
12656
12657 #define spin_lock(x) \
12658     do { \
12659         unsigned long __spinflags; \
12660         save_flags(__spinflags); \
12661         cli(); \
12662         if ((x)->lock&&(x)->babble) { \
12663             printk("%s:%d: spin_lock(%s:%p) already locked\n", \
12664                 __BASE_FILE__, __LINE__, (x)->module, (x)); \
12665             (x)->babble--; \
12666         } \
12667         (x)->lock = 1; \
12668         restore_flags(__spinflags); \
12669     } while (0)
12670 #define spin_unlock_wait(x) \
12671     do { \
12672         unsigned long __spinflags; \
12673         save_flags(__spinflags); \
12674         cli(); \
12675         if ((x)->lock&&(x)->babble) { \
12676             printk("%s:%d: spin_unlock_wait(%s:%p) deadlock\n", \
12677                 __BASE_FILE__, __LINE__, (x)->module, (x)); \
12678             (x)->babble--; \
12679         } \
12680         restore_flags(__spinflags); \
12681     } while (0)
12682 #define spin_unlock(x) \
12683     do { \
12684         unsigned long __spinflags; \
12685         save_flags(__spinflags); \
12686         cli(); \
12687         if (!(x)->lock&&(x)->babble) { \
12688             printk("%s:%d: spin_unlock(%s:%p) not locked\n", \

```

```

12689     __BASE_FILE__, __LINE__, (x)->module, (x)); \
12690     (x)->babble--; \
12691     } \
12692     (x)->lock = 0; \
12693     restore_flags(__spinflags); \
12694 } while (0) \
12695 #define spin_lock_irq(x) \
12696 do { \
12697     cli(); \
12698     if ((x)->lock&&(x)->babble) { \
12699         printk("%s:%d: spin_lock_irq(%s:%p) already locked" \
12700             "\n", __BASE_FILE__, __LINE__, (x)->module, (x)); \
12701         (x)->babble--; \
12702     } \
12703     (x)->lock = 1; \
12704 } while (0) \
12705 #define spin_unlock_irq(x) \
12706 do { \
12707     cli(); \
12708     if (!(x)->lock&&(x)->babble) { \
12709         printk("%s:%d: spin_lock(%s:%p) not locked\n", \
12710             __BASE_FILE__, __LINE__, (x)->module, (x)); \
12711         (x)->babble--; \
12712     } \
12713     (x)->lock = 0; \
12714     sti(); \
12715 } while (0) \
12716 #define spin_lock_irqsave(x, flags) \
12717 do { \
12718     save_flags(flags); \
12719     cli(); \
12720     if ((x)->lock&&(x)->babble) { \
12721         printk("%s:%d: spin_lock_irqsave(%s:%p) already " \
12722             "locked\n", __BASE_FILE__, __LINE__, \
12723             (x)->module, (x)); \
12724         (x)->babble--; \
12725     } \
12726     (x)->lock = 1; \
12727 } while (0) \
12728 #define spin_unlock_irqrestore(x, flags) \
12729 do { \
12730     cli(); \
12731     if (!(x)->lock&&(x)->babble) { \
12732         .printk("%s:%d: spin_unlock_irqrestore(%s:%p) " \
12733             "not locked\n", __BASE_FILE__, __LINE__, \
12734             (x)->module, (x)); \
12735         (x)->babble--; \
12736     } \
12737     (x)->lock = 0; \
12738     restore_flags(flags); \
12739 } while (0) \
12740 #endif /* DEBUG_SPINLOCKS */ \
12741 \
12742 /* Read-write spinlocks, allowing multiple readers but \
12743 * only one writer. \
12744 * \
12745 * NOTE! it is quite common to have readers in interrupts \
12746 * but no interrupt writers. For those circumstances we \
12747 * can "mix" irq-safe locks - any writer needs to get a \
12748 * irq-safe write-lock, but readers can get non-irqsafe \
12749 * read-locks. \
12750 * \
12751 * Gcc-2.7.x has a nasty bug with empty initializers. */ \
12752 #if (__GNUC__ > 2) || \
12753     (__GNUC__ == 2 && __GNUC_MINOR__ >= 8) \
12754     typedef struct { } rwlock_t; \
12755 #define RW_LOCK_UNLOCKED (rwlock_t) { } \
12756 #else \
12757     typedef struct { int gcc_is_buggy; } rwlock_t; \
12758 #define RW_LOCK_UNLOCKED (rwlock_t) { 0 } \
12759 #endif \
12760 \
12761 #define read_lock(lock) do { } while(0) \
12762 #define read_unlock(lock) do { } while(0) \
12763 #define write_lock(lock) do { } while(0) \
12764 #define write_unlock(lock) do { } while(0) \
12765 #define read_lock_irq(lock) cli() \
12766 #define read_unlock_irq(lock) sti() \
12767 #define write_lock_irq(lock) cli() \
12768 #define write_unlock_irq(lock) sti() \
12769 \
12770 #define read_lock_irqsave(lock, flags) \
12771 do { save_flags(flags); cli(); } while (0) \
12772 #define read_unlock_irqrestore(lock, flags) \
12773 restore_flags(flags) \
12774 #define write_lock_irqsave(lock, flags) \
12775 do { save_flags(flags); cli(); } while (0) \
12776 #define write_unlock_irqrestore(lock, flags) \
12777 restore_flags(flags) \
12778 \
12779 #else /* __SMP__ */ \
12780 \
12781 /* Your basic spinlocks, allowing only a single CPU \
12782 * anywhere */ \
12783 \

```

```

12785 typedef struct {
12786     volatile unsigned int lock;
12787 } spinlock_t;
12788
12789 #define SPIN_LOCK_UNLOCKED (spinlock_t) { 0 }
12790
12791 #define spin_lock_init(x) do { (x)->lock = 0; } while(0)
12792 /* Simple spin lock operations. There are two variants,
12793  * one clears IRQ's on the local processor, one does not.
12794  *
12795  * We make no fairness assumptions. They have a cost. */
12796
12797 #define spin_unlock_wait(x) \
12798 do { \
12799     barrier(); \
12800 } while(((volatile spinlock_t *) (x))->lock)
12801
12802 typedef struct { unsigned long a[100]; } __dummy_lock_t;
12803 #define __dummy_lock(lock) (*(__dummy_lock_t *) (lock))
12804
12805 #define spin_lock_string \
12806     "\n1:\t" \
12807     "lock : btsl $0,%0\n\t" \
12808     "jc 2f\n" \
12809     ".section .text.lock,\"ax\"\n" \
12810     "2:\t" \
12811     "testb $1,%0\n\t" \
12812     "jne 2b\n\t" \
12813     "jmp 1b\n" \
12814     ".previous"
12815
12816 #define spin_unlock_string \
12817     "lock : btrl $0,%0"
12818
12819 #define spin_lock(lock) \
12820 __asm__ __volatile__( \
12821     spin_lock_string \
12822     : "=m" (__dummy_lock(lock)))
12823
12824 #define spin_unlock(lock) \
12825 __asm__ __volatile__( \
12826     spin_unlock_string \
12827     : "=m" (__dummy_lock(lock)))
12828
12829 #define spin_trylock(lock) (!test_and_set_bit(0,(lock)))
12830
12831 #define spin_lock_irq(lock) \
12832 do { __cli(); spin_lock(lock); } while (0)

```

```

12833
12834 #define spin_unlock_irq(lock) \
12835 do { spin_unlock(lock); __sti(); } while (0)
12836
12837 #define spin_lock_irqsave(lock, flags) \
12838 do { __save_flags(flags); __cli(); \
12839     spin_lock(lock); } while (0)
12840
12841 #define spin_unlock_irqrestore(lock, flags) \
12842 do { spin_unlock(lock); \
12843     __restore_flags(flags); } while (0)
12844
12845 /* Read-write spinlocks, allowing multiple readers but
12846  * only one writer.
12847  *
12848  * NOTE! it is quite common to have readers in interrupts
12849  * but no interrupt writers. For those circumstances we
12850  * can "mix" irq-safe locks - any writer needs to get a
12851  * irq safe write-lock, but readers can get non-irqsafe
12852  * read-locks. */
12853 typedef struct {
12854     volatile unsigned int lock;
12855     unsigned long previous;
12856 } rwlock_t;
12857
12858 #define RW_LOCK_UNLOCKED (rwlock_t) { 0, 0 }
12859
12860 /* On x86, we implement read-write locks as a 32-bit
12861  * counter with the high bit (sign) being the "write"
12862  * bit.
12863  *
12864  * The inline assembly is non-obvious. Think about it. */
12865 #define read_lock(rw) \
12866 asm volatile("\n1:\t" \
12867     "lock ; incl %0\n\t" \
12868     "js 2f\n" \
12869     ".section .text.lock,\"ax\"\n" \
12870     "2:\tlock ; decl %0\n\t" \
12871     "3:\tcmpl $0,%0\n\t" \
12872     "js 3b\n\t" \
12873     "jmp 1b\n" \
12874     ".previous" \
12875     : "=m" (__dummy_lock(&(rw)->lock)))
12876
12877 #define read_unlock(rw) \
12878 asm volatile("lock ; decl %0" \
12879     : "=m" (__dummy_lock(&(rw) >lock)))
12880

```

```

12881 #define write_lock(rw) \
12882     asm volatile("\n1:\t" \
12883         "lock ; btsl $31,%0\n\t" \
12884         "jc 4f\n" \
12885         "2:\ttestl $0x7fffffff,%0\n\t" \
12886         "jne 3f\n" \
12887         ".section .text.lock,\"ax\"\n" \
12888         "3:\tlock ; btrl $31,%0\n" \
12889         "4:\tcmp $0,%0\n\t" \
12890         "jne 4b\n\t" \
12891         "jmp 1b\n" \
12892         ".previous" \
12893         : "=m" (__dummy_lock(&(rw)->lock)) \
12894         ;
12895 #define write_unlock(rw) \
12896     asm volatile("lock ; btrl $31,%0":"=m" \
12897         (__dummy_lock(&(rw)->lock)) \
12898         ;
12899 #define read_lock_irq(lock) \
12900     do { __cli(); read_lock(lock); } while (0)
12901 #define read_unlock_irq(lock) \
12902     do { read_unlock(lock); __sti(); } while (0)
12903 #define write_lock_irq(lock) \
12904     do { __cli(); write_lock(lock); } while (0)
12905 #define write_unlock_irq(lock) \
12906     do { write_unlock(lock); __sti(); } while (0)
12907
12908 #define read_lock_irqsave(lock, flags) \
12909     do { __save_flags(flags); __cli(); \
12910         read_lock(lock); } while (0)
12911 #define read_unlock_irqrestore(lock, flags) \
12912     do { read_unlock(lock); \
12913         __restore_flags(flags); } while (0)
12914 #define write_lock_irqsave(lock, flags) \
12915     do { __save_flags(flags); __cli(); \
12916         write_lock(lock); } while (0)
12917 #define write_unlock_irqrestore(lock, flags) \
12918     do { write_unlock(lock); \
12919         __restore_flags(flags); } while (0)
12920
12921 #endif /* __SMP__ */
12922 #endif /* __ASM_SPINLOCK_H */

include/asm-i386/system.h
12923 #ifndef __ASM_SYSTEM_H
12924 #define __ASM_SYSTEM_H
12925

```

```

12926 #include <linux/kernel.h>
12927 #include <asm/segment.h>
12928
12929 #ifdef __KERNEL__
12930
12931 /* one of the stranger aspects of C forward decls. */
12932 struct task_struct;
12933 extern void
12934 FASTCALL(__switch_to(struct task_struct *prev,
12935                     struct task_struct *next));
12936
12937 /* We do most of the task switching in C, but we need to
12938  * do the EIP/ESP switch in assembly.. */
12939 #define switch_to(prev,next) do { \
12940     unsigned long eax, edx, ecx; \
12941     asm volatile("pushl %%ebx\n\t" \
12942         "pushl %%esi\n\t" \
12943         "pushl %%edi\n\t" \
12944         "pushl %%ebp\n\t" \
12945         "movl %%esp,%0\n\t" /* save ESP */ \
12946         "movl %5,%%esp\n\t" /* restore ESP */ \
12947         "movl $1f,%1\n\t" /* save EIP */ \
12948         "pushl %6\n\t" /* restore EIP */ \
12949         "jmp __switch_to\n" \
12950         "1:\t" \
12951         "popl %%ebp\n\t" \
12952         "popl %%edi\n\t" \
12953         "popl %%esi\n\t" \
12954         "popl %%ebx" \
12955         : "=m" (prev->tss.esp), "=m" (prev->tss.eip), \
12956         "=a" (eax), "=d" (edx), "=c" (ecx) \
12957         : "m" (next->tss.esp), "m" (next->tss.eip), \
12958         "a" (prev), "d" (next)); \
12959     } while (0)
12960
12961 #define _set_base(addr,base) do { unsigned long __pr; \
12962     __asm__ __volatile__ ("movw %%dx,%1\n\t" \
12963         "rorl $16,%%edx\n\t" \
12964         "movb %%dl,%2\n\t" \
12965         "movb %%dh,%3" \
12966         : "=&d" (__pr) \
12967         : "m" (*((addr)+2)), \
12968         "m" (*((addr)+4)), \
12969         "m" (*((addr)+7)), \
12970         "0" (base) \
12971         ); } while(0)
12972
12973 #define _set_limit(addr,limit) do { unsigned long 1r; \

```

```

12974 __asm__ __volatile__ ("movw %%dx,%l\n\t" \
12975 "rorl $16,%%edx\n\t" \
12976 "movb %2,%%dh\n\t" \
12977 "andb $0xf0,%%dh\n\t" \
12978 "orb %%dh,%%dl\n\t" \
12979 "movb %%dl,%2" \
12980 : "=&d" (&l_r) \
12981 : "m" (*(addr)), \
12982 "m" (*(addr)+6)), \
12983 "0" (limit) \
12984 ); } while(0) \
12985 \
12986 #define set_base(ldt,base) \
12987 _set_base( ((char *)&(ldt)) , (base) ) \
12988 #define set_limit(ldt,limit) \
12989 _set_limit( ((char *)&(ldt)) , ((limit)-1)>>12 ) \
12990 \
12991 static inline unsigned long _get_base(char * addr) \
12992 { \
12993     unsigned long __base; \
12994     __asm__ ("movb %3,%%dh\n\t" \
12995 "movb %2,%%dl\n\t" \
12996 "shll $16,%%edx\n\t" \
12997 "movw %1,%%dx" \
12998 : "=&d" (&__base) \
12999 : "m" (*(addr)+2)), \
13000 "m" (*(addr)+4)), \
13001 "m" (*(addr)+7)); \
13002     return __base; \
13003 } \
13004 \
13005 #define get_base(ldt) _get_base( ((char *)&(ldt)) ) \
13006 \
13007 /* Load a segment. Fall back on loading the zero segment \
13008 * if something goes wrong.. */ \
13009 #define loadsegment(seg,value) \
13010     asm volatile("\n" \
13011 "1:\t" \
13012 "movl %0,%" #seg "\n" \
13013 "2:\n" \
13014 ".section .fixup,\"ax\"\n" \
13015 "3:\t" \
13016 "pushl $0\n\t" \
13017 "popl %" #seg "\n\t" \
13018 "jmp 2b\n" \
13019 ".previous\n" \
13020 ".section __ex_table,\"a\"\n\t" \
13021 ".align 4\n\t" \
13022 ".long 1b,3b\n" \
13023 ".previous" \
13024 : : "m" (*(unsigned int *)&(value))) \
13025 \
13026 /* Clear and set 'TS' bit respectively */ \
13027 #define clts() __asm__ __volatile__ ("clts") \
13028 #define read_cr0() ( { \
13029     unsigned int __dummy; \
13030     __asm__( \
13031         "movl %%cr0,%0\n\t" \
13032         : "=r" (&__dummy); \
13033         __dummy; \
13034     ) \
13035 #define write_cr0(x) \
13036     __asm__("movl %0,%%cr0": : "r" (x)); \
13037 #define stts() write_cr0(8 | read_cr0()) \
13038 \
13039 #endif /* __KERNEL__ */ \
13040 \
13041 static inline unsigned long get_limit( \
13042     unsigned long segment) \
13043 { \
13044     unsigned long __limit; \
13045     __asm__("lsl %1,%0" \
13046         : "=r" (&__limit): "r" (segment)); \
13047     return __limit+1; \
13048 } \
13049 \
13050 #define nop() __asm__ __volatile__ ("nop") \
13051 \
13052 #define xchg(ptr,x) ((__typeof__(*(ptr))) \
13053     __xchg((unsigned long)(x),(ptr),sizeof(*(ptr)))) \
13054 #define tas(ptr) (xchg((ptr),1)) \
13055 \
13056 struct __xchg_dummy { unsigned long a[100]; }; \
13057 #define __xg(x) ((struct __xchg_dummy *) (x)) \
13058 \
13059 /* Note: no "lock" prefix even on SMP: xchg always \
13060 * implies lock anyway */ \
13061 static inline unsigned long __xchg(unsigned long x, \
13062     void * ptr, int size) \
13063 { \
13064     switch (size) { \
13065     case 1: \
13066         __asm__("xchgb %b0,%1" \
13067             : "=q" (x) \
13068             : "m" (*__xg(ptr)), "0" (x) \
13069             : "memory"); \

```

```

13070     break;
13071     case 2:
13072         __asm__("xchgw %w0,%1"
13073             : "=r" (x)
13074             : "m" (*__xg(ptr)), "0" (x)
13075             : "memory");
13076     break;
13077     case 4:
13078         __asm__("xchgl %0,%1"
13079             : "=r" (x)
13080             : "m" (*__xg(ptr)), "0" (x)
13081             : "memory");
13082     break;
13083 }
13084 return x;
13085 }
13086
13087 /* Force strict CPU ordering.  And yes, this is required
13088 * on UP too when we're talking to devices.
13089 *
13090 * For now, "wmb()" doesn't actually do anything, as all
13091 * Intel CPU's follow what Intel calls a *Processor
13092 * Order*, in which all writes are seen in the program
13093 * order even outside the CPU.
13094 *
13095 * I expect future Intel CPU's to have a weaker ordering,
13096 * but I'd also expect them to finally get their act
13097 * together and add some real memory barriers if so. */
13098 #define mb()    __asm__ __volatile__ \
13099     ("lock; addl $0,0(%%esp)": : : "memory")
13100 #define rmb()    mb()
13101 #define wmb()    __asm__ __volatile__ ("" : : "memory")
13102
13103 /* interrupt control.. */
13104 #define __sti() __asm__ __volatile__ ("sti": : : "memory")
13105 #define __cli() __asm__ __volatile__ ("cli": : : "memory")
13106 #define __save_flags(x) __asm__ __volatile__ \
13107     ("pushfl ; popl %0": "=g" (x): /* no input */ : "memory")
13108 #define __restore_flags(x) __asm__ __volatile__ \
13109     ("pushl %0 ; popfl": /* no output */ : "g" (x): "memory")
13110
13111 #ifdef __SMP__
13112
13113 extern void __global_cli(void);
13114 extern void __global_sti(void);
13115 extern unsigned long __global_save_flags(void);
13116 extern void __global_restore_flags(unsigned long);
13117 #define cli() __global_cli()

```

```

13118 #define sti() __global_sti()
13119 #define save_flags(x) ((x)=__global_save_flags())
13120 #define restore_flags(x) __global_restore_flags(x)
13121
13122 #else
13123
13124 #define cli() __cli()
13125 #define sti() __sti()
13126 #define save_flags(x) __save_flags(x)
13127 #define restore_flags(x) __restore_flags(x)
13128
13129 #endif
13130
13131 /* disable hlt during certain critical i/o operations */
13132 #define HAVE_DISABLE_HLT
13133 void disable_hlt(void);
13134 void enable_hlt(void);
13135
13136 #endif

```

include/asm-i386/uaccess.h

```

13137 #ifndef __i386_UACCESS_H
13138 #define __i386_UACCESS_H
13139
13140 /* User space memory access functions */
13141 #include <linux/config.h>
13142 #include <linux/sched.h>
13143 #include <asm/page.h>
13144
13145 #define VERIFY_READ 0
13146 #define VERIFY_WRITE 1
13147
13148 /* The fs value determines whether argument validity
13149 * checking should be performed or not.  If get_fs() ==
13150 * USER_DS, checking is performed, with get_fs() ==
13151 * KERNEL_DS, checking is bypassed.
13152 *
13153 * For historical reasons, these macros are grossly
13154 * misnamed.  */
13155
13156 #define MAKE_MM_SEG(s) ((mm_segment_t) { (s) })
13157
13158 #define KERNEL_DS    MAKE_MM_SEG(0xFFFFFFFF)
13160 #define USER_DS    MAKE_MM_SEG(PAGE_OFFSET)
13161
13162 #define get_ds()    (KERNEL_DS)

```

```

13163 #define get_fs()          (current->addr_limit)
13164 #define set_fs(x)         (current->addr_limit = (x))
13165
13166 #define segment_eq(a,b) ((a).seg == (b).seg)
13167
13168 extern int __verify_write(const void *, unsigned long);
13169
13170 #define __addr_ok(addr)    \
13171     ((unsigned long)(addr) < (current->addr_limit.seg))
13172
13173 /* Uhhuh, this needs 33-bit arithmetic. We have a carry*/
13174 #define __range_ok(addr,size) ({ \
13175     unsigned long flag,sum; \
13176     asm("addl %3,%1 : sbb1 %0,%0; cmpl %1,%4; sbb1 $0,%0" \
13177         : "=&r" (flag), "=r" (sum) \
13178         : "1" (addr),"g" (size),"g" \
13179         (current->addr_limit.seg)); \
13180     flag; })
13181
13182 #ifdef CONFIG_X86_WP_WORKS_OK
13183
13184 #define access_ok(type,addr,size) \
13185     (__range_ok(addr,size) == 0)
13186
13187 #else
13188
13189 #define access_ok(type,addr,size) \
13190     ((__range_ok(addr,size) == 0) && \
13191     ((type) == VERIFY_READ || boot_cpu_data.wp_works_ok || \
13192     segment_eq(get_fs(),KERNEL_DS) || \
13193     __verify_write((void *) (addr),(size))))
13194
13195 #endif /* CPU */
13196
13197 extern inline int verify_area(int type, const void * addr
13198                             , unsigned long size)
13199 {
13200     return access_ok(type,addr,size) ? 0 : -EFAULT;
13201 }
13202
13203
13204 /* The exception table consists of pairs of addresses:
13205  * the first is the address of an instruction that is
13206  * allowed to fault, and the second is the address at
13207  * which the program should continue. No registers are
13208  * modified, so it is entirely up to the continuation
13209  * code to figure out what to do.
13210  *

```

```

13211  * All the routines below use bits of fixup code that are
13212  * out of line with the main instruction path. This
13213  * means when everything is well, we don't even have to
13214  * jump over them. Further, they do not intrude on our
13215  * cache or tlb entries. */
13216
13217 struct exception_table_entry
13218 {
13219     unsigned long insn, fixup;
13220 };
13221
13222 /* Returns 0 if exception not found, fixup otherwise. */
13223 extern unsigned long search_exception_table(
13224     unsigned long);
13225
13226
13227 /* These are the main single-value transfer routines.
13228  * They automatically use the right size if we just have
13229  * the right pointer type.
13230  *
13231  * This gets kind of ugly. We want to return _two_ values
13232  * in "get_user()" and yet we don't want to do any
13233  * pointers, because that is too much of a performance
13234  * impact. Thus we have a few rather ugly macros here.
13235  * and hide all the ugliness from the user.
13236  *
13237  * The "__xxx" versions of the user access functions are
13238  * versions that do not verify the address space, that
13239  * must have been done previously with a separate
13240  * "access_ok()" call (this is used when we do multiple
13241  * accesses to the same area of user memory). */
13242
13243 extern void __get_user_1(void);
13244 extern void __get_user_2(void);
13245 extern void __get_user_4(void);
13246
13247 #define __get_user_x(size,ret,x,ptr) \
13248     __asm__ __volatile__ ("call __get_user_" #size \
13249         : "=a" (ret),"=d" (x) \
13250         : "0" (ptr))
13251
13252 /* Careful: we have to cast the result to the type of the
13253  * pointer for sign reasons */
13254 #define get_user(x,ptr) \
13255     ({ \
13256         int __ret_gu,__val_gu; \
13257         switch(sizeof (*(ptr))) { \
13258             case 1: __get_user_x(1,__ret_gu,__val_gu,ptr); break; \
13259             case 2: __get_user_x(2,__ret_gu,__val_gu,ptr); break; \

```



```

13259 case 4: __get_user_x(4,__ret_gu,__val_gu,ptr); break;\
13260 default: __get_user_x(X,__ret_gu,__val_gu,ptr); break;\
13261 } \
13262 (x) = (__typeof__(*(ptr)))__val_gu; \
13263 __ret_gu; \
13264 })
13265
13266 extern void __put_user_1(void);
13267 extern void __put_user_2(void);
13268 extern void __put_user_4(void);
13269
13270 extern void __put_user_bad(void);
13271
13272 #define __put_user_x(size,ret,x,ptr) \
13273 __asm__ __volatile__("call __put_user_" #size \
13274 :"-a" (ret) \
13275 : "0" (ptr),"d" (x) \
13276 : "cx")
13277
13278 #define put_user(x,ptr) \
13279 ({ int __ret_pu; \
13280 switch(sizeof (*(ptr))) { \
13281 case 1: __put_user_x(1,__ret_pu, \
13282 (__typeof__(*(ptr)))(x),ptr); \
13283 break; \
13284 case 2: __put_user_x(2,__ret_pu, \
13285 (__typeof__(*(ptr)))(x),ptr); \
13286 break; \
13287 case 4: __put_user_x(4,__ret_pu, \
13288 (__typeof__(*(ptr)))(x),ptr); \
13289 break; \
13290 default: __put_user_x(X,__ret_pu,x,ptr); break; \
13291 } \
13292 __ret_pu; \
13293 })
13294
13295 #define __get_user(x,ptr) \
13296 __get_user_nocheck((x),(ptr),sizeof(*(ptr)))
13297 #define __put_user(x,ptr) \
13298 __put_user_nocheck((__typeof__(*(ptr)))(x),(ptr), \
13299 sizeof(*(ptr)))
13300
13301 #define __put_user_nocheck(x,ptr,size) \
13302 ({ \
13303 long __pu_err; \
13304 __put_user_size((x),(ptr),(size),__pu_err); \
13305 __pu_err; \
13306 })

```

```

13307
13308 #define __put_user_size(x,ptr,size,retval) \
13309 do { \
13310 retval = 0; \
13311 switch (size) { \
13312 case 1: __put_user_asm(x,ptr,retval,"b","b","iq"); \
13313 break; \
13314 case 2: __put_user_asm(x,ptr,retval,"w"."w"."ir"); \
13315 break; \
13316 case 4: __put_user_asm(x,ptr,retval,"l".""."ir"); \
13317 break; \
13318 default: __put_user_bad(); \
13319 } \
13320 } while (0).
13321
13322 struct __large_struct { unsigned long buf[100]; };
13323 #define __m(x) (*(struct __large_struct *) (x))
13324
13325 /* Tell gcc we read from memory instead of writing: this
13326 * is because we do not write to any memory gcc knows
13327 * about, so there are no aliasing issues. */
13328 #define __put_user_asm(x,addr,err,itype,rtype,ltype) \
13329 __asm__ __volatile__( \
13330 "1: mov"itype" %"rtype"l,%2\n" \
13331 "2:\n" \
13332 ".section .fixup,\"ax\"\n" \
13333 "3: movl %3,%0\n" \
13334 " jmp 2b\n" \
13335 ".previous\n" \
13336 ".section __ex_table,\"a\"\n" \
13337 " .align 4\n" \
13338 " .long 1b,3b\n" \
13339 ".previous" \
13340 : "=r"(err) \
13341 : ltype (x), "m"(__m(addr)), "i"(-EFAULT), "0"(err)) \
13342
13343
13344 #define __get_user_nocheck(x,ptr,size) \
13345 ({ \
13346 long __gu_err, __gu_val; \
13347 __get_user_size(__gu_val,(ptr),(size),__gu_err); \
13348 (x) = (__typeof__(*(ptr)))__gu_val; \
13349 __gu_err; \
13350 })
13351
13352 extern long __get_user_bad(void);
13353
13354 #define __get_user_size(x,ptr,size,retval) \

```

```

13355 do {
13356     retval = 0;
13357     switch (size) {
13358         case 1: __get_user_asm(x,ptr,retval,"b"."b","-q");
13359                 break;
13360         case 2: __get_user_asm(x,ptr,retval,"w"."w","-r");
13361                 break;
13362         case 4: __get_user_asm(x,ptr,retval,"l"."l","-r");
13363                 break;
13364         default: (x) = __get_user_bad();
13365     }
13366 } while (0)
13367
13368 #define __get_user_asm(x,addr,err,itype,rtype,ltype)
13369     __asm__ __volatile__(
13370     "1:   movl itype" %2,%"rtype"l\n"
13371     "2:\n"
13372     ".section .fixup,\"ax\"\n"
13373     "3:   movl %3,%0\n"
13374     "     xor itype" %"rtype"l,%"rtype"l\n"
13375     "     jmp 2b\n"
13376     ".previous\n"
13377     ".section __ex_table,\"a\"\n"
13378     "     .align 4\n"
13379     "     .long 1b,3b\n"
13380     ".previous"
13381     : "=r"(err), ltype (x)
13382     : "m"(__m(addr)), "i"(-EFAULT), "0"(err))
13383
13384 /* The "xxx_ret" versions return constant specified in
13385  * third argument, if something bad happens. These macros
13386  * can be optimized for the case of just returning from
13387  * the function xxx_ret is used. */
13388
13389 #define put_user_ret(x,ptr,ret)
13390     ({ if (put_user(x,ptr)) return ret; })
13391
13392 #define get_user_ret(x,ptr,ret)
13393     ({ if (get_user(x,ptr)) return ret; })
13394
13395 #define __put_user_ret(x,ptr,ret)
13396     ({ if (__put_user(x,ptr)) return ret; })
13397
13398 #define __get_user_ret(x,ptr,ret)
13399     ({ if (__get_user(x,ptr)) return ret; })
13400
13401 /* Copy To/From Userspace */
13402
13403 /* Generic arbitrary sized copy. */
13404 #define __copy_user(to,from,size)
13405 do {
13406     int __d0, __d1;
13407     __asm__ __volatile__(
13408     "0:   rep; movsl\n"
13409     "     movl %3,%0\n"
13410     "1:   rep; movsb\n"
13411     "2:\n"
13412     ".section .fixup,\"ax\"\n"
13413     "3:   lea 0(%3,%0,4),%0\n"
13414     "     jmp 2b\n"
13415     ".previous\n"
13416     ".section __ex_table,\"a\"\n"
13417     "     .align 4\n"
13418     "     .long 0b,3b\n"
13419     "     .long 1b,2b\n"
13420     ".previous"
13421     : "&c"(size), "-&D" (__d0), "-&S" (__d1)
13422     : "r"(size & 3), "0"(size / 4), "1"(to), "2"(from)
13423     : "memory");
13424 } while (0)
13425
13426 #define __copy_user_zeroing(to,from,size)
13427 do {
13428     int __d0, __d1;
13429     __asm__ __volatile__(
13430     "0:   rep; movsl\n"
13431     "     movl %3,%0\n"
13432     "1:   rep; movsb\n"
13433     "2:\n"
13434     ".section .fixup,\"ax\"\n"
13435     "3:   lea 0(%3,%0,4),%0\n"
13436     "4:   pushl %0\n"
13437     "     pushl %%eax\n"
13438     "     xorl %%eax,%%eax\n"
13439     "     rep; stosb\n"
13440     "     popl %%eax\n"
13441     "     popl %0\n"
13442     "     jmp 2b\n"
13443     ".previous\n"
13444     ".section __ex_table,\"a\"\n"
13445     "     .align 4\n"
13446     "     .long 0b,3b\n"
13447     "     .long 1b,4b\n"
13448     ".previous"
13449     : "&c"(size), "-&D" (__d0), "-&S" (__d1)
13450     : "r"(size & 3), "0"(size / 4), "1"(to), "2"(from)

```

```

13451     : "memory"); \
13452 } while (0) \
13453 \
13454 /* We let the __ versions of copy_from/to_user inline, \
13455  * because they're often used in fast paths and have only \
13456  * a small space overhead. */ \
13457 static inline unsigned long \
13458 __generic_copy_from_user_nocheck( \
13459     void *to, const void *from, unsigned long n) \
13460 { \
13461     __copy_user_zeroing(to,from,n); \
13462     return n; \
13463 } \
13464 \
13465 static inline unsigned long \
13466 __generic_copy_to_user_nocheck( \
13467     void *to, const void *from, unsigned long n) \
13468 { \
13469     __copy_user(to,from,n); \
13470     return n; \
13471 } \
13472 \
13473 \
13474 /* Optimize just a little bit when we know the size of \
13475  * the move. */ \
13476 #define __constant_copy_user(to, from, size) \
13477 do { \
13478     int __d0, __d1; \
13479     switch (size & 3) { \
13480     default: \
13481         __asm__ __volatile__( \
13482             "0:    rep; movsl\n" \
13483             "1:\n" \
13484             ".section .fixup,\"ax\"\n" \
13485             "2:    shl $2,%0\n" \
13486             "        jmp 1b\n" \
13487             ".previous\n" \
13488             ".section __ex_table,\"a\"\n" \
13489             "        .align 4\n" \
13490             "        .long 0b,2b\n" \
13491             ".previous" \
13492             : "=c"(size), "=&S" (__d0), "=&D" (__d1) \
13493             : "1"(from), "2"(to), "0"(size/4) \
13494             : "memory"); \
13495         break; \
13496     case 1: \
13497         __asm__ __volatile__( \
13498             "0:    rep; movsl\n" \

```

```

13499             "1:    movsb\n" \
13500             "2:\n" \
13501             ".section .fixup,\"ax\"\n" \
13502             "3:    shl $2,%0\n" \
13503             "4:    incl %0\n" \
13504             "        jmp 2b\n" \
13505             ".previous\n" \
13506             ".section __ex_table,\"a\"\n" \
13507             "        .align 4\n" \
13508             "        .long 0b,3b\n" \
13509             "        .long 1b,4b\n" \
13510             ".previous" \
13511             : "=c"(size), "=&S" (__d0), "=&D" (__d1) \
13512             : "1"(from), "2"(to), "0"(size/4) \
13513             : "memory"); \
13514         break; \
13515     case 2: \
13516         __asm__ __volatile__( \
13517             "0:    rep; movsl\n" \
13518             "1:    movsw\n" \
13519             "2:\n" \
13520             ".section .fixup,\"ax\"\n" \
13521             "3:    shl $2,%0\n" \
13522             "4:    addl $2,%0\n" \
13523             "        jmp 2b\n" \
13524             ".previous\n" \
13525             ".section __ex_table,\"a\"\n" \
13526             "        .align 4\n" \
13527             "        .long 0b,3b\n" \
13528             "        .long 1b,4b\n" \
13529             ".previous" \
13530             : "=c"(size), "=&S" (__d0), "=&D" (__d1) \
13531             : "1"(from), "2"(to), "0"(size/4) \
13532             : "memory"); \
13533         break; \
13534     case 3: \
13535         __asm__ __volatile__( \
13536             "0:    rep; movsl\n" \
13537             "1:    movsw\n" \
13538             "2:    movsb\n" \
13539             "3:\n" \
13540             ".section .fixup,\"ax\"\n" \
13541             "4:    shl $2,%0\n" \
13542             "5:    addl $2,%0\n" \
13543             "6:    incl %0\n" \
13544             "        jmp 3b\n" \
13545             ".previous\n" \
13546             ".section __ex_table,\"a\"\n" \

```

```

13547     "        .align 4\n"           \
13548     "        .long 0b,4b\n"       \
13549     "        .long 1b,5b\n"       \
13550     "        .long 2b,6b\n"       \
13551     ".previous"                   \
13552     : "=c"(size), "=&S" (__d0), "-&D" (__d1) \
13553     : "1"(from), "2"(to), "0"(size/4) \
13554     : "memory");                   \
13555     break;                         \
13556   }                               \
13557 } while (0)
13558
13559 /* Optimize just a little bit when we know the size of
13560  * the move. */
13561 #define __constant_copy_user_zeroing(to, from, size) \
13562 do {
13563     int __d0, __d1;
13564     switch (size & 3) {
13565     default:
13566         __asm__ __volatile__(
13567             "0:   rep; movsl\n"
13568             "1:\n"
13569             ".section .fixup,\"ax\"\n"
13570             "2:   pushl %0\n"
13571             "      pushl %%eax\n"
13572             "      xorl %%eax,%%eax\n"
13573             "      rep; stosl\n"
13574             "      popl %%eax\n"
13575             "      popl %0\n"
13576             "      shl $2,%0\n"
13577             "      jmp 1b\n"
13578             ".previous\n"
13579             ".section __ex_table,\"a\"\n"
13580             "      .align 4\n"
13581             "      .long 0b,2b\n"
13582             ".previous"
13583             : "=c"(size), "=&S" (__d0), "-&D" (__d1) \
13584             : "1"(from), "2"(to), "0"(size/4) \
13585             : "memory");
13586     break;
13587     case 1:
13588         __asm__ __volatile__(
13589             "0:   rep; movsl\n"
13590             "1:   movsb\n"
13591             "2:\n"
13592             ".section .fixup,\"ax\"\n"
13593             "3:   pushl %0\n"
13594             "      pushl %%eax\n"

```

```

13595     "      xorl %%eax,%%eax\n"
13596     "      rep; stosl\n"
13597     "      stosb\n"
13598     "      popl %%eax\n"
13599     "      popl %0\n"
13600     "      shl $2,%0\n"
13601     "      incl %0\n"
13602     "      jmp 2b\n"
13603     "4:   pushl %%eax\n"
13604     "      xorl %%eax,%%eax\n"
13605     "      stosb\n"
13606     "      popl %%eax\n"
13607     "      incl %0\n"
13608     "      jmp 2b\n"
13609     ".previous\n"
13610     ".section __ex_table,\"a\"\n"
13611     "      .align 4\n"
13612     "      .long 0b,3b\n"
13613     "      .long 1b,4b\n"
13614     ".previous"
13615     : "=c"(size), "=&S" (__d0), "-&D" (__d1) \
13616     : "1"(from), "2"(to), "0"(size/4) \
13617     : "memory");
13618     break;
13619     case 2:
13620         __asm__ __volatile__(
13621             "0:   rep; movsl\n"
13622             "1:   movsw\n"
13623             "2:\n"
13624             ".section .fixup,\"ax\"\n"
13625             "3:   pushl %0\n"
13626             "      pushl %%eax\n"
13627             "      xorl %%eax,%%eax\n"
13628             "      rep; stosl\n"
13629             "      stosw\n"
13630             "      popl %%eax\n"
13631             "      popl %0\n"
13632             "      shl $2,%0\n"
13633             "      addl $2,%0\n"
13634             "      jmp 2b\n"
13635             "4:   pushl %%eax\n"
13636             "      xorl %%eax,%%eax\n"
13637             "      stosw\n"
13638             "      popl %%eax\n"
13639             "      addl $2,%0\n"
13640             "      jmp 2b\n"
13641             ".previous\n"
13642             ".section __ex_table,\"a\"\n"

```



```

13739
13740 #define copy_from_user(to,from,n) \
13741     (__builtin_constant_p(n) ? \
13742     __constant_copy_from_user((to),(from),(n)) : \
13743     __generic_copy_from_user((to),(from),(n)))
13744
13745 #define copy_to_user_ret(to,from,n,retval) \
13746     (( if (copy_to_user(to,from,n)) return retval; ))
13747
13748 #define copy_from_user_ret(to,from,n,retval) \
13749     (( if (copy_from_user(to,from,n)) return retval; ))
13750
13751 #define __copy_to_user(to,from,n) \
13752     (__builtin_constant_p(n) ? \
13753     __constant_copy_to_user_nocheck((to),(from),(n)) : \
13754     __generic_copy_to_user_nocheck((to),(from),(n)))
13755
13756 #define __copy_from_user(to,from,n) \
13757     (__builtin_constant_p(n) ? \
13758     __constant_copy_from_user_nocheck((to),(from),(n)) : \
13759     __generic_copy_from_user_nocheck((to),(from),(n)))
13760
13761 long strncpy_from_user(char *dst, const char *src,
13762                       long count);
13763 long __strncpy_from_user(char *dst, const char *src,
13764                          long count);
13765 long strlen_user(const char *str);
13766 unsigned long clear_user(void *mem, unsigned long len);
13767 unsigned long __clear_user(void *mem, unsigned long len);
13768
13769 #endif /* __i386_UACCESS_H */

include/linux/binfmts.h
13770 #ifndef _LINUX_BINFMTS_H
13771 #define _LINUX_BINFMTS_H
13772
13773 #include <linux/ptrace.h>
13774 #include <linux/capability.h>
13775
13776 /* MAX_ARG_PAGES defines the number of pages allocated
13777  * for arguments and envelope for the new program. 32
13778  * should suffice, this gives a maximum env+arg of 128kB
13779  * w/4KB pages! */
13780 #define MAX_ARG_PAGES 32
13781
13782 #ifdef __KERNEL__
13783
13784 /* This structure is used to hold the arguments that are
13785  * used when loading binaries. */
13786 struct linux_binprm{
13787     char buf[128];
13788     unsigned long page[MAX_ARG_PAGES];
13789     unsigned long p;
13790     int sh_bang;
13791     int java; /* Java bin, prevent recursive invocation */
13792     struct dentry * dentry;
13793     int e_uid, e_gid;
13794     kernel_cap_t cap_inheritable, cap_permitted,
13795         cap_effective;
13796     int argc, envc;
13797     char * filename; /* Name of binary */
13798     unsigned long loader, exec;
13799 };
13800
13801 /* This structure defines the functions that are used to
13802  * load the binary formats that linux accepts. */
13803 struct linux_binfmt {
13804     struct linux_binfmt * next;
13805     struct module *module;
13806     int (*load_binary)(struct linux_binprm *,
13807                       struct pt_regs * regs);
13808     int (*load_shlib)(int fd);
13809     int (*core_dump)(long signr, struct pt_regs * regs);
13810 };
13811
13812 extern int register_binfmt(struct linux_binfmt *);
13813 extern int unregister_binfmt(struct linux_binfmt *);
13814
13815 extern int read_exec(struct dentry *,
13816                     unsigned long offset, char * addr, unsigned long count,
13817                     int to_kmem);
13818
13819 extern int open_dentry(struct dentry *, int mode);
13820
13821 extern int init_elf_binfmt(void);
13822 extern int init_elf32_binfmt(void);
13823 extern int init_aout_binfmt(void);
13824 extern int init_aout32_binfmt(void);
13825 extern int init_script_binfmt(void);
13826 extern int init_java_binfmt(void);
13827 extern int init_em86_binfmt(void);
13828 extern int init_misc_binfmt(void);
13829
13830 extern int prepare_binprm(struct linux_binprm *);
13831 extern void remove_arg_zero(struct linux_binprm *);

```

```

13832 extern int search_binary_handler(struct linux_binprm *,
13833                                 struct pt_regs *);
13834 extern int flush_old_exec(struct linux_binprm * bprm);
13835 extern unsigned long setup_arg_pages(unsigned long p,
13836                                     struct linux_binprm * bprm);
13837 extern unsigned long copy_strings(int argc,char ** argv,
13838                                 unsigned long *page, unsigned long p, int from_kmem);
13839
13840 extern void compute_creds(struct linux_binprm *binprm);
13841
13842 /* this eventually goes away */
13843 #define change_ldt(a,b) setup_arg_pages(a,b)
13844
13845 #endif /* __KERNEL__ */
13846 #endif /* _LINUX_BINFMTS_H */

```

include/linux/capability.h

```

13847 /*
13848  * This is <linux/capability.h>
13849  *
13850  * Andrew G. Morgan <morgan@transmeta.com>
13851  * Alexander Kjeldaas <astor@guardian.no>
13852  * with help from Aleph1, Roland Buresund and Andrew
13853  * Main. */
13854
13855 #ifndef _LINUX_CAPABILITY_H
13856 #define _LINUX_CAPABILITY_H
13857
13858 #include <linux/types.h>
13859 #include <linux/fs.h>
13860
13861 /* User-level do most of the mapping between kernel and
13862  * user capabilities based on the version tag given by
13863  * the kernel. The kernel might be somewhat backwards
13864  * compatible, but don't bet on it. */
13865
13866 /* XXX - Note, cap_t, is defined by POSIX to be an
13867  * "opaque" pointer to a set of three capability sets.
13868  * The transposition of 3*the following structure to such
13869  * a composite is better handled in a user library since
13870  * the draft standard requires the use of malloc/free
13871  * etc.. */
13872
13873 #define _LINUX_CAPABILITY_VERSION 0x19980330
13874
13875 typedef struct __user_cap_header_struct {
13876     __u32 version;

```

```

13877     int pid;
13878 } *cap_user_header_t;
13879
13880 typedef struct __user_cap_data_struct {
13881     __u32 effective;
13882     __u32 permitted;
13883     __u32 inheritable;
13884 } *cap_user_data_t;
13885
13886 #ifdef __KERNEL__
13887
13888 /* #define STRICT_CAP_T_TYPECHECKS */
13889
13890 #ifdef STRICT_CAP_T_TYPECHECKS
13891
13892 typedef struct kernel_cap_struct {
13893     __u32 cap;
13894 } kernel_cap_t;
13895
13896 #else
13897
13898 typedef __u32 kernel_cap_t;
13899
13900 #endif
13901
13902 #define _USER_CAP_HEADER_SIZE (2*sizeof(__u32))
13903 #define _KERNEL_CAP_T_SIZE (sizeof(kernel_cap_t))
13904
13905 #endif
13906
13907
13908 /**
13909  ** POSIX-draft defined capabilities.
13910  **/
13911
13912 /* In a system with the [_POSIX_CHOWN_RESTRICTED] option
13913  * defined, this overrides the restriction of changing
13914  * file ownership and group ownership. */
13915
13916 #define CAP_CHOWN 0
13917
13918 /* Override all DAC access, including ACL execute access
13919  * if [_POSIX_ACL] is defined. Excluding DAC access
13920  * covered by CAP_LINUX_IMMUTABLE. */
13921
13922 #define CAP_DAC_OVERRIDE 1
13923
13924 /* Overrides all DAC restrictions regarding read and

```

0.528

```

13925 search on files and directories, including ACL
13926 restrictions if [_POSIX_ACL] is defined. Excluding DAC
13927 access covered by CAP_LINUX_IMMUTABLE. */
13928
13929 #define CAP_DAC_READ_SEARCH 2
13930
13931 /* Overrides all restrictions about allowed operations on
13932 files, where file owner ID must be equal to the user
13933 ID, except where CAP_FSETID is applicable. It doesn't
13934 override MAC and DAC restrictions. */
13935
13936 #define CAP_FOWNER 3
13937
13938 /* Overrides the following restrictions that the
13939 effective user ID shall match the file owner ID when
13940 setting the S_ISUID and S_ISGID bits on that file;
13941 that the effective group ID (or one of the
13942 supplementary group IDs shall match the file owner ID
13943 when setting the S_ISGID bit on that file: that the
13944 S_ISUID and S_ISGID bits are cleared on successful
13945 return from chown(2). */
13946
13947 #define CAP_FSETID 4
13948
13949 /* Used to decide between falling back on the old suser()
13950 or fsuser(). */
13951
13952 #define CAP_FS_MASK 0x1f
13953
13954 /* Overrides the restriction that the real or effective
13955 user ID of a process sending a signal must match the
13956 real or effective user ID of the process receiving the
13957 signal. */
13958
13959 #define CAP_KILL 5
13960
13961 /* Allows setgid(2) manipulation */
13962 /* Allows setgroups(2) */
13963 /* Allows forged gids on socket credentials passing. */
13964
13965 #define CAP_SETGID 6
13966
13967 /* Allows set*uid(2) manipulation (including fsuid). */
13968 /* Allows forged pids on socket credentials passing. */
13969
13970 #define CAP_SETUID 7
13971
13972

```

```

13973 /**
13974 ** Linux-specific capabilities
13975 **/
13976
13977 /* Transfer any capability in your permitted set to any
13978 pid, remove any capability in your permitted set from
13979 any pid */
13980
13981 #define CAP_SETPCAP 8
13982
13983 /* Allow modification of S_IMMUTABLE and S_APPEND file
13984 attributes */
13985
13986 #define CAP_LINUX_IMMUTABLE 9
13987
13988 /* Allows binding to TCP/UDP sockets below 1024 */
13989
13990 #define CAP_NET_BIND_SERVICE 10
13991
13992 /* Allow broadcasting, listen to multicast */
13993
13994 #define CAP_NET_BROADCAST 11
13995
13996 /* Allow interface configuration */
13997 /* Allow administration of IP firewall, masquerading and
13998 accounting */
13999 /* Allow setting debug option on sockets */
14000 /* Allow modification of routing tables */
14001 /* Allow setting arbitrary process / process group
14002 ownership on sockets */
14003 /* Allow binding to any addr for transparent proxying */
14004 /* Allow setting TOS (type of service) */
14005 /* Allow setting promiscuous mode */
14006 /* Allow clearing driver statistics */
14007 /* Allow multicasting */
14008 /* Allow read/write of device-specific registers */
14009
14010 #define CAP_NET_ADMIN 12
14011
14012 /* Allow use of RAW sockets */
14013 /* Allow use of PACKET sockets */
14014
14015 #define CAP_NET_RAW 13
14016
14017 /* Allow locking of shared memory segments */
14018 /* Allow mlock and mlockall (which doesn't really have
14019 anything to do with IPC) */
14020

```



```

14021 #define CAP_IPC_LOCK          14
14022
14023 /* Override IPC ownership checks */
14024
14025 #define CAP_IPC_OWNER          15
14026
14027 /* Insert and remove kernel modules */
14028
14029 #define CAP_SYS_MODULE        16
14030
14031 /* Allow ioperm/iopl access */
14032
14033 #define CAP_SYS_RAWIO         17
14034
14035 /* Allow use of chroot() */
14036
14037 #define CAP_SYS_CHROOT        18
14038
14039 /* Allow ptrace() of any process */
14040
14041 #define CAP_SYS_PTRACE        19
14042
14043 /* Allow configuration of process accounting */
14044
14045 #define CAP_SYS_PACCT         20
14046
14047 /* Allow configuration of the secure attention key */
14048 /* Allow administration of the random device */
14049 /* Allow device administration (mknod)*/
14050 /* Allow examination and configuration of disk quotas */
14051 /* Allow configuring the kernel's syslog (printk
14052  behaviour) */
14053 /* Allow sending a signal to any process */
14054 /* Allow setting the domainname */
14055 /* Allow setting the hostname */
14056 /* Allow calling bdflush() */
14057 /* Allow mount() and umount(), setting up new smb
14058  connection */
14059 /* Allow some autofs root ioctls */
14060 /* Allow nfsservctl */
14061 /* Allow VMB6_REQUEST_IRQ */
14062 /* Allow to read/write pci config on alpha */
14063 /* Allow irix_prctl on mips (setstacksize) */
14064 /* Allow flushing all cache on m68k (sys_cacheflush) */
14065 /* Allow removing semaphores */
14066 /* Used instead of CAP_CHOWN to "chown" IPC message
14067  queues, semaphores and shared memory */
14068 /* Allow locking/unlocking of shared memory segment */

```

```

14069 /* Allow turning swap on/off */
14070 /* Allow forged pids on socket credentials passing */
14071 /* Allow setting readahead and flushing buffers on block
14072  devices */
14073 /* Allow setting geometry in floppy driver */
14074 /* Allow turning DMA on/off in xd driver */
14075 /* Allow administration of md devices (mostly the above,
14076  but some extra ioctls) */
14077 /* Allow tuning the ide driver */
14078 /* Allow access to the nvram device */
14079 /* Allow administration of apm_bios, serial and bttv (TV)
14080  device */
14081 /* Allow manufacturer commands in isdn CAPI support
14082  driver */
14083 /* Allow reading non-standardized portions of pci
14084  configuration space */
14085 /* Allow DDI debug ioctl on sbpcd driver */
14086 /* Allow setting up serial ports */
14087 /* Allow sending raw qic-117 commands */
14088 /* Allow enabling/disabling tagged queuing on SCSI
14089  controllers and sending arbitrary SCSI commands */
14090 /* Allow setting encryption key on loopback filesystem */
14091
14092 #define CAP_SYS_ADMIN          21
14093
14094 /* Allow use of reboot() */
14095
14096 #define CAP_SYS_BOOT          22
14097
14098 /* Allow raising priority and setting priority on other
14099  (different UID) processes */
14100 /* Allow use of FIFO and round-robin (realtime)
14101  scheduling on own processes and setting the scheduling
14102  algorithm used by another process. */
14103
14104 #define CAP_SYS_NICE          23
14105
14106 /* Override resource limits. Set resource limits. */
14107 /* Override quota limits. */
14108 /* Override reserved space on ext2 filesystem */
14109 /* NOTE: ext2 honors fsuid when checking for resource
14110  overrides, so you can override using fsuid too */
14111 /* Override size restrictions on IPC message queues */
14112 /* Allow more than 64hz interrupts from the real-time
14113  clock */
14114 /* Override max # of consoles on console allocation */
14115 /* Override max # of keymaps */
14116

```

```

14117 #define CAP_SYS_RESOURCE      24
14118
14119 /* Allow manipulation of system clock */
14120 /* Allow irix_stime on mips */
14121 /* Allow setting the real-time clock */
14122
14123 #define CAP_SYS_TIME            25
14124
14125 /* Allow configuration of tty devices */
14126 /* Allow vhangup() of tty */
14127
14128 #define CAP_SYS_TTY_CONFIG      26
14129
14130 #ifdef __KERNEL__
14131
14132 /* Internal kernel functions only */
14133
14134 #ifdef STRICT_CAP_T_TYPECHECKS
14135
14136 #define to_cap_t(x) { x }
14137 #define cap_t(x) (x).cap
14138
14139 #else
14140
14141 #define to_cap_t(x) (x)
14142 #define cap_t(x) (x)
14143
14144 #endif
14145
14146 #define CAP_EMPTY_SET    to_cap_t(0)
14147 #define CAP_FULL_SET    to_cap_t(~0)
14148 #define CAP_INIT_EFF_SET to_cap_t(~0 &
14149                               ~CAP_TO_MASK(CAP_SETPCAP))
14150 #define CAP_INIT_INH_SET to_cap_t(~0 &
14151                               ~CAP_TO_MASK(CAP_SETPCAP))
14152
14153 #define CAP_TO_MASK(x) (1 << (x))
14154 #define cap_raise(c, flag) \
14155     (cap_t(c) |= CAP_TO_MASK(flag))
14156 #define cap_lower(c, flag) \
14157     (cap_t(c) &= ~CAP_TO_MASK(flag))
14158 #define cap_raised(c, flag) \
14159     (cap_t(c) & CAP_TO_MASK(flag))
14160
14161 static inline kernel_cap_t cap_combine(kernel_cap_t a,
14162                                       kernel_cap_t b)
14163 {
14164     kernel_cap_t dest;

```

```

14165     cap_t(dest) = cap_t(a) | cap_t(b);
14166     return dest;
14167 }
14168
14169 static inline kernel_cap_t cap_intersect(kernel_cap_t a,
14170                                       kernel_cap_t b)
14171 {
14172     kernel_cap_t dest;
14173     cap_t(dest) = cap_t(a) & cap_t(b);
14174     return dest;
14175 }
14176
14177 static inline kernel_cap_t cap_drop(kernel_cap_t a,
14178                                    kernel_cap_t drop)
14179 {
14180     kernel_cap_t dest;
14181     cap_t(dest) = cap_t(a) & ~cap_t(drop);
14182     return dest;
14183 }
14184
14185 static inline kernel_cap_t cap_invert(kernel_cap_t c)
14186 {
14187     kernel_cap_t dest;
14188     cap_t(dest) = ~cap_t(c);
14189     return dest;
14190 }
14191
14192 #define cap_isclear(c)    (!cap_t(c))
14193 #define cap_issubset(a,set) (!(cap_t(a) & ~cap_t(set)))
14194
14195 #define cap_clear(c)      do { cap_t(c) = 0; } while(0)
14196 #define cap_set_full(c)   do { cap_t(c) = ~0; } while(0)
14197 #define cap_mask(c,mask)  do { cap_t(c) &= cap_t(mask); \
14198                               } while(0)
14199
14200 #define cap_is_fs_cap(c) (CAP_TO_MASK(c) & CAP_FS_MASK)
14201
14202 #endif /* __KERNEL__ */
14203
14204 #endif /* !_LINUX_CAPABILITY_H */

include/linux/elf.h
14205 #ifndef _LINUX_ELF_H
14206 #define _LINUX_ELF_H
14207
14208 #include <linux/types.h>
14209 #include <asm/elf.h>

```

```

14210
14211 /* 32-bit ELF base types. */
14212 typedef __u32   Elf32_Addr;
14213 typedef __u16   Elf32_Half;
14214 typedef __u32   Elf32_Off;
14215 typedef __s32   Elf32_Sword;
14216 typedef __u32   Elf32_Word;
14217
14218 /* 64-bit ELF base types. */
14219 typedef __u64   Elf64_Addr;
14220 typedef __u16   Elf64_Half;
14221 typedef __s16   Elf64_SHalf;
14222 typedef __u64   Elf64_Off;
14223 typedef __s64   Elf64_Sword;
14224 typedef __u64   Elf64_Word;
14225
14226 /* These constants are for the segment types stored in
14227  * the image headers */
14228 #define PT_NULL    0
14229 #define PT_LOAD    1
14230 #define PT_DYNAMIC 2
14231 #define PT_INTERP  3
14232 #define PT_NOTE    4
14233 #define PT_SHLIB   5
14234 #define PT_PHDR    6
14235 #define PT_LOPROC 0x70000000
14236 #define PT_HIPROC 0x7fffffff
14237
14238 /* These constants define the different elf file types */
14239 #define ET_NONE    0
14240 #define ET_REL     1
14241 #define ET_EXEC    2
14242 #define ET_DYN     3
14243 #define ET_CORE    4
14244 #define ET_LOPROC 5
14245 #define ET_HIPROC 6
14246
14247 /* These constants define the various ELF target machs */
14248 #define EM_NONE    0
14249 #define EM_M32     1
14250 #define EM_SPARC   2
14251 #define EM_386     3
14252 #define EM_68K     4
14253 #define EM_88K     5
14254 #define EM_486     6 /* Perhaps disused */
14255 #define EM_860     7
14256
14257 /* MIPS R3000 (officially, big-endian only) */
14258 #define EM_MIPS    8
14259
14260 #define EM_MIPS_RS4_BE 10 /* MIPS R4000 big-endian */
14261
14262 #define EM_PARISC   15 /* HPPA */
14263
14264 #define EM_SPARC32PLUS 18 /* Sun's "v8plus" */
14265
14266 #define EM_PPC      20 /* PowerPC */
14267
14268 #define EM_SPARCV9   43 /* SPARC v9 64-bit */
14269
14270 /* This is an interim value that we will use until the
14271  * committee comes up with a final number. */
14272 #define EM_ALPHA    0x9026
14273
14274
14275 /* This is the info that is needed to parse the dynamic
14276  * section of the file */
14277 #define DT_NULL      0
14278 #define DT_NEEDED    1
14279 #define DT_PLTRELSZ  2
14280 #define DT_PLTGOT    3
14281 #define DT_HASH      4
14282 #define DT_STRTAB    5
14283 #define DT_SYMTAB    6
14284 #define DT_RELA      7
14285 #define DT_RELASZ    8
14286 #define DT_RELAENT   9
14287 #define DT_STRSZ     10
14288 #define DT_SYMENT    11
14289 #define DT_INIT      12
14290 #define DT_FINI      13
14291 #define DT_SONAME    14
14292 #define DT_RPATH     15
14293 #define DT_SYMBOLIC  16
14294 #define DT_REL       17
14295 #define DT_RELSZ     18
14296 #define DT_RELENT    19
14297 #define DT_PLTREL    20
14298 #define DT_DEBUG     21
14299 #define DT_TEXTREL   22
14300 #define DT_JMPREL    23
14301 #define DT_LOPROC    0x70000000
14302 #define DT_HIPROC    0x7fffffff
14303
14304 /* This info is needed when parsing the symbol table */
14305 #define STB_LOCAL    0

```

```

14306 #define STB_GLOBAL 1
14307 #define STB_WEAK 2
14308
14309 #define STT_NOTYPE 0
14310 #define STT_OBJECT 1
14311 #define STT_FUNC 2
14312 #define STT_SECTION 3
14313 #define STT_FILE 4
14314
14315 #define ELF32_ST_BIND(x) ((x) >> 4)
14316 #define ELF32_ST_TYPE(x) (((unsigned int) x) & 0xf)
14317
14318 /* Symbolic values for the entries in the auxiliary table
14319 put on the initial stack */
14320 #define AT_NULL 0 /* end of vector */
14321 #define AT_IGNORE 1 /* entry should be ignored */
14322 #define AT_EXECFD 2 /* file descriptor of program */
14323 #define AT_PHDR 3 /* program headers for program */
14324 #define AT_PHENT 4 /* size of program header entry */
14325 #define AT_PHNUM 5 /* number of program headers */
14326 #define AT_PAGESZ 6 /* system page size */
14327 #define AT_BASE 7 /* base address of interpreter */
14328 #define AT_FLAGS 8 /* flags */
14329 #define AT_ENTRY 9 /* entry point of program */
14330 #define AT_NOTELF 10 /* program is not ELF */
14331 #define AT_UID 11 /* real uid */
14332 #define AT_EUID 12 /* effective uid */
14333 #define AT_GID 13 /* real gid */
14334 #define AT_EGID 14 /* effective gid */
14335 #define AT_PLATFORM 15 /* string identifying CPU for
14336 * optimizations */
14337 #define AT_HWCAP 16 /* arch dependent hints at CPU
14338 * capabilities */
14339
14340 typedef struct dynamic{
14341 Elf32_Sword d_tag;
14342 union{
14343 Elf32_Sword d_val;
14344 Elf32_Addr d_ptr;
14345 } d_un;
14346 } Elf32_Dyn;
14347
14348 typedef struct {
14349 Elf64_Word d_tag; /* entry tag value */
14350 union {
14351 Elf64_Word d_val;
14352 Elf64_Word d_ptr;
14353 } d_un;

```

```

14354 } Elf64_Dyn;
14355
14356 /* The following are used with relocations */
14357 #define ELF32_R_SYM(x) ((x) >> 8)
14358 #define ELF32_R_TYPE(x) ((x) & 0xff)
14359
14360 #define R_386_NONE 0
14361 #define R_386_32 1
14362 #define R_386_PC32 2
14363 #define R_386_GOT32 3
14364 #define R_386_PLT32 4
14365 #define R_386_COPY 5
14366 #define R_386_GLOB_DAT 6
14367 #define R_386_JMP_SLOT 7
14368 #define R_386_RELATIVE 8
14369 #define R_386_GOTOFF 9
14370 #define R_386_GOTPC 10
14371 #define R_386_NUM 11
14372
14373 /* Sparc ELF relocation types */
14374 #define R_SPARC_NONE 0
14375 #define R_SPARC_8 1
14376 #define R_SPARC_16 2
14377 #define R_SPARC_32 3
14378 #define R_SPARC_DISP8 4
14379 #define R_SPARC_DISP16 5
14380 #define R_SPARC_DISP32 6
14381 #define R_SPARC_WDISP30 7
14382 #define R_SPARC_WDISP22 8
14383 #define R_SPARC_HI22 9
14384 #define R_SPARC_22 10
14385 #define R_SPARC_13 11
14386 #define R_SPARC_LO10 12
14387 #define R_SPARC_GOT10 13
14388 #define R_SPARC_GOT13 14
14389 #define R_SPARC_GOT22 15
14390 #define R_SPARC_PC10 16
14391 #define R_SPARC_PC22 17
14392 #define R_SPARC_WPLT30 18
14393 #define R_SPARC_COPY 19
14394 #define R_SPARC_GLOB_DAT 20
14395 #define R_SPARC_JMP_SLOT 21
14396 #define R_SPARC_RELATIVE 22
14397 #define R_SPARC_UA32 23
14398 #define R_SPARC_PLT32 24
14399 #define R_SPARC_HIPLT22 25
14400 #define R_SPARC_LOPLT10 26
14401 #define R_SPARC_PCPLT32 27

```

```

14402 #define R_SPARC_PCPLT22      28
14403 #define R_SPARC_PCPLT10     29
14404 #define R_SPARC_10          30
14405 #define R_SPARC_11          31
14406 #define R_SPARC_WDISP16     40
14407 #define R_SPARC_WDISP19     41
14408 #define R_SPARC_7            43
14409 #define R_SPARC_5            44
14410 #define R_SPARC_6            45
14411
14412 /* Bits present in AT_HWCAP, primarily for Sparc32. */
14413
14414 #define HWCAP_SPARC_FLUSH      1 /* CPU supports flush
14415                               * instruction. */
14416 #define HWCAP_SPARC_STBAR      2
14417 #define HWCAP_SPARC_SWAP      4
14418 #define HWCAP_SPARC_MULDIV     8
14419 #define HWCAP_SPARC_V9        16
14420
14421
14422 /* 68k ELF relocation types */
14423 #define R_68K_NONE             0
14424 #define R_68K_32               1
14425 #define R_68K_16              2
14426 #define R_68K_B               3
14427 #define R_68K_PC32           4
14428 #define R_68K_PC16           5
14429 #define R_68K_PC8            6
14430 #define R_68K_GOT32          7
14431 #define R_68K_GOT16          8
14432 #define R_68K_GOT8           9
14433 #define R_68K_GOT320         10
14434 #define R_68K_GOT160         11
14435 #define R_68K_GOT80          12
14436 #define R_68K_PLT32          13
14437 #define R_68K_PLT16          14
14438 #define R_68K_PLT8           15
14439 #define R_68K_PLT320         16
14440 #define R_68K_PLT160         17
14441 #define R_68K_PLT80          18
14442 #define R_68K_COPY           19
14443 #define R_68K_GLOB_DAT        20
14444 #define R_68K_JMP_SLOT        21
14445 #define R_68K_RELATIVE        22
14446
14447 /* Alpha ELF relocation types */
14448 #define R_ALPHA_NONE           0 /* No reloc */
14449 #define R_ALPHA_REFLONG        1 /* Direct 32 bit */
14450 #define R_ALPHA_REFQUAD        2 /* Direct 64 bit */
14451 #define R_ALPHA_GPREL32        3 /* GP relative 32 bit*/
14452 #define R_ALPHA_LITERAL        4 /* GP relative 16 bit
14453                               * w/optimization */
14454 #define R_ALPHA_LITUSE         5 /* Optimization hint
14455                               * for LITERAL */
14456 #define R_ALPHA_GPREL16        6 /* Add displacement to
14457                               * GP */
14458 #define R_ALPHA_BRADDR         7 /* PC+4 relative 23 bit
14459                               * shifted */
14460 #define R_ALPHA_HINT           8 /* PC+4 relative 16 bit
14461                               * shifted */
14462 #define R_ALPHA_SREL16         9 /* PC relative 16 bit*/
14463 #define R_ALPHA_SREL32        10 /* PC relative 32 bit*/
14464 #define R_ALPHA_SREL64        11 /* PC relative 64 bit*/
14465 #define R_ALPHA_OP_PUSH        12 /* OP stack push */
14466 #define R_ALPHA_OP_STORE       13 /* OP stack pop and
14467                               * store */
14468 #define R_ALPHA_OP_PSUB        14 /* OP stack subtract */
14469 #define R_ALPHA_OP_PRSHIFT     15 /* OP stack right shift
14470                               */
14471 #define R_ALPHA_GPVALUE        16
14472 #define R_ALPHA_GPRELHIGH      17
14473 #define R_ALPHA_GPRELLOW      18
14474 #define R_ALPHA_IMMED_GP_16    19
14475 #define R_ALPHA_IMMED_GP_HI32  20
14476 #define R_ALPHA_IMMED_SCN_HI32 21
14477 #define R_ALPHA_IMMED_BR_HI32 22
14478 #define R_ALPHA_IMMED_LO32     23
14479 #define R_ALPHA_COPY           24 /* Copy symbol at
14480                               * runtime */
14481 #define R_ALPHA_GLOB_DAT        25 /* Create GOT entry */
14482 #define R_ALPHA_JMP_SLOT        26 /* Create PLT entry */
14483 #define R_ALPHA_RELATIVE        27 /* Adjust by program
14484                               * base */
14485
14486 /* Legal values for e_flags field of Elf64_Ehdr. */
14487
14488 #define EF_ALPHA_32BIT          1 /* All addresses are
14489                               * below 2GB */
14490
14491
14492 typedef struct elf32_rel {
14493     Elf32_Addr    r_offset;
14494     Elf32_Word    r_info;
14495 } Elf32_Rel;
14496
14497 typedef struct elf64_rel {

```

```

14498  /* Location at which to apply the action */
14499  Elf64_Addr r_offset;
14500  Elf64_Word r_info; /* index and type of relocation */
14501 } Elf64_Rel;
14502
14503 typedef struct elf32_rela {
14504   Elf32_Addr r_offset;
14505   Elf32_Word r_info;
14506   Elf32_Sword r_addend;
14507 } Elf32_Rela;
14508
14509 typedef struct elf64_rela {
14510   /* Location at which to apply the action */
14511   Elf64_Addr r_offset;
14512   /* index and type of relocation */
14513   Elf64_Word r_info;
14514   /* Constant addend used to compute value */
14515   Elf64_Word r_addend;
14516 } Elf64_Rela;
14517
14518 typedef struct elf32_sym {
14519   Elf32_Word st_name;
14520   Elf32_Addr st_value;
14521   Elf32_Word st_size;
14522   unsigned char st_info;
14523   unsigned char st_other;
14524   Elf32_Half st_shndx;
14525 } Elf32_Sym;
14526
14527 typedef struct elf64_sym {
14528   Elf32_Word st_name; /* Symbol name, index in
14529                       * string tbl (yes, Elf32) */
14530   unsigned char st_info; /* Type and binding
14531                          * attributes */
14532   unsigned char st_other; /* No defined meaning, 0 */
14533   Elf64_Half st_shndx; /* Associated section index */
14534   Elf64_Addr st_value; /* Value of the symbol */
14535   Elf64_Word st_size; /* Associated symbol size */
14536 } Elf64_Sym;
14537
14538
14539 #define EI_NIDENT 16
14540
14541 typedef struct elf32_hdr {
14542   unsigned char e_ident[EI_NIDENT];
14543   Elf32_Half e_type;
14544   Elf32_Half e_machine;
14545   Elf32_Word e_version;
14546   Elf32_Addr e_entry; /* Entry point */
14547   Elf32_Off e_phoff;
14548   Elf32_Off e_shoff;
14549   Elf32_Word e_flags;
14550   Elf32_Half e_ehsize;
14551   Elf32_Half e_phentsize;
14552   Elf32_Half e_phnum;
14553   Elf32_Half e_shentsize;
14554   Elf32_Half e_shnum;
14555   Elf32_Half e_shstrndx;
14556 } Elf32_Ehdr;
14557
14558 typedef struct elf64_hdr {
14559   unsigned char e_ident[16]; /* ELF "magic number" */
14560   Elf64_SHalf e_type;
14561   Elf64_Half e_machine;
14562   __s32 e_version;
14563   Elf64_Addr e_entry; /* Entry point virtual address */
14564   Elf64_Off e_phoff; /* Program hdr table file offset */
14565   Elf64_Off e_shoff; /* Section hdr table file offset */
14566   __s32 e_flags;
14567   Elf64_SHalf e_ehsize;
14568   Elf64_SHalf e_phentsize;
14569   Elf64_SHalf e_phnum;
14570   Elf64_SHalf e_shentsize;
14571   Elf64_SHalf e_shnum;
14572   Elf64_SHalf e_shstrndx;
14573 } Elf64_Ehdr;
14574
14575 /* These constants define the permissions on sections in
14576    the program header, p_flags. */
14577 #define PF_R 0x4
14578 #define PF_W 0x2
14579 #define PF_X 0x1
14580
14581 typedef struct elf32_phdr {
14582   Elf32_Word p_type;
14583   Elf32_Off p_offset;
14584   Elf32_Addr p_vaddr;
14585   Elf32_Addr p_paddr;
14586   Elf32_Word p_filesz;
14587   Elf32_Word p_memsz;
14588   Elf32_Word p_flags;
14589   Elf32_Word p_align;
14590 } Elf32_Phdr;
14591
14592 typedef struct elf64_phdr {
14593   __s32 p_type;

```

```

14594 __s32 p_flags;
14595 Elf64_Off p_offset; /* Segment file offset */
14596 Elf64_Addr p_vaddr; /* Segment virtual address */
14597 Elf64_Addr p_paddr; /* Segment physical address */
14598 Elf64_Word p_filesz; /* Segment size in file */
14599 Elf64_Word p_memsz; /* Segment size in memory */
14600 Elf64_Word p_align; /* Segment alignment,
14601 * file & memory */
14602 } Elf64_Phdr;
14603
14604 /* sh_type */
14605 #define SHT_NULL 0
14606 #define SHT_PROGBITS 1
14607 #define SHT_SYMTAB 2
14608 #define SHT_STRTAB 3
14609 #define SHT_RELA 4
14610 #define SHT_HASH 5
14611 #define SHT_DYNAMIC 6
14612 #define SHT_NOTE 7
14613 #define SHT_NOBITS 8
14614 #define SHT_REL 9
14615 #define SHT_SHLIB 10
14616 #define SHT_DYNSYM 11
14617 #define SHT_NUM 12
14618 #define SHT_LOPROC 0x70000000
14619 #define SHT_HIPROC 0x7fffffff
14620 #define SHT_LOUSER 0x80000000
14621 #define SHT_HIUSER 0xffffffff
14622
14623 /* sh_flags */
14624 #define SHF_WRITE 0x1
14625 #define SHF_ALLOC 0x2
14626 #define SHF_EXECINSTR 0x4
14627 #define SHF_MASKPROC 0xf0000000
14628
14629 /* special section indexes */
14630 #define SHN_UNDEF 0
14631 #define SHN_LORESERVE 0xfff0
14632 #define SHN_LOPROC 0xff0
14633 #define SHN_HIPROC 0xff1f
14634 #define SHN_ABS 0xffff1
14635 #define SHN_COMMON 0xffff2
14636 #define SHN_HIRESERVE 0xfffff
14637
14638 typedef struct {
14639 Elf32_Word sh_name;
14640 Elf32_Word sh_type;
14641 Elf32_Word sh_flags;

```

```

14642 Elf32_Addr sh_addr;
14643 Elf32_Off sh_offset;
14644 Elf32_Word sh_size;
14645 Elf32_Word sh_link;
14646 Elf32_Word sh_info;
14647 Elf32_Word sh_addralign;
14648 Elf32_Word sh_entsize;
14649 } Elf32_Shdr;
14650
14651 typedef struct elf64_shdr {
14652 Elf32_Word sh_name; /* Section name, index in
14653 * string tbl (yes Elf32) */
14654 Elf32_Word sh_type; /* Type of section
14655 * (yes Elf32) */
14656 Elf64_Word sh_flags; /* Miscellaneous section
14657 * attributes */
14658 Elf64_Addr sh_addr; /* Section virtual addr at
14659 * execution */
14660 Elf64_Off sh_offset; /* Section file offset */
14661 Elf64_Word sh_size; /* Size of section in bytes */
14662 Elf32_Word sh_link; /* Index of another section
14663 * (yes Elf32) */
14664 Elf32_Word sh_info; /* Additional section
14665 * information (yes Elf32) */
14666 Elf64_Word sh_addralign; /* Section alignment */
14667 Elf64_Word sh_entsize; /* Entry size if section holds
14668 * table */
14669 } Elf64_Shdr;
14670
14671 #define EI_MAG0 0 /* e_ident[] indexes */
14672 #define EI_MAG1 1
14673 #define EI_MAG2 2
14674 #define EI_MAG3 3
14675 #define EI_CLASS 4
14676 #define EI_DATA 5
14677 #define EI_VERSION 6
14678 #define EI_PAD 7
14679
14680 #define ELFMAG 0x7f /* EI_MAG */
14681 #define ELF_MAG1 'E'
14682 #define ELF_MAG2 'L'
14683 #define ELF_MAG3 'F'
14684 #define ELFMAG "\177ELF"
14685 #define SELF_MAG 4
14686
14687 #define ELFCLASSNONE 0 /* EI_CLASS */
14688 #define ELFCLASS32 1
14689 #define ELFCLASS64 2

```

```

14690 #define ELFCLASSNUM      3
14691
14692 #define ELFDATANONE        0          /* e_ident[EI_DATA] */
14693 #define ELFDATA2LSB        1
14694 #define ELFDATA2MSB        2
14695
14696 #define EV_NONE             0          /* e_version, EI_VERSION */
14697 #define EV_CURRENT          1
14698 #define EV_NUM              2
14699
14700 /* Notes used in ET_CORE */
14701 #define NT_PRSTATUS         1
14702 #define NT_PRFPREG          2
14703 #define NT_PRPSINFO         3
14704 #define NT_TASKSTRUCT       4
14705
14706 /* Note header in a PT_NOTE section */
14707 typedef struct elf32_note {
14708     Elf32_Word  n_namesz;      /* Name size */
14709     Elf32_Word  n_descsz;     /* Content size */
14710     Elf32_Word  n_type;       /* Content type */
14711 } Elf32_Nhdr;
14712
14713 /* Note header in a PT_NOTE section */
14714 /* For now we use the 32 bit version of the structure
14715  * until we figure out whether we need anything better.
14716  * Note - on the Alpha, "unsigned int" is only 32 bits.*/
14717 typedef struct elf64_note {
14718     Elf32_Word n_namesz; /* Name size */
14719     Elf32_Word n_descsz; /* Content size */
14720     Elf32_Word n_type;   /* Content type */
14721 } Elf64_Nhdr;
14722
14723 #if ELF_CLASS == ELFCLASS32
14724
14725 extern Elf32_Dyn _DYNAMIC [];
14726 #define elfhdr      elf32_hdr
14727 #define elf_phdr    elf32_phdr
14728 #define elf_note    elf32_note
14729
14730 #else
14731
14732 extern Elf64_Dyn _DYNAMIC [];
14733 #define elfhdr      elf64_hdr
14734 #define elf_phdr    elf64_phdr
14735 #define elf_note    elf64_note
14736
14737 #endif

```

```

14738
14739
14740 #endif /* _LINUX_ELF_H */

```

include/linux/elfcore.h

```

14741 #ifndef _LINUX_ELFCORE_H
14742 #define _LINUX_ELFCORE_H
14743
14744 #include <linux/types.h>
14745 #include <linux/signal.h>
14746 #include <linux/time.h>
14747 #include <linux/ptrace.h>
14748 #include <linux/user.h>
14749
14750 struct elf_siginfo
14751 {
14752     int     si_signo; /* signal number */
14753     int     si_code; /* extra code */
14754     int     si_errno; /* errno */
14755 };
14756
14757 #include <asm/elf.h>
14758
14759 #ifndef _KERNEL
14760 typedef elf_greg_t greg_t;
14761 typedef elf_gregset_t gregset_t;
14762 typedef elf_fpregset_t fpregset_t;
14763 #define NGREG ELF_NGREG
14764 #endif
14765
14766 /* Definitions to generate Intel SVR4 like core files.
14767  * These mostly have the same names as the SVR4 types
14768  * with "elf_" tacked on the front to prevent clashes
14769  * with linux definitions, and the typedef forms have
14770  * been avoided. This is mostly like the SVR4 structure,
14771  * but more Linuxy, with things that linux does not
14772  * support and which gdb doesn't really use excluded.
14773  * Fields present but not used are marked with "XXX". */
14774 struct elf_prstatus
14775 {
14776     #if 0
14777     long    pr_flags;      /* XXX Process flags */
14778     short   pr_why;       /* XXX Reason for process halt */
14779     short   pr_what;      /* XXX More detailed reason */
14780     #endif
14781     struct elf_siginfo pr_info; /* Info associated with
14782     * signal */

```



```

14783 short pr_cursig; /* Current signal */
14784 unsigned long pr_sigpend; /* Set of pending
14785 * signals */
14786 unsigned long pr_sighold; /* Set of held signals
14787 */
14788 #if 0
14789 struct sigaltstack pr_altstack; /* Alternate stack info
14790 */
14791 struct sigaction pr_action; /* Signal action for
14792 * current sig */
14793 #endif
14794 pid_t pr_pid;
14795 pid_t pr_ppid;
14796 pid_t pr_pgrp;
14797 pid_t pr_sid;
14798 struct timeval pr_utime; /* User time */
14799 struct timeval pr_stime; /* System time */
14800 struct timeval pr_cutime; /* Cumulative user time */
14801 struct timeval pr_cstime; /* Cumulative system time */
14802 #if 0
14803 long pr_instr; /* Current instruction */
14804 #endif
14805 elf_gregset_t pr_reg; /* GP registers */
14806 int pr_fpvalid; /* True if math co-processor
14807 * being used. */
14808 };
14809
14810 #define ELF_PRARGSZ (80) /* Number of chars for
14811 * args */
14812
14813 struct elf_prpsinfo
14814 {
14815 char pr_state; /* numeric process state */
14816 char pr_sname; /* char for pr_state */
14817 char pr_zomb; /* zombie */
14818 char pr_nice; /* nice val */
14819 unsigned long pr_flag; /* flags */
14820 uid_t pr_uid;
14821 gid_t pr_gid;
14822 pid_t pr_pid, pr_ppid, pr_pgrp, pr_sid;
14823 /* Lots missing */
14824 char pr_fname[16]; /* filename of executable */
14825 char pr_psargs[ELF_PRARGSZ]; /* initial part of
14826 * arg list */
14827 };
14828
14829 #ifndef _KERNEL
14830 typedef struct elf_prstatus prstatus_t;

```

```

14831 typedef struct elf_prpsinfo prpsinfo_t;
14832 #define PRARGSZ ELF_PRARGSZ
14833 #endif
14834
14835 #endif /* _LINUX_ELFCORE_H */

```

include/linux/interrupt.h

```

14836 /* interrupt.h */
14837 #ifndef _LINUX_INTERRUPT_H
14838 #define _LINUX_INTERRUPT_H
14839
14840 #include <linux/kernel.h>
14841 #include <asm/bitops.h>
14842 #include <asm/atomic.h>
14843
14844 struct irqaction {
14845 void (*handler)(int, void *, struct pt_regs *);
14846 unsigned long flags;
14847 unsigned long mask;
14848 const char *name;
14849 void *dev_id;
14850 struct irqaction *next;
14851 };
14852
14853 extern volatile unsigned char bh_running;
14854
14855 extern atomic_t bh_mask_count[32];
14856 extern unsigned long bh_active;
14857 extern unsigned long bh_mask;
14858 extern void (*bh_base[32])(void);
14859
14860 asmlinkage void do_bottom_half(void);
14861
14862 /* Who gets which entry in bh_base. Things which will
14863 occur most often should come first - in which case NET
14864 should be up the top with SERIAL/TQUEUE! */
14865
14866 enum {
14867 TIMER_BH = 0,
14868 CONSOLE_BH,
14869 TQUEUE_BH,
14870 DIGI_BH,
14871 SERIAL_BH,
14872 RISCOM8_BH,
14873 SPECIALIX_BH,
14874 ESP_BH,
14875 NET_BH,

```

```

14876 SCSI_BH,
14877 IMMEDIATE_BH,
14878 KEYBOARD_BH,
14879 CYCLADES_BH,
14880 CM206_BH,
14881 JS_BH,
14882 MACSERIAL_BH,
14883 ISICOM_BH
14884 };
14885
14886 #include <asm/hardirq.h>
14887 #include <asm/softirq.h>
14888
14889 /* Autoprobing for irqs:
14890 *
14891 * probe_irq_on() and probe_irq_off() provide robust
14892 * primitives for accurate IRQ probing during kernel
14893 * initialization. They are reasonably simple to use,
14894 * are not "fooled" by spurious interrupts, and, unlike
14895 * other attempts at IRQ probing, they do not get hung on
14896 * stuck interrupts (such as unused PS2 mouse interfaces
14897 * on ASUS boards).
14898 *
14899 * For reasonably foolproof probing, use them as follows:
14900 *
14901 * 1. clear and/or mask the device's internal interrupt.
14902 * 2. sti();
14903 * 3. irqs = probe_irq_on();
14904 * // "take over" all unassigned idle IRQs
14905 * 4. enable the device and cause it to trigger
14906 * an interrupt.
14907 * 5. wait for the device to interrupt, using
14908 * non-intrusive polling or a delay.
14909 * 6. irq = probe_irq_off(irqs);
14910 * // get IRQ number, 0=none, negative=multiple
14911 * 7. service the device to clear its pending interrupt.
14912 * 8. loop again if paranoia is required.
14913 *
14914 * probe_irq_on() returns a mask of allocated irq's.
14915 *
14916 * probe_irq_off() takes the mask as a parameter,
14917 * and returns the irq number which occurred,
14918 * or zero if none occurred, or a negative irq number
14919 * if more than one irq occurred. */
14920 extern unsigned long probe_irq_on(void); /*== 0 on fail*/
14921 extern int probe_irq_off(unsigned long); /*<= 0 on fail*/
14922
14923 #endif

```

include/linux/kernel.h

```

14924 #ifndef _LINUX_KERNEL_H
14925 #define _LINUX_KERNEL_H
14926
14927 /* 'kernel.h' contains some often-used function
14928 * prototypes etc */
14929
14930 #ifdef __KERNEL__
14931
14932 #include <stdarg.h>
14933 #include <linux/linkage.h>
14934
14935 /* Optimization barrier */
14936 /* The "volatile" is due to gcc bugs */
14937 #define barrier() __asm__ __volatile__("" : : "memory")
14938
14939 #define INT_MAX ((int)(-0U>>1))
14940 #define UINT_MAX (~0U)
14941 #define LONG_MAX ((long)(~0UL>>1))
14942 #define ULONG_MAX (~0UL)
14943
14944 #define STACK_MAGIC 0xdeadbeef
14945
14946 #define KERN_EMERG "<0>" /* system is unusable */
14947 #define KERN_ALERT "<1>" /* need immediate action */
14948 #define KERN_CRIT "<2>" /* critical conditions */
14949 #define KERN_ERR "<3>" /* error conditions */
14950 #define KERN_WARNING "<4>" /* warning conditions */
14951 #define KERN_NOTICE "<5>" /* normal but significant */
14952 #define KERN_INFO "<6>" /* informational */
14953 #define KERN_DEBUG "<7>" /* debug-level messages */
14954
14955 # define NORET_TYPE /**/
14956 # define ATTRIB_NORET __attribute__((noreturn))
14957 # define NORET_AND noreturn,
14958
14959 #ifdef __i386__
14960 #define FASTCALL(x) x __attribute__((regparm(3)))
14961 #else
14962 #define FASTCALL(x) x
14963 #endif
14964
14965 extern void math_error(void);
14966 NORET_TYPE void panic(const char * fmt, ...)
14967 __attribute__((NORET_AND format (printf, 1, 2)));
14968 NORET_TYPE void do_exit(long error_code)
14969 ATTRIB_NORET;

```

```

14970 extern unsigned long simple_strtoul(const char *,char **,
14971                                     unsigned int);
14972 extern long simple_strtol(const char *,char **,
14973                           unsigned int);
14974 extern int sprintf(char * buf, const char * fmt, ...);
14975 extern int vsprintf(char *buf, const char *, va_list);
14976
14977 extern int session_of_pgrp(int pgrp);
14978
14979 asmlinkage int printk(const char * fmt, ...)
14980     __attribute__((format (printf, 1, 2)));
14981
14982 #if DEBUG
14983 #define pr_debug(fmt,arg...) \
14984     printk(KERN_DEBUG fmt,##arg)
14985 #else
14986 #define pr_debug(fmt,arg...) \
14987     do { } while (0)
14988 #endif
14989
14990 #define pr_info(fmt,arg...) \
14991     printk(KERN_INFO fmt,##arg)
14992
14993 /* Display an IP address in readable format. */
14994
14995 #define NIPQUAD(addr) \
14996     ((unsigned char *)&addr)[0], \
14997     ((unsigned char *)&addr)[1], \
14998     ((unsigned char *)&addr)[2], \
14999     ((unsigned char *)&addr)[3]
15000
15001 #endif /* __KERNEL__ */
15002
15003 #define SI_LOAD_SHIFT 16
15004 struct sysinfo {
15005     long uptime; /* Seconds since boot */
15006     unsigned long loads[3]; /* 1/5/15-min load averages */
15007     unsigned long totalram; /* Total usable main mem sz */
15008     unsigned long freeram; /* Available memory size */
15009     unsigned long sharedram; /* Amount of shared memory */
15010     unsigned long bufferram; /* Memory used by buffers */
15011     unsigned long totalswap; /* Total swap space size */
15012     unsigned long freeswap; /* swap spc still available */
15013     unsigned short procs; /* # of current processes */
15014     char _f[22]; /* Pads struct to 64 bytes */
15015 };
15016
15017 #endif

```

include/linux/kernel_stat.h

```

15018 #ifndef _LINUX_KERNEL_STAT_H
15019 #define _LINUX_KERNEL_STAT_H
15020
15021 #include <asm/irq.h>
15022 #include <linux/smp.h>
15023 #include <linux/tasks.h>
15024
15025 /* 'kernel_stat.h' contains the definitions needed for
15026  * doing some kernel statistics (CPU usage, context
15027  * switches ...), used by rstatd/perfmer */
15028
15029 #define DK_NDRIVE 4
15030
15031 struct kernel_stat {
15032     unsigned int cpu_user, cpu_nice, cpu_system;
15033     unsigned int per_cpu_user[NR_CPUS],
15034                 per_cpu_nice[NR_CPUS],
15035                 per_cpu_system[NR_CPUS];
15036     unsigned int dk_drive[DK_NDRIVE];
15037     unsigned int dk_drive_rio[DK_NDRIVE];
15038     unsigned int dk_drive_wio[DK_NDRIVE];
15039     unsigned int dk_drive_rblk[DK_NDRIVE];
15040     unsigned int dk_drive_wblk[DK_NDRIVE];
15041     unsigned int pggpin, pggout;
15042     unsigned int pswpin, pswpout;
15043     unsigned int irqs[NR_CPUS][NR_IRQS];
15044     unsigned int ipackets, opackets;
15045     unsigned int ierrors, oerrors;
15046     unsigned int collisions;
15047     unsigned int context_swch;
15048 };
15049
15050 extern struct kernel_stat kstat;
15051
15052 /* # of interrupts per specific IRQ src, since bootup */
15053 extern inline int kstat_irqs (int irq)
15054 {
15055     int i, sum=0;
15056
15057     for (i = 0 ; i < smp_num_cpus ; i++)
15058         sum += kstat.irqs[cpu_logical_map(i)][irq];
15059
15060     return sum;
15061 }
15062
15063 #endif /* LINUX KERNEL STAT H */

```

include/linux/limits.h

```

15064 #ifndef _LINUX_LIMITS_H
15065 #define _LINUX_LIMITS_H
15066
15067 #define NR_OPEN    1024
15068
15069 #define NGROUPS_MAX 32 /* suppl. group IDs are avail */
15070 #define ARG_MAX 131072 /* bytes of args+env for exec() */
15071 #define CHILD_MAX 999 /* no limit :-) */
15072 #define OPEN_MAX 256 /* # open files a proc may have */
15073 #define LINK_MAX 127 /* # links a file may have */
15074 #define MAX_CANON 255 /* size of the canonical input q */
15075 #define MAX_INPUT 255 /* size of the type-ahead buf */
15076 #define NAME_MAX 255 /* # chars in a file name */
15077 #define PATH_MAX 4095 /* # chars in a path name */
15078 #define PIPE_BUF 4096 /* # bytes in atomic pipe write */
15079
15080 #define RTSIG_MAX    32
15081
15082 #endif

```

include/linux/mm.h

```

15083 #ifndef _LINUX_MM_H
15084 #define _LINUX_MM_H
15085
15086 #include <linux/sched.h>
15087 #include <linux/errno.h>
15088
15089 #ifdef __KERNEL__
15090
15091 #include <linux/string.h>
15092
15093 extern unsigned long max_mapnr;
15094 extern unsigned long num_physpages;
15095 extern void * high_memory;
15096 extern int page_cluster;
15097
15098 #include <asm/page.h>
15099 #include <asm/atomic.h>
15100
15101 /* Linux kernel virtual memory manager primitives. The
15102  * idea being to have a "virtual" mm in the same way we
15103  * have a virtual fs - giving a cleaner interface to the
15104  * mm details, and allowing different kinds of memory
15105  * mappings (from shared memory to executable loading to
15106  * arbitrary mmap() functions). */
15107

```

```

15108 /* This struct defines a memory VMM memory area. There is
15109  * one of these per VM-area/task. A VM area is any part
15110  * of the process virtual memory space that has a special
15111  * rule for the page-fault handlers (ie a shared library,
15112  * the executable area etc). */
15113 struct vm_area_struct {
15114     struct mm_struct * vm_mm; /* VM area parameters */
15115     unsigned long vm_start;
15116     unsigned long vm_end;
15117
15118     /* linked list of VM areas per task, sorted by addr */
15119     struct vm_area_struct *vm_next;
15120
15121     pgprot_t vm_page_prot;
15122     unsigned short vm_flags;
15123
15124     /* AVL tree of VM areas per task, sorted by address */
15125     short vm_avl_height;
15126     struct vm_area_struct * vm_avl_left;
15127     struct vm_area_struct * vm_avl_right;
15128
15129     /* For areas with inode, the list inode->i_mmap, for
15130      * shm areas, the list of attaches, else unused. */
15131     struct vm_area_struct *vm_next_share;
15132     struct vm_area_struct **vm_pprev_share;
15133
15134     struct vm_operations_struct * vm_ops;
15135     unsigned long vm_offset;
15136     struct file * vm_file;
15137     unsigned long vm_pte; /* shared mem */
15138 };
15139
15140 /* vm_flags.. */
15141 #define VM_READ    0x0001 /* currently active flags */
15142 #define VM_WRITE   0x0002
15143 #define VM_EXEC    0x0004
15144 #define VM_SHARED  0x0008
15145
15146 #define VM_MAYREAD 0x0010 /* lms for mprotect() etc*/
15147 #define VM_MAYWRITE 0x0020
15148 #define VM_MAYEXEC 0x0040
15149 #define VM_MAYSHARE 0x0080
15150
15151 #define VM_GROWSDOWN 0x0100 /* general info on segment*/
15152 #define VM_GROWSUP 0x0200
15153 #define VM_SHM 0x0400 /* shrd mem,don't swap out*/
15154 #define VM_DENYWRITE 0x0800 /* ETXTRSY on write. */
15155

```

```

15156 #define VM_EXECUTABLE 0x1000
15157 #define VM_LOCKED      0x2000
15158 #define VM_IO          0x4000 /* Mem-mapped I/O /similar*/
15159
15160 #define VM_STACK_FLAGS 0x0177
15161
15162 /* mapping from the currently active vm_flags protection
15163  * bits (the low four bits) to a page protection mask. */
15164 extern pgprot_t protection_map[16];
15165
15166
15167 /* These are the virtual MM functions - opening of an
15168  * area, closing and unmapping it (needed to keep files
15169  * on disk up-to-date etc), pointer to the functions
15170  * called when a no-page or a wp-page exception occurs.*/
15171 struct vm_operations_struct {
15172     void (*open)(struct vm_area_struct * area);
15173     void (*close)(struct vm_area_struct * area);
15174     void (*unmap)(struct vm_area_struct *area,
15175                 unsigned long, size_t);
15176     void (*protect)(struct vm_area_struct *area,
15177                    unsigned long, size_t, unsigned int newprot);
15178     int (*sync)(struct vm_area_struct *area,
15179                unsigned long, size_t, unsigned int flags);
15180     void (*advise)(struct vm_area_struct *area,
15181                  unsigned long, size_t, unsigned int advise);
15182     unsigned long (*nopage)(struct vm_area_struct * area,
15183                            unsigned long address, int write_access);
15184     unsigned long (*wppage)(struct vm_area_struct * area,
15185                             unsigned long address, unsigned long page);
15186     int (*swapout)(struct vm_area_struct *, struct page *);
15187     pte_t (*swpin)(struct vm_area_struct *, unsigned long,
15188                   unsigned long);
15189 };
15190
15191 /* Try to keep the most commonly accessed fields in
15192  * single cache lines here (16 bytes or greater). This
15193  * ordering should be particularly beneficial on 32-bit
15194  * processors.
15195  *
15196  * The first line is data used in page cache lookup, the
15197  * second line is used for linear searches (eg. clock
15198  * algorithm scans). */
15199 typedef struct page {
15200     /* these must be first (free area handling) */
15201     struct page *next;
15202     struct page *prev;
15203     struct inode *inode;
15204     unsigned long offset;
15205     struct page *next_hash;
15206     atomic_t count;
15207     /* atomic flags, some possibly updated asynchronously*/
15208     unsigned long flags;
15209     struct wait_queue *wait;
15210     struct page **pprev_hash;
15211     struct buffer_head * buffers;
15212 } mem_map_t;
15213
15214 /* Page flag bit values */
15215 #define PG_locked          0
15216 #define PG_error          1
15217 #define PG_referenced     2
15218 #define PG_dirty          3
15219 #define PG_uptodate       4
15220 #define PG_free_after     5
15221 #define PG_decr_after     6
15222 #define PG_swap_unlock_after 7
15223 #define PG_DMA            8
15224 #define PG_Slab           9
15225 #define PG_swap_cache    10
15226 #define PG_skip          11
15227 #define PG_reserved      31
15228
15229 /* Make it prettier to test the above... */
15230 #define PageLocked(page) \
15231     (test_bit(PG_locked, &(page)->flags))
15232 #define PageError(page) \
15233     (test_bit(PG_error, &(page)->flags))
15234 #define PageReferenced(page) \
15235     (test_bit(PG_referenced, &(page)->flags))
15236 #define PageDirty(page) \
15237     (test_bit(PG_dirty, &(page)->flags))
15238 #define PageUptodate(page) \
15239     (test_bit(PG_uptodate, &(page)->flags))
15240 #define PageFreeAfter(page) \
15241     (test_bit(PG_free_after, &(page)->flags))
15242 #define PageDecrAfter(page) \
15243     (test_bit(PG_decr_after, &(page)->flags))
15244 #define PageSwapUnlockAfter(page) \
15245     (test_bit(PG_swap_unlock_after, &(page)->flags))
15246 #define PageDMA(page) \
15247     (test_bit(PG_DMA, &(page)->flags))
15248 #define PageSlab(page) \
15249     (test_bit(PG_Slab, &(page)->flags))
15250 #define PageSwapCache(page) \
15251     (test_bit(PG_swap_cache, &(page)->flags))

```

```

15252 #define PageReserved(page) \
15253     (test_bit(PG_reserved, &(page)->flags))
15254
15255 #define PageSetSlab(page) \
15256     (set_bit(PG_Slab, &(page)->flags))
15257 #define PageSetSwapCache(page) \
15258     (set_bit(PG_swap_cache, &(page)->flags))
15259
15260 #define PageTestandSetDirty(page) \
15261     (test_and_set_bit(PG_dirty, &(page)->flags))
15262 #define PageTestandSetSwapCache(page) \
15263     (test_and_set_bit(PG_swap_cache, &(page)->flags))
15264
15265 #define PageClearSlab(page) \
15266     (clear_bit(PG_Slab, &(page)->flags))
15267 #define PageClearSwapCache(page) \
15268     (clear_bit(PG_swap_cache, &(page)->flags))
15269
15270 #define PageTestandClearDirty(page) \
15271     (test_and_clear_bit(PG_dirty, &(page)->flags))
15272 #define PageTestandClearSwapCache(page) \
15273     (test_and_clear_bit(PG_swap_cache, &(page)->flags))
15274
15275 /* Various page->flags bits:
15276  *
15277  * PG_reserved is set for a page which must never be
15278  * accessed (which may not even be present).
15279  *
15280  * PG_DMA is set for those pages which lie in the range
15281  * of physical addresses capable of carrying DMA
15282  * transfers.
15283  *
15284  * Multiple processes may "see" the same page. E.g. for
15285  * untouched mappings of /dev/null, all processes see the
15286  * same page full of zeroes, and text pages of
15287  * executables and shared libraries have only one copy in
15288  * memory, at most, normally.
15289  *
15290  * For the non-reserved pages, page->count denotes a
15291  * reference count.
15292  * page->count == 0 means the page is free.
15293  * page->count == 1 means the page is used for exactly
15294  * one purpose
15295  * (e.g. a private data page of one process).
15296  *
15297  * A page may be used for kmalloc() or anyone else who
15298  * does a get_free_page(). In this case the page->count
15299  * is at least 1, and all other fields are unused but

```

```

15300  * should be 0 or NULL. The management of this page is
15301  * the responsibility of the one who uses it.
15302  *
15303  * The other pages (we may call them "process pages") are
15304  * completely managed by the Linux memory manager: I/O,
15305  * buffers, swapping etc. The following discussion
15306  * applies only to them.
15307  *
15308  * A page may belong to an inode's memory mapping. In
15309  * this case, page->inode is the pointer to the inode,
15310  * and page->offset is the file offset of the page (not
15311  * necessarily a multiple of PAGE_SIZE).
15312  *
15313  * A page may have buffers allocated to it. In this case,
15314  * page->buffers is a circular list of these buffer
15315  * heads. Else, page->buffers == NULL.
15316  *
15317  * For pages belonging to inodes, the page->count is the
15318  * number of attaches, plus 1 if buffers are allocated to
15319  * the page.
15320  *
15321  * All pages belonging to an inode make up a doubly
15322  * linked list inode->i_pages, using the fields
15323  * page->next and page->prev. (These fields are also used
15324  * for freelist management when page->count==0.) There
15325  * is also a hash table mapping (inode,offset) to the
15326  * page in memory if present. The lists for this hash
15327  * table use the fields page->next hash and
15328  * page->pprev_hash.
15329  *
15330  * All process pages can do I/O:
15331  * - inode pages may need to be read from disk,
15332  * - inode pages which have been modified and are
15333  *   MAP_SHARED may need to be written to disk,
15334  * - private pages which have been modified may need to
15335  *   be swapped out to swap space and (later) to be read
15336  *   back into memory.
15337  * During disk I/O, PG_locked is used. This bit is set
15338  * before I/O and reset when I/O completes. page->wait is
15339  * a wait queue of all tasks waiting for the I/O on this
15340  * page to complete.
15341  * PG_uptodate tells whether the page's contents is
15342  * valid. When a read completes, the page becomes
15343  * uptodate, unless a disk I/O error happened.
15344  * When a write completes, and PG_free_after is set, the
15345  * page is freed without any further delay.
15346  *
15347  * For choosing which pages to swap out, inode pages

```

```

15348 * carry a PG_referenced bit, which is set any time the
15349 * system accesses that page through the (inode,offset)
15350 * hash table.
15351 *
15352 * PG_skip is used on sparc/sparc64 architectures to
15353 * "skip" certain parts of the address space.
15354 *
15355 * PG_error is set to indicate that an I/O error occurred
15356 * on this page. */
15357
15358 extern mem_map_t * mem_map;
15359
15360 /* This is timing-critical - most of the time in getting
15361 * a new page goes to clearing the page. If you want a
15362 * page without the clearing overhead, just use
15363 * __get_free_page() directly.. */
15364 #define __get_free_page(gfp_mask) \
15365     __get_free_pages((gfp_mask),0)
15366 #define __get_dma_pages(gfp_mask, order) \
15367     __get_free_pages((gfp_mask) | GFP_DMA,(order))
15368 extern unsigned long
15369 FASTCALL(__get_free_pages(int gfp_mask,
15370                          unsigned long gfp_order));
15371
15372 extern inline unsigned long get_free_page(int gfp_mask)
15373 {
15374     unsigned long page;
15375
15376     page = __get_free_page(gfp_mask);
15377     if (page)
15378         clear_page(page);
15379     return page;
15380 }
15381
15382 extern int low_on_memory;
15383
15384 /* memory.c & swap.c*/
15385
15386 #define free_page(addr) free_pages((addr),0)
15387 extern void FASTCALL(free_pages(unsigned long addr,
15388                             unsigned long order));
15389 extern void FASTCALL(__free_page(struct page *));
15390
15391 extern void show_free_areas(void);
15392 extern unsigned long put_dirty_page(
15393     struct task_struct * tsk, unsigned long page,
15394     unsigned long address);
15395

```

```

15396 extern void free_page_tables(struct mm_struct * mm);
15397 extern void clear_page_tables(struct mm_struct *,
15398                             unsigned long, int);
15399 extern int new_page_tables(struct task_struct * tsk);
15400
15401 extern void zap_page_range(struct mm_struct *mm,
15402                          unsigned long address, unsigned long size);
15403 extern int copy_page_range(struct mm_struct *dst,
15404                          struct mm_struct *src, struct vm_area_struct *vma);
15405 extern int remap_page_range(unsigned long from,
15406                          unsigned long to, unsigned long size, pgprot_t prot);
15407 extern int zeromap_page_range(unsigned long from,
15408                          unsigned long size, pgprot_t prot);
15409
15410 extern void vmtruncate(struct inode * inode,
15411                      unsigned long offset);
15412 extern int handle_mm_fault(struct task_struct *tsk,
15413                          struct vm_area_struct *vma, unsigned long address,
15414                          int write_access);
15415 extern void make_pages_present(unsigned long addr,
15416                              unsigned long end);
15417
15418 extern int pgt_cache_water[2];
15419 extern int check_pgt_cache(void);
15420
15421 extern unsigned long paging_init(unsigned long start_mem,
15422                              unsigned long end_mem);
15423 extern void mem_init(unsigned long start_mem,
15424                    unsigned long end_mem);
15425 extern void show_mem(void);
15426 extern void oom(struct task_struct * tsk);
15427 extern void si_meminfo(struct sysinfo * val);
15428
15429 /* mmap.c */
15430 extern void vma_init(void);
15431 extern void merge_segments(struct mm_struct *,
15432                          unsigned long, unsigned long);
15433 extern void insert_vm_struct(struct mm_struct *,
15434                          struct vm_area_struct *);
15435 extern void build_mmap_avl(struct mm_struct *);
15436 extern void exit_mmap(struct mm_struct *);
15437 extern unsigned long get_unmapped_area(unsigned long,
15438                                       unsigned long);
15439
15440 extern unsigned long do_mmap(struct file *,
15441                          unsigned long, unsigned long, unsigned long,
15442                          unsigned long, unsigned long);
15443 extern int do_munmap(unsigned long, size_t);

```

```

15444 .
15445 /* filemap.c */
15446 extern void remove_inode_page(struct page *);
15447 extern unsigned long page_unuse(struct page *);
15448 extern int shrink_mmap(int, int);
15449 extern void truncate_inode_pages(struct inode *,
15450                                 unsigned long);
15451 extern unsigned long get_cached_page(struct inode *,
15452                                     unsigned long, int);
15453 extern void put_cached_page(unsigned long);
15454
15455 /* GFP bitmasks.. */
15456 #define __GFP_WAIT      0x01
15457 #define __GFP_LOW      0x02
15458 #define __GFP_MED      0x04
15459 #define __GFP_HIGH     0x08
15460 #define __GFP_IO       0x10
15461 #define __GFP_SWAP     0x20
15462
15463 #define __GFP_DMA      0x80
15464
15465 #define GFP_BUFFER     (__GFP_LOW | __GFP_WAIT)
15466 #define GFP_ATOMIC     (__GFP_HIGH)
15467 #define GFP_USER       (__GFP_LOW | __GFP_WAIT | __GFP_IO)
15468 #define GFP_KERNEL     (__GFP_MED | __GFP_WAIT | __GFP_IO)
15469 #define GFP_NFS        (__GFP_HIGH | __GFP_WAIT | __GFP_IO)
15470 #define GFP_KSWAPD     (__GFP_IO | __GFP_SWAP)
15471
15472 /* Flag - indicates that the buffer will be suitable for
15473    DMA. Ignored on some platforms, used as appropriate
15474    on others */
15475
15476 #define GFP_DMA        __GFP_DMA
15477
15478 /* vma is the first one with address < vma->vm_end, and
15479    * even address < vma->vm_start. Have to extend vma. */
15480 static inline int expand_stack(
15481     struct vm_area_struct * vma, unsigned long address)
15482 {
15483     unsigned long grow;
15484
15485     address &= PAGE_MASK;
15486     grow = vma->vm_start - address;
15487     if (vma->vm_end - address
15488         >(unsigned long)current->rlim[RLIMIT_STACK].rlim_cur
15489         || (vma->vm_mm->total_vm << PAGE_SHIFT) + grow
15490         >(unsigned long)current->rlim[RLIMIT_AS].rlim_cur)
15491         return -ENOMEM;

```

```

15492     vma->vm_start = address;
15493     vma->vm_offset -= grow;
15494     vma->vm_mm->total_vm += grow >> PAGE_SHIFT;
15495     if (vma->vm_flags & VM_LOCKED)
15496         vma->vm_mm->locked_vm += grow >> PAGE_SHIFT;
15497     return 0;
15498 }
15499
15500 /* Look up the first VMA which satisfies addr < vm_end,
15501    NULL if none. */
15502 extern struct vm_area_struct * find_vma(
15503     struct mm_struct * mm, unsigned long addr);
15504
15505 /* Look up the first VMA which intersects the interval
15506    start_addr..end_addr 1, NULL if none. Assume
15507    start_addr < end_addr. */
15508 static inline struct vm_area_struct *
15509 find_vma_intersection(struct mm_struct * mm,
15510                      unsigned long start_addr, unsigned long end_addr)
15511 {
15512     struct vm_area_struct * vma = find_vma(mm, start_addr);
15513
15514     if (vma && end_addr <= vma->vm_start)
15515         vma = NULL;
15516     return vma;
15517 }
15518
15519 #define buffer_under_min() \
15520     ((buffermem >> PAGE_SHIFT) * 100 < \
15521     buffer_mem.min_percent * num_physpages) \
15522 #define pgcache_under_min() \
15523     (page_cache_size * 100 < \
15524     page_cache.min_percent * num_physpages) \
15525
15526 #endif /* KERNEL */
15527
15528 #endif

```

include/linux/module.h

```

15529 /*
15530  * Dynamic loading of modules into the kernel.
15531  *
15532  * Rewritten by Richard Henderson <rth@tamu.edu> Dec 1996
15533  */
15534
15535 #ifndef LINUX_MODULE_H
15536 #define LINUX_MODULE_H

```



```

15537
15538 #include <linux/config.h>
15539
15540 #ifdef __GENKSYMS__
15541 # define _set_ver(sym) sym
15542 # undef MODVERSIONS
15543 # define MODVERSIONS
15544 #else /* ! __GENKSYMS__ */
15545 # if !defined(MODVERSIONS) && defined(EXPORT_SYMTAB)
15546 # define _set_ver(sym) sym
15547 # include <linux/modversions.h>
15548 # endif
15549 #endif /* __GENKSYMS__ */
15550
15551 #include <asm/atomic.h>
15552
15553 /* Don't need to bring in all of uaccess.h just for this
15554 decl. */
15555 struct exception_table_entry;
15556
15557 /* Used by get_kernel_syms, which is obsolete. */
15558 struct kernel_sym
15559 {
15560     unsigned long value;
15561     /* should have been 64-sizeof(long); oh well */
15562     char name[60];
15563 };
15564
15565 struct module_symbol
15566 {
15567     unsigned long value;
15568     const char *name;
15569 };
15570
15571 struct module_ref
15572 {
15573     struct module *dep; /* "parent" pointer */
15574     struct module *ref; /* "child" pointer */
15575     struct module_ref *next_ref;
15576 };
15577
15578 /* TBD */
15579 struct module_persist;
15580
15581 struct module
15582 {
15583     unsigned long size of struct; /* == sizeof(module) */
15584     struct module *next;

```

```

15585     const char *name;
15586     unsigned long size;
15587
15588     union
15589     {
15590         atomic_t usecount;
15591         long pad;
15592     } uc; /* Needs to keep its size - so says rth */
15593
15594     unsigned long flags; /* AUTOCLEAN et al */
15595
15596     unsigned nsyms;
15597     unsigned ndeps;
15598
15599     struct module_symbol *syms;
15600     struct module_ref *deps;
15601     struct module_ref *refs;
15602     int (*init)(void);
15603     void (*cleanup)(void);
15604     const struct exception_table_entry *ex_table_start;
15605     const struct exception_table_entry *ex_table_end;
15606 #ifdef __alpha__
15607     unsigned long gp;
15608 #endif
15609     /* Members past this point are extensions to the basic
15610     module support and are optional. Use
15611     mod_opt_member() to examine them. */
15612     const struct module_persist *persist_start;
15613     const struct module_persist *persist_end;
15614     int (*can_unload)(void);
15615 };
15616
15617 struct module_info
15618 {
15619     unsigned long addr;
15620     unsigned long size;
15621     unsigned long flags;
15622     long usecount;
15623 };
15624
15625 /* Bits of module.flags. */
15626
15627 #define MOD_UNINITIALIZED 0
15628 #define MOD_RUNNING 1
15629 #define MOD_DELETED 2
15630 #define MOD_AUTOCLEAN 4
15631 #define MOD_VISITED 8
15632 #define MOD_USED_ONCE 16

```

```

15633 #define MOD_JUST_FREED          32
15634
15635 /* Values for query_module's which. */
15636
15637 #define QM_MODULES          1
15638 #define QM_DEPS            2
15639 #define QM_REFS            3
15640 #define QM_SYMBOLS         4
15641 #define QM_INFO            5
15642
15643 /* When struct module is extended, we must test whether
15644    the new member is present in the header received from
15645    insmod before we can use it. This function returns
15646    true if the member is present. */
15647
15648 #define mod_member_present(mod,member) \
15649    ((unsigned long)((struct module *)0L->member + 1) \
15650     <- (mod)->size_of_struct)
15651
15652 /* Backwards compatibility definition. */
15653
15654 #define GET_USE_COUNT(module) \
15655    (atomic_read(&(module)->uc.usecount))
15656
15657 /* Poke the use count of a module. */
15658
15659 #define __MOD_INC_USE_COUNT(mod) \
15660    (atomic_inc(&(mod)->uc.usecount), \
15661     (mod)->flags |= MOD_VISITED|MOD_USED_ONCE)
15662 #define __MOD_DEC_USE_COUNT(mod) \
15663    (atomic_dec(&(mod)->uc.usecount), \
15664     (mod)->flags |= MOD_VISITED)
15665 #define __MOD_IN_USE(mod) \
15666    (mod_member_present((mod), can_unload) \
15667     && (mod)->can_unload \
15668     ? (mod)->can_unload() \
15669     : atomic_read(&(mod)->uc.usecount))
15670
15671 /* Indirect stringification. */
15672
15673 #define __MODULE_STRING_1(x)    #x
15674 #define __MODULE_STRING(x)     __MODULE_STRING_1(x)
15675
15676 /* Find a symbol exported by the kernel or another
15677    * module */
15678 extern unsigned long get_module_symbol(char *, char *);
15679
15680 #if defined(MODULE) && !defined(__GENSYMS__)

```

```

15681
15682 /* Embedded module documentation macros. */
15683
15684 /* For documentation purposes only. */
15685
15686 #define MODULE_AUTHOR(name) \
15687    const char __module_author[] \
15688    __attribute__((section(".modinfo"))) = "author=" name
15689
15690 #define MODULE_DESCRIPTION(desc) \
15691    const char __module_description[] \
15692    __attribute__((section(".modinfo"))) = \
15693    "description=" desc
15694
15695 /* Could potentially be used by kmod... */
15696
15697 #define MODULE_SUPPORTED_DEVICE(dev) \
15698    const char __module_device[] \
15699    __attribute__((section(".modinfo"))) = "device=" dev
15700
15701 /* Used to verify parameters given to the module. The
15702    TYPE arg should be a string in the following format:
15703    [min[-max]][b,h,i,l,s]
15704
15705    The MIN and MAX specifiers delimit the length of the
15706    array. If MAX is omitted, it defaults to MIN; if both
15707    are omitted, the default is 1. The final character is
15708    a type specifier:
15709
15710    b      byte
15711    h      short
15712    i      int
15713    l      long
15714    s      string */
15715
15716 #define MODULE_PARM(var,type) \
15717    const char __module_parm_##var[] \
15718    __attribute__((section(".modinfo"))) = \
15719    "parm_" __MODULE_STRING(var) "=" type
15720
15721 #define MODULE_PARM_DESC(var,desc) \
15722    const char __module_parm_desc_##var[] \
15723    __attribute__((section(".modinfo"))) = \
15724    "parm_desc_" __MODULE_STRING(var) "=" desc
15725
15726 /* The attributes of a section are set the first time the
15727    * section is seen; we want .modinfo to not be allocated.
15728    */

```

```

15729
15730 __asm__(".section .modinfo\n\t.previous");
15731
15732 /* Define the module variable, and usage macros. */
15733 extern struct module __this_module;
15734
15735 #define MOD_INC_USE_COUNT          \
15736     __MOD_INC_USE_COUNT(&__this_module)
15737 #define MOD_DEC_USE_COUNT          \
15738     __MOD_DEC_USE_COUNT(&__this_module)
15739 #define MOD_IN_USE                  \
15740     __MOD_IN_USE(&__this_module)
15741
15742 #ifndef __NO_VERSION__
15743 #include <linux/version.h>
15744 const char __module_kernel_version[]
15745     __attribute__((section(".modinfo"))) =
15746     "kernel_version=" UTS_RELEASE;
15747 #ifdef MODVERSIONS
15748 const char __module_using_checksums[]
15749     __attribute__((section(".modinfo"))) =
15750     "using_checksums=1";
15751 #endif
15752 #endif
15753
15754 #else /* MODULE */
15755
15756 #define MODULE_AUTHOR(name)
15757 #define MODULE_DESCRIPTION(desc)
15758 #define MODULE_SUPPORTED_DEVICE(name)
15759 #define MODULE_PARM(var,type)
15760 #define MODULE_PARM_DESC(var,desc)
15761
15762 #ifndef __GENKSYMS__
15763
15764 #define MOD_INC_USE_COUNT          do { } while (0)
15765 #define MOD_DEC_USE_COUNT          do { } while (0)
15766 #define MOD_IN_USE                  1
15767
15768 extern struct module *module_list;
15769
15770 #endif /* !__GENKSYMS__ */
15771
15772 #endif /* MODULE */
15773
15774 /* Export a symbol either from the kernel or a module.
15775
15776     In the kernel, the symbol is added to the kernel's

```

```

15777     global symbol table.
15778
15779     In a module, it controls which variables are exported.
15780     If no variables are explicitly exported, the action is
15781     controlled by the insmod -[xX] flags. Otherwise, only
15782     the variables listed are exported. This obviates the
15783     need for the old register_syntab() function. */
15784
15785 #if defined(__GENKSYMS__)
15786
15787 /* We want the EXPORT_SYMBOL tag left intact for
15788     recognition. */
15789
15790 #elif !defined(AUTOCONF_INCLUDED)
15791
15792 #define __EXPORT_SYMBOL(sym, str)          \
15793     error config_must_be_included_before_module
15794 #define EXPORT_SYMBOL(var)                \
15795     error config_must_be_included_before_module
15796 #define EXPORT_SYMBOL_NOVERS(var)         \
15797     error config_must_be_included_before_module
15798
15799 #elif !defined(CONFIG_MODULES)
15800
15801 #define __EXPORT_SYMBOL(sym, str)
15802 #define EXPORT_SYMBOL(var)
15803 #define EXPORT_SYMBOL_NOVERS(var)
15804
15805 #elif !defined(EXPORT_SYMTAB)
15806
15807 /* If things weren't set up in the Makefiles to get
15808     EXPORT_SYMTAB defined, then they weren't set up to run
15809     gensyms properly so MODVERSIONS breaks. */
15810 #define __EXPORT_SYMBOL(sym, str)          \
15811     error EXPORT_SYMTAB_not_defined
15812 #define EXPORT_SYMBOL(var)                \
15813     error EXPORT_SYMTAB_not_defined
15814 #define EXPORT_SYMBOL_NOVERS(var)         \
15815     error EXPORT_SYMTAB_not_defined
15816
15817 #else
15818
15819 #define __EXPORT_SYMBOL(sym, str)          \
15820     const char __kstrtab_##sym[]
15821     __attribute__((section(".kstrtab"))) = str;
15822     const struct module_symbol __ksymtab_##sym
15823     __attribute__((section("ksymtab"))) =
15824     { (unsigned long)&sym, __kstrtab_##sym }

```

```

15825
15826 #if defined(MODVERSIONS) || !defined(CONFIG_MODVERSIONS)
15827 #define EXPORT_SYMBOL(var) \
15828     __EXPORT_SYMBOL(var, __MODULE_STRING(var))
15829 #else
15830 #define EXPORT_SYMBOL(var) \
15831     __EXPORT_SYMBOL(var, \
15832         __MODULE_STRING(__VERSIONED_SYMBOL(var)))
15833 #endif
15834
15835 #define EXPORT_SYMBOL_NOVERS(var) \
15836     __EXPORT_SYMBOL(var, __MODULE_STRING(var))
15837
15838 #endif /* __GENKSYMS__ */
15839
15840 #ifndef MODULE
15841 /* Force a module to export no symbols. */
15842 #define EXPORT_NO_SYMBOLS \
15843     __asm__(".section __ksymtab\n.previous")
15844 #else
15845 #define EXPORT_NO_SYMBOLS
15846 #endif /* MODULE */
15847
15848 #endif /* _LINUX_MODULE_H */

```

include/linux/msg.h

```

15849 #ifndef _LINUX_MSG_H
15850 #define _LINUX_MSG_H
15851
15852 #include <linux/ipc.h>
15853
15854 /* ipcs ctl commands */
15855 #define MSG_STAT 11
15856 #define MSG_INFO 12
15857
15858 /* msgrcv options */
15859 /* no error if message is too big */
15860 #define MSG_NOERROR 010000
15861 /* recv any msg except of specified type.*/
15862 #define MSG_EXCEPT 020000
15863
15864 /* one msqid structure for each queue on the system */
15865 struct msqid_ds {
15866     struct ipc_perm msg_perm;
15867     struct msg *msg_first; /* first msg on queue */
15868     struct msg *msg_last; /* last msg in queue */
15869     __kernel_time_t msg_stime; /* last msgsnd time */

```

```

15870     __kernel_time_t msg_rtime; /* last msgrcv time */
15871     __kernel_time_t msg_ctime; /* last change time */
15872     struct wait_queue *wait;
15873     struct wait_queue *rwait;
15874     unsigned short msg_cbytes; /* current # bytes on q */
15875     unsigned short msg_qnum; /* # of msgs in queue */
15876     unsigned short msg_qbytes; /* max # of bytes on q */
15877     __kernel_ipc_pid_t msg_lspid; /* pid of last msgsnd */
15878     __kernel_ipc_pid_t msg_lrpid; /* last receive pid */
15879 };
15880
15881 /* message buffer for msgsnd and msgrcv calls */
15882 struct msgbuf {
15883     long mtype; /* type of message */
15884     char mtext[1]; /* message text */
15885 };
15886
15887 /* buffer for msgctl calls IPC_INFO, MSG_INFO */
15888 struct msginfo {
15889     int msgpool;
15890     int msgmap;
15891     int msgmax;
15892     int msgmnb;
15893     int msgmni;
15894     int msgssz;
15895     int msgtql;
15896     unsigned short msgseg;
15897 };
15898
15899 /* max # of msg queue identifiers */
15900 #define MSGMNI 128 /* <= 1K */
15901 /* max size of message (bytes) */
15902 #define MSGMAX 4056 /* <= 4056 */
15903 /* default max size of a message queue */
15904 #define MSGMNB 16384 /* ? */
15905
15906 /* unused */
15907 /* size in kilobytes of message pool */
15908 #define MSGPOOL (MSGMNI*MSGMNB/1024)
15909 #define MSGTQL MSGMNB /* number of system msg headers */
15910 #define MSGMAP MSGMNB /* number of entries in msg map */
15911 #define MSGSSZ 16 /* message segment size */
15912 /* max no. of segments */
15913 #define __MSGSEG ((MSGPOOL*1024)/MSGSSZ)
15914 #define MSGSEG ( __MSGSEG <= 0xffff ? __MSGSEG : 0xffff)
15915
15916 #ifdef __KERNEL__
15917

```

```

15918 /* one msg structure for each message */
15919 struct msg {
15920     struct msg *msg_next; /* next message on queue */
15921     long msg_type;
15922     char *msg_spot; /* message text address */
15923     time_t msg_stime; /* msgsnd time */
15924     short msg_ts; /* message text size */
15925 };
15926
15927 asmlinkage int sys_msgget(key_t key, int msgflg);
15928 asmlinkage int sys_msgsnd(int msqid, struct msgbuf *msgp,
15929     size_t msgsz, int msgflg);
15930 asmlinkage int sys_msgrcv(int msqid, struct msgbuf *msgp,
15931     size_t msgsz, long msgtyp,
15932     int msgflg);
15933 asmlinkage int sys_msgctl(int msqid, int cmd,
15934     struct msqid_ds *buf);
15935
15936 #endif /* __KERNEL__ */
15937
15938 #endif /* _LINUX_MSG_H */

```

include/linux/personality.h

```

15939 #ifndef _PERSONALITY_H
15940 #define _PERSONALITY_H
15941
15942 #include <linux/linkage.h>
15943 #include <linux/ptrace.h>
15944
15945
15946 /* Flags for bug emulation. These occupy the top three
15947     bytes. */
15948 #define STICKY_TIMEOITS      0x4000000
15949 #define WHOLE_SECONDS        0x2000000
15950 #define ADDR_LIMIT_32BIT    0x0800000
15951
15952 /* Personality types. These go in the low byte. Avoid
15953     * using the top bit, it will conflict with error
15954     * returns. */
15955 #define PER_MASK             (0x00ff)
15956 #define PER_LINUX           (0x0000)
15957 #define PER_LINUX_32BIT    (0x0000 | ADDR_LIMIT_32BIT)
15958 #define PER_SVR4           (0x0001 | STICKY_TIMEOITS)
15959 #define PER_SVR3           (0x0002 | STICKY_TIMEOITS)
15960 #define PER_SCOSVR3       (0x0003 | STICKY_TIMEOITS |
15961     WHOLE_SECONDS)
15962 #define PER_WYSEV386      (0x0004 | STICKY TIMEOITS)

```

```

15963 #define PER_ISCR4          (0x0005 | STICKY_TIMEOITS)
15964 #define PER_BSD           (0x0006)
15965 #define PER_XENIX         (0x0007 | STICKY_TIMEOITS)
15966 #define PER_LINUX32      (0x0008)
15967
15968 /* Prototype for an lcall7 syscall handler. */
15969 typedef void (*lcall7_func)(struct pt_regs *);
15970
15971
15972 /* Description of an execution domain - personality range
15973     * supported, lcall7 syscall handler, start up / shut
15974     * down functions etc. N.B. The name and lcall7 handler
15975     * must be where they are since the offset of the handler
15976     * is hard coded in kernel/sys call.S. */
15977 struct exec_domain {
15978     const char *name;
15979     lcall7_func handler;
15980     unsigned char pers_low, pers_high;
15981     unsigned long * signal_map;
15982     unsigned long * signal_invmmap;
15983     struct module * module;
15984     struct exec_domain *next;
15985 };
15986
15987 extern struct exec_domain default_exec_domain;
15988
15989 extern struct exec_domain *lookup_exec_domain(
15990     unsigned long personality);
15991 extern int register_exec_domain(struct exec_domain *it);
15992 extern int unregister_exec_domain(struct exec_domain*it);
15993 asmlinkage int sys_personality(
15994     unsigned long personality);
15995
15996 #endif /* _PERSONALITY_H */

```

include/linux/reboot.h

```

15997 #ifndef _LINUX_REBOOT_H
15998 #define _LINUX_REBOOT_H
15999
16000 /* Magic values required to use _reboot() system call. */
16001
16002 #define LINUX_REBOOT_MAGIC1      0xfeeldead
16003 #define LINUX_REBOOT_MAGIC2      672274793
16004 #define LINUX_REBOOT_MAGIC2A     85072278
16005 #define LINUX_REBOOT_MAGIC2B     369367448
16006
16007

```

```

16008 /* Commands accepted by the _reboot() system call.
16009 *
16010 * RESTART
16011     Restart system using default command and mode.
16012 * HALT
16013 *     Stop OS & give sys control to ROM mon, if any.
16014 * CAD_ON
16015 *     Ctrl-Alt-Del sequence causes RESTART command.
16016 * CAD_OFF
16017 *     Ctrl-Alt-Del sequence sends SIGINT to init task.
16018 * POWER_OFF
16019 *     Stop OS & remove all pwr from system, if poss.
16020 * RESTART2     Restart system using given command string.
16021 */
16022
16023 #define LINUX_REBOOT_CMD_RESTART     0x01234567
16024 #define LINUX_REBOOT_CMD_HALT       0xCDEF0123
16025 #define LINUX_REBOOT_CMD_CAD_ON     0x89ABCDEF
16026 #define LINUX_REBOOT_CMD_CAD_OFF    0x00000000
16027 #define LINUX_REBOOT_CMD_POWER_OFF  0x4321FEDC
16028 #define LINUX_REBOOT_CMD_RESTART2   0xA1B2C3D4
16029
16030
16031 #ifdef __KERNEL__
16032
16033 #include <linux/notifier.h>
16034
16035 extern struct notifier_block *reboot_notifier_list;
16036 extern int register_reboot_notifier(
16037     struct notifier_block *);
16038 extern int unregister_reboot_notifier(
16039     struct notifier_block *);
16040
16041
16042 /* Architecture-specific implementations of sys_reboot
16043 * commands. */
16044 extern void machine_restart(char *cmd);
16045 extern void machine_halt(void);
16046 extern void machine_power_off(void);
16047
16048 #endif
16049
16050 #endif /* _LINUX_REBOOT_H */

include/linux/resource.h
16051 #ifndef _LINUX_RESOURCE_H
16052 #define _LINUX_RESOURCE_H
16053
16054 #include <linux/time.h>
16055
16056 /* Resource control/accounting header file for linux */
16057
16058 /* Definition of struct rusage taken from BSD 4.3 Reno
16059 *
16060 * We don't support all of these yet, but we might as
16061 * well have them.... Otherwise, each time we add new
16062 * items, programs which depend on this structure will
16063 * lose. This reduces the chances of that happening. */
16064 #define RUSAGE_SELF     0
16065 #define RUSAGE_CHILDREN (-1)
16066 #define RUSAGE_BOTH     (-2) /* sys_wait4() uses this */
16067
16068 struct rusage {
16069     struct timeval ru_utime; /* user time used */
16070     struct timeval ru_stime; /* system time used */
16071     long ru_maxrss; /* max resident set size */
16072     long ru_ixrss; /* shared mem size */
16073     long ru_idrss; /* unshared data size */
16074     long ru_isrss; /* unshared stack size */
16075     long ru_minflt; /* page reclaims */
16076     long ru_majflt; /* page faults */
16077     long ru_nswap; /* swaps */
16078     long ru_inblock; /* block input operations */
16079     long ru_oublock; /* block output operations */
16080     long ru_msgsnd; /* messages sent */
16081     long ru_msgrcv; /* messages received */
16082     long ru_nsignals; /* signals received */
16083     long ru_nvcsw; /*voluntary context switches*/
16084     long ru_nivcsw; /* involuntary " */
16085 };
16086
16087 #define RLIM_INFINITY ((long)(-0UL>>1))
16088
16089 struct rlimit {
16090     long rlim_cur;
16091     long rlim_max;
16092 };
16093
16094 #define PRIO_MIN (-20)
16095 #define PRIO_MAX 20
16096
16097 #define PRIO_PROCESS 0
16098 #define PRIO_PGRP 1
16099 #define PRIO_USER 2
16100

```

```

16101 /* Due to binary compatibility, the actual resource
16102 * numbers may be different for different linux
16103 * versions.. */
16104 #include <asm/resource.h>
16105
16106 #endif

```

include/linux/sched.h

```

16107 #ifndef _LINUX_SCHED_H
16108 #define _LINUX_SCHED_H
16109
16110 #include <asm/param.h> /* for HZ */
16111
16112 extern unsigned long event;
16113
16114 #include <linux/binfmts.h>
16115 #include <linux/personality.h>
16116 #include <linux/tasks.h>
16117 #include <linux/kernel.h>
16118 #include <linux/types.h>
16119 #include <linux/times.h>
16120 #include <linux/timex.h>
16121
16122 #include <asm/system.h>
16123 #include <asm/semaphore.h>
16124 #include <asm/page.h>
16125
16126 #include <linux/smp.h>
16127 #include <linux/tty.h>
16128 #include <linux/sem.h>
16129 #include <linux/signal.h>
16130 #include <linux/securebits.h>
16131
16132 /* cloning flags: */
16133 /* signal mask to be sent at exit */
16134 #define CSIGNAL 0x000000ff
16135 /* set if VM shared between processes */
16136 #define CLONE_VM 0x00000100
16137 /* set if fs info shared between processes */
16138 #define CLONE_FS 0x00000200
16139 /* set if open files shared between processes */
16140 #define CLONE_FILES 0x00000400
16141 /* set if signal handlers shared */
16142 #define CLONE_SIGHAND 0x00000800
16143 /* set if pid shared */
16144 #define CLONE_PID 0x00001000
16145 /* set to let tracing continue on the child too */

```

```

16146 #define CLONE_PTRACE 0x00002000
16147 /* set if parent wants child to wake it on mm_release */
16148 #define CLONE_VFORK 0x00004000
16149
16150 /* These are the constant used to fake the fixed-point
16151 * load-average counting. Some notes:
16152 * - 11 bit fractions expand to 22 bits by the
16153 * multiplies: this gives a load-average precision of 10
16154 * bits integer + 11 bits fractional
16155 * - if you want to count load-averages more often, you
16156 * need more precision, or rounding will get you. With
16157 * 2-second counting freq, the EXP_n values would be
16158 * 1981, 2034 and 2043 if still using only 11 bit
16159 * fractions. */
16160 extern unsigned long avenrun[]; /* Load averages */
16161
16162 #define FSHIFT 11 /* # bits of precision */
16163 #define FIXED_1 (1<<FSHIFT) /* 1.0 as fixed-pt */
16164 #define LOAD_FREQ (5*HZ) /* 5 sec intervals */
16165 #define EXP_1 1884 /* 1/exp(5sec/1min) as FP*/
16166 #define EXP_5 2014 /* 1/exp(5sec/5min) */
16167 #define EXP_15 2037 /* 1/exp(5sec/15min) */
16168
16169 #define CALC_LOAD(load,exp,n) \
16170     load *= exp; \
16171     load += n*(FIXED_1-exp); \
16172     load >>= FSHIFT;
16173
16174 #define CT_TO_SECS(x) ((x) / HZ)
16175 #define CT_TO_USECS(x) (((x) % HZ) * 1000000/HZ)
16176
16177 extern int nr_running, nr_tasks;
16178 extern int last_pid;
16179
16180 #include <linux/fs.h>
16181 #include <linux/time.h>
16182 #include <linux/param.h>
16183 #include <linux/resource.h>
16184 #include <linux/timer.h>
16185
16186 #include <asm/processor.h>
16187
16188 #define TASK_RUNNING 0
16189 #define TASK_INTERRUPTIBLE 1
16190 #define TASK_UNINTERRUPTIBLE 2
16191 #define TASK_ZOMBIE 4
16192 #define TASK_STOPPED 8
16193 #define TASK_SWAPPING 16

```

▶ P-51

```

16194
16195 /* Scheduling policies */
16196 #define SCHED_OTHER          0
16197 #define SCHED_FIFO          1
16198 #define SCHED_RR            2
16199
16200 /* This is an additional bit set when we want to yield
16201 * the CPU for one re-schedule.. */
16202 #define SCHED_YIELD          0x10
16203
16204 struct sched_param {
16205     int sched_priority;
16206 };
16207
16208 #ifndef NULL
16209 #define NULL ((void *) 0)
16210 #endif
16211
16212 #ifdef __KERNEL__
16213
16214 #include <asm/spinlock.h>
16215
16216 /* This serializes "schedule()" and also protects the
16217 * run-queue from deletions/modifications (but _adding_
16218 * to the beginning of the run-queue has a separate
16219 * lock). */
16220 extern rwlock_t tasklist_lock;
16221 extern spinlock_t scheduler_lock;
16222 extern spinlock_t runqueue_lock;
16223
16224 extern void sched_init(void);
16225 extern void show_state(void);
16226 extern void trap_init(void);
16227
16228 #define MAX_SCHEDULE_TIMEOUT    LONG_MAX
16229 extern signed long
16230 FASTCALL(schedule_timeout(signed long timeout));
16231 asmlinkage void schedule(void);
16232
16233 /* Open file table structure */
16234 struct files_struct {
16235     atomic_t count;
16236     int max_fds;
16237     struct file ** fd;      /* current fd array */
16238     fd_set close_on_exec;
16239     fd_set open_fds;
16240 };
16241

```

```

16242 #define INIT_FILES {
16243     ATOMIC_INIT(1),
16244     NR_OPEN,
16245     &init_fd_array[0],
16246     { { 0, } },
16247     { { 0, } }
16248 }
16249
16250 struct fs_struct {
16251     atomic_t count;
16252     int umask;
16253     struct dentry * root, * pwd;
16254 };
16255
16256 #define INIT_FS {
16257     ATOMIC_INIT(1),
16258     0022,
16259     NULL, NULL
16260 }
16261
16262 /* Maximum number of active map areas.. This is a random
16263 * (large) number */
16264 #define MAX_MAP_COUNT    (65536)
16265
16266 /* Number of map areas at which the AVL tree is
16267 * activated. This is arbitrary. */
16268 #define AVL_MIN_MAP_COUNT    32
16269
16270 struct mm_struct {
16271     struct vm_area_struct *mmap; /* list of VMAs */
16272     struct vm_area_struct *mmap_avl; /* tree of VMAs */
16273     /* last find_vma result */
16274     struct vm_area_struct *mmap_cache;
16275     pgd_t * pgd;
16276     atomic_t count;
16277     int map_count; /* number of VMAs */
16278     struct semaphore mmap_sem;
16279     unsigned long context;
16280     unsigned long start_code, end_code,
16281         start_data, end_data;
16282     unsigned long start_brk, brk, start_stack;
16283     unsigned long arg_start, arg_end, env_start, env_end;
16284     unsigned long rss, total_vm, locked_vm;
16285     unsigned long def_flags;
16286     unsigned long cpu_vm_mask;
16287     /* number of pages to swap on next pass */
16288     unsigned long swap_cnt;
16289     unsigned long swap_address;

```



```

16290 /* This is an architecture-specific pointer: the
16291  * portable part of Linux does not know about any
16292  * segments. */
16293 void * segments;
16294 };
16295
16296 #define INIT_MM {
16297     &init_mmap, NULL, NULL,
16298     swapper_pg_dir,
16299     ATOMIC_INIT(1), 1,
16300     MUTEX,
16301     0,
16302     0, 0, 0, 0,
16303     0, 0, 0,
16304     0, 0, 0, 0,
16305     0, 0, 0,
16306     0, 0, 0, 0, NULL }
16307
16308 struct signal_struct {
16309     atomic_t          count;
16310     struct k_sigaction action[_NSIG];
16311     spinlock_t        siglock;
16312 };
16313
16314 #define INIT_SIGNALS {
16315     ATOMIC_INIT(1),
16316     { {{0}}, },
16317     SPIN_LOCK_UNLOCKED }
16318
16319 /* Some day this will be a full-fledged user tracking
16320  * system.. Right now it is only used to track how many
16321  * processes a user has, but it has the potential to
16322  * track memory usage etc. */
16323 struct user_struct;
16324
16325 struct task_struct {
16326     /* these are hardcoded - don't touch */
16327     /* -1 unrunnable, 0 runnable, >0 stopped */
16328     volatile long state;
16329     /* per process flags, defined below */
16330     unsigned long flags;
16331     int sigpending;
16332     mm_segment_t addr_limit;
16333     /* thread address space:
16334     * 0-0xBFFFFFFF for user thread
16335     * 0-0xFFFFFFFF for kernel thread */
16336     struct exec_domain *exec_domain;
16337     long need_resched;

```

```

16338
16339 /* various fields */
16340     long counter;
16341     long priority;
16342     cycles_t avg_slice;
16343     /* SMP and runqueue state */
16344     int has_cpu;
16345     int processor;
16346     int last_processor;
16347     /* Lock depth. We can context switch in and out of
16348     * holding a syscall kernel lock... */
16349     int lock_depth;
16350     struct task_struct *next_task, *prev_task;
16351     struct task_struct *next_run, *prev_run;
16352
16353     /* task state */
16354     struct linux_binfmt *binfmt;
16355     int exit_code, exit_signal;
16356     int pdeath_signal; /* Signal sent when parent dies */
16357     /* ??? */
16358     unsigned long personality;
16359     int dumpable:1;
16360     int did_exec:1;
16361     pid_t pid;
16362     pid_t pgrp;
16363     pid_t tty_old_pgrp;
16364     pid_t session;
16365     /* boolean value for session group leader */
16366     int leader;
16367     /* pointers to (original) parent process, youngest
16368     * child, younger sibling, older sibling, respectively.
16369     * (p->father can be replaced with p->p_pptr->pid) */
16370     struct task_struct *p_opptr, *p_pptr, *p_cpptr,
16371     *p_ysptr, *p_osptr;
16372
16373     /* PID hash table linkage. */
16374     struct task_struct *pidhash_next;
16375     struct task_struct **pidhash_pprev;
16376
16377     /* Pointer to task[] array linkage. */
16378     struct task_struct **tarray_ptr;
16379
16380     struct wait_queue *wait_chldexit; /* for wait4() */
16381     struct semaphore *vfork_sem; /* for vfork() */
16382     unsigned long policy, rt_priority;
16383     unsigned long it_real_value, it_prof_value,
16384     it_virt_value;
16385     unsigned long it_real_incr, it_prof_incr, it_virt_incr;

```

```

16386 struct timer_list real_timer;
16387 struct tms times;
16388 unsigned long start_time;
16389 long per_cpu_utime[NR_CPUS], per_cpu_stime[NR_CPUS];
16390 /* mm fault and swap info: this can arguably be seen as
16391 either mm-specific or thread-specific */
16392 unsigned long min_flt, maj_flt, nswap,
16393 cmin_flt, cmaj_flt, cnswap;
16394 int swappable:1;
16395 /* process credentials */
16396 uid_t uid,euid,suid,fsuid;
16397 gid_t gid,egid,sgid,fsgid;
16398 int ngroups;
16399 gid_t groups[NGROUPS];
16400 kernel_cap_t cap_effective, cap_inheritable,
16401 cap_permitted;
16402 struct user_struct *user;
16403 /* limits */
16404 struct rlimit rlim[RLIM_NLIMITS];
16405 unsigned short used_math;
16406 char comm[16];
16407 /* file system info */
16408 int link_count;
16409 struct tty_struct *tty; /* NULL if no tty */
16410 /* ipc stuff */
16411 struct sem_undo *semundo;
16412 struct sem_queue *semsleeping;
16413 /* tss for this task */
16414 struct thread_struct tss;
16415 /* filesystem information */
16416 struct fs_struct *fs;
16417 /* open file information */
16418 struct files_struct *files;
16419 /* memory management info */
16420 struct mm_struct *mm;
16421
16422 /* signal handlers */
16423 spinlock_t sigmask_lock; /* Protects signal & blocked*/
16424 struct signal_struct *sig;
16425 sigset_t signal, blocked;
16426 struct signal_queue *sigqueue, **sigqueue_tail;
16427 unsigned long sas_ss_sp;
16428 size_t sas_ss_size;
16429 );
16430
16431 /* Per process flags */
16432 /* Print alignment warning msgs */
16433 #define PF_ALIGNWARN 0x00000001

```

```

16434 /* Not implemented yet, only for 486*/
16435 /* being created */
16436 #define PF_STARTING 0x00000002
16437 /* getting shut down */
16438 #define PF_EXITING 0x00000004
16439 /* set if ptrace (0) has been called */
16440 #define PF_PTRACED 0x00000010
16441 /* tracing system calls */
16442 #define PF_TRACESYS 0x00000020
16443 /* forked but didn't exec */
16444 #define PF_FORKNOEXEC 0x00000040
16445 /* used super-user privileges */
16446 #define PF_SUPERPRIV 0x00000100
16447 /* dumped core */
16448 #define PF_DUMPCORE 0x00000200
16449 /* killed by a signal */
16450 #define PF_SIGNALED 0x00000400
16451 /* Allocating memory */
16452 #define PF_MEMALLOC 0x00000800
16453 /* Wake up parent in mm_release */
16454 #define PF_VFORK 0x00001000
16455
16456 /* task used FPU this quantum (SMP) */
16457 #define PF_USEDMPU 0x00100000
16458 /* delayed trace (used on m68k, i386) */
16459 #define PF_DTRACE 0x00200000
16460
16461 /* Limit the stack by to some sane default: root can
16462 * always increase this limit if needed.. 8MB seems
16463 * reasonable. */
16464 #define _STK_LIM (8*1024*1024)
16465
16466 #define DEF_PRIORITY (20*HZ/100) /* 210-ms tm slices*/
16467
16468 /* INIT_TASK is used to set up the first task table,
16469 * touch at your own risk!. Base=0, limit=0x1fffff (=2MB)
16470 */
16471 #define INIT_TASK \
16472 /* state etc */\{0,0,0,KERNEL_DS,&default_exec_domain,0, \
16473 /* counter */ DEF_PRIORITY,DEF_PRIORITY,0, \
16474 /* SMP */ 0,0,0,-1, \
16475 /* schedlink */ &init_task,&init_task, &init_task, \
16476 &init_task, \
16477 /* binfmt */ NULL, \
16478 /* ec,brk... */ 0,0,0,0,0,0, \
16479 /* pid etc.. */ 0,0,0,0,0, \
16480 /* proc links*/ &init_task,&init_task,NULL,NULL,NULL, \
16481 /* pidhash */ NULL, NULL, \

```

```

16482 /* tarray */ &task[0], \
16483 /* chld wait */ NULL, NULL, \
16484 /* timeout */ SCHED_OTHER,0,0,0,0,0,0, \
16485 /* timer */ { NULL, NULL, 0, 0, it_real_fn }, \
16486 /* utime */ {0,0,0,0},0, \
16487 /* per CPU times */ {0, }, {0, }, \
16488 /* fit */ 0,0,0,0,0,0, \
16489 /* swp */ 0, \
16490 /* process credentials */ \
16491 /* uid etc */ 0,0,0,0,0,0,0,0, \
16492 /* suppl grps*/ 0, {0,}, \
16493 /* caps */ CAP_INIT_EFF_SET,CAP_INIT_INH_SET, \
16494 CAP_FULL_SET, \
16495 /* user */ NULL, \
16496 /* rlimits */ INIT_RLIMITS, \
16497 /* math */ 0, \
16498 /* comm */ "swapper", \
16499 /* fs info */ 0,NULL, \
16500 /* ipc */ NULL, NULL, \
16501 /* tss */ INIT_TSS, \
16502 /* fs */ &init_fs, \
16503 /* files */ &init_files, \
16504 /* mm */ &init_mm, \
16505 /* signals */ SPIN_LOCK_UNLOCKED, &init_signals, \
16506 {{0}}, {0}}, NULL, &init_task.sigqueue, 0, 0, \
16507 }
16508
16509 union task_union {
16510 struct task_struct task;
16511 unsigned long stack[2048];
16512 };
16513
16514 extern union task_union init_task_union;
16515
16516 extern struct mm_struct init_mm;
16517 extern struct task_struct *task[NR_TASKS];
16518
16519 extern struct task_struct **tarray_freelist;
16520 extern spinlock_t taskslot_lock;
16521
16522 extern __inline__ void
16523 add_free_taskslot(struct task_struct **t)
16524 {
16525 spin_lock(&taskslot_lock);
16526 *t = (struct task_struct *) tarray_freelist;
16527 tarray_freelist = t;
16528 spin_unlock(&taskslot_lock);
16529 }

```

p 510

```

16530
16531 extern __inline__ struct task_struct **
16532 get_free_taskslot(void)
16533 {
16534 struct task_struct **tslot;
16535
16536 spin_lock(&taskslot_lock);
16537 if((tslot = tarray_freelist) != NULL)
16538 tarray_freelist = (struct task_struct **) *tslot;
16539 spin_unlock(&taskslot_lock);
16540
16541 return tslot;
16542 }
16543
16544 /* PID hashing. */
16545 #define PIDHASH_SZ (NR_TASKS >> 2)
16546 extern struct task_struct *pidhash[PIDHASH_SZ];
16547
16548 #define pid_hashfn(x)
16549 (((x) >> 8) ^ (x)) & (PIDHASH_SZ - 1)
16550
16551 extern __inline__ void hash_pid(struct task_struct *p)
16552 {
16553 struct task_struct **htable =
16554 &pidhash[pid_hashfn(p->pid)];
16555
16556 if((p->pidhash_next = *htable) != NULL)
16557 (*htable)->pidhash_pprev = &p->pidhash_next;
16558 *htable = p;
16559 p->pidhash_pprev = htable;
16560 }
16561
16562 extern __inline__ void unhash_pid(struct task_struct *p)
16563 {
16564 if(p->pidhash_next)
16565 p->pidhash_next->pidhash_pprev = p->pidhash_pprev;
16566 *p->pidhash_pprev = p->pidhash_next;
16567 }
16568
16569 extern __inline__ struct task_struct *
16570 find_task_by_pid(int pid)
16571 {
16572 struct task_struct *p, **htable =
16573 &pidhash[pid_hashfn(pid)];
16574
16575 for(p = *htable; p && p->pid != pid;
16576 p = p->pidhash_next)
16577 ;

```

```

16578
16579     return p;
16580 }
16581
16582 /* per-UID process charging. */
16583 extern int alloc_uid(struct task_struct *p);
16584 void free_uid(struct task_struct *p);
16585
16586 #include <asm/current.h>
16587
16588 extern unsigned long volatile jiffies;
16589 extern unsigned long itimer_ticks;
16590 extern unsigned long itimer_next;
16591 extern struct timeval xtime;
16592 extern void do_timer(struct pt_regs *);
16593
16594 extern unsigned int * prof_buffer;
16595 extern unsigned long prof_len;
16596 extern unsigned long prof_shift;
16597
16598 #define CURRENT_TIME (xtime.tv_sec)
16599
16600 extern void FASTCALL(__wake_up(struct wait_queue ** p,
16601                               unsigned int mode));
16602 extern void FASTCALL(sleep_on(struct wait_queue ** p));
16603 extern long FASTCALL(sleep_on_timeout(
16604     struct wait_queue ** p, signed long timeout));
16605 extern void FASTCALL(interruptible_sleep_on(
16606     struct wait_queue ** p));
16607 extern long FASTCALL(interruptible_sleep_on_timeout(
16608     struct wait_queue ** p, signed long timeout));
16609 extern void FASTCALL(wake_up_process(
16610     struct task_struct * tsk));
16611
16612 #define wake_up(x) \
16613     __wake_up((x), TASK_UNINTERRUPTIBLE | TASK_INTERRUPTIBLE) \
16614 #define wake_up_interruptible(x) \
16615     __wake_up((x), TASK_INTERRUPTIBLE)
16616
16617 extern int in_group_p(gid_t grp);
16618
16619 extern void flush_signals(struct task_struct *);
16620 extern void flush_signal_handlers(struct task_struct *);
16621 extern int dequeue_signal(sigset_t *block, siginfo_t *);
16622 extern int send_sig_info(int, struct siginfo *info,
16623                          struct task_struct *);
16624 extern int force_sig_info(int, struct siginfo *info,
16625                          struct task_struct *);
16626 extern int kill_pg_info(int, struct siginfo *info, pid_t);
16627 extern int kill_sl_info(int, struct siginfo *info, pid_t);
16628 extern int kill_proc_info(int, struct siginfo *info,
16629                          pid_t);
16630 extern int kill_something_info(int, struct siginfo *info,
16631                               int);
16632 extern void notify_parent(struct task_struct * tsk, int);
16633 extern void force_sig(int sig, struct task_struct * p);
16634 extern int send_sig(int sig, struct task_struct * p,
16635                   int priv);
16636 extern int kill_pg(pid_t, int, int);
16637 extern int kill_sl(pid_t, int, int);
16638 extern int kill_proc(pid_t, int, int);
16639 extern int do_sigaction(int sig,
16640     const struct k_sigaction *act, struct k_sigaction *oact);
16641 extern int do_sigaltstack(const stack_t *ss,
16642     stack_t *oss, unsigned long sp);
16643
16644 extern inline int signal_pending(struct task_struct *p)
16645 {
16646     return (p->sigpending != 0);
16647 }
16648
16649 /* Reevaluate whether the task has signals pending
16650 * delivery. This is required every time the blocked
16651 * sigset_t changes. All callers should have
16652 * t->sigmask_lock. */
16653
16654 static inline void recalc_sigpending(
16655     struct task_struct *t)
16656 {
16657     unsigned long ready;
16658     long i;
16659
16660     switch (_NSIG_WORDS) {
16661     default:
16662         for (i = _NSIG_WORDS, ready = 0; --i >= 0 ; )
16663             ready |= t->signal.sig[i] &~ t->blocked.sig[i];
16664         break;
16665
16666     case 4: ready = t->signal.sig[3] &~ t->blocked.sig[3];
16667             ready |= t->signal.sig[2] &~ t->blocked.sig[2];
16668             ready |= t->signal.sig[1] &~ t->blocked.sig[1];
16669             ready |= t->signal.sig[0] &~ t->blocked.sig[0];
16670         break;
16671
16672     case 2: ready = t->signal.sig[1] &~ t->blocked.sig[1];
16673             ready |= t->signal.sig[0] &~ t->blocked.sig[0];

```

p.485

```

16674     break;
16675
16676     case 1: ready = t->signal.sig[0] &~ t->blocked.sig[0];
16677     }
16678
16679     t->sigpending = (ready != 0);
16680 }
16681
16682 /* True if we are on the alternate signal stack. */
16683
16684 static inline int on_sig_stack(unsigned long sp)
16685 {
16686     return (sp >= current->sas_ss_sp
16687         && sp < current->sas_ss_sp + current->sas_ss_size);
16688 }
16689
16690 static inline int sas_ss_flags(unsigned long sp)
16691 {
16692     return (current->sas_ss_size == 0 ? SS_DISABLE
16693         : on_sig_stack(sp) ? SS_ONSTACK : 0);
16694 }
16695
16696 extern int request_irq(unsigned int irq,
16697     void (*handler)(int, void *, struct pt_regs *),
16698     unsigned long flags, const char *device, void *dev_id);
16699 extern void free_irq(unsigned int irq, void *dev_id);
16700
16701 /* This has now become a routine instead of a macro, it
16702  * sets a flag if it returns true (to do BSD-style
16703  * accounting where the process is flagged if it uses
16704  * root privs). The implication of this is that you
16705  * should do normal permissions checks first, and check
16706  * suser() last.
16707  *
16708  * [Dec 1997 -- Chris Evans] For correctness, the above
16709  * considerations need to be extended to fsuser(). This
16710  * is done, along with moving fsuser() checks to be last.
16711  *
16712  * These will be removed, but in the mean time, when the
16713  * SECURE_NOROOT flag is set, uids don't grant privilege.
16714  */
16715 extern inline int suser(void)
16716 {
16717     if (!issecure(SECURE_NOROOT) && current->euid == 0) {
16718         current->flags |= PF_SUPERPRIV;
16719         return 1;
16720     }
16721     return 0;

```

```

16722 }
16723
16724 extern inline int fsuser(void)
16725 {
16726     if (!issecure(SECURE_NOROOT) && current->fsuid == 0) {
16727         current->flags |= PF_SUPERPRIV;
16728         return 1;
16729     }
16730     return 0;
16731 }
16732
16733 /* capable() checks for a particular capability. New
16734  * privilege checks should use this interface, rather
16735  * than suser() or fsuser(). See
16736  * include/linux/capability.h for defined capabilities.*/
16737
16738 extern inline int capable(int cap)
16739 {
16740     #if 1 /* ok now */
16741         if (cap_raised(current->cap_effective, cap))
16742             #else
16743                 if (cap_is_fs_cap(cap)
16744                     ? current->fsuid == 0 : current->euid == 0)
16745                     #endif
16746                     {
16747                         current->flags |= PF_SUPERPRIV;
16748                         return 1;
16749                     }
16750     return 0;
16751 }
16752
16753 /* Routines for handling mm_structs */
16754 extern struct mm_struct * mm_alloc(void);
16755 static inline void mmget(struct mm_struct * mm)
16756 {
16757     atomic_inc(&mm->count);
16758 }
16759 extern void mmput(struct mm_struct *);
16760 /* Remove the current tasks stale references to the old
16761  * mm_struct */
16762 extern void mm_release(void);
16763
16764 extern int copy_thread(int, unsigned long,
16765     unsigned long, struct task_struct *, struct pt_regs *);
16766 extern void flush_thread(void);
16767 extern void exit_thread(void);
16768
16769 extern void exit_mm(struct task_struct *);

```

```

16770 extern void exit_fs(struct task_struct *);
16771 extern void exit_files(struct task_struct *);
16772 extern void exit_sighand(struct task_struct *);
16773
16774 extern int do_execve(char *, char **, char **,
16775                     struct pt_regs *);
16776 extern int do_fork(unsigned long, unsigned long,
16777                   struct pt_regs *);
16778
16779 /* The wait-queues are circular lists, and you have to be
16780  * *very* sure to keep them correct. Use only these two
16781  * functions to add/remove entries in the queues. */
16782 extern inline void __add_wait_queue(
16783     struct wait_queue ** p, struct wait_queue * wait)
16784 {
16785     wait->next = *p ? : WAIT_QUEUE_HEAD(p);
16786     *p = wait;
16787 }
16788
16789 extern rwlock_t waitqueue_lock;
16790
16791 extern inline void add_wait_queue(struct wait_queue ** p,
16792                                 struct wait_queue * wait)
16793 {
16794     unsigned long flags;
16795
16796     write_lock_irqsave(&waitqueue_lock, flags);
16797     __add_wait_queue(p, wait);
16798     write_unlock_irqrestore(&waitqueue_lock, flags);
16799 }
16800
16801 extern inline void __remove_wait_queue(
16802     struct wait_queue ** p, struct wait_queue * wait)
16803 {
16804     struct wait_queue * next = wait->next;
16805     struct wait_queue * head = next;
16806     struct wait_queue * tmp;
16807
16808     while ((tmp = head->next) != wait) {
16809         head = tmp;
16810     }
16811     head->next = next;
16812 }
16813
16814 extern inline void remove_wait_queue(
16815     struct wait_queue ** p, struct wait_queue * wait)
16816 {
16817     unsigned long flags;

```

```

16818
16819     write_lock_irqsave(&waitqueue_lock, flags);
16820     __remove_wait_queue(p, wait);
16821     write_unlock_irqrestore(&waitqueue_lock, flags);
16822 }
16823
16824 #define __wait_event(wq, condition) \
16825 do { \
16826     struct wait_queue __wait; \
16827 \
16828     __wait.task = current; \
16829     add_wait_queue(&wq, &__wait); \
16830     for (;;) { \
16831         current->state = TASK_UNINTERRUPTIBLE; \
16832         if (condition) \
16833             break; \
16834         schedule(); \
16835     } \
16836     current->state = TASK_RUNNING; \
16837     remove_wait_queue(&wq, &__wait); \
16838 } while (0)
16839
16840 #define wait_event(wq, condition) \
16841 do { \
16842     if (condition) \
16843         break; \
16844     __wait_event(wq, condition); \
16845 } while (0)
16846
16847 #define __wait_event_interruptible(wq, condition, ret) \
16848 do { \
16849     struct wait_queue __wait; \
16850 \
16851     __wait.task = current; \
16852     add_wait_queue(&wq, &__wait); \
16853     for (;;) { \
16854         current->state = TASK_INTERRUPTIBLE; \
16855         if (condition) \
16856             break; \
16857         if (!signal_pending(current)) { \
16858             schedule(); \
16859             continue; \
16860         } \
16861         ret = -ERESTARTSYS; \
16862         break; \
16863     } \
16864     current->state = TASK_RUNNING; \
16865     remove_wait_queue(&wq, &__wait); \

```

```

16866 } while (0)
16867
16868 #define wait_event_interruptible(wq, condition) \
16869 ({ \
16870     int __ret = 0; \
16871     if (!(condition)) \
16872         __wait_event_interruptible(wq, condition, __ret); \
16873     __ret; \
16874 })
16875
16876 #define REMOVE_LINKS(p) do { \
16877     (p)->next_task->prev_task = (p)->prev_task; \
16878     (p)->prev_task->next_task = (p)->next_task; \
16879     if ((p)->p_osptr) \
16880         (p)->p_osptr->p_ysptr = (p)->p_ysptr; \
16881     if ((p)->p_ysptr) \
16882         (p)->p_ysptr->p_osptr = (p)->p_osptr; \
16883     else \
16884         (p)->p_pptr->p_cpctr = (p)->p_osptr; \
16885     } while (0)
16886
16887 #define SET_LINKS(p) do { \
16888     (p)->next_task = &init_task; \
16889     (p)->prev_task = init_task.prev_task; \
16890     init_task.prev_task->next_task = (p); \
16891     init_task.prev_task = (p); \
16892     (p)->p_ysptr = NULL; \
16893     if (((p)->p_osptr = (p)->p_pptr->p_cpctr) != NULL) \
16894         (p)->p_osptr->p_ysptr = p; \
16895     (p)->p_pptr->p_cpctr = p; \
16896     } while (0)
16897
16898 #define for_each_task(p) \
16899     for(p = &init_task; (p = p->next_task) != &init_task; )
16900
16901 #endif /* __KERNEL__ */
16902
16903 #endif

include/linux/sem.h
16904 #ifndef _LINUX_SEM_H
16905 #define _LINUX_SEM_H
16906
16907 #include <linux/ipc.h>
16908
16909 /* semop flags */
16910 #define SEM_UNDO 0x1000 /* undo the operation on exit */

```

```

16911
16912 /* semctl Command Definitions. */
16913 #define GETPID 11 /* get semid */
16914 #define GETVAL 12 /* get semval */
16915 #define GETALL 13 /* get all semval's */
16916 #define GETNCNT 14 /* get semncnt */
16917 #define GETZCNT 15 /* get semzcnt */
16918 #define SETVAL 16 /* set semval */
16919 #define SETALL 17 /* set all semval's */
16920
16921 /* ipcs ctl cmds */
16922 #define SEM_STAT 18
16923 #define SEM_INFO 19
16924
16925 /* One semid data structure for each set of semaphores in
16926 the system. */
16927 struct semid_ds {
16928     struct ipc_perm sem_perm; /* permissions; see ipc.h */
16929     __kernel_time_t sem_otime; /* last semop time */
16930     __kernel_time_t sem_ctime; /* last change time */
16931     struct sem *sem_base; /* ptr to 1st sem in arr */
16932     struct sem_queue *sem_pending; /* op to be processed */
16933     struct sem_queue **sem_pending_last; /* last op */
16934     struct sem_undo *undo; /* undo reqs on this arr */
16935     unsigned short sem_nsems; /* # sems in array */
16936 };
16937
16938 /* semop system calls takes an array of these. */
16939 struct sembuf {
16940     unsigned short sem_num; /* sem index in array */
16941     short sem_op; /* semaphore operation */
16942     short sem_flg; /* operation flags */
16943 };
16944
16945 /* arg for semctl system calls. */
16946 union semun {
16947     int val; /* value for SETVAL */
16948     struct semid_ds *buf; /* buf for IPC_STAT&IPC_SET */
16949     unsigned short *array; /* array for GETALL & SETALL */
16950     struct seminfo * buf; /* buffer for IPC_INFO */
16951     void *__pad;
16952 };
16953
16954 struct seminfo {
16955     int semmap;
16956     int semmni;
16957     int semmns;
16958     int semmnu;

```

```

16959 int semsl;
16960 int semopm;
16961 int semume;
16962 int semusz;
16963 int semvmx;
16964 int semaem;
16965 };
16966
16967 #define SEMMNI 128 /* ? max # of sem identifiers */
16968 #define SEMMSL 32 /* <= 512 max semaphores per id */
16969 #define SEMMNS (SEMMNI*SEMMSL) /* ? max sems in system*/
16970 #define SEMOPM 32 /* ~ 100 max ops per semop call */
16971 #define SEMVMX 32767 /* semaphore maximum value */
16972
16973 /* unused */
16974 #define SEMUME SEMOPM /* max undo entries per process */
16975 #define SEMMNU SEMMNS /* # undo structures sys-wide */
16976 #define SEMAEM (SEMVMX >> 1) /* adjust on exit max val*/
16977 #define SEMMAP SEMMNS /* # of entries in sem map */
16978 #define SEMUSZ 20 /* sizeof struct sem_undo */
16979
16980 #ifdef __KERNEL__
16981
16982 /* One sem structure for each sem in the system. */
16983 struct sem {
16984     int semval; /* current value */
16985     int sempid; /* pid of last operation */
16986 };
16987
16988 /* One queue for each semaphore set in the system. */
16989 struct sem_queue {
16990     /* next entry in the queue */
16991     struct sem_queue * next;
16992     /* previous entry in the queue, *(q->prev) == q */
16993     struct sem_queue ** prev;
16994     /* sleeping process */
16995     struct wait_queue * sleeper;
16996     /* undo structure */
16997     struct sem_undo * undo;
16998     /* process id of requesting process */
16999     int pid;
17000     /* completion status of operation */
17001     int status;
17002     /* semaphore array for operations */
17003     struct semid_ds * sma;
17004     /* array of pending operations */
17005     struct sembuf * sops;
17006     /* number of operations */

```

```

17007     int nsops;
17008     /* operation will alter semaphore */
17009     int alter;
17010 };
17011
17012 /* Each task has a list of undo requests. They are
17013 * executed automatically when the process exits. */
17014 struct sem_undo {
17015     /* next entry on this process */
17016     struct sem_undo * proc_next;
17017     /* next entry on this semaphore set */
17018     struct sem_undo * id_next;
17019     /* semaphore set identifier */
17020     int semid;
17021     /* array of adjustments, one per semaphore */
17022     short * semadj;
17023 };
17024
17025 asmlinkage int sys_semget (key_t key, int nsems,
17026                             int semflg);
17027 asmlinkage int sys_semop (int semid, struct sembuf *sops,
17028                             unsigned nsops);
17029 asmlinkage int sys_semctl (int semid, int semnum,
17030                             int cmd, union semun arg);
17031
17032 #endif /* __KERNEL__ */
17033
17034 #endif /* _LINUX_SEM_H */

```

include/linux/shm.h

```

17035 #ifndef _LINUX_SHM_H_
17036 #define _LINUX_SHM_H_
17037
17038 #include <linux/ipc.h>
17039
17040 #include <asm/shmparam.h>
17041
17042 struct shmids {
17043     struct ipc_perm shm_perm; /* operation perms */
17044     int shm_segsz; /* seg sz (bytes) */
17045     __kernel_time_t shm_atime; /* last attach time */
17046     __kernel_time_t shm_dtime; /* last detach time */
17047     __kernel_time_t shm_ctime; /* last change time */
17048     __kernel_ipc_pid_t shm_cpid; /* pid of creator */
17049     __kernel_ipc_pid_t shm_lpid; /* pid of last op */
17050     unsigned short shm_nattch; /* #current attaches */
17051     unsigned short shm_unused; /* compatibility */

```



```

17052 void          *shm_unused2; /* ditto - for DIPC */
17053 void          *shm_unused3; /* unused */
17054 };
17055
17056 struct shm_id_kernel
17057 {
17058     struct shm_id_ds    u;
17059     /* the following are private */
17060     unsigned long      shm_npages; /* seg sz (pages) */
17061     /* array of ptrs to frames -> SHMMAX */
17062     unsigned long      *shm_pages;
17063     /* descriptors for attaches */
17064     struct vm_area_struct *attaches;
17065 };
17066
17067 /* permission flag for shmget */
17068 #define SHM_R        0400 /*or S_IRUGO from linux/stat.h*/
17069 #define SHM_W        0200 /*or S_IWUGO from linux/stat.h*/
17070
17071 /* mode for attach */
17072 #define SHM_RDONLY  010000 /* read-only access */
17073 #define SHM_RND     020000 /* round attach addr to
17074                             SHMLBA boundary */
17075 #define SHM_REMAP   040000 /*take over region on attach*/
17076
17077 /* super user shmctl commands */
17078 #define SHM_LOCK     11
17079 #define SHM_UNLOCK   12
17080
17081 /* ipcs ctl commands */
17082 #define SHM_STAT     13
17083 #define SHM_INFO     14
17084
17085 struct shm_info {
17086     int shmmax;
17087     int shmmn;
17088     int shmni;
17089     int shmseg;
17090     int shmall;
17091 };
17092
17093 struct shm_info {
17094     int used_ids;
17095     unsigned long shm_tot; /* total allocated shm */
17096     unsigned long shm_rss; /* total resident shm */
17097     unsigned long shm_swp; /* total swapped shm */
17098     unsigned long swap_attempts;
17099     unsigned long swap_successes;

```

```

17100 };
17101
17102 #ifdef __KERNEL__
17103
17104 /* shm_mode upper byte flags */
17105 /* segment will be destroyed on last detach */
17106 #define SHM_DEST    01000
17107 /* segment will not be swapped */
17108 #define SHM_LOCKED  02000
17109
17110 asmlinkage int sys_shmget(key_t key, int size,
17111                          int flag);
17112 asmlinkage int sys_shmat(int shm_id, char *shmaddr,
17113                          int shmflg, unsigned long *addr);
17114 asmlinkage int sys_shmctl(char *shmaddr);
17115 asmlinkage int sys_shmctl(int shm_id, int cmd,
17116                          struct shm_id_ds *buf);
17117 extern void shm_unuse(unsigned long entry,
17118                      unsigned long page);
17119
17120 #endif /* __KERNEL__ */
17121
17122 #endif /* _LINUX_SHM_H_ */

```

include/linux/signal.h

```

17123 #ifndef _LINUX_SIGNAL_H
17124 #define _LINUX_SIGNAL_H
17125
17126 #include <asm/signal.h>
17127 #include <asm/siginfo.h>
17128
17129 #ifdef __KERNEL__
17130 /* Real Time signals may be queued. */
17131
17132 struct signal_queue
17133 {
17134     struct signal_queue *next;
17135     siginfo_t info;
17136 };
17137
17138 /* Define some primitives to manipulate sigset_t. */
17139
17140 #ifndef HAVE_ARCH_SIG_BITOPS
17141 #include <asm/bitops.h>
17142
17143 /* We don't use <asm/bitops.h> for these because there is
17144  * no need to be atomic. */

```

```

p 177 17145 extern inline void sigaddset(sigset_t *set, int _sig)
17146 {
17147     unsigned long sig = _sig - 1;
17148     if (_NSIG_WORDS == 1)
17149         set->sig[0] |= 1UL << sig;
17150     else
17151         set->sig[sig/_NSIG_BPW] |= 1UL << (sig % _NSIG_BPW);
17152 }
17153
p 478 17154 extern inline void sigdelset(sigset_t *set, int _sig)
17155 {
17156     unsigned long sig = _sig - 1;
17157     if (_NSIG_WORDS == 1)
17158         set->sig[0] &= ~(1UL << sig);
17159     else
17160         set->sig[sig/_NSIG_BPW] &= ~(1UL << (sig%_NSIG_BPW));
17161 }
17162
p 478 17163 extern inline int sigismember(sigset_t *set, int _sig)
17164 {
17165     unsigned long sig = _sig - 1;
17166     if (_NSIG_WORDS == 1)
17167         return 1 & (set->sig[0] >> sig);
17168     else
17169         return 1 & (set->sig[sig/_NSIG_BPW]>>(sig%_NSIG_BPW));
17170 }
17171
p 478 17172 extern inline int sigfindinword(unsigned long word)
17173 {
17174     return ffz(~word);
17175 }
17176
p 478 17177 #define sigmask(sig)    (1UL << ((sig) - 1))
17178
17179 #endif /* __HAVE_ARCH_SIG_BITOPS */
17180
17181 #ifndef __HAVE_ARCH_SIG_SETOPS
17182 #include <linux/string.h>
17183
p 179 17184 #define _SIG_SET_BINOP(name, op) \
17185 extern inline void name(sigset_t *r, const sigset_t *a, \
17186                        const sigset_t *b) \
17187 { \
17188     unsigned long a0, a1, a2, a3, b0, b1, b2, b3; \
17189     unsigned long i; \
17190     \
17191     for (i = 0; i < _NSIG_WORDS/4; ++i) { \
17192         a0 = a->sig[4*i+0]; a1 = a->sig[4*i+1];
17193         a2 = a->sig[4*i+2]; a3 = a->sig[4*i+3];
17194         b0 = b->sig[4*i+0]; b1 = b->sig[4*i+1];
17195         b2 = b->sig[4*i+2]; b3 = b->sig[4*i+3];
17196         r->sig[4*i+0] = op(a0, b0);
17197         r->sig[4*i+1] = op(a1, b1);
17198         r->sig[4*i+2] = op(a2, b2);
17199         r->sig[4*i+3] = op(a3, b3);
17200     }
17201     switch (_NSIG_WORDS % 4) {
17202     case 3:
17203         a0 = a->sig[4*i+0]; a1 = a->sig[4*i+1];
17204         a2 = a->sig[4*i+2];
17205         b0 = b->sig[4*i+0]; b1 = b->sig[4*i+1];
17206         b2 = b->sig[4*i+2];
17207         r->sig[4*i+0] = op(a0, b0);
17208         r->sig[4*i+1] = op(a1, b1);
17209         r->sig[4*i+2] = op(a2, b2);
17210         break;
17211     case 2:
17212         a0 = a->sig[4*i+0]; a1 = a->sig[4*i+1];
17213         b0 = b->sig[4*i+0]; b1 = b->sig[4*i+1];
17214         r->sig[4*i+0] = op(a0, b0);
17215         r->sig[4*i+1] = op(a1, b1);
17216         break;
17217     case 1:
17218         a0 = a->sig[4*i+0]; b0 = b->sig[4*i+0];
17219         r->sig[4*i+0] = op(a0, b0);
17220         break;
17221     }
17222 }
17223
17224 #define _sig_or(x,y)    ((x) | (y))
17225 _SIG_SET_BINOP(sigorsets, _sig_or)
17226
17227 #define _sig_and(x,y)  ((x) & (y))
17228 _SIG_SET_BINOP(sigandsets, _sig_and)
17229
17230 #define _sig_nand(x,y) ((x) & ~(y))
17231 _SIG_SET_BINOP(signandsets, _sig_nand)
17232
17233 #undef _SIG_SET_BINOP
17234 #undef _sig_or
17235 #undef _sig_and
17236 #undef _sig_nand
17237
p 180 17238 #define _SIG_SET_OP(name, op) \
17239 extern inline void name(sigset_t *set) \
17240 {

```

```

17241 unsigned long i;
17242
17243 for (i = 0; i < _NSIG_WORDS/4; ++i) {
17244     set->sig[4*i+0] = op(set->sig[4*i+0]);
17245     set->sig[4*i+1] = op(set->sig[4*i+1]);
17246     set->sig[4*i+2] = op(set->sig[4*i+2]);
17247     set->sig[4*i+3] = op(set->sig[4*i+3]);
17248 }
17249 switch (_NSIG_WORDS % 4) {
17250     case 3: set->sig[4*i+2] = op(set->sig[4*i+2]);
17251     case 2: set->sig[4*i+1] = op(set->sig[4*i+1]);
17252     case 1: set->sig[4*i+0] = op(set->sig[4*i+0]);
17253 }
17254 }
17255
17256 #define _sig_not(x)    ~(x)
17257 _SIG_SET_OP(signotset, _sig_not)
17258
17259 #undef _SIG_SET_OP
17260 #undef _sig_not
17261
p 480 17262 extern inline void sigemptyset(sigset_t *set)
17263 {
17264     switch (_NSIG_WORDS) {
17265     default:
17266         memset(set, 0, sizeof(sigset_t));
17267         break;
17268     case 2: set->sig[1] = 0;
17269     case 1: set->sig[0] = 0;
17270         break;
17271     }
17272 }
17273
17274 extern inline void sigfillset(sigset_t *set)
17275 {
17276     switch (_NSIG_WORDS) {
17277     default:
17278         memset(set, -1, sizeof(sigset_t));
17279         break;
17280     case 2: set->sig[1] = -1;
17281     case 1: set->sig[0] = -1;
17282         break;
17283     }
17284 }
17285
17286 extern char * render_sigset_t(sigset_t *set,
17287                               char *buffer);
17288

```

```

17289 /* Some extensions for manipulating the low 32 signals in
17290    particular. */
17291
p 480 17292 extern inline void sigaddsetmask(sigset_t *set,
17293                                  unsigned long mask)
17294 {
17295     set->sig[0] |= mask;
17296 }
17297
17298 extern inline void sigdelsetmask(sigset_t *set,
17299                                  unsigned long mask)
17300 {
17301     set->sig[0] &= ~mask;
17302 }
17303
17304 extern inline int sigtestsetmask(sigset_t *set,
17305                                  unsigned long mask)
17306 {
17307     return (set->sig[0] & mask) != 0;
17308 }
17309
p 480 17310 extern inline void siginitset(sigset_t *set,
17311                               unsigned long mask)
17312 {
17313     set->sig[0] = mask;
17314     switch (_NSIG_WORDS) {
17315     default:
17316         memset(&set->sig[1], 0, sizeof(long)*(_NSIG_WORDS-1));
17317         break;
17318     case 2: set->sig[1] = 0;
17319     case 1:
17320     }
17321 }
17322
17323 extern inline void siginitsetinv(sigset_t *set,
17324                                  unsigned long mask)
17325 {
17326     set->sig[0] = ~mask;
17327     switch (_NSIG_WORDS) {
17328     default:
17329         memset(&set->sig[1], -1, sizeof(long)*(_NSIG_WORDS-1));
17330         break;
17331     case 2: set->sig[1] = -1;
17332     case 1:
17333     }
17334 }
17335
17336 #endif /* HAVE_ARCH_SIG_SETOPS */

```

```

17337
17338 #endif /* __KERNEL__ */
17339
17340 #endif /* _LINUX_SIGNAL_H */

```

include/linux/slab.h

```

17341 /*
17342  * linux/mm/slab.h
17343  * Written by Mark Hemment, 1996.
17344  * (markhe@nextd.demon.co.uk)
17345  */
17346
17347 #if !defined(_LINUX_SLAB_H)
17348 #define _LINUX_SLAB_H
17349
17350 #if defined(__KERNEL__)
17351
17352 typedef struct kmem_cache_s kmem_cache_t;
17353
17354 #include <linux/mm.h>
17355 #include <asm/cache.h>
17356
17357 /* flags for kmem_cache_alloc() */
17358 #define SLAB_BUFFER GFP_BUFFER
17359 #define SLAB_ATOMIC GFP_ATOMIC
17360 #define SLAB_USER GFP_USER
17361 #define SLAB_KERNEL GFP_KERNEL
17362 #define SLAB_NFS GFP_NFS
17363 #define SLAB_DMA GFP_DMA
17364
17365 #define SLAB_LEVEL_MASK 0x0000007FUL
17366 /* don't grow a cache */
17367 #define SLAB_NO_GROW 0x00001000UL
17368
17369 /* flags to pass to kmem_cache_create(). The first 3 are
17370  * only valid when the allocator as been build
17371  * SLAB_DEBUG_SUPPORT. */
17372 /* Perform (expensive) checks on free */
17373 #define SLAB_DEBUG_FREE 0x00000100UL
17374 /* Call constructor (as verifier) */
17375 #define SLAB_DEBUG_INITIAL 0x00000200UL
17376 /* Red zone objs in a cache */
17377 #define SLAB_RED_ZONE 0x00000400UL
17378 /* Poison objects */
17379 #define SLAB_POISON 0x00000800UL
17380 /* never reap from the cache */
17381 #define SLAB_NO_REAP 0x00001000UL

```

```

17382 /* align objs on a h/w cache lines */
17383 #define SLAB_HWCACHE_ALIGN 0x00002000UL
17384 #if 0
17385 #define SLAB_HIGH_PACK 0x00004000UL /* XXX */
17386 #endif
17387
17388 /* flags passed to a constructor func */
17389 /* if not set, then deconstructor */
17390 #define SLAB_CTOR_CONSTRUCTOR 0x001UL
17391 /* tell constructor it can't sleep */
17392 #define SLAB_CTOR_ATOMIC 0x002UL
17393 /* tell constructor it's a verify call */
17394 #define SLAB_CTOR_VERIFY 0x004UL
17395
17396 /* prototypes */
17397 extern long kmem_cache_init(long, long);
17398 extern void kmem_cache_sizes_init(void);
17399 extern kmem_cache_t *kmem_find_general_cachep(size_t);
17400 extern kmem_cache_t *kmem_cache_create(const char *,
17401 size_t, size_t, unsigned long,
17402 void (*)(void *, kmem_cache_t *, unsigned long),
17403 void (*)(void *, kmem_cache_t *, unsigned long));
17404 extern int kmem_cache_shrink(kmem_cache_t *);
17405 extern void *kmem_cache_alloc(kmem_cache_t *, int);
17406 extern void kmem_cache_free(kmem_cache_t *, void *);
17407
17408 extern void *kmallocc(size_t, int);
17409 extern void kfree(const void *);
17410 extern void kfree_s(const void *, size_t);
17411
17412 extern void kmem_cache_reap(int);
17413 extern int get_slabinfo(char *);
17414
17415 /* System wide caches */
17416 extern kmem_cache_t *vm_area_cachep;
17417 extern kmem_cache_t *mm_cachep;
17418
17419 #endif /* __KERNEL__ */
17420
17421 #endif /* _LINUX_SLAB_H */

```

include/linux/smp.h

```

17422 #ifndef __LINUX_SMP_H
17423 #define __LINUX_SMP_H
17424
17425 /* Generic SMP support
17426  * Alan Cox, <alan@cymru.net>

```

```

17427 */
17428
17429 #ifdef __SMP__
17430
17431 #include <asm/smp.h>
17432
17433 /* main cross-CPU interfaces, handles INIT, TLB flush,
17434 * STOP, etc. (defined in asm header): */
17435
17436 /* stops all CPUs but the current one: */
17437 extern void smp_send_stop(void);
17438
17439 /* sends a 'reschedule' event to another CPU: */
17440 extern void FASTCALL(smp_send_reschedule(int cpu));
17441
17442 /* Boot processor call to load the other CPU's */
17443 extern void smp_boot_cpus(void);
17444
17445 /* Processor call in. Must hold processors until .. */
17446 extern void smp_callin(void);
17447
17448 /* Multiprocessors may now schedule */
17449 extern void smp_commence(void);
17450
17451 /* True once the per process idle is forked */
17452 extern int smp_threads_ready;
17453
17454 extern int smp_num_cpus;
17455
17456 extern volatile unsigned long smp_msg_data;
17457 extern volatile int smp_src_cpu;
17458 extern volatile int smp_msg_id;
17459
17460 /* Assume <32768 CPUs */
17461 #define MSG_ALL_BUT_SELF 0x8000
17462 #define MSG_ALL 0x8001
17463
17464 /* Remote processor TLB invalidate */
17465 #define MSG_INVALIDATE_TLB 0x0001
17466 /* Sent to shut down slave CPUs when rebooting */
17467 #define MSG_STOP_CPU 0x0002
17468 /* Reschedule request from master CPU */
17469 #define MSG_RESCHEDULE 0x0003
17470 /* Change MTRR */
17471 #define MSG_MTRR_CHANGE 0x0004
17472
17473 #else
17474

```

```

17475 /* These macros fold the SMP functionality into a single
17476 * CPU system */
17477
17478 #define smp_num_cpus 1
17479 #define smp_processor_id() 0
17480 #define hard_smp_processor_id() 0
17481 #define smp_threads_ready 1
17482 #define kernel_lock()
17483 #define cpu_logical_map(cpu) 0
17484
17485 #endif
17486 #endif

```

include/linux/smp_lock.h

```

17487 #ifndef __LINUX_SMPLOCK_H
17488 #define __LINUX_SMPLOCK_H
17489
17490 #ifdef __SMP__
17491
17492 #define lock_kernel() do { } while(0)
17493 #define unlock_kernel() do { } while(0)
17494 #define release_kernel_lock(task, cpu) do { } while(0)
17495 #define reacquire_kernel_lock(task) do { } while(0)
17496
17497 #else
17498
17499 #include <asm/smplock.h>
17500
17501 #endif /* __SMP__ */
17502
17503 #endif

```

include/linux/swap.h

```

17504 #ifndef _LINUX_SWAP_H
17505 #define _LINUX_SWAP_H
17506
17507 #include <asm/page.h>
17508
17509 /* set if swap priority specified */
17510 #define SWAP_FLAG_PREFER 0x8000
17511 #define SWAP_FLAG_PRIO_MASK 0x7fff
17512 #define SWAP_FLAG_PRIO_SHIFT 0
17513
17514 #define MAX_SWAPFILES 8
17515
17516 union swap_header {

```

```

17517 struct
17518 {
17519     char reserved[PAGE_SIZE - 10];
17520     char magic[10];
17521 } magic;
17522 struct
17523 {
17524     char bootbits[1024]; /* Space for disklabel etc. */
17525     unsigned int version;
17526     unsigned int last_page;
17527     unsigned int nr_badpages;
17528     unsigned int padding[125];
17529     unsigned int badpages[1];
17530 } info;
17531 };
17532
17533 #ifdef __KERNEL__
17534
17535 /* Max bad pages in the new format.. */
17536 #define __swapoffset(x) \
17537     ((unsigned long)&((union swap_header *)0)->x)
17538 #define MAX_SWAP_BADPAGES \
17539     ((__swapoffset(magic.magic) - \
17540      __swapoffset(info.badpages)) / sizeof(int))
17541
17542 #undef DEBUG_SWAP
17543
17544 #include <asm/atomic.h>
17545
17546 #define SWP_USED      1
17547 #define SWP_WRITEOK  3
17548
17549 #define SWAP_CLUSTER_MAX 32
17550
17551 #define SWAP_MAP_MAX  0x7fff
17552 #define SWAP_MAP_BAD  0x8000
17553
17554 struct swap_info_struct {
17555     unsigned int flags;
17556     kdev_t swap_device;
17557     struct dentry * swap_file;
17558     unsigned short * swap_map;
17559     unsigned char * swap_lockmap;
17560     unsigned int lowest_bit;
17561     unsigned int highest_bit;
17562     unsigned int cluster_next;
17563     unsigned int cluster_nr;
17564     int prio; /* swap priority */
17565     int pages;
17566     unsigned long max;
17567     int next; /* next entry on swap list */
17568 };
17569
17570 extern int nr_swap_pages;
17571 extern int nr_free_pages;
17572 extern atomic_t nr_async_pages;
17573 extern struct inode swapper_inode;
17574 extern unsigned long page_cache_size;
17575 extern int buffermem;
17576
17577 /* Incomplete types for prototype declarations: */
17578 struct task_struct;
17579 struct vm_area_struct;
17580 struct sysinfo;
17581
17582 /* linux/ipc/shm.c */
17583 extern int shm_swap (int, int);
17584
17585 /* linux/mm/swap.c */
17586 extern void swap_setup (void);
17587
17588 /* linux/mm/vmscan.c */
17589 extern int try_to_free_pages(unsigned int gfp_mask);
17590
17591 /* linux/mm/page_io.c */
17592 extern void rw_swap_page(int, unsigned long, char *,int);
17593 extern void rw_swap_page_nocache(int, unsigned long,
17594                                  char *);
17595 extern void rw_swap_page_nolock(int, unsigned long,
17596                                 char *, int);
17597 extern void swap_after_unlock_page (unsigned long entry);
17598
17599 /* linux/mm/page_alloc.c */
17600 extern void swap_in(struct task_struct *,
17601                    struct vm_area_struct *, pte_t *, unsigned long, int);
17602
17603
17604 /* linux/mm/swap_state.c */
17605 extern void show_swap_cache_info(void);
17606 extern int add_to_swap_cache(struct page *,
17607                             unsigned long);
17608 extern int swap_duplicate(unsigned long);
17609 extern int swap_check_entry(unsigned long);
17610 struct page * lookup_swap_cache(unsigned long);
17611 extern struct page * read_swap_cache_async(unsigned long,
17612                                             int);

```

```

17613 #define read_swap_cache(entry) \
17614     read_swap_cache_async(entry, 1);
17615 extern int FASTCALL(swap_count(unsigned long));
17616 /* Make these inline later once they are working
17617  * properly. */
17618 extern void delete_from_swap_cache(struct page *page);
17619 extern void free_page_and_swap_cache(unsigned long addr);
17620
17621 /* linux/mm/swapfile.c */
17622 extern unsigned int nr_swapfiles;
17623 extern struct swap_info_struct swap_info[];
17624 void si_swapinfo(struct sysinfo *);
17625 unsigned long get_swap_page(void);
17626 extern void FASTCALL(swap_free(unsigned long));
17627 struct swap_list_t {
17628     int head; /* head of priority-ordered swapfile list */
17629     int next; /* swapfile to be used next */
17630 };
17631 extern struct swap_list_t swap_list;
17632 asmlinkage int sys_swapoff(const char *);
17633 asmlinkage int sys_swapon(const char *, int);
17634
17635 /* vm_ops not present page codes for shared memory.
17636  *
17637  * Will go away eventually.. */
17638 #define SHM_SWP_TYPE 0x20
17639
17640 /* swap cache stuff (in linux/mm/swap_state.c) */
17641
17642 #define SWAP_CACHE_INFO
17643
17644 #ifdef SWAP_CACHE_INFO
17645 extern unsigned long swap_cache_add_total;
17646 extern unsigned long swap_cache_del_total;
17647 extern unsigned long swap_cache_find_total;
17648 extern unsigned long swap_cache_find_success;
17649 #endif
17650
17651 extern inline unsigned long
17652 in_swap_cache(struct page *page)
17653 {
17654     if (PageSwapCache(page))
17655         return page->offset;
17656     return 0;
17657 }
17658
17659 /* Work out if there are any other processes sharing this
17660  * page, ignoring any page reference coming from the swap

```

```

17661  * cache, or from outstanding swap IO on this page. (The
17662  * page cache _does_ count as another valid reference to
17663  * the page, however.) */
17664 static inline int is_page_shared(struct page *page)
17665 {
17666     unsigned int count;
17667     if (PageReserved(page))
17668         return 1;
17669     count = atomic_read(&page->count);
17670     if (PageSwapCache(page))
17671         count += swap_count(page->offset) - 2;
17672     if (PageFreeAfter(page))
17673         count--;
17674     return count > 1;
17675 }
17676
17677 #endif /* __KERNEL__ */
17678
17679 #endif /* _LINUX_SWAP_H */

```

include/linux/swapctl.h

```

17680 #ifndef _LINUX_SWAPCTL_H
17681 #define _LINUX_SWAPCTL_H
17682
17683 #include <asm/page.h>
17684 #include <linux/fs.h>
17685
17686 typedef struct buffer_mem_v1
17687 {
17688     unsigned int    min_percent;
17689     unsigned int    borrow_percent;
17690     unsigned int    max_percent;
17691 } buffer_mem_v1;
17692 typedef buffer_mem_v1 buffer_mem_t;
17693 extern buffer_mem_t buffer_mem;
17694 extern buffer_mem_t page_cache;
17695
17696 typedef struct freepages_v1
17697 {
17698     unsigned int    min;
17699     unsigned int    low;
17700     unsigned int    high;
17701 } freepages_v1;
17702 typedef freepages_v1 freepages_t;
17703 extern freepages_t freepages;
17704
17705 typedef struct pager daemon_v1

```

```

17706 {
17707     unsigned int    tries_base;
17708     unsigned int    tries_min;
17709     unsigned int    swap_cluster;
17710 } pager_daemon_v1;
17711 typedef pager_daemon_v1 pager_daemon_t;
17712 extern pager_daemon_t pager_daemon;
17713
17714 #endif /* _LINUX_SWAPCTL_H */

```

include/linux/sysctl.h

```

17715 /*
17716  * sysctl.h: General linux system control interface
17717  *
17718  * Begun 24 March 1995, Stephen Tweedie
17719  *
17720  * *****
17721  * *****
17722  **
17723  ** WARNING:
17724  ** The values in this file are exported to user space
17725  ** via the sysctl() binary interface. Do *NOT* change
17726  ** the numbering of any existing values here, and do
17727  ** not change any numbers within any one set of values.
17728  ** If you have to redefine an existing interface, use a
17729  ** new number for it. The kernel will then return
17730  ** ENOTDIR to any application using the old binary
17731  ** interface.
17732  **
17733  ** --sct
17734  **
17735  * *****
17736  * *****/
17737
17738 #include <linux/lists.h>
17739
17740 #ifndef _LINUX_SYSCTL_H
17741 #define _LINUX_SYSCTL_H
17742
17743 #define CTL_MAXNAME 10
17744
17745 struct __sysctl_args {
17746     int *name;
17747     int nlen;
17748     void *oldval;
17749     size_t *oldlenp;
17750     void *newval;

```

```

17751     size_t newlen;
17752     unsigned long __unused[4];
17753 };
17754
17755 /* Define sysctl names first */
17756
17757 /* Top-level names: */
17758
17759 /* For internal pattern-matching use only: */
17760 #ifdef __KERNEL__
17761 #define CTL_ANY -1 /* Matches any name */
17762 #define CTL_NONE 0
17763 #endif
17764
17765 enum
17766 {
17767     CTL_KERN=1, /* General kernel info and control */
17768     CTL_VM=2, /* VM management */
17769     CTL_NET=3, /* Networking */
17770     CTL_PROC=4, /* Process info */
17771     CTL_FS=5, /* Filesystems */
17772     CTL_DEBUG=6, /* Debugging */
17773     CTL_DEV=7, /* Devices */
17774 };
17775
17776
17777 /* CTL_KERN names: */
17778 enum
17779 {
17780     /* string: system version */
17781     KERN_OSTYPE=1,
17782     /* string: system release */
17783     KERN_OSRELEASE=2,
17784     /* int: system revision */
17785     KERN_OSREV=3,
17786     /* string: compile time info */
17787     KERN_VERSION=4,
17788     /* struct: maximum rights mask */
17789     KERN_SECUREMASK=5,
17790     /* table: profiling information */
17791     KERN_PROF=6,
17792     KERN_NODENAME=7,
17793     KERN_DOMAINNAME=8,
17794
17795     /* int: system security level */
17796     KERN_SECURELVL=14,
17797     /* int: panic timeout */
17798     KERN_PANIC=15,

```



```

17799 /* real root device to mount after initrd */
17800 KERN_REALROOTDEV=16,
17801
17802 /* path to Java(tm) interpreter */
17803 KERN_JAVA_INTERPRETER=19,
17804 /* path to Java(tm) appletviewer */
17805 KERN_JAVA_APPLETVIEWER=20,
17806 /* reboot command on Sparc */
17807 KERN_SPARC_REBOOT=21,
17808 /* int: allow ctrl-alt-del to reboot */
17809 KERN_CTLALTDDEL=22,
17810 /* struct: control printk logging parameters */
17811 KERN_PRINTK=23,
17812 /* Name translation */
17813 KERN_NAMETRANS=24,
17814 /* turn htab reclamation on/off on PPC */
17815 KERN_PPC_HTABRECLAIM=25,
17816 /* turn idle page zeroing on/off on PPC */
17817 KERN_PPC_ZEROPAGED=26,
17818 /* use nap mode for power saving */
17819 KERN_PPC_POWERSAVE_NAP=27,
17820 KERN_MODPROBE=28,
17821 KERN_SG_BIG_BUFF=29,
17822 /* BSD process accounting parameters */
17823 KERN_ACCT=30,
17824 /* l2cr register on PPC */
17825 KERN_PPC_L2CR=31,
17826
17827 /* Number of rt sigs queued */
17828 KERN_RTISGNR=32,
17829 /* Max queueable */
17830 KERN_RTIGMAX=33,
17831
17832 /* int: Maximum shared memory segment */
17833 KERN_SHMMAX=34,
17834 /* int: Maximum size of a message */
17835 KERN_MSGMAX=35,
17836 /* int: Maximum message queue size */
17837 KERN_MSGMNB=36,
17838 /* int: Maximum system message pool size */
17839 KERN_MSGPOOL=37
17840 };
17841
17842
17843 /* CTL_VM names: */
17844 enum
17845 {
17846 /* struct: Set vm swapping control */

```

```

17847 VM_SWAPCTL=1,
17848 /* int: Background pageout interval */
17849 VM_SWAPOUT=2,
17850 /* struct: Set free page thresholds */
17851 VM_FREEPG=3,
17852 /* struct: Control buffer cache flushing */
17853 VM_BDFLUSH=4,
17854 /* Turn off the virtual memory safety limit */
17855 VM_OVERCOMMIT_MEMORY=5,
17856 /* struct: Set buffer memory thresholds */
17857 VM_BUFFERMEM=6,
17858 /* struct: Set cache memory thresholds */
17859 VM_PAGECACHE=7,
17860 /* struct: Control kswapd behaviour */
17861 VM_PAGERDAEMON=8,
17862 /* struct: Set page table cache parameters */
17863 VM_PGT_CACHE=9,
17864 /* int: set number of pages to swap together */
17865 VM_PAGE_CLUSTER=10
17866 };
17867
17868
17869 /* CTL_NET names: */
17870 enum
17871 {
17872 NET_CORE=1,
17873 NET_ETHER=2,
17874 NET_802=3,
17875 NET_UNIX=4,
17876 NET_IPV4=5,
17877 NET_IPX=6,
17878 NET_ATALK=7,
17879 NET_NETROM=8,
17880 NET_AX25=9,
17881 NET_BRIDGE=10,
17882 NET_ROSE=11,
17883 NET_IPV6=12,
17884 NET_X25=13,
17885 NET_TR=14,
17886 NET_DEFCONF=15
17887 };
17888
17889
17890 /* /proc/sys/net/core */
17891 enum
17892 {
17893 NET_CORE_WMEM_MAX=1,
17894 NET_CORE_RMEM_MAX=2,

```

```

17895 NET_CORE_WMEM_DEFAULT=3,
17896 NET_CORE_RMEM_DEFAULT=4,
17897 /* was NET_CORE_DESTROY_DELAY */
17898 NET_CORE_MAX_BACKLOG=6,
17899 NET_CORE_FASTROUTE=7,
17900 NET_CORE_MSG_COST=8,
17901 NET_CORE_MSG_BURST=9,
17902 NET_CORE_OPTMEM_MAX=10
17903 };
17904
17905 /* /proc/sys/net/ethernet */
17906
17907 /* /proc/sys/net/802 */
17908
17909 /* /proc/sys/net/unix */
17910
17911 enum
17912 {
17913     NET_UNIX_DESTROY_DELAY=1,
17914     NET_UNIX_DELETE_DELAY=2,
17915     NET_UNIX_MAX_DGRAM_QLEN=3,
17916 };
17917
17918 /* /proc/sys/net/ipv4 */
17919 enum
17920 {
17921     /* v2.0 compatible variables */
17922     NET_IPV4_FORWARD=8,
17923     NET_IPV4_DYNADDR=9,
17924
17925     NET_IPV4_CONF=16,
17926     NET_IPV4_NEIGH=17,
17927     NET_IPV4_ROUTE=18,
17928     NET_IPV4_FIB_HASH=19,
17929
17930     NET_IPV4_TCP_TIMESTAMPS=33,
17931     NET_IPV4_TCP_WINDOW_SCALING=34,
17932     NET_IPV4_TCP_SACK=35,
17933     NET_IPV4_TCP_RETRANS_COLLAPSE=36,
17934     NET_IPV4_DEFAULT_TTL=37,
17935     NET_IPV4_AUTOCONFIG=38,
17936     NET_IPV4_NO_PMTU_DISC=39,
17937     NET_IPV4_TCP_SYN_RETRIES=40,
17938     NET_IPV4_IPFRAG_HIGH_THRESH=41,
17939     NET_IPV4_IPFRAG_LOW_THRESH=42,
17940     NET_IPV4_IPFRAG_TIME=43,
17941     NET_IPV4_TCP_MAX_KA_PROBES=44,
17942     NET_IPV4_TCP_KEEPA_LIVE_TIME=45,
17943     NET_IPV4_TCP_KEEPA_LIVE_PROBES=46,
17944     NET_IPV4_TCP_RETRIES1=47,
17945     NET_IPV4_TCP_RETRIES2=48,
17946     NET_IPV4_TCP_FIN_TIMEOUT=49,
17947     NET_IPV4_IP_MASQ_DEBUG=50,
17948     NET_TCP_SYNCOOKIES=51,
17949     NET_TCP_STDURG=52,
17950     NET_TCP_RFC1337=53,
17951     NET_TCP_SYN_TAILDROP=54,
17952     NET_TCP_MAX_SYN_BACKLOG=55,
17953     NET_IPV4_LOCAL_PORT_RANGE=56,
17954     NET_IPV4_ICMP_ECHO_IGNORE_ALL=57,
17955     NET_IPV4_ICMP_ECHO_IGNORE_BROADCASTS=58,
17956     NET_IPV4_ICMP_SOURCEQUENCH_RATE=59,
17957     NET_IPV4_ICMP_DESTUNREACH_RATE=60,
17958     NET_IPV4_ICMP_TIMEEXCEED_RATE=61,
17959     NET_IPV4_ICMP_PARAMPROB_RATE=62,
17960     NET_IPV4_ICMP_ECHOREPLY_RATE=63,
17961     NET_IPV4_ICMP_IGNORE_BOGUS_ERROR_RESPONSES=64,
17962     NET_IPV4_IGMP_MAX_MEMBERSHIPS=65
17963 };
17964
17965 enum {
17966     NET_IPV4_ROUTE_FLUSH=1,
17967     NET_IPV4_ROUTE_MIN_DELAY=2,
17968     NET_IPV4_ROUTE_MAX_DELAY=3,
17969     NET_IPV4_ROUTE_GC_THRESH=4,
17970     NET_IPV4_ROUTE_MAX_SIZE=5,
17971     NET_IPV4_ROUTE_GC_MIN_INTERVAL=6,
17972     NET_IPV4_ROUTE_GC_TIMEOUT=7,
17973     NET_IPV4_ROUTE_GC_INTERVAL=8,
17974     NET_IPV4_ROUTE_REDIRECT_LOAD=9,
17975     NET_IPV4_ROUTE_REDIRECT_NUMBER=10,
17976     NET_IPV4_ROUTE_REDIRECT_SILENCE=11,
17977     NET_IPV4_ROUTE_ERROR_COST=12,
17978     NET_IPV4_ROUTE_ERROR_BURST=13,
17979     NET_IPV4_ROUTE_GC_ELASTICITY=14,
17980     NET_IPV4_ROUTE_MTU_EXPIRES=15
17981 };
17982
17983 enum
17984 {
17985     NET_PROTO_CONF_ALL=-2,
17986     NET_PROTO_CONF_DEFAULT= 3
17987
17988     /* And device ifindices ... */
17989 };
17990

```

```

17991 enum
17992 {
17993     NET_IPV4_CONF_FORWARDING=1,
17994     NET_IPV4_CONF_MC_FORWARDING=2,
17995     NET_IPV4_CONF_PROXY_ARP=3,
17996     NET_IPV4_CONF_ACCEPT_REDIRECTS=4,
17997     NET_IPV4_CONF_SECURE_REDIRECTS=5,
17998     NET_IPV4_CONF_SEND_REDIRECTS=6,
17999     NET_IPV4_CONF_SHARED_MEDIA=7,
18000     NET_IPV4_CONF_RP_FILTER=8,
18001     NET_IPV4_CONF_ACCEPT_SOURCE_ROUTE=9,
18002     NET_IPV4_CONF_BOOTP_RELAY=10,
18003     NET_IPV4_CONF_LOG_MARTIANS=11
18004 };
18005
18006 /* /proc/sys/net/ipv6 */
18007 enum {
18008     NET_IPV6_CONF=16,
18009     NET_IPV6_NEIGH=17,
18010     NET_IPV6_ROUTE=18
18011 };
18012
18013 enum {
18014     NET_IPV6_ROUTE_FLUSH=1,
18015     NET_IPV6_ROUTE_GC_THRESH=2,
18016     NET_IPV6_ROUTE_MAX_SIZE=3,
18017     NET_IPV6_ROUTE_GC_MIN_INTERVAL=4,
18018     NET_IPV6_ROUTE_GC_TIMEOUT=5,
18019     NET_IPV6_ROUTE_GC_INTERVAL=6,
18020     NET_IPV6_ROUTE_GC_ELASTICITY=7,
18021     NET_IPV6_ROUTE_MTU_EXPIRES=8
18022 };
18023
18024 enum {
18025     NET_IPV6_FORWARDING=1,
18026     NET_IPV6_HOP_LIMIT=2,
18027     NET_IPV6_MTU=3,
18028     NET_IPV6_ACCEPT_RA=4,
18029     NET_IPV6_ACCEPT_REDIRECTS=5,
18030     NET_IPV6_AUTOCONF=6,
18031     NET_IPV6_DAD_TRANSMITS=7,
18032     NET_IPV6_RTR_SOLICITS=8,
18033     NET_IPV6_RTR_SOLICIT_INTERVAL=9,
18034     NET_IPV6_RTR_SOLICIT_DELAY=10
18035 };
18036
18037 /* /proc/sys/net/<protocol>/neigh/<dev> */
18038 enum {

```

```

18039     NET_NEIGH_MCAST_SOLICIT=1,
18040     NET_NEIGH_UCAST_SOLICIT=2,
18041     NET_NEIGH_APP_SOLICIT=3,
18042     NET_NEIGH_RETRANS_TIME=4,
18043     NET_NEIGH_REACHABLE_TIME=5,
18044     NET_NEIGH_DELAY_PROBE_TIME=6,
18045     NET_NEIGH_GC_STALE_TIME=7,
18046     NET_NEIGH_UNRES_QLEN=8,
18047     NET_NEIGH_PROXY_QLEN=9,
18048     NET_NEIGH_ANYCAST_DELAY=10,
18049     NET_NEIGH_PROXY_DELAY=11,
18050     NET_NEIGH_LOCKTIME=12,
18051     NET_NEIGH_GC_INTERVAL=13,
18052     NET_NEIGH_GC_THRESH1=14,
18053     NET_NEIGH_GC_THRESH2=15,
18054     NET_NEIGH_GC_THRESH3=16
18055 };
18056
18057 /* /proc/sys/net/ipv6 */
18058
18059
18060 /* /proc/sys/net/appletalk */
18061 enum {
18062     NET_ATALK_AARP_EXPIRY_TIME=1,
18063     NET_ATALK_AARP_TICK_TIME=2,
18064     NET_ATALK_AARP_RETRANSMIT_LIMIT=3,
18065     NET_ATALK_AARP_RESOLVE_TIME=4
18066 };
18067
18068
18069 /* /proc/sys/net/netrom */
18070 enum {
18071     NET_NETROM_DEFAULT_PATH_QUALITY=1,
18072     NET_NETROM_OBSOLESCENCE_COUNT_INITIALISER=2,
18073     NET_NETROM_NETWORK_TTL_INITIALISER=3,
18074     NET_NETROM_TRANSPORT_TIMEOUT=4,
18075     NET_NETROM_TRANSPORT_MAXIMUM_TRIES=5,
18076     NET_NETROM_TRANSPORT_ACKNOWLEDGE_DELAY=6,
18077     NET_NETROM_TRANSPORT_BUSY_DELAY=7,
18078     NET_NETROM_TRANSPORT_REQUESTED_WINDOW_SIZE=8,
18079     NET_NETROM_TRANSPORT_NO_ACTIVITY_TIMEOUT=9,
18080     NET_NETROM_ROUTING_CONTROL=10,
18081     NET_NETROM_LINK_FAILS_COUNT=11
18082 };
18083
18084 /* /proc/sys/net/ax25 */
18085 enum {
18086     NET_AX25_IP_DEFAULT_MODE=1,

```

```

18087 NET_AX25_DEFAULT_MODE=2,
18088 NET_AX25_BACKOFF_TYPE=3,
18089 NET_AX25_CONNECT_MODE=4,
18090 NET_AX25_STANDARD_WINDOW=5,
18091 NET_AX25_EXTENDED_WINDOW=6,
18092 NET_AX25_T1_TIMEOUT=7,
18093 NET_AX25_T2_TIMEOUT=8,
18094 NET_AX25_T3_TIMEOUT=9,
18095 NET_AX25_IDLE_TIMEOUT=10,
18096 NET_AX25_N2=11,
18097 NET_AX25_PACLEN=12,
18098 NET_AX25_PROTOCOL=13,
18099 NET_AX25_DAMA_SLAVE_TIMEOUT=14
18100 };
18101
18102 /* /proc/sys/net/rose */
18103 enum {
18104     NET_ROSE_RESTART_REQUEST_TIMEOUT=1,
18105     NET_ROSE_CALL_REQUEST_TIMEOUT=2,
18106     NET_ROSE_RESET_REQUEST_TIMEOUT=3,
18107     NET_ROSE_CLEAR_REQUEST_TIMEOUT=4,
18108     NET_ROSE_ACK_HOLD_BACK_TIMEOUT=5,
18109     NET_ROSE_ROUTING_CONTROL=6,
18110     NET_ROSE_LINK_FAIL_TIMEOUT=7,
18111     NET_ROSE_MAX_VCS=8,
18112     NET_ROSE_WINDOW_SIZE=9,
18113     NET_ROSE_NO_ACTIVITY_TIMEOUT=10
18114 };
18115
18116 /* /proc/sys/net/x25 */
18117 enum {
18118     NET_X25_RESTART_REQUEST_TIMEOUT=1,
18119     NET_X25_CALL_REQUEST_TIMEOUT=2,
18120     NET_X25_RESET_REQUEST_TIMEOUT=3,
18121     NET_X25_CLEAR_REQUEST_TIMEOUT=4,
18122     NET_X25_ACK_HOLD_BACK_TIMEOUT=5
18123 };
18124
18125 /* /proc/sys/net/token-ring */
18126 enum
18127 {
18128     NET_TR_RIF_TIMEOUT=1
18129 };
18130
18131 /* /proc/sys/net/decnet */
18132 enum {
18133     NET_DECNET_DEF_T3_BROADCAST=1,
18134     NET_DECNET_DEF_T3_POINTTOPOINT=2,

```

```

18135     NET_DECNET_DEF_T1=3,
18136     NET_DECNET_DEF_BCT1=4,
18137     NET_DECNET_CACHETIMEOUT=5,
18138     NET_DECNET_DEBUG_LEVEL=6
18139 };
18140
18141 /* CTL_PROC names: */
18142
18143 /* CTL_FS names: */
18144 enum
18145 {
18146     /* int:current number of allocated inodes */
18147     FS_NRINODE=1,
18148     FS_STATINODE=2,
18149     /* int:max number of inodes that can be allocated */
18150     FS_MAXINODE=3,
18151     /* int:current number of allocated dquots */
18152     FS_NRDQUOT=4,
18153     /* int:max number of dquots that can be allocated */
18154     FS_MAXDQUOT=5,
18155     /* int:current number of allocated filedescriptors */
18156     FS_NRFIL=6,
18157     /* int:max #of filedescriptors that can be allocated */
18158     FS_MAXFILE=7,
18159     FS_DENTRY=8,
18160     /* int:current number of allocated super blocks */
18161     FS_NRSUPER=9,
18162     /* int:max # of super blocks that can be allocated */
18163     FS_MAXSUPER=10
18164 };
18165
18166 /* CTL_DEBUG names: */
18167
18168 /* CTL_DEV names: */
18169 enum {
18170     DEV_CDROM=1,
18171     DEV_HWMON=2
18172 };
18173
18174 /* /proc/sys/dev/cdrom */
18175 enum {
18176     DEV_CDROM_INFO=1
18177 };
18178
18179 #ifdef KERNEL
18180
18181 extern asmlinkage int sys_sysctl(struct sysctl_args *);
18182 extern void sysctl_init(void);

```

```

18183
18184 typedef struct ctl_table ctl_table;
18185
18186 typedef int ctl_handler(ctl_table *table,
18187                         int *name, int nlen,
18188                         void *oldval, size_t *oldlenp,
18189                         void *newval, size_t newlen,
18190                         void **context);
18191
18192 typedef int proc_handler(ctl_table *ctl, int write,
18193                          struct file * filp, void *buffer, size_t *lenp);
18194
18195 extern int proc_dostring(ctl_table *, int, struct file *,
18196                          void *, size_t *);
18197 extern int proc_dointvec(ctl_table *, int, struct file *,
18198                          void *, size_t *);
18199 extern int proc_dointvec_minmax(ctl_table *, int,
18200                                  struct file *, void *, size_t *);
18201 extern int proc_dointvec_jiffies(ctl_table *, int,
18202                                  struct file *, void *, size_t *);
18203
18204 extern int do_sysctl (int *name, int nlen,
18205                      void *oldval, size_t *oldlenp,
18206                      void *newval, size_t newlen);
18207
18208 extern int do_sysctl_strategy (ctl_table *table,
18209                               int *name, int nlen,
18210                               void *oldval, size_t *oldlenp,
18211                               void *newval, size_t newlen,
18212                               void ** context);
18213
18214 extern ctl_handler sysctl_string;
18215 extern ctl_handler sysctl_intvec;
18216
18217 extern int do_string (
18218     void *oldval, size_t *oldlenp,
18219     void *newval, size_t newlen,
18220     int rdwr, char *data, size_t max);
18221 extern int do_int (
18222     void *oldval, size_t *oldlenp,
18223     void *newval, size_t newlen,
18224     int rdwr, int *data);
18225 extern int do_struct (
18226     void *oldval, size_t *oldlenp,
18227     void *newval, size_t newlen,
18228     int rdwr, void *data, size_t len);
18229
18230

```

```

18231 /* Register a set of sysctl names by calling
18232 * register_sysctl_table with an initialised array of
18233 * ctl_table's. An entry with zero ctl_name terminates
18234 * the table. table->de will be set up by the
18235 * registration and need not be initialised in advance.
18236 *
18237 * sysctl names can be mirrored automatically under
18238 * /proc/sys. The procname supplied controls /proc
18239 * naming.
18240 *
18241 * The table's mode will be honoured both for
18242 * sys_sysctl(2) and proc fs access.
18243 *
18244 * Leaf nodes in the sysctl tree will be represented by a
18245 * single file under /proc; non-leaf nodes will be
18246 * represented by directories. A null procname disables
18247 * /proc mirroring at this node.
18248 *
18249 * sysctl(2) can automatically manage read and write
18250 * requests through the sysctl table. The data and
18251 * maxlen fields of the ctl_table struct enable minimal
18252 * validation of the values being written to be
18253 * performed, and the mode field allows minimal
18254 * authentication.
18255 *
18256 * More sophisticated management can be enabled by the
18257 * provision of a strategy routine with the table entry.
18258 * This will be called before any automatic read or write
18259 * of the data is performed.
18260 *
18261 * The strategy routine may return:
18262 * <0: Error occurred (error is passed to user process)
18263 * 0: OK - proceed with automatic read or write.
18264 * >0: OK - read or write has been done by the strategy
18265 * routine, so return immediately.
18266 *
18267 * There must be a proc_handler routine for any terminal
18268 * nodes mirrored under /proc/sys (non-terminals are
18269 * handled by a built-in directory handler). Several
18270 * default handlers are available to cover common cases.
18271 */
18272
18273 /* A sysctl table is an array of struct ctl_table: */
18274 struct ctl_table
18275 {
18276     int ctl_name; /* Binary ID */
18277     const char *procname; /* Text ID for /proc/sys, or 0 */
18278     void *data;

```

```

18279 int maxlen;
18280 mode_t mode;
18281 ctl_table *child;
18282 proc_handler *proc_handler; /* CB for text formatting*/
18283 ctl_handler *strategy; /* CB fn for all r/w */
18284 struct proc_dir_entry *de; /* /proc control block */
18285 void *extra1;
18286 void *extra2;
18287 };
18288
18289 /* struct ctl_table_header is used to maintain dynamic
18290 * lists of ctl_table trees. */
18291 struct ctl_table_header
18292 {
18293     ctl_table *ctl_table;
18294     DLNODE(struct ctl_table_header) ctl_entry;
18295 };
18296
18297 struct ctl_table_header * register_sysctl_table(
18298     ctl_table * table, int insert_at_head);
18299 void unregister_sysctl_table(
18300     struct ctl_table_header * table);
18301
18302 #else /* __KERNEL__ */
18303
18304 #endif /* __KERNEL__ */
18305
18306 #endif /* _LINUX_SYSCTL_H */

```

include/linux/tasks.h

```

18307 #ifndef _LINUX_TASKS_H
18308 #define _LINUX_TASKS_H
18309
18310 /* This is the maximum nr of tasks - change it if you
18311 * need to */
18312 #ifdef __SMP__
18313 /* Max processors that can be running in SMP */
18314 #define NR_CPUS 32
18315 #else
18316 #define NR_CPUS 1
18317 #endif
18318
18319 /* On x86 Max 4092, or 4090 w/APM configured. */
18320 #define NR_TASKS 512
18321
18322 #define MAX_TASKS_PER_USER (NR_TASKS/2)
18323 #define MIN_TASKS_LEFT_FOR_ROOT 4
18324

```

```

18325
18326 /* This controls the max pid allocated to a process */
18327 #define PID_MAX 0x8000
18328
18329 #endif

```

include/linux/time.h

```

18330 #ifndef _LINUX_TIME_H
18331 #define _LINUX_TIME_H
18332
18333 #include <asm/param.h>
18334 #include <linux/types.h>
18335
18336 #ifndef _STRUCT_TIMESPEC
18337 #define _STRUCT_TIMESPEC
18338 struct timespec {
18339     time_t tv_sec; /* seconds */
18340     long tv_nsec; /* nanoseconds */
18341 };
18342 #endif /* _STRUCT_TIMESPEC */
18343
18344 /* Change timeval to jiffies, trying to avoid the most
18345 * obvious overflows..
18346 *
18347 * And some not so obvious.
18348 *
18349 * Note that we don't want to return MAX LONG, because
18350 * for various timeout reasons we often end up having to
18351 * wait "jiffies+1" in order to guarantee that we wait at
18352 * _least_ "jiffies" - so "jiffies+1" had better still be
18353 * positive. */
18354 #define MAX_JIFFY_OFFSET ((~0UL >> 1) 1)
18355
18356 static __inline__ unsigned long
18357 timespec_to_jiffies(struct timespec *value)
18358 {
18359     unsigned long sec = value->tv_sec;
18360     long nsec = value->tv_nsec;
18361
18362     if (sec >= (MAX_JIFFY_OFFSET / HZ))
18363         return MAX_JIFFY_OFFSET;
18364     nsec += 1000000000L / HZ - 1;
18365     nsec /= 1000000000L / HZ;
18366     return HZ * sec + nsec;
18367 }
18368
18369 static __inline__ void
18370 jiffies_to_timespec(unsigned long jiffies,

```

```

18371         struct timespec *value)
18372 {
18373     value->tv_nsec = (jiffies % HZ) * (1000000000L / HZ);
18374     value->tv_sec = jiffies / HZ;
18375 }
18376
18377 struct timeval {
18378     time_t         tv_sec;           /* seconds */
18379     suseconds_t    tv_usec;         /* microseconds */
18380 };
18381
18382 struct timezone {
18383     int         tz_minuteswest; /* minutes west of Greenwich */
18384     int         tz_dsttime;     /* type of dst correction */
18385 };
18386
18387 #define NFDBITS          __NFDBITS
18388
18389 #ifdef __KERNEL__
18390 extern void do_gettimeofday(struct timeval *tv);
18391 extern void do_settimeofday(struct timeval *tv);
18392 extern void get_fast_time(struct timeval *tv);
18393 extern void (*do_get_fast_time)(struct timeval *);
18394 #endif
18395
18396 #define FD_SETSIZE          __FD_SETSIZE
18397 #define FD_SET(fd,fdsetp)  __FD_SET(fd,fdsetp)
18398 #define FD_CLR(fd,fdsetp)  __FD_CLR(fd,fdsetp)
18399 #define FD_ISSET(fd,fdsetp) __FD_ISSET(fd,fdsetp)
18400 #define FD_ZERO(fdsetp)    __FD_ZERO(fdsetp)
18401
18402 /* Names of the interval timers, and structure defining a
18403  * timer setting. */
18404 #define ITIMER_REAL        0
18405 #define ITIMER_VIRTUAL    1
18406 #define ITIMER_PROF       2
18407
18408 struct itimerspec {
18409     struct timespec it_interval; /* timer period */
18410     struct timespec it_value;    /* timer expiration */
18411 };
18412
18413 struct itimerval {
18414     struct timeval it_interval; /* timer interval */
18415     struct timeval it_value;    /* current value */
18416 };
18417
18418 #endif

```

include/linux/timer.h

```

18419 #ifndef _LINUX_TIMER_H
18420 #define _LINUX_TIMER_H
18421
18422 /* Old-style timers. Please don't use for any new code.
18423  *
18424  * Numbering of these timers should be consecutive to
18425  * minimize processing delays. [MJ] */
18426
18427 #define BLANK_TIMER        0 /* Console screen-saver */
18428 #define BEEP_TIMER        1 /* Console beep */
18429 #define RS_TIMER          2 /* RS-232 ports */
18430 #define SWAP_TIMER        3 /* Background pageout */
18431 #define BACKGR_TIMER      4 /* io_request background I/O */
18432 #define HD_TIMER          5 /* Old IDE driver */
18433 #define FLOPPY_TIMER      6 /* Floppy */
18434 #define QIC02_TAPE_TIMER  7 /* QIC 02 tape */
18435 #define MCD_TIMER         8 /* Mitsumi CDROM */
18436 #define GSCD_TIMER        9 /* Goldstar CDROM */
18437 #define CONTROL_TIMER     10 /* Control serial */
18438 #define DIGI_TIMER        11 /* Digi serial */
18439 #define GDTH_TIMER        12 /* Ugh - gdth scsi driver */
18440
18441 #define COPRO_TIMER       31 /* 387 timeout for buggy
18442                               hardware (boot only) */
18443
18444 struct timer_struct {
18445     unsigned long expires;
18446     void (*fn)(void);
18447 };
18448
18449 extern unsigned long timer_active;
18450 extern struct timer_struct timer_table[32];
18451
18452 /* This is completely separate from the above, and is the
18453  * "new and improved" way of handling timers more
18454  * dynamically. Hopefully efficient and general enough
18455  * for most things.
18456  *
18457  * The "hardcoded" timers above are still useful for
18458  * well-defined problems, but the timer_list is probably
18459  * better when you need multiple outstanding timers or
18460  * similar.
18461  *
18462  * The "data" field is in case you want to use the same
18463  * timeout function for several timeouts. You can use
18464  * this to distinguish between the different invocations.

```

```

18465 */
18466 struct timer_list {
18467     struct timer_list *next; /* MUST be first element */
18468     struct timer_list *prev;
18469     unsigned long expires;
18470     unsigned long data;
18471     void (*function)(unsigned long);
18472 };
18473
18474 extern void add_timer(struct timer_list * timer);
18475 extern int del_timer(struct timer_list * timer);
18476
18477 /* mod_timer is a more efficient way to update the expire
18478 * field of an active timer (if the timer is inactive it
18479 * will be activated)
18480 * mod_timer(a,b) is equivalent to
18481 * del_timer(a); a->expires = b; add_timer(a) */
18482 void mod_timer(struct timer_list *timer,
18483                unsigned long expires);
18484
18485 extern void it_real_fn(unsigned long);
18486
18487 extern inline void init_timer(struct timer_list * timer)
18488 {
18489     timer->next = NULL;
18490     timer->prev = NULL;
18491 }
18492
18493 extern inline int
18494 timer_pending(struct timer_list * timer)
18495 {
18496     return timer->prev != NULL;
18497 }
18498
18499 /* These inlines deal with timer wrapping correctly. You
18500 * are strongly encouraged to use them
18501 *     1. Because people otherwise forget
18502 *     2. Because if the timer wrap changes in future
18503 *     you wont have to alter your driver code.
18504 *
18505 * Do this with "<0" and ">=0" to only test the sign of
18506 * the result. A good compiler would generate better code
18507 * (and a really good compiler wouldn't care). Gcc is
18508 * currently neither. */
18509 #define time_after(a,b)      ((long)(b) - (long)(a) < 0)
18510 #define time_before(a,b)    time_after(b,a)
18511
18512 #define time_after_eq(a,b)  ((long)(a) - (long)(b) >= 0)

```

```

18513 #define time_before_eq(a,b) time_after_eq(b,a)
18514
18515 #endif

```

include/linux/times.h

```

18516 #ifndef _LINUX_TIMES_H
18517 #define _LINUX_TIMES_H
18518
18519 struct tms {
18520     clock_t tms_utime;
18521     clock_t tms_stime;
18522     clock_t tms_cutime;
18523     clock_t tms_cstime;
18524 };
18525
18526 #endif

```

include/linux/tqueue.h

```

18527 /*
18528 * tqueue.h -- task queue handling for Linux.
18529 *
18530 * Mostly based on a proposed bottom half replacement
18531 * code written by Kai Petzke,
18532 * wpp@marie.physik.tu-berlin.de.
18533 *
18534 * Modified for use in the Linux kernel by Theodore Ts'o,
18535 * tytso@mit.edu. Any bugs are my fault, not Kai's.
18536 *
18537 * The original comment follows below. */
18538
18539 #ifndef _LINUX_TQUEUE_H
18540 #define _LINUX_TQUEUE_H
18541
18542 #include <asm/bitops.h>
18543 #include <asm/system.h>
18544 #include <asm/spinlock.h>
18545
18546 /* New proposed "bottom half" handlers:
18547 * (C) 1994 Kai Petzke, wpp@marie.physik.tu-berlin.de
18548 *
18549 * Advantages:
18550 * - Bottom halves are implemented as a linked list. You
18551 *   can have as many of them, as you want.
18552 * - No more scanning of a bit field is required upon
18553 *   call of a bottom half.
18554 * - Support for chained bottom half lists. The

```



```

18555 * run_task_queue() function can be used as a bottom half
18556 * handler. This is for example useful for bottom halves,
18557 * which want to be delayed until the next clock tick.
18558 *
18559 * Problems:
18560 * - The queue_task_irq() inline function is only atomic
18561 * with respect to itself. Problems can occur, when
18562 * queue_task_irq() is called from a normal system
18563 * call, and an interrupt comes in. No problems occur,
18564 * when queue_task_irq() is called from an interrupt or
18565 * bottom half, and interrupted, as run_task_queue()
18566 * will not be executed/continued before the last
18567 * interrupt returns. If in doubt, use queue_task(),
18568 * not queue_task_irq().
18569 * - Bottom halves are called in the reverse order that
18570 * they were linked into the list. */
18571
18572 struct tq_struct {
18573     struct tq_struct *next; /* linked list of active BHs*/
18574     unsigned long sync; /* must be initialized to 0 */
18575     void (*routine)(void *); /* function to call */
18576     void *data; /* arg to function */
18577 };
18578
18579 typedef struct tq_struct * task_queue;
18580
18581 #define DECLARF_TASK_QUEUE(q) task_queue q = NULL
18582
18583 extern task_queue tq_timer, tq_immediate, tq_scheduler,
18584 tq_disk;
18585
18586 /* To implement your own list of active bottom halves, use
18587 * the following two definitions:
18588 *
18589 * struct tq_struct *my_bh = NULL;
18590 * struct tq_struct run_my_bh = {
18591 *     0, 0, (void (*)(void *)) run_task_queue, &my_bh
18592 * };
18593 *
18594 * To activate a bottom half on your list, use:
18595 *
18596 *     queue_task(tq_pointer, &my_bh);
18597 *
18598 * To run the bottom halves on your list put them on the
18599 * immediate list by:
18600 *
18601 *     queue_task(&run_my_bh, &tq_immediate);
18602 *

```

```

18603 * This allows you to do deferred procession. For
18604 * example, you could have a bottom half list tq_timer,
18605 * which is marked active by the timer interrupt. */
18606
18607 extern spinlock_t tqueue_lock;
18608
18609 /* queue_task */
18610 extern __inline__ void queue_task(
18611     struct tq_struct *bh_pointer, task_queue *bh_list)
18612 {
18613     if (!test_and_set_bit(0,&bh_pointer->sync)) {
18614         unsigned long flags;
18615         spin_lock_irqsave(&tqueue_lock, flags);
18616         bh_pointer->next = *bh_list;
18617         *bh_list = bh_pointer;
18618         spin_unlock_irqrestore(&tqueue_lock, flags);
18619     }
18620 }
18621
18622 /* Call all "bottom halves" on a given list. */
18623 extern __inline__ void run_task_queue(task_queue *list)
18624 {
18625     if (*list) {
18626         unsigned long flags;
18627         struct tq_struct *p;
18628
18629         spin_lock_irqsave(&tqueue_lock, flags);
18630         p = *list;
18631         *list = NULL;
18632         spin_unlock_irqrestore(&tqueue_lock, flags);
18633
18634         while (p) {
18635             void *arg;
18636             void (*f)(void *);
18637             struct tq_struct *save_p;
18638             arg = p -> data;
18639             f = p -> routine;
18640             save_p = p;
18641             p = p -> next;
18642             mb();
18643             save_p -> sync = 0;
18644             (*f)(arg);
18645         }
18646     }
18647 }
18648
18649 #endif /* LINUX TOUFU H */

```

include/linux/wait.h

```

18650 #ifndef _LINUX_WAIT_H
18651 #define _LINUX_WAIT_H
18652
18653 #define WNOHANG          0x00000001
18654 #define WUNTRACED       0x00000002
18655
18656 #define __WCLONE         0x80000000
18657
18658 #ifdef __KERNEL__
18659
18660 #include <asm/page.h>
18661
18662 struct wait_queue {
18663     struct task_struct * task;
18664     struct wait_queue * next;
18665 };
18666
18667 #define WAIT_QUEUE_HEAD(x) ((struct wait_queue *)((x)-1))
18668
18669 static inline void init_waitqueue(struct wait_queue **q)
18670 {
18671     *q = WAIT_QUEUE_HEAD(q);
18672 }
18673
18674 static inline int waitqueue_active(struct wait_queue **q)
18675 {
18676     struct wait_queue *head = *q;
18677     return head && head != WAIT_QUEUE_HEAD(q);
18678 }
18679
18680 #endif /* __KERNEL__ */
18681
18682 #endif

```

init/main.c

```

18683 /*
18684  * linux/init/main.c
18685  *
18686  * Copyright (C) 1991, 1992 Linus Torvalds
18687  *
18688  * GK 2/5/95 - Changed to support mounting root fs via
18689  * NFS
18690  * Added initrd & change_root: Werner Almesberger & Hans
18691  * Lermen, Feb '96
18692  * Moan early if gcc is old, avoiding bogus kernels -
18693  * Paul Gortmaker, May '96

```

```

18694  * Simplified starting of init: Michael A. Griffith
18695  * <grif@acm.org> */
18696
18697 #define __KERNEL_SYSCALLS__
18698
18699 #include <linux/config.h>
18700 #include <linux/proc_fs.h>
18701 #include <linux/unistd.h>
18702 #include <linux/ctype.h>
18703 #include <linux/delay.h>
18704 #include <linux/utsname.h>
18705 #include <linux/ioport.h>
18706 #include <linux/init.h>
18707 #include <linux/smp_lock.h>
18708 #include <linux/blk.h>
18709 #include <linux/hdreg.h>
18710
18711 #include <asm/io.h>
18712 #include <asm/bugs.h>
18713
18714 #ifdef CONFIG_PCI
18715 #include <linux/pci.h>
18716 #endif
18717
18718 #ifdef CONFIG_DIO
18719 #include <linux/dio.h>
18720 #endif
18721
18722 #ifdef CONFIG_ZORRO
18723 #include <linux/zorro.h>
18724 #endif
18725
18726 #ifdef CONFIG_MTRR
18727 #include <asm/mtrr.h>
18728 #endif
18729
18730 #ifdef CONFIG_APM
18731 #include <linux/apm_bios.h>
18732 #endif
18733
18734 /* Versions of gcc older than that listed below may
18735  * actually compile and link okay, but the end product
18736  * can have subtle run time bugs. To avoid associated
18737  * bogus bug reports, we flatly refuse to compile with a
18738  * gcc that is known to be too old from the very
18739  * beginning. */
18740 #if __GNUC__ < 2 || ( __GNUC__ == 2 && __GNUC_MINOR__ < 6)
18741 #error sorry, your GCC is too old.

```

```

18742 It builds incorrect kernels.
18743 #endif
18744
18745 extern char _stext, _etext;
18746 extern char *linux_banner;
18747
18748 extern int console_ldglevel;
18749
18750 static int init(void *);
18751 extern int bdflush(void *);
18752 extern int kswapd(void *);
18753 extern int kpiod(void *);
18754 extern void kswapd_setup(void);
18755
18756 extern void init_IRQ(void);
18757 extern void init_modules(void);
18758 extern long console_init(long, long);
18759 extern void sock_init(void);
18760 extern void uidcache_init(void);
18761 extern void mca_init(void);
18762 extern void sbus_init(void);
18763 extern void powermac_init(void);
18764 extern void sysctl_init(void);
18765 extern void filescache_init(void);
18766 extern void signals_init(void);
18767
18768 extern void device_setup(void);
18769 extern void binfmt_setup(void);
18770 extern void free_initmem(void);
18771 extern void filesystem_setup(void);
18772
18773 #ifdef CONFIG_ARCH_ACORN
18774 extern void ecard_init(void);
18775 #endif
18776
18777 extern void smp_setup(char *str, int *ints);
18778 #ifdef __i386__
18779 extern void ioapic_pirq_setup(char *str, int *ints);
18780 extern void ioapic_setup(char *str, int *ints);
18781 #endif
18782 extern void no_scroll(char *str, int *ints);
18783 extern void kbd_reset_setup(char *str, int *ints);
18784 extern void panic_setup(char *str, int *ints);
18785 extern void bmouse_setup(char *str, int *ints);
18786 extern void msmouse_setup(char *str, int *ints);
18787 extern void console_setup(char *str, int *ints);
18788 #ifdef CONFIG_PRINTER
18789 extern void lp_setup(char *str, int *ints);

```

```

18790 #endif
18791 #ifdef CONFIG_JOY_AMIGA
18792 extern void js_am_setup(char *str, int *ints);
18793 #endif
18794 #ifdef CONFIG_JOY_ANALOG
18795 extern void js_an_setup(char *str, int *ints);
18796 #endif
18797 #ifdef CONFIG_JOY_ASSASIN
18798 extern void js_as_setup(char *str, int *ints);
18799 #endif
18800 #ifdef CONFIG_JOY_CONSOLE
18801 extern void js_console_setup(char *str, int *ints);
18802 #endif
18803 #ifdef CONFIG_JOY_DB9
18804 extern void js_db9_setup(char *str, int *ints);
18805 #endif
18806 #ifdef CONFIG_JOY_TURBOGRAFX
18807 extern void js_tg_setup(char *str, int *ints);
18808 #endif
18809 #ifdef CONFIG_JOY_LIGHTNING
18810 extern void js_l4_setup(char *str, int *ints);
18811 #endif
18812 extern void eth_setup(char *str, int *ints);
18813 #ifdef CONFIG_ARCNET_COM20020
18814 extern void com20020_setup(char *str, int *ints);
18815 #endif
18816 #ifdef CONFIG_ARCNET_RIM_I
18817 extern void arcrimi_setup(char *str, int *ints);
18818 #endif
18819 #ifdef CONFIG_ARCNET_COM90xx10
18820 extern void com90io_setup(char *str, int *ints);
18821 #endif
18822 #ifdef CONFIG_ARCNET_COM90xx
18823 extern void com90xx_setup(char *str, int *ints);
18824 #endif
18825 #ifdef CONFIG_DECNET
18826 extern void decnet_setup(char *str, int *ints);
18827 #endif
18828 #ifdef CONFIG_BLK_DEV_XD
18829 extern void xd_setup(char *str, int *ints);
18830 extern void xd_manual_geo_init(char *str, int *ints);
18831 #endif
18832 #ifdef CONFIG_BLK_DEV_IDE
18833 extern void ide_setup(char *);
18834 #endif
18835 #ifdef CONFIG_PARIDE_PD
18836 extern void pd_setup(char *str, int *ints);
18837 #endif

```

```

18838 #ifdef CONFIG_PARIDE_PF
18839 extern void pf_setup(char *str, int *ints);
18840 #endif
18841 #ifdef CONFIG_PARIDE_PT
18842 extern void pt_setup(char *str, int *ints);
18843 #endif
18844 #ifdef CONFIG_PARIDE_PG
18845 extern void pg_setup(char *str, int *ints);
18846 #endif
18847 #ifdef CONFIG_PARIDE_PCD
18848 extern void pcd_setup(char *str, int *ints);
18849 #endif
18850 extern void floppy_setup(char *str, int *ints);
18851 extern void st_setup(char *str, int *ints);
18852 extern void st0x_setup(char *str, int *ints);
18853 extern void advansys_setup(char *str, int *ints);
18854 extern void tmc8xx_setup(char *str, int *ints);
18855 extern void tl28_setup(char *str, int *ints);
18856 extern void pas16_setup(char *str, int *ints);
18857 extern void generic_NCR5380_setup(char *str, int *intr);
18858 extern void generic_NCR53C400_setup(char *str, int *intr);
18859 extern void generic_NCR53C400A_setup(char *str,
18860                                     int *intr);
18861 extern void generic_DTC3181E_setup(char *str, int *intr);
18862 extern void aha152x_setup(char *str, int *ints);
18863 extern void aha1542_setup(char *str, int *ints);
18864 extern void gdth_setup(char *str, int *ints);
18865 extern void aic7xxx_setup(char *str, int *ints);
18866 extern void AM53C974_setup(char *str, int *ints);
18867 extern void BusLogic_Setup(char *str, int *ints);
18868 extern void ncr53c8xx_setup(char *str, int *ints);
18869 extern void eata2x_setup(char *str, int *ints);
18870 extern void ul4_34f_setup(char *str, int *ints);
18871 extern void fdomain_setup(char *str, int *ints);
18872 extern void ibmmca_scsi_setup(char *str, int *ints);
18873 extern void in2000_setup(char *str, int *ints);
18874 extern void NCR53c406a_setup(char *str, int *ints);
18875 extern void sym53c416_setup(char *str, int *ints);
18876 extern void wd7000_setup(char *str, int *ints);
18877 extern void dc390_setup(char *str, int *ints);
18878 extern void scsi_luns_setup(char *str, int *ints);
18879 extern void scsi_logging_setup(char *str, int *ints);
18880 extern void sound_setup(char *str, int *ints);
18881 extern void reboot_setup(char *str, int *ints);
18882 extern void video_setup(char *str, int *ints);
18883 #ifdef CONFIG_CDU31A
18884 extern void cdu31a_setup(char *str, int *ints);
18885 #endif CONFIG_CDU31A
18886 #ifdef CONFIG_BLK_DEV_PS2
18887 extern void ed_setup(char *str, int *ints);
18888 extern void tp720_setup(char *str, int *ints);
18889 #endif CONFIG_BLK_DEV_PS2
18890 #ifdef CONFIG_MCD
18891 extern void mcd_setup(char *str, int *ints);
18892 #endif CONFIG_MCD
18893 #ifdef CONFIG_MCDX
18894 extern void mcdx_setup(char *str, int *ints);
18895 #endif CONFIG_MCDX
18896 #ifdef CONFIG_SBPCD
18897 extern void sbpcd_setup(char *str, int *ints);
18898 #endif CONFIG_SBPCD
18899 #ifdef CONFIG_AZTCD
18900 extern void aztcd_setup(char *str, int *ints);
18901 #endif CONFIG_AZTCD
18902 #ifdef CONFIG_CDU535
18903 extern void sonycd535_setup(char *str, int *ints);
18904 #endif CONFIG_CDU535
18905 #ifdef CONFIG_GSCD
18906 extern void gscd_setup(char *str, int *ints);
18907 #endif CONFIG_GSCD
18908 #ifdef CONFIG_CM206
18909 extern void cm206_setup(char *str, int *ints);
18910 #endif CONFIG_CM206
18911 #ifdef CONFIG_OPTCD
18912 extern void optcd_setup(char *str, int *ints);
18913 #endif CONFIG_OPTCD
18914 #ifdef CONFIG_SJCD
18915 extern void sjcd_setup(char *str, int *ints);
18916 #endif CONFIG_SJCD
18917 #ifdef CONFIG_ISP16_CDI
18918 extern void ispl6_setup(char *str, int *ints);
18919 #endif CONFIG_ISP16_CDI
18920 #ifdef CONFIG_BLK_DEV_RAM
18921 static void ramdisk_start_setup(char *str, int *ints);
18922 static void load_ramdisk(char *str, int *ints);
18923 static void prompt_ramdisk(char *str, int *ints);
18924 static void ramdisk_size(char *str, int *ints);
18925 #ifdef CONFIG_BLK_DEV_INITRD
18926 static void no_initrd(char *s, int *ints);
18927 #endif
18928 #endif CONFIG_BLK_DEV_RAM
18929 #ifdef CONFIG_ISDN_DRV_ICN
18930 extern void icn_setup(char *str, int *ints);
18931 #endif
18932 #ifdef CONFIG_ISDN_DRV_HISAX
18933 extern void HiSax_setup(char *str, int *ints);

```

```

18934 #endif
18935 #ifdef CONFIG_DIGIEPCA
18936 extern void epca_setup(char *str, int *ints);
18937 #endif
18938 #ifdef CONFIG_ISDN_DRV_PCBIT
18939 extern void pcbit_setup(char *str, int *ints);
18940 #endif
18941
18942 #ifdef CONFIG_ATARIMOUSE
18943 extern void atari_mouse_setup (char *str, int *ints);
18944 #endif
18945 #ifdef CONFIG_DMASOUND
18946 extern void dmasound_setup (char *str, int *ints);
18947 #endif
18948 #ifdef CONFIG_ATARI_SCSI
18949 extern void atari_scsi_setup (char *str, int *ints);
18950 #endif
18951 extern void stram_swap_setup (char *str, int *ints);
18952 extern void wd33c93_setup (char *str, int *ints);
18953 extern void gvpl1_setup (char *str, int *ints);
18954 extern void ncr53c7xx_setup (char *str, int *ints);
18955 #ifdef CONFIG_MAC_SCSI
18956 extern void mac_scsi_setup (char *str, int *ints);
18957 #endif
18958
18959 #ifdef CONFIG_CYCLADES
18960 extern void cy_setup(char *str, int *ints);
18961 #endif
18962 #ifdef CONFIG_DIGI
18963 extern void pcxx_setup(char *str, int *ints);
18964 #endif
18965 #ifdef CONFIG_RISCOM8
18966 extern void riscom8_setup(char *str, int *ints);
18967 #endif
18968 #ifdef CONFIG_SPECIALIX
18969 extern void specialix_setup(char *str, int *ints);
18970 #endif
18971 #ifdef CONFIG_DMASCC
18972 extern void dmascc_setup(char *str, int *ints);
18973 #endif
18974 #ifdef CONFIG_BAYCOM_PAR
18975 extern void baycom_par_setup(char *str, int *ints);
18976 #endif
18977 #ifdef CONFIG_BAYCOM_SER_FDX
18978 extern void baycom_ser_fdx_setup(char *str, int *ints);
18979 #endif
18980 #ifdef CONFIG_BAYCOM_SER_HDX
18981 extern void baycom_ser_hdx_setup(char *str, int *ints);

```

```

18982 #endif
18983 #ifdef CONFIG_SOUNDMODEM
18984 extern void sm_setup(char *str, int *ints);
18985 #endif
18986 #ifdef CONFIG_ADBMOUSE
18987 extern void adb_mouse_setup(char *str, int *ints);
18988 #endif
18989 #ifdef CONFIG_WDT
18990 extern void wdt_setup(char *str, int *ints);
18991 #endif
18992 #ifdef CONFIG_PARPORT
18993 extern void parport_setup(char *str, int *ints);
18994 #endif
18995 #ifdef CONFIG_PLIP
18996 extern void plip_setup(char *str, int *ints);
18997 #endif
18998 #ifdef CONFIG_HFMODEM
18999 extern void hfmodem_setup(char *str, int *ints);
19000 #endif
19001 #ifdef CONFIG_IP_PNP
19002 extern void ip_auto_config_setup(char *str, int *ints);
19003 #endif
19004 #ifdef CONFIG_ROOT_NFS
19005 extern void nfs_root_setup(char *str, int *ints);
19006 #endif
19007 #ifdef CONFIG_FTAPE
19008 extern void ftape_setup(char *str, int *ints);
19009 #endif
19010 #ifdef CONFIG_MDA_CONSOLE
19011 extern void mdacon_setup(char *str, int *ints);
19012 #endif
19013 #ifdef CONFIG_LTPC
19014 extern void ltpc_setup(char *str, int *ints);
19015 #endif
19016
19017 #if defined(CONFIG_SYSVIPC)
19018 extern void ipc_init(void);
19019 #endif
19020 #if defined(CONFIG_QUOTA)
19021 extern void dquot_init_hash(void);
19022 #endif
19023
19024 #ifdef CONFIG_MD_BOOT
19025 extern void md_setup(char *str, int *ints) __init;
19026 #endif
19027
19028 /* Boot command-line arguments */
19029 #define MAX_INIT_ARGS 8

```

```

19030 #define MAX_INIT_ENVS 8
19031
19032 extern void time_init(void);
19033
19034 static unsigned long memory_start = 0;
19035 static unsigned long memory_end = 0;
19036
19037 int rows, cols;
19038
19039 #ifdef CONFIG_BLK_DEV_RAM
19040 /* 1 = load ramdisk, 0 = don't load */
19041 extern int rd_doload;
19042 /* 1 = prompt for ramdisk, 0 = don't prompt */
19043 extern int rd_prompt;
19044 /* Size of the ramdisk(s) */
19045 extern int rd_size;
19046 /* starting block # of image */
19047 extern int rd_image_start;
19048
19049 #ifdef CONFIG_BLK_DEV_INITRD
19050 kdev_t real_root_dev;
19051 #endif
19052 #endif
19053
19054 int root_mountflags = MS_RDONLY;
19055 char *execute_command = NULL;
19056
19057 static char * argv_init[MAX_INIT_ARGS+2] =
19058 { "init", NULL, };
19059 static char * envp_init[MAX_INIT_ENVS+2] =
19060 { "HOME=/", "TERM=linux", NULL, };
19061
19062 char *get_options(char *str, int *ints)
19063 {
19064     char *cur = str;
19065     int i=1;
19066
19067     while (cur && (*cur--'-' || isdigit(*cur)) && i <= 10){
19068         ints[i++] = simple_strtol(cur,NULL,0);
19069         if ((cur = strchr(cur, '.')) != NULL)
19070             cur++;
19071     }
19072     ints[0] = i-1;
19073     return(cur);
19074 }
19075
19076 static void __init profile_setup(char *str, int *ints)
19077 {

```

```

19078     if (ints[0] > 0)
19079         prof_shift = (unsigned long) ints[1];
19080     else
19081         prof_shift = 2;
19082 }
19083
19084
19085 static struct dev_name_struct {
19086     const char *name;
19087     const int num;
19088 } root_dev_names[] __initdata = {
19089 #ifdef CONFIG_ROOT_NFS
19090     { "nfs", 0x00ff },
19091 #endif
19092 #ifdef CONFIG_BLK_DEV_IDE
19093     { "hda", 0x0300 },
19094     { "hdb", 0x0340 },
19095     { "hdc", 0x1600 },
19096     { "hdd", 0x1640 },
19097     { "hde", 0x2100 },
19098     { "hdf", 0x2140 },
19099     { "hdg", 0x2200 },
19100     { "hdh", 0x2240 },
19101     { "hdi", 0x3800 },
19102     { "hdj", 0x3840 },
19103     { "hdk", 0x3900 },
19104     { "hdl", 0x3940 },
19105 #endif
19106 #ifdef CONFIG_BLK_DEV_SD
19107     { "sda", 0x0800 },
19108     { "sdb", 0x0810 },
19109     { "sdc", 0x0820 },
19110     { "sdd", 0x0830 },
19111     { "sde", 0x0840 },
19112     { "sdf", 0x0850 },
19113     { "sdg", 0x0860 },
19114     { "sdh", 0x0870 },
19115     { "sdi", 0x0880 },
19116     { "sdj", 0x0890 },
19117     { "sdk", 0x08a0 },
19118     { "sdl", 0x08b0 },
19119     { "sdm", 0x08c0 },
19120     { "sdn", 0x08d0 },
19121     { "sdo", 0x08e0 },
19122     { "sdp", 0x08f0 },
19123 #endif
19124 #ifdef CONFIG_ATARI_ACPI
19125     { "ada", 0x1c00 },

```

```

19126 { "adb",    0x1c10 },
19127 { "adc",    0x1c20 },
19128 { "add",    0x1c30 },
19129 { "ade",    0x1c40 },
19130 #endif
19131 #ifdef CONFIG_BLK_DEV_FD
19132 { "fd",      0x0200 },
19133 #endif
19134 #ifdef CONFIG_MD_BOOT
19135 { "md",      0x0900 },
19136 #endif
19137 #ifdef CONFIG_BLK_DEV_XD
19138 { "xda",     0x0d00 },
19139 { "xdb",     0x0d40 },
19140 #endif
19141 #ifdef CONFIG_BLK_DEV_RAM
19142 { "ram",     0x0100 },
19143 #endif
19144 #ifdef CONFIG_BLK_DEV_SR
19145 { "scd",     0x0b00 },
19146 #endif
19147 #ifdef CONFIG_MCD
19148 { "mcd",     0x1700 },
19149 #endif
19150 #ifdef CONFIG_CDU535
19151 { "cdu535", 0x1800 },
19152 { "sonycd", 0x1800 },
19153 #endif
19154 #ifdef CONFIG_AZTCD
19155 { "aztcd",  0x1d00 },
19156 #endif
19157 #ifdef CONFIG_CM206
19158 { "cm206cd", 0x2000 },
19159 #endif
19160 #ifdef CONFIG_GSCD
19161 { "gscd",   0x1000 },
19162 #endif
19163 #ifdef CONFIG_SBP_CD
19164 { "sbpcd",  0x1900 },
19165 #endif
19166 #ifdef CONFIG_BLK_DEV_PS2
19167 { "eda",    0x2400 },
19168 { "edb",    0x2440 },
19169 #endif
19170 #ifdef CONFIG_PARIDE_PD
19171 { "pda",    0x2d00 },
19172 { "pdb",    0x2d10 },
19173 { "pdc",    0x2d20 },

```

```

19174 { "pdd",    0x2d30 },
19175 #endif
19176 #ifdef CONFIG_PARIDE_PCD
19177 { "pcd",    0x2e00 },
19178 #endif
19179 #ifdef CONFIG_PARIDE_PF
19180 { "pf",     0x2f00 },
19181 #endif
19182 #if CONFIG_APBLOCK
19183 { "apblock", APBLOCK_MAJOR << 8 },
19184 #endif
19185 #if CONFIG_DDV
19186 { "ddv",    DDV_MAJOR << 8 },
19187 #endif
19188 { NULL, 0 }
19189 };
19190
19191 kdev_t __init name_to_kdev_t(char *line)
19192 {
19193     int base = 0;
19194     if (strncmp(line, "/dev/", 5) == 0) {
19195         struct dev_name_struct *dev = root_dev_names;
19196         line += 5;
19197         do {
19198             int len = strlen(dev->name);
19199             if (strncmp(line, dev->name, len) == 0) {
19200                 line += len;
19201                 base = dev->num;
19202                 break;
19203             }
19204             dev++;
19205         } while (dev->name);
19206     }
19207     return to_kdev_t(base + simple_strtoul(line, NULL,
19208                                             base ? 10 : 16));
19209 }
19210
19211 static void __init root_dev_setup(char *line, int *num)
19212 {
19213     ROOT_DEV = name_to_kdev_t(line);
19214 }
19215
19216 /* List of kernel command line parameters. The first
19217 * table lists parameters which are subject to values
19218 * parsing (leading numbers are converted to an array of
19219 * ints and chopped off the string), the second table
19220 * contains the few exceptions which obey their own
19221 * syntax rules. */

```

```

19222
19223 struct kernel_param {
19224     const char *str;
19225     void (*setup_func)(char *, int *);
19226 };
19227
19228 static struct kernel_param cooked_params[] __initdata = {
19229     /* FIXME: make PNP just become reserve_setup */
19230     #ifndef CONFIG_KERNEL_PNP_RESOURCE
19231     { "reserve=", reserve_setup },
19232     #else
19233     { "reserve=", pnp_reserve_setup },
19234     #endif
19235     { "profile=", profile_setup },
19236     #ifdef __SMP__
19237     { "nosmp", smp_setup },
19238     { "maxcpus=", smp_setup },
19239     #ifdef CONFIG_X86_IO_APIC
19240     { "noapic", ioapic_setup },
19241     { "pirq=", ioapic_pirq_setup },
19242     #endif
19243     #endif
19244     #ifdef CONFIG_BLK_DEV_RAM
19245     { "ramdisk_start=", ramdisk_start_setup },
19246     { "load_ramdisk=", load_ramdisk },
19247     { "prompt_ramdisk=", prompt_ramdisk },
19248     { "ramdisk=", ramdisk_size },
19249     { "ramdisk_size=", ramdisk_size },
19250     #ifdef CONFIG_BLK_DEV_INITRD
19251     { "noinitrd", no_initrd },
19252     #endif
19253     #endif
19254     #ifdef CONFIG_FB
19255     { "video=", video_setup },
19256     #endif
19257     { "panic=", panic_setup },
19258     { "console=", console_setup },
19259     #ifdef CONFIG_VGA_CONSOLE
19260     { "no-scroll", no_scroll },
19261     #endif
19262     #ifdef CONFIG_MDA_CONSOLE
19263     { "mdacon=", mdacon_setup },
19264     #endif
19265     #ifdef CONFIG_VT
19266     { "kbd-reset", kbd_reset_setup },
19267     #endif
19268     #ifdef CONFIG_BUGi386
19269     { "no-hlt", no_halt },

```

```

19270     { "no387", no_387 },
19271     { "reboot=", reboot_setup },
19272     #endif
19273     #ifdef CONFIG_INET
19274     { "ether=", eth_setup },
19275     #endif
19276     #ifdef CONFIG_ARCNET_COM20020
19277     { "com20020=", com20020_setup },
19278     #endif
19279     #ifdef CONFIG_ARCNET_RIM_I
19280     { "arcrimi=", arcrimi_setup },
19281     #endif
19282     #ifdef CONFIG_ARCNET_COM90xxIO
19283     { "com90io=", com90io_setup },
19284     #endif
19285     #ifdef CONFIG_ARCNET_COM90xx
19286     { "com90xx=", com90xx_setup },
19287     #endif
19288     #ifdef CONFIG_DECNET
19289     { "decnet=", decnet_setup },
19290     #endif
19291     #ifdef CONFIG_PRINTER
19292     { "lp=", lp_setup },
19293     #endif
19294     #ifdef CONFIG_JOY_AMIGA
19295     { "js_am=", js_am_setup },
19296     #endif
19297     #ifdef CONFIG_JOY_ANALOG
19298     { "js_an=", js_an_setup },
19299     #endif
19300     #ifdef CONFIG_JOY_ASSASIN
19301     { "js_as=", js_as_setup },
19302     #endif
19303     #ifdef CONFIG_JOY_CONSOLE
19304     { "js_console=", js_console_setup },
19305     { "js_console2=", js_console_setup },
19306     { "js_console3=", js_console_setup },
19307     #endif
19308     #ifdef CONFIG_JOY_DB9
19309     { "js_db9=", js_db9_setup },
19310     { "js_db9_2=", js_db9_setup },
19311     { "js_db9_3=", js_db9_setup },
19312     #endif
19313     #ifdef CONFIG_JOY_TURBOGRAFX
19314     { "js_tg=", js_tg_setup },
19315     { "js_tg_2=", js_tg_setup },
19316     { "js_tg_3=", js_tg_setup },
19317     #endif

```



```

19318 #ifdef CONFIG_SCSI
19319     { "max_scsi_luns=", scsi_luns_setup },
19320     { "scsi_logging=", scsi_logging_setup },
19321 #endif
19322 #ifdef CONFIG_JOY_LIGHTNING
19323     { "js_l4=", js_l4_setup },
19324 #endif
19325 #ifdef CONFIG_SCSI_ADVANSYS
19326     { "advansys=", advansys_setup },
19327 #endif
19328 #if defined(CONFIG_BLK_DEV_HD)
19329     { "hd=", hd_setup },
19330 #endif
19331 #ifdef CONFIG_CHR_DEV_ST
19332     { "st=", st_setup },
19333 #endif
19334 #ifdef CONFIG_BUSMOUSE
19335     { "bmouse=", bmouse_setup },
19336 #endif
19337 #ifdef CONFIG_MS_BUSMOUSE
19338     { "msmouse=", msmouse_setup },
19339 #endif
19340 #ifdef CONFIG_SCSI_SEAGATE
19341     { "st0x=", st0x_setup },
19342     { "tmc8xx=", tmc8xx_setup },
19343 #endif
19344 #ifdef CONFIG_SCSI_T128
19345     { "t128=", t128_setup },
19346 #endif
19347 #ifdef CONFIG_SCSI_PAS16
19348     { "pas16=", pas16_setup },
19349 #endif
19350 #ifdef CONFIG_SCSI_GENERIC_NCR5380
19351     { "ncr5380=", generic_NCR5380_setup },
19352     { "ncr53c400=", generic_NCR53C400_setup },
19353     { "ncr53c400a=", generic_NCR53C400A_setup },
19354     { "dtc3181e=", generic_DTC3181E_setup },
19355 #endif
19356 #ifdef CONFIG_SCSI_AHA152X
19357     { "aha152x=", aha152x_setup },
19358 #endif
19359 #ifdef CONFIG_SCSI_AHA1542
19360     { "aha1542=", aha1542_setup },
19361 #endif
19362 #ifdef CONFIG_SCSI_GDTH
19363     { "gdth=", gdth_setup },
19364 #endif
19365 #ifdef CONFIG_SCSI_AIC7XXX

```

```

19366     { "aic7xxx=", aic7xxx_setup },
19367 #endif
19368 #ifdef CONFIG_SCSI_BUSLOGIC
19369     { "BusLogic=", BusLogic_Setup },
19370 #endif
19371 #ifdef CONFIG_SCSI_NCR53C8XX
19372     { "ncr53c8xx=", ncr53c8xx_setup },
19373 #endif
19374 #ifdef CONFIG_SCSI_EATA
19375     { "eata=", eata2x_setup },
19376 #endif
19377 #ifdef CONFIG_SCSI_U14_34F
19378     { "u14-34f=", u14_34f_setup },
19379 #endif
19380 #ifdef CONFIG_SCSI_AM53C974
19381     { "AM53C974=", AM53C974_setup },
19382 #endif
19383 #ifdef CONFIG_SCSI_NCR53C406A
19384     { "ncr53c406a=", NCR53c406a_setup },
19385 #endif
19386 #ifdef CONFIG_SCSI_SYM53C416
19387     { "sym53c416=", sym53c416_setup },
19388 #endif
19389 #ifdef CONFIG_SCSI_FUTURE_DOMAIN
19390     { "fdomain=", fdomain_setup },
19391 #endif
19392 #ifdef CONFIG_SCSI_IN2000
19393     { "in2000=", in2000_setup },
19394 #endif
19395 #ifdef CONFIG_SCSI_7000FASST
19396     { "wd7000=", wd7000_setup },
19397 #endif
19398 #ifdef CONFIG_SCSI_IBMMCA
19399     { "ibmmcascsi=", ibmmca_scsi_setup },
19400 #endif
19401 #if defined(CONFIG_SCSI_DC390T) &&
19402     !defined(CONFIG_SCSI_DC390T_NOGENSUPP)
19403     { "tmcsim=", dc390_setup },
19404 #endif
19405 #ifdef CONFIG_BLK_DEV_XD
19406     { "xd=", xd_setup },
19407     { "xd_geo=", xd_manual_geo_init },
19408 #endif
19409 #if defined(CONFIG_BLK_DEV_FD) ||
19410     defined(CONFIG_AMIGA_FLOPPY) ||
19411     defined(CONFIG_ATARI_FLOPPY)
19412     { "floppy=", floppy_setup },
19413 #endif

```

```

19414 #ifdef CONFIG_BLK_DEV_PS2
19415 { "eda=", ed_setup },
19416 { "edb=", ed_setup },
19417 { "tp720=", tp720_setup },
19418 #endif
19419 #ifdef CONFIG_CDU31A
19420 { "cdu31a=", cdu31a_setup },
19421 #endif CONFIG_CDU31A
19422 #ifdef CONFIG_MCD
19423 { "mcd=", mcd_setup },
19424 #endif CONFIG_MCD
19425 #ifdef CONFIG_MCDX
19426 { "mcdx=", mcdx_setup },
19427 #endif CONFIG_MCDX
19428 #ifdef CONFIG_SBP_CD
19429 { "sbpcd=", sbpcd_setup },
19430 #endif CONFIG_SBP_CD
19431 #ifdef CONFIG_AZTCD
19432 { "aztcd=", aztcd_setup },
19433 #endif CONFIG_AZTCD
19434 #ifdef CONFIG_CDU535
19435 { "sonycd535=", sonycd535_setup },
19436 #endif CONFIG_CDU535
19437 #ifdef CONFIG_GSCD
19438 { "gscd=", gscd_setup },
19439 #endif CONFIG_GSCD
19440 #ifdef CONFIG_CM206
19441 { "cm206=", cm206_setup },
19442 #endif CONFIG_CM206
19443 #ifdef CONFIG_OPTCD
19444 { "optcd=", optcd_setup },
19445 #endif CONFIG_OPTCD
19446 #ifdef CONFIG_SJCD
19447 { "sjcd=", sjcd_setup },
19448 #endif CONFIG_SJCD
19449 #ifdef CONFIG_ISP16_CDI
19450 { "isp16=", isp16_setup },
19451 #endif CONFIG_ISP16_CDI
19452 #ifdef CONFIG_SOUND_OSS
19453 { "sound=", sound_setup },
19454 #endif
19455 #ifdef CONFIG_ISDN_DRV_ICN
19456 { "icn=", icn_setup },
19457 #endif
19458 #ifdef CONFIG_ISDN_DRV_HISAX
19459 { "hisax=", HiSax_setup },
19460 { "HiSax=", HiSax_setup },
19461 #endif

```

```

19462 #ifdef CONFIG_ISDN_DRV_PCBI1
19463 { "pccbi1=", pccbi1_setup },
19464 #endif
19465 #ifdef CONFIG_ATARIMOUSE
19466 { "atamouse=", atari_mouse_setup },
19467 #endif
19468 #ifdef CONFIG_DMASOUND
19469 { "dmasound=", dmasound_setup },
19470 #endif
19471 #ifdef CONFIG_ATARI_SCSI
19472 { "atascsi=", atari_scsi_setup },
19473 #endif
19474 #ifdef CONFIG_STRAM_SWAP
19475 { "stram_swap=", stram_swap_setup },
19476 #endif
19477 #if defined(CONFIG_A4000T_SCSI) || \
19478     defined(CONFIG_WARPENGINE_SCSI) || \
19479     defined(CONFIG_A4091_SCSI) || \
19480     defined(CONFIG_MVME16x_SCSI) || \
19481     defined(CONFIG_BVME6000_SCSI) || \
19482 { "53c7xx=", ncr53c7xx_setup },
19483 #endif
19484 #if defined(CONFIG_A3000_SCSI) || \
19485     defined(CONFIG_A2091_SCSI) || \
19486     defined(CONFIG_GVP11_SCSI) || \
19487 { "wd33c93=", wd33c93_setup },
19488 #endif
19489 #if defined(CONFIG_GVP11_SCSI)
19490 { "gvp11=", gvp11_setup },
19491 #endif
19492 #ifdef CONFIG_MAC_SCSI
19493 { "mac5380=", mac_scsi_setup },
19494 #endif
19495 #ifdef CONFIG_CYCLADES
19496 { "cyclades=", cy_setup },
19497 #endif
19498 #ifdef CONFIG_DIGI
19499 { "digi=", pcxx_setup },
19500 #endif
19501 #ifdef CONFIG_DIGIEPCA
19502 { "digiepca=", epca_setup },
19503 #endif
19504 #ifdef CONFIG_RISCOM8
19505 { "riscom8=", riscom8_setup },
19506 #endif
19507 #ifdef CONFIG_DMASCC
19508 { "dmascc=", dmascc_setup },
19509 #endif

```

```

19510 #ifdef CONFIG_SPECIALIX
19511 { "specialix=", specialix_setup },
19512 #endif
19513 #ifdef CONFIG_BAYCOM_PAR
19514 { "baycom_par=", baycom_par_setup },
19515 #endif
19516 #ifdef CONFIG_BAYCOM_SER_FDX
19517 { "baycom_ser_fdx=", baycom_ser_fdx_setup },
19518 #endif
19519 #ifdef CONFIG_BAYCOM_SER_HDX
19520 { "baycom_ser_hdx=", baycom_ser_hdx_setup },
19521 #endif
19522 #ifdef CONFIG_SOUNDMODEM
19523 { "soundmodem=", sm_setup },
19524 #endif
19525 #ifdef CONFIG_WDT
19526 { "wdt=", wdt_setup },
19527 #endif
19528 #ifdef CONFIG_PARPORT
19529 { "parport=", parport_setup },
19530 #endif
19531 #ifdef CONFIG_PLIP
19532 { "plip=", plip_setup },
19533 #endif
19534 #ifdef CONFIG_HFMODEM
19535 { "hfmodem=", hfmodem_setup },
19536 #endif
19537 #ifdef CONFIG_FTAPE
19538 { "ftape=", ftape_setup },
19539 #endif
19540 #ifdef CONFIG_MD_BOOT
19541 { "md=", md_setup },
19542 #endif
19543 #ifdef CONFIG_ADBMOUSE
19544 { "adb_buttons=", adb_mouse_setup },
19545 #endif
19546 #ifdef CONFIG_LTPC
19547 { "ltpc=", ltpc_setup },
19548 #endif
19549 { 0, 0 }
19550 );
19551
19552 static struct kernel_param raw_params[] __initdata = {
19553 { "root=", root_dev_setup },
19554 #ifdef CONFIG_ROOT_NFS
19555 { "nfsroot=", nfs_root_setup },
19556 { "nfsaddr=", ip_auto_config_setup },
19557 #endif

```

```

19558 #ifdef CONFIG_IP_PNP
19559 { "ip=", ip_auto_config_setup },
19560 #endif
19561 #ifdef CONFIG_PCI
19562 { "pci=", pci_setup },
19563 #endif
19564 #ifdef CONFIG_PARIDE_PD
19565 { "pd.", pd_setup },
19566 #endif
19567 #ifdef CONFIG_PARIDE_PCD
19568 { "pcd.", pcd_setup },
19569 #endif
19570 #ifdef CONFIG_PARIDE_PF
19571 { "pf.", pf_setup },
19572 #endif
19573 #ifdef CONFIG_PARIDE_PT
19574 { "pt.", pt_setup },
19575 #endif
19576 #ifdef CONFIG_PARIDE_PG
19577 { "pg.", pg_setup },
19578 #endif
19579 #ifdef CONFIG_APM
19580 { "apm=", apm_setup },
19581 #endif
19582 { 0, 0 }
19583 };
19584
19585 #ifdef CONFIG_BLK_DEV_RAM
19586 static void __init ramdisk_start_setup(char *str,
19587                                         int *ints)
19588 {
19589     if (ints[0] > 0 && ints[1] >= 0)
19590         rd_image_start = ints[1];
19591 }
19592
19593 static void __init load_ramdisk(char *str, int *ints)
19594 {
19595     if (ints[0] > 0 && ints[1] >= 0)
19596         rd_doload = ints[1] & 1;
19597 }
19598
19599 static void __init prompt_ramdisk(char *str, int *ints)
19600 {
19601     if (ints[0] > 0 && ints[1] >= 0)
19602         rd_prompt = ints[1] & 1;
19603 }
19604
19605 static void __init ramdisk_size(char *str, int *ints)

```

```

19606 {
19607     if (ints[0] > 0 && ints[1] >= 0)
19608         rd_size = ints[1];
19609 }
19610 #endif
19611
19612 static int __init checksetup(char *line)
19613 {
19614     int i, ints[11];
19615
19616 #ifdef CONFIG_BLK_DEV_IDE
19617     /* ide driver needs the basic string, rather than
19618      * pre-processed values */
19619     if (!strncmp(line, "ide", 3) ||
19620         (!strncmp(line, "hd", 2) && line[2] != '=')) {
19621         ide_setup(line);
19622         return 1;
19623     }
19624 #endif
19625     for (i=0; raw_params[i].str; i++) {
19626         int n = strlen(raw_params[i].str);
19627         if (!strncmp(line, raw_params[i].str, n)) {
19628             raw_params[i].setup_func(line+n, NULL);
19629             return 1;
19630         }
19631     }
19632     for (i=0; cooked_params[i].str; i++) {
19633         int n = strlen(cooked_params[i].str);
19634         if (!strncmp(line, cooked_params[i].str, n)) {
19635             cooked_params[i].setup_func(get_options(line+n,
19636                                                     ints), ints);
19637             return 1;
19638         }
19639     }
19640     return 0;
19641 }
19642
19643 /* this should be approx 2 Bo*Mips to start (note
19644  * initial shift), and will still work even if initially
19645  * too large, it will just take slightly longer */
19646 unsigned long loops_per_sec = (1<<12);
19647
19648 /* This is the number of bits of precision for the
19649  * loops_per_second. Each bit takes on average 1.5/HZ
19650  * seconds. This (like the original) is a little better
19651  * than 1% */
19652 #define LPS_PREC 8
19653

```

```

19654 void __init calibrate_delay(void)
19655 {
19656     unsigned long ticks, loopbit;
19657     int lps_precision = LPS_PREC;
19658
19659     loops_per_sec = (1<<12);
19660
19661     printk("Calibrating delay loop... ");
19662     while (loops_per_sec <<= 1) {
19663         /* wait for "start of" clock tick */
19664         ticks = jiffies;
19665         while (ticks == jiffies)
19666             /* nothing */;
19667         /* Go .. */
19668         ticks = jiffies;
19669         __delay(loops_per_sec);
19670         ticks = jiffies - ticks;
19671         if (ticks)
19672             break;
19673     }
19674
19675     /* Do a binary approximation to get loops_per_second set
19676      * to equal one clock (up to lps_precision bits) */
19677     loops_per_sec >>= 1;
19678     loopbit = loops_per_sec;
19679     while ( lps_precision-- && (loopbit >>= 1) ) {
19680         loops_per_sec |= loopbit;
19681         ticks = jiffies;
19682         while (ticks == jiffies);
19683         ticks = jiffies;
19684         __delay(loops_per_sec);
19685         if (jiffies != ticks) /* longer than 1 tick */
19686             loops_per_sec &= ~loopbit;
19687     }
19688
19689     /* finally, adjust loops per second in terms of seconds
19690      * instead of clocks */
19691     loops_per_sec *= HZ;
19692     /* Round the value and print it */
19693     printk("%lu.%02lu BogoMIPS\n",
19694            (loops_per_sec+2500)/500000,
19695            ((loops_per_sec+2500)/500000 % 100));
19696 }
19697
19698 /* This is a simple kernel command line parsing function:
19699  * it parses the command line, and fills in the
19700  * arguments/environment to init as appropriate. Any
19701  * cmd-line option is taken to be an environment variable

```

```

19702 * if it contains the character '='.
19703 *
19704 * This routine also checks for options meant for the
19705 * kernel. These options are not given to init - they
19706 * are for internal kernel use only. */
19707 static void __init parse_options(char *line)
19708 {
19709     char *next;
19710     int args, envs;
19711
19712     if (!*line)
19713         return;
19714     args = 0;
19715     envs = 1;          /* TERM is set to 'linux' by default */
19716     next = line;
19717     while ((line = next) != NULL) {
19718         if ((next = strchr(line, ' ')) != NULL)
19719             *next++ = 0;
19720         /* check for kernel options first.. */
19721         if (!strcmp(line, "ro")) {
19722             root_mountflags |= MS_RDONLY;
19723             continue;
19724         }
19725         if (!strcmp(line, "rw")) {
19726             root_mountflags &= ~MS_RDONLY;
19727             continue;
19728         }
19729         if (!strcmp(line, "debug")) {
19730             console_loglevel = 10;
19731             continue;
19732         }
19733         if (!strncmp(line, "init=", 5)) {
19734             line += 5;
19735             execute_command = line;
19736             /* In case LILO is going to boot us with default
19737              * command line, it prepends "auto" before the
19738              * whole cmdline which makes the shell think it
19739              * should execute a script with such name. So we
19740              * ignore all arguments entered _before_
19741              * init=... [MJ] */
19742             args = 0;
19743             continue;
19744         }
19745         if (checksetup(line))
19746             continue;
19747
19748         /* Then check if it's an environment variable or an
19749          * option. */

```

```

19750         if (strchr(line, '=') {
19751             if (envs >= MAX_INIT_ENVS)
19752                 break;
19753             envp_init[++envs] = line;
19754         } else {
19755             if (args >= MAX_INIT_ARGS)
19756                 break;
19757             argv_init[++args] = line;
19758         }
19759     }
19760     argv_init[args+1] = NULL;
19761     envp_init[envs+1] = NULL;
19762 }
19763
19764 extern void setup_arch(char **, unsigned long *,
19765                       unsigned long *);
19766
19767 #ifndef __SMP__
19768
19769 /* Uniprocessor idle thread */
19770
19771 int cpu_idle(void *unused)
19772 {
19773     for(;;)
19774         idle();
19775 }
19776
19777 #define smp_init()      do { } while (0)
19778
19779 #else
19780
19781 /* Multiprocessor idle thread is in arch/... */
19782
19783 extern int cpu_idle(void * unused);
19784
19785 /* Called by boot processor to activate the rest. */
19786 static void __init smp_init(void)
19787 {
19788     /* Get other processors into their bootup holding
19789      * patterns. */
19790     smp_boot_cpus();
19791     smp_threads_ready=1;
19792     smp_commence();
19793 }
19794
19795 #endif
19796
19797

```

```

19798 extern void initialize_secondary(void);
19799
19800 /* Activate the first processor. */
19801
19802 ↪ asmlinkage void __init start_kernel(void)
19803 {
19804     char * command_line;
19805
19806 #ifdef __SMP__
19807     static int boot_cpu = 1;
19808     /* "current" has been set up, we need to load it now */
19809     if (!boot_cpu)
19810         initialize_secondary();
19811     boot_cpu = 0;
19812 #endif
19813
19814 /* Interrupts are still disabled. Do necessary setups,
19815  * then enable them */
19816     printk(linux_banner);
19817     setup_arch(&command_line, &memory_start, &memory_end);
19818     memory_start = paging_init(memory_start, memory_end);
19819     trap_init();
19820     init_IRQ();
19821     sched_init();
19822     time_init();
19823     parse_options(command_line);
19824
19825     /* HACK ALERT! This is early. We're enabling the
19826      * console before we've done PCI setups etc, and
19827      * console_init() must be aware of this. But we do want
19828      * output early, in case something goes wrong. */
19829     memory_start = console_init(memory_start, memory_end);
19830 #ifdef CONFIG_MODULES
19831     init_modules();
19832 #endif
19833     if (prof_shift) {
19834         prof_buffer = (unsigned int *) memory_start;
19835         /* only text is profiled */
19836         prof_len = (unsigned long) &etext -
19837                 (unsigned long) &stext;
19838         prof_len >>= prof_shift;
19839         memory_start += prof_len * sizeof(unsigned int);
19840         memset(prof_buffer, 0,
19841                prof_len * sizeof(unsigned int));
19842     }
19843
19844     memory_start = kmem_cache_init(memory_start,
19845                                   memory_end);

```

```

p45 ↪ 19846     sti();
19847     calibrate_delay();
19848 #ifdef CONFIG_BLK_DEV_INITRD
19849     if (initrd_start && !initrd_below_start_ok &&
19850         initrd_start < memory_start) {
19851         printk(KERN_CRIT
19852                "initrd overwritten (0x%08lx < 0x%08lx) - "
19853                "disabling it.\n", initrd_start, memory_start);
19854         initrd_start = 0;
19855     }
19856 #endif
19857     mem_init(memory_start, memory_end);
19858     kmem_cache_sizes_init();
19859 #ifdef CONFIG_PROC_FS
19860     proc_root_init();
19861 #endif
19862     uidcache_init();
19863     filescache_init();
19864     dcache_init();
19865     vma_init();
19866     buffer_init();
19867     signals_init();
19868     inode_init();
19869     file_table_init();
19870 #if defined(CONFIG_SYSVIPC)
19871     ipc_init();
19872 #endif
19873 #if defined(CONFIG_QUOTA)
19874     dquot_init_hash();
19875 #endif
19876     check_bugs();
19877     printk("POSIX conformance testing by UNIFIX\n");
19878
19879     /* We count on the initial thread going ok Like idlers
19880      * init is an unlocked kernel thread, which will make
19881      * syscalls (and thus be locked). */
19882     smp_init();
19883     kernel_thread(init, NULL,
19884                  CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
19885     current->need_resched = 1;
19886     cpu_idle(NULL);
19887 }
19888
19889 #ifdef CONFIG_BLK_DEV_INITRD
19890 static int do_linuxrc(void * shell)
19891 {
19892     static char *argv[] = { "linuxrc", NULL, };
19893

```

```

19894 close(0);close(1);close(2);
19895 setsid();
19896 (void) open("/dev/console",O_RDWR,0);
19897 (void) dup(0);
19898 (void) dup(0);
19899 return execve(shell, argv, envp_init);
19900 }
19901
19902 static void __init no_initrd(char *s,int *ints)
19903 {
19904     mount_initrd = 0;
19905 }
19906 #endif
19907
19908 struct task_struct *child_reaper = &init_task;
19909
19910 /* Ok, the machine is now initialized. None of the
19911  * devices have been touched yet, but the CPU subsystem
19912  * is up and running, and memory and process management
19913  * works.
19914  *
19915  * Now we can finally start doing some real work.. */
19916 static void __init do_basic_setup(void)
19917 {
19918     #ifdef CONFIG_BLK_DEV_INITRD
19919         int real_root_mountflags;
19920     #endif
19921
19922     /* Tell the world that we're going to be the grim
19923      * reaper of innocent orphaned children.
19924      *
19925      * We don't want people to have to make incorrect
19926      * assumptions about where in the task array this can
19927      * be found. */
19928     child_reaper = current;
19929
19930     #if defined(CONFIG_MTRR) /* After SMP initialization */
19931     /* We should probably create some architecture-dependent
19932      * "fixup after everything is up" style function where
19933      * this would belong better than in init/main.c.. */
19934     mtrr_init();
19935     #endif
19936
19937     #ifdef CONFIG_SYSCTL
19938     sysctl_init();
19939     #endif
19940
19941     /* Ok, at this point all CPU's should be initialized,

```

```

19942      * so we can start looking into devices.. */
19943     #ifdef CONFIG_PCI
19944     pci_init();
19945     #endif
19946     #ifdef CONFIG_SBUS
19947     sbus_init();
19948     #endif
19949     #if defined(CONFIG_PPC)
19950     powermac_init();
19951     #endif
19952     #ifdef CONFIG_MCA
19953     mca_init();
19954     #endif
19955     #ifdef CONFIG_ARCH_ACORN
19956     ecard_init();
19957     #endif
19958     #ifdef CONFIG_ZORRO
19959     zorro_init();
19960     #endif
19961     #ifdef CONFIG_DIO
19962     dio_init();
19963     #endif
19964
19965     /* Net initialization needs a process context */
19966     sock_init();
19967
19968     /* Launch bdflush from here, instead of the old syscall
19969      * way. */
19970     kernel_thread(bdflush, NULL,
19971                  CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
19972     /* Start the background pageout daemon. */
19973     kswapd_setup();
19974     kernel_thread(kpiod, NULL,
19975                  CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
19976     kernel_thread(kswapd, NULL,
19977                  CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
19978
19979     #if CONFIG_API1000
19980     /* Start the async paging daemon. */
19981     {
19982         extern int asyncd(void *);
19983         kernel_thread(asyncd, NULL,
19984                       CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
19985     }
19986     #endif
19987
19988     #ifdef CONFIG_BLK_DEV_INITRD
19989

```

```

19990 real_root_dev = ROOT_DEV;
19991 real_root_mountflags = root_mountflags;
19992 if (initrd_start && mount_initrd)
19993     root_mountflags &= ~MS_RDONLY;
19994 else mount_initrd = 0;
19995 #endif
19996
19997 /* Set up devices .. */
19998 device_setup();
19999
20000 /* .. executable formats .. */
20001 binfmt_setup();
20002
20003 /* .. filesystems .. */
20004 filesystem_setup();
20005
20006 /* Mount the root filesystem.. */
20007 mount_root();
20008
20009 #ifdef CONFIG_UMSDOS_FS
20010 {
20011     /* When mounting a umsdos fs as root, we detect the
20012      * pseudo_root (/linux) and initialise it here.
20013      * pseudo_root is defined in fs/umsdos/inode.c */
20014     extern struct inode *pseudo_root;
20015     if (pseudo_root != NULL){
20016         current->fs->root = pseudo_root->i_sb->s_root;
20017         current->fs->pwd = pseudo_root->i_sb->s_root;
20018     }
20019 }
20020 #endif
20021
20022 #ifdef CONFIG_BLK_DEV_INITRD
20023 root_mountflags = real_root_mountflags;
20024 if (mount_initrd && ROOT_DEV != real_root_dev
20025     && MAJOR(ROOT_DEV) == RAMDISK_MAJOR
20026     && MINOR(ROOT_DEV) == 0) {
20027     int error;
20028     int i, pid;
20029
20030     pid = kernel_thread(do_linuxrc, "/linuxrc", SIGCHLD);
20031     if (pid > 0)
20032         while (pid != wait(&i));
20033     if (MAJOR(real_root_dev) != RAMDISK_MAJOR
20034         || MINOR(real_root_dev) != 0) {
20035         error = change_root(real_root_dev, "/initrd");
20036         if (error)
20037             printk(KERN_ERR "Change root to /initrd: "

```

```

20038         "error %d\n", error);
20039     }
20040 }
20041 #endif
20042 }
20043
20044 static int init(void * unused)
20045 {
20046     lock_kernel();
20047     do_basic_setup();
20048
20049     /* OK, we have completed the initial bootup, and we're
20050      * essentially up and running. Get rid of the initmem
20051      * segments and start the user-mode stuff.. */
20052     free_initmem();
20053     unlock_kernel();
20054
20055     if (open("/dev/console", O_RDWR, 0) < 0)
20056         printk("Warning: unable to open an initial "
20057             "console.\n");
20058
20059     (void) dup(0);
20060     (void) dup(0);
20061
20062     /* We try each of these until one succeeds.
20063      *
20064      * The Bourne shell can be used instead of init if we
20065      * are trying to recover a really broken machine. */
20066
20067     if (execute_command)
20068         execve(execute_command, argv_init, envp_init);
20069     execve("/sbin/init", argv_init, envp_init);
20070     execve("/etc/init", argv_init, envp_init);
20071     execve("/bin/init", argv_init, envp_init);
20072     execve("/bin/sh", argv_init, envp_init);
20073     panic("No init found. "
20074         "Try passing init= option to kernel.");
20075 }

```

init/version.c

```

20076 /*
20077  * linux/version.c
20078  *
20079  * Copyright (C) 1992 Theodore Ts'o
20080  *
20081  * May be freely distributed as part of Linux.
20082  */

```



```

20083
20084 #include <linux/uts.h>
20085 #include <linux/utsname.h>
20086 #include <linux/version.h>
20087 #include <linux/compile.h>
20088
20089 #define version(a) Version_ ## a
20090 #define version_string(a) version(a)
20091
20092 int version_string(LINUX_VERSION_CODE) = 0;
20093
20094 struct new_utsname system_utsname = {
20095     UTS_SYSNAME, UTS_NODENAME, UTS_RELEASE, UTS_VERSION,
20096     UTS_MACHINE, UTS_DOMAINNAME
20097 };
20098
20099 const char *linux_banner =
20100     "Linux version " UTS_RELEASE " (" LINUX_COMPILE_BY "@"
20101     LINUX_COMPILE_HOST ") (" LINUX_COMPILER ") "
20102     UTS_VERSION "\n";

```

ipc/msg.c

```

20103 /*
20104  * linux/ipc/msg.c
20105  * Copyright (C) 1992 Krishna Balasubramanian
20106  *
20107  * Removed all the remaining kernel mess
20108  * Catch the -EFAULT stuff properly
20109  * Use GFP_KERNEL for messages as in 1.2
20110  * Fixed up the unchecked user space derefs
20111  * Copyright (C) 1998 Alan Cox & Andi Kleen
20112  *
20113  */
20114
20115 #include <linux/malloc.h>
20116 #include <linux/msg.h>
20117 #include <linux/interrupt.h>
20118 #include <linux/smp_lock.h>
20119 #include <linux/init.h>
20120
20121 #include <asm/uaccess.h>
20122
20123 extern int ipcperms(struct ipc_perm *ipcp, short msgflg);
20124
20125 static void freeque (int id);
20126 static int newque (key_t key, int msgflg);
20127 static int findkey (key_t key);

```

```

20128
p570 20129 static struct msqid_ds *msgque[MSGMNI];
20130 static int msgbytes = 0;
20131 static int msghdrs = 0;
20132 static unsigned short msg_seq = 0;
20133 static int used_queues = 0;
20134 static int max_msgid = 0;
20135 static struct wait_queue *msg_lock = NULL;
20136
p570 20137 void __init msg_init (void)
20138 {
20139     int id;
20140
20141     for (id = 0; id < MSGMNI; id++)
p570 20142         msgque[id] = (struct msqid_ds *) IPC_UNUSED;
20143     msgbytes = msghdrs = msg_seq = max_msgid =
20144         used_queues = 0;
20145     msg_lock = NULL;
20146     return;
20147 }
20148
p571 20149 static int real_msgsnd(int msqid, struct msgbuf *msgp,
20150                        size_t msgsz, int msgflg)
20151 {
20152     int id;
20153     struct msqid_ds *msq;
20154     struct ipc_perm *ipcp;
20155     struct msg *msgh;
20156     long mtype;
20157
20158     if (msgsz > MSGMAX || (long) msgsz < 0 || msqid < 0)
20159         return -EINVAL;
20160     if (get_user(mtype, &msgp->mtype))
20161         return -EFAULT;
20162     if (mtype < 1)
20163         return -EINVAL;
20164     id = (unsigned int) msqid % MSGMNI;
20165     msq = msgque [id];
20166     if (msq == IPC_UNUSED || msq == IPC_NOID)
20167         return -EINVAL;
20168     ipcp = &msq->msg_perm;
20169
20170     slept:
20171     if (msq->msg_perm.seq != (unsigned int) msqid / MSGMNI)
20172         return -EIDRM;
20173
p572 20174     if (ipcperms(ipcp, S_IWUGO))
20175         return -EACCS;

```

```

20176
20177 if (msgsz + msq->msg_cbytes > msq->msg_qbytes) {
20178     if (msgsz + msq->msg_cbytes > msq->msg_qbytes) {
20179         /* still no space in queue */
20180         if (msgflg & IPC_NOWAIT)
20181             return -EAGAIN;
20182         if (signal_pending(current))
20183             return -EINTR;
20184         interruptible_sleep_on (&msq->wwait);
20185         goto slept;
20186     }
20187 }
20188
20189 /* allocate message header and text space*/
20190 msgh = (struct msg *) kmalloc(sizeof(*msgh) + msgsz,
20191                               GFP_KERNEL);
20192 if (!msgh)
20193     return -ENOMEM;
20194 msgh->msg_spot = (char *) (msgh + 1);
20195
20196 if (copy_from_user(msgh->msg_spot, msgp->mtext, msgsz))
20197 {
20198     kfree(msgh);
20199     return -EFAULT;
20200 }
20201
20202 if (msgque[id] == IPC_UNUSED || msgque[id] == IPC_NOID
20203     || msq->msg_perm.seq !=
20204     (unsigned int) msqid / MSGMNI) {
20205     kfree(msgh);
20206     return -EIDRM;
20207 }
20208
20209 msgh->msg_next = NULL;
20210 msgh->msg_ts = msgsz;
20211 msgh->msg_type = mtype;
20212 msgh->msg_stime = CURRENT_TIME;
20213
20214 if (!msq->msg_first)
20215     msq->msg_first = msq->msg_last = msgh;
20216 else {
20217     msq->msg_last->msg_next = msgh;
20218     msq->msg_last = msgh;
20219 }
20220 msq->msg_cbytes += msgsz;
20221 msgbytes += msgsz;
20222 msghdrs++;
20223 msq->msg_qnum++;

```



```

20224 msq->msg_lspid = current->pid;
20225 msq->msg_stime = CURRENT_TIME;
20226 wake_up (&msq->rwait);
20227 return 0;
20228 }
20229
20230 static int real_msgrcv(int msqid, struct msgbuf *msgp,
20231                       size_t msgsz, long msgtyp, int msgflg)
20232 {
20233     struct msqid_ds *msq;
20234     struct ipc_perm *ipcp;
20235     struct msg *tmsg, *leastp = NULL;
20236     struct msg *nmsg = NULL;
20237     int id;
20238
20239     if (msqid < 0 || (long) msgsz < 0)
20240         return -EINVAL;
20241
20242     id = (unsigned int) msqid % MSGMNI;
20243     msq = msgque [id];
20244     if (msq == IPC_NOID || msq == IPC_UNUSED)
20245         return -EINVAL;
20246     ipcp = &msq->msg_perm;
20247
20248     /* find message of correct type.
20249      * msgtyp = 0 => get first.
20250      * msgtyp > 0 => get first message of matching type.
20251      * msgtyp < 0 => get message with least type
20252      * must be < abs(msgtype). */
20253     while (!nmsg) {
20254         if (msq->msg_perm.seq !=
20255             (unsigned int) msqid / MSGMNI) {
20256             return -EIDRM;
20257         }
20258         if (ipcp->ipcp == S_IRUGO) {
20259             return -EACCES;
20260         }
20261
20262         if (msgtyp == 0)
20263             nmsg = msq->msg_first;
20264         else if (msgtyp > 0) {
20265             if (msgflg & MSG_EXCEPT) {
20266                 for (tmsg = msq->msg_first; tmsg;
20267                     tmsg = tmsg->msg_next)
20268                     if (tmsg->msg_type != msgtyp)
20269                         break;
20270                 nmsg = tmsg;
20271             } else {

```

```

20272     for (tmsg = msq->msg_first; tmsg;
20273           tmsg = tmsg->msg_next)
20274         if (tmsg->msg_type == msgtyp)
20275             break;
20276     nmsg = tmsg;
20277 }
20278 } else {
20279     for (leastp = tmsg = msq->msg_first; tmsg;
20280           tmsg = tmsg->msg_next)
20281         if (tmsg->msg_type < leastp->msg_type)
20282             leastp = tmsg;
20283     if (leastp && leastp->msg_type <= - msgtyp)
20284         nmsg = leastp;
20285 }
20286
20287 if (nmsg) { /* done finding a message */
20288     if ((msgsz < nmsg->msg_ts) &&
20289         !(msgflg & MSG_NOERROR)) {
20290         return -E2BIG;
20291     }
20292     msgsz = (msgsz > nmsg->msg_ts)
20293             ? nmsg->msg_ts : msgsz;
20294     if (nmsg == msq->msg_first)
20295         msq->msg_first = nmsg->msg_next;
20296     else {
20297         for (tmsg = msq->msg_first; tmsg;
20298               tmsg = tmsg->msg_next)
20299             if (tmsg->msg_next == nmsg)
20300                 break;
20301         tmsg->msg_next = nmsg->msg_next;
20302         if (nmsg == msq->msg_last)
20303             msq->msg_last = tmsg;
20304     }
20305     if (!(--msq->msg_qnum))
20306         msq->msg_last = msq->msg_first = NULL;
20307
20308     msq->msg_rtime = CURRENT_TIME;
20309     msq->msg_lrpid = current->pid;
20310     msgbytes -= nmsg->msg_ts;
20311     msghdrs--;
20312     msq->msg_cbytes -= nmsg->msg_ts;
20313     wake_up(&msq->wwait);
20314     if (put_user(nmsg->msg_type, &msgp->mtype) ||
20315         copy_to_user(msgp->mtext, nmsg->msg_spot, msgsz))
20316         msgsz = -EFAULT;
20317     kfree(nmsg);
20318     return msgsz;
20319 } else { /* did not find a message */

```

```

20320     if (msgflg & IPC_NOWAIT) {
20321         return -ENOMSG;
20322     }
20323     if (signal_pending(current)) {
20324         return -EINTR;
20325     }
20326     interruptible_sleep_on(&msq->rwait);
20327 }
20328 } /* end while */
20329 return -1;
20330 }
20331
20332 asmlinkage int sys_msgsnd(int msqid, struct msgbuf *msgp
20333                             , size_t msgsz, int msgflg)
20334 {
20335     int ret;
20336
20337     lock_kernel();
20338     ret = real_msgsnd(msqid, msgp, msgsz, msgflg);
20339     unlock_kernel();
20340     return ret;
20341 }
20342
20343 asmlinkage int sys_msgrcv(int msqid, struct msgbuf *msgp
20344                             , size_t msgsz, long msgtyp, int msgflg)
20345 {
20346     int ret;
20347
20348     lock_kernel();
20349     ret = real_msgrcv(msqid, msgp, msgsz, msgtyp, msgflg);
20350     unlock_kernel();
20351     return ret;
20352 }
20353
20354 static int findkey (key_t key)
20355 {
20356     int id;
20357     struct msqid_ds *msq;
20358
20359     for (id = 0; id <= max_msqid; id++) {
20360         while ((msq = msgqueue[id]) == IPC_NOID)
20361             interruptible_sleep_on(&msg_lock);
20362         if (msq == IPC_UNUSED)
20363             continue;
20364         if (key == msq->msg_perm.key)
20365             return id;
20366     }
20367     return -1;

```

→ p 580

```

20368 }
20369
20370 static int newque (key_t key, int msgflg)
20371 {
20372     int id;
20373     struct msqid_ds *msq;
20374     struct ipc_perm *ipcp;
20375
20376     for (id = 0; id < MSGMNI; id++)
20377         if (msgque[id] == IPC_UNUSED) {
20378             msqque[id] = (struct msqid_ds *) IPC_NOID;
20379             goto found;
20380         }
20381     return -ENOSPC;
20382
20383 found:
20384     msq =
20385         (struct msqid_ds *)kmalloc(sizeof(*msq), GFP_KERNEL);
20386     if (!msq) {
20387         msgque[id] = (struct msqid_ds *) IPC_UNUSED;
20388         wake_up (&msg_lock);
20389         return -ENOMEM;
20390     }
20391     ipcp = &msq->msg_perm;
20392     ipcp->mode = (msgflg & S_IRWXUGO);
20393     ipcp->key = key;
20394     ipcp->cuid = ipcp->uid = current->euid;
20395     ipcp->gid = ipcp->cgid = current->egid;
20396     msq->msg_perm.seq = msg_seq;
20397     msq->msg_first = msq->msg_last = NULL;
20398     msq->rwait = msq->wwait = NULL;
20399     msq->msg_cbytes = msq->msg_qnum = 0;
20400     msq->msg_lspid = msq->msg_lrpid = 0;
20401     msq->msg_stime = msq->msg_rtime = 0;
20402     msq->msg_qbytes = MSGMNB;
20403     msq->msg_ctime = CURRENT_TIME;
20404     if (id > max_msqid)
20405         max_msqid = id;
20406     msgque[id] = msq;
20407     used_queues++;
20408     wake_up (&msg_lock);
20409     return (unsigned int) msq->msg_perm.seq * MSGMNI + id;
20410 }
20411
20412 asmlinkage int sys_msqget (key_t key, int msgflg)
20413 {
20414     int id, ret = -EPERM;
20415     struct msqid_ds *msq;

```

```

20416     lock_kernel();
20417     if (key == IPC_PRIVATE)
20418         ret = newque(key, msgflg);
20419     else if ((id = findkey(key)) == 1) { /*key not used*/
20420         if (!(msgflg & IPC_CREAT))
20421             ret = -ENOENT;
20422         else
20423             ret = newque(key, msgflg);
20424     } else if (msgflg & IPC_CREAT && msgflg & IPC_EXCL) {
20425         ret = -EEXIST;
20426     } else {
20427         msq = msgque[id];
20428         if (msq == IPC_UNUSED || msq == IPC_NOID)
20429             ret = -EIDRM;
20430         else if (ipcperms(&msq->msg_perm, msgflg))
20431             ret = -EACCES;
20432         else
20433             ret = (unsigned int) msq->msg_perm.seq*MSGMNI + id;
20434     }
20435     unlock_kernel();
20436     return ret;
20437 }
20438
20439
20440 static void freeque (int id)
20441 {
20442     struct msqid_ds *msq = msgque[id];
20443     struct msq *msgp, *msqh;
20444
20445     msq->msg_perm.seq++;
20446     /* increment, but avoid overflow */
20447     msg_seq = (msg_seq+1) % ((unsigned)(1<<31)/MSGMNI);
20448     msgbytes = msq->msg_cbytes;
20449     if (id == max_msqid)
20450         while (max_msqid &&
20451             (msgque[ max_msqid] == IPC_UNUSED));
20452     msgque[id] = (struct msqid_ds *) IPC_UNUSED;
20453     used_queues--;
20454     while (waitqueue_active(&msq->rwait) ||
20455         waitqueue_active(&msq->wwait)) {
20456         wake_up (&msq->rwait);
20457         wake_up (&msq->wwait);
20458         schedule();
20459     }
20460     for (msgp = msq->msg_first; msgp; msgp = msqh) {
20461         msqh = msgp->msg_next;
20462         msghdrs++;
20463         kfree(msgp);

```

```

20464     }
20465     kfree(msq);
20466 }
20467
20468 asmlinkage int sys_msgctl(int msqid, int cmd,
20469                          struct msqid_ds *buf)
20470 {
20471     int id, err = -EINVAL;
20472     struct msqid_ds *msq;
20473     struct msqid_ds tbuf;
20474     struct ipc_perm *ipcp;
20475
20476     lock_kernel();
20477     if (msqid < 0 || cmd < 0)
20478         goto out;
20479     err = -EFAULT;
20480     switch (cmd) {
20481     case IPC_INFO:
20482     case MSG_INFO:
20483         if (!buf)
20484             goto out;
20485         {
20486             struct msginfo msginfo;
20487             msginfo.msgmni = MSGMNI;
20488             msginfo.msgmax = MSGMAX;
20489             msginfo.msgmnb = MSGMNB;
20490             msginfo.msgmap = MSGMAP;
20491             msginfo.msgpool = MSGPOOL;
20492             msginfo.msgtql = MSGTQL;
20493             msginfo.msgssz = MSGSSZ;
20494             msginfo.msgseg = MSGSEG;
20495             if (cmd == MSG_INFO) {
20496                 msginfo.msgpool = used_queues;
20497                 msginfo.msgmap = msghdrs;
20498                 msginfo.msgtql = msgbytes;
20499             }
20500
20501             err = -EFAULT;
20502             if (copy_to_user(buf, &msginfo,
20503                             sizeof(struct msginfo)))
20504                 goto out;
20505             err = max_msqid;
20506             goto out;
20507         }
20508     case MSG_STAT:
20509         if (!buf)
20510             goto out;
20511         err = -EINVAL;

```

```

20512         if (msqid > max_msqid)
20513             goto out;
20514         msq = msgque[msqid];
20515         if (msq == IPC_UNUSED || msq == IPC_NOID)
20516             goto out;
20517         err = -EACCES;
20518         if (ipcp && (&msq->msg_perm, S_IRUGO))
20519             goto out;
20520         id = (unsigned int) msq->msg_perm.seq*MSGMNI + msqid;
20521         tbuf.msg_perm = msq->msg_perm;
20522         tbuf.msg_stime = msq->msg_stime;
20523         tbuf.msg_rtime = msq->msg_rtime;
20524         tbuf.msg_ctime = msq->msg_ctime;
20525         tbuf.msg_cbytes = msq->msg_cbytes;
20526         tbuf.msg_qnum = msq->msg_qnum;
20527         tbuf.msg_qbytes = msq->msg_qbytes;
20528         tbuf.msg_lspid = msq->msg_lspid;
20529         tbuf.msg_lrpid = msq->msg_lrpid;
20530         err = -EFAULT;
20531         if (copy_to_user(buf, &tbuf, sizeof(*buf)))
20532             goto out;
20533         err = id;
20534         goto out;
20535     case IPC_SET:
20536         if (!buf)
20537             goto out;
20538         err = -EFAULT;
20539         if (!copy_from_user(&tbuf, buf, sizeof(*buf)))
20540             err = 0;
20541         break;
20542     case IPC_STAT:
20543         if (!buf)
20544             goto out;
20545         break;
20546     }
20547
20548     id = (unsigned int) msqid % MSGMNI;
20549     msq = msgque[id];
20550     err = -EINVAL;
20551     if (msq == IPC_UNUSED || msq == IPC_NOID)
20552         goto out;
20553     err = -EIDRM;
20554     if (msq->msg_perm.seq != (unsigned int) msqid / MSGMNI)
20555         goto out;
20556     ipcp = &msq->msg_perm;
20557
20558     switch (cmd) {
20559     case IPC_STAT:

```

```

20560     err = -EACCES;
20561     if (ipcp->ipcperms (ipcp, S_IRUGO))
20562         goto out;
20563     tbuf.msg_perm   = msq->msg_perm;
20564     tbuf.msg_stime  = msq->msg_stime;
20565     tbuf.msg_rtime  = msq->msg_rtime;
20566     tbuf.msg_ctime  = msq->msg_ctime;
20567     tbuf.msg_cbytes = msq->msg_cbytes;
20568     tbuf.msg_qnum   = msq->msg_qnum;
20569     tbuf.msg_qbytes = msq->msg_qbytes;
20570     tbuf.msg_lspid  = msq->msg_lspid;
20571     tbuf.msg_lrpid  = msq->msg_lrpid;
20572     err = -EFAULT;
20573     if (!copy_to_user (buf, &tbuf, sizeof (*buf)))
20574         err = 0;
20575     goto out;
20576 case IPC_SET:
20577     err = -EPERM;
20578     if (current->euid != ipcp->cuid &&
20579         current->euid != ipcp->uid &&
20580         !capable(CAP_SYS_ADMIN))
20581         /* We _could_ check for CAP_CHOWN above, but we
20582            don't */
20583         goto out;
20584     if (tbuf.msg_qbytes > MSGMNB &&
20585         !capable(CAP_SYS_RESOURCE))
20586         goto out;
20587     msq->msg_qbytes = tbuf.msg_qbytes;
20588     ipcp->uid = tbuf.msg_perm.uid;
20589     ipcp->gid = tbuf.msg_perm.gid;
20590     ipcp->mode = (ipcp->mode & ~S_IRWXUGO) |
20591         (S_IRWXUGO & tbuf.msg_perm.mode);
20592     msq->msg_ctime = CURRENT_TIME;
20593     err = 0;
20594     goto out;
20595 case IPC_RMID:
20596     err = -EPERM;
20597     if (current->euid != ipcp->cuid &&
20598         current->euid != ipcp->uid &&
20599         !capable(CAP_SYS_ADMIN))
20600         goto out;
20601
20602     freeque (id);
20603     err = 0;
20604     goto out;
20605 default:
20606     err = -EINVAL;
20607     goto out;

```

```

20608     }
20609 out:
20610     unlock_kernel();
20611     return err;
20612 }
20613

```

ipc/sem.c

```

20614 /*
20615  * linux/ipc/sem.c
20616  * Copyright (C) 1992 Krishna Balasubramanian
20617  * Copyright (C) 1995 Eric Schenk, Bruno Haible
20618  *
20619  * IMPLEMENTATION NOTES ON CODE REWRITE (Eric Schenk,
20620  * January 1995): This code underwent a massive rewrite
20621  * in order to solve some problems with the original
20622  * code. In particular the original code failed to wake
20623  * up processes that were waiting for semval to go to 0
20624  * if the value went to 0 and was then incremented
20625  * rapidly enough. In solving this problem I have also
20626  * modified the implementation so that it processes
20627  * pending operations in a FIFO manner, thus give a
20628  * guarantee that processes waiting for a lock on the
20629  * semaphore won't starve unless another locking process
20630  * fails to unlock.
20631  * In addition the following two changes in behavior have
20632  * been introduced:
20633  * - The original implementation of semop returned the
20634  *   value last semaphore element examined on
20635  *   success. This does not match the manual page
20636  *   specifications, and effectively allows the user to
20637  *   read the semaphore even if they do not have read
20638  *   permissions. The implementation now returns 0 on
20639  *   success as stated in the manual page.
20640  * - There is some confusion over whether the set of undo
20641  *   adjustments to be performed at exit should be done in
20642  *   an atomic manner. That is, if we are attempting to
20643  *   decrement the semval should we queue up and wait until
20644  *   we can do so legally? The original implementation
20645  *   attempted to do this. The current implementation does
20646  *   not do so. This is because I don't think it is the
20647  *   right thing (TM) to do, and because I couldn't see a
20648  *   clean way to get the old behavior with the new design.
20649  *   The POSIX standard and SVID should be consulted to
20650  *   determine what behavior is mandated.
20651  *
20652  * Further notes on refinement (Christoph Rohland,

```

```

20653 * December 1998):
20654 * - The POSIX standard says, that the undo adjustments
20655 * simply should redo. So the current implementation is
20656 * o.K.
20657 * - The previous code had two flaws:
20658 * 1) It actively gave the semaphore to the next waiting
20659 * process sleeping on the semaphore. Since this
20660 * process did not have the cpu this led to many
20661 * unnecessary context switches and bad
20662 * performance. Now we only check which process
20663 * should be able to get the semaphore and if this
20664 * process wants to reduce some semaphore value we
20665 * simply wake it up without doing the operation. So
20666 * it has to try to get it later. Thus e.g. the
20667 * running process may require the semaphore during
20668 * the current time slice. If it only waits for zero
20669 * or increases the semaphore, we do the operation
20670 * in advance and wake it up.
20671 * 2) It did not wake up all zero waiting processes. We
20672 * try to do better but only get the semops right
20673 * which only wait for zero or increase. If there
20674 * are decrement operations in the operations array
20675 * we do the same as before. */
20676
20677 #include <linux/malloc.h>
20678 #include <linux/smp_lock.h>
20679 #include <linux/init.h>
20680
20681 #include <asm/uaccess.h>
20682
20683 extern int ipcperms(struct ipc_perm *ipcp, short semflg);
20684 static int newary (key_t, int, int);
20685 static int findkey (key_t key);
20686 static void freeary (int id);
20687
20688 static struct semid_ds *semary[SEMMNI];
20689 static int used_sems = 0, used_semids = 0;
20690 static struct wait_queue *sem_lock = NULL;
20691 static int max_semid = 0;
20692
20693 static unsigned short sem_seq = 0;
20694
20695 void __init sem_init (void)
20696 {
20697     int i;
20698
20699     sem_lock = NULL;
20700     used_sems = used_semids = max_semid = sem_seq = 0;

```

```

20701     for (i = 0; i < SEMMNI; i++)
20702         semary[i] = (struct semid_ds *) IPC_UNUSED;
20703     return;
20704 }
20705
20706 static int findkey (key_t key)
20707 {
20708     int id;
20709     struct semid_ds *sma;
20710
20711     for (id = 0; id <= max_semid; id++) {
20712         while ((sma = semary[id]) == IPC_NOID)
20713             interruptible_sleep_on (&sem_lock);
20714         if (sma == IPC_UNUSED)
20715             continue;
20716         if (key == sma->sem_perm.key)
20717             return id;
20718     }
20719     return -1;
20720 }
20721
20722 static int newary (key_t key, int nsems, int semflg)
20723 {
20724     int id;
20725     struct semid_ds *sma;
20726     struct ipc_perm *ipcp;
20727     int size;
20728
20729     if (!nsems)
20730         return -EINVAL;
20731     if (used_sems + nsems > SEMMNS)
20732         return -ENOSPC;
20733     for (id = 0; id < SEMMNI; id++)
20734         if (semary[id] == IPC_UNUSED) {
20735             semary[id] = (struct semid_ds *) IPC_NOID;
20736             goto found;
20737         }
20738     return -ENOSPC;
20739 found:
20740     size = sizeof (*sma) + nsems * sizeof (struct sem);
20741     used_sems += nsems;
20742     sma = (struct semid_ds *) kmalloc (size, GFP_KERNEL);
20743     if (!sma) {
20744         semary[id] = (struct semid_ds *) IPC_UNUSED;
20745         used_sems -= nsems;
20746         wake_up (&sem_lock);
20747         return -ENOMEM;
20748     }

```

```

20749 memset (sma, 0, size);
20750 sma->sem_base = (struct sem *) &sma[1];
20751 ipcpc = &sma->sem_perm;
20752 ipcpc->mode = (semflg & S_IRWXUGO);
20753 ipcpc->key = key;
20754 ipcpc->cuid = ipcpc->uid = current->euid;
20755 ipcpc->gid = ipcpc->cgid = current->egid;
20756 sma->sem_perm.seq = sem_seq;
20757 /* sma->sem_pending = NULL; */
20758 sma->sem_pending_last = &sma->sem_pending;
20759 /* sma->undo = NULL; */
20760 sma->sem_nsems = nsems;
20761 sma->sem_ctime = CURRENT_TIME;
20762 if (id > max_semid)
20763     max_semid = id;
20764 used_semid++;
20765 semary[id] = sma;
20766 wake_up (&sem_lock);
20767 return (unsigned int) sma->sem_perm.seq * SEMMNI + id;
20768 }
20769
20770 asmlinkage int sys_semget(key_t key,int nsems,int semflg)
20771 {
20772     int id, err = -EINVAL;
20773     struct semid_ds *sma;
20774
20775     lock_kernel();
20776     if (nsems < 0 || nsems > SEMMSL)
20777         goto out;
20778     if (key == IPC_PRIVATE) {
20779         err = newary(key, nsems, semflg);
20780     } else if ((id = findkey (key)) == 1) {
20781         /* key not used */
20782         if (!(semflg & IPC_CREAT))
20783             err = -ENOENT;
20784         else
20785             err = newary(key, nsems, semflg);
20786     } else if (semflg & IPC_CREAT && semflg & IPC_EXCL) {
20787         err = -EEXIST;
20788     } else {
20789         sma = semary[id];
20790         if (nsems > sma->sem_nsems)
20791             err = -EINVAL;
20792         else if (ipcperms(&sma->sem_perm, semflg))
20793             err = -EACCES;
20794         else
20795             err = (int) sma->sem_perm.seq * SEMMNI + id;
20796     }

```

```

20797 out:
20798     unlock_kernel();
20799     return err;
20800 }
20801
20802 /* Manage the doubly linked list sma->sem_pending as a
20803 * FIFO: insert new queue elements at the tail
20804 * sma->sem_pending_last. */
20805 static inline void append_to_queue(struct semid ds * sma,
20806                                   struct sem_queue * q)
20807 {
20808     *(q->prev = sma->sem_pending_last) = q;
20809     *(sma->sem_pending_last = &q->next) = NULL;
20810 }
20811
20812 static inline void prepend_to_queue(struct semid ds *sma,
20813                                    struct sem_queue * q)
20814 {
20815     q->next = sma->sem_pending;
20816     *(q->prev = &sma->sem_pending) = q;
20817     if (q->next)
20818         q->next->prev = &q->next;
20819     else /* sma->sem_pending_last == &sma->sem_pending */
20820         sma->sem_pending_last = &q->next;
20821 }
20822
20823 static inline void remove_from_queue(
20824     struct semid ds * sma, struct sem_queue * q)
20825 {
20826     *(q->prev) = q->next;
20827     if (q->next)
20828         q->next->prev = q->prev;
20829     else /* sma->sem_pending_last == &q->next */
20830         sma->sem_pending_last = q->prev;
20831     q->prev = NULL; /* mark as removed */
20832 }
20833
20834 /* Determine whether a sequence of semaphore operations
20835 * would succeed all at once. Return 0 if yes, 1 if need
20836 * to sleep, else return error code. */
20837
20838 static int try_atomic_semop(struct semid ds * sma,
20839                            struct sembuf * sops, int nsops, struct sem_undo *un,
20840                            int pid, int do_undo)
20841 {
20842     int result, sem_op;
20843     struct sembuf *sop;
20844     struct sem * curr;

```



```

20845
20846 for (sop = sops; sop < sops + nsops; sop++) {
20847     curr = sma->sem_base + sop->sem_num;
20848     sem_op = sop->sem_op;
20849
20850     if (!sem_op && curr->semval)
20851         goto would_block;
20852
20853     curr->sempid = (curr->sempid << 16) | pid;
20854     curr->semval += sem_op;
20855     if (sop->sem_flg & SEM_UNDO)
20856         un->semadj[sop->sem_num] -= sem_op;
20857
20858     if (curr->semval < 0)
20859         goto would_block;
20860     if (curr->semval > SEMVMX)
20861         goto out_of_range;
20862 }
20863
20864 if (do_undo)
20865 {
20866     sop--;
20867     result = 0;
20868     goto undo;
20869 }
20870
20871 sma->sem_otime = CURRENT_TIME;
20872 return 0;
20873
20874 out_of_range:
20875 result = ERANGE;
20876 goto undo;
20877
20878 would_block:
20879 if (sop->sem_flg & IPC_NOWAIT)
20880     result = EAGAIN;
20881 else
20882     result = 1;
20883
20884 undo:
20885 while (sop >= sops) {
20886     curr = sma->sem_base + sop->sem_num;
20887     curr->semval -= sop->sem_op;
20888     curr->sempid >>= 16;
20889
20890     if (sop->sem_flg & SEM_UNDO)
20891         un->semadj[sop->sem_num] += sop->sem_op;
20892     sop--;

```

```

20893 }
20894
20895 return result;
20896 }
20897
20898 /* Go through the pending queue for the indicated
20899 * semaphore looking for tasks that can be completed. */
20900 static void update_queue (struct semid_ds * sma)
20901 {
20902     int error;
20903     struct sem queue * q;
20904
20905     for (q = sma->sem_pending; q; q = q->next) {
20906
20907         if (q->status == 1)
20908             return; /* wait for other process */
20909
20910         error = try_atomic_semop(sma, q->sops, q->nsops,
20911                                 q->undo, q->pid, q->alter);
20912
20913         /* Does q->sleeper still need to sleep? */
20914         if (error <= 0) {
20915             /* Found one, wake it up */
20916             wake_up_interruptible(&q->sleeper);
20917             if (error == 0 && q->alter) {
20918                 /* if q->alter let it self try */
20919                 q->status = 1;
20920                 return;
20921             }
20922             q->status = error;
20923             remove_from_queue(sma, q);
20924         }
20925     }
20926 }
20927
20928 /* The following counts are associated to each semaphore:
20929 *   semcnt: # tasks waiting on semval being != 0.
20930 *   semzcnt: # tasks waiting on semval being == 0.
20931 * This model assumes that a task waits on exactly one
20932 * semaphore. Since semaphore operations are to be
20933 * performed atomically, tasks actually wait on a whole
20934 * sequence of semaphores simultaneously. The counts we
20935 * return here are a rough approximation, but still
20936 * warrant that semcnt+semzcnt>0 if the task is on the
20937 * pending queue. */
20938 static int count_semaphore (struct semid_ds * sma,
20939                             unsigned semnum)
20940 {

```

```

20941 int semncnt;
20942 struct sem_queue * q;
20943
20944 semncnt = 0;
20945 for (q = sma->sem_pending; q; q = q->next) {
20946     struct sembuf * sops = q->sops;
20947     int nsops = q->nsops;
20948     int i;
20949     for (i = 0; i < nsops; i++)
20950         if (sops[i].sem_num == semnum
20951             && (sops[i].sem_op < 0)
20952             && !(sops[i].sem_flg & IPC_NOWAIT))
20953             semncnt++;
20954     }
20955     return semncnt;
20956 }
20957 static int count_semzcnt (struct semid_ds * sma,
20958                          ushort semnum)
20959 {
20960     int semzcnt;
20961     struct sem_queue * q;
20962
20963     semzcnt = 0;
20964     for (q = sma->sem_pending; q; q = q->next) {
20965         struct sembuf * sops = q->sops;
20966         int nsops = q->nsops;
20967         int i;
20968         for (i = 0; i < nsops; i++)
20969             if (sops[i].sem_num == semnum
20970                 && (sops[i].sem_op == 0)
20971                 && !(sops[i].sem_flg & IPC_NOWAIT))
20972                 semzcnt++;
20973     }
20974     return semzcnt;
20975 }
20976
20977 /* Free a semaphore set. */
20978 static void freeary (int id)
20979 {
20980     struct semid_ds *sma = semary[id];
20981     struct sem_undo *un;
20982     struct sem_queue *q;
20983
20984     /* Invalidate this semaphore set */
20985     sma->sem_perm.seq++;
20986     /* increment, but avoid overflow */
20987     sem_seq = (sem_seq+1) % ((unsigned)(1<<31)/SEMMNI);
20988     used_sems -= sma->sem_nsems;

```

```

20989     if (id == max_semid)
20990         while (max_semid &&
20991              (semary[--max_semid] == IPC_UNUSED));
20992     semary[id] = (struct semid_ds *) IPC_UNUSED;
20993     used_semid--;
20994
20995     /* Invalidate the existing undo structures for this
20996      * semaphore set. (They will be freed without any
20997      * further action in sem_exit().) */
20998     for (un = sma->undo; un; un = un->id_next)
20999         un->semid = -1;
21000
21001     /* Wake up all pending processes and let them fail with
21002      * EIDRM. */
21003     for (q = sma->sem_pending; q; q = q->next) {
21004         q->status = -EIDRM;
21005         q->prev = NULL;
21006         /* doesn't sleep! */
21007         wake_up_interruptible(&q->sleeper);
21008     }
21009
21010     kfree(sma);
21011 }
21012
21013 asmlinkage int sys_semctl(int semid, int semnum, int cmd,
21014                          union semun arg)
21015 {
21016     struct semid_ds *buf = NULL;
21017     struct semid_ds tbuf;
21018     int i, id, val = 0;
21019     struct semid_ds *sma;
21020     struct ipc_perm *ipcp;
21021     struct sem *curr = NULL;
21022     struct sem_undo *un;
21023     unsigned int nsems;
21024     ushort *array = NULL;
21025     ushort sem_io[SEMMSL];
21026     int err = -EINVAL;
21027
21028     lock_kernel();
21029     if (semid < 0 || semnum < 0 || cmd < 0)
21030         goto out;
21031
21032     switch (cmd) {
21033     case IPC_INFO:
21034     case SEM_INFO:
21035     {
21036         struct seminfo seminfo, *tmp = arg, buf;

```

p 580

p 586

```

21037     seminfo.semmni = SEMMNI;
21038     seminfo.semmns = SEMMNS;
21039     seminfo.semmsl = SEMMSL;
21040     seminfo.semopm = SEMOPM;
21041     seminfo.semvmx = SEMVMX;
21042     seminfo.semmnu = SEMMNU;
21043     seminfo.semmap = SEMMAP;
21044     seminfo.semume = SEMUME;
21045     seminfo.semusz = SEMUSZ;
21046     seminfo.semaem = SEMAEM;
21047     if (cmd == SEM_INFO) {
21048         seminfo.semusz = used_semids;
21049         seminfo.semaem = used_sems;
21050     }
21051     err = -EFAULT;
21052     if (copy_to_user(tmp, &seminfo,
21053         sizeof(struct seminfo)))
21054         goto out;
21055     err = max_semid;
21056     goto out;
21057 }
21058
21059 case SEM_STAT:
21060     buf = arg.buf;
21061     err = -EINVAL;
21062     if (semid > max_semid)
21063         goto out;
21064     sma = semory[semid];
21065     if (sma == IPC_UNUSED || sma == IPC_NOID)
21066         goto out;
21067     err = -EACCES;
21068     if (ipcperms (&sma->sem_perm, S_IRUGO))
21069         goto out;
21070     id = (unsigned int) sma->sem_perm.seq*SEMMNI + semid;
21071     tbuf.sem_perm = sma->sem_perm;
21072     tbuf.sem_otime = sma->sem_otime;
21073     tbuf.sem_ctime = sma->sem_ctime;
21074     tbuf.sem_nsems = sma->sem_nsems;
21075     err = -EFAULT;
21076     if (copy_to_user (buf, &tbuf, sizeof(*buf)) == 0)
21077         err = id;
21078     goto out;
21079 }
21080
21081 id = (unsigned int) semid % SEMMNI;
21082 sma = semory [id];
21083 err = -EINVAL;
21084 if (sma == IPC_UNUSED || sma == IPC_NOID)

```

```

21085     goto out;
21086     ipcperm = &sma->sem_perm;
21087     nsems = sma->sem_nsems;
21088     err = -EIDRM;
21089     if (sma->sem_perm.seq != (unsigned int) semid / SEMMNI)
21090         goto out;
21091
21092     switch (cmd) {
21093     case GETVAL:
21094     case GETPID:
21095     case GETCNT:
21096     case GETZCNT:
21097     case SETVAL:
21098         err = -EINVAL;
21099         if (semnum >= nsems)
21100             goto out;
21101         curr = &sma->sem_base[semnum];
21102         break;
21103     }
21104
21105     switch (cmd) {
21106     case GETVAL:
21107     case GETPID:
21108     case GETCNT:
21109     case GETZCNT:
21110     case GETALL:
21111         err = -EACCES;
21112         if (ipcperms (ipcperm, S_IRUGO))
21113             goto out;
21114         switch (cmd) {
21115         case GETVAL : err = curr->semval; goto out;
21116         case GETPID : err = curr->sempid & 0xffff; goto out;
21117         case GETCNT : err = count_semncnt(sma,semnum);
21118             goto out;
21119         case GETZCNT : err = count_semzcnt(sma,semnum);
21120             goto out;
21121         case GETALL:
21122             array = arg.array;
21123             break;
21124         }
21125         break;
21126     case SETVAL:
21127         val = arg.val;
21128         err = -ERANGE;
21129         if (val > SEMVMX || val < 0)
21130             goto out;
21131         break;
21132     case IPC_RMID:

```

```

21133     if (current->euid == ipcp->cuid {
21134         current->euid == ipcp->uid ||
21135         capable(CAP_SYS_ADMIN)) {
21136         freeary (id);
21137         err = 0;
21138         goto out;
21139     }
21140     err = -EPERM;
21141     goto out;
21142 case SETALL: /* arg is a ptr to an array of ushort */
21143     array = arg.array;
21144     err = -EFAULT;
21145     if (copy_from_user(sem_io, array,
21146         nsems * sizeof(ushort)))
21147         goto out;
21148     err = 0;
21149     for (i = 0; i < nsems; i++)
21150         if (sem_io[i] > SEMVMX) {
21151             err = -ERANGE;
21152             goto out;
21153         }
21154     break;
21155 case IPC_STAT:
21156     buf = arg.buf;
21157     break;
21158 case IPC_SET:
21159     buf = arg.buf;
21160     err = copy_from_user (&tbuf, buf, sizeof (*buf));
21161     if (err)
21162         err = -EFAULT;
21163     break;
21164 }
21165
21166 err = -EIDRM;
21167 if (semary[id] == IPC_UNUSED || semary[id] == IPC_NOID)
21168     goto out;
21169 if (sma->sem_perm.seq != (unsigned int) semid / SEMMNI)
21170     goto out;
21171
21172 switch (cmd) {
21173 case GETALL:
21174     err = -EACCES;
21175     if (ipcperms (ipcp, S_IRUGO))
21176         goto out;
21177     for (i = 0; i < sma->sem_nsems; i++)
21178         sem_io[i] = sma->sem_base[i].semval;
21179     if (copy_to_user(array, sem_io,
21180         nsems * sizeof(ushort)))

```

```

21181     err = -EFAULT;
21182     break;
21183 case SETVAL:
21184     err = -EACCES;
21185     if (ipcperms (ipcp, S_IWUGO))
21186         goto out;
21187     for (un = sma->undo; un; un = un->id_next)
21188         un->semadj[semnum] = 0;
21189     curr->semval = val;
21190     sma->sem_ctime = CURRENT_TIME;
21191     /* maybe some queued-up processes were waiting for
21192     * this */
21193     update_queue(sma);
21194     break;
21195 case IPC_SET:
21196     if (current->euid == ipcp->cuid ||
21197         current->euid == ipcp->uid ||
21198         capable(CAP_SYS_ADMIN)) {
21199         ipcp->uid = tbuf.sem_perm.uid;
21200         ipcp->gid = tbuf.sem_perm.gid;
21201         ipcp->mode = (ipcp->mode & ~S_IRWXUGO)
21202             | (tbuf.sem_perm.mode & S_IRWXUGO);
21203         sma->sem_ctime = CURRENT_TIME;
21204         err = 0;
21205         goto out;
21206     }
21207     err = -EPERM;
21208     goto out;
21209 case IPC_STAT:
21210     err = -EACCES;
21211     if (ipcperms (ipcp, S_IRUGO))
21212         goto out;
21213     tbuf.sem_perm = sma->sem_perm;
21214     tbuf.sem_otime = sma->sem_otime;
21215     tbuf.sem_ctime = sma->sem_ctime;
21216     tbuf.sem_nsems = sma->sem_nsems;
21217     if (copy_to_user (buf, &tbuf, sizeof(*buf)))
21218         err = -EFAULT;
21219     break;
21220 case SETALL:
21221     err = -EACCES;
21222     if (ipcperms (ipcp, S_IWUGO))
21223         goto out;
21224     for (i = 0; i < nsems; i++)
21225         sma->sem_base[i].semval = sem_io[i];
21226     for (un = sma->undo; un; un = un->id_next)
21227         for (i = 0; i < nsems; i++)
21228             un->semadj[i] = 0;

```

```

21229     sma->sem_ctime = CURRENT_TIME;
21230     /* maybe some queued-up processes were waiting for
21231      * this */
21232     update_queue(sma);
21233     break;
21234 default:
21235     err = -EINVAL;
21236     goto out;
21237 }
21238 err = 0;
21239 out:
21240 unlock_kernel();
21241 return err;
21242 }
21243
21244 asmlinkage int sys_semop(int semid, struct sembuf *tsops,
21245                          unsigned nsops)
21246 {
21247     int id, size, error = -EINVAL;
21248     struct semid_ds *sma;
21249     struct sembuf sops[SEMOPM], *sop;
21250     struct sem_undo *un;
21251     int undos = 0, decrease = 0, alter = 0;
21252     struct sem_queue queue;
21253
21254     lock_kernel();
21255     if (nsops < 1 || semid < 0)
21256         goto out;
21257     error = -E2BIG;
21258     if (nsops > SEMOPM)
21259         goto out;
21260     error = -EFAULT;
21261     if (copy_from_user(sops, tsops, nsops * sizeof(*tsops)))
21262         goto out;
21263     id = (unsigned int) semid % SEMMNI;
21264     error = -EINVAL;
21265     if ((sma = semary[id]) == IPC_UNUSED || sma == IPC_NOID)
21266         goto out;
21267     error = -EIDRM;
21268     if (sma->sem_perm.seq != (unsigned int) semid / SEMMNI)
21269         goto out;
21270
21271     error = -EFBIG;
21272     for (sop = sops; sop < sops + nsops; sop++) {
21273         if (sop->sem_num >= sma->sem_nsems)
21274             goto out;
21275         if (sop->sem_flg & SEM_UNDO)
21276             undos++;

```

```

21277         if (sop->sem_op < 0)
21278             decrease = 1;
21279         if (sop->sem_op > 0)
21280             alter = 1;
21281     }
21282     alter |= decrease;
21283
21284     error = -EACCES;
21285     if (ipcperms(&sma->sem_perm, alter ? S_IWUGO : S_IRUGO))
21286         goto out;
21287     if (undos) {
21288         /* Make sure we have an undo structure
21289          * for this process and this semaphore set.
21290          */
21291         for (un = current->semundo; un; un = un->proc_next)
21292             if (un->semid == semid)
21293                 break;
21294         if (!un) {
21295             size = sizeof(struct sem_undo) +
21296                   sizeof(short) * sma->sem_nsems;
21297             un = (struct sem_undo *) kmalloc(size, GFP_ATOMIC);
21298             if (!un) {
21299                 error = -ENOMEM;
21300                 goto out;
21301             }
21302             memset(un, 0, size);
21303             un->semadj = (short *) &un[1];
21304             un->semid = semid;
21305             un->proc_next = current->semundo;
21306             current->semundo = un;
21307             un->id_next = sma->undo;
21308             sma->undo = un;
21309         }
21310     } else
21311         un = NULL;
21312
21313     error = try_atomic_semop(sma, sops, nsops, un,
21314                             current->pid, 0);
21315     if (error <= 0)
21316         goto update;
21317
21318     /* We need to sleep on this operation, so we put the
21319      * current task into the pending queue and go to sleep.
21320      */
21321     queue.sma = sma;
21322     queue.sops = sops;
21323     queue.nsops = nsops;
21324     queue.undo = un;

```

```

21325 queue.pid = current->pid;
21326 queue.alter = decrease;
21327 current->semsleeping = &queue;
21328 if (alter)
21329     append_to_queue(sma, &queue);
21330 else
21331     prepend_to_queue(sma, &queue);
21332
21333 for (;;) {
21334     queue.status = -EINTR;
21335     queue.sleeper = NULL;
21336     interruptible_sleep_on(&queue.sleeper);
21337
21338     /* If queue.status == 1 we were woken up and have to
21339      * retry else we simply return. If an interrupt
21340      * occurred we have to clean up the queue
21341      */
21342     if (queue.status == 1)
21343     {
21344         error = try_atomic_semop (sma, sops, nsops, un,
21345             current->pid, 0);
21346         if (error <= 0)
21347             break;
21348     } else {
21349         error = queue.status;;
21350         if (queue.prev) /* got Interrupt */
21351             break;
21352         /* Everything done by update_queue */
21353         current->semsleeping = NULL;
21354         goto out;
21355     }
21356 }
21357 current->semsleeping = NULL;
21358 remove_from_queue(sma, &queue);
21359 update:
21360 if (alter)
21361     update_queue (sma);
21362 out:
21363 unlock_kernel();
21364 return error;
21365 }
21366
21367 /* add semadj values to semaphores, free undo structures.
21368 * undo structures are not freed when semaphore arrays
21369 * are destroyed so some of them may be out of date.
21370 * IMPLEMENTATION NOTE: There is some confusion over
21371 * whether the set of adjustments that needs to be done
21372 * should be done in an atomic manner or not. That is, if

```

p 587

```

21373 * we are attempting to decrement the semval should we
21374 * queue up and wait until we can do so legally? The
21375 * original implementation attempted to do this (queue
21376 * and wait). The current implementation does not do
21377 * so. The POSIX standard and SVID should be consulted to
21378 * determine what behavior is mandated. */
21379 void sem_exit (void)
21380 {
21381     struct sem_queue *q;
21382     struct sem_undo *u, *un = NULL, **up, **unp;
21383     struct semid_ds *sma;
21384     int nsems, i;
21385
21386     /* If the current process was sleeping for a semaphore,
21387      * remove it from the queue.
21388      */
21389     if ((q = current->semsleeping)) {
21390         if (q->prev)
21391             remove_from_queue(q->sma, q);
21392         current->semsleeping = NULL;
21393     }
21394
21395     for (up = &current->semundo; (u = *up);
21396          *up = u->proc_next, kfree(u)) {
21397         if (u->semid == -1)
21398             continue;
21399         sma = semary[(unsigned int) u->semid % SEMMNI];
21400         if (sma == IPC_UNUSED || sma == IPC_NOID)
21401             continue;
21402         if (sma->sem_perm.seq !=
21403             (unsigned int) u->semid / SEMMNI)
21404             continue;
21405         /* remove u from the sma->undo list */
21406         for (unp = &sma->undo; (un = *unp);
21407              unp = &un->id_next) {
21408             if (u == un)
21409                 goto found;
21410         }
21411         printk("sem_exit undo list error id=%d\n", u->semid);
21412         break;
21413     found:
21414         *unp = un->id_next;
21415         /* perform adjustments registered in u */
21416         nsems = sma->sem_nsems;
21417         for (i = 0; i < nsems; i++) {
21418             struct sem *sem = &sma->sem_base[i];
21419             sem->semval += u->semadj[i];
21420             if (sem->semval < 0)

```

```

21421     sem->semval = 0; /* shouldn't happen */
21422     sem->sempid = current->pid;
21423 }
21424 sma->sem_otime = CURRENT_TIME;
21425 /* maybe some queued-up processes were waiting for
21426  * this */
21427     update_queue(sma);
21428 }
21429     current->semundo = NULL;
21430 }

```

ipc/shm.c

```

21431 /*
21432  * linux/ipc/shm.c
21433  * Copyright (C) 1992, 1993 Krishna Balasubramanian
21434  *      Many improvements/fixes by Bruno Haible.
21435  * Replaced 'struct shm_desc' by 'struct vm_area_struct',
21436  * July 1994.
21437  * Fixed the shm swap deallocation (shm_unuse()), August
21438  * 1998 Andrea Arcangeli.
21439  */
21440
21441 #include <linux/malloc.h>
21442 #include <linux/shm.h>
21443 #include <linux/swap.h>
21444 #include <linux/smp_lock.h>
21445 #include <linux/init.h>
21446 #include <linux/vmalloc.h>
21447
21448 #include <asm/uaccess.h>
21449 #include <asm/pgtable.h>
21450
21451 extern int ipcperms(struct ipc_perm *ipcp, short shmflg);
21452 extern unsigned long get_swap_page(void);
21453 static int findkey(key_t key);
21454 static int newseg(key_t key, int shmflg, int size);
21455 static int shm_map(struct vm_area_struct *shmd);
21456 static void killseg(int id);
21457 static void shm_open(struct vm_area_struct *shmd);
21458 static void shm_close(struct vm_area_struct *shmd);
21459 static pte_t shm_swap_in(struct vm_area_struct *,
21460                          unsigned long, unsigned long);
21461
21462 /* total number of shared memory pages */
21463 static int shm_tot = 0;
21464 /* number of shared memory pages that are in memory */
21465 static int shm_rss = 0;

```

```

21466 /* number of shared memory pages that are in swap */
21467 static int shm_swp = 0;
21468 /* every used id is <= max_shmid */
21469 static int max_shmid = 0;
21470 /* calling findkey() may need to wait */
21471 static struct wait_queue *shm_lock = NULL;
21472 static struct shm_kernel *shm_segs[SHMMNI];
21473
21474 /* incremented, for recognizing stale ids */
21475 static unsigned short shm_seq = 0;
21476
21477 /* some statistics */
21478 static unsigned long swap_attempts = 0;
21479 static unsigned long swap_successes = 0;
21480 static unsigned long used_segs = 0;
21481
21482 void __init shm_init (void)
21483 {
21484     int id;
21485
21486     for (id = 0; id < SHMMNI; id++)
21487         shm_segs[id] = (struct shm_kernel *) IPC_UNUSED;
21488     shm_tot = shm_rss = shm_seq = max_shmid = used_segs = 0;
21489     shm_lock = NULL;
21490     return;
21491 }
21492
21493 static int findkey (key_t key)
21494 {
21495     int id;
21496     struct shm_kernel *shp;
21497
21498     for (id = 0; id <= max_shmid; id++) {
21499         while ((shp = shm_segs[id]) == IPC_NOID)
21500             sleep_on (&shm_lock);
21501         if (shp == IPC_UNUSED)
21502             continue;
21503         if (key == shp->u.shm_perm.key)
21504             return id;
21505     }
21506     return -1;
21507 }
21508
21509 /* allocate new shm kernel and pgtable. protected by
21510  * shm_segs[id] = NOID. */
21511 static int newseg (key_t key, int shmflg, int size)
21512 {
21513     struct shm_kernel *shp;

```

```

21514 int numpages = (size + PAGE_SIZE - 1) >> PAGE_SHIFT;
21515 int id, i;
21516
21517 if (size < SHMMIN)
21518     return -EINVAL;
21519 if (shm_tot + numpages >= SHMALL)
21520     return -ENOSPC;
21521 for (id = 0; id < SHMMNI; id++)
21522     if (shm_segs[id] == IPC_UNUSED) {
21523         shm_segs[id] = (struct shmid_kernel *) IPC_NOID;
21524         goto found;
21525     }
21526 return -ENOSPC;
21527
21528 found:
21529 shp = (struct shmid_kernel *) kmalloc(sizeof(*shp),
21530                                     GFP_KERNEL);
21531 if (!shp) {
21532     shm_segs[id] = (struct shmid_kernel *) IPC_UNUSED;
21533     wake_up(&shm_lock);
21534     return -ENOMEM;
21535 }
21536
21537 shp->shm_pages =
21538     (ulong *) vmalloc(numpages*sizeof(ulong));
21539 if (!shp->shm_pages) {
21540     shm_segs[id] = (struct shmid_kernel *) IPC_UNUSED;
21541     wake_up(&shm_lock);
21542     kfree(shp);
21543     return -ENOMEM;
21544 }
21545
21546 for (i = 0; i < numpages; shp->shm_pages[i++] = 0);
21547 shm_tot += numpages;
21548 shp->u.shm_perm.key = key;
21549 shp->u.shm_perm.mode = (shmflg & S_IRWXUGO);
21550 shp->u.shm_perm.cuid = shp->u.shm_perm.uid =
21551     current->euid;
21552 shp->u.shm_perm.cgid = shp->u.shm_perm.gid =
21553     current->egid;
21554 shp->u.shm_perm.seq = shm_seq;
21555 shp->u.shm_segsz = size;
21556 shp->u.shm_cpid = current->pid;
21557 shp->attaches = NULL;
21558 shp->u.shm_lpid = shp->u.shm_nattch = 0;
21559 shp->u.shm_atime = shp->u.shm_dtime = 0;
21560 shp->u.shm_ctime = CURRENT_TIME;
21561 shp->shm_npages = numpages;

```



```

21562
21563 if (id > max_shmid)
21564     max_shmid = id;
21565 shm_segs[id] = shp;
21566 used_segs++;
21567 wake_up(&shm_lock);
21568 return (unsigned int) shp->u.shm_perm.seq*SHMMNI + id;
21569 }
21570
21571 int shmmax = SHMMAX;
21572
21573 asmlinkage int sys_shmget(key_t key,int size,int shmflg)
21574 {
21575     struct shmid_kernel *shp;
21576     int err, id = 0;
21577
21578     down(&current->mm->mmap_sem);
21579     lock_kernel();
21580     if (size < 0 || size > shmmax) {
21581         err = -EINVAL;
21582     } else if (key == IPC_PRIVATE) {
21583         err = newseg(key, shmflg, size);
21584     } else if ((id = findkey(key)) == -1) {
21585         if (!(shmflg & IPC_CREAT))
21586             err = -ENOENT;
21587         else
21588             err = newseg(key, shmflg, size);
21589     } else if ((shmflg & IPC_CREAT) &&
21590                (shmflg & IPC_EXCL)) {
21591         err = -EEXIST;
21592     } else {
21593         shp = shm_segs[id];
21594         if (shp->u.shm_perm.mode & SHM_DEST)
21595             err = -IDRM;
21596         else if (size > shp->u.shm_segsz)
21597             err = -EINVAL;
21598         else if (ipcperms(&shp->u.shm_perm, shmflg))
21599             err = -EACCES;
21600         else
21601             err = (int) shp->u.shm_perm.seq * SHMMNI + id;
21602     }
21603     unlock_kernel();
21604     up(&current->mm->mmap_sem);
21605     return err;
21606 }
21607
21608 /* Only called after testing nattch and SHM_DEST. Here
21609 * pages, pgtable and shmid kernel are freed. */

```



```

p 592 21610 static void killseg (int id)
21611 {
21612     struct shmid_kernel *shp;
21613     int i, numpages;
21614
21615     shp = shm_segs[id];
21616     if (shp == IPC_NOID || shp == IPC_UNUSED) {
21617         printk("shm nono: killseg called on unused seq "
21618             "id=%d\n", id);
21619         return;
21620     }
21621     shp->u.shm_perm.seq++; /* for shmat */
21622     /* increment, but avoid overflow */
21623     shm_seq = (shm_seq+1) % ((unsigned)(1<<31)/SHMMNI);
21624     shm_segs[id] = (struct shmid_kernel *) IPC_UNUSED;
21625     used_segs--;
21626     if (id == max_shmid)
21627         while (max_shmid &&
21628             (shm_segs[- max_shmid] == IPC_UNUSED));
21629     if (!shp->shm_pages) {
21630         printk("shm nono: killseg shp->pages=NULL. "
21631             "id=%d\n", id);
21632         return;
21633     }
21634     numpages = shp->shm_npages;
p 592 21635     for (i = 0; i < numpages ; i++) {
21636         pte_t pte;
21637         pte = __pte(shp->shm_pages[i]);
21638         if (pte_none(pte))
21639             continue;
21640         if (pte_present(pte)) {
21641             free_page (pte_page(pte));
21642             shm_rss--;
21643         } else {
21644             swap_free(pte_val(pte));
21645             shm_swp--;
21646         }
21647     }
21648     vfree(shp->shm_pages);
21649     shm_tot -= numpages;
21650     kfree(shp);
21651     return;
21652 }
21653
p 592 21654 asmlinkage int sys_shmctl (int shmid, int cmd,
21655                             struct shmid_ds *buf)
21656 {
21657     struct shmid_ds tbuf;

```

```

21658     struct shmid_kernel *shp;
21659     struct ipc_perm *ipcp;
21660     int id, err = EINVAL;
21661
21662     lock_kernel();
21663     if (cmd < 0 || shmid < 0)
21664         goto out;
21665     if (cmd == IPC_SET) {
21666         err = EFAULT;
21667         if(copy_from_user (&tbuf, buf, sizeof (*buf)))
21668             goto out;
21669     }
21670
21671     switch (cmd) { /* replace with proc interface : */
21672     case IPC_INFO:
21673     {
21674         struct shminfo shminfo;
21675         err = EFAULT;
21676         if (!buf)
21677             goto out;
21678         shminfo.shmmni = SHMMNI;
21679         shminfo.shmmax = shmmax;
21680         shminfo.shmmin = SHMMIN;
21681         shminfo.shmall = SHMALL;
21682         shminfo.shmseg = SHMSEG;
21683         if (copy_to_user(buf, &shminfo,
21684             sizeof(struct shminfo)))
21685             goto out;
21686         err = max_shmid;
21687         goto out;
21688     }
21689     case SHM_INFO:
21690     {
21691         struct shm_info shm_info;
21692         err = EFAULT;
21693         shm_info.used_ids = used_segs;
21694         shm_info.shm_rss = shm_rss;
21695         shm_info.shm_tot = shm_tot;
21696         shm_info.shm_swp = shm_swp;
21697         shm_info.swap_attempts = swap_attempts;
21698         shm_info.swap_successes = swap_successes;
21699         if(copy_to_user (buf, &shm_info, sizeof(shm_info)))
21700             goto out;
21701         err = max_shmid;
21702         goto out;
21703     }
21704     case SHM_STAT:
21705         err = EINVAL;

```

```

21706     if (shmid > max_shmid)
21707         goto out;
21708     shp = shm_segs[shmid];
21709     if (shp == IPC_UNUSED || shp == IPC_NOID)
21710         goto out;
21711     if (ipcperms (&shp->u.shm_perm, S_IRUGO))
21712         goto out;
21713     id =
21714         (unsigned int)shp->u.shm_perm.seq * SHMMNI + shmid;
21715     err = -EFAULT;
21716     if (copy_to_user (buf, &shp->u, sizeof(*buf)))
21717         goto out;
21718     err = id;
21719     goto out;
21720 )
21721
21722 shp = shm_segs[id - (unsigned int) shmid % SHMMNI];
21723 err = -EINVAL;
21724 if (shp == IPC_UNUSED || shp == IPC_NOID)
21725     goto out;
21726 err = -EIDRM;
21727 if (shp->u.shm_perm.seq !=
21728     (unsigned int) shmid / SHMMNI)
21729     goto out;
21730 ipc = &shp->u.shm_perm;
21731
21732 switch (cmd) {
21733 case SHM_UNLOCK:
21734     err = -EPERM;
21735     if (!capable(CAP_IPC_LOCK))
21736         goto out;
21737     err = -EINVAL;
21738     if (!(ipc->mode & SHM_LOCKED))
21739         goto out;
21740     ipc->mode &= ~SHM_LOCKED;
21741     break;
21742 case SHM_LOCK:
21743 /* Allow superuser to lock segment in memory */
21744 /* Should the pages be faulted in here or leave it to
21745  * user? */
21746 /* need to determine interaction w/ current->swappable */
21747     err = -EPERM;
21748     if (!capable(CAP_IPC_LOCK))
21749         goto out;
21750     err = -EINVAL;
21751     if (ipc->mode & SHM_LOCKED)
21752         goto out;
21753     ipc->mode |= SHM_LOCKED;
21754     break;
21755 case IPC_STAT:
21756     err = -EACCES;
21757     if (ipcperms (ipc, S_IRUGO))
21758         goto out;
21759     err = -EFAULT;
21760     if (copy_to_user (buf, &shp->u, sizeof(shp->u)))
21761         goto out;
21762     break;
21763 case IPC_SET:
21764     if (current->euid == shp->u.shm_perm.uid ||
21765         current->euid == shp->u.shm_perm.cuid ||
21766         capable(CAP_SYS_ADMIN)) {
21767         ipc->uid = tbuf.shm_perm.uid;
21768         ipc->gid = tbuf.shm_perm.gid;
21769         ipc->mode = (ipc->mode & ~S_IRWXUGO)
21770             | (tbuf.shm_perm.mode & S_IRWXUGO);
21771         shp->u.shm_ctime = CURRENT_TIME;
21772         break;
21773     }
21774     err = -EPERM;
21775     goto out;
21776 case IPC_RMID:
21777     if (current->euid == shp->u.shm_perm.uid ||
21778         current->euid == shp->u.shm_perm.cuid ||
21779         capable(CAP_SYS_ADMIN)) {
21780         shp->u.shm_perm.mode |= SHM_DEST;
21781         if (shp->u.shm_nattch <= 0)
21782             killseg (id);
21783         break;
21784     }
21785     err = -EPERM;
21786     goto out;
21787 default:
21788     err = -EINVAL;
21789     goto out;
21790 )
21791 err = 0;
21792 out:
21793 unlock_kernel();
21794 return err;
21795 }
21796
21797 /* The per process internal structure for managing
21798 * segments is 'struct vm_area_struct'. A shmat will add
21799 * to and shmdt will remove from the list.
21800 * shmd->vm_mm         the attacher
21801 * shmd->vm_start      virt addr of attach.

```

```

21802 *           multiple of SHMLBA
21803 * shmd->vm_end   multiple of SHMLBA
21804 * shmd->vm_next  next attach for segment
21805 * shmd->vm_next_share next attach for task
21806 * shmd->vm_offset offset into segment
21807 * shmd->vm_pte   signature for this attach */
21808
21809 static struct vm_operations_struct shm_vm_ops = {
21810     shm_open, /* open - callback for a new vm-area open */
21811     shm_close, /* close - CB for when vm-area is released */
21812     NULL,      /* no need to sync pages at unmap */
21813     NULL,      /* protect */
21814     NULL,      /* sync */
21815     NULL,      /* advise */
21816     NULL,      /* nopage (done with swapin) */
21817     NULL,      /* wppage */
21818     NULL,      /* swapout (hardcoded right now) */
21819     shm_swapin /* swapin */
21820 };
21821
21822 /* Insert shmd into the list shp->attaches */
21823 static inline void insert_attach(
21824     struct shmid_kernel *shp, struct vm_area_struct *shmd)
21825 {
21826     if((shmd->vm_next_share = shp->attaches) != NULL)
21827         shp->attaches->vm_pprev_share = &shmd->vm_next_share;
21828     shp->attaches = shmd;
21829     shmd->vm_pprev_share = &shp->attaches;
21830 }
21831
21832 /* Remove shmd from list shp->attaches */
21833 static inline void remove_attach(
21834     struct shmid_kernel *shp, struct vm_area_struct *shmd)
21835 {
21836     if(shmd->vm_next_share)
21837         shmd->vm_next_share->vm_pprev_share =
21838             shmd->vm_pprev_share;
21839     *shmd->vm_pprev_share = shmd->vm_next_share;
21840 }
21841
21842 /* ensure page tables exist
21843 * mark page table entries with shm_sgn. */
21844 static int shm_map (struct vm_area_struct *shmd)
21845 {
21846     pgd_t *page_dir;
21847     pmd_t *page_middle;
21848     pte_t *page_table;
21849     unsigned long tmp, shm_sgn;

```

→ p 592

→ p 593

```

21850     int error;
21851
21852     /* clear old mappings */
21853     do_munmap(shmd->vm_start,
21854             shm_sgn, shmd->vm_end - shm_sgn);
21855
21856     /* add new mapping */
21857     tmp = shm_sgn - shmd->vm_start;
21858     if((current->mm->total_vm << PAGE_SHIFT) + tmp
21859        > (unsigned long) current->rlim[RLIMIT_AS].rlim_cur)
21860         return -ENOMEM;
21861     current->mm->total_vm += tmp >> PAGE_SHIFT;
21862     insert_vm_struct(current->mm, shmd);
21863     merge_segments(current->mm, shmd->vm_start,
21864                 shm_sgn, shmd->vm_end);
21865
21866     /* map page range */
21867     error = 0;
21868     shm_sgn = shmd->vm_pte +
21869             SWP_ENTRY(0, (shmd->vm_offset >> PAGE_SHIFT)
21870                 << SHM_IDX_SHIFT);
21871     flush_cache_range(shmd->vm_mm, shmd->vm_start,
21872                     shm_sgn);
21873     for (tmp = shmd->vm_start;
21874          tmp < shmd->vm_end;
21875          tmp += PAGE_SIZE,
21876          shm_sgn += SWP_ENTRY(0, 1 << SHM_IDX_SHIFT))
21877     {
21878         page_dir = pgd_offset(shmd->vm_mm, tmp);
21879         page_middle = pmd_alloc(page_dir, tmp);
21880         if (!page_middle) {
21881             error = -ENOMEM;
21882             break;
21883         }
21884         page_table = pte_alloc(page_middle, tmp);
21885         if (!page_table) {
21886             error = -ENOMEM;
21887             break;
21888         }
21889         set_pte(page_table, __pte(shm_sgn));
21890     }
21891     flush_tlb_range(shmd->vm_mm, shmd->vm_start,
21892                 shm_sgn);
21893     return error;
21894 }
21895
21896 /* Fix shmaddr, allocate descriptor, map shm, add attach
21897 * descriptor to lists. */

```

```

p 593 21898 asmlinkage int sys_shmat (int shmid, char *shmat,
21899                               int shmflg, ulong *raddr)
21900 {
21901     struct shmid_kernel *shp;
21902     struct vm_area_struct *shmd;
21903     int err = -EINVAL;
21904     unsigned int id;
21905     unsigned long addr;
21906     unsigned long len;
21907
21908     down(&current->mm->mmap_sem);
21909     lock_kernel();
21910     if (shmid < 0) {
21911         /* printk("shmat() -> EINVAL because shmid = "
21912            *      "%d < 0\n", shmid); */
21913         goto out;
21914     }
21915
21916     shp = shm_segs[id = (unsigned int) shmid % SHMMNI];
21917     if (shp == IPC_UNUSED || shp == IPC_NOID) {
21918         /* printk("shmat() -> EINVAL because shmid = %d "
21919            *      "is invalid\n", shmid); */
21920         goto out;
21921     }
21922
21923     if (!(addr = (ulong) shmat)) {
21924         if (shmflg & SHM_REMAP)
21925             goto out;
21926         err = -ENOMEM;
21927         addr = 0;
21928     again:
21929         if (!(addr = get_unmapped_area(addr,
21930                                       shp->u.shm_segsz)))
21931             goto out;
21932         if (addr & (SHMLBA - 1)) {
21933             addr = (addr + (SHMLBA - 1)) & ~(SHMLBA - 1);
21934             goto again;
21935         }
21936     } else if (addr & (SHMLBA-1)) {
21937         if (shmflg & SHM_RND)
21938             addr &= ~(SHMLBA-1); /* round down */
21939         else
21940             goto out;
21941     }
21942     /* Check if addr exceeds TASK_SIZE (from do_mmap) */
21943     len = PAGE_SIZE*shp->shm_npages;
21944     err = -EINVAL;
21945     if (addr >= TASK_SIZE || len > TASK_SIZE ||

```

```

21946         addr > TASK_SIZE - len)
21947         goto out;
21948     /* If shm segment goes below stack, make sure there is
21949     * some space left for the stack to grow (presently 4
21950     * pages). */
p 594 21951     if (addr < current->mm->start_stack &&
21952         addr > current->mm->start_stack -
21953             PAGE_SIZE*(shp->shm_npages + 4))
21954     {
21955         /* printk("shmat() -> EINVAL because segment "
21956            *      "intersects stack\n"); */
21957         goto out;
21958     }
21959     if (!(shmflg & SHM_REMAP))
21960         if ((shmd = find_vma_intersection(current->mm, addr,
21961                                           addr + shp->u.shm_segsz))) {
21962             /* printk("shmat() -> EINVAL because the interval "
21963                *      "[0x%x,0x%x] intersects an already "
21964                *      "mapped interval [0x%x,0x%x).\n",
21965                *      addr, addr + shp->shm_segsz,
21966                *      shmd->vm_start, shmd->vm_end); */
21967             goto out;
21968         }
21969
21970     err = -EACCES;
21971     if (ipcperms(&shp->u.shm_perm, shmflg & SHM_RDONLY
21972                ? S_IRUGO : S_IRUGO|S_IWUGO))
21973         goto out;
21974     err = -EIDRM;
21975     if (shp->u.shm_perm.seq !=
21976         (unsigned int) shmid / SHMMNI)
21977         goto out;
21978
21979     err = -ENOMEM;
21980     shmd = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
21981     if (!shmd)
21982         goto out;
21983     if ((shp != shm_segs[id]) ||
21984         (shp->u.shm_perm.seq !=
21985          (unsigned int) shmid / SHMMNI)) {
21986         kmem_cache_free(vm_area_cachep, shmd);
21987         err = -EIDRM;
21988         goto out;
21989     }
21990
21991     shmd->vm_pte = SWP_ENTRY(SHM_SWP_TYPE, id);
21992     shmd->vm_start = addr;
21993     shmd->vm_end = addr + shp->shm_npages * PAGE_SIZE;

```

```

21994 shmd->vm_mm = current->mm;
21995 shmd->vm_page_prot = (shmflg & SHM_RDONLY)
21996         ? PAGE_READONLY : PAGE_SHARED;
21997 shmd->vm_flags = VM_SHM | VM_MAYSHARE | VM_SHARED
21998         | VM_MAYREAD | VM_MAYEXEC | VM_READ | VM_EXEC
21999         | ((shmflg & SHM_RDONLY) ? 0 : VM_MAYWRITE|VM_WRITE);
22000 shmd->vm_file = NULL;
22001 shmd->vm_offset = 0;
22002 shmd->vm_ops = &shm_vm_ops;
22003
22004 shp->u.shm_nattch++; /* prevent destruction */
22005 if ((err = shm_map(shmd)) {
22006     if (--shp->u.shm_nattch <= 0 &&
22007         shp->u.shm_perm.mode & SHM_DEST)
22008         killseg(id);
22009     kmem_cache_free(vm_area_cache, shmd);
22010     goto out;
22011 }
22012
22013 /* insert shmd into shp->attaches */
22014 insert_attach(shp,shmd);
22015
22016 shp->u.shm_lpid = current->pid;
22017 shp->u.shm_atime = CURRENT_TIME;
22018
22019 *raddr = addr;
22020 err = 0;
22021 out:
22022 unlock_kernel();
22023 up(&current->mm->mmap_sem);
22024 return err;
22025 }
22026
22027 /* This is called by fork, once for every shm attach. */
22028 static void shm_open (struct vm_area_struct *shmd)
22029 {
22030     unsigned int id;
22031     struct shmid_kernel *shp;
22032
22033     id = SWP_OFFSET(shmd->vm_pte) & SHM_ID_MASK;
22034     shp = shm_segs[id];
22035     if (shp == IPC_UNUSED) {
22036         printk("shm_open: unused id=%d PANIC\n", id);
22037         return;
22038     }
22039     /* insert shmd into shp->attaches */
22040     insert_attach(shp,shmd);
22041     shp->u.shm_nattch++;

```

p 595

p 595

p 595

```

22042 shp->u.shm_atime = CURRENT_TIME;
22043 shp->u.shm_lpid = current->pid;
22044 }
22045
22046 /* remove the attach descriptor shmd.
22047 * free memory for segment if it is marked destroyed.
22048 * The descriptor has already been removed from the
22049 * current->mm->mmap list and will later be kfree()d. */
22050 static void shm_close (struct vm_area_struct *shmd)
22051 {
22052     struct shmid_kernel *shp;
22053     int id;
22054
22055     /* remove from the list of attaches of the shm seg */
22056     id = SWP_OFFSET(shmd->vm_pte) & SHM_ID_MASK;
22057     shp = shm_segs[id];
22058     remove_attach(shp,shmd); /* remove from shp->attaches*/
22059     shp->u.shm_lpid = current->pid;
22060     shp->u.shm_dtime = CURRENT_TIME;
22061     if (--shp->u.shm_nattch <= 0 &&
22062         shp->u.shm_perm.mode & SHM_DEST)
22063         killseg(id);
22064 }
22065
22066 /* detach and kill segment if marked destroyed. The work
22067 * is done in shm_close. */
22068 asmlinkage int sys_shmdt (char *shmaddr)
22069 {
22070     struct vm_area_struct *shmd, *shmdnext;
22071
22072     down(&current->mm->mmap_sem);
22073     lock_kernel();
22074     for (shmd = current->mm->mmap; shmd; shmd = shmdnext) {
22075         shmdnext = shmd->vm_next;
22076         if (shmd->vm_ops == &shm_vm_ops &&
22077             shmd->vm_start - shmd->vm_offset ==
22078                 (ulong) shmaddr)
22079             do_munmap(shmd->vm_start,
22080                 shmd->vm_end - shmd->vm_start);
22081     }
22082     unlock_kernel();
22083     up(&current->mm->mmap_sem);
22084     return 0;
22085 }
22086
22087 /* page not present ... go through shm_pages */
22088 static pte_t shm_swap_in(struct vm_area_struct * shmd,
22089     unsigned long offset, unsigned long code)

```

```

22090 {
22091     pte_t pte;
22092     struct shmid_kernel *shp;
22093     unsigned int id, idx;
22094
22095     id = SWP_OFFSET(code) & SHM_ID_MASK;
22096 #ifdef DEBUG_SHM
22097     if (id != (SWP_OFFSET(shmd->vm_pte) & SHM_ID_MASK)) {
22098         printk("shm_swap_in: code id = %d and shmd id = %ld "
22099             "differ\n",
22100             id, SWP_OFFSET(shmd->vm_pte) & SHM_ID_MASK);
22101         return BAD_PAGE;
22102     }
22103     if (id > max_shmid) {
22104         printk("shm_swap_in: id=%d too big. proc mem "
22105             "corrupted\n", id);
22106         return BAD_PAGE;
22107     }
22108 #endif
22109     shp = shm_segs[id];
22110
22111 #ifdef DEBUG_SHM
22112     if (shp == IPC_UNUSED || shp == IPC_NOID) {
22113         printk("shm_swap_in: id=%d invalid. Race.\n", id);
22114         return BAD_PAGE;
22115     }
22116 #endif
22117     idx =
22118         (SWP_OFFSET(code) >> SHM_IDX_SHIFT) & SHM_IDX_MASK;
22119 #ifdef DEBUG_SHM
22120     if (idx != (offset >> PAGE_SHIFT)) {
22121         printk("shm_swap_in: code idx = %u and shmd idx = "
22122             "%lu differ\n", idx, offset >> PAGE_SHIFT);
22123         return BAD_PAGE;
22124     }
22125     if (idx >= shp->shm_npages) {
22126         printk("shm_swap_in: too large page index. id=%d\n",
22127             id);
22128         return BAD_PAGE;
22129     }
22130 #endif
22131
22132     pte = __pte(shp->shm_pages[idx]);
22133     if (!pte_present(pte)) {
22134         unsigned long page = get_free_page(GFP_KERNEL);
22135         if (!page) {
22136             oom(current);
22137             return BAD_PAGE;

```

```

22138     }
22139     pte = __pte(shp->shm_pages[idx]);
22140     if (pte_present(pte)) {
22141         free_page(page); /* doesn't sleep */
22142         goto done;
22143     }
22144     if (!pte_none(pte)) {
22145         rw_swap_page_nocache(READ, pte_val(pte),
22146             (char *)page);
22147         pte = __pte(shp->shm_pages[idx]);
22148         if (pte_present(pte)) {
22149             free_page(page); /* doesn't sleep */
22150             goto done;
22151         }
22152         swap_free(pte_val(pte));
22153         shm_swp--;
22154     }
22155     shm_rss++;
22156     pte = pte_mkdirty(mk_pte(page, PAGE_SHARED));
22157     shp->shm_pages[idx] = pte_val(pte);
22158 } else
22159     --current->maj_flt; /* was incd in do_no_page */
22160
22161 done: /* pte_val(pte) == shp->shm_pages[idx] */
22162     current->min_flt++;
22163     atomic_inc(&mem_map[MAP_NR(pte_page(pte))].count);
22164     return pte_modify(pte, shmd->vm_page_prot);
22165 }
22166
22167 /* Goes through counter = (shm_rss >> prio) present shm
22168 * pages. */
22169 static unsigned long swap_id = 0; /* now being swapped */
22170 static unsigned long swap_idx = 0; /* next to swap */
22171
22172 int shm_swap (int prio, int gfp_mask)
22173 {
22174     pte_t page;
22175     struct shmid_kernel *shp;
22176     struct vm_area_struct *shmd;
22177     unsigned long swap_nr;
22178     unsigned long id, idx;
22179     int loop = 0;
22180     int counter;
22181
22182     counter = shm_rss >> prio;
22183     if (!counter || !(swap_nr = get_swap_page()))
22184         return 0;
22185

```

```

22186 check_id:
22187 shp = shm_segs[swap_id];
22188 if (shp == IPC_UNUSED || shp == IPC_NOID ||
22189     shp->u.shm_perm.mode & SHM_LOCKED ) {
22190     next_id:
22191     swap_idx = 0;
22192     if (++swap_id > max_shmid) {
22193         if (loop)
22194             goto failed;
22195         loop = 1;
22196         swap_id = 0;
22197     }
22198     goto check_id;
22199 }
22200 id = swap_id;
22201
22202 check_table:
22203 idx = swap_idx++;
22204 if (idx >= shp->shm_npages)
22205     goto next_id;
22206
22207 page = __pte(shp->shm_pages[idx]);
22208 if (!pte_present(page))
22209     goto check_table;
22210 if ((gfp_mask & __GFP_DMA) &&
22211     !PageDMA(&mem_map[MAP_NR(pte_page(page))]))
22212     goto check_table;
22213 swap_attempts++;
22214
22215 if (--counter < 0) { /* failed */
22216     failed:
22217     swap_free (swap_nr);
22218     return 0;
22219 }
22220 if (shp->attaches)
22221     for (shmd = shp->attaches; ; ) {
22222         do {
22223             pgd_t *page_dir;
22224             pmd_t *page_middle;
22225             pte_t *page_table, pte;
22226             unsigned long tmp;
22227
22228             if ((SWP_OFFSET(shmd->vm_pte) & SHM_ID_MASK) != id) {
22229                 printk("shm_swap: id=%ld does not match "
22230                        "shmd->vm_pte.id=%ld\n",
22231                        id, SWP_OFFSET(shmd->vm_pte) & SHM_ID_MASK);
22232                 continue;
22233             }

```

```

22234     tmp = shmd->vm_start + (idx << PAGE_SHIFT) -
22235           shmd->vm_offset;
22236     if (!(tmp >= shmd->vm_start && tmp < shmd->vm_end))
22237         continue;
22238     page_dir = pgd_offset(shmd->vm_mm, tmp);
22239     if (pgd_none(*page_dir) || pgd_bad(*page_dir)) {
22240         printk("shm_swap: bad pgtbl! id=%ld start=%lx "
22241                "idx=%ld\n", id, shmd->vm_start, idx);
22242         pgd_clear(page_dir);
22243         continue;
22244     }
22245     page_middle = pmd_offset(page_dir, tmp);
22246     if (pmd_none(*page_middle) || pmd_bad(*page_middle)) {
22247         printk("shm_swap: bad pgmid! id=%ld start=%lx "
22248                "idx=%ld\n", id, shmd->vm_start, idx);
22249         pmd_clear(page_middle);
22250         continue;
22251     }
22252     page_table = pte_offset(page_middle, tmp);
22253     pte = *page_table;
22254     if (!pte_present(pte))
22255         continue;
22256     if (pte_young(pte)) {
22257         set_pte(page_table, pte_mkold(pte));
22258         continue;
22259     }
22260     if (pte_page(pte) != pte_page(page))
22261         printk("shm_swap_out: page and pte mismatch "
22262                "%lx %lx\n", pte_page(pte), pte_page(page));
22263     flush_cache_page(shmd, tmp);
22264     set_pte(page_table,
22265             __pte(shmd->vm_pte +
22266                  SWP_ENTRY(0, idx << SHM_IDX_SHIFT)));
22267     atomic_dec(&mem_map[MAP_NR(pte_page(pte))].count);
22268     if (shmd->vm_mm->rss > 0)
22269         shmd->vm_mm->rss--;
22270     flush_tlb_page(shmd, tmp);
22271     /* continue looping through the linked list */
22272     } while (0);
22273     shmd = shmd->vm_next_share;
22274     if (!shmd)
22275         break;
22276 }
22277
22278 if (atomic_read(&mem_map[MAP_NR(pte_page(page))].count)
22279     != 1)
22280     goto check_table;
22281 shp->shm_pages[idx] = swap_nr;

```

```

22282 rw_swap_page_nocache(WRITE, swap_nr,
22283                      (char *) pte_page(page));
22284 free_page(pte_page(page));
22285 swap_successes++;
22286 shm_swp++;
22287 shm_rss--;
22288 return 1;
22289 }
22290
22291 /* Free the swap entry and set the new pte for the shm
22292  * page. */
22293 static void shm_unuse_page(struct shmid_kernel *shp,
22294                          unsigned long idx, unsigned long page,
22295                          unsigned long entry)
22296 {
22297     pte_t pte;
22298
22299     pte = pte_mkdirty(mk_pte(page, PAGE_SHARED));
22300     shp->shm_pages[idx] = pte_val(pte);
22301     atomic_inc(&mem_map[MAP_NR(page)].count);
22302     shm_rss++;
22303
22304     swap_free(entry);
22305     shm_swp--;
22306 }
22307
22308 /* unuse_shm() search for an eventually swapped out shm
22309  * page. */
22310 void shm_unuse(unsigned long entry, unsigned long page)
22311 {
22312     int i, n;
22313
22314     for (i = 0; i < SHMMNI; i++)
22315         if (shm_segs[i] != IPC_UNUSED &&
22316             shm_segs[i] != IPC_NOID)
22317             for (n = 0; n < shm_segs[i]->shm_npages; n++)
22318                 if (shm_segs[i]->shm_pages[n] == entry)
22319                     {
22320                         shm_unuse_page(shm_segs[i], n,
22321                                       page, entry);
22322                     }
22323     return;
22324 }

```

ipc/util.c

```

22325 /*
22326  * linux/ipc/util.c

```

```

22327  * Copyright (C) 1992 Krishna Balasubramanian
22328  *
22329  * Sep 1997 - Call suser() last after "normal" permission
22330  * checks so we get BSD style process accounting right.
22331  * Occurs in several places in the IPC code. Chris
22332  * Evans. <chris@ferret.lmh.ox.ac.uk>
22333  */
22334
22335 #include <linux/config.h>
22336 #include <linux/mm.h>
22337 #include <linux/shm.h>
22338 #include <linux/init.h>
22339 #include <linux/msg.h>
22340
22341 #if defined(CONFIG_SYSVIPC)
22342
22343 extern void sem_init(void), msg_init(void),
22344          shm_init(void);
22345
22346 void __init ipc_init(void)
22347 {
22348     sem_init(),
22349     msg_init(),
22350     shm_init();
22351     return;
22352 }
22353
22354 /* Check user, group, other permissions for access to ipc
22355  * resources. return 0 if allowed */
22356 int ipcperms (struct ipc_perm *ipcp, short flag)
22357 {
22358     /* flag will most probably be 0 or S_...UGO from
22359      * <linux/stat.h> */
22360     int requested_mode, granted_mode;
22361
22362     requested_mode = (flag >> 6) | (flag >> 3) | flag;
22363     granted_mode = ipcp->mode;
22364     if (current->euid == ipcp->cuid ||
22365         current->euid == ipcp->uid)
22366         granted_mode >>= 6;
22367     else if (in_group_p(ipcp->cgid) ||
22368             in_group_p(ipcp->gid))
22369         granted_mode >>= 3;
22370     /* is there some bit set in requested_mode but not in
22371      * granted_mode? */
22372     if ((requested_mode & ~granted_mode & 0007) &&
22373         !capable(CAP_IPC_OWNER))
22374         return -1;

```



```

22375
22376     return 0;
22377 }
22378
22379 #else
22380 /* Dummy functions when SYSV IPC isn't configured */
22381
22382 void sem_exit (void)
22383 {
22384     return;
22385 }
22386
22387 int shm_swap (int prio, int gfp_mask)
22388 {
22389     return 0;
22390 }
22391
22392 asmlinkage int sys_semget (key_t key, int nsems,
22393                           int semflg)
22394 {
22395     return -ENOSYS;
22396 }
22397
22398 asmlinkage int sys_semop (int semid, struct sembuf *sops,
22399                          unsigned nsops)
22400 {
22401     return -ENOSYS;
22402 }
22403
22404 asmlinkage int sys_semctl (int semid, int semnum,
22405                           int cmd, union semun arg)
22406 {
22407     return -ENOSYS;
22408 }
22409
22410 asmlinkage int sys_msgget (key_t key, int msgflg)
22411 {
22412     return -ENOSYS;
22413 }
22414
22415 asmlinkage int sys_msgsnd (int msqid,
22416                          struct msgbuf *msgp, size_t msgsz, int msgflg)
22417 {
22418     return -ENOSYS;
22419 }
22420
22421 asmlinkage int sys_msgrcv (int msqid, struct msgbuf *msgp,
22422                          size_t msgsz, long msgtyp, int msgflg)

```

```

22423 {
22424     return -ENOSYS;
22425 }
22426
22427 asmlinkage int sys_msgctl (int msqid, int cmd,
22428                          struct msqid_ds *buf)
22429 {
22430     return -ENOSYS;
22431 }
22432
22433 asmlinkage int sys_shmget (key_t key, int size, int flag)
22434 {
22435     return -ENOSYS;
22436 }
22437
22438 asmlinkage int sys_shmat (int shmid, char *shmaddr,
22439                          int shmflg, ulong *addr)
22440 {
22441     return -ENOSYS;
22442 }
22443
22444 asmlinkage int sys_shmdt (char *shmaddr)
22445 {
22446     return -ENOSYS;
22447 }
22448
22449 asmlinkage int sys_shmctl (int shmid, int cmd,
22450                          struct shmctl_ds *buf)
22451 {
22452     return -ENOSYS;
22453 }
22454
22455 void shm_unuse (unsigned long entry, unsigned long page)
22456 {
22457 }
22458
22459 #endif /* CONFIG_SYSVIPC */

```

kernel/capability.c

```

22460 /*
22461  * linux/kernel/capability.c
22462  *
22463  * Copyright (C) 1997 Andrew Main <zefram@fysh.org>
22464  * Integrated into 2.1.97+, Andrew G. Morgan
22465  * <morgan@transmeta.com>
22466  */
22467

```

```

22468 #include <linux/mm.h>
22469 #include <asm/uaccess.h>
22470
22471 /* Note: never hold tasklist_lock while spinning for this
22472  * one */
22473 spinlock_t task_capability_lock;
22474
22475 /* For sys_getproccap() and sys_setproccap(), any of the
22476  * three capability set pointers may be NULL --
22477  * indicating that that set is uninteresting and/or not
22478  * to be changed. */
22479
22480 asmlinkage int sys_capget(cap_user_header_t header,
22481                          cap_user_data_t dataptr)
22482 {
22483     int error, pid;
22484     __u32 version;
22485     struct task_struct *target;
22486     struct __user_cap_data_struct data;
22487
22488     if (get_user(version, &header->version))
22489         return -EFAULT;
22490
22491     error = -EINVAL;
22492     if (version != _LINUX_CAPABILITY_VERSION) {
22493         version = _LINUX_CAPABILITY_VERSION;
22494         if (put_user(version, &header->version))
22495             error = -EFAULT;
22496         return error;
22497     }
22498
22499     if (get_user(pid, &header->pid))
22500         return -EFAULT;
22501
22502     if (pid < 0)
22503         return -EINVAL;
22504
22505     error = 0;
22506
22507     spin_lock(&task_capability_lock);
22508
22509     if (pid && pid != current->pid) {
22510         read_lock(&tasklist_lock);
22511         /* identify target of query */
22512         target = find_task_by_pid(pid);
22513         if (!target)
22514             error = -ESRCH;
22515     } else {
22516         target = current;
22517     }
22518
22519     if (!error) {
22520         data.permitted = cap_t(target->cap_permitted);
22521         data.inheritable = cap_t(target->cap_inheritable);
22522         data.effective = cap_t(target->cap_effective);
22523     }
22524
22525     if (target != current)
22526         read_unlock(&tasklist_lock);
22527     spin_unlock(&task_capability_lock);
22528
22529     if (!error) {
22530         if (copy_to_user(dataptr, &data, sizeof data))
22531             return -EFAULT;
22532     }
22533
22534     return error;
22535 }
22536
22537 /* set capabilities for all processes in a given process
22538  * group */
22539 static void cap_set_pg(int pgrp,
22540                       kernel_cap_t *effective,
22541                       kernel_cap_t *inheritable,
22542                       kernel_cap_t *permitted)
22543 {
22544     struct task_struct *target;
22545
22546     /* FIXME: do we need to have a write lock here..? */
22547     read_lock(&tasklist_lock);
22548     for_each_task(target) {
22549         if (target->pgrp != pgrp)
22550             continue;
22551         target->cap_effective = *effective;
22552         target->cap_inheritable = *inheritable;
22553         target->cap_permitted = *permitted;
22554     }
22555     read_unlock(&tasklist_lock);
22556 }
22557
22558 /* set capabilities for all processes other than 1 and
22559  * self */
22560
22561 static void cap_set_all(kernel_cap_t *effective,
22562                       kernel_cap_t *inheritable,
22563                       kernel_cap_t *permitted)

```

```

22564 {
22565     struct task_struct *target;
22566
22567     /* FIXME: do we need to have a write lock here..? */
22568     read_lock(&tasklist_lock);
22569     /* ALL means everyone other than self or 'init' */
22570     for_each_task(target) {
22571         if (target == current || target->pid == 1)
22572             continue;
22573         target->cap_effective = *effective;
22574         target->cap_inheritable = *inheritable;
22575         target->cap_permitted = *permitted;
22576     }
22577     read_unlock(&tasklist_lock);
22578 }
22579
22580 /* The restrictions on setting capabilities are specified
22581  * as:
22582  *
22583  * [pid is for the 'target' task. 'current' is the
22584  * calling task.]
22585  *
22586  * I: any raised capabilities must be a subset of the
22587  * (old current) Permitted
22588  * P: any raised capabilities must be a subset of the
22589  * (old current) permitted
22590  * E: must be set to a subset of (new target) Permitted
22591  */
22592 asmlinkage int sys_capset(cap_user_header_t header,
22593                          const cap_user_data_t data)
22594 {
22595     kernel_cap_t inheritable, permitted, effective;
22596     __u32 version;
22597     struct task_struct *target;
22598     int error, pid;
22599
22600     if (get_user(version, &header->version))
22601         return -EFAULT;
22602
22603     if (version != _LINUX_CAPABILITY_VERSION) {
22604         version = _LINUX_CAPABILITY_VERSION;
22605         if (put_user(version, &header->version))
22606             return -EFAULT;
22607         return -EINVAL;
22608     }
22609
22610     if (get_user(pid, &header->pid))
22611         return -EFAULT;

```

```

22612
22613     if (pid && !capable(CAP_SETPCAP))
22614         return -EPERM;
22615
22616     if (copy_from_user(&effective, &data->effective,
22617                      sizeof(effective)) ||
22618         copy_from_user(&inheritable, &data->inheritable,
22619                      sizeof(inheritable)) ||
22620         copy_from_user(&permitted, &data->permitted,
22621                      sizeof(permitted)))
22622         return -EFAULT;
22623
22624     error = -EPERM;
22625     spin_lock(&task_capability_lock);
22626
22627     if (pid > 0 && pid != current->pid) {
22628         read_lock(&tasklist_lock);
22629         /* identify target of query */
22630         target = find_task_by_pid(pid);
22631         if (!target) {
22632             error = -ESRCH;
22633             goto out;
22634         }
22635     } else {
22636         target = current;
22637     }
22638
22639     /* verify restrictions on target's new Inheritable
22640     * set */
22641     if (!cap_issubset(inheritable,
22642                     cap_combine(target->cap_inheritable,
22643                                 current->cap_permitted))) {
22644         goto out;
22645     }
22646
22647     /* verify restrictions on target's new Permitted
22648     * set */
22649     if (!cap_issubset(permitted,
22650                     cap_combine(target->cap_permitted,
22651                                 current->cap_permitted))) {
22652         goto out;
22653     }
22654
22655     /* verify the _new_Effective_ is a subset of the
22656     * _new_Permitted_ */
22657     if (!cap_issubset(effective, permitted)) {
22658         goto out;
22659

```

```

22660     }
22661
22662     /* having verified that the proposed changes are
22663      * legal, we now put them into effect. */
22664     error = 0;
22665
22666     if (pid < 0) {
22667         if (pid == -1) /* all procs but current & init */
22668             cap_set_all(&effective,&inheritable,&permitted);
22669
22670         else /* all procs in process group */
22671             cap_set_pg(-pid, &effective, &inheritable,
22672                       &permitted);
22673         goto spin_out;
22674     } else {
22675         /* FIXME: do we need a write lock here..? */
22676         target->cap_effective = effective;
22677         target->cap_inheritable = inheritable;
22678         target->cap_permitted = permitted;
22679     }
22680
22681 out:
22682     if (target != current) {
22683         read_unlock(&tasklist_lock);
22684     }
22685 spin_out:
22686     spin_unlock(&task_capability_lock);
22687     return error;
22688 }

```

kernel/dma.c

```

22689 /* $Id: dma.c,v 1.7 1994/12/28 03:35:33 root Exp root $
22690 * linux/kernel/dma.c: A DMA channel allocator. Inspired
22691 * by linux/kernel/irq.c.
22692 *
22693 * Written by Hennus Bergman, 1992.
22694 *
22695 * 1994/12/26: Changes by Alex Nash to fix a minor bug in
22696 * /proc/dma. In the previous version the reported
22697 * device could end up being wrong, if a device requested
22698 * a DMA channel that was already in use. [It also
22699 * happened to remove the sizeof(char *) == sizeof(int)
22700 * assumption introduced because of those /proc/dma
22701 * patches. -- Hennus]
22702 */
22703
22704 #include <linux/kernel.h>

```

```

22705 #include <linux/errno.h>
22706 #include <asm/dma.h>
22707 #include <asm/system.h>
22708 #include <asm/spinlock.h>
22709
22710 /* A note on resource allocation:
22711 *
22712 * All drivers needing DMA channels, should allocate and
22713 * release them through the public routines
22714 * 'request_dma()' and 'free_dma()'.
22715 *
22716 * In order to avoid problems, all processes should
22717 * allocate resources in the same sequence and release
22718 * them in the reverse order.
22719 *
22720 * So, when allocating DMAs and IRQs, first allocate the
22721 * IRQ, then the DMA. When releasing them, first release
22722 * the DMA, then release the IRQ. If you don't, you may
22723 * cause allocation requests to fail unnecessarily. This
22724 * doesn't really matter now, but it will once we get
22725 * real semaphores in the kernel. */
22726
22727 spinlock_t dma_spin_lock = SPIN_LOCK_UNLOCKED;
22728
22729 /* Channel n is busy iff dma_chan_busy[n].lock != 0.
22730 * DMA0 used to be reserved for DRAM refresh, but
22731 * apparently not any more... DMA4 is reserved for
22732 * cascading. */
22733 struct dma_chan {
22734     int lock;
22735     const char *device_id;
22736 };
22737
22738 static struct dma_chan dma_chan_busy[MAX_DMA_CHANNELS] = {
22739     { 0, 0 },
22740     { 0, 0 },
22741     { 0, 0 },
22742     { 0, 0 },
22743     { 1, "cascade" },
22744     { 0, 0 },
22745     { 0, 0 },
22746     { 0, 0 }
22747 };
22748
22749 int get_dma_list(char *buf)
22750 {
22751     int i, len = 0;
22752

```

```

22753 for (i = 0 ; i < MAX_DMA_CHANNELS ; i++) {
22754     if (dma_chan_busy[i].lock) {
22755         len += sprintf(buf+len, "%2d: %s\n",
22756             i,
22757             dma_chan_busy[i].device_id);
22758     }
22759 }
22760 return len;
22761 } /* get_dma_list */
22762
22763
22764 int request_dma(unsigned int dmanr,
22765     const char * device_id)
22766 {
22767     if (dmanr >= MAX_DMA_CHANNELS)
22768         return -EINVAL;
22769
22770     if (xchg(&dma_chan_busy[dmanr].lock, 1) != 0)
22771         return -EBUSY;
22772
22773     dma_chan_busy[dmanr].device_id = device_id;
22774
22775     /* old flag was 0, now contains 1 to indicate busy */
22776     return 0;
22777 } /* request_dma */
22778
22779
22780 void free_dma(unsigned int dmanr)
22781 {
22782     if (dmanr >= MAX_DMA_CHANNELS) {
22783         printk("Trying to free DMA%d\n", dmanr);
22784         return;
22785     }
22786
22787     if (xchg(&dma_chan_busy[dmanr].lock, 0) == 0) {
22788         printk("Trying to free free DMA%d\n", dmanr);
22789         return;
22790     }
22791
22792 } /* free_dma */

```

kernel/exec_domain.c

```

22793 #include <linux/mm.h>
22794 #include <linux/smp_lock.h>
22795 #include <linux/module.h>
22796
22797 static asmlinkage void no_lcall7(struct pt_regs * regs);

```

```

22798
22799
22800 static unsigned long ident_map[32] = {
22801     0, 1, 2, 3, 4, 5, 6, 7,
22802     8, 9, 10, 11, 12, 13, 14, 15,
22803     16, 17, 18, 19, 20, 21, 22, 23,
22804     24, 25, 26, 27, 28, 29, 30, 31
22805 };
22806
22807 struct exec_domain default_exec_domain = {
22808     "Linux", /* name */
22809     no_lcall7, /* lcall7 causes a seg fault. */
22810     0, 0xff, /* All personalities. */
22811     ident_map, /* Identity map signals. */
22812     ident_map, /* - both ways. */
22813     NULL, /* No usage counter. */
22814     NULL /* Nothing after this in the list. */
22815 };
22816
22817 static struct exec_domain *exec_domains =
22818     &default_exec_domain;
22819
22820 static asmlinkage void no_lcall7(struct pt_regs * regs)
22821 {
22822
22823     /* This may have been a static linked SVr4 binary, so
22824     * we would have the personality set incorrectly.
22825     * Check to see whether SVr4 is available, and use it.
22826     * otherwise give the user a SEGV. */
22827     if (current->exec_domain &&
22828         current->exec_domain->module)
22829         __MOD_DEC_USE_COUNT(current->exec_domain->module);
22830
22831     current->personality = PER_SVR4;
22832     current->exec_domain =
22833         lookup_exec_domain(current->personality);
22834
22835     if (current->exec_domain &&
22836         current->exec_domain->module)
22837         __MOD_INC_USE_COUNT(current->exec_domain->module);
22838
22839     if (current->exec_domain &&
22840         current->exec_domain->handler &&
22841         current->exec_domain->handler != no_lcall7) {
22842         current->exec_domain->handler(regs);
22843         return;
22844     }
22845

```

```

22846 send_sig(SIGSEGV, current, 1);
22847 }
22848
22849 struct exec_domain *lookup_exec_domain(
22850 unsigned long personality)
22851 {
22852 unsigned long pers = personality & PER_MASK;
22853 struct exec_domain *it;
22854
22855 for (it=exec_domains; it; it=it->next)
22856     if (pers >= it->pers_low
22857         && pers <= it->pers_high)
22858         return it;
22859
22860 /* Should never get this far. */
22861 printk(KERN_ERR "No execution domain for personality "
22862         "0x%02lx\n", pers);
22863 return NULL;
22864 }
22865
22866 int register_exec_domain(struct exec_domain *it)
22867 {
22868     struct exec_domain *tmp;
22869
22870     if (!it)
22871         return -EINVAL;
22872     if (it->next)
22873         return -EBUSY;
22874     for (tmp=exec_domains; tmp; tmp=tmp->next)
22875         if (tmp == it)
22876             return -EBUSY;
22877     it->next = exec_domains;
22878     exec_domains = it;
22879     return 0;
22880 }
22881
22882 int unregister_exec_domain(struct exec_domain *it)
22883 {
22884     struct exec_domain ** tmp;
22885
22886     tmp = &exec_domains;
22887     while (*tmp) {
22888         if (it == *tmp) {
22889             *tmp = it->next;
22890             it->next = NULL;
22891             return 0;
22892         }
22893         tmp = &(*tmp)->next;

```

```

22894     }
22895     return -EINVAL;
22896 }
22897
22898 asmlinkage int sys_personality(unsigned long personality)
22899 {
22900     struct exec_domain *it;
22901     unsigned long old_personality;
22902     int ret;
22903
22904     lock_kernel();
22905     ret = current->personality;
22906     if (personality == 0xffffffff)
22907         goto out;
22908
22909     ret = -EINVAL;
22910     it = lookup_exec_domain(personality);
22911     if (!it)
22912         goto out;
22913
22914     old_personality = current->personality;
22915     if (current->exec_domain &&
22916         current->exec_domain->module)
22917         __MOD_DEC_USE_COUNT(current->exec_domain->module);
22918     current->personality = personality;
22919     current->exec_domain = it;
22920     if (current->exec_domain->module)
22921         __MOD_INC_USE_COUNT(current->exec_domain->module);
22922     ret = old_personality;
22923 out:
22924     unlock_kernel();
22925     return ret;
22926 }

```

kernel/exit.c

```

22927 /*
22928  * linux/kernel/exit.c
22929  *
22930  * Copyright (C) 1991, 1992 Linus Torvalds
22931  */
22932
22933 #include <linux/config.h>
22934 #include <linux/malloc.h>
22935 #include <linux/interrupt.h>
22936 #include <linux/smp_lock.h>
22937 #include <linux/module.h>
22938 #ifdef CONFIG_BSD_PROCESS_ACCT

```

```

22939 #include <linux/acct.h>
22940 #endif
22941
22942 #include <asm/uaccess.h>
22943 #include <asm/pgtable.h>
22944 #include <asm/mmu_context.h>
22945
22946 extern void sem_exit (void);
22947 extern struct task_struct *child_reaper;
22948
22949 int getrusage(struct task_struct *, int, struct rusage*);
22950
22951 static void release(struct task_struct * p)
22952 {
22953     if (p != current) {
22954 #ifdef __SMP__
22955         /* Wait to make sure the process isn't active on any
22956          * other CPU */
22957         for (;;) {
22958             int has_cpu;
22959             spin_lock(&scheduler_lock);
22960             has_cpu = p->has_cpu;
22961             spin_unlock(&scheduler_lock);
22962             if (!has_cpu)
22963                 break;
22964             do {
22965                 barrier();
22966             } while (p->has_cpu);
22967         }
22968 #endif
22969         free_uid(p);
22970         nr_tasks--;
22971         add_free_taskslot(p->tarray_ptr);
22972
22973         write_lock_irq(&tasklist_lock);
22974         unhash_pid(p);
22975         REMOVE_LINKS(p);
22976         write_unlock_irq(&tasklist_lock);
22977
22978         release_thread(p);
22979         current->cmin_flt += p->min_flt + p->cmin_flt;
22980         current->cmaj_flt += p->maj_flt + p->cmaj_flt;
22981         current->cnswap += p->nswap + p->cnswap;
22982         free_task_struct(p);
22983     } else {
22984         printk("task releasing itself\n");
22985     }
22986 }

```

```

22987
22988 /* This checks not only the pgrp, but falls back on the
22989 * pid if no satisfactory pgrp is found. I dunno - gdb
22990 * doesn't work correctly without this... */
22991 int session_of_pgrp(int pgrp)
22992 {
22993     struct task_struct *p;
22994     int fallback;
22995
22996     fallback = -1;
22997     read_lock(&tasklist_lock);
22998     for_each_task(p) {
22999         if (p->session <= 0)
23000             continue;
23001         if (p->pgrp == pgrp) {
23002             fallback = p->session;
23003             break;
23004         }
23005         if (p->pid == pgrp)
23006             fallback = p->session;
23007     }
23008     read_unlock(&tasklist_lock);
23009     return fallback;
23010 }
23011
23012 /* Determine if a process group is "orphaned", according
23013 * to the POSIX definition in 2.2.2.52. Orphaned process
23014 * groups are not to be affected by terminal-generated
23015 * stop signals. Newly orphaned process groups are to
23016 * receive a SIGHUP and a SIGCONT.
23017 *
23018 * "I ask you, have you ever known what it is to be an
23019 * orphan?" */
23020 static int will_become_orphaned_pgrp(int pgrp,
23021 struct task_struct * ignored_task)
23022 {
23023     struct task_struct *p;
23024
23025     read_lock(&tasklist_lock);
23026     for each task(p) {
23027         if ((p == ignored_task) || (p->pgrp != pgrp) ||
23028             (p->state == TASK_ZOMBIE) ||
23029             (p->p_pptr->pid == 1))
23030             continue;
23031         if ((p->p_pptr->pgrp != pgrp) &&
23032             (p->p_pptr->session == p->session)) {
23033             read_unlock(&tasklist_lock);
23034             return 0;

```

```

23035     }
23036 }
23037 read_unlock(&tasklist_lock);
23038 return 1;      /* (sighing) "Often!" */
23039 }
23040
23041 int is_orphaned_pgrp(int pgrp)
23042 {
23043     return will_become_orphaned_pgrp(pgrp, 0);
23044 }
23045
23046 static inline int has_stopped_jobs(int pgrp)
23047 {
23048     int retval = 0;
23049     struct task_struct * p;
23050
23051     read_lock(&tasklist_lock);
23052     for_each_task(p) {
23053         if (p->pgrp != pgrp)
23054             continue;
23055         if (p->state != TASK_STOPPED)
23056             continue;
23057         retval = 1;
23058         break;
23059     }
23060     read_unlock(&tasklist_lock);
23061     return retval;
23062 }
23063
23064 static inline void forget_original_parent(
23065     struct task_struct * father)
23066 {
23067     struct task_struct * p;
23068
23069     read_lock(&tasklist_lock);
23070     for_each_task(p) {
23071         if (p->p_opptr == father) {
23072             p->exit_signal = SIGCHLD;
23073             p->p_opptr = child_reaper; /* init */
23074             if (p->pdeath_signal)
23075                 send_sig(p->pdeath_signal, p, 0);
23076         }
23077     }
23078     read_unlock(&tasklist_lock);
23079 }
23080
23081 static inline void close_files(
23082     struct files_struct * files)

```

```

23083 {
23084     int i, j;
23085
23086     j = 0;
23087     for (;;) {
23088         unsigned long set = files->open_fds.fds_bits[j];
23089         i = j * __NFDBITS;
23090         j++;
23091         if (i >= files->max_fds)
23092             break;
23093         while (set) {
23094             if (set & 1) {
23095                 struct file * file = files->fd[i];
23096                 if (file) {
23097                     files->fd[i] = NULL;
23098                     filp_close(file, files);
23099                 }
23100             }
23101             i++;
23102             set >>= 1;
23103         }
23104     }
23105 }
23106
23107 extern kmem_cache_t *files_cachep;
23108
23109 static inline void __exit_files(struct task_struct *tsk)
23110 {
23111     struct files_struct * files = tsk->files;
23112
23113     if (files) {
23114         tsk->files = NULL;
23115         if (atomic_dec_and_test(&files->count)) {
23116             close_files(files);
23117             /* Free the fd array as appropriate ... */
23118             if (NR_OPEN * sizeof(struct file *) == PAGE_SIZE)
23119                 free_page((unsigned long) files->fd);
23120             else
23121                 kfree(files->fd);
23122             kmem_cache_free(files_cachep, files);
23123         }
23124     }
23125 }
23126
23127 void exit_files(struct task_struct *tsk)
23128 {
23129     __exit_files(tsk);
23130 }

```



```

23131
23132 static inline void __exit_fs(struct task_struct *tsk)
23133 {
23134     struct fs_struct * fs = tsk->fs;
23135
23136     if (fs) {
23137         tsk->fs = NULL;
23138         if (atomic_dec_and_test(&fs->count)) {
23139             dput(fs->root);
23140             dput(fs->pwd);
23141             kfree(fs);
23142         }
23143     }
23144 }
23145
23146 void exit_fs(struct task_struct *tsk)
23147 {
23148     __exit_fs(tsk);
23149 }
23150
23151 static inline void __exit_sighand(
23152     struct task_struct *tsk)
23153 {
23154     struct signal_struct * sig = tsk->sig;
23155
23156     if (sig) {
23157         unsigned long flags;
23158
23159         spin_lock_irqsave(&tsk->sigmask_lock, flags);
23160         tsk->sig = NULL;
23161         spin_unlock_irqrestore(&tsk->sigmask_lock, flags);
23162         if (atomic_dec_and_test(&sig->count))
23163             kfree(sig);
23164     }
23165
23166     flush_signals(tsk);
23167 }
23168
23169 void exit_sighand(struct task_struct *tsk)
23170 {
23171     __exit_sighand(tsk);
23172 }
23173
23174 static inline void __exit_mm(struct task_struct * tsk)
23175 {
23176     struct mm_struct * mm = tsk->mm;
23177
23178     /* Set us up to use the kernel mm state */

```

```

23179     if (mm != &init_mm) {
23180         flush_cache_mm(mm);
23181         flush_tlb_mm(mm);
23182         destroy_context(mm);
23183         tsk->mm = &init_mm;
23184         tsk->swappable = 0;
23185         SET_PAGE_DIR(tsk, swapper_pg_dir);
23186         mm_release();
23187         mmput(mm);
23188     }
23189 }
23190
23191 void exit_mm(struct task_struct *tsk)
23192 {
23193     __exit_mm(tsk);
23194 }
23195
23196 /* Send signals to all our closest relatives so that they
23197 * know to properly mourn us.. */
23198 static void exit_notify(void)
23199 {
23200     struct task_struct * p;
23201
23202     forget_original_parent(current);
23203     /* Check to see if any process groups have become
23204     * orphaned as a result of our exiting, and if they
23205     * have any stopped jobs, send them a SIGHUP and then a
23206     * SIGCONT. (POSIX 3.2.2.2)
23207     *
23208     * Case i: Our father is in a different pgrp than we
23209     * are and we were the only connection outside, so our
23210     * pgrp is about to become orphaned. */
23211     if ((current->p_pptr->pgrp != current->pgrp) &&
23212         (current->p_pptr->session == current->session) &&
23213         will_become_orphaned_pgrp(current->pgrp, current) &&
23214         has_stopped_jobs(current->pgrp)) {
23215         kill_pg(current->pgrp, SIGHUP, 1);
23216         kill_pg(current->pgrp, SIGCONT, 1);
23217     }
23218
23219     /* Let father know we died */
23220     notify_parent(current, current->exit_signal);
23221
23222     /* This loop does two things:
23223     *
23224     * A. Make init inherit all the child processes
23225     * B. Check to see if any process groups have become
23226     * orphaned as a result of our exiting, and if they

```

```

23227 * have any stopped jobs, send them a SIGHUP and then a
23228 * SIGCONT. (POSIX 3.2.2.2) */
23229
23230 write_lock_irq(&tasklist_lock);
23231 while (current->p_cptr != NULL) {
23232     p = current->p_cptr;
23233     current->p_cptr = p->p_osptr;
23234     p->p_ysptr = NULL;
23235     p->flags &= ~(PF_PTRACED|PF_TRACESYS);
23236
23237     p->p_pptr = p->p_opptr;
23238     p->p_osptr = p->p_pptr->p_cptr;
23239     if (p->p_osptr)
23240         p->p_osptr->p_ysptr = p;
23241     p->p_pptr->p_cptr = p;
23242     if (p->state == TASK_ZOMBIE)
23243         notify_parent(p, p->exit_signal);
23244     /* process group orphan check
23245     * Case ii: Our child is in a different pgrp than we
23246     * are, and it was the only connection outside, so
23247     * the child pgrp is now orphaned. */
23248     if ((p->pgrp != current->pgrp) &&
23249         (p->session == current->session)) {
23250         int pgrp = p->pgrp;
23251
23252         write_unlock_irq(&tasklist_lock);
23253         if (is_orphaned_pgrp(pgrp) &&
23254             has_stopped_jobs(pgrp)) {
23255             kill_pg(pgrp, SIGHUP, 1);
23256             kill_pg(pgrp, SIGCONT, 1);
23257         }
23258         write_lock_irq(&tasklist_lock);
23259     }
23260 }
23261 write_unlock_irq(&tasklist_lock);
23262
23263 if (current->leader)
23264     disassociate_ctty(1);
23265 }
23266
23267 NORET_TYPE void do_exit(long code)
23268 {
23269     struct task_struct *tsk = current;
23270
23271     if (in_interrupt())
23272         printk("Aiee, killing interrupt handler\n");
23273     if (!tsk->pid)
23274         panic("Attempted to kill the idle task!");
23275     tsk->flags |= PF_EXITING;
23276     start_bh_atomic();
23277     del_timer(&tsk->real_timer);
23278     end_bh_atomic();
23279
23280     lock_kernel();
23281     fake_volatile:
23282     #ifdef CONFIG_BSD_PROCESS_ACCT
23283     acct_process(code);
23284     #endif
23285     sem_exit();
23286     __exit_mm(tsk);
23287     #if CONFIG_API000
23288     exit_msc(tsk);
23289     #endif
23290     __exit_files(tsk);
23291     __exit_fs(tsk);
23292     __exit_sighand(tsk);
23293     exit_thread();
23294     tsk->state = TASK_ZOMBIE;
23295     tsk->exit_code = code;
23296     exit_notify();
23297     #ifdef DEBUG_PROC_TREE
23298     audit_ptree();
23299     #endif
23300     if (tsk->exec_domain && tsk->exec_domain->module)
23301         __MOD_DEC_USE_COUNT(tsk->exec_domain->module);
23302     if (tsk->binfmt && tsk->binfmt->module)
23303         __MOD_DEC_USE_COUNT(tsk->binfmt->module);
23304     schedule();
23305     /* In order to get rid of the "volatile function does
23306     * return" message I did this little loop that confuses
23307     * gcc to think do_exit really is volatile. In fact it's
23308     * schedule() that is volatile in some circumstances:
23309     * when current->state = ZOMBIE, schedule() never
23310     * returns.
23311     */
23312     /* In fact the natural way to do all this is to have the
23313     * label and the goto right after each other, but I put
23314     * the fake_volatile label at the start of the function
23315     * just in case something /really/ bad happens, and the
23316     * schedule returns. This way we can try again. I'm not
23317     * paranoid: it's just that everybody is out to get me.
23318     */
23319     goto fake_volatile;
23320 }
23321
23322 asmlinkage int sys_exit(int error_code)

```

→ 23267

```

23323 (
23324 do_exit((error_code&0xff)<<8);
23325 )
23326
23327 asmlinkage int sys_wait4(pid_t pid,
23328 unsigned int * stat_addr, int options,
23329 struct rusage * ru)
23330 {
23331 int flag, retval;
23332 struct wait_queue wait = { current, NULL };
23333 struct task_struct *p;
23334
23335 if (options & ~(WNOHANG|WUNTRACED|__WCLONE))
23336 return -EINVAL;
23337
23338 add_wait_queue(&current->wait_chldexit,&wait);
23339 repeat:
23340 flag = 0;
23341 read_lock(&tasklist_lock);
23342 for (p = current->p_cptr ; p ; p = p->p_osptr) {
23343 if (pid>0) {
23344 if (p->pid != pid)
23345 continue;
23346 } else if (!pid) {
23347 if (p->pgrp != current->pgrp)
23348 continue;
23349 } else if (pid != -1) {
23350 if (p->pgrp != -pid)
23351 continue;
23352 }
23353 /* wait for cloned processes iff the __WCLONE flag is
23354 * set */
23355 if ((p->exit_signal != SIGCHLD) ^
23356 ((options & __WCLONE) != 0))
23357 continue;
23358 flag = 1;
23359 switch (p->state) {
23360 case TASK_STOPPED:
23361 if (!p->exit_code)
23362 continue;
23363 if (!(options & WUNTRACED) &&
23364 !(p->flags & PF_PTRACED))
23365 continue;
23366 read_unlock(&tasklist_lock);
23367 retval = ru ? getrusage(p, RUSAGE_BOTH, ru) : 0;
23368 if (!retval && stat_addr)
23369 retval = put_user((p->exit_code << 8) | 0x7f,
23370 stat_addr);

```

→ p 532

→ p 533

```

23371 if (!retval) {
23372 p->exit_code = 0;
23373 retval = p->pid;
23374 }
23375 goto end_wait4;
23376 case TASK_ZOMBIE:
23377 current->times.tms_cutime += p->times.tms_utime +
23378 p->times.tms_cutime;
23379 current->times.tms_cstime += p->times.tms_stime +
23380 p->times.tms_cstime;
23381 read_unlock(&tasklist_lock);
23382 retval = ru ? getrusage(p, RUSAGE_BOTH, ru) : 0;
23383 if (!retval && stat_addr)
23384 retval = put_user(p->exit_code, stat_addr);
23385 if (retval)
23386 goto end_wait4;
23387 retval = p->pid;
23388 if (p->p_opptr != p->p_pptr) {
23389 write_lock_irq(&tasklist_lock);
23390 REMOVE_LINKS(p);
23391 p->p_pptr = p->p_opptr;
23392 SET_LINKS(p);
23393 write_unlock_irq(&tasklist_lock);
23394 notify_parent(p, SIGCHLD);
23395 } else
23396 release(p);
23397 #ifdef DEBUG_PROC_TREE
23398 audit_ptree();
23399 #endif
23400 goto end_wait4;
23401 default:
23402 continue;
23403 }
23404 }
23405 read_unlock(&tasklist_lock);
23406 if (flag) {
23407 retval = 0;
23408 if (options & WNOHANG)
23409 goto end_wait4;
23410 retval = -ERESTARTSYS;
23411 if (signal_pending(current))
23412 goto end_wait4;
23413 current->state=TASK_INTERRUPTIBLE;
23414 schedule();
23415 goto repeat;
23416 }
23417 retval = -ECHILD;
23418 end_wait4:

```

```

23419 remove_wait_queue(&current->wait_chldexit,&wait);
23420 return retval;
23421 }
23422
23423 #ifndef __alpha__
23424
23425 /* sys_waitpid() remains for compatibility. waitpid()
23426 * should be implemented by calling sys_wait4() from
23427 * libc.a. */
23428 asmlinkage int sys_waitpid(pid_t pid,
23429 unsigned int * stat_addr, int options)
23430 {
23431 return sys_wait4(pid, stat_addr, options, NULL);
23432 }
23433
23434 #endif

```

kernel/fork.c

```

23435 /*
23436 * linux/kernel/fork.c
23437 *
23438 * Copyright (C) 1991, 1992 Linus Torvalds
23439 */
23440
23441 /* 'fork.c' contains the help-routines for the 'fork'
23442 * system call (see also system_call.s). Fork is rather
23443 * simple, once you get the hang of it, but the memory
23444 * management can be a bitch. See 'mm/mm.c':
23445 * 'copy_page_tables()' */
23446
23447 #include <linux/malloc.h>
23448 #include <linux/init.h>
23449 #include <linux/unistd.h>
23450 #include <linux/smp_lock.h>
23451 #include <linux/module.h>
23452 #include <linux/vmalloc.h>
23453
23454 #include <asm/pgtable.h>
23455 #include <asm/mmu_context.h>
23456 #include <asm/uaccess.h>
23457
23458 /* The idle tasks do not count.. */
23459 int nr_tasks=0;
23460 int nr_running=0;
23461
23462 /* Handle normal Linux uptimes. */
23463 unsigned long int total_forks=0;

```

```

23464 int last_pid=0;
23465
23466 /* SLAB cache for mm_struct's. */
23467 kmem_cache_t *mm_cachep;
23468
23469 /* SLAB cache for files structs */
23470 kmem_cache_t *files_cachep;
23471
23472 struct task_struct *pidhash[PIDHASH_SZ];
23473
23474 struct task_struct **tarray_freelist = NULL;
23475 spinlock_t taskslot_lock = SPIN_LOCK_UNLOCKED;
23476
23477 /* UID task count cache, to prevent walking entire
23478 * process list every single fork() operation. */
23479 #define UIDHASH_SZ (PIDHASH_SZ >> 2)
23480
23481 static struct user_struct {
23482 atomic_t count;
23483 struct user_struct *next, **pprev;
23484 unsigned int uid;
23485 } *uidhash[UIDHASH_SZ];
23486
23487 spinlock_t uidhash_lock = SPIN_LOCK_UNLOCKED;
23488
23489 kmem_cache_t *uid_cachep;
23490
23491 #define uidhashfn(uid)
23492 (((uid >> 8) ^ uid) & (UIDHASH_SZ - 1))
23493
23494 /* These routines must be called with the uidhash
23495 * spinlock held! */
23496 static inline void uid_hash_insert(
23497 struct user_struct *up, unsigned int hashent)
23498 {
23499 if ((up->next = uidhash[hashent]) != NULL)
23500 uidhash[hashent]->pprev = &up->next;
23501 up->pprev = &uidhash[hashent];
23502 uidhash[hashent] = up;
23503 }
23504
23505 static inline void uid_hash_remove(
23506 struct user_struct *up)
23507 {
23508 if (up->next)
23509 up->next->pprev = up->pprev;
23510 *up->pprev = up->next;
23511 }

```

```

23512
23513 static inline struct user_struct *uid_hash_find(
23514     unsigned short uid, unsigned int hashent)
23515 {
23516     struct user_struct *up, *next;
23517
23518     next = uidhash[hashent];
23519     for (;;) {
23520         up = next;
23521         if (next) {
23522             next = up->next;
23523             if (up->uid != uid)
23524                 continue;
23525             atomic_inc(&up->count);
23526         }
23527         break;
23528     }
23529     return up;
23530 }
23531
23532 void free_uid(struct task_struct *p)
23533 {
23534     struct user_struct *up = p->user;
23535
23536     if (up) {
23537         p->user = NULL;
23538         if (atomic_dec_and_test(&up->count)) {
23539             spin_lock(&uidhash_lock);
23540             uid_hash_remove(up);
23541             spin_unlock(&uidhash_lock);
23542             kmem_cache_free(uid_cachep, up);
23543         }
23544     }
23545 }
23546
23547 int alloc_uid(struct task_struct *p)
23548 {
23549     unsigned int hashent = uidhashfn(p->uid);
23550     struct user_struct *up;
23551
23552     spin_lock(&uidhash_lock);
23553     up = uid_hash_find(p->uid, hashent);
23554     spin_unlock(&uidhash_lock);
23555
23556     if (!up) {
23557         struct user_struct *new;
23558
23559         new = kmem_cache_alloc(uid_cachep, SLAB_KERNEL);

```

```

23560     if (!new)
23561         return -EAGAIN;
23562     new->uid = p->uid;
23563     atomic_set(&new->count, 1);
23564
23565     /* Before adding this, check whether we raced on
23566      * adding the same user already.. */
23567     spin_lock(&uidhash_lock);
23568     up = uid_hash_find(p->uid, hashent);
23569     if (up) {
23570         kmem_cache_free(uid_cachep, new);
23571     } else {
23572         uid_hash_insert(new, hashent);
23573         up = new;
23574     }
23575     spin_unlock(&uidhash_lock);
23576
23577 }
23578 p->user = up;
23579 return 0;
23580 }
23581
23582 void __init uidcache_init(void)
23583 {
23584     int i;
23585
23586     uid_cachep =
23587         kmem_cache_create("uid_cache",
23588             sizeof(struct user_struct),
23589             0, SLAB_HWCACHE_ALIGN, NULL, NULL);
23590     if (!uid_cachep)
23591         panic("Cannot create uid taskcount SLAB cache\n");
23592
23593     for (i = 0; i < UIDHASH_SZ; i++)
23594         uidhash[i] = 0;
23595 }
23596
23597 static inline struct task_struct **
23598 find_empty_process(void)
23599 {
23600     struct task_struct **tslot = NULL;
23601
23602     if ((nr_tasks < NR_TASKS - MIN_TASKS_LEFT_FOR_ROOT) ||
23603         !current->uid)
23604         tslot = get_free_taskslot();
23605     return tslot;
23606 }
23607

```

```

23608 /* Protects next_safe and last_pid. */
23609 spinlock_t lastpid_lock = SPIN_LOCK_UNLOCKED;
23610
23611 static int get_pid(unsigned long flags)
23612 {
23613     static int next_safe = PID_MAX;
23614     struct task_struct *p;
23615
23616     if (flags & CLONE_PID)
23617         return current->pid;
23618
23619     spin_lock(&lastpid_lock);
23620     if(++last_pid & 0xffff8000) {
23621         last_pid = 300;          /* Skip daemons etc. */
23622         goto inside;
23623     }
23624     if(last_pid >= next_safe) {
23625     inside:
23626         next_safe = PID_MAX;
23627         read_lock(&tasklist_lock);
23628         repeat:
23629         for_each_task(p) {
23630             if(p->pid == last_pid ||
23631                p->pgrp == last_pid ||
23632                p->session == last_pid) {
23633                 if(++last_pid >= next_safe) {
23634                     if(last_pid & 0xffff8000)
23635                         last_pid = 300;
23636                     next_safe = PID_MAX;
23637                 }
23638                 goto repeat;
23639             }
23640             if(p->pid > last_pid && next_safe > p->pid)
23641                 next_safe = p->pid;
23642             if(p->pgrp > last_pid && next_safe > p->pgrp)
23643                 next_safe = p->pgrp;
23644             if(p->session > last_pid && next_safe > p->session)
23645                 next_safe = p->session;
23646         }
23647         read_unlock(&tasklist_lock);
23648     }
23649     spin_unlock(&lastpid_lock);
23650
23651     return last_pid;
23652 }
23653
23654 static inline int dup_mmap(struct mm_struct * mm)
23655 {

```

```

23656     struct vm_area_struct * mpnt, *tmp, **pprev;
23657     int retval;
23658
23659     flush_cache_mm(current->mm);
23660     pprev = &mm->mmap;
23661     for (mpnt = current->mm->mmap; mpnt;
23662          mpnt = mpnt->vm_next) {
23663         struct file *file;
23664
23665         retval = -ENOMEM;
23666         tmp = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
23667         if (!tmp)
23668             goto fail_nomem;
23669         *tmp = *mpnt;
23670         tmp->vm_flags &= ~VM_LOCKED;
23671         tmp->vm_mm = mm;
23672         mm->map_count++;
23673         tmp->vm_next = NULL;
23674         file = tmp->vm_file;
23675         if (file) {
23676             file->f_count++;
23677             if (tmp->vm_flags & VM_DENYWRITE)
23678                 file->f_dentry->d_inode->i_writecount--;
23679
23680             /*insert tmp into the share list, just after mpnt*/
23681             if ((tmp->vm_next_share = mpnt->vm_next_share) !=
23682                 NULL)
23683                 mpnt->vm_next_share->vm_pprev_share =
23684                     &tmp->vm_next_share;
23685             mpnt->vm_next_share = tmp;
23686             tmp->vm_pprev_share = &mpnt->vm_next_share;
23687         }
23688
23689         /* Copy the pages, but defer checking for errors */
23690         retval = copy_page_range(mm, current->mm, tmp);
23691         if (!retval && tmp->vm_ops && tmp->vm_ops->open)
23692             tmp->vm_ops->open(tmp);
23693
23694         /* Link in the new vma even if an error occurred, so
23695          * that exit_mmap() can clean up the mess. */
23696         tmp->vm_next = *pprev;
23697         *pprev = tmp;
23698
23699         pprev = &tmp->vm_next;
23700         if (retval)
23701             goto fail_nomem;
23702     }
23703     retval = 0;

```

```

23704 if (mm->map_count >= AVL_MIN_MAP_COUNT)
23705     build_mmap_avl(mm);
23706
23707 fail_nomem:
23708 flush_tlb_mm(current->mm);
23709 return retval;
23710 }
23711
23712 /* Allocate and initialize an mm_struct.
23713 *
23714 * NOTE! The mm mutex will be locked until the caller
23715 * decides that all systems are go.. */
23716 struct mm_struct * mm_alloc(void)
23717 {
23718     struct mm_struct * mm;
23719
23720     mm = kmem_cache_alloc(mm_cachep, SLAB_KERNEL);
23721     if (mm) {
23722         *mm = *current->mm;
23723         init_new_context(mm);
23724         atomic_set(&mm->count, 1);
23725         mm->map_count = 0;
23726         mm->def_flags = 0;
23727         mm->mmap_sem = MUTEX_LOCKED;
23728         /* Leave mm->pgd set to the parent's pgd so that
23729          * pgd_offset() is always valid. */
23730         mm->mmap = mm->mmap_avl = mm->mmap_cache = NULL;
23731
23732         /* It has not run yet, so cannot be present in
23733          * anyone's cache or tlb. */
23734         mm->cpu_vm_mask = 0;
23735     }
23736     return mm;
23737 }
23738
23739 /* Please note the differences between mmput and
23740 * mm_release. mmput is called whenever we stop holding
23741 * onto a mm_struct, error success whatever.
23742 *
23743 * mm_release is called after a mm_struct has been
23744 * removed from the current process.
23745 *
23746 * This difference is important for error handling, when
23747 * we only half set up a mm_struct for a new process and
23748 * need to restore the old one. Because we mmput the new
23749 * mm_struct before restoring the old one. . . Eric
23750 * Biederman 10 January 1998 */
23751 void mm_release(void)

```

```

23752 {
23753     struct task_struct *tsk = current;
23754     forget_segments();
23755     /* notify parent sleeping on vfork() */
23756     if (tsk->flags & PF_VFORK) {
23757         tsk->flags &= ~PF_VFORK;
23758         up(tsk->p_opptr->vfork_sem);
23759     }
23760 }
23761
23762 /* Decrement the use count and release all resources for
23763 * an mm. */
23764 void mmput(struct mm_struct *mm)
23765 {
23766     if (atomic_dec_and_test(&mm->count)) {
23767         release_segments(mm);
23768         exit_mmap(mm);
23769         free_page_tables(mm);
23770         kmem_cache_free(mm_cachep, mm);
23771     }
23772 }
23773
23774 static inline int copy_mm(int nr,
23775 unsigned long clone_flags, struct task_struct * tsk)
23776 {
23777     struct mm_struct * mm;
23778     int retval;
23779
23780     if (clone_flags & CLONE_VM) {
23781         mmget(current->mm);
23782         /* Set up the LDT descriptor for the clone task. */
23783         copy_segments(nr, tsk, NULL);
23784         SET_PAGE_DIR(tsk, current->mm->pgd);
23785         return 0;
23786     }
23787
23788     retval = -ENOMEM;
23789     mm = mm_alloc();
23790     if (!mm)
23791         goto fail_nomem;
23792
23793     tsk->mm = mm;
23794     tsk->min_flt = tsk->maj_flt = 0;
23795     tsk->cmin_flt = tsk->cmaj_flt = 0;
23796     tsk->nswap = tsk->cnsnap = 0;
23797     copy_segments(nr, tsk, mm);
23798     retval = new_page_tables(tsk);
23799     if (retval)

```

```

23800     goto free_mm;
23801     retval = dup_mmap(mm);
23802     if (retval)
23803         goto free_pt;
23804     up(&mm->mmap_sem);
23805     return 0;
23806
23807     free_mm:
23808     mm->pgd = NULL;
23809     free_pt:
23810     tsk->mm = NULL;
23811     mmput(mm);
23812     fail_nomem:
23813     return retval;
23814 }
23815
23816 static inline int copy_fs(unsigned long clone_flags,
23817                          struct task_struct * tsk)
23818 {
23819     if (clone_flags & CLONE_FS) {
23820         atomic_inc(&current->fs->count);
23821         return 0;
23822     }
23823     tsk->fs = kcalloc(sizeof(*tsk->fs), GFP_KERNEL);
23824     if (!tsk->fs)
23825         return -1;
23826     atomic_set(&tsk->fs->count, 1);
23827     tsk->fs->umask = current->fs->umask;
23828     tsk->fs->root = dget(current->fs->root);
23829     tsk->fs->pwd = dget(current->fs->pwd);
23830     return 0;
23831 }
23832
23833 /* Copy a fd_set and compute the maximum fd it contains.
23834 */
23835 static inline int __copy_fdset(unsigned long *d,
23836                               unsigned long *src)
23837 {
23838     int i;
23839     unsigned long *p = src;
23840     unsigned long *max = src;
23841
23842     for (i = __FDSET_LONGS; i; --i) {
23843         if ((*d++ = *p++) != 0)
23844             max = p;
23845     }
23846     return (max - src)*sizeof(long)*8;
23847 }

```

```

23848
23849 static inline int copy_fdset(fd_set *dst, fd_set *src)
23850 {
23851     return __copy_fdset(dst->fds_bits, src->fds_bits);
23852 }
23853
23854 static int copy_files(unsigned long clone_flags,
23855                      struct task_struct * tsk)
23856 {
23857     struct files_struct *oldf, *newf;
23858     struct file **old_fds, **new_fds;
23859     int size, i, error = 0;
23860
23861     /* A background process may not have any files ... */
23862     oldf = current->files;
23863     if (!oldf)
23864         goto out;
23865
23866     if (clone_flags & CLONE_FILES) {
23867         atomic_inc(&oldf->count);
23868         goto out;
23869     }
23870
23871     tsk->files = NULL;
23872     error = -ENOMEM;
23873     newf = kmem_cache_alloc(files_cachep, SLAB_KERNEL);
23874     if (!newf)
23875         goto out;
23876
23877     /* Allocate the fd array, using get_free_page() if
23878      * possible. Eventually we want to make the array size
23879      * variable ... */
23880     size = NR_OPEN * sizeof(struct file *);
23881     if (size == PAGE_SIZE)
23882         new_fds =
23883             (struct file **) __get_free_page(GFP_KERNEL);
23884     else
23885         new_fds = (struct file **) kcalloc(size, GFP_KERNEL);
23886     if (!new_fds)
23887         goto out_release;
23888
23889     atomic_set(&newf->count, 1);
23890     newf->max_fds = NR_OPEN;
23891     newf->fd = new_fds;
23892     newf->close_on_exec = oldf->close_on_exec;
23893     i = copy_fdset(&newf->open_fds, &oldf->open_fds);
23894
23895     old_fds = oldf->fd;

```



```

23896 for (; i != 0; i--) {
23897     struct file *f = *old_fds++;
23898     *new_fds = f;
23899     if (f)
23900         f->f_count++;
23901     new_fds++;
23902 }
23903 /* This is long word aligned thus could use a optimized
23904  * version */
23905 memset(new_fds, 0,
23906        (char *)newf->fd + size * (char *)new_fds);
23907
23908 tsk->files = newf;
23909 error = 0;
23910 out:
23911 return error;
23912
23913 out_release:
23914 kmem_cache_free(files_cachep, newf);
23915 goto out;
23916 }
23917
23918 static inline int copy_sighand(unsigned long clone_flags,
23919                               struct task_struct * tsk)
23920 {
23921     if (clone_flags & CLONE_SIGHAND) {
23922         atomic_inc(&current->sig->count);
23923         return 0;
23924     }
23925     tsk->sig = kmalloc(sizeof(*tsk->sig), GFP_KERNEL);
23926     if (!tsk->sig)
23927         return -1;
23928     spin_lock_init(&tsk->sig->siglock);
23929     atomic_set(&tsk->sig->count, 1);
23930     memcpy(tsk->sig->action, current->sig->action,
23931           sizeof(tsk->sig->action));
23932     return 0;
23933 }
23934
23935 static inline void copy_flags(unsigned long clone_flags,
23936                              struct task_struct *p)
23937 {
23938     unsigned long new_flags = p->flags;
23939
23940     new_flags &= ~(PF_SUPERPRIV | PF_USEDFPU | PF_VFORK);
23941     new_flags |= PF_FORKNOEXEC;
23942     if (!(clone_flags & CLONE_PTRACE))
23943         new_flags &= ~(PF_PTRACED|PF_TRACESYS);

```

→ p 512

```

23944     if (clone_flags & CLONE_VFORK)
23945         new_flags |= PF_VFORK;
23946     p->flags = new_flags;
23947 }
23948
23949 /* Ok, this is the main fork-routine. It copies the
23950  * system process information (task[nr]) and sets up the
23951  * necessary registers. It also copies the data segment
23952  * in its entirety. */
23953 int do_fork(unsigned long clone_flags, unsigned long usp,
23954            struct pt_regs *regs)
23955 {
23956     int nr;
23957     int retval = -ENOMEM;
23958     struct task_struct *p;
23959     struct semaphore sem = MUTEX_LOCKED;
23960
23961     current->vfork sem = &sem;
23962
23963     p = alloc_task_struct();
23964     if (!p)
23965         goto fork_out;
23966
23967     *p = *current;
23968
23969     down(&current->mm->mmap_sem);
23970     lock_kernel();
23971
23972     retval = -EAGAIN;
23973     if (p->user) {
23974         if (atomic_read(&p->user->count) >=
23975             p->rlim[RLIMIT_NPROC].rlim_cur)
23976             goto bad_fork_free;
23977     }
23978
23979     {
23980         struct task_struct **tslot;
23981         tslot = find_empty_process();
23982         if (!tslot)
23983             goto bad_fork_free;
23984         p->tarray_ptr = tslot;
23985         *tslot = p;
23986         nr = tslot - &task[0];
23987     }
23988
23989     if (p->exec_domain && p->exec_domain->module)
23990         __MOD_INC_USE_COUNT(p->exec_domain->module);
23991     if (p->binfmt && p->binfmt->module)

```

```

23992     __MOD_INC_USE_COUNT(p->binfmt->module);
23993
23994     p->did_exec = 0;
23995     p->swappable = 0;
23996     p->state = TASK_UNINTERRUPTIBLE;
23997
23998     copy_flags(clone_flags, p);
23999     p->pid = get_pid(clone_flags);
24000
24001     /* This is a "shadow run" state. The process is marked
24002      * runnable, but isn't actually on any run queue
24003      * yet.. (that happens at the very end). */
24004     p->state = TASK_RUNNING;
24005     p->next_run = p;
24006     p->prev_run = p;
24007
24008     p->p_pptr = p->p_opptr = current;
24009     p->p_cpnr = NULL;
24010     init_waitqueue(&p->wait_chldexit);
24011     p->vfork_sem = NULL;
24012
24013     p->sigpending = 0;
24014     sigemptyset(&p->signal);
24015     p->sigqueue = NULL;
24016     p->sigqueue_tail = &p->sigqueue;
24017
24018     p->it_real_value = p->it_virt_value = p->it_prof_value
24019     = 0;
24020     p->it_real_incr = p->it_virt_incr = p->it_prof_incr
24021     = 0;
24022     init_timer(&p->real_timer);
24023     p->real_timer.data = (unsigned long) p;
24024
24025     p->leader = 0; /* session leadership doesn't inherit */
24026     p->tty_old_pgrp = 0;
24027     p->times.tms_utime = p->times.tms_stime = 0;
24028     p->times.tms_cutime = p->times.tms_cstime = 0;
24029     #ifdef __SMP__
24030     {
24031         int i;
24032         p->has_cpu = 0;
24033         p->processor = NO_PROC_ID;
24034         /* ?? should we just memset this ?? */
24035         for(i = 0; i < smp_num_cpus; i++)
24036             p->per_cpu_utime[i] = p->per_cpu_stime[i] = 0;
24037         spin_lock_init(&p->sigmask_lock);
24038     }
24039     #endif
24040     p->lock_depth = -1; /* -1 = no lock */
24041     p->start_time = jiffies;
24042
24043     retval = -ENOMEM;
24044     /* copy all the process information */
24045     if (copy_files(clone_flags, p))
24046         goto bad_fork_cleanup;
24047     if (copy_fs(clone_flags, p))
24048         goto bad_fork_cleanup_files;
24049     if (copy_sighand(clone_flags, p))
24050         goto bad_fork_cleanup_fs;
24051     if (copy_mm(nr, clone_flags, p))
24052         goto bad_fork_cleanup_sighand;
24053     retval = copy_thread(nr, clone_flags, usp, p, regs);
24054     if (retval)
24055         goto bad_fork_cleanup_sighand;
24056     p->semundo = NULL;
24057
24058     /* ok, now we should be set up.. */
24059     p->swappable = 1;
24060     p->exit_signal = clone_flags & CSIGNAL;
24061     p->pdeath_signal = 0;
24062
24063     /* "share" dynamic priority between parent and child,
24064      * thus the total amount of dynamic priorities in the
24065      * system doesn't change, more scheduling fairness. This
24066      * is only important in the first timeslice, on the
24067      * long run the scheduling behaviour is unchanged. */
24068     current->counter >>= 1;
24069     p->counter = current->counter;
24070
24071     /* OK, add it to the run-queues and make it visible to
24072      * the rest of the system.
24073      *
24074      * Let it rip! */
24075     retval = p->pid;
24076     if (retval) {
24077         write_lock_irq(&tasklist_lock);
24078         SET_LINKS(p);
24079         hash_pid(p);
24080         write_unlock_irq(&tasklist_lock);
24081
24082         nr_tasks++;
24083         if (p->user)
24084             atomic_inc(&p->user->count);
24085
24086         p->next_run = NULL;
24087         p->prev_run = NULL;

```

```

24088     wake_up_process(p);           /* do this last */
24089 }
24090 ++total_forks;
24091 bad_fork:
24092     unlock_kernel();
24093     up(&current->mm->mmap_sem);
24094     fork_out:
24095     if ((clone_flags & CLONE_VFORK) && (retval > 0))
24096         down(&sem);
24097     return retval;
24098
24099     bad_fork_cleanup_sighand:
24100     exit_sighand(p);
24101     bad_fork_cleanup_fs:
24102     exit_fs(p); /* blocking */
24103     bad_fork_cleanup_files:
24104     exit_files(p); /* blocking */
24105     bad_fork_cleanup:
24106     if (p->exec_domain && p->exec_domain->module)
24107         __MOD_DEC_USE_COUNT(p->exec_domain->module);
24108     if (p->binfmt && p->binfmt->module)
24109         __MOD_DEC_USE_COUNT(p->binfmt->module);
24110
24111     add_free_taskslot(p->tarray_ptr);
24112     bad_fork_free:
24113     free_task_struct(p);
24114     goto bad_fork;
24115 }
24116
24117 void __init files_cache_init(void)
24118 {
24119     files_cachep = kmem_cache_create("files_cache",
24120                                     sizeof(struct files_struct),
24121                                     0,
24122                                     SLAB_HWCACHE_ALIGN,
24123                                     NULL, NULL);
24124     if (!files_cachep)
24125         panic("Cannot create files cache");
24126 }

```

kernel/info.c

```

24127 /*
24128 * linux/kernel/info.c
24129 *
24130 * Copyright (C) 1992 Darren Senn
24131 */
24132

```

```

24133 /* This implements the sysinfo() system call */
24134
24135 #include <linux/mm.h>
24136 #include <linux/unistd.h>
24137 #include <linux/swap.h>
24138 #include <linux/smp_lock.h>
24139
24140 #include <asm/uaccess.h>
24141
24142 ↗ p472 asmlinkage int sys_sysinfo(struct sysinfo *info)
24143 {
24144     struct sysinfo val;
24145
24146     memset((char *)&val, 0, sizeof(struct sysinfo));
24147
24148     cli();
24149     val.uptime = jiffies / HZ;
24150
24151     val.loads[0] = avenrun[0] << (SI_LOAD_SHIFT - FSHIFT);
24152     val.loads[1] = avenrun[1] << (SI_LOAD_SHIFT - FSHIFT);
24153     val.loads[2] = avenrun[2] << (SI_LOAD_SHIFT - FSHIFT);
24154
24155     val.procs = nr_tasks-1;
24156     sti();
24157
24158     si_meminfo(&val);
24159     si_swapinfo(&val);
24160
24161     if (copy_to_user(info, &val, sizeof(struct sysinfo)))
24162         return -EFAULT;
24163     return 0;
24164 }

```

kernel/itimer.c

```

24165 /*
24166 * linux/kernel/itimer.c
24167 *
24168 * Copyright (C) 1992 Darren Senn
24169 */
24170
24171 /* These are all the functions necessary to implement
24172 * itimers */
24173
24174 #include <linux/mm.h>
24175 #include <linux/smp_lock.h>
24176 #include <linux/interrupt.h>
24177

```

```

24178 #include <asm/uaccess.h>
24179
24180 /* change timeval to jiffies, trying to avoid the most
24181 * obvious overflows..
24182 *
24183 * The tv_*sec values are signed, but nothing seems to
24184 * indicate whether we really should use them as signed
24185 * values when doing itimers. POSIX doesn't mention this
24186 * (but if alarm() uses itimers without checking, we have
24187 * to use unsigned arithmetic). */
24188 static unsigned long tvtojiffies(struct timeval *value)
24189 {
24190     unsigned long sec = (unsigned) value->tv_sec;
24191     unsigned long usec = (unsigned) value->tv_usec;
24192
24193     if (sec > (ULONG_MAX / HZ))
24194         return ULONG_MAX;
24195     usec += 1000000 / HZ - 1;
24196     usec /= 1000000 / HZ;
24197     return HZ*sec+usec;
24198 }
24199
24200 static void jiffiestotv(unsigned long jiffies,
24201                        struct timeval *value)
24202 {
24203     value->tv_usec = (jiffies % HZ) * (1000000 / HZ);
24204     value->tv_sec = jiffies / HZ;
24205 }
24206
24207 int do_getitimer(int which, struct itimerval *value)
24208 {
24209     register unsigned long val, interval;
24210
24211     switch (which) {
24212     case ITIMER_REAL:
24213         interval = current->it_real_incr;
24214         val = 0;
24215         start_bh_atomic();
24216         if (timer_pending(&current->real_timer)) {
24217             val = current->real_timer.expires - jiffies;
24218
24219             /* look out for negative/zero itimer.. */
24220             if ((long) val <= 0)
24221                 val = 1;
24222         }
24223         end_bh_atomic();
24224         break;
24225     case ITIMER_VIRTUAL:
24226         val = current->it_virt_value;
24227         interval = current->it_virt_incr;
24228         break;
24229     case ITIMER_PROF:
24230         val = current->it_prof_value;
24231         interval = current->it_prof_incr;
24232         break;
24233     default:
24234         return(-EINVAL);
24235     }
24236     jiffiestotv(val, &value->it_value);
24237     jiffiestotv(interval, &value->it_interval);
24238     return 0;
24239 }
24240
24241 /* SMP: Only we modify our itimer values. */
24242 asmlinkage int sys_getitimer(int which,
24243                               struct itimerval *value)
24244 {
24245     int error = -EFAULT;
24246     struct itimerval get_buffer;
24247
24248     if (value) {
24249         error = do_getitimer(which, &get_buffer);
24250         if (!error &&
24251             copy_to_user(value, &get_buffer,
24252                          sizeof(get_buffer)))
24253             error = -EFAULT;
24254     }
24255     return error;
24256 }
24257
24258 void it_real_fn(unsigned long ____data)
24259 {
24260     struct task_struct * p = (struct task_struct *) ____data;
24261     unsigned long interval;
24262
24263     send_sig(SIGALRM, p, 1);
24264     interval = p->it_real_incr;
24265     if (interval) {
24266         if (interval > (unsigned long) ULONG_MAX)
24267             interval = ULONG_MAX;
24268         p->real_timer.expires = jiffies + interval;
24269         add_timer(&p->real_timer);
24270     }
24271 }
24272
24273 int do_setitimer(int which, struct itimerval *value,

```

```

24274         struct itimerval *ovalue)
24275 {
24276     register unsigned long i, j;
24277     int k;
24278
24279     i = tvtojiffies(&value->it_interval);
24280     j = tvtojiffies(&value->it_value);
24281     if (ovalue && (k = do_getitimer(which, ovalue)) < 0)
24282         return k;
24283     switch (which) {
24284     case ITIMER_REAL:
24285         start_bh_atomic();
24286         del_timer(&current->real_timer);
24287         end_bh_atomic();
24288         current->it_real_value = j;
24289         current->it_real_incr = i;
24290         if (!j)
24291             break;
24292         if (j > (unsigned long) LONG_MAX)
24293             j = LONG_MAX;
24294         i = j + jiffies;
24295         current->real_timer.expires = i;
24296         add_timer(&current->real_timer);
24297         break;
24298     case ITIMER_VIRTUAL:
24299         if (j)
24300             j++;
24301         current->it_virt_value = j;
24302         current->it_virt_incr = i;
24303         break;
24304     case ITIMER_PROF:
24305         if (j)
24306             j++;
24307         current->it_prof_value = j;
24308         current->it_prof_incr = i;
24309         break;
24310     default:
24311         return -EINVAL;
24312     }
24313     return 0;
24314 }
24315
24316 /* SMP: Again, only we play with our itimers, and signals
24317  *   are SMP safe now so that is not an issue at all
24318  *   anymore. */
24319 asmlinkage int sys_setitimer(int which,
24320     struct itimerval *value, struct itimerval *ovalue)
24321 {

```

```

24322     struct itimerval set_buffer, get_buffer;
24323     int error;
24324
24325     if (value) {
24326         if (verify_area(VERIFY_READ, value, sizeof(*value)))
24327             return -EFAULT;
24328         if (copy_from_user(&set_buffer, value,
24329             sizeof(set_buffer)))
24330             return -EFAULT;
24331     } else
24332         memset((char *) &set_buffer, 0, sizeof(set_buffer));
24333
24334     error = do_setitimer(which, &set_buffer,
24335         ovalue ? &get_buffer : 0);
24336     if (error || !ovalue)
24337         return error;
24338
24339     if (copy_to_user(ovalue, &get_buffer,
24340         sizeof(get_buffer)))
24341         return -EFAULT;
24342     return 0;
24343 }

```

kernel/kmod.c

```

24344 /*
24345  * kmod, the new module loader (replaces kernelld)
24346  * Kirk Petersen
24347
24348  * Reorganized not to be a daemon by Adam Richter, with
24349  * guidance from Greg Zornetzer.
24350
24351  * Modified to avoid chroot and file sharing problems.
24352  * Mikael Pettersson */
24353
24354 #define __KERNEL_SYSCALLS__
24355
24356 #include <linux/sched.h>
24357 #include <linux/unistd.h>
24358 #include <linux/smp_lock.h>
24359
24360 #include <asm/uaccess.h>
24361
24362 /* modprobe_path is set via /proc/sys. */
24363 char modprobe_path[256] = "/sbin/modprobe";
24364
24365 static inline void
24366 use_init_file_context(void)

```

```

24367 {
24368     struct fs_struct * fs;
24369
24370     lock_kernel();
24371
24372     /* Don't use the user's root, use init's root instead.
24373      * Note that we can use "init_task" (which is not
24374      * actually the same as the user-level "init" process)
24375      * because we started "init" with a CLONE_FS */
24376     exit_fs(current);      /* current->fs->count--; */
24377     fs = init_task.fs;
24378     current->fs = fs;
24379     atomic_inc(&fs->count);
24380
24381     unlock_kernel();
24382 }
24383
24384 static int exec_modprobe(void * module_name)
24385 {
24386     static char * envp[] = { "HOME=/", "TERM=linux",
24387         "PATH=/sbin:/usr/sbin:/bin:/usr/bin", NULL };
24388     char *argv[] = { modprobe_path, "-s", "-k",
24389         (char*)module_name, NULL };
24390     int i;
24391
24392     use_init_file_context();
24393
24394     /* Prevent parent user process from sending signals to
24395      * child. Otherwise, if the modprobe program does not
24396      * exist, it might be possible to get a user defined
24397      * signal handler to execute as the super user right
24398      * after the execve fails if you time the signal just
24399      * right. */
24400     spin_lock_irq(&current->sigmask_lock);
24401     flush_signals(current);
24402     flush_signal_handlers(current);
24403     spin_unlock_irq(&current->sigmask_lock);
24404
24405     for (i = 0; i < current->files->max_fds; i++) {
24406         if (current->files->fd[i]) close(i);
24407     }
24408
24409     /* Drop the "current user" thing */
24410     free_uid(current);
24411
24412     /* Give kmod all privileges.. */
24413     current->uid = current->euid = current->fsuid = 0;
24414     cap_set_full(current->cap_inheritable);

```

→ p. 433

```

24415     cap_set_full(current->cap_effective);
24416
24417     /* Allow execve args to be in kernel space. */
24418     set_fs(KERNEL_DS);
24419
24420     /* Go, go, go... */
24421     if (execve(modprobe_path, argv, envp) < 0) {
24422         printk(KERN_ERR
24423             "kmod: failed to exec %s -s -k %s, errno = %d\n",
24424             modprobe_path, (char*) module_name, errno);
24425         return -errno;
24426     }
24427     return 0;
24428 }
24429
24430 /* request_module: the function that everyone calls when
24431  * they need a module. */
24432 int request_module(const char * module_name)
24433 {
24434     int pid;
24435     int waitpid_result;
24436     sigset_t tmsig;
24437
24438     /* Don't allow request_module() before the root fs is
24439      * mounted! */
24440     if (!current->fs->root) {
24441         printk(KERN_ERR "request_module[%s]: Root fs "
24442             "not mounted\n", module_name);
24443         return -EPERM;
24444     }
24445
24446     pid = kernel_thread(exec_modprobe,
24447         (void*) module_name, CLONE_FS);
24448     if (pid < 0) {
24449         printk(KERN_ERR "request_module[%s]: fork failed, "
24450             "errno %d\n", module_name, -pid);
24451         return pid;
24452     }
24453
24454     /* Block everything but SIGKILL/SIGSTOP */
24455     spin_lock_irq(&current->sigmask_lock);
24456     tmsig = current->blocked;
24457     siginitsetinv(&current->blocked,
24458         sigmask(SIGKILL) | sigmask(SIGSTOP));
24459     recalc_sigpending(current);
24460     spin_unlock_irq(&current->sigmask_lock);
24461
24462     waitpid_result = waitpid(pid, NULL, __WCLONE);

```

```

24463
24464 /* Allow signals again.. */
24465 spin_lock_irq(&current->sigmask_lock);
24466 current->blocked = tmpsig;
24467 recalc_sigpending(current);
24468 spin_unlock_irq(&current->sigmask_lock);
24469
24470 if (waitpid_result != pid) {
24471     printk (KERN_ERR "kmod: waitpid(%d,NULL,0) failed, "
24472            "returning %d.\n", pid, waitpid_result);
24473 }
24474 return 0;
24475 }

```

kernel/module.c

```

24476 #include <linux/config.h>
24477 #include <linux/mm.h>
24478 #include <linux/module.h>
24479 #include <asm/uaccess.h>
24480 #include <linux/vmalloc.h>
24481 #include <linux/smp_lock.h>
24482 #include <asm/pgtable.h>
24483 #include <linux/init.h>
24484
24485 /* Originally by Anonymous (as far as I know...)
24486 * Linux version by Bas Laarhoven <bas@vimec.nl>
24487 * 0.99.14 version by Jon Tombs <jon@gtex02.us.es>,
24488 * Heavily modified by Bjorn Ekwall <bj0rn@blox.se> May
24489 * 1994 (C)
24490 * Rewritten by Richard Henderson <rth@tam.u.edu> Dec 1996
24491 *
24492 * This source is covered by the GNU GPL, the same as all
24493 * kernel sources. */
24494
24495 #ifdef CONFIG_MODULES /* a *big* #ifdef block... */
24496
24497 extern struct module_symbol __start___ksymtab[];
24498 extern struct module_symbol __stop___ksymtab[];
24499
24500 extern const struct exception_table_entry
24501 __start___ex_table[];
24502 extern const struct exception_table_entry
24503 __stop___ex_table[];
24504
24505 static struct module kernel_module =
24506 {
24507     sizeof(struct module), /* size_of_struct */

```

```

24508 NULL, /* next */
24509 "", /* name */
24510 0, /* size */
24511 {ATOMIC_INIT(1)}, /* usecount */
24512 MOD_RUNNING, /* flags */
24513 0, /* nsyms -- filled in in init_modules */
24514 0, /* ndeps */
24515 __start___ksymtab, /* syms */
24516 NULL, /* deps */
24517 NULL, /* refs */
24518 NULL, /* init */
24519 NULL, /* cleanup */
24520 __start___ex_table, /* ex_table_start */
24521 __stop___ex_table, /* ex_table_end */
24522 /* Rest are NULL */
24523 };
24524
24525 struct module *module_list = &kernel_module;
24526
24527 static long get_mod_name(const char *user_name,
24528                          char **buf);
24529 static void put_mod_name(char *buf);
24530 static struct module *find_module(const char *name);
24531 static void free_module(struct module *, int tag_freed);
24532
24533
24534 /* Called at boot time */
24535
24536 __initfunc(void init_modules(void))
24537 {
24538     kernel_module.nsyms =
24539         __stop___ksymtab - __start___ksymtab;
24540
24541 #ifdef __alpha__
24542     __asm__("stq $29,%0" : "=m"(kernel_module.gp));
24543 #endif
24544 }
24545
24546 /* Copy the name of a module from user space. */
24547
24548 static inline long
24549 get_mod_name(const char *user_name, char **buf)
24550 {
24551     unsigned long page;
24552     long retval;
24553
24554     if (((unsigned long)user_name >= TASK_SIZE
24555         && !segment_eq(get_fs (), KERNEL_DS))

```

```

24556     return -EFAULT;
24557
24558     page = __get_free_page(GFP_KERNEL);
24559     if (!page)
24560         return -ENOMEM;
24561
24562     retval = strncpy_from_user((char *)page, user_name,
24563                               PAGE_SIZE);
24564     if (retval > 0) {
24565         if (retval < PAGE_SIZE) {
24566             *buf = (char *)page;
24567             return retval;
24568         }
24569         retval = -ENAMETOOLONG;
24570     } else if (!retval)
24571         retval = -EINVAL;
24572
24573     free_page(page);
24574     return retval;
24575 }
24576
24577 static inline void
24578 put_mod_name(char *buf)
24579 {
24580     free_page((unsigned long)buf);
24581 }
24582
24583 /* Allocate space for a module. */
24584
24585 asmlinkage unsigned long
24586 sys_create_module(const char *name_user, size_t size)
24587 {
24588     char *name;
24589     long namelen, error;
24590     struct module *mod;
24591
24592     lock_kernel();
24593     if (!capable(CAP_SYS_MODULE)) {
24594         error = -EPERM;
24595         goto err0;
24596     }
24597     if ((namelen = get_mod_name(name_user, &name)) < 0) {
24598         error = namelen;
24599         goto err0;
24600     }
24601     if (size < sizeof(struct module)+namelen) {
24602         error = -EINVAL;
24603         goto err1;

```

```

24604     }
24605     if (find_module(name) != NULL) {
24606         error = -EEXIST;
24607         goto err1;
24608     }
24609     if ((mod = (struct module *)module_map(size)) == NULL) {
24610         error = -ENOMEM;
24611         goto err1;
24612     }
24613
24614     memset(mod, 0, sizeof(*mod));
24615     mod->size_of_struct = sizeof(*mod);
24616     mod->next = module_list;
24617     mod->name = (char *)mod + 1;
24618     mod->size = size;
24619     memcpy((char*)(mod+1), name, namelen+1);
24620
24621     put_mod_name(name);
24622
24623     module_list = mod;      /* link it in */
24624
24625     error = (long) mod;
24626     goto err0;
24627 err1:
24628     put_mod_name(name);
24629 err0:
24630     unlock_kernel();
24631     return error;
24632 }
24633
24634 /* Initialize a module. */
24635
24636 asmlinkage int
24637 sys_init_module(const char *name_user,
24638                struct module *mod_user)
24639 {
24640     struct module mod_tmp, *mod;
24641     char *name, *n_name;
24642     long namelen, n_namelen, i, error = -EPERM;
24643     unsigned long mod_user_size;
24644     struct module_ref *dep;
24645
24646     lock_kernel();
24647     if (!capable(CAP_SYS_MODULE))
24648         goto err0;
24649     if ((namelen = get_mod_name(name_user, &name)) < 0) {
24650         error = namelen;
24651         goto err0;

```



```

24652 }
24653 if ((mod = find_module(name)) == NULL) {
24654     error = -ENOENT;
24655     goto err1;
24656 }
24657
24658 /* Check module header size. We allow a bit of slop
24659  * over the size we are familiar with to cope with a
24660  * version of insmod for a newer kernel. But don't
24661  * over do it. */
24662 if ((error = get_user(mod_user_size,
24663     &mod_user->size_of_struct)) != 0)
24664     goto err1;
24665 if (mod_user_size <
24666     (unsigned long)&((struct module *)0L)->persist_start
24667     || mod_user_size >
24668     sizeof(struct module) + 16*sizeof(void*)) {
24669     printk(KERN_ERR
24670         "init_module: Invalid module header size.\n"
24671         KERN_ERR
24672         "A new version of the modutils is likely "
24673         "needed.\n");
24674     error = -EINVAL;
24675     goto err1;
24676 }
24677
24678 /* Hold the current contents while we play with the
24679  * user's idea of righteousness. */
24680 mod_tmp = *mod;
24681
24682 error = copy_from_user(mod, mod_user,
24683     sizeof(struct module));
24684 if (error) {
24685     error = -EFAULT;
24686     goto err2;
24687 }
24688
24689 /* Sanity check the size of the module. */
24690 error = -EINVAL;
24691
24692 if (mod->size > mod_tmp.size) {
24693     printk(KERN_ERR
24694         "init_module: Size of initialized module "
24695         "exceeds size of created module.\n");
24696     goto err2;
24697 }
24698
24699 /* Make sure all interesting pointers are sane. */

```

```

24700
24701 #define bound(p, n, m)
24702     ((unsigned long)(p) >= (unsigned long)(m+1) &&
24703     (unsigned long)((p)+(n)) <=
24704     (unsigned long)(m) + (m)->size)
24705
24706 if (!bound(mod->name, namelen, mod)) {
24707     printk(KERN_ERR
24708         "init_module: mod->name out of bounds.\n");
24709     goto err2;
24710 }
24711 if (mod->nsyms && !bound(mod->syms, mod->nsyms, mod)) {
24712     printk(KERN_ERR
24713         "init_module: mod->syms out of bounds.\n");
24714     goto err2;
24715 }
24716 if (mod->ndeps && !bound(mod->deps, mod->ndeps, mod)) {
24717     printk(KERN_ERR
24718         "init_module: mod->deps out of bounds.\n");
24719     goto err2;
24720 }
24721 if (mod->init && !bound(mod->init, 0, mod)) {
24722     printk(KERN_ERR
24723         "init_module: mod->init out of bounds.\n");
24724     goto err2;
24725 }
24726 if (mod->cleanup && !bound(mod->cleanup, 0, mod)) {
24727     printk(KERN_ERR
24728         "init_module: mod->cleanup out of bounds.\n");
24729     goto err2;
24730 }
24731 if (mod->ex_table_start > mod->ex_table_end
24732     || (mod->ex_table_start &&
24733     !((unsigned long)mod->ex_table_start >=
24734     (unsigned long)(mod+1)
24735     && ((unsigned long)mod->ex_table_end
24736     < (unsigned long)mod + mod->size)))
24737     || (((unsigned long)mod->ex_table_start
24738     - (unsigned long)mod->ex_table_end)
24739     % sizeof(struct exception_table_entry)) {
24740     printk(KERN_ERR
24741         "init module: mod->ex_table_* invalid.\n");
24742     goto err2;
24743 }
24744 if (mod->flags & ~MOD_AUTOCLFAN) {
24745     printk(KERN_ERR
24746         "init_module: mod >flags invalid.\n");
24747     goto err2;

```

```

24748 }
24749 #ifdef __alpha__
24750 if (!bound(mod->gp - 0x8000, 0, mod)) {
24751     printk(KERN_ERR
24752         "init_module: mod->gp out of bounds.\n");
24753     goto err2;
24754 }
24755 #endif
24756 if (mod_member_present(mod, can_unload) &&
24757     mod->can_unload &&
24758     !bound(mod->can_unload, 0, mod)) {
24759     printk(KERN_ERR "init_module: mod->can_unload out "
24760         "of bounds.\n");
24761     goto err2;
24762 }
24763
24764 #undef bound
24765
24766 /* Check that the user isn't doing something silly with
24767  * the name. */
24768
24769 if ((n_namelen =
24770     get_mod_name(mod->name - (unsigned long) mod
24771         + (unsigned long) mod_user,
24772         &n_name)) < 0) {
24773     error = n_namelen;
24774     goto err2;
24775 }
24776 if (namelen != n_namelen ||
24777     strcmp(n_name, mod_tmp.name) != 0) {
24778     printk(KERN_ERR
24779         "init_module: changed module name to "
24780         "'%s' from '%s'\n", n_name, mod_tmp.name);
24781     goto err3;
24782 }
24783
24784 /* Ok, that's about all the sanity we can stomach; copy
24785  * the rest. */
24786 if (copy_from_user(mod+1, mod_user+1,
24787     mod->size - sizeof(*mod))) {
24788     error = -EFAULT;
24789     goto err3;
24790 }
24791
24792 /* On some machines it is necessary to do something
24793  * here to make the I and D caches consistent. */
24794 flush_icache_range((unsigned long)mod,
24795     (unsigned long)mod + mod->size);

```

```

24796
24797 /* Update module references. */
24798 mod->next = mod_tmp.next;
24799 mod->refs = NULL;
24800 for (i = 0, dep = mod->deps; i < mod->ndeps;
24801     ++i, ++dep) {
24802     struct module *o, *d = dep->dep;
24803
24804     /* Make sure the indicated dependencies are really
24805      * modules. */
24806     if (d == mod) {
24807         printk(KERN_ERR "init_module: self-referential "
24808             "dependency in mod->deps.\n");
24809         goto err3;
24810     }
24811
24812     for (o = module_list; o != &kernel_module;
24813         o = o->next)
24814         if (o == d) goto found_dep;
24815
24816     printk(KERN_ERR
24817         "init_module: found dependency that is "
24818         "(no longer?) a module.\n");
24819     goto err3;
24820
24821 found_dep:
24822     dep->ref = mod;
24823     dep->next_ref = d->refs;
24824     d->refs = dep;
24825     /* Being referenced by a dependent module counts as a
24826      * use as far as kmod is concerned. */
24827     d->flags |= MOD_USED.ONCE;
24828 }
24829
24830 /* Free our temporary memory. */
24831 put_mod_name(n_name);
24832 put_mod_name(name);
24833
24834 /* Initialize the module. */
24835 atomic_set(&mod->uc.usecount,1);
24836 if (mod->init && mod >init() != 0) {
24837     atomic_set(&mod->uc.usecount,0);
24838     error = -EBUSY;
24839     goto err0;
24840 }
24841 atomic_dec(&mod->uc.usecount);
24842
24843 /* And set it running. */

```

```

24844 mod->flags |= MOD_RUNNING;
24845 error = 0;
24846 goto err0;
24847
24848 err3:
24849 put_mod_name(n_name);
24850 err2:
24851 *mod = mod_tmp;
24852 err1:
24853 put_mod_name(name);
24854 err0:
24855 unlock_kernel();
24856 return error;
24857 }
24858
24859 asmlinkage int
24860 sys_delete_module(const char *name_user)
24861 {
24862     struct module *mod, *next;
24863     char *name;
24864     long error = -EPERM;
24865     int something_changed;
24866
24867     lock_kernel();
24868     if (!capable(CAP_SYS_MODULE))
24869         goto out;
24870
24871     if (name_user) {
24872         if ((error = get_mod_name(name_user, &name)) < 0)
24873             goto out;
24874         if (error == 0) {
24875             error = -EINVAL;
24876             put_mod_name(name);
24877             goto out;
24878         }
24879         error = -ENOENT;
24880         if ((mod = find_module(name)) == NULL) {
24881             put_mod_name(name);
24882             goto out;
24883         }
24884         put_mod_name(name);
24885         error = -EBUSY;
24886         if (mod->refs != NULL || __MOD_IN_USE(mod))
24887             goto out;
24888
24889         free_module(mod, 0);
24890         error = 0;
24891         goto out;

```

```

24892     }
24893
24894     /* Do automatic reaping */
24895 restart:
24896     something_changed = 0;
24897     for (mod = module_list; mod != &kernel_module;
24898         mod = next) {
24899         next = mod->next;
24900         if (mod->refs == NULL
24901             && (mod->flags & MOD_AUTOCLEAN)
24902             && (mod->flags & MOD_RUNNING)
24903             && !(mod->flags & MOD_DELETED)
24904             && (mod->flags & MOD_USED_ONCE)
24905             && !__MOD_IN_USE(mod)) {
24906             if ((mod->flags & MOD_VISITED)
24907                 && !(mod->flags & MOD_JUST_FREED)) {
24908                 mod->flags &= ~MOD_VISITED;
24909             } else {
24910                 free_module(mod, 1);
24911                 something_changed = 1;
24912             }
24913         }
24914     }
24915     if (something_changed)
24916         goto restart;
24917     for (mod = module_list; mod != &kernel_module;
24918         mod = mod->next)
24919         mod->flags &= ~MOD_JUST_FREED;
24920     error = 0;
24921 out:
24922     unlock_kernel();
24923     return error;
24924 }
24925
24926 /* Query various bits about modules. */
24927
24928 static int
24929 qm_modules(char *buf, size_t bufsize, size_t *ret)
24930 {
24931     struct module *mod;
24932     size_t nmod, space, len;
24933
24934     nmod = space = 0;
24935
24936     for (mod = module_list; mod != &kernel_module;
24937         mod = mod->next, ++nmod) {
24938         len = strlen(mod->name)+1;
24939         if (len > bufsize)

```

```

24940     goto calc_space_needed;
24941     if (copy_to_user(buf, mod->name, len))
24942         return -EFAULT;
24943     buf += len;
24944     bufsize -= len;
24945     space += len;
24946 }
24947
24948 if (put_user(nmod, ret))
24949     return -EFAULT;
24950 else
24951     return 0;
24952
24953 calc_space_needed:
24954     space += len;
24955     while ((mod = mod->next) != &kernel_module)
24956         space += strlen(mod->name)+1;
24957
24958     if (put_user(space, ret))
24959         return -EFAULT;
24960     else
24961         return -ENOSPC;
24962 }
24963
24964 static int
24965 qm_deps(struct module *mod, char *buf, size_t bufsize,
24966         size_t *ret)
24967 {
24968     size_t i, space, len;
24969
24970     if (mod == &kernel_module)
24971         return -EINVAL;
24972     if ((mod->flags & (MOD_RUNNING | MOD_DELETED)) != MOD_
24973         RUNNING)
24974         if (put_user(0, ret))
24975             return -EFAULT;
24976         else
24977             return 0;
24978
24979     space = 0;
24980     for (i = 0; i < mod->ndeps; ++i) {
24981         const char *dep_name = mod->deps[i].dep->name;
24982
24983         len = strlen(dep_name)+1;
24984         if (len > bufsize)
24985             goto calc_space_needed;
24986         if (copy_to_user(buf, dep_name, len))
24987             return -EFAULT;

```

```

24988     buf += len;
24989     bufsize -= len;
24990     space += len;
24991 }
24992
24993 if (put_user(i, ret))
24994     return -EFAULT;
24995 else
24996     return 0;
24997
24998 calc_space_needed:
24999     space += len;
25000     while (++i < mod->ndeps)
25001         space += strlen(mod->deps[i].dep->name)+1;
25002
25003     if (put_user(space, ret))
25004         return -EFAULT;
25005     else
25006         return -ENOSPC;
25007 }
25008
25009 static int
25010 qm_refs(struct module *mod, char *buf, size_t bufsize,
25011         size_t *ret)
25012 {
25013     size_t nrefs, space, len;
25014     struct module_ref *ref;
25015
25016     if (mod == &kernel_module)
25017         return -EINVAL;
25018     if ((mod->flags & (MOD_RUNNING | MOD_DELETED)) !=
25019         MOD_RUNNING)
25020         if (put_user(0, ret))
25021             return -EFAULT;
25022         else
25023             return 0;
25024
25025     space = 0;
25026     for (nrefs = 0, ref = mod->refs; ref;
25027         ++nrefs, ref = ref->next_ref) {
25028         const char *ref_name = ref->ref->name;
25029
25030         len = strlen(ref_name)+1;
25031         if (len > bufsize)
25032             goto calc_space_needed;
25033         if (copy_to_user(buf, ref_name, len))
25034             return -EFAULT;
25035         buf += len;

```

```

25036     bufsize -= len;
25037     space += len;
25038 }
25039
25040 if (put_user(nrefs, ret))
25041     return -EFAULT;
25042 else
25043     return 0;
25044
25045 calc_space_needed:
25046     space += len;
25047     while ((ref = ref->next_ref) != NULL)
25048         space += strlen(ref->ref->name)+1;
25049
25050 if (put_user(space, ret))
25051     return -EFAULT;
25052 else
25053     return -ENOSPC;
25054 )
25055
25056 static int
25057 qm_symbols(struct module *mod, char *buf, size_t bufsize,
25058           size_t *ret)
25059 {
25060     size_t i, space, len;
25061     struct module_symbol *s;
25062     char *strings;
25063     unsigned long *vals;
25064
25065     if ((mod->flags & (MOD_RUNNING | MOD_DELETED)) !=
25066         MOD_RUNNING)
25067         if (put_user(0, ret))
25068             return -EFAULT;
25069         else
25070             return 0;
25071
25072     space = mod->nsyms * 2*sizeof(void *);
25073
25074     i = len = 0;
25075     s = mod->syms;
25076
25077     if (space > bufsize)
25078         goto calc_space_needed;
25079
25080     if (!access_ok(VERIFY_WRITE, buf, space))
25081         return -EFAULT;
25082
25083     bufsize -= space;

```

```

25084     vals = (unsigned long *)buf;
25085     strings = buf+space;
25086
25087     for (; i < mod->nsyms; ++i, ++s, vals += 2) {
25088         len = strlen(s->name)+1;
25089         if (len > bufsize)
25090             goto calc_space_needed;
25091
25092         if (copy_to_user(strings, s->name, len)
25093             || __put_user(s->value, vals+0)
25094             || __put_user(space, vals+1))
25095             return -EFAULT;
25096
25097         strings += len;
25098         bufsize -= len;
25099         space += len;
25100     }
25101
25102     if (put_user(i, ret))
25103         return -EFAULT;
25104     else
25105         return 0;
25106
25107 calc_space_needed:
25108     for (; i < mod->nsyms; ++i, ++s)
25109         space += strlen(s->name)+1;
25110
25111     if (put_user(space, ret))
25112         return -EFAULT;
25113     else
25114         return -ENOSPC;
25115 }
25116
25117 static int
25118 qm_info(struct module *mod, char *buf, size_t bufsize,
25119        size_t *ret)
25120 {
25121     int error = 0;
25122
25123     if (mod == &kernel_module)
25124         return -EINVAL;
25125
25126     if (sizeof(struct module_info) <= bufsize) {
25127         struct module_info info;
25128         info.addr = (unsigned long)mod;
25129         info.size = mod->size;
25130         info.flags = mod->flags;
25131         info.usecount = (mod_member_present(mod, can_unload)

```

```

25132     && mod->can_unload
25133     ? -1 : atomic_read(&mod->uc.usecount));
25134
25135     if (copy_to_user(buf, &info,
25136                     sizeof(struct module_info)))
25137         return -EFAULT;
25138 } else
25139     error = -ENOSPC;
25140
25141     if (put_user(sizeof(struct module_info), ret))
25142         return -EFAULT;
25143
25144     return error;
25145 }
25146
25147 asmlinkage int
25148 sys_query_module(const char *name_user, int which,
25149                 char *buf, size_t bufsize, size_t *ret)
25150 {
25151     struct module *mod;
25152     int err;
25153
25154     lock_kernel();
25155     if (name_user == NULL)
25156         mod = &kernel_module;
25157     else {
25158         long namelen;
25159         char *name;
25160
25161         if ((namelen = get_mod_name(name_user, &name)) < 0) {
25162             err = namelen;
25163             goto out;
25164         }
25165         err = -ENOENT;
25166         if (namelen == 0)
25167             mod = &kernel_module;
25168         else if ((mod = find_module(name)) == NULL) {
25169             put_mod_name(name);
25170             goto out;
25171         }
25172         put_mod_name(name);
25173     }
25174
25175     switch (which)
25176     {
25177     case 0:
25178         err = 0;
25179         break;

```

```

25180     case QM_MODULES:
25181         err = qm_modules(buf, bufsize, ret);
25182         break;
25183     case QM_DEPS:
25184         err = qm_deps(mod, buf, bufsize, ret);
25185         break;
25186     case QM_REFS:
25187         err = qm_refs(mod, buf, bufsize, ret);
25188         break;
25189     case QM_SYMBOLS:
25190         err = qm_symbols(mod, buf, bufsize, ret);
25191         break;
25192     case QM_INFO:
25193         err = qm_info(mod, buf, bufsize, ret);
25194         break;
25195     default:
25196         err = -EINVAL;
25197         break;
25198     }
25199     out:
25200     unlock_kernel();
25201     return err;
25202 }
25203
25204 /* Copy the kernel symbol table to user space. If the
25205 * argument is NULL, just return the size of the table.
25206 *
25207 * This call is obsolete. New programs should use
25208 * query_module+QM_SYMBOLS which does not arbitrarily
25209 * limit the length of symbols. */
25210 asmlinkage int
25211 sys_get_kernel_syms(struct kernel_sym *table)
25212 {
25213     struct module *mod;
25214     int i;
25215     struct kernel_sym ksym;
25216
25217     lock_kernel();
25218     for (mod = module_list, i = 0; mod; mod = mod->next) {
25219         /* include the count for the module name! */
25220         i += mod->nsyms + 1;
25221     }
25222
25223     if (table == NULL)
25224         goto out;
25225
25226     /* So that we don't give the user our stack content */
25227     memset (&ksym, 0, sizeof (ksym));

```

```

25228
25229 for (mod = module_list, i = 0; mod; mod = mod->next) {
25230     struct module_symbol *msym;
25231     unsigned int j;
25232
25233     if ((mod->flags & (MOD_RUNNING|MOD_DELETED)) !=
25234         MOD_RUNNING)
25235         continue;
25236
25237     /* magic: write module info as a pseudo symbol */
25238     ksym.value = (unsigned long)mod;
25239     ksym.name[0] = '#';
25240     strncpy(ksym.name+1, mod->name, sizeof(ksym.name)-1);
25241     ksym.name[sizeof(ksym.name)-1] = '\0';
25242
25243     if (copy_to_user(table, &ksym, sizeof(ksym)) != 0)
25244         goto out;
25245     ++i, ++table;
25246
25247     if (mod->nsyms == 0)
25248         continue;
25249
25250     for (j = 0, msym = mod->syms; j < mod->nsyms;
25251         ++j, ++msym) {
25252         ksym.value = msym->value;
25253         strncpy(ksym.name, msym->name, sizeof(ksym.name));
25254         ksym.name[sizeof(ksym.name)-1] = '\0';
25255
25256         if (copy_to_user(table, &ksym, sizeof(ksym)) != 0)
25257             goto out;
25258         ++i, ++table;
25259     }
25260 }
25261 out:
25262     unlock_kernel();
25263     return i;
25264 }
25265
25266 /* Look for a module by name, ignoring modules marked for
25267 * deletion. */
25268 static struct module *
25269 find_module(const char *name)
25270 {
25271     struct module *mod;
25272
25273     for (mod = module_list; mod ; mod = mod->next) {
25274         if (mod->flags & MOD_DELETED)
25275             continue;

```

```

25276         if (!strcmp(mod->name, name))
25277             break;
25278     }
25279
25280     return mod;
25281 }
25282
25283 /* Free the given module. */
25284
25285 static void
25286 free_module(struct module *mod, int tag_freed)
25287 {
25288     struct module_ref *dep;
25289     unsigned i;
25290
25291     /* Let the module clean up. */
25292
25293     mod->flags |= MOD_DELETED;
25294     if (mod->flags & MOD_RUNNING)
25295     {
25296         if(mod->cleanup)
25297             mod->cleanup();
25298         mod->flags &= ~MOD_RUNNING;
25299     }
25300
25301     /* Remove the module from the dependency lists. */
25302
25303     for (i = 0, dep = mod->deps; i < mod->ndeps;
25304         ++i, ++dep) {
25305         struct module_ref **pp;
25306         for (pp = &dep->dep->refs; *pp != dep;
25307             pp = &(*pp)->next_ref)
25308             continue;
25309         *pp = dep->next_ref;
25310         if (tag_freed && dep->dep->refs == NULL)
25311             dep->dep->flags |= MOD_JUST_FREED;
25312     }
25313
25314     /* And from the main module list. */
25315
25316     if (mod == module_list) {
25317         module_list = mod->next;
25318     } else {
25319         struct module *p;
25320         for (p = module_list; p->next != mod; p = p->next)
25321             continue;
25322         p->next = mod->next;
25323     }

```

```

25324
25325 /* And free the memory. */
25326
25327 module_unmap(mod);
25328 }
25329
25330 /* Called by the /proc file system to return a current
25331 * list of modules. */
25332 int get_module_list(char *p)
25333 {
25334     size_t left = PAGE_SIZE;
25335     struct module *mod;
25336     char tmpstr[64];
25337     struct module_ref *ref;
25338
25339     for (mod = module_list; mod != &kernel_module;
25340         mod = mod->next) {
25341         long len;
25342         const char *q;
25343
25344         #define safe_copy_str(str, len) \
25345         do { \
25346             if (left < len) \
25347                 goto fini; \
25348             memcpy(p, str, len); p += len, left -= len; \
25349         } while (0)
25350         #define safe_copy_cstr(str) \
25351         safe_copy_str(str, sizeof(str)-1)
25352
25353         len = strlen(mod->name);
25354         safe_copy_str(mod->name, len);
25355
25356         if ((len = 20 - len) > 0) {
25357             if (left < len)
25358                 goto fini;
25359             memset(p, ' ', len);
25360             p += len;
25361             left -= len;
25362         }
25363
25364         len = sprintf(tmpstr, "%8lu", mod->size);
25365         safe_copy_str(tmpstr, len);
25366
25367         if (mod->flags & MOD_RUNNING) {
25368             len = sprintf(tmpstr, "%4ld",
25369                 (mod_member_present(mod, can_unload)
25370                  && mod->can_unload
25371                  ? -1L

```

```

25372                 : (long)atomic_read(&mod->uc.usecount)));
25373         safe_copy_str(tmpstr, len);
25374     }
25375
25376     if (mod->flags & MOD_DELETED)
25377         safe_copy_cstr(" (deleted)");
25378     else if (mod->flags & MOD_RUNNING) {
25379         if (mod->flags & MOD_AUTOCLEAN)
25380             safe_copy_cstr(" (autoclean)");
25381         if (!(mod->flags & MOD_USED_ONCE))
25382             safe_copy_cstr(" (unused)");
25383     } else
25384         safe_copy_cstr(" (uninitialized)");
25385
25386     if ((ref = mod->refs) != NULL) {
25387         safe_copy_cstr(" [");
25388         while (1) {
25389             q = ref->ref->name;
25390             len = strlen(q);
25391             safe_copy_str(q, len);
25392
25393             if ((ref = ref->next_ref) != NULL)
25394                 safe_copy_cstr(" ");
25395             else
25396                 break;
25397         }
25398         safe_copy_cstr("]");
25399     }
25400     safe_copy_cstr("\n");
25401
25402 #undef safe_copy_str
25403 #undef safe_copy_cstr
25404 }
25405
25406 fini:
25407     return PAGE_SIZE - left;
25408 }
25409
25410 /* Called by the /proc file system to return a current
25411 * list of ksyms. */
25412 int
25413 get_ksyms_list(char *buf, char **start, off_t offset,
25414               int length)
25415 {
25416     struct module *mod;
25417     char *p = buf;
25418     int len = 0; /* code from net/ipv4/proc.c */
25419     off_t pos = 0;

```



```

25420 off_t begin = 0;
25421
25422 for (mod = module_list; mod; mod = mod->next) {
25423     unsigned i;
25424     struct module_symbol *sym;
25425
25426     if (!(mod->flags & MOD_RUNNING) ||
25427         (mod->flags & MOD_DELETED))
25428         continue;
25429
25430     for (i = mod->nsyms, sym = mod->syms; i > 0;
25431          --i, ++sym) {
25432         p = buf + len;
25433         if (*mod->name) {
25434             len += sprintf(p, "%0*lx %s\t[%s]\n",
25435                            (int)(2*sizeof(void*)),
25436                            sym->value, sym->name,
25437                            mod->name);
25438         } else {
25439             len += sprintf(p, "%0*lx %s\n",
25440                            (int)(2*sizeof(void*)),
25441                            sym->value, sym->name);
25442         }
25443         pos = begin + len;
25444         if (pos < offset) {
25445             len = 0;
25446             begin = pos;
25447         }
25448         pos = begin + len;
25449         if (pos > offset+length)
25450             goto leave_the_loop;
25451     }
25452 }
25453 leave_the_loop:
25454 *start = buf + (offset - begin);
25455 len -= (offset - begin);
25456 if (len > length)
25457     len = length;
25458 return len;
25459 }
25460
25461 /* Gets the address for a symbol in the given module. If
25462 * modname is NULL, it looks for the name in any
25463 * registered symbol table. If the modname is an empty
25464 * string, it looks for the symbol in kernel exported
25465 * symbol tables. */
25466 unsigned long
25467 get_module_symbol(char *modname, char *symname)

```

```

25468 {
25469     struct module *mp;
25470     struct module_symbol *sym;
25471     int i;
25472
25473     for (mp = module_list; mp; mp = mp->next) {
25474         if (((modname == NULL) ||
25475             (strcmp(mp->name, modname) == 0)) &&
25476             (mp->flags & (MOD_RUNNING | MOD_DELETED)) ==
25477             MOD_RUNNING && (mp->nsyms > 0)) {
25478             for (i = mp->nsyms, sym = mp->syms;
25479                  i > 0; --i, ++sym) {
25480
25481                 if (strcmp(sym->name, symname) == 0) {
25482                     return sym->value;
25483                 }
25484             }
25485         }
25486     }
25487     return 0;
25488 }
25489
25490 #else /* CONFIG_MODULES */
25491
25492 /* Dummy syscalls for people who don't want modules */
25493
25494 asmlinkage unsigned long
25495 sys_create_module(const char *name, user, size_t size)
25496 {
25497     return -ENOSYS;
25498 }
25499
25500 asmlinkage int
25501 sys_init_module(const char *name, user,
25502                struct module *mod, user)
25503 {
25504     return -ENOSYS;
25505 }
25506
25507 asmlinkage int
25508 sys_delete_module(const char *name, user)
25509 {
25510     return -ENOSYS;
25511 }
25512
25513 asmlinkage int
25514 sys_query_module(const char *name, user, int which,
25515                  char *buf, size_t bufsize, size_t *ret)

```

```

25516 {
25517 /* Let the program know about the new interface. Not
25518 * that it'll do them much good. */
25519 if (which == 0)
25520     return 0;
25521
25522 return -ENOSYS;
25523 }
25524
25525 asmlinkage int
25526 sys_get_kernel_syms(struct kernel_sym *table)
25527 {
25528     return -ENOSYS;
25529 }
25530
25531 #endif /* CONFIG_MODULES */

```

kernel/panic.c

```

25532 /*
25533 * linux/kernel/panic.c
25534 *
25535 * Copyright (C) 1991, 1992 Linus Torvalds
25536 */
25537
25538 /* This function is used through-out the kernel
25539 * (including mm and fs) to indicate a major problem. */
25540 #include <linux/sched.h>
25541 #include <linux/delay.h>
25542 #include <linux/reboot.h>
25543 #include <linux/init.h>
25544 #include <linux/sysrq.h>
25545 #include <linux/interrupt.h>
25546
25547 #ifdef __alpha__
25548 #include <asm/machvec.h>
25549 #endif
25550
25551 asmlinkage void sys_sync(void); /* it's really int */
25552 extern void unblank_console(void);
25553 extern int C_A_D;
25554
25555 int panic_timeout = 0;
25556
25557 void __init panic_setup(char *str, int *ints)
25558 {
25559     if (ints[0] == 1)
25560         panic_timeout = ints[1];
25561 }

```

```

25562
25563 NORET_TYPE void panic(const char * fmt, ...)
25564 {
25565     static char buf[1024];
25566     va_list args;
25567
25568     va_start(args, fmt);
25569     vsprintf(buf, fmt, args);
25570     va_end(args);
25571     printk(KERN_EMERG "Kernel panic: %s\n", buf);
25572     if (current == task[0])
25573         printk(KERN_EMERG "In swapper task - not syncing\n");
25574     else if (in_interrupt())
25575         printk(KERN_EMERG
25576                "In interrupt handler - not syncing\n");
25577     else
25578         sys_sync();
25579
25580     unblank_console();
25581
25582 #ifdef __SMP__
25583     smp_send_stop();
25584 #endif
25585     if (panic_timeout > 0)
25586     {
25587         /* Delay timeout seconds before rebooting the
25588          * machine. We can't use the "normal" timers since
25589          * we just panicked.. */
25590         printk(KERN_EMERG
25591                "Rebooting in %d seconds..", panic_timeout);
25592         mdelay(panic_timeout*1000);
25593         /* Should we run the reboot notifier. For the moment
25594          * I'm choosing not to. It might crash, be corrupt,
25595          * or do more harm than good for other reasons. */
25596         machine_restart(NULL);
25597     }
25598 #ifdef __sparc__
25599     printk("Press LI-A to return to the boot prom\n");
25600 #endif
25601 #ifdef __alpha__
25602     if (alpha_using_srm)
25603         halt();
25604 #endif
25605     sti();
25606     for(;;) {
25607         CHECK_EMERGENCY_SYNC
25608     }
25609 }

```

kernel/printk.c

```
25610 /*
25611 * linux/kernel/printk.c
25612 *
25613 * Copyright (C) 1991, 1992 Linus Torvalds
25614 *
25615 * Modified to make sys_syslog() more flexible: added
25616 * commands to return the last 4k of kernel messages,
25617 * regardless of whether they've been read or not. Added
25618 * option to suppress kernel printk's to the console.
25619 * Added hook for sending the console messages elsewhere,
25620 * in preparation for a serial line console (someday).
25621 * Ted Ts'o, 2/11/93.
25622 * Modified for sysctl support, 1/8/97, Chris Horn.
25623 */
25624 #include <linux/mm.h>
25625 #include <linux/tty_driver.h>
25626 #include <linux/smp_lock.h>
25627 #include <linux/console.h>
25628 #include <linux/init.h>
25629
25630 #include <asm/uaccess.h>
25631
25632 #define LOG_BUF_LEN      (16384)
25633
25634 static char buf[1024];
25635
25636 /* printk's without a loglevel use this.. */
25637 #define DEFAULT_MESSAGE_LOGLEVEL 4 /* KERN_WARNING */
25638
25639 /* We show everything that is MORE important than
25640 * this.. */
25641 /* Minimum loglevel we let people use */
25642 #define MINIMUM_CONSOLE_LOGLEVEL 1
25643 /* anything MORE serious than KERN_DEBUG */
25644 #define DEFAULT_CONSOLE_LOGLEVEL 7
25645
25646 unsigned long log_size = 0;
25647 struct wait_queue * log_wait = NULL;
25648
25649 /* Keep together for sysctl support */
25650 int console_loglevel = DEFAULT_CONSOLE_LOGLEVEL;
25651 int default_message_loglevel = DEFAULT_MESSAGE_LOGLEVEL;
25652 int minimum_console_loglevel = MINIMUM_CONSOLE_LOGLEVEL;
25653 int default_console_loglevel = DEFAULT_CONSOLE_LOGLEVEL;
25654
25655 struct console *console_drivers = NULL;
25656 static char log_buf[LOG_BUF_LEN];
```

```
25657 static unsigned long log_start = 0;
25658 static unsigned long logged_chars = 0;
25659 struct console_cmdline
25660 console_cmdline[MAX_CMDLINECONSOLES];
25661 static int preferred_console = -1;
25662
25663 /* Setup a list of consoles. Called from init/main.c */
25664 void __init console_setup(char *str, int *ints)
25665 {
25666     struct console_cmdline *c;
25667     char name[sizeof(c->name)];
25668     char *s, *options;
25669     int i, idx;
25670
25671     /* Decode str into name, index, options. */
25672     if (str[0] >= '0' && str[0] <= '9') {
25673         strcpy(name, "ttyS");
25674         strncpy(name + 4, str, sizeof(name) - 5);
25675     } else
25676         strncpy(name, str, sizeof(name) - 1);
25677     name[sizeof(name) - 1] = 0;
25678     if ((options = strchr(str, ',')) != NULL)
25679         *(options++) = 0;
25680 #ifdef __sparc__
25681     if (!strcmp(str, "ttya"))
25682         strcpy(name, "ttyS0");
25683     if (!strcmp(str, "ttyb"))
25684         strcpy(name, "ttyS1");
25685 #endif
25686     for(s = name; *s; s++)
25687         if (*s >= '0' && *s <= '9')
25688             break;
25689     idx = simple_strtoul(s, NULL, 10);
25690     *s = 0;
25691
25692     /* See if this tty is not yet registered, and if we
25693     * have a slot free. */
25694     for(i = 0; i < MAX_CMDLINECONSOLES &&
25695         console_cmdline[i].name[0]; i++)
25696         if (strcmp(console_cmdline[i].name, name) == 0 &&
25697             console_cmdline[i].index == idx) {
25698             preferred_console = i;
25699             return;
25700         }
25701     if (i == MAX_CMDLINECONSOLES)
25702         return;
25703     preferred_console = i;
25704     c = &console_cmdline[i];
```

```

25705 memcpy(c->name, name, sizeof(c->name));
25706 c->options = options;
25707 c->index = idx;
25708 }
25709
25710 /* Commands to do_syslog:
25711 *
25712 * 0 -- Close the log. Currently a NOP.
25713 * 1 -- Open the log. Currently a NOP.
25714 * 2 -- Read from the log.
25715 * 3 -- Read up to the last 4k of messages in the
25716 *      ring buffer.
25717 * 4 -- Read and clear last 4k of messages in the
25718 *      ring buffer.
25719 * 5 -- Clear ring buffer.
25720 * 6 -- Disable printk's to console
25721 * 7 -- Enable printk's to console
25722 * 8 -- Set level of messages printed to console
25723 */
25724 int do_syslog(int type, char * buf, int len)
25725 {
25726     unsigned long i, j, count, flags;
25727     int do_clear = 0;
25728     char c;
25729     int error = -EPERM;
25730
25731     lock_kernel();
25732     error = 0;
25733     switch (type) {
25734     case 0:          /* Close log */
25735         break;
25736     case 1:          /* Open log */
25737         break;
25738     case 2:          /* Read from log */
25739         error = -EINVAL;
25740         if (!buf || len < 0)
25741             goto out;
25742         error = 0;
25743         if (!len)
25744             goto out;
25745         error = verify_area(VRFIFY_WRITE, buf, len);
25746         if (error)
25747             goto out;
25748         error = wait_event_interruptible(log_wait, log_size);
25749         if (error)
25750             goto out;
25751         i = 0;
25752         while (log_size && i < len) {
25753             c = *((char *) log_buf + log_start);
25754             log_start++;
25755             log_size--;
25756             log_start &= LOG_BUF_LEN - 1;
25757             sti();
25758             __put_user(c, buf);
25759             buf++;
25760             i++;
25761             cli();
25762         }
25763         sti();
25764         error = i;
25765         break;
25766     case 4:          /* Read/clear last kernel messages */
25767         do_clear = 1;
25768         /* FALL THRU */
25769     case 3:          /* Read last kernel messages */
25770         error = -EINVAL;
25771         if (!buf || len < 0)
25772             goto out;
25773         error = 0;
25774         if (!len)
25775             goto out;
25776         error = verify_area(VERIFY_WRITE, buf, len);
25777         if (error)
25778             goto out;
25779         /* The logged_chars, log_start, and log_size values
25780          * may change from an interrupt, so we disable
25781          * interrupts. */
25782         __save_flags(flags);
25783         __cli();
25784         count = len;
25785         if (count > LOG_BUF_LEN)
25786             count = LOG_BUF_LEN;
25787         if (count > logged_chars)
25788             count = logged_chars;
25789         j = log_start + log_size - count;
25790         __restore_flags(flags);
25791         for (i = 0; i < count; i++) {
25792             c = *((char *) log_buf + (j++ & (LOG_BUF_LEN - 1)));
25793             __put_user(c, buf++);
25794         }
25795         if (do_clear)
25796             logged_chars = 0;
25797         error = i;
25798         break;
25799     case 5:          /* Clear ring buffer */
25800         logged_chars = 0;

```

```

25801     break;
25802     case 6:         /* Disable logging to console */
25803         console_loglevel = minimum_console_loglevel;
25804         break;
25805     case 7:         /* Enable logging to console */
25806         console_loglevel = default_console_loglevel;
25807         break;
25808     case 8:
25809         error = -EINVAL;
25810         if (len < 1 || len > 8)
25811             goto out;
25812         if (len < minimum_console_loglevel)
25813             len = minimum_console_loglevel;
25814         console_loglevel = len;
25815         error = 0;
25816         break;
25817     default:
25818         error = -EINVAL;
25819         break;
25820     }
25821 out:
25822     unlock_kernel();
25823     return error;
25824 }
25825
25826 asmlinkage int sys_syslog(int type, char * buf, int len)
25827 {
25828     if ((type != 3) && !capable(CAP_SYS_ADMIN))
25829         return -EPERM;
25830     return do_syslog(type, buf, len);
25831 }
25832
25833
25834 spinlock_t console_lock;
25835
25836 asmlinkage int printk(const char *fmt, ...)
25837 {
25838     va_list args;
25839     int i;
25840     char *msg, *p, *buf_end;
25841     int line_feed;
25842     static signed char msg_level = -1;
25843     long flags;
25844
25845     spin_lock_irqsave(&console_lock, flags);
25846     va_start(args, fmt);
25847     /* hopefully i < sizeof(buf) 4 */
25848     i = vsprintf(buf + 3, fmt, args);

```

▶ 125

```

25849     buf_end = buf + 3 + i;
25850     va_end(args);
25851     for (p = buf + 3; p < buf_end; p++) {
25852         msg = p;
25853         if (msg_level < 0) {
25854             if (
25855                 p[0] != '<' ||
25856                 p[1] < '0' ||
25857                 p[1] > '7' ||
25858                 p[2] != '>'
25859             ) {
25860                 p -= 3;
25861                 p[0] = '<';
25862                 p[1] = default_message_loglevel + '0';
25863                 p[2] = '>';
25864             } else
25865                 msg += 3;
25866             msg_level = p[1] - '0';
25867         }
25868         line_feed = 0;
25869         for (; p < buf_end; p++) {
25870             log_buf[(log_start+log_size) & (LOG_BUF_LEN-1)]
25871                 = *p;
25872             if (log_size < LOG_BUF_LEN)
25873                 log_size++;
25874             else {
25875                 log_start++;
25876                 log_start &= LOG_BUF_LEN-1;
25877             }
25878             logged_chars++;
25879             if (*p == '\n') {
25880                 line_feed = 1;
25881                 break;
25882             }
25883         }
25884         if (msg_level < console_loglevel && console_drivers) {
25885             struct console *c = console_drivers;
25886             while(c) {
25887                 if ((c->flags & CON_ENABLED) && c->write)
25888                     c->write(c, msg, p - msg + line_feed);
25889                 c = c->next;
25890             }
25891         }
25892         if (line_feed)
25893             msg_level = -1;
25894     }
25895     spin_unlock_irqrestore(&console_lock, flags);
25896     wake_up_interruptible(&log_wait);

```

```

25897 return i;
25898 }
25899
25900 void console_print(const char *s)
25901 {
25902     struct console *c = console_drivers;
25903     int len = strlen(s);
25904
25905     while(c) {
25906         if ((c->flags & CON_ENABLED) && c->write)
25907             c->write(c, s, len);
25908         c = c->next;
25909     }
25910 }
25911
25912 void unblank_console(void)
25913 {
25914     struct console *c = console_drivers;
25915     while(c) {
25916         if ((c->flags & CON_ENABLED) && c->unblank)
25917             c->unblank();
25918         c = c->next;
25919     }
25920 }
25921
25922 /* The console driver calls this routine during kernel
25923  * initialization to register the console printing
25924  * procedure with printk() and to print any messages that
25925  * were printed by the kernel before the console driver
25926  * was initialized. */
25927 void register_console(struct console * console)
25928 {
25929     int i, j, len;
25930     int p = log_start;
25931     char buf[16];
25932     signed char msg_level = -1;
25933     char *q;
25934
25935     /* See if we want to use this console driver. If we
25936      * didn't select a console we take the first one that
25937      * registers here. */
25938     if (preferred_console < 0) {
25939         if (console->index < 0)
25940             console->index = 0;
25941         if (console->setup == NULL ||
25942             console->setup(console, NULL) == 0) {
25943             console->flags |= CON_ENABLED | CON_CONSDEV;
25944             preferred_console = 0;

```

```

25945     }
25946 }
25947
25948 /* See if this console matches one we selected on the
25949  * command line. */
25950 for (i = 0;
25951      i < MAX_CMDLINECONSOLES &&
25952      console_cmdline[i].name[0];
25953      i++) {
25954     if (strcmp(console_cmdline[i].name, console->name))
25955         continue;
25956     if (console->index >= 0 &&
25957         console->index != console_cmdline[i].index)
25958         continue;
25959     if (console->index < 0)
25960         console->index = console_cmdline[i].index;
25961     if (console->setup &&
25962         console->setup(console,
25963                       console_cmdline[i].options) != 0)
25964         break;
25965     console->flags |= CON_ENABLED;
25966     console->index = console_cmdline[i].index;
25967     if (i == preferred_console)
25968         console->flags |= CON_CONSDEV;
25969     break;
25970 }
25971
25972 if (!(console->flags & CON_ENABLED))
25973     return;
25974
25975 /* Put this console in the list - keep the preferred
25976  * driver at the head of the list. */
25977 if ((console->flags & CON_CONSDEV) ||
25978     console_drivers == NULL) {
25979     console->next = console_drivers;
25980     console_drivers = console;
25981 } else {
25982     console->next = console_drivers->next;
25983     console_drivers->next = console;
25984 }
25985 if ((console->flags & CON_PRINTBUFFER) == 0) return;
25986
25987 /* Print out buffered log messages. */
25988 for (i=0, j=0; i < log_size; i++) {
25989     buf[j++] = log_bufp[i];
25990     p++; p &= LOG_BUF_LEN-1;
25991     if (buf[j-1] != '\n' && i < log_size - 1 &&
25992         j < sizeof(buf)-1)

```

```

25993     continue;
25994     buf[j] = 0;
25995     q = buf;
25996     len = j;
25997     if (msg_level < 0) {
25998         msg_level = buf[1] - '0';
25999         q = buf + 3;
26000         len -= 3;
26001     }
26002     if (msg_level < console_loglevel)
26003         console->write(console, q, len);
26004     if (buf[j-1] == '\n')
26005         msg_level = -1;
26006     j = 0;
26007 }
26008 }
26009
26010
26011 int unregister_console(struct console * console)
26012 {
26013     struct console *a,*b;
26014
26015     if (console_drivers == console) {
26016         console_drivers=console->next;
26017         return (0);
26018     }
26019     for (a = console_drivers->next, b = console_drivers;
26020          a; b = a, a = b->next) {
26021         if (a == console) {
26022             b->next = a->next;
26023             return 0;
26024         }
26025     }
26026
26027     return (1);
26028 }
26029
26030 /* Write a message to a certain tty, not just the
26031  * console. This is used for messages that need to be
26032  * redirected to a specific tty. We don't put it into
26033  * the syslog queue right now maybe in the future if
26034  * really needed. */
26035 void tty_write_message(struct tty_struct *tty, char *msg)
26036 {
26037     if (tty && tty->driver.write)
26038         tty->driver.write(tty, 0, msg, strlen(msg));
26039     return;
26040 }

```

kernel/sched.c

```

26041 /*
26042  * linux/kernel/sched.c
26043  *
26044  * Copyright (C) 1991, 1992 Linus Torvalds
26045  *
26046  * 1996-12-23 Modified by Dave Grothe to fix bugs in
26047  * semaphores and make semaphores SMP safe
26048  * 1997-01-28 Modified by Finn Arne Gangstad to make
26049  * timers scale better.
26050  * 1997-09-10 Updated NTP code according to technical
26051  * memorandum Jan '96 "A Kernel Model for Precision
26052  * Timekeeping" by Dave Mills
26053  * 1998-11-19 Implemented schedule_timeout() and related
26054  * stuff by Andrea Arcangeli
26055  * 1998-12-24 Fixed a xtime SMP race (we need the
26056  * xtime_lock rw spinlock to serialize accesses to
26057  * xtime/lost_ticks). Copyright (C) 1998 Andrea
26058  * Arcangeli
26059  * 1998-12-28 Implemented better SMP scheduling by Ingo
26060  * Molnar
26061  * 1999-03-10 Improved NTP compatibility by Ulrich Windl
26062  */
26063
26064 /* 'sched.c' is the main kernel file. It contains
26065  * scheduling primitives (sleep_on, wakeup, schedule etc)
26066  * as well as a number of simple system call functions
26067  * (type getpid()), which just extract a field from
26068  * current-task */
26069
26070 #include <linux/mm.h>
26071 #include <linux/kernel_stat.h>
26072 #include <linux/fdreg.h>
26073 #include <linux/delay.h>
26074 #include <linux/interrupt.h>
26075 #include <linux/smp_lock.h>
26076 #include <linux/init.h>
26077
26078 #include <asm/io.h>
26079 #include <asm/uaccess.h>
26080 #include <asm/pgtable.h>
26081 #include <asm/mmu_context.h>
26082 #include <asm/semaphore-helper.h>
26083
26084 #include <linux/timex.h>
26085
26086 /* kernel variables */
26087

```

```

26088 /* systemwide security settings */
26089 unsigned securebits = SECUREBITS_DEFAULT;
26090
26091 /* timer interrupt period */
26092 long tick = (1000000 + HZ/2) / HZ;
26093
26094 /* The current time */
26095 volatile struct timeval
26096     xtime __attribute__((aligned(16)));
26097
26098 /* Don't completely fail for HZ > 500. */
26099 int tickadj = 500/HZ ? : 1; /* microsecs */
26100
26101 DECLARE_TASK_QUEUE(tq_timer);
26102 DECLARE_TASK_QUEUE(tq_immediate);
26103 DECLARE_TASK_QUEUE(tq_scheduler);
26104
26105 /* phase-lock loop variables */
26106 /* TIME_ERROR prevents overwriting the CMOS clock */
26107 /* clock synchronization status */
26108 int time_state = TIME_OK;
26109 /* clock status bits */
26110 int time_status = STA_UNSYNC;
26111 /* time adjustment (us) */
26112 long time_offset = 0;
26113 /* pll time constant */
26114 long time_constant = 2;
26115 /* frequency tolerance (ppm) */
26116 long time_tolerance = MAXFREQ;
26117 /* clock precision (us) */
26118 long time_precision = 1;
26119 /* maximum error (us) */
26120 long time_maxerror = NTP_PHASE_LIMIT;
26121 /* estimated error (us) */
26122 long time_esterror = NTP_PHASE_LIMIT;
26123 /* phase offset (scaled us) */
26124 long time_phase = 0;
26125 /* frequency offset (scaled ppm) */
26126 long time_freq =
26127     ((1000000 + HZ/2) % HZ - HZ/2) << SHIFT_USEC;
26128 /* tick adjust (scaled 1 / HZ) */
26129 long time_adj = 0;
26130 /* time at last adjustment (s) */
26131 long time_reftime = 0;
26132
26133 long time_adjust = 0;
26134 long time_adjust_step = 0;
26135
26136 unsigned long event = 0;
26137
26138 extern int do_setitimer(int, struct itimerval *,
26139                       struct itimerval *);
26140 unsigned int * prof_buffer = NULL;
26141 unsigned long prof_len = 0;
26142 unsigned long prof_shift = 0;
26143
26144 extern void mem_use(void);
26145
26146 unsigned long volatile jiffies=0;
26147
26148 /* Init task must be ok at boot for the ix86 as we will
26149  * check its signals via the SMP irq return path. */
26150 struct task_struct * task[NR_TASKS] = {&init_task, };
26151
26152 struct kernel_stat kstat = { 0 };
26153
26154 void scheduling_functions_start_here(void) { }
26155
26156 #ifdef __SMP__
26157 static void reschedule_idle_slow(struct task_struct * p)
26158 {
26159     /* (see reschedule_idle() for an explanation first ...)
26160     *
26161     * Pass #2
26162     *
26163     * We try to find another (idle) CPU for this woken-up
26164     * process.
26165     *
26166     * On SMP, we mostly try to see if the CPU the task used
26167     * to run on is idle.. but we will use another idle CPU
26168     * too, at this point we already know that this CPU is
26169     * not willing to reschedule in the near future.
26170     *
26171     * An idle CPU is definitely wasted, especially if this
26172     * CPU is running long-timeslice processes. The following
26173     * algorithm is pretty good at finding the best idle CPU
26174     * to send this process to.
26175     *
26176     * [We can try to preempt low-priority processes on other
26177     * CPUs in 2.3. Also we can try to use the avg_slice
26178     * value to predict 'likely reschedule' events even on
26179     * other CPUs.] */
26180     int best_cpu = p->processor;
26181     this_cpu = smp_processor_id();
26182     struct task_struct **idle = task, *tsk, *target_tsk;
26183     int i = smp_num_cpus;

```

p 609


```

26184
26185 target_tsk = NULL;
26186 do {
26187     tsk = *idle;
26188     idle++;
26189     if (tsk->has_cpu) {
26190         if (tsk->processor == this_cpu)
26191             continue;
26192         target_tsk = tsk;
26193         if (tsk->processor == best_cpu) {
26194             /* bingo, we couldn't get a better CPU, activate
26195              * it. */
26196             goto send; /* this one helps GCC ... */
26197         }
26198     }
26199 } while (--i > 0);
26200
26201 /* found any idle CPU? */
26202 if (target_tsk) {
26203     send:
26204     target_tsk->need_resched = 1;
26205     smp_send_reschedule(target_tsk->processor);
26206     return;
26207 }
26208 }
26209 #endif /* __SMP__ */
26210
26211 /* If there is a dependency between p1 and p2, don't be
26212  * too eager to go into the slow schedule. In
26213  * particular, if p1 and p2 both want the kernel lock,
26214  * there is no point in trying to make them extremely
26215  * parallel..
26216  *
26217  * (No lock - lock_depth < 0) */
26218 #define related(p1,p2)
26219 ((p1)->lock_depth >= 0 && (p2)->lock_depth >= 0)
26220
26221 static inline void reschedule_idle(
26222     struct task_struct * p)
26223 {
26224
26225     if (p->policy != SCHED_OTHER ||
26226         p->counter > current->counter + 3) {
26227         current->need_resched = 1;
26228         return;
26229     }
26230
26231 #ifdef __SMP__

```

→ p 610

→ p 608

```

26232 /* ("wakeup()") should not be called before we've
26233  * initialized SMP completely. Basically a not-yet
26234  * initialized SMP subsystem can be considered as a
26235  * not-yet working scheduler, simply dont use it before
26236  * it's up and running ...)
26237  *
26238  * SMP rescheduling is done in 2 passes:
26239  * - pass #1: faster: quick decisions
26240  * - pass #2: slower: let's try to find another CPU */
26241
26242 /* Pass #1
26243  *
26244  * There are two metrics here:
26245  *
26246  * first, a 'cutoff' interval, currently 0-200 usecs on
26247  * x86 CPUs, depending on the size of the 'SMP-local
26248  * cache'. If the current process has longer average
26249  * timeslices than this, then we utilize the idle CPU.
26250  *
26251  * second, if the wakeup comes from a process context,
26252  * then the two processes are 'related'. (they form a
26253  * 'gang')
26254  *
26255  * An idle CPU is almost always a bad thing, thus we
26256  * skip the idle-CPU utilization only if both these
26257  * conditions are true. (ie. a 'process-gang'
26258  * rescheduling with rather high frequency should stay
26259  * on the same CPU).
26260  *
26261  * [We can switch to something more finegrained in
26262  * 2.3.] */
26263 if ((current->avg_slice < cacheflush_time) &&
26264     related(current, p))
26265     return;
26266
26267 reschedule_idle_slow(p);
26268 #endif /* __SMP__ */
26269 }
26270
26271 /* Careful!
26272  *
26273  * This has to add the process to the _beginning_ of the
26274  * run-queue, not the end. See the comment about "This is
26275  * subtle" in the scheduler proper.. */
26276 static inline void add_to_runqueue(struct task_struct *p)
26277 {
26278     struct task_struct *next = init_task.next_run;
26279

```

```

26280 p->prev_run = &init_task;
26281 init_task.next_run = p;
26282 p->next_run = next;
26283 next->prev_run = p;
26284 nr_running++;
26285 }
26286
26287 static inline void del_from_runqueue(
26288     struct task_struct * p)
26289 {
26290     struct task_struct *next = p->next_run;
26291     struct task_struct *prev = p->prev_run;
26292
26293     nr_running--;
26294     next->prev_run = prev;
26295     prev->next_run = next;
26296     p->next_run = NULL;
26297     p->prev_run = NULL;
26298 }
26299
26300 static inline void move_last_runqueue(
26301     struct task_struct * p)
26302 {
26303     struct task_struct *next = p->next_run;
26304     struct task_struct *prev = p->prev_run;
26305
26306     /* remove from list */
26307     next->prev_run = prev;
26308     prev->next_run = next;
26309     /* add back to list */
26310     p->next_run = &init_task;
26311     prev = init_task.prev_run;
26312     init_task.prev_run = p;
26313     p->prev_run = prev;
26314     prev->next_run = p;
26315 }
26316
26317 static inline void
26318 move_first_runqueue(struct task_struct * p)
26319 {
26320     struct task_struct *next = p->next_run;
26321     struct task_struct *prev = p->prev_run;
26322
26323     /* remove from list */
26324     next->prev_run = prev;
26325     prev->next_run = next;
26326     /* add back to list */
26327     p->prev_run = &init_task;

```

```

26328     next = init_task.next_run;
26329     init_task.next_run = p;
26330     p->next_run = next;
26331     next->prev_run = p;
26332 }
26333
26334 /* The tasklist_lock protects the linked list of
26335 * processes.
26336 *
26337 * The scheduler lock is protecting against multiple
26338 * entry into the scheduling code, and doesn't need to
26339 * worry about interrupts (because interrupts cannot call
26340 * the scheduler).
26341 *
26342 * The run-queue lock locks the parts that actually
26343 * access and change the run-queues, and have to be
26344 * interrupt-safe. */
26345 /* should be acquired first */
26346 spinlock_t scheduler_lock = SPIN_LOCK_UNLOCKED;
26347 spinlock_t runqueue_lock = SPIN_LOCK_UNLOCKED; /* 2nd */
26348 rwlock_t tasklist_lock = RW_LOCK_UNLOCKED; /* 3rd */
26349
26350 /* Wake up a process. Put it on the run-queue if it's not
26351 * already there. The "current" process is always on the
26352 * run-queue (except when the actual re-schedule is in
26353 * progress), and as such you're allowed to do the
26354 * simpler "current->state = TASK_RUNNING" to mark
26355 * yourself runnable without the overhead of this. */
26356 void wake_up_process(struct task_struct * p)
26357 {
26358     unsigned long flags;
26359
26360     spin_lock_irqsave(&runqueue_lock, flags);
26361     p->state = TASK_RUNNING;
26362     if (!p->next_run) {
26363         add_to_runqueue(p);
26364         reschedule_idle(p);
26365     }
26366     spin_unlock_irqrestore(&runqueue_lock, flags);
26367 }
26368
26369 static void process_timeout(unsigned long __data)
26370 {
26371     struct task_struct * p = (struct task_struct *) __data;
26372
26373     wake_up_process(p);
26374 }
26375

```

```

26376 /* This is the function that decides how desirable a
26377 * process is.. You can weigh different processes
26378 * against each other depending on what CPU they've run
26379 * on lately etc to try to handle cache and TLB miss
26380 * penalties.
26381 *
26382 * Return values:
26383 *     -1000: never select this
26384 *         0: out of time, recalculate counters
26385 *         (but it might still be selected)
26386 *     +ve: "goodness" value (the larger, the better)
26387 *     +1000: realtime process, select this. */
26388 static inline int goodness(struct task_struct * p,
26389 struct task_struct * prev, int this_cpu)
26390 {
26391     int policy = p->policy;
26392     int weight;
26393
26394     if (policy & SCHED_YIELD) {
26395         p->policy = policy & ~SCHED_YIELD;
26396         return 0;
26397     }
26398
26399     /* Realtime process, select the first one on the
26400     * runqueue (taking priorities within processes into
26401     * account). */
26402     if (policy != SCHED_OTHER)
26403         return 1000 + p->rt_priority;
26404
26405     /* Give the process a first-approximation goodness
26406     * value according to the number of clock-ticks it has
26407     * left.
26408     *
26409     * Don't do any other calculations if the time slice is
26410     * over.. */
26411     weight = p->counter;
26412     if (weight) {
26413
26414 #ifdef __SMP__
26415         /* Give a largish advantage to the same processor...
26416         * (this is equivalent to penalizing other
26417         * processors) */
26418         if (p->processor == this_cpu)
26419             weight += PROC_CHANGE_PENALTY;
26420 #endif
26421
26422         /* .. and a slight advantage to the current thread */
26423         if (p->mm == prev->mm)

```



```

26424         weight += 1;
26425         weight += p->priority;
26426     }
26427
26428     return weight;
26429 }
26430
26431 /* Event timer code */
26432 #define TVN_BITS 6
26433 #define TVR_BITS 8
26434 #define TVN_SIZE (1 << TVN_BITS)
26435 #define TVR_SIZE (1 << TVR_BITS)
26436 #define TVN_MASK (TVN_SIZE - 1)
26437 #define TVR_MASK (TVR_SIZE - 1)
26438
26439 struct timer_vec {
26440     int index;
26441     struct timer_list *vec[TVN_SIZE];
26442 };
26443
26444 struct timer_vec_root {
26445     int index;
26446     struct timer_list *vec[TVR_SIZE];
26447 };
26448
26449 static struct timer_vec tv5 = { 0 };
26450 static struct timer_vec tv4 = { 0 };
26451 static struct timer_vec tv3 = { 0 };
26452 static struct timer_vec tv2 = { 0 };
26453 static struct timer_vec_root tv1 = { 0 };
26454
26455 static struct timer_vec * const tvecs[] = {
26456     (struct timer_vec *)&tv1, &tv2, &tv3, &tv4, &tv5
26457 };
26458
26459 #define N00F_TVECS (sizeof(tvecs) / sizeof(tvecs[0]))
26460
26461 static unsigned long timer_jiffies = 0;
26462
26463 static inline void insert_timer(struct timer_list *timer,
26464 struct timer_list **vec, int idx)
26465 {
26466     if ((timer->next = vec[idx]))
26467         vec[idx]->prev = timer;
26468     vec[idx] = timer;
26469     timer->prev = (struct timer_list *)&vec[idx];
26470 }
26471

```

```

26472 static inline void internal_add_timer(
26473     struct timer_list *timer)
26474 {
26475     /* must be cli-ed when calling this */
26476     unsigned long expires = timer->expires;
26477     unsigned long idx = expires - timer_jiffies;
26478
26479     if (idx < TVR_SIZE) {
26480         int i = expires & TVR_MASK;
26481         insert_timer(timer, tv1.vec, i);
26482     } else if (idx < 1 << (TVR_BITS + TVN_BITS)) {
26483         int i = (expires >> TVR_BITS) & TVN_MASK;
26484         insert_timer(timer, tv2.vec, i);
26485     } else if (idx < 1 << (TVR_BITS + 2 * TVN_BITS)) {
26486         int i = (expires >> (TVR_BITS + TVN_BITS)) & TVN_MASK;
26487         insert_timer(timer, tv3.vec, i);
26488     } else if (idx < 1 << (TVR_BITS + 3 * TVN_BITS)) {
26489         int i =
26490             (expires >> (TVR_BITS + 2 * TVN_BITS)) & TVN_MASK;
26491         insert_timer(timer, tv4.vec, i);
26492     } else if ((signed long) idx < 0) {
26493         /* can happen if you add a timer with expires ==
26494          * jiffies. or you set a timer to go off in the past
26495          */
26496         insert_timer(timer, tv1.vec, tv1.index);
26497     } else if (idx <= 0xffffffffUL) {
26498         int i =
26499             (expires >> (TVR_BITS + 3 * TVN_BITS)) & TVN_MASK;
26500         insert_timer(timer, tv5.vec, i);
26501     } else {
26502         /* Can only get here on architectures with 64-bit
26503          * jiffies */
26504         timer->next = timer->prev = timer;
26505     }
26506 }
26507
26508 spinlock_t timerlist_lock = SPIN_LOCK_UNLOCKED;
26509
26510 void add_timer(struct timer_list *timer)
26511 {
26512     unsigned long flags;
26513
26514     spin_lock_irqsave(&timerlist_lock, flags);
26515     if (timer->prev)
26516         goto bug;
26517     internal_add_timer(timer);
26518 out:
26519     spin_unlock_irqrestore(&timerlist_lock, flags);

```

```

26520     return;
26521
26522 bug:
26523     printk("bug: kernel timer added twice at %p.\n",
26524         __builtin_return_address(0));
26525     goto out;
26526 }
26527
26528 static inline int detach_timer(struct timer_list *timer)
26529 {
26530     struct timer_list *prev = timer->prev;
26531     if (prev) {
26532         struct timer_list *next = timer->next;
26533         prev->next = next;
26534         if (next)
26535             next->prev = prev;
26536         return 1;
26537     }
26538     return 0;
26539 }
26540
26541 void mod_timer(struct timer_list *timer,
26542               unsigned long expires)
26543 {
26544     unsigned long flags;
26545
26546     spin_lock_irqsave(&timerlist_lock, flags);
26547     timer->expires = expires;
26548     detach_timer(timer);
26549     internal_add_timer(timer);
26550     spin_unlock_irqrestore(&timerlist_lock, flags);
26551 }
26552
26553 int del_timer(struct timer_list * timer)
26554 {
26555     int ret;
26556     unsigned long flags;
26557
26558     spin_lock_irqsave(&timerlist_lock, flags);
26559     ret = detach_timer(timer);
26560     timer->next = timer->prev = 0;
26561     spin_unlock_irqrestore(&timerlist_lock, flags);
26562     return ret;
26563 }
26564
26565 #ifdef __SMP__
26566 #define idle_task (task[cpu_number_map[this cpu]])

```

```

26568 #define can_schedule(p) (!(p)->has_cpu)
26569
26570 #else
26571
26572 #define idle_task (&init_task)
26573 #define can_schedule(p) (1)
26574
26575 #endif
26576
26577 signed long schedule_timeout(signed long timeout)
26578 {
26579     struct timer_list timer;
26580     unsigned long expire;
26581
26582     switch (timeout)
26583     {
26584     case MAX_SCHEDULE_TIMEOUT:
26585         /* These two special cases are useful to be
26586          * comfortable in the caller. Nothing more. We could
26587          * take MAX_SCHEDULE_TIMEOUT from one of the negative
26588          * value but I'd like to return a valid offset (>=0)
26589          * to allow the caller to do everything it want with
26590          * the retval. */
26591         schedule();
26592         goto out;
26593     default:
26594         /* Another bit of PARANOID. Note that the retval will
26595          * be 0 since no piece of kernel is supposed to do a
26596          * check for a negative retval of schedule_timeout()
26597          * (since it should never happens anyway). You just
26598          * have the printk() that will tell you if something
26599          * is gone wrong and where. */
26600         if (timeout < 0)
26601         {
26602             printk(KERN_ERR "schedule_timeout: wrong timeout "
26603                    "value %lx from %p\n", timeout,
26604                    __builtin_return_address(0));
26605             goto out;
26606         }
26607     }
26608
26609     expire = timeout + jiffies;
26610
26611     init_timer(&timer);
26612     timer.expires = expire;
26613     timer.data = (unsigned long) current;
26614     timer.function = process_timeout;
26615

```

```

26616     add_timer(&timer);
26617     schedule();
26618     del_timer(&timer);
26619
26620     timeout = expire - jiffies;
26621
26622     out:
26623     return timeout < 0 ? 0 : timeout;
26624 }
26625
26626 /* This one aligns per-CPU data on cacheline boundaries.
26627 */
26628 static union {
26629     struct schedule_data {
26630         struct task_struct * prev;
26631         long prevstate;
26632         cycles_t last_schedule;
26633     } schedule_data;
26634     char __pad [SMP_CACHE_BYTES];
26635 } aligned_data [NR_CPUS] __cacheline_aligned =
26636 { {&init_task,0}};
26637
26638 static inline void __schedule_tail (void)
26639 {
26640 #ifdef __SMP__
26641     struct schedule_data * sched_data;
26642
26643     /* We might have switched CPUs: */
26644     sched_data =
26645         &aligned_data[smp_processor_id()].schedule_data;
26646
26647     /* Subtle. In the rare event that we got a wakeup to
26648      * 'prev' just during the reschedule (this is possible,
26649      * the scheduler is pretty parallel), we should do
26650      * another reschedule in the next task's
26651      * context. schedule() will do the right thing next
26652      * time around. This is equivalent to 'delaying' the
26653      * wakeup until the reschedule has finished. */
26654     if (sched_data->prev->state != sched_data->prevstate)
26655         current->need_resched = 1;
26656
26657     /* Release the previous process ...
26658      *
26659      * We have dropped all locks, and we must make sure
26660      * that we only mark the previous process as no longer
26661      * having a CPU after all other state has been seen by
26662      * other CPUs. Thus the write memory barrier! */
26663     wmb();

```

→ p 608

```

26664 sched_data->prev->has_cpu = 0;
26665 #endif /* __SMP__ */
26666 }
26667
26668 /* schedule_tail() is getting called from the fork return
26669 * path. This cleans up all remaining scheduler things,
26670 * without impacting the common case. */
26671 void schedule_tail (void)
26672 {
26673     __schedule_tail();
26674 }
26675
26676 /* 'schedule()' is the scheduler function. It's a very
26677 * simple and nice scheduler: it's not perfect, but
26678 * certainly works for most things.
26679 *
26680 * The goto is "interesting".
26681 *
26682 * NOTE!! Task 0 is the 'idle' task, which gets called
26683 * when no other tasks can run. It can not be killed, and
26684 * it cannot sleep. The 'state' information in task[0] is
26685 * never used. */
26686 asmlinkage void schedule(void)
26687 {
26688     struct schedule_data * sched_data;
26689     struct task_struct * prev, * next;
26690     int this_cpu;
26691
26692     run_task_queue(&tq_scheduler);
26693
26694     prev = current;
26695     this_cpu = prev->processor;
26696     /* 'sched_data' is protected by the fact that we can
26697      * run only one process per CPU. */
26698     sched_data = & aligned_data[this_cpu].schedule_data;
26699
26700     if (in_interrupt())
26701         goto scheduling_in_interrupt;
26702     release_kernel_lock(prev, this_cpu);
26703
26704     /* Do "administrative" work here while we don't hold
26705      * any locks */
26706     if (bh_active & bh_mask)
26707         do_bottom_half();
26708
26709     spin_lock(&scheduler_lock);
26710     spin_lock_irq(&runqueue_lock);
26711

```

```

26712     /* move an exhausted RR process to be last.. */
26713     prev->need_resched = 0;
26714
26715     if (!prev->counter && prev->policy == SCHED_RR) {
26716         prev->counter = prev->priority;
26717         move_last_runqueue(prev);
26718     }
26719
26720     switch (prev->state) {
26721     case TASK_INTERRUPTIBLE:
26722         if (signal_pending(prev)) {
26723             prev->state = TASK_RUNNING;
26724             break;
26725         }
26726     default:
26727         del_from_runqueue(prev);
26728     case TASK_RUNNING:
26729     }
26730
26731     sched_data->prevstate = prev->state;
26732
26733     /* this is the scheduler proper: */
26734     {
26735         struct task_struct * p = init_task.next_run;
26736         int c = -1000;
26737
26738         /* Default process to select.. */
26739         next = idle_task;
26740         if (prev->state == TASK_RUNNING) {
26741             c = goodness(prev, prev, this_cpu);
26742             next = prev;
26743         }
26744
26745         /* This is subtle. Note how we can enable interrupts
26746          * here, even though interrupts can add processes to
26747          * the run queue. This is because any new processes
26748          * will be added to the front of the queue, so "p"
26749          * above is a safe starting point. run-queue
26750          * deletion and re-ordering is protected by the
26751          * scheduler lock */
26752         spin_unlock_irq(&runqueue_lock);
26753         /* Note! there may appear new tasks on the run-queue
26754          * during this, as interrupts are enabled. However, they
26755          * will be put on front of the list, so our list starting
26756          * at "p" is essentially fixed. */
26757         while (p != &init_task) {
26758             if (can_schedule(p)) {
26759                 int weight = goodness(p, prev, this_cpu);

```

```

26760     if (weight > c)
26761         c = weight, next = p;
26762     }
26763     p = p->next_run;
26764 }
26765
26766 /* Do we need to re-calculate counters? */
26767 if (!c) {
26768     struct task_struct *p;
26769     read_lock(&tasklist_lock);
26770     for_each_task(p)
26771         p->counter = (p->counter >> 1) + p->priority;
26772     read_unlock(&tasklist_lock);
26773 }
26774 }
26775
26776 /* maintain the per-process 'average timeslice' value.
26777  * (this has to be recalculated even if we reschedule
26778  * to the same process) Currently this is only used on
26779  * SMP: */
26780 #ifdef __SMP__
26781 {
26782     cycles_t t, this_slice;
26783
26784     t = get_cycles();
26785     this_slice = t - sched_data->last_schedule;
26786     sched_data->last_schedule = t;
26787
26788     /* Simple, exponentially fading average calculation:
26789     */
26790     prev->avg_slice = this_slice + prev->avg_slice;
26791     prev->avg_slice >>= 1;
26792 }
26793
26794 /* We drop the scheduler lock early (it's a global
26795  * spinlock), thus we have to lock the previous process
26796  * from getting rescheduled during switch_to(). */
26797 next->processor = this_cpu;
26798 next->has_cpu = 1;
26799 spin_unlock(&scheduler_lock);
26800 #endif /* __SMP__ */
26801 if (prev != next) {
26802 #ifdef __SMP__
26803     sched_data->prev = prev;
26804 #endif
26805     kstat.context_swch++;
26806     get_mmu_context(next);
26807     switch_to(prev,next);

```

→ 208

→ 209

```

26808
26809     __schedule_tail();
26810 }
26811
26812 reacquire_kernel_lock(current);
26813 return;
26814
26815 scheduling_in_interrupt:
26816 printk("Scheduling in interrupt\n");
26817 *(int *)0 = 0;
26818 }
26819
26820 rwlock_t waitqueue_lock = RW_LOCK_UNLOCKED;
26821
26822 /* wake_up doesn't wake up stopped processes - they have
26823  * to be awakened with signals or similar.
26824  */
26825 * Note that we only need a read lock for the wait queue
26826 * (and thus do not have to protect against interrupts).
26827 * as the actual removal from the queue is handled by the
26828 * process itself. */
26829 void __wake_up(struct wait_queue **q, unsigned int mode)
26830 {
26831     struct wait_queue *next;
26832
26833     read_lock(&waitqueue_lock);
26834     if (q && (next = *q)) {
26835         struct wait_queue *head;
26836
26837         head = WAIT_QUEUE_HEAD(q);
26838         while (next != head) {
26839             struct task_struct *p = next->task;
26840             next = next->next;
26841             if (p->state & mode)
26842                 wake_up_process(p);
26843         }
26844     }
26845     read_unlock(&waitqueue_lock);
26846 }
26847
26848 /* Semaphores are implemented using a two-way counter:
26849  * The "count" variable is decremented for each process
26850  * that tries to sleep, while the "waking" variable is
26851  * incremented when the "up()" code goes to wake up
26852  * waiting processes.
26853  */
26854 * Notably, the inline "up()" and "down()" functions can
26855 * efficiently test if they need to do any extra work (up

```

→ 131

```

26856 * needs to do something only if count was negative
26857 * before the increment operation.
26858 *
26859 * waking_non_zero() (from asm/semaphore.h) must execute
26860 * atomically.
26861 *
26862 * When __up() is called, the count was negative before
26863 * incrementing it, and we need to wake up somebody.
26864 *
26865 * This routine adds one to the count of processes that
26866 * need to wake up and exit. ALL waiting processes
26867 * actually wake up but only the one that gets to the
26868 * "waking" field first will gate through and acquire the
26869 * semaphore. The others will go back to sleep.
26870 *
26871 * Note that these functions are only called when there
26872 * is contention on the lock, and as such all this is the
26873 * "non-critical" part of the whole semaphore
26874 * business. The critical part is the inline stuff in
26875 * <asm/semaphore.h> where we want to avoid any extra
26876 * jumps and calls. */
p 601 26877 void __up(struct semaphore *sem)
26878 {
26879     wake_one_more(sem);
26880     wake_up(&sem->wait);
26881 }
26882
26883 /* Perform the "down" function. Return zero for
26884 * semaphore acquired, return negative for signalled out
26885 * of the function.
26886 *
26887 * If called from __down, the return is ignored and the
26888 * wait loop is not interruptible. This means that a
26889 * task waiting on a semaphore using "down()" cannot be
26890 * killed until someone does an "up()" on the semaphore.
26891 *
26892 * If called from __down_interruptible, the return value
26893 * gets checked upon return. If the return value is
26894 * negative then the task continues with the negative
26895 * value in the return register (it can be tested by the
26896 * caller).
26897 *
26898 * Either form may be used in conjunction with "up()". */
26899
p 602 26900 #define DOWN_VAR \
26901     struct task_struct *tsk = current; \
26902     struct wait_queue wait = { tsk, NULL };
26903

```

```

p 602 26904 #define DOWN_HEAD(task_state) \
26905 \
26906     tsk->state = (task_state); \
26907     add_wait_queue(&sem->wait, &wait); \
26908 \
26909     /* Ok, we're set up. sem->count is known to be less \
26910      * than zero so we must wait. \
26911      * \
26912      * We can let go the lock for purposes of waiting. \
26913      * We re-acquire it after awaking so as to protect \
26914      * all semaphore operations. \
26915      * \
26916      * If "up()" is called before we call \
26917      * waking_non_zero() then we will catch it right away. \
26918      * If it is called later then we will have to go \
26919      * through a wakeup cycle to catch it. \
26920      * \
26921      * Multiple waiters contend for the semaphore lock to \
26922      * see who gets to gate through and who has to wait \
26923      * some more. */
26924     for (;;) {
26925
p 602 26926 #define DOWN_TAIL(task_state) \
26927     tsk->state = (task_state); \
26928     } \
26929     tsk->state = TASK_RUNNING; \
26930     remove_wait_queue(&sem->wait, &wait);
26931
p 602 26932 void __down(struct semaphore * sem)
26933 {
26934     DOWN_VAR
26935     DOWN_HEAD(TASK_UNINTERRUPTIBLE)
26936     if (waking_non_zero(sem))
26937         break;
26938     schedule();
26939     DOWN_TAIL(TASK_UNINTERRUPTIBLE)
26940 }
26941
p 603 26942 int __down_interruptible(struct semaphore * sem)
26943 {
26944     DOWN_VAR
26945     int ret = 0;
26946     DOWN_HEAD(TASK_INTERRUPTIBLE)
26947
26948     ret = waking_non_zero_interruptible(sem, tsk);
26949     if (ret)
26950     {
26951         if (ret == -1)

```



```

26952     /* ret != 0 only if we get interrupted -arca */
26953     ret = 0;
26954     break;
26955 }
26956 schedule();
26957 DOWN_TAIL(TASK_INTERRUPTIBLE)
26958 return ret;
26959 }
26960
26961 int __down_trylock(struct semaphore * sem)
26962 {
26963     return waking_non_zero_trylock(sem);
26964 }
26965
26966 #define SLEEP_ON_VAR \
26967     unsigned long flags; \
26968     struct wait_queue wait;
26969
26970 #define SLEEP_ON_HEAD \
26971     wait.task = current; \
26972     write_lock_irqsave(&waitqueue_lock, flags); \
26973     __add_wait_queue(p, &wait); \
26974     write_unlock(&waitqueue_lock);
26975
26976 #define SLEEP_ON_TAIL \
26977     write_lock_irq(&waitqueue_lock); \
26978     __remove_wait_queue(p, &wait); \
26979     write_unlock_irqrestore(&waitqueue_lock, flags);
26980
26981 void interruptible_sleep_on(struct wait_queue **p)
26982 {
26983     SLEEP_ON_VAR
26984
26985     current->state = TASK_INTERRUPTIBLE;
26986
26987     SLEEP_ON_HEAD
26988     schedule();
26989     SLEEP_ON_TAIL
26990 }
26991
26992 long interruptible_sleep_on_timeout(
26993     struct wait_queue **p, long timeout)
26994 {
26995     SLEEP_ON_VAR
26996
26997     current->state = TASK_INTERRUPTIBLE;
26998
26999     SLEEP_ON_HEAD

```

→ p.603

```

27000     timeout = schedule_timeout(timeout);
27001     SLEEP_ON_TAIL
27002
27003     return timeout;
27004 }
27005
27006 void sleep_on(struct wait_queue **p)
27007 {
27008     SLEEP_ON_VAR
27009
27010     current->state = TASK_UNINTERRUPTIBLE;
27011
27012     SLEEP_ON_HEAD
27013     schedule();
27014     SLEEP_ON_TAIL
27015 }
27016
27017 long sleep_on_timeout(struct wait_queue **p,
27018     long timeout)
27019 {
27020     SLEEP_ON_VAR
27021
27022     current->state = TASK_UNINTERRUPTIBLE;
27023
27024     SLEEP_ON_HEAD
27025     timeout = schedule_timeout(timeout);
27026     SLEEP_ON_TAIL
27027
27028     return timeout;
27029 }
27030
27031 void scheduling_functions_end_here(void) { }
27032
27033 static inline void cascade_timers(struct timer_vec *tv)
27034 {
27035     /* cascade all the timers from tv up one level */
27036     struct timer_list *timer;
27037     timer = tv->vec[tv->index];
27038     /* We are removing all timers from the list, so we
27039      * don't have to detach them individually, just clear
27040      * the list afterwards. */
27041     while (timer) {
27042         struct timer_list *tmp = timer;
27043         timer = timer->next;
27044         internal_add_timer(tmp);
27045     }
27046     tv->vec[tv->index] = NULL;
27047     tv->index = (tv->index + 1) & TVN_MASK;

```

```

27048 }
27049
27050 static inline void run_timer_list(void)
27051 {
27052     spin_lock_irq(&timerlist_lock);
27053     while ((long)(jiffies - timer_jiffies) >= 0) {
27054         struct timer_list *timer;
27055         if (!tv1.index) {
27056             int n = 1;
27057             do {
27058                 cascade_timers(tvecs[n]);
27059             } while (tvecs[n]->index -- 1 && ++n < NOOF_TVECS);
27060         }
27061         while ((timer = tv1.vec[tv1.index])) {
27062             void (*fn)(unsigned long) = timer->function;
27063             unsigned long data = timer->data;
27064             detach_timer(timer);
27065             timer->next = timer->prev = NULL;
27066             spin_unlock_irq(&timerlist_lock);
27067             fn(data);
27068             spin_lock_irq(&timerlist_lock);
27069         }
27070         ++timer_jiffies;
27071         tv1.index = (tv1.index + 1) & TVR_MASK;
27072     }
27073     spin_unlock_irq(&timerlist_lock);
27074 }
27075
27076
27077 static inline void run_old_timers(void)
27078 {
27079     struct timer_struct *tp;
27080     unsigned long mask;
27081
27082     for (mask = 1, tp = timer_table+0; mask;
27083         tp++, mask += mask) {
27084         if (mask > timer_active)
27085             break;
27086         if (!(mask & timer_active))
27087             continue;
27088         if (time_after(tp->expires, jiffies))
27089             continue;
27090         timer_active &= ~mask;
27091         tp->fn();
27092         sti();
27093     }
27094 }
27095

```

```

27096 spinlock_t tqueue_lock;
27097
27098 void tqueue_bh(void)
27099 {
27100     run_task_queue(&tq_timer);
27101 }
27102
27103 void immediate_bh(void)
27104 {
27105     run_task_queue(&tq_immediate);
27106 }
27107
27108 unsigned long timer_active = 0;
27109 struct timer_struct timer_table[32];
27110
27111 /* Hmm.. Changed this, as the GNU make sources (load.c)
27112  * seems to imply that avenrun[] is the standard name for
27113  * this kind of thing. Nothing else seems to be
27114  * standardized: the fractional size etc all seem to
27115  * differ on different machines. */
27116 unsigned long avenrun[3] = { 0,0,0 };
27117
27118 /* Nr of active tasks - counted in fixed-point numbers */
27119 static unsigned long count_active_tasks(void)
27120 {
27121     struct task_struct *p;
27122     unsigned long nr = 0;
27123
27124     read_lock(&tasklist_lock);
27125     for_each_task(p) {
27126         if ((p->state == TASK_RUNNING ||
27127             p->state == TASK_UNINTERRUPTIBLE ||
27128             p->state == TASK_SWAPPING))
27129             nr += FIXED_1;
27130     }
27131     read_unlock(&tasklist_lock);
27132     return nr;
27133 }
27134
27135 static inline void calc_load(unsigned long ticks)
27136 {
27137     unsigned long active_tasks; /* fixed-point */
27138     static int count = LOAD_FREQ;
27139
27140     count -= ticks;
27141     if (count < 0) {
27142         count += LOAD_FREQ;
27143         active_tasks = count_active_tasks();

```

p 504

p 503

```

27144     CALC_LOAD(avenrun[0], EXP_1, active_tasks);
27145     CALC_LOAD(avenrun[1], EXP_5, active_tasks);
27146     CALC_LOAD(avenrun[2], EXP_15, active_tasks);
27147 }
27148 }
27149
27150 /* this routine handles the overflow of the microsecond
27151 * field
27152 *
27153 * The tricky bits of code to handle the accurate clock
27154 * support were provided by Dave Mills (Mills@UDEL.EDU)
27155 * of NTP fame. They were originally developed for SUN
27156 * and DEC kernels. All the kudos should go to Dave for
27157 * this stuff. */
27158 static void second_overflow(void)
27159 {
27160     long ltemp;
27161
27162     /* Bump the maxerror field */
27163     time_maxerror += time_tolerance >> SHIFT_USEC;
27164     if ( time_maxerror > NTP_PHASE_LIMIT ) {
27165         time_maxerror = NTP_PHASE_LIMIT;
27166         time_status |= STA_UNSYNC;
27167     }
27168
27169     /* Leap second processing. If in leap-insert state at
27170     * the end of the day, the system clock is set back one
27171     * second; if in leap-delete state, the system clock is
27172     * set ahead one second. The microtime() routine or
27173     * external clock driver will insure that reported time
27174     * is always monotonic. The ugly divides should be
27175     * replaced. */
27176     switch (time_state) {
27177
27178     case TIME_OK:
27179         if (time_status & STA_INS)
27180             time_state = TIME_INS;
27181         else if (time_status & STA_DEL)
27182             time_state = TIME_DEL;
27183         break;
27184
27185     case TIME_INS:
27186         if (xtime.tv_sec % 86400 == 0) {
27187             xtime.tv_sec--;
27188             time_state = TIME_OOP;
27189             printk(KERN_NOTICE "Clock: "
27190                  "inserting leap second 23:59:60 UTC\n");
27191         }

```

```

27192         break;
27193
27194     case TIME_DEL:
27195         if ((xtime.tv_sec + 1) % 86400 == 0) {
27196             xtime.tv_sec++;
27197             time_state = TIME_WAIT;
27198             printk(KERN_NOTICE "Clock: "
27199                  "deleting leap second 23:59:59 UTC\n");
27200         }
27201         break;
27202
27203     case TIME_OOP:
27204         time_state = TIME_WAIT;
27205         break;
27206
27207     case TIME_WAIT:
27208         if (!(time_status & (STA_INS | STA_DEL)))
27209             time_state = TIME_OK;
27210     }
27211
27212     /* Compute the phase adjustment for the next second. In
27213     * PLL mode, the offset is reduced by a fixed factor
27214     * times the time constant. In FLL mode the offset is
27215     * used directly. In either mode, the maximum phase
27216     * adjustment for each second is clamped so as to
27217     * spread the adjustment over not more than the number
27218     * of seconds between updates. */
27219     if (time_offset < 0) {
27220         ltemp = -time_offset;
27221         if (!(time_status & STA_FLL))
27222             ltemp >>= SHIFT_KG + time_constant;
27223         if (ltemp > (MAXPHASE / MINSEC) << SHIFT_UPDATE)
27224             ltemp = (MAXPHASE / MINSEC) << SHIFT_UPDATE;
27225         time_offset += ltemp;
27226         time_adj = -ltemp <<
27227             (SHIFT_SCALE - SHIFT_HZ - SHIFT_UPDATE);
27228     } else {
27229         ltemp = time_offset;
27230         if (!(time_status & STA_FLL))
27231             ltemp >>= SHIFT_KG + time_constant;
27232         if (ltemp > (MAXPHASE / MINSEC) << SHIFT_UPDATE)
27233             ltemp = (MAXPHASE / MINSEC) << SHIFT_UPDATE;
27234         time_offset -= ltemp;
27235         time_adj = ltemp <<
27236             (SHIFT_SCALE - SHIFT_HZ - SHIFT_UPDATE);
27237     }
27238
27239     /* Compute the frequency estimate and additional phase

```

```

27240 * adjustment due to frequency error for the next
27241 * second. When the PPS signal is engaged, gnaw on the
27242 * watchdog counter and update the frequency computed
27243 * by the pll and the PPS signal. */
27244 pps_valid++;
27245 if (pps_valid == PPS_VALID) { /* PPS signal lost */
27246     pps_jitter = MAXTIME;
27247     pps_stabil = MAXFREQ;
27248     time_status &= ~(STA_PPSSIGNAL | STA_PPSJITTER |
27249                    STA_PPSWANDER | STA_PPSERROR);
27250 }
27251 ltemp = time_freq + pps_freq;
27252 if (ltemp < 0)
27253     time_adj -= -ltemp >>
27254     (SHIFT_USEC + SHIFT_HZ - SHIFT_SCALE);
27255 else
27256     time_adj += ltemp >>
27257     (SHIFT_USEC + SHIFT_HZ - SHIFT_SCALE);
27258
27259 #if HZ == 100
27260 /* Compensate for (HZ==100) != (1 << SHIFT_HZ). Add
27261 * 25% and 3.125% to get 128.125; => only 0.125% error
27262 * (p. 14) */
27263 if (time_adj < 0)
27264     time_adj -= (-time_adj >> 2) + (-time_adj >> 5);
27265 else
27266     time_adj += (time_adj >> 2) + (time_adj >> 5);
27267 #endif
27268 }
27269
27270 /* in the NTP reference this is called "hardclock()" */
27271 static void update_wall_time_one_tick(void)
27272 {
27273     if ( (time_adjust_step - time_adjust) != 0 ) {
27274         /* We are doing an adjtime thing.
27275          *
27276          * Prepare time_adjust_step to be within bounds.
27277          * Note that a positive time_adjust means we want the
27278          * clock to run faster.
27279          *
27280          * Limit the amount of the step to be in the range
27281          * -tickadj .. +tickadj */
27282         if (time_adjust > tickadj)
27283             time_adjust_step = tickadj;
27284         else if (time_adjust < -tickadj)
27285             time_adjust_step = -tickadj;
27286
27287         /* Reduce by this step the amount of time left */
27288         time_adjust -= time_adjust_step;
27289     }
27290     xtime.tv_usec += tick + time_adjust_step;
27291     /* Advance the phase, once it gets to one microsecond,
27292      * then advance the tick more. */
27293     time_phase += time_adj;
27294     if (time_phase <= -FINEUSEC) {
27295         long ltemp = -time_phase >> SHIFT_SCALE;
27296         time_phase += ltemp << SHIFT_SCALE;
27297         xtime.tv_usec -= ltemp;
27298     }
27299     else if (time_phase >= FINEUSEC) {
27300         long ltemp = time_phase >> SHIFT_SCALE;
27301         time_phase -= ltemp << SHIFT_SCALE;
27302         xtime.tv_usec += ltemp;
27303     }
27304 }
27305
27306 /* Using a loop looks inefficient, but "ticks" is usually
27307 * just one (we shouldn't be losing ticks, we're doing
27308 * this this way mainly for interrupt latency reasons,
27309 * not because we think we'll have lots of lost timer
27310 * ticks */
27311 static void update_wall_time(unsigned long ticks)
27312 {
27313     do {
27314         ticks--;
27315         update_wall_time_one_tick();
27316     } while (ticks);
27317
27318     if (xtime.tv_usec >= 1000000) {
27319         xtime.tv_usec -= 1000000;
27320         xtime.tv_sec++;
27321         second_overflow();
27322     }
27323 }
27324
27325 static inline void do_process_times(
27326     struct task_struct *p, unsigned long user,
27327     unsigned long system)
27328 {
27329     long psecs;
27330
27331     psecs = (p->times.tms_utime += user);
27332     psecs += (p->times.tms_stime += system);
27333     if (psecs / HZ > p->rlim[RLIMIT_CPU].rlim_cur) {
27334         /* Send SIGXCPU every second.. */
27335         if (!(psecs % HZ))

```

```

27336     send_sig(SIGXCPU, p, 1);
27337     /* and SIGKILL when we go over max.. */
27338     if (psecs / HZ > p->rlim[RLIMIT_CPU].rlim_max)
27339         send_sig(SIGKILL, p, 1);
27340 }
27341 }
27342
27343 static inline void do_it_virt(struct task_struct * p,
27344                             unsigned long ticks)
27345 {
27346     unsigned long it_virt = p->it_virt_value;
27347
27348     if (it_virt) {
27349         if (it_virt <= ticks) {
27350             it_virt = ticks + p->it_virt_incr;
27351             send_sig(SIGVTALRM, p, 1);
27352         }
27353         p->it_virt_value = it_virt - ticks;
27354     }
27355 }
27356
27357 static inline void do_it_prof(struct task_struct * p,
27358                              unsigned long ticks)
27359 {
27360     unsigned long it_prof = p->it_prof_value;
27361
27362     if (it_prof) {
27363         if (it_prof <= ticks) {
27364             it_prof = ticks + p->it_prof_incr;
27365             send_sig(SIGPROF, p, 1);
27366         }
27367         p->it_prof_value = it_prof - ticks;
27368     }
27369 }
27370
27371 void update_one_process(struct task_struct *p,
27372                       unsigned long ticks, unsigned long user,
27373                       unsigned long system, int cpu)
27374 {
27375     p->per_cpu_utime[cpu] += user;
27376     p->per_cpu_stime[cpu] += system;
27377     do_process_times(p, user, system);
27378     do_it_virt(p, user);
27379     do_it_prof(p, ticks);
27380 }
27381
27382 static void update_process_times(unsigned long ticks,
27383                                 unsigned long system)

```

```

27384 {
27385     /* SMP does this on a per-CPU basis elsewhere */
27386     #ifndef __SMP__
27387         struct task_struct * p = current;
27388         unsigned long user = ticks - system;
27389         if (p->pid) {
27390             p->counter -= ticks;
27391             if (p->counter < 0) {
27392                 p->counter = 0;
27393                 p->need_resched = 1;
27394             }
27395             if (p->priority < DEF_PRIORITY)
27396                 kstat.cpu_nice += user;
27397             else
27398                 kstat.cpu_user += user;
27399             kstat.cpu_system += system;
27400         }
27401         update_one_process(p, ticks, user, system, 0);
27402     #endif
27403 }
27404
27405 volatile unsigned long lost_ticks = 0;
27406 static unsigned long lost_ticks_system = 0;
27407
27408 /* This spinlock protect us from races in SMP while
27409  * playing with xtime. -arca */
27410 rwlock_t xtime_lock = RW_LOCK_UNLOCKED;
27411
27412 static inline void update_times(void)
27413 {
27414     unsigned long ticks;
27415
27416     /* update_times() is run from the raw timer_bh handler
27417      * so we just know that the irqs are locally enabled
27418      * and so we don't need to save/restore the flags of
27419      * the local CPU here. -arca */
27420     write_lock_irq(&xtime_lock);
27421
27422     ticks = lost_ticks;
27423     lost_ticks = 0;
27424
27425     if (ticks) {
27426         unsigned long system;
27427         system = xchg(&lost_ticks_system, 0);
27428
27429         calc_load(ticks);
27430         update_wall_time(ticks);
27431         write_unlock_irq(&xtime_lock);

```

```

27432
27433     update_process_times(ticks, system);
27434
27435     } else
27436     write_unlock_irq(&xtime_lock);
27437 }
27438
27439 static void timer_bh(void)
27440 {
27441     update_times();
27442     run_old_timers();
27443     run_timer_list();
27444 }
27445
27446 void do_timer(struct pt_regs * regs)
27447 {
27448     (*(unsigned long *)&jiffies)++;
27449     lost_ticks++;
27450     mark_bh(TIMER_BH);
27451     if (!user_mode(regs))
27452         lost_ticks_system++;
27453     if (tq_timer)
27454         mark_bh(TQUEUE_BH);
27455 }
27456
27457 #ifndef __alpha__
27458
27459 /* For backwards compatibility? This can be done in libc
27460  * so Alpha and all newer ports shouldn't need it. */
27461 asmlinkage unsigned int sys_alarm(unsigned int seconds)
27462 {
27463     struct itimerval it_new, it_old;
27464     unsigned int oldalarm;
27465
27466     it_new.it_interval.tv_sec = it_new.it_interval.tv_usec
27467     = 0;
27468     it_new.it_value.tv_sec = seconds;
27469     it_new.it_value.tv_usec = 0;
27470     do_setitimer(ITIMER_REAL, &it_new, &it_old);
27471     oldalarm = it_old.it_value.tv_sec;
27472     /* eh... We can't return 0 if we have an alarm
27473      * pending.. And we'd better return too much than too
27474      * little anyway */
27475     if (it_old.it_value.tv_usec)
27476         oldalarm++;
27477     return oldalarm;
27478 }
27479

```

```

27480 /* The Alpha uses getxpid, getxuid, and getxgid instead.
27481  * Maybe this should be moved into arch/i386 instead? */
27482
27483 asmlinkage int sys_getpid(void)
27484 {
27485     /* This is SMP safe - current->pid doesn't change */
27486     return current->pid;
27487 }
27488
27489 /* This is not strictly SMP safe: p_opptr could change
27490  * from under us. However, rather than getting any lock
27491  * we can use an optimistic algorithm: get the parent
27492  * pid, and go back and check that the parent is still
27493  * the same. If it has changed (which is extremely
27494  * unlikely indeed), we just try again..
27495  *
27496  * NOTE! This depends on the fact that even if we _do_
27497  * get an old value of "parent", we can happily
27498  * dereference the pointer: we just can't necessarily
27499  * trust the result until we know that the parent pointer
27500  * is valid.
27501  *
27502  * The "mb()" macro is a memory barrier - a synchronizing
27503  * event. It also makes sure that gcc doesn't optimize
27504  * away the necessary memory references.. The barrier
27505  * doesn't have to have all that strong semantics: on x86
27506  * we don't really require a synchronizing instruction,
27507  * for example. The barrier is more important for code
27508  * generation than for any real memory ordering semantics
27509  * (even if there is a small window for a race, using the
27510  * old pointer is harmless for a while). */
27511 asmlinkage int sys_getppid(void)
27512 {
27513     int pid;
27514     struct task_struct * me = current;
27515     struct task_struct * parent;
27516
27517     parent = me->p_opptr;
27518     for (;;) {
27519         pid = parent->pid;
27520         #if __SMP__
27521         {
27522             struct task_struct *old = parent;
27523             mb();
27524             parent = me->p_opptr;
27525             if (old != parent)
27526                 continue;
27527 }

```

```

27528 #endif
27529     break;
27530 }
27531 return pid;
27532 }
27533
27534 asmlinkage int sys_getuid(void)
27535 {
27536     /* Only we change this so SMP safe */
27537     return current->uid;
27538 }
27539
27540 asmlinkage int sys_geteuid(void)
27541 {
27542     /* Only we change this so SMP safe */
27543     return current->euid;
27544 }
27545
27546 asmlinkage int sys_getgid(void)
27547 {
27548     /* Only we change this so SMP safe */
27549     return current->gid;
27550 }
27551
27552 asmlinkage int sys_getegid(void)
27553 {
27554     /* Only we change this so SMP safe */
27555     return current->egid;
27556 }
27557
27558 /* This has been replaced by sys_setpriority. Maybe it
27559  * should be moved into the arch dependent tree for those
27560  * ports that require it for backward compatibility? */
27561
27562 asmlinkage int sys_nice(int increment)
27563 {
27564     unsigned long newprio;
27565     int increase = 0;
27566
27567     /* Setpriority might change our priority at the same
27568      * moment. We don't have to worry. Conceptually one
27569      * call occurs first and we have a single winner. */
27570
27571     newprio = increment;
27572     if (increment < 0) {
27573         if (!capable(CAP_SYS_NICE))
27574             return -EPERM;
27575         newprio = -increment;

```

```

27576         increase = 1;
27577     }
27578
27579     if (newprio > 40)
27580         newprio = 40;
27581     /* do a "normalization" of the priority (traditionally
27582      * Unix nice values are -20 to 20; Linux doesn't really
27583      * use that kind of thing, but uses the length of the
27584      * timeslice instead (default 210 ms). The rounding is
27585      * why we want to avoid negative values. */
27586     newprio = (newprio * DEF_PRIORITY + 10) / 20;
27587     increment = newprio;
27588     if (increase)
27589         increment = -increment;
27590     /* Current->priority can change between this point and
27591      * the assignment. We are assigning not doing add/subs
27592      * so thats ok. Conceptually a process might just
27593      * instantaneously read the value we stomp over. I
27594      * don't think that is an issue unless posix makes it
27595      * one. If so we can loop on changes to
27596      * current->priority. */
27597     newprio = current->priority - increment;
27598     if ((signed) newprio < 1)
27599         newprio = 1;
27600     if (newprio > DEF_PRIORITY*2)
27601         newprio = DEF_PRIORITY*2;
27602     current->priority = newprio;
27603     return 0;
27604 }
27605
27606 #endif
27607
27608 static inline struct task_struct *
27609 find_process_by_pid(pid_t pid)
27610 {
27611     struct task_struct *tsk = current;
27612
27613     if (pid)
27614         tsk = find_task_by_pid(pid);
27615     return tsk;
27616 }
27617
27618 static int setscheduler(pid_t pid, int policy,
27619                        struct sched_param *param)
27620 {
27621     struct sched_param lp;
27622     struct task_struct *p;
27623     int retval;

```

→ p 525

```

27624
27625 retval = -EINVAL;
27626 if (!param || pid < 0)
27627     goto out_nounlock;
27628
27629 retval = -EFAULT;
27630 if (copy_from_user(&lp, param,
27631                 sizeof(struct sched_param)))
27632     goto out_nounlock;
27633
27634 /* We play safe to avoid deadlocks. */
27635 spin_lock(&scheduler_lock);
27636 spin_lock_irq(&runqueue_lock);
27637 read_lock(&tasklist_lock);
27638
27639 p = find_process_by_pid(pid);
27640
27641 retval = -ESRCH;
27642 if (!p)
27643     goto out_unlock;
27644
27645 if (policy < 0)
27646     policy = p->policy;
27647 else {
27648     retval = -EINVAL;
27649     if (policy != SCHED_FIFO && policy != SCHED_RR &&
27650         policy != SCHED_OTHER)
27651         goto out_unlock;
27652 }
27653
27654 /* Valid priorities for SCHED_FIFO and SCHED_RR are
27655  * 1..99, valid priority for SCHED_OTHER is 0. */
27656 retval = -EINVAL;
27657 if (lp.sched_priority < 0 || lp.sched_priority > 99)
27658     goto out_unlock;
27659 if((policy == SCHED_OTHER) != (lp.sched_priority == 0))
27660     goto out_unlock;
27661
27662 retval = -EPERM;
27663 if ((policy == SCHED_FIFO || policy == SCHED_RR) &&
27664     !capable(CAP_SYS_NICE))
27665     goto out_unlock;
27666 if ((current->euid != p->euid) &&
27667     (current->euid != p->uid) &&
27668     !capable(CAP_SYS_NICE))
27669     goto out_unlock;
27670
27671 retval = 0;

```

```

27672 p->policy = policy;
27673 p->rt_priority = lp.sched_priority;
27674 if (p->next_run)
27675     move_first_runqueue(p);
27676
27677 current->need_resched = 1;
27678
27679 out_unlock:
27680 read_unlock(&tasklist_lock);
27681 spin_unlock_irq(&runqueue_lock);
27682 spin_unlock(&scheduler_lock);
27683
27684 out_nounlock:
27685 return retval;
27686 }
27687
27688 asmlinkage int sys_sched_setscheduler(pid_t pid,
27689 int policy, struct sched_param *param)
27690 {
27691     return setscheduler(pid, policy, param);
27692 }
27693
27694 asmlinkage int sys_sched_setparam(pid_t pid,
27695 struct sched_param *param)
27696 {
27697     return setscheduler(pid, -1, param);
27698 }
27699
27700 asmlinkage int sys_sched_getscheduler(pid_t pid)
27701 {
27702     struct task_struct *p;
27703     int retval;
27704
27705     retval = -EINVAL;
27706     if (pid < 0)
27707         goto out_nounlock;
27708
27709     read_lock(&tasklist_lock);
27710
27711     retval = -ESRCH;
27712     p = find_process_by_pid(pid);
27713     if (!p)
27714         goto out_unlock;
27715
27716     retval = p->policy;
27717
27718 out_unlock:
27719     read_unlock(&tasklist_lock);

```



```

27720
27721 out_nounlock:
27722     return retval;
27723 }
27724
27725 asmlinkage int sys_sched_getparam(pid_t pid,
27726     struct sched_param *param)
27727 {
27728     struct task_struct *p;
27729     struct sched_param lp;
27730     int retval;
27731
27732     retval = -EINVAL;
27733     if (!param || pid < 0)
27734         goto out_nounlock;
27735
27736     read_lock(&tasklist_lock);
27737     p = find_process_by_pid(pid);
27738     retval = -ESRCH;
27739     if (!p)
27740         goto out_unlock;
27741     lp.sched_priority = p->rt_priority;
27742     read_unlock(&tasklist_lock);
27743
27744     /* This one might sleep, we cannot do it with a
27745      * spinlock held ... */
27746     retval = copy_to_user(param, &lp,
27747         sizeof(*param)) ? -EFAULT : 0;
27748
27749 out_nounlock:
27750     return retval;
27751
27752 out_unlock:
27753     read_unlock(&tasklist_lock);
27754     return retval;
27755 }
27756
27757 asmlinkage int sys_sched_yield(void)
27758 {
27759     spin_lock(&scheduler_lock);
27760     spin_lock_irq(&runqueue_lock);
27761     if (current->policy == SCHED_OTHER)
27762         current->policy |= SCHED_YIELD;
27763     current->need_resched = 1;
27764     move_last_runqueue(current);
27765     spin_unlock_irq(&runqueue_lock);
27766     spin_unlock(&scheduler_lock);
27767     return 0;

```

```

27768 }
27769
27770 asmlinkage int sys_sched_get_priority_max(int policy)
27771 {
27772     int ret = -EINVAL;
27773
27774     switch (policy) {
27775     case SCHED_FIFO:
27776     case SCHED_RR:
27777         ret = 99;
27778         break;
27779     case SCHED_OTHER:
27780         ret = 0;
27781         break;
27782     }
27783     return ret;
27784 }
27785
27786 asmlinkage int sys_sched_get_priority_min(int policy)
27787 {
27788     int ret = -EINVAL;
27789
27790     switch (policy) {
27791     case SCHED_FIFO:
27792     case SCHED_RR:
27793         ret = 1;
27794         break;
27795     case SCHED_OTHER:
27796         ret = 0;
27797     }
27798     return ret;
27799 }
27800
27801 asmlinkage int sys_sched_rr_get_interval(pid_t pid,
27802     struct timespec *interval)
27803 {
27804     struct timespec t;
27805
27806     t.tv_sec = 0;
27807     t.tv_nsec = 150000;
27808     if (copy_to_user(interval, &t,
27809         sizeof(struct timespec)))
27810         return -EFAULT;
27811     return 0;
27812 }
27813
27814 asmlinkage int sys_nanosleep(struct timespec *rqtp,
27815     struct timespec *rmtp)

```

```

27816 {
27817     struct timespec t;
27818     unsigned long expire;
27819
27820     if (copy_from_user(&t, rqtp, sizeof(struct timespec)))
27821         return -EFAULT;
27822
27823     if (t.tv_nsec >= 1000000000L || t.tv_nsec < 0 ||
27824         t.tv_sec < 0)
27825         return -EINVAL;
27826
27827
27828     if (t.tv_sec == 0 && t.tv_nsec <= 2000000L &&
27829         current->policy != SCHED_OTHER)
27830     {
27831         /* Short delay requests up to 2 ms will be handled
27832          * with high precision by a busy wait for all
27833          * real-time processes.
27834          *
27835          * It's important on SMP not to do this holding
27836          * locks. */
27837         udelay((t.tv_nsec + 999) / 1000);
27838         return 0;
27839     }
27840
27841     expire =
27842         timespec_to_jiffies(&t) + (t.tv_sec || t.tv_nsec);
27843
27844     current->state = TASK_INTERRUPTIBLE;
27845     expire = schedule_timeout(expire);
27846
27847     if (expire) {
27848         if (rmtp) {
27849             jiffies_to_timespec(expire, &t);
27850             if (copy_to_user(rmtp, &t, sizeof(struct timespec)))
27851                 return -EFAULT;
27852         }
27853         return -EINTR;
27854     }
27855     return 0;
27856 }
27857
27858 static void show_task(int nr, struct task_struct * p)
27859 {
27860     unsigned long free = 0;
27861     int state;
27862     static const char * stat_nam[] =
27863         { "R", "S", "D", "Z", "T", "W" };

```

```

27864
27865     printk("%-8s %3d ",
27866         p->comm, (p == current) ? -nr : nr);
27867     state = p->state ? ffz(~p->state) + 1 : 0;
27868     if (((unsigned) state) <
27869         sizeof(stat_nam)/sizeof(char *))
27870         printk(stat_nam[state]);
27871     else
27872         printk(" ");
27873     #if (BITS_PER_LONG == 32)
27874     if (p == current)
27875         printk(" current ");
27876     else
27877         printk(" %08lx ", thread_saved_pc(&p->tss));
27878     #else
27879     if (p == current)
27880         printk(" current task ");
27881     else
27882         printk(" %016lx ", thread_saved_pc(&p->tss));
27883     #endif
27884     {
27885         unsigned long * n = (unsigned long *) (p+1);
27886         while (!*n)
27887             n++;
27888         free = (unsigned long) n - (unsigned long)(p+1);
27889     }
27890     printk("%5lu %5d %6d ", free, p->pid, p->p_pptr->pid);
27891     if (p->p_cptr)
27892         printk("%5d ", p->p_cptr->pid);
27893     else
27894         printk(" ");
27895     if (p->p_ysptr)
27896         printk("%7d", p->p_ysptr->pid);
27897     else
27898         printk(" ");
27899     if (p->p_osptr)
27900         printk(" %5d\n", p->p_osptr->pid);
27901     else
27902         printk("\n");
27903
27904     {
27905         struct signal_queue *q;
27906         char s[sizeof(sigset_t)*2+1], b[sizeof(sigset_t)*2+1];
27907
27908         render_sigset_t(&p->signal, s);
27909         render_sigset_t(&p->blocked, b);
27910         printk(" sig: %d %s %s :",
27911             signal_pending(p), s, b);

```

```

27912     for (q = p->sigqueue; q ; q = q->next)
27913         printk(" %d", q->info.si_signo);
27914     printk(" X\n");
27915 }
27916 }
27917
27918 char * render_sigset_t(sigset_t *set, char *buffer)
27919 {
27920     int i = _NSIG, x;
27921     do {
27922         i -= 4, x = 0;
27923         if (sigismember(set, i+1)) x |= 1;
27924         if (sigismember(set, i+2)) x |= 2;
27925         if (sigismember(set, i+3)) x |= 4;
27926         if (sigismember(set, i+4)) x |= 8;
27927         *buffer++ = (x < 10 ? '0' : 'a' - 10) + x;
27928     } while (i >= 4);
27929     *buffer = 0;
27930     return buffer;
27931 }
27932
27933 void show_state(void)
27934 {
27935     struct task_struct *p;
27936
27937     #if (BITS_PER_LONG == 32)
27938     printk("\n"
27939          "          sibling\n")
27940          "          free          "
27941     printk(" task          PC          stack          pid father "
27942          "child younger older\n");
27943     #else
27944     printk("\n"
27945          "          sibling\n")
27946          "          free          "
27947     printk(" task          PC          stack          pid "
27948          "father child younger older\n");
27949     #endif
27950     read_lock(&tasklist_lock);
27951     for_each_task(p)
27952         show_task((p->tarray_ptr - &task[0]),p);
27953     read_unlock(&tasklist_lock);
27954 }
27955
27956 void __init sched_init(void)
27957 {
27958     /* We have to do a little magic to get the first
27959      * process right in SMP mode. */

```

```

27960     int cpu=hard_smp_processor_id();
27961     int nr = NR_TASKS;
27962
27963     init_task.processor=cpu;
27964
27965     /* Init task array free list and pidhash table. */
27966     while(--nr > 0)
27967         add_free_taskslot(&task[nr]);
27968
27969     for(nr = 0; nr < PIDHASH_SZ; nr++)
27970         pidhash[nr] = NULL;
27971
27972     init_bh(TIMER_BH, timer_bh);
27973     init_bh(TQUEUE_BH, tqueue_bh);
27974     init_bh(IMMEDIATE_BH, immediate_bh);
27975 }

```

kernel/signal.c

```

27976 /*
27977  * linux/kernel/signal.c
27978  *
27979  * Copyright (C) 1991, 1992 Linus Torvalds
27980  *
27981  * 1997-11-02 Modified for POSIX.1b signals by Richard
27982  * Henderson
27983  */
27984
27985 #include <linux/slab.h>
27986 #include <linux/module.h>
27987 #include <linux/unistd.h>
27988 #include <linux/smp_lock.h>
27989 #include <linux/init.h>
27990
27991 #include <asm/uaccess.h>
27992
27993 /* SLAB caches for signal bits. */
27994
27995 #define DEBUG_SIG 0
27996
27997 #if DEBUG_SIG
27998 #define SIG_SLAB_DEBUG (SLAB_DEBUG_FREE | SLAB_RED_ZONE)
27999 /* | SLAB_POISON */
28000 #else
28001 #define SIG_SLAB_DEBUG 0
28002 #endif
28003
28004 static kmem_cache_t *signal_queue_cachep;

```

```

28005
28006 int nr_queued_signals;
28007 int max_queued_signals = 1024;
28008
28009 void __init signals_init(void)
28010 {
28011     signal_queue_cachep =
28012         kmem_cache_create("signal_queue",
28013             sizeof(struct signal_queue),
28014             __alignof__(struct signal_queue),
28015             SIG_SLAB_DEBUG, NULL, NULL);
28016 }
28017
28018
28019 /* Flush all pending signals for a task. */
28020 void
28021 flush_signals(struct task_struct *t)
28022 {
28023     struct signal_queue *q, *n;
28024
28025     t->sigpending = 0;
28026     sigemptyset(&t->signal);
28027     q = t->sigqueue;
28028     t->sigqueue = NULL;
28029     t->sigqueue_tail = &t->sigqueue;
28030
28031     while (q) {
28032         n = q->next;
28033         kmem_cache_free(signal_queue_cachep, q);
28034         nr_queued_signals--;
28035         q = n;
28036     }
28037 }
28038
28039 /* Flush all handlers for a task. */
28040 void
28041 flush_signal_handlers(struct task_struct *t)
28042 {
28043     int i;
28044     struct k_sigaction *ka = &t->sig->action[0];
28045     for (i = _NSIG; i != 0; i--) {
28046         if (ka->sa.sa_handler != SIG_IGN)
28047             ka->sa.sa_handler = SIG_DFL;
28048         ka->sa.sa_flags = 0;
28049         sigemptyset(&ka->sa.sa_mask);
28050         ka++;
28051     }
28052 }

```

```

28053
28054 /* Dequeue a signal and return the element to the caller,
28055  * which is expected to free it.
28056  *
28057  * All callers of must be holding current->sigmask_lock.
28058  */
28059 int
28060 dequeue_signal(sigset_t *mask, siginfo_t *info)
28061 {
28062     unsigned long i, *s, *m, x;
28063     int sig = 0;
28064
28065     #if DEBUG_SIG
28066     printk("SIG dequeue (%s:%d): %d ", current->comm,
28067         current->pid, signal_pending(current));
28068     #endif
28069
28070     /* Find the first desired signal that is pending. */
28071     s = current->signal.sig;
28072     m = mask->sig;
28073     switch (_NSIG_WORDS) {
28074     default:
28075         for (i = 0; i < _NSIG_WORDS; ++i, ++s, ++m)
28076             if ((x = *s &~ *m) != 0) {
28077                 sig = ffz(~x) + i*_NSIG_BPW + 1;
28078                 break;
28079             }
28080         break;
28081
28082     case 2: if ((x = s[0] &~ m[0]) != 0)
28083             sig = 1;
28084             else if ((x = s[1] &~ m[1]) != 0)
28085                 sig = _NSIG_BPW + 1;
28086             else
28087                 break;
28088             sig += ffz(~x);
28089             break;
28090
28091     case 1: if ((x = *s &~ *m) != 0)
28092             sig = ffz(~x) + 1;
28093             break;
28094     }
28095
28096     if (sig) {
28097         int reset = 1;
28098
28099         /* Collect the siginfo appropriate to this signal. */
28100         if (sig < SIGRTMIN) {

```

```

28101      /* XXX: As an extension, support queueing exactly
28102         one non-rt signal if SA_SIGINFO is set, so that
28103         we can get more detailed information about the
28104         cause of the signal. */
28105      /* Deciding not to init these couple of fields is
28106         more expensive than just initializing them. */
28107      info->si_signo = sig;
28108      info->si_errno = 0;
28109      info->si_code = 0;
28110      info->si_pid = 0;
28111      info->si_uid = 0;
28112  } else {
28113      struct signal_queue *q, **pp;
28114      pp = &current->sigqueue;
28115      q = current->sigqueue;
28116
28117      /* Find the one we're interested in ... */
28118      for ( ; q ; pp = &q->next, q = q->next)
28119          if (q->info.si_signo == sig)
28120              break;
28121      if (q) {
28122          if ((*pp = q->next) == NULL)
28123              current->sigqueue_tail = pp;
28124          *info = q->info;
28125          kmem_cache_free(signal_queue_cache, q);
28126          nr_queued_signals--;
28127
28128          /* then see if this signal is still pending. */
28129          q = *pp;
28130          while (q) {
28131              if (q->info.si_signo == sig) {
28132                  reset = 0;
28133                  break;
28134              }
28135              q = q->next;
28136          }
28137      } else {
28138          /* Ok, it wasn't in the queue. It must have
28139             been sent either by a non-rt mechanism and
28140             we ran out of queue space. So zero out the
28141             info. */
28142          info->si_signo = sig;
28143          info->si_errno = 0;
28144          info->si_code = 0;
28145          info->si_pid = 0;
28146          info->si_uid = 0;
28147      }
28148  }

```

```

28149
28150      if (reset)
28151          sigdelset(&current->signal, sig);
28152      recalc_sigpending(current);
28153
28154      /* XXX: Once POSIX.1b timers are in, if si_code ==
28155         SI_TIMER, we need to xchg out the timer overrun
28156         values. */
28157  } else {
28158      /* XXX: Once CLONE_PID is in to join those "threads"
28159         that are part of the same "process", look for
28160         signals sent to the "process" as well. */
28161
28162      /* Sanity check... */
28163      if (mask == &current->blocked &&
28164          signal_pending(current)) {
28165          printk(KERN_CRIT "SIG: sigpending lied\n");
28166          current->sigpending = 0;
28167      }
28168  }
28169
28170 #if DEBUG_SIG
28171 printk(" %d -> %d\n", signal_pending(current), sig);
28172 #endif
28173
28174      return sig;
28175  }
28176
28177 /* Determine whether a signal should be posted or not.
28178 *
28179 * Signals with SIG_IGN can be ignored, except for the
28180 * special case of a SIGCHLD.
28181 *
28182 * Some signals with SIG_DFL default to a non-action. */
28183 static int ignored_signal(int sig, struct task_struct *t)
28184 {
28185     struct signal_struct *signals;
28186     struct k_sigaction *ka;
28187
28188     /* Don't ignore traced or blocked signals */
28189     if ((t->flags & PF_PTRACED) ||
28190         sigismember(&t->blocked, sig))
28191         return 0;
28192
28193     signals = t->sig;
28194     if (!signals)
28195         return 1;
28196

```

```

28197 ka = &signals->action[sig-1];
28198 switch ((unsigned long) ka->sa_handler) {
28199     case (unsigned long) SIG_DFL:
28200         if (sig == SIGCONT ||
28201             sig == SIGWINCH ||
28202             sig == SIGCHLD ||
28203             sig == SIGURG)
28204             break;
28205         return 0;
28206
28207     case (unsigned long) SIG_IGN:
28208         if (sig != SIGCHLD)
28209             break;
28210         /* fallthrough */
28211     default:
28212         return 0;
28213 }
28214 return 1;
28215 }
28216
28217 int
28218 send_sig_info(int sig, struct siginfo *info,
28219              struct task_struct *t)
28220 {
28221     unsigned long flags;
28222     int ret;
28223
28224     #if DEBUG_SIG
28225     printk("SIG queue (%s:%d): %d ", t->comm, t->pid, sig);
28226     #endif
28227
28228     ret = EINVAL;
28229     if (sig < 0 || sig > _NSIG)
28230         goto out_nolock;
28231     /* The somewhat baroque permissions check... */
28232     ret = EPERM;
28233     if ((!info || ((unsigned long)info != 1 &&
28234                  SI_FROMUSER(info)))
28235         && ((sig != SIGCONT) ||
28236            (current->session != t->session))
28237         && (current->euid ^ t->suid)
28238         && (current->euid ^ t->uid)
28239         && (current->uid ^ t->suid)
28240         && (current->uid ^ t->uid)
28241         && !capable(CAP_SYS_ADMIN))
28242         goto out_nolock;
28243
28244     /* The null signal is a permissions and process

```

```

28245     * existence probe. No signal is actually delivered.
28246     * Same goes for zombies. */
28247     ret = 0;
28248     if (!sig || !t->sig)
28249         goto out_nolock;
28250
28251     spin_lock_irqsave(&t->sigmask lock, flags);
28252     switch (sig) {
28253     case SIGKILL: case SIGCONT:
28254         /* Wake up the process if stopped. */
28255         if (t->state == TASK_STOPPED)
28256             wake_up_process(t);
28257         t->exit_code = 0;
28258         sigdelsetmask(&t->signal,
28259                      (sigmask(SIGSTOP)|sigmask(SIGTSTP) |
28260                       sigmask(SIGTTOU)|sigmask(SIGTTIN)));
28261         /* Inflict this corner case with recalculations, not
28262          * mainline */
28263         recalc_sigpending(t);
28264         break;
28265
28266     case SIGSTOP: case SIGTSTP:
28267     case SIGTTIN: case SIGTTOU:
28268         /* If we're stopping again, cancel SIGCONT */
28269         sigdelset(&t->signal, SIGCONT);
28270         /* Inflict this corner case with recalculations, not
28271          * mainline */
28272         recalc_sigpending(t);
28273         break;
28274     }
28275
28276     /* Optimize away the signal, if it's a signal that can
28277     * be handled immediately (ie non-blocked and untraced)
28278     * and that is ignored (either explicitly or by
28279     * default). */
28280
28281     if (ignored_signal(sig, t))
28282         goto out;
28283
28284     if (sig < SIGRTMIN) {
28285         /* Non-real-time signals are not queued. */
28286         /* XXX: As an extension, support queueing exactly one
28287          * non-rt signal if SA_SIGINFO is set, so that we can
28288          * get more detailed information about the cause of
28289          * the signal. */
28290         if (sigismember(&t->signal, sig))
28291             goto out;
28292     } else {

```

```

28293 /* Real-time signals must be queued if sent by
28294 * sigqueue, or some other real-time mechanism. It
28295 * is implementation defined whether kill() does so.
28296 * We attempt to do so, on the principle of least
28297 * surprise, but since kill is not allowed to fail
28298 * with EAGAIN when low on memory we just make sure
28299 * at least one signal gets delivered and don't pass
28300 * on the info struct. */
28301
28302 struct signal_queue *q = 0;
28303
28304 if (nr_queued_signals < max_queued_signals) {
28305     q = (struct signal_queue *)
28306         kmem_cache_alloc(signal_queue_cachep,
28307                         GFP_KERNEL);
28308 }
28309
28310 if (q) {
28311     nr_queued_signals++;
28312     q->next = NULL;
28313     *t->sigqueue_tail = q;
28314     t->sigqueue_tail = &q->next;
28315     switch ((unsigned long) info) {
28316     case 0:
28317         q->info.si_signo = sig;
28318         q->info.si_errno = 0;
28319         q->info.si_code = SI_USER;
28320         q->info.si_pid = current->pid;
28321         q->info.si_uid = current->uid;
28322         break;
28323     case 1:
28324         q->info.si_signo = sig;
28325         q->info.si_errno = 0;
28326         q->info.si_code = SI_KERNEL;
28327         q->info.si_pid = 0;
28328         q->info.si_uid = 0;
28329         break;
28330     default:
28331         q->info = *info;
28332         break;
28333     }
28334 } else {
28335     /* If this was sent by a rt mechanism, try again.*/
28336     if (info->si_code < 0) {
28337         ret = -EAGAIN;
28338         goto out;
28339     }
28340     /* Otherwise, mention that the signal is pending,

```

```

28341         but don't queue the info. */
28342     }
28343 }
28344
28345 sigaddset(&t->signal, sig);
28346 if (!sigismember(&t->blocked, sig)) {
28347     t->sigpending = 1;
28348 #ifdef __SMP__
28349     /* If the task is running on a different CPU force a
28350     * reschedule on the other CPU - note that the code
28351     * below is a tad loose and might occasionally kick
28352     * the wrong CPU if we catch the process in the
28353     * process of changing - but no harm is done by that
28354     * other than doing an extra (lightweight) IPI
28355     * interrupt.
28356     *
28357     * note that we rely on the previous spin_lock to
28358     * lock interrupts for us! No need to set
28359     * need_resched since signal event passing goes
28360     * through ->blocked. */
28361     spin_lock(&runqueue_lock);
28362     if (t->has_cpu && t->processor != smp_processor_id())
28363         smp_send_reschedule(t->processor);
28364     spin_unlock(&runqueue_lock);
28365 #endif /* __SMP__ */
28366 }
28367
28368 out:
28369     spin_unlock_irqrestore(&t->sigmask_lock, flags);
28370     if(t->state == TASK_INTERRUPTIBLE && signal_pending(t))
28371         wake_up_process(t);
28372
28373 out_nolock:
28374 #if DEBUG_SIG
28375     printk(" %d -> %d\n", signal_pending(t), ret);
28376 #endif
28377
28378     return ret;
28379 }
28380
28381 /* Force a signal that the process can't ignore: if
28382 * necessary we unblock the signal and change any SIG_IGN
28383 * to SIG_DFL. */
28384
28385 int
28386 force_sig_info(int sig, struct siginfo *info,
28387               struct task_struct *)
28388 {

```

```

28389 unsigned long int flags;
28390
28391 spin_lock_irqsave(&t->sigmask_lock, flags);
28392 if (t->sig == NULL) {
28393     spin_unlock_irqrestore(&t->sigmask_lock, flags);
28394     return -ESRCH;
28395 }
28396
28397 if (t->sig->action[sig-1].sa.sa_handler == SIG_IGN)
28398     t->sig->action[sig-1].sa.sa_handler = SIG_DFL;
28399 sigdelset(&t->blocked, sig);
28400 spin_unlock_irqrestore(&t->sigmask_lock, flags);
28401
28402 return send_sig_info(sig, info, t);
28403 }
28404
28405 /* kill_pg() sends a signal to a process group: this is
28406 * what the tty control characters do (^C, ^Z etc) */
28407 int
28408 kill_pg_info(int sig, struct siginfo *info, pid_t pgrp)
28409 {
28410     int retval = -EINVAL;
28411     if (pgrp > 0) {
28412         struct task_struct *p;
28413         int found = 0;
28414
28415         retval = -ESRCH;
28416         read_lock(&tasklist_lock);
28417         for_each_task(p) {
28418             if (p->pgrp == pgrp) {
28419                 int err = send_sig_info(sig, info, p);
28420                 if (err != 0)
28421                     retval = err;
28422             } else
28423                 found++;
28424         }
28425     }
28426     read_unlock(&tasklist_lock);
28427     if (found)
28428         retval = 0;
28429 }
28430 return retval;
28431 }
28432
28433 /* kill_sl() sends a signal to the session leader: this
28434 * is used to send SIGHUP to the controlling process of a
28435 * terminal when the connection is lost. */
28436 int

```

p 481

```

28437 kill_sl_info(int sig, struct siginfo *info, pid_t sess)
28438 {
28439     int retval = -EINVAL;
28440     if (sess > 0) {
28441         struct task_struct *p;
28442         int found = 0;
28443
28444         retval = -ESRCH;
28445         read_lock(&tasklist_lock);
28446         for_each_task(p) {
28447             if (p->leader && p->session == sess) {
28448                 int err = send_sig_info(sig, info, p);
28449                 if (err)
28450                     retval = err;
28451             } else
28452                 found++;
28453         }
28454     }
28455     read_unlock(&tasklist_lock);
28456     if (found)
28457         retval = 0;
28458 }
28459 return retval;
28460 }
28461
28462 inline int
28463 kill_proc_info(int sig, struct siginfo *info, pid_t pid)
28464 {
28465     int error;
28466     struct task_struct *p;
28467
28468     read_lock(&tasklist_lock);
28469     p = find_task_by_pid(pid);
28470     error = -ESRCH;
28471     if (p)
28472         error = send_sig_info(sig, info, p);
28473     read_unlock(&tasklist_lock);
28474     return error;
28475 }
28476
28477 /* kill_something() interprets pid in interesting ways
28478 * just like kill(2).
28479 *
28480 * POSIX specifies that kill(-1, sig) is unspecified, but
28481 * what we have is probably wrong. Should make it like
28482 * BSD or SYSV. */
28483 int
28484 kill_something_info(int sig, struct siginfo *info,

```

p 482

p 483


```

28485             int pid)
28486 {
28487     if (!pid) {
28488         return kill_pg_info(sig, info, current->pggrp);
28489     } else if (pid == -1) {
28490         int retval = 0, count = 0;
28491         struct task_struct * p;
28492
28493         read_lock(&tasklist_lock);
28494         for_each_task(p) {
28495             if (p->pid > 1 && p != current) {
28496                 int err = send_sig_info(sig, info, p);
28497                 ++count;
28498                 if (err != -EPERM)
28499                     retval = err;
28500             }
28501         }
28502         read_unlock(&tasklist_lock);
28503         return count ? retval : -ESRCH;
28504     } else if (pid < 0) {
28505         return kill_pg_info(sig, info, -pid);
28506     } else {
28507         return kill_proc_info(sig, info, pid);
28508     }
28509 }
28510
28511 /* These are for backward compatibility with the rest of
28512  * the kernel source. */
28513 int
28514 send_sig(int sig, struct task_struct *p, int priv)
28515 {
28516     return send_sig_info(sig, (void*)(long)(priv != 0), p);
28517 }
28518
28519 void
28520 force_sig(int sig, struct task_struct *p)
28521 {
28522     force_sig_info(sig, (void*)1L, p);
28523 }
28524
28525 int
28526 kill_pg(pid_t pgrp, int sig, int priv)
28527 {
28528     return kill_pg_info(sig,
28529                         (void*)(long)(priv != 0), pgrp);
28530 }
28531
28532 int

```

```

28533 kill_sl(pid_t sess, int sig, int priv)
28534 {
28535     return kill_sl_info(sig,
28536                         (void*)(long)(priv != 0), sess);
28537 }
28538
28539 int
28540 kill_proc(pid_t pid, int sig, int priv)
28541 {
28542     return kill_proc_info(sig,
28543                           (void*)(long)(priv != 0), pid);
28544 }
28545
28546 /* Let a parent know about a status change of a child. */
28547 void
28548 notify_parent(struct task_struct *tsk, int sig)
28549 {
28550     struct siginfo info;
28551     int why;
28552
28553     info.si_signo = sig;
28554     info.si_errno = 0;
28555     info.si_pid = tsk->pid;
28556
28557     /* FIXME: find out whether or not this is supposed to
28558      * be c*time. */
28559     info.si_utime = tsk->times.tms_utime;
28560     info.si_stime = tsk->times.tms_stime;
28561
28562     why = SI_KERNEL;          /* shouldn't happen */
28563     switch (tsk->state) {
28564     case TASK_ZOMBIE:
28565         if (tsk->exit_code & 0x80)
28566             why = CLD_DUMPED;
28567         else if (tsk->exit_code & 0x7f)
28568             why = CLD_KILLED;
28569         else
28570             why = CLD_EXITED;
28571         break;
28572     case TASK_STOPPED:
28573         /* FIXME -- can we deduce CLD_TRAPPED or
28574          * CLD_CONTINUED? */
28575         why = CLD_STOPPED;
28576         break;
28577     default:
28578         printk(KERN_DEBUG
28579                "eh? notify parent with state %ld?\n",

```

→ p 488

```

28581         tsk->state);
28582     break;
28583 }
28584 info.si_code = why;
28585
28586 send_sig_info(sig, &info, tsk->p_pptr);
28587 wake_up_interruptible(&tsk->p_pptr->wait_chldexit);
28588 }
28589
28590 EXPORT_SYMBOL(dequeue_signal);
28591 EXPORT_SYMBOL(flush_signals);
28592 EXPORT_SYMBOL(force_sig);
28593 EXPORT_SYMBOL(force_sig_info);
28594 EXPORT_SYMBOL(kill_pg);
28595 EXPORT_SYMBOL(kill_pg_info);
28596 EXPORT_SYMBOL(kill_proc);
28597 EXPORT_SYMBOL(kill_proc_info);
28598 EXPORT_SYMBOL(kill_sl);
28599 EXPORT_SYMBOL(kill_sl_info);
28600 EXPORT_SYMBOL(notify_parent);
28601 EXPORT_SYMBOL(recalc_sigpending);
28602 EXPORT_SYMBOL(send_sig);
28603 EXPORT_SYMBOL(send_sig_info);
28604
28605
28606 /* System call entry points. */
28607
28608 /* We don't need to get the kernel lock - this is all
28609  * local to this particular thread.. (and that's good,
28610  * because this is _heavily_ used by various programs) */
28611 asmlinkage int
28612 sys_rt_sigprocmask(int how, sigset_t *set,
28613                   sigset_t *oset, size_t sigsetsize)
28614 {
28615     int error = -EINVAL;
28616     sigset_t old_set, new_set;
28617
28618     /* XXX: Don't preclude handling different sized
28619      * sigset_t's. */
28620     if (sigsetsize != sizeof(sigset_t))
28621         goto out;
28622
28623     if (set) {
28624         error = -EFAULT;
28625         if (copy_from_user(&new_set, set, sizeof(*set)))
28626             goto out;
28627         sigdelsetmask(&new_set,
28628                     sigmask(SIGKILL)|sigmask(SIGSTOP));
28629
28630         spin_lock_irq(&current->sigmask_lock);
28631         old_set = current->blocked;
28632
28633         error = 0;
28634         switch (how) {
28635             default:
28636                 error = -EINVAL;
28637                 break;
28638             case SIG_BLOCK:
28639                 sigorsets(&new_set, &old_set, &new_set);
28640                 break;
28641             case SIG_UNBLOCK:
28642                 signandsets(&new_set, &old_set, &new_set);
28643                 break;
28644             case SIG_SETMASK:
28645                 break;
28646         }
28647
28648         current->blocked = new_set;
28649         recalc_sigpending(current);
28650         spin_unlock_irq(&current->sigmask_lock);
28651         if (error)
28652             goto out;
28653         if (oset)
28654             goto set_old;
28655     } else if (oset) {
28656         spin_lock_irq(&current->sigmask_lock);
28657         old_set = current->blocked;
28658         spin_unlock_irq(&current->sigmask_lock);
28659
28660         set_old:
28661         error = -EFAULT;
28662         if (copy_to_user(oset, &old_set, sizeof(*oset)))
28663             goto out;
28664     }
28665     error = 0;
28666 out:
28667     return error;
28668 }
28669
28670 asmlinkage int
28671 sys_rt_sigpending(sigset_t *set, size_t sigsetsize)
28672 {
28673     int error = -EINVAL;
28674     sigset_t pending;
28675
28676     /* XXX: Don't preclude handling different sized

```

```

28677  * sigset_t's. */
28678  if (sigsetsize != sizeof(sigset_t))
28679      goto out;
28680
28681  spin_lock_irq(&current->sigmask_lock);
28682  sigandsets(&pending,
28683             &current->blocked, &current->signal);
28684  spin_unlock_irq(&current->sigmask_lock);
28685
28686  error = -EFAULT;
28687  if (!copy_to_user(set, &pending, sizeof(*set)))
28688      error = 0;
28689  out:
28690  return error;
28691  }
28692
28693  asmlinkage int
28694  sys_rt_sigtimedwait(const sigset_t *uthese,
28695  siginfo_t *uinfo, const struct timespec *uts,
28696  size_t sigsetsize)
28697  {
28698  int ret, sig;
28699  sigset_t these;
28700  struct timespec ts;
28701  siginfo_t info;
28702  long timeout = 0;
28703
28704  /* XXX: Don't preclude handling different sized
28705   * sigset_t's. */
28706  if (sigsetsize != sizeof(sigset_t))
28707      return -EINVAL;
28708
28709  if (copy_from_user(&these, uthese, sizeof(these)))
28710      return -EFAULT;
28711  else {
28712      /* Invert the set of allowed signals to get those we
28713       * want to block. */
28714      signotset(&these);
28715  }
28716
28717  if (uts) {
28718      if (copy_from_user(&ts, uts, sizeof(ts)))
28719          return -EFAULT;
28720      if (ts.tv_nsec >= 1000000000L || ts.tv_nsec < 0
28721          || ts.tv_sec < 0)
28722          return -EINVAL;
28723  }
28724

```

```

28725  spin_lock_irq(&current->sigmask_lock);
28726  sig = dequeue_signal(&these, &info);
28727  if (!sig) {
28728      /* None ready -- temporarily unblock those we're
28729       * interested in so that we'll be awakened when they
28730       * arrive. */
28731      sigset_t oldblocked = current->blocked;
28732      sigandsets(&current->blocked, &current->blocked,
28733                &these);
28734      recalc_sigpending(current);
28735      spin_unlock_irq(&current->sigmask_lock);
28736
28737      timeout = MAX_SCHEDULE_TIMEOUT;
28738      if (uts)
28739          timeout = (timespec_to_jiffies(&ts)
28740                   + (ts.tv_sec || ts.tv_nsec));
28741
28742      current->state = TASK_INTERRUPTIBLE;
28743      timeout = schedule_timeout(timeout);
28744
28745      spin_lock_irq(&current->sigmask_lock);
28746      sig = dequeue_signal(&these, &info);
28747      current->blocked = oldblocked;
28748      recalc_sigpending(current);
28749  }
28750  spin_unlock_irq(&current->sigmask_lock);
28751
28752  if (sig) {
28753      ret = sig;
28754      if (uinfo) {
28755          if (copy_to_user(uinfo, &info, sizeof(siginfo_t)))
28756              ret = -EFAULT;
28757      }
28758  } else {
28759      ret = -EAGAIN;
28760      if (timeout)
28761          ret = -EINTR;
28762  }
28763
28764  return ret;
28765  }
28766
28767  asmlinkage int
28768  sys_kill(int pid, int sig)
28769  {
28770  struct siginfo info;
28771
28772  info.si_signo = sig;

```

```

28773 info.si_errno = 0;
28774 info.si_code = SI_USER;
28775 info.si_pid = current->pid;
28776 info.si_uid = current->uid;
28777
28778 return kill_something_info(sig, &info, pid);
28779 }
28780
28781 asmlinkage int
28782 sys_rt_sigqueueinfo(int pid, int sig, siginfo_t *uinfo)
28783 {
28784     siginfo_t info;
28785
28786     if (copy_from_user(&info, uinfo, sizeof(siginfo_t)))
28787         return -EFAULT;
28788
28789     /* Not even root can pretend to send signals from the
28790      * kernel. Nor can they impersonate a kill(), which
28791      * adds source info. */
28792     if (info.si_code >= 0)
28793         return -EPERM;
28794     info.si_signo = sig;
28795
28796     /* POSIX.1b doesn't mention process groups. */
28797     return kill_proc_info(sig, &info, pid);
28798 }
28799
28800 int
28801 do_sigaction(int sig, const struct k_sigaction *act,
28802             struct k_sigaction *oact)
28803 {
28804     struct k_sigaction *k;
28805
28806     if (sig < 1 || sig > _NSIG ||
28807         (act && (sig == SIGKILL || sig == SIGSTOP)))
28808         return -EINVAL;
28809
28810     spin_lock_irq(&current->sigmask_lock);
28811     k = &current->sig->action[sig - 1];
28812
28813     if (oact) *oact = *k;
28814
28815     if (act) {
28816         *k = *act;
28817         sigdelsetmask(&k->sa.sa_mask,
28818                     sigmask(SIGKILL) | sigmask(SIGSTOP));
28819
28820         /* POSIX 3.3.1.3: "Setting a signal action to SIG_IGN

```

```

28821      * for a signal that is pending shall cause the
28822      * pending signal to be discarded, whether or not it
28823      * is blocked."
28824      *
28825      * "Setting a signal action to SIG_DFL for a signal
28826      * that is pending and whose default action is to
28827      * ignore the signal (for example, SIGCHLD), shall
28828      * cause the pending signal to be discarded, whether
28829      * or not it is blocked"
28830      *
28831      * Note the silly behaviour of SIGCHLD: SIG_IGN means
28832      * that the signal isn't actually ignored, but does
28833      * automatic child reaping, while SIG_DFL is
28834      * explicitly said by POSIX to force the signal to be
28835      * ignored. */
28836     if (k->sa.sa_handler == SIG_IGN
28837         || (k->sa.sa_handler == SIG_DFL
28838             && (sig == SIGCONT ||
28839                sig == SIGCHLD ||
28840                sig == SIGWINCH))) {
28841         /* So dequeue any that might be pending. XXX:
28842          * process-wide signals? */
28843         if (sig >= SIGRTMIN &&
28844             sigismember(&current->signal, sig)) {
28845             struct signal_queue *q, **pp;
28846             pp = &current->sigqueue;
28847             q = current->sigqueue;
28848             while (q) {
28849                 if (q->info.si_signo != sig)
28850                     pp = &q->next;
28851                 else {
28852                     *pp = q->next;
28853                     kmem_cache_free(signal_queue_cache, q);
28854                     nr_queued_signals--;
28855                 }
28856                 q = *pp;
28857             }
28858         }
28859     }
28860     sigdelset(&current->signal, sig);
28861     recalc_sigpending(current);
28862 }
28863 }
28864
28865 spin_unlock_irq(&current->sigmask_lock);
28866
28867 return 0;
28868 }

```

```

28869
28870 int
28871 do_sigaltstack(const stack_t *uss, stack_t *uoss,
28872                unsigned long sp)
28873 {
28874     stack_t oss;
28875     int error;
28876
28877     if (uoss) {
28878         oss.ss_sp = (void *) current->sas_ss_sp;
28879         oss.ss_size = current->sas_ss_size;
28880         oss.ss_flags = sas_ss_flags(sp);
28881     }
28882
28883     if (uss) {
28884         void *ss_sp;
28885         size_t ss_size;
28886         int ss_flags;
28887
28888         error = -EFAULT;
28889         if (verify_area(VERIFY_READ, uss, sizeof(*uss))
28890             || __get_user(ss_sp, &uss->ss_sp)
28891             || __get_user(ss_flags, &uss->ss_flags)
28892             || __get_user(ss_size, &uss->ss_size))
28893             goto out;
28894
28895         error = -EPERM;
28896         if (on_sig_stack (sp))
28897             goto out;
28898
28899         error = -EINVAL;
28900         if (ss_flags & ~SS_DISABLE)
28901             goto out;
28902
28903         if (ss_flags & SS_DISABLE) {
28904             ss_size = 0;
28905             ss_sp = NULL;
28906         } else {
28907             error = -ENOMEM;
28908             if (ss_size < MINSIGSTKSZ)
28909                 goto out;
28910         }
28911
28912         current->sas_ss_sp = (unsigned long) ss_sp;
28913         current->sas_ss_size = ss_size;
28914     }
28915
28916     if (uoss) {

```

```

28917         error = -EFAULT;
28918         if (copy_to_user(uoss, &oss, sizeof(oss)))
28919             goto out;
28920     }
28921
28922     error = 0;
28923 out:
28924     return error;
28925 }
28926
28927 #if !defined(__alpha__)
28928 /* Alpha has its own versions with special arguments. */
28929
28930 asmlinkage int
28931 sys_sigprocmask(int how, old_sigset_t *set,
28932                old_sigset_t *oset)
28933 {
28934     int error;
28935     old_sigset_t old_set, new_set;
28936
28937     if (set) {
28938         error = -EFAULT;
28939         if (copy_from_user(&new_set, set, sizeof(*set)))
28940             goto out;
28941         new_set &= ~(sigmask(SIGKILL)|sigmask(SIGSTOP));
28942
28943         spin_lock_irq(&current->sigmask_lock);
28944         old_set = current->blocked.sig[0];
28945
28946         error = 0;
28947         switch (how) {
28948         default:
28949             error = -EINVAL;
28950             break;
28951         case SIG_BLOCK:
28952             sigaddsetmask(&current->blocked, new_set);
28953             break;
28954         case SIG_UNBLOCK:
28955             sigdelsetmask(&current->blocked, new_set);
28956             break;
28957         case SIG_SETMASK:
28958             current->blocked.sig[0] = new_set;
28959             break;
28960         }
28961
28962         recalc_sigpending(current);
28963         spin_unlock_irq(&current->sigmask_lock);
28964         if (error)

```

▶ p 49

```

28965     goto out;
28966     if (oset)
28967         goto set_old;
28968 } else if (oset) {
28969     old_set = current->blocked.sig[0];
28970     set_old:
28971     error = -EFAULT;
28972     if (copy_to_user(oset, &old_set, sizeof(*oset)))
28973         goto out;
28974 }
28975 error = 0;
28976 out:
28977     return error;
28978 }
28979
28980 asmlinkage int
28981 sys_sigpending(old_sigset_t *set)
28982 {
28983     int error;
28984     old_sigset_t pending;
28985
28986     spin_lock_irq(&current->sigmask_lock);
28987     pending =
28988         current->blocked.sig[0] & current->signal.sig[0];
28989     spin_unlock_irq(&current->sigmask_lock);
28990
28991     error = -EFAULT;
28992     if (!copy_to_user(set, &pending, sizeof(*set)))
28993         error = 0;
28994     return error;
28995 }
28996
28997 #ifndef __sparc__
28998 asmlinkage int
28999 sys_rt_sigaction(int sig, const struct sigaction *act,
29000                 struct sigaction *oact, size_t sigsetsize)
29001 {
29002     struct k_sigaction new_sa, old_sa;
29003     int ret = -EINVAL;
29004
29005     /* XXX: Don't preclude handling different sized
29006      * sigset_t's. */
29007     if (sigsetsize != sizeof(sigset_t))
29008         goto out;
29009
29010     if (act) {
29011         if (copy_from_user(&new_sa.sa, act, sizeof(new_sa.sa)))
29012             return -EFAULT;

```

```

29013     }
29014
29015     ret = do_sigaction(sig, act ? &new_sa : NULL,
29016                      oact ? &old_sa : NULL);
29017
29018     if (!ret && oact) {
29019         if (copy_to_user(oact, &old_sa.sa, sizeof(old_sa.sa)))
29020             return -EFAULT;
29021     }
29022 out:
29023     return ret;
29024 }
29025 #endif /* __sparc__ */
29026 #endif
29027
29028 #if !defined(__alpha__)
29029 /* For backwards compatibility. Functionality superseded
29030  * by sigprocmask. */
29031 asmlinkage int
29032 sys_sgetmask(void)
29033 {
29034     /* SMP safe */
29035     return current->blocked.sig[0];
29036 }
29037
29038 asmlinkage int
29039 sys_ssetmask(int newmask)
29040 {
29041     int old;
29042
29043     spin_lock_irq(&current->sigmask_lock);
29044     old = current->blocked.sig[0];
29045
29046     siginitset(&current->blocked,
29047              newmask & ~(sigmask(SIGKILL)|sigmask(SIGSTOP)));
29048     recalc_sigpending(current);
29049     spin_unlock_irq(&current->sigmask_lock);
29050
29051     return old;
29052 }
29053
29054 /* For backwards compatibility. Functionality superseded
29055  * by sigaction. */
29056 asmlinkage unsigned long
29057 sys_signal(int sig, __sighandler_t handler)
29058 {
29059     struct k_sigaction new_sa, old_sa;
29060     int ret;

```

```

29061
29062 new_sa.sa.sa_handler = handler;
29063 new_sa.sa.sa_flags = SA_ONESHOT | SA_NOMASK;
29064
29065 ret = do_sigaction(sig, &new_sa, &old_sa);
29066
29067 return ret ? ret : (unsigned long)old_sa.sa.sa_handler;
29068 }
29069 #endif /* !alpha */

```

kernel/softirq.c

```

29070 /*
29071 *      linux/kernel/softirq.c
29072 *
29073 *      Copyright (C) 1992 Linus Torvalds
29074 *
29075 * do_bottom_half() runs at normal kernel priority: all
29076 * interrupts enabled. do_bottom_half() is atomic with
29077 * respect to itself: a bottom_half handler need not be
29078 * re-entrant.
29079 *
29080 * Fixed a disable_bh()/enable_bh() race (was causing a
29081 * console lockup) due bh_mask_count not atomic
29082 * handling. Copyright (C) 1998 Andrea Arcangeli
29083 */
29084
29085 #include <linux/mm.h>
29086 #include <linux/kernel_stat.h>
29087 #include <linux/interrupt.h>
29088 #include <linux/smp_lock.h>
29089
29090 #include <asm/io.h>
29091
29092 /* intr_count died a painless death... -DaveM */
29093
29094 atomic_t bh_mask_count[32];
29095 unsigned long bh_active = 0;
29096 unsigned long bh_mask = 0;
29097 void (*bh_base[32])(void);
29098
29099 /* This needs to make sure that only one bottom half
29100 * handler is ever active at a time. We do this without
29101 * locking by doing an atomic increment on the
29102 * intr_count, and checking (nonatomically) against
29103 * 1. Only if it's 1 do we schedule the bottom half.
29104 *
29105 * Note that the non-atomicity of the test (as opposed to

```

```

29106 * the actual update) means that the test may fail, and
29107 * _nobody_ runs the handlers if there is a race that
29108 * makes multiple CPU's get here at the same time. That's
29109 * ok, we'll run them next time around. */
29110 static inline void run_bottom_halfes(void)
29111 {
29112     unsigned long active;
29113     void (**bh)(void);
29114
29115     active = get_active_bhs();
29116     clear_active_bhs(active);
29117     bh = bh_base;
29118     do {
29119         if (active & 1)
29120             (*bh)();
29121         bh++;
29122         active >>= 1;
29123     } while (active);
29124 }
29125
29126 asmlinkage void do_bottom_half(void)
29127 {
29128     int cpu = smp_processor_id();
29129
29130     if (softirq_trylock(cpu)) {
29131         if (hardirq_trylock(cpu)) {
29132             __sti();
29133             run_bottom_halfes();
29134             __cli();
29135             hardirq_endlock(cpu);
29136         }
29137         softirq_endlock(cpu);
29138     }
29139 }

```

kernel/sys.c

```

29140 /*
29141 *      linux/kernel/sys.c
29142 *
29143 *      Copyright (C) 1991, 1992 Linus Torvalds
29144 */
29145
29146 #include <linux/mm.h>
29147 #include <linux/utsname.h>
29148 #include <linux/mman.h>
29149 #include <linux/smp_lock.h>
29150 #include <linux/notifier.h>

```

```

29151 #include <linux/reboot.h>
29152 #include <linux/prctl.h>
29153
29154 #include <asm/uaccess.h>
29155 #include <asm/io.h>
29156
29157 /* this indicates whether you can reboot with
29158  * ctrl-alt-del: the default is yes */
29159
29160 int C_A_D = 1;
29161
29162
29163 /* Notifier list for kernel code which wants to be called
29164  * at shutdown. This is used to stop any idling DMA
29165  * operations and the like. */
29166
29167 struct notifier_block *reboot_notifier_list = NULL;
29168
29169 int register_reboot_notifier(struct notifier_block * nb)
29170 {
29171     return notifier_chain_register(&reboot_notifier_list,
29172                                   nb);
29173 }
29174
29175 int unregister_reboot_notifier(struct notifier_block *nb)
29176 {
29177     return notifier_chain_unregister(&reboot_notifier_list,
29178                                     nb);
29179 }
29180
29181
29182
29183 extern void adjust_clock(void);
29184
29185 asmlinkage int sys_ni_syscall(void)
29186 {
29187     return -ENOSYS;
29188 }
29189
29190 static int proc_sel(struct task_struct *p, int which,
29191                    int who)
29192 {
29193     if(p->pid)
29194     {
29195         switch (which) {
29196             case PRIO_PROCESS:
29197                 if (!who && p == current)
29198                     return 1;

```

```

29199             return(p->pid == who);
29200         case PRIO_PGRP:
29201             if (!who)
29202                 who = current->pgrp;
29203             return(p->pgrp == who);
29204         case PRIO_USER:
29205             if (!who)
29206                 who = current->uid;
29207             return(p->uid == who);
29208     }
29209 }
29210 return 0;
29211 }
29212
29213 asmlinkage int sys_setpriority(int which, int who,
29214                               int niceval)
29215 {
29216     struct task_struct *p;
29217     unsigned int priority;
29218     int error;
29219
29220     if (which > 2 || which < 0)
29221         return -EINVAL;
29222
29223     /* normalize: avoid signed division
29224      * (rounding problems) */
29225     error = ESRCH;
29226     priority = niceval;
29227     if (niceval < 0)
29228         priority = -niceval;
29229     if (priority > 20)
29230         priority = 20;
29231     priority = (priority * DEF_PRIORITY + 10) / 20 +
29232               DEF_PRIORITY;
29233
29234     if (niceval >= 0) {
29235         priority = 2*DEF_PRIORITY - priority;
29236         if (!priority)
29237             priority = 1;
29238     }
29239
29240     read_lock(&tasklist_lock);
29241     for_each_task(p) {
29242         if (!proc_sel(p, which, who))
29243             continue;
29244         if (p->uid != current->euid &&
29245             p->uid != current->uid && !capable(CAP_SYS_NICE)) {
29246             error = EPERM;

```

p 469

p 523

p 524


```

29247     continue;
29248 }
29249 if (error == ESRCH)
29250     error = 0;
29251 if (priority > p->priority && !capable(CAP_SYS_NICE))
29252     error = EACCES;
29253 else
29254     p->priority = priority;
29255 }
29256 read_unlock(&tasklist_lock);
29257
29258 return -error;
29259 }
29260
29261 /* Ugh. To avoid negative return values, "getpriority()"
29262 * will not return the normal nice-value, but a value
29263 * that has been offset by 20 (ie it returns 0..40
29264 * instead of -20..20) */
29265 asmlinkage int sys_getpriority(int which, int who)
29266 {
29267     struct task_struct *p;
29268     long max_prio = -ESRCH;
29269
29270     if (which > 2 || which < 0)
29271         return -EINVAL;
29272
29273     read_lock(&tasklist_lock);
29274     for_each_task (p) {
29275         if (!proc_sel(p, which, who))
29276             continue;
29277         if (p->priority > max_prio)
29278             max_prio = p->priority;
29279     }
29280     read_unlock(&tasklist_lock);
29281
29282     /* scale the priority from timeslice to 0..40 */
29283     if (max_prio > 0)
29284         max_prio = (max_prio * 20 + DEF_PRIORITY/2) /
29285             DEF_PRIORITY;
29286     return max_prio;
29287 }
29288
29289
29290 /* Reboot system call: for obvious reasons only root may
29291 * call it, and even root needs to set up some magic
29292 * numbers in the registers so that some mistake won't
29293 * make this reboot the whole machine. You can also set
29294 * the meaning of the ctrl-alt-del key here.

```

```

29295 *
29296 * reboot doesn't sync: do that yourself before calling
29297 * this. */
29298 asmlinkage int sys_reboot(int magic1, int magic2,
29299     int cmd, void * arg)
29300 {
29301     char buffer[256];
29302
29303     /* We only trust the superuser with rebooting the
29304     * system. */
29305     if (!capable(CAP_SYS_BOOT))
29306         return -EPERM;
29307
29308     /* For safety, we require "magic" arguments. */
29309     if (magic1 != LINUX_REBOOT_MAGIC1 ||
29310         (magic2 != LINUX_REBOOT_MAGIC2 &&
29311          magic2 != LINUX_REBOOT_MAGIC2A &&
29312          magic2 != LINUX_REBOOT_MAGIC2B))
29313         return -EINVAL;
29314
29315     lock_kernel();
29316     switch (cmd) {
29317     case LINUX_REBOOT_CMD_RESTART:
29318         notifier_call_chain(&reboot_notifier_list,
29319             SYS_RESTART, NULL);
29320         printk(KERN_EMERG "Restarting system.\n");
29321         machine_restart(NULL);
29322         break;
29323
29324     case LINUX_REBOOT_CMD_CAD_ON:
29325         C_A_D = 1;
29326         break;
29327
29328     case LINUX_REBOOT_CMD_CAD_OFF:
29329         C_A_D = 0;
29330         break;
29331
29332     case LINUX_REBOOT_CMD_HALT:
29333         notifier_call_chain(&reboot_notifier_list,
29334             SYS_HALT, NULL);
29335         printk(KERN_EMERG "System halted.\n");
29336         machine_halt();
29337         do_exit(0);
29338         break;
29339
29340     case LINUX_REBOOT_CMD_POWER_OFF:
29341         notifier_call_chain(&reboot_notifier_list,
29342             SYS_POWER_OFF, NULL);

```

```

29343     printk(KERN_EMERG "Power down.\n");
29344     machine_power_off();
29345     do_exit(0);
29346     break;
29347
29348     case LINUX_REBOOT_CMD_RESTART2:
29349         if (strncpy_from_user(&buffer[0], (char *)arg,
29350                             sizeof(buffer) - 1) < 0) {
29351             unlock_kernel();
29352             return -EFAULT;
29353         }
29354         buffer[sizeof(buffer) - 1] = '\0';
29355
29356         notifier_call_chain(&reboot_notifier_list,
29357                             SYS_RESTART, buffer);
29358         printk(KERN_EMERG
29359                "Restarting system with command '%s'.\n",
29360                buffer);
29361         machine_restart(buffer);
29362         break;
29363
29364     default:
29365         unlock_kernel();
29366         return -EINVAL;
29367         break;
29368 };
29369 unlock_kernel();
29370 return 0;
29371 }
29372
29373 /* This function gets called by ctrl-alt-del - ie the
29374 * keyboard interrupt. As it's called within an
29375 * interrupt, it may NOT sync: the only choice is whether
29376 * to reboot at once, or just ignore the ctrl-alt-del.
29377 */
29378 void ctrl_alt_del(void)
29379 {
29380     if (C_A_D) {
29381         notifier_call_chain(&reboot_notifier_list,
29382                             SYS_RESTART, NULL);
29383         machine_restart(NULL);
29384     } else
29385         kill_proc(1, SIGINT, 1);
29386 }
29387
29388
29389 /* Unprivileged users may change the real gid to the
29390 * effective gid or vice versa. (BSD-style)

```

```

29391 *
29392 * If you set the real gid at all, or set the effective
29393 * gid to a value not equal to the real gid, then the
29394 * saved gid is set to the new effective gid.
29395 *
29396 * This makes it possible for a setgid program to
29397 * completely drop its privileges, which is often a
29398 * useful assertion to make when you are doing a security
29399 * audit over a program.
29400 *
29401 * The general idea is that a program which uses just
29402 * setregid() will be 100% compatible with BSD. A
29403 * program which uses just setgid() will be 100%
29404 * compatible with POSIX with saved IDs.
29405 *
29406 * SMP: There are not races, the GIDs are checked only by
29407 * filesystem operations (as far as semantic preservation
29408 * is concerned). */
29409 asmlinkage int sys_setregid(gid_t rgid, gid_t egid)
29410 {
29411     int old_rgid = current->gid;
29412     int old_egid = current->egid;
29413
29414     if (rgid != (gid_t) -1) {
29415         if ((old_rgid == rgid) ||
29416             (current->egid == rgid) ||
29417             capable(CAP_SETGID))
29418             current->gid = rgid;
29419         else
29420             return -EPERM;
29421     }
29422     if (egid != (gid_t) -1) {
29423         if ((old_rgid == egid) ||
29424             (current->egid == egid) ||
29425             (current->sgid == egid) ||
29426             capable(CAP_SETGID))
29427             current->fsgid = current->egid = egid;
29428         else {
29429             current->gid = old_rgid;
29430             return -EPERM;
29431         }
29432     }
29433     if (rgid != (gid_t) -1 ||
29434         (egid != (gid_t) -1 && egid != old_rgid))
29435         current->sgid = current->egid;
29436     current->fsgid = current->egid;
29437     if (current->egid != old_egid)
29438         current->dumpable = 0;

```

```

29439 return 0;
29440 }
29441
29442 /* setgid() is implemented like SysV w/ SAVED_IDS
29443 *
29444 * SMP: Same implicit races as above. */
29445 asmlinkage int sys_setgid(gid_t gid)
29446 {
29447     int old_egid = current->egid;
29448
29449     if (capable(CAP_SETGID))
29450         current->gid = current->egid - current->sgid -
29451             current->fsgid = gid;
29452     else if ((gid == current->gid) ||
29453             (gid == current->sgid))
29454         current->egid - current->fsgid - gid;
29455     else
29456         return -EPERM;
29457
29458     if (current->egid != old_egid)
29459         current->dumpable = 0;
29460     return 0;
29461 }
29462
29463 /* cap_emulate_setuid() fixes the effective / permitted
29464 * capabilities of a process after a call to setuid,
29465 * setreuid, or setresuid.
29466 *
29467 * 1) When set*uiding _from_ one of {r,e,s}uid == 0 _to_
29468 * all of {r,e,s}uid != 0, the permitted and effective
29469 * capabilities are cleared.
29470 *
29471 * 2) When set*uiding _from_ euid == 0 _to_ euid != 0,
29472 * the effective capabilities of the process are
29473 * cleared.
29474 *
29475 * 3) When set*uiding _from_ euid != 0 _to_ euid == 0,
29476 * the effective capabilities are set to the permitted
29477 * capabilities.
29478 *
29479 * fsuid is handled elsewhere. fsuid == 0 and
29480 * {r,e,s}uid != 0 should never happen.
29481 *
29482 * -astor */
29483 extern inline void cap_emulate setxuid(int old_ruid,
29484                                       int old_euid,
29485                                       int old_suid)
29486 {

```

```

29487     if ((old_ruid == 0 || old_euid == 0 || old_suid == 0)
29488         && (current->uid != 0 && current->euid != 0 &&
29489             current->suid != 0)) {
29490         cap_clear(current->cap_permitted);
29491         cap_clear(current->cap_effective);
29492     }
29493     if (old_euid == 0 && current->euid != 0) {
29494         cap_clear(current->cap_effective);
29495     }
29496     if (old_euid != 0 && current->euid == 0) {
29497         current->cap_effective = current->cap_permitted;
29498     }
29499 }
29500
29501 /* Unprivileged users may change the real uid to the
29502 * effective uid or vice versa. (BSD-style)
29503 *
29504 * If you set the real uid at all, or set the effective
29505 * uid to a value not equal to the real uid, then the
29506 * saved uid is set to the new effective uid.
29507 *
29508 * This makes it possible for a setuid program to
29509 * completely drop its privileges, which is often a
29510 * useful assertion to make when you are doing a security
29511 * audit over a program.
29512 *
29513 * The general idea is that a program which uses just
29514 * setreuid() will be 100% compatible with BSD. A
29515 * program which uses just setuid() will be 100%
29516 * compatible with POSIX with saved IDs. */
29517 asmlinkage int sys_setreuid(uid_t ruid, uid_t euid)
29518 {
29519     int old_ruid, old_euid, old_suid, new_ruid;
29520
29521     new_ruid = old_ruid = current->uid;
29522     old_euid = current->euid;
29523     old_suid = current->suid;
29524     if (ruid != (uid_t) -1) {
29525         if ((old_ruid == ruid) ||
29526             (current->euid == ruid) ||
29527             capable(CAP_SETUID))
29528             new_ruid = ruid;
29529         else
29530             return -EPERM;
29531     }
29532     if (euid != (uid_t) -1) {
29533         if ((old_ruid == euid) ||
29534             (current->euid == euid) ||

```

```

29535     (current->suid == euid) ||
29536     capable(CAP_SETUID))
29537     current->fsuid = current->euid = euid;
29538     else
29539     return -EPERM;
29540 }
29541 if (ruid != (uid_t) -1 ||
29542     (euid != (uid_t) -1 && euid != old_ruid))
29543     current->suid = current->euid;
29544     current->fsuid = current->euid;
29545     if (current->euid != old_euid)
29546         current->dumpable = 0;
29547
29548     if(new_ruid != old_ruid) {
29549         /* What if a process setreuid()'s and this brings the
29550          * new uid over his NPROC rlimit? We can check this
29551          * now cheaply with the new uid cache, so if it
29552          * matters we should be checking for it. -DaveM */
29553         free_uid(current);
29554         current->uid = new_ruid;
29555         alloc_uid(current);
29556     }
29557
29558     if (!issecure(SECURE_NO_SETUID_FIXUP)) {
29559         cap_emulate_setxuid(old_ruid, old_euid, old_suid);
29560     }
29561
29562     return 0;
29563 }
29564
29565
29566
29567 /* setuid() is implemented like SysV with SAVED_IDS
29568 *
29569 * Note that SAVED_ID's is deficient in that a setuid
29570 * root program like sendmail, for example, cannot set
29571 * its uid to be a normal user and then switch back,
29572 * because if you're root, setuid() sets the saved uid
29573 * too. If you don't like this, blame the bright people
29574 * in the POSIX committee and/or USG. Note that the
29575 * BSD-style setreuid() will allow a root program to
29576 * temporarily drop privileges and be able to regain them
29577 * by swapping the real and effective uid. */
29578 asmlinkage int sys_setuid(uid_t uid)
29579 {
29580     int old_euid = current->euid;
29581     int old_ruid, old_suid, new_ruid;
29582

```

```

29583     old_ruid = new_ruid = current->uid;
29584     old_suid = current->suid;
29585     if (capable(CAP_SETUID))
29586         new_ruid = current->euid = current->suid =
29587         current->fsuid = uid;
29588     else if ((uid == current->uid) ||
29589             (uid == current->suid))
29590         current->fsuid = current->euid = uid;
29591     else
29592         return -EPERM;
29593
29594     if (current->euid != old_euid)
29595         current->dumpable = 0;
29596
29597     if (new_ruid != old_ruid) {
29598         /* See comment above about NPROC rlimit issues... */
29599         free_uid(current);
29600         current->uid = new_ruid;
29601         alloc_uid(current);
29602     }
29603
29604     if (!issecure(SECURE_NO_SETUID_FIXUP)) {
29605         cap_emulate_setxuid(old_ruid, old_euid, old_suid);
29606     }
29607
29608     return 0;
29609 }
29610
29611
29612 /* This function implements a generic ability to update
29613 * ruid, euid, and suid. This allows you to implement
29614 * the 4.4 compatible setreuid(). */
29615 asmlinkage int sys_setresuid(uid_t ruid, uid_t euid,
29616                             uid_t suid)
29617 {
29618     int old_ruid = current->uid;
29619     int old_euid = current->euid;
29620     int old_suid = current->suid;
29621
29622     if (!capable(CAP_SETUID)) {
29623         if ((ruid != (uid_t) -1) && (ruid != current->uid) &&
29624             (ruid != current->euid) &&
29625             (ruid != current->suid))
29626             return -EPERM;
29627         if ((euid != (uid_t) -1) && (euid != current->uid) &&
29628             (euid != current->euid) &&
29629             (euid != current->suid))
29630             return -EPERM;

```

```

29631     if ((suid != (uid_t) -1) && (suid != current->uid) &&
29632         (suid != current->euid) &&
29633         (suid != current->suid))
29634         return -EPERM;
29635     }
29636     if (ruid != (uid_t) -1) {
29637         /* See above commentary about NPROC rlimit issues
29638          * here. */
29639         free_uid(current);
29640         current->uid = ruid;
29641         alloc_uid(current);
29642     }
29643     if (euid != (uid_t) -1) {
29644         if (euid != current->euid)
29645             current->dumpable = 0;
29646         current->euid = euid;
29647         current->fsuid = euid;
29648     }
29649     if (suid != (uid_t) -1)
29650         current->suid = suid;
29651
29652     if (!issecure(SECURE_NO_SETUID_FIXUP)) {
29653         cap_emulate_setxuid(old_ruid, old_euid, old_suid);
29654     }
29655
29656     return 0;
29657 }
29658
29659 asmlinkage int sys_getresuid(uid_t *ruid, uid_t *euid,
29660                             uid_t *suid)
29661 {
29662     int retval;
29663
29664     if (!(retval = put_user(current->uid, ruid)) &&
29665         !(retval = put_user(current->euid, euid)))
29666         retval = put_user(current->suid, suid);
29667
29668     return retval;
29669 }
29670
29671 /* Same as above, but for rgid, egid, sgid. */
29672 asmlinkage int sys_setresgid(gid_t rgid, gid_t egid,
29673                             gid_t sgid)
29674 {
29675     if (!capable(CAP_SETGID)) {
29676         if ((rgid != (gid_t) -1) &&
29677             (rgid != current->gid) &&
29678             (rgid != current->egid) &&

```

```

29679             (rgid != current->sgid))
29680         return -EPERM;
29681     if ((egid != (gid_t) -1) &&
29682         (egid != current->gid) &&
29683         (egid != current->egid) &&
29684         (egid != current->sgid))
29685         return -EPERM;
29686     if ((sgid != (gid_t) -1) &&
29687         (sgid != current->gid) &&
29688         (sgid != current->egid) &&
29689         (sgid != current->sgid))
29690         return -EPERM;
29691     }
29692     if (rgid != (gid_t) -1)
29693         current->gid = rgid;
29694     if (egid != (gid_t) -1) {
29695         if (egid != current->egid)
29696             current->dumpable = 0;
29697         current->egid = egid;
29698         current->fsgid = egid;
29699     }
29700     if (sgid != (gid_t) -1)
29701         current->sgid = sgid;
29702     return 0;
29703 }
29704
29705 asmlinkage int sys_getresgid(gid_t *rgid, gid_t *egid,
29706                             gid_t *sgid)
29707 {
29708     int retval;
29709
29710     if (!(retval = put_user(current->gid, rgid)) &&
29711         !(retval = put_user(current->egid, egid)))
29712         retval = put_user(current->sgid, sgid);
29713
29714     return retval;
29715 }
29716
29717
29718 /* "setfsuid()" sets the fsuid - the uid used for
29719  * filesystem checks. This is used for "access()" and for
29720  * the NFS daemon (letting nfsd stay at whatever uid it
29721  * wants to). It normally shadows "euid". except when
29722  * explicitly set by setfsuid() or for access.. */
29723 asmlinkage int sys_setfsuid(uid_t uid)
29724 {
29725     int old_fsuid;
29726

```

```

29727 old_fsuuid = current->fsuid;
29728 if (uid == current->uid || uid == current->euid ||
29729     uid == current->suid || uid == current->fsuid ||
29730     capable(CAP_SETUID))
29731     current->fsuid = uid;
29732 if (current->fsuid != old_fsuuid)
29733     current->dumpable = 0;
29734
29735 /* We emulate fsuid by essentially doing a scaled-down
29736 * version of what we did in setresuid and
29737 * friends. However, we only operate on the fs-specific
29738 * bits of the process' effective capabilities
29739 *
29740 * FIXME - is fsuser used for all CAP_FS_MASK
29741 * capabilities? if not, we might be a bit too harsh
29742 * here. */
29743 if (!issecure(SECURE_NO_SETUID_FIXUP)) {
29744     if (old_fsuuid == 0 && current->fsuid != 0) {
29745         cap_t(current->cap_effective) &= ~CAP_FS_MASK;
29746     }
29747     if (old_fsuuid != 0 && current->fsuid == 0) {
29748         cap_t(current->cap_effective) |=
29749             (cap_t(current->cap_permitted) & CAP_FS_MASK);
29750     }
29751 }
29752
29753 return old_fsuuid;
29754 }
29755
29756 /* Samma på svenska. */
29757 asmlinkage int sys_setfsgid(gid_t gid)
29758 {
29759     int old_fsgid;
29760
29761     old_fsgid = current->fsgid;
29762     if (gid == current->gid || gid == current->egid ||
29763         gid == current->sgid || gid == current->fsgid ||
29764         capable(CAP_SETGID))
29765         current->fsgid = gid;
29766     if (current->fsgid != old_fsgid)
29767         current->dumpable = 0;
29768
29769     return old_fsgid;
29770 }
29771
29772 asmlinkage long sys_times(struct tms * tbuf)
29773 {
29774     /* In the SMP world we might just be unlucky and have
29775      * one of the times increment as we use it. Since the
29776      * value is an atomically safe type this is just
29777      * fine. Conceptually it's as if the syscall took an
29778      * instant longer to occur. */
29779     if (tbuf)
29780         if (copy_to_user(tbuf, &current->times,
29781                         sizeof(struct tms)))
29782             return -EFAULT;
29783     return jiffies;
29784 }
29785
29786 /* This needs some heavy checking ... I just haven't the
29787 * stomach for it. I also don't fully understand
29788 * sessions/pggrp etc. Let somebody who does explain it.
29789 *
29790 * OK, I think I have the protection semantics
29791 * right... this is really only important on a
29792 * multi-user system anyway, to make sure one user can't
29793 * send a signal to a process owned by another. -TYT,
29794 * 12/12/91
29795 *
29796 * Auch. Had to add the 'did_exec' flag to conform
29797 * completely to POSIX. LBT 04.03.94 */
29798 asmlinkage int sys_setpgid(pid_t pid, pid_t pgid)
29799 {
29800     struct task_struct * p;
29801     int err = -EINVAL;
29802
29803     if (!pid)
29804         pid = current->pid;
29805     if (!pgid)
29806         pgid = pid;
29807     if (pgid < 0)
29808         return -EINVAL;
29809
29810     /* From this point forward we keep holding onto the
29811      * tasklist lock so that our parent does not change
29812      * from under us. -DaveM */
29813     read_lock(&tasklist_lock);
29814
29815     err = -ESRCH;
29816     p = find_task_by_pid(pid);
29817     if (!p)
29818         goto out;
29819
29820     if (p->p_pptr == current || p->p_opptr == current) {
29821         err = -EPERM;
29822         if (p->session != current->session)

```

```

29823     goto out;
29824     err = -EACCES;
29825     if (p->did_exec)
29826         goto out;
29827 ) else if (p != current)
29828     goto out;
29829 err = -EPERM;
29830 if (p->leader)
29831     goto out;
29832 if (pgid != pid) {
29833     struct task_struct * tmp;
29834     for_each_task (tmp) {
29835         if (tmp->pgrp == pgid &&
29836             tmp->session == current->session)
29837             goto ok_pgid;
29838     }
29839     goto out;
29840 }
29841
29842 ok_pgid:
29843 p->pgrp = pgid;
29844 err = 0;
29845 out:
29846 /* All paths lead to here, thus we are safe. -DaveM */
29847 read_unlock(&tasklist_lock);
29848 return err;
29849 }
29850
29851 asmlinkage int sys_getpgid(pid_t pid)
29852 {
29853     if (!pid) {
29854         return current->pgrp;
29855     } else {
29856         int retval;
29857         struct task_struct *p;
29858
29859         read_lock(&tasklist_lock);
29860         p = find_task_by_pid(pid);
29861
29862         retval = -ESRCH;
29863         if (p)
29864             retval = p->pgrp;
29865         read_unlock(&tasklist_lock);
29866         return retval;
29867     }
29868 }
29869
29870 asmlinkage int sys_getpgrp(void)

```

```

29871 {
29872     /* SMP: assuming writes are word atomic this is fine */
29873     return current->pgrp;
29874 }
29875
29876 asmlinkage int sys_getsid(pid_t pid)
29877 {
29878     if (!pid) {
29879         return current->session;
29880     } else {
29881         int retval;
29882         struct task_struct *p;
29883
29884         read_lock(&tasklist_lock);
29885         p = find_task_by_pid(pid);
29886
29887         retval = -ESRCH;
29888         if (p)
29889             retval = p->session;
29890         read_unlock(&tasklist_lock);
29891         return retval;
29892     }
29893 }
29894
29895 asmlinkage int sys_setsid(void)
29896 {
29897     struct task_struct * p;
29898     int err = -EPERM;
29899
29900     read_lock(&tasklist_lock);
29901     for_each_task(p) {
29902         if (p->pgrp == current->pid)
29903             goto out;
29904     }
29905
29906     current->leader = 1;
29907     current->session = current->pgrp = current->pid;
29908     current->tty = NULL;
29909     current->tty_old_pgrp = 0;
29910     err = current->pgrp;
29911 out:
29912     read_unlock(&tasklist_lock);
29913     return err;
29914 }
29915
29916 /* Supplementary group IDs */
29917 asmlinkage int sys_getgroups(int gidsetsize,
29918                             gid_t *grouplist)

```

```

29919 {
29920     int i;
29921
29922     /* SMP: Nobody else can change our grouplist. Thus we
29923      * are safe. */
29924     if (gidsetsize < 0)
29925         return -EINVAL;
29926     i = current->ngroups;
29927     if (gidsetsize) {
29928         if (i > gidsetsize)
29929             return -EINVAL;
29930         if (copy_to_user(grouplist, current->groups,
29931             sizeof(gid_t)*i))
29932             return -EFAULT;
29933     }
29934     return i;
29935 }
29936
29937 /* SMP: Our groups are not shared. We can copy to/from
29938  * them safely without another task interfering. */
29939 asmlinkage int sys_setgroups(int gidsetsize,
29940     gid_t *grouplist)
29941 {
29942     if (!capable(CAP_SETGID))
29943         return -EPERM;
29944     if ((unsigned) gidsetsize > NGROUPS)
29945         return -EINVAL;
29946     if (copy_from_user(current->groups, grouplist,
29947         gidsetsize * sizeof(gid_t)))
29948         return -EFAULT;
29949     current->ngroups = gidsetsize;
29950     return 0;
29951 }
29952
29953 int in_group_p(gid_t grp)
29954 {
29955     if (grp != current->fsgid) {
29956         int i = current->ngroups;
29957         if (i) {
29958             gid_t *groups = current->groups;
29959             do {
29960                 if (*groups == grp)
29961                     goto out;
29962                 groups++;
29963                 i--;
29964             } while (i);
29965         }
29966         return 0;

```

```

29967     }
29968 out:
29969     return 1;
29970 }
29971
29972 /* This should really be a blocking read-write lock
29973  * rather than a semaphore. Anybody want to implement
29974  * one? */
29975 struct semaphore uts_sem = MUTEX;
29976
29977 asmlinkage int sys_newuname(struct new_utsname * name)
29978 {
29979     int errno = 0;
29980
29981     down(&uts_sem);
29982     if (copy_to_user(name, &system_utsname, sizeof *name))
29983         errno = -EFAULT;
29984     up(&uts_sem);
29985     return errno;
29986 }
29987
29988 asmlinkage int sys_sethostname(char *name, int len)
29989 {
29990     int errno;
29991
29992     if (!capable(CAP_SYS_ADMIN))
29993         return -EPERM;
29994     if (len < 0 || len > __NEW_UTS_LEN)
29995         return -EINVAL;
29996     down(&uts_sem);
29997     errno = -EFAULT;
29998     if (!copy_from_user(system_utsname.nodename, name,
29999         len)) {
30000         system_utsname.nodename[len] = 0;
30001         errno = 0;
30002     }
30003     up(&uts_sem);
30004     return errno;
30005 }
30006
30007 asmlinkage int sys_gethostname(char *name, int len)
30008 {
30009     int i, errno;
30010
30011     if (len < 0)
30012         return -EINVAL;
30013     down(&uts_sem);
30014     i = 1 + strlen(system_utsname.nodename);

```



```

30015 if (i > len)
30016     i = len;
30017 errno = 0;
30018 if (copy_to_user(name, system_utsname.nodename, i))
30019     errno = -EFAULT;
30020 up(&uts_sem);
30021 return errno;
30022 }
30023
30024 /* Only setdomainname; getdomainname can be implemented
30025  * by calling uname() */
30026 asmlinkage int sys_setdomainname(char *name, int len)
30027 {
30028     int errno;
30029
30030     if (!capable(CAP_SYS_ADMIN))
30031         return -EPERM;
30032     if (len < 0 || len > __NEW_UTS_LEN)
30033         return -EINVAL;
30034
30035     down(&uts_sem);
30036     errno = -EFAULT;
30037     if (!copy_from_user(system_utsname.domainname, name,
30038                         len)) {
30039         errno = 0;
30040         system_utsname.domainname[len] = 0;
30041     }
30042     up(&uts_sem);
30043     return errno;
30044 }
30045
30046 asmlinkage int sys_getrlimit(unsigned int resource,
30047                               struct rlimit *rlim)
30048 {
30049     if (resource >= RLIM_NLIMITS)
30050         return -EINVAL;
30051     else
30052         return copy_to_user(rlim, current->rlim + resource,
30053                             sizeof(*rlim))
30054             ? -EFAULT : 0;
30055 }
30056
30057 asmlinkage int sys_setrlimit(unsigned int resource,
30058                               struct rlimit *rlim)
30059 {
30060     struct rlimit new_rlim, *old_rlim;
30061
30062     if (resource >= RLIM_NLIMITS)

```

```

30063         return -EINVAL;
30064     if(copy_from_user(&new_rlim, rlim, sizeof(*rlim)))
30065         return -EFAULT;
30066     old_rlim = current->rlim + resource;
30067     if (((new_rlim.rlim_cur > old_rlim->rlim_max) ||
30068         (new_rlim.rlim_max > old_rlim->rlim_max)) &&
30069         !capable(CAP_SYS_RESOURCE))
30070         return -EPERM;
30071     if (resource == RLIMIT_NOFILE) {
30072         if (new_rlim.rlim_cur > NR_OPEN ||
30073             new_rlim.rlim_max > NR_OPEN)
30074             return -EPERM;
30075     }
30076     *old_rlim = new_rlim;
30077     return 0;
30078 }
30079
30080 /* It would make sense to put struct rusage in the
30081  * task_struct, except that would make the task_struct be
30082  * *really big*. After task_struct gets moved into
30083  * malloc'ed memory, it would make sense to do this. It
30084  * will make moving the rest of the information a lot
30085  * simpler! (Which we're not doing right now because
30086  * we're not measuring them yet).
30087  *
30088  * This is SMP safe. Either we are called from
30089  * sys.getrusage on ourselves below (we know we aren't
30090  * going to exit/disappear and only we change our rusage
30091  * counters), or we are called from wait4() on a process
30092  * which is either stopped or zombied. In the zombied
30093  * case the task won't get reaped till shortly after the
30094  * call to getrusage(), in both cases the task being
30095  * examined is in a frozen state so the counters won't
30096  * change. */
30097 int getrusage(struct task_struct *p, int who,
30098              struct rusage *ru)
30099 {
30100     struct rusage r;
30101
30102     memset((char *) &r, 0, sizeof(r));
30103     switch (who) {
30104     case RUSAGE_SELF:
30105         r.ru_utime.tv_sec = CT_TO_SECS(p->times.tms_utime);
30106         r.ru_utime.tv_usec = CT_TO_USECS(p->times.tms_utime);
30107         r.ru_stime.tv_sec = CT_TO_SECS(p->times.tms_stime);
30108         r.ru_stime.tv_usec = CT_TO_USECS(p->times.tms_stime);
30109         r.ru_minflt = p->min_flt;
30110         r.ru_majflt = p->maj_flt;

```

```

30111     r.ru_nswap = p->nswap;
30112     break;
30113 case RUSAGE_CHILDREN:
30114     r.ru_utime.tv_sec =
30115         CT_TO_SECS(p->times.tms_cutime);
30116     r.ru_utime.tv_usec =
30117         CT_TO_USECS(p->times.tms_cutime);
30118     r.ru_stime.tv_sec =
30119         CT_TO_SECS(p->times.tms_cstime);
30120     r.ru_stime.tv_usec =
30121         CT_TO_USECS(p->times.tms_cstime);
30122     r.ru_minflt = p->cmin_flt;
30123     r.ru_majflt = p->cmaj_flt;
30124     r.ru_nswap = p->cnswap;
30125     break;
30126 default:
30127     r.ru_utime.tv_sec = CT_TO_SECS(p->times.tms_utime
30128         + p->times.tms_cutime);
30129     r.ru_utime.tv_usec = CT_TO_USECS(p->times.tms_utime
30130         + p->times.tms_cutime);
30131     r.ru_stime.tv_sec = CT_TO_SECS(p->times.tms_stime
30132         + p->times.tms_cstime);
30133     r.ru_stime.tv_usec = CT_TO_USECS(p->times.tms_stime
30134         + p->times.tms_cstime);
30135     r.ru_minflt = p->min_flt + p->cmin_flt;
30136     r.ru_majflt = p->maj_flt + p->cmaj_flt;
30137     r.ru_nswap = p->nswap + p->cnswap;
30138     break;
30139 }
30140 return copy_to_user(ru, &r, sizeof(r)) ? -EFAULT : 0;
30141 }
30142
30143 asmlinkage int sys_getrusage(int who, struct rusage *ru)
30144 {
30145     if (who != RUSAGE_SELF && who != RUSAGE_CHILDREN)
30146         return -EINVAL;
30147     return getrusage(current, who, ru);
30148 }
30149
30150 asmlinkage int sys_umask(int mask)
30151 {
30152     mask = xchg(&current->fs->umask, mask & S_IRWXUGO);
30153     return mask;
30154 }
30155
30156 asmlinkage int sys_prctl(int option, unsigned long arg2,
30157     unsigned long arg3, unsigned long arg4,
30158     unsigned long arg5)

```

```

30159 {
30160     int error = 0;
30161     int sig;
30162
30163     switch (option) {
30164         case PR_SET_PDEATHSIG:
30165             sig = arg2;
30166             if (sig > _NSIG) {
30167                 error = -EINVAL;
30168                 break;
30169             }
30170             current->pdeath_signal = sig;
30171             break;
30172         default:
30173             error = -EINVAL;
30174             break;
30175     }
30176     return error;
30177 }
30178

```

kernel/sysctl.c

```

30179 /*
30180 * sysctl.c: General linux system control interface
30181 *
30182 * Begun 24 March 1995, Stephen Tweedie
30183 * Added /proc support, Dec 1995
30184 * Added bdflush entry and intvec min/max checking,
30185 * 2/23/96, Tom Dyas.
30186 * Added hooks for /proc/sys/net (minor, minor patch),
30187 * 96/4/1, Mike Shaver.
30188 * Added kernel/java-{interpreter,appletviewer}, 96/5/10,
30189 * Mike Shaver.
30190 * Dynamic registration fixes, Stephen Tweedie.
30191 * Added kswapd-interval, ctrl-alt-del, printk stuff,
30192 * 1/8/97, Chris Horn.
30193 * Made sysctl support optional via CONFIG_SYSCTL.
30194 * 1/10/97, Chris Horn. */
30195
30196 #include <linux/config.h>
30197 #include <linux/malloc.h>
30198 #include <linux/sysctl.h>
30199 #include <linux/swapctl.h>
30200 #include <linux/proc_fs.h>
30201 #include <linux/ctype.h>
30202 #include <linux/utsname.h>
30203 #include <linux/swapctl.h>

```

```

30204 #include <linux/smp_lock.h>
30205 #include <linux/init.h>
30206
30207 #include <asm/uaccess.h>
30208
30209 #ifdef CONFIG_ROOT_NFS
30210 #include <linux/nfs_fs.h>
30211 #endif
30212
30213 #if defined(CONFIG_SYSCTL)
30214
30215 /* External variables not in a header file. */
30216 extern int panic_timeout;
30217 extern int console_loglevel, C_A_D;
30218 extern int bdf_prm[], bdflush_min[], bdflush_max[];
30219 extern char binfmt_java_interpreter[],
30220         binfmt_java_appletviewer[];
30221 extern int sysctl_overcommit_memory;
30222 extern int nr_queued_signals, max_queued_signals;
30223
30224 #ifdef CONFIG_KMOD
30225 extern char modprobe_path[];
30226 #endif
30227 #ifdef CONFIG_CHR_DEV_SG
30228 extern int sg_big_buff;
30229 #endif
30230 #ifdef CONFIG_SYSVIPC
30231 extern int shmmax;
30232 #endif
30233
30234 #ifdef __sparc__
30235 extern char reboot_command [];
30236 #endif
30237 #ifdef __powerpc__
30238 extern unsigned long htab_reclaim_on, zero_paged_on,
30239         powersave_nap;
30240 int proc_dol2crvec(ctl_table *table, int write,
30241         struct file *filp, void *buffer, size_t *lenp);
30242 #endif
30243
30244 #ifdef CONFIG_BSD_PROCESS_ACCT
30245 extern int acct_parm[];
30246 #endif
30247
30248 extern int pgt_cache_water[];
30249
30250 static int parse_table(int *, int, void *, size_t *,
30251         void *, size_t, ctl_table *, void **);

```

```

30252 static int proc_doutsstring(ctl_table *table, int write,
30253         struct file *filp, void *buffer, size_t *lenp);
30254
30255 static ctl_table root_table[];
30256 static struct ctl_table_header root_table_header =
30257         {root_table, DNODE_SINGLE(&root_table_header)};
30258
30259 static ctl_table kern_table[];
30260 static ctl_table vm_table[];
30261 #ifdef CONFIG_NET
30262 extern ctl_table net_table[];
30263 #endif
30264 static ctl_table proc_table[];
30265 static ctl_table fs_table[];
30266 static ctl_table debug_table[];
30267 static ctl_table dev_table[];
30268
30269
30270 /* /proc declarations: */
30271
30272 #ifdef CONFIG_PROC_FS
30273
30274 static ssize_t proc_readsys(struct file *, char *,
30275         size_t, loff_t *);
30276 static ssize_t proc_writesys(struct file *, const char *,
30277         size_t, loff_t *);
30278 static int proc_sys_permission(struct inode *, int);
30279
30280 struct file_operations proc_sys_file_operations =
30281 {
30282     NULL,          /* lseek */
30283     proc_readsys, /* read */
30284     proc_writesys, /* write */
30285     NULL,         /* readdir */
30286     NULL,        /* poll */
30287     NULL,        /* ioctl */
30288     NULL,        /* mmap */
30289     NULL,        /* no special open code */
30290     NULL,        /* no special flush code */
30291     NULL,        /* no special release code */
30292     NULL,        /* can't fsync */
30293 };
30294
30295 struct inode_operations proc_sys_inode_operations =
30296 {
30297     &proc_sys_file_operations,
30298     NULL,          /* create */
30299     NULL,          /* lookup */

```

```

30300 NULL,          /* link */
30301 NULL,          /* unlink */
30302 NULL,          /* symlink */
30303 NULL,          /* mkdir */
30304 NULL,          /* rmdir */
30305 NULL,          /* mknod */
30306 NULL,          /* rename */
30307 NULL,          /* readlink */
30308 NULL,          /* follow_link */
30309 NULL,          /* readpage */
30310 NULL,          /* writepage */
30311 NULL,          /* bmap */
30312 NULL,          /* truncate */
30313 proc_sys_permission
30314 };
30315
30316 extern struct proc_dir_entry proc_sys_root;
30317
30318 static void register_proc_table(ctl_table *,
30319                               struct proc_dir_entry *);
30320 static void unregister_proc_table(ctl_table *,
30321                                  struct proc_dir_entry *);
30322 #endif
30323 extern int inodes_stat[];
30324 extern int dentry_stat[];
30325
30326 /* The default sysctl tables: */
30327
30328 static ctl_table root_table[] = {
30329     {CTL_KERN, "kernel", NULL, 0, 0555, kern_table},
30330     {CTL_VM, "vm", NULL, 0, 0555, vm_table},
30331 #ifdef CONFIG_NET
30332     {CTL_NET, "net", NULL, 0, 0555, net_table},
30333 #endif
30334     {CTL_PROC, "proc", NULL, 0, 0555, proc_table},
30335     {CTL_FS, "fs", NULL, 0, 0555, fs_table},
30336     {CTL_DEBUG, "debug", NULL, 0, 0555, debug_table},
30337     {CTL_DEV, "dev", NULL, 0, 0555, dev_table},
30338     {0}
30339 };
30340
30341 static ctl_table kern_table[] = {
30342     {KERN_OSTYPE, "ostype", system_utsname.sysname, 64,
30343      0444, NULL, &proc_doutsstring, &sysctl_string},
30344     {KERN_OSRELEASE, "osrelease", system_utsname.release,
30345      64, 0444, NULL, &proc_doutsstring, &sysctl_string},
30346     {KERN_VERSION, "version", system_utsname.version, 64,
30347      0444, NULL, &proc_doutsstring, &sysctl_string},

```

```

30348     {KERN_NODENAME, "hostname", system_utsname.nodename,
30349      64, 0644, NULL, &proc_doutsstring, &sysctl_string},
30350     {KERN_DOMAINNAME, "domainname",
30351      system_utsname.domainname, 64,
30352      0644, NULL, &proc_doutsstring, &sysctl_string},
30353     {KERN_PANIC, "panic", &panic_timeout, sizeof(int),
30354      0644, NULL, &proc_dointvec},
30355 #ifdef CONFIG_BLK_DEV_INITRD
30356     {KERN_REALROOTDEV, "real-root-dev", &real_root_dev,
30357      sizeof(int), 0644, NULL, &proc_dointvec},
30358 #endif
30359 #ifdef CONFIG_BINFORMAT_JAVA
30360     {KERN_JAVA_INTERPRETER, "java-interpreter",
30361      binfmt_java_interpreter,
30362      64, 0644, NULL, &proc_dostring, &sysctl_string },
30363     {KERN_JAVA_APPLETVIEWER, "java-appletviewer",
30364      binfmt_java_appletviewer,
30365      64, 0644, NULL, &proc_dostring, &sysctl_string },
30366 #endif
30367 #ifdef __sparc__
30368     {KERN_SPARC_REBOOT, "reboot-cmd", reboot_command,
30369      256, 0644, NULL, &proc_dostring, &sysctl_string },
30370 #endif
30371 #ifdef __powerpc__
30372     {KERN_PPC_HTABRECLAIM, "htab-reclaim",
30373      &htab_reclaim_on, sizeof(int),
30374      0644, NULL, &proc_dointvec},
30375     {KERN_PPC_ZEROPAGED, "zero-paged", &zero_paged_on,
30376      sizeof(int), 0644, NULL, &proc_dointvec},
30377     {KERN_PPC_POWERSAVE_NAP, "powersave-nap",
30378      &powersave_nap, sizeof(int),
30379      0644, NULL, &proc_dointvec},
30380     {KERN_PPC_L2CR, "l2cr", NULL, 0,
30381      0644, NULL, &proc_dol2crvec},
30382 #endif
30383     {KERN_CTLALTD, "ctrl-alt-del", &C_A_D, sizeof(int),
30384      0644, NULL, &proc_dointvec},
30385     {KERN_PRINTK, "printk", &console_loglevel,
30386      4*sizeof(int), 0644, NULL, &proc_dointvec},
30387 #ifdef CONFIG_KMOD
30388     {KERN_MODPROBE, "modprobe", &modprobe_path, 256,
30389      0644, NULL, &proc_dostring, &sysctl_string },
30390 #endif
30391 #ifdef CONFIG_CHR_DEV_SG
30392     {KERN_SG_BIG_BUFF, "sg-big-buff", &sg_big_buff,
30393      sizeof(int), 0444, NULL, &proc_dointvec},
30394 #endif
30395 #ifdef CONFIG_BSD_PROCESS_ACCT

```

```

30396 {KERN_ACCT, "acct", &acct_parm, 3*sizeof(int),
30397 0644, NULL, &proc_dointvec},
30398 #endif
30399 {KERN_RTSGNRR, "rtsig-nr", &nr_queued_signals,
30400 sizeof(int), 0444, NULL, &proc_dointvec},
30401 {KERN_RTSGMAX, "rtsig-max", &max_queued_signals,
30402 sizeof(int), 0644, NULL, &proc_dointvec},
30403 #ifdef CONFIG_SYSVIPC
30404 {KERN_SHMMAX, "shmmx", &shmmx, sizeof(int),
30405 0644, NULL, &proc_dointvec},
30406 #endif
30407 {0}
30408 };
30409
30410 static ctl_table vm_table[] = {
30411 {VM_FREEPG, "freepages", &freepages,
30412 sizeof(freepages_t), 0644, NULL, &proc_dointvec},
30413 {VM_BDFLUSH, "bdflush", &bdf_prm, 9*sizeof(int), 0600,
30414 NULL, &proc_dointvec_minmax, &sysctl_intvec, NULL,
30415 &bdflush_min, &bdflush_max},
30416 {VM_OVERCOMMIT_MEMORY, "overcommit_memory",
30417 &sysctl_overcommit_memory,
30418 sizeof(sysctl_overcommit_memory), 0644, NULL,
30419 &proc_dointvec},
30420 {VM_BUFFERMEM, "buffermem", &buffer_mem,
30421 sizeof(buffer_mem_t), 0644, NULL, &proc_dointvec},
30422 {VM_PAGECACHE, "pagecache", &page_cache,
30423 sizeof(buffer_mem_t), 0644, NULL, &proc_dointvec},
30424 {VM_PAGERDAEMON, "kswapd", &pager_daemon,
30425 sizeof(pager_daemon_t), 0644, NULL, &proc_dointvec},
30426 {VM_PGT_CACHE, "pagetable_cache", &pgt_cache_water,
30427 2*sizeof(int), 0600, NULL, &proc_dointvec},
30428 {VM_PAGE_CLUSTER, "page-cluster", &page_cluster,
30429 sizeof(int), 0600, NULL, &proc_dointvec},
30430 {0}
30431 };
30432
30433 static ctl_table proc_table[] = { {0} };
30434
30435 static ctl_table fs_table[] = {
30436 {FS_NRINODE, "inode-nr", &inodes_stat, 2*sizeof(int),
30437 0444, NULL, &proc_dointvec},
30438 {FS_STATINODE, "inode-state", &inodes_stat,
30439 7*sizeof(int), 0444, NULL, &proc_dointvec},
30440 {FS_MAXINODE, "inode-max", &max_inodes, sizeof(int),
30441 0644, NULL, &proc_dointvec},
30442 {FS_NRFILE, "file-nr", &nr_files, 3*sizeof(int),
30443 0444, NULL, &proc_dointvec},

```

```

30444 {FS_MAXFILE, "file-max", &max_files, sizeof(int),
30445 0644, NULL, &proc_dointvec},
30446 {FS_NRSUPER, "super-nr", &nr_super_blocks, sizeof(int),
30447 0444, NULL, &proc_dointvec},
30448 {FS_MAXSUPER, "super-max", &max_super_blocks,
30449 sizeof(int), 0644, NULL, &proc_dointvec},
30450 {FS_NRDQUOT, "dquot-nr", &nr_dquots, 2*sizeof(int),
30451 0444, NULL, &proc_dointvec},
30452 {FS_MAXDQUOT, "dquot-max", &max_dquots, sizeof(int),
30453 0644, NULL, &proc_dointvec},
30454 {FS_DENTRY, "dentry-state", &dentry_stat,
30455 6*sizeof(int), 0444, NULL, &proc_dointvec},
30456 {0}
30457 };
30458
30459 static ctl_table debug_table[] = { {0} };
30460
30461 static ctl_table dev_table[] = { {0} };
30462
30463 void __init sysctl_init(void)
30464 {
30465 #ifdef CONFIG_PROC_FS
30466 register_proc_table(root_table, &proc_sys_root);
30467 #endif
30468 }
30469
30470
30471 int do_sysctl (int *name, int nlen,
30472 void *oldval, size_t *oldlenp,
30473 void *newval, size_t newlen)
30474 {
30475 int error;
30476 struct ctl_table_header *tmp;
30477 void *context;
30478
30479 if (nlen == 0 || nlen >= CTL_MAXNAME)
30480 return -ENOTDIR;
30481
30482 if (oldval)
30483 {
30484 int old_len;
30485 if (!oldlenp)
30486 return -EFAULT;
30487 if (get_user(old_len, oldlenp))
30488 return -EFAULT;
30489 }
30490 tmp = &root_table_header;
30491 do {

```

```

30492 context = NULL;
30493 error = parse_table(name, nlen, oldval, oldlenp,
30494                   newval, newlen, tmp->ctl_table, &context);
30495 if (context)
30496     kfree(context);
30497 if (error != -ENOTDIR)
30498     return error;
30499 tmp = tmp->DLIST_NEXT(ctl_entry);
30500 } while (tmp != &root_table_header);
30501 return -ENOTDIR;
30502 }
30503
30504 extern asmlinkage int sys_sysctl(
30505     struct __sysctl_args *args)
30506 {
30507     struct __sysctl_args tmp;
30508     int error;
30509
30510     if (copy_from_user(&tmp, args, sizeof(tmp)))
30511         return -EFAULT;
30512
30513     lock_kernel();
30514     error = do_sysctl(tmp.name, tmp.nlen, tmp.oldval,
30515                    tmp.oldlenp, tmp.newval, tmp.newlen);
30516     unlock_kernel();
30517     return error;
30518 }
30519
30520 /* Like in_group_p, but testing against egid,
30521  * not fsgid */
30522 static int in_egroup_p(gid_t grp)
30523 {
30524     if (grp != current->egid) {
30525         int i = current->ngroups;
30526         if (i) {
30527             gid_t *groups = current->groups;
30528             do {
30529                 if (*groups == grp)
30530                     goto out;
30531                 groups++;
30532                 i--;
30533             } while (i);
30534         }
30535         return 0;
30536     }
30537 out:
30538     return 1;
30539 }
30540
30541 /* ctl_perm does NOT grant the superuser all rights
30542  * automatically, because some sysctl variables are
30543  * readonly even to root. */
30544 static int test_perm(int mode, int op)
30545 {
30546     if (!current->euid)
30547         mode >>= 6;
30548     else if (in_egroup_p(0))
30549         mode >>= 3;
30550     if ((mode & op & 0007) == op)
30551         return 0;
30552     return -EACCES;
30553 }
30554
30555 static inline int ctl_perm(ctl_table *table, int op)
30556 {
30557     return test_perm(table->mode, op);
30558 }
30559
30560 static int parse_table(int *name, int nlen,
30561                       void *oldval, size_t *oldlenp,
30562                       void *newval, size_t newlen,
30563                       ctl_table *table, void **context)
30564 {
30565     int error;
30566     repeat:
30567     if (!nlen)
30568         return -ENOTDIR;
30569
30570     for ( ; table->ctl_name; table++) {
30571         int n;
30572         if (get_user(n, name))
30573             return -EFAULT;
30574         if (n == table->ctl_name ||
30575             table->ctl_name == CTL_ANY) {
30576             if (table->child) {
30577                 if (ctl_perm(table, 001))
30578                     return -EPERM;
30579                 if (table->strategy) {
30580                     error = table->strategy(
30581                         table, name, nlen,
30582                         oldval, oldlenp,
30583                         newval, newlen, context);
30584                     if (error)
30585                         return error;
30586                 }
30587                 name++;

```

```

30588     nlen--;
30589     table = table->child;
30590     goto repeat;
30591     }
30592     error = do_sysctl_strategy(table, name, nlen,
30593     oldval, oldlenp,
30594     newval, newlen, context);
30595     return error;
30596     }
30597     };
30598     return -ENOTDIR;
30599     }
30600
30601     /* Perform the actual read/write of a sysctl table
30602     * entry. */
30603     int do_sysctl_strategy(ctl_table *table,
30604     int *name, int nlen,
30605     void *oldval, size_t *oldlenp,
30606     void *newval, size_t newlen, void **context)
30607     {
30608     int op = 0, rc, len;
30609
30610     if (oldval)
30611         op |= 004;
30612     if (newval)
30613         op |= 002;
30614     if (ctl_perm(table, op))
30615         return -EPERM;
30616
30617     if (table->strategy) {
30618         rc = table->strategy(table, name, nlen, oldval,
30619         oldlenp, newval, newlen, context);
30620         if (rc < 0)
30621             return rc;
30622         if (rc > 0)
30623             return 0;
30624     }
30625
30626     /* If there is no strategy routine, or if the strategy
30627     * returns zero, proceed with automatic r/w */
30628     if (table->data && table->maxlen) {
30629         if (oldval && oldlenp) {
30630             get_user(len, oldlenp);
30631             if (len) {
30632                 if (len > table->maxlen)
30633                     len = table->maxlen;
30634                 if(copy_to_user(oldval, table->data, len))
30635                     return -EFAULT;

```

```

30636             if(put_user(len, oldlenp))
30637                 return -EFAULT;
30638         }
30639     }
30640     if (newval && newlen) {
30641         len = newlen;
30642         if (len > table->maxlen)
30643             len = table->maxlen;
30644         if(copy_from_user(table->data, newval, len))
30645             return -EFAULT;
30646     }
30647     }
30648     return 0;
30649     }
30650
30651     struct ctl_table_header *register_sysctl_table(
30652     ctl_table * table, int insert_at_head)
30653     {
30654     struct ctl_table_header *tmp;
30655     tmp = kmalloc(sizeof(*tmp), GFP_KERNEL);
30656     if (!tmp)
30657         return 0;
30658     *tmp = ((struct ctl_table_header) {table, DNODE_NULL});
30659     if (insert_at_head)
30660         DLIST_INSERT_AFTER(&root_table_header, tmp,
30661         ctl_entry);
30662     else
30663         DLIST_INSERT_BEFORE(&root_table_header, tmp,
30664         ctl_entry);
30665     #ifdef CONFIG_PROC_FS
30666     register_proc_table(table, &proc_sys_root);
30667     #endif
30668     return tmp;
30669     }
30670
30671     /* Unlink and free a ctl_table. */
30672     void unregister_sysctl_table(
30673     struct ctl_table_header * header)
30674     {
30675     DLIST_DELETE(header, ctl_entry);
30676     #ifdef CONFIG_PROC_FS
30677     unregister_proc_table(header->ctl_table,
30678     &proc_sys_root);
30679     #endif
30680     kfree(header);
30681     }
30682
30683     /* /proc/sys support */

```

```

30684
30685 #ifdef CONFIG_PROC_FS
30686
30687 /* Scan the sysctl entries in table and add them all into
30688  * /proc */
p 616 30689 static void register_proc_table(ctl_table * table,
30690                                struct proc_dir_entry *root)
30691 {
30692     struct proc_dir_entry *de;
30693     int len;
30694     mode_t mode;
30695
30696     for (; table->ctl_name; table++) {
30697         /* Can't do anything without a proc name. */
30698         if (!table->procname)
30699             continue;
30700         /* Maybe we can't do anything with it... */
30701         if (!table->proc_handler && !table->child) {
30702             printk(KERN_WARNING "SYSCTL: Can't register %s\n",
30703                    table->procname);
30704             continue;
30705         }
30706
30707         len = strlen(table->procname);
30708         mode = table->mode;
30709
30710         de = NULL;
30711         if (table->proc_handler)
30712             mode |= S_IFREG;
30713         else {
30714             mode |= S_IFDIR;
30715             for (de = root->subdir; de; de = de->next) {
30716                 if (proc_match(len, table->procname, de))
30717                     break;
30718             }
30719             /* If the subdir exists already, de is non-NULL */
30720         }
30721
30722         if (!de) {
30723             de = create_proc_entry(table->procname, mode, root);
30724             if (!de)
30725                 continue;
30726             de->data = (void *) table;
30727             if (table->proc_handler)
30728                 de->ops = &proc_sys_inode_operations;
30729         }
30730     }
30731     table->de = de;

```

```

30732     if (de->mode & S_IFDIR)
30733         register_proc_table(table->child, de);
30734 }
30735 }
30736
30737 /* Unregister a /proc sysctl table and any
30738  * subdirectories. */
p 617 30739 static void unregister_proc_table(ctl_table * table,
30740                                   struct proc_dir_entry *root)
30741 {
30742     struct proc_dir_entry *de;
30743     for (; table->ctl_name; table++) {
30744         if (!(de = table->de))
30745             continue;
30746         if (de->mode & S_IFDIR) {
30747             if (!table->child) {
30748                 printk(KERN_ALERT
30749                        "Help - malformed sysctl tree on free\n");
30750                 continue;
30751             }
30752             unregister_proc_table(table->child, de);
30753
30754             /* Don't unregister directories which still have
30755              * entries.. */
30756             if (de->subdir)
30757                 continue;
30758         }
30759
30760         /* Don't unregister proc entries that are still being
30761          * used.. */
30762         if (de->count)
30763             continue;
30764
30765         proc_unregister(root, de->low_ino);
30766         table->de = NULL;
30767         kfree(de);
30768     }
30769 }
30770
p 618 30771 static ssize_t do_rw_proc(int write, struct file * file,
30772                          char * buf, size_t count, loff_t *ppos)
30773 {
30774     int op;
30775     struct proc_dir_entry *de;
30776     struct ctl_table *table;
30777     size_t res;
30778     ssize_t error;
30779

```



```

30780 de = (struct proc_dir_entry *)
30781     file->f_dentry->d_inode->u.generic_ip;
30782 if (!de || !de->data)
30783     return -ENOTDIR;
30784 table = (struct ctl_table *) de->data;
30785 if (!table || !table->proc_handler)
30786     return -ENOTDIR;
30787 op = (write ? 002 : 004);
30788 if (ctl_perm(table, op))
30789     return -EPERM;
30790
30791 res = count;
30792
30793 /* FIXME: we need to pass on ppos to the handler. */
30794
30795 error =
30796     (*table->proc_handler)(table, write, file, buf,&res);
30797 if (error)
30798     return error;
30799 return res;
30800 }
30801
30802 static ssize_t proc_readsys(struct file * file,
30803     char * buf, size_t count, loff_t *ppos)
30804 {
30805     return do_rw_proc(0, file, buf, count, ppos);
30806 }
30807
30808 static ssize_t proc_writesys(struct file * file,
30809     const char * buf, size_t count, loff_t *ppos)
30810 {
30811     return do_rw_proc(1, file, (char *) buf, count, ppos);
30812 }
30813
30814 static int proc_sys_permission(struct inode *inode,
30815     int op)
30816 {
30817     return test_perm(inode->i_mode, op);
30818 }
30819
30820 int proc_dostring(ctl_table *table, int write,
30821     struct file *filp, void *buffer, size_t *lenp)
30822 {
30823     int len;
30824     char *p, c;
30825
30826     if (!table->data || !table->maxlen || !*lenp ||
30827         (filp->f_pos && !write)) {

```

▶ p 618

```

30828     *lenp = 0;
30829     return 0;
30830 }
30831
30832 if (write) {
30833     len = 0;
30834     p = buffer;
30835     while (len < *lenp) {
30836         if(get_user(c, p++))
30837             return -EFAULT;
30838         if (c == 0 || c == '\n')
30839             break;
30840         len++;
30841     }
30842     if (len >= table->maxlen)
30843         len = table->maxlen-1;
30844     if(copy_from_user(table->data, buffer, len))
30845         return -EFAULT;
30846     ((char *) table->data)[len] = 0;
30847     filp->f_pos += *lenp;
30848 } else {
30849     len = strlen(table->data);
30850     if (len > table->maxlen)
30851         len = table->maxlen;
30852     if (len > *lenp)
30853         len = *lenp;
30854     if (len)
30855         if(copy_to_user(buffer, table->data, len))
30856             return -EFAULT;
30857     if (len < *lenp) {
30858         if(put_user('\n', ((char *) buffer) + len))
30859             return -EFAULT;
30860         len++;
30861     }
30862     *lenp = len;
30863     filp->f_pos += len;
30864 }
30865 return 0;
30866 }
30867
30868 /* Special case of dostring for the UTS structure. This
30869 * has locks to observe. Should this be in kernel/sys.c
30870 * ??? */
30871 static int proc_doutsstring(ctl_table *table, int write,
30872     struct file *filp, void *buffer, size_t *lenp)
30873 {
30874     int r;
30875     down(&uts.sem);

```

▶ p 618

```

30876 r=proc_dostring(table,write,filp,buffer,lenp);
30877 up(&uts_sem);
30878 return r;
30879 }
30880
30881 static int do_proc_dointvec(ctl_table *table, int write,
30882 struct file *filp, void *buffer, size_t *lenp, int conv)
30883 {
30884     int *i, vleft, first=1, len, left, neg, val;
30885     #define TMPBUFLEN 20
30886     char buf[TMPBUFLEN], *p;
30887
30888     if (!table->data || !table->maxlen || !*lenp ||
30889         (filp->f_pos && !write)) {
30890         *lenp = 0;
30891         return 0;
30892     }
30893
30894     i = (int *) table->data;
30895     vleft = table->maxlen / sizeof(int);
30896     left = *lenp;
30897
30898     for (; left && vleft--; i++, first=0) {
30899         if (write) {
30900             while (left) {
30901                 char c;
30902                 if(get_user(c,(char *) buffer))
30903                     return -EFAULT;
30904                 if (!isspace(c))
30905                     break;
30906                 left--;
30907                 ((char *) buffer)++;
30908             }
30909             if (!left)
30910                 break;
30911             neg = 0;
30912             len = left;
30913             if (len > TMPBUFLEN-1)
30914                 len = TMPBUFLEN-1;
30915             if(copy_from_user(buf, buffer, len))
30916                 return -EFAULT;
30917             buf[len] = 0;
30918             p = buf;
30919             if (*p == '-' && left > 1) {
30920                 neg = 1;
30921                 left--, p++;
30922             }
30923             if (*p < '0' || *p > '9')

```

```

30924                 break;
30925                 val = simple_strtoul(p, &p, 0) * conv;
30926                 len = p-buf;
30927                 if ((len < left) && *p && !isspace(*p))
30928                     break;
30929                 if (neg)
30930                     val = -val;
30931                 buffer += len;
30932                 left -= len;
30933                 *i = val;
30934             } else {
30935                 p = buf;
30936                 if (!first)
30937                     *p++ = '\t';
30938                 sprintf(p, "%d", (*i) / conv);
30939                 len = strlen(buf);
30940                 if (len > left)
30941                     len = left;
30942                 if(copy_to_user(buffer, buf, len))
30943                     return -EFAULT;
30944                 left -= len;
30945                 buffer += len;
30946             }
30947         }
30948
30949         if (!write && !first && left) {
30950             if(put_user('\n', (char *) buffer))
30951                 return -EFAULT;
30952             left--, buffer++;
30953         }
30954         if (write) {
30955             p = (char *) buffer;
30956             while (left) {
30957                 char c;
30958                 if(get_user(c, p++))
30959                     return -EFAULT;
30960                 if (!isspace(c))
30961                     break;
30962                 left--;
30963             }
30964         }
30965         if (write && first)
30966             return -EINVAL;
30967         *lenp -= left;
30968         filp->f_pos += *lenp;
30969         return 0;
30970     }
30971

```

```

30972 int proc_dointvec(ctl_table *table, int write,
30973 struct file *filp, void *buf, size_t *lenp)
30974 {
30975     return do_proc_dointvec(table,write,filp,buf,lenp,1);
30976 }
30977
30978 int proc_dointvec_minmax(ctl_table *table, int write,
30979 struct file *filp, void *buffer, size_t *lenp)
30980 {
30981     int *i, *min, *max, vleft, first=1, len, left, neg,val;
30982     #define TMPBUFLLEN 20
30983     char buf[TMPBUFLLEN], *p;
30984
30985     if (!table->data || !table->maxlen || !*lenp ||
30986         (filp->f_pos && !write)) {
30987         *lenp = 0;
30988         return 0;
30989     }
30990
30991     i = (int *) table->data;
30992     min = (int *) table->extral;
30993     max = (int *) table->extra2;
30994     vleft = table->maxlen / sizeof(int);
30995     left = *lenp;
30996
30997     for (; left && vleft--; i++, first=0) {
30998         if (write) {
30999             while (left) {
31000                 char c;
31001                 if(get_user(c, (char *) buffer))
31002                     return -EFAULT;
31003                 if (!isspace(c))
31004                     break;
31005                 left--;
31006                 ((char *) buffer)++;
31007             }
31008             if (!left)
31009                 break;
31010             neg = 0;
31011             len = left;
31012             if (len > TMPBUFLLEN-1)
31013                 len = TMPBUFLLEN-1;
31014             if(copy_from_user(buf, buffer, len))
31015                 return -EFAULT;
31016             buf[len] = 0;
31017             p = buf;
31018             if (*p == '-' && left > 1) {
31019                 neg = 1;

```

```

31020         left--, p++;
31021     }
31022     if (*p < '0' || *p > '9')
31023         break;
31024     val = simple_strtoul(p, &p, 0);
31025     len = p-buf;
31026     if ((len < left) && *p && !isspace(*p))
31027         break;
31028     if (neg)
31029         val = -val;
31030     buffer += len;
31031     left -= len;
31032
31033     if (min && val < *min++)
31034         continue;
31035     if (max && val > *max++)
31036         continue;
31037     *i = val;
31038 } else {
31039     p = buf;
31040     if (!first)
31041         *p++ = '\t';
31042     sprintf(p, "%d", *i);
31043     len = strlen(buf);
31044     if (len > left)
31045         len = left;
31046     if(copy_to_user(buffer, buf, len))
31047         return -EFAULT;
31048     left -= len;
31049     buffer += len;
31050 }
31051 }
31052
31053 if (!write && !first && left) {
31054     if(put_user('\n', (char *) buffer))
31055         return -EFAULT;
31056     left--, buffer++;
31057 }
31058 if (write) {
31059     p = (char *) buffer;
31060     while (left) {
31061         char c;
31062         if(get_user(c, p++))
31063             return -EFAULT;
31064         if (!isspace(c))
31065             break;
31066         left--;
31067     }

```

```

31068 }
31069 if (write && first)
31070     return -EINVAL;
31071 *lenp -- left;
31072 filp->f_pos += *lenp;
31073 return 0;
31074 }
31075
31076 /* Like proc_dointvec, but converts seconds to jiffies */
31077 int proc_dointvec_jiffies(ctl_table *tbl, int write,
31078     struct file *filp, void *buf, size_t *lenp)
31079 {
31080     return do_proc_dointvec(tbl, write, filp, buf, lenp, HZ);
31081 }
31082
31083 #else /* CONFIG_PROC_FS */
31084
31085 int proc_dostring(ctl_table *table, int write,
31086     struct file *filp, void *buffer, size_t *lenp)
31087 {
31088     return -ENOSYS;
31089 }
31090
31091 static int proc_doutsstring(ctl_table *table, int write,
31092     struct file *filp, void *buffer, size_t *lenp)
31093 {
31094     return -ENOSYS;
31095 }
31096
31097 int proc_dointvec(ctl_table *table, int write,
31098     struct file *filp, void *buffer, size_t *lenp)
31099 {
31100     return -ENOSYS;
31101 }
31102
31103 int proc_dointvec_minmax(ctl_table *table, int write,
31104     struct file *filp, void *buffer, size_t *lenp)
31105 {
31106     return -ENOSYS;
31107 }
31108
31109 int proc_dointvec_jiffies(ctl_table *table, int write,
31110     struct file *filp, void *buffer, size_t *lenp)
31111 {
31112     return -ENOSYS;
31113 }
31114
31115 #endif /* CONFIG_PROC_FS */

```

```

31116
31117
31118 /* General sysctl support routines */
31119
31120 /* The generic string strategy routine: */
31121 int sysctl_string(ctl_table *table, int *name, int nlen,
31122     void *oldval, size_t *oldlenp,
31123     void *newval, size_t newlen, void **context)
31124 {
31125     int l, len;
31126
31127     if (!table->data || !table->maxlen)
31128         return -ENOTDIR;
31129
31130     if (oldval && oldlenp) {
31131         if (get_user(len, oldlenp))
31132             return -EFAULT;
31133         if (len) {
31134             l = strlen(table->data);
31135             if (len > l) len = l;
31136             if (len >= table->maxlen)
31137                 len = table->maxlen;
31138             if (copy_to_user(oldval, table->data, len))
31139                 return -EFAULT;
31140             if (put_user(0, ((char *) oldval) + len))
31141                 return -EFAULT;
31142             if (put_user(len, oldlenp))
31143                 return -EFAULT;
31144         }
31145     }
31146     if (newval && newlen) {
31147         len = newlen;
31148         if (len > table->maxlen)
31149             len = table->maxlen;
31150         if (copy_from_user(table->data, newval, len))
31151             return -EFAULT;
31152         if (len == table->maxlen)
31153             len--;
31154         ((char *) table->data)[len] = 0;
31155     }
31156     return 0;
31157 }
31158
31159 /* This function makes sure that all of the integers in
31160 * the vector are between the minimum and maximum values
31161 * given in the arrays table->extra1 and table->extra2,
31162 * respectively. */
31163 int sysctl_intvec(ctl_table *table, int *name, int nlen,

```

```

31164 void *oldval, size_t *oldlenp,
31165 void *newval, size_t newlen, void **context)
31166 {
31167     int i, length, *vec, *min, *max;
31168
31169     if (newval && newlen) {
31170         if (newlen % sizeof(int) != 0)
31171             return -EINVAL;
31172
31173         if (!table->extral && !table->extra2)
31174             return 0;
31175
31176         if (newlen > table->maxlen)
31177             newlen = table->maxlen;
31178         length = newlen / sizeof(int);
31179
31180         vec = (int *) newval;
31181         min = (int *) table->extral;
31182         max = (int *) table->extra2;
31183
31184         for (i = 0; i < length; i++) {
31185             int value;
31186             get_user(value, vec + i);
31187             if (min && value < min[i])
31188                 return -EINVAL;
31189             if (max && value > max[i])
31190                 return -EINVAL;
31191         }
31192     }
31193     return 0;
31194 }
31195
31196 int do_string (void *oldval, size_t *oldlenp,
31197               void *newval, size_t newlen,
31198               int rdwr, char *data, size_t max)
31199 {
31200     int l = strlen(data) + 1;
31201     if (newval && !rdwr)
31202         return -EPERM;
31203     if (newval && newlen >= max)
31204         return -EINVAL;
31205     if (oldval) {
31206         int old_l;
31207         if (get_user(old_l, oldlenp))
31208             return -EFAULT;
31209         if (l > old_l)
31210             return -ENOMEM;
31211         if (put_user(l, oldlenp) ||

```

```

31212         copy_to_user(oldval, data, l))
31213         return -EFAULT;
31214     }
31215     if (newval) {
31216         if (copy_from_user(data, newval, newlen))
31217             return -EFAULT;
31218         data[newlen] = 0;
31219     }
31220     return 0;
31221 }
31222
31223 int do_int (void *oldval, size_t *oldlenp,
31224            void *newval, size_t newlen,
31225            int rdwr, int *data)
31226 {
31227     if (newval && !rdwr)
31228         return -EPERM;
31229     if (newval && newlen != sizeof(int))
31230         return -EINVAL;
31231     if (oldval) {
31232         int old_l;
31233         if (get_user(old_l, oldlenp))
31234             return -EFAULT;
31235         if (old_l < sizeof(int))
31236             return -ENOMEM;
31237         if (put_user(sizeof(int), oldlenp) ||
31238             copy_to_user(oldval, data, sizeof(int)))
31239             return -EFAULT;
31240     }
31241     if (newval)
31242         if (copy_from_user(data, newval, sizeof(int)))
31243             return -EFAULT;
31244     return 0;
31245 }
31246
31247 int do_struct (void *oldval, size_t *oldlenp,
31248               void *newval, size_t newlen,
31249               int rdwr, void *data, size_t len)
31250 {
31251     if (newval && !rdwr)
31252         return -EPERM;
31253     if (newval && newlen != len)
31254         return -EINVAL;
31255     if (oldval) {
31256         int old_l;
31257         if (get_user(old_l, oldlenp))
31258             return -EFAULT;
31259         if (old_l < len)

```

```

31260     return -ENOMEM;
31261     if (put_user(len, oldlenp) ||
31262         copy_to_user(oldval, data, len))
31263         return -EFAULT;
31264 }
31265 if (newval)
31266     if (copy_from_user(data, newval, len))
31267         return -EFAULT;
31268 return 0;
31269 }
31270
31271
31272 #else /* CONFIG_SYSCTL */
31273
31274
31275 extern asmlinkage int sys_sysctl(
31276     struct __sysctl_args *args)
31277 {
31278     return -ENOSYS;
31279 }
31280
31281 int sysctl_string(ctl_table *table, int *name, int nlen,
31282     void *oldval, size_t *oldlenp,
31283     void *newval, size_t newlen, void **context)
31284 {
31285     return -ENOSYS;
31286 }
31287
31288 int sysctl_intvec(ctl_table *table, int *name, int nlen,
31289     void *oldval, size_t *oldlenp,
31290     void *newval, size_t newlen, void **context)
31291 {
31292     return -ENOSYS;
31293 }
31294
31295 int proc_dostring(ctl_table *table, int write,
31296     struct file *filp, void *buffer, size_t *lenp)
31297 {
31298     return -ENOSYS;
31299 }
31300
31301 int proc_dointvec(ctl_table *table, int write,
31302     struct file *filp, void *buffer, size_t *lenp)
31303 {
31304     return -ENOSYS;
31305 }
31306
31307 int proc_dointvec_minmax(ctl_table *table, int write,

```

```

31308     struct file *filp, void *buffer, size_t *lenp)
31309 {
31310     return -ENOSYS;
31311 }
31312
31313 int proc_dointvec_jiffies(ctl_table *table, int write,
31314     struct file *filp, void *buffer, size_t *lenp)
31315 {
31316     return -ENOSYS;
31317 }
31318
31319 struct ctl_table_header *
31320 register_sysctl_table(ctl_table *tbl, int insert_at_head)
31321 {
31322     return 0;
31323 }
31324
31325 void unregister_sysctl_table(
31326     struct ctl_table_header * table)
31327 {
31328 }
31329
31330 #endif /* CONFIG_SYSCTL */

```

kernel/time.c

```

31331 /*
31332  * linux/kernel/time.c
31333  *
31334  * Copyright (C) 1991, 1992 Linus Torvalds
31335  *
31336  * This file contains the interface functions for the
31337  * various time related system calls: time, stime,
31338  * gettimeofday, settimeofday, adjtime */
31339 /*
31340  * Modification history kernel/time.c
31341  *
31342  * 1993-09-02 Philip Gladstone
31343  * Created file with time related functions from
31344  * sched.c and adjtimex()
31345  * 1993-10-08 Torsten Duwe
31346  * adjtime interface update and CMOS clock write
31347  * code
31348  * 1995-08-13 Torsten Duwe
31349  * kernel PLL updated to 1994-12-13 specs (rfc-1589)
31350  * 1999-01-16 Ulrich Windl
31351  * Introduced error checking for many cases in
31352  * adjtimex(). Updated NTP code according to

```

```

31353 *      technical memorandum Jan '96 "A Kernel Model for
31354 *      Precision Timekeeping" by Dave Mills Allow
31355 *      time_constant larger than MAXTC(6) for NTP v4
31356 *      (MAXTC == 10) (Even though the technical
31357 *      memorandum forbids it) */
31358
31359 #include <linux/mm.h>
31360 #include <linux/timex.h>
31361 #include <linux/smp_lock.h>
31362
31363 #include <asm/uaccess.h>
31364
31365 /* The timezone where the local system is located. Used
31366 * as a default by some programs who obtain this value by
31367 * using gettimeofday. */
31368 struct timezone sys_tz = { 0, 0};
31369
31370 static void do_normal_gettime(struct timeval * tm)
31371 {
31372     *tm=xtime;
31373 }
31374
31375 void (*do_get_fast_time)(struct timeval *) =
31376     do_normal_gettime;
31377
31378 /* Generic way to access 'xtime' (the current time of
31379 * day). This can be changed if the platform provides a
31380 * more accurate (and fast!) version. */
31381
31382 void get_fast_time(struct timeval * t)
31383 {
31384     do_get_fast_time(t);
31385 }
31386
31387 #ifndef __alpha__
31388
31389 /* sys_time() can be implemented in user-level using
31390 * sys_gettimeofday(). Is this for backwards
31391 * compatibility? If so, why not move it into the
31392 * appropriate arch directory (for those architectures
31393 * that need it). */
31394 asmlinkage int sys_time(int * tloc)
31395 {
31396     int i;
31397
31398     /* SMP: This is fairly trivial. We grab CURRENT_TIME
31399     * and stuff it to user space. No side effects */
31400     i = CURRENT_TIME;

```

```

31401     if (tloc) {
31402         if (put_user(i,tloc))
31403             i = -EFAULT;
31404     }
31405     return i;
31406 }
31407
31408 /* sys_stime() can be implemented in user-level using
31409 * sys_settimeofday(). Is this for backwards
31410 * compatibility? If so, why not move it into the
31411 * appropriate arch directory (for those architectures
31412 * that need it). */
31413 asmlinkage int sys_stime(int * tptr)
31414 {
31415     int value;
31416
31417     if (!capable(CAP_SYS_TIME))
31418         return -EPERM;
31419     if (get_user(value, tptr))
31420         return -EFAULT;
31421     cli();
31422     xtime.tv_sec = value;
31423     xtime.tv_usec = 0;
31424     time_adjust = 0;        /* stop active adjtime() */
31425     time_status |= STA_UNSYNC;
31426     time_maxerror = NTP_PHASE_LIMIT;
31427     time_esterror = NTP_PHASE_LIMIT;
31428     sti();
31429     return 0;
31430 }
31431
31432 #endif
31433
31434 asmlinkage int sys_gettimeofday(struct timeval *tv,
31435                                 struct timezone *tz)
31436 {
31437     if (tv) {
31438         struct timeval ktv;
31439         do_gettimeofday(&ktv);
31440         if (copy_to_user(tv, &ktv, sizeof(ktv)))
31441             return -EFAULT;
31442     }
31443     if (tz) {
31444         if (copy_to_user(tz, &sys_tz, sizeof(sys_tz)))
31445             return -EFAULT;
31446     }
31447     return 0;
31448 }

```

```

31449
31450 /* Adjust the time obtained from the CMOS to be UTC time
31451 * instead of local time.
31452 *
31453 * This is ugly, but preferable to the alternatives.
31454 * Otherwise we would either need to write a program to
31455 * do it in /etc/rc (and risk confusion if the program
31456 * gets run more than once; it would also be hard to make
31457 * the program warp the clock precisely n hours) or
31458 * compile in the timezone information into the kernel.
31459 * Bad, bad....
31460 *
31461 * - TYT, 1992-01-01
31462 *
31463 * The best thing to do is to keep the CMOS clock in
31464 * universal time (UTC) as real UNIX machines always do
31465 * it. This avoids all headaches about daylight saving
31466 * times and warping kernel clocks. */
31467 inline static void warp_clock(void)
31468 {
31469     cli();
31470     xtime.tv_sec += sys_tz.tz_minuteswest * 60;
31471     sti();
31472 }
31473
31474 /* In case for some reason the CMOS clock has not already
31475 * been running in UTC, but in some local time: The first
31476 * time we set the timezone, we will warp the clock so
31477 * that it is ticking UTC time instead of local
31478 * time. Presumably, if someone is setting the timezone
31479 * then we are running in an environment where the
31480 * programs understand about timezones. This should be
31481 * done at boot time in the /etc/rc script, as soon as
31482 * possible, so that the clock can be set
31483 * right. Otherwise, various programs will get confused
31484 * when the clock gets warped. */
31485 int do_sys_settimeofday(struct timeval *tv,
31486                        struct timezone *tz)
31487 {
31488     static int firsttime = 1;
31489
31490     if (!capable(CAP_SYS_TIME))
31491         return -EPERM;
31492
31493     if (tz) {
31494         /* SMP safe, global irq locking makes it work. */
31495         sys_tz = *tz;
31496         if (firsttime) {
31497             firsttime = 0;
31498             if (!tv)
31499                 warp_clock();
31500         }
31501     }
31502     if (tv)
31503     {
31504         /* SMP safe, again the code in arch/foo/time.c should
31505          * globally block out interrupts when it runs.
31506          */
31507         do_settimeofday(tv);
31508     }
31509     return 0;
31510 }
31511
31512 asmlinkage int sys_settimeofday(struct timeval *tv,
31513                                struct timezone *tz)
31514 {
31515     struct timeval new_tv;
31516     struct timezone new_tz;
31517
31518     if (tv) {
31519         if (copy_from_user(&new_tv, tv, sizeof(*tv)))
31520             return -EFAULT;
31521     }
31522     if (tz) {
31523         if (copy_from_user(&new_tz, tz, sizeof(*tz)))
31524             return -EFAULT;
31525     }
31526
31527     return do_sys_settimeofday(tv ? &new_tv : NULL,
31528                               tz ? &new_tz : NULL);
31529 }
31530
31531 long pps_offset = 0; /* pps time offset (us) */
31532 long pps_jitter = MAXTIME; /* time dispersion (jitter)
31533 (us) */
31534 long pps_freq = 0; /* frequency offset (scaled
31535 ppm) */
31536 long pps_stabil = MAXFREQ; /* frequency dispersion
31537 (scaled ppm) */
31538 long pps_valid = PPS_VALID; /* pps signal watchdog
31539 counter */
31540 int pps_shift = PPS_SHIFT; /* interval duration (s)
31541 (shift) */
31542 long pps_jitcnt = 0; /* jitter limit exceeded */
31543 long pps_calcnt = 0; /* calibration intervals */
31544 long pps_errcnt = 0; /* calibration errors */

```



```

31545 long pps_stbcnt = 0;      /* stability limit exceeded*/
31546
31547 /* hook for a loadable hardpps kernel module */
31548 void (*hardpps_ptr)(struct timeval *) =
31549     (void (*)(struct timeval *)0);
31550
31551 /* adjtimex mainly allows reading (and writing, if
31552  * superuser) of kernel time-keeping variables. used by
31553  * xntpd. */
31554 int do_adjtimex(struct timex *txc)
31555 {
31556     long ltemp, mtemp, save_adjust;
31557     int result = time_state;      /* mostly 'TIME_OK' */
31558
31559     /* In order to modify anything, you gotta be
31560      * super-user! */
31561     if (txc->modes && !capable(CAP_SYS_TIME))
31562         return -EPERM;
31563
31564     /* Now we validate the data before disabling interrupts
31565      */
31566     if (txc->modes != ADJ_OFFSET_SINGLESHOT &&
31567         (txc->modes & ADJ_OFFSET))
31568         /* adjustment Offset limited to +- .512 seconds */
31569         if (txc->offset <= - MAXPHASE ||
31570             txc->offset >= MAXPHASE)
31571             return -EINVAL;
31572
31573     /* if the quartz is off by more than 10% something is
31574      * VERY wrong ! */
31575     if (txc->modes & ADJ_TICK)
31576         if (txc->tick < 900000/HZ || txc->tick > 1100000/HZ)
31577             return -EINVAL;
31578
31579     cli(); /* SMP: global cli() is enough protection. */
31580
31581     /* Save for later - semantics of adjtime is to return
31582      * old value */
31583     save_adjust = time_adjust;
31584
31585     #if 0 /* STA_CLOCKERR is never set yet */
31586     time_status &= ~STA_CLOCKERR; /* reset STA_CLOCKERR */
31587     #endif
31588     /* If there are input parameters, then process them */
31589     if (txc->modes)
31590     {
31591         if (txc->modes & ADJ_STATUS)
31592             /* only set allowed bits */

```

```

31593     time_status = (txc->status & ~STA_RONLY) |
31594         (time_status & STA_RONLY);
31595
31596     if (txc->modes & ADJ_FREQUENCY) { /* p. 22 */
31597         if (txc->freq > MAXFREQ || txc->freq < -MAXFREQ) {
31598             result = -EINVAL;
31599             goto leave;
31600         }
31601         time_freq = txc->freq - pps_freq;
31602     }
31603
31604     if (txc->modes & ADJ_MAXERROR) {
31605         if (txc->maxerror < 0 ||
31606             txc->maxerror >= NTP_PHASE_LIMIT) {
31607             result = -EINVAL;
31608             goto leave;
31609         }
31610         time_maxerror = txc->maxerror;
31611     }
31612
31613     if (txc->modes & ADJ_ESTERROR) {
31614         if (txc->esterror < 0 ||
31615             txc->esterror >= NTP_PHASE_LIMIT) {
31616             result = -EINVAL;
31617             goto leave;
31618         }
31619         time_esterror = txc->esterror;
31620     }
31621
31622     if (txc->modes & ADJ_TIMECONST) { /* p. 24 */
31623         if (txc->constant < 0) { /*NTP v4 uses values > 6*/
31624             result = -EINVAL;
31625             goto leave;
31626         }
31627         time_constant = txc->constant;
31628     }
31629
31630     /* values checked earlier */
31631     if (txc->modes & ADJ_OFFSET) {
31632         if (txc->modes == ADJ_OFFSET_SINGLESHOT) {
31633             /* adjtime() is independent from ntp_adjtime() */
31634             time_adjust = txc->offset;
31635         }
31636         else if ( (time_status & (STA_PLL | STA_PPSTIME)) |
31637             (time_status & (STA_PPSTIME | STA_PPSSIGNAL))
31638             == (STA_PPSTIME | STA_PPSSIGNAL)
31639             ? pps_offset : txc->offset;
31640

```

```

31641
31642 /* Scale the phase adjustment and clamp to the
31643 * operating range. */
31644 if (ltemp > MAXPHASE)
31645     time_offset = MAXPHASE << SHIFT_UPDATE;
31646 else if (ltemp < -MAXPHASE)
31647     time_offset = -(MAXPHASE << SHIFT_UPDATE);
31648 else
31649     time_offset = ltemp << SHIFT_UPDATE;
31650
31651 /* Select whether the frequency is to be
31652 * controlled and in which mode (PLL or
31653 * FLL). Clamp to the operating range. Ugly
31654 * multiply/divide should be replaced someday.
31655 */
31656
31657 if (time_status & STA_FREQHOLD ||
31658     time_reftime == 0)
31659     time_reftime = xtime.tv_sec;
31660 mtemp = xtime.tv_sec - time_reftime;
31661 time_reftime = xtime.tv_sec;
31662 if (time_status & STA_FLL) {
31663     if (mtemp >= MINSEC) {
31664         ltemp = (time_offset / mtemp) <<
31665             (SHIFT_USEC - SHIFT_UPDATE);
31666         if (ltemp < 0)
31667             time_freq -= -ltemp >> SHIFT_KH;
31668         else
31669             time_freq += ltemp >> SHIFT_KH;
31670     } else
31671         /* calibration interval too short (p. 12) */
31672         result = TIME_ERROR;
31673 } else { /* PLL mode */
31674     if (mtemp < MAXSEC) {
31675         ltemp *= mtemp;
31676         if (ltemp < 0)
31677             time_freq -= -ltemp >>
31678                 (time_constant +
31679                  time_constant +
31680                  SHIFT_KF - SHIFT_USEC);
31681         else
31682             time_freq += ltemp >>
31683                 (time_constant +
31684                  time_constant +
31685                  SHIFT_KF - SHIFT_USEC);
31686     } else
31687         /* calibration interval too long (p. 12) */
31688         result = TIME_ERROR;

```

```

31689     }
31690     if (time_freq > time_tolerance)
31691         time_freq = time_tolerance;
31692     else if (time_freq < -time_tolerance)
31693         time_freq = -time_tolerance;
31694 } /* STA_PLL || STA_PPSTIME */
31695 } /* txc->modes & ADJ_OFFSET */
31696 if (txc->modes & ADJ_TICK) {
31697     /* if the quartz is off by more than 10% something
31698     is VERY wrong ! */
31699     if (txc->tick < 900000/HZ ||
31700         txc->tick > 1100000/HZ) {
31701         result = -EINVAL;
31702         goto leave;
31703     }
31704     tick = txc->tick;
31705 }
31706 } /* txc->modes */
31707 leave:
31708 if ((time_status & (STA_UNSYNC|STA_CLOCKERR)) != 0
31709     || ((time_status & (STA_PPSFREQ|STA_PPSTIME)) != 0
31710         && (time_status & STA_PPSSIGNAL) == 0)
31711     /* p. 24, (b) */
31712     || ((time_status & (STA_PPSTIME|STA_PPSJITTER))
31713         == (STA_PPSTIME|STA_PPSJITTER))
31714     /* p. 24, (c) */
31715     || ((time_status & STA_PPSFREQ) != 0
31716         && (time_status &
31717             (STA_PPSWANDER|STA_PPSERROR)) != 0))
31718     /* p. 24, (d) */
31719     result = TIME_ERROR;
31720
31721 if ((txc->modes & ADJ_OFFSET_SINGLESHOT) ==
31722     ADJ_OFFSET_SINGLESHOT)
31723     txc->offset = save_adjust;
31724 else {
31725     if (time_offset < 0)
31726         txc->offset = -(time_offset >> SHIFT_UPDATE);
31727     else
31728         txc->offset = time_offset >> SHIFT_UPDATE;
31729 }
31730 txc->freq = time_freq + pps_freq;
31731 txc->maxerror = time_maxerror;
31732 txc->esterror = time_esterror;
31733 txc->status = time_status;
31734 txc->constant = time_constant;
31735 txc->precision = time_precision;
31736 txc->tolerance = time_tolerance;

```

```

31737 do_gettimeofday(&txc->time);
31738 txc->tick = tick;
31739 txc->ppsfreq = pps_freq;
31740 txc->jitter = pps_jitter >> PPS_AVG;
31741 txc->shift = pps_shift;
31742 txc->stabil = pps_stabil;
31743 txc->jitcnt = pps_jitcnt;
31744 txc->calcnt = pps_calcnt;
31745 txc->errcnt = pps_errcnt;
31746 txc->stbcnt = pps_stbcnt;
31747
31748 sti();
31749 return(result);
31750 }
31751
31752 asmlinkage int sys_adjtimex(struct timex *txc_p)
31753 {
31754     struct timex txc; /* Local copy of parameter */
31755     int ret;
31756
31757     /* Copy the user data space into the kernel copy
31758      * structure. But bear in mind that the structures may
31759      * change */
31760     if(copy_from_user(&txc, txc_p, sizeof(struct timex)))
31761         return -EFAULT;
31762     ret = do_adjtimex(&txc);
31763     return copy_to_user(txc_p, &txc, sizeof(struct timex))
31764         ? -EFAULT : ret;
31765 }

```

mm/memory.c

```

31766 /*
31767  * linux/mm/memory.c
31768  *
31769  * Copyright (C) 1991, 1992, 1993, 1994 Linus Torvalds
31770  */
31771
31772 /* demand-loading started 01.12.91 - seems it is high on
31773  * the list of things wanted, and it should be easy to
31774  * implement. - Linus */
31775
31776 /* Ok, demand-loading was easy, shared pages a little bit
31777  * trickier. Shared pages started 02.12.91, seems to
31778  * work. - Linus.
31779  *
31780  * Tested sharing by executing about 30 /bin/sh; under
31781  * the old kernel it would have taken more than the 6M I

```

```

31782  * have free, but it worked well as far as I could see.
31783  *
31784  * Also corrected some "invalidate()"s - I wasn't doing
31785  * enough of them. */
31786
31787 /* Real VM (paging to/from disk) started 18.12.91. Much
31788  * more work and thought has to go into this. Oh, well..
31789  * 19.12.91 - works, somewhat. Sometimes I get faults,
31790  * don't know why. Fund it. Everything seems to work
31791  * now.
31792  * 20.12.91 - Ok, making the swap-device changeable like
31793  * the root. */
31794 /* 05.04.94 - Multi-pg memory management added for v1.1.
31795  * Idea by Alex Bligh (alex@concepts.co.uk)
31796  */
31797
31798 #include <linux/mm.h>
31799 #include <linux/mman.h>
31800 #include <linux/swap.h>
31801 #include <linux/smp_lock.h>
31802
31803 #include <asm/uaccess.h>
31804 #include <asm/pgtable.h>
31805
31806 unsigned long max_mapnr = 0;
31807 unsigned long num_physpages = 0;
31808 void * high_memory = NULL;
31809
31810 /* We special-case the C-0-W ZERO_PAGE, because it's such
31811  * a common occurrence (no need to read the page to know
31812  * that it's zero - better for the cache and memory
31813  * subsystem). */
31814 static inline void copy_cow_page(unsigned long from,
31815                                 unsigned long to)
31816 {
31817     if (from == ZERO_PAGE) {
31818         clear_page(to);
31819         return;
31820     }
31821     copy_page(to, from);
31822 }
31823
31824 mem_map_t * mem_map = NULL;
31825
31826 /* oom() prints a message (so that the user knows why the
31827  * process died), and gives the process an untrappable
31828  * SIGKILL. */
31829 void oom(struct task_struct * task)

```

```

31830 {
31831     printk("\nOut of memory for %s.\n", task->comm);
31832     force_sig(SIGKILL, task);
31833 }
31834
31835 /* Note: this doesn't free the actual pages
31836 * themselves. That has been handled earlier when
31837 * unmapping all the memory regions. */
31838 static inline void free_one_pmd(pmd_t * dir)
31839 {
31840     pte_t * pte;
31841
31842     if (pmd_none(*dir))
31843         return;
31844     if (pmd_bad(*dir)) {
31845         printk("free_one_pmd: bad directory entry %08lx\n",
31846             pmd_val(*dir));
31847         pmd_clear(dir);
31848         return;
31849     }
31850     pte = pte_offset(dir, 0);
31851     pmd_clear(dir);
31852     pte_free(pte);
31853 }
31854
31855 static inline void free_one_pgdp(pgd_t * dir)
31856 {
31857     int j;
31858     pmd_t * pmd;
31859
31860     if (pgd_none(*dir))
31861         return;
31862     if (pgd_bad(*dir)) {
31863         printk("free_one_pgdp: bad directory entry %08lx\n",
31864             pgd_val(*dir));
31865         pgd_clear(dir);
31866         return;
31867     }
31868     pmd = pmd_offset(dir, 0);
31869     pgd_clear(dir);
31870     for (j = 0; j < PTRS_PER_PMD ; j++)
31871         free_one_pmd(pmd+j);
31872     pmd_free(pmd);
31873 }
31874
31875 /* Low and high watermarks for page table cache. The
31876 * system should try to have pgt_water[0] <= cache
31877 * elements <= pgt_water[1] */

```

```

31878 int pgt_cache_water[2] = { 25, 50 };
31879
31880 /* Returns the number of pages freed */
31881 int check_pgt_cache(void)
31882 {
31883     return do_check_pgt_cache(pgt_cache_water[0],
31884                             pgt_cache_water[1]);
31885 }
31886
31887
31888 /* This function clears all user-level page tables of a
31889 * process - this is needed by execve(), so that old
31890 * pages aren't in the way. */
31891 void clear_page_tables(struct mm_struct *mm,
31892                       unsigned long first, int nr)
31893 {
31894     pgd_t * page_dir = mm->pgd;
31895
31896     if (page_dir && page_dir != swapper_pg_dir) {
31897         page_dir += first;
31898         do {
31899             free_one_pgdp(page_dir);
31900             page_dir++;
31901         } while (--nr);
31902
31903         /* keep the page table cache within bounds */
31904         check_pgt_cache();
31905     }
31906 }
31907
31908 /* This function just free's the page directory - the
31909 * page tables themselves have been freed earlier by
31910 * clear_page_tables(). */
31911 void free_page_tables(struct mm_struct * mm)
31912 {
31913     pgd_t * page_dir = mm->pgd;
31914
31915     if (page_dir) {
31916         if (page_dir == swapper_pg_dir)
31917             goto out_bad;
31918         pgd_free(page_dir);
31919     }
31920     return;
31921
31922 out_bad:
31923     printk(KERN_ERR
31924         "free_page_tables: Trying to free kernel pgd\n");
31925     return;

```

```

31926 }
31927
31928 int new_page_tables(struct task_struct * tsk)
31929 {
31930     pgd_t * new_pg;
31931
31932     if (!(new_pg = pgd_alloc()))
31933         return -ENOMEM;
31934     SET_PAGE_DIR(tsk, new_pg);
31935     tsk->mm->pgd = new_pg;
31936     return 0;
31937 }
31938
31939 #define PTE_TABLE_MASK ((PTRS_PER_PTE-1) * sizeof(pte_t))
31940 #define PMD_TABLE_MASK ((PTRS_PER_PMD-1) * sizeof(pmd_t))
31941
31942 /* copy one vm_area from one task to the other. Assumes
31943  * the page tables already present in the new task to be
31944  * cleared in the whole range covered by this vma.
31945  *
31946  * 08Jan98 Merged into one routine from several inline
31947  * routines to reduce variable count and make things
31948  * faster. -jj */
31949 int copy_page_range(struct mm_struct *dst,
31950                    struct mm_struct *src,
31951                    struct vm_area_struct *vma)
31952 {
31953     pgd_t * src_pgd, * dst_pgd;
31954     unsigned long address = vma->vm_start;
31955     unsigned long end = vma->vm_end;
31956     unsigned long cow =
31957         (vma->vm_flags & (VM_SHARED | VM_MAYWRITE))
31958         == VM_MAYWRITE;
31959
31960     src_pgd = pgd_offset(src, address)-1;
31961     dst_pgd = pgd_offset(dst, address)-1;
31962
31963     for (;;) {
31964         pmd_t * src_pmd, * dst_pmd;
31965
31966         src_pgd++; dst_pgd++;
31967
31968         /* copy_pmd_range */
31969
31970         if (pgd_none(*src_pgd))
31971             goto skip_copy_pmd_range;
31972         if (pgd_bad(*src_pgd)) {
31973             printk("copy pmd range: bad pgd (%08lx)\n",

```

```

31974             pgd_val(*src_pgd));
31975             pgd_clear(src_pgd);
31976         skip_copy_pmd_range:
31977             address = (address + PGDIR_SIZE) & PGDIR_MASK;
31978             if (address >= end)
31979                 goto out;
31980             continue;
31981         }
31982         if (pgd_none(*dst_pgd)) {
31983             if (!pmd_alloc(dst_pgd, 0))
31984                 goto nomem;
31985         }
31986
31987         src_pmd = pmd_offset(src_pgd, address);
31988         dst_pmd = pmd_offset(dst_pgd, address);
31989
31990         do {
31991             pte_t * src_pte, * dst_pte;
31992
31993             /* copy_pte_range */
31994
31995             if (pmd_none(*src_pmd))
31996                 goto skip_copy_pte_range;
31997             if (pmd_bad(*src_pmd)) {
31998                 printk("copy_pte_range: bad pmd (%08lx)\n",
31999                     pmd_val(*src_pmd));
32000                 pmd_clear(src_pmd);
32001             skip_copy_pte_range:
32002                 address = (address + PMD_SIZE) & PMD_MASK;
32003                 if (address >= end)
32004                     goto out;
32005                 goto cont_copy_pmd_range;
32006             }
32007             if (pmd_none(*dst_pmd)) {
32008                 if (!pte_alloc(dst_pmd, 0))
32009                     goto nomem;
32010             }
32011
32012             src_pte = pte_offset(src_pmd, address);
32013             dst_pte = pte_offset(dst_pmd, address);
32014
32015             do {
32016                 pte_t pte = *src_pte;
32017                 unsigned long page_nr;
32018
32019                 /* copy_one_pte */
32020
32021                 if (pte_none(pte))

```

```

32022     goto cont_copy_pte_range;
32023     if (!pte_present(pte)) {
32024         swap_duplicate(pte_val(pte));
32025         set_pte(dst_pte, pte);
32026         goto cont_copy_pte_range;
32027     }
32028     page_nr = MAP_NR(pte_page(pte));
32029     if (page_nr >= max_mapnr ||
32030         PageReserved(mem_map+page_nr)) {
32031         set_pte(dst_pte, pte);
32032         goto cont_copy_pte_range;
32033     }
32034     /* If it's a COW mapping, write protect it both
32035      * in the parent and the child */
32036     if (cow) {
32037         pte = pte_wrprotect(pte);
32038         set_pte(src_pte, pte);
32039     }
32040     /* If it's a shared mapping, mark it clean in the
32041      * child */
32042     if (vma->vm_flags & VM_SHARED)
32043         pte = pte_mkclean(pte);
32044     set_pte(dst_pte, pte_mkold(pte));
32045     atomic_inc(&mem_map[page_nr].count);
32046
32047 cont_copy_pte_range:
32048     address += PAGE_SIZE;
32049     if (address >= end)
32050         goto out;
32051     src_pte++;
32052     dst_pte++;
32053     } while ((unsigned long)src_pte & PTE_TABLE_MASK);
32054
32055 cont_copy_pmd_range:
32056     src_pmd++;
32057     dst_pmd++;
32058     } while ((unsigned long)src_pmd & PMD_TABLE_MASK);
32059     }
32060 out:
32061     return 0;
32062
32063 nomem:
32064     return -ENOMEM;
32065 }
32066
32067 /* Return indicates whether a page was freed so caller
32068  * can adjust rss */
32069 static inline int free_pte(pte_t page)

```

```

32070 {
32071     if (pte_present(page)) {
32072         unsigned long addr = pte_page(page);
32073         if (MAP_NR(addr) >= max_mapnr ||
32074             PageReserved(mem_map+MAP_NR(addr)))
32075             return 0;
32076         /* free_page() used to be able to clear swap cache
32077          * entries. We may now have to do it manually. */
32078         free_page_and_swap_cache(addr);
32079         return 1;
32080     }
32081     swap_free(pte_val(page));
32082     return 0;
32083 }
32084
32085 static inline void forget_pte(pte_t page)
32086 {
32087     if (!pte_none(page)) {
32088         printk("forget_pte: old mapping existed!\n");
32089         free_pte(page);
32090     }
32091 }
32092
32093 static inline int zap_pte_range(pmd_t * pmd,
32094     unsigned long address, unsigned long size)
32095 {
32096     pte_t * pte;
32097     int freed;
32098
32099     if (pmd_none(*pmd))
32100         return 0;
32101     if (pmd_bad(*pmd)) {
32102         printk("zap_pte_range: bad pmd (%08lx)\n",
32103             pmd_val(*pmd));
32104         pmd_clear(pmd);
32105         return 0;
32106     }
32107     pte = pte_offset(pmd, address);
32108     address &= ~PMD_MASK;
32109     if (address + size > PMD_SIZE)
32110         size = PMD_SIZE - address;
32111     size >>= PAGE_SHIFT;
32112     freed = 0;
32113     for (;;) {
32114         pte_t page;
32115         if (!size)
32116             break;
32117         page = *pte;

```

```

32118     pte++;
32119     size--;
32120     if (pte_none(page))
32121         continue;
32122     pte_clear(pte-1);
32123     freed += free_pte(page);
32124 }
32125 return freed;
32126 }
32127
32128 static inline int zap_pmd_range(pgd_t * dir,
32129 unsigned long address, unsigned long size)
32130 {
32131     pmd_t * pmd;
32132     unsigned long end;
32133     int freed;
32134
32135     if (pgd_none(*dir))
32136         return 0;
32137     if (pgd_bad(*dir)) {
32138         printk("zap_pmd_range: bad pgd (%08lx)\n",
32139             pgd_val(*dir));
32140         pgd_clear(dir);
32141         return 0;
32142     }
32143     pmd = pmd_offset(dir, address);
32144     address &= ~PGDIR_MASK;
32145     end = address + size;
32146     if (end > PGDIR_SIZE)
32147         end = PGDIR_SIZE;
32148     freed = 0;
32149     do {
32150         freed += zap_pte_range(pmd, address, end - address);
32151         address = (address + PMD_SIZE) & PMD_MASK;
32152         pmd++;
32153     } while (address < end);
32154     return freed;
32155 }
32156
32157 /* remove user pages in a given range. */
32158 void zap_page_range(struct mm_struct *mm,
32159 unsigned long address, unsigned long size)
32160 {
32161     pgd_t * dir;
32162     unsigned long end = address + size;
32163     int freed = 0;
32164
32165     dir = pgd_offset(mm, address);

```

```

32166     while (address < end) {
32167         freed += zap_pmd_range(dir, address, end - address);
32168         address = (address + PGDIR_SIZE) & PGDIR_MASK;
32169         dir++;
32170     }
32171     /* Update rss for the mm_struct (not necessarily
32172      * current->mm) */
32173     if (mm->rss > 0) {
32174         mm->rss -= freed;
32175         if (mm->rss < 0)
32176             mm->rss = 0;
32177     }
32178 }
32179
32180 static inline void zeromap_pte_range(pte_t * pte,
32181 unsigned long address, unsigned long size,
32182 pte_t zero_pte)
32183 {
32184     unsigned long end;
32185
32186     address &= ~PMD_MASK;
32187     end = address + size;
32188     if (end > PMD_SIZE)
32189         end = PMD_SIZE;
32190     do {
32191         pte_t oldpage = *pte;
32192         set_pte(pte, zero_pte);
32193         forget_pte(oldpage);
32194         address += PAGE_SIZE;
32195         pte++;
32196     } while (address < end);
32197 }
32198
32199 static inline int zeromap_pmd_range(pmd_t * pmd,
32200 unsigned long address, unsigned long size,
32201 pte_t zero_pte)
32202 {
32203     unsigned long end;
32204
32205     address &= ~PGDIR_MASK;
32206     end = address + size;
32207     if (end > PGDIR_SIZE)
32208         end = PGDIR_SIZE;
32209     do {
32210         pte_t * pte = pte_alloc(pmd, address);
32211         if (!pte)
32212             return -ENOMEM;
32213         zeromap_pte_range(pte, address, end - address,

```

```

32214         zero_pte);
32215     address = (address + PMD_SIZE) & PMD_MASK;
32216     pmd++;
32217 } while (address < end);
32218 return 0;
32219 }
32220
32221 int zeromap_page_range(unsigned long address,
32222                       unsigned long size, pgprot_t prot)
32223 {
32224     int error = 0;
32225     pgd_t * dir;
32226     unsigned long beg = address;
32227     unsigned long end = address + size;
32228     pte_t zero_pte;
32229
32230     zero_pte = pte_wrprotect(mk_pte(ZERO_PAGE, prot));
32231     dir = pgd_offset(current->mm, address);
32232     flush_cache_range(current->mm, beg, end);
32233     while (address < end) {
32234         pmd_t *pmd = pmd_alloc(dir, address);
32235         error = -ENOMEM;
32236         if (!pmd)
32237             break;
32238         error = zeromap_pmd_range(pmd, address,
32239                                 end - address, zero_pte);
32240         if (error)
32241             break;
32242         address = (address + PGDIR_SIZE) & PGDIR_MASK;
32243         dir++;
32244     }
32245     flush_tlb_range(current->mm, beg, end);
32246     return error;
32247 }
32248
32249 /* maps a range of physical memory into the requested
32250 * pages. the old mappings are removed. any references to
32251 * nonexistent pages results in null mappings (currently
32252 * treated as "copy-on-access") */
32253 static inline void remap_pte_range(pte_t * pte,
32254     unsigned long address, unsigned long size,
32255     unsigned long phys_addr, pgprot_t prot)
32256 {
32257     unsigned long end;
32258
32259     address &= ~PMD_MASK;
32260     end = address + size;
32261     if (end > PMD_SIZE)

```

```

32262     end = PMD_SIZE;
32263     do {
32264         unsigned long mapnr;
32265         pte_t oldpage = *pte;
32266         pte_clear(pte);
32267
32268         mapnr = MAP_NR(__va(phys_addr));
32269         if (mapnr >= max_mapnr ||
32270             PageReserved(mem_map+mapnr))
32271             set_pte(pte, mk_pte_phys(phys_addr, prot));
32272         forget_pte(oldpage);
32273         address += PAGE_SIZE;
32274         phys_addr += PAGE_SIZE;
32275         pte++;
32276     } while (address < end);
32277 }
32278
32279 static inline int remap_pmd_range(pmd_t * pmd,
32280     unsigned long address, unsigned long size,
32281     unsigned long phys_addr, pgprot_t prot)
32282 {
32283     unsigned long end;
32284
32285     address &= ~PGDIR_MASK;
32286     end = address + size;
32287     if (end > PGDIR_SIZE)
32288         end = PGDIR_SIZE;
32289     phys_addr -= address;
32290     do {
32291         pte_t * pte = pte_alloc(pmd, address);
32292         if (!pte)
32293             return -ENOMEM;
32294         remap_pte_range(pte, address, end - address,
32295             address + phys_addr, prot);
32296         address = (address + PMD_SIZE) & PMD_MASK;
32297         pmd++;
32298     } while (address < end);
32299     return 0;
32300 }
32301
32302 int remap_page_range(unsigned long from,
32303     unsigned long phys_addr, unsigned long size,
32304     pgprot_t prot)
32305 {
32306     int error = 0;
32307     pgd_t * dir;
32308     unsigned long beg = from;
32309     unsigned long end = from + size;

```



```

32310
32311 phys_addr -= from;
32312 dir = pgd_offset(current->mm, from);
32313 flush_cache_range(current->mm, beg, end);
32314 while (from < end) {
32315     pmd_t *pmd = pmd_alloc(dir, from);
32316     error = -ENOMEM;
32317     if (!pmd)
32318         break;
32319     error = remap_pmd_range(pmd, from, end - from,
32320                             phys_addr + from, prot);
32321     if (error)
32322         break;
32323     from = (from + PGDIR_SIZE) & PGDIR_MASK;
32324     dir++;
32325 }
32326 flush_tlb_range(current->mm, beg, end);
32327 return error;
32328 }
32329
32330 /* sanity-check function.. */
32331 static void put_page(pte_t * page_table, pte_t pte)
32332 {
32333     if (!pte_none(*page_table)) {
32334         free_page_and_swap_cache(pte_page(pte));
32335         return;
32336     }
32337 /* no need for flush_tlb */
32338 set_pte(page_table, pte);
32339 }
32340
32341 /* This routine is used to map in a page into an address
32342 * space: needed by execve() for the initial stack and
32343 * environment pages. */
32344 unsigned long put_dirty_page(struct task_struct * tsk,
32345                             unsigned long page, unsigned long address)
32346 {
32347     pgd_t * pgd;
32348     pmd_t * pmd;
32349     pte_t * pte;
32350
32351     if (MAP_NR(page) >= max_mapnr)
32352         printk("put_dirty_page: trying to put page %08lx at "
32353                "%08lx\n", page, address);
32354     if (atomic_read(&mem_map[MAP_NR(page)].count) != 1)
32355         printk("mem_map disagrees with %08lx at %08lx\n",
32356                page, address);
32357     pgd = pgd_offset(tsk->mm, address);

```

```

32358     pmd = pmd_alloc(pgd, address);
32359     if (!pmd) {
32360         free_page(page);
32361         oom(tsk);
32362         return 0;
32363     }
32364     pte = pte_alloc(pmd, address);
32365     if (!pte) {
32366         free_page(page);
32367         oom(tsk);
32368         return 0;
32369     }
32370     if (!pte_none(*pte)) {
32371         printk("put_dirty_page: page already exists\n");
32372         free_page(page);
32373         return 0;
32374     }
32375     flush_page_to_ram(page);
32376     set_pte(pte, pte_mkwrite(pte_mkdirty(mk_pte(page,
32377                                             PAGE_COPY))));
32378     /* no need for flush_tlb */
32379     return page;
32380 }
32381
32382 /* This routine handles present pages, when users try to
32383 * write to a shared page. It is done by copying the page
32384 * to a new address and decrementing the shared-page
32385 * counter for the old page.
32386 *
32387 * Goto-purists beware: the only reason for goto's here
32388 * is that it results in better assembly code.. The
32389 * "default" path will see no jumps at all.
32390 *
32391 * Note that this routine assumes that the protection
32392 * checks have been done by the caller (the low-level
32393 * page fault routine in most cases). Thus we can safely
32394 * just mark it writable once we've done any necessary
32395 * COW.
32396 *
32397 * We also mark the page dirty at this point even though
32398 * the page will change only once the write actually
32399 * happens. This avoids a few races, and potentially
32400 * makes it more efficient. */
32401 static int do_wp_page(struct task_struct * tsk,
32402                      struct vm_area_struct * vma, unsigned long address,
32403                      pte_t *page_table)
32404 {
32405     pte_t pte;

```

▶ p 590

```

32406 unsigned long old_page, new_page;
32407 struct page * page_map;
32408
32409 pte = *page_table;
32410 new_page = __get_free_page(GFP_USER);
32411 /* Did someone else copy this page for us while we
32412  * slept? */
32413 if (pte_val(*page_table) != pte_val(pte))
32414     goto end_wp_page;
32415 if (!pte_present(pte))
32416     goto end_wp_page;
32417 if (pte_write(pte))
32418     goto end_wp_page;
32419 old_page = pte_page(pte);
32420 if (MAP_NR(old_page) >= max_mapnr)
32421     goto bad_wp_page;
32422 tsk->min_flt++;
32423 page_map = mem_map + MAP_NR(old_page);
32424
32425 /* We can avoid the copy if:
32426  * - we're the only user (count == 1)
32427  * - the only other user is the swap cache,
32428  *   and the only swap cache user is itself,
32429  *   in which case we can remove the page
32430  *   from the swap cache.
32431  */
32432 switch (atomic_read(&page_map->count)) {
32433     case 2:
32434         if (!PageSwapCache(page_map))
32435             break;
32436         if (swap_count(page_map->offset) != 1)
32437             break;
32438         delete_from_swap_cache(page_map);
32439         /* FallThrough */
32440     case 1:
32441         /* We can release the kernel lock now.. */
32442         unlock_kernel();
32443
32444         flush_cache_page(vma, address);
32445         set_pte(page_table, pte_mkdirty(pte_mkwrite(pte)));
32446         flush_tlb_page(vma, address);
32447     end_wp_page:
32448         if (new_page)
32449             free_page(new_page);
32450         return 1;
32451     }
32452
32453     unlock_kernel();

```

```

32454     if (!new_page)
32455         return 0;
32456
32457     if (PageReserved(mem_map + MAP_NR(old_page)))
32458         ++vma->vm_mm->rss;
32459     copy_cow_page(old_page, new_page);
32460     flush_page_to_ram(old_page);
32461     flush_page_to_ram(new_page);
32462     flush_cache_page(vma, address);
32463     set_pte(page_table,
32464             pte_mkwrite(pte_mkdirty(mk_pte(new_page,
32465                                         vma->vm_page_prot))));
32466     free_page(old_page);
32467     flush_tlb_page(vma, address);
32468     return 1;
32469
32470 bad_wp_page:
32471     printk("do_wp_page: bogus page at address "
32472           "%08lx (%08lx)\n", address, old_page);
32473     send_sig(SIGKILL, tsk, 1);
32474     if (new_page)
32475         free_page(new_page);
32476     return 0;
32477 }
32478
32479 /* This function zeroes out partial mmap'ed pages at
32480  truncation time.. */
32481 static void partial_clear(struct vm_area_struct *vma,
32482                          unsigned long address)
32483 {
32484     pgd_t *page_dir;
32485     pmd_t *page_middle;
32486     pte_t *page_table, pte;
32487
32488     page_dir = pgd_offset(vma->vm_mm, address);
32489     if (pgd_none(*page_dir))
32490         return;
32491     if (pgd_bad(*page_dir)) {
32492         printk("bad page table directory entry %p:[%lx]\n",
32493               page_dir, pgd_val(*page_dir));
32494         pgd_clear(page_dir);
32495         return;
32496     }
32497     page_middle = pmd_offset(page_dir, address);
32498     if (pmd_none(*page_middle))
32499         return;
32500     if (pmd_bad(*page_middle)) {
32501         printk("bad page table directory entry %p:[%lx]\n",

```

```

32502         page_dir, pgd_val(*page_dir));
32503     pmd_clear(page_middle);
32504     return;
32505 }
32506     page_table = pte_offset(page_middle, address);
32507     pte = *page_table;
32508     if (!pte_present(pte))
32509         return;
32510     flush_cache_page(vma, address);
32511     address &= ~PAGE_MASK;
32512     address += pte_page(pte);
32513     if (MAP_NR(address) >= max_mapnr)
32514         return;
32515     memset((void *) address, 0,
32516           PAGE_SIZE - (address & ~PAGE_MASK));
32517     flush_page_to_ram(pte_page(pte));
32518 }
32519
32520 /* Handle all mappings that got truncated by a
32521  * "truncate()" system call.
32522  *
32523  * NOTE! We have to be ready to update the memory sharing
32524  * between the file and the memory map for a potential
32525  * last incomplete page. Ugly, but necessary. */
32526 void vmtruncate(struct inode * inode,
32527                unsigned long offset)
32528 {
32529     struct vm_area_struct * mpnt;
32530
32531     truncate_inode_pages(inode, offset);
32532     if (!inode->i_mmap)
32533         return;
32534     mpnt = inode->i_mmap;
32535     do {
32536         struct mm_struct *mm = mpnt->vm_mm;
32537         unsigned long start = mpnt->vm_start;
32538         unsigned long end = mpnt->vm_end;
32539         unsigned long len = end - start;
32540         unsigned long diff;
32541
32542         /* mapping wholly truncated? */
32543         if (mpnt->vm_offset >= offset) {
32544             flush_cache_range(mm, start, end);
32545             zap_page_range(mm, start, len);
32546             flush_tlb_range(mm, start, end);
32547             continue;
32548         }
32549         /* mapping wholly unaffected? */

```

```

32550         diff = offset - mpnt->vm_offset;
32551         if (diff >= len)
32552             continue;
32553         /* Ok, partially affected.. */
32554         start += diff;
32555         len = (len - diff) & PAGE_MASK;
32556         if (start & ~PAGE_MASK) {
32557             partial_clear(mpnt, start);
32558             start = (start + ~PAGE_MASK) & PAGE_MASK;
32559         }
32560         flush_cache_range(mm, start, end);
32561         zap_page_range(mm, start, len);
32562         flush_tlb_range(mm, start, end);
32563     } while ((mpnt = mpnt->vm_next_share) != NULL);
32564 }
32565
32566
32567 /* This is called with the kernel lock held, we need to
32568  * return without it. */
32569 static int do_swap_page(struct task_struct * tsk,
32570                        struct vm_area_struct * vma, unsigned long address,
32571                        pte_t * page_table, pte_t entry, int write_access)
32572 {
32573     if (!vma->vm_ops || !vma->vm_ops->swapon) {
32574         swapon_in(tsk, vma, page_table, pte_val(entry),
32575                 write_access);
32576         flush_page_to_ram(pte_page(*page_table));
32577     } else {
32578         pte_t page =
32579             vma->vm_ops->swapon(vma,
32580                                address - vma->vm_start + vma->vm_offset,
32581                                pte_val(entry));
32582         if (pte_val(*page_table) != pte_val(entry)) {
32583             free_page(pte_page(page));
32584         } else {
32585             if (atomic_read(&mem_map[MAP_NR(pte_page(page))].
32586                             count) > 1 &&
32587                 !(vma->vm_flags & VM_SHARED))
32588                 page = pte_wrprotect(page);
32589             ++vma->vm_mm->rss;
32590             ++tsk->maj_flt;
32591             flush_page_to_ram(pte_page(page));
32592             set_pte(page_table, page);
32593         }
32594     }
32595     unlock_kernel();
32596     return 1;
32597 }

```

```

32598
32599 /* This only needs the MM semaphore */
32600 static int do_anonymous_page(struct task_struct * tsk,
32601     struct vm_area_struct * vma, pte_t *page_table,
32602     int write_access)
32603 {
32604     pte_t entry = pte_wrprotect(mk_pte(ZERO_PAGE,
32605         vma->vm_page_prot));
32606     if (write_access) {
32607         unsigned long page = __get_free_page(GFP_USER);
32608         if (!page)
32609             return 0;
32610         clear_page(page);
32611         entry = pte_mkwrite(pte_mkdirty(mk_pte(page,
32612             vma->vm_page_prot)));
32613         vma->vm_mm->rss++;
32614         tsk->min_flt++;
32615         flush_page_to_ram(page);
32616     }
32617     put_page(page_table, entry);
32618     return 1;
32619 }
32620
32621 /* do_no_page() tries to create a new page mapping. It
32622  * aggressively tries to share with existing pages, but
32623  * makes a separate copy if the "write_access" parameter
32624  * is true in order to avoid the next page fault.
32625  *
32626  * As this is called only for pages that do not currently
32627  * exist, we do not need to flush old virtual caches or
32628  * the TLB.
32629  *
32630  * This is called with the MM semaphore and the kernel
32631  * lock held. We need to release the kernel lock as soon
32632  * as possible.. */
32633 static int do_no_page(struct task_struct * tsk,
32634     struct vm_area_struct * vma, unsigned long address,
32635     int write_access, pte_t *page_table)
32636 {
32637     unsigned long page;
32638     pte_t entry;
32639
32640     if (!vma->vm_ops || !vma->vm_ops->nopage) {
32641         unlock_kernel();
32642         return do_anonymous_page(tsk, vma, page_table,
32643             write_access);
32644     }
32645
32646     /* The third argument is "no_share", which tells the
32647     * low-level code to copy, not share the page even if
32648     * sharing is possible. It's essentially an early COW
32649     * detection. */
32650     page = vma->vm_ops->nopage(vma, address & PAGE_MASK,
32651         (vma->vm_flags & VM_SHARED)?0:write_access);
32652
32653     unlock_kernel();
32654     if (!page)
32655         return 0;
32656
32657     ++tsk->maj_flt;
32658     ++vma->vm_mm->rss;
32659     /* This silly early PAGE_DIRTY setting removes a race
32660     * due to the bad i386 page protection. But it's valid
32661     * for other architectures too.
32662     *
32663     * Note that if write_access is true, we either now
32664     * have an exclusive copy of the page, or this is a
32665     * shared mapping, so we can make it writable and dirty
32666     * to avoid having to handle that later. */
32667     flush_page_to_ram(page);
32668     entry = mk_pte(page, vma->vm_page_prot);
32669     if (write_access) {
32670         entry = pte_mkwrite(pte_mkdirty(entry));
32671     } else if (atomic_read(&mem_map[MAP_NR(page)].
32672         count) > 1 &&
32673         !(vma->vm_flags & VM_SHARED))
32674         entry = pte_wrprotect(entry);
32675     put_page(page_table, entry);
32676     /* no need to invalidate: a not present page shouldn't
32677     * be cached */
32678     return 1;
32679 }
32680
32681 /* These routines also need to handle stuff like marking
32682  * pages dirty and/or accessed for architectures that
32683  * don't do it in hardware (most RISC architectures).
32684  * The early dirtying is also good on the i386.
32685  *
32686  * There is also a hook called "update_mmu_cache()" that
32687  * architectures with external mmu caches can use to
32688  * update those (ie the Sparc or PowerPC hashed page
32689  * tables that act as extended TLBs). */
32690 static inline int handle_pte_fault(
32691     struct task_struct *tsk,
32692     struct vm_area_struct * vma, unsigned long address,
32693     int write_access, pte_t * pte)

```

```

32694 {
32695     pte_t entry;
32696
32697     lock_kernel();
32698     entry = *pte;
32699
32700     if (!pte_present(entry)) {
32701         if (pte_none(entry))
32702             return do_no_page(tsk, vma, address, write_access,
32703                               pte);
32704         return do_swap_page(tsk, vma, address, pte, entry,
32705                             write_access);
32706     }
32707
32708     entry = pte_mkyoung(entry);
32709     set_pte(pte, entry);
32710     flush_tlb_page(vma, address);
32711     if (write_access) {
32712         if (!pte_write(entry))
32713             return do_wp_page(tsk, vma, address, pte);
32714
32715         entry = pte_mkdirty(entry);
32716         set_pte(pte, entry);
32717         flush_tlb_page(vma, address);
32718     }
32719     unlock_kernel();
32720     return 1;
32721 }
32722
32723 /* By the time we get here, we already hold the mm
32724 * semaphore */
32725 int handle_mm_fault(struct task_struct *tsk,
32726                    struct vm_area_struct * vma, unsigned long address,
32727                    int write_access)
32728 {
32729     pgd_t *pgd;
32730     pmd_t *pmd;
32731
32732     pgd = pgd_offset(vma->vm_mm, address);
32733     pmd = pmd_allqc(pgd, address);
32734     if (pmd) {
32735         pte_t *pte = pte_alloc(pmd, address);
32736         if (pte) {
32737             if (handle_pte_fault(tsk, vma, address,
32738                                 write_access, pte)) {
32739                 update_mmu_cache(vma, address, *pte);
32740                 return 1;
32741             }

```

```

32742     }
32743 }
32744 return 0;
32745 }
32746
32747 /* Simplistic page force-in.. */
32748 void make_pages_present(unsigned long addr,
32749                          unsigned long end)
32750 {
32751     int write;
32752     struct vm_area_struct * vma;
32753
32754     vma = find_vma(current->mm, addr);
32755     write = (vma->vm_flags & VM_WRITE) != 0;
32756     while (addr < end) {
32757         handle_mm_fault(current, vma, addr, write);
32758         addr += PAGE_SIZE;
32759     }
32760 }

```

mm/mlock.c

```

32761 /*
32762 *      linux/mm/mlock.c
32763 *
32764 *      (C) Copyright 1995 Linus Torvalds
32765 */
32766 #include <linux/slab.h>
32767 #include <linux/shm.h>
32768 #include <linux/mman.h>
32769 #include <linux/smp_lock.h>
32770
32771 #include <asm/uaccess.h>
32772 #include <asm/pgtable.h>
32773
32774 static inline int mlock_fixup_all(
32775     struct vm_area_struct * vma, int newflags)
32776 {
32777     vma->vm_flags = newflags;
32778     return 0;
32779 }
32780
32781 static inline int mlock_fixup_start(
32782     struct vm_area_struct * vma, unsigned long end,
32783     int newflags)
32784 {
32785     struct vm_area_struct * n;
32786

```

```

32787 n = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
32788 if (!n)
32789     return -EAGAIN;
32790 *n = *vma;
32791 vma->vm_start = end;
32792 n->vm_end = end;
32793 vma->vm_offset += vma->vm_start - n->vm_start;
32794 n->vm_flags = newflags;
32795 if (n->vm_file)
32796     n->vm_file->f_count++;
32797 if (n->vm_ops && n->vm_ops->open)
32798     n->vm_ops->open(n);
32799 insert_vm_struct(current->mm, n);
32800 return 0;
32801 }
32802
32803 static inline int mlock_fixup_end(
32804     struct vm_area_struct * vma,
32805     unsigned long start, int newflags)
32806 {
32807     struct vm_area_struct * n;
32808
32809     n = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
32810     if (!n)
32811         return -EAGAIN;
32812     *n = *vma;
32813     vma->vm_end = start;
32814     n->vm_start = start;
32815     n->vm_offset += n->vm_start - vma->vm_start;
32816     n->vm_flags = newflags;
32817     if (n->vm_file)
32818         n->vm_file->f_count++;
32819     if (n->vm_ops && n->vm_ops->open)
32820         n->vm_ops->open(n);
32821     insert_vm_struct(current->mm, n);
32822     return 0;
32823 }
32824
32825 static inline int mlock_fixup_middle(
32826     struct vm_area_struct * vma,
32827     unsigned long start, unsigned long end, int newflags)
32828 {
32829     struct vm_area_struct * left, * right;
32830
32831     left = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
32832     if (!left)
32833         return -EAGAIN;
32834     right = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);

```

```

32835     if (!right) {
32836         kmem_cache_free(vm_area_cachep, left);
32837         return -EAGAIN;
32838     }
32839     *left = *vma;
32840     *right = *vma;
32841     left->vm_end = start;
32842     vma->vm_start = start;
32843     vma->vm_end = end;
32844     right->vm_start = end;
32845     vma->vm_offset += vma->vm_start - left->vm_start;
32846     right->vm_offset += right->vm_start - left->vm_start;
32847     vma->vm_flags = newflags;
32848     if (vma->vm_file)
32849         vma->vm_file->f_count += 2;
32850
32851     if (vma->vm_ops && vma->vm_ops->open) {
32852         vma->vm_ops->open(left);
32853         vma->vm_ops->open(right);
32854     }
32855     insert_vm_struct(current->mm, left);
32856     insert_vm_struct(current->mm, right);
32857     return 0;
32858 }
32859
32860 static int mlock_fixup(struct vm_area_struct * vma,
32861     unsigned long start, unsigned long end,
32862     unsigned int newflags)
32863 {
32864     int pages, retval;
32865
32866     if (newflags == vma->vm_flags)
32867         return 0;
32868
32869     if (start == vma->vm_start) {
32870         if (end == vma->vm_end)
32871             retval = mlock_fixup_all(vma, newflags);
32872         else
32873             retval = mlock_fixup_start(vma, end, newflags);
32874     } else {
32875         if (end == vma->vm_end)
32876             retval = mlock_fixup_end(vma, start, newflags);
32877         else
32878             retval = mlock_fixup_middle(vma, start, end,
32879                 newflags);
32880     }
32881     if (!retval) {
32882         /* keep track of amount of locked VM */

```

```

32883     pages = (end - start) >> PAGE_SHIFT;
32884     if (!(newflags & VM_LOCKED))
32885         pages = -pages;
32886     vma->vm_mm->locked_vm += pages;
32887     make_pages_present(start, end);
32888 }
32889     return retval;
32890 }
32891
32892 static int do_mlock(unsigned long start, size_t len,
32893                    int on)
32894 {
32895     unsigned long nstart, end, tmp;
32896     struct vm_area_struct * vma, * next;
32897     int error;
32898
32899     if (!capable(CAP_IPC_LOCK))
32900         return -EPERM;
32901     len = (len + ~PAGE_MASK) & PAGE_MASK;
32902     end = start + len;
32903     if (end < start)
32904         return -EINVAL;
32905     if (end == start)
32906         return 0;
32907     vma = find_vma(current->mm, start);
32908     if (!vma || vma->vm_start > start)
32909         return -ENOMEM;
32910
32911     for (nstart = start ; ; ) {
32912         unsigned int newflags;
32913
32914         /* Here we know that
32915          * vma->vm_start <= nstart < vma->vm_end. */
32916         newflags = vma->vm_flags | VM_LOCKED;
32917         if (!on)
32918             newflags &= ~VM_LOCKED;
32919
32920         if (vma->vm_end >= end) {
32921             error = mlock_fixup(vma, nstart, end, newflags);
32922             break;
32923         }
32924
32925         tmp = vma->vm_end;
32926         next = vma->vm_next;
32927         error = mlock_fixup(vma, nstart, tmp, newflags);
32928         if (error)
32929             break;
32930         nstart = tmp;

```

```

32931     vma = next;
32932     if (!vma || vma->vm_start != nstart) {
32933         error = -ENOMEM;
32934         break;
32935     }
32936 }
32937 merge_segments(current->mm, start, end);
32938 return error;
32939 }
32940
32941 asmlinkage int sys_mlock(unsigned long start, size_t len)
32942 {
32943     unsigned long locked;
32944     unsigned long lock_limit;
32945     int error = -ENOMEM;
32946
32947     down(&current->mm->mmap_sem);
32948     lock_kernel();
32949     len = (len + (start & ~PAGE_MASK) + ~PAGE_MASK) &
32950         PAGE_MASK;
32951     start &= PAGE_MASK;
32952
32953     locked = len >> PAGE_SHIFT;
32954     locked += current->mm->locked_vm;
32955
32956     lock_limit = current->rlim[RLIMIT_MEMLOCK].rlim_cur;
32957     lock_limit >>= PAGE_SHIFT;
32958
32959     /* check against resource limits */
32960     if (locked > lock_limit)
32961         goto out;
32962
32963     /* we may lock at most half of physical memory... */
32964     /* (this check is pretty bogus, but doesn't hurt) */
32965     if (locked > num_physpages/2)
32966         goto out;
32967
32968     error = do_mlock(start, len, 1);
32969 out:
32970     unlock_kernel();
32971     up(&current->mm->mmap_sem);
32972     return error;
32973 }
32974
32975 asmlinkage int sys_munlock(unsigned long start,
32976                             size_t len)
32977 {
32978     int ret;

```

```

32979     down(&current->mm->mmap_sem);
32980     lock_kernel();
32981     len = (len + (start & ~PAGE_MASK) + ~PAGE_MASK) &
32982         PAGE_MASK;
32984     start &= PAGE_MASK;
32985     ret = do_mlock(start, len, 0);
32986     unlock_kernel();
32987     up(&current->mm->mmap_sem);
32988     return ret;
32989 }
32990
32991 static int do_mlockall(int flags)
32992 {
32993     int error;
32994     unsigned int def_flags;
32995     struct vm_area_struct * vma;
32996
32997     if (!capable(CAP_IPC_LOCK))
32998         return -EPERM;
32999
33000     def_flags = 0;
33001     if (flags & MCL_FUTURE)
33002         def_flags = VM_LOCKED;
33003     current->mm->def_flags = def_flags;
33004
33005     error = 0;
33006     for (vma = current->mm->mmap; vma;
33007         vma = vma->vm_next) {
33008         unsigned int newflags;
33009
33010         newflags = vma->vm_flags | VM_LOCKED;
33011         if (!(flags & MCL_CURRENT))
33012             newflags &= ~VM_LOCKED;
33013         error = mlock_fixup(vma, vma->vm_start, vma->vm_end,
33014                             newflags);
33015         if (error)
33016             break;
33017     }
33018     merge_segments(current->mm, 0, TASK_SIZE);
33019     return error;
33020 }
33021
33022 asmlinkage int sys_mlockall(int flags)
33023 {
33024     unsigned long lock_limit;
33025     int ret = -EINVAL;
33026

```

```

33027     down(&current->mm->mmap_sem);
33028     lock_kernel();
33029     if (!flags || (flags & ~(MCL_CURRENT | MCL_FUTURE)))
33030         goto out;
33031
33032     lock_limit = current->rlim[RLIMIT_MEMLOCK].rlim_cur;
33033     lock_limit >>= PAGE_SHIFT;
33034
33035     ret = -ENOMEM;
33036     if (current->mm->total_vm > lock_limit)
33037         goto out;
33038
33039     /* we may lock at most half of physical memory... */
33040     /* (this check is pretty bogus, but doesn't hurt) */
33041     if (current->mm->total_vm > num_physpages/2)
33042         goto out;
33043
33044     ret = do_mlockall(flags);
33045 out:
33046     unlock_kernel();
33047     up(&current->mm->mmap_sem);
33048     return ret;
33049 }
33050
33051 asmlinkage int sys_munlockall(void)
33052 {
33053     int ret;
33054
33055     down(&current->mm->mmap_sem);
33056     lock_kernel();
33057     ret = do_mlockall(0);
33058     unlock_kernel();
33059     up(&current->mm->mmap_sem);
33060     return ret;
33061 }

```

mm/mmap.c

```

33062 /*
33063  *      linux/mm/mmap.c
33064  *
33065  * Written by obz.
33066  */
33067 #include <linux/slab.h>
33068 #include <linux/shm.h>
33069 #include <linux/mman.h>
33070 #include <linux/pagemap.h>
33071 #include <linux/swap.h>

```



```

33072 #include <linux/swapctl.h>
33073 #include <linux/smp_lock.h>
33074 #include <linux/init.h>
33075 #include <linux/file.h>
33076
33077 #include <asm/uaccess.h>
33078 #include <asm/pgtable.h>
33079
33080 /* description of effects of mapping type and prot in
33081  * current implementation. this is due to the limited
33082  * x86 page protection hardware. The expected behavior
33083  * is in parens (Y = yes, N = no, C = copy):
33084  *
33085  * map_type      prot
33086  *               PROT_NONE PROT_READ PROT_WRITE PROT_EXEC
33087  * MAP_SHARED   r: (N) N   r: (Y) Y   r: (N) Y   r: (N) Y
33088  *               w: (N) N   w: (N) N   w: (Y) Y   w: (N) N
33089  *               x: (N) N   x: (N) Y   x: (N) Y   x: (Y) Y
33090  *
33091  * MAP_PRIVATE r: (N) N   r: (Y) Y   r: (N) Y   r: (N) Y
33092  *               w: (N) N   w: (N) N   w: (C) C   w: (N) N
33093  *               x: (N) N   x: (N) Y   x: (N) Y   x: (Y) Y
33094  */
33095 pgprot_t protection_map[16] = {
33096     __P000, __P001, __P010, __P011,
33097     __P100, __P101, __P110, __P111,
33098     __S000, __S001, __S010, __S011,
33099     __S100, __S101, __S110, __S111
33100 };
33101
33102 /* SLAB cache for vm_area_struct's. */
33103 kmem_cache_t *vm_area_cachep;
33104
33105 int sysctl_overcommit_memory;
33106
33107 /* Check that a process has enough memory to allocate a
33108  * new virtual mapping.
33109  */
33110 int vm_enough_memory(long pages)
33111 {
33112     /* Stupid algorithm to decide if we have enough memory:
33113      * while simple, it hopefully works in most obvious
33114      * cases.. Easy to fool it, but this should catch most
33115      * mistakes. */
33116     /* 23/11/98 NJC: Somewhat less stupid version of
33117      * algorithm, which tries to do "TheRightThing".
33118      * Instead of using half of (buffers+cache), use the
33119      * minimum values. Allow an extra 2% of num_physpages

```

```

33120     * for safety margin. */
33121
33122     long free;
33123
33124     /* Sometimes we want to use more memory than we
33125      * have. */
33126     if (sysctl_overcommit_memory)
33127         return 1;
33128
33129     free = buffermem >> PAGE_SHIFT;
33130     free += page_cache_size;
33131     free += nr_free_pages;
33132     free += nr_swap_pages;
33133     free -= (page_cache.min_percent +
33134             buffer_mem.min_percent + 2)*num_physpages/100;
33135     return free > pages;
33136 }
33137
33138 /* Remove one vm structure from the inode's i_mmap
33139  * ring. */
33140 static inline void remove_shared_vm_struct(
33141     struct vm_area_struct *vma)
33142 {
33143     struct file * file = vma->vm_file;
33144
33145     if (file) {
33146         if (vma->vm_flags & VM_DENYWRITE)
33147             file->f_dentry->d_inode->i_writcount++;
33148         if(vma->vm_next_share)
33149             vma->vm_next_share->vm_pprev_share =
33150                 vma->vm_pprev_share;
33151         *vma->vm_pprev_share = vma->vm_next_share;
33152     }
33153 }
33154
33155 asmlinkage unsigned long sys_brk(unsigned long brk)
33156 {
33157     unsigned long rlim, retval;
33158     unsigned long newbrk, oldbrk;
33159     struct mm_struct *mm = current->mm;
33160
33161     down(&mm->mmap_sem);
33162
33163     /* This lock-kernel is one of the main contention
33164      * points for certain normal loads. And it really
33165      * should not be here: almost everything in
33166      * brk()/mmap()/munmap() is protected sufficiently by
33167      * the mmap semaphore that we got above.

```

→ p 561

```

33168 *
33169 * We should move this into the few things that really
33170 * want the lock, namely anything that actually touches
33171 * a file descriptor etc. We can do all the normal
33172 * anonymous mapping cases without ever getting the
33173 * lock at all - the actual memory management code is
33174 * already completely thread-safe. */
33175 lock_kernel();
33176
33177 if (brk < mm->end_code)
33178     goto out;
33179 newbrk = PAGE_ALIGN(brk);
33180 oldbrk = PAGE_ALIGN(mm->brk);
33181 if (oldbrk == newbrk)
33182     goto set_brk;
33183
33184 /* Always allow shrinking brk. */
33185 if (brk <= mm->brk) {
33186     if (!do_munmap(newbrk, oldbrk-newbrk))
33187         goto set_brk;
33188     goto out;
33189 }
33190
33191 /* Check against rlimit and stack. */
33192 rlim = current->rlim[RLIMIT_DATA].rlim_cur;
33193 if (rlim < RLIM_INFINITY && brk - mm->end_code > rlim)
33194     goto out;
33195
33196 /* Check against existing mmap mappings. */
33197 if (find_vma_intersection(mm, oldbrk, newbrk+PAGE_SIZE))
33198     goto out;
33199
33200 /* Check if we have enough memory. */
33201 if (!vm_enough_memory((newbrk-oldbrk) >> PAGE_SHIFT))
33202     goto out;
33203
33204 /* Ok, looks good - let it rip. */
33205 if (do_mmap(NULL, oldbrk, newbrk-oldbrk,
33206     PROT_READ|PROT_WRITE|PROT_EXEC,
33207     MAP_FIXED|MAP_PRIVATE, 0) != oldbrk)
33208     goto out;
33209 set_brk:
33210     mm->brk = brk;
33211 out:
33212     retval = mm->brk;
33213     unlock_kernel();
33214     up(&mm->mmap_sem);
33215     return retval;

```

p 56

```

33216 }
33217
33218 /* Combine the mmap "prot" and "flags" argument into one
33219 * "vm_flags" used internally. Essentially, translate the
33220 * "PROT_xxx" and "MAP_xxx" bits into "VM_xxx". */
33221 static inline unsigned long vm_flags(unsigned long prot,
33222                                     unsigned long flags)
33223 {
33224     #define _trans(x,bit1,bit2)
33225     ((bit1==bit2)?(x&bit1):(x&bit1)?bit2:0)
33226
33227     unsigned long prot_bits, flag_bits;
33228     prot_bits =
33229         _trans(prot, PROT_READ, VM_READ) |
33230         _trans(prot, PROT_WRITE, VM_WRITE) |
33231         _trans(prot, PROT_EXEC, VM_EXEC);
33232     flag_bits =
33233         _trans(flags, MAP_GROWSDOWN, VM_GROWSDOWN) |
33234         _trans(flags, MAP_DENYWRITE, VM_DENYWRITE) |
33235         _trans(flags, MAP_EXECUTABLE, VM_EXECUTABLE);
33236     return prot_bits | flag_bits;
33237 #undef _trans
33238 }
33239
33240 unsigned long do_mmap(struct file * file,
33241                     unsigned long addr, unsigned long len,
33242                     unsigned long prot, unsigned long flags,
33243                     unsigned long off)
33244 {
33245     struct mm_struct * mm = current->mm;
33246     struct vm_area_struct * vma;
33247     int error;
33248
33249     if ((len = PAGE_ALIGN(len)) == 0)
33250         return addr;
33251
33252     if (len > TASK_SIZE || addr > TASK_SIZE-len)
33253         return -EINVAL;
33254
33255     /* offset overflow? */
33256     if (off + len < off)
33257         return -EINVAL;
33258
33259     /* Too many mappings? */
33260     if (mm->map_count > MAX_MAP_COUNT)
33261         return -ENOMEM;
33262
33263     /* mlock MCL_FUTURE? */

```

p 55

```

33264 if (mm->def_flags & VM_LOCKED) {
33265     unsigned long locked = mm->locked_vm << PAGE_SHIFT;
33266     locked += len;
33267     if (locked > current->rlim[RLIMIT_MEMLOCK].rlim_cur)
33268         return -EAGAIN;
33269 }
33270
33271 /* Do simple checking here so the lower-level routines
33272  * won't have to. we assume access permissions have
33273  * been handled by the open of the memory object, so we
33274  * don't do any here. */
33275 if (file != NULL) {
33276     switch (flags & MAP_TYPE) {
33277     case MAP_SHARED:
33278         if ((prot & PROT_WRITE) && !(file->f_mode & 2))
33279             return -EACCESS;
33280
33281         /* Make sure we don't allow writing to an
33282          * append-only file.. */
33283         if (IS_APPEND(file->f_dentry->d_inode) &&
33284             (file->f_mode & 2))
33285             return -EACCESS;
33286
33287         /* make sure there are no mandatory locks on the
33288          * file. */
33289         if (locks_verify_locked(file->f_dentry->d_inode))
33290             return -EAGAIN;
33291
33292         /* fall through */
33293     case MAP_PRIVATE:
33294         if (!(file->f_mode & 1))
33295             return -EACCESS;
33296         break;
33297
33298     default:
33299         return -EINVAL;
33300     }
33301 } else if ((flags & MAP_TYPE) != MAP_PRIVATE)
33302     return -EINVAL;
33303
33304 /* Obtain the address to map to. we verify (or select)
33305  * it and ensure that it represents a valid section of
33306  * the address space. */
33307 if (flags & MAP_FIXED) {
33308     if (addr & ~PAGE_MASK)
33309         return -EINVAL;
33310 } else {
33311     addr = get_unmapped_area(addr, len);

```

```

33312     if (!addr)
33313         return -ENOMEM;
33314 }
33315
33316 /* Determine the object being mapped and call the
33317  * appropriate specific mapper. the address has already
33318  * been validated, but not unmapped, but the maps are
33319  * removed from the list. */
33320 if (file && (!file->f_op || !file->f_op->mmap))
33321     return -ENODEV;
33322
33323 vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
33324 if (!vma)
33325     return -ENOMEM;
33326
33327 vma->vm_mm = mm;
33328 vma->vm_start = addr;
33329 vma->vm_end = addr + len;
33330 vma->vm_flags = vm_flags(prot, flags) | mm->def_flags;
33331
33332 if (file) {
33333     if (file->f_mode & 1)
33334         vma->vm_flags |= VM_MAYREAD|VM_MAYWRITE|VM_MAYEXEC;
33335     if (flags & MAP_SHARED) {
33336         vma->vm_flags |= VM_SHARED | VM_MAYSHARE;
33337
33338         /* This looks strange, but when we don't have the
33339          * file open for writing, we can demote the shared
33340          * mapping to a simpler private mapping. That also
33341          * takes care of a security hole with ptrace()
33342          * writing to a shared mapping without write
33343          * permissions.
33344          *
33345          * We leave the VM_MAYSHARE bit on, just to get
33346          * correct output from /proc/xxx/maps.. */
33347         if (!(file->f_mode & 2))
33348             vma->vm_flags &= ~(VM_MAYWRITE | VM_SHARED);
33349     }
33350 } else
33351     vma->vm_flags |= VM_MAYREAD|VM_MAYWRITE|VM_MAYEXEC;
33352 vma->vm_page_prot =
33353     protection_map[vma->vm_flags & 0x0f];
33354 vma->vm_ops = NULL;
33355 vma->vm_offset = 0;
33356 vma->vm_file = NULL;
33357 vma->vm_pte = 0;
33358
33359 /* Clear old maps */

```

```

33360 error = -ENOMEM;
33361 if (do_munmap(addr, len))
33362     goto free_vma;
33363
33364 /* Check against address space limit. */
33365 if ((mm->total_vm << PAGE_SHIFT) + len
33366     > current->rlim[RLIMIT_AS].rlim_cur)
33367     goto free_vma;
33368
33369 /* Private writable mapping? Check memory
33370  * availability.. */
33371 if ((vma->vm_flags & (VM_SHARED | VM_WRITE)) ==
33372     VM_WRITE &&
33373     !(flags & MAP_NORESERVE) &&
33374     !vm_enough_memory(len >> PAGE_SHIFT))
33375     goto free_vma;
33376
33377 if (file) {
33378     int correct_wcount = 0;
33379     if (vma->vm_flags & VM_DENYWRITE) {
33380         if (file->f_dentry->d_inode->i_writecount > 0) {
33381             error = -ETXTBSY;
33382             goto free_vma;
33383         }
33384         /* f_op->mmap might possibly sleep
33385          * (generic_file_mmap doesn't, but other code
33386          * might). In any case, this takes care of any
33387          * race that this might cause.
33388          */
33389         file->f_dentry->d_inode->i_writecount--;
33390         correct_wcount = 1;
33391     }
33392     error = file->f_op->mmap(file, vma);
33393     /* Fix up the count if necessary, then check for an
33394      * error */
33395     if (correct_wcount)
33396         file->f_dentry->d_inode->i_writecount++;
33397     if (error)
33398         goto unmap_and_free_vma;
33399     vma->vm_file = file;
33400     file->f_count++;
33401 }
33402
33403 /* merge_segments may merge our vma, so we can't refer
33404  * to it after the call. Save the values we need now
33405  * ... */
33406 flags = vma->vm_flags;
33407 addr = vma->vm_start; /* can addr have changed?? */

```

```

33408 insert_vm_struct(mm, vma);
33409 merge_segments(mm, vma->vm_start, vma->vm_end);
33410
33411 mm->total_vm += len >> PAGE_SHIFT;
33412 if (flags & VM_LOCKED) {
33413     mm->locked_vm += len >> PAGE_SHIFT;
33414     make_pages_present(addr, addr + len);
33415 }
33416 return addr;
33417
33418 unmap_and_free_vma:
33419 /* Undo any partial mapping done by a device driver. */
33420 flush_cache_range(mm, vma->vm_start, vma->vm_end);
33421 zap_page_range(mm, vma->vm_start,
33422                vma->vm_end - vma->vm_start);
33423* flush_tlb_range(mm, vma->vm_start, vma->vm_end);
33424 free_vma:
33425 kmem_cache_free(vm_area_cachep, vma);
33426 return error;
33427 }
33428
33429 /* Get an address range which is currently unmapped. For
33430  * mmap() without MAP_FIXED and shmat() with addr=0.
33431  * Return value 0 means ENOMEM. */
33432 unsigned long get_unmapped_area(unsigned long addr,
33433                                unsigned long len)
33434 {
33435     struct vm_area_struct * vmm;
33436
33437     if (len > TASK_SIZE)
33438         return 0;
33439     if (!addr)
33440         addr = TASK_UNMAPPED_BASE;
33441     addr = PAGE_ALIGN(addr);
33442
33443     for (vmm = find_vma(current->mm, addr); ;
33444          vmm = vmm->vm_next) {
33445         /* At this point: (!vmm || addr < vmm->vm_end). */
33446         if (TASK_SIZE - len < addr)
33447             return 0;
33448         if (!vmm || addr + len <= vmm->vm_start)
33449             return addr;
33450         addr = vmm->vm_end;
33451     }
33452 }
33453
33454 #define vm_avl_empty (struct vm_area_struct *) NULL
33455

```

```

33456 #include "mmap_avl.c"
33457
33458 /* Look up the first VMA which satisfies addr < vm_end,
33459  * NULL if none. */
33460 struct vm_area_struct * find_vma(struct mm_struct * mm,
33461                                unsigned long addr)
33462 {
33463     struct vm_area_struct *vma = NULL;
33464
33465     if (mm) {
33466         /* Check the cache first. */
33467         /* (Cache hit rate is typically around 35%. */
33468         vma = mm->mmap_cache;
33469         if (!(vma && vma->vm_end > addr &&
33470             vma->vm_start <= addr)) {
33471             if (!mm->mmap_avl) {
33472                 /* Go through the linear list. */
33473                 vma = mm->mmap;
33474                 while (vma && vma->vm_end <= addr)
33475                     vma = vma->vm_next;
33476             } else {
33477                 /* Then go through the AVL tree quickly. */
33478                 struct vm_area_struct * tree = mm->mmap_avl;
33479                 vma = NULL;
33480                 for (;;) {
33481                     if (tree == vm_avl_empty)
33482                         break;
33483                     if (tree->vm_end > addr) {
33484                         vma = tree;
33485                         if (tree->vm_start <= addr)
33486                             break;
33487                         tree = tree->vm_avl_left;
33488                     } else
33489                         tree = tree->vm_avl_right;
33490                 }
33491             }
33492             if (vma)
33493                 mm->mmap_cache = vma;
33494         }
33495     }
33496     return vma;
33497 }
33498
33499 /* Same as find_vma, but also return a pointer to the
33500  * previous VMA in *pprev. */
33501 struct vm_area_struct * find_vma_prev(
33502     struct mm_struct * mm, unsigned long addr,
33503     struct vm_area_struct **pprev)

```

```

33504 {
33505     if (mm) {
33506         if (!mm->mmap_avl) {
33507             /* Go through the linear list. */
33508             struct vm_area_struct * prev = NULL;
33509             struct vm_area_struct * vma = mm->mmap;
33510             while (vma && vma->vm_end <= addr) {
33511                 prev = vma;
33512                 vma = vma->vm_next;
33513             }
33514             *pprev = prev;
33515             return vma;
33516         } else {
33517             /* Go through the AVL tree quickly. */
33518             struct vm_area_struct * vma = NULL;
33519             struct vm_area_struct * last_turn_right = NULL;
33520             struct vm_area_struct * prev = NULL;
33521             struct vm_area_struct * tree = mm->mmap_avl;
33522             for (;;) {
33523                 if (tree == vm_avl_empty)
33524                     break;
33525                 if (tree->vm_end > addr) {
33526                     vma = tree;
33527                     prev = last_turn_right;
33528                     if (tree->vm_start <= addr)
33529                         break;
33530                     tree = tree->vm_avl_left;
33531                 } else {
33532                     last_turn_right = tree;
33533                     tree = tree->vm_avl_right;
33534                 }
33535             }
33536             if (vma) {
33537                 if (vma->vm_avl_left != vm_avl_empty) {
33538                     prev = vma->vm_avl_left;
33539                     while (prev->vm_avl_right != vm_avl_empty)
33540                         prev = prev->vm_avl_right;
33541                 }
33542                 if ((prev ? prev->vm_next : mm->mmap) != vma)
33543                     printk("find_vma_prev: tree inconsistent with
33544                             "list\n");
33545                 *pprev = prev;
33546                 return vma;
33547             }
33548         }
33549     }
33550     *pprev = NULL;
33551     return NULL;

```

```

33552 }
33553
33554 /* Normal function to fix up a mapping
33555 * This function is the default for when an area has no
33556 * specific function. This may be used as part of a more
33557 * specific routine. This function works out what part
33558 * of an area is affected and adjusts the mapping
33559 * information. Since the actual page manipulation is
33560 * done in do_mmap(), none need be done here, though it
33561 * would probably be more appropriate.
33562 *
33563 * By the time this function is called, the area struct
33564 * has been removed from the process mapping list, so it
33565 * needs to be reinserted if necessary.
33566 *
33567 * The 4 main cases are:
33568 *   Unmapping the whole area
33569 *   Unmapping from the start of the seg to a point in it
33570 *   Unmapping from an intermediate point to the end
33571 *   Unmapping between to intermediate points, making a
33572 *   hole.
33573 *
33574 * Case 4 involves the creation of 2 new areas, for each
33575 * side of the hole. If possible, we reuse the existing
33576 * area rather than allocate a new one, and the return
33577 * indicates whether the old area was reused. */
33578 static int unmap_fixup(struct vm_area_struct *area,
33579 unsigned long addr, size_t len,
33580 struct vm_area_struct **extra)
33581 {
33582     struct vm_area_struct *mpnt;
33583     unsigned long end = addr + len;
33584
33585     area->vm_mm->total_vm -= len >> PAGE_SHIFT;
33586     if (area->vm_flags & VM_LOCKED)
33587         area->vm_mm->locked_vm -= len >> PAGE_SHIFT;
33588
33589     /* Unmapping the whole area. */
33590     if (addr == area->vm_start && end == area->vm_end) {
33591         if (area->vm_ops && area->vm_ops->close)
33592             area->vm_ops->close(area);
33593         if (area->vm_file)
33594             fput(area->vm_file);
33595         return 0;
33596     }
33597
33598     /* Work out to one of the ends. */
33599     if (end == area->vm_end)
33600         area->vm_end = addr;
33601     else if (addr == area->vm_start) {
33602         area->vm_offset += (end - area->vm_start);
33603         area->vm_start = end;
33604     } else {
33605         /* Unmapping a hole:
33606          * area->vm_start < addr <= end < area->vm_end */
33607         /* Add end mapping -- leave beginning for below */
33608         mpnt = *extra;
33609         *extra = NULL;
33610
33611         mpnt->vm_mm = area->vm_mm;
33612         mpnt->vm_start = end;
33613         mpnt->vm_end = area->vm_end;
33614         mpnt->vm_page_prot = area->vm_page_prot;
33615         mpnt->vm_flags = area->vm_flags;
33616         mpnt->vm_ops = area->vm_ops;
33617         mpnt->vm_offset =
33618             area->vm_offset + (end - area->vm_start);
33619         mpnt->vm_file = area->vm_file;
33620         mpnt->vm_pte = area->vm_pte;
33621         if (mpnt->vm_file)
33622             mpnt->vm_file->f_count++;
33623         if (mpnt->vm_ops && mpnt->vm_ops->open)
33624             mpnt->vm_ops->open(mpnt);
33625         area->vm_end = addr; /* Truncate area */
33626         insert_vm_struct(current->mm, mpnt);
33627     }
33628
33629     insert_vm_struct(current->mm, area);
33630     return 1;
33631 }
33632
33633 /* Try to free as many page directory entries as we can,
33634 * without having to work very hard at actually scanning
33635 * the page tables themselves.
33636 *
33637 * Right now we try to free page tables if we have a nice
33638 * PGDIR-aligned area that got free'd up. We could be
33639 * more granular if we want to, but this is fast and
33640 * simple, and covers the bad cases.
33641 *
33642 * "prev", if it exists, points to a vma before the one
33643 * we just free'd - but there's no telling how much
33644 * before. */
33645 static void free_pgtables(struct mm_struct * mm,
33646 struct vm_area_struct *prev,
33647 unsigned long start, unsigned long end)

```

```

33648 {
33649     unsigned long first = start & PGDIR_MASK;
33650     unsigned long last = (end + PGDIR_SIZE - 1) &
33651         PGDIR_MASK;
33652
33653     if (!prev) {
33654         prev = mm->mmap;
33655         if (!prev)
33656             goto no_mmmaps;
33657         if (prev->vm_end > start) {
33658             if (last > prev->vm_end)
33659                 last = prev->vm_end;
33660             goto no_mmmaps;
33661         }
33662     }
33663     for (;;) {
33664         struct vm_area_struct *next = prev->vm_next;
33665
33666         if (next) {
33667             if (next->vm_start < start) {
33668                 prev = next;
33669                 continue;
33670             }
33671             if (last > next->vm_start)
33672                 last = next->vm_start;
33673         }
33674         if (prev->vm_end > first)
33675             first = prev->vm_end + PGDIR_SIZE - 1;
33676         break;
33677     }
33678 no_mmmaps:
33679     first = first >> PGDIR_SHIFT;
33680     last = last >> PGDIR_SHIFT;
33681     if (last > first)
33682         clear_page_tables(mm, first, last-first);
33683 }
33684
33685 /* Munmap is split into 2 main parts -- this part which
33686 * finds what needs doing, and the areas themselves,
33687 * which do the work. This now handles partial
33688 * unmappings. Jeremy Fitzhardine <jeremy@sw.oz.au> */
33689 int do_munmap(unsigned long addr, size_t len)
33690 {
33691     struct mm_struct * mm;
33692     struct vm_area_struct *mpnt, *prev, **npp, *free,
33693         *extra;
33694
33695     if ((addr & ~PAGE_MASK) || addr > TASK_SIZE ||

```

→ 558

```

33696     len > TASK_SIZE-addr)
33697         return -EINVAL;
33698
33699     if ((len = PAGE_ALIGN(len)) == 0)
33700         return -EINVAL;
33701
33702     /* Check if this memory area is ok - put it on the
33703     * temporary list if so.. The checks here are pretty
33704     * simple -- every area affected in some way (by any
33705     * overlap) is put on the list. If nothing is put on,
33706     * nothing is affected. */
33707     mm = current->mm;
33708     mpnt = find_vma_prev(mm, addr, &prev);
33709     if (!mpnt)
33710         return 0;
33711     /* we have addr < mpnt->vm_end */
33712
33713     if (mpnt->vm_start >= addr+len)
33714         return 0;
33715
33716     /* If we'll make "hole", check the vm areas limit */
33717     if ((mpnt->vm_start < addr && mpnt->vm_end > addr+len)
33718         && mm->map_count >= MAX_MAP_COUNT)
33719         return -ENOMEM;
33720
33721     /* We may need one additional vma to fix up the
33722     * mappings ... and this is the last chance for an
33723     * easy error exit. */
33724     extra = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
33725     if (!extra)
33726         return -ENOMEM;
33727
33728     npp = (prev ? &prev->vm_next : &mm->mmap);
33729     free = NULL;
33730     for (; mpnt && mpnt->vm_start < addr+len; mpnt = *npp){
33731         *npp = mpnt->vm_next;
33732         mpnt->vm_next = free;
33733         free = mpnt;
33734         if (mm->mmap_avl)
33735             avl_remove(mpnt, &mm->mmap_avl);
33736     }
33737
33738     /* Ok - we have the memory areas we should free on the
33739     * 'free' list, so release them, and unmap the page
33740     * range.. If the one of the segments is only being
33741     * partially unmapped, it will put new
33742     * vm_area_struct(s) into the address space. */
33743     while ((mpnt = free) != NULL) {

```

```

33744 unsigned long st, end, size;
33745
33746 free = free->vm_next;
33747
33748 st = addr < mpnt->vm_start ? mpnt->vm_start : addr;
33749 end = addr+len;
33750 end = end > mpnt->vm_end ? mpnt->vm_end : end;
33751 size = end - st;
33752
33753 if (mpnt->vm_ops && mpnt->vm_ops->unmap)
33754     mpnt->vm_ops->unmap(mpnt, st, size);
33755
33756 remove_shared_vm_struct(mpnt);
33757 mm->map_count--;
33758
33759 flush_cache_range(mm, st, end);
33760 zap_page_range(mm, st, size);
33761 flush_tlb_range(mm, st, end);
33762
33763 /* Fix the mapping, and free the old area if it
33764    * wasn't reused. */
33765 if (!unmap_fixup(mpnt, st, size, &extra))
33766     kmem_cache_free(vm_area_cachep, mpnt);
33767 }
33768
33769 /* Release the extra vma struct if it wasn't used */
33770 if (extra)
33771     kmem_cache_free(vm_area_cachep, extra);
33772
33773 free_pgtables(mm, prev, addr, addr+len);
33774
33775 mm->mmap_cache = NULL; /* Kill the cache. */
33776 return 0;
33777 }
33778
33779 asmlinkage int sys_munmap(unsigned long addr, size_t len)
33780 {
33781     int ret;
33782
33783     down(&current->mm->mmap_sem);
33784     lock_kernel();
33785     ret = do_munmap(addr, len);
33786     unlock_kernel();
33787     up(&current->mm->mmap_sem);
33788     return ret;
33789 }
33790
33791 /* Build the AVL tree corresponding to the VMA list. */
33792 void build_mmap_avl(struct mm_struct * mm)
33793 {
33794     struct vm_area_struct * vma;
33795
33796     mm->mmap_avl = NULL;
33797     for (vma = mm->mmap; vma; vma = vma->vm_next)
33798         avl_insert(vma, &mm->mmap_avl);
33799 }
33800
33801 /* Release all mmmaps. */
33802 void exit_mmap(struct mm_struct * mm)
33803 {
33804     struct vm_area_struct * mpnt;
33805
33806     mpnt = mm->mmap;
33807     mm->mmap = mm->mmap_avl = mm->mmap_cache = NULL;
33808     mm->rss = 0;
33809     mm->total_vm = 0;
33810     mm->locked_vm = 0;
33811     while (mpnt) {
33812         struct vm_area_struct * next = mpnt->vm_next;
33813         unsigned long start = mpnt->vm_start;
33814         unsigned long end = mpnt->vm_end;
33815         unsigned long size = end - start;
33816
33817         if (mpnt->vm_ops) {
33818             if (mpnt->vm_ops->unmap)
33819                 mpnt->vm_ops->unmap(mpnt, start, size);
33820             if (mpnt->vm_ops->close)
33821                 mpnt->vm_ops->close(mpnt);
33822         }
33823         mm->map_count--;
33824         remove_shared_vm_struct(mpnt);
33825         zap_page_range(mm, start, size);
33826         if (mpnt->vm_file)
33827             fput(mpnt->vm_file);
33828         kmem_cache_free(vm_area_cachep, mpnt);
33829         mpnt = next;
33830     }
33831
33832     /* This is just debugging */
33833     if (mm->map_count)
33834         printk("exit_mmap: map count is %d\n",
33835             mm->map_count);
33836
33837     clear_page_tables(mm, 0, USER_PTRS_PER_PGD);
33838 }
33839

```



```

33840 /* Insert vm structure into process list sorted by
33841 * address and into the inode's i_mmap ring. */
33842 void insert_vm_struct(struct mm_struct *mm,
33843                      struct vm_area_struct *vmp)
33844 {
33845     struct vm_area_struct **pprev;
33846     struct file * file;
33847
33848     if (!mm->mmap_avl) {
33849         pprev = &mm->mmap;
33850         while (*pprev && (*pprev)->vm_start <= vmp->vm_start)
33851             pprev = &(*pprev)->vm_next;
33852     } else {
33853         struct vm_area_struct *prev, *next;
33854         avl_insert_neighbours(vmp, &mm->mmap_avl,
33855                             &prev, &next);
33856         pprev = (prev ? &prev->vm_next : &mm->mmap);
33857         if (*pprev != next)
33858             printk("insert_vm_struct: tree inconsistent with "
33859                   "list\n");
33860     }
33861     vmp->vm_next = *pprev;
33862     *pprev = vmp;
33863
33864     mm->map_count++;
33865     if (mm->map_count >= AVL_MIN_MAP_COUNT &&
33866         !mm->mmap_avl)
33867         build_mmap_avl(mm);
33868
33869     file = vmp->vm_file;
33870     if (file) {
33871         struct inode * inode = file->f_dentry->d_inode;
33872         if (vmp->vm_flags & VM_DENYWRITE)
33873             inode->i_writecount--;
33874
33875         /* insert vmp into inode's share list */
33876         if((vmp->vm_next_share = inode->i_mmap) != NULL)
33877             inode->i_mmap->vm_pprev_share =
33878                 &vmp->vm_next_share;
33879         inode->i_mmap = vmp;
33880         vmp->vm_pprev_share = &inode->i_mmap;
33881     }
33882 }
33883
33884 /* Merge the list of memory segments if possible.
33885 * Redundant vm_area_structs are freed. This assumes
33886 * that the list is ordered by address. We don't need to
33887 * traverse the entire list, only those segments which

```

→ p 558

```

33888 * intersect or are adjacent to a given interval.
33889 *
33890 * We must already hold the mm semaphore when we get
33891 * here.. */
33892 void merge_segments (struct mm_struct * mm,
33893                    unsigned long start_addr, unsigned long end_addr)
33894 {
33895     struct vm_area_struct *prev, *mpnt, *next, *prev1;
33896
33897     mpnt = find_vma_prev(mm, start_addr, &prev1);
33898     if (!mpnt)
33899         return;
33900
33901     if (prev1) {
33902         prev = prev1;
33903     } else {
33904         prev = mpnt;
33905         mpnt = mpnt->vm_next;
33906     }
33907
33908     /* prev and mpnt cycle through the list, as long as
33909     * start_addr < mpnt->vm_end &&
33910     * prev->vm_start < end_addr */
33911     for ( ; mpnt && prev->vm_start < end_addr;
33912          prev = mpnt, mpnt = next) {
33913         next = mpnt->vm_next;
33914
33915         /* To share, we must have the same file,
33916         * operations.. */
33917         if ((mpnt->vm_file != prev->vm_file)||
33918             (mpnt->vm_pte != prev->vm_pte) ||
33919             (mpnt->vm_ops != prev->vm_ops) ||
33920             (mpnt->vm_flags != prev->vm_flags) ||
33921             (prev->vm_end != mpnt->vm_start))
33922             continue;
33923
33924         /* If we have a file or it's a shared memory area the
33925         * offsets must be contiguous.. */
33926         if ((mpnt->vm_file != NULL) ||
33927             (mpnt->vm_flags & VM_SHM)) {
33928             unsigned long off =
33929                 prev->vm_offset+prev->vm_end-prev->vm_start;
33930             if (off != mpnt->vm_offset)
33931                 continue;
33932         }
33933
33934         /* merge prev with mpnt and set up pointers so the
33935         * new big segment can possibly merge with the next

```

```

33936      * one. The old unused mpnt is freed. */
33937      if (mm->mmap_avl)
33938          avl_remove(mpnt, &mm->mmap_avl);
33939      prev->vm_end = mpnt->vm_end;
33940      prev->vm_next = mpnt->vm_next;
33941      if (mpnt->vm_ops && mpnt->vm_ops->close) {
33942          mpnt->vm_offset += mpnt->vm_end - mpnt->vm_start;
33943          mpnt->vm_start = mpnt->vm_end;
33944          mpnt->vm_ops->close(mpnt);
33945      }
33946      mm->map_count--;
33947      remove_shared_vm_struct(mpnt);
33948      if (mpnt->vm_file)
33949          fput(mpnt->vm_file);
33950      kmem_cache_free(vm_area_cachep, mpnt);
33951      mpnt = prev;
33952  }
33953  mm->mmap_cache = NULL;          /* Kill the cache. */
33954  }
33955
33956  void __init vma_init(void)
33957  {
33958      vm_area_cachep = kmem_cache_create("vm_area_struct",
33959          sizeof(struct vm_area_struct),
33960          0, SLAB_HWCACHE_ALIGN,
33961          NULL, NULL);
33962      if(!vm_area_cachep)
33963          panic("vma_init: Cannot alloc vm_area_struct cache.");
33964
33965      mm_cachep = kmem_cache_create("mm_struct",
33966          sizeof(struct mm_struct),
33967          0, SLAB_HWCACHE_ALIGN,
33968          NULL, NULL);
33969      if(!mm_cachep)
33970          panic("vma_init: Cannot alloc mm_struct cache.");
33971  }

```

mm/mprotect.c

```

33972  /*
33973   *      linux/mm/mprotect.c
33974   *
33975   *      (C) Copyright 1994 Linus Torvalds
33976   */
33977  #include <linux/slab.h>
33978  #include <linux/smp_lock.h>
33979  #include <linux/shm.h>
33980  #include <linux/mman.h>

```

```

33981
33982  #include <asm/uaccess.h>
33983  #include <asm/pgtable.h>
33984
33985  static inline void change_pte_range(pmd_t * pmd,
33986      unsigned long address, unsigned long size,
33987      pgprot_t newprot)
33988  {
33989      pte_t * pte;
33990      unsigned long end;
33991
33992      if (pmd_none(*pmd))
33993          return;
33994      if (pmd_bad(*pmd)) {
33995          printk("change_pte_range: bad pmd (%08lx)\n",
33996              pmd_val(*pmd));
33997          pmd_clear(pmd);
33998          return;
33999      }
34000      pte = pte_offset(pmd, address);
34001      address &= ~PMD_MASK;
34002      end = address + size;
34003      if (end > PMD_SIZE)
34004          end = PMD_SIZE;
34005      do {
34006          pte_t entry = *pte;
34007          if (pte_present(entry))
34008              set_pte(pte, pte_modify(entry, newprot));
34009          address += PAGE_SIZE;
34010          pte++;
34011      } while (address < end);
34012  }
34013
34014  static inline void change_pmd_range(pgd_t * pgd,
34015      unsigned long address, unsigned long size,
34016      pgprot_t newprot)
34017  {
34018      pmd_t * pmd;
34019      unsigned long end;
34020
34021      if (pgd_none(*pgd))
34022          return;
34023      if (pgd_bad(*pgd)) {
34024          printk("change_pmd_range: bad pgd (%08lx)\n",
34025              pgd_val(*pgd));
34026          pgd_clear(pgd);
34027          return;
34028      }

```

```

34029 pmd = pmd_offset(pgd, address);
34030 address &= ~PGDIR_MASK;
34031 end = address + size;
34032 if (end > PGDIR_SIZE)
34033     end = PGDIR_SIZE;
34034 do {
34035     change_pte_range(pmd, address, end - address,
34036                     newprot);
34037     address = (address + PMD_SIZE) & PMD_MASK;
34038     pmd++;
34039 } while (address < end);
34040 }
34041
34042 static void change_protection(unsigned long start,
34043 unsigned long end, pgprot_t newprot)
34044 {
34045     pgd_t *dir;
34046     unsigned long beg = start;
34047
34048     dir = pgd_offset(current->mm, start);
34049     flush_cache_range(current->mm, beg, end);
34050     while (start < end) {
34051         change_pmd_range(dir, start, end - start, newprot);
34052         start = (start + PGDIR_SIZE) & PGDIR_MASK;
34053         dir++;
34054     }
34055     flush_tlb_range(current->mm, beg, end);
34056     return;
34057 }
34058
34059 static inline int mprotect_fixup_all(
34060 struct vm_area_struct * vma,
34061 int newflags, pgprot_t prot)
34062 {
34063     vma->vm_flags = newflags;
34064     vma->vm_page_prot = prot;
34065     return 0;
34066 }
34067
34068 static inline int mprotect_fixup_start(
34069 struct vm_area_struct * vma,
34070 unsigned long end, int newflags, pgprot_t prot)
34071 {
34072     struct vm_area_struct * n;
34073
34074     n = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
34075     if (!n)
34076         return -ENOMEM;

```

```

34077     *n = *vma;
34078     vma->vm_start = end;
34079     n->vm_end = end;
34080     vma->vm_offset += vma->vm_start - n->vm_start;
34081     n->vm_flags = newflags;
34082     n->vm_page_prot = prot;
34083     if (n->vm_file)
34084         n->vm_file->f_count++;
34085     if (n->vm_ops && n->vm_ops->open)
34086         n->vm_ops->open(n);
34087     insert_vm_struct(current->mm, n);
34088     return 0;
34089 }
34090
34091 static inline int mprotect_fixup_end(
34092 struct vm_area_struct * vma,
34093 unsigned long start,
34094 int newflags, pgprot_t prot)
34095 {
34096     struct vm_area_struct * n;
34097
34098     n = kmem_cache_alloc(vm_area_cachep, GFP_KERNEL);
34099     if (!n)
34100         return -ENOMEM;
34101     *n = *vma;
34102     vma->vm_end = start;
34103     n->vm_start = start;
34104     n->vm_offset += n->vm_start - vma->vm_start;
34105     n->vm_flags = newflags;
34106     n->vm_page_prot = prot;
34107     if (n->vm_file)
34108         n->vm_file->f_count++;
34109     if (n->vm_ops && n->vm_ops->open)
34110         n->vm_ops->open(n);
34111     insert_vm_struct(current->mm, n);
34112     return 0;
34113 }
34114
34115 static inline int mprotect_fixup_middle(
34116 struct vm_area_struct * vma,
34117 unsigned long start, unsigned long end,
34118 int newflags, pgprot_t prot)
34119 {
34120     struct vm_area_struct * left, * right;
34121
34122     left = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
34123     if (!left)
34124         return -ENOMEM;

```

```

34125 right = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
34126 if (!right) {
34127     kmem_cache_free(vm_area_cachep, left);
34128     return -ENOMEM;
34129 }
34130 *left = *vma;
34131 *right = *vma;
34132 left->vm_end = start;
34133 vma->vm_start = start;
34134 vma->vm_end = end;
34135 right->vm_start = end;
34136 vma->vm_offset += vma->vm_start - left->vm_start;
34137 right->vm_offset += right->vm_start - left->vm_start;
34138 vma->vm_flags = newflags;
34139 vma->vm_page_prot = prot;
34140 if (vma->vm_file)
34141     vma->vm_file->f_count += 2;
34142 if (vma->vm_ops && vma->vm_ops->open) {
34143     vma->vm_ops->open(left);
34144     vma->vm_ops->open(right);
34145 }
34146 insert_vm_struct(current->mm, left);
34147 insert_vm_struct(current->mm, right);
34148 return 0;
34149 }
34150
34151 static int mprotect_fixup(struct vm_area_struct * vma,
34152 unsigned long start, unsigned long end,
34153 unsigned int newflags)
34154 {
34155     pgprot_t newprot;
34156     int error;
34157
34158     if (newflags == vma->vm_flags)
34159         return 0;
34160     newprot = protection_map[newflags & 0xf];
34161     if (start == vma->vm_start) {
34162         if (end == vma->vm_end)
34163             error = mprotect_fixup_all(vma, newflags, newprot);
34164         else
34165             error = mprotect_fixup_start(vma, end, newflags,
34166 newprot);
34167     } else if (end == vma->vm_end)
34168         error = mprotect_fixup_end(vma, start, newflags,
34169 newprot);
34170     else
34171         error = mprotect_fixup_middle(vma, start, end,
34172 newflags, newprot);

```

```

34173
34174     if (error)
34175         return error;
34176
34177     change_protection(start, end, newprot);
34178     return 0;
34179 }
34180
34181 asmlinkage int sys_mprotect(unsigned long start,
34182 size_t len, unsigned long prot)
34183 {
34184     unsigned long nstart, end, tmp;
34185     struct vm_area_struct * vma, * next;
34186     int error = -EINVAL;
34187
34188     if (start & ~PAGE_MASK)
34189         return -EINVAL;
34190     len = (len + ~PAGE_MASK) & PAGE_MASK;
34191     end = start + len;
34192     if (end < start)
34193         return -EINVAL;
34194     if (prot & ~(PROT_READ | PROT_WRITE | PROT_EXEC))
34195         return -EINVAL;
34196     if (end == start)
34197         return 0;
34198
34199     down(&current->mm->mmap_sem);
34200     lock_kernel();
34201
34202     vma = find_vma(current->mm, start);
34203     error = -EFAULT;
34204     if (!vma || vma->vm_start > start)
34205         goto out;
34206
34207     for (nstart = start ; ; ) {
34208         unsigned int newflags;
34209
34210         /* Here we know that
34211          * vma->vm_start <= nstart < vma->vm_end. */
34212
34213         newflags =
34214             prot | (vma->vm_flags &
34215 ~ (PROT_READ | PROT_WRITE | PROT_EXEC));
34216         if ((newflags & ~(newflags >> 4)) & 0xf) {
34217             error = -EACCES;
34218             break;
34219         }
34220

```

```

34221     if (vma->vm_end >= end) {
34222         error = mprotect_fixup(vma, nstart, end, newflags);
34223         break;
34224     }
34225
34226     tmp = vma->vm_end;
34227     next = vma->vm_next;
34228     error = mprotect_fixup(vma, nstart, tmp, newflags);
34229     if (error)
34230         break;
34231     nstart = tmp;
34232     vma = next;
34233     if (!vma || vma->vm_start != nstart) {
34234         error = -EFAULT;
34235         break;
34236     }
34237 }
34238 merge_segments(current->mm, start, end);
34239 out:
34240 unlock_kernel();
34241 up(&current->mm->mmap_sem);
34242 return error;
34243 }

```

mm/mremap.c

```

34244 /*
34245  *    linux/mm/remap.c
34246  *
34247  *    (C) Copyright 1996 Linus Torvalds
34248  */
34249
34250 #include <linux/slab.h>
34251 #include <linux/smp_lock.h>
34252 #include <linux/shm.h>
34253 #include <linux/mman.h>
34254 #include <linux/swap.h>
34255
34256 #include <asm/uaccess.h>
34257 #include <asm/pgtable.h>
34258
34259 extern int vm_enough_memory(long pages);
34260
34261 static inline pte_t *get_one_pte(struct mm_struct *mm,
34262                                 unsigned long addr)
34263 {
34264     pgd_t *pgd;
34265     pmd_t *pmd;

```

```

34266     pte_t *pte = NULL;
34267
34268     pgd = pgd_offset(mm, addr);
34269     if (pgd_none(*pgd))
34270         goto end;
34271     if (pgd_bad(*pgd)) {
34272         printk("move_one_page: bad source pgd (%08lx)\n",
34273             pgd_val(*pgd));
34274         pgd_clear(pgd);
34275         goto end;
34276     }
34277
34278     pmd = pmd_offset(pgd, addr);
34279     if (pmd_none(*pmd))
34280         goto end;
34281     if (pmd_bad(*pmd)) {
34282         printk("move_one_page: bad source pmd (%08lx)\n",
34283             pmd_val(*pmd));
34284         pmd_clear(pmd);
34285         goto end;
34286     }
34287
34288     pte = pte_offset(pmd, addr);
34289     if (pte_none(*pte))
34290         pte = NULL;
34291 end:
34292     return pte;
34293 }
34294
34295 static inline pte_t *alloc_one_pte(struct mm_struct *mm,
34296                                   unsigned long addr)
34297 {
34298     pmd_t *pmd;
34299     pte_t *pte = NULL;
34300
34301     pmd = pmd_alloc(pgd_offset(mm, addr), addr);
34302     if (pmd)
34303         pte = pte_alloc(pmd, addr);
34304     return pte;
34305 }
34306
34307 static inline int copy_one_pte(pte_t *src, pte_t *dst)
34308 {
34309     int error = 0;
34310     pte_t pte = *src;
34311
34312     if (!pte_none(pte)) {
34313         error++;

```

```

34314     if (dst) {
34315         pte_clear(src);
34316         set_pte(dst, pte);
34317         error--;
34318     }
34319 }
34320 return error;
34321 }
34322
34323 static int move_one_page(struct mm_struct *mm,
34324 unsigned long old_addr, unsigned long new_addr)
34325 {
34326     int error = 0;
34327     pte_t * src;
34328
34329     src = get_one_pte(mm, old_addr);
34330     if (src)
34331         error = copy_one_pte(src,
34332                               alloc_one_pte(mm, new_addr));
34333     return error;
34334 }
34335
34336 static int move_page_tables(struct mm_struct * mm,
34337 unsigned long new_addr, unsigned long old_addr,
34338 unsigned long len)
34339 {
34340     unsigned long offset = len;
34341
34342     flush_cache_range(mm, old_addr, old_addr + len);
34343     flush_tlb_range(mm, old_addr, old_addr + len);
34344
34345     /* This is not the clever way to do this, but we're
34346      * taking the easy way out on the assumption that most
34347      * remappings will be only a few pages.. This also
34348      * makes error recovery easier. */
34349     while (offset) {
34350         offset -= PAGE_SIZE;
34351         if (move_one_page(mm, old_addr + offset,
34352                           new_addr + offset))
34353             goto oops_we_failed;
34354     }
34355     return 0;
34356
34357     /* Ok, the move failed because we didn't have enough
34358      * pages for the new page table tree. This is unlikely,
34359      * but we have to take the possibility into account. In
34360      * that case we just move all the pages back (this will
34361      * work, because we still have the old page tables) */

```

```

34362 oops_we_failed:
34363     flush_cache_range(mm, new_addr, new_addr + len);
34364     while ((offset += PAGE_SIZE) < len)
34365         move_one_page(mm, new_addr + offset,
34366                       old_addr + offset);
34367     zap_page_range(mm, new_addr, new_addr + len);
34368     flush_tlb_range(mm, new_addr, new_addr + len);
34369     return 1;
34370 }
34371
34372 static inline unsigned long move_vma(
34373 struct vm_area_struct * vma, unsigned long addr,
34374 unsigned long old_len, unsigned long new_len)
34375 {
34376     struct vm_area_struct * new_vma;
34377
34378     new_vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
34379     if (new_vma) {
34380         unsigned long new_addr = get_unmapped_area(addr,
34381                                                       new_len);
34382
34383         if (new_addr &&
34384             !move_page_tables(current->mm, new_addr, addr,
34385                               old_len)) {
34386             *new_vma = *vma;
34387             new_vma->vm_start = new_addr;
34388             new_vma->vm_end = new_addr+new_len;
34389             new_vma->vm_offset =
34390                 vma->vm_offset + (addr - vma->vm_start);
34391             if (new_vma->vm_file)
34392                 new_vma->vm_file->f_count++;
34393             if (new_vma->vm_ops && new_vma->vm_ops->open)
34394                 new_vma->vm_ops->open(new_vma);
34395             insert_vm_struct(current->mm, new_vma);
34396             merge_segments(current->mm, new_vma->vm_start,
34397                            new_vma->vm_end);
34398             do_munmap(addr, old_len);
34399             current->mm->total_vm += new_len >> PAGE_SHIFT;
34400             if (new_vma->vm_flags & VM_LOCKED) {
34401                 current->mm->locked_vm += new_len >> PAGE_SHIFT;
34402                 make_pages_present(new_vma->vm_start,
34403                                    new_vma->vm_end);
34404             }
34405             return new_addr;
34406         }
34407         kmem_cache_free(vm_area_cachep, new_vma);
34408     }
34409     return -ENOMEM;

```

```

34410 )
34411
34412 /* Expand (or shrink) an existing mapping, potentially
34413  * moving it at the same time (controlled by the
34414  * MREMAP_MAYMOVE flag and available VM space) */
34415 asmlinkage unsigned long sys_mremap(unsigned long addr,
34416   unsigned long old_len, unsigned long new_len,
34417   unsigned long flags)
34418 {
34419   struct vm_area_struct *vma;
34420   unsigned long ret = -EINVAL;
34421
34422   down(&current->mm->mmap_sem);
34423   lock_kernel();
34424   if (addr & ~PAGE_MASK)
34425     goto out;
34426   old_len = PAGE_ALIGN(old_len);
34427   new_len = PAGE_ALIGN(new_len);
34428
34429   /* Always allow a shrinking remap: that just unmaps the
34430    * unnecessary pages.. */
34431   ret = addr;
34432   if (old_len >= new_len) {
34433     do_munmap(addr+new_len, old_len - new_len);
34434     goto out;
34435   }
34436
34437   /* Ok, we need to grow.. */
34438   ret = -EFAULT;
34439   vma = find_vma(current->mm, addr);
34440   if (!vma || vma->vm_start > addr)
34441     goto out;
34442   /* We can't remap across vm area boundaries */
34443   if (old_len > vma->vm_end - addr)
34444     goto out;
34445   if (vma->vm_flags & VM_LOCKED) {
34446     unsigned long locked =
34447       current->mm->locked_vm << PAGE_SHIFT;
34448     locked += new_len - old_len;
34449     ret = -EAGAIN;
34450     if (locked > current->rlim[RLIMIT_MEMLOCK].rlim_cur)
34451       goto out;
34452   }
34453   ret = -ENOMEM;
34454   if ((current->mm->total_vm << PAGE_SHIFT) +
34455       (new_len - old_len)
34456       > current->rlim[RLIMIT_AS].rlim_cur)
34457     goto out;

```

```

34458   /* Private writable mapping? Check memory
34459    * availability.. */
34460   if ((vma->vm_flags & (VM_SHARED|VM_WRITE)) == VM_WRITE
34461       && !(flags & MAP_NORESERVE) &&
34462       !vm_enough_memory((new_len - old_len) >>
34463         PAGE_SHIFT))
34464     goto out;
34465
34466   /* old_len exactly to the end of the area.. */
34467   if (old_len == vma->vm_end - addr &&
34468       (old_len != new_len || !(flags & MREMAP_MAYMOVE))){
34469     unsigned long max_addr = TASK_SIZE;
34470     if (vma->vm_next)
34471       max_addr = vma->vm_next->vm_start;
34472     /* can we just expand the current mapping? */
34473     if (max_addr - addr >= new_len) {
34474       int pages = (new_len - old_len) >> PAGE_SHIFT;
34475       vma->vm_end = addr + new_len;
34476       current->mm->total_vm += pages;
34477       if (vma->vm_flags & VM_LOCKED) {
34478         current->mm->locked_vm += pages;
34479         make_pages_present(addr + old_len,
34480           addr + new_len);
34481       }
34482       ret = addr;
34483       goto out;
34484     }
34485   }
34486
34487   /* We weren't able to just expand or shrink the area,
34488    * we need to create a new one and move it.. */
34489   if (flags & MREMAP_MAYMOVE)
34490     ret = move_vma(vma, addr, old_len, new_len);
34491   else
34492     ret = -ENOMEM;
34493 out:
34494   unlock_kernel();
34495   up(&current->mm->mmap_sem);
34496   return ret;
34497 }

```

mm/page_alloc.c

```

34498 /*
34499  * linux/mm/page alloc.c
34500  *
34501  * Copyright (C) 1991, 1992, 1993, 1994 Linus Torvalds
34502  * Swap reorganised 29.12.95. Stephen Tweedie

```

```

34503 */
34504
34505 #include <linux/config.h>
34506 #include <linux/mm.h>
34507 #include <linux/kernel_stat.h>
34508 #include <linux/swap.h>
34509 #include <linux/swapctl.h>
34510 #include <linux/interrupt.h>
34511 #include <linux/init.h>
34512 #include <linux/pagemap.h>
34513
34514 #include <asm/dma.h>
34515 #include <asm/uaccess.h> /* for copy_to/from_user */
34516 #include <asm/pgtable.h>
34517
34518 int nr_swap_pages = 0;
34519 int nr_free_pages = 0;
34520
34521 /* Free area management
34522  *
34523  * The free_area_list arrays point to the queue heads of
34524  * the free areas of different sizes */
34525
34526 #if CONFIG_API1000
34527 /* the AP+ needs to allocate 8MB contiguous, aligned
34528  * chunks of ram for the ring buffers */
34529 #define NR_MEM_LISTS 12
34530 #else
34531 #define NR_MEM_LISTS 6
34532 #endif
34533
34534 /* The start of this MUST match the start of "struct
34535  * page" */
34536 struct free_area_struct {
34537     struct page *next;
34538     struct page *prev;
34539     unsigned int *map;
34540 };
34541
34542 #define memory_head(x) ((struct page *) (x))
34543
34544 static struct free_area_struct free_area[NR_MEM_LISTS];
34545
34546 static inline void init_mem_queue(
34547     struct free_area_struct * head)
34548 {
34549     head->next = memory_head(head);
34550     head->prev = memory_head(head);

```

```

34551 }
34552
34553 static inline void add_mem_queue(
34554     struct free_area_struct * head, struct page * entry)
34555 {
34556     struct page * next = head->next;
34557
34558     entry->prev = memory_head(head);
34559     entry->next = next;
34560     next->prev = entry;
34561     head->next = entry;
34562 }
34563
34564 static inline void remove_mem_queue(struct page * entry)
34565 {
34566     struct page * next = entry->next;
34567     struct page * prev = entry->prev;
34568     next->prev = prev;
34569     prev->next = next;
34570 }
34571
34572 /* Free_page() adds the page to the free lists. This is
34573  * optimized for fast normal cases (no error jumps taken
34574  * normally).
34575  *
34576  * The way to optimize jumps for gcc-2.2.2 is to:
34577  * - select the "normal" case and put it inside the
34578  *   if () { XXX }
34579  * - no else-statements if you can avoid them
34580  *
34581  * With the above two rules, you get a straight-line
34582  * execution path for the normal case, giving better
34583  * asm code. */
34584
34585 /* Buddy system. Hairy. You really aren't expected to
34586  * understand this
34587  *
34588  * Hint: -mask = 1+-mask */
34589 spinlock_t page_alloc_lock = SPIN_LOCK_UNLOCKED;
34590
34591 static inline void free_pages_ok(unsigned long map_nr,
34592                                 unsigned long order)
34593 {
34594     struct free_area_struct *area = free_area + order;
34595     unsigned long index = map_nr >> (1 + order);
34596     unsigned long mask = (~0UL) << order;
34597     unsigned long flags;
34598

```



```

34599 spin_lock_irqsave(&page_alloc_lock, flags);
34600
34601 #define list(x) (mem_map+(x))
34602
34603 map_nr &= mask;
34604 nr_free_pages -= mask;
34605 while (mask + (1 << (NR_MEM_LISTS-1))) {
34606     if (!test_and_change_bit(index, area->map))
34607         break;
34608     remove_mem_queue(list(map_nr ^ ~mask));
34609     mask <<= 1;
34610     area++;
34611     index >>= 1;
34612     map_nr &= mask;
34613 }
34614 add_mem_queue(area, list(map_nr));
34615
34616 #undef list
34617
34618 spin_unlock_irqrestore(&page_alloc_lock, flags);
34619 }
34620
34621 void __free_page(struct page *page)
34622 {
34623     if (!PageReserved(page) &&
34624         atomic_dec_and_test(&page->count)) {
34625         if (PageSwapCache(page))
34626             panic ("Freeing swap cache page");
34627         page->flags &= ~(1 << PG_referenced);
34628         free_pages_ok(page - mem_map, 0);
34629         return;
34630     }
34631 }
34632
34633 void free_pages(unsigned long addr, unsigned long order)
34634 {
34635     unsigned long map_nr = MAP_NR(addr);
34636
34637     if (map_nr < max_mapnr) {
34638         mem_map_t * map = mem_map + map_nr;
34639         if (PageReserved(map))
34640             return;
34641         if (atomic_dec_and_test(&map->count)) {
34642             if (PageSwapCache(map))
34643                 panic ("Freeing swap cache pages");
34644             map->flags &= ~(1 << PG_referenced);
34645             free_pages_ok(map_nr, order);
34646             return;

```

```

34647     }
34648 }
34649 }
34650
34651 /* Some ugly macros to speed up __get_free_pages().. */
34652 #define MARK_USED(index, order, area) \
34653     change_bit((index) >> (1+(order)), (area)->map)
34654 #define CAN_DMA(x) (PageDMA(x))
34655 #define ADDRESS(x) (PAGE_OFFSET + ((x) << PAGE_SHIFT))
34656 #define RMQUEUE(order, gfp_mask) \
34657 do { \
34658     struct free_area_struct * area = free_area+order; \
34659     unsigned long new_order = order; \
34660     do { \
34661         struct page *prev = memory_head(area), \
34662             *ret = prev->next; \
34663         while (memory_head(area) != ret) { \
34664             if (!(gfp_mask & __GFP_DMA) || CAN_DMA(ret)) { \
34665                 unsigned long map_nr; \
34666                 (prev->next = ret->next)->prev = prev; \
34667                 map_nr = ret - mem_map; \
34668                 MARK_USED(map_nr, new_order, area); \
34669                 nr_free_pages -= 1 << order; \
34670                 EXPAND(ret, map_nr, order, new_order, area); \
34671                 spin_unlock_irqrestore(&page_alloc_lock, flags); \
34672                 return ADDRESS(map_nr); \
34673             } \
34674             prev = ret; \
34675             ret = ret->next; \
34676         } \
34677         new_order++; area++; \
34678     } while (new_order < NR_MEM_LISTS); \
34679 } while (0)
34680
34681 #define EXPAND(map,index,low,high,area) \
34682 do { \
34683     unsigned long size = 1 << high; \
34684     while (high > low) { \
34685         area--; high--; size >>= 1; \
34686         add_mem_queue(area, map); \
34687         MARK_USED(index, high, area); \
34688         index += size; \
34689         map += size; \
34690     } \
34691     atomic_set(&map->count, 1); \
34692 } while (0)
34693
34694 int low_on_memory = 0;

```

```

34695
34696 unsigned long __get_free_pages(int gfp_mask,
34697                               unsigned long order)
34698 {
34699     unsigned long flags;
34700
34701     if (order >= NR_MEM_LISTS)
34702         goto nopage;
34703
34704 #ifdef ATOMIC_MEMORY_DEBUGGING
34705     if ((gfp_mask & __GFP_WAIT) && in_interrupt()) {
34706         static int count = 0;
34707         if (++count < 5) {
34708             printk("gfp called nonatomically from interrupt "
34709                  "%p\n", __builtin_return_address(0));
34710         }
34711         goto nopage;
34712     }
34713 #endif
34714
34715     /* If this is a recursive call, we'd better do our best
34716      * to just allocate things without further thought. */
34717     if (!(current->flags & PF_MEMALLOC)) {
34718         int freed;
34719
34720         if (nr_free_pages > freepages.min) {
34721             if (!low_on_memory)
34722                 goto ok_to_allocate;
34723             if (nr_free_pages >= freepages.high) {
34724                 low_on_memory = 0;
34725                 goto ok_to_allocate;
34726             }
34727         }
34728
34729         low_on_memory = 1;
34730         current->flags |= PF_MEMALLOC;
34731         freed = try_to_free_pages(gfp_mask);
34732         current->flags &= ~PF_MEMALLOC;
34733
34734         if (!freed && !(gfp_mask & (__GFP_MED | __GFP_HIGH)))
34735             goto nopage;
34736     }
34737 ok_to_allocate:
34738     spin_lock_irqsave(&page_alloc_lock, flags);
34739     RMOQUEUE(order, gfp_mask);
34740     spin_unlock_irqrestore(&page_alloc_lock, flags);
34741
34742     /* If we can schedule, do so, and make sure to yield.

```

```

34743     * We may be a real-time process, and if kswapd is
34744     * waiting for us we need to allow it to run a bit. */
34745     if (gfp_mask & __GFP_WAIT) {
34746         current->policy |= SCHED_YIELD;
34747         schedule();
34748     }
34749
34750 nopage:
34751     return 0;
34752 }
34753
34754 /* Show free area list (used inside shift_scroll-lock
34755  * stuff) We also calculate the percentage
34756  * fragmentation. We do this by counting the memory on
34757  * each free list with the exception of the first item on
34758  * the list. */
34759 void show_free_areas(void)
34760 {
34761     unsigned long order, flags;
34762     unsigned long total = 0;
34763
34764     printk("Free pages:      %dkB\n ( ",
34765           nr_free_pages<<(PAGE_SHIFT-10));
34766     printk("Free: %d (%d %d %d)\n",
34767           nr_free_pages,
34768           freepages.min,
34769           freepages.low,
34770           freepages.high);
34771     spin_lock_irqsave(&page_alloc_lock, flags);
34772     for (order=0; order < NR_MEM_LISTS; order++) {
34773         struct page * tmp;
34774         unsigned long nr = 0;
34775         for (tmp = free_area[order].next;
34776              tmp != memory_head(free_area+order);
34777              tmp = tmp->next) {
34778             nr ++;
34779         }
34780         total += nr * ((PAGE_SIZE>>10) << order);
34781         printk("%lu*%lukB ", nr,
34782              (unsigned long)((PAGE_SIZE>>10) << order));
34783     }
34784     spin_unlock_irqrestore(&page_alloc_lock, flags);
34785     printk("= %lukB)\n", total);
34786 #ifdef SWAP_CACHE_INFO
34787     show_swap_cache_info();
34788 #endif
34789 }
34790

```

```

34791 #define LONG_ALIGN(x)
34792 (((x)+(sizeof(long))-1)&~((sizeof(long))-1))
34793
34794 /* set up the free-area data structures:
34795 * - mark all pages reserved
34796 * - mark all memory queues empty
34797 * - clear the memory bitmaps */
34798 unsigned long __init free_area_init(
34799 unsigned long start_mem, unsigned long end_mem)
34800 {
34801 mem_map_t * p;
34802 unsigned long mask = PAGE_MASK;
34803 unsigned long i;
34804
34805 /* Select nr of pages we try to keep free for important
34806 * stuff with a minimum of 10 pages and a maximum of
34807 * 256 pages, so that we don't waste too much memory on
34808 * large systems. This is fairly arbitrary, but based
34809 * on some behaviour analysis. */
34810 i = (end_mem - PAGE_OFFSET) >> (PAGE_SHIFT+7);
34811 if (i < 10)
34812     i = 10;
34813 if (i > 256)
34814     i = 256;
34815 freepages.min = i;
34816 freepages.low = i * 2;
34817 freepages.high = i * 3;
34818 mem_map = (mem_map_t *) LONG_ALIGN(start_mem);
34819 p = mem_map + MAP_NR(end_mem);
34820 start_mem = LONG_ALIGN(unsigned long) p;
34821 memset(mem_map, 0,
34822         start_mem - (unsigned long) mem_map);
34823 do {
34824     --p;
34825     atomic_set(&p->count, 0);
34826     p->flags = (1 << PG_DMA) | (1 << PG_reserved);
34827 } while (p > mem_map);
34828
34829 for (i = 0; i < NR_MEM_LISTS; i++) {
34830     unsigned long bitmap_size;
34831     init_mem_queue(free_area+i);
34832     mask += mask;
34833     end_mem = (end_mem + ~mask) & mask;
34834     bitmap_size =
34835         (end_mem - PAGE_OFFSET) >> (PAGE_SHIFT + i);
34836     bitmap_size = (bitmap_size + 7) >> 3;
34837     bitmap_size = LONG_ALIGN(bitmap_size);
34838     free_area[i].map = (unsigned int *) start_mem;

```

```

34839     memset((void *) start_mem, 0, bitmap_size);
34840     start_mem += bitmap_size;
34841 }
34842 return start_mem;
34843 }
34844
34845 /* Primitive swap readahead code. We simply read an
34846 * aligned block of (1 << page_cluster) entries in the
34847 * swap area. This method is chosen because it doesn't
34848 * cost us any seek time. We also make sure to queue the
34849 * 'original' request together with the readahead ones...
34850 */
34851 void swapin_readahead(unsigned long entry)
34852 {
34853     int i;
34854     struct page *new_page;
34855     unsigned long offset = SWP_OFFSET(entry);
34856     struct swap_info_struct *swapdev =
34857         SWP_TYPE(entry) + swap_info;
34858
34859     offset = (offset >> page_cluster) << page_cluster;
34860
34861     *i = 1 << page_cluster;
34862     do {
34863         /* Don't read-ahead past the end of the swap area */
34864         if (offset >= swapdev->max)
34865             break;
34866         /* Don't block on I/O for read-ahead */
34867         if (atomic_read(&nr_async_pages) >=
34868             pager_daemon.swap_cluster)
34869             break;
34870         /* Don't read in bad or busy pages */
34871         if (!swapdev->swap_map[offset])
34872             break;
34873         if (swapdev->swap_map[offset] == SWAP_MAP_BAD)
34874             break;
34875         if (test_bit(offset, swapdev->swap_lockmap))
34876             break;
34877
34878         /* Ok, do the async read-ahead now */
34879         new_page =
34880             read_swap_cache_async(SWP_ENTRY(SWP_TYPE(entry),
34881                                         offset), 0);
34882         if (new_page != NULL)
34883             __free_page(new_page);
34884         offset++;
34885     } while (--i);
34886     return;

```

```

34887 }
34888
34889 /* The tests may look silly, but it essentially makes
34890 * sure that no other process did a swap-in on us just as
34891 * we were waiting.
34892 *
34893 * Also, don't bother to add to the swap cache if this
34894 * page-in was due to a write access. */
34895 void swap_in(struct task_struct * tsk,
34896 struct vm_area_struct * vma, pte_t * page_table,
34897 unsigned long entry, int write_access)
34898 {
34899     unsigned long page;
34900     struct page *page_map = lookup_swap_cache(entry);
34901
34902     if (!page_map) {
34903         swapin_readahead(entry);
34904         page_map = read_swap_cache(entry);
34905     }
34906     if (pte_val(*page_table) != entry) {
34907         if (page_map)
34908             free_page_and_swap_cache(page_address(page_map));
34909         return;
34910     }
34911     if (!page_map) {
34912         set_pte(page_table, BAD_PAGE);
34913         swap_free(entry);
34914         oom(tsk);
34915         return;
34916     }
34917
34918     page = page_address(page_map);
34919     vma->vm_mm->rss++;
34920     tsk->min_flt++;
34921     swap_free(entry);
34922
34923     if (!write_access || is_page_shared(page_map)) {
34924         set_pte(page_table, mk_pte(page, vma->vm_page_prot));
34925         return;
34926     }
34927
34928     /* The page is unshared, and we want write access. In
34929     * this case, it is safe to tear down the swap cache
34930     * and give the page over entirely to this process. */
34931     if (PageSwapCache(page_map))
34932         delete_from_swap_cache(page_map);
34933     set_pte(page_table,
34934             pte_mkwrite(pte_mkdirty(mk_pte(page,

```

```

34935             vma->vm_page_prot))));
34936     return;
34937 }

```

mm/page_io.c

```

34938 /*
34939 * linux/mm/page_io.c
34940 *
34941 * Copyright (C) 1991, 1992, 1993, 1994 Linus Torvalds
34942 *
34943 * Swap reorganised 29.12.95,
34944 * Asynchronous swapping added 30.12.95. Stephen Tweedie
34945 * Removed race in async swapping, 14.4.1996. Bruno
34946 * Haible
34947 * Add swap of shared pages through the page
34948 * cache. 20.2.1998. Stephen Tweedie
34949 * Always use brw_page, life becomes simpler. 12 May
34950 * 1998 Eric Biederman */
34951
34952 #include <linux/mm.h>
34953 #include <linux/kernel_stat.h>
34954 #include <linux/swap.h>
34955 #include <linux/locks.h>
34956 #include <linux/swapctl.h>
34957
34958 #include <asm/pgtable.h>
34959
34960 static struct wait_queue * lock_queue = NULL;
34961
34962 /* Reads or writes a swap page.
34963 * wait=1: start I/O and wait for completion.
34964 * wait=0: start asynchronous I/O.
34965 *
34966 * Important prevention of race condition: the caller
34967 * *must* atomically create a unique swap cache entry for
34968 * this swap page before calling rw_swap_page, and must
34969 * lock that page. By ensuring that there is a single
34970 * page of memory reserved for the swap entry, the normal
34971 * VM page lock on that page also doubles as a lock on
34972 * swap entries. Having only one lock to deal with per
34973 * swap entry (rather than locking swap and memory
34974 * independently) also makes it easier to make certain
34975 * swapping operations atomic, which is particularly
34976 * important when we are trying to ensure that shared
34977 * pages stay shared while being swapped. */
34978
34979 static void rw_swap_page base(int rw,

```

```

34980 unsigned long entry, struct page *page, int wait)
34981 {
34982     unsigned long type, offset;
34983     struct swap_info struct * p;
34984     int zones[PAGE_SIZE/512];
34985     int zones_used;
34986     kdev_t dev = 0;
34987     int block_size;
34988
34989 #ifdef DEBUG_SWAP
34990     printk("DebugVM: %s_swap_page entry %08lx, "
34991           "page %p (count %d), %s\n",
34992           (rw == READ) ? "read" : "write",
34993           entry, (char *) page_address(page),
34994           atomic_read(&page->count),
34995           wait ? "wait" : "nowait");
34996 #endif
34997
34998     type = SWP_TYPE(entry);
34999     if (type >= nr_swapfiles) {
35000         printk("Internal error: bad swap-device\n");
35001         return;
35002     }
35003
35004     /* Don't allow too many pending pages in flight.. */
35005     if (atomic_read(&nr_async_pages) >
35006         pager_daemon.swap_cluster)
35007         wait = 1;
35008
35009     p = &swap_info[type];
35010     offset = SWP_OFFSET(entry);
35011     if (offset >= p->max) {
35012         printk("rw_swap_page: weirdness\n");
35013         return;
35014     }
35015     if (p->swap_map && !p->swap_map[offset]) {
35016         printk(KERN_ERR "rw_swap_page: "
35017               "Trying to %s unallocated swap (%08lx)\n",
35018               (rw == READ) ? "read" : "write", entry);
35019         return;
35020     }
35021     if (!(p->flags & SWP_USED)) {
35022         printk(KERN_ERR "rw_swap_page: "
35023               "Trying to swap to unused swap-device\n");
35024         return;
35025     }
35026
35027     if (!PageLocked(page)) {

```

```

35028         printk(KERN_ERR "VM: swap page is unlocked\n");
35029         return;
35030     }
35031
35032     if (PageSwapCache(page)) {
35033         /* Make sure we are the only process doing I/O with
35034          * this swap page. */
35035         while (test_and_set_bit(offset,p->swap_lockmap)) {
35036             run_task_queue(&tq_disk);
35037             sleep_on(&lock_queue);
35038         }
35039
35040         /* Make sure that we have a swap cache association
35041          * for this page. We need this to find which swap
35042          * page to unlock once the swap IO has completed to
35043          * the physical page. If the page is not already in
35044          * the cache, just overload the offset entry as if it
35045          * were: we are not allowed to manipulate the inode
35046          * hashing for locked pages. */
35047         if (page->offset != entry) {
35048             printk ("swap entry mismatch");
35049             return;
35050         }
35051     }
35052     if (rw == READ) {
35053         clear_bit(PG_uptodate, &page->flags);
35054         kstat.pswpin++;
35055     } else
35056         kstat.pswpout++;
35057
35058     atomic_inc(&page->count);
35059     if (p->swap_device) {
35060         zones[0] = offset;
35061         zones_used = 1;
35062         dev = p->swap_device;
35063         block_size = PAGE_SIZE;
35064     } else if (p->swap_file) {
35065         struct inode *swapf = p->swap_file->d_inode;
35066         int i;
35067         if (swapf->i_op->bmap == NULL
35068             && swapf->i_op->smap != NULL) {
35069             /* With MS-DOS, we use msdos_smap which returns a
35070              * sector number (not a cluster or block number).
35071              * It is a patch to enable the UMSDOS project.
35072              * Other people are working on better solution.
35073              *
35074              * It sounds like !! rw_swap_file defined its
35075              * operation size (sector size) based on PAGE_SIZE

```

```

35076     * and the number of blocks to read. So using bmap
35077     * or smap should work even if smap will require
35078     * more blocks. */
35079     int j;
35080     unsigned int block = offset << 3;
35081
35082     for (i=0, j=0; j< PAGE_SIZE ; i++, j += 512){
35083         if (!(zones[i] =
35084             swapf->i_op->smap(swapf,block++))) {
35085             printk("rw_swap_page: bad swap file\n");
35086             return;
35087         }
35088     }
35089     block_size = 512;
35090 } else {
35091     int j;
35092     unsigned int block = offset
35093         << (PAGE_SHIFT - swapf->i_sb->s_blocksize_bits);
35094
35095     block_size = swapf->i_sb->s_blocksize;
35096     for (i=0, j=0; j< PAGE_SIZE ; i++, j += block_size)
35097         if (!(zones[i] = bmap(swapf,block++))) {
35098             printk("rw_swap_page: bad swap file\n");
35099             return;
35100         }
35101     zones_used = i;
35102     dev = swapf->i_dev;
35103 }
35104 } else {
35105     printk(KERN_ERR
35106         "rw_swap_page: no swap file or device\n");
35107     /* Do some cleaning up so if this ever happens we can
35108     * hopefully trigger controlled shutdown. */
35109     if (PageSwapCache(page)) {
35110         if (!test_and_clear_bit(offset,p->swap_lockmap))
35111             printk("swap_after_unlock_page: lock already "
35112                 "cleared\n");
35113         wake_up(&lock_queue);
35114     }
35115     atomic_dec(&page->count);
35116     return;
35117 }
35118 if (!wait) {
35119     set_bit(PG_decr_after, &page->flags);
35120     atomic_inc(&nr_async_pages);
35121 }
35122 if (PageSwapCache(page)) {
35123     /* only lock/unlock swap cache pages! */
35124     set_bit(PG_swap_unlock_after, &page->flags);
35125 }
35126 set_bit(PG_free_after, &page->flags);
35127
35128 /* block_size == PAGE_SIZE/zones_used */
35129 brw_page(rw, page, dev, zones, block_size, 0);
35130
35131 /* Note! For consistency we do all of the logic,
35132 * decrementing the page count, and unlocking the page
35133 * in the swap lock map - in the IO completion handler.
35134 */
35135 if (!wait)
35136     return;
35137 wait_on_page(page);
35138 /* This shouldn't happen, but check to be sure. */
35139 if (atomic_read(&page->count) == 0)
35140     printk(KERN_ERR
35141         "rw_swap_page: page unused while waiting!\n");
35142
35143 #ifdef DEBUG_SWAP
35144     printk("DebugVM: %s_swap_page finished on page %p "
35145         "(count %d)\n",
35146         (rw == READ) ? "read" : "write",
35147         (char *) page_address(page),
35148         atomic_read(&page->count));
35149 #endif
35150 }
35151
35152 /* Note: We could remove this totally asynchronous
35153 * function, and improve swap performance, and remove the
35154 * need for the swap lock map, by not removing pages from
35155 * the swap cache until after I/O has been processed and
35156 * letting remove_from_page_cache decrement the swap
35157 * count just before it removes the page from the page
35158 * cache. */
35159 /* This is run when asynchronous page I/O has
35160 * completed. */
35161 void swap_after_unlock_page (unsigned long entry)
35162 {
35163     unsigned long type, offset;
35164     struct swap_info_struct * p;
35165
35166     type = SWP_TYPE(entry);
35167     if (type >= nr_swapfiles) {
35168         printk("swap_after_unlock_page: bad swap-device\n");
35169         return;
35170     }
35171     p = &swap_info[type];

```

```

35172 offset = SWP_OFFSET(entry);
35173 if (offset >= p->max) {
35174     printk("swap_after_unlock_page: weirdness\n");
35175     return;
35176 }
35177 if (!test_and_clear_bit(offset,p->swap_lockmap))
35178     printk("swap_after_unlock_page: "
35179         "lock already cleared\n");
35180 wake_up(&lock queue);
35181 }
35182
35183 /* A simple wrapper so the base function doesn't need to
35184  * enforce that all swap pages go through the swap cache!
35185  */
35186 void rw_swap_page(int rw, unsigned long entry, char *buf,
35187                 int wait)
35188 {
35189     struct page *page = mem_map + MAP_NR(buf);
35190
35191     if (page->inode && page->inode != &swapper_inode)
35192         panic("Tried to swap a non-swapper page");
35193
35194     /* Make sure that we have a swap cache association for
35195      * this page. We need this to find which swap page to
35196      * unlock once the swap IO has completed to the
35197      * physical page. If the page is not already in the
35198      * cache, just overload the offset entry as if it were:
35199      * we are not allowed to manipulate the inode hashing
35200      * for locked pages. */
35201     if (!PageSwapCache(page)) {
35202         printk("VM: swap page is not in swap cache\n");
35203         return;
35204     }
35205     if (page->offset != entry) {
35206         printk("swap entry mismatch");
35207         return;
35208     }
35209     rw_swap_page_base(rw, entry, page, wait);
35210 }
35211
35212 /* Setting up a new swap file needs a simple wrapper just
35213  * to read the swap signature. SysV shared memory also
35214  * needs a simple wrapper. */
35215 void rw_swap_page_nocache(int rw,
35216                          unsigned long entry, char *buffer)
35217 {
35218     struct page *page;

```

```

35219
35220     page = mem_map + MAP_NR((unsigned long) buffer);
35221     wait_on_page(page);
35222     set_bit(PG_locked, &page->flags);
35223     if (test_and_set_bit(PG_swap_cache, &page->flags)) {
35224         printk("VM: read_swap_page: "
35225             "page already in swap cache!\n");
35226         return;
35227     }
35228     if (page->inode) {
35229         printk("VM: read_swap_page: "
35230             "page already in page cache!\n");
35231         return;
35232     }
35233     page->inode = &swapper_inode;
35234     page->offset = entry;
35235     /* Protect from shrink_mmap() */
35236     atomic_inc(&page->count);
35237     rw_swap_page(rw, entry, buffer, 1);
35238     atomic_dec(&page->count);
35239     page->inode = 0;
35240     clear_bit(PG_swap_cache, &page->flags);
35241 }
35242
35243 /* shmf5 needs a version that doesn't put the page in the
35244  * page cache! The swap lock map insists that pages be
35245  * in the page cache! Therefore we can't use it. Later
35246  * when we can remove the need for the lock map and we
35247  * can reduce the number of functions exported. */
35248 void rw_swap_page_nolock(int rw, unsigned long entry,
35249                          char *buffer, int wait)
35250 {
35251     struct page *page =
35252         mem_map + MAP_NR((unsigned long) buffer);
35253
35254     if (!PageLocked(page)) {
35255         printk("VM: rw_swap_page_nolock: "
35256             "page not locked!\n");
35257         return;
35258     }
35259     if (PageSwapCache(page)) {
35260         printk("VM: rw_swap_page_nolock: "
35261             "page in swap cache!\n");
35262         return;
35263     }
35264     rw_swap_page_base(rw, entry, page, wait);
35265 }

```

mm/slab.c

```

35266 /*
35267 * linux/mm/slab.c
35268 * Written by Mark Hemment, 1996/97.
35269 * (markhe@nextd.demon.co.uk)
35270 *
35271 * 11 April '97. Started multi-threading - markhe
35272 * The global cache-chain is protected by the
35273 * semaphore 'cache_chain_sem'. The sem is only
35274 * needed when accessing/extending the cache-chain,
35275 * which can never happen inside an interrupt
35276 * (kmem_cache_create(), kmem_cache_shrink() and
35277 * kmem_cache_reap()). This is a medium-term
35278 * exclusion lock.
35279 *
35280 * Each cache has its own lock; 'c_spinlock'. This
35281 * lock is needed only when accessing non-constant
35282 * members of a cache-struct. Note: 'constant
35283 * members' are assigned a value in
35284 * kmem_cache_create() before the cache is linked
35285 * into the cache-chain. The values never change,
35286 * so not even a multi-reader lock is needed for
35287 * these members. The c_spinlock is only ever held
35288 * for a few cycles.
35289 *
35290 * To prevent kmem_cache_shrink() trying to shrink a
35291 * 'growing' cache (which maybe be sleeping and
35292 * therefore not holding the semaphore/lock), the
35293 * c_growing field is used. This also prevents
35294 * reaping from a cache.
35295 *
35296 * Note, caches can _never_ be destroyed. When a
35297 * sub-system (eg module) has finished with a cache,
35298 * it can only be shrunk. This leaves the cache
35299 * empty, but already enabled for re-use, eg. during
35300 * a module re load.
35301 *
35302 * Notes:
35303 * o Constructors/deconstructors are called
35304 * while the cache-lock is _not_ held.
35305 * Therefore they _must_ be threaded.
35306 * o Constructors must not attempt to
35307 * allocate memory from the same cache that
35308 * they are a constructor for - infinite
35309 * loop! (There is no easy way to trap
35310 * this.)
35311 * o The per-cache locks must be obtained
35312 *
35313 * with local-interrupts disabled. o When
35314 * compiled with debug support, and an
35315 * object-verify (upon release) is request
35316 * for a cache, the verify-function is
35317 * called with the cache lock held. This
35318 * helps debugging.
35319 * o The functions called from
35320 * try_to_free_page() must not attempt to
35321 * allocate memory from a cache which is
35322 * being grown. The buffer sub-system might
35323 * try to allocate memory, via
35324 * buffer_cachep. As this pri is passed to
35325 * the SLAB, and then (if necessary) onto
35326 * the gfp() funcs (which avoid calling
35327 * try_to_free_page()), no deadlock should
35328 * happen.
35329 *
35330 * The positioning of the per-cache lock is tricky.
35331 * If the lock is placed on the same h/w cache line
35332 * as commonly accessed members the number of L1
35333 * cache-line faults is reduced. However, this can
35334 * lead to the cache-line ping-ponging between
35335 * processors when the lock is in contention (and
35336 * the common members are being accessed). Decided
35337 * to keep it away from common members.
35338 *
35339 * More fine-graining is possible, with per-slab
35340 * locks...but this might be taking fine graining
35341 * too far, but would have the advantage;
35342 *
35343 * During most allocs/frees no writes occur
35344 * to the cache-struct. Therefore a
35345 * multi-reader/one writer lock could be
35346 * used (the writer needed when the slab
35347 * chain is being link/unlinked). As we
35348 * would not have an exclusion lock for the
35349 * cache-structure, one would be needed
35350 * per-slab (for updating s_free_ptr, and/or
35351 * the contents of s_index).
35352 *
35353 * The above locking would allow parallel operations
35354 * to different slabs within the same cache with
35355 * reduced spinning.
35356 *
35357 * Per-engine slab caches, backed by a global cache
35358 * (as in Mach's Zone allocator), would allow most
35359 * allocations from the same cache to execute in
35360 * parallel.

```



```

35360 *
35361 *   At present, each engine can be growing a cache.
35362 *   This should be blocked.
35363 *
35364 *   It is not currently 100% safe to examine the
35365 *   page_struct outside of a kernel or global cli
35366 *   lock. The risk is v. small, and non-fatal.
35367 *
35368 *   Calls to printk() are not 100% safe (the function
35369 *   is not threaded). However, printk() is only used
35370 *   under an error condition, and the risk is
35371 *   v. small (not sure if the console write functions
35372 *   'enjoy' executing multiple contexts in parallel.
35373 *   I guess they don't...). Note, for most calls to
35374 *   printk() any held cache-lock is dropped. This is
35375 *   not always done for text size reasons - having
35376 *   *_unlock() everywhere is bloat. */
35377
35378 /* An implementation of the Slab Allocator as described
35379 * in outline in;
35380 *   UNIX Internals: The New Frontiers by Uresh Vahalia
35381 *   Pub: Prentice Hall   ISBN 0-13-101908-2
35382 * or with a little more detail in;
35383 *   The Slab Allocator: An Object-Caching Kernel
35384 *   Memory Allocator
35385 *   Jeff Bonwick (Sun Microsystems).
35386 *   Presented at: USENIX Summer 1994 Technical
35387 *   Conference */
35388
35389 /* This implementation deviates from Bonwick's paper as
35390 * it does not use a hash-table for large objects, but
35391 * rather a per slab index to hold the bufctls. This
35392 * allows the bufctl structure to be small (one word),
35393 * but limits the number of objects a slab (not a cache)
35394 * can contain when off-slab bufctls are used. The limit
35395 * is the size of the largest general cache that does not
35396 * use off-slab bufctls, divided by the size of a bufctl.
35397 * For 32bit archs, is this 256/4 = 64. This is not
35398 * serious, as it is only for large objects, when it is
35399 * unwise to have too many per slab.
35400 *
35401 * Note: This limit can be raised by introducing a
35402 * general cache whose size is less than 512
35403 * (PAGE_SIZE<<3), but greater than 256. */
35404
35405 #include      <linux/config.h>
35406 #include      <linux/slab.h>
35407 #include      <linux/interrupt.h>

```

```

35408 #include      <linux/init.h>
35409
35410 /* If there is a different PAGE_SIZE around, and it works
35411 * with this allocator, then change the following. */
35412 #if      (PAGE_SIZE != 8192 && PAGE_SIZE != 4096)
35413 #error Your page size is probably not correctly \
35414 supported - please check
35415 #endif
35416
35417 /* SLAB_MGMT_CHECKS - 1 to enable extra checks in
35418 * kmem_cache_create().
35419 * 0 if you wish to reduce memory
35420 * usage.
35421 *
35422 * SLAB_DEBUG_SUPPORT - 1 for kmem cache create() to
35423 * honour; SLAB_DEBUG_FREE,
35424 * SLAB_DEBUG_INITIAL,
35425 * SLAB_RED_ZONE & SLAB_POISON.
35426 * 0 for faster, smaller, code
35427 * (especially in the critical
35428 * paths).
35429 *
35430 * SLAB_STATS - 1 to collect stats for
35431 * /proc/slabinfo.
35432 * 0 for faster, smaller, code
35433 * (especially in the critical
35434 * paths).
35435 *
35436 * SLAB_SELFTEST - 1 to perform a few tests, mainly
35437 * for development. */
35438 #define      SLAB_MGMT_CHECKS      1
35439 #define      SLAB_DEBUG_SUPPORT      0
35440 #define      SLAB_STATS      0
35441 #define      SLAB_SELFTEST      0
35442
35443 /* Shouldn't this be in a header file somewhere? */
35444 #define      BYTES_PER_WORD      sizeof(void *)
35445
35446 /* Legal flag mask for kmem_cache_create(). */
35447 #if      SLAB_DEBUG_SUPPORT
35448 #if      0
35449 #define      SLAB_C_MASK      \
35450 (SLAB_DEBUG_FREE|SLAB_DEBUG_INITIAL|SLAB_RED_ZONE| \
35451 SLAB_POISON|SLAB_HWCACHE_ALIGN|SLAB_NO_REAP| \
35452 SLAB_HIGH_PACK)
35453 #endif
35454 #define      SLAB_C_MASK      \
35455 (SLAB_DEBUG_FREE|SLAB_DEBUG_INITIAL|SLAB_RED_ZONE| \

```

```

35456 SLAB_POISON|SLAB_HWCACHE_ALIGN|SLAB_NO_REAP)
35457 #else
35458 #if 0
35459 #define SLAB_C_MASK
35460 (SLAB_HWCACHE_ALIGN|SLAB_NO_REAP|SLAB_HIGH_PACK)
35461 #endif
35462 #define SLAB_C_MASK
35463 (SLAB_HWCACHE_ALIGN|SLAB_NO_REAP)
35464 #endif /* SLAB_DEBUG_SUPPORT */
35465
35466 /* Slab management struct. Manages the objs in a slab.
35467 * Placed either at the end of mem allocated for a slab,
35468 * or from an internal obj cache (cache_slabp). Slabs
35469 * are chained into a partially ordered list; fully used
35470 * first, partial next, and then fully free slabs. The
35471 * first 4 members are referenced during an alloc/free
35472 * operation, and should always appear on the same cache
35473 * line. Note: The offset between some members _must_
35474 * match offsets within the kmem_cache_t - see
35475 * kmem_cache_init() for the checks. */
35476
35477 /* could make this larger for 64bit archs */
35478 #define SLAB_OFFSET_BITS 16
35479
35480 typedef struct kmem_slab_s {
35481 /* ptr to first inactive obj in slab */
35482 struct kmem_bufctl_s *s_freep;
35483 struct kmem_bufctl_s *s_index;
35484 unsigned long s_magic;
35485 /* num of objs active in slab */
35486 unsigned long s_inuse;
35487
35488 struct kmem_slab_s *s_nextp;
35489 struct kmem_slab_s *s_prevp;
35490 /* addr of first obj in slab */
35491 void *s_mem;
35492 unsigned long s_offset:SLAB_OFFSET_BITS,
35493 s_dma:1;
35494 } kmem_slab_t;
35495
35496 /* When the slab management is on-slab, this gives the
35497 * size to use. */
35498 #define slab_align_size
35499 (L1_CACHE_ALIGN(sizeof(kmem_slab_t)))
35500
35501 /* Test for end of slab chain. */
35502 #define kmem_slab_end(x)
35503 ((kmem_slab_t*)&(x)->c_offset))
35504
35505 /* s_magic */
35506 #define SLAB_MAGIC_ALLOC 0xA5C32F2BUL /* alive */
35507 #define SLAB_MAGIC_DESTROYED 0xB2F23C5AUL /* destroyed */
35508
35509 /* Bufctl's are used for linking objs within a slab,
35510 * identifying what slab an obj is in, and the address of
35511 * the associated obj (for sanity checking with off-slab
35512 * bufctls). What a bufctl contains depends upon the
35513 * state of the obj and the organisation of the cache. */
35514 typedef struct kmem_bufctl_s {
35515 union {
35516 struct kmem_bufctl_s *buf_nextp;
35517 kmem_slab_t *buf_slabp; /* slab for obj */
35518 void *buf_objp;
35519 } u;
35520 } kmem_bufctl_t;
35521
35522 /* ...shorthand... */
35523 #define buf_nextp u.buf_nextp
35524 #define buf_slabp u.buf_slabp
35525 #define buf_objp u.buf_objp
35526
35527 #if SLAB_DEBUG_SUPPORT
35528 /* Magic nums for obj red zoning. Placed in the first
35529 * word before and the first word after an obj. */
35530 #define SLAB_RED_MAGIC1 0x5A2CF071UL /* obj active */
35531 #define SLAB_RED_MAGIC2 0x170FC2A5UL /* obj inactive */
35532
35533 /* ...and for poisoning */
35534 #define SLAB_POISON_BYTE 0x5a /* byte val for poisoning */
35535 #define SLAB_POISON_END 0xa5 /* end-byte of poisoning */
35536
35537 #endif /* SLAB_DEBUG_SUPPORT */
35538
35539 /* Cache struct - manages a cache. First four members
35540 * are commonly referenced during an alloc/free
35541 * operation. */
35542 struct kmem_cache_s {
35543 kmem_slab_t *c_freep; /* first w/ free objs */
35544 unsigned long c_flags; /* constant flags */
35545 unsigned long c_offset;
35546 unsigned long c_num; /* # of objs per slab */
35547
35548 unsigned long c_magic;
35549 unsigned long c_inuse; /* kept at zero */
35550 kmem_slab_t *c_firstp; /* first slab in chain */
35551 kmem_slab_t *c_lastp; /* last slab in chain */

```

```

35552
35553 spinlock_t    c_spinlock;
35554 unsigned long c_growing;
35555 unsigned long c_dflgs;      /* dynamic flags */
35556 size_t        c_org_size;
35557 unsigned long c_gfporder; /* ord pgs per slab (2^n) */
35558 /* constructor func */
35559 void (*c_ctor)(void *, kmem_cache_t *, unsigned long);
35560 /* de-constructor func */
35561 void (*c_dtor)(void *, kmem_cache_t *, unsigned long);
35562 unsigned long c_align;      /* alignment of objs */
35563 size_t        c_colour;     /* cache coloring range*/
35564 size_t        c_colour_next; /* cache coloring */
35565 unsigned long c_failures;
35566 const char    *c_name;
35567 struct kmem_cache_s *c_nextp;
35568 kmem_cache_t  *c_index_cache;
35569 #if SLAB_STATS
35570 unsigned long c_num_active;
35571 unsigned long c_num_allocations;
35572 unsigned long c_high_mark;
35573 unsigned long c_grown;
35574 unsigned long c_reaped;
35575 atomic_t c_errors;
35576 #endif /* SLAB_STATS */
35577 };
35578
35579 /* internal c_flags */
35580 /* slab management in own cache */
35581 #define SLAB_CFLGS_OFF_SLAB 0x010000UL
35582 /* bufctls in own cache */
35583 #define SLAB_CFLGS_BUFCTL 0x020000UL
35584 /* a general cache */
35585 #define SLAB_CFLGS_GENERAL 0x080000UL
35586
35587 /* c_dflgs (dynamic flags). Need to hold the spinlock
35588 * to access this member */
35589 /* don't reap a recently grown */
35590 #define SLAB_CFLGS_GROWN 0x000002UL
35591
35592 #define SLAB_OFF_SLAB(x) ((x) & SLAB_CFLGS_OFF_SLAB)
35593 #define SLAB_BUFCTL(x) ((x) & SLAB_CFLGS_BUFCTL)
35594 #define SLAB_GROWN(x) ((x) & SLAB_CFLGS_GROWN)
35595
35596 #if SLAB_STATS
35597 #define SLAB_STATS_INC_ACTIVE(x) ((x)->c_num_active++)
35598 #define SLAB_STATS_DEC_ACTIVE(x) ((x)->c_num_active)
35599 #define SLAB_STATS_INC_ALLOCED(x) \

```

```

((x)->c_num_allocations++)
35601 #define SLAB_STATS_INC_GROWN(x) ((x)->c_grown++)
35602 #define SLAB_STATS_INC_REAPED(x) ((x)->c_reaped++)
35603 #define SLAB_STATS_SET_HIGH(x) \
35604 do { \
35605 if ((x)->c_num_active > (x)->c_high_mark) \
35606 (x)->c_high_mark = (x)->c_num_active; \
35607 } while (0)
35608 #define SLAB_STATS_INC_ERR(x) \
35609 (atomic_inc(&(x)->c_errors))
35610 #else
35611 #define SLAB_STATS_INC_ACTIVE(x)
35612 #define SLAB_STATS_DEC_ACTIVE(x)
35613 #define SLAB_STATS_INC_ALLOCED(x)
35614 #define SLAB_STATS_INC_GROWN(x)
35615 #define SLAB_STATS_INC_REAPED(x)
35616 #define SLAB_STATS_SET_HIGH(x)
35617 #define SLAB_STATS_INC_ERR(x)
35618 #endif /* SLAB_STATS */
35619
35620 #if SLAB_SELFTEST
35621 #if !SLAB_DEBUG_SUPPORT
35622 #error Debug support needed for self-test
35623 #endif
35624 static void kmem_self_test(void);
35625 #endif /* SLAB_SELFTEST */
35626
35627 /* c_magic - used to detect 'out of slabs' in
35628 * __kmem_cache_alloc() */
35629 #define SLAB_C_MAGIC 0x4F17A36DUL
35630
35631 /* maximum size of an obj (in 2^order pages) */
35632 #define SLAB_OBJ_MAX_ORDER 5 /* 32 pages */
35633
35634 /* maximum num of pages for a slab (prevents large
35635 * requests to the VM layer) */
35636 #define SLAB_MAX_GFP_ORDER 5 /* 32 pages */
35637
35638 /* the 'preferred' minimum num of objs per slab - maybe
35639 * less for large objs */
35640 #define SLAB_MIN_OBJS_PER_SLAB 4
35641
35642 /* If the num of objs per slab is <=
35643 * SLAB_MIN_OBJS_PER_SLAB, then the page order must be
35644 * less than this before trying the next order. */
35645 #define SLAB_BREAK_GFP_ORDER_HI 2
35646 #define SLAB_BREAK_GFP_ORDER_LO 1
35647 static int slab_break_gfp_order =

```

```

35648 SLAB_BREAK_GFP_ORDER_LO;
35649
35650 /* Macros for storing/retrieving the cachep and or slab
35651 * from the global 'mem_map'. With off-slab bufctls,
35652 * these are used to find the slab an obj belongs to.
35653 * With kmalloc(), and kfree(), these are used to find
35654 * the cache which an obj belongs to. */
35655 #define SLAB_SET_PAGE_CACHE(pg, x) \
35656 ((pg)->next = (struct page *)(x))
35657 #define SLAB_GET_PAGE_CACHE(pg) \
35658 ((kmem_cache_t *)(pg)->next)
35659 #define SLAB_SET_PAGE_SLAB(pg, x) \
35660 ((pg)->prev = (struct page *)(x))
35661 #define SLAB_GET_PAGE_SLAB(pg) \
35662 ((kmem_slab_t *) (pg)->prev)
35663
35664 /* Size description struct for general caches. */
35665 typedef struct cache_sizes {
35666     size_t      cs_size;
35667     kmem_cache_t *cs_cachep;
35668 } cache_sizes_t;
35669
35670 static cache_sizes_t cache_sizes[] = {
35671 #if PAGE_SIZE == 4096
35672     { 32,      NULL},
35673 #endif
35674     { 64,      NULL},
35675     { 128,     NULL},
35676     { 256,     NULL},
35677     { 512,     NULL},
35678     {1024,     NULL},
35679     {2048,     NULL},
35680     {4096,     NULL},
35681     {8192,     NULL},
35682     {16384,    NULL},
35683     {32768,    NULL},
35684     {65536,    NULL},
35685     {131072,   NULL},
35686     {0,        NULL}
35687 };
35688
35689 /* Names for the general caches. Not placed into the
35690 * sizes struct for a good reason; the string ptr is not
35691 * needed while searching in kmalloc(), and would
35692 * 'get-in-the-way' in the h/w cache. */
35693 static char *cache_sizes_name[] = {
35694 #if PAGE_SIZE == 4096
35695     "size-32",

```

```

35696 #endif
35697     "size-64",
35698     "size-128",
35699     "size-256",
35700     "size-512",
35701     "size-1024",
35702     "size-2048",
35703     "size-4096",
35704     "size-8192",
35705     "size-16384",
35706     "size-32768",
35707     "size-65536",
35708     "size-131072"
35709 };
35710
35711 /* internal cache of cache description objs */
35712 static kmem_cache_t cache_cache = {
35713     /* freep, flags */      kmem_slab_end(&cache_cache),
35714                             SLAB_NO_REAP,
35715     /* offset, num */      sizeof(kmem_cache_t), 0,
35716     /* c_magic, c_inuse */ SLAB_C_MAGIC, 0,
35717     /* firstp, lastp */   kmem_slab_end(&cache_cache),
35718                             kmem_slab_end(&cache_cache),
35719     /* spinlock */        SPIN_LOCK_UNLOCKED,
35720     /* growing */         0,
35721     /* dflags */          0,
35722     /* org_size, gfp */   0, 0,
35723     /* ctor, dtor, align */ NULL, NULL, LI_CACHE_BYTES,
35724     /* colour, colour next */ 0, 0,
35725     /* failures */        0,
35726     /* name */             "kmem_cache",
35727     /* nextp */            &cache_cache,
35728     /* index */           NULL,
35729 };
35730
35731 /* Guard access to the cache chain. */
35732 static struct semaphore cache_chain sem;
35733
35734 /* Place maintainer for reaping. */
35735 static kmem_cache_t *clock_searchp = &cache_cache,
35736
35737 /* Internal slab management cache, for when slab
35738 * management is off-slab. */
35739 static kmem_cache_t *cache_slabp = NULL;
35740
35741 /* Max number of objs-per-slab for caches which use
35742 * bufctl's. Needed to avoid a possible looping
35743 * condition in kmem_cache_grow(). */

```

```

35744 static unsigned long bufctl_limit = 0;
35745
35746 /* Initialisation - setup the 'cache' cache. */
35747 long __init kmem_cache_init(long start, long end)
35748 {
35749     size_t size, i;
35750
35751 #define kmem_slab_offset(x) \
35752     ((unsigned long)&((kmem_slab_t *)0)->x)
35753 #define kmem_slab_diff(a,b) \
35754     (kmem_slab_offset(a) - kmem_slab_offset(b))
35755 #define kmem_cache_offset(x) \
35756     ((unsigned long)&((kmem_cache_t *)0)->x)
35757 #define kmem_cache_diff(a,b) \
35758     (kmem_cache_offset(a) - kmem_cache_offset(b))
35759
35760 /* Sanity checks... */
35761 if (kmem_cache_diff(c_firstp, c_magic) !=
35762     kmem_slab_diff(s_nextp, s_magic) ||
35763     kmem_cache_diff(c_firstp, c_inuse) !=
35764     kmem_slab_diff(s_nextp, s_inuse) ||
35765     ((kmem_cache_offset(c_lastp)
35766      ((unsigned long)
35767       kmem_slab_end((kmem_cache_t*)NULL))) !=
35768      kmem_slab_offset(s_prevp)) ||
35769     kmem_cache_diff(c_lastp, c_firstp) !=
35770     kmem_slab_diff(s_prevp, s_nextp)) {
35771     /* Offsets to the magic are incorrect, either the
35772      * structures have been incorrectly changed, or
35773      * adjustments are needed for your architecture. */
35774     panic("kmem_cache_init(): Offsets are wrong - "
35775          "I've been messed with!");
35776     /* NOTREACHED */
35777 }
35778 #undef kmem_cache_offset
35779 #undef kmem_cache_diff
35780 #undef kmem_slab_offset
35781 #undef kmem_slab_diff
35782
35783 cache_chain_sem = MUTEX;
35784
35785 size = cache_cache.c_offset + sizeof(kmem_bufctl_t);
35786 size += (L1_CACHE_BYTES-1);
35787 size &- (L1_CACHE_BYTES-1);
35788 cache_cache.c_offset = size-sizeof(kmem_bufctl_t);
35789
35790 i = (PAGE_SIZE << cache_cache.c_gfporder) -
35791     slab_align_size;
35792 cache_cache.c_num = i / size; /* objs / slab */
35793
35794 /* Cache colouring. */
35795 cache_cache.c_colour =
35796     (i-(cache_cache.c_num*size))/L1_CACHE_BYTES;
35797 cache_cache.c_colour_next = cache_cache.c_colour;
35798
35799 /* Fragmentation resistance on low memory - only use
35800  * bigger page orders on machines with more than 32MB
35801  * of memory. */
35802 if (num_physpages > (32 << 20) >> PAGE_SHIFT)
35803     slab_break_gfp_order = SLAB_BREAK_GFP_ORDER_HI;
35804 return start;
35805 }
35806
35807 /* Initialisation - setup remaining internal and general
35808  * caches. Called after the gfp() functions have been
35809  * enabled, and before smp init(). */
35810 void init_kmem_cache_sizes_init(void)
35811 {
35812     unsigned int found = 0;
35813
35814     cache_slabp = kmem_cache_create("slab cache",
35815     sizeof(kmem_slab_t), 0, SLAB_HWCACHE_ALIGN,
35816     NULL, NULL);
35817     if (cache_slabp) {
35818         char **names = cache_sizes_name;
35819         cache_sizes_t *sizes = cache_sizes;
35820         do {
35821             /* For performance, all the general caches are 11
35822              * aligned. This should be particularly beneficial
35823              * on SMP boxes, as it eliminates "false sharing".
35824              * Note for systems short on memory removing the
35825              * alignment will allow tighter packing of the
35826              * smaller caches. */
35827             if (!(sizes->cs_cachep =
35828                 kmem_cache_create(*names++, sizes->cs_size,
35829                 0, SLAB_HWCACHE_ALIGN, NULL, NULL)))
35830                 goto panic_time;
35831             if (!found) {
35832                 /* Inc off-slab bufctl limit until the ceiling is
35833                  * hit. */
35834                 if (SLAB_BUFCTL(sizes->cs_cachep->c_flags))
35835                     found++;
35836             } else
35837                 bufctl_limit -=
35838                     (sizes->cs_size/sizeof(kmem_bufctl_t));
35839         }

```

```

35840     sizes->cs_cachep->c_flags |= SLAB_CFLGS_GENERAL;
35841     sizes++;
35842     } while (sizes->cs_size);
35843 #if     SLAB_SELFTEST
35844     kmem_self_test();
35845 #endif /* SLAB_SELFTEST */
35846     return;
35847 }
35848 panic_time:
35849 panic("kmem_cache_sizes_init: Error creating caches");
35850 /* NOTREACHED */
35851 }
35852
35853 /* Interface to system's page allocator. Dma pts to
35854  * non-zero if all of memory is DMAable. No need to hold
35855  * the cache-lock. */
35856 static inline void *
35857 kmem_getpages(kmem_cache_t *cachep, unsigned long flags,
35858              unsigned int *dma)
35859 {
35860     void *addr;
35861
35862     *dma = flags & SLAB_DMA;
35863     addr =
35864         (void*) __get_free_pages(flags, cachep->c_gfporder);
35865     /* Assume that now we have the pages no one else can
35866      * legally mess with the 'struct page's. However
35867      * vm_scan() might try to test the structure to see if
35868      * it is a named-page or buffer-page. The members it
35869      * tests are of no interest here..... */
35870     if (!*dma && addr) {
35871         /* Need to check if can dma. */
35872         struct page *page = mem_map + MAP_NR(addr);
35873         *dma = 1 << cachep->c_gfporder;
35874         while ((*dma)--> {
35875             if (!PageDMA(page)) {
35876                 *dma = 0;
35877                 break;
35878             }
35879             page++;
35880         }
35881     }
35882     return addr;
35883 }
35884
35885 /* Interface to system's page release. */
35886 static inline void
35887 kmem_freepages(kmem_cache_t *cachep, void *addr)

```

```

35888 {
35889     unsigned long i = (1 << cachep->c_gfporder);
35890     struct page *page = &mem_map[MAP_NR(addr)];
35891
35892     /* free_pages() does not clear the type bit - we do
35893      * that. The pages have been unlinked from their
35894      * cache-slab, but their 'struct page's might be
35895      * accessed in vm_scan(). Shouldn't be a worry. */
35896     while (i--> {
35897         PageClearSlab(page);
35898         page++;
35899     }
35900     free_pages((unsigned long)addr, cachep->c_gfporder);
35901 }
35902
35903 #if     SLAB_DEBUG_SUPPORT
35904 static inline void
35905 kmem_poison_obj(kmem_cache_t *cachep, void *addr)
35906 {
35907     memset(addr, SLAB_POISON_BYTE, cachep->c_org_size);
35908     *(unsigned char *) (addr+cachep->c_org_size-1) =
35909         SLAB_POISON_END;
35910 }
35911
35912 static inline int
35913 kmem_check_poison_obj(kmem_cache_t *cachep, void *addr)
35914 {
35915     void *end;
35916     end = memchr(addr, SLAB_POISON_END,
35917                 cachep->c_org_size);
35918     if (end != (addr+cachep->c_org_size-1))
35919         return 1;
35920     return 0;
35921 }
35922 #endif /* SLAB_DEBUG_SUPPORT */
35923
35924 /* Three slab chain funcs - all called with ints disabled
35925  * and the appropriate cache lock held. */
35926 static inline void
35927 kmem_slab_unlink(kmem_slab_t *slabp)
35928 {
35929     kmem_slab_t *prevp = slabp->s_prevp;
35930     kmem_slab_t *nextp = slabp->s_nextp;
35931     prevp->s_nextp = nextp;
35932     nextp->s_prevp = prevp;
35933 }
35934
35935 static inline void

```

```

35936 kmem_slab_link_end(kmem_cache_t *cachep,
35937                     kmem_slab_t *slabp)
35938 {
35939     kmem_slab_t *lastp = cachep->c_lastp;
35940     slabp->s_nextp = kmem_slab_end(cachep);
35941     slabp->s_prevp = lastp;
35942     cachep->c_lastp = slabp;
35943     lastp->s_nextp = slabp;
35944 }
35945
35946 static inline void
35947 kmem_slab_link_free(kmem_cache_t *cachep,
35948                   kmem_slab_t *slabp)
35949 {
35950     kmem_slab_t *nextp = cachep->c_freep;
35951     kmem_slab_t *prevp = nextp->s_prevp;
35952     slabp->s_nextp = nextp;
35953     slabp->s_prevp = prevp;
35954     nextp->s_prevp = slabp;
35955     slabp->s_prevp->s_nextp = slabp;
35956 }
35957
35958 /* Destroy all the objs in a slab, and release the mem
35959 * back to the system. Before calling the slab must have
35960 * been unlinked from the cache. The cache-lock is not
35961 * held/needed. */
35962 static void
35963 kmem_slab_destroy(kmem_cache_t *cachep,
35964                 kmem_slab_t *slabp)
35965 {
35966     if (cachep->c_dtor
35967 #if SLAB_DEBUG_SUPPORT
35968         || cachep->c_flags & (SLAB_POISON | SLAB_RED_ZONE)
35969 #endif /*SLAB_DEBUG_SUPPORT*/
35970     ) {
35971         /* Doesn't use the bufctl ptrs to find objs. */
35972         unsigned long num = cachep->c_num;
35973         void *objp = slabp->s_mem;
35974         do {
35975 #if SLAB_DEBUG_SUPPORT
35976             if (cachep->c_flags & SLAB_RED_ZONE) {
35977                 if (*((unsigned long*)(objp)) != SLAB_RED_MAGIC1)
35978                     printk(KERN_ERR "kmem_slab_destroy: "
35979                            "Bad front redzone - %s\n",
35980                            cachep->c_name);
35981                 objp += BYTES_PER_WORD;
35982             } if (*((unsigned long*)(objp+cachep->c_org_size))
35983                 != SLAB_RED_MAGIC1)

```

```

35984             printk(KERN_ERR "kmem_slab_destroy: "
35985                    "Bad rear redzone - %s\n",
35986                    cachep->c_name);
35987         }
35988         if (cachep->c_dtor)
35989 #endif /*SLAB_DEBUG_SUPPORT*/
35990             (cachep->c_dtor)(objp, cachep, 0);
35991 #if SLAB_DEBUG_SUPPORT
35992         else if (cachep->c_flags & SLAB_POISON) {
35993             if (kmem_check_poison_obj(cachep, objp))
35994                 printk(KERN_ERR "kmem_slab_destroy: "
35995                        "Bad poison - %s\n", cachep->c_name);
35996         }
35997         if (cachep->c_flags & SLAB_RED_ZONE)
35998             objp -= BYTES_PER_WORD;
35999 #endif /* SLAB_DEBUG_SUPPORT */
36000         objp += cachep->c_offset;
36001         if (!slabp->s_index)
36002             objp += sizeof(kmem_bufctl_t);
36003         } while (--num);
36004     }
36005
36006     slabp->s_magic = SLAB_MAGIC_DESTROYED;
36007     if (slabp->s_index)
36008         kmem_cache_free(cachep->c_index_cachep,
36009                        slabp->s_index);
36010     kmem_freepages(cachep, slabp->s_mem+slabp->s_offset);
36011     if (SLAB_OFF_SLAB(cachep->c_flags))
36012         kmem_cache_free(cache_slabp, slabp);
36013 }
36014
36015 /* Call the num objs, wastage, and bytes left over for a
36016 * given slab size. */
36017 static inline size_t
36018 kmem_cache_cal_waste(unsigned long gfporder, size_t size,
36019                     size_t extra, unsigned long flags, size_t *left_over,
36020                     unsigned long *num)
36021 {
36022     size_t wastage = PAGE_SIZE<<gfporder;
36023
36024     if (SLAB_OFF_SLAB(flags))
36025         gfporder = 0;
36026     else
36027         gfporder = slab_align_size;
36028     wastage -= gfporder;
36029     *num = wastage / size;
36030     wastage -= (*num * size);
36031     *left_over = wastage;

```

```

36032
36033 return (wastage + gfporder + (extra * *num));
36034 )
36035
36036 /* Create a cache: Returns a ptr to the cache on success,
36037 * NULL on failure. Cannot be called within a int, but
36038 * can be interrupted. NOTE: The 'name' is assumed to be
36039 * memory that is _not_ going to disappear. */
36040 kmem_cache_t *
36041 kmem_cache_create(const char *name, size_t size,
36042 size_t offset, unsigned long flags,
36043 void (*ctor)(void*, kmem_cache_t *, unsigned long),
36044 void (*dtor)(void*, kmem_cache_t *, unsigned long))
36045 {
36046 const char *func_nm= KERN_ERR "kmem_create: ";
36047 kmem_cache_t *searchp;
36048 kmem_cache_t *cachep=NULL;
36049 size_t extra;
36050 size_t left_over;
36051 size_t align;
36052
36053 /* Sanity checks... */
36054 #if SLAB_MGMT_CHECKS
36055 if (!name) {
36056 printk("%sNULL ptr\n", func_nm);
36057 goto opps;
36058 }
36059 if (in_interrupt()) {
36060 printk("%sCalled during int - %s\n", func_nm, name);
36061 goto opps;
36062 }
36063
36064 if (size < BYTES_PER_WORD) {
36065 printk("%sSize too small %d - %s\n",
36066 func_nm, (int) size, name);
36067 size = BYTES_PER_WORD;
36068 }
36069
36070 if (size > ((1<<SLAB_OBJ_MAX_ORDER)*PAGE_SIZE)) {
36071 printk("%sSize too large %d - %s\n",
36072 func_nm, (int) size, name);
36073 goto opps;
36074 }
36075
36076 if (dtor && !ctor) {
36077 /* Decon, but no con - doesn't make sense */
36078 printk("%sDecon but no con - %s\n", func_nm, name);
36079 goto opps;

```

```

36080 }
36081
36082 if (offset < 0 || offset > size) {
36083 printk("%soffset weird %d - %s\n",
36084 func_nm, (int) offset, name);
36085 offset = 0;
36086 }
36087
36088 #if SLAB_DEBUG_SUPPORT
36089 if ((flags & SLAB_DEBUG_INITIAL) && !ctor) {
36090 /* No ctor, but initial state check requested */
36091 printk("%sNo con, but init state check requested "
36092 "%s\n", func_nm, name);
36093 flags &= ~SLAB_DEBUG_INITIAL;
36094 }
36095
36096 if ((flags & SLAB_POISON) && ctor) {
36097 /* request for poisoning, but we can't do that with a
36098 * constructor */
36099 printk("%sPoisoning requested, but con given %s\n",
36100 func_nm, name);
36101 flags &= ~SLAB_POISON;
36102 }
36103 #if 0
36104 if ((flags & SLAB_HIGH_PACK) && ctor) {
36105 printk("%sHigh pack requested, but con given %s\n",
36106 func_nm, name);
36107 flags &= ~SLAB_HIGH_PACK;
36108 }
36109 if ((flags & SLAB_HIGH_PACK) &&
36110 (flags & (SLAB_POISON|SLAB_RFD_ZONE))) {
36111 printk("%sHigh pack requested, but with "
36112 "poisoning/red-zoning %s\n",
36113 func_nm, name);
36114 flags &= ~SLAB_HIGH_PACK;
36115 }
36116 #endif
36117 #endif /* SLAB_DEBUG_SUPPORT */
36118 #endif /* SLAB_MGMT_CHECKS */
36119
36120 /* Always checks flags, a caller might be expecting
36121 * debug support which isn't available. */
36122 if (flags & ~SLAB_C_MASK) {
36123 printk("%sIllgl flg %ix - %s\n",
36124 func_nm, flags, name);
36125 flags &= SLAB_C_MASK;
36126 }
36127

```



```

36128 /* Get cache's description obj. */
36129 cachep =
36130     (kmem_cache_t *) kmem_cache_alloc(&cache_cache,
36131                                       SLAB_KERNEL);
36132 if (!cachep)
36133     goto opps;
36134 memset(cachep, 0, sizeof(kmem_cache_t));
36135
36136 /* Check that size is in terms of words. This is
36137  * needed to avoid unaligned accesses for some archs
36138  * when redzoning is used, and makes sure any on-slab
36139  * bufctl's are also correctly aligned. */
36140 if (size & (BYTES_PER_WORD-1)) {
36141     size += (BYTES_PER_WORD-1);
36142     size &= ~(BYTES_PER_WORD-1);
36143     printk("%sForcing size word alignment - %s\n",
36144           func_nm, name);
36145 }
36146
36147 cachep->c_org_size = size;
36148 #if SLAB_DEBUG_SUPPORT
36149 if (flags & SLAB_RED_ZONE) {
36150     /* There is no point trying to honour cache alignment
36151      * when redzoning. */
36152     flags &= ~SLAB_HWCACHE_ALIGN;
36153     size += 2*BYTES_PER_WORD; /* words for redzone */
36154 }
36155 #endif /* SLAB_DEBUG_SUPPORT */
36156
36157 align = BYTES_PER_WORD;
36158 if (flags & SLAB_HWCACHE_ALIGN)
36159     align = L1_CACHE_BYTES;
36160
36161 /* Determine if the slab management and/or bufctls are
36162  * 'on' or 'off' slab. */
36163 extra = sizeof(kmem_bufctl_t);
36164 if (size < (PAGE_SIZE>>3)) {
36165     /* Size is small(ish). Use packing where bufctl size
36166      * per obj is low, and slab management is on-slab. */
36167     #if 0
36168     if ((flags & SLAB_HIGH_PACK)) {
36169         /* Special high packing for small objects (mainly
36170          * for vm_mapping structs, but others can use it).
36171          */
36172         if (size == (L1_CACHE_BYTES/4) ||
36173             size == (L1_CACHE_BYTES/2) ||
36174             size == L1_CACHE_BYTES) {
36175             /* The bufctl is stored with the object. */

```

```

36176         extra = 0;
36177     } else
36178         flags &= ~SLAB_HIGH_PACK;
36179     }
36180 #endif
36181 } else {
36182     /* Size is large, assume best to place the slab
36183      * management obj off-slab (should allow better
36184      * packing of objs). */
36185     flags |= SLAB_CFLGS_OFF_SLAB;
36186     if (!(size & ~PAGE_MASK) || size == (PAGE_SIZE/2) ||
36187         size == (PAGE_SIZE/4) || size == (PAGE_SIZE/8)) {
36188         /* To avoid waste the bufctls are off slab... */
36189         flags |= SLAB_CFLGS_BUFCTL;
36190         extra = 0;
36191     } /* else slab management is off slab, but freelist
36192        * pointers are on. */
36193 }
36194 size += extra;
36195
36196 if (flags & SLAB_HWCACHE_ALIGN) {
36197     /* Need to adjust size so that objs are cache
36198      * aligned. */
36199     if (size > (L1_CACHE_BYTES/2)) {
36200         size_t words = size % L1_CACHE_BYTES;
36201         if (words)
36202             size += (L1_CACHE_BYTES - words);
36203     } else {
36204         /* Small obj size, can get at least two per cache
36205          * line. */
36206         int num_per_line = L1_CACHE_BYTES/size;
36207         left_over = L1_CACHE_BYTES - (num_per_line*size);
36208         if (left_over) {
36209             /* Need to adjust size so objs cache align. */
36210             if (left_over%num_per_line) {
36211                 /* Odd num of objs per line - fixup. */
36212                 num_per_line--;
36213                 left_over += size;
36214             }
36215             size += (left_over/num_per_line);
36216         }
36217     }
36218 } else if (!(size%L1_CACHE_BYTES)) {
36219     /* Size happens to cache align... */
36220     flags |= SLAB_HWCACHE_ALIGN;
36221     align = L1_CACHE_BYTES;
36222 }
36223 }

```

```

36224 /* Cal size (in pages) of slabs, and the num of objs
36225 * per slab. This could be made much more intelligent.
36226 * For now, try to avoid using high page-orders for
36227 * slabs. When the gfp() funcs are more friendly
36228 * towards high-order requests, this should be changed.
36229 */
36230 do {
36231     size_t wastage;
36232     unsigned int break_flag = 0;
36233 cal_wastage:
36234     wastage = kmem_cache_cal_waste(cachep->c_gfporder,
36235     size, extra, flags, &left_over, &cachep->c_num);
36236     if (!cachep->c_num)
36237         goto next;
36238     if (break_flag)
36239         break;
36240     if (SLAB_BUFCTL(flags) &&
36241         cachep->c_num > bufctl_limit) {
36242         /* Oops, this num of objs will cause problems. */
36243         cachep->c_gfporder--;
36244         break_flag++;
36245         goto cal_wastage;
36246     }
36247     if (cachep->c_gfporder == SLAB_MAX_GFP_ORDER)
36248         break;
36249
36250     /* Large num of objs is good, but v. large slabs are
36251     * currently bad for the gfp()s. */
36252     if (cachep->c_num <= SLAB_MIN_OBJS_PER_SLAB) {
36253         if (cachep->c_gfporder < slab_break_gfp_order)
36254             goto next;
36255     }
36256
36257     /* Stop caches with small objs having a large num of
36258     * pages. */
36259     if (left_over <= slab_align_size)
36260         break;
36261     if ((wastage*8) <= (PAGE_SIZE<<<cachep->c_gfporder))
36262         break; /* Acceptable internal fragmentation. */
36263 next:
36264     cachep->c_gfporder++;
36265 } while (1);
36266
36267 /* If the slab has been placed off-slab, and we have
36268 * enough space then move it on-slab. This is at the
36269 * expense of any extra colouring. */
36270 if ((flags & SLAB_CFLGS_OFF_SLAB) &&
36271     !SLAB_BUFCTL(flags) &&

```

```

36272     left_over >= slab_align_size) {
36273     flags &= ~SLAB_CFLGS_OFF_SLAB;
36274     left_over -= slab_align_size;
36275 }
36276
36277 /* Offset must be a factor of the alignment. */
36278 offset += (align-1);
36279 offset &= ~(align-1);
36280
36281 /* Mess around with the offset alignment. */
36282 if (!left_over) {
36283     offset = 0;
36284 } else if (left_over < offset) {
36285     offset = align;
36286     if (flags & SLAB_HWCACHE_ALIGN) {
36287         if (left_over < offset)
36288             offset = 0;
36289     } else {
36290         /* Offset is BYTES_PER_WORD, and left over is at
36291         * least BYTES PER WORD.
36292         */
36293         if (left_over >= (BYTES_PER_WORD*2)) {
36294             offset >>= 1;
36295             if (left_over >= (BYTES_PER_WORD*4))
36296                 offset >>= 1;
36297         }
36298     }
36299 } else if (!offset) {
36300     /* No offset requested, but space enough give
36301     * one. */
36302     offset = left_over/align;
36303     if (flags & SLAB_HWCACHE_ALIGN) {
36304         if (offset >= 8) {
36305             /* A large number of colours use a larger
36306             * alignment. */
36307             align <<= 1;
36308         }
36309     } else {
36310         if (offset >= 10) {
36311             align <<= 1;
36312             if (offset >= 16)
36313                 align <<= 1;
36314         }
36315     }
36316     offset = align;
36317 }
36318
36319 #if 0

```

```

36320 printk("%s: Left_over:%d Align:%d Size:%d\n",
36321         name, left_over, offset, size);
36322 #endif
36323
36324 if ((cachep->c_align = (unsigned long) offset))
36325     cachep->c_colour = (left_over/offset);
36326 cachep->c_colour_next = cachep->c_colour;
36327
36328 /* If the bufctl's are on-slab, c_offset does not
36329  * include the size of bufctl. */
36330 if (!SLAB_BUFCTL(flags))
36331     size -= sizeof(kmem_bufctl_t);
36332 else
36333     cachep->c_index_cachep =
36334     kmem_find_general_cachep(cachep->c_num *
36335                             sizeof(kmem_bufctl_t));
36336 cachep->c_offset = (unsigned long) size;
36337 cachep->c_freep = kmem_slab_end(cachep);
36338 cachep->c_firstp = kmem_slab_end(cachep);
36339 cachep->c_lastp = kmem_slab_end(cachep);
36340 cachep->c_flags = flags;
36341 cachep->c_ctor = ctor;
36342 cachep->c_dtor = dtor;
36343 cachep->c_magic = SLAB_C_MAGIC;
36344 cachep->c_name = name; /* Simply point to the name. */
36345 spin_lock_init(&cachep->c_spinlock);
36346
36347 /* Need the semaphore to access the chain. */
36348 down(&cache_chain_sem);
36349 searchp = &cache_cache;
36350 do {
36351     /* The name field is constant - no lock needed. */
36352     if (!strcmp(searchp->c_name, name)) {
36353         printk("%sDup name - %s\n", func_nm, name);
36354         break;
36355     }
36356     searchp = searchp->c_nextp;
36357 } while (searchp != &cache_cache);
36358
36359 /* There is no reason to lock our new cache before we
36360  * link it in - no one knows about it yet...
36361  */
36362 cachep->c_nextp = cache_cache.c_nextp;
36363 cache_cache.c_nextp = cachep;
36364 up(&cache_chain_sem);
36365 opps:
36366 return cachep;
36367 }

```

```

36368
36369 /* Shrink a cache. Releases as many slabs as possible
36370  * for a cache. It is expected this function will be
36371  * called by a module when it is unloaded. The cache is
36372  * .not_ removed, this creates too many problems and the
36373  * cache structure does not take up much room. A module
36374  * should keep its cache pointer(s) in unloaded memory,
36375  * so when reloaded it knows the cache is available. To
36376  * help debugging, a zero exit status indicates all slabs
36377  * were released. */
36378 int
36379 kmem_cache_shrink(kmem_cache_t *cachep)
36380 {
36381     kmem_cache_t *searchp;
36382     kmem_slab_t *slabp;
36383     int ret;
36384
36385     if (!cachep) {
36386         printk(KERN_ERR "kmem shrink: NULL ptr\n");
36387         return 2;
36388     }
36389     if (in_interrupt()) {
36390         printk(KERN_ERR "kmem_shrink: Called during int - "
36391                "%s\n", cachep->c_name);
36392         return 2;
36393     }
36394
36395     /* Find the cache in the chain of caches. */
36396     down(&cache_chain_sem); /* Semaphore is needed. */
36397     searchp = &cache_cache;
36398     for (; searchp->c_nextp != &cache_cache;
36399         searchp = searchp->c_nextp) {
36400         if (searchp->c_nextp != cachep)
36401             continue;
36402
36403         /* Accessing clock searchp is safe we hold the
36404          * mutex. */
36405         if (cachep == clock_searchp)
36406             clock_searchp = cachep->c_nextp;
36407         goto found;
36408     }
36409     up(&cache_chain_sem);
36410     printk(KERN_ERR "kmem_shrink: Invalid cache addr %p\n",
36411            cachep);
36412     return 2;
36413 found:
36414     /* Release the semaphore before getting the cache lock.
36415      * This could mean multiple engines are shrinking the

```



```

36512         "after constructor - %s\n",
36513         cachep->c_name);
36514     }
36515 }
36516 #endif /* SLAB_DEBUG_SUPPORT */
36517
36518     objp += cachep->c_offset;
36519     if (!slab->s_index) {
36520         *bufpp = objp;
36521         objp += sizeof(kmem_bufctl_t);
36522     } else
36523         *bufpp = &slab->s_index[num];
36524     bufpp = &(*bufpp)->buf_nextp;
36525     } while (num--);
36526
36527     *bufpp = NULL;
36528 }
36529
36530 /* Grow (by 1) the number of slabs within a cache. This
36531  * is called by kmem_cache_alloc() when there are no
36532  * active objs left in a cache. */
36533 static int
36534 kmem_cache_grow(kmem_cache_t *cachep, int flags)
36535 {
36536     kmem_slab_t    *slabp;
36537     struct page    *page;
36538     void           *objp;
36539     size_t         offset;
36540     unsigned int   dma, local_flags;
36541     unsigned long  ctor_flags;
36542     unsigned long  save_flags;
36543
36544     /* Be lazy and only check for valid flags here, keeping
36545      * it out of the critical path in kmem_cache_alloc().
36546      */
36547     if (flags & ~(SLAB_DMA|SLAB_LEVEL_MASK|SLAB_NO_GROW)) {
36548         printk(KERN_WARNING "kmem_grow: Illegal flgs %X "
36549                "(correcting) - %s\n", flags, cachep->c_name);
36550         flags &= (SLAB_DMA|SLAB_LEVEL_MASK|SLAB_NO_GROW);
36551     }
36552
36553     if (flags & SLAB_NO_GROW)
36554         return 0;
36555
36556     /* The test for missing atomic flag is performed here,
36557      * rather than the more obvious place, simply to reduce
36558      * the critical path length in kmem_cache_alloc(). If
36559      * a caller is slightly mis-behaving they will

```

```

36560     * eventually be caught here (where it matters). */
36561     if (in_interrupt() &&
36562         (flags & SLAB_LEVEL_MASK) != SLAB_ATOMIC) {
36563         printk(KERN_ERR "kmem_grow: Called nonatomically "
36564                "from int - %s\n", cachep->c_name);
36565         flags &= ~SLAB_LEVEL_MASK;
36566         flags |= SLAB_ATOMIC;
36567     }
36568     ctor_flags = SLAB_CTOR_CONSTRUCTOR;
36569     local_flags = (flags & SLAB_LEVEL_MASK);
36570     if (local_flags == SLAB_ATOMIC) {
36571         /* Not allowed to sleep. Need to tell a constructor
36572          * about this - it might need to know... */
36573         ctor_flags |= SLAB_CTOR_ATOMIC;
36574     }
36575
36576     /* About to mess with non-constant members - lock. */
36577     spin_lock_irqsave(&cachep->c_spinlock, save_flags);
36578
36579     /* Get colour for the slab, and cal the next value. */
36580     if (!(offset = cachep->c_colour_next))
36581         cachep->c_colour_next = cachep->c_colour;
36582     offset += cachep->c_align;
36583     cachep->c_dflgs = SLAB_FLAGS_GROWN;
36584
36585     cachep->c_growing++;
36586     spin_unlock_irqrestore(&cachep->c_spinlock,
36587                          save_flags);
36588
36589     /* A series of memory allocations for a new slab.
36590      * Neither the cache-chain semaphore, or cache-lock,
36591      * are held, but the incrementing c_growing prevents
36592      * this this cache from being reaped or shrunk. Note:
36593      * The cache could be selected in for reaping in
36594      * kmem_cache_reap(), but when the final test is made
36595      * the growing value will be seer. */
36596
36597     /* Get mem for the objs. */
36598     if (!(objp = kmem_getpages(cachep, flags, &dma)))
36599         goto failed;
36600
36601     /* Get slab management. */
36602     if (!(slabp = kmem_cache_slabmgmt(cachep,
36603                                       objp+offset,
36604                                       local_flags)))
36605         goto opps1;
36606     if (dma)
36607         slabp->s_dma = 1;

```

```

36608 if (SLAB_BUFCTL(cachep->c_flags)) {
36609     slabp->s_index =
36610         kmem_cache_alloc(cachep->c_index_cachep,
36611             local_flags);
36612     if (!slabp->s_index)
36613         goto opps2;
36614 }
36615
36616 /* Nasty!!!!!! I hope this is OK. */
36617 dma = 1 << cachep->c_gfporder;
36618 page = &mem_map[MAP_NR(objp)];
36619 do {
36620     SLAB_SET_PAGE_CACHE(page, cachep);
36621     SLAB_SET_PAGE_SLAB(page, slabp);
36622     PageSetSlab(page);
36623     page++;
36624 } while (--dma);
36625
36626 slabp->s_offset = offset; /* It will fit... */
36627 objp += offset; /* Address of first object. */
36628 slabp->s_mem = objp;
36629
36630 /* For on-slab bufctls, c_offset is the distance
36631  * between the start of an obj and its related bufctl.
36632  * For off-slab bufctls, c_offset is the distance
36633  * between objs in the slab. */
36634 kmem_cache_init_objs(cachep, slabp, objp, ctor_flags);
36635
36636 spin_lock_irq(&cachep->c_spinlock);
36637
36638 /* Make slab active. */
36639 slabp->s_magic = SLAB_MAGIC_ALLOC;
36640 kmem_slab_link_end(cachep, slabp);
36641 if (cachep->c_freep == kmem_slab_end(cachep))
36642     cachep->c_freep = slabp;
36643 SLAB_STATS_INC_GROWN(cachep);
36644 cachep->c_failures = 0;
36645 cachep->c_growing--;
36646
36647 spin_unlock_irqrestore(&cachep->c_spinlock,
36648     save_flags);
36649 return 1;
36650 opps2:
36651 if (SLAB_OFF_SLAB(cachep->c_flags))
36652     kmem_cache_free(cache_slabp, slabp);
36653 opps1:
36654 kmem_freepages(cachep, objp);
36655 failed:

```

```

36656 spin_lock_irq(&cachep->c_spinlock);
36657 cachep->c_growing--;
36658 spin_unlock_irqrestore(&cachep->c_spinlock,
36659     save_flags);
36660 return 0;
36661 }
36662
36663 static void
36664 kmem_report_alloc_err(const char *str,
36665     kmem_cache_t * cachep)
36666 {
36667     if (cachep)
36668         SLAB_STATS_INC_ERR(cachep); /* this is atomic */
36669     printk(KERN_ERR "kmem_alloc: %s (name=%s)\n",
36670         str, cachep ? cachep->c_name : "unknown");
36671 }
36672
36673 static void
36674 kmem_report_free_err(const char *str, const void *objp,
36675     kmem_cache_t * cachep)
36676 {
36677     if (cachep)
36678         SLAB_STATS_INC_ERR(cachep);
36679     printk(KERN_ERR "kmem_free: %s (objp=%p, name=%s)\n",
36680         str, objp, cachep ? cachep->c_name : "unknown");
36681 }
36682
36683 /* Search for a slab whose objs are suitable for DMA.
36684  * Note: since testing the first free slab (in
36685  * __kmem_cache_alloc()), ints must not have been
36686  * enabled, or the cache-lock released! */
36687 static inline kmem_slab_t *
36688 kmem_cache_search_dma(kmem_cache_t * cachep)
36689 {
36690     kmem_slab_t *slabp = cachep->c_freep->s_nextp;
36691
36692     for (; slabp != kmem_slab_end(cachep);
36693         slabp = slabp->s_nextp) {
36694         if (!(slabp->s_dma))
36695             continue;
36696         kmem_slab_unlink(slabp);
36697         kmem_slab_link_free(cachep, slabp);
36698         cachep->c_freep = slabp;
36699         break;
36700     }
36701     return slabp;
36702 }
36703

```

```

36704 #if SLAB_DEBUG_SUPPORT
36705 /* Perform extra freeing checks. Currently, this check
36706 * is only for caches that use bufctl structures within
36707 * the slab. Those which use bufctl's from the internal
36708 * cache have a reasonable check when the address is
36709 * searched for. Called with the cache-lock held. */
36710 static void *
36711 kmem_extra_free_checks(kmem_cache_t *cachep,
36712                       kmem_bufctl_t *search_bufp,
36713                       kmem_bufctl_t *bufp, void *objp)
36714 {
36715     if (SLAB_BUFCTL(cachep->c_flags))
36716         return objp;
36717
36718     /* Check slab's freelist to see if this obj is
36719      * there. */
36720     for (; search_bufp;
36721          search_bufp = search_bufp->buf_nextp) {
36722         if (search_bufp != bufp)
36723             continue;
36724         return NULL;
36725     }
36726     return objp;
36727 }
36728 #endif /* SLAB_DEBUG_SUPPORT */
36729
36730 /* Called with cache lock held. */
36731 static inline void
36732 kmem_cache_full_free(kmem_cache_t *cachep,
36733                     kmem_slab_t *slabp)
36734 {
36735     if (slabp->s_nextp->s_inuse) {
36736         /* Not at correct position. */
36737         if (cachep->c_freep == slabp)
36738             cachep->c_freep = slabp->s_nextp;
36739         kmem_slab_unlink(slabp);
36740         kmem_slab_link_end(cachep, slabp);
36741     }
36742 }
36743
36744 /* Called with cache lock held. */
36745 static inline void
36746 kmem_cache_one_free(kmem_cache_t *cachep,
36747                    kmem_slab_t *slabp)
36748 {
36749     if (slabp->s_nextp->s_inuse == cachep->c_num) {
36750         kmem_slab_unlink(slabp);
36751         kmem_slab_link_free(cachep, slabp);

```

```

36752     }
36753     cachep->c_freep = slabp;
36754 }
36755
36756 /* Returns a ptr to an obj in the given cache. */
36757 static inline void *
36758 __kmem_cache_alloc(kmem_cache_t *cachep, int flags)
36759 {
36760     kmem_slab_t *slabp;
36761     kmem_bufctl_t *bufp;
36762     void *objp;
36763     unsigned long save_flags;
36764
36765     /* Sanity check. */
36766     if (!cachep)
36767         goto nul_ptr;
36768     spin_lock_irqsave(&cachep->c_spinlock, save_flags);
36769     try_again:
36770     /* Get slab alloc is to come from. */
36771     slabp = cachep->c_freep;
36772
36773     /* Magic is a sanity check _and_ says if we need a new
36774      * slab. */
36775     if (slabp->s_magic != SLAB_MAGIC_ALLOC)
36776         goto alloc_new_slab;
36777     /* DMA requests are 'rare' - keep out of the critical
36778      * path. */
36779     if (flags & SLAB_DMA)
36780         goto search_dma;
36781     try_again_dma:
36782     SLAB_STATS_INC_ALLOCED(cachep);
36783     SLAB_STATS_INC_ACTIVE(cachep);
36784     SLAB_STATS_SET_HIGH(cachep);
36785     slabp->s_inuse++;
36786     bufp = slabp->s_freep;
36787     slabp->s_freep = bufp->buf_nextp;
36788     if (slabp->s_freep) {
36789         ret_obj:
36790         if (!slabp->s_index) {
36791             bufp->buf_slabp = slabp;
36792             objp = ((void*)bufp) + cachep->c_offset;
36793         finished:
36794         /* The lock is not needed by the red-zone or poison
36795          * ops, and the obj has been removed from the slab.
36796          * Should be safe to drop the lock here. */
36797         spin_unlock_irqrestore(&cachep->c_spinlock,
36798                               save_flags);
36799     #if SLAB_DEBUG_SUPPORT

```

```

36800     if (cachep->c_flags & SLAB_RED_ZONE)
36801         goto red_zone;
36802 ret_red:
36803     if ((cachep->c_flags & SLAB_POISON) &&
36804         kmem_check_poison_obj(cachep, objp))
36805         kmem_report_alloc_err("Bad poison", cachep);
36806 #endif /* SLAB_DEBUG_SUPPORT */
36807     return objp;
36808 }
36809 /* Update index ptr. */
36810 objp = ((bufp->slabp->s_index)*cachep->c_offset) +
36811         slabp->s_mem;
36812 bufp->buf_objp = objp;
36813 goto finished;
36814 }
36815 cachep->c_freep = slabp->s_nextp;
36816 goto ret_obj;
36817
36818 #if SLAB_DEBUG_SUPPORT
36819 red_zone:
36820 /* Set alloc red-zone, and check old one. */
36821 if (xchg((unsigned long *)objp, SLAB_RED_MAGIC2)
36822     != SLAB_RED_MAGIC1)
36823     kmem_report_alloc_err("Bad front redzone", cachep);
36824 objp += BYTES_PER_WORD;
36825 if (xchg((unsigned long *) (objp+cachep->c_org_size),
36826         SLAB_RED_MAGIC2) != SLAB_RED_MAGIC1)
36827     kmem_report_alloc_err("Bad rear redzone", cachep);
36828 goto ret_red;
36829 #endif /* SLAB_DEBUG_SUPPORT */
36830
36831 search_dma:
36832 if (slabp->s_dma ||
36833     (slabp = kmem_cache_search_dma(cachep)) !=
36834     kmem_slab_end(cachep))
36835     goto try_again_dma;
36836 alloc_new_slab:
36837 /* Either out of slabs, or magic number corruption. */
36838 if (slabp == kmem_slab_end(cachep)) {
36839     /* Need a new slab. Release the lock before calling
36840      * kmem_cache_grow(). This allows objs to be
36841      * released back into the cache while growing. */
36842     spin_unlock_irqrestore(&cachep->c_spinlock,
36843         save_flags);
36844     if (kmem_cache_grow(cachep, flags)) {
36845         /* Someone may have stolen our objs. Doesn't
36846          * matter, we'll just come back here again. */
36847         spin_lock_irq(&cachep->c_spinlock);

```

```

36848         goto try_again;
36849     }
36850     /* Couldn't grow, but some objs may have been
36851      * freed. */
36852     spin_lock_irq(&cachep->c_spinlock);
36853     if (cachep->c_freep != kmem_slab_end(cachep)) {
36854         if ((flags & SLAB_ATOMIC) == 0)
36855             goto try_again;
36856     }
36857 } else {
36858     /* Very serious error - maybe panic() here? */
36859     kmem_report_alloc_err("Bad slab magic (corrupt)",
36860         cachep);
36861 }
36862 spin_unlock_irqrestore(&cachep->c_spinlock,
36863     save_flags);
36864 err_exit:
36865 return NULL;
36866 nul_ptr:
36867 kmem_report_alloc_err("NULL ptr", NULL);
36868 goto err_exit;
36869 }
36870
36871 /* Release an obj back to its cache. If the obj has a
36872 * constructed state, it should be in this state before
36873 * it is released. */
36874 static inline void
36875 kmem_cache_free(kmem_cache_t *cachep, const void *objp)
36876 {
36877     kmem_slab_t *slabp;
36878     kmem_bufctl_t *bufp;
36879     unsigned long save_flags;
36880
36881     /* Basic sanity checks. */
36882     if (!cachep || !objp)
36883         goto null_addr;
36884
36885 #if SLAB_DEBUG_SUPPORT
36886     /* A verify func is called without the cache-lock
36887      * held. */
36888     if (cachep->c_flags & SLAB_DEBUG_INITIAL)
36889         goto init_state_check;
36890     finished_initial:
36891
36892     if (cachep->c_flags & SLAB_RED_ZONE)
36893         goto red_zone;
36894     return_red;
36895 #endif /* SLAB_DEBUG_SUPPORT */

```



```

36896
36897 spin_lock_irqsave(&cachep->c_spinlock, save_flags);
36898
36899 if (SLAB_BUFCTL(cachep->c_flags))
36900     goto bufctl;
36901 bufp = (kmem_bufctl_t *) (objp+cachep->c_offset);
36902
36903 /* Get slab for the object. */
36904 #if 0
36905 /* _NASTY_IF/ELSE_, but avoids a 'distant' memory ref
36906  * for some objects. Is this worth while? XXX */
36907 if (cachep->c_flags & SLAB_HIGH_PACK)
36908     slabp = SLAB_GET_PAGE_SLAB(&mem_map[MAP_NR(bufp)]);
36909 else
36910 #endif
36911     slabp = bufp->buf_slabp;
36912
36913 check_magic:
36914 /* Sanity check. */
36915 if (slabp->s_magic != SLAB_MAGIC_ALLOC)
36916     goto bad_slab;
36917
36918 #if SLAB_DEBUG_SUPPORT
36919 if (cachep->c_flags & SLAB_DEBUG_FREE)
36920     goto extra_checks;
36921 passed_extra:
36922 #endif /* SLAB_DEBUG_SUPPORT */
36923
36924 if (slabp->s_inuse) { /* Sanity check. */
36925     SLAB_STATS_DEC_ACTIVE(cachep);
36926     slabp->s_inuse--;
36927     bufp->buf_nextp = slabp->s_freep;
36928     slabp->s_freep = bufp;
36929     if (bufp->buf_nextp) {
36930         if (slabp->s_inuse) {
36931             /* (hopefully) The most common case. */
36932             finished:
36933             #if SLAB_DEBUG_SUPPORT
36934             if (cachep->c_flags & SLAB_POISON) {
36935                 if (cachep->c_flags & SLAB_RED_ZONE)
36936                     objp += BYTES_PER_WORD;
36937                 kmem_poison_obj(cachep, objp);
36938             }
36939             #endif /* SLAB_DEBUG_SUPPORT */
36940             spin_unlock_irqrestore(&cachep->c_spinlock,
36941                 save_flags);
36942             return;
36943         }

```

```

36944         kmem_cache_full free(cachep, slabp);
36945         goto finished;
36946     }
36947     kmem_cache_one_free(cachep, slabp);
36948     goto finished;
36949 }
36950
36951 /* Don't add to freelist. */
36952 spin_unlock_irqrestore(&cachep->c_spinlock,
36953     save_flags);
36954 kmem_report_free_err("free with no active objs",
36955     objp, cachep);
36956 return;
36957 bufctl:
36958 /* No 'extra' checks are performed for objs stored this
36959  * way, finding the obj is check enough. */
36960 slabp = SLAB_GET_PAGE_SLAB(&mem_map[MAP_NR(objp)]);
36961 bufp = &slabp->s_index[(objp - slabp->s_mem) /
36962     cachep->c_offset];
36963 if (bufp->buf_objp == objp)
36964     goto check_magic;
36965 spin_unlock_irqrestore(&cachep->c_spinlock,
36966     save_flags);
36967 kmem_report_free_err("Either bad obj addr or double "
36968     "free", objp, cachep);
36969 return;
36970 #if SLAB_DEBUG_SUPPORT
36971 init_state_check:
36972 /* Need to call the slab's constructor so the caller
36973  * can perform a verify of its state (debugging). */
36974 cachep->c_ctor(objp, cachep,
36975     SLAB_CTOR_CONSTRUCTOR|SLAB_CTOR_VERIFY);
36976 goto finished_initial;
36977 extra_checks:
36978 if (!kmem_extra_free_checks(cachep, slabp->s_freep,
36979     bufp, objp)) {
36980     spin_unlock_irqrestore(&cachep->c_spinlock,
36981         save_flags);
36982     kmem_report_free_err("Double free detected during "
36983         "checks", objp, cachep);
36984     return;
36985 }
36986 goto passed_extra;
36987 red_zone:
36988 /* We do not hold the cache-lock while checking the
36989  * red-zone. */
36990 objp += BYTES_PER_WORD;
36991 if (xchg((unsigned long *)objp, SLAB_RED_MAGIC1) !=

```

```

36992     SLAB_RED_MAGIC2) (
36993     /* Either write before start of obj. or a double
36994      * free. */
36995     kmem_report_free_err("Bad front redzone", objp,
36996                          cachep);
36997 }
36998 if (xchg((unsigned long *)
36999         (objp+cachep->c_org_size+BYTES_PER_WORD),
37000         SLAB_RED_MAGIC1) != SLAB_RED_MAGIC2) {
37001     /* Either write past end of obj. or a double free. */
37002     kmem_report_free_err("Bad rear redzone",
37003                          objp, cachep);
37004 }
37005 goto return_red;
37006 #endif /* SLAB_DEBUG_SUPPORT */
37007
37008 bad_slab:
37009 /* Slab doesn't contain the correct magic num. */
37010 if (slabp->s_magic == SLAB_MAGIC_DESTROYED) {
37011     /* Magic num says this is a destroyed slab. */
37012     kmem_report_free_err("free from inactive slab",
37013                          objp, cachep);
37014 } else
37015     kmem_report_free_err("Bad obj addr", objp, cachep);
37016 spin_unlock_irqrestore(&cachep->c_spinlock,
37017                        save_flags);
37018
37019 #if 1
37020 /* FORCE A KERNEL DUMP WHEN THIS HAPPENS. SPEAK IN ALL
37021  * CAPS. GET THE CALL CHAIN. */
37022 *(int *) 0 = 0;
37023 #endif
37024
37025 return;
37026 null_addr:
37027 kmem_report_free_err("NULL ptr", objp, cachep);
37028 return;
37029 }
37030
37031 void *
37032 kmem_cache_alloc(kmem_cache_t *cachep, int flags)
37033 {
37034     return __kmem_cache_alloc(cachep, flags);
37035 }
37036
37037 void
37038 kmem_cache_free(kmem_cache_t *cachep, void *objp)
37039 {

```

```

37040     __kmem_cache_free(cachep, objp);
37041 }
37042
37043 void *
37044 kmem_alloc(size_t size, int flags)
37045 {
37046     cache_sizes_t *csizes = cache_sizes;
37047
37048     for (; csizes->cs_size; csizes++) {
37049         if (size > csizes->cs_size)
37050             continue;
37051         return __kmem_cache_alloc(csizes->cs_cachep, flags);
37052     }
37053     printk(KERN_ERR "kmalloc: Size (%lu) too large\n",
37054            (unsigned long) size);
37055     return NULL;
37056 }
37057
37058 void
37059 kfree(const void *objp)
37060 {
37061     struct page *page;
37062     int nr;
37063
37064     if (!objp)
37065         goto null_ptr;
37066     nr = MAP_NR(objp);
37067     if (nr >= max_mapnr)
37068         goto bad_ptr;
37069
37070     /* Assume we own the page structure - hence no locking.
37071      * If someone is misbehaving (for example, calling us
37072      * with a bad address), then access to the page
37073      * structure can race with the kmem_slab_destroy()
37074      * code. Need to add a spin_lock to each page
37075      * structure, which would be useful in threading the
37076      * gfp() functions.... */
37077     page = &mem_map[nr];
37078     if (PageSlab(page)) {
37079         kmem_cache_t *cachep;
37080
37081         /* Here, we again assume the obj address is good. If
37082          * it isn't, and happens to map onto another general
37083          * cache page which has no active objs, then we race.
37084          */
37085         cachep = SLAB_GET_PAGE_CACHE(page);
37086         if (cachep &&
37087             (cachep->c_flags & SLAB_CFLGS_GENERAL)) {

```

```

37088     __kmem_cache_free(cachep, objp);
37089     return;
37090     }
37091     }
37092 bad_ptr:
37093     printk(KERN_ERR "kfree: Bad obj %p\n", objp);
37094
37095 #if 1
37096 /* FORCE A KERNEL DUMP WHEN THIS HAPPENS. SPEAK IN ALL
37097  * CAPS. GET THE CALL CHAIN. */
37098 *(int *) 0 = 0;
37099 #endif
37100
37101 null_ptr:
37102     return;
37103 }
37104
37105 void
37106 kfree_s(const void *objp, size_t size)
37107 {
37108     struct page *page;
37109     int nr;
37110
37111     if (!objp)
37112         goto null_ptr;
37113     nr = MAP_NR(objp);
37114     if (nr >= max_mapnr)
37115         goto null_ptr;
37116     /* See comment in kfree() */
37117     page = &mem_map[nr];
37118     if (PageSlab(page)) {
37119         kmem_cache_t *cachep;
37120         /* See comment in kfree() */
37121         cachep = SLAB_GET_PAGE_CACHE(page);
37122         if (cachep && cachep->c_flags & SLAB_CFLGS_GENERAL) {
37123             if (size <= cachep->c_org_size) {
37124                 /* XXX better check */
37125                 __kmem_cache_free(cachep, objp);
37126                 return;
37127             }
37128         }
37129     }
37130     null_ptr:
37131     printk(KERN_ERR "kfree_s: Bad obj %p\n", objp);
37132     return;
37133 }
37134
37135 kmem_cache_t *

```

```

37136 kmem_find_general_cachep(size_t size)
37137 {
37138     cache_sizes_t *csizp = cache_sizes;
37139
37140     /* This function could be moved to the header file, and
37141      * made inline so consumers can quickly determine what
37142      * cache pointer they require.
37143      */
37144     for (; csizp->cs_size; csizp++) {
37145         if (size > csizp->cs_size)
37146             continue;
37147         break;
37148     }
37149     return csizp->cs_cachep;
37150 }
37151
37152
37153 /* Called from try to free page().
37154  * This function cannot be called within a int, but it
37155  * can be interrupted.
37156  */
37157 void
37158 kmem_cache_reap(int gfp_mask)
37159 {
37160     kmem_slab_t *slabp;
37161     kmem_cache_t *searchp;
37162     kmem_cache_t *best_cachep;
37163     unsigned int scan;
37164     unsigned int reap_level;
37165
37166     if (in_interrupt()) {
37167         printk("kmem_cache_reap() called within int!\n");
37168         return;
37169     }
37170
37171     /* We really need a test semaphore op so we can avoid
37172      * sleeping when !wait is true. */
37173     down(&cache_chain_sem);
37174
37175     scan = 10;
37176     reap_level = 0;
37177
37178     best_cachep = NULL;
37179     searchp = clock_searchp;
37180     do {
37181         unsigned int full free;
37182         unsigned int dma_flag;
37183

```

```

37184 /* It's safe to test this without holding the
37185  * cache-lock. */
37186 if (searchp->c_flags & SLAB_NO_REAP)
37187     goto next;
37188 spin_lock_irq(&searchp->c_spinlock);
37189 if (searchp->c_growing)
37190     goto next_unlock;
37191 if (searchp->c_dflags & SLAB_CFLGS_GROWN) {
37192     searchp->c_dflags &= ~SLAB_CFLGS_GROWN;
37193     goto next_unlock;
37194 }
37195 /* Sanity check for corruption of static values. */
37196 if (searchp->c_inuse ||
37197     searchp->c_magic != SLAB_C_MAGIC) {
37198     spin_unlock_irq(&searchp->c_spinlock);
37199     printk(KERN_ERR "kmem_reap: Corrupted cache struct"
37200             " for %s\n", searchp->c_name);
37201     goto next;
37202 }
37203 dma_flag = 0;
37204 full_free = 0;
37205
37206 /* Count the fully free slabs. There should not be
37207  * not many, since we are holding the cache lock. */
37208 slabp = searchp->c_lastp;
37209 while (!slabp->s_inuse &&
37210        slabp != kmem_slab_end(searchp)) {
37211     slabp = slabp->s_prevp;
37212     full_free++;
37213     if (slabp->s_dma)
37214         dma_flag++;
37215 }
37216 spin_unlock_irq(&searchp->c_spinlock);
37217
37218 if ((gfp_mask & GFP_DMA) && !dma_flag)
37219     goto next;
37220
37221 if (full_free) {
37222     if (full_free >= 10) {
37223         best_cachep = searchp;
37224         break;
37225     }
37226
37227     /* Try to avoid slabs with constructors and/or more
37228      * than one page per slab (as it can be difficult
37229      * to get high orders from gfp()). */
37230     if (full_free >= reap_level) {
37231
37232         best_cachep = searchp;
37233     }
37234 }
37235 goto next;
37236 next_unlock:
37237     spin_unlock_irq(&searchp->c_spinlock);
37238 next:
37239     searchp = searchp->c_nextp;
37240 } while (--scan && searchp != clock_searchp);
37241
37242 clock_searchp = searchp;
37243 up(&cache_chain sem);
37244
37245 if (!best_cachep) {
37246     /* couldn't find anything to reap */
37247     return;
37248 }
37249
37250 spin_lock_irq(&best_cachep->c_spinlock);
37251 while (!best_cachep->c_growing &&
37252        !(slabp = best_cachep->c_lastp) > s_inuse &&
37253        slabp != kmem_slab_end(best_cachep)) {
37254     if (gfp_mask & GFP_DMA) {
37255         do {
37256             if (slabp->s_dma)
37257                 goto good_dma;
37258             slabp = slabp->s_prevp;
37259         } while (!slabp > s_inuse &&
37260                slabp != kmem_slab_end(best_cachep));
37261
37262         /* Didn't found a DMA slab (there was a free one
37263          * must have been become active). */
37264         goto dma_fail;
37265     good_dma:
37266     }
37267     if (slabp == best_cachep->c_freep)
37268         best_cachep->c_freep = slabp->s_nextp;
37269     kmem_slab_unlink(slabp);
37270     SLAB_STATS_INC_REAPED(best_cachep);
37271
37272     /* Safe to drop the lock. The slab is no longer
37273      * linked to the cache. */
37274     spin_unlock_irq(&best_cachep->c_spinlock);
37275     kmem_slab_destroy(best_cachep, slabp);
37276     spin_lock_irq(&best_cachep->c_spinlock);
37277 }
37278 dma_fail:
37279     spin_unlock_irq(&best_cachep->c_spinlock);

```

```

37280 return;
37281 }
37282
37283 #if SLAB_SELFTEST
37284 /* A few v. simple tests */
37285 static void
37286 kmem_self_test(void)
37287 {
37288     kmem_cache_t *test_cache;
37289
37290     printk(KERN_INFO "kmem_test() - start\n");
37291     test_cache =
37292         kmem_cache_create("test-cache", 16, 0,
37293             SLAB_RED_ZONE|SLAB_POISON,
37294             NULL, NULL);
37295     if (test_cache) {
37296         char *objp =
37297             kmem_cache_alloc(test_cache, SLAB_KERNEL);
37298         if (objp) {
37299             /* Write in front and past end, red zone test. */
37300             *(objp-1) = 1;
37301             *(objp+16) = 1;
37302             kmem_cache_free(test_cache, objp);
37303
37304             /* Mess up poisoning. */
37305             *objp = 10;
37306             objp = kmem_cache_alloc(test_cache, SLAB_KERNEL);
37307             kmem_cache_free(test_cache, objp);
37308
37309             /* Mess up poisoning (again). */
37310             *objp = 10;
37311             kmem_cache_shrink(test_cache);
37312         }
37313     }
37314     printk(KERN_INFO "kmem_test() - finished\n");
37315 }
37316 #endif /* SLAB_SELFTEST */
37317
37318 #if defined(CONFIG_PROC_FS)
37319 /* /proc/slabinfo
37320 * cache-name num-active-objs total-objs num-active-slabs
37321 * ... total-slabs num-pages-per-slab
37322 */
37323 int
37324 get_slabinfo(char *buf)
37325 {
37326     kmem_cache_t *cachep;
37327     kmem_slab_t *slabp;

```

```

37328     unsigned long active_objs;
37329     unsigned long save_flags;
37330     unsigned long num_slabs;
37331     unsigned long num_objs;
37332     int len=0;
37333 #if SLAB_STATS
37334     unsigned long active_slabs;
37335 #endif /* SLAB_STATS */
37336
37337     __save_flags(save_flags);
37338
37339     /* Output format version, so at least we can change it
37340     * without _too_ many complaints. */
37341     #if SLAB_STATS
37342     len = sprintf(buf,
37343         "slabinfo - version: 1.0 (statistics)\n");
37344     #else
37345     len = sprintf(buf, "slabinfo - version: 1.0\n");
37346     #endif /* SLAB_STATS */
37347     down(&cache_chain_sem);
37348     cachep = &cache_cache;
37349     do {
37350         #if SLAB_STATS
37351         active_slabs = 0;
37352         #endif /* SLAB_STATS */
37353         num_slabs = active_objs = 0;
37354         spin_lock_irq(&cachep->c_spinlock);
37355         for (slabp = cachep->c_firstp;
37356             slabp != kmem_slab_end(cachep);
37357             slabp = slabp->s_nextp) {
37358             active_objs += slabp->s_inuse;
37359             num_slabs++;
37360         #if SLAB_STATS
37361         if (slabp->s_inuse)
37362             active_slabs++;
37363         #endif /* SLAB_STATS */
37364     }
37365     num_objs = cachep->c_num*num_slabs;
37366     #if SLAB_STATS
37367     {
37368         unsigned long errors;
37369         unsigned long high = cachep->c_high_mark;
37370         unsigned long grown = cachep->c_grown;
37371         unsigned long reaped = cachep->c_reaped;
37372         unsigned long allocs = cachep->c_num_allocations;
37373         errors =
37374             (unsigned long) atomic_read(&cachep->c_errors);
37375         spin_unlock_irqrestore(&cachep->c_spinlock,

```

```

37376         save_flags);
37377     len += sprintf(buf+len,
37378         "%-16s %6lu %6lu %4lu %4lu %4lu "
37379         "%6lu %7lu %5lu %4lu %4lu\n",
37380         cachep->c_name, active_objs,
37381         num_objs, active_slabs, num_slabs,
37382         (1<<cachep->c_gfporder)*num_slabs,
37383         high, allocs, grown, reaped,errors);
37384     }
37385 #else
37386     spin_unlock_irqrestore(&cachep->c_spinlock,
37387         save_flags);
37388     len += sprintf(buf+len, "%-17s %6lu %6lu\n",
37389         cachep->c_name, active_objs,num_objs);
37390 #endif /* SLAB_STATS */
37391 } while ((cachep = cachep->c_nextp) != &cache_cache);
37392 up(&cache_chain_sem);
37393
37394 return len;
37395 }
37396 #endif /* CONFIG_PROC_FS */

```

mm/swap.c

```

37397 /*
37398  * linux/mm/swap.c
37399  *
37400  * Copyright (C) 1991, 1992, 1993, 1994 Linus Torvalds
37401  */
37402
37403 /* This file contains the default values for the
37404  * operation of the Linux VM subsystem. Fine-tuning
37405  * documentation can be found in
37406  * linux/Documentation/sysctl/vm.txt.
37407  * Started 18.12.91
37408  * Swap aging added 23.2.95, Stephen Tweedie.
37409  * Buffermem limits added 12.3.98, Rik van Riel.
37410  */
37411
37412 #include <linux/mm.h>
37413 #include <linux/kernel_stat.h>
37414 #include <linux/swap.h>
37415 #include <linux/swapctl.h>
37416 #include <linux/pagemap.h>
37417 #include <linux/init.h>
37418
37419 #include <asm/dma.h>
37420 #include <asm/uaccess.h> /* for copy_to/from_user */

```

```

37421 #include <asm/pgtable.h>
37422
37423 /* We identify three levels of free memory. We never let
37424  * free mem fall below the freepages.min except for
37425  * atomic allocations. We start background swapping if
37426  * we fall below freepages.high free pages, and we begin
37427  * intensive swapping below freepages.low.
37428  *
37429  * These values are there to keep GCC from
37430  * complaining. Actual initialization is done in
37431  * mm/page_alloc.c or arch/sparc(64)/mm/init.c. */
37432 freepages_t freepages = {
37433     48, /* freepages.min */
37434     96, /* freepages.low */
37435     144 /* freepages.high */
37436 };
37437
37438 /* How many pages do we try to swap or page in/out
37439  * together? */
37440 /* Default modified in swap_setup() */
37441 int page_cluster = 4;
37442
37443 /* We track the number of pages currently being
37444  * asynchronously swapped out, so that we don't try to
37445  * swap TOO many pages out at once */
37446 atomic_t nr_async_pages = ATOMIC_INIT(0);
37447
37448 buffer_mem_t buffer_mem = {
37449     2, /* minimum percent buffer */
37450     10, /* borrow percent buffer */
37451     60 /* maximum percent buffer */
37452 };
37453
37454 buffer_mem_t page_cache = {
37455     2, /* minimum percent page cache */
37456     15, /* borrow percent page cache */
37457     75 /* maximum */
37458 };
37459
37460 pager_daemon_t pager_daemon = {
37461     512, /* base # for calculating the number of tries */
37462     SWAP_CLUSTER_MAX, /* minimum number of tries */
37463     SWAP_CLUSTER_MAX, /* swap I/O in clusters of this sz */
37464 };
37465
37466 /* Perform any setup for the swap system */
37467
37468 void __init swap_setup(void)

```

```

37469 (
37470 /* Use a smaller cluster for memory <16MB or <32MB */
37471 if (num_physpages < ((16 * 1024 * 1024) >> PAGE_SHIFT))
37472     page_cluster = 2;
37473 else if (num_physpages <
37474         ((32 * 1024 * 1024) >> PAGE_SHIFT))
37475     page_cluster = 3;
37476 else
37477     page_cluster = 4;
37478 )

```

mm/swap_state.c

```

37479 /*
37480 * linux/mm/swap_state.c
37481 *
37482 * Copyright (C) 1991, 1992, 1993, 1994 Linus Torvalds
37483 * Swap reorganised 29.12.95, Stephen Tweedie
37484 *
37485 * Rewritten to use page cache. (C) 1998 Stephen Tweedie
37486 */
37487
37488 #include <linux/mm.h>
37489 #include <linux/kernel_stat.h>
37490 #include <linux/swap.h>
37491 #include <linux/swapctl.h>
37492 #include <linux/init.h>
37493 #include <linux/pagemap.h>
37494
37495 #include <asm/pgtable.h>
37496
37497 /* Keep a reserved false inode which we will use to mark
37498 * pages in the page cache are acting as swap cache
37499 * instead of file cache.
37500 *
37501 * We only need a unique pointer to satisfy the page
37502 * cache, but we'll reserve an entire zeroed inode
37503 * structure for the purpose just to ensure that any
37504 * mistaken dereferences of this structure cause a kernel
37505 * oops. */
37506 struct inode swapper_inode;
37507
37508 #ifdef SWAP_CACHE_INFO
37509 unsigned long swap_cache_add_total = 0;
37510 unsigned long swap_cache_del_total = 0;
37511 unsigned long swap_cache_find_total = 0;
37512 unsigned long swap_cache_find_success = 0;
37513

```

```

37514 void show_swap_cache_info(void)
37515 {
37516     printk("Swap cache: add %ld, delete %ld, "
37517           "find %ld/%ld\n",
37518           swap_cache_add_total,
37519           swap_cache_del_total,
37520           swap_cache_find_success, swap_cache_find_total);
37521 }
37522 #endif
37523
37524 int add_to_swap_cache(struct page *page,
37525                     unsigned long entry)
37526 {
37527     #ifdef SWAP_CACHE_INFO
37528     swap_cache_add_total++;
37529     #endif
37530     #ifdef DEBUG_SWAP
37531     printk("DebugVM: add to swap cache(%08lx count %d, "
37532           "entry %08lx)\n", page_address(page),
37533           atomic_read(&page->count), entry);
37534     #endif
37535     if (PageTestandSetSwapCache(page)) {
37536         printk(KERN_ERR
37537                "swap_cache: replacing non-empty entry %08lx "
37538                "on page %08lx\n",
37539                page->offset, page_address(page));
37540         return 0;
37541     }
37542     if (page->inode) {
37543         printk(KERN_ERR
37544                "swap cache: replacing page-cached entry "
37545                "on page %08lx\n", page_address(page));
37546         return 0;
37547     }
37548     atomic_inc(&page->count);
37549     page->inode = &swapper_inode;
37550     page->offset = entry;
37551     add_page_to_hash_queue(page, &swapper_inode, entry);
37552     add_page_to_inode_queue(&swapper_inode, page);
37553     return 1;
37554 }
37555
37556 /* Verify that a swap entry is valid and increment its
37557 * swap map count.
37558 *
37559 * Note: if swap map[] reaches SWAP_MAP_MAX the entries
37560 * are treated as "permanent", but will be reclaimed by
37561 * the next swapon. */

```

```

37562 int swap_duplicate(unsigned long entry)
37563 {
37564     struct swap_info_struct * p;
37565     unsigned long offset, type;
37566     int result = 0;
37567
37568     if (!entry)
37569         goto out;
37570     type = SWP_TYPE(entry);
37571     if (type & SHM_SWAP_TYPE)
37572         goto out;
37573     if (type >= nr_swapfiles)
37574         goto bad_file;
37575     p = type + swap_info;
37576     offset = SWP_OFFSET(entry);
37577     if (offset >= p->max)
37578         goto bad_offset;
37579     if (!p->swap_map[offset])
37580         goto bad_unused;
37581     /* Entry is valid, so increment the map count. */
37582     if (p->swap_map[offset] < SWAP_MAP_MAX)
37583         p->swap_map[offset]++;
37584     else {
37585         static int overflow = 0;
37586         if (overflow++ < 5)
37587             printk(KERN_WARNING
37588                 "swap_duplicate: entry %08lx map count=%d\n",
37589                 entry, p->swap_map[offset]);
37590         p->swap_map[offset] = SWAP_MAP_MAX;
37591     }
37592     result = 1;
37593 #ifdef DEBUG_SWAP
37594     printk("DebugVM: swap_duplicate(entry %08lx, "
37595         "count now %d)\n", entry, p->swap_map[offset]);
37596 #endif
37597 out:
37598     return result;
37599
37600 bad_file:
37601     printk(KERN_ERR "swap_duplicate: entry %08lx, "
37602         "nonexistent swap file\n", entry);
37603     goto out;
37604 bad_offset:
37605     printk(KERN_ERR "swap_duplicate: entry %08lx, "
37606         "offset exceeds max\n", entry);
37607     goto out;
37608 bad_unused:
37609     printk(KERN_ERR "swap_duplicate at %8p: "

```

```

37610         "entry %08lx, unused page\n",
37611         __builtin_return_address(0), entry);
37612     goto out;
37613 }
37614
37615 int swap_count(unsigned long entry)
37616 {
37617     struct swap_info_struct * p;
37618     unsigned long offset, type;
37619     int retval = 0;
37620
37621     if (!entry)
37622         goto bad_entry;
37623     type = SWP_TYPE(entry);
37624     if (type & SHM_SWAP_TYPE)
37625         goto out;
37626     if (type >= nr_swapfiles)
37627         goto bad_file;
37628     p = type + swap_info;
37629     offset = SWP_OFFSET(entry);
37630     if (offset >= p->max)
37631         goto bad_offset;
37632     if (!p->swap_map[offset])
37633         goto bad_unused;
37634     retval = p->swap_map[offset];
37635 #ifdef DEBUG_SWAP
37636     printk("DebugVM: swap count(entry %08lx, count %d)\n",
37637         entry, retval);
37638 #endif
37639 out:
37640     return retval;
37641
37642 bad_entry:
37643     printk(KERN_ERR "swap count: null entry!\n");
37644     goto out;
37645 bad_file:
37646     printk(KERN_ERR "swap_count: entry %08lx, "
37647         "nonexistent swap file!\n", entry);
37648     goto out;
37649 bad_offset:
37650     printk(KERN_ERR "swap count: entry %08lx, offset "
37651         "exceeds max!\n", entry);
37652     goto out;
37653 bad_unused:
37654     printk(KERN_ERR "swap_count at %8p: entry %08lx, "
37655         "unused page!\n", __builtin_return_address(0),
37656         entry);
37657     goto out;

```



```

37658 }
37659
37660 static inline void remove_from_swap_cache(
37661     struct page *page)
37662 {
37663     if (!page->inode) {
37664         printk("VM: Removing swap cache page with zero inode"
37665             " hash on page %08lx\n", page_address(page));
37666         return;
37667     }
37668     if (page->inode != &swapper_inode) {
37669         printk("VM: Removing swap cache page with wrong "
37670             "inode hash on page %08lx\n",
37671             page_address(page));
37672     }
37673
37674 #ifdef DEBUG_SWAP
37675     printk("DebugVM: remove from swap cache(%08lx "
37676         "count %d)\n",
37677         page_address(page), atomic_read(&page->count));
37678 #endif
37679     PageClearSwapCache (page);
37680     remove_inode_page(page);
37681 }
37682
37683
37684 /* This must be called only on pages that have been
37685  * verified to be in the swap cache. */
37686 void delete_from_swap_cache(struct page *page)
37687 {
37688     long entry = page->offset;
37689
37690 #ifdef SWAP_CACHE_INFO
37691     swap_cache_del_total++;
37692 #endif
37693 #ifdef DEBUG_SWAP
37694     printk("DebugVM: delete from swap cache(%08lx "
37695         "count %d, entry %08lx)\n",
37696         page_address(page), atomic_read(&page->count),
37697         entry);
37698 #endif
37699     remove_from_swap_cache (page);
37700     swap_free (entry);
37701 }
37702
37703 /* Perform a free page(), also freeing any swap cache
37704  * associated with this page if it is the last user of
37705  * the page. */

```

```

37706
37707 void free_page_and_swap_cache(unsigned long addr)
37708 {
37709     struct page *page = mem_map + MAP_NR(addr);
37710
37711     /* If we are the only user, then free up the swap
37712      * cache. */
37713     if (PageSwapCache(page) && !is_page_shared(page)) {
37714         delete_from_swap_cache(page);
37715     }
37716     __free_page(page);
37717 }
37718
37719
37720 /* Lookup a swap entry in the swap cache. We need to be
37721  * careful about locked pages. A found page will be
37722  * returned with its refcount incremented. */
37723 struct page *lookup_swap_cache(unsigned long entry)
37724 {
37725     struct page *found;
37726
37727 #ifdef SWAP_CACHE_INFO
37728     swap_cache_find_total++;
37729 #endif
37730     while (1) {
37731         found = find_page(&swapper_inode, entry);
37732         if (!found)
37733             return 0;
37734         if (found->inode != &swapper_inode ||
37735             !PageSwapCache(found))
37736             goto out_bad;
37737         if (!PageLocked(found)) {
37738 #ifdef SWAP_CACHE_INFO
37739             swap_cache_find_success++;
37740 #endif
37741             return found;
37742         }
37743         __free_page(found);
37744         wait_on_page(found);
37745     }
37746
37747 out_bad:
37748     printk(KERN_ERR
37749         "VM: Found a non-swapper swap page!\n");
37750     free_page(found);
37751     return 0;
37752 }
37753

```

```

37754 /* Locate a page of swap in physical memory, reserving
37755 * swap cache space and reading the disk if it is not
37756 * already cached. If wait==0, we are only doing
37757 * readahead, so don't worry if the page is already
37758 * locked.
37759 *
37760 * A failure return means that either the page allocation
37761 * failed or that the swap entry is no longer in use. */
37762 struct page * read_swap_cache_async(unsigned long entry,
37763                                     int wait)
37764 {
37765     struct page *found_page = 0, *new_page;
37766     unsigned long new_page_addr;
37767
37768 #ifdef DEBUG_SWAP
37769     printk("DebugVM: read_swap_cache_async entry %08lx%s\n"
37770           ", entry, wait ? ", wait" : "");
37771 #endif
37772     /* Make sure the swap entry is still in use. */
37773     /* Account for the swap cache */
37774     if (!swap_duplicate(entry))
37775         goto out;
37776     /* Look for the page in the swap cache. */
37777     found_page = lookup_swap_cache(entry);
37778     if (found_page)
37779         goto out_free_swap;
37780
37781     new_page_addr = __get_free_page(GFP_USER);
37782     if (!new_page_addr)
37783         goto out_free_swap; /* Out of memory */
37784     new_page = mem_map + MAP_NR(new_page_addr);
37785
37786     /* Check the swap cache again, in case we stalled
37787      * above. */
37788     found_page = lookup_swap_cache(entry);
37789     if (found_page)
37790         goto out_free_page;
37791     /* Add it to the swap cache and read its contents. */
37792     if (!add_to_swap_cache(new_page, entry))
37793         goto out_free_page;
37794
37795     set_bit(PG_locked, &new_page->flags);
37796     rw_swap_page(READ, entry, (char *)new_page_addr, wait);
37797 #ifdef DEBUG_SWAP
37798     printk("DebugVM: read_swap_cache_async created "
37799           "entry %08lx at %p\n",
37800           entry, (char *) page_address(new_page));
37801 #endif

```

```

37802     return new_page;
37803
37804 out_free_page:
37805     __free_page(new_page);
37806 out_free_swap:
37807     swap_free(entry);
37808 out:
37809     return found_page;
37810 }

```

mm/swapfile.c

```

37811 /*
37812 * linux/mm/swapfile.c
37813 *
37814 * Copyright (C) 1991, 1992, 1993, 1994 Linus Torvalds
37815 * Swap reorganised 29.12.95, Stephen Tweedie
37816 */
37817
37818 #include <linux/malloc.h>
37819 #include <linux/smp_lock.h>
37820 #include <linux/kernel_stat.h>
37821 #include <linux/swap.h>
37822 #include <linux/swapctl.h>
37823 #include <linux/blkdev.h> /* for blk_size */
37824 #include <linux/vmalloc.h>
37825 #include <linux/pagemap.h>
37826 #include <linux/shm.h>
37827
37828 #include <asm/pgtable.h>
37829
37830 unsigned int nr_swapfiles = 0;
37831
37832 struct swap_list_t swap_list = {-1, -1};
37833
37834 struct swap_info_struct swap_info[MAX_SWAPFILES];
37835
37836 #define SWAPFILE_CLUSTER 256
37837
37838 static inline int scan_swap_map(
37839     struct swap_info_struct *si)
37840 {
37841     unsigned long offset;
37842     /* We try to cluster swap pages by allocating them
37843      * sequentially in swap. Once we've allocated
37844      * SWAPFILE_CLUSTER pages this way, however, we resort
37845      * to first-free allocation, starting a new cluster.
37846      * This prevents us from scattering swap pages all over

```

```

37847  * the entire swap partition, so that we reduce overall
37848  * disk seek times between swap pages.  -- sct */
37849  if (si->cluster_nr) {
37850      while (si->cluster_next <= si->highest_bit) {
37851          offset = si->cluster_next++;
37852          if (si->swap_map[offset])
37853              continue;
37854          if (test_bit(offset, si->swap_lockmap))
37855              continue;
37856          si->cluster_nr--;
37857          goto got_page;
37858      }
37859  }
37860  si->cluster_nr = SWAPFILE_CLUSTER;
37861  for (offset = si->lowest_bit;
37862       offset <= si->highest_bit ; offset++) {
37863      if (si->swap_map[offset])
37864          continue;
37865      if (test_bit(offset, si->swap_lockmap))
37866          continue;
37867      si->lowest_bit = offset;
37868  got_page:
37869      si->swap_map[offset] = 1;
37870      nr_swap_pages--;
37871      if (offset == si->highest_bit)
37872          si->highest_bit--;
37873      si->cluster_next = offset;
37874      return offset;
37875  }
37876  return 0;
37877 }
37878
37879 unsigned long get_swap_page(void)
37880 {
37881     struct swap_info_struct * p;
37882     unsigned long offset, entry;
37883     int type, wrapped = 0;
37884
37885     type = swap_list.next;
37886     if (type < 0)
37887         return 0;
37888     if (nr_swap_pages == 0)
37889         return 0;
37890
37891     while (1) {
37892         p = &swap_info[type];
37893         if ((p->flags & SWP_WRITEOK) == SWP_WRITEOK) {
37894             offset = scan_swap_map(p);

```

```

37895         if (offset) {
37896             entry = SWP_ENTRY(type,offset);
37897             type = swap_info[type].next;
37898             if (type < 0) {
37899                 p->prio != swap_info[type].prio
37900             }
37901                 swap_list.next = swap_list.head;
37902             }
37903             else
37904             {
37905                 swap_list.next = type;
37906             }
37907             return entry;
37908         }
37909     }
37910     type = p->next;
37911     if (!wrapped) {
37912         if (type < 0 || p->prio != swap_info[type].prio) {
37913             type = swap_list.head;
37914             wrapped = 1;
37915         }
37916     } else if (type < 0) {
37917         return 0; /* out of swap space */
37918     }
37919 }
37920 }
37921
37922
37923 void swap_free(unsigned long entry)
37924 {
37925     struct swap_info_struct * p;
37926     unsigned long offset, type;
37927
37928     if (!entry)
37929         goto out;
37930
37931     type = SWP_TYPE(entry);
37932     if (type & SHM_SWAP_TYPE)
37933         goto out;
37934     if (type >= nr_swapfiles)
37935         goto bad_nofile;
37936     p = &swap_info[type];
37937     if (!(p->flags & SWP_USED))
37938         goto bad_device;
37939     if (p->prio > swap_info[swap_list.next].prio)
37940         swap_list.next = swap_list.head;
37941     offset = SWP_OFFSET(entry);
37942     if (offset >= p->max)

```

```

37943     goto bad_offset;
37944     if (offset < p->lowest_bit)
37945         p->lowest_bit = offset;
37946     if (offset > p->highest_bit)
37947         p->highest_bit = offset;
37948     if (!p->swap_map[offset])
37949         goto bad_free;
37950     if (p->swap_map[offset] < SWAP_MAP_MAX) {
37951         if (!--p->swap_map[offset])
37952             nr_swap_pages++;
37953     }
37954 #ifdef DEBUG_SWAP
37955     printk("DebugVM: "
37956           "swap_free(entry %08lx, count now %d)\n",
37957           entry, p->swap_map[offset]);
37958 #endif
37959 out:
37960     return;
37961 bad_nofile:
37962     printk("swap_free: "
37963           "Trying to free nonexistent swap-page\n");
37964     goto out;
37965 bad_device:
37966     printk("swap_free: "
37967           "Trying to free swap from unused swap-device\n");
37968     goto out;
37969 bad_offset:
37970     printk("swap_free: offset exceeds max\n");
37971     goto out;
37972 bad_free:
37973     printk("swap_free: "
37974           "swap-space map bad (entry %08lx)\n",entry);
37975     goto out;
37976 }
37977 }
37978
37979 /* The swap entry has been read in advance, and we return
37980 * 1 to indicate that the page has been used or is no
37981 * longer needed.
37982 *
37983 * Always set the resulting pte to be nowrite (the same
37984 * as COW pages after one process has exited). We don't
37985 * know just how many PTEs will share this swap entry, so
37986 * be cautious and let do_wp_page work out what to do if
37987 * a write is requested later. */
37988 static inline void unuse_pte(struct vm_area_struct * vma,
37989 unsigned long address, pte_t *dir, unsigned long entry,
37990 unsigned long page)

```

```

37991 {
37992     pte_t pte = *dir;
37993
37994     if (pte_none(pte))
37995         return;
37996     if (pte_present(pte)) {
37997         /* If this entry is swap-cached, then page must
37998          * already hold the right address for any copies in
37999          * physical memory */
38000         if (pte_page(pte) != page)
38001             return;
38002         /* We will be removing the swap cache in a moment,
38003          * so... */
38004         set_pte(dir, pte_mkdirty(pte));
38005         return;
38006     }
38007     if (pte_val(pte) != entry)
38008         return;
38009     set_pte(dir, pte_mkdirty(mk_pte(page,
38010                                   vma->vm_page_prot)));
38011     swap_free(entry);
38012     atomic_inc(&mem_map[MAP_NR(page)].count);
38013     ++vma->vm_mm->rss;
38014 }
38015
38016 static inline void unuse_pmd(struct vm_area_struct * vma,
38017 pmd_t *dir, unsigned long address, unsigned long size,
38018 unsigned long offset, unsigned long entry,
38019 unsigned long page)
38020 {
38021     pte_t * pte;
38022     unsigned long end;
38023
38024     if (pmd_none(*dir))
38025         return;
38026     if (pmd_bad(*dir)) {
38027         printk("unuse pmd: bad pmd (%08lx)\n",pmd_val(*dir));
38028         pmd_clear(dir);
38029         return;
38030     }
38031     pte = pte_offset(dir, address);
38032     offset += address & PMD_MASK;
38033     address &= ~PMD_MASK;
38034     end = address + size;
38035     if (end > PMD_SIZE)
38036         end = PMD_SIZE;
38037     do {
38038         unuse_pte(vma, offset+address-vma->vm_start, pte,

```

```

38039         entry, page);
38040     address += PAGE_SIZE;
38041     pte++;
38042 } while (address < end);
38043 }
38044
38045 static inline void unuse_pgd(struct vm_area_struct * vma,
38046     pgd_t *dir, unsigned long address, unsigned long size,
38047     unsigned long entry, unsigned long page)
38048 {
38049     pmd_t * pmd;
38050     unsigned long offset, end;
38051
38052     if (pgd_none(*dir))
38053         return;
38054     if (pgd_bad(*dir)) {
38055         printk("unuse_pgd: bad pgd (%08lx)\n", pgd_val(*dir));
38056         pgd_clear(dir);
38057         return;
38058     }
38059     pmd = pmd_offset(dir, address);
38060     offset = address & PGDIR_MASK;
38061     address &= ~PGDIR_MASK;
38062     end = address + size;
38063     if (end > PGDIR_SIZE)
38064         end = PGDIR_SIZE;
38065     do {
38066         unuse_pmd(vma, pmd, address, end - address, offset,
38067             entry, page);
38068         address = (address + PMD_SIZE) & PMD_MASK;
38069         pmd++;
38070     } while (address < end);
38071 }
38072
38073 static void unuse_vma(struct vm_area_struct * vma,
38074     pgd_t *pgdir, unsigned long entry, unsigned long page)
38075 {
38076     unsigned long start = vma->vm_start, end = vma->vm_end;
38077
38078     while (start < end) {
38079         unuse_pgd(vma, pgdir, start, end - start, entry,
38080             page);
38081         start = (start + PGDIR_SIZE) & PGDIR_MASK;
38082         pgdir++;
38083     }
38084 }
38085
38086 static void unuse_process(struct mm_struct * mm,

```

```

38087     unsigned long entry, unsigned long page)
38088 {
38089     struct vm_area_struct* vma;
38090
38091     /* Go through process's page directory. */
38092     if (!mm || mm == &init_mm)
38093         return;
38094     for (vma = mm->mmap; vma; vma = vma->vm_next) {
38095         pgd_t * pgd = pgd_offset(mm, vma->vm_start);
38096         unuse_vma(vma, pgd, entry, page);
38097     }
38098     return;
38099 }
38100
38101 /* We completely avoid races by reading each swap page in
38102 * advance, and then search for the process using it.
38103 * All the necessary page table adjustments can then be
38104 * made atomically. */
38105 static int try_to_unuse(unsigned int type)
38106 {
38107     struct swap_info_struct * si = &swap_info[type];
38108     struct task_struct *p;
38109     struct page *page_map;
38110     unsigned long entry, page;
38111     int i;
38112
38113     while (1) {
38114         /* Find a swap page in use and read it in. */
38115         for (i = 1; i < si->max; i++) {
38116             if (si->swap_map[i] > 0 &&
38117                 si->swap_map[i] != SWAP_MAP_BAD) {
38118                 goto found_entry;
38119             }
38120         }
38121         break;
38122
38123     found_entry:
38124         entry = SWP_ENTRY(type, i);
38125
38126         /* Get a page for the entry, using the existing swap
38127          * cache page if there is one. Otherwise, get a
38128          * clean page and read the swap into it. */
38129         page_map = read_swap_cache(entry);
38130         if (!page_map) {
38131             /* Continue searching if entry became unused. */
38132             if (si->swap_map[i] == 0)
38133                 continue;
38134             return -ENOMEM;

```

```

38135     }
38136     page = page_address(page_map);
38137     read_lock(&tasklist_lock);
38138     for_each_task(p)
38139         unuse_process(p->mm, entry, page);
38140     read_unlock(&tasklist_lock);
38141     shm_unuse(entry, page);
38142     /* Now get rid of the extra reference to the
38143      * temporary page we've been using. */
38144     if (PageSwapCache(page_map))
38145         delete_from_swap_cache(page_map);
38146     __free_page(page_map);
38147     /* Check for and clear any overflowed swap map
38148      * counts. */
38149     if (si->swap_map[i] != 0) {
38150         if (si->swap_map[i] != SWAP_MAP_MAX)
38151             printk(KERN_ERR
38152                 "try_to_unuse: entry %08lx count=%d\n",
38153                 entry, si->swap_map[i]);
38154         si->swap_map[i] = 0;
38155         nr_swap_pages++;
38156     }
38157 }
38158 return 0;
38159 }
38160
38161 asmlinkage int sys_swapon(const char * specialfile)
38162 {
38163     struct swap_info_struct * p = NULL;
38164     struct dentry * dentry;
38165     struct file filp;
38166     int i, type, prev;
38167     int err = -EPERM;
38168
38169     lock_kernel();
38170     if (!capable(CAP_SYS_ADMIN))
38171         goto out;
38172
38173     dentry = namei(specialfile);
38174     err = PTR_ERR(dentry);
38175     if (IS_ERR(dentry))
38176         goto out;
38177
38178     prev = -1;
38179     for (type = swap_list.head; type >= 0;
38180         type = swap_info[type].next) {
38181         p = swap_info + type;
38182         if ((p->flags & SWP_WRITEOK) == SWP_WRITEOK) {

```

```

38183         if (p->swap_file) {
38184             if (p->swap_file == dentry)
38185                 break;
38186         } else {
38187             if (S_ISBLK(dentry->d_inode->i_mode) &&
38188                 (p->swap_device == dentry->d_inode->i_rdev))
38189                 break;
38190         }
38191     }
38192     prev = type;
38193 }
38194 err = EINVAL;
38195 if (type < 0)
38196     goto out_dput;
38197
38198 if (prev < 0) {
38199     swap_list.head = p->next;
38200 } else {
38201     swap_info[prev].next = p->next;
38202 }
38203 if (type == swap_list.next) {
38204     /* just pick something that's safe... */
38205     swap_list.next = swap_list.head;
38206 }
38207 p->flags = SWP_USED;
38208 err = try_to_unuse(type);
38209 if (err) {
38210     /* re-insert swap space back into swap_list */
38211     for (prev = -1, i = swap_list.head; i >= 0;
38212         prev = i, i = swap_info[i].next)
38213         if (p->prio >= swap_info[i].prio)
38214             break;
38215     p->next = i;
38216     if (prev < 0)
38217         swap_list.head = swap_list.next + p - swap_info;
38218     else
38219         swap_info[prev].next = p - swap_info;
38220     p->flags = SWP_WRITEOK;
38221     goto out_dput;
38222 }
38223 if (p->swap_device) {
38224     memset(&filp, 0, sizeof(filp));
38225     filp.f_dentry = dentry;
38226     filp.f_mode = 3; /* read write */
38227     /* open it again to get fops */
38228     if (!blkdev_open(dentry->d_inode, &filp) &&
38229         filp.f_op && filp.f_op->release) {
38230         filp.f_op->release(dentry->d_inode, &filp);

```

```

38231     filp_f_op->release(dentry->d_inode,&filp);
38232     }
38233 }
38234 dput(dentry);
38235
38236 dentry = p->swap_file;
38237 p->swap_file = NULL;
38238 nr_swap_pages -= p->pages;
38239 p->swap_device = 0;
38240 vfree(p->swap_map);
38241 p->swap_map = NULL;
38242 vfree(p->swap_lockmap);
38243 p->swap_lockmap = NULL;
38244 p->flags = 0;
38245 err = 0;
38246
38247 out_dput:
38248 dput(dentry);
38249 out:
38250 unlock_kernel();
38251 return err;
38252 }
38253
38254 int get_swaparea_info(char *buf)
38255 {
38256     char * page = (char *) __get_free_page(GFP_KERNEL);
38257     struct swap_info_struct *ptr = swap_info;
38258     int i, j, len = 0, usedswap;
38259
38260     if (!page)
38261         return -ENOMEM;
38262
38263     len += sprintf(buf, "Filename\t\t\tType\t\tSize\tUsed"
38264                    "\tPriority\n");
38265     for (i = 0 ; i < nr_swapfiles ; i++, ptr++) {
38266         if (ptr->flags & SWP_USED) {
38267             char * path = d_path(ptr->swap_file, page,
38268                                PAGE_SIZE);
38269
38270             len += sprintf(buf + len, "%-31s ", path);
38271
38272             if (!ptr->swap_device)
38273                 len += sprintf(buf + len, "file\t\t");
38274             else
38275                 len += sprintf(buf + len, "partition\t");
38276
38277             usedswap = 0;
38278             for (j = 0; j < ptr->max; ++j)

```

→ p 555

```

38279         switch (ptr->swap_map[j]) {
38280             case SWAP_MAP_BAD:
38281                 case 0:
38282                     continue;
38283             default:
38284                 usedswap++;
38285         }
38286         len +=
38287             sprintf(buf + len, "%d\t%d\t%d\n",
38288                    ptr->pages << (PAGE_SHIFT - 10),
38289                    usedswap << (PAGE_SHIFT - 10),ptr->prio);
38290     }
38291 }
38292 free_page((unsigned long) page);
38293 return len;
38294 }
38295
38296 /* Written 01/25/92 by Simmule Turner, heavily changed by
38297  * Linus.
38298  *
38299  * The swapon system call */
38300 asmlinkage int sys_swapon(const char * specialfile,
38301                          int swap_flags)
38302 {
38303     struct swap_info_struct * p;
38304     struct dentry * swap_dentry;
38305     unsigned int type;
38306     int i, j, prev;
38307     int error = -EPERM;
38308     struct file filp;
38309     static int least_priority = 0;
38310     union swap_header *swap_header = 0;
38311     int swap_header_version;
38312     int lock_map_size = PAGE_SIZE;
38313     int nr_good_pages = 0;
38314     unsigned long tmp_lock_map = 0;
38315
38316     lock_kernel();
38317     if (!capable(CAP_SYS_ADMIN))
38318         goto out;
38319     memset(&filp, 0, sizeof(filp));
38320     p = swap_info;
38321     for (type = 0 ; type < nr_swapfiles ; type++,p++)
38322         if (!(p->flags & SWP_USED))
38323             break;
38324     if (type >= MAX_SWAPFILES)
38325         goto out;
38326     if (type >= nr_swapfiles)

```

```

38327     nr_swapfiles = type+1;
38328     p->flags = SWP_USED;
38329     p->swap_file = NULL;
38330     p->swap_device = 0;
38331     p->swap_map = NULL;
38332     p->swap_lockmap = NULL;
38333     p->lowest_bit = 0;
38334     p->highest_bit = 0;
38335     p->cluster_nr = 0;
38336     p->max = 1;
38337     p->next = -1;
38338     if (swap_flags & SWAP_FLAG_PREFER) {
38339         p->prio = (swap_flags & SWAP_FLAG_PRIO_MASK)
38340             >> SWAP_FLAG_PRIO_SHIFT;
38341     } else {
38342         p->prio = --least_priority;
38343     }
38344     swap_dentry = namei(specialfile);
38345     error = PTR_ERR(swap_dentry);
38346     if (IS_ERR(swap_dentry))
38347         goto bad_swap_2;
38348
38349     p->swap_file = swap_dentry;
38350     error = -EINVAL;
38351
38352     if (S_ISBLK(swap_dentry->d_inode->i_mode)) {
38353         p->swap_device = swap_dentry->d_inode->i_rdev;
38354         set_blocksize(p->swap_device, PAGE_SIZE);
38355
38356         filp.f_dentry = swap_dentry;
38357         filp.f_mode = 3; /* read write */
38358         error = blkdev_open(swap_dentry->d_inode, &filp);
38359         if (error)
38360             goto bad_swap_2;
38361         set_blocksize(p->swap_device, PAGE_SIZE);
38362         error = -ENODEV;
38363         if (!p->swap_device ||
38364             (blk_size[MAJOR(p->swap_device)] &&
38365              !blk_size[MAJOR(p->swap_device)]
38366               [MINOR(p->swap_device)]))
38367             goto bad_swap;
38368         error = -EBUSY;
38369         for (i = 0 ; i < nr_swapfiles ; i++) {
38370             if (i == type)
38371                 continue;
38372             if (p->swap_device == swap_info[i].swap_device)
38373                 goto bad_swap;
38374         }

```

```

38375     } else if (S_ISREG(swap_dentry->d_inode->i_mode)) {
38376         error = -EBUSY;
38377         for (i = 0 ; i < nr_swapfiles ; i++) {
38378             if (i == type)
38379                 continue;
38380             if (swap_dentry->d_inode ==
38381                 swap_info[i].swap_file->d_inode)
38382                 goto bad_swap;
38383         }
38384     } else
38385         goto bad_swap;
38386
38387     swap_header = (void *) __get_free_page(GFP_USER);
38388     if (!swap_header) {
38389         printk("Unable to start swapping: "
38390             "out of memory :-)\n");
38391         error = -ENOMEM;
38392         goto bad_swap;
38393     }
38394
38395     p->swap_lockmap = (char *) &tmp_lock_map;
38396     rw_swap_page_nocache(READ, SWP_ENTRY(type,0),
38397         (char *) swap_header);
38398     p->swap_lockmap = NULL;
38399
38400     if (!memcmp("SWAP-SPACE", swap_header->magic.magic,10))
38401         swap_header_version = 1;
38402     else if (!memcmp("SWAPSPACE2",
38403         swap_header->magic.magic, 10))
38404         swap_header_version = 2;
38405     else {
38406         printk("Unable to find swap space signature\n");
38407         error = -EINVAL;
38408         goto bad_swap;
38409     }
38410
38411     switch (swap_header_version) {
38412     case 1:
38413         memset(((char *) swap_header)+PAGE_SIZE-10,0,10);
38414         j = 0;
38415         p->lowest_bit = 0;
38416         p->highest_bit = 0;
38417         for (i = 1 ; i < 8*PAGE_SIZE ; i++) {
38418             if (test_bit(i,(char *) swap_header)) {
38419                 if (!p->lowest_bit)
38420                     p->lowest_bit = i;
38421                 p->highest_bit = i;
38422                 p->max = i+1;

```



```

38423     j++;
38424     }
38425     }
38426     nr_good_pages = j;
38427     p->swap_map = vmalloc(p->max * sizeof(short));
38428     if (!p->swap_map) {
38429         error = -ENOMEM;
38430         goto bad_swap;
38431     }
38432     for (i = 1; i < p->max; i++) {
38433         if (test_bit(i, (char *) swap_header))
38434             p->swap_map[i] = 0;
38435         else
38436             p->swap_map[i] = SWAP_MAP_BAD;
38437     }
38438     break;
38439
38440 case 2:
38441     /* Check the swap header's sub version and the size
38442      * of the swap file and bad block lists */
38443     if (swap_header->info.version != 1) {
38444         printk(KERN_WARNING
38445             "Unable to handle swap header version %d\n",
38446             swap_header->info.version);
38447         error = -EINVAL;
38448         goto bad_swap;
38449     }
38450
38451     p->lowest_bit = 1;
38452     p->highest_bit = swap_header->info.last_page - 1;
38453     p->max = swap_header->info.last_page;
38454
38455     error = -EINVAL;
38456     if (swap_header->info.nr_badpages >
38457         MAX_SWAP_BADPAGES)
38458         goto bad_swap;
38459     if (p->max >= SWP_OFFSET(SWP_ENTRY(0, -0UL)))
38460         goto bad_swap;
38461
38462     /* OK, set up the swap map and apply the bad block
38463      * list */
38464     if (!(p->swap_map =
38465         vmalloc(p->max * sizeof(short)))) {
38466         error = -ENOMEM;
38467         goto bad_swap;
38468     }
38469
38470     error = 0;

```

```

38471     memset(p->swap_map, 0, p->max * sizeof(short));
38472     for (i=0; i<swap_header->info.nr_badpages; i++) {
38473         int page = swap_header->info.badpages[i];
38474         if (page <= 0 ||
38475             page >= swap_header->info.last_page)
38476             error = -EINVAL;
38477         else
38478             p->swap_map[page] = SWAP_MAP_BAD;
38479     }
38480     nr_good_pages = swap_header->info.last_page - i;
38481     lock_map_size = (p->max + 7) / 8;
38482     if (error)
38483         goto bad_swap;
38484 }
38485
38486 if (!nr_good_pages) {
38487     printk(KERN_WARNING "Empty swap-file\n");
38488     error = -EINVAL;
38489     goto bad_swap;
38490 }
38491 p->swap_map[0] = SWAP_MAP_BAD;
38492 if (!(p->swap_lockmap = vmalloc(lock_map_size))) {
38493     error = -ENOMEM;
38494     goto bad_swap;
38495 }
38496 memset(p->swap_lockmap, 0, lock_map_size);
38497 p->flags = SWP_WRITEOK;
38498 p->pages = nr_good_pages;
38499 nr_swap_pages += nr_good_pages;
38500 printk(KERN_INFO
38501     "Adding Swap: %dk swap-space (priority %d)\n",
38502     nr_good_pages<<(PAGE_SHIFT-10), p->prio);
38503
38504 /* insert swap space into swap list: */
38505 prev = -1;
38506 for (i = swap_list.head; i >= 0;
38507     i = swap_info[i].next) {
38508     if (p->prio >= swap_info[i].prio) {
38509         break;
38510     }
38511     prev = i;
38512 }
38513 p->next = i;
38514 if (prev < 0) {
38515     swap_list.head = swap_list.next = p->swap_info;
38516 } else {
38517     swap_info[prev].next = p->swap_info;
38518 }

```

```

38519 error = 0;
38520 goto out;
38521 bad_swap:
38522 if(filp.f_op && filp.f_op->release)
38523     filp.f_op->release(filp.f_dentry->d_inode,&filp);
38524 bad_swap_2:
38525 if (p->swap_lockmap)
38526     vfree(p->swap_lockmap);
38527 if (p->swap_map)
38528     vfree(p->swap_map);
38529 dput(p->swap_file);
38530 p->swap_device = 0;
38531 p->swap_file = NULL;
38532 p->swap_map = NULL;
38533 p->swap_lockmap = NULL;
38534 p->flags = 0;
38535 if (!(swap_flags & SWAP_FLAG_PREFER))
38536     ++least_priority;
38537 out:
38538 if (swap_header)
38539     free_page((long) swap_header);
38540 unlock_kernel();
38541 return error;
38542 }
38543
38544 void si_swapinfo(struct sysinfo *val)
38545 {
38546     unsigned int i, j;
38547
38548     val->freeswap = val->totalswap = 0;
38549     for (i = 0; i < nr_swapfiles; i++) {
38550         if ((swap_info[i].flags & SWP_WRITEOK) !=
38551             SWP_WRITEOK)
38552             continue;
38553         for (j = 0; j < swap_info[i].max; ++j)
38554             switch (swap_info[i].swap_map[j]) {
38555                 case SWAP_MAP_BAD:
38556                     continue;
38557                 case 0:
38558                     ++val->freeswap;
38559                 default:
38560                     ++val->totalswap;
38561             }
38562     }
38563     val->freeswap <<= PAGE_SHIFT;
38564     val->totalswap <<= PAGE_SHIFT;
38565     return;
38566 }

```

mm/vmalloc.c

```

38567 /*
38568  * linux/mm/vmalloc.c
38569  *
38570  * Copyright (C) 1993 Linus Torvalds
38571  */
38572
38573 #include <linux/malloc.h>
38574 #include <linux/vmalloc.h>
38575
38576 #include <asm/uaccess.h>
38577
38578 static struct vm_struct * vmlist = NULL;
38579
38580 static inline void free_area_pte(pmd_t * pmd,
38581     unsigned long address, unsigned long size)
38582 {
38583     pte_t * pte;
38584     unsigned long end;
38585
38586     if (pmd_none(*pmd))
38587         return;
38588     if (pmd_bad(*pmd)) {
38589         printk("free_area_pte: bad pmd (%08lx)\n",
38590             pmd_val(*pmd));
38591         pmd_clear(pmd);
38592         return;
38593     }
38594     pte = pte_offset(pmd, address);
38595     address &= ~PMD_MASK;
38596     end = address + size;
38597     if (end > PMD_SIZE)
38598         end = PMD_SIZE;
38599     while (address < end) {
38600         pte_t page = *pte;
38601         pte_clear(pte);
38602         address += PAGE_SIZE;
38603         pte++;
38604         if (pte_none(page))
38605             continue;
38606         if (pte_present(page)) {
38607             free_page(pte_page(page));
38608             continue;
38609         }
38610         printk("Whee.. Swapped out page in "
38611             "kernel page table\n");
38612     }

```

```

38613 }
38614
38615 static inline void free_area_pmd(pgd_t * dir,
38616 unsigned long address, unsigned long size)
38617 {
38618     pmd_t * pmd;
38619     unsigned long end;
38620
38621     if (pgd_none(*dir))
38622         return;
38623     if (pgd_bad(*dir)) {
38624         printk("free_area_pmd: bad pgd (%08lx)\n",
38625             pgd_val(*dir));
38626         pgd_clear(dir);
38627         return;
38628     }
38629     pmd = pmd_offset(dir, address);
38630     address &= ~PGDIR_MASK;
38631     end = address + size;
38632     if (end > PGDIR_SIZE)
38633         end = PGDIR_SIZE;
38634     while (address < end) {
38635         free_area_pte(pmd, address, end - address);
38636         address = (address + PMD_SIZE) & PMD_MASK;
38637         pmd++;
38638     }
38639 }
38640
38641 void vmfree_area_pages(unsigned long address,
38642 unsigned long size)
38643 {
38644     pgd_t * dir;
38645     unsigned long end = address + size;
38646
38647     dir = pgd_offset_k(address);
38648     flush_cache_all();
38649     while (address < end) {
38650         free_area_pmd(dir, address, end - address);
38651         address = (address + PGDIR_SIZE) & PGDIR_MASK;
38652         dir++;
38653     }
38654     flush_tlb_all();
38655 }
38656
38657 static inline int alloc_area_pte(pte_t * pte,
38658 unsigned long address, unsigned long size)
38659 {
38660     unsigned long end;

```

```

38661
38662     address &= ~PMD_MASK;
38663     end = address + size;
38664     if (end > PMD_SIZE)
38665         end = PMD_SIZE;
38666     while (address < end) {
38667         unsigned long page;
38668         if (!pte_none(*pte))
38669             printk("alloc_area_pte: page already exists\n");
38670         page = __get_free_page(GFP_KERNEL);
38671         if (!page)
38672             return -ENOMEM;
38673         set_pte(pte, mk_pte(page, PAGE_KERNEL));
38674         address += PAGE_SIZE;
38675         pte++;
38676     }
38677     return 0;
38678 }
38679
38680 static inline int alloc_area_pmd(pmd_t * pmd,
38681 unsigned long address, unsigned long size)
38682 {
38683     unsigned long end;
38684
38685     address &= ~PGDIR_MASK;
38686     end = address + size;
38687     if (end > PGDIR_SIZE)
38688         end = PGDIR_SIZE;
38689     while (address < end) {
38690         pte_t * pte = pte_alloc_kernel(pmd, address);
38691         if (!pte)
38692             return -ENOMEM;
38693         if (alloc_area_pte(pte, address, end - address))
38694             return -ENOMEM;
38695         address = (address + PMD_SIZE) & PMD_MASK;
38696         pmd++;
38697     }
38698     return 0;
38699 }
38700
38701 int vmalloc_area_pages(unsigned long address,
38702 unsigned long size)
38703 {
38704     pgd_t * dir;
38705     unsigned long end = address + size;
38706
38707     dir = pgd_offset_k(address);
38708     flush_cache_all();

```

```

38709 while (address < end) (
38710     pmd_t *pmd;
38711     pgd_t olddir = *dir;
38712
38713     pmd = pmd_alloc_kernel(dir, address);
38714     if (!pmd)
38715         return -ENOMEM;
38716     if (alloc_area_pmd(pmd, address, end - address))
38717         return -ENOMEM;
38718     if (pgd_val(olddir) != pgd_val(*dir))
38719         set_pgdire(address, *dir);
38720     address = (address + PGDIR_SIZE) & PGDIR_MASK;
38721     dir++;
38722 }
38723 flush_tlb_all();
38724 return 0;
38725 }
38726
p 563 38727 struct vm_struct * get_vm_area(unsigned long size)
38728 {
38729     unsigned long addr;
38730     struct vm_struct **p, *tmp, *area;
38731
p 563 38732     area = (struct vm_struct *) kmalloc(sizeof(*area),
38733                                         GFP_KERNEL);
38734     if (!area)
38735         return NULL;
38736     addr = VMALLOC_START;
38737     for (p = &vmlist; (tmp = *p); p = &tmp->next) {
38738         if (size + addr < (unsigned long) tmp->addr)
38739             break;
38740         if (addr > VMALLOC_END-size) {
38741             kfree(area);
38742             return NULL;
38743         }
38744         addr = tmp->size + (unsigned long) tmp->addr;
38745     }
38746     area->addr = (void *)addr;
38747     area->size = size + PAGE_SIZE;
38748     area->next = *p;
38749     *p = area;
38750     return area;
38751 }
38752
p 564 38753 void vfree(void * addr)
38754 {
38755     struct vm_struct **p, *tmp;
38756
38757     if (!addr)
38758         return;
38759     if ((PAGE_SIZE-1) & (unsigned long) addr) {
38760         printk("Trying to vfree() bad address (%p)\n", addr);
38761         return;
38762     }
38763     for (p = &vmlist; (tmp = *p); p = &tmp->next) {
38764         if (tmp->addr == addr) {
38765             *p = tmp->next;
38766             vmfree_area_pages(VMALLOC_VMADDR(tmp->addr),
38767                               tmp->size);
38768             kfree(tmp);
38769             return;
38770         }
38771     }
38772     printk("Trying to vfree() nonexistent vm area (%p)\n",
38773           addr);
38774 }
38775
p 563 38776 void * vmalloc(unsigned long size)
38777 {
38778     void * addr;
38779     struct vm_struct *area;
38780
38781     size = PAGE_ALIGN(size);
38782     if (!size || size > (max_mapnr << PAGE_SHIFT))
38783         return NULL;
38784     area = get_vm_area(size);
38785     if (!area)
38786         return NULL;
38787     addr = area->addr;
38788     if (vmalloc_area_pages(VMALLOC_VMADDR(addr), size)) {
38789         vfree(addr);
38790         return NULL;
38791     }
38792     return addr;
38793 }
38794
38795 long vread(char *buf, char *addr, unsigned long count)
38796 {
38797     struct vm_struct *tmp;
38798     char *vaddr, *buf_start = buf;
38799     unsigned long n;
38800
38801     /* Don't allow overflow */
38802     if ((unsigned long) addr + count < count)
38803         count = -(unsigned long) addr;
38804

```

```

38805 for (tmp = vmlist; tmp; tmp = tmp->next) {
38806     vaddr = (char *) tmp->addr;
38807     if (addr >= vaddr + tmp->size - PAGE_SIZE)
38808         continue;
38809     while (addr < vaddr) {
38810         if (count == 0)
38811             goto finished;
38812         put_user('\0', buf);
38813         buf++;
38814         addr++;
38815         count--;
38816     }
38817     n = vaddr + tmp->size - PAGE_SIZE - addr;
38818     do {
38819         if (count == 0)
38820             goto finished;
38821         put_user(*addr, buf);
38822         buf++;
38823         addr++;
38824         count--;
38825     } while (--n > 0);
38826 }
38827 finished:
38828 return buf - buf_start;
38829 }

```

mm/vmscan.c

```

38830 /*
38831  * linux/mm/vmscan.c
38832  *
38833  * Copyright (C) 1991, 1992, 1993, 1994 Linus Torvalds
38834  *
38835  * Swap reorganised 29.12.95, Stephen Tweedie.
38836  * kswapd added: 7.1.96 sct
38837  * Removed kswapd_ctl limits, and swap out as many pages
38838  * as needed to bring the system back to freepages.high:
38839  * 2.4.97, Rik van Riel.
38840  * Version: $Id: vmscan.c,v 1.5 1998/02/23 22:14:28 sct
38841  * Exp $ */
38842
38843 #include <linux/slab.h>
38844 #include <linux/kernel_stat.h>
38845 #include <linux/swap.h>
38846 #include <linux/swapctl.h>
38847 #include <linux/smp_lock.h>
38848 #include <linux/pagemap.h>
38849 #include <linux/init.h>

```

```

38850
38851 #include <asm/pgtable.h>
38852
38853 /* The swap-out functions return 1 if they successfully
38854  * threw something out, and we got a free page. It
38855  * returns zero if it couldn't do anything, and any other
38856  * value indicates it decreased rss, but the page was
38857  * shared.
38858  *
38859  * NOTE! If it sleeps, it *must* return 1 to make sure we
38860  * don't continue with the swap-out. Otherwise we may be
38861  * using a process that no longer actually exists (it
38862  * might have died while we slept). */
38863 static int try_to_swap_out(struct task_struct * ts,
38864     struct vm_area_struct* vma, unsigned long address,
38865     pte_t * page_table, int gfp_mask)
38866 {
38867     pte_t pte;
38868     unsigned long entry;
38869     unsigned long page;
38870     struct page * page_map;
38871
38872     pte = *page_table;
38873     if (!pte_present(pte))
38874         return 0;
38875     page = pte_page(pte);
38876     if (MAP_NR(page) >= max_mapnr)
38877         return 0;
38878
38879     page_map = mem_map + MAP_NR(page);
38880     if (PageReserved(page_map)
38881         || PageLocked(page_map)
38882         || ((gfp_mask & __GFP_DMA) && !PageDMA(page_map)))
38883         return 0;
38884
38885     if (pte_young(pte)) {
38886         /* Transfer the "accessed" bit from the page tables
38887          * to the global page map. */
38888         set_pte(page_table, pte_mkold(pte));
38889         set_bit(PG_referenced, &page_map->flags);
38890         return 0;
38891     }
38892
38893     /* Is the page already in the swap cache? If so, then
38894     * we can just drop our reference to it without doing
38895     * any IO - it's already up-to-date on disk.
38896     *
38897     * Return 0, as we didn't actually free any real

```

→ p 551

```

38898 * memory, and we should just continue our scan. */
38899 if (PageSwapCache(page_map)) {
38900     entry = page_map->offset;
38901     swap_duplicate(entry);
38902     set_pte(page_table, __pte(entry));
38903 drop_pte:
38904     vma->vm_mm->rss--;
38905     flush_tlb_page(vma, address);
38906     __free_page(page_map);
38907     return 0;
38908 }
38909
38910 /* Is it a clean page? Then it must be recoverable by
38911 * just paging it in again, and we can just drop it..
38912 *
38913 * However, this won't actually free any real memory,
38914 * as the page will just be in the page cache
38915 * somewhere, and as such we should just continue our
38916 * scan.
38917 *
38918 * Basically, this just makes it possible for us to do
38919 * some real work in the future in "shrink_mmap()". */
38920 if (!pte_dirty(pte)) {
38921     pte_clear(page_table);
38922     goto drop_pte;
38923 }
38924
38925 /* Don't go down into the swap-out stuff if we cannot
38926 * do I/O! Avoid recursing on FS locks etc. */
38927 if (!(gfp_mask & __GFP_IO))
38928     return 0;
38929
38930 /* Ok, it's really dirty. That means that we should
38931 * either create a new swap cache entry for it, or we
38932 * should write it back to its own backing store.
38933 *
38934 * Note that in neither case do we actually know that
38935 * we make a page available, but as we potentially
38936 * sleep we can no longer continue scanning, so we
38937 * might as well assume we free'd something.
38938 *
38939 * NOTE NOTE NOTE! This should just set a dirty bit in
38940 * page_map, and just drop the pte. All the hard work
38941 * would be done by shrink_mmap().
38942 *
38943 * That would get rid of a lot of problems. */
38944 flush_cache_page(vma, address);
38945 if (vma->vm_ops && vma->vm_ops->swapout) {

```

```

38946     pid_t pid = tsk->pid;
38947     pte_clear(page_table);
38948     flush_tlb_page(vma, address);
38949     vma->vm_mm->rss--;
38950
38951     if (vma->vm_ops->swapout(vma, page_map))
38952         kill_proc(pid, SIGBUS, 1);
38953     __free_page(page_map);
38954     return 1;
38955 }
38956
38957 /* This is a dirty, swappable page. First of all, get
38958 * a suitable swap entry for it, and make sure we have
38959 * the swap cache set up to associate the page with
38960 * that swap entry. */
38961 entry = get_swap_page();
38962 if (!entry)
38963     return 0; /* No swap space left */
38964
38965 vma->vm_mm->rss--;
38966 tsk->nswap++;
38967 set_pte(page_table, __pte(entry));
38968 flush_tlb_page(vma, address);
38969 /* One for the process, one for the swap cache */
38970 swap_duplicate(entry);
38971 add_to_swap_cache(page_map, entry);
38972 /* We checked we were unlocked way up above, and we
38973 * have been careful not to stall until here */
38974 set_bit(PG_locked, &page_map->flags);
38975
38976 /* OK, do a physical asynchronous write to swap. */
38977 rw_swap_page(WRITE, entry, (char *) page, 0);
38978
38979 __free_page(page_map);
38980 return 1;
38981 }
38982
38983 /* A new implementation of swap_out(). We do not swap
38984 * complete processes, but only a small number of blocks.
38985 * before we continue with the next process. The number
38986 * of blocks actually swapped is determined on the number
38987 * of page faults, that this process actually had in the
38988 * last time, so we won't swap heavily used processes all
38989 * the time ...
38990 *
38991 * Note: the priority argument is a hint on much CPU to
38992 * waste with the swap block search, not a hint, of how
38993 * much blocks to swap with each process.

```

```

38994 *
38995 * (C) 1993 Kai Petzke, wpp@marie.physik.tu-berlin.de */
38996 static inline int swap_out_pmd(struct task_struct * tsk,
38997 struct vm_area_struct * vma, pmd_t * dir,
38998 unsigned long address, unsigned long end, int gfp_mask)
38999 {
39000 pte_t * pte;
39001 unsigned long pmd_end;
39002
39003 if (pmd_none(*dir))
39004     return 0;
39005 if (pmd_bad(*dir)) {
39006     printk("swap_out_pmd: bad pmd (%08lx)\n",
39007           pmd_val(*dir));
39008     pmd_clear(dir);
39009     return 0;
39010 }
39011
39012 pte = pte_offset(dir, address);
39013
39014 pmd_end = (address + PMD_SIZE) & PMD_MASK;
39015 if (end > pmd_end)
39016     end = pmd_end;
39017
39018 do {
39019     int result;
39020     tsk->mm->swap_address = address + PAGE_SIZE;
39021     result = try_to_swap_out(tsk, vma, address, pte,
39022                             gfp_mask);
39023     if (result)
39024         return result;
39025     address += PAGE_SIZE;
39026     pte++;
39027 } while (address < end);
39028 return 0;
39029 }
39030
39031 static inline int swap_out_pgd(struct task_struct * tsk,
39032 struct vm_area_struct * vma, pgd_t * dir,
39033 unsigned long address, unsigned long end, int gfp_mask)
39034 {
39035 pmd_t * pmd;
39036 unsigned long pgd_end;
39037
39038 if (pgd_none(*dir))
39039     return 0;
39040 if (pgd_bad(*dir)) {
39041     printk("swap_out_pgd: bad pgd (%08lx)\n",

```

```

39042           pgd_val(*dir));
39043     pgd_clear(dir);
39044     return 0;
39045 }
39046
39047 pmd = pmd_offset(dir, address);
39048
39049 pgd_end = (address + PGDIR_SIZE) & PGDIR_MASK;
39050 if (end > pgd_end)
39051     end = pgd_end;
39052
39053 do {
39054     int result = swap_out_pmd(tsk, vma, pmd, address,
39055                             end, gfp_mask);
39056     if (result)
39057         return result;
39058     address = (address + PMD_SIZE) & PMD_MASK;
39059     pmd++;
39060 } while (address < end);
39061 return 0;
39062 }
39063
39064 static int swap_out_vma(struct task_struct * tsk,
39065 struct vm_area_struct * vma, unsigned long address,
39066 int gfp_mask)
39067 {
39068 pgd_t *pgdir;
39069 unsigned long end;
39070
39071 /* Don't swap out areas like shared memory which have
39072 * their own separate swapping mechanism or areas which
39073 * are locked down */
39074 if (vma->vm_flags & (VM_SHM | VM_LOCKED))
39075     return 0;
39076
39077 pgdir = pgd_offset(tsk->mm, address);
39078
39079 end = vma->vm_end;
39080 while (address < end) {
39081     int result = swap_out_pgd(tsk, vma, pgdir, address,
39082                             end, gfp_mask);
39083     if (result)
39084         return result;
39085     address = (address + PGDIR_SIZE) & PGDIR_MASK;
39086     pgdir++;
39087 }
39088 return 0;
39089 }

```

```

39090
39091 static int swap_out_process(struct task_struct * p,
39092                             int gfp_mask)
39093 {
39094     unsigned long address;
39095     struct vm_area_struct* vma;
39096
39097     /* Go through process' page directory. */
39098     address = p->mm->swap_address;
39099
39100     /* Find the proper vm-area */
39101     vma = find_vma(p->mm, address);
39102     if (vma) {
39103         if (address < vma->vm_start)
39104             address = vma->vm_start;
39105
39106         for (;;) {
39107             int result =
39108                 swap_out_vma(p, vma, address, gfp_mask);
39109             if (result)
39110                 return result;
39111             vma = vma->vm_next;
39112             if (!vma)
39113                 break;
39114             address = vma->vm_start;
39115         }
39116     }
39117
39118     /* We didn't find anything for the process */
39119     p->mm->swap_cnt = 0;
39120     p->mm->swap_address = 0;
39121     return 0;
39122 }
39123
39124 /* Select the task with maximal swap_cnt and try to swap
39125  * out a page. N.B. This function returns only 0 or 1.
39126  * Return values != 1 from the lower-level routines
39127  * result in continued processing. */
39128 static int swap_out(unsigned int priority, int gfp_mask)
39129 {
39130     struct task_struct * p, * pbest;
39131     int counter, assign, max_cnt;
39132
39133     /* We make one or two passes through the task list,
39134     * indexed by assign = {0, 1}:
39135     * Pass 1: select the swappable task with maximal
39136     *         RSS that HAS not yet been swapped out.
39137     * Pass 2: re-assign rss swap_cnt values, then

```

```

39138     *         select as above.
39139     *
39140     * With this approach, there's no need to remember the
39141     * last task swapped out. If the swap-out fails, we
39142     * clear swap cnt so the task won't be selected again
39143     * until all others have been tried.
39144     *
39145     * Think of swap_cnt as a "shadow rss" - it tells us
39146     * which process we want to page out (always try
39147     * largest first). */
39148     counter = nr_tasks / (priority+1);
39149     if (counter < 1)
39150         counter = 1;
39151     if (counter > nr_tasks)
39152         counter = nr_tasks;
39153
39154     for (; counter >= 0; counter--) {
39155         assign = 0;
39156         max_cnt = 0;
39157         pbest = NULL;
39158         select:
39159         read_lock(&tasklist_lock);
39160         p = init_task.next_task;
39161         for (; p != &init_task; p = p->next_task) {
39162             if (!p->swappable)
39163                 continue;
39164             if (p->mm->rss <= 0)
39165                 continue;
39166             /* Refresh swap_cnt? */
39167             if (assign)
39168                 p->mm->swap_cnt = p->mm->rss;
39169             if (p->mm->swap_cnt > max_cnt) {
39170                 max_cnt = p->mm->swap_cnt;
39171                 pbest = p;
39172             }
39173         }
39174         read_unlock(&tasklist_lock);
39175         if (!pbest) {
39176             if (!assign) {
39177                 assign = 1;
39178                 goto select;
39179             }
39180             goto out;
39181         }
39182
39183         if (swap_out_process(pbest, gfp_mask))
39184             return 1;
39185     }

```



```

39186 out:
39187     return 0;
39188 }
39189
39190 /* We need to make the locks finer granularity, but right
39191 * now we need this so that we can do page allocations
39192 * without holding the kernel lock etc.
39193 *
39194 * We want to try to free "count" pages, and we need to
39195 * cluster them so that we get good swap-out
39196 * behaviour. See the "free_memory()" macro for details.
39197 */
39198 static int do_try_to_free_pages(unsigned int gfp_mask)
39199 {
39200     int priority;
39201     int count = SWAP_CLUSTER_MAX;
39202
39203     lock_kernel();
39204
39205     /* Always trim SLAB caches when memory gets low. */
39206     kmem_cache_reap(gfp_mask);
39207
39208     priority = 6;
39209     do {
39210         while (shrink_mmap(priority, gfp_mask)) {
39211             if (!--count)
39212                 goto done;
39213         }
39214
39215         /* Try to get rid of some shared memory pages.. */
39216         if (gfp_mask & __GFP_IO) {
39217             while (shm_swap(priority, gfp_mask)) {
39218                 if (!--count)
39219                     goto done;
39220             }
39221         }
39222
39223         /* Then, try to page stuff out.. */
39224         while (swap_out(priority, gfp_mask)) {
39225             if (!--count)
39226                 goto done;
39227         }
39228
39229         shrink_dcache_memory(priority, gfp_mask);
39230     } while (--priority >= 0);
39231 done:
39232     unlock_kernel();
39233

```

```

39234     return priority >= 0;
39235 }
39236
39237 /* Before we start the kernel thread, print out the
39238 * kswapd initialization message (otherwise the init
39239 * message may be printed in the middle of another
39240 * driver's init message). It looks very bad when that
39241 * happens. */
39242 void __init kswapd_setup(void)
39243 {
39244     int i;
39245     char *revision="$Revision: 1.5 $", *s, *e;
39246
39247     swap_setup();
39248
39249     if ((s = strchr(revision, ':')) &&
39250         (e = strchr(s, '$')))
39251         s++, i = e - s;
39252     else
39253         s = revision, i = -1;
39254     printk ("Starting kswapd v%.*s\n", i, s);
39255 }
39256
39257 static struct task_struct *kswapd_process;
39258
39259 /* The background pageout daemon, started as a kernel
39260 * thread from the init process.
39261 *
39262 * This basically executes once a second, trickling out
39263 * pages so that we have _some_ free memory available
39264 * even if there is no other activity that frees anything
39265 * up. This is needed for things like routing etc, where
39266 * we otherwise might have all activity going on in
39267 * asynchronous contexts that cannot page things out.
39268 *
39269 * If there are applications that are active
39270 * memory-allocators (most normal use), this basically
39271 * shouldn't matter. */
39272 int kswapd(void *unused)
39273 {
39274     struct task_struct *tsk = current;
39275
39276     kswapd_process = tsk;
39277     tsk->session = 1;
39278     tsk->pgrp = 1;
39279     strcpy(tsk->comm, "kswapd");
39280     sigfillset(&tsk->blocked);
39281

```

```

39282 /* Tell the memory management that we're a "memory
39283 * allocator", and that if we need more memory we
39284 * should get access to it regardless (see
39285 * "__get_free_pages()"). "kswapd" should never get
39286 * caught in the normal page freeing logic.
39287 *
39288 * (Kswapd normally doesn't need memory anyway, but
39289 * sometimes you need a small amount of memory in order
39290 * to be able to page out something else, and this flag
39291 * essentially protects us from recursively trying to
39292 * free more memory as we're trying to free the first
39293 * piece of memory in the first place). */
39294 tsk->flags |= PF_MEMALLOC;
39295
39296 while (1) {
39297     /* Wake up once a second to see if we need to make
39298      * more memory available.
39299      *
39300      * If we actually get into a low-memory situation,
39301      * the processes needing more memory will wake us up
39302      * on a more timely basis. */
39303     do {
39304         if (nr_free_pages >= freepages.high)
39305             break;
39306
39307         if (!do_try_to_free_pages(GFP_KSWAPD))
39308             break;
39309     } while (!tsk->need_resched);
39310     run_task_queue(&tq_disk);
39311     tsk->state = TASK_INTERRUPTIBLE;
39312     schedule_timeout(HZ);
39313 }
39314 }
39315
39316 /* Called by non-kswapd processes when they want more
39317 * memory.
39318 *
39319 * In a perfect world, this should just wake up kswapd
39320 * and return. We don't actually want to swap stuff out
39321 * from user processes, because the locking issues are
39322 * nasty to the extreme (file write locks, and MM
39323 * locking)
39324 *
39325 * One option might be to let kswapd do all the page-out
39326 * and VM page table scanning that needs locking, and
39327 * this process thread could do just the mmap shrink
39328 * stage that can be done by just dropping cached pages
39329 * without having any deadlock issues. */
39330 int try_to_free_pages(unsigned int gfp_mask)
39331 {
39332     int retval = 1;
39333
39334     wake_up_process(kswapd_process);
39335     if (gfp_mask & __GFP_WAIT)
39336         retval = do_try_to_free_pages(gfp_mask);
39337     return retval;
39338 }

```

第二部分 Linux 内核源代码分析

第1章 Linux 简介

让用户很详细地了解大多数现有操作系统的实际工作方式是不可能的，因为大多数操作系统的源代码都是严格保密的。除了一些研究用的及为操作系统教学而设计的系统外。尽管研究和教学目的都很好，但是这类系统很少能够通过正式操作系统的小部分实现来体现操作系统的实际功能。对于操作系统的一些特殊问题，这种折衷系统所能够表现的就更是少得可怜了。

在以实际使用为目标的操作系统中，让任何人都可以自由获取系统源代码，无论目的是要了解、学习还是改进，这样的现实系统并不多。本书的主题就是这些少数操作系统中的一个：Linux。

Linux的工作方式类似于Unix，它是免费的，源代码也是开放的，符合标准规范的32位（在64位CPU上是64位）操作系统。Linux拥有现代操作系统的所具有的内容，例如：

- 真正的抢先式多任务处理，支持多用户。
- 内存保护。
- 虚拟内存。
- 支持对称多处理机SMP (symmetric multiprocessing)，即多个CPU机器以及通常的单个CPU (UP) 机器。
- 符合POSIX标准。
- 联网。
- 图形用户接口和桌面环境（实际上桌面环境并不只一个）。
- 速度和稳定性。

严格说来，Linux并不是一个完整的操作系统。当我们在安装通常所说的Linux时，我们实际安装的是很多工具的集合。这些工具协同工作以组成一个功能强大的实用系统。Linux本身只是这个操作系统的内核，是操作系统的核心、灵魂、指挥中心（整个系统应该称为GNU/Linux，其原因在本章的后续内容中将会给以介绍）。内核以独占的方式执行最底层任务，保证系统正常运行——协调多个并发进程，管理进程使用的内存，使它们相互之间不产生冲突，满足进程访问磁盘的请求等等。

在本书中，我们给大家揭示的就是Linux是如何完成这一具有挑战性的工作的。

1.1 Linux和Unix的简明历史

为了让大家对本书所讨论的内容有更清楚的了解，让我们先来简要回顾一下Linux的历史。由于Linux是在Unix的基础上发展而来的，我们的话题就从Unix开始。

Unix是由AT&T贝尔实验室的Ken Thompson和Dennis Ritchie于1969年在一台已经废弃了的PDP-7上开发的；它最初是一个用汇编语言写成的单用户操作系统。不久，Thompson和Ritchie成功地说服管理部门为他们购买更新的机器，以便该开发小组可以实现一个文本处理系统，Unix就在PDP-11上用C语言重新编写（发明C语言的部分目的就在于此）。它果真变成

了一个文本处理系统——不久之后。只不过问题是他们先实现了一个操作系统而已……

最终，他们实现了该文本处理工具，而且Unix（以及Unix上运行的工具）也在AT&T得到广泛应用。在1973年，Thompson和Ritchie在一个操作系统会议上就这个系统发表了一篇文章，该论文引起了学术界对Unix系统的极大兴趣。

由于1956年反托拉斯法案的限制，AT&T不能涉足计算机业务，但允许它象征性地收取费用发售该系统。就这样，Unix被广泛发布，首先是学术科研用户，后来又扩展到政府和商业用户。

伯克利加州大学是学术用户中的一个。在这里，Unix得到了计算机系统研究小组（CSRG）的广泛应用。并且在这里所进行的修改引发了Unix的一大系列，这就是广为人知的伯克利软件开发（BSD）Unix。除了AT&T所提供的Unix系列之外，BSD是最有影响力的Unix系列。BSD在Unix中增加了很多显著特性，例如TCP/IP网络，更好的用户文件系统（UFS），工作控制，并且改进了AT&T的内存管理代码。

多年以来，BSD版本的Unix一直在学术环境中占据主导地位，但最终发展成为System V版本的AT&T的Unix则成为商业领域的领头羊。从某种程度上来说，这是有社会原因的：学校倾向于使用非正式但通常更好用的BSD风格的Unix，而商业界则倾向于从AT&T获取Unix。

在用户需求和用户编程改进特性的促进下，BSD风格的Unix一般要比AT&T的Unix更具有创新性，而且改进也更为迅速。但是，在AT&T发布最后一个正式版本System V Release 4（SVR4）时，System V Unix已经吸收了BSD的大多数重要的优点，并且还增加了一些自己的优势。这部分由于从1984年开始，AT&T逐渐可以将Unix商业化，而伯克利Unix的开发工作在1993年BSD4.4版本完成以后就逐渐收缩，以至终止了。然而，BSD的进一步改进由外界开发者延续下来，到今天还在继续进行。正在进行的Unix系列开发中至少有四个独立的版本是直接起源于BSD4.4，这还不包括几个厂商的Unix版本，例如惠普的HP-UX，都是部分地或者全部基于BSD而发展起来的。

实际上Unix的变种并不止BSD和System V。由于Unix主要使用C语言来编写，这就使得它移植到新的机器上相对比较容易，它的简单性也使其重新设计与开发相对比较容易。Unix的这些特点大受商业界硬件供应商的欢迎，比如Sun、SGI、HP、IBM、DEC、Amdahl等等；IBM还不止一次对Unix进行了再开发。厂商们设计开发出新的硬件，并简单地将Unix移植到新的硬件上，这样新的硬件一经发布便具备一定的功能。经过一段时间之后，这些厂商都拥有了自己的专有Unix版本。而且为了占有市场，这些版本故意以不同的侧重点发布出来，以更好地占有用户。

版本混乱的状态促进了标准化工作的进行。其中最主要的就是POSIX系列标准，它定义了一套标准的操作系统接口和工具。从理论上说，POSIX标准代码很容易移植到任何遵守POSIX标准的操作系统中，而且严格的POSIX测试已经把这种理论上的可移植性转化为现实。直到今天，几乎所有的正式操作系统都以支持POSIX标准为目标。

现在让我们回顾一下，在1984年，杰出的电脑黑客Richard Stallman独立开发出一个类Unix的操作系统，该操作系统具有完全的内核、开发工具和终端用户应用程序。在GNU（“GNU's Not Unix”首字母的缩写）计划的配合下，Stallman开发这个产品有自己的技术理想：他想开发出一个质量高而且自由的操作系统。Stallman使用了“自由”（free）这个词，不仅意味着用户可以免费获取软件；而且更重要的是，它将意味着某种程度的“解放”：用户可以

自由使用、拷贝、查询、重用、修改甚至是分发这份软件，完全没有软件使用协议的限制。这也正是Stallman创建自由软件基金会（FSF）资助GNU软件开发的本意（FSF也在资助其他科研方面的开发工作）。

15年来，GNU工程已经吸收、产生了大量的程序，这不仅包括Emacs、gcc（GNU的C编译器）、bash（shell命令），还有大部分Linux用户所熟知的许多应用程序。现在正在进行开发的项目是GNU Hurd内核，这是GNU操作系统的最后一个主要部件（实际上Hurd内核早已能够使用了，不过当前的版本号为0.3的系统在什么时候能够完成，还是未知数）。

尽管Linux大受欢迎，但是Hurd内核还在继续开发。原因有几个方面，其一是Hurd的体系结构十分清晰地体现了Stallman关于操作系统工作方式的思想，例如，在运行期间，任何用户都可以部分地改变或替换Hurd（这种替换不是对每个用户都是可见的，而是只对申请修改的用户可见，而且还必须符合安全规范）。另一个原因是据介绍Hurd对于多处理器的支持比Linux本身的内核要好。还有一个简单的原因是兴趣的驱动，因为程序员们希望能够自由地进行自己所喜欢的工作。只要有人希望为Hurd工作，Hurd的开发就不会停止。如果他们能够如愿以偿，Hurd有朝一日将成为Linux的强劲对手。不过在今天，Linux还是自由内核王国里无可争议的统治者。

在GNU发展的中期，也就是1991年，一个名叫Linus Torvalds的芬兰大学生想要了解Intel的新CPU——80386。他认为比较好的学习方法是自己编写一个操作系统的内核。出于这种目的，加上他对当时Unix变种版本对于80386类机器的脆弱支持十分不满，他决定要开发出一个全功能的、支持POSIX标准的、类Unix的操作系统内核，该系统吸收了BSD和System V的优点，同时摒弃了它们的缺点。Linus（虽然我知道我应该称他为Torvalds，但是所有人都称他为Linus）独立把这个内核开发到0.02版，这个版本已经可以运行gcc、bash和很少的一些应用程序。这些就是他开始的全部工作了。后来，他又开始在因特网上寻求广泛的帮助。

不到三年，Linus的Unix——Linux，已经升级到1.0版本。它的源代码量也呈指数形式增长，实现了基本的TCP/IP功能（网络部分的代码后来重写过，而且还可能会再次重写）。此时Linux就已经拥有大约10万用户了。

现在的Linux内核由150多万行代码组成，Linux也已经拥有了大约1000万用户（由于Linux可以自由获取和拷贝，获取具体的统计数字是不可能的）。Linux内核GNU/Linux附同GNU工具已经占据了Unix 50%的市场。一些公司正在把内核和一些应用程序同安装软件打包在一起，生产出Linux的发行版本，这些公司包括Red Hat和Caldera公司。现在的GNU/Linux已经备受瞩目，得到了诸如Sun、IBM、SGI等公司的广泛支持。SGI最近决定在其基于Intel的Merced的系列机器上不再搭载自己的Unix变种版本IRIX，而是直接采用GNU/Linux；Linux甚至被指定为Amiga将要发布的新操作系统的基础。

1.2 GNU通用公共许可证

这样一个如此流行的操作系统当然值得我们学习。按照通用公共许可证(GPL, General Public License)的规定，Linux的源代码可以自由获取，这满足了我们学习该系统的强烈愿望。GPL这份非同寻常的软件许可证，充分体现了上面提到的Stallman的思想：只要用户所做的修改是同等自由的，用户可以自由地使用、拷贝、查询、重用、修改甚至重新发布这个软件。通过这种方式，GPL保证了Linux（以及同一许可证保证下的大量其他软件）不仅现在自由可

用，而且以后经过任何修改之后都仍然可以自由使用。

请注意这里的自由并不是说没有人靠这个软件盈利，有一些日益兴起的公司，比如发行最流行的Linux发行版本的Red Hat就是一个例子（Red Hat自从上市以来，市值已经突破数十亿美元，每年盈利数十万美元，而且这些数字还在不断增长）。但是任何人都不能限制其他用户涉足本软件领域，而且所做的修改不能减少其自由程度。

本书的附录B中收录了GNU通用公共许可证协议的全文。

1.3 Linux开发过程

如上所述，由于Linux是一个自由软件，它可以免费获取以供学习研究。Linux之所以值得学习研究，是因为它是相当优秀的操作系统。如果Linux操作系统相当糟糕，那它就根本不值得我们使用，也就没有必要去研究相关的书籍。Linux是一个十分优秀的操作系统还在于几个相互关联的原因。

原因之一在于它是基于天才的思想开发而成的。在学生时代就开始推动整个系统开发的Linus Torvalds是一个天才，他的才能不仅展现在编程能力方面，而且组织技巧也相当杰出。Linux的内核是由世界上一些最优秀的程序员开发并不断完善的，他们通过Internet相互协作，开发理想的操作系统；他们享受着工作中的乐趣，而且也获得了充分的自豪感。

Linux优秀的另外一个原因在于它是基于一组优秀的概念。Unix是一个简单却非常优秀的模型。在Linux创建之前，Unix已经有20年的发展历史。Linux从Unix的各个流派中不断吸取成功经验，模仿Unix的优点，抛弃Unix的缺点。这样做的结果是Linux成为了Unix系列中的佼佼者：高速、健壮、完整，而且抛弃了历史包袱。

然而，Linux最强大的生命力还在于其公开的开发过程。每个人都可以自由获取内核源程序，每个人都可以对源程序加以修改，而后他人也可以自由获取你修改后的源程序。如果你发现了缺陷，你可以对它进行修正，而不用去乞求不知名的公司来为你修正。如果你有什么最优化或者新特点的创意，你也可以直接在系统中增加功能，而不用向操作系统供应商解释你的想法，指望他们将来会增加相应的功能。当发现一个安全漏洞后，你可以通过编程来弥补这个漏洞，而不用关闭系统直到你的供应商为你提供修补程序。由于你拥有直接访问源代码的能力，你也可以直接阅读代码来寻找缺陷，或是效率不高的代码，或是安全漏洞，以防患于未然。

除非你是一个程序员，否则这一点听起来仿佛没有多少吸引力。实际上，即使你不是程序员，这种开发模型也将使你受益匪浅，这主要体现在以下两个方面：

- 可以间接受益于世界各地成千上万的程序员随时进行的改进工作。
- 如果你需要对系统进行修改，你可以雇用程序员为你完成工作。这部分人将根据你的需求定义单独为你服务。可以设想，这在源程序不公开的操作系统中将是什么样子。

Linux这种独特的自由流畅的开发模型已被命名为bazaar（集市模型），它是相对于cathedral（教堂）模型而言的。在cathedral模型中，源程序代码被锁定在一个保密的小范围内。只有开发者（很多情况下是市场）认为能够发行一个新版本，这个新版本才会被推向市场。这些术语在Eric S. Raymond的《教堂与集市》（The Cathedral and the Bazaar）一文中有所介绍，大家可以在<http://www.tuxedo.org/~esr/writings/>找到这篇文章。bazaar开发模型通过重视实验，征集并充分利用早期的反馈，对巨大数量的脑力资源进行平衡配置，可以开发出更优

秀的软件。(顺便说一下,虽然Linux是最为明显的使用bazaar开发模型的例子,但是它却远不是第一个使用这个模型的系统。)

为了确保这些无序的开发过程能够有序地进行, Linux采用了双树系统。一个树是稳定树(stable tree), 另一个树是非稳定树(unstable tree) 或者开发树(development tree)。一些新特性、实验性改进等都将首先在开发树中进行。如果在开发树中所做的改进也可以应用于稳定树, 那么在开发树中经过测试以后, 在稳定树中将进行相同的改进。按照Linus的观点, 一旦开发树经过了足够的发展, 开发树就会成为新的稳定树, 如此周而复始的进行下去。

源程序版本号的形式为x.y.z。对于稳定树来说, y是偶数; 对于开发树来说, y比相应的稳定树大一(因此, 是奇数)。截至到本书截稿时, 最新的稳定内核版本号是2.2.10, 最新的开发内核的版本号是2.3.12。对2.3树的缺陷修正会回溯影响(back-propagated) 2.2树, 而当2.3树足够成熟的时候会发展成为2.4.0。(顺便说一下, 这种开发会比常规惯例要快, 因为每一版本所包含的改变比以前更少了, 内核开发人员只需花很短的时间就能够完成一个实验开发周期。)

<http://www.kernel.org>及其镜像站点提供了最新的可供下载的内核版本, 而且同时包括稳定和开发版本。如果你愿意的话, 不需要很长时间, 这些站点所提供的最新版本中就可能包含了你的一部分源程序代码。

第2章 代码初识

本章首先从较高层次介绍Linux内核源程序的概况，这些都是大家关心的一些基本特点。随后将简要介绍一些实际代码。最后介绍如何编译内核。

2.1 Linux内核源程序的部分特点

在过去的一段时期，Linux内核同时使用C语言和汇编语言来实现。这两种语言需要一定的平衡：C语言编写的代码移植性较好、易于维护，而汇编语言编写的程序则速度较快。一般只有在速度是关键因素或者一些因平台相关特性而产生的特殊要求（例如直接和内存管理硬件进行通讯）时才使用汇编语言。

正如实际中所做的，即使内核并未使用C++的对象特性，部分内核也可以在g++（GNU的C++编译器）下进行编译。同其他面向对象的编程语言相比较，相对而言C++的开销是较低的，但是对于内核开发人员来说，这已经是太多了。

内核开发人员不断发展编程风格，形成了Linux代码独有的特色。本节将讨论其中的一些问题。

2.1.1 gcc特性的使用

Linux内核被设计为必须使用GNU的C编译器gcc来编译，而不是任何一种C编译器都可以使用。内核代码有时要使用gcc特性，本书将陆续介绍其中的一部分。

一些gcc特有代码只是简单地使用gcc语言扩展，例如允许在C（不只是C++）中使用inline关键字指示内联函数。也就是说，代码中被调用的函数在每次函数调用时都会被扩充，因而就可以节约实际函数调用的开销。

一般情况下，代码的编写方式比较复杂。因为对于某些类型的输入，gcc能够产生比其他输入效率更高的执行代码。从理论上讲，编译器可以优化具有相同功能的两种对等的方法，并且得到相同的结果。因此，代码的编写方式是无关紧要的。但在实际上，用某种方法编写所产生的代码要比用另外一些方法编写所产生的代码执行速度快许多。内核开发人员知道怎样才能产生更高效的执行代码，这不断地在他们编写的代码中反映出来。

例如，考虑内核中经常使用的goto语句——为了提高速度，内核中经常大量使用这种一般要避免使用的语句。在本书中所包含的不到40 000行代码中，一共有500多条goto语句，大约是每80行一个。除汇编文件外，精确的统计数字是接近每72行一个goto语句。公平地说，这是选择偏向的结果：比例如此高的原因之一是本书中涉及的是内核源程序的核心，在这里速度比其他因素都需要优先考虑。整个内核的比例大概是每260行一个goto语句。然而，这仍然是我不再使用Basic进行编程以来见过的使用goto频率最高的地方。

代码必需受特定编译器限制的特性不仅与普通应用程序的开发有很大不同，而且也不同干大多数内核的开发。大多数的开发人员使用C语言编写代码来保持较高的可移植性，即使在编写操作系统时也是如此。这样做的优点是显而易见的，最为重要的一点是一旦出现更好的

编译器，程序员们可以随时进行更换。

内核对于gcc特性的完全依赖使得内核向新的编译器上移植更加困难。最近Linus对这一问题在有关内核的邮件列表上表明了自己的观点：“记住，编译器只是一个工具。”这是对依赖于gcc特性的一个很好的基本思想的表述：编译器只是为了完成工作。如果通过遵守标准还不能达到工作要求，那就不是工作要求有问题，而是对于标准的依赖有问题。

在大多数情况下，这种观点是不能被人所接受的。通常情况下，为了保证和程序语言标准的一致，开发人员可能需要牺牲某些特性、速度或者其他相关因素。其他的选择可能会为后期开发造成很大的麻烦。

但是，在这种特定的情况下，Linus是正确的。Linux内核是一个特例，因为其执行速度要比向其他编译器的可移植性远为重要。如果设计目标是编写一个可移植性好而不要求快速运行的内核，或者是编写一个任何人都可以使用自己喜欢的编译器进行编译的内核，那么结论就可能会有所不同了；而这些恰好不是Linux的设计目标。实际上，gcc几乎可以为所有能够运行Linux的CPU生成代码，因此，对于gcc的依赖并不是可移植性的严重障碍。

在第3章中我们将对内核设计目标进行详细说明。

2.1.2 内核代码习惯用语

内核代码中使用了一些显著的习惯用语，本节将介绍常用的几个。当通读源代码时，真正重要的问题并不在这些习惯用语本身，而是这种类型的习惯用语的确存在，而且是不断被使用和发展的。如果你需要编写内核代码，你应该注意到内核中所使用的习惯用语，并把这些习惯用语应用到你的代码中。当通读本书（或者代码）时，看看你还能找到多少习惯用语。

为了讨论这些习惯用语，我们首先需要对它们进行命名。为了便于讨论，笔者创造了这些名字。而在实际中，大家不一定非要参考这些用语，它们只是对内核工作方式的描述而已。

一个普通的习惯用语，笔者称之为“资源获取”（resource acquisition idiom）。在这个用语中，一个函数必须实现一系列资源的获取，包括内存、锁等等（这些资源的类型未必相同）。只有成功地获取当前所需要的资源之后，才能处理后面的资源请求。最后，该函数还必须释放所有已经获取的资源，而不必考虑没有获取的资源。

我采用“错误变量”这一用语（error variable idiom）来辅助说明资源获取用语，它使用一个临时变量来记录函数的期望返回值。当然，相当多的函数都能实现这个功能。但是错误变量的不同点在于它通常是用来处理由于速度的因素而变得非常复杂的流程控制中的问题。错误变量有两个典型的值，0（表示成功）和负数（表示有错）。

这两个用语结合使用，我们就可以十分自然地得到符合模式的代码如下：

```
int f(void)
{
    int err;
    resource * r1, * r2;
    err = -ERR1;          /* Assume failure. */
    r1 = acquire_resource1();
    if (!r1)             /* Not acquired. */
        goto out;      /* Returns -ERR1. */

    /* Got resource r1; try for r2. */
```

```

err = -ERR2;          /* Assume failure. */
r2 = acquire_resource2();
if (!r2)              /* Not acquired. */
    goto out1;        /* Returns -ERR2. */

/* Have both r1 and r2. */
err = 0;              /* No error. */

/* ... Use r1 and r2 .... */

out2:
    release_resource(r2);

out1:
    release_resource(r1);
out:
    return err;
}

```

(注意变量err是使用错误变量的一个明确实例，同样，诸如out之类的标号则指明了资源获取用语的使用。)

如果执行到标号out2，则都已经获取了r1和r2资源，而且也都需要进行释放。如果执行到标号out1（不管是顺序执行还是使用goto语句进行跳转到），则r2资源是无效的（也可能刚被释放），但是r1资源却是有效的，而且必需在此将其释放。同理，如果标号out能被执行，则r1和r2资源都无效，err所返回的是错误或成功标志。

在这个简单的例子中，对err的一些赋值是没有必要的。在实践中，实际代码必须遵守这种模式。这样做的原因主要在于同一行中可能包含有多种测试，而这些测试应该返回相同的错误代码，因此对错误变量统一赋值要比多次赋值更为简单。虽然在这个例子中对于这种属性的必要性并不非常迫切，但是我还是倾向于保留这种特点。有关的实际应用可以参考sys_shmctl（第21654行），在第9章中还将详细介绍这个例子。

2.1.3 减少#if和#endif的使用

现在的Linux内核已经移植到不同的平台上，但是我们还必须解决移植过程中所出现的问题。大部分支持各种不同平台的代码由于包含许多预处理代码而已经变得非常不规范，例如：

```

#ifdef SOLARIS
/* ... do things the Solaris way .... */
#elif defined(HPUX)
/* ... do things the HP-UX way .... */
#elif defined(LINUX)
/* ... do things the right way .... */
#else
#error Unsupported platform.
#endif

```

这个例子试图实现操作系统的可移植性，虽然Linux关注的焦点很明显是实现代码在各种CPU上的可移植性，但是二者的基本原理是一致的。对于这类问题来说，预处理器是一种错误的解决方式。这些杂乱的问题使得代码晦涩难懂。更为糟糕的是，增加对新平台的支持有

可能要求重新遍历这些杂乱分布的低质量代码段（实际上你很难能找到这类代码段的全部）。

与现有方式不同的是，Linux一般通过简单函数（或者是宏）调用来抽象出不同平台间的差异。内核的移植可以通过实现适合于相应平台的函数（或宏）来实现。这样不仅使代码的主体简单易懂，而且在移植的过程中还可以比较容易地自动检测出你没有注意到的内容：如引用未声明函数时会出现链接错误。有时用预处理器来支持不同的体系结构，但这种方式并不常用，而相对于代码风格的变化就更是微不足道了。

顺便说一下，我们可以注意到这种解决方法和使用用户对象（或者C语言中充满函数指针的struct结构）来代替离散的switch语句处理不同类型的方法十分相似。在某些层次上，这些问题和解决方法是统一的。

可移植性的问题并不仅限于平台和CPU的移植，编译器也是一个重要的问题。此处为了简化，假设Linux只使用gcc来编译。由于Linux只使用同一个编译器，所以就没有必要使用`#if`块（或者`#ifdef`块）来选择不同的编译器。

内核代码主要使用`#ifdef`来区分需要编译或不需要编译的部分，从而对不同的结构提供支持。例如，代码经常测试SMP宏是否定义过，从而决定是否支持SMP机。

2.2 代码样例

了解Linux代码风格最好的方法就是实际研究一下它的部分代码。即使你不完全理解本节所讨论代码的细节也无关紧要，毕竟本节的主要目的不是理解代码，一些读者可以只对本节进行浏览。本节的主要目的是让读者对Linux代码进行初步了解，为今后的工作提供必要基础。该讨论将涉及部分广泛使用的内核代码。

2.2.1 printk

`printk`（25836行）是内核内部消息日志记录函数。在出现诸如内核检测到其数据结构出现不一致的事件时，内核会使用`printk`把相关信息打印到系统控制台上。对于`printk`的调用一般分为如下几类：

- 紧急事件（emergency）——例如，`panic`函数（25563行）多次使用了`printk`。当内核检测到发生不可恢复的内部错误时就会调用`panic`函数，然后尽其所能地安全关闭计算机。这个函数中调用`printk`以提示用户系统将要关闭。
- 调试——从3816行开始的`#ifdef`块使用`printk`来打印SMP逻辑单元（box）中每一个处理器的相关配置信息，但是此过程只有在使用`SMP_DEBUG`标志编译代码的情况下才能够被执行。
- 普通信息——例如，当机器启动时，内核必须估计系统速度以确保设备驱动程序能够忙等待（busy-wait）一个精确的极短周期。计算这种估计值的函数名为`calibrate_delay`（19654行），它既在19661行使用`printk`声明马上开始计算，又在19693行报告计算结果。另外，在第4章将详细的介绍`calibrate_delay`函数。

如果你已经浏览过这些参照行，你可能已经注意到`printk`和`printf`的参数十分类似：一个格式化字符串，后跟零个或者多个参数加入字符串中。格式化字符串可能是以一组“`<N>`”开始，这里的`N`是从0到7的数字，包括0和7在内。数字区分了消息的日志等级（log level），只有当日志等级高于当前控制台定义的日志等级（`console_loglevel`，25650行）时，才会打印消

息。root可以通过适当减小控制台的日志等级来过滤不是很紧急的消息。如果内核在格式化字符串中检测不到日志等级序列，那么就会一直打印消息（实际上，日志等级序列并不一定要在格式化字符串中出现，可以在格式化文本中查找到它的代码）。

从14946行开始的#define块说明了这些特殊序列，这些定义可以帮助调用者正确区分对printk的调用。简单地说，我称日志等级0到4为“紧急事件”，等级5到等级6为“普通信息”，等级7自然就是我所说的“调试”（这种分类方法并不意味着其他更好的分类方法没有用处，而只是目前我们还不关心它而已）。

在上面讨论的基础上，我们研究一下代码本身。

printk

25836: 参数fmt是printf类型的格式化字符串。如果你对“...”部分的内容不熟悉，那就需要参阅一本好的C语言参考书（在其索引中查找“变参函数，variadic function”）。另外，在安装的GNU/Linux中的stdarg帮助里也包含了一个有关变参函数的简明描述，在这儿只需要敲入“man stdarg”就可以看到。

简单地说，“...”部分提示编译器fmt后面可能紧跟着数量不定的任何类型的参数。由于这些参数在编译的时候还没有类型和名字，内核使用由三个宏va_start、va_arg和va_end组成的特殊组及一个特殊类型——va_list对它们进行处理。

25842: msg_level记录了当前消息的日志等级。它是静态的，这看起来可能会有些奇怪——为什么下一次对printk的调用需要记录日志等级呢？问题的答案是只有打印出新行（\n）或者赋给一个新的日志等级序列以后，当前消息才会结束。这样，通过在包含消息结束的新行里调用printk，就保证了在多个短期冲突的情况下，调用者只打印唯一的一个长消息。

25845: 在SMP逻辑单元中，内核可能试图从不同的CPU向控制台同时打印信息（有时在单处理机（UP）逻辑单元中也会发生同样问题，但由于中断还未被覆盖掉，所以问题也并不十分明显）。如果不进行任何协同的话，结果就将处于完全无法让人了解的杂乱无章的状态，每个消息的各个部分都和其他消息的各个部分混杂交织在一起。

相反，内核使用旋转锁（spin-lock）来控制对控制台的访问。旋转锁将在第10章进行深入介绍。

如果你对flags在传递给spin_lock_irqsave之前为什么不对它初始化感到疑惑，请不要担心：spin_lock_irqsave（对于不同的版本请分别参看12614行，12637行，12716行和12837行）是一个宏，而不是一个函数。该宏实际上是将值写入flags中，而不是从flags中读出值（在25895行中，存储在flags中的信息被spin_unlock_irqrestore回读，请参看12616行，12639行，12728行和12841行）。

25846: 初始化变量args，该变量代表printk参数中的“...”部分。

25848: 调用内核自身的vsprintf（为节省空间而省略）实现。该函数的功能与标准vsprintf函数非常相似，向buf中写入格式化文本（25634行）并返回写入字符串的长度（长度不包括最后一位终止字符0字节）。很快，你将可以看到为什么这种机制会忽略buf的前三个字符。

（正如25847行的注释中所述）我们应该注意到在这里并没有采取严格的措施

来保证缓冲器不会过载。这里系统假定1024个字符长度的buf已经足够使用（参阅25634行）。如果内核在这里能够使用vsprintf函数的话，情况就会好许多。然而，vsprintf还有另外一个参数限制了它能够写入缓冲器的字符长度。

- 25849: 计算buf中最近使用的元素，调用va_end终止对“...”参数的处理。
- 25851: 开始格式化消息的循环。其中存在一个内部循环能够处理更多内容（这一点随后就能看到），因此，每次内循环开始，都开始一个新的打印行。由于通常情况下printk只用于打印单行，所以在每次调用中，这种循环通常只执行一次。
- 25853: 如果预先不知道消息的日志等级，printk会检查当前行是否以日志等级序列开头。
- 25860: 如果不是，buf中开始未使用的三个字符就能够起作用了（第一次以后的每次循环，都会覆盖部分消息文本，但是这样并不会引起问题，因为这里的文本只是前面行中的一部分，它们已经被打印过，而且以后也不再需要了）。这样，就可以将日志等级插入buf中。
- 25866: 此处有如下属性：p指向日志等级序列（消息文本紧随其后），msg指向消息文本——请注意25852行和25865行中对msg的赋值。
- 由于已知p用来指示日志等级序列的开头——该日志等级序列可能是由函数自身所创建的，日志等级可以从p中抽出并存入msg_level中。
- 25868: 没有检测到新行，清空line_feed标志。
- 25869: 这是前面谈到过的内循环，循环将运行到本行结束（也就是检测到新行标志）或者缓冲器的末尾为止。
- 25870: 除了将消息打印到控制台之外，printk还能够记录最近打印的长度为LOG_BUF_LEN的字符组（LOG_BUF_LEN为16K，请参看25632行）。如果在控制台打开之前，内核就已经调用printk，则显然不能在控制台上正确打印消息，但是这些消息将被尽可能地存储到log_buf中（25656行）。当控制台打开以后，缓存在log_buf中的数据就可以转储并在控制台上打印出来，请参看25988行。
- log_buf是一个循环缓冲器，log_start和log_size变量（25657行和25646行）分别记录当前缓冲器的开始位置和长度。本行中的按位与（AND）操作实际上是快速求模（%）运算，它的正确性依赖于LOG_BUF_LEN的值是2的幂。
- 25872: 保存变量跟踪记录循环日志的值。显然，日志大小会不断增长，直至达到LOG_BUF_LEN的值为止。此后，log_size将保持不变，而插入新字符将导致log_start的增长。
- 25878: 请注意logged_chars（25658行）记录从机器启动之后由printk写入的所有字符的长度，它在每次循环中都会被更新，而不是在循环结束后才改变一次。基于同样的道理，log_start和log_size的处理方式也是一样。这实际上是一种优化的时机，本书将在结束对函数的介绍之后再对它进行详细讨论。
- 25879: 消息被分为若干行，这当然要使用新行标志符来进行分割。一旦内核检测到新行标志符，就写入一个完整行，从而内循环的执行也可以提前终止。
- 25884: 在这里我们先不考虑内部循环是否会提前退出，从msg到p的字符序列是专门提供给控制台使用的（这种字符序列我称之为行，但是不要忘了，这里的行可能并不意味着新行终止，因为buf也许还没有终止）。如果该行的日志等级高于系统控制

台定义的日志等级，而且当前又有控制台可供打印，那么就能够正确打印该行。（记住，`printk`可能在所有控制台打开之前就已经被调用过了。）

如果在该消息块中没有发现日志等级序列，并且在前面的`printk`调用中也没有对`msg_level`赋值，那么本行中的`msg_level`就是-1。由于`console_loglevel`总不小于1（除非`root`通过`sysctl`接口锁定），于是总是可以打印这些行。

25886: 本行应该能够被打印。`printk`通过遍历打开的控制台驱动链表告知每一个控制台驱动去打印当前行设备驱动在本书的讨论范围之外，因此，控制台驱动代码则并不包含在内）。

25888: 请注意这里消息文本的开头使用的是`msg`而不是`p`，这样就在没有日志等级序列的情况下写入消息了。然而，日志等级序列已经被存储到`log_buf`缓冲器中了。这样就使后来能够访问`log_buf`以获取消息日志等级的代码（请参看25998行），不会再产生显示混乱信息序列的现象。

25892: 如果内层`for`循环发现一新行，那么`buf`中的剩余字符（如果有的话）将被认为是新的消息，因此`msg_level`会被重置。但是无论怎样，外层循环都会持续到`buf`清空为止。

25895: 释放在25845行获取的控制台锁（`console lock`）。

25896: 唤醒等待被写入控制台日志的所有进程。注意即使没有文本被实际写入任何控制台，这个过程也仍然会发生。这样处理是正确的，因为无论是否要往控制台中写入文本，等待进程实际上都是在等待从`log_buf`中读出信息。在25748行，进程被转入休眠状态以等待`log_buf`的活动。在休眠、唤醒和等待队列中所使用的机制将在下一节中进行讨论。

25897: 返回日志中写入的字符长度。

如果对于每个字符的处理工作都能减少一点，那么从25869行开始的`for`循环就执行得更快一点。当循环存在时，我们可以通过只在循环退出时将`logged_chars`更新一次来稍微提高运行速度。然而我们还可以通过其他努力来提高速度。由于我们可以预知消息的长度，因此`log_size`和`log_start`可以到最后再增长。让我们来实验一下这样能否提高速度，下面是一段经过理想优化的代码：

```
do {
    static int wrapped = 0;
    const int x = wrapped
        ? log_start
        : log_size;
    const int lim = LOG_BUF_LEN - x;
    int n = buf_end - p;
    if (n >= lim)
        n = lim;

    memcpy(log_buf + x, p, n);
    p += n;

    if (log_size < LOG_BUF_LEN)
        log_size += n;
    else {
```

```

        wrapped = 1;
        log_start += n;
        log_start &= LOG_BUF_LEN - 1;
    }
} while (p < buf_end);

```

请注意循环通常只需要执行一次，只有在log_buf末尾写入信息需要折行时才会多次执行。因而log_size和log_buf只需要更新一次（或者当写入需要换行时是两次）。

这时速度的确提高了，但是有两个原因使我们并不能这样做。首先，内核可能有自己特有的memcpy函数，我们必须确保对memcpy的调用不会再次进入对printk的调用（有一部分内核移植版定义了自己特有的速度较快的memcpy函数版本，因此所有的移植都要在这一点上保持一致）。如果memcpy调用printk来报告失败，那么就有可能触发无限循环。

然而在这一点也不并不是真的无药可救。使用这种解决方案的最大问题在于该内核循环的形式中也要留意新行标志符，因此使用memcpy将整个消息拷贝到log_buf中是不正确的：如果此处存在新行，我们将无法对其进行处理。

我们可以试验一个一箭双雕的办法。下面这种替代的尝试虽然可能比前面那种初步解决方法速度要慢，但是它保持了内核版本的语意：

```

/* In declarations section: */
int n;
char * start;
static char * log = log_buf;
/* ... */

for (start = p; p < buf_end; p++) {
    *log++ = *p;
    if (log >= (log_buf + LOG_BUF_LEN))
        log = log_buf; /* Wrap. */

    if (*p == '\n') {
        line_feed = 1;
        break;
    }
}

/* p - start is number of chars copied. */
n = p - start;
logged_chars += n;
/*
 * Exercise for the reader:
 * Also use n to update log_size and log_start.
 * (It's not as simple as it may look.)
 */

```

（请注意gcc的优化器十分灵敏，它足以能检测到循环内部的表达式log_buf+LOG_BUF_LEN并没有改变，因此在上面的循环中试图手工加速计算是没有任何效果的。）

不幸的是，这种方法并不能比现在的内核版本在速度上快许多，而且那样会使得代码晦涩难懂（如果你编写过更新log_size和log_start的代码，你就能清楚地了解这一点）。你可以自

已决定这种折衷是否值得。然而无论怎样，我们学到了一些东西，通常，不管成功与否，改进内核代码都可以加深你对内核工作原理的理解。

2.2.2 等待队列

前一节我们曾简要的提到进程（也就是正在运行的程序）可以转入休眠状态以等待某个特定事件，当该事件发生时这些进程能够被再次唤醒。内核实现这一功能的技术要点是把等待队列（wait queue）和每一个事件联系起来。需要等待事件的进程在转入休眠状态后插入到队列中。当事件发生之后，内核遍历相应队列，唤醒休眠的任务让它投入运行状态。任务负责将自己从等待队列中清除。

等待队列的功能强大得令人吃惊，它们被广泛应用于整个内核中。更重要的是，实现等待队列的代码量并不大。

1. wait_queue结构

18662: 简单的数据结构就是等待队列节点，它包含两个元素：

- task——指向struct task_struct结构的指针，它代表一个进程。从16325行开始的struct task_struct结构将在第7章中进行介绍。
- next——指向队列中下一节点的指针。因而，等待队列实际上是一个单链表。

通常，我们用指向等待队列队首的指针来表示等待队列。例如，printk使用的等待队列log_wait（25647行）。

2. wait_event

16840: 通过使用这个宏，内核代码能够使当前执行的进程在等待队列wq中等待直至给定condition（可能是任何的表达式）得到满足。

16842: 如果条件已经为真，当前进程显然也就无需等待了。

16844: 否则，进程必须等待给定条件转变为真。这可以通过调用__wait_event来实现（16824行），我们将在下一节介绍它。由于__wait_event已经同wait_event分离，已知条件为假的部分内核代码可以直接调用__wait_queue，而不用通过宏来进行冗余的（特别是在这些情况下）测试，实际上也没有代码会真正这样处理。更为重要的是，如果条件已经为真，wait_event会跳过将进程插入等待队列的代码。

注意wait_event的主体是用一个比较特殊的结构封闭起来的：

```
do {
    /* ... */
} while (0)
```

奇怪的是，这个小技巧并没有得到应有的重视。这里的主要思路是使被封闭的代码能够像一个单句一样使用。考虑下面这个宏，该宏的目的是如果p是一个非空指针，则调用free：

```
#define FREE1(p) if (p) free(p)
```

除非你在如下所述的情况下使用FREE1，否则所有调用都是正确有效的：

```
if (expression)
    FREE1(p);
else
    printf("expression was false.\n");
```

FREE1经扩展以后，else就和错误的if（FREE1的if）联系在一起。

有些程序员通过如下途径解决这种问题:

```
#define FREE2(p) if (p) { free(p); }
#define FREE3(p) { if (p) { free(p); } }
```

这两种方法都不尽人意, 程序员在调用宏以后自然而然使用的分号会把扩展信息弄乱。以FREE2为例, 在宏展开之后, 为了使编译器能更准确地识别, 我们还需要进行一定的缩进调节, 最终代码如下所示:

```
if (expression)
    if (p) { free(p); }
;
else
    printf("expression was false.\n");
```

这样就会引起语法错误——else和任何一个if都不匹配。FREE3从本质上讲也存在同样的问题。而且在研究问题产生原因的同时, 就能够明白为什么宏体里是否包含if是无关紧要的。不管宏体内部内容如何, 只要使用一组括号来指定宏体, 就会碰到相同的问题。

引入do/while(0)技巧能够克服前面所出现的所有问题, 现在我们可以编写FREE4。

```
#define FREE4(p) \
do { \
    if (p) \
        free(p); \
} while (0)
```

将FREE4和其他宏一样插入相同代码之后, 宏展开后其代码如下所示(为清晰起见, 我们再次调整了缩进格式):

```
if (expression)
do {
    if (p)
        free(p);
} while (0); /* ";" following macro. */
else
    printf("expression was false.\n");
```

这段代码当然可以正确执行。编译器能够优化这个伪循环, 舍弃循环控制, 因此执行代码并没有速度的损失, 我们也从而得到了能够实现理想功能的宏。

虽然这是一个可以接受的解决方案, 但是我们不能不提到的是编写函数要比编写宏好得多。不过如果你不能提供函数调用所需的开销, 那么就需要使用内联函数。这种情况虽然在内核中经常出现, 但是在其他地方就要少得多。(不可否认, 当使用C++、gcc或者任何实现了将要出现的修正版ISO标准C的编译器时, 这种方案只是一种选择, 就是最后为C增加内联函数。)

3. __wait_event

16824: __wait_event使当前进程在等待队列wq中等待, 直至condition为真。

16829: 通过调用add_wait_queue (16791行), 局部变量__wait可以被链接到队列上。注意__wait是在堆栈中而不是在内核堆中分配空间, 这是内核中常用的一种技巧。在宏运行结束之前, __wait就已经被从等待队列中移走了, 因此等待队列中指向它的指针总是有效的。

16830: 重复分配CPU给另一个进程直至条件满足，这一点将在下面几节中讨论。

16831: 进程被置为TASK_UNINTERRUPTIBLE状态 (16190行)。这意味着进程处于休眠状态，不应被唤醒，即使是信号也不能打断该进程的休眠。信号在第6章中介绍，而进程状态则在第7章中介绍。

16832: 如果条件已经满足，则可以退出循环。

请注意如果在第一次循环时条件就已经满足，那么前面一行的赋值就浪费了（因为在循环结束之后进程状态会立刻被再次赋值）。__wait_event假定宏开始执行时条件还没有得到满足。而且，这种对进程状态变量state的延迟赋值也并没有什么害处。在某些特殊情况下，这种方法还十分有益。例如当__wait_event开始执行时条件为假，但是在执行到16832行时就为真了。这种变化只有在为有关进程状态的代码计算condition变量值时才会出现问题。但是在代码中这种情况我没有发现。

16834: 调用schedule (26686行，在第7章中讨论) 将CPU转移给另一个进程。直到进程再次获得CPU时，对schedule的调用才会返回。这种情况只有当等待队列中的进程被唤醒时才会发生。

16836: 进程已经退出了，因此条件必定已经得到了满足。进程重置TASK_RUNNING的状态 (16188行)，使其适合CPU运行。

16837: 通过调用remove_wait_queue (16814行) 将进程从等待队列中移去。wait_event_interruptible和__wait_event_interruptible (分别参见16868行和16847) 基本上与wait_event和__wait_event相同，但不同的是它们允许休眠的进程可以被信号中断。信号将在第6章中介绍。

请注意wait_event是被如下结构所包含的。

```
((
/* ... */
))
```

和do/while(0)技巧一样，这样可以使被封闭起来的代码能够像一个单元一样运行。这样的封闭代码就是一个独立的表达式，而不是一个独立的语句。也就是说，它可以求值以供其他更复杂的表达式使用。发生这种情况的原因主要在于一些不可移植的gcc特有代码的存在。通过使用这类技巧，一个程序块中的最后一个表达式的值将定义为整个程序块的最终值。当在表达式中使用wait_event_interruptible时，执行宏体后赋__ret的值为宏体的值 (参见16873行)。对于有Lisp背景知识的程序员来说，这是个很常见的概念。但是如果你仅仅了解一点C和其他一些相关的过程性程序设计语言，你可能就会觉得比较奇怪。

__wake_up

26829: 该函数用来唤醒等待队列中正在休眠的进程。它由wake_up和wake_up_interruptible调用 (请分别参见16612行和16614行)。这些宏提供mode参数，只有状态满足mode所包含的状态之一的进程才可能被唤醒。

26833: 正如将在第10章中详细讨论的那样，锁 (lock) 是用来限制对资源的访问，这在SMP逻辑单元中尤其重要，因为在这种情况下当一个CPU在修改某数据结构时，另一个CPU可能正在从该数据结构中读取数据，或者也有可能两个CPU同时对同

一个数据结构进行修改，等等。在这种情况下，受保护的资源显然是等待队列。非常有趣的是所有的等待队列都使用同一个锁来保护。虽然这种方法要比为每一个等待队列定义一个新锁简单得多，但是这就意味着SMP逻辑单元可能经常会发现自己正在等待一个实际上并不必须的锁。

26838: 本段代码遍历非空队列，为队列中正确状态的每一个进程调用wake_up_process (26356行)。如前所述，进程（队列节点）在此可能并没有从队列中移走。这在很大程度上是由于即使队列中的进程正在被唤醒，它仍然可能希望继续存在于等待队列中，这一点正如我们在__wait_event中发现的问题一样。

2.2.3 内核模块

整个内核并不需要同时装入内存。应该确认，为保证系统能够正常运行，一些特定的内核必须总是驻留在内存中，例如，进程调度代码就必须常驻内存。但是内核其他部分，例如大部分的设备驱动就应该仅在内核需要的时候才装载，而在其他情况下则无需占用内存。

举例来说，只有在内核真正和CD-ROM通讯时才需要使用完成内核与CD-ROM通讯的设备驱动程序，因此内核可以被设置为在和设备通讯之前才装载相应代码。内核完成和设备的通讯之后可以将这部分代码丢弃。也就是说，一旦代码不再需要，就可以从内存中移走。系统运行过程中可以增减的这部分内核称为内核模块。

内核模块的优点是可以简化内核自身的开发。假设你购买了一个新的高速CD-ROM驱动器，但是现有的CD-ROM驱动程序并不支持该设备。你自然就希望增加对这种高速模式的支持以提高系统光驱设备的性能。如果作为内核模块来编译驱动程序，你的工作将会方便得多：编译驱动程序、加载到内核、测试、卸载驱动程序、修改驱动程序、再次加载驱动程序到内核、测试，如此周而复始。如果你的驱动程序是直接编辑在内核中的，那么你就必须重新编译整个内核并且在每次修改驱动程序之后重新启动机器。这样慢得很多。

自然，你也必须留意内核模块。对于指明其他内核模块在磁盘上的驻留位置的那些模块，一定不能从内存中卸载，否则，内核将只能通过访问磁盘来装载处理磁盘访问的内核模块，这是不可能实现的。这也是我们要选择把部分内核作为模块编译还是直接编译进内核使其常驻内存的又一个原因。知道自己系统的设置方式，因而也就可以选择正确使用的方式（如果为了确保安全，可以简单的忽略内核模块系统的优点，而把所有的内容都编译到内核里面）。

内核模块会带来一些速度上的损失，这是因为一些必需的代码现在并不在RAM中，必需要从磁盘读入。但是整个系统的性能通常会有所提高，这主要是因为通过丢弃暂时不使用的模块可以释放出额外的RAM供应用程序使用。如果这部分内存被内核所占用，应用程序将只能更加频繁地进行磁盘交换，而这种磁盘交换会显著地降低应用程序的性能（磁盘交换将在第8章中讨论）。

内核模块还会带来因复杂度的增加所造成的开销，这是因为在系统运行的过程中，移出部分内核需要额外的代码。然而，复杂度的开销是可以管理的。通过使用外部程序来代理一些必需的工作还可以更进一步降低复杂度的开销（更为确切的说法是，这样做不是减少了复杂度的开销，而是把复杂度的开销重新分配了一下）。这是对内核模块原理的一个小小的扩展：即使是内核的支持模块，对于内核来说也只是外部的、部分可用的，只有在需要的时候才被装入内存。

通常用于这种目的程序称为modprobe。有关的modprobe代码超出了本书的范围，但是在Linux的每个发行版本中都包含有它。本节的剩余部分将讨论同modprobe协同工作，以装载内核模块的内核代码。

1. request_module

24432: 作为函数说明之前的注释，request_module是一个函数。内核的其他模块在需要装载其他内核模块的时候，都必须调用这个函数。就像内核处理其他工作一样，这种调用也是为当前运行的进程进行的。从进程的角度来看，这种调用的请求通常是隐含的——正在执行进程其他请求的内核可能会发现，必须调入一个模块才能够完成该请求。例如，请参见10070行，这里是一些将在第7章中讨论的代码。

24446: 以内核中的一个独立进程的形式执行exec_modprobe函数（24384行）。这并不能只通过函数的简单调用实现，因为exec_modprobe要继续调用exec来执行一个程序。因此，对函数exec_modprobe的简单调用将永远不会有返回。

这和使用fork以准备exec调用十分类似，你可以认为kernel_thread对内核来说就是较低版本的fork，虽然两者有很大不同。fork是从指定函数开始执行新的进程，而不是从调用者的当前位置开始运行。正如fork一样，kernel_thread返回的值是新进程的进程号。

24448: 和fork一样，从kernel_thread返回的负值表示内部错误。

24455: 正如函数中论述的一样，大部分的信号将因当前进程而被暂时阻塞。

24462: 等待exec_modprobe执行完毕，同时指出所需要的模块是已经成功装入内存，还是装载失败了。

24465: 结束运行，恢复信号。如果exec_modprobe返回错误代码，则打印错误消息。

2. exec_modprobe

24384: exec_modprobe运行在内核增加内核模块的程序。这里的模块名是一个void*的指针，而不是char*的指针。原因简单说来就是kernel_thread产生的函数通常都使用void*指针参数。

24386: 设置modprobe的参数列表和环境。modprobe_path（24363行）用来定位modprobe程序的位置。它可以通过内核的sysctl特性来修改，这一点将在第11章中介绍（参见30388行）。这意味着root可以动态选择不同于/sbin/modprobe的程序来运行，以适应当modprobe被安装到其他地方或者使用修改过的modprobe替换掉了原有的modprobe之类的情况。

24400: （正如代码中描述的一样）出于安全性考虑，丢弃所有挂起的信号和信号句柄（handlers）。这里最重要的部分是对flush_signal_handlers的调用（28041行），它使用内核默认的信号句柄代替所有用户定义的信号句柄。如果在此时有信号被传送到内核，它将获得默认响应——通常是忽略信号或杀死进程。但是不管怎样都不会引起安全风险。由于该函数从触发它的进程中分离出来（如前所述），所以，不管原始进程在此处是否改变其原来分配的信号、句柄都不会产生任何影响。

24405: 关闭调用进程打开的所有文件。最重要的是，这意味着modprobe程序不再从调用进程中继承标准输入输出和标准错误。这很有可能会引起安全漏洞（这可能是在替代modprobe的程序中引起的问题，但是modprobe本身实际上并不关心这个差异）。

24413: modprobe程序作为root运行, 它拥有root所拥有的所有权限。和整个内核中其他地方一样, 请注意root使用用户ID号0的假定在这里已经被写入程序。用户ID号和权能系统 (capability system, 在接下来的几行中会用到) 将在第7章中介绍。

24421: 试图执行modprobe程序。如果尝试失败, 内核将使用printk打印错误消息并返回错误代码。这里是可能产生printk的缓冲器过载的地点之一。module_name的长度并没有明确限制, 就我们对该调用的看法而言, 它可能长达一百万个字符。为防止printk缓冲器过载, 你必需遍历所有对于该函数的调用 (实际上是对request_module的调用), 以保证每个调用者使用足够短的、不会为printk造成麻烦的模块名。

24427: 当execve成功执行时, 它不会返回任何结果, 因此本处是不可能执行到的。但是编译器却并不知道这一点, 因此, 此处使用了return语句以保证gcc不出错。

对于内核的进一步讨论将超出本章的既定范围, 因此在这个问题上我们到此为止。然而本书中也包括了其他必需的内核代码。在读完第4章和第5章之后, 也许你会希望再次仔细阅读一下这部分内容。有关这个问题的两个文件是include/linux/module.h (从15529行开始) 和/kernel/module.c (从24476行开始)。和sys_create_module (24586行)、sys_init_module (24637行)、sys_delete_module (24860行) 和sys_query_module (25148行) 四个函数需要特别注意一样, struct module (15581行) 也要特别引起注意。这些函数实现了modprobe及insmod、lsmod和rmmod所使用的系统调用, 以完成模块的装载、定位和卸载。

内核触发直接回调内核程序的现象看起来很令人奇怪。但是, 实际上进行的工作不止于此。例如, modprobe必须实际访问磁盘以搜寻要装载的模块。而且更为重要的一点是, 这种方法赋予root对内核模块系统更多的控制能力。这主要是因为root也可以运行modprobe及相关程序。因此, root既可以手工装载、查询、卸载模块, 也可以由内核自动完成。

2.3 配置与编译内核

你可能仅仅研读、欣赏而并不修改Linux内核源代码。但是, 更普遍的情况是, 用户有强烈的愿望去改进内核代码并完成相应的测试, 这样我们就需要知道如何重建内核。本节就是要告诉你如何实现这一点, 而最终则归结于如何把你所做的修改发行给别人, 以使得每个人都能从你的工作中受益。

2.3.1 配置内核

编译内核的第一步就是配置内核, 这是增加或者减少对内核特性的支持及修改内核的一些特性的必要步骤。例如, 你可以要求内核为自己的声卡指定一个不同的DMA通道。如果内核配置和你的需要相同, 那么你可以直接跳过本节, 否则请继续阅读以下内容。

为了完成内核的配置, 请先切换到root用户, 然后转入如下内核源程序目录:

```
cd /usr/src/linux
```

接着敲入如下命令组:

```
make config  
make menuconfig  
make xconfig
```

这三条命令都可以用来配置内核，但它们发挥作用的方式各不相同：

- **make config**——三种方法中最简单也是最枯燥的一种。但是最基本的一点是，它可以适应任何情况。通过为每一个内核支持的特性向用户提问的方式来决定在内核中需要包含哪些特性。对于大多数问题，你只要回答y (yes, 把该特性编译进内核中)、m (作为模块编译) 或者n (no, 根本不对该特性提供支持)。在决定之前用户应该考虑清楚，因为这个过程是不可逆的。如果你在该过程中犯了错误，就只能按Ctrl+C退出。你也可以敲入?以获取帮助。图2-1显示了这种方法在X终端上运行的情况。

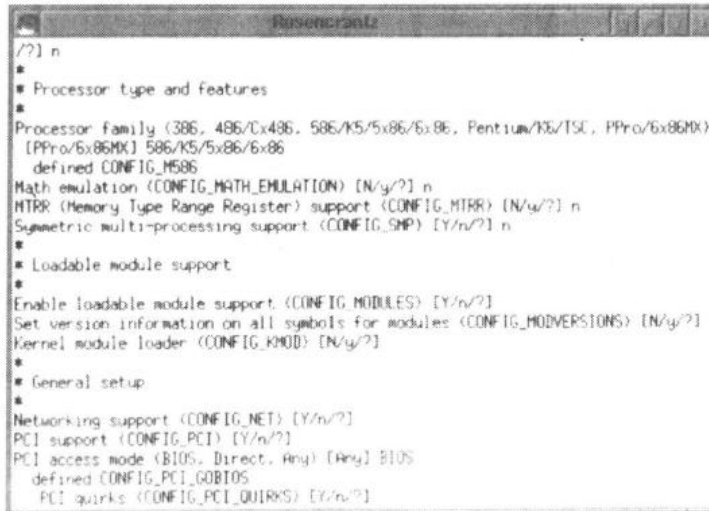


图2-1 运行中的make config

幸运的是，这种方法还有一些智能。例如，如果你对SCSI支持回答no，那么系统就不会再询问你有关SCSI的细节问题了。而且你可以只按回车键以接受默认的选择，也就是当前的设置（因此，如果当前内核将对于SCSI的支持编译进了内核，在这个问题上按回车键就意味着继续把对SCSI的支持编译进内核中）。即使是这样，大部分用户还是宁愿使用另外的两种方法。

- **make menuconfig**——一种基于终端的配置机制，用户拥有通过移动光标来进行浏览等功能。图2-2显示了在X终端上运行的make menuconfig。虽然在控制台上显示的是彩色，但是在终端上的显示仍然相当单调。使用menuconfig必须要有相应的ncurses类库。

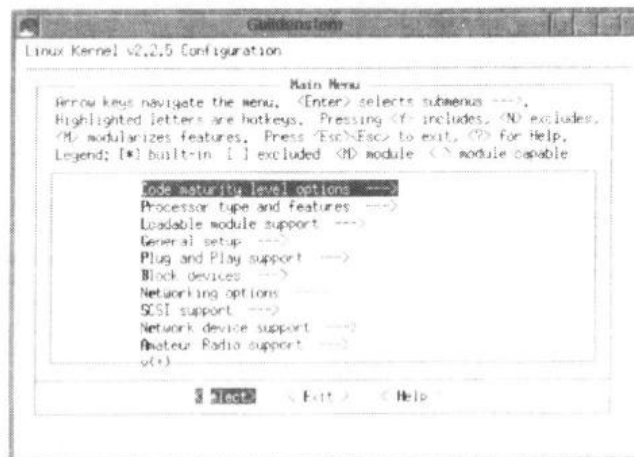


图2-2 运行中的make menuconfig

- `make xconfig`——这是我最喜欢的一种配置方式。只有你能够在X server上用root用户身份运行X应用程序时，这种配置方式才可以使用（有些偏执的用户就不愿意使用这种方式）。你还必须拥有Tcl窗口系统，这实际上还意味着你必须拥有Tcl、Tk以及一个正在运行的X安装程序。作为补偿，用户获得的是更漂亮的、基于X系统的以及和menuconfig功能相同的配置方法。图2-3显示了在这种方法运行过程中打开“可装载模块支持（Loadable module support）”子窗口的情况。

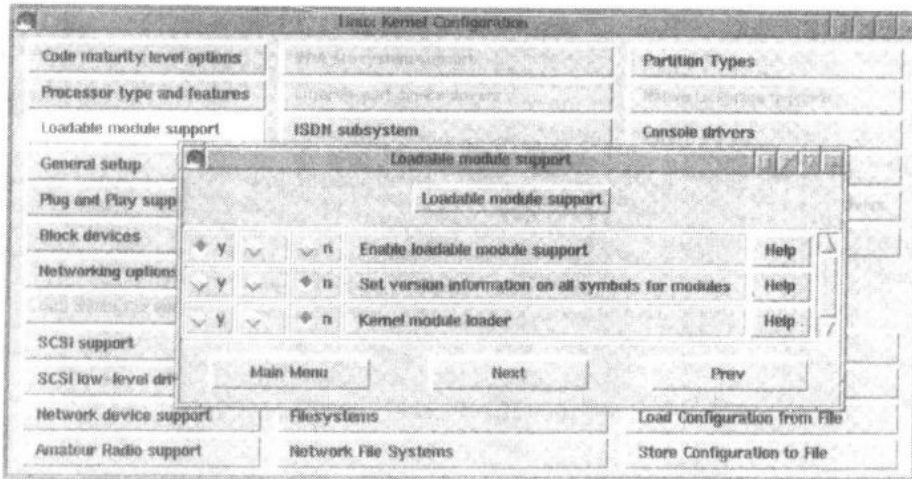


图2-3 运行中的make xconfig

如上所述，这三种方法都实现了相同的功能：它们都生成在构建内核时使用的.config文件。而唯一的区别在于创建这个文件时的难易程度不同。

2.3.2 构建内核

构建内核要做的工作要比配置内核所做的工作少得多。虽然有几种方式都能实现这一功能，但是选择哪一种依赖于你希望怎样对系统进行设置。长期以来，我已经形成了如下的习惯。虽然这种习惯比我所必须要做的略微多一些，但是它包含了所有基本的问题。首先，如果你还不在内核源程序目录中，请先再次转入这一目录：

```
cd /usr/src/linux
```

现在，切换到root用户，使用下面显示的命令生成内核。现在在shell中敲入下面的命令，注意make命令因为空间关系分成了两行，但实际上这在shell输入时是一个只有一行的命令：

```
make dep clean zliilo boot
modules modules_install
```

当给出了如上多个目标时，除非前面所有的目标都成功了，否则make能够知道没有必要继续尝试下面的目标。因此，如果make能够运行结束，成功退出，那么这就意味着所有的目标都正确构建了。现在你可以重新启动机器以运行新的内核。

2.3.3 备份的重要性

当修改（fooling）内核时，你必须准备一个能够启动的备用内核。实现该目的的一种方式是通过配置Linux加载程序（LILO）以允许用户选择启动的内核映象，其中之一是从没有

修改过的内核的备份（我总是这样做的）。

如果你比较有耐心，那么你就可以使用zdisk目标而不使用zli10目标；它可以把能够启动的内核映像写入软盘中。这样你就可以通过在启动时插入软盘的方式启动你的测试内核；如果没有插入软盘，则启动正常的内核。

但是请注意：内核模块并没有被装载到软盘中，它们实际上是装在硬盘中的（除非你愿意承担更多的麻烦）。因此，如果你弄乱了内核模块，即使是zdisk目标也救不了你。实际上，上面提到的这两种方法都存在这个问题。虽然有比较好的解决方法可用，但是最简单的方法（也就是我所使用的方法）是把备份内核作为严格独立的内核来编译，而不使用可装载模块的支持。通过这种方法，即使我弄乱了内核而不得不使用备份启动系统，那么不管问题是实验性内核不正确还是内核模块的原因都无关紧要。不管怎样，在备份的内核中已经有我需要的所有东西了。

由于用户所做的修改可能导致系统的崩溃，如损坏磁盘上的数据等等，并不仅仅只是打乱设备驱动程序或文件系统，在测试新内核之前，备份系统的最新数据也是一个英明的决策（虽然设备驱动程序的开发不是本书的主题，但是必需指出的是，设备驱动程序的缺陷可能会引起系统的物理损坏。例如显示器是不能备份的，而且因价格昂贵而不易替换）。作为一个潜在的内核黑客，你的最佳投资（当然是读过本书以后）是一个磁带驱动器和充足的磁带。

2.3.4 发布你的改进

下面是有关发布你所做修改的一些基本规则：

- 检查最新发行版本，确保你所处理的不是已经解决了的问题。
- 遵守Linux 内核代码编写的风格。简要的说就是8字符缩进以及K&R括号风格（if, else, for, while, switch或者do后面同一行中紧跟着开括号）。在内核源程序目录下面的文档编写和代码风格文件给出了完整的规则，不过我们已经介绍了其中的关键部分。注意本书中包含的源程序代码为节省空间而进行了大量的重新编辑，在该过程中我可能打破了其中的一些规则。
- 独立发行相对无关的修改。这样，只想使用你所做的某部分修改的人就可以十分方便地获得想要的东西，而不用一次检验所有的修改内容。
- 让使用你所做修改的用户清楚他们可以从你的修改中获取什么。同样，你也应该给出这些问题的可信度。你是15min之前才匆匆完成你的修改，甚至还没有时间对它们进行编译，还是已经在你和你的朋友的系统中已长期稳定地运行过这个修改？

假设现在你已经准备好发行自己的修改版本了，那么要做的第一步是建立一个说明你所做的修改的文件。你可以使用diff程序自动创建这个文件。结果或者被称为diffs、或者在Linux中更普遍的被称作补丁（patch）。

发布的过程十分简单。假设原来没有修改过的源程序代码在linux-2.2.5目录下，而你修改过的源程序代码在linux-my目录下，那么只要进行如下的简单工作就可以了（只有在链接不存在的情况下才需要执行ln）：

```
ln -s linux my linux
make -C linux-2.2.5 distclean
make -C linux distclean
diff -urN linux-2.2.5 linux >my.patch
```

现在，输出文件my.patch包含了其他用户应用这个修改程序时所需要的一切内容。（警告：如上所述，两个源程序间的所有差别都会包含在这个补丁文件中。diff不能区分修改部分之间的关系，所以就把它们都罗列了出来。）如果补丁文件相对较小，你可以使用邮件直接发往内核邮件列表。如果补丁很大，那么就需要通过FTP或者Web站点发布。这时发给邮件列表的信件中就只需要包含一个URL。

Linux内核邮件列表的常见问题解答（FAQ）文件位于<http://www.eecs.uc.edu/~rreilova/linux/lkmlfaq.html>。该FAQ中包含了邮件列表的订阅、邮件发布及阅读邮件列表的注意事项等等。

顺便提一下，如果你想随时了解内核更新开发的进程，向你推荐下面这个具有很高价值的内核交流站点：<http://www.kt.opensrc.org>。

第3章 内核体系结构概述

本章从较高层次上对内核进行说明。从顺序上来说，本章首先介绍内核设计目标，接下来介绍内核体系结构，最后介绍内核源程序目录结构。

3.1 内核设计目标

Linux 的内核展现出了几个相互关联的设计目标，它们依次是：清晰性、兼容性、可移植性、健壮性、安全性和速度。这些目标有时是互补的，有时则是矛盾的。但是它们被尽可能的保持在相互一致的状态，内核设计和实现的特性通常都要回归到这些问题上来。本节接下来的部分将分别讨论这些设计目标，同时还将对它们之间的取舍与平衡进行简要的说明。

3.1.1 清晰性

稍微简化点说，内核目标是在保证速度和健壮性的前提下尽量清晰。这和现在的大多数应用程序的开发有所区别，后者的目标通常是在保证清晰性和健壮性的基础上尽量提高速度。因而在内核内部，速度和清晰性经常是一对矛盾。

在某种程度上，清晰性是健壮性的必要补充：一个很容易理解的实现方法比较容易证明是正确的；或者即使不正确，也能比较容易的找出其问题所在。从而这两个目标很少会发生冲突。

但是清晰性和速度通常却是一对矛盾。经过仔细手工优化的算法通常都使用了编译器生成代码的类似技术，很少可能是最清晰的解决方案。当内核中清晰性和速度要求不一致时，通常都是以牺牲清晰性来保证速度的。即便如此，程序员仍然清楚地知道清晰性的重要性，而且他们也做了大量的工作以使用最清晰的方法保证速度。

3.1.2 兼容性

正如第1章中所述，Linux最初的编写目的是为了实现在一个完整的、与Unix兼容的操作系统内核。随着开发过程的展开，它也开始以符合POSIX标准为目标。就内核而言，兼容Unix（至少是同某一现代的Unix实现相兼容）和符合POSIX标准并没有什么区别，因此我们也不会在这个问题上详细追究。

内核提供了另外一种类型的兼容性。基于Linux的系统能够提供可选择的对Java.class文件的本地运行支持（据说Linux是第一个提供这种支持的操作系统）。尽管实际负责Java程序解释执行的是另外一个Java虚拟机进程，该虚拟机并没有内置到内核中。但是内核提供的这种机制可以使得这种支持对用户是透明的。通过内核本身提供的程度不同的支持（这并不代表大部分工作像Java的解决方式一样能够通过外部进程实现），对其他可执行文件格式的支持也能够以同样的方式插入内核中。这方面的内容将在第7章中详细介绍。

另外需要说明的是，GNU/Linux系统作为一个整体通过DOSEMU仿真机器提供了对DOS可执行程序的支持，而且也通过WINE设计提供了对Windows可执行程序的部分支持。系统还

以同样的方式通过SAMBAs提供了对Windows兼容文件和打印服务的支持。但是这些都不是同内核密切相关的问题，因此在本书中我们不再对它们进行讨论。

兼容性的另外一个方面是兼容异种文件系统，本章中稍后会有更为详细的介绍，但是大部分内容已经超出了本书的范围。Linux能够支持很多文件系统，例如ext2（“本地”文件系统）、ISO-9660（CD-ROM使用的文件系统）、MS-DOS、网络文件系统（NFS）等许多其他文件系统。如果你有使用其他操作系统格式的磁盘或者一个网络磁盘服务器，那么Linux将能够和这些不同的文件系统进行交互。

兼容性的另外一个问题是网络，这在当今Internet流行的时代尤为重要。作为Unix的一个变种，Linux自然从很早就开始提供对TCP/IP的支持。内核还支持其他许多网络协议，它们包括AppleTalk协议的代码，这使得Linux单元可以和Macintosh机自由通讯；Novell的网络协议，也就是网络报文交换（IPX）、分组报文交换（SPX）和NetWare核心协议（NCP）；IP协议的新版本IPv6以及其它一些不太出名的协议。

兼容性考虑的最后一个方面是硬件兼容性。似乎每个不常见的显卡、市场份额小的网卡、非标准的CD-ROM接口和专用磁带设备都有Linux的驱动程序（只要它不是专为特定操作系统设计的专用硬件）。而且只要越来越多的厂商也逐渐认识到Linux的优势，并为更容易地实现向Linux上移植而开放相应的源程序代码，Linux对硬件支持会越来越越好。

这些兼容性必须通过一个重要的子目标：模块度（modularity）来实现。在可能的情况下，内核只定义了子系统的抽象接口，这种抽象接口可以通过任何方法来实现。例如，内核对于新文件系统的支持将简化为对虚拟文件系统（VFS）接口的代码实现。第7章中介绍的是另外一个例子，内核对二进制句柄的抽象支持是实现诸如Java之类的新可执行格式的支持的方法。增加新的可执行格式的支持将转变为对相应的二进制句柄接口的实现。

3.1.3 可移植性

与硬件兼容性相关的设计目标是可移植性，也就是在不同硬件平台上运行Linux的能力。系统最初是为运行在标准IBM兼容机上的Intel x86 CPU而设计的，当时根本没有考虑到可移植性的问题。但是从那以后情况已经发生了很大的变化。现在正式的内核移植包括向基于Alpha、ARM、Motorola 68x0、MIPS、PowerPC、SPARC及SPARC-64 CPU系统的移植。因而，Linux可以在Amigas、旧版或新版的Macintosh、Sun和SGI工作站及NeXT机等机器上运行。而且这些还只是标准内核发行版本的移植范围。从老的DEC VAX到3Com掌上系列个人数字助理（例如Palm III）的非正式的移植工作也在不断进行中。成功的非正式移植版本后来通常都会变成正式的移植版本，因此这些非正式的移植版本很多最终都会出现在主开发树中。

广泛平台支持之所以能够成功的部分原因在于内核把源程序代码清晰地划分为体系结构无关部分和体系结构相关部分。在本章的后续部分将对这个问题进行更深入的讨论。

3.1.4 健壮性和安全性

Linux必须健壮、稳定。系统自身应该没有任何缺陷，它还应该可以保护进程（用户）以防止互相干扰，这就像把整个系统从其他系统中隔离开来加以保护一样。后一种考虑很大程度上是受信任的用户空间应用程序领域的问题，但是内核至少也应该提供支撑安全体系的原语（primitive）。健壮性和安全性比任何别的目标都要重要，包括速度。（系统崩溃的速度很

快又有什么好处呢?)

保证Linux健壮性和安全性的唯一最重要的因素是其开放的开发过程,它可以被看作是一种广泛而严格的检查。内核中的每一行代码、每一个改变都会很快由世界上数不清的程序员检验。还有一些程序员专门负责寻找和报告潜在的缺陷——他们这样做完全是出于自己的个人爱好,因为他们也希望自己的Linux系统能够健壮安全。以前检查中所没有发现的缺陷可以通过这类人的努力来定位、修复,而这种修复又合并进主开发树以使所有的人都能够受益。安全警告和缺陷报告通常在几天甚至几个小时内就能够得到处理和修复。

Linux可能并不一定是现有的最安全的操作系统(很多人认为这项桂冠应该属于OpenBSD,它是一个以安全性为主要目标的Unix变种),但是它是一个有力的竞争者。而且Linux健壮性远没有发展到尽头。

3.1.5 速度

这个术语自己就可以说明问题。速度几乎是最重要的衡量标准,虽然其等级比健壮性、安全性和(在有些时候)兼容性的等级要低。然而它却是代码最直观的几个方面之一。Linux内核代码经过了彻底的优化,而最经常使用的部分——例如调度,则是优化工作的重点。几乎在任何时候都有一些不可思议的代码,这是由于这种方式的执行速度比较快(这并不总是很明显,但是你经常不得不通过自己的试验来对这种优化代码进行确认)。虽然有时一些更直接的实现方法速度也很快,但是我所见过的这种情况很少。

在某些情况下,本书推荐用可读性更好的代码来替代那些打着速度的名义而被故意扭曲了的代码。虽然速度是一个设计目标,但我基本上只在以下两种情况时才会这样做:a)在所考虑的问题中,速度明显不是关键问题 b)没有其他的办法。

3.2 内核体系结构初识

图3-1是一种类Unix操作系统的相当标准的视图,实际上,更细致地说,该图能够说明所有期望具有平台无关特性的操作系统。它着重强调了内核的两个特性:

- 内核将应用程序和硬件分离开来。
- 部分内核是体系结构和硬件特有的,而部分内核则是可移植的。

第一点我们在前面章节中已经讨论清楚了,在这里没有必要重复说明。第二点,也就是与体系结构无关和与体系结构相关代码的内容对于我们的讨论比较有意义。内核通过使用与处理用户应用程序相同的技巧来实现部分可移植性。这也就是说,如同内核把用户应用程序和硬件分离一样,部分内核将会因为与硬件的联系而同其他内核分离开来。通过这种分离,用户应用程序和部分内核都成为可移植的。

虽然这通常并不能够使得内核本身更清楚,但是源程序代码的体系结构无关部分通常定义了与低层,也就是体系结构相关部分(或假定)的接口。作为一个简单的例子,内存管理代码中的体系结构无关部分假定只要包含特定的头文件就可以获得合适的PAGE_SIZE宏(参见10791行)的定义,该宏定义了系统的内存管理硬件用于分割系统地址空间的内存块的大小(参见第8章)。体系结构无关代码并不关心宏的确切定义,而把这些问题都留给体系结构相关代码去处理(顺便一提,这比到处使用#ifdef/endif程序块来定义平台相关代码要清晰易懂得多)。

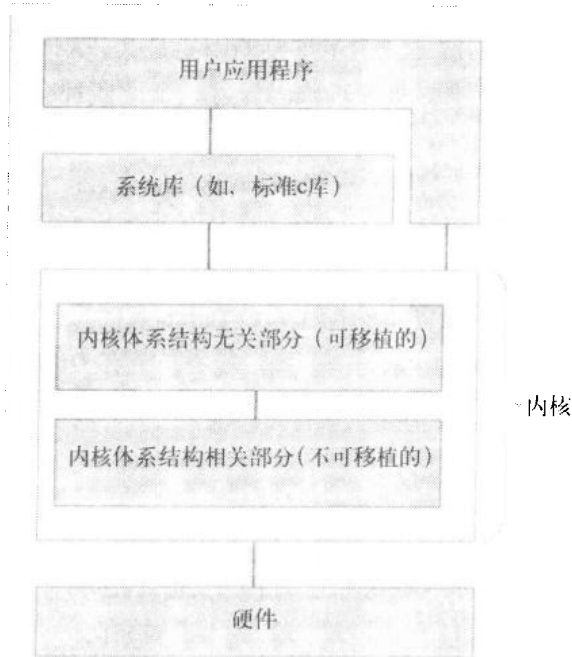


图3-1 内核体系结构基本结构图

这样，内核向新的体系结构的移植就转变成为确认这些特性及在新内核上实现它们的问题。

另外，用户应用程序的可移植性还可以通过它和内核的中间层次——标准C库（libc）——的协助来实现。应用程序实际上从不和内核直接通讯，而只通过libc来实现。图3-1中显示应用程序和内核直接通讯的唯一原因在于它们能够和内核通讯。虽然在实际上应用程序并不同内核直接通讯——这样做是毫无意义的。通过直接和内核通讯所能处理的问题都可以通过使用libc实现，而且更容易。

libc和内核通讯的方式是体系结构相关的（这和图中有一点矛盾），libc负责将用户代码从实现细节中解放出来。有趣的是，甚至大部分libc都不了解这些细节。大部分的libc，例如atoi和rand的实现，都根本不需要和内核进行通讯。剩余的大部分libc，例如printf函数，在涉及到内核之前或之后就已经处理了大量的工作（printf必需首先解释格式化字符串，分析相应参数，设定打印方法，在临时内部缓冲器中记录预期输出。直到此时它才调用底层系统，调用write来实际打印该缓冲区）。其他部分的libc则只是相应系统调用的简单代理。因而一旦发生函数调用时，它们会立即调用内核相应函数以完成主要工作。在最低层次上，大部分libc通过单通道同内核进行交流，而它们所使用的机制将在第5章进行详细介绍。

由于这种设计，所有的用户应用程序，甚至大部分的C库，都是通过体系结构无关的方式和内核通讯的。

3.3 内核体系结构的深入了解

图3-2显示了内核概念化的一种可能方式。该图和区分内核的体系结构无关及体系结构相关的方法有所不同，它是一种更具有普遍性的结构视图。在“Kernel”框内的本书中有所涉

及的内核部分都用括号注明了相应的章节编号。虽然有关对称多处理（SMP）的支持也属于本书的范围，但是在这里我们却没有标明章号。部分原因在于相当多的SMP代码广泛地分布于整个内核中，因此很难将它与某一个模块联系起来。同样，对于内核初始化的支持也属于本书的范围，但是也没有标明章号。这样做是因为从设计的观点上看，该问题并不重要。最后，虽然在图中我们将第6章和“进程间通讯”框联系在一起，但是该章只涉及一部分进程间通讯的内容。

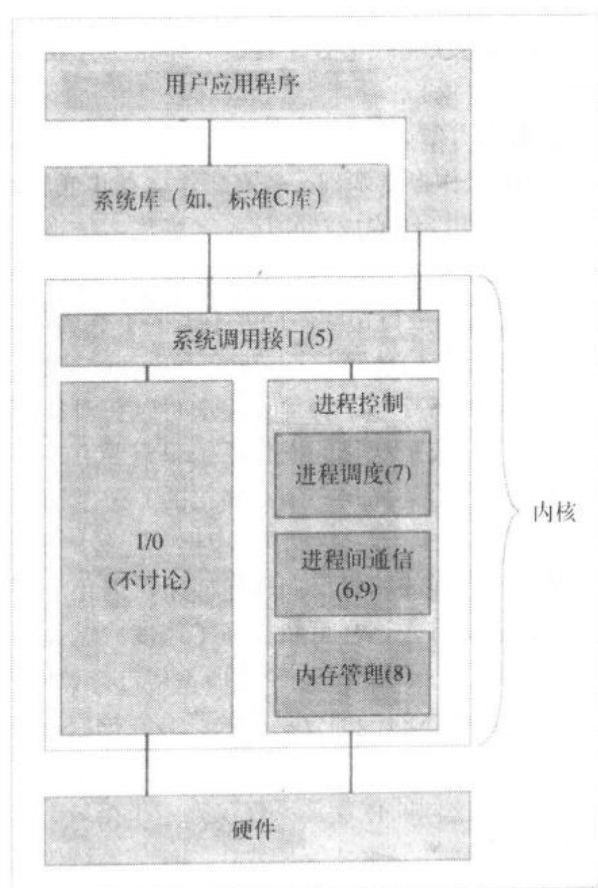


图3-2 详细的内核体系结构图

进程和内核的交互通常需要通过如下步骤：

- 1) 用户应用程序调用系统调用，通常是使用libc。
- 2) 该调用被内核的system_call函数截获（第5章，171行），此后该函数会将调用请求转发给另外的执行请求的内核函数。
- 3) 该函数随即和相关内部代码模块建立通讯，而这些模块还可能需要和其他的代码模块或者底层硬件通讯。
- 4) 结果按照同样的路径依次返回。

然而，并不是所有内核和进程间的交互都是由进程发起的。内核有时也会自行决定同哪个进程交互，例如通过释放信号或者简单的采用直接杀死进程的方法终止该进程的执行（如当进程用完所有可用的CPU时间片），以便使其他进程有机会运行。这些交互过程在该图中并没有表示，主要是因为它们通常都只是内核对内部的内部数据结构的修改（信号传递对于这

种规则来说是一个例外)。

3.4 Linux 内核的类型

解决复杂性的所有方法都基于一个基本原理：问题分解和各个击破。也就是说，都是把大型的、难以解决的问题（或系统）分解成一定数量的复杂度较低的子问题（或子系统），再根据需要重复这一过程直到每一部分都小到可以解决为止，而各种方法只是这种原理的一些不同运用而已。

计算机科学中有三种经典的方法比较适合于构建大型系统（我首先必须说明的是，这些定义都是经过我深思熟虑的讨论对象）。

1) 层次 (Layer) —— 将解决方案分解成若干部分，在这些部分中存在一个问题域的最底层，它为上层的抽象层次较高的工作提供基础。较高层建立在其低层基础之上。OSI和TCP/IP协议堆栈是众所周知的层次化软件设计的成功的例子。操作系统设计的层次化解决方案可能会包含一个可以直接和硬件通讯的层次，然后在其上提供为更高层提供抽象支持的层次。这样更高层就可以对磁盘、网卡等硬件进行访问，而并不需要了解这些设备的具体细节。

层次化设计的一个特征是要逐步构建符号集 (vocabulary)。随着层次的升高，符号集的功能将越来越强大。层次化设计的另外一个特征是完全可以在对其上下层透明的条件下替换某一层。在最理想的情况下，移植层次化的操作系统只需要重写最底层的代码。纯层次化模型实现的执行速度可能会很慢，因为高层必须（间接的）通过调用一系列连续的低层才能处理完自己的任务——N层调用N-1层，N-1层调用N-2层，等等，直到实际的工作在0层被处理完成。接着，结果当然是通过同样的路径反向传递回来。因此，层次化设计通常会包含对某些高层直接和某些低层通讯的支持；这样虽然提高了速度，但是却使得各个层次的替换工作更加困难（因为不止一个高层会直接依赖于这个你所希望进行替换的层次）。

- 模块 (Module) —— 模块将具体的一部分功能块隐藏在抽象的接口背后。模块的最大特点是将接口和其实现分离开来，这样就能够保证一个模块可以在不影响其他模块的情况下进行改变。这样也将模块之间的依赖关系仅仅限定于接口。模块的范围是试图反映求解域内一些方面的自然的概念性界限。纯模块化的操作系统因而就可能有一个磁盘子系统模块，一个内存管理子系统模块，等等。纯模块化和纯层次化的操作系统之间的主要区别是，一个可以由其他模块自由调用，模块间没有上层和下层的概念（从这个意义上来说，模块是广义的层次。按照纯粹的观点，层次是最多可供一个其它模块调用的模块，这个模块也就是它的直接上层模块）。
- 对象 (Object) —— 对象和模块不同，因为对于初学者来说它们具有不同的问题考虑方式，实现的方法也可能各自独立。但是，就我们当前的目的来说，对象不过是结构化使用模块的方法。组件 (component) 作为对象思想的进一步改进，目前还没有在操作系统设计中广泛使用。即便如此（按照我们的观点），我们也没有足够的理由将其和模块划分在不同的范畴中。

图3-1强调了内核的层次化的视图，而且是体系结构无关层次位于体系结构相关层次之上（更为精确的视图是在顶层增加一个附加的体系结构相关的层次。这是因为系统调用接口位于应用程序和内核之间，而且是体系结构相关的）。图3-2着重强调了更加模块化的内核视图。

从合理的表述层次上看，这两种观点都是正确的。但也可以说这两种观点都是错误的。

我可以用大量的图片向你证明内核是遵从所有你所能够指出的设计原则集合的，因为它就是从众多思想中抽取出来的。简单说来，事实是Linux内核既不是严格层次化的，也不是严格模块化的，也不是严格意义上的任何类型，而是以实用为主要依据的（实际上，如果要用一个词来概括Linux从设计到实现的所有特点，那么实用就是最确切的）。也许最保守的观点是内核的实现是模块化的，虽然这些模块有时会为了追求速度而有意跨越模块的界限。

这样，Linux的设计同时兼顾了理论和实际。Linux并没有忽视设计方法；相反，在Linux的开发基本思想中，设计方法的作用就像是编译器：它是完成工作的有力工具。选择一个基本的设计原则（例如对象）并完全使用这种原则，不允许有任何例外，这对于测试该原则的限制，或者构建以说明这些方法为目的的教学系统来说都是一个不错的方法。但是如果要用它来达到Linux的设计目标则会引起许多问题。而且Linux的设计目标中也并不包括要使内核成为一个完全纯化的系统。Linux开发者为了达到设计目标宁愿违背妨碍目标实现的原则。

实际上，如果对于Linux来说是正确的，那么它们对于所有最成功的设计来说都是正确的。最成功、应用最广泛的实际系统必然是实用的系统。有些开发人员试图寻找功能强大的可以解决所有问题的特殊方法。他们一旦找到了这种方法，所有的问题就都迎刃而解了。像Linux内核一样的成功设计通常需要为系统的不同部分和描述上的不同层次使用不同的方法。这样做的结果可能不是很清晰，也不是很纯粹，但是这种混合产物比同等功能的纯粹系统要强大而且优秀得多。

Linux大部分都是单内核的

操作系统内核可能是微内核，也可能是单内核（后者有时称之为宏内核Macrokernel）。按照类似封装的形式，这些术语定义如下：

- 微内核（microkernel）——在微内核中，大部分内核都作为独立的进程在特权状态下运行，它们通过消息传递进行通讯。在典型情况下，每个概念模块都有一个进程。因此，如果在设计中有一个系统调用模块，那么就必然有一个相应的进程来接收系统调用，并和能够执行系统调用的其他进程（或模块）通讯以完成所需任务。

在这些设计中，微内核部分经常只不过是一个消息转发站：当系统调用模块要给文件系统模块发送消息时，消息直接通过内核转发。这种方式有助于实现模块间的隔离（某些时候，模块也可以直接给其他模块传递消息）。在一些微内核的设计中，更多的功能，如I/O等，也都被封装在内核中了。但是最根本的思想还是要保持微内核尽量小，这样只需要把微内核本身进行移植就可以完成将整个内核移植到新的平台上。其他模块都只依赖于微内核或其他模块，并不直接依赖硬件。

微内核设计的一个优点是在不影响系统其他部分的情况下，用更高效的实现代替现有文件系统模块将会更加容易。我们甚至可以在系统运行时将开发出的新系统模块或者需要替换现有模块的模块直接而迅速地加入系统。另外一个优点是不需要的模块将不会被加载到内存中，因此，微内核就可以更有效地利用内存。

- 单内核（monolithic kernel）——单内核是一个很大的进程。它的内部又可以被分为若干模块（或者是层次或其他）。但是在运行的时候，它是一个独立的二进制大映像。其模块间的通讯是通过直接调用其他模块中的函数实现的，而不是消息传递。

单内核的支持者声称微内核的消息传递开销引起了效率的损失。微内核的支持者则认为

因此而增加的内核设计的灵活性和可维护性可以弥补任何损失。

我并不想讨论这些问题，但必须说明非常有趣的一点是，这种争论经常会令人想到前几年CPU领域中RISC和CISC的斗争。现代成功的CPU设计中包含了所有这两种技术，就像Linux内核是微内核和单内核的混合产物一样。Linux内核基本上是单一的，但是它并不是一个纯粹的集成内核。前面一章所介绍的内核模块系统将微内核的许多优点引入到Linux的单内核设计中。（顺便提一下，我考虑过一种有趣的情况，就是Linux的内核模块系统可以将系统内核转化成为简单的不传递消息的微内核设计。虽然我并不赞成，但是它仍然是一个有趣的想法。）

为什么Linux必然是单内核的呢？一个方面是历史的原因：在Linus的观点看来，通过把内核以单一的方式进行组织并在最初的空间中运行是相当容易的事情。这种决策避免了有关消息传递体系结构、计算模块装载方式等相关工作。（内核模块系统在随后的几年中又进行了不断地改进。）

另外一个原因是充足的开发时间的结果。Linux既没有开发时间的限制，也没有来自于市场压力的发行进度。所有的限制只有并不过分的对内核的修改与扩充。内核的单一设计在内部实现了充分的模块化，在这种条件下的修改或增加都并不怎么困难。而且问题还在于没有必要为了追求尚未证实的可维护性的微小增长而重写Linux的内核（Linus曾多次特别强调了如下的观点：为了这点利益而损耗速度是不值得的）。后面章节中将详细讨论充足开发时间的效果。

如果Linux是纯微内核设计，那么向其他体系结构上的移植将会比较容易。实际上，有一些微内核，如Mach微内核，就已经成功地证明了这种可移植性的优点。实际的情况是，Linux内核的移植虽然不是很简单，但也绝不是不可能的：大约的数字是，向一个全新的体系结构上的典型的移植工作需要30 000到60 000行代码，再加上不到20 000行的驱动程序代码（并不是所有的移植都需要新的驱动程序代码）。粗略计算一下，一个典型的移植大约平均需要50 000行代码。这对于一个程序员或者最多一个程序小组来说是力所能及的，可以在一年之内完成。虽然这比微内核的移植需要更多的代码，但是Linux的支持者将会提出，这样的Linux内核移植版本比微内核更能够有效地利用底层硬件，因而移植过程中的额外工作是能够从系统性能的提高上得到补偿的。

这种特殊设计的权衡也不是很轻松就可以达到的，单内核的实现策略公然违背了传统的看法，后者认为微内核是未来发展的趋势。但是由于单一模式（大部分情况下）在Linux中运行状态良好，而且内核移植相对来说比较困难，但没有明显地阻碍程序员团体的工作，他们已经成功地把内核移植到了现存的大部分实际系统中，更不用说掌上型电脑了。只要Linux的众多特点仍然值得移植，新的移植版本就会不断涌现。

3.5 设计和实现的关系

接下来的部分将介绍一些内核设计和实现之间的关系。本部分最重要的内容是对内核源程序目录结构的概述，这一点随后就会提到。本章最后以实现中体系结构无关代码和体系结构相关代码的相对大小的估算作为总结。

3.5.1 内核源程序目录结构

按照惯例，内核源程序代码安装在/usr/src/linux目录下。在该目录下还有几个其他目录，

每一个都代表一个特定的内核功能性子集（或者非常粗略地说是高层代码模块）。

1. Documentation

这个目录下面没有内核代码，只有一套有用的文档。但是这些文档的质量不一。有一部分内核文档，例如文件系统，在该目录下有相当优秀而且相当完整的文档；而另外一部分内核，例如进程调度，则根本就没有文档。但是在这里你可以不时地发现自己所最需要的东西。

2. arch

arch目录下的所有子目录中都是体系结构相关的代码。每个体系结构特有的子目录下都又至少包含三个子目录：kernel，存放支持体系结构特有的诸如信号处理和SMP之类特征的实现；lib，存放高速的体系结构特有的诸如strlen和memcpy之类的通用函数的实现；mm，存放体系结构特有的内存管理程序的实现。

除了这三个子目录以外，大多数体系结构在必要的情况下还都有一个boot子目录，该目录中包含有在这种平台上启动内核所使用的部分或全部平台特有代码。这些启动代码中的部分或全部也可以在平台特有的内核目录下找到。

最后，大部分体系结构所特有的目录还可以根据需要包含了供附加特性或改进组织使用的其他子目录。例如，i386目录包含一个math-emu子目录，其中包括了在缺少数学协处理器（FPU）的CPU上运行模拟FPU的代码。作为另外一个例子，m68k移植版本中为每一个该移植版本所支持的基于680x0的机器建立了一个子目录，从而这些机器所特有的代码都有一个自然的根目录。

下面几个是arch目录下的子目录：

- arch/alpha/——Linux内核到基于DEC Alpha CPU工作站的移植。
- arch/arm/——Linux到ARM系列CPU的移植，该类CPU主要用于诸如Corel的NetWinder和Acorn RiscPC之类的机器。
- arch/i386/——最接近于Linux内核原始平台或标准平台。这是为Intel的80386结构使用的，当然包括对同一系列后来的CPU（80486、Pentium等等）的支持。它还包括了对AMD、Cyrix和IDT等公司的一些兼容产品的支持。

本书基本上将这种体系结构称为“x86”。即使这样，严格说来“x86”对于我们的目标来说还是要求得过于宽泛。早期的Intel CPU，例如80286，并没有包括Linux运行所需的所有特性。对于这些机器，Linux也没有正式的支持版本（顺便提一下，Linux对这种CPU的独立移植版本是存在的，不过它在功能上有部分损失）。当本书中提到“x86平台”时，通常是指80386或更新的CPU。

- arch/m68k/——到Motorola的680x0 CPU系列的移植。该版本可以提供对基于从68020（只要它同内存管理单元（MMU）68851一起使用）到68060的一切机器的支持。很多公司在他们的产品中使用680x0系列芯片，例如Commodore（现在是Gateway）的Amiga、Apple的Macintosh、Atari ST等等。这些老机器中的很多现在正充当可靠的Linux工作站。另外，到NeXT工作站和SUN 3工作站的移植也正在进行中。
- arch/mips/——到MIPS的CPU系列的移植。虽然有其他几个厂商也使用MIPS开发了一些系统，但是基于这种CPU的最出名的机器是Silicon Graphics（SGI）工作站。
- arch/ppc/——到Motorola/IBM的PowerPC系列CPU的移植。这包括对基于PowerPC的Macintosh和Amiga以及BeBox、IBM的RS/6000等其他一些机器的支持。

- arch/sparc/——到32位SPARC CPU的移植。这包括对从Sun SPARC 1到SPARC 20的全部支持。
- arch/sparc64/——到基于64位SPARC CPU (UltraSPARC系列)系统的移植。这里所能够支持的机器包括Sun的Ultra 1、Ultra 2和更高配置的机器，直到Sun的最新产品Enterprise 10000。注意32位和64位的SPARC的移植版本正在合并中。

不幸的是，本书必须将注意力集中在x86上，因此只应用到了arch/i386/目录下的代码，而其他体系结构所特有的代码将不再涉及了。

3. drivers

这个目录是内核中非常大的一块。实际上，drivers目录下包含的代码占整个内核发行版本代码的一半以上。它包括显卡、网卡、SCSI适配器、软盘驱动器、PCI设备和其他任何你可以说出的Linux支持的外围设备的软件驱动程序。

drivers目录下的一些子目录是平台特有的，例如，zorro子目录中包含有和Zorro总线通讯的代码。而Zorro总线只在Amiga中使用过，因此这些代码必然是Amiga特有的。而其他一些子目录，例如pci子目录，则至少是部分平台无关的。

4. fs

Linux支持的所有文件系统在fs目录下面都有一个对应的子目录。一个文件系统是存储设备和需要访问存储设备的进程之间的媒介。

文件系统可能是本地的物理上可访问的存储设备，例如硬盘或CD-ROM驱动器；在这两种情况下将分别使用ext2和isofs文件系统。文件系统也可能是可以通过网络访问的存储设备；这种情况下使用的文件系统是NFS。

还有一些伪文件系统，例如proc文件系统，可以以伪文件的形式提供其他信息（例如，在proc的情况下是提供内核的内部变量和数据结构）。虽然在底层并没有实际的存储设备与这些文件系统相对应，但是进程可以像有实际存储设备一样处理（NFS也可以作为伪文件系统来使用）。

5. include

这个目录包含了Linux源程序树中大部分的包含(.h)文件。这些文件按照下面的子目录进行分组：

- include/asm-*/——这样的子目录有多个，每一个都对应着一个arch的子目录，例如include/asm-alpha、include/asm-arm、include/asm-i386等等。每个目录下的文件中包含了支持给定体系结构所必须的预处理器宏和短小的内联函数。这些内联函数很多都是全部或部分地使用汇编语言实现的，而且在C或者汇编代码中都会应用到这些文件。

当编译内核时，系统将建立一个从include/asm到目标体系结构特有的目录的符号链接。结果是体系结构无关的内核源程序代码可以使用如下形式的代码来实现所需功能：

```
#include <asm/some-file>
```

这样就能够将适当地体系结构特有的文件包含(#include)进来。

- include/linux/——内核和用户应用程序请求特定内核服务时所使用的常量和数据结构在头文件中定义，而该目录中就包含了这些头文件。这些文件大都是平台独立的。这个目录被全部复制（更多的情况是链接）到usr/include/linux下。这样用户应用程序就可以使用#include包含这些头文件，而且能够保证所包含进来的头文件的内容和内核中的定

义一致。第9章将会给出有关的一个样例。

- 对这些文件的移植只有对于内核来说才是必须的，对用户应用程序则没有必要。移植工作可以按照如下的方式封装处理：

```
/* ... Stuff for user apps and kernel ... */
#ifdef __KERNEL__

/* ... Stuff for kernel only ... */

#endif /* __KERNEL__ */
```

- include/net/——这个目录供与网络子系统有关的头文件使用。
- include/scsi/——这个目录供与SCSI控制器和SCSI设备有关的头文件使用。
- include/video/——这个目录供与显卡和帧显示缓存有关的头文件使用。

6. init

这个目录下面的两个文件中比较重要的一个是main.c，它包含了大部分协调内核初始化的代码。第4章将详细介绍这部分代码。

7. ipc

这个目录下的文件实现了System V的进程间通讯（IPC）。在第9章中将会对它们进行详细介绍。

8. kernel

这个目录中包含了Linux中最重要的部分：实现平台独立的基本功能。这部分内容包括进程调度（kernel/sched.c）及创建和撤销进程的代码（kernel/fork.c和kernel/exist.c）；以上所有内容将在第7章中有所涉及。但是我并不想给你留下这样的印象：需要了解的内容都在这个目录下。实际上在其他目录下也有很多重要的内容。但是，不管怎样说，最重要部分的代码是在这个目录下的。

9. lib

lib目录包含两部分的内容。lib/inflate.c中的函数能够在系统启动时展开经过压缩的内核（请参见第4章）。lib目录下剩余的其他文件实现一个标准C库的有用子集。这些实现的焦点集中在字符串和内存操作的函数（strlen、memcpy和其他类似的函数）及有关sprintf和atoi的系列函数上。

这些文件都是使用C语言编写的，因此在新的内核移植版本中可以立即使用这些文件。正如本章前面部分说明的那样，一些移植提供了它们独有的高速的函数版本，这些函数通常是经过手工调整过的汇编程序，在移植后的系统使用这些函数来代替原来的通用函数。

10. mm

该目录包含了体系结构无关的内存管理代码。正如我们前面说明的那样，为每个平台实现最底层的原语的体系结构特有的内存管理程序是存储在arch/platform/mm中的。大部分平台独立和x86特有的内存管理代码将在第8章中介绍。

11. net

这个目录包含了Linux应用的网络协议代码，例如AppleTalk、TCP/IP、IPX等等。

12. scripts

该目录下没有内核代码，它包含了用来配置内核的脚本。当运行make menuconfig或者

make xconfig之类的命令配置内核时，用户就是和位于这个目录下的脚本进行交互的。

3.5.2 体系结构相关和体系结构无关的代码

现在我们来估计一下体系结构相关和体系结构无关代码的相对大小。我们首先给出一些数字。完整的2.2.5的内核总共有1 725 645行代码（顺便一提，请注意本书只包含了39 000行代码，但是我们仍然努力涵盖了相当部分的核心函数）。其中一共有392 844行代码在体系结构特有的目录之内，也就是arch/*和include/asm-*下面。我估计还有超过64 000行的代码是仅供一种体系结构专用的驱动程序。这意味着大约26%的代码是专用于体系结构相关代码的。

但是，对于单一一种体系结构，体系结构相关代码比例相对较小。不妨理想一点，如果某种单一体系结构所需要的特有代码约有50 000行，而体系结构无关代码则大约有1 250 000行，那么体系结构相关代码大概只占到4%。当然，在特定的一个内核中，并不是所有这些体系结构无关代码都会被用到，因此体系结构相关代码在特定内核中所占的比重与内核的配置有关。但是不管怎样，很显然大部分内核代码是平台独立的。

第4章 系统初始化

当你想要运行程序时，你需要把程序的文件名敲入shell——或者更为流行的，在如GNOME或者KDE等之类桌面环境中点击相应的图标，这样就能将其装载进内核并运行。但是，首先必须有其他的软件来装载并运行内核；这通常是诸如LOADLIN或者LILO之类的内核引导程序。更进一步说，我们还需要其他的软件来装载运行内核引导程序——称之为“内核引导程序的引导程序”，而且看起来似乎运行内核引导程序的引导程序也需要内核引导程序的引导程序的引导程序，等等，这个过程是无限的。

这个无限循环的过程必然最终在某个地方终止，这就是硬件。因此，在最低的层次上，启动系统的第一步是从硬件中获得帮助。该硬件总是运行一些短小的内置程序——软件，但是这些软件是被固化在只读存储器中，存储在已知地址中。因此，在这种情况下就不需要软件引导程序了——它能够运行更大更复杂的程序，直到内核自身装载成功为止。按照这种方式，系统自己的引导过程（bootstrap）会引发系统的启动，当然这只是术语“系统引导（booting）”的一个比喻。虽然不同体系结构的引导过程的具体细节差异很大，但是它们的原则都基本相同。

前面的工作都完成以后，内核就已经成功装载了。随后内核可以初始化自身以及系统的其他部分。

本章首先将简单介绍基于x86 PC机的典型自启动方式，接着回顾一下每一步工作在什么时机发生，最后我们还要介绍的是内核的相应部分。

4.1 引导PC机

本节简要介绍x86 PC是如何引导的。本节的目的不是让你精通PC是怎样引导的（这超出了本书的范围），而是向你展示特定体系结构一般的引导方式，为下文中的内核初始化进行铺垫。

首先，机器中的每个CPU都要自行初始化，接着可能要用几分之一秒的时间来执行自测试。在多处理器的系统中，这个过程会更复杂些——但是实际上也并不多。在双处理器的Pentium系统中，一个CPU总是作为主CPU存在，另外一个CPU则是辅CPU。主CPU执行启动过程中的剩余工作，随后内核才会激活辅CPU。在多处理器的Pentium Pro系统中，CPU必须根据Intel定义的算法“抢夺标志”来动态决定由哪个CPU启动系统。取得标志的CPU启动系统，随后内核激活其他的CPU。无论是哪种情况，启动程序的剩余部分只与一个CPU有关。这样，在随后的一段时间内，我们可以认为该系统中只有一个CPU是可用的，而不考虑其他的CPU，或者说这些CPU被暂时隐藏了。另一方面，内核还需要明确地激活所有其他的CPU——这一点你可以在本章后续部分看到。

接下来，CPU从0xfffff0地址单元中取得指令并执行，这个地址非常接近于32位CPU的最后可用的地址。因为大多数PC都没有4GB的RAM，所以通常在这个地址上并没有实际内存。内存硬件可以虚拟使用它。对那些确实有4GB内存的机器来说，它们也只是仅仅损失了供BIOS使用的顶端地址空间末尾的少量内存（实际上BIOS在这里只保留了64K的空间——这种损失在4GB的机器中是可以忽略的）。

该地址单元中存储的指令是一条跳转指令，这条指令跳转到基本输入输出系统（BIOS）代码的首部。BIOS内置在主板中，它主要负责控制系统的启动。请注意CPU实际上并不真正关心BIOS是否存在，这样就使得在诸如用户定制的嵌入系统之类的非PC体系结构的计算机中使用Intel的CPU成为可能。CPU执行在目标地址中发现的任何指令，在这里使用跳转指令转移到BIOS只是PC体系结构的一部分（实际上，跳转指令自己是BIOS的一部分，但是这不是考虑这个问题的最方便的方法）。

BIOS使用内置的规则来选择启动设备。通常情况下，这些规则是可以改变的，方法是在启动过程开始时按下一个键（例如，在我的系统中是Del键）并通过一些菜单选项浏览选择。但是，通常的过程是BIOS首先试图从软盘启动，如果失败了，就再试图从主硬盘上启动。如果又失败了，就再试图从CD-ROM上启动。为了使问题更具体，这里讨论的情况假定是最普通的，也就是启动设备是硬盘。

从这种启动设备上启动，BIOS读取第一个扇区的信息——首512个字节，称之为主引导记录（MBR）。接下来发生的内容有赖于Linux是怎样在系统上安装的。为使讨论形象具体，我们假定LILO是内核的载入程序。在典型的设置中，BIOS检测MBR中的关键数字（为了确认该数据段的确是MBR），并在MBR中检测引导扇区的位置。这一扇区包含了LILO的开始部分，然后BIOS将其装入内存，开始执行。

注意我们现在已经实现了从硬件和内置软件的范围到实际软件范围的转变，从有形范围到无形范围，也就是说从你可以接触的部分到不可接触的部分。

下面就是LILO的责任了。它把自己其余的部分装载进来，在磁盘上找到配置数据，这些数据指明从什么地方可以得到内核，启动时要通过什么选项。LILO接着装载内核到内存并跳转到内核。

通常，内核以压缩形式存储，只有少量足以完成解压缩任务的指令，也就是自解压可执行文件，是以非压缩形式存储的。因此，内核的下一步工作是自解压缩内核镜像。到这里，内核就已经完成了装载的过程。

下面是对所进行步骤的简要描述：

- 1) CPU初始化自身，接着在固定位置执行一条指令。
- 2) 这条指令跳转到BIOS中。
- 3) BIOS找到启动设备并获取MBR，该MBR指向LILO。
- 4) BIOS装载并把控制权转交给LILO。
- 5) LILO装载压缩内核。
- 6) 压缩内核自解压，并把控制权转交给解压的内核。

正如你所见到的，引导过程每一步都将你带入更大量更复杂的代码块中，一直到最后成功地运行了内核为止。

依赖于你计算层次的方式，CPU成为内核引导程序的引导程序的引导程序的引导程序（CPU装载BIOS，BIOS装载LILO，LILO装载压缩内核，压缩内核装载解压内核；但是你可以考虑是否这些步骤都满足引导程序的定义）。

4.2 初始化Linux内核

在内核成功装入内存（如果需要就解压缩）以及一些关键硬件，例如已经在低层设置过

的内存管理器（MMU，请参见第8章）之后，内核将跳转到start_kernel（19802行）。这个函数完成其余的系统初始化工作——实际上，几乎所有的初始化工作都是由这个函数实现的。因此，start_kernel就是本节的核心。

start_kernel

19802: __init标示符在gcc编译器中指定将该函数置于内核的特定区域。在内核完成自身初始化之后，就试图释放这个特定区域。实际上，内核中存在两个这样的区域，.text.init和.data.init——第一个是代码初始化使用的，另外一个则是数据初始化使用的（可以在进程间共享的代码和字符串常量之类的“文本（Text）”是在可执行程序中的“纯区域”中使用的一个术语）。另外你也可以看到__initfunc和__initdata标志，前者和__init类似，标志初始化专用代码，后者则标志初始化专用数据。

19807: 如前所述，即使在多处理器系统中，在启动时也只使用一个CPU。Intel称之为引导程序处理器（bootstrap processor，简称为BSP），它在内核代码的某些地方有时也称之为BP。BSP首次运行这一行时，跳过后面的if语句，并减小boot_cpu标志，从而当其他CPU运行到此处时，都要运行if语句。等到其他CPU被激活执行到这里时，BSP已经在idle循环中了（本章稍后会更详细地讨论这个问题），initialize_secondary（4355行）负责把其他CPU加入到BSP中。这样，其他CPU就不用执行start_kernel的剩余部分了——这也是一件好事，因为这意味着不用再对许多硬件进行冗余初始化等工作了。

顺便说一下，这种奇异的小小的改动只有对于x86是必需的；对于其他平台，调用smp_init完全可以处理SMP设置的其他部分。因此，其他平台的initialize_secondary的定义都是空的。

19816: 打印内核标题信息（20099行），这里显示了有关内核如何编译的信息，包括在什么机器上编译，什么时间编译，使用什么版本的编译器，等等。如果中间任何一步发生了错误，在寻找机器不能启动的原因时查明内核的来源是一个有用的线索。

19817: 初始化内核自身的部分组件——内存、硬件中断、调度程序，等等。尤其是setup_arch函数（19765行）完成体系结构相关的设置，此后在command_line（传递到内核的参数，在下面讨论）、memory_start和memory_end（内核可用物理地址范围）中返回结果。下面这些函数都希望驻留在内存低端，它们使用memory_start和memory_end来传递该信息。在函数获得所希望的值后，返回值指明了新的memory_start的值。

19823: 分析传给内核的各种选项。parse_options函数（19707行，在随后的“分析内核选项”一节中讨论）也设置了argv和envp的初值。

19833: 内核运行过程中也可以自行对所进行的工作进行记录，周期性地对所执行的指令进行抽样，并使用所获得的结果更新表格。这在定时器中断过程中通过调用x86_do_profile（1896行）来实现，该部分将在第6章中介绍。

如图4-1中说明的那样，这个表格把内核划分为几个大小相同的范围，并简单跟踪在一次中断的时间内每个范围中运行多少条指令。这种记录当然是非常粗

糙的——甚至不是依据函数和行号进行划分的，而只是使用近似的地址——但是这样代价很低，且快速、短小，而且有助于专家判断最关键的问题。每个表格条目所涉及到地址的多少——还有问题发生地点的不确定性——可以通过简单修改 `prof_shift` (26142行) 来调节。`profile_setup` (19076行，在本章中后面讨论) 可以让你在启动的时候设置 `prof_shift` 的值，这样比为修改这个数字而重新编译内核要清晰方便得多。

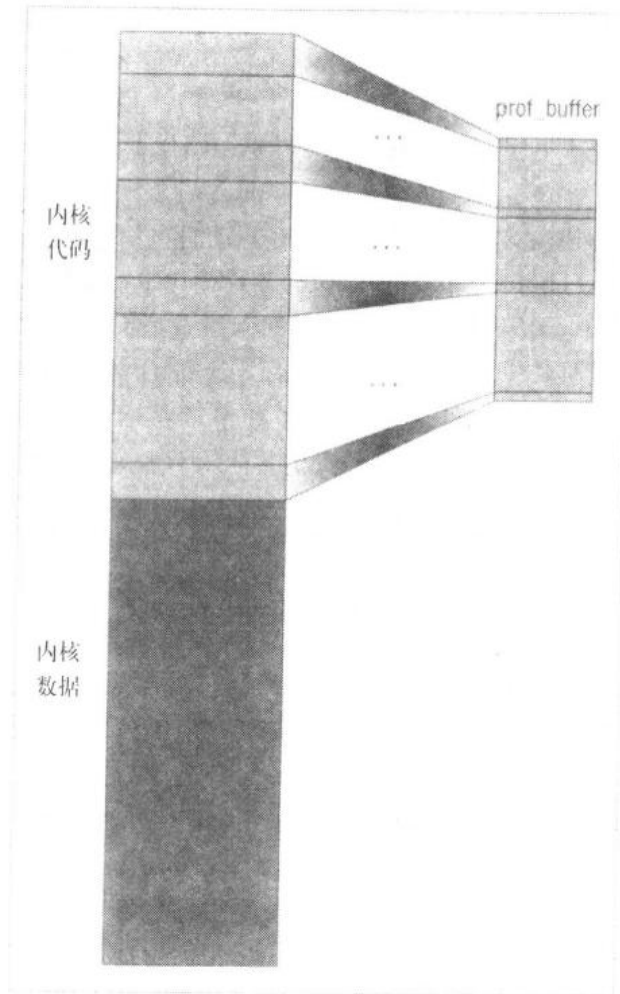


图4-1 描述用缓存

这个 `if` 程序块为记录表格分配内存，并把所有项都清零。注意到如果 `prof_shift` 是 0 (默认值)，那么记录功能就被关掉了，`if` 程序块不再被执行，也不为表格分配空间。

- 19846: 内核通过调用 `sti` (UP版本的13104行，注意该主题在第6章中有更详细的介绍) 开始接收硬件中断。首先需要激活定时器中断，以便后来对 `calibrate_delay` (19654行) 的调用可以计算机器的BogoMIPS的值 (在下一节“BogoMIPS”中介绍)。因为一些设备驱动程序需要BogoMIPS的值，所以内核必需在大部分硬件、文件系统等等初始化之前计算出这个值来。
- 19876: 测试该CPU的各种缺陷，比如Pentium F00F缺陷 (请参见第8章)，记录检测到的

缺陷，以便于内核的其他部分以后可以使用它们工作。（为了节省空间起见，我们省略掉了check_bugs函数。）

- 19882: 调用smp_init (19787行)，它又调用了其他的函数来激活SMP系统中的其他CPU: 在x86的平台上，smp_boot_cpus (4614行) 初始化一些内核数据结构，这些数据结构跟踪检测另外的CPU并简单的将其改为保持模式；最后smp_commence (4195行) 使这些CPU继续执行。
- 19883: 把init函数作为内核线程终止，这比较复杂；请参阅本章后面有关init的讨论。
- 19885: 增加idle进程的need_resched标志，这样做的原因此时可能还比较模糊。当读完了第5、6、7章以后，就会有清楚的概念；但是，在下一个定时器中断结束之前（在第6章中讨论），system_call (171行、在第5章中讨论) 函数中会注意到idle进程的need_resched标志增加了，并且调用schedule (26686行，第7章) 释放CPU，并将其赋给更应该获取CPU的进程。
- 19886: 已经完成了内核初始化的工作——或者不管怎样，已经把需要完成的少量责任传递给了init，所剩余的工作不过是进入idle循环以消耗空闲的CPU时间片。因此，本行调用cpu_idle (2014行)——idle循环。正如你可以从cpu_idle本身可以发现的一样，该函数从不返回。然而，当有实际工作要处理时，该函数就会被抢占。

注意到cpu_idle只是反复调用idle系统调用（下一章将讨论系统调用），它通过sys_idle (2064行) 实现真正的idle循环——2014行对应UP版本，2044行针对SMP版本。它们通过执行hlt（对应“halt”）指令把CPU转入低功耗的“睡眠”状态。只要没有实际的工作处理，CPU都将转入这种状态。

4.2.1 BogomIPS

BogomIPS的数字由内核计算并在系统初始化的时候打印。它近似给出了每秒钟CPU可以执行一个短延迟循环的次数。在内核中，这个结果主要用于需要等待非常短周期的设备驱动程序——例如，等待几微秒并查看设备的某些信息是否已经可用。

由于没有正确理解BogomIPS的含义，BogomIPS在各处都被滥用，就仿佛它可以满足人类最原始、最深层次的需求：把所有计算机性能的信息简化为一个数字。“BogomIPS”中的“Bogo”部分来源于“伪 (bogus)”，就正是为了防止这种用法：虽然这个数字比大多数基准测试数大，但是它仍然是不准确的、容易引起误解的、无用的和不真实的，根本不适合将它用于机器间差别的对比。但是这个数字仍然非常吸引人，这也正是我们在这里讨论这个问题的原因。（BogomIPS中“MIPS”部分是“millions of instructions per second (百万条指令每秒)”的缩写，这是CPU基准测试中的一个常用单位。）

calibrate_delay

- 19654: calibrate_delay是近似计算BogomIPS数字的内核函数。
- 19622: 作为第一次估算，calibrate_delay计算出在每一秒内执行多少次__delay循环 (6866行)，也就是每个定时器滴答 (timer tick)——百分之一秒——内延时循环可以执行多少次。
- 19664: 计算一个定时器滴答内可以执行多少次循环需要在滴答开始时就开始计数，或者应该尽可能与它接近。全局变量jiffies (16588行) 中存储了从内核开始保持跟踪

时间开始到现在已经经过的定时器滴答数；第6章中将介绍它的实现方式。jiffies保持异步更新，在一个中断内——每秒一百次，内核暂时挂起正在处理的内容，更新变量，然后继续刚才的工作。如果不这样处理，下一行的循环就永远不可能退出。从而，如果jiffies不声明为volatile——简单地说，这个值变化的原因对于编译器是透明的，gcc仍然可能对该循环进行优化，并引起该循环进入不能退出的状态。虽然目前的gcc还没有如此高的智能，然而它的维护者应该完全能够为它实现这种智能。

19669: 定时器又前移了一个滴答，因此又产生一个新的滴答。下一步是要等待loops_per_sec延时循环调用定时器循环，接着检测是否最少有一个完整的滴答已经完成。如果是这样，就退出首次近似估算循环；如果没有，就把loops_per_sec的值加倍，然后重新启动这个过程。

这个循环的正确性依赖于如下的事实：现有的机器在任何地方都不能每秒执行 2^{32} 次延时循环（对于64位机来说则远低于每秒 2^{64} 次），虽然这只是一个微不足道的问题。

19677: 现在内核已经清楚loops_per_sec循环调用延时循环在这台机器上要花费超过百分之一秒的时间才能完成，因此，内核将重新开始进行估算。为了提高效率，内核使用折半查找算法计算loops_per_sec的实际值，我们假定开始的时候，实际值在现在计算结果和其一半之间——实际值不可能比现在计算值还大，但是可以（而且可能）稍微小一点。

19681: 和前面使用的方式一样，calibrate_delay查看这个loops_per_sec已经减小了的值是否还是比较大，而需要耗费一个完整的定时器间隔。如果还是相当大，实际值应该小于当前计算值或者就是当前值，因此，使用更小的值继续查询；如果不够大，就使用一个更大的值继续查询。

19691: 内核有一种很好的方法来计算一个定时器滴答中执行延时循环的次数。这个数字乘以一秒内滴答的数量就得到了每秒内可以执行的延时循环的次数。这种计算只是一种估算，乘法也累积了误差，因此结果并不能精确到纳秒。但是这个数字供内核使用已经足够精确了。

19693: 为了让用户感到激动，内核打印出这个数字。注意这里明显省略了%f的格式限定——内核尽量避免浮点数运算。这个计算过程中最有用的常量是500 000；它是用一百万除以2得来，理由是每秒钟执行一百万条指令，而每个delay循环的核心是2条指令（decl和一条跳转指令）。

4.2.2 分析内核选项

parse_options函数分析由内核引导程序发送给内核的启动选项，在初始化过程中按照某些选项运行，并将剩余部分传送给init进程（在本章后面部分提到）。这些选项可能已经存储在配置文件中了，也可能是由用户在系统启动时敲入的——内核并不关心这些。类似的细节全部是内核引导程序应该关注的内容。

1. parse_options

19707: 参数已经收集在一条长的命令行中，内核被赋给指向该命令行头部的一个指针；

内核引导程序在前面已经将该行存储在一个指定地址中。

- 19718: 中断下一个参数，保持指向下一个参数的指针以供下一次循环使用。注意系统使用空格而不是通常的空白来分隔内核参数；制表符并不能把当前参数和下一个参数分隔开。如果发现了分隔字符空格，下一行就使用字节0覆盖，这样line可以作为包含有唯一一个内核选项的标准C字符串来使用了。如果没有发现空格，就该函数关心的内容而言，其余的部分都具有相同的属性——这只有在处理line中最后一个选项的情况下才会发生，循环就会在下次开始时结束。

注意该代码不会跳过多个空格。假设line值如下所述（两个空格）：

```
rw debug
```

这会被当作三个选项：“rw”、“ ”（空字符串）和“debug”。因为空字符串不是有效的内核选项，它将会被传递到初始化的过程（这一点随后就可以看到）——这当然不是用户所希望的。因此，内核引导程序应该负责对多个空格进行压缩。LILO通过忽略用户多敲的空格，完美地解决了这个问题。

- 19721: 现在开始解释这些选项。最前面的两个选项ro和rw指明内核要装载根文件系统，也就是根目录（/目录）所在的位置，而分别处于只读和读/写模式。
- 19729: 第三种可能性，debug，增加了调试信息的数量；这些调试信息要通过调用do_syslog打印出来（25724行）。
- 19733: 开始几个选项是简单的独立标志，它们并不使用参数。内核也可以辨认形为option=value的选项。本行就是一个例子，这里内核引导程序定义了一个命令来代替init运行；它使用init=/some/other/program的形式。这里的代码舍弃了init=部分，为随后init的使用而把剩余部分在execute_command中保存起来（20044行）。和其他大部分参数的处理方法不同，本处功能不能在checksetup（19612行）中实现，这是因为它改变了该函数的局部变量。很快，你就可以看到前面三个选项之所以也在这里处理，而不是在checksetup中处理的原因。
- 19745: 大部分内核选项都是由checksetup函数分析的。如果checksetup处理了某个选项，就返回真值，循环继续进行。
- 19750: 否则，line中没有已经被辨认的内核选项。在这种情况下，它被作为一个供init进程使用的选项或者环境变量来处理——如果其形式为envar=value，就作为环境变量处理；否则，就作为选项处理。如果argv_init和envp_init（分别见19057和19059行）数组中有足够的空间，选项和环境变量就存储在里面供以后init函数使用。

这解释了从19736行开始的注释。字符串auto并不是任何内核选项的前缀，因此它应该被作为init的一个参数存储在argv_init数组中——这在大多数情况下都是可行的，因为auto是init可以识别的选项。但是，当使用init=的形式给出内核选项时，通常是执行shell而不是init，auto会使shell混淆；因此，安全一点的方法是，parse_options在此处忽略所有与此有关的init参数。

奇怪的是，当argv_init或者envp_init数组空间用完时，整个循环就结束了。仅仅因为argv_init的空间用完了并不意味着line中就不再含有init使用的环境变量，反之亦然。此外，可能还剩下许多内核选项没有处理。当你考虑到MAX_INIT_ARGS

(19029行)和MAX_INIT_ENVS(19030行)都通过使用#define被预定义为8——这是一个很容易超过的下限,这种行为就更奇怪了。如果在19752行和19756行的break改成continue,那么循环可以继续处理内核选项,而不会写入超过argv_init和envp_init数组界限的空间。如果command_line中仍然包含有并不是为init而定义的内核选项,那么这一点就是非常重要的。

19760: 所有的内核选项都处理完成了。最后一步是要使用NULL填充argv_init和envp_init数组的末尾,从而使得init可以知道在哪里终止。

2. checksetup

19612: checksetup函数负责进行大部分内核选项的处理过程。它把这些内核选项分为三类:一类使用内核普通参数来分析=sign之后的部分;另一类自行分析=sign之后的部分;还有一类自行分析整个行,包括=sign前面的部分和=sign后面的部分。第一类被认为是使用“现成”的参数,这与为第二类提供的“原始”参数相对应。最后一类只由一个IDE驱动程序组成;内核首先在19619行检查并处理这种情况,以使其不会在随后的处理中造成麻烦。

19625: 接下来,checksetup扫描整个raw_params数组(19552行),并判断是否该内核选项应该不加处理地保留。raw_params中的元素是struct kernel_param类型(19223行)的,它把内核选项前缀和装载选项时调用的函数联系起来。如果数组中的某些项的str成员以line为前缀,就会调用line后面的相应函数(也就是前缀之后的部分),随后checksetup会返回一个非零值以表明它已经对该内核选项进行了处理。raw_params数组以两个NULL结束,因此在检测到str成员是NULL时,循环就可以结束了。在这种情况下,显然循环已经到达了raw_params数组的结尾,但是仍然没有找到匹配的情况。当然,测试setup_func成员也可以取得同样好的效果。

这个循环说明了一点:与大多数内核非常不同的是,这里的初始化并不需要尽可能地快。如果内核比从前多用几微秒来启动,这并没有什么实际的损失——毕竟用户应用程序还没有开始运行,所以他们并没有损失什么东西。

最终结果是代码效率很低,而且存在很多优化的可能。例如,raw_params数组中字符串的长度可以在raw_params中暂存,而不用在19626行多次重复计算。更好的解决方法是,可以把raw_params数组中的项按照字符顺序排序,这样checksetup就可以进行折半查找。

在raw_params的情况中实现排序并没有什么障碍,但是这样也可能并不能获得很大的优势,因为折半查找的优点只有在比较大的数组中才能充分表现出来(所谓比较大的确切值在不同的环境中也有所不同)。raw_params的姊妹数组cooked_params(19228行)当然是足够大的,可以显示出折半查找的优势;但是这样就引发了一个新的问题:对cooked_params进行排序比较难用,因为这可能需要分隔一些#ifdef程序段——请参看从19268行到19272行的例子。进一步说,因为算法只是查找前缀,而不使用完全匹配,在遍历数组中的各个项时对遍历次序比较敏感,所以这种特性在使用不同的查找次序时就很难再保持了。然而,这些问题并不是不可克服的(程序员可以预先静态地为引导程序建立一颗前缀树),如果性能在这里是主要因素,那么这种努力也是值得的。但是,由于性能在这里

并不是主要问题，所以简单性才被作为最重要的因素体现出来。

即使这样，在类似的`root_dev_names`数组（19085行）中——这个数组把硬件设备名的前缀映射到它们的主ID号上，开发者仍然可以简单地通过把比较常用的项（IDE和SCSI磁盘）放在不太常用的项（并口IDE CD）的前面以节省出一点性能。但是我在`raw_params`或`cooked_params`中并没有发现与之类似的模式。

另外一件需要注意的事是：现在你可以猜想一下为什么`ro`、`rw`和`debug`选项在`parse_options`中测试而不在这里测试，`parse_options`要检测精确的匹配，但是`checksetup`只检测前缀。作为一个特殊的情况，`ro`选项碰巧正好是`root=`（19553行）的前缀，这样如果这三个选项彼此合并，就需要仔细处理了。这似乎仍然是一个相当无力的原因。考虑一下`noinitrd`选项（19251行）。这是`cooked_params`的一个项，因而只需要匹配前缀，而且与之相关联的设置函数（`no_initrd`，19902行）将忽略所有可能已经传递给它们的参数——这正像`ro`、`rw`和`debug`被包含在`cooked_params`中时所可能进行的工作一样。

19632: 这个循环为`cooked_params`数组的处理工作和前面一个循环为`raw_params`数组的处理工作相同。这两个循环（当然不包括循环使用的数组）间的唯一区别是本循环在调用设置函数之前，使用`get_options`（19062行）处理`line`中`=sign`后面的部分。简单地说，`get_options`使用10个负整数填充`ints[1]`到`ints[10]`。`ints[0]`中是`ints`中使用元素的个数——也就是，它记录了存储在`ints`中的`ints get_options`数量。接着这个数组将被传递给设置函数，该设置函数则会按照自己喜欢的方式对该数组内容进行解释。

19640: 返回0，说明`line`中所包含的内核选项不能被函数理解。

3. `profile_setup`

19076: `profile_setup`是`checksetup`调用的设置函数的一个完美的例子：这个函数十分短小，使用`ints`参数处理了部分内容。而且到目前为止你也应该对它的目的有了一定了解。正如前面提到的一样，用户可以在启动的时候设置`prof_shift`的值——这里正是它的实现方式。当内核启动过程提供`profile=`选项时，就调用`profile_setup`函数。前缀字符串和函数在19235行被联系在一起。注意这是在`cooked_params`中，因此`profile_setup`取得的是处理过的参数。

19079: 如果参数中存在`profile=`的形式，就使用`profile=`后面的第一个数字作为`prof_shift`的新值。选项给出的其他参数都被简单地忽略了。

19081: 如果给出了`profile=`选项，但是没有为它提供参数，`prof_shift`的缺省值就是2。这个缺省值有些奇怪，因为我们已经知道，这意味着使用四分之一的内核可用内存来配置其余部分——这是一个很大的开销。但是另一方面，使用这些内存有助于更精确地定位问题热点——只有很少的几条指令存在不确定性，这样应该比较容易地把问题限制在一两行源程序代码内。那张图也并不是像我所画的那样简单：因为图中只描述了内核代码，这种开销还不到内核所有内存空间的25%，但是对于所覆盖的代码量来说却并不止25%。

4.3 `init`

`init`从许多方面看都是一个非常特殊的进程。这是内核运行的第一个用户进程，它要负责

触发其他必需的进程以使系统作为一个整体进入可用的状态。这些工作由/etc/inittab文件控制，通常包括设置getty进程以接受用户登录，建立网络服务，例如FTP和HTTP守护进程，等等。如果没有这些进程，用户就不可能完成很多工作，这样成功启动内核就显得没有多大意义了。

这种设计的另外一个主要的副作用是init是系统中所有进程的祖先。init产生getty进程，getty进程产生login进程，login进程产生你自己的shell，使用自己的shell，可以产生每一个你运行的进程。在所有的结果中，这有助于确保内核进程表中的所有项最终都能够得到处理。进程结束以后将其清除（回收）的工作首先应由其父进程完成；如果父进程已经退出，那么祖父进程就要担负起这种责任；如果祖父进程已经退出，那么曾祖父进程就要担负起这种责任，周而复始。通过这种方式，从不退出的init进程就可能要负责回收其他进程。

因此，为了确保这些重要的工作都能正确执行，内核初始化进程所需要做的最后一步工作就是创建init进程，接下来就加以描述。

init

20044: unused参数来源于该函数的非常规调用。init函数——不要和init进程搞混了，后者是它随后要创建的——作为内核线程开始生命周期，一个作为内核的一部分运行的进程（如果你编写过多线程的程序，这里的内核线程可能会同你所已经知道的线程意义有所不同——在那种意义下，它不是一个内核线程）。实际上，init函数就像是新进程使用的剥离出来了的main函数，unused参数是一个独立的指针，其值指向为给定进程所提供的信息——这比通常使用argc、argv和envp参数传递的信息要少得多。init函数碰巧不需要额外的信息，这个参数命名为unused，就是要强调这一点。

为了确保在这一点上你不会产生困惑，我们在这里再对整个机制进行扼要重复：init函数是内核的一部分；它在内核中作为内核的一个独立的执行部分运行；也就是说，无论从哪个方面看它都是内核代码。但是，init进程就不是这样了。在某些方面，init进程是一个特殊的进程，但是不属于内核本身；其代码存储在磁盘上单独的可执行映像中，这和其他程序一样。因为init函数后来产生init进程，而它自己又恰好作为进程运行，这样就很容易产生混淆。

因为idle进程已经占据了进程ID号（PID）0，init（当然是init）就被赋值为下一个可用的PID，也就是1（进程ID在第7章中讨论）。内核重复假定PID为1的进程是init，因此这种特性在没有充分地相互作用，也就是没有同步地进行修改的情况下是不能改变的。

20046: 调用lock_kernel（对应UP版本17492行，对应SMP版本10174行）执行后续几行，而不会受到其他会受随后工作的影响的内核模块的干扰。内核锁随后在20053行被释放。

20047: 调用do_basic_setup（19916行）初始化总线，并随同其他工作产生一些其他内核线程。

20052: 内核已完全完成初始化了，因此free_initmem（7620行）可以舍弃内核的.text.init节的函数和.data.init节的数据。所有使用__initfunc标记过的函数和使用__initdata标记过的数据现在都不能使用了，它们曾经获得的内存现在也可能重新用于其他目的了。

- 20055: 如果可能, 打开控制台设备, 这样init进程就拥有一个控制台, 可以向其中写入信息, 也可以从其中读取输入信息。实际上init进程除了打印错误信息以外, 并不使用控制台, 但是如果调用的是shell或者其他需要交互的进程, 而不是init, 那么就需要一个可以交互的输入源。如果成功执行open, /dev/console就成为init的标准输入源(文件描述符0)。
- 20059: 调用dup打开/dev/console文件描述符两次, 这样, init就也使用它供标准输出和标准错误使用(文件描述符1和2)。假设20055行的open成功执行(正常情况), init现在就有三个文件描述符——标准输入、标准输出及标准错误——全都加载在系统控制台之上。
- 20067: 如果内核命令行中给出了到init的直接路径(或者别的可替代的程序), 现在就试图执行init。
- 因为当execve成功执行目标程序时并不返回, 只有当前面的所有处理过程都失败时, 才能执行相关的表达式。接下来的几行在几个地方查找init, 按照可能性由高到低的顺序依次是: 首先是/sbin/init, 这是init标准的位置; 接下来是两个可能的位置, /etc/init和/bin/init。
- 20072: 这些是init可能出现的所有地方。如果现在还没有出现, init就无法找到它的这个同名者了, 机器可能就崩溃了。因此, 它就会试图建立一个交互的shell(/bin/sh)来代替。现在init最后的希望就是root用户可以修复这种错误并重新启动机器(可以肯定, root也正是希望如此)。
- 20073: init甚至不能创建shell——一定是发生了什么问题! 按照它们所说的, 当所有其他情况都失败时, 调用panic(25563行)。这样内核就会试图同步磁盘, 确保其状态一致, 然后暂停进程的执行。如果超过了内核选项中定义的时间, 它也可能会重新启动机器。

第5章 系统调用

大部分介绍Unix内核的书籍都没有详细说明系统调用，我认为这是一个失误。实际上，我们实际需要的系统调用现在已经十分完美。因此，从某种意义上来说，研究系统调用的实现是无意义的——如果你想为Linux内核的改进贡献自己的力量，还有其他许多方面更值得投入精力。

然而，对于我们来说，仔细研究少量系统调用是十分值得的。这样就有机会初步了解一些概念，这些概念将在本书中逐步详细介绍，例如进程处理和内存。这使你可以详细了解Linux内核编程的特点。这包括一些和你过去在学校里（或工作中）所学的内容不同的方法。和其他编程任务相比，Linux内核编程的一个显著特点是它不断同三个成见进行斗争——这三个成见就是速度、正确和清晰，我们不可能同时获取这三个方面，至少并不总是能够。

5.1 什么是系统调用

系统调用发生在用户进程（比如emacs）通过调用特殊函数（例如open）以请求内核提供服务的时候。在这里，用户进程被暂时挂起。内核检验用户请求，尝试执行，并把结果反馈给用户进程，接着用户进程重新启动，随后我们将详细讨论这种机制。

系统调用负责保护对内核所管理的资源的访问，系统调用中的几个大类主要有：处理I/O请求（open、close、read、write、poll等等）、进程（fork、execve、kill等等）、时间（time、settimeofday等等）以及内存（mmap、brk等等）的系统调用。几乎所有的系统调用都可以归入这几类中。

然而，从根本上来说，系统调用可能和它表面上有所不同。首先，在Linux中，C库中对于一些系统调用的实现是建立在其他系统调用的基础之上的。例如，waitpid是通过简单调用wait4实现的，但是它们两个都是作为独立的系统调用说明的。其他的传统系统调用，如sigmask和ftime是由C库而不是由Linux内核本身实现的；即使不是全部，至少大部分是如此。

当然，从技巧的一面来看这是无害的——从应用程序的观点来看，系统调用就和其他的函数调用一样。只要结果符合预计的情况，应用程序就不能确定是否真正使用到了内核（这种处理方式还有一个潜在的优点：用户可以直接触发的内核代码越少，出现安全漏洞的机会也就越少）。但是，由于使用这种技巧所引起的困扰将会使我们的讨论更为困难。实际上，系统调用这一术语通常被演讲者用来说明在第一个Unix版本中的任何对系统的调用。但是在本章中我们只对“真正”的系统调用感兴趣——真正的系统调用至少包括用户进程对部分内核代码的调用。

系统调用必须返回int的值，并且也只能返回int的值。为了方便起见，返回值如果为零或者为正，就说明调用成功；为负则说明发生了错误。就像老练的C程序员所知道的一样，当标准C库中的函数发生错误时，会通过设置全局整型变量errno指明发生错误的属性，系统调用的原理和它相同。然而，仅仅研究内核源程序代码并不能够获得这种系统调用方式的全部意义。如果发生了错误，系统调用简单返回自己所期望的负数错误号，其余部分则由标准C库实

现（正常情况下，用户代码并不直接调用内核系统函数，而是要通过标准C库中专门负责翻译的一个小层次（thin layer）实现）。我们随便举一个例子，27825行（`sys_nanosleep`的一部分）返回-EINVAL指明所提供的值越界了。标准C库中实际处理`sys_nanosleep`的代码会注意到返回的负值，从而设置`errno`和EINVAL，并且自己返回-1给原始的调用者。

在最近的内核版本中，系统调用返回负值偶尔也不一定表示错误。在目前的几个系统调用中（例如`lseek`），即使结果正确也会返回一个很大的负值。最近，错误返回值是在-1到-4095范围之内。现在，标准C库实现能够以更加成熟和高级的方式解释系统调用的返回值；当返回值为负时，内核本身就不用再做任何特殊的处理了。

中断、内核空间和用户空间

我们将在第6章中介绍中断和在第8章中介绍内存时再次明确这些概念。但是在本章中，我们只需要粗略地了解一些术语。

第一个术语是中断（interrupt），它来源于两个方面：硬件中断，例如磁盘指明其中存放一些数据（这与本章无关）；软件中断，一种等价的软件机制。在x86系列CPU中，软件中断是用户进程通知内核需要触发系统调用的基本方法（出于这种目的使用的中断号是0x80，对于Intel芯片的研究者来说更为熟悉的是INT 80h）。内核通过`system_call`（171行）函数响应中断，这一点我们马上就会介绍。

另外两个术语是内核空间（kernel space）和用户空间（user space），它们分别对应内核保留的内存和用户进程保留的内存。当然，多用户进程也经常同时运行，而且各个进程之间通常不会共享它们的内存，但是，任何一个用户进程使用的内存都称为用户空间。内核在某一个时刻通常只和一个用户进程交互，因此实际上不会引起任何混乱。

由于这些内存空间是相互独立的，用户进程根本不能直接访问内核空间，内核也只能通过`put_user`（13278行）和`get_user`（13254行）宏和类似的宏才可以访问用户空间。因为系统调用是进程和进程所运行的操作系统之间的接口，所以系统调用需要频繁地和用户空间交互，因此这些宏也就会不时的在系统调用中出现。在通过数值传递参数的情况下并不需要它们，但是当用户把指针（内核通过这个指针进行读写）传递给系统调用时，就需要这些宏了。

5.2 如何激活系统调用

系统调用的激活有两种方法：`system_call`函数和`lcall7`调用门（call gate）（请参见135行）。（你可能还听说过一种机制，`syscall`函数，是通过调用`lcall7`实现的——至少在x86平台上是如此，因此，它并不是一个特有的方法）。本节将细致地讨论一下这两种机制。

在阅读的过程中请注意系统调用本身并不关心它们是由`system_call`还是由`lcall7`激活的。这种把系统调用和其实现方式区别开来的方法是十分精巧的。这样，如果出于某种原因我们不得不增加一种激活系统调用的方法，我们也不必修改系统调用本身来支持这种方法。

在你浏览这些汇编代码之前要注意这些机器指令中操作数的顺序和普通Intel的次序相反。虽然还有一些其他的语法区别，但是操作数反序是最令人迷惑的。如果你还记得Intel的语法：

```
mov eax, 0
```

（本句代码的意思是把常数0传送到寄存器EAX中）在这里应该写作：

```
movl $0, %eax
```

这样你就能够正确通过（内核使用的语法是AT&T的汇编语法。在GNU汇编文档中有更多资料）。

5.2.1 system_call

system_call (171行) 是所有系统调用的入口点（这是对于内部代码来说的，lcall7用来支持iBCS2，这一点我们很快就会讨论）。正如前面标题注释中说明的一样，目的是为普通情况简单地实现直接的流程，不采用跳转，因此函数的各个部分都是离散的——整体的流量控制已经因为要避免普通情况下的多分支而变得非常复杂（分支的避免是十分值得的，因为它们引起的代价非常昂贵。它们可以清空CPU管道，使现存CPU的并行加速机制失效）。

图5-1显示了作为system_call的一部分出现的分支目标标号以及它们之间的流程控制方向，该图可以在你阅读本部分讨论内容时提供很大的帮助。图中system_call和restore_all两个标号比其他标号都要大，因为这两处是该函数正常的出口点和入口点；然而，还有另外两个入口点，这一点在本章的后续内容中很快就可以看到。

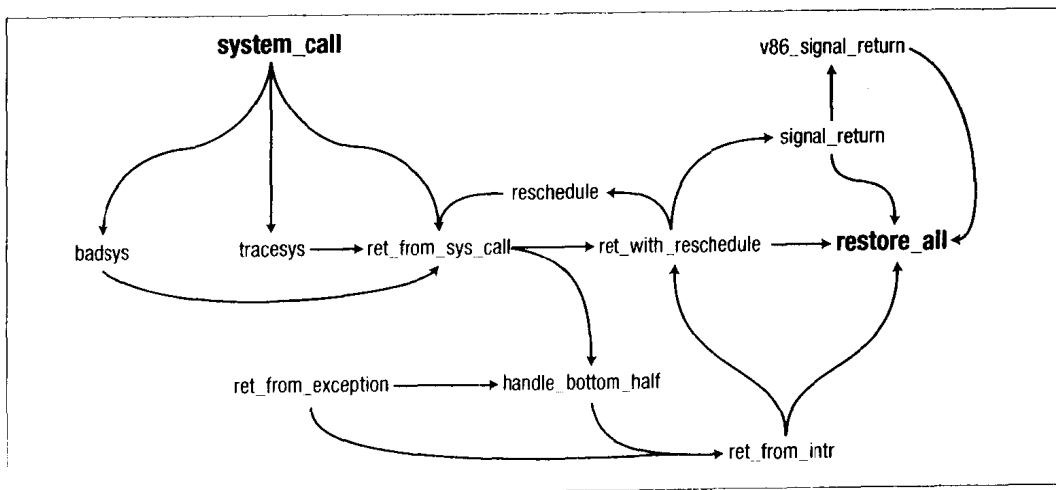


图5-1 system_call的流程控制

system_call是由标准C库激活的，该标准C库会把自己希望传递的参数装载到CPU寄存器中，并触发0x80软件中断（system_call在这里是一个中断处理程序）。内核记录了软件中断和6828行的system_call函数的联系（SYSCALL_VECTOR是在1713行宏定义为0x80的）。

system_call

171: system_call的第一个参数是所希望激活的系统调用的数目；它存储在EAX寄存器中。system_call还允许有多达四个的参数和系统调用一起传送。在一些极其罕见的情况下使用四个参数的限制是负担繁重的，通常可以建立一个指向结构的指针参数来巧妙地完成同样功能，指针指向的结构中可以包含你所需要的一切信息。

随后可能需要EAX值的一个额外拷贝，因此通过将其压栈而保存起来；这个值就是218行的ORIG_EAX (%esp) 表达式的值。

173: SAVE_ALL宏是在85行定义的，它把所有寄存器的值拷贝压入CPU的堆栈。随后，就在system_call返回之前，使用RESTORE_ALL (100行) 把栈中的值弹出。在这中间，system_call可以根据需要自由使用寄存器的值。更重要的是，任何它所调用

的C函数都可以从栈中查找到所希望的参数，因为SAVE_ALL已经把所有寄存器的值都压入栈中了。

结果栈的结构从26行开始描述。像0(%esp)和4(%esp)一样的表达式指明了堆栈指针(ESP寄存器)的一种替换形式——分别表示ESP上的0字节，ESP上的4字节等等。特别要注意的是在前面一行中压入堆栈的EAX的拷贝已经变成本标题注释作为orig_eax所描述的内容，它们是由SAVE_ALL压入寄存器之上的堆栈的(orig_eax之上的寄存器在这里早已就绪了)。

还需注意：这可能有点令人迷惑——由于我们调用orig_eax时EAX的拷贝已经压入了堆栈，它是否有可能在其他寄存器下面，而不是在其他寄存器上面呢？答案既是肯定的，也是否定的。x86的堆栈指针寄存器ESP在有数据压入堆栈时会减少——堆栈会向内存低地址发展。因此，orig_eax逻辑上是在其他值的下面，但是物理上却是在其他值的上面。

从51行开始的一系列宏有助于使这些替换更容易理解。例如，EAX(%esp)就和18(%esp)相同——然而前一种方法通过表达式引用存储在堆栈中的EAX寄存器拷贝的决定可以使整个过程更加简单。

- 174: 从EBX寄存器中取得指向当前任务的指针。完成这个工作的宏GET_CURRENT(131行)对于在大部分代码中使用的C函数get_current(10277行)来说是一个无限循环。

此后，当看到类似于foo(%ebx)或者foo(%esp)的表达式时，这意味着这些代码正在引用代表当前进程的结构中的字段——16325行的struct task_struct——在第7章中将对它进行更详细的介绍(更确切的描述是，%ebx的置换在struct task_struct中，%esp的置换在与struct task_struct相关联的struct pt_regs结构中。但是这些细节在这里都并不重要)。

- 175: 检查(EAX中的)系统调用的数目是否超过系统调用的最大数量(此处EAX为一个无符号数，因此不可能为负值)。如果的确超过了，就向前跳转到badsys(223行)。
- 177: 检测系统调用是否正被跟踪。如strace之类的程序为有兴趣的人提供了系统调用的跟踪工具，或者额外的调试信息：如果能够监测到正在执行的系统调用，那么你就可以了解到当前程序正在处理的内容。如果系统调用正被跟踪，控制流程就向前跳转到tracesys(215行)。
- 179: 调用系统函数。此处有很多工作需要处理。首先，SYMBOL_NAME宏不处理任何工作，只是简单的为参数文本所替换，因此可以将其忽略。sys_call_table是在当前文件(arch/i386/kernel/entry.S)的末尾从373行开始定义的。这是一张由指向实现各种系统调用的内核函数的函数指针组成的表。

本行中第二对圆括号中包含了三个使用逗号分隔开的参数(第一个参数为空)；这里就是实现数组索引的地方。当然，这个数组是以sys_call_table作为索引的，这称为偏移(displacement)。这三个参数是数组的基地址、索引(EAX，系统调用的数目)和大小，或者每个数组元素中的字节数——在这里就是4。由于数组基地址为空，就将其当作0——但是它要和偏移地址sys_call_table相加，简单的说就是sys_call_table被当作数组的基地址。本行基本上等同于如下的C表达式：

```
/* Call a function in an array of functions. */
(sys_call_table[eax]);
```

然而，C当然还要处理许多繁重的工作，例如为你记录数组元素的大小。不要忘记，系统调用的参数早已经存储在堆栈中了，这主要由调用者提供给system_call并使用SAVE_ALL把它们压栈。

- 180: 系统调用已经返回。它在EAX寄存器中的返回值（这个值同时也是system_call的返回值）被存储起来。返回值被存储在堆栈中的EAX内，以使得RESTORE_ALL可以迅速地恢复实际的EAX寄存器及其他寄存器的值。
- 182: 接下来的代码仍然是system_call的一部分，它是一个也可以命名为ret_from_sys_call和ret_from_intr的独立入口点。它们偶尔会被C直接调用，也可以从system_call的其他部分跳转过来的。
- 185: 接下来的几行检测“下半部分（bottom half）”是否激活；如果激活了，就跳转到handle_bottom_half标号（242行）并立即开始处理。下半部分是中断进程的一部分，将在下一章中讨论。
- 189: 检查该进程是否为再次调度做了标记（记住表达式\$0就是常量0的系统简单表示）。如果的确如此，就跳转到reschedule标号（247行）。
- 191: 检测是否还有挂起的信号，如果有的话，下一行就向前跳转到signal_return（197行）。
- 193: restore_all标号是system_call的退出点。其主体就是简单的RESTORE_ALL宏（100行），该宏将恢复早先由SAVE_ALL存储的参数并返回给system_call的调用者。
- 197: 当system_call从系统调用返回前，如果它检测到需要将信号传送给当前的进程时，才会执行到signal_return。它通过使中断再次可用开始执行，有关内容将在第6章中介绍。
- 199: 如果返回虚拟8086模式（这不是本书的主题），就向前跳转到v86_signal_return（207行）。
- 202: system_call要调用C函数do_signal（3364行，在第6章中讨论）来释放信号。do_signal需要两个参数，这两个参数都是通过寄存器传递的；第一个是EAX寄存器，另一个是EDX寄存器。system_call（在200行）早已把第一个参数的值赋给了EAX；现在，就把EDX寄存器和寄存器本身进行XOR操作，从而将其清0，这样do_signal就认为这是一个空指针。
- 203: 调用do_signal传递信号，并且跳回到restore_all（193行）结束。
- 207: 由于虚拟8086模式不是本书的主题，我们将忽略大部分v86_signal_return。然而，它和signal_return的情况非常类似。
- 215: 如果当前进程的系统调用正由其祖先跟踪，就像strace程序中那样，那么就可以执行到tracesys标号。这一部分的基本思想如同179行一样是通过syscall_table调用系统函数，但是这里把该调用和对syscall_trace函数的调用捆绑在一起。后面的这个函数在本书中并没有涉及到，它能够终止当前进程并通知其祖先注意当前进程将要激活一个系统调用。

EAX操作和这些代码的交错使用最初可能容易令人产生困惑。system_call把存

储在堆栈中的EAX拷贝赋给-ENOSYS，调用syscall_trace，在172行再从所做的拷贝中恢复EAX的值，调用实际的系统调用，把系统调用的返回值置入堆栈中EAX的位置，再次调用syscall_trace。

这种方式背后的原因是syscall_trace(或者更准确地说是它所要用的跟踪程序)需要知道它是在实际系统调用之前还是之后被调用的。-ENOSYS的值能够用来指示它是在实际系统调用执行之前被调用的，因为实际中所有实现的系统调用的执行都不会返回-ENOSYS。因此，EAX在堆栈中的备份在第一次调用syscall_trace之前是-ENOSYS，但是在第二次调用syscall_trace之前就不再是了(除非是调用sys_ni_syscall的时候，在这种情况下，我们并不关心是怎样跟踪的)。218行和219行中EAX的作用只是找出要调用的系统调用，这和无须跟踪的情况是一致的。

- 222: 被跟踪的系统调用已经返回，流程控制跳转回ret_from_sys_call(184行)并以与普通的无须跟踪的情况相同的方式结束。
- 223: 当系统调用的数目越界时，就可以执行到badsys标号。在这种情况下，system_call必须返回-ENOSYS(ENOSYS在82行将它赋值为38)。正如前面提到的一样，调用者会识别出这是一个错误，因为返回值在-1到-4095之间。
- 228: 在诸如除零错误(请参见279行)之类的CPU异常中断情况下将执行到ret_from_exception标号；但是system_call内部的所有代码都不会执行到这个标号。如果有下半部分是激活的，现在就是它在起作用了。
- 233: 处理完下半部分之后或者从上面的情况简单执行下来(虽然没有下半部分是激活的，但是同样也触发了CPU异常)，就执行到了ret_from_intr标号。这是一个全局可访问符号，因此可能在内核的其他部分也会有对它的调用。
- 237: 被保存的CPU的EFLAGS和CS寄存器在此已经被并入EAX，因而高24位的值(其中恰好包含了一位在70行定义的非常有用的VM_MASK)来源于EFLAGS，其他低8位的值来源于CS。该行隐式的同时对这两部分进行测试以判断进程是返回虚拟8086模式(这是VM_MASK的部分)还是用户模式(这是3的部分——用户模式的优先等级是3)。下面是近似的等价C代码：

```
/* Mix eflags and cs in eax. */
eax = eflags & ~0xff;
eax |= cs & 0xff;
/* Simultaneously test lower 2 bits
 * and VM_MASK bit. */
if (eax & (VM_MASK | 3))
    goto ret_with_reschedule;
goto restore_all;
```

- 238: 如果这些条件中有一个能得到满足，流程控制就跳转到ret_with_reschedule(188行)标号，测试在system_call返回之前进程是否需要再次调度。否则，调用者就是一个内核任务，因此system_call通过跳转到restore_all(193行)来跳过重新调度的内容。
- 242: 无论何时system_call使用一个下半部分服务时都可以执行到handle_bottom_half标号。它简单地调用第6章中介绍的C函数bottom_half(29126行)，然后跳回到ret_from_intr(233行)。
- 248: system_call的最后一个部分在reschedule标号之下。当产生系统调用的进程已经被标

记为需要进行重新调度时，就可以执行到这个标号；一般说来，这是因为进程的时间片已经用完了——也就是说，进程到目前为止已经尽可能地拥有CPU了，应该给其他进程一个机会来运行了。因此，在必要的情况下就可以调用C函数schedule (26686行) 交出CPU，同时流程控制转向249行。CPU调度是第7章中讨论的一个主题。

5.2.2 lcall7

Linux支持Intel二进制兼容规范标准的版本2 (iBCS2) (iBCS2中的小写字母i显然是有意的，但是该标准却没有对此进行解释，这样看来似乎和现实的Intel系列的CPU例如i386、i486等等是一致的)。iBCS2的规范中规定了所有基于x86的Unix系统的应用程序的标准内核接口，这些系统不仅包括Linux，而且还包括其他自由的x86 Unix (例如FreeBSD)，也还包括Solaris/x86、SCO Unix等等。这些标准接口使得为其他Unix系统开发的二进制商业软件在Linux系统中能够直接运行，反之亦然 (而且，近期新开发软件向其他Unix移植的情况越来越多)。例如，Corel公司的WordPerfect的SCO Unix的二进制代码在还没有Linux的本地版本的WordPerfect之前就可以使用iBCS2在Linux上良好地运行。

iBCS2标准有很多组成部分，但是我们现在关心的是这些系统调用如何协调一致来适应这些迥然不同的Unix系统。这是通过lcall7调用门实现的。它是一个相当简单的汇编函数 (尤其是和system_call相比而言更是如此)，仅仅定位并全权委托一个C函数来处理细节。(调用门是x86 CPU的一种特性，通过这种特性用户任务可以在安全受控的模式下调用内核代码。) 这种调用门在6802行进行设定。

lcall7

135: 前面的几行将通过调整处理器堆栈以使堆栈的内容和system_call预期的相同——system_call中的一些代码将会完成清理工作，这样所有的内容都可以连续存放了。

145: 基于同样的思想，lcall7把指向当前任务的指针置入EBX寄存器，这一点和system_call的情况是相同的。但是，它的执行方式却与system_call不同，这就比较奇怪了。这三行可以等价地按如下形式书写：

```
pushl %esp
GET_CURRENT(%ebx)
```

这种实现的执行速度并不比原有的更快，在将宏展开以后，实际上这还是同样的三条指令以不同的次序组合在一起而已。这样做的优点是可以和文件中的其他代码更为一致，而且代码也许会更清晰一些。

148: 取得指向当前任务exec_domain域的指针，使用这个域以获取指向其lcall7处理程序的指针，接着调用这个处理程序。

本书中并没有对执行域 (execution domains) 进行详细说明——但是简单说来，内核使用执行域实现了部分iBCS2标准。在15977行你可以找到struct exec_domain结构。default_exec_domain (22807行) 是缺省的执行域，它拥有一个缺省的lcall7处理程序。它就是no_lcall7 (22820行)。其基本的执行方式类似于SVR4风格的Unix，如果调用进程没有成功，就传送一个分段违例信号 (segmentation violation signal) 给调用的进程。

152: 跳转到ret_from_sys_call标号（184行——注意这是在system_call内部的）清除并返回，就像是正常的系统调用一样。

5.3 系统调用样例

现在你已经知道了系统调用是如何激活的，接下来我们将通过对几个系统调用例子的剖析来了解一下它们的工作方式。注意系统调用foo几乎都是使用名为sys_foo的内核函数实现的，但是在某些情况下该函数也会使用一个名为do_foo的辅助函数。

1. sys_ni_syscall

29185: sys_ni_syscall的确是最简单的系统调用；它只是简单地返回ENOSYS错误。最初的时候这可能显得没有什么作用，但是它的确是有用的。实际上，sys_ni_syscall在sys_call_table中占据了很多位置——而且其原因并不只有一个。开始的时候，sys_ni_syscall在位置0（374行），因为，如果漏洞百出的代码错误地调用了system_call——例如，没有初始化作为参数传递给system_call的变量，在这种偶然的变量定义中，0是最可能的值。如果我们能够避免这种情况，那么在错误发生时就不用采取像杀掉进程这样的措施（当然，只要允许有用工作的进行，就不可能防止所有的错误）。这种使用表的元素0作为抵御错误的手段在内核中作为良好的经验而被广泛使用。

而且，你还会发现sys_ni_syscall在表中明显出现的地方就多达十几处。这些条目代表了那些已经从内核中移出的系统调用——例如在418行，就代替了已经废弃了的prof系统调用。我们不能简单地把另外的实际系统调用放在这里，因为老的二进制代码可能还会使用到这些已经废弃了的系统调用号。如果一个程序试图调用这些老的系统调用，但是结果却与预期的完全不同，例如打开了一个文件，这会令人感到奇怪的。

最后，sys_ni_syscall将占据表尾部所有未用的空间；这一点是在从572行到574行的代码中实现的，它根据需要重复使用这些项来填充表。由于sys_ni_syscall只是简单返回ENOSYS错误号，对它的调用和跳转到system_call中的badsys标号作用是相同的——也就是说，使用指向这些表项的系统调用号和在外对整个表进行全部索引具有相同的作用。因此，我们不用改变NR_syscalls就可以在表中增加（或者删除）系统调用，但是其效果与我们真的对NR_syscalls进行了修改一样（不管怎样，这都是由NR_syscalls所建立的限制条件所决定的）。

到现在也许你已经猜到了，sys_ni_syscall中的“ni”并不是指Monty Python的“说‘Ni’的骑士”；而是指“not implemented（没有实现）”。

对于这个简单的函数我们需要研究的另外一个问题是asmlinkage标志。这是为一些gcc功能定义的一个宏，它告诉编译器该函数不希望从寄存器中（这是一种普通的优化）取得任何参数，而希望仅仅从CPU堆栈中取得参数。回忆一下我们前面提到过system_call使用第一个参数作为系统调用的数目，同时还允许另外四个参数和系统调用一起传递。system_call通过把其他参数（这些参数是通过寄存器传递过来的）滞留在堆栈中简单地实现了这种技巧。所有的系统调用都使用asmlinkage标志作了标记，因此它们都要查找堆栈以获得参数。当然，在sys_ni_syscall的情况下

这并没有任何区别，因为`sys_ni_syscall`并不需要任何参数。但是对于其他大部分系统调用来说，这就是个问题了。并且，由于在其他很多函数前面都有`asm` linkage标志，我想你也应该对它有些了解。

2. `sys_time`

31394: `sys_time`是包含几个重要概念的简单系统调用。它实现了系统调用`time`，返回值是从某个特定的时间点（1970年1月1日午夜UTC）以来经过的秒数。这个数字被作为全局变量`xtime`（请参见26095行；它被声明为`volatile`型的变量，因为它可以通过中断加以修改，这一点我们在第6章中就会看到）的一部分，通过`CURRENT_TIME`宏（请参见16598行）可以访问它。

31400: 该函数非常直接地实现了它的简单定义。当前时间首先被存储在局部变量`i`中。

31402: 如果所提供的指针`tloc`是非空的，返回值也将被拷贝到指针指向的位置。该函数的一个微妙之处就在于此；它把`i`拷贝到用户空间中，而不是使用`CURRENT_TIME`宏来重新对其进行计算，这基于两个原因：

- `CURRENT_TIME`宏的定义以后可能会改变，新的实现方法可能会由于某种原因而速度比较慢，但是对于`i`的访问至少应该和`CURRENT_TIME`宏展开的速度同样快。
- 使用这种方式处理，确保结果的一致性：如果代码刚好执行到31400行和31402行之间时时间发生了改变，`sys_time`可能把一个值拷贝到`*tloc`中，但是在结束之后却返回另一个值。

另外还有一个小的方面需要注意，此处的代码不使用`&&`来编写，而是使用两个`if`，这可能有一点令人奇怪。内核中采用这些看起来非常特殊的代码，一般原因都是由于速度的要求，但是`gcc`为`&&`版本和两个`if`版本生成的代码是等同的，因此这里的原因就不可能是速度的要求——除非这些代码是在早期`gcc`版本下开发的，这样才有些意义。

31403: 如果`sys_time`不能访问所提供的位置（一般都是因为`tloc`无效），它就把`-EFAULT`的值赋给`i`，从而在31405行返回错误代码。

31405: 为调用者返回的`i`或者是当前时间，或者是`-EFAULT`。

3. `sys_reboot`

29298: 内核中其他地方可能都没有`sys_reboot`的实现方法先进。其原因可以理解为：根据调用的名字我们就可以知道，`reboot`系统调用可以用来重新启动机器。根据所提供的参数，它还能够挂起机器、关闭电源、允许或者禁止使用`Ctrl+Alt+Del`组合键来重启机器。如果你要使用这个函数编写代码，需要特别注意它上面的注释标题的警告：首先同步磁盘，否则磁盘缓冲区中的数据可能会丢失。

由于它可能为系统引发的潜在后果，`sys_reboot`需要几个特殊参数，这一点马上就会讨论。

29305: 如果调用者不具有`CAP_SYS_BOOT`（14096行）权能（`capability`），系统就会返回`EPERM`错误。权能在第7章中会详细讨论。现在，简单地说就是：权能是检测用户是否具有特定权限的方法。

29309: 在这里，这种偏执的思想充分发挥了作用。`sys_reboot`根据从16002到16005行定

义的特殊数字检测参数magic1和magic2。这种思想是：如果sys_reboot在某种程度上是被偶然调用的，那么就不太可能再从由magic1和magic2组成的小集合中同时提取值。注意，这并不意味着这是一个防止粗心的安全措施。

顺便说一下，这些特殊数字并不是随机选取的。第一个参数的关系是十分明显的，它是“感受死亡 (feel dead)”的双关语。后面的三个参数要用十六进制才能了解它们全部的意思：它们分别是0x28121969, 0x5121996, 0x16041998。这似乎代表Linus的妻子（或者就是Linus自己）和他两个女儿的生日。由此推论，当Linus和他的妻子养育了更多儿女的时候，重启动需要的特殊参数可能在某种程度上会增加。不过我想在他们用尽32位可能空间之前，他的妻子就会制止他的行为了。

29315: 请求内核锁，这样能保证这段代码在某一时间只能由一个处理器执行。使用lock_kernel/unlock_kernel“函数对”所保护起来的任何其他代码对其他CPU都同样是不可访问的。在单处理器的机器中，这只是一个no-op（不处理任何事情）；而详细讨论它在多处理器上的作用则是第10章的内容。

29317: 在LINUX_REBOOT_CMD_RESTART的情况中，sys_reboot调用一系列基于reboot_notifier_list的函数来通知它们系统正在重新启动。正常情况下，这些函数都是操作系统关闭时需要清除的模块的一部分。这个列表函数似乎并不在内核中的其他地方使用——至少在标准内核发行版本中是这样，也许之外的其他模块可能使用这个列表。不管怎样，这个列表的存在可以方便其他人使用。

LINUX_REBOOT_CMD_RESTART和其他cmd识别出的值从16023行开始通过#define进行宏定义。这些值并没有潜在的意义，选用它们的简单原因是它们一般不会发生意外，并且相互之间各不相同。（有趣的是，LINUX_REBOOT_CMD_OFF为零，这是在意外情况下最不可能出现的一个值。但是，由于LINUX_REBOOT_CMD_OFF简单地禁止用户使用Ctrl+Alt+Del重新启动机器，它就是一种“安全”的意外了。）

29321: 打印警告信息以后，sys_reboot调用machine_restart（2185行）重启机器。正如从2298行中所看到的一样，machine_restart函数从来不会返回。但是不管怎样，对于machine_restart的调用后面都跟着一个break语句。

这仅仅是经典的良好的编程风格吗？的确如此，但是却不仅仅如此。文件kernel/sys.c的代码是属于体系结构无关部分的。但是machine_restart，它显然是体系结构所特有的，属于代码的体系结构特有的部分（arch/i386/kernel/process.c）。因而对于不同的移植版本也有所不同。我们并不清楚以后内核的每个移植版本的实现都不会返回——例如，它可能调度底层硬件重启但是本身要仍然持续运行几分钟，这就需要首先从函数中返回。或者更为确切的说法是，由于某些特定的原因，系统可能并不总是能够重启；或许某些软件所控制的硬件根本就不能重启。在这种平台上，machine_restart就应该可以返回，因此体系结构无关的代码应该对这种可能性有所准备。

针对这个问题，正式的发行版本中都至少包含一个退出端口，使machine_restart函数可以从这个端口返回：m68k端口。不同的基于m68k的机器支持的代

码也各不相同，由于本书主要是针对x86的，我不希望花费过多的时间来解析所有的细节。但是这的确是可能的（在其他情况下，`machine_restart`简单进入一个无限循环——既不重新启动机器，也不返回。但是这里我们担心的是需要返回的情况）。

因此，我们毕竟是需要`break`的。前面看起来只是简单的习惯甚至是偏执的思想在这里为了内核的移植性已经变成必须的了。

- 29324: 接下来的两种情况分别允许和禁止臭名昭著的`Ctrl+Alt+Del`组合键（这三个组合键也被称为“Vulcan神经收缩（Vulcan nerve pinch）”，“黑客之手（hacker's claw）”，“三指之礼（three-fingered salute）”，我最喜欢后面这个）。这些只是简单地设置全局`C_A_D`标志（在29160行定义，在29378行检测）。
- 29332: 这种情况和`LINUX_REBOOT_CMD_RESTART`类似，但只是暂停系统而不是将其重新启动。两者之间的一个区别是它调用`machine_halt`（2304行）——这是x86上的一条`no-op`指令，但是在其他平台上却要完成关闭系统的实际工作——而不是调用`machine_restart`。并且它会把`machine_halt`不能暂停的机器转入低功耗模式运行。它使用`do_exit`（23267行）杀死内核本身。
- 29340: 到现在为止，这已经是一种比较熟悉的模式了。这里，`sys_reboot`关闭机器电源，除了为可以使用软件自行关闭电源的系统调用`machine_power_off`（2307行）之外，其他的应该和暂停机器情况完全相同。
- 29348: `LINUX_REBOOT_CMD_RESTART2`的情况是已建立主题的一个变种。它接收命令，将其作为ASCII字符串传递，该字符串说明了机器应该如何关闭。字符串不会由`sys_reboot`本身来解释，而是使用`machine_restart`函数来解释；因而这种模式的意义，如果有的话，就是这些代码是平台相关的（我使用“如果有”的原因是启动机器——特别是在x86中——一般只有一种方法，因此其他的信息都可以被`machine_restart`忽略）。
- 29365: 调用者传递了一个无法识别的命令。`sys_reboot`不做任何处理，仅仅返回一个错误。因此，即使由`magic1`和`magic2`传递给`sys_reboot`正确的`magic`数值，它也无须处理任何内容。
- 29369: 一个可识别的命令被传递给`sys_reboot`。如果流程执行到这里，它可能就是两个设置`C_A_D`的命令之一，因为其他情况通常都是停止或者重新启动机器。在任何情况下，`sys_reboot`都简单把内核解锁并返回0以表示成功。

4. `sys_sysinfo`

- 24142: 一个只能返回一个整型值的系统调用。如果需要返回更多的信息，我们只需要使用类似于在系统调用中传递多于四个参数时所使用的技巧就可以了：我们通过一个指向结构的指针将结果返回。收集系统资源使用情况的`sysinfo`系统调用就是这种函数的一个样例。
- 24144: 分配并清空一个`struct sysinfo`结构（15004行）以暂时存储返回值。`sys_sysinfo`可以把结构中的每个域都独立地拷贝出来，但是这样会速度很慢、很不方便，而且必然不容易阅读。
- 24148: 禁止中断。这在第6章中会有详细的介绍；作为目前来说，我们只要说明这种模

式在使用的过程中能够确保sys_sysinfo正在使用的值不会改变就足够了。

- 24149: struct sysinfo结构的uptime域用来指明系统已经启动并运行了的秒数。这个值是使用jiffies (26146行) 和HZ来计算的。jiffies计算了系统运行过程中时钟的滴答次数; HZ是系统相关的一个参数, 它十分简单, 就是每秒内部时钟滴答的次数。
- 24151: 数组avenrun (27116行) 记录了运行队列的平均长度——也就是等待CPU的平均进程数——在最后的1秒钟、5秒钟和15秒钟。calc_load (27135行) 周期性地重复计算它的值。由于内核中是要严格禁止浮点数运算的, 所以只能通过计算变化的次数这一修正值来计算。
- 24155: 同样记录系统中当前运行的进程数。
- 24158: si_meminfo (07635行) 写入这个结构中的内存相关成员, si_swapinfo (38544行) 写入与虚拟内存相关的部分。
- 24161: 现在整个结构都已经全部填充了。sys_sysinfo试图将其拷贝回用户空间, 如果失败就返回EFAULT错误, 如果成功就返回0。

第6章 信号、中断和时间

信号 (signal) 是进程间通讯 (IPC) 的一种形式——是一个进程给另一个进程发送信息的方法。但是信息不可能很多——一个信号不可能携带详细的信息，即使是传送者的身份也不能被传递；唯一能够确定的事实是信号的确被发送了。(然而和经典信号不同，POSIX实时信号允许传送稍微多一点的信息。) 实际上，信号对于双向通讯是没有用处的。另外，根据某些限定，信号的接收者不必以任何方式作出响应，甚至可以直接忽略大部分信号。

虽然有这么多的限制，然而信号仍然是一种功能强大的十分有用的机制——毋庸置疑，这是Unix IPC中使用最频繁的机制。每当进程退出或者废弃一个空指针时，每当使用Ctrl+C键终止程序运行时，都要传递信号。

第9章会更详细地讨论IPC机制。

正如在Linux内核本身的代码注释中所说明的一样，中断 (Interrupt) 对于内核来说和信号是类似的。中断一般都是从磁盘之类的硬件设备送往内核，用以提示内核该设备需要加以注意。一个重要的硬件中断源就是定时器设备，它周期性地通知内核已经通过的时间。如同第5章中阐述的一样，中断也可以由用户进程通过软件产生。

在本章中，我们首先讨论一下Linux中信号和中断的实现，最后再浏览一下Linux的时间处理方式。

虽然内核对代码的要求标准非常严格，本章所涉及的代码仍然特别清晰明白。本章首先介绍相关的数据结构和它们之间的关系，接下来讨论操纵和查询它们的函数。

6.1 锁的概述

锁的基本思想是限制对共享资源的访问——共享资源包括共享的文件、共享的内存片及一次只能由一个CPU执行的代码段。概括地说，在单处理器上运行的Linux内核并不需要锁，这是因为在编写Linux内核时就已经注意到要尽量避免各种可能需要锁的情况了。但是，在多处理器机器上，一个处理器有时需要防止其他处理器对它的有害的介入。

include/asm-i386/spinlock.h文件 (从12582行开始) 并不使用难看的#ifdef把所有对锁函数的调用封装起来，它包含一系列对单处理器平台 (UP) 基本为空的宏，然而在多处理器平台 (SMP) 上这些宏将展开成为实际代码。因而内核的其他代码对UP和SMP (当涉及到这种特性时) 都是相同的，但是它们两个的效果却是迥然不同的。

第10章中涉及SMP的部分会对锁做深入的介绍。但是，由于你在代码中到处都能看到对锁宏的调用，特别是在本章所讨论到的代码中，这一点尤为明显，所以你应该首先对宏的用途有初步了解——以及为什么现在在大多数情况下我们都可以安全地将其忽略 (我们将在讨论的过程中对其中的异常情况进行说明)。

6.2 信号

Linux内核将信号分为两类:

- 非实时的 (nonrealtime) —— 大部分是些传统的信号, 例如SIGSEGV、SIGHUP和SIGKILL。
- 实时的 (realtime) —— 由POSIX 1003.1b标准规定, 它们同非实时信号有细微的区别。特别是实时信号具有进程可以配置的意义——就像是非实时信号SIGUSR1和SIGUSR2一样, 额外的信息能够和这些信号一起传送。它们也会排队, 因此如果在第一个信号处理完成之前有多个信号实例到达, 所有的信号都能够被正确传送; 这对于非实时信号则是不可能的。

在第7章中我们将会对“实时性”对于Linux内核的意义进行更详细的介绍——特别是“实时性”所不能够说明的内容。

信号数目的宏定义从12048行开始。实时信号的数目在SIGRTMIN和SIGRTMAX (分别在12087行和12088行) 所定义的范围之内。

6.2.1 数据结构

本节讨论信号代码使用的最重要的数据结构。

1. sigset_t

12035: sigset_t表示信号的集合。根据使用地点的不同, 它的意思也不同——例如, 它可能记录着正在等待某一个进程的信号 (如16425行struct task_struct的signal成员) 的集合, 也可能是某个进程已经请求阻塞了的信号 (如同一行中定义的同结构的blocked成员) 的集合。本书中我们会逐渐看到这些类似的应用。

12036: sigset_t的唯一一个组成部分是一组unsigned long (无符号长整型数), 其中的每一位都代表一个信号。注意到无符号长整型类型在整个内核代码中是作为一个字来处理的, 这和你所希望的可能有所出入——即使是在当前x86 CPU的讨论中, 有时候字也被用于说明16位类型。由于Linux是一个真32位操作系统, 将32位看作是一个字在绝大多数情况下是正确的 (将Linux称为真32位操作系统也有一些不准确, 因为在64位CPU上它也是一个真64位操作系统)。

这个数组的大小_NSIG_WORDS在12031行直接计算 (_NSIG_BPW中的“BPW”是“bits per word (每字位数)”的缩写)。在不同的平台上, _NSIG_WORDS的大小从1 (Alpha平台中) 到4 (MIPS平台中) 不等。如你所见, 在x86平台中, 该值正好是2, 这意味着在x86平台上2个无符号数就可以包含足够的位数来代表所有Linux使用的信号。

2. struct sigaction

12165: struct sigaction代表信号到达时进程应该执行的动作。它被封装在struct k_sigaction (12172行) 结构中, 而该结构又是被封装在struct signal_struct 结构中的, 后者是struct task_struct结构的sig成员所指向的一个实例 (16424行)。如果这个指针为空, 进程就会退出而不必接收任何信号。否则, 每个进程对于每个信号数目都需要若干_NSIG struct sigaction结构和一个struct sigaction结构。

12166: sa_handler (__sig_handler_t类型——一个在12148行定义的函数指针类型) 描述了进程希望处理信号的方式。其值可以是下面中的一个:

- SIG_DFL (12151行) 申请处理信号的缺省操作, 不管该操作是什么——这是由信号所决定的。注意它和NULL是等同的。

- SIG_IGN (12153行) 意味着信号应该忽略。但是, 并不是所有的信号都可以被忽略的。
- 所有的其他值都是在信号到达时所需要调用的用户空间函数的地址。

12167: sa_flags进一步调整信号处理代码所完成的工作。可能的标志集合从12108行开始定义。这些标志允许用户代码在信号实例发送以后(或者保留用户定制的操作时)请求恢复缺省操作, 等等。这一点在宏定义块前面的标签注释中已经说明了。

12168: sa_restorer是本书中所没有涉及的一些信号处理代码细节所使用的。

12169: sa_mask是一系列其他信号的集合, 进程在处理这些信号的过程中可能需要进行锁定。例如, 如果一个进程在处理SIGCHLD的时候希望锁定SIGHUP和SIGINT, 进程的第SIGCHLD个sa_mask就会对与SIGHUP和SIGINT相关的位进行置位。

3. siginfo_t

11851: struct siginfo (也称为siginfo_t) 是伴随着信号(特别是在实时信号)所传递的额外信息。

11852: 毋庸置疑, si_signo是信号的数目。

11853: si_errno应该是信号传递时传送者的errno的值, 这样接收者就可以对它进行检测。内核本身并不关心这个值; 当在某些情况下需要设置这个值时, 内核将其设置为0。我推测如果这样, 即使调用者没有设置这个值, 它们仍然会发现si_errno的值被设为已知状态。

11854: si_code记录了信号的来源(不是发送者的进程ID号, 也就是PID——它在别处记录)。有效的信号来源在11915行及其随后部分使用宏进行了定义。

11856: 该结构的最后一部分是union类型的; 该union类型依赖于si_code的值。

11857: union的第一部分是_pad, 它将siginfo_t的长度扩展填充为 $128 \times \text{sizeof}(\text{int})$ 字节(在x86平台上一共是512个字节)。留意一下这个数组的大小, 也就是SI_PAD_SIZE (11849行), 代表了该结构的前三个成员——如果增加了更多的成员, SI_PAD_SIZE就需要进行相应修改。

4. struct signal_queue

17132: struct signal_queue结构用来确保所有的实时信号都被正确传送了, 如果可能, 每一个都包含着额外信息(siginfo_t)。如同后面你将会看到的一样, 内核会为每个进程都设置一个队列, 用来存放该进程的挂起的实时信号。这个队列类型本身很小, 仅仅由一个指向下一个节点的指针和siginfo_t本身组成。

6.2.2 应用函数

有关信号的一个最重要的数据结构是sigset_t, 它是由一系列在include/linux/signal.h文件中定义的简单函数所操纵的, 这些函数的定义从17123行开始。在x86平台上, 这些相同的函数可以——而且已经——使用汇编语言更加有效地实现了; 这些更高效的版本从12204行开始(m68k端口是唯一一个例外的端口, 它使用体系结构特有的代码实现)。由于平台无关的版本和x86特有的版本都很重要, 我们会对两者都加以介绍。

6.2.2.1 平台无关的sigset_t函数

配合sigset_t使用的平台无关的函数在include/linux/signal.h文件中, 从17123行开始。称为

“bitops”（位级的操作）的函数将在后面介绍。

1. sigaddset

17145: sigaddset把一个信号加入集合——也就是说，它修改了集合中的一位。

17147: 为了便于位操作，将基于0的信号转化为基于1的信号。

17149: 如果信号中填入一个无符号长整型数，恰当的位就会被设置。

17151: 否则，sigaddset就需要绕很多弯路，首先装入恰当的数组元素，接着设置该元素中相关位。

17148: 行的代码和该文件中后面的其他代码一样，第一次见到时可能会令人感到有些困惑。在内核代码中，速度是压倒一切的因素。从而，也许你并不会看到类似于下面的运行期决定的代码：

```
if ( _NSIG_WORDS == 1)
    set->sig[0] |= 1UL << sig;
else
    set->sig[sig / _NSIG_BPW]
        |= 1UL << (sig % _NSIG_BPW);
```

而你看到的是类似于下面的在编译期决定的代码：

```
#if ( _NSIG_WORDS == 1)
    set->sig[0] |= 1UL << sig;
#else
    set->sig[sig / _NSIG_BPW]
        |= 1UL << (sig % _NSIG_BPW);
#endif
```

难道这样不会运行的更快些吗？不要忘了，if条件是能够在编译期进行计算的，因此预处理器可以使系统没有必要在运行期执行检测工作。

当你认识到优化工作的实现方式时，这也就没有什么神秘的了。gcc的优化器的敏锐程度足以注意到if表达式只有一个出口，因此它可以把那些不必要的代码移走。作为内核“运行期”版本的结果目标代码和“编译期”的版本是等同的。

但是在我们使用优化器很糟糕的编译器时，基于预处理器的版本还会更好吗？这一点并不确定。问题之一是，基于预处理器的（编译期的）版本有一点更难懂。当代码的复杂程度比前面的简单例子要高时，可读性的差别就会明显地显示出来。例如，让我们考虑一下sigemptyset中的从17264行开始的switch。现在的switch类似于这样：

```
switch ( _NSIG_WORDS) {
default:
    memset(set, 0, sizeof(sigset_t));
    break;
case 2: set->sig[1] = 0;
case 1: set->sig[0] = 0;
    break;
}
```

（请注意经周密考虑的case 2随case 1连续执行的情况。）为了更好地利用预处理器而将其重写，它就可能类似于：

```
#if ( ( _NSIG_WORDS != 2) && \
      ( _NSIG_WORDS != 1))
```

```

    memset(set, 0, sizeof(sigset_t));
#else /* (_NSIG_WORDS is 2 or 1. */
#if (_NSIG_WORDS == 2)
    set->sig[1] = 0;
#endif
    set->sig[0] = 0;
#endif /* _NSIG_WORDS test. */

```

gcc的优化器为两者产生的目标代码是相同的。你更希望读哪一种版本的源程序代码呢？

另外，即使编译器的优化器并没有这么好——这种优化实在相当简单——那么编译器就不可能生成很好的代码。不管我们提供多少帮助都注定是不够的，因此我们可能要编写一些更容易读、更容易维护的代码——这是又一项工程技术的权衡。最后，就像我们在前面的内容中已经看到而且还要不断看到的那样，使用除gcc之外的编译器编译内核本身就是个挑战——增加一段gcc特有代码不会引起更多问题的。

2. sigdelset

17154: 这些代码和sigaddset非常类似；区别在于这里从集合中删去了一位——就是把相应的位设置为关。

3. sigismember

17163: 这些代码和sigaddset也非常类似；这里是要测试某一位是否被设置。注意到17167行可能和下面的这种写法有同样的好处：

```
return set->sig[0] & (1UL << sig);
```

这种写法与17169行非常相似。虽然这样能够和其他函数的编写风格更加一致，但是这并没有什么改进。

这些修改将对函数的行为方式稍有改动：它现在返回0或1，经过这种修改，就可以在一个位被设置时返回其他的非0值。但是，这种改变不会终止没有退出的代码，因为其调用者只关心返回值是否为0（它们并不特别在意是否为1）。

4. sigfindinword

17172: 这个函数返回word中设置的第一个位的位置。函数ffz（在本书中没有涉及）返回其参数中第一个0位的位置。在将位求补的字中的第一个0的位置——这正是这个函数搜寻的内容——显然是原始顺序中的第一个1的位置。它从最小位0开始计算。

5. sigmask

17177: 最后，这个有用的sigmask宏简单地把信号数目通过一个相应的位集合转化为一个位掩码。

6.2.2.2 平台相关的sigset_t函数

即使平台无关的版本已经使用了简单有效的C代码，它也可以通过使用x86 CPU家族的、方便而功能强大的位集指令在x86平台上更加有效地实现。这些函数中的大部分都可以减少为单独的机器指令，因此这里的讨论也都很精简。

在x86平台（例如m68k）上，平台无关的函数对于编译器甚至是不可见的。17126行包含进了asm/signal.h文件，在x86上这个文件被分解为include/asm-i386/signal.h，这都应该归功于

设置文件所建立的符号链接。12202行定义了预处理器符号`__HAVE_ARCH_SIG_BITOPS`，它消除了这些平台无关的函数的定义（请参见17140行）。

1. sigaddset

12204: x86特有的使用`btsl`指令的`sigaddset`实现，它仅对操作数的一个位进行设置。

2. sigdelset

12210: 同样，这是x86特有的使用`btrl`指令的`sigdelset`实现，它对操作数的一个位进行重置（清除）。

3. sigismember

12233: `sigismember`根据其`sig`参数是否是一个编译期常量表达式来选择实现方法。文档中所没有说明的gcc编译器的强大的特殊参数`__builtin_constant_p`是一个编译期操作符（就像`sizeof`一样），它能够报告是否可以在编译期计算其参数值。

如果可以，`sigismember`使用`__const_sigismember`函数（12216行）完成这项工作，因为它的大部分表达式都可以在编译期计算。否则就使用更为普遍的版本`__gen_sigismember`函数（12224行）来代替。更普遍的版本中使用的是x86的`btl`指令，它需要测试其操作数中的某一位。

注意，在编译期的常量合并和死锁代码消除通常意味着这样的完整测试只能在编译期执行——关键是`sigismember`要根据需要使用`__const_sigismember`或者`__gen_sigismember`替换，在作为结果的目标代码中甚至完全看不出来根本就没有对另一部分进行考虑。这样相当精简，难道不是吗？

4. sigmask

12238: x86特有的`sigmask`的实现，这与平台无关的版本是等同的。

5. sigfindinword

12240: 最后，x86特有的`sigfindinword`实现只使用了x86的`bsfl`指令，它在自己的操作数中寻找一个设置位。

6.2.2.3 设置函数

除了前面的那一组函数之外，还有一组对`sigset_t`执行设置操作的函数和宏。和前面一组类似，这些函数使用`__HAVE_ARCH_SIG_SETOPS`预处理器符号保护起来。然而现在没有一种体系结构能够提供自己独有的这些函数的实现，正因为如此，体系结构无关的版本是现存的唯一版本。

1. _SIG_SET_BINOP

17184: 我们希望定义的全部三个二进制操作符——`sigorsets`、`sigandsets`和`signandsets`——的实现方式从本质上来说是相同的。这个宏简单的把这三个的共同代码分解出来，从而只给它们提供一个操作符和一个名字。当然，这同C++模板函数类似，不过这样我们不得不自己处理一些记录工作，而不能完全信任编译器——这是我们使用C工作所付出的一部分代价。

17191: 程序在`sigset_t`中全部四个字节的无符号长整型数的循环开始，同时对这些操作符进行应用。这个循环是由于速度的原因而展开的——通过减少循环，控制开销来提高速度，这是众所周知的一种增加速度的方法。然而，大多数情况下，这个循

环根本就不执行。例如，在x86平台上，编译器可以在运行期就证实不会执行该循环体，因为截断取整以后，`_NSIG_WORDS/4`的结果是0（回忆一下，`_NSIG_WORDS`在x86平台上的值为2）。

17201: `switch`从循环末尾处理剩余工作的这行开始。如果在某些平台上`_NSIG_WORDS`正好为6，那么该循环就可以执行一次，而且`switch`的情况2也可以被执行。在x86平台上，循环永远不会执行；只有`switch`的情况2才可能执行。

顺便说一下，我并不清楚为什么`switch`不和其类似的`_SIG_SET_OP`一样使用直接流程的方式实现。通常情况下，现存的版本可以更充分地利用缓存（如果你试图重新编写它，那么你就可以清楚地认识到这一点）——但是如果实际原因的确如此，那么`_SIG_SET_OP`也应该使用相同的参数。

2. `_SIG_SET_OP`

17238: `_SIG_SET_OP`和`_SIG_SET_BINOP`类似，但是它使用的是一元操作符而不是二元操作符，因此我们并不需要详细地介绍它。但是你应该注意的是，这只能使用一次——在17257行生成`signotset`——这和`_SIG_SET_BINOP`不同。因此，在某种程度上这是不需要的——其实现者可以直接编写`signotset`，而不必借助`_SIG_SET_OP`，这并没有产生任何重复代码。然而，二者生成的目标代码是相同的，这样如果我们以后选择增加一元操作符，意义也就不大了。

3. `sigemptyset`

17262: `sigemptyset`清空所提供的集合——要把其中的每一位都清空。（下面一个函数`sigfillset`和这个函数功能相同，不过它要设置所有的位，而不是清除所有的位，因此我们就不再详细介绍了。）

17265: 普通情况下使用`memset`把集合中的每一位都置为0。

17268: 对于`_NSIG_WORDS`的一些比较小的值来说，简单地直接设置`sigset_t`的一两个元素可能速度更快。在这里采用的就是这种直接流程实现。

4. `sigaddsetmask`

17292: 该函数和下面的几个函数是更简单、快速设置和读取最低的32位（或者根据字的大小）信号的一系列函数。`sigaddsetmask`简单地把`mask`所指定的位置位，而不对剩余的位进行任何处理——这是一个集合的联合操作。

5. `siginitset`

17310: 根据提供的掩码对最低32位（或者是别的）置位，并将其他位设置为0。下面一个函数`siginitsetinv`（17323行）正好相反：它根据掩码的补数设置最低32位（或者是别的），并对其余的位置位。

6.2.3 传送信号

从用户的观点来看，传送信号相当简单：调用系统调用`kill`，该调用只需要进程ID号和一个信号。但是，正如本节中所显示的那样，其实现要复杂得多。

1. `sys_kill`

28768: `sys_kill`是系统调用`kill`的一个具有欺骗性的实现样例；真正的实际工作是在`kill_somethig_info`中实现的。`sys_kill`的参数是要传递的信号`sig`和信号的目的`pid`。

就像你将看到的那样，参数pid并不仅是进程ID（PID和进程的其他概念都在第7章中详细介绍）。

28770: 根据提供给sys_kill的信息声明并填充struct siginfo结构。特别要注意的是si_code是SI_USER（因为只有用户进程才可以调用该系统调用；内核本身是不会调用系统调用的，它更倾向于使用低层函数）。

28778: 传递这些信息给kill_something_info，该函数处理实际的工作。

2. kill_something_info

28484: 该函数的参数和sys_kill类似，但是增加了一项struct siginfo结构的指针。

28487: 如果pid为0，就意味着当前进程希望把信号传递给整个进程组，该工作由kill_pg_info（28408行）完成。

28489: 如果pid是-1，信号（几乎）被送往系统中的每一个进程，这在下面的段落中介绍。

28494: 使用for_each_task宏（在16898行宏定义，第7章中详细介绍）开始循环处理现存进程列表的每一项。

28496: 如果这不是idle进程（或init），就使用send_sig_info（28218行，后面将会讨论）传递信号。每次发现合适的任务时count的值都会增加，虽然kill_something_info并不关心count的实际值，而是在意是否能够发现合适的进程。如果所有试图发送信号的努力都失败了，将记录失败的过程以使得kill_something_info可以在28503行返回错误代码；如果发生了多次错误，则只返回最后一次失败的情况。

28503: 如果发现了合适的候选进程，kill_something_info就返回最近失败的错误代码，或者成功就返回0。如果没有发现任何合适的候选进程，就返回ESRCH错误。

28504: 其他负的pid（是负值，但不是-1）定义了接收信号的进程组；pid的绝对值是进程组号。和前面一样，kill_pg_info的使用就是出于这种目的。

28506: 其他的所有可能性都已经进行了说明；pid必须为正数。在这种情况下，它是信号传送的目的进程的PID。这由kill_proc_info实现（28463行，很快就会讨论）。

3. kill_pg_info

28408: 这个函数给进程组中的每一个进程发送一个信号和一个struct siginfo结构。其函数体和前面介绍的kill_something_info类似，因此我只是简单介绍一下。

28417: 开始循环处理系统中的所有进程。

28418: 如果进程在正确的进程组中，那么信号就发送给它。

28427: 如果信号成功发送给任何进程，retval就设置为0，从而在28430行成功返回。如果信号不能被发往任何进程，那么，要么是所给的进程组中没有进程，在这种情况下，retval仍然会在28415行赋值为-ESRCH；或者kill_pg_info发送信号给一个或多个进程，但是每次都失败了，在这种情况下retval值为从send_sig_info得到的最近错误代码。注意它和kill_something_info的细微区别，后者如果发送信号失败时就返回错误。但是这里的kill_pg_info即使在某些情况下出错了，只要信号能成功地传递给任意进程，就会返回成功信息。

28430: 在28410行中，如果进程组号无效，retval或者是如前所述的赋值，或者就是-EINVAL。

4. kill_proc_info

28463: kill_proc_info是一个相当简单的函数，它把信号和struct siginfo结构传递给由PID定义的单个进程。

28469: 通过所提供的PID查找相应的进程；如果成功find_task_by_pid（16570行）返回一个指向该进程的指针，如果没有找到该进程就返回NULL。

28472: 如果找到匹配进程，就使用send_sig_info把信号传送给目的进程。

28474: 返回错误指示，或者是在28470行由于没有发现匹配进程而返回-ESRCH，或者是其他情况下从send_sig_info中返回的值。

5. send_sig_info

28218: 我们最后看的几个函数中最重要的是send_sig_info。这个函数使用不同的方法装载进程并处理实际的工作。现在应该了解一下实际的工作是如何完成的。send_sig_info将使用info指针（该指针也可能为NULL）指向额外信息的信号sig传送给t指针（调用者应该保证t不会为NULL）指向的进程。

28229: 确保sig在范围之内。注意使用的是如下的测试

```
sig > _NSIG
```

而不是你可能预期的：

```
sig >= _NSIG
```

这是因为信号的计数是从1开始的，而不是从0开始的。因此虽然不存在对这个信号编号的定义，有效信号编号的标识符_NSIG本身也是有效的信号编号。

28233: 这是另外一个严密性检查——实际上包含多个检验。基本的思想是检测信号的传送是否合法。虽然内核本身可以给任何进程传送信号，但是除了在涉及SIGCONT的情况之外，除root之外的用户都不能给其他用户的进程传送信号。总之，这个长的if条件说明了如下问题：

- （28233行）如果不存在补充信息，或者虽然存在补充信息，但是信号来源于用户而不是内核，并且.....
- （28235行）.....信号不是SIGCONT，或者虽然信号是SIGCONT，但是并不是传送给同一会话过程中的其他进程，并且.....
- （28237行和28238行）.....发送者有效的用户ID既不是已经存储了的目标进程的用户ID，也不是目标进程的当前用户ID，并且.....
- （28239行和28240行）.....发送者的当前用户ID既不是已经存储了的目标进程的用户ID，也不是目标进程的当前用户ID，并且.....
- （28241行）.....此处不会允许用户超越普通许可（例如，由于用户是root）.....那么，就不应该发送信号了；可以跳过这段发送信号的代码。

对于前面的if条件必须明白两点。首先，当将info映射为无符号长整型数的时候，如果它为1，这就不是一个实际指向struct siginfo结构的指针。相反的，它是说明信号来自于内核的特殊值，但是并没有进一步的附加信息可供使用。内核本身在最低的页（内存页在第8章中讨论）中并不分配空间，因此在4 096之下的任何地址（除了0，NULL之外）都可以作为这种特殊值使用。

其次，在这几种条件的情况中，位XOR运算操作符（^）比不等运算操作符

(!=) 使用得更为普遍。在这些情况下，两个操作符意义相同，因为如果两个相比较的正数之间有一位不同，在XOR运算的结构中就至少有一位被置位，所以结果非零、逻辑值为真。推测起来，gcc的早期版本为(^)生成的代码比为(!=)生成的代码更为有效，但是在现在的编译器版本中就不是这样了。

- 28248: 忽略信号0并拒绝将信号传送给僵进程（已经退出但是还尚未从系统的数据结构中移走的进程；请参看第7章的“进程状态”一节，它讨论了函数exit）。
- 28252: 对于一些信号，在实际发送之前必须进行一些额外的工作。这些工作是在这里的switch中处理的。
- 28253: 如果正在发送SIGKILL或者SIGCONT，send_sig_info就唤醒进程（也就是说，如果当前被停止了就允许它再次运行）。
- 28257: 设置进程的返回代码为0——如果进程已经使用SIGSTOP停止了，返回代码域就被用来在停止等待的信号和其祖先间建立通讯。
- 28258: 取消任何挂起的SIGSTOP（被调试器阻塞）、SIGSTP（由键盘输入的Ctrl+Z终止）、SIGTTIN（试图从TTY中读取信息的后台运行进程）、SIGTTOU（试图向TTY中写入信息的后台运行的进程）；这些是所有可能中断进程的条件，也是SIGCONT或者SIGKILL最可能作为响应出现的情况。
- 28263: 在一些信号被删除之后，调用recalc_sigpending（16654行，将在后面讨论）来判断是否还有信号仍然处于挂起状态以等待进程。
- 28266: 在前面的情况中，如果SIGCONT或者SIGKILL到达了，这四个信号就会都被取消。但是看起来有些不太对称，如果这四个信号有一个到达了，任何挂起等待的SIGCONT都会被取消。然而SIGKILL却不会被取消，这遵循SIGKILL永远不会被锁定或者取消的规律。
- 28281: 如果目标进程希望忽略信号并且允许不接收信号，那么就跳过了信号的接收过程。
- 28284: 非实时信号并不排队等待，这就意味着如果在进程处理第一个信号实例之前，同一信号的第二个实例就到达了，那么第二个实例就会被忽略。这一点就是在这里确保的（回想一下struct task_struct结构的signal成员中保存着一个进程的当前正在挂起等待的信号集合）。
- 28304: 在限制条件的控制下，实时信号需要排队等待。最重要的限制是可以同时排队等待的实时信号总数的可配置限制；这一限制值为max_queued_signals，它是在28007行定义的，而且可以使用Linux的系统控制特性加以修改。如果有空间来容纳更多的信号，就分配struct signal_queue结构来容纳排队等待的信息。
- 为什么要首先限制排队等待的信号的数目呢？这是为了防止服务拒绝的攻击：如果没有这个限制，用户可以持续发送实时信号直到内核内存溢出，这样就会阻碍内核为其他进程提供该服务及其他服务。
- 28310: 如果一个队列节点已经被分配，现在send_sig_info就必须使有关这个信号的信息进入队列。
- 28311: 把信息加入队列是很直接的：send_sig_info把挂起等待的信号数量（全局变量）增加1，接着把新的节点增加到目标进程的信号队列中。

- 28315: 根据提供给send_sig_info的info参数填充队列节点的info成员。
- 28316: 0 (NULL) 意味着信号是从用户发送而来的, 而且可能使用了从28513行到28544行定义的向后兼容的信号发送函数。目标siginfo_t使用相对比较明确的值来填写。
- 28323: 值1是指示信号来源于内核的一个特殊值——再一次的使用了向后兼容的函数。和前面的情况一样, 目标siginfo_t使用相对比较明确的值来填写。
- 28331: 正常情况下, send_sig_info得到一个实际的siginfo_t, 它可以简单地将其拷贝到队列节点中。
- 28334: 没有分配队列节点——或者因为系统内存溢出而造成kmem_cache_alloc在28306行返回NULL; 或者因为已经达到了信号队列的最大值, send_sig_info根本就没有试图分配节点。不管怎样, send_sig_info所处理的内容是相同的: 除非该信号是通过内核或者老式的信号函数(例如kill)发出的, 否则send_sig_info就返回EAGAIN错误, 通知调用者现在信号不能排队等待, 但是后来调用者应该可以再次使用相同的参数成功执行调用。否则, send_sig_info就传送该信号, 但并不将其排入队列中。
- 28345: 最后, send_sig_info从实际上准备好发送信号。首先, 信号进入该进程的挂起等待的信号的集合中。注意, 即使信号被锁定了, 这个过程也要执行, 这可能有点奇怪。但是这样处理是有原因的: 内核必须提供sys_sigpending (28981行, 本章后面将讨论), 它允许进程查询在锁定时传送进来什么信号。
- 28346: 如果信号没有被锁定, 那么进程应该被通知有信号到达了。相应的, 其sigpending标志被设置。
- 28370: 如果进程正在等待信号的到达并且有信号也正在等待它, 那么这个进程就被唤醒(使用wake_up_process, 26356行)来处理信号。

6. force_sig_info

- 28386: 这个函数被内核用来保证不管进程是否需要, 它都确实接收了一个信号。例如, 在进程释放未用指针时, 可以使用这个函数来确保该进程接收了SIGSEGV(请参见7070行——实际上是调用了向后兼容的函数force_sig, 但是force_sig完全是按照force_sig_info实现的)。force_sig_info的参数和send_sig_info的参数相同, 两者的意义也相同。
- 28392: 如果目标进程是僵进程, 即使是内核也不应该给它发送任何信号; 所进行的尝试将被拒绝。
- 28397: 如果进程将要忽略这个信号, force_sig_info将通过强制它执行缺省操作的方式进行纠正。实际上它并不像外表所表现出来的那样无害: 在内核使用该函数的情况下, 对这个信号的缺省操作是杀死进程。
- 28399: 把信号从t所锁定的集合中移走。
- 28402: force_sig_info现在已经建立了一些条件使得t必须接收信号, 因此该信号就可以使用send_sig_info进行发送。如果send_sig_info的实现改变了, 这仍然可能造成信号不能发送, 因此这两个函数必须保持同步。

7. recalc_sigpending

- 16654: 这个函数重新计算进程的sigpending标志; 当进程的signal或blocked集合改变时就调用该函数。
- 16676: 在最简单的情况中, recalc_sigpending将信号和锁定集合求补的结果执行位AND操作(对锁定集合求补就是允许的集合。)其他的情况仅仅是这种情况的泛化。
- 16679: 如果前面操作中的任何一个在ready中遗留下了任何一位, 那么挂起等待的信号集合中最少有一个信号现在还被锁定; 因此recalc_sigpending将增加sigpending标志的值。

由于recalc_sigpending所实际需要了解的全部内容只是是否至少有一个信号在挂起等待——例如, 如果不止一个, 也并不需要知道有多少信号在挂起等待——非平凡情况下的代码只要发现ready的值被置为非0值就应该停止对其进行修改(例如, 前面16662行通过中断循环)。但是, 任何可能来对此优化所产生的效率增进都必须同为此而进行的额外测试进行权衡。正是由于这个原因, 又加上_NSIG_WORDS很小(在实际中无论如何都是如此), 改进的版本可能要比标准情况快一点。

8. ignored_signal

- 28183: ignored_signal有助于send_sig_info决定是否给一个进程发送信号。
- 28189: 如果进程正被其祖先跟踪(可能是调试器), 或者信号是在进程锁定的集合中, 那么它就不能被忽略。第二种情况可能是我们过去所没有考虑过的; 如果信号被锁定了, send_sig_info(还有ignored_signal)难道不应该将其忽略吗? 如果情况的确如此, 还真不应该忽略。这个函数通过信号是否应该被忽略, 表明了进程的信号的signal集合的相应位是否应该被置位。如同前面我们已经看到的那样, 对sigpending系统调用的支持要求如果在锁定过程中有信号到达, 内核就应该设置相应的位。因此, 被锁定信号不能简单地忽略。
- 28194: 如果进程是一个僵进程, 信号就应该被忽略。这种测试是不必要的, 因为这种情况甚至在28248行的ignored_signal调用之前就会被发现。
- 28199: 在大多数情况下, SIG_DFL(缺省的)操作是处理信号而不是将其忽略。你所能看到的例外是SIGCONT、SIGWINCH、SIGCHLD和SIGURG。
- 28207: 进程允许忽略大部分信号, 但是不能忽略SIGCHLD。对于SIGCHLD, POSIX赋予SIG_IGN一种特殊的意义, 这一点在28831行将会说明。这里所提到的“automatic child reaping”(自动子进程空间回收)将在3426行执行。
- 28211: 在缺省的情况下, 可以假定ignored_signal有一个实际的函数指针, 而不是SIG_DFL或者SIG_IGN两个伪值的一个。这样, 信号就和用户定义的处理句柄联系起来, 这意味着进程希望处理这个信号。它通过返回0来指明信号不应该被忽略。

9. do_signal

- 3364: do_signal在信号到达进程时使用。这个函数在内核中被调用的地方不止一次——如我们在第5章中看到的从203行和211行, 还有从2797行和2827行。通常所有这些情况都是当前进程希望处理挂起等待的信号(如果有的话)。
- 3375: 如果非空, oldset用来返回当前进程锁定的信号集合。由于do_signal不会修改锁定的集合, 它可以简单的返回一个指向现有锁定集合的指针。

- 3378: 进入几乎扩展到该函数末尾 (3478行) 的循环。退出该循环的方法只有两种: 把所有可能的信号都处理了, 或者处理唯一一个信号。
- 3382: 使用`dequeue_signal`使信号出队列 (28060, 后面将会介绍)。`dequeue_signal`或者返回0, 或者返回需要处理的信号的编号, 并且它还会填充`info`中的附加信息。
- 3385: 如果没有信号处于等待状态, 将在这里中断循环。正常情况下, 它在循环第一次执行过程中是不会发生的。
- 3388: 如果当前进程正在被其祖先跟踪 (可能是调试器), 而且信号也并不是不可锁定的SIGKILL, 那么在信号到达之前, 进程的祖先就必须已经得到通知了。
- 3391: 将传递给子孙进程的信号编号传送到祖先进程中对应子孙进程的`exit_code`域; 祖先使用`sys_wait4`收集这些信息 (23327行, 在第7章中介绍)。`do_signal`停止子孙进程的运行, 然后使用`notify_parent` (3393行) 给祖先进程发送SIGCHLD信号, 接着调用调度函数`schedule` (26686行, 第7章中介绍), 给其他进程——尤其是其祖先进程——运行的机会。`schedule`会把CPU分配给其他进程, 因此直到内核跳转回这个进程才会返回。
- 3397: 如果调试器取消了信号, `do_signal`在这里就不应该处理它; 循环继续进行。
- 3402: SIGSTOP可能只是由于进程正在被跟踪而产生。这里没有必要处理它; 循环继续进行。
- 3406: 如果调试器修改了`do_signal`要处理的信号编号, `do_signal`将根据新的信息填充`info`。
- 3415: 正如注释中所说明的一样, 如果新的信号被锁定了, 就需要重新排队, 循环继续进行。否则, 控制流程将直接执行下面的代码。
- 3421: 在这里, 或者进程未被跟踪, 或者进程正被跟踪, 但是得到了一个SIGKILL信号, 或者控制流程直接从前面的代码块中执行下来。在任何一种情况中, `do_signal`都有一个应该现在处理的信号。它从获取`struct k_sigaction`结构开始, 这个结构指明了怎样处理这个信号编号。
- 3423: 如果进程试图忽略信号, 那么除非这个信号是SIGCHLD, 否则`do_signal`就继续执行循环从而忽略该信号。为什么这个测试不能同时保证该进程不会忽略掉SIGKILL这个注定不可忽略也不可锁定的信号呢? 答案在于和SIGKILL相关的操作永远不会是SIG_IGN的, 实际上也不会是除SIG_DFL之外的任何操作——28807行就保证了这一点 (在`do_sigaction`函数中)。这样, 如果操作是SIG_IGN, 那么信号编号就不可能是SIGKILL。
- 3426: 如同在从28820行开始的标题注释中说明的一样, POSIX标准说明了忽略SIGCHLD的操作实际上意味着自动回收其子孙进程。子孙进程是通过使用`sys_wait4`来回收的 (23327行, 在第7章中介绍), 此后循环继续运行。
- 3435: 进程为这个信号采用缺省操作。专用的初始化进程接收到全部信号所对应的缺省操作是把信号整个忽略掉。
- 3439: 对信号SIGCONT、SIGCHLD和SIGWINCH所采取的缺省操作是不加处理, 只是简单地继续执行循环。
- 3442: 对于信号SIGSTP、SIGTTIN和SIGTTOU, 缺省的操作各自不同。如果该进程所归

属的进程组是孤立的——简单地说就是没有连接到TTY上——那么POSIX规定对于这些基于终端的信号缺省操作是将其忽略。如果进程的进程组不是孤立的，缺省的操作是停止进程的运行——这和SIGSTOP的情况是相同的，在这种情况下，控制流程直接向下运行。

3447: 在对SIGSTOP的响应中（或者是从前面情况中直接执行下来），do_signal终止了进程。另外，除非祖先进程已经规定对其子孙进程的终止运行不加理会，否则祖先进程将会在其子孙进程退出时被通知。和3394行一样，调用schedule交出CPU给其他某一进程。当内核把CPU再次分配给当前进程的时候，该循环继续运行以处理队列中的另外一个信号。

这不是我们希望的——我认为当schedule返回时，循环应该退出，因为信号已经处理完了。其原理在于如果进程被终止了，唤醒进程的最可能的原因是进程又得到了信号，可能是SIGCONT，因此该进程现在就可以检测并处理信号了。

3456: 对于其他信号的缺省操作是退出进程。它们中的一些将使进程首先清空内核（详细的介绍请参看第8章），这些信号就是SIGQUIT、SIGILL、SIGTRAP、SIGABRT、SIGFPE和SIGSEGV。如果此二进制格式（详细的介绍请参看第7章）知道如何清空内核并且成功地清空了内核，那么在进程的返回代码中就会有一位被设置来指明进程在退出之前就已经清空了内核。接着流程按照default的情况继续执行，终止进程的运行。注意do_exit（23267行，在第7章中也会有介绍）是从来不会返回的——因而在3471行中会有“NOTREACHED”注释。

3476: 此处，do_signal从队列中取出一个信号，该信号既不和SIG_IGN的操作有关，也不和SIG_DFL的操作有关。唯一的另外一种可能性是：这是用户定义的信号处理函数。do_signal调用handle_signal（3314行，本章随后将会讨论）来触发这个信号处理函数，接着返回1向调用者声明这个信号已经处理过了。

3481: 此处，do_signal不能为当前进程从队列中取出信号（只有从3386行的break退出时才能执行到本行）。如果在系统调用的处理过程中被中断了，do_signal就要调整寄存器，从而系统调用将可以重新执行。

3490: 返回0以通知调用者do_signal没有处理任何信号。

10. dequeue_signal

28060: dequeue_signal将信号从进程信号队列中移出，同时忽略那些由掩码说明的信号。它返回信号的编号并使用指针参数info返回相关的siginfo_t。

28071: 为了避免重复的非关联化，建立一些别名：s是进程的挂起等待的信号的集合（记住其中可能包括了一些锁定的信号），m是掩码的集合。特别要注意的是*s表达式，在该函数中这个表达式出现了不止一次，但是它只是current->signal.sig[0]的一种简单写法。

28073: 在这个switch条件分支中，sig被设置为第一个挂起等待的信号。从最简单的情况入手最容易理解；其他的情况只是这种情况的泛化。

28091: 最简单的情况是：它把挂起等待的信号和掩码求补后的结果进行位AND运算，结果被存储在临时变量x中；x现在就是掩码不能忽略的挂起等待的信号的集合。如果x不为0，那么就存在挂起等待的信号（x至少有一位被置位）；dequeue_signal

使用ffz（本书中没有涉及）取得相应的信号编号，并将其转化为从1开始计数的信号编号，将结果存储在sig中。正如前面所说明的一样，其他情况只是这种情况的泛化；最重要的结果是sig被置位，如果可能的话在每种情况下都是如此——此后其他变量（i、s、m或者x）的状态就不难理解了。如果在switch之后的sig是0，掩码中就没有传递挂起等待的信号。

- 28097: 如果一个信号正在挂起等待，那么`dequeue_signal`应该试图将其从队列中释放出来。`reset`跟踪`dequeue_signal`以判定是否应该把信号从进程的挂起等待的信号队列中删除。将`reset`初始化为1仅仅是由于在函数处理过程中它可能会改变的假定。
- 28107: 对于非实时信号，内核不会保持原始的`siginfo_t`（如果曾经有过的话），因此`dequeue_signal`应该尽可能重新组织有关的信息。不幸的是，当前实现方法中并没有多少信息——只有信号编号自身而已。`info`的其他成员都简单地被设置为0。
- 28118: 在另一种情况，也就是实时信号情况下，`siginfo_t`只是一种点缀。`dequeue_signal`会在进程的`sigqueue`中进行扫描以确定其值。
- 28122: 如果找到了`siginfo_t`，`dequeue_signal`现在就使其出队列，将`siginfo_t`的内容拷贝到`info`中，并释放为这个队列节点分配的内存。
- 28129: 如果队列中没有这个信号的更多实例，那么信号就不会再挂起等待了。但是为了弄清楚队列中是否还有信号的实例，`dequeue_signal`需要遍历整个队列。因此，该函数需要扫描这个队列的其余元素来查询是否存在相同信号的其他实例。如果发现了实例，`dequeue_signal`就清空`reset`标志——只有在这种独特的情况下才会进行的操作。
- 28142: 正在出队列的信号是实时信号，但是在进程的挂起等待的实时信号队列中却没有发现它，其原因在代码中已经进行了阐述。现在，`dequeue_signal`和它有非实时信号的情况相同了——它知道信号是可以访问的，但是没有方法可以访问其原始值——并且其响应过程处理的工作也完全相同，仅仅使用信号编号来填充`info`，而没有其他属性值。
- 28150: 除非`reset`标志被清空了——也就是说除非这是一个实时信号并且同一个信号的其他实例仍然在挂起等待队列中，该信号已经被处理过；它应该从进程的挂起等待集合中删除。
- 28152: 信号脱离队列，因此`dequeue_signal`应该重新计算进程的`sigpending`标志。我认为这里有一个可以进行少量优化的机会：只用当`reset`为真的时候`dequeue_signal`才需要这样处理。`recalc_sigpending`从进程的锁定集合和挂起等待集合中计算结果；锁定的集合没有改变，因此只有当挂起等待的集合发生改变时，`dequeue_signal`才需要调用`recalc_sigpending`。如果`reset`为假，挂起等待的集合就不会改变，因此对于`recalc_sigpending`的调用就是不必要的。
- 28163: `switch`没有发现信号，因此没有信号正在挂起等待。作为内部正确性的检测，`dequeue_signal`确保内核不会认为有信号正在为某任务挂起等待。
- 28174: 返回出队列的信号编号，或者如果没有信号出队列，就返回0。

`notify_parent`

- 28548: `notify_parent`寻找进程的祖先进程并通知它其子孙进程的状态发生了改变——通

常情况是其子孙进程或者被终止了，或者被杀死了。

- 28553: 使用有关信号发生的上下文的信息填充局部变量info。
- 28564: 如果子孙进程已经退出，why被赋予适当的值以指明其原因，是因为它清空了内核，或者被某信号将其杀死，或者因为执行了非法操作。
- 28572: 同样，如果使用信号终止了进程，对why赋值以说明发生的情况。
- 28578: 前面的情况几乎覆盖了所有的可能性。如果没有，函数打印出警告信息并继续运行；在这种情况下，系统会在28562行将why的值赋为SI_KERNEL。
- 28586: 给进程的祖先进程发送信号。下面一行唤醒任何等待这个子孙进程的进程，并为其提供CPU。

handle_signal

- 3314: handle_signal在需要调用用户定义的信号处理程序时由do_signal调用。
- 3338: 建立一个用户处理程序可以在其中运行的堆栈帧。如果进程已经请求了内核所拥有的有关信号的原始值和其上下文的附加信息，那么堆栈帧就使用setup_rt_frame (3231行) 构建起来；否则就使用setup_frame (3161行) 构建。这两种方法都可以实现构建工作，这样控制流程会返回信号处理程序。当它返回时，实际返回的是信号到达的时候正在执行的代码。
- 3343: 如果SA_ONESHOT标志被设置，则信号处理程序应该只执行一次（注意sys_signal是signal系统调用的实现，它使用SA_ONESHOT类型的信号处理程序——请参看29063行）。在这种情况下，缺省的操作是立即将其恢复。
- 3346: SA_NODEFER意味着在执行这个信号的处理程序时，不应该有其他信号被锁定。如果位没有设置，其他的位现在就会被加入进程的锁定的集合中。

6.2.4 其他有关信号的函数

其他一些有关信号处理的函数。

1. sys_sigpending

- 28981: 这个简短的系统调用允许进程询问在信号锁定期间是否有非实时信号到达。通过所提供的指针，该函数返回一个位集以指明它们是哪些信号。
- 28987: 这个函数的核心是进程的blocked集合和signal集合间的简单位AND操作。它只对最低32位感兴趣，这些都是非实时信号。
- 28992: 使用所提供的指针把挂起等待的集合拷贝回用户空间。如果失败就返回-EFAULT，如果成功就返回0。注意是否有信号正在挂起等待——也就是说，返回值是否为空——并不是成功的判据之一。

2. do_sigaction

- 28801: do_sigaction实现了系统调用sigaction有意义的部分（其余部分在2833行的sys_sigaction中）。sigaction是POSIX中等价于ISO C的函数signal——它把信号和操作关联起来，这样进程接收到信号时就能够执行相应的操作。
- 28806: 稳健性检测：确保sig在范围之内并且进程没有试图把SIGKILL或者SIGCONT和某种操作相关联。进程被简单地剥夺了覆盖这两个信号的缺省操作的权力。然而，和signal实现的处理程序不同，使用sigaction实现的处理程序不是SA_ONESHOT类

型的，因此在处理程序被调用的时候就不用每次都将其重新装载。

- 28811: 获取和这个信号相关的指向struct k_sigaction结构的指针。
- 28813: sigaction可以通过一个过去所提供的指针返回旧有的操作。这在以堆栈方式存在的处理程序中是很有用的，在这里处理程序被临时覆盖，以后再恢复出来。如果oact指针非空，旧有的操作就会被拷贝到其中（但是这并不会把信息拷贝回用户空间；调用者必须执行这样的处理）。
- 28815: 如果do_sigaction被赋予一个需要同信号相关联的操作，那么二者现在就相互关联起来。SIGKILL和SIGSTOP也必须被从操作的掩码中删除，以确保这些信号不会被锁定或者覆盖。
- 28836: 正如在28820行开始的标题中注释的一样，为了遵守POSIX标准，下面的几行代码必须要经过一定变形，并且在必要情况下还会舍弃某些信号。对于这些细节情况，我们即使跳过也不会有什么损失。

3. sys_rt_sigtimedwait

- 28694: sys_rt_sigtimedwait等待信号的到达，它可能在经过一段特定的时间间隔以后超时退出。并不是所有的信号都会被接收；指针uthese所指明的sigset_t说明了调用者所感兴趣的信号。
- 28714: uthese（它已经被拷贝到局部变量these中了）是允许的信号的集合，于是内核原语只知道如何锁定信号。但这样也没有关系：对允许的信号集合进行求补运算就得到了应该锁定的信号，所得到的结果就可以直接使用了。
- 28717: 如果调用者提供了超时时间，该超时时间将被拷贝到用户空间中，而且其值也必须经过稳健检测。
- 28726: 检查是否已经有信号正在挂起了——如果有，就没有必要为其等待了。否则，调用者必须等待。
- 28731: 保存原来的锁定信号集合，然后阻塞由these定义的所有信号。
- 28737: 如果用户没有提供超时时间，那么超时时间会是MAX_SCHEDULE_TIMEOUT（宏定义为LONG_MAX，或者是 $2^{31}-1$ ，16228行）。但是并不永远都是如此——超时时间是以瞬间（jiffy）计数的，它的系统时钟以每秒100次的速度跳动着，因此大约有248天，超时时间就耗尽了（在64位机器中，这大约需要三十亿年）。
- 28739: 如果用户确实提供了超时时间，就将其转化为以瞬间计算的值。“+”后面的表达式是对下一个瞬间进行向上舍入的明智方法——其思想是timespec_to_jiffies可能已经向下舍入了，但是内核必须是上舍入的，因为它必须等够用户请求的瞬间个数。它虽然可以检测timespec_to_jiffies（18357行）是否是下舍入的，但是下面这种方法更为简单：如果用户提供的超时时间不是0，就为其增加一个瞬间，并且认为是对它进行了调整。毕竟Linux不是一个真正的实时操作系统——当你指定了超时时间时，Linux只能保证至少等待如此长的时间。
- 28742: 设置当前用户的状态为TASK_INTERRUPTIBLE（请参看第7章）。schedule_timeout（26577行）用来让出CPU；在指定的时间用完以后或者其他事件到达并唤醒进程（比如接收了一个信号）时，该进程才可以继续运行。
- 28746: 进程希望被信号唤醒。sys_rt_sigtimedwait再次尝试从进程的等待信号队列中取

出信号，并恢复原来锁定的集合。

- 28752: 此处，该函数仍然不知道信号是否已经到达了——它可能无需等待就可以得到一个信号，或者在等待期间可能有另一个信号到达，又或者该函数一直在等待，但是没有信号到达。
- 28753: 如果信号到达，该函数就给用户进程传递信息，并且返回信号编号。
- 28759: 否则，虽然进行了等待，但是没有信号到达。在这种情况下，该函数或者返回-EAGAIN（说明用户进程可以再次使用相同的参数尝试），或者返回-EINTR（说明其等待过程被由于某些原因而不能传递的信号中断了）。

6.2.5 内核如何区分实时信号和非实时信号

简单地说，答案并不复杂。我几乎掩盖了其中的绝大部分区别，这是有一定原因的：退出语句不多。现在，为了使这一点更加清楚，让我们来看一下系统调用sigprocmask的两个版本，它允许进程处理自己的锁定信号的集合——增加，删除，或者简单地对信号集合进行设置。

1. sys_sigprocmask

- 28931: sys_sigprocmask是这个函数的原始版本，这一版本并不知道或者是不关心实时信号。参数how指明了要执行的操作；如果set不为NULL，就是这个操作的操作数；如果oset是非空的，那么oset返回的就是原始的锁定集合。
- 28937: 如果set为空，那么how的值就没有什么用处了：该操作就没有操作数了，因此该函数不会处理有关的内容。否则，就继续执行该操作。
- 28939: 在新的锁定集合中的拷贝，其中删除了不可锁定的SIGKILL和SIGSTOP。
- 28944: 为了避免以后将当前锁定集合拷贝回用户空间，在old_set中存储当前锁定集合的一个备份。由于当前锁定集合在以后的代码中可能会被修改，因此在它改变之前必须对其值进行存储。
- 28948: 当然是忽略无效的操作。
- 28951: SIG_BLOCK操作符指明new_set应该解释为要锁定的附加信号的集合。这些信号将被加入该锁定集合中。
- 28954: SIG_UNBLOCK操作符指明new_set应该解释为要从锁定的信号的集合移出的信号集合。这些信号现在被移出锁定集合。
- 28957: SIG_SETMASK操作符指明new_set应该解释为新的锁定集合，简单覆盖该锁定集合原有的值。因此，sys_sigprocmask正是实现这一点的。注意它只设置了blocked.set数组的最低的元素——这个元素包含低32位非实时信号，这是该函数所关心的内容。
- 28966: 如果调用者已经请求查询锁定集合的原来的值，执行流程就向前跳到set_old标号（28970行）。
- 28968: 如果set为空，意味着调用者没有请求对锁定集合进行修改，但是调用者可能仍然希望了解锁定集合的当前值。
- 28972: oset非空（set也可能为非空）。不管哪一种情况，old_set都包含一个原来锁定的集合的备份，在返回之前，sys_sigprocmask会试图将其拷贝回用户空间。

2. sys_rt_sigprocmask

28612: `sys_rt_sigprocmask`和`sys_sigprocmask`非常类似,但是它也能够处理新的实时信号。

由于这两者之间的相似性,在这里我仅仅介绍一下它们之间的区别。

28638: 替代的代码类似于:

```
/* How sys_sigprocmask does SIG_BLOCK.
 */
new_set = *set;      /* Line 28939 */
blocked |= new_set; /* Line 28952 */
```

(作为采用SIG_BLOCK情况的一个例子),实际代码类似如下代码:

```
/* How sys_rt_sigprocmask does
 SIG_BLOCK. */
new_set = *set;      /* Line 28625 */
new_set |= old_set; /* Line 28639 */
blocked = new_set;  /* Line 28648 */
```

我不明白为什么`sys_rt_sigprocmask`不使用和`sys_sigprocmask`相同的方式实现,而且这样还可以节约一点效率。

6.3 中断

中断的名字十分形象,因为它们终止了系统正常的处理过程。在前面第5章中就已经看到了中断的例子:提供系统调用基本机制的软件中断。在本章中,我们来了解一下硬件中断。

和系统调用中断一样,硬件中断也可能转化为内核模式,然后返回。如果用户进程运行时发生了中断,系统就转化为内核模式,并且内核要对中断做出响应。接着,内核将控制返回给用户进程,用户进程能够从当时离开的位置继续运行。

同系统调用中断的另一个区别是,硬件中断可能在内核已经在内核模式下运行时发生。这在系统调用中很少发生——通常内核不会麻烦地触发系统调用中断,因为它可以直接调用目标内核函数。如果中断发生时系统处于内核模式,结果就同在用户模式下的机制相一致——唯一的区别是内核自身所特有的执行过程,而不是用户进程的执行过程暂时被中断。

如果内核在一段时期内不希望被中断,那么就可以使用`cli`和`sti`函数(13105行和13104行,UP版本;1216行和1229行,SMP版本)屏蔽和开启中断。这些函数根据底层的x86指令命名:`cli`代表“清除中断标志”,`sti`代表“设置中断标志”。其工作方式和其名称类似:CPU有一个“中断允许”标志,如果对其置位就允许中断,如果将其清空就禁止中断。因此,你可以使用`cli`清空这个标志从而禁止中断,也可以使用`sti`设置这个标志从而允许中断。在UP代码中,你可以选择调用两个等价的宏`__cli`和`__sti`——分别见13105行和13104行。

当然,把内核移植到非x86平台上会使用不同的底层指令——在这些体系结构中`cli`和`sti`函数的实现都不相同。

6.3.1 中断请求: IRQ

IRQ,或者中断请求,是从硬件设备发往CPU的中断信号。作为对IRQ的响应,CPU跳转到某个地址——中断服务例行程序(ISR),更普通的情况是调用中断处理程序——内核在前面已经对这些处理程序进行了登记。中断处理程序是内核执行的为中断服务的函数;从中断处理程序中返回就继续执行中断前所在位置的代码。

IRQ是有编号的，每一个硬件设备在系统中都对应一个IRQ号码。例如在IBM PC体系结构中，IRQ 0就关联着一个每秒产生100次中断的定时器。把IRQ号码和设备关联起来，使得CPU可以区分每个中断是哪个设备产生的，从而允许它跳转到正确的中断处理程序。（在某些情况中，在一个系统中一个IRQ号可以被多个设备所共用，当然这不是非常普遍的情况。）

6.3.2 下半部分

中断处理程序的下半部分（bottom half）是无须立即执行的部分。在某些中断之后，你甚至可能根本就不需要执行它。

给定的中断处理程序从概念上可以被分为上半部分（top half）和下半部分；在中断发生时上半部分的处理过程立即执行，但是下半部分（如果有的话）却推迟执行。这是通过把上半部分和下半部分处理为独立的函数并对其区别对待实现的。总之，上半部分要决定其相关的下半部分是否需要执行。不能推迟的部分显然不会属于下半部分，但是可以推迟的部分只是可能属于下半部分。

你也许会很奇怪为什么Linux会辛苦地把它们区分开——为什么要延迟呢？一个原因是要把中断的总延迟时间最小化。Linux内核定义了两种类型的中断，快速的和慢速的，这两者之间的一个区别是慢速中断自身还可以被中断，而快速中断则不能。因此，当处理快速中断时，如果有其他中断到达——不管是快速中断还是慢速中断——它们都必须等待。为了尽可能快地处理这些其他的中断，内核就需要尽可能地将处理延迟到下半部分执行。

另外一个原因是，在最低层，当内核执行上半部分时，中断控制芯片将被告知禁止正在服务的这个特殊IRQ（这和CPU级别的中断禁止不同，它把快速中断和慢速中断区别开来）。我们并不希望这种状态会持续的比需要的时间还长，因此只有上半部分中时间最为关键的部分才被处理，但是下半部分中其他的工作就要延迟处理了。

区分上下部分还有一个原因是处理程序的下半部分包含有一些中断所不一定非要处理的操作，只要内核可以在一系列设备中断之后能从某些地方得到。在这种情况下，执行对于每个中断的下半部分的处理完全是一种浪费，它可以稍稍延迟并在后来只执行一次。

最后一段的一个暗示是值得说明的：没有必要每次中断都调用下半部分。相反，是上半部分（或者也可能是其他代码）简单地标记下半部分，通过设置某一位来指明下半部分必须执行。如果下半部分已经标记过需要执行了，现在又再次标记，那么内核就简单地保持这个标记；当情况允许的时候，内核就对它进行处理。如果在内核有机会运行其下半部分之前给定的设备就已经发生了100次中断，那么内核的上半部分就运行100次，下半部分运行1次。

下半部分在内核中有时被认为是“软IRQ”或者“软中断处理程序”，这有助于你理解今后要遇到的一些文件名和术语。

在本节的剩余内容中，我们将保持下半部分抽象的概念。下一节深入介绍定时器中断，包括其下半部分的处理，并展示了下半部分概念的一个有趣的滥用现象——我的意思是一个有趣的变种。

6.3.3 数据结构

同对信号的处理一样，我们首先介绍一下中断和下半部分使用的重要的数据结构。图6-1

阐述了这些数据类型之间的关系。

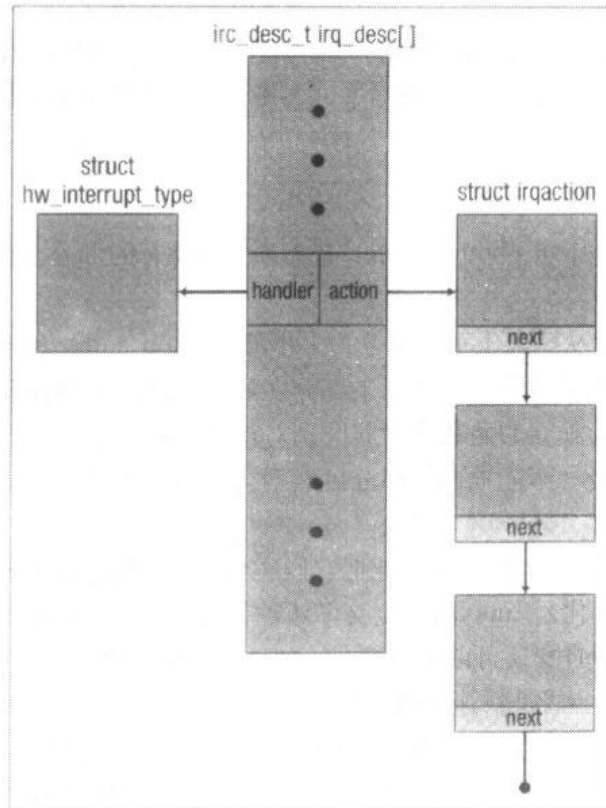


图6-1 有关中断的数据结构

我们从这里开始，体系结构无关的头文件linux/interrupt.h定义了struct irqaction结构（14844行），它代表了内核接收到特定IRQ之后应该采取的操作（在本章后面将看到struct irqaction结构是如何与IRQ关联的）。其成员如下：

- handler——指向某一函数的指针，该函数是作为对中断的响应所执行的操作。
- flags——从与前面已经介绍过的sa_flags相同的集合中提取出来；这个集合从12108行开始。该集合中仅仅为此目的而出现的值只有SA_INTERRUPT（使用另外一个中断来中断这个中断也是可以的），SA_SAMPLE_RANDOM（考虑到这个中断也是源于物理随机性）和SA_SHIRQ（这个IRQ和其他struct irqaction共享）。
- mask——在x86或者体系结构无关的代码中不会使用（除非将其设置为0）；看起来只有在SPARC64的移植版本中要跟踪有关软盘的信息时才会使用它。
- name——生成中断的硬件设备的名字。由于不止一个硬件可以共享一个IRQ，这在打印人工阅读程序时就有助于区分它们。
- dev_id——标识硬件类型的一个唯一的ID——Linux支持的所有硬件设备的每一种类型都有一个由制造厂商定义的在此成员中记录的设备ID。其所有的可能值都是从一个巨大的集合中抽取出来的，这个集合在本书中没有介绍，因为它包含的内容是十分繁琐的，而且都是重复的——它仅仅是结构上类似于下面一小段代码的巨大宏定义块。

```
#define PCI_DEVICE_ID_S3_868 0x8880
#define PCI_DEVICE_ID_S3_928 0x88b0
```

```
#define PCI_DEVICE_ID_S3_864_1 0x88c0
#define PCI_DEVICE_ID_S3_864_2 0x88c1
```

在看完这些部分之后，也就相当于将其完整地看了一下。可能你已经发现了，摘录的这部分内容是从包含针对基于S3的PCI显卡的设备ID的文件中选取的。

虽然dev_id是一个指针，可它并不指向任何内容，但若不关联化就会引起错误。能够说明问题的是它的位结构模式。

- next——如果IRQ是共享的，那么这就是指向队列中下一个struct irqaction结构的指针。通常情况下，IRQ不是共享的，因此这个成员就为空。

接下来我们感兴趣的两个数据结构存在于体系结构相关的文件arch/i386/kernel/irq.h中。第一个是struct hw_interrupt_type结构（1673行），它是一个抽象的中断控制器。这是一系列的指向函数的指针，这些函数处理控制器特有的操作：

- typename——赋给控制器的人工可读的名字。
- startup——允许从给定的控制器的IRQ所产生的事件。
- shutdown——禁止从给定的控制器的IRQ所产生的事件。
- handle——根据提供给该函数的IRQ处理唯一的中断。
- enable和disable——这两个函数基本上和startup和shutdown相同；存在的差异对于本书中涉及的代码都不很重要（实际上，对于本书中包含的所有代码来说，enable/disable函数对和startup/shutdown函数对都是相同的）。

这个文件中我们感兴趣的另外一个数据结构是irq_desc_t（1698行），它具有如下成员：

- status——一个整数，它的位或者为0，或者对应从1685行到1689行定义的集合中抽取出的标志。这些标志的集合代表了IRQ的状态——IRQ是否被禁止了，有关IRQ的设备当前是否正被自动检测，等等。
- handler——指向struct hw_interrupt_type的指针。
- action——指向struct irqaction结构组成的队列的头。如同前面说明的一样，正常情况下每个IRQ只有一个操作，因此链接列表的正常长度是1（或者0）。但是，如果IRQ被两个或者多个设备所共享，那么这个队列中就有多个操作了。
- depth——irq_desc_t的当前用户的个数。主要是用来保证事件正在处理的过程中IRQ不会被禁止。

irq_desc_t是在irq_desc数组中（733行）积聚起来的。对于每一个IRQ都有一个数组入口，因此数组把每一个IRQ映射到和它相关的处理程序和irq_desc_t中的其他信息上。

最后一个需要说明的数据结构集合从29094行开始；这些都与前面所讨论的下半部分有关：

- bh_mask_count（29094行）——跟踪为每个下半部分提出的enable/disable请求嵌套对的数组。这些请求通过调用enable_bh（12575行）和disable_bh（12568行）实现。每个禁止请求都增加计数器；每个使能请求都减小计数器。当计数器达到0时，所有未完成的禁止语句都已经被使能语句所匹配了，因此下半部分最终被重新使能。
- bh_mask和bh_active（14856行和14857行）——它们共同控制下半部分是否运行。它们两个都有32位，而每一个下半部分都占用一位。当一个上半部分（或者一些其他代码）决定其下半部分需要运行时，就通过设置bh_active（12498行中使用mark_bh）中的一

位来标记下半部分。不管是否经过了这样的标记，下半部分可能会通过清空bh_mask中的相关位来整个跳过——通过调整bh_mask_count入口，enable_bh和disable_bh完成了这个功能。

因此，对bh_mask和bh_active进行位AND运算就能够表明应该运行哪一个下半部分。特别是如果位与运算的结果是0，就没有下半部分需要运行。这种技术在内核中多次使用，例如在宏get_active_bhs（12480行）中就使用了这种技术

- bh_base（14858行）——这是一组简单的指向下半部分函数的指针。
- 未命名的enum——从14866行开始的未命名的enum为内核使用的每一个下半部分指定了一个符号名称。例如，为了把计数器的下半部分标记为活动的，你可以这样写语句：

```
mark_bh (TIMER_BH);
```

27450行的确就是这样处理的。

6.3.4 操作和IRQ

一个经过仔细选择的小型函数集合处理了操作和IRQ之间的链接和解除链接。本节就是来讨论这些函数及那些从整体上对IRQ系统进行初始化的函数。

1. init_IRQ

1597: init_IRQ初始化IRQ的处理。

1601: 符号CONFIG_X86_VISWS_APIC是为SGI虚拟工作站及SGI的基于x86的工作站流水线而设置的。虽然同样基于x86的CPU，虚拟工作站不能和基于IBM PC的体系结构共享很多其他特性——特别是如同你看到的，它们的中断处理有些不同。我们以后将忽略虚拟工作站所特有的代码。

1609: 建立中断描述符表，给32项到95项（十进制）赋缺省值。在这个过程中使用了set_intr_gate（6647行），该函数很快就会介绍到。

1651: 建立IRQ 2（级联中断）和IRQ13（为FPU使用——请参看955行）。和这两个IRQ有关的irqaction结构分别是irq2（979行）和irq13（974行）。

2. init_ISA_irqs

1578: 该函数填充irq_desc数组，为ISA总线类型的机器（也就是所有标准PC）初始化所有IRQ。虽然该函数没有声明为static类型的，也没有使用__initfunc标签标记，但是它只会被init_IRQ调用。因此，只有在内核初始化过程中这个函数才是必要的。

1583: 对irq_desc中的每一个元素，系统为status、action和depth成员赋予了不会惹人反对的、也不会使人吃惊的缺省值。

1589: 原来的（在PCI之前）IRQ使用i8259A_irq_type（723行）处理。

1592: 编号比较高的IRQ初始化为no_irq_type（701行），这是一个必要的空处理程序。后来它们可能会改变——实际上，如果你使用了PCI卡，就确实会改变，就像现在的大多数PC一样。

4. set_intr_gate

6647: set_intr_gate在x86CPU的中断描述符表（IDT）中建立一个项。在基于x86的系统中发生的每一个软件中断和硬件中断都有一个编号，这个编号被CPU用作是对这个表的索引（包括系统调用中断——编号为0x80——在第5章中我们已经介绍过了）。

表中相关的项是中断发生时（内核）函数需要跳转到的地址。

4. setup_x86_irq

- 1388: setup_x86_irq给指定的IRQ增加了一个操作（一个struct irqaction结构）。例如，在6088行使用它来记录定时器的中断。它还可以通过request_irq（1439行）使用，这在下一节介绍。
- 1398: Linux使用了几种物理的随机源——例如中断——把一系列不可预知的值提供给设备/dev/random，这是一个有限却具有很高随机性的数据源，还有/dev/urandom，这是对/dev/random的无限的但是随机性较小的对应版本。随机系统作为一个整体在本书中并没有涉及，但是如果你不知道这个概念，这一大部分代码就会显得十分神秘。
- 1412: 如果现存的操作列表非空，setup_x86_irq必须保证现存的操作和新的操作可以共享这个IRQ。
- 1414: 验证这个IRQ可以和其上现存的struct irqaction结构共享。这种测试是十分有效的，它部分是基于我们的一些认识：没有必要遍历执行队列中的所有操作，也没有必要检测它们可能共享的所有情况。除非这两个操作和第一个操作都允许共享IRQ，否则不会允许第一个操作后的所有操作都进入队列。因此，如果第一个操作可以共享IRQ，那么队列中的其他操作也就可以共享IRQ；如果第一个操作不能共享，那么队列中的其他任何操作也都不能共享IRQ。
- 1420: IRQ正在被共享。setup_x86_irq利用p向前执行操作队列直到末尾，离开时p指向队列的最后一个元素的next域。它也会增加shared标志的值，这将会在1429行中被使用。
- 1427: p现在指向队列中的最后一个元素的next域，如果要共享IRQ，或者p在不共享的情况下指向irq_desc[irq].action——指向队列的头节点的指针。不管怎样，指针现在被设置为新的元素了。
- 1429: 如果还没有操作和这个IRQ关联，irq_desc[irq]的其他部分也就还没有设置，在这里就需要对其初始化了。特别要注意1433行中为这个IRQ调用了startup函数。

5. request_irq

- 1439: request_irq从提供的值中创建一个struct irqaction结构，并将其加入对应给定的IRQ的struct irqaction列表中（如果你对C++和Java比较熟悉，可以把它当作是操作的构造函数）。它的实现非常简单明了。
- 1448: 对一对输入值进行稳健性检测。注意没有必要测试irq是否小于0，因为它是一个无符号数。
- 1453: 动态分配新的struct irqaction结构。为此目的使用的函数kmalloc在第8章中简单介绍。
- 1458: 填充新的操作并使用setup_x86_irq将其加入操作列表。

6. free_irq

- 1472: free_irq是request_irq的补数（inverse）。如果request_irq类似于操作的构造程序，那么这就是解除程序最接近的事。
- 1481: 在确保irq在范围内以后，free_irq找到有关的irq_desc项并且开始遍历操作列表。

- 1483: 除非它有正确的设备ID, 否则就忽略这个队列元素。
- 1487: 把这个元素从队列中分离出来并且释放其所占用的内存。
- 1489: 如果现在操作队列为空——也就是如果队列中只有唯一一个元素没有被链接, 设备就会被关闭。
- 1495: 如果控制流程执行到这里, 就意味着free_irq处理了整个操作列表而没有发现匹配的dev_id。如果发现了匹配对象, 1493行的goto语句就已经跳过了本行。因此, 这个试图释放IRQ操作的努力是错误的; 在这种情况下free_irq会打印出一条警告信息对当前状况进行描述。

7. probe_irq_on

- 1506: probe_irq_on实现了内核IRQ自动探测的重要的一部分。阅读14889行开始的标题注释就得到了对整个进程的描述。根据描述我们知道这里要做的工作(只)是执行步骤三: 暂时使能所有没有定义的IRQ, 以使得probe_irq_on的调用者可以检测它们。
- 1514: 对于除IRQ 0之外的每一个IRQ, 如果这个IRQ还没有与之相关的操作, probe_irq_on会记录下这个IRQ正在自动探测的事实并启动关联设备。顺便说明一下, 我不认为有任何的原因使这个循环向后执行。
- 1524: 忙等待约十分之一秒时间以允许生成伪中断的设备取消自己。
- 1530: 循环再次遍历所有的IRQ, 这一次要过滤出所有生成伪中断的设备。这个循环每次重复执行都是从1开始而不是从0开始, 这是因为不需要自动检测的IRQ都被忽略掉了, 而IRQ 0是从来都不会自动检测的。速度在这里也是一个问题; 在十分之一秒的延时之后——这是很长的一段时间, 即使是从慢速的CPU的观点来看也是如此——一个循环或多或少都是有些不合理的。
- 1537: 如果设备在1524行的等待过程中触发了中断, 这个中断可能就是伪中断: 在此期间系统应该还没有和设备通讯过, 因此设备也应该还没有和系统通讯过。因此自动探测位将被清空, 处理程序再次关闭。
- 1544: 返回特殊数字0x12345678, 其原因将在下面的讨论中进行说明。

8. probe_irq_off

- 1547: probe_irq_off实现了IRQ自动探测的另外一部分重要的内容。这里的工作是决定对探测到的哪一个IRQ做出响应, 并返回其中的一个IRQ。
- 1551: 检测名字很容易让人误解的参数unused和probe_irq_on返回的特殊数字是否相同。调用者假定象下面这样处理:

```
magic = probe_irq_on();
/* ... */
probe_irq_off(magic);
```

如果偶然使用了其他的方法调用了probe_irq_off(例如, 如果由于其他一些逻辑调用者偶尔跳过了对probe_irq_on的调用), 那么提供的参数可能不会包含正确的值。可能更重要的是这个参数给正在编写代码使用这个函数的程序员提供了一些信息: 在研究其参数应该是什么的时候, 你会发现在调用它的时候一直遵守的规则。这种规则很容易就被过度使用。

通过对紧随的错误消息的严格调整, 似乎该函数的早期版本中可能已经在其

参数中采用了调用者的地址。如果的确如此，这个测试就具有了第三种目的：把仍然不正确使用这个函数的调用者检测出来。

- 1557: 循环遍历所有的IRQ，搜寻响应调用者探测的所有设备。这个循环也可以从1开始循环，这和前面讨论的probe_irq_on的原因是相同的。
- 1560: 内核没有试图自动检测这个IRQ上的任何内容；它跳到了下一个IRQ。
- 1563: IRQ_INPROGRESS标志指明了该IRQ的一个中断已经到达。由于probe_irq_on可能捕获所有的伪中断，假定这是对探测的真实响应。成功地自动检测到IRQ的数量因此而增一，同时保存第一次的数字。
- 1568: 不管是否成功地自动探测到IRQ，自动探测标志都要减少，并且再次结束处理程序。
- 1573: 如果不止一个IRQ被成功地自动探测到，就通过否定的irq_found来通知调用者。
- 1575: 返回irq_found——0，或者（可能是经过求反的）第一个成功地自动探测到的IRQ号。注意如果发现了设备，则返回值决不会是0，因为内核不会试图自动检测IRQ 0。因此当没有自动检测到IRQ时，probe_irq_off就返回0。

6.3.5 硬件中断处理程序和下半部分

x86系列的实际中断处理程序是微不足道的；在最低的层次上，这是通过反复使用BUILD_IRQ宏（1886行）建立了一系列小汇编函数而实现的。BUILD_IRQ自己被BI宏调用（866行），这个宏又顺次被BUILD_16_IRQS宏（869行）调用，该宏在878行到895行的代码中用来建立汇编程序。这一连串的宏调用的目的仅仅是试图减少必须编写的代码数量和复杂度——我们应该使用256次对BUILD_IRQ的调用，而不是16次对BUILD_16_IRQS的调用。

汇编程序和如下代码相类似：

```
IRQ0x00_interrupt:
    pushl 0x00-256
    jmp common_interrupt
```

也就是每一次都简单地把它的IRQ号（减去256，原因在1897行有论述）压入堆栈，并且跳转到正常的中断程序。

正常的中断处理程序是调用common_interrupt，该函数也十分简短。它是使用BUILD_COMMON_IRQ宏（1871行）建立的，在为do_IRQ进行安排之后简单调用do_IRQ返回给from_intr（233行）——这是第5章中介绍的系统调用的一部分。随后将要介绍的do_IRQ（1362行）负责查看中断是否已经被处理了。

在介绍这些代码之前，从总体上观察一下在处理单个中断时这些部分如何组织在一起是很有帮助的：

1) CPU跳转到IRQ0xNN_interrupt程序（其中的NN是中断号），它将其唯一的中断号压入堆栈并跳转到common_interrupt。

2) common_interrupt调用do_IRQ，并保证当do_IRQ返回时控制流程能够转向ret_from_intr。

3) do_IRQ调用中断处理器芯片独有的代码——直接和芯片通讯的代码，如果需要就用来处理中断。对于PC体系结构中流行的8259A控制器芯片，处理函数是do_8259A_IRQ，在这里提到它仅仅是为了举一个例子而已。

4) `do_8259A_IRQ`暂时禁止正在处理的特殊IRQ，调用`handle_IRQ_event`，接着重新使能这个IRQ。

5) `handle_IRQ_event`为慢速IRQ使能中断，或者为处理快速IRQ而将这些中断保持在禁止状态。接着遍历一个已经和这个IRQ建立联系的函数队列，并依次调用这些函数。由于中断为慢速IRQ而使能，这里就是慢速IRQ的处理程序可能被其他中断所中断的地方。在执行完队列中的所有函数之后，`handle_IRQ_event`禁止中断并返回控制器所特有的处理函数，而该函数将返回到`do_IRQ`。

6) `do_IRQ`处理所有挂起等待的下半部分，接着返回。如同你已经知道的那样，它要返回到`ret_from_intr`。第5章中介绍了从此之后的处理内容。

1. `do_IRQ`

1375: 更新内核的一些统计数字并调用与该IRQ相关的处理函数。对于一些老式PC上的标号较小的IRQ来说，其处理程序是`handle_IRQ_event`。

1383: 如果下半部分是激活的，内核现在就使用`do_bottom_half` (29126行) 对它们进行处理。

2. `handle_IRQ_event`

1292: `handle_IRQ_event`为`do_8259A_IRQ` (821行) 负担了大半重要的工作。本书中没有涉及到的其他一些代码也会调用这个函数。

1302: 与它的描述 (12094行) 正好相反，`SA_INTERRUPT`标志并不是一个`no_op`。如果没有设置该标志，在接下来的代码中就允许中断。这是快速中断和慢速中断之间历史上遗留下来的区别，这一点我们已经讨论过了 (处理这两种类型中断的代码通常有很多差异，但是结果是相同的——代码已经被处理得更加出色了)。恰当地说，这个标志似乎是大多数情况下都为慢速设备使用——顾名思义，就是软盘设备。

1305: 通过调用每一个的处理函数来遍历执行这个IRQ的操作队列 (队列头是由调用者提供的)。

1310: 这里的中断触发是用来为`/dev/random`和`/dev/urandom`增加一些随机信息——从大体上看来，大部分中断都是随机发生的。

1312: 禁止中断 (当条件具备时调用者可以再次允许这些中断)。

3. `do_bottom_half`

29126: Linux代码中有三处调用了`do_bottom_half`: 26707行、243行、1384行 (你会发现，其中的两个是在体系结构特有的文件中的；在非x86平台体系结构特有的文件中对应部分也会调用这个函数)。因此，下半部分是用来处理如下三种情况的：

- 当决定随后哪一个进程应该获得CPU时。
- 当从系统调用中返回时。
- 在从`do_IRQ`中返回之前——也就是说，在每个中断之后。代码中的注释暗示了在内核的未来版本中不一定总在这里运行下半部分。

29130: 下半部分的一个理想特性是在某一时刻只能有一个下半部分处于运行状态。这种特性，也就是锁定 (locking) 在UP代码中体现出重要意义的位置之一，在这里得到了强化。首先调用`softirq_trylock` (UP版本在12559行——第10章中可以查看

所有的SMP版本), 只有在`local_bh_count[cpu]`原来为0时这个函数才把`local_bh_count[cpu]`设置为1并返回真值。根据17479行, 对于UP来说`cpu`总是0, 而且你应该注意到`softirq_trylock`自己是不能被中断的, 因为在这里中断已经被禁止了。`softirq_trylock`和对应的`softirq_endlock` (12561行), 就是仅仅为了以下目的而退出的, 而不存在其他原因: 协助保证这个下半部分不会被其他下半部分中断 (虽然它们可以被上半部分所中断)。

29131: 如果成功获得了锁, 那么该函数就尝试另一个函数, 10736行的`hardirq_trylock`。它只报告当前执行进程是否位于`hardirq_enter/hardirq_exit`对 (10739行和10740行) 之间。对于UP, 这两者的定义是和`irq_enter/irq_exit` (1810行和1811行) 两者的定义相同的; 后两个函数在`handle_IRQ_event`中使用, 当然在其他我们所没有讨论到的地方也对它们有所引用。这些宏协同工作来保证`__cli`和`__sti`对能够正确的进行嵌套——由于CPU不会嵌套使用这些宏, 我们必须保证不会使用`__sti`处理其他的`__cli`, 而这也不是我们所希望的。

29132: 没有其他下半部分在运行, 而且`do_bottom_half`有权使能硬件中断。因此, 它就使能硬件中断, 运行下半部分, 接着再次禁止中断。

29135: 释放该函数已经获取的锁并返回。

4. `run_bottom_halves`

29110: 现在内核能够运行挂起等待的下半部分。

29115: 存储当前局部变量`active`中活动的——也就是被标记过的——下半部分的集合, 并使用`clear_active_bh`宏 (12481行) 清空全局变量`bh_active`中的设置位。对`bh_active`中这些位的清除将同时取消对所有下半部分的标记。

现在你就应该可以看到下半部分有时候是批量处理的, 这一点在前面已经进行了没有论证的说明。此处中断是被使能的, 因此, 如果在`run_bottom_halves`把`bh_active`拷贝到`active`之前就有中断触发, 并标记已经标记过的下半部分, 那么上半部分就已经运行了两次, 然而下半部分只运行了一次。还有, 由于这个中断可以被自己中断, 在下半部分运行一次的时候上半部分就已经运行了三次, 等等。但是随着递归调用数量的增长, 这很快就会变得不再可能了。

代码有可能忽略某个下半部分吗? 假定中断在最坏的时刻发生: 在29115行和29116行之间——也就是在拷贝`bh_active`之后, 但是在清空其中的设置位之前。下面是三种可能的情况:

- 新的中断没有标记下半部分: 这种情况显然不会引起什么问题——中断之后处理的下半部分集合和前面的集合相同, 因此`run_bottom_halves`仍将运行所有它应该运行的下半部分。
- 新的中断标记了一个已经标记过的下半部分: 这种情况也不会引起问题——`run_bottom_halves`不管怎样都要运行这个下半部分, 而且在新中断返回之后就运行它。
- 新的中断标记了一个前面没有标记过的下半部分: 在这种情况下, 当`run_bottom_halves`遍历执行所有的下半部分时, `active`就不再和`bh_active`匹配了。然而, 由于`clear_active_bhs`只会清空`active`集合中设置的位, 所以29116行不会清空`bh_active`中新近标记的位。`clear_active_bhs`使用`atomic_clear_mask` (10262行), 后者简单地对`active`集合中的设置位进行位AND运算, 而并不处理其余部分。因此, 当`run_bottom_halves`执行循环时, 就

不会立刻对新近标记的下半部分进行处理；但是由于它的位仍然在bh_active中设置，run_bottom_halves就仍然会在最后对它进行处理。更为准确的说法是，通过这种方法跳过的下半部分会在处理随后的一个定时器中断的过程中一起处理，这只是一个瞬间的延迟而已——或者如果有其他中断首先发生了，那么这段时间延迟会更短。因此，“迷途的”下半部分通常的等待时间不会超过百分之一秒，而且根据定义，下半部分毕竟不是时效性要求非常高的，这种少量延时不会引起任何问题。

29118: 同时遍历执行bh_base数组和active中的位。如果active中最低的位被设置了，就调用相关的下半部分；接着循环推进到下一个bh_base项和active中的下一位继续执行。

因为这是一个do/while循环，所以它最少执行一次。部分原因是由于在调用do_bottom_half之前，调用者都要检测是否有下半部分需要处理，所以这个循环最少要执行一次。在一些情况下，这种检测会执行到do_bottom_half本身中，但是如果下半部分需要运行，在调用之前执行测试就能够节省函数调用的开销。不管怎样，我们很容易就可以看出即使没有下半部分需要运行，这个循环也可以正确执行；虽然这会浪费一些时间，但是不这样就会引起错误。

29123: 当active中没有任何位被设置时，循环就终止退出。由于在循环执行的过程中active是不断移位的，这样就同时测试了其余的位，而没有必要对它们的每一个都进行循环处理。

6.4 时间

本节通过观察一个中断的例子——定时器中断——的工作方式来使你能够将中断和下半部分的知识融会贯通起来。

定时器中断函数timer_interrupt是和6086行的IRQ 0相关的。此处使用的irq0变量是在5937行定义的。27972行通过使用init_bh（12484行）把timer_bh函数注册为定时器的下半部分。

当触发IRQ 0时，timer_interrupt从CPU时间戳计数器中读取一些属性值，如果CPU中有值（这在本书中没有涉及到的一些代码中使用），就调用do_timer_interrupt（5758行）。除了其他一些工作之外，它会调用do_timer，这是定时器中断非常有趣的一部分。

1. do_timer

27446: 从我们的出发点来看，这是我们感兴趣的定时器的上半部分。

27448: 更新全局变量jiffies，这个值记录了机器启动以来系统时钟滴答的次数（显然，这实际上记录的是从定时器中断装载以来已经经过的定时器滴答的次数，定时器中断在系统启动的瞬间是不会发生的）。

27449: 递增丢失的定时器滴答的数目——也就是那些没有被下半部分处理的定时器的滴答。很快你就会看到有定时器的下半部分是怎样使用这个变量的。

27450: 上半部分已经运行了，因此其下半部分被标记为只要可能就运行。

27451: 除了要记录从上一次定时器的下半部分运行以来定时器滴答发生的次数之外，我们还要知道有多少这种滴答发生在系统模式下。很快我们会再次看到为什么下半部分需要它；但是在目前，如果进程运行在系统（内核）模式下而不是用户模式下，那么只需要递增lost_ticks_system的值。

27453: 如果定时器队列中有任务在等待, 定时器队列的下半部分被标记为准备好运行(我们很快就会对定时器队列进行讨论)。而这就是整个定时器中断。虽然这看起来十分简单, 但是这很大程度上是由于主要的工作都适当地延迟到下半部分处理了。

2. timer_bh

27439: 这是定时器的下半部分。它调用函数为进程和内核本身更新有关时间的统计数字, 并同时为老式内核定时器和新式内核定时器进行处理。

3. update_times

27412: 这个函数主要是更新统计数字: 计算系统的平均负载, 更新记录当前时间的全局变量, 并更新内核的当前进程使用的CPU时间的估计值。

27422: 取得从上次下半部分运行以来发生的定时器滴答的数目, 并重置计数器。

27427: 如果数字非空——正常情况下都是这样——update_times会找出有多少这种滴答发生在系统模式下。

27429: 调用calc_load (27135行, 马上就介绍) 更新内核有关系统负载要素的估计值。

27430: 调用update_wall_time (27311行, 后面会讨论) 更新xtime, 它记录了当前的wall_clock时间。

27433: 调用update_process_times (27382行), 该函数同个辅助函数update_one_process (27371行) 共同作用, 更新内核中有关当前进程已经运行的时间长度的统计。这些统计数字可以使用诸如time、top和ps之类的普通程序得到。现在你就可以看出, 这些统计资料未必一定要正确: 如果一个进程能够做到在所有时钟中断触发的时刻都不处于运行状态, 那么它就可以偷偷地使用大部分CPU, 而且还不会被认为使用了任何CPU资源。然而, 还是比较难以理解为什么(或者怎样)恶意的进程要试图执行它, 对于意图良好的进程来说, 统计资料自然要进行平均——它们要为一些自己几乎不能使用的CPU定时器滴答负责, 但是却不用为那些自己几乎在整个定时器滴答期间都在使用CPU的另一种情况负责, 但是这些最终都不存在了。

4. update_wall_time

27313: 为每一个必须要处理的滴答调用update_wall_time_one_tick (27271行)。它更新了全局变量xtime, 如果可能, 就通过遵守网络时间协议(Network Time Protocol)努力将其和实际时间保持同步。

27318: 将xtime标准化, 使得微秒数在0到999 999的范围内。在极端的情况下可能会丢失多于一秒的定时器滴答, 那么xtime的tv_usec部分就可能超出200万, 这段代码就可能在标准化xtime时失败。但是随后的调用可以把xtime完全标准化。

5. calc_load

27135: calc_load从定时器的下半部分中被调用, 以更新内核对于当前系统负载的估计值。虽然对于它的介绍有点离题, 但是对于每个对这个随处可见的数字是如何计算的感到疑惑的人都会对这个函数很感兴趣, 因此它还是值得一看的。

27138: 静态变量count记录了从上次计算平均负载以来已经经过了多少时间。它被初始化为LOAD_FREQ, 这在16164行中进行宏定义以代表相当于5秒时间的定时器间

隔。对于这个函数的每一个调用都要递减遗留到下一次计算负载的定时器滴答数量。

27142: 当剩余的滴答数小于0时, 就应该重新计算了。(我倒是希望这个数是小于或者等于0, 而不是仅仅小于0, 但是实际上这两个值差不了多少。)

27143: 为了能在另一5秒内可以再次触发, 重新初始化count, 此后, count_active_tasks (27119行) 被用来监视系统中当前有多少任务。现在count_active_tasks的实现可能已经不是什么奇妙的事情了, 但是所有有关它的问题在阅读完第7章后都应该得到解答。

27144: 在16169行定义的CALC_LOAD宏用来更新avenrun数组 (27116行) 的三个项, 它的三个元素分别记录了前面5s、10s、15s的系统负载。与前一章中说明的一样, 内核中尽量避免浮点数运算, 因此这些计算都是在固定点进行的。

6. run_old_timers

27077: 内核提供了类似于下半部分的已经不再使用的技巧, 通过这种技巧内核函数可以被登记到表中, 和超时时间建立联系, 并在指定时间到达时调用。这个函数根据需要调用其他函数。由于现在使用这个函数的唯一目的是为了支持原来的代码, 所以我就不仔细地介绍它了。它简单地遍历处理timer_table数组 (27109行) 的列表项, 如果定时器已经触发就调用相关函数。

定时器队列

定时器队列最初的思想是基于与上半部分相关的下半部分并不一定非要与中断处理有关。相反的, 它也可能是我们所需要周期性处理的任何内容。将一个函数定义为定时器中断处理程序下半部分的一部分能够保证这个函数每秒钟大约被调用100次 (如果每秒运行100次太频繁了, 那么这个函数可以保持一个计数器并每10次调用都简单返回9次, 例如——我们看到calc_load就是这样处理的)。结果就像是创建了一个进程, 它的main部分在无穷循环中调用这个函数, 于是它没有占用通常的和进程相关的开销。

但是, 下半部分是一种有限的资源——只存在32个, 这是因为我们希望bh_mask和bh_active各自都匹配一个无符号长整型数。我们可以通过使用系统中类似于实现信号的方法来扩展下半部分的数量, 但是这样仅仅能增加静态可用的下半部分的数量——它并不能使我们动态地扩展下半部分列表。

从本质上说, 这是定时器队列所提供的内容: 一个动态的可增长的下半部分的列表, 所有项都和定时器中断有关。这里有一个独立的下半部分以处理这种情况——TQUEUE_BH——如同你看到的一样, 如果定时器队列中有任务, 它就和TIMER_BH一起被标记。因此, 定时器中断有两个下半部分。

定时器队列实际上只是更普通的内核特性——任务队列的一个实例。根据说明文档, 任务队列在内核本身中是相当自由的——请参看文件include/linux/tqueue.h从18527行开始的部分。因此, 接下来我们就不再介绍它们了。但是, 它们是很重要的内核服务, 而且那个简短的文件也很值得一读。

第7章 进程和线程

操作系统的存在归根结底是为了提供一个运行程序的空间。按照Unix的术语，将正在运行的程序称为进程。Linux内核和其他Unix变种一样，都是采用了多任务技术；它可以在许多进程之间分配时间片，从而使这些进程看起来似乎在同时运行一样。这里通常是内核对有关资源的访问作出仲裁；在这种情况下，资源就是CPU时间。

进程传统上都有唯一的执行程序的上下文——这是说明在某个时刻它正在处理一项内容的流行的方法。在给定的时刻，我们可以精确地知道代码的哪一部分正在执行。但是有时我们希望一个进程同时处理多件事情。例如，我们可能希望Web浏览器获取并显示Web页，同时也要监视用户是否点击停止按钮。只为监视停止按钮而运行一个全新的程序显然是不必要的，但是对于Web浏览器来说要对其时间进行分隔也并不总是非常方便——获取一些Web页信息、检测停止按钮，再获取一些Web页信息，再重新检测停止按钮，等等。

对于这个问题的比较流行的解决方法是线程。从概念上来说，线程是同一个进程中独立的执行上下文——更简单一点说，它们为单一进程提供了一种同时处理多件事情的方法，就像是进程是一个自行控制的微缩化了的多任务操作系统。同一线程组中的线程共享它们的全局变量，并有相同的堆（heap），因此使用malloc给线程组中的一个线程分配的内存可以被该线程组中的其他线程读写。但是它们拥有不同的堆栈（它们的局部变量是不共享的），并可以同时进程代码不同的地方运行。这样，你的Web浏览器可以让一个线程来获取并显示Web页，同时另外一个线程观测停止按钮是否被点击，并且在停止按钮被点击时停止第一个线程。

和线程等价的一种观点——这是Linux内核使用的观点——线程只是偶然的共享相同的全局内存空间的进程。这意味着内核无需为线程创建一种全新的机制，否则必然会和现在已经编写完成的进程处理代码造成重复，而且有关进程的讨论绝大多数也都可以应用到线程上。

当然，以上的说明仅仅适用于内核空间的线程。实际中也有用户空间的线程，它执行相同的功能，但是却是在应用层实现的。用户空间的线程和内核空间的线程相比有很多优点，也有很多缺点，但是有关这些问题的讨论超出了本书的范围。而使人更加容易产生混淆的是一个名为kernel_thread（2426行）的函数，尽管该函数被赋予了这样一个名字，但是它实际和内核空间的线程没有任何关系。

部分是由于历史的原因，部分是由于Linux内核并没有真正区分进程和线程这两者在概念上的不同，在内核代码中进程和线程都使用更通用的名字“任务”来引用。根据同样的思路，本书中所出现的“任务”和“进程”具有相同的意义。

7.1 调度和时间片

对CPU访问的裁决过程被称为调度（scheduling）。良好的调度决策要尊重用户赋予的优先级，这可以建立一种所有进程都在同时运行的十分逼真的假象。糟糕的调度决策会使操作系统变得沉闷缓慢。这是Linux调度程序必须经过高度优化的一个原因。

从概念上来说，调度程序把时间分为小片断，并根据一定的原则把这些片断分配给进程。

你可能已经猜到，时间的这些小片断称为时间片。

7.2 实时进程

Linux提供了三种调度算法：一种传统的Unix调度程序和两个由POSIX.1b（原名为POSIX.4）操作系统标准所规定的“实时”调度程序。因此，本书中有时会使用实时进程（从技术上考虑，系统使用术语“非实时进程（nonrealtime process）”来作为实时进程的对应，虽然我更倾向于使用另外一个术语（unrealtime process）。不要过分计较“实时”这个术语，虽然——如果从硬件的角度来看待这个问题，实时意味着你可以得到有关操作系统的某种性能保证，例如有关中断等待时间的承诺，但是这一点在Linux实时调度规则中并没有提供。相反的，Linux的调度规则是“软件实时”，也就是说如果实时进程需要，它们就只把CPU分配给实时进程；否则就把CPU时间让出给非实时进程。

如果你真正需要，一些Linux的变种也承诺提供一种“硬实时”。但是，在当前的Linux内核中——因此也就是在本章中——“实时”仅指“软件实时”。

7.3 优先级

非实时进程有两种优先级，一种是静态优先级，另一种是动态优先级。实时进程又增加了第三种优先级，实时优先级。优先级是一些简单的整数，为了决定应该允许哪一个进程使用CPU的资源，用优先级代表相对权值——优先级越高，它得到CPU时间的机会也就越大：

- 静态优先级——被称为“静态”是因为它不随时间而改变，只能由用户进行修改。它指明了在被迫和其他进程竞争CPU之前，该进程所应该被允许的时间片的最大值（但是也可能由于其他原因，在该时间片耗尽之前进程就被迫交出了CPU）。
- 动态优先级——只要进程拥有CPU，它就随着时间不断减小；当它小于0时，标记进程重新调度。它指明了在这个时间片中所剩余的时间量。
- 实时优先级——指明这个进程自动把CPU交给哪一个其他进程：较高权值的进程总是优先于较低权值的进程。因为如果一个进程不是实时进程，其优先级就是0，所以实时进程总是优先于非实时进程的（这并不完全正确；如同后面论述的一样，实时进程也会明确地交出CPU，而在等待I/O时也会被迫交出CPU。前面的描述仅限于能够交付CPU运行的进程）。

7.4 进程ID: PID

传统上每个Unix进程都有一个唯一的标识符，它是一个被称为进程标识符（PID）的、范围在0到32 767之间的整数。PID 0和PID 1对于系统有特定的意义；其他的进程标识符都被认为是普通进程。在本章后面对get_pid的讨论中，会看到PID是如何生成和赋值的。

在Linux中，PID不一定非要唯一——虽然通常都是唯一的，但是两个任务也可以共享一个PID。这是Linux对线程支持的一个副作用，这些线程从概念上讲应该共享一个PID，因为它们是同一个进程的一部分。在Linux中，你可以创建两个任务，并且共享且仅共享它们的PID——从实际使用角度讲它们不会是线程，但是它们可以使用同一个PID。这并没有多大的意义，但是如果你希望这样处理，Linux是支持的。

7.5 引用计数

引用计数是多个对象之间为共享普通信息而广泛使用的技术。使用更通用的术语来说，

一个或多个“容器对象”携带指向共享数据对象的指针，其中包含了一个称为“引用计数 (reference count)”的整数；这个引用计数的值和共享数据的容器对象的个数相同。希望共享数据的新容器对象将被赋予一个指向同一结构的指针，并且递增该共享数据对象的引用计数。

当容器对象离开时，就递减共享数据的引用计数，并做到“人走灯熄”——也就是当引用计数减小到0时，容器对象回收共享对象。图7-1阐述了这种技术。

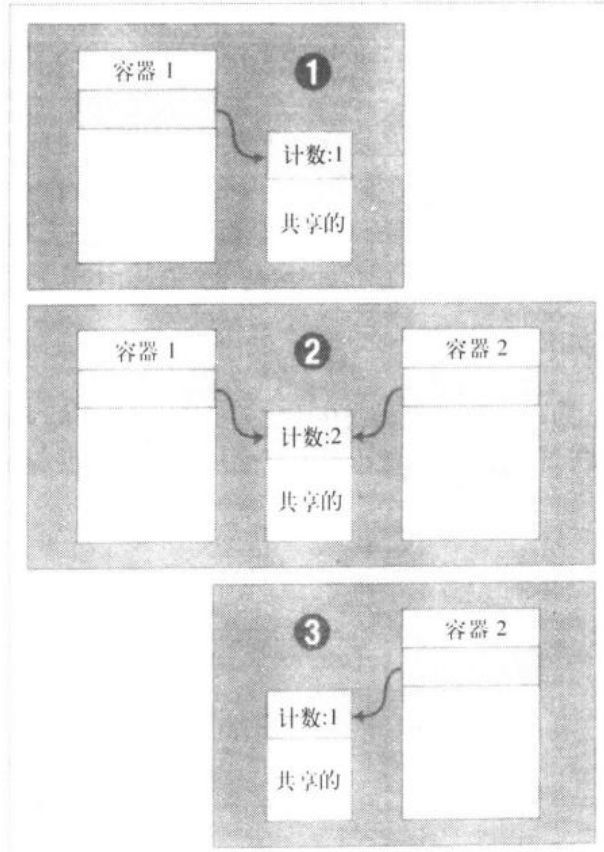


图7-1 引用计数

就像你随后会看到的那样，Linux通过使用引用计数技术来实现线程间的数据共享。

7.6 权能

在早期的Unix中，你或者是root用户，或者不是。如果你是root，你几乎可以进行任何希望进行的操作，即使你的想法实际上十分糟糕，例如删除系统引导盘上的所有文件。如果你不是root，那么你就不会对系统造成太大的损害，但是你也无法执行任何重要的系统管理任务。

不幸的是，很多应用程序的需要都介于这两个安全性极端之间。例如，修改系统时间只有root才能执行的操作，因此实现它的程序必须作为root运行。但是因为是作为root运行的，修改系统时间的进程也就能处理root可以完成的任何事情。对于编写良好的程序来说并不会造成问题，但是程序仍然会有意无意地把系统搞得一团糟（数不清的计算机攻击事件都是欺骗root去运行一些看似值得信任的可执行代码，造成了一些恶作剧）。

这些问题中有一些可以通过正确使用组和诸如sudo之类的程序而避免，但是有一些则不

行。对于某些重要的操作，虽然你可能只想允许它们执行一两种权限操作，你也只能给予这些进程普通root访问许可。Linux对于这个问题的解决方法是使用从现在已经舍弃了的POSIX草案标准中抽取出来的思想——权能。

权能使你可以更精确地定义经授权的进程所允许处理的事情。例如，你可以给一个进程授予修改系统时间的权力，而没有授予它可以杀掉系统中的其他进程、毁坏你的文件、并胡乱运行的权力。而且，为了帮助防止意外地滥用其优先级，长时间运行的进程可以暂时获得权能（如果允许），只要时间足够处理特殊的零碎工作就可以了，在处理完这个零碎的工作以后再收回权能。

在本书的编写期间，权能仍然处于开发状态。为了完全实现权能的预期功能，开发者们还必须要实现一些新的特性——例如，目前还没有内核支持将程序的权能附加到文件本身中。这样所造成的一个后果是Linux有时仍要检测进程是否作为root运行，而不是检测进程所需要的特殊权能。但是迄今为止已经实现了的内容仍然是十分有用的。

7.7 进程在内核中是如何表示的

内核使用几个数据结构来跟踪进程；其中有一些和进程自身的表示方法是密切相关的，另外一些则是独立的。图7-2阐述了这些数据结构，随后会对它们进行详细介绍。

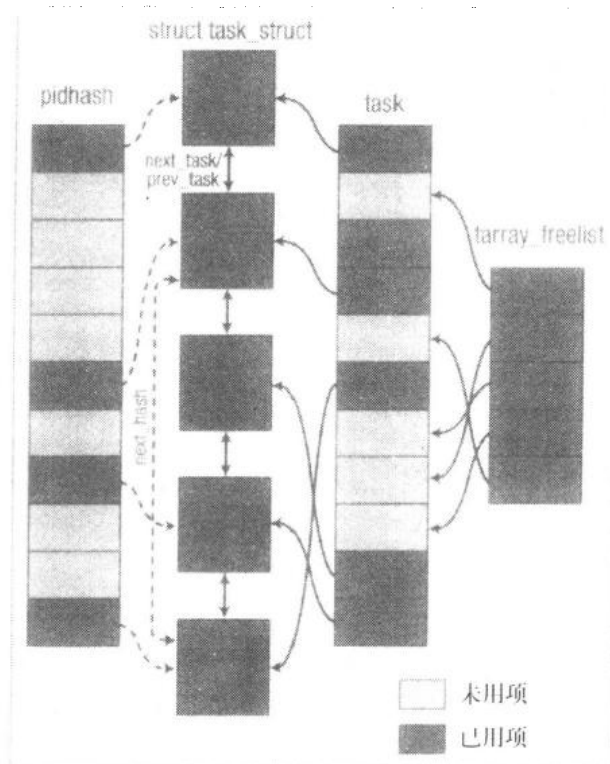


图7-2 管理任务使用的内核数据结构

16325: 表示进程的 kernel 数据结构是 `struct task_struct`。我们暂时向前跳过这个结构的定义，继续往下看。它相当大，但是可以从逻辑上划分为很多部分。随着本章讨论的展开，你将会逐渐清楚它们每一部分的意义。在阅读的过程中，要注意这个结构的很多部分都是指向其他结构的指针；这在子孙进程和祖先进程希望共享指针所指

向的信息时可以灵活运用——很多指针都指向正在被引用计数的信息。

16350: 任务本身使用struct task_struct结构的next_task和prev_task成员组成一个循环的双向链接列表，它被称为任务队列。的确，这忽略了一个事实，它们在中心数组task（很快就会讨论）中早已存在了。最初这看起来可能有些奇怪，但实际上这是十分有用的，因为这样允许内核代码可以遍历执行所有现存的任务——也就是task中所有经过填充的时间片——而无须浪费时间跳过空时间片。实际上对这个循环的访问是如此频繁，以至于在16898行单独为它定义了一个宏for_each_task。

虽然for_each_task是单向的，但是它有一些值得注意的特性。首先，注意到循环的开始和末尾都是init_task。这是很安全的，因为init_task从来不会退出；因此，作为标记它一直都是可用的。但是，注意到init_task本身不是作为循环的一部分而访问的——这恰好就是你使用这个宏时所需要的东西。还有，作为我们关心的一小部分，你总是使用next_task成员直接向前遍历执行列表的；不存在相关的向后执行的宏。也没有必要需要这样一个宏——只有在需要及时把任务从列表中清除时才需要使用prev_task成员。

16351: Linux还保持一个和这个任务列表类似的循环的双向任务列表。这个列表使用struct task_struct结构的prev_run成员和next_run成员进行链接，基本上作为队列来处理的（这真值得让人举杯庆祝）；出于这个原因，这个列表通常被称为运行队列（run queue）。对于next_task来说，只是因为需要高效地将一个项移出队列才会使用到prev_run成员；对于这个列表的遍历循环执行通常都是使用next_run向前的。同样，在这个任务队列中也使用init_task来标记队列的开始和末尾。

通过使用add_to_runqueue（26276行）能够将任务加入队列，而使用del_from_runqueue（26287行）则把任务移出队列。有时候分别使用move_first_runqueue（26318行）和move_last_runqueue（26300行）把它们强制移动到队列的开头和末尾。注意这些函数都是局限于kernel/sched.c的，在别的文件中不会使用prev_run和next_run域（特别是在kernel/fork.c文件中的进程创建期间）；这是十分恰当的，因为只有当在调度时才需要运行队列。

16370: 首先，任务能够组成一个图，该图的结构表达了任务之间的家族关系；由于我不清楚这个图所使用的通用术语，我就称它为进程图（process graph）。这和next_task/prev_task之间的链接根本没有关系，在那里任务的位置是毫无意义的——只是一个偶然的偶然的历史事件而已。每一个struct task_struct中有五个指向进程图中自己位置的指针。这五个指针在从16370行到16371行的代码中被定义。

- p_opptr指向进程的原始祖先；通常和p_pptr类似。
- p_pptr指向进程的当前祖先。
- p_cprr指向进程的最年青（最近）子孙。
- p_ysptr指向进程的下一个最年青（下一个最近）兄弟。
- p_osptr指向进程的下一个最古老（下一个最远）兄弟。

图7-3说明了它们之间的关系（整个链接集合都以标号为“Me”的节点为核心）。这个指针的集合还提供了浏览系统中进程集合的另外一种方法；显然，在处理诸如查找进程祖先或者查找列表中进程子孙时这个指针特别有用。这个指针是由两个宏维护的：

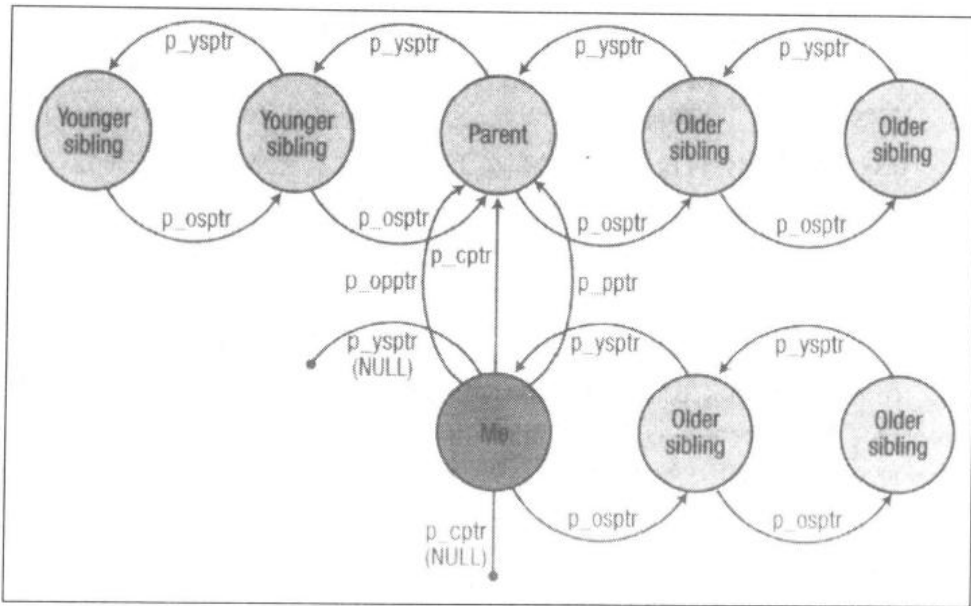


图7-3 进程图

- REMOVE_LINKS (16876行) 从图中移出指针。
- SET_LINKS (16887行) 向图中插入指针。

这两个宏都可以调整next_task/prev_task的链接。如果你仔细研究一下这两个宏，你就会发现它们只是增加或者删除叶子进程——而不会对拥有子孙进程的进程进行处理。

16517: task定义为由指向struct task_struct结构的指针组成的数组。这个数组中的每一项代表系统中的任务。数组的大小是NR_TASKS (在18320行设置为512)，它规定了系统中可以同时运行的任务数量的上限。因为一共有32 768个可能的PID，由于数组不够大，要通过它们的PID直接索引系统中所有任务显然是不可能的（也就是task[i]未必是由PID i指明的任务）。相反，Linux使用其他的数据结构来帮助系统管理这种有限的资源。

16519: 自由时间片列表tarray_freelist拥有一个说明task数组中自由位置的列表（实际上是一个堆栈）。它在27966行和27967行初始化，接着被两个在16522行到16542行定义的内联函数所使用。在SMP平台上，对于tarray_freelist的访问必须受自旋锁taskslot_lock (23475行)的限制（自旋锁在第10章中详细讨论）。

16546: pidhash数组有助于把PID映象到指向struct task_struct的指针。pidhash在27969行和27970行初始化，此后它被一系列在16548行到16580行定义的宏和内联函数所操纵。这些最终实现了一个普通的散列表。注意，为了处理散列表元，维护pidhash的函数使用了struct task_struct结构中的两个成员——pidhash_next (16374行)和pidhash_pprev (16375行)。通过使用pidhash，内核可以通过其PID有效地发现任务——虽然这种方式仍然比直接查找要慢。

仅仅是为了好玩，你可以自己证明这个散列函数——pid_hashfn，16548行——提供了一个均匀覆盖其域0到32 767（所有有效的PID）的发行版本。除非你所谓的“好玩”的概念和我不一样，否则你会同我一样感到有趣。

这些数据结构提供了有关当前运行系统的很多信息，但是这也需要付出代价：

每当增加或删除进程时，这些信息必须能够得到正确维护，否则系统就会变得混乱不堪。考虑到实现这种正确的维护非常困难，进程只在一个地方创建（使用 `do_fork`，后面会讨论），也只有一个地方删除（使用 `release`，也在后面中讨论）。

如果我们能把 `task` 处理为 32 768 个 `struct task_struct` 结构组成的数组，其中的每一项代表一个可能的 PID，那么至少可以消除一部分这种类型的复杂性。但是这样处理会大量增加内核对于内存的需求。每一个 `struct task_struct` 结构在 UP 平台上占用 964 字节，在 SMP 平台上占用 1212 字节——取整以后，近似的数字是 1K。为了容纳所有这些结构，`task` 会像气球一样迅速膨胀到 32 768K，也就是 32M！（实际情况会更糟糕：我们尚未提到的有关任务的额外内存开销会把这个数字增长 8 倍——也就是 256M——而且不要忘记了，这些开销实际上都还没有运行一个任务）。此外，x86 的内存管理硬件把活动任务的数量限制在 4 000 左右；这一主题在下一章介绍。因此，数组中大多数的空间都会不可避免地被浪费了。

在目前的实现中，如果没有进程在运行，`task` 仅仅是 512 个 4 字节的指针，总共才 2K。如果我们考虑到那些附加的数据结构会占用一些额外开销，可能有一些超过这个数字，但是比起 32M 来还差得远呢。即使是 `task` 中的每一项都使用了，而且每个 `struct task_struct` 结构也都分配了，总共使用的内存也才不过 512K 左右。应用程序能够忽略这种微小的区别。

进程状态

在一个给定的时间，进程处于下面注释中描述的六种状态中的一种。进程的当前状态被记录在 `struct task_struct` 结构的 `state` 成员中（16328 行）。

- 16188: `TASK_RUNNING` 意味着进程准备好运行了。即使是在 UP 系统中，也有不止一个任务同时处于 `TASK_RUNNING` 状态——`TASK_RUNNING` 并不意味着该进程可以立即获得 CPU（虽然有时候是这样），而是仅仅说明只要 CPU 一旦可用，进程就可以立即准备好执行了。
- 16189: `TASK_INTERRUPTIBLE` 是两种等待状态的一种——这种状态意味着进程在等待特定事件，但是也可以被信号中断。
- 16190: `TASK_UNINTERRUPTIBLE` 是另外一种等待状态。这种状态意味着进程在等待硬件条件，而且不能被信号中断。
- 16191: `TASK_ZOMBIE` 意味着进程已经退出了（或者已经被杀掉了），但是其相关的 `struct task_struct` 结构并没有被删除。这样即使子孙进程已经退出，也允许祖先进程对已经死去的子孙进程的状态进行查询。在本章后面我们会详细介绍这一点。
- 16192: `TASK_STOPPED` 意味着进程已经停止运行了。一般情况下，这意味着进程已经接收到了 `SIGSTOP`、`SIGSTP`、`SIGTTIN` 或者 `SIGTTOU` 信号中的一个，但是它也可能意味着当前进程正在被跟踪（例如，进程正在调试器下运行，用户正在单步执行代码）。
- 16193: `TASK_SWAPPING` 主要用于表明进程正在执行磁盘交换工作。然而，这种状态似乎是没有用的——虽然该标识符在整个内核中出现了好几次，但是其值从来没有被赋给进程的 `state` 成员。这种状态正在被逐渐淘汰。

7.8 进程来源：fork 和 `_clone`

传统的 Unix 实现方法在系统运行以后只给出了一种创建新进程的方法：系统调用 `fork`。

(如果你奇怪第一个进程是哪来的, 实际上该进程是init, 在第4章中已经讨论过。) 当进程调用fork时, 该进程从概念上被分成了两部分——这就像是道路中的分支——祖先和子孙可以自由选择不同的路径。在fork之后, 祖先进程和其子进程几乎是等同的——它们所有的变量都有相同的值, 它们打开的文件都相同, 等等。但是, 如果祖先进程改变了一个变量的值, 子进程将不会看到这个变化, 反之亦然。子进程是祖先进程的一个拷贝 (至少最初是这样), 但是它们并不共享内容。

Linux保留了传统的fork并增加了一个更通用的函数`__clone` (前面的两个下划线有助于强调普通应用程序代码不应该直接调用`__clone`, 应该从在`__clone`之上建立的线程库中调用这个函数)。鉴于fork创建一个新的子孙进程后, 子孙进程虽然是其祖先进程的拷贝, 但是它们并不共享任何内容, `__clone`允许你定义祖先进程和子孙进程所应该共享的内容。如果你没有给`__clone`提供它能够识别的五个标志, 子孙进程和祖先进程之间就不会共享任何内容, 这样它就和fork类似。如果你提供了全部的五个标志, 子孙进程就可以和祖先进程共享任何内容, 这就和传统线程类似。其他标记的不同组合可以使你完成介于两者之间的功能。

顺便提一下, 内核使用`kernel_thread`函数 (2426行) 为自己的使用创建了几个任务。用户从来不会调用这个函数——实际上, 用户也不能调用这个函数; 它只在创建例如`kswapd` (在第8章中介绍) 之类的特殊进程时才会使用, 这些特殊进程有效地把内核分为很多部分, 为了简单起见也把它们当作任务处理。使用`kernel_thread`创建的任务具有一些特殊的性质, 这些性质我们在此不再深入介绍 (例如, 它们不能被抢占); 但是现在主要需要引起注意的是`kernel_thread`使用`do_fork`处理其垃圾工作。因此, 即使是这些特殊进程, 它们最终也要使用普通进程的创建方法来创建。

`do_fork`

23953: `do_fork`是实现fork和`__clone`的内核程序。

23963: 分配`struct task_struct`结构以代表一个新的进程。

23967: 给新的`struct task_struct`结构赋予初始值, 该值直接从当前进程中拷贝而来。`do_fork`的剩余工作主要包含为祖先进程和子孙进程不会共享的信息建立新的拷贝。(在本行和整个内核中你可以看到的`current`是一个宏, 它把一个指针指向代表当前正在执行的进程的`struct task_struct`结构。这在10285行定义, 但实际上只是对`get_current`函数的一个调用, 而后者的定义在10277行。)

23981: 新到达者需要`task`数组中的一个项; 这个项是使用`find_empty_process` (23598行——它严格依赖于16532行的`get_free_taskslot`) 找到的。然而, 它工作的方式有点不明显: `task`数组没有使用的成员不是设置为空, 而是设置为自由列表的下一个元素 (使用`add_free_taskslot`, 16523行)。因此, `task`中没有使用的项指向链接列表中另外一个`task`没有使用的项, 而`tarray_freelist`仅仅指向这个列表的表头。那么, 返回一个自由位置就简单地变成了返回列表头的问题了 (当然要把这个头指针指向下一个元素)。更传统的方法是使用一个独立的数据结构来管理这些信息, 但是在内核中, 空间总会显得有些不足。

23999: 给新的任务赋PID (其中的细节很快就会介绍)。

24045: 本行和下面几行, 使用该文件中别处定义的辅助函数, 根据所提供的`clone_flags`参数的值为子孙进程建立祖先进程的数据结构中子孙进程所选择部分的拷贝。如

果`clone_flags`指明相关的部分应该共享而不是拷贝，这时辅助函数（help function）就简单地增加引用计数接着返回；否则，它就创建新进程所独有的新的拷贝。

24078: 到现在为止，所有进程所有的数据结构都已经设置过了，但是大部分跟踪进程的数据结构还没有被设置。系统将通过把进程增加到进程图表中开始设置它们。

24079: 通过调用`hash_pid`把新的进程置入`pidhash`表中。

24088: 通过调用`wake_up_process`（26356行）把新的进程设置为`TASK_RUNNING`状态，并将其置入运行队列。

注意到现在不止是`struct task_struct`结构被设置了，而且所有相关的数据结构——自由时间片列表、任务列表、进程图、运行队列和PID hash表——这些都已经为新的到达者而正确进行了修改。恭喜你，你现在已经得到了一个健康的子孙任务。

PID的分配

PID是使用`get_pid`函数（23611行）生成的，该函数能够返回一个没有使用的PID。它从`last_pid`（23464行）开始——这是最近分配的PID。

内核中使用的`get_pid`的版本是内核复杂性和速度频繁折中的一个例子；这里速度更为重要一些。`get_pid`经过了高度优化——它比直接向前的实现方法要复杂得多，但是速度也要快得多。最直接的实现方法将遍历执行整个任务列表——典型的情况可能有几十项，有时候也可能成百上千项——对每一个可能的PID进程检测并找出适当的值。我们见到的版本有时是必须执行这些步骤的，但是在大多数情况下都可以跳过。这一结果被用来帮助加速进程创建的操作，它在Unix上慢得臭名卓著。

如果我们所需要的只是要为每一个运行进程都快速计算一个各不相同的整数，那么这里已经有现实可用的方法：只要取在`task`数组中进程的索引就可以了。平均说来，这肯定要比现在的`get_pid`速度要快——毕竟，这无须遍历任务列表。不幸的是，很多现存的应用程序都假定在一个PID可以再重用之前都需要等待一段时间。这种假定在任何情况下都是不安全的，但是，如果为了这些程序的问题而将内核牵涉进去可能仍然是一个很糟糕的思想。现存的PID分配策略速度仍然很快，并且它偶尔还有可以暴露这些应用程序中的潜在的缺陷，如果有的话。

`get_pid`

23613: `next_safe`变量是一个为加快系统运行速度而设定的变量；它保持记录了可能保留的次最低的候选PID（更正确地说应该把它命名为`next_unsafe`）。当`last_pid`递增并超过这个范围时，系统应该检测整个任务列表来保证这个候选PID是否仍在被保留着（原来保留这个PID的进程现在可能已经运行完了）。由于遍历这个任务列表可能会很慢，所以，只要可能就应该避免执行这样的操作。因此，在执行这个遍历的过程中，`get_pid`要重新计算`next_safe`——如果有些进程已经死掉了，这个数字可能现在更大了，因此`get_pid`可以避免一些将来对任务列表的遍历（`next_safe`是静态的，因此其值在下次`get_pid`需要分配PID时就会保留下来）。

23616: 如果新的进程要和其祖先共享PID，就返回祖先进程的PID。

23620: 开始搜寻候选PID寻找未使用的值。位与运算只是通过测试低15位是否置位来简单测试`last_pid`的新值是否超过了32 767（最大允许的PID）。我怀疑这些内核开发者真的需要通过这样做来获得微小的速度优势，但是你永远也不会知道；至少在

这段代码编写期间，gcc还不够敏锐到足以注意到它们的等价性，并在生成的代码中选择稍微快速的形式。

- 23621: 如果last_pid已经超出了允许的最大值，它就会滚动到300。300这个数字并没有什么魔力——它对于内核并没有特别的意义——这是另外一个加速变量。其思想是数字比较小的PID通常都属于系统开始运行时就已经创建的、从不会退出的长时间运行的后台监控程序。由于它们总是占据着数字比较小的PID，所以，如果不考虑对前面几百个值的重用问题，我们将会发现寻找可以使用的PID的过程会快许多。而且，由于PID的空间是同时允许的任务数（512）的64倍，为了追求速度而损失一些空间是非常值得的。
- 23622: 由于last_pid超出了最大允许的PID，它必然也就超出了next_safe；因此，后面的if测试也可以跳过。
- 23624: 如果last_pid仍然小于next_safe，其值就可以再用。否则，必须检查任务列表。
- 23633: 如果取得了last_pid的当前值，它就简单地递增，如果需要就跳转到300，重新开始循环。初次看的时候，仿佛这个循环会一直运行下去——如果所有的PID都已经被使用了会出现什么情况呢？但是稍微考虑一下，我们就可以排除这种可能性：任务列表的最大值和同时并发的任务的最大数是相同的，有效的PID数目要比这两个数都大得多。因此，循环最终会找到有效的PID；这仅仅是个时间的问题。
- 23651: get_pid已经发现了一个没有被使用的PID，随后返回该PID。

7.9 运行新程序

如果我们能够进行的所有工作只是fork（或者__clone），那么我们就一次次建立同一个进程的拷贝就可以了——这样我们的Linux系统就只能运行在系统中第一个创建的用户进程init了。init是很有用的，但是功能还没有如此强大；我们也还需要处理其他事情。

在我们创建新的进程以后，它通过调用exec就能够变成独立于其他进程的进程了（这实际上不止是一个名为exec的函数；exec通常用作一个引用一系列函数的通用术语，所有这些函数基本上都处理相同的事情，但是使用的参数稍微有些不同）。

因此，创建一个“真正”的新进程——与其祖先不同的程序运行镜像——任务分为两步，一步是fork，另一步是exec，最后能够得出下面风格的C代码：

```
/* Ignore the possibility of error below. */
if (fork()) {
    /* I'm the parent; continue normally. */
} else {
    /* I'm the child */
    /* Become /some/other/program. */
    execl("/some/other/program",
          "/some/other/program")
}
```

（execl是exec家族若干函数中的一个。）

实现所有exec家族函数的底层内核函数是10079行到10141行的do_execve。do_execve处理三种工作：

- 把一些定义信息从文件读入内存（do_execve把这个工作交给prepare_binprm处理）。
- 准备新的参数和环境——这是C应用程序将它作为argc、argv和envp使用的内容。

- 装载可以解析可执行文件的二进制处理程序，并让它处理剩余的修改内核数据结构的工作。

记住这些任务，现在让我们开始仔细研究一下do_execve。

1. do_execve

10082: 代表在使用exec处理进程时所需要记录的全部信息的数据类型是struct linux_binprm结构（请参看13786行）——我认为binprm是“binary parameters（二进制参数）”的缩写。do_execve处理自己的工作，并使用这种类型的变量bprm同那些负责处理其部分工作的函数进行通信。注意到当do_execve返回时bprm就会被废弃——只有在执行exec时才需要bprm，它并不在该进程的整个生命期中存在。

10087: do_execve通过初始化一个记录新进程参数和环境分配的内存页的微型页表开始执行。它为这个目的总共需要申请MAX_ARG_PAGES（在13780行宏定义为32）个页，在x86平台上每一页是4K，因此参数总共可以使用的空间加起来就是 $32 \times 4K = 128K$ 。作为我个人而言，我很高兴了解到这个内容，因为我偶而会超过这个限定，通常是在一个具有成百个文件的目录下运行`cat x >/tmp/joined`之类的东西的时候——所有这些文件名连接起来可能就超过了128K。我通常是使用xargs程序解决这个问题，但是我现在也可以通过为MAX_ARG_PAGES重新定义一个比较大的值并重新编译内核来解决这个问题。至少现在如果这个问题再困扰我，我也知道该如何增加这一限制了（可能一些热心的读者会重新编写程序来去掉这段糟糕的限制）。所以我非常喜欢拥有内核的源代码。

10091: 下一步是要打开可执行文件。这不是简单地从文件中读出数据——现在的焦点是要确保文件存在，这样do_execve就可以清楚是否有必要继续进行处理。如果这是第一步，而不是首先填充bprm的页表的话，do_execve在执行时有时能够获得很高的边际效应——如果这样失败了，用来初始化页表的时间就浪费了。然而，这只在文件不存在时才有用——这不是普通的情况，不值得优化。

10096: 继续填充bprm，特别是其argv和envc成员。为了填充这些成员，do_execve使用count函数（9480行），它通过使用被传递进来的argv和envp数组计算非空指针的个数。第一个空指针标志着列表结束，因此在到达空指针时就可以得到非空指针的个数并将其返回。起初看起来似乎会造成一些效率的损失：调用do_execve的函数有时早就知道了argv和envp数组的长度。因此可以再给do_execve增加两个整型参数argc和envc。如果这两个参数都是非负的，那么它们就可以分别代表两个数组的长度。但是事情并没有这么简单：count同时要检测它扫描的数组中是否有访问内存的错误发生。强迫（更多的情况是完全信任）do_execve的调用者来对这些内容进行检测是不正确的。所以目前这样的处理方式要更好一些。

10115: 主要使用copy_strings（9519行）把参数和环境变量拷贝到新进程中。copy_strings看起来很复杂，但是它要处理的工作十分简单：把字符串拷贝到新进程的内存空间中，如果需要就给它们分配页。这种复杂性的增长主要出现在对页表的管理需要和跨越内核/用户空间限制的需要，这一点将在第8章中更详细地介绍。

10126: 如果前面的工作可以很好地执行到此处, 最后一步是要为新的可执行程序寻找一个二进制处理程序。如果`search_binary_handler`成功地找到了这种程序, 整个过程就成功运行结束, 并返回一个非负值以说明成功。

10134: 如果程序运行到了此处, 那么前面的几步中肯定发生了错误。系统释放为新进程的参数和环境分配的所有页, 接着必须返回一个负值通知调用者调用过程失败了。

2. `prepare_binprm`

9832: `prepare_binprm`填写`do_execve`的重要部分`bprm`。

9839: 本行开始一些稳健性检测, 例如要确保执行的是文件而不是目录, 并且文件的可执行位已经设置了。

9858: 如果已经被设置过`setuid`和`setgid`位, 就根据它们的提示新进程应该把当前执行的用户作为一个不同的用户(如果`setuid`被设置), 并且/或者把它作为一个不同组的成员(如果`setgid`被设置)。

9933: 最后, `prepare_binprm`从文件中读取前128个字节(而不是像该函数标题注释里说明的一样是前512个字节)到`bprm`的`buf`成员中。

顺便说一下, 这里有一个延续已久的争论: 在13787行, `struct linux_binprm`结构的`buf`成员被声明为是128字节长, 在9933行读入了128字节。但是字面上常量128用在两个地方——没有宏定义表示有必要保持两个数字的一致; 因此, 有可能会对其中一个进行改变而不改变相关的另一个的情况, 这样就很可能摧毁系统。即使不从学术上考虑, 这种忽略在保证效率的基础上是不能防止的——我不能想像出还有什么其他理由。

这是一个很好的对内核做点简短却有用的修改的机会: 在每处这样使用128的地方都使用一个`#define`语句(或者是使用类似于`sizeof(bprm->buf)`的语句)代替; 存在几个其他实例。如果你实验一下, 你就会发现在这种情况下`#define`为什么比`sizeof`要好。(把这种重复出现的神奇数字加以定义和修正对于内核是更好的贡献。但是总体的修正工作要比看起来的困难, 由于正确的对所有相关部分进行定位是很困难的; 让我们一点一点地开始, 最终会将其全部解决。)

3. `search_binary_handler`

二进制处理程序是Linux内核统一处理各种二进制格式的机制, 这是我们需要, 因为不是所有的文件都是以相同的文件格式存储的。一个很合适的例子是Java的`.class`文件。Java定义了一种平台无关的二进制可执行格式——无论它们是在什么平台上运行, 它们的文件本身都是相同的——因此这些文件显然应该和Linux特有的可执行格式一样构建。通过使用适当的二进制处理程序, Linux可以把它们当作是自己特有的可执行文件一样处理。

后面我们会详细介绍二进制处理程序, 现在应该了解一些有关内容, 以便理解`do_execve`是如何发现匹配的。它把这一工作交给`search_binary_handler`(9996行)处理。

10037: 开始遍历处理内核的二进制处理程序链接列表, 依次将`bprm`传递给它们(我们现在并不关心`regs`参数)。更确切地说, 二进制处理程序的链接列表的每一个元素都包含一组指向函数的指针, 这些函数一起提供了对一种二进制格式的支持(13803行定义的`struct linux_binfmt`结构显示了其中包含的内容: 我们感兴趣的部分是装载二进制的部分`load_binary`; 装载共享库的部分`load_shlib`; 创建内核转储

映象的部分`core_dump`)。 `search_binary_handler`简单调用每一个`load_binary`函数,知道其中一个返回非负值,指明它成功识别并装载了文件。`search_binary_handler`返回负值指明发生的错误,其中包括不能找到匹配的二进制处理程序的错误。

10070: 如果10037行开始的循环不能找到匹配的二进制处理程序,本行就试图装载新的二进制格式,它会引起第二次尝试,并应该取得成功。因此整个操作被包含在从10036行开始的两次执行的循环中。

7.10 可执行格式

正如前面一节中说明的一样,不是所有程序都使用相同的文件格式存储,Linux使用二进制处理程序把它们之间的区别掩盖掉了。

Linux当前“本地的”可执行格式(如果“本地”在系统中可以给各种格式提供良好支持)是可执行链接格式(ELF)。ELF只是全部替换了原来的称为`a.out`的格式,替换之前的格式很难说是灵活的——除了有一些其他缺点以外,`a.out`还很难适用于动态链接,这会使得共享库难于实现。Linux仍然为`a.out`保留了一个二进制处理程序,但通常是使用ELF。

二进制处理程序通过某种内嵌在文件开头的“特征序列”(一个特殊字节序列)来识别文件,有时也会通过文件名的一些特性。例如,你会看到的Java处理程序可以保证文件名以`.class`结尾,并且前四个字节是(以十六进制) `0xcafebabe`,这是Java标准所定义的。

下面是2.2版本内核所提供的二进制处理程序(这是在我的Intel系统中的;Linux的其他平台的移植版本,例如PowerPC和SPARC上,需要使用其他的处理程序):

- `a.out` (在文件`fs/binfmt_aout.c`中)——这是为了支持原来风格的Linux二进制文件。这仍然是为了满足一些系统的向后兼容的需要,但是,基本上`a.out`很快就会光荣退役了。
- ELF (在文件`fs/binfmt_elf.c`中)——只是为了支持现在新风格的Linux二进制文件。这在可执行文件和共享库中都广泛使用。最新的Linux系统(例如Red Hat 5.2)一般只预装了ELF二进制文件,但是特殊情况下如果你决定装载`a.out`二进制文件,那么系统也可以对它提供支持。注意即使ELF被作为惯用的Linux本地格式,也要和其他格式一样使用二进制处理程序——内核并没有特殊的偏好。避免特殊情况的惯例能够简化内核代码。
- EM86 (在文件`fs/binfmt_em86.c`中)——帮你在Alpha机器上运行Intel的Linux二进制文件,仿佛它们就是Alpha的本地二进制文件。
- Java (在文件`fs/binfmt_java.c`中)——使你不必每次都麻烦地定义Java字节码的解释程序就可以执行Java的`.class`文件。这种机制和脚本中使用的机制类似;通过把`.class`文件的文件名作为参数传递,处理程序返回来为你执行字节码解释程序。从用户的观点来看,Java二进制文件是作为本地可执行文件处理的。在本章的后面我们会详细介绍这个处理程序。
- Misc (在文件`fs/binfmt_misc.c`中)——这是最明智地使用二进制处理程序的方法,这个处理程序通过内嵌的特征数字或者文件名后缀可以识别出各种二进制格式——但是其最优秀的特性是它在运行期可以配置,而不是只能在编译期可以配置。因此,遵守这些限制,你就可以快速地增加对新二进制文件的支持,而不用重新编译内核,也无须重新启动机器。源程序文件中的注释建议最终使用它来取代Java和EM86二进制处理程序。

- 脚本（在文件fs/binfmt_script.c中）——对于shell脚本、Perl脚本等提供支持。宽松一点说，所有前面两个字符是#!的可执行文件都由这个二进制处理程序进行处理。

在上面这些二进制处理程序中，本书中只对Java和ELF处理程序进行了说明（分别从9083行和7656行开始），因为作为我们关心的基本内容，我们更关心内核如何处理各种不同格式间的区别，而不是每一种单个二进制处理程序的细节（虽然它自己也是一个很有趣的主题）。

例子：Java二进制处理程序

如同前面你看到的一样，do_execve遍历一个代表二进制处理程序的struct linux_binfmt结构的链接列表，调用每个结构的load_binary成员指向的函数直到其中一个成功（当然也可能到试验完了所有的格式为止）。但是这些结构又从何而来呢？函数load_binary是如何实现的？为了寻找这些答案，让我们来看一下fs/binfmt_java.c文件。

这个模块处理一些不是涉及在Web浏览器上使用java_format（9236行）执行的Java程序的Java二进制文件和相关的函数。它使用applet_format（9254行）及相关函数处理Java小程序（applet）。在本节剩余部分的内容中，我们会集中看一下对于非Java小程序的支持；对于Java小程序的支持实际上是相同的。

如果重写fs/binfmt_java.c中的函数来加强Java小程序函数和非Java小程序函数之间的相同代码的数量就更好了。虽然它注定最终要被“misc”二进制处理程序取代，但是现在还只是在讨论，尚未实行。

1. do_load_java

9108: 这是实际处理装载Java的.class文件工作的函数。

9117: 通过检测特征数字0xcafebabe开始，这是因为Java标准规定所有有效的类文件都使用这个字符序列开始。接着开始执行稳健性检测，一直到9147行，确保没有递归调用，而且正在请求执行的可执行文件是以.class结尾的。

9148: 此处，所有的稳健性检测已经通过了。现在，do_load_java取得文件的基本名字，将其和Java字节码解释程序一起放置到程序空间中，并试图执行Java字节码解释程序。

9165: 使用我们在do_execve中见到的同一个进程执行解释程序。特殊情况下，就像查询do_load_java的方法一样，使用search_binary_handler为解释程序查询二进制处理程序（实际上，虽然它不一定非要是ELF二进制文件，但是它也可能是）。

记住其他处理程序不会分配新的struct task_struct结构——我们在使用fork的时候也碰到了这个问题。其他处理程序只是修改现存进程的struct task_struct结构。如果你希望细致地了解这是如何实现的，你的入手点应该是do_load_elf_binary（8072行）——我们关心的部分从8273行开始。

2. load_java

9226: load_java是其他外部对象装载.class文件时所使用的函数。它首先递增内核模块使用的计数（如果作为内核模块编译），随后又将其递减，但是实际的工作是由do_load_java（9108行）处理的。

3. java_format

9236: 通过比较java_format的初始化和struct linux_binfmt结构（13803行）的定义，你可

以看出这个模块没有提供对共享库和内核卸载的支持，只提供了对装载可执行程序的支持；而且这种支持是通过load_java函数实现的。

4. init_java_binfmt

9262: 指向这个模块的项是init_java_binfmt，它把两个静态struct linux_binfmt结构java_format和applet_format的地址压入系统列表中。如果对Java二进制文件的支持被编译进了内核，就在9355行调用init_java_binfmt，或者如果Java二进制文件的支持被作为一个内核模块编译进了内核，就使用kmod任务。

7.11 调度及它们是如何运行的

在应用程序被装载以后，必须获得对CPU的访问。这是调度程序涉及的领域。操作系统调度程序基本上划分为两类：

- 复杂调度程序——运行需要花费相当长的时间，但是希望可以全面提高系统性能。
- 快餐式（quick-and-dirty）调度程序——只是试图处理一些尽量简单的合理的工作就退出，从而进程本身将可以尽可能多地获得CPU。

Linux调度程序是后面一种情况。不要把“quick-and-dirty”解释成贬义的词，虽然实际的情况是：Linux的调度程序在商业和自由领域中都从根本上痛击了其竞争者。

7.11.1 调度函数和调度策略

内核主要的调度函数经过仔细挑选使用schedule这个名字，该函数从26686行开始。这实际上是个很简单的函数，比它看起来还要简单，虽然由于它把三种调度策略合成了一种，其意义显得有些不是很明显。而且对于SMP的支持也增加了一定的复杂性，这一点将在第10章中详细讨论。

通常情况下使用的调度策略和进程有关。给定进程使用的调度算法称为调度策略，这在进程的struct task_struct结构的policy成员中有所反映。一般情况下，policy是SCHED_OTHER、SCHED_FIFO或SCHED_RR其中一个的位集。但是它也可能含有SCHED_YIELD位集，如果进程决定交出CPU——例如，通过调用sched_yield系统调用（请参看sched_yield，27757行）。

SCHED_XXX常量在16196行到16202行宏定义。

16196: SCHED_OTHER意味着传统Unix调度使用它——这不是一个实时进程。

16197: SCHED_FIFO意味着这是一个实时进程，这要遵守POSIX.1b标准的FIFO（先进先出）调度程序。它会一直运行，直到有一个进程在I/O阻塞，因而明确释放CPU，或者是CPU被另一个具有更高rt_priority的实时进程抢占了。在Linux实现中，SCHED_FIFO进程拥有时间片——只有当时间片结束时它们才被迫释放CPU。因此，如同POSIX.1b中规定的一样，这样的进程就像没有时间片一样运行。因此进程要保持对其时间片进行记录主要是为了实现的方便，因此我们就不必使用if(!(current->policy & SCHED_FIFO)) { ... }来弄乱这些代码。还有，这样处理速度可能会快一些——其他实际可行的策略都需要记录时间片，并持续检测是否需要记录时间片会比简单的跟踪它速度更慢。

16198: SCHED_RR意味着这是一个实时进程，要遵守POSIX.1b的RR（循环：round-robin）调度规则。除了时间片有些不同之外，这和SCHED_FIFO类似。当

SCHEM_RR进程的时间片用完后，就使用相同的rt_priority跳转到SCHEM_FIFO和SCHEM_RR列表的最后。

16202: SCHEM_YIELD并不是一种调度策略，而是截取调度策略的一个附加位。如同前面说明的一样，如果有其他进程需要CPU，它就提示调度程序释放CPU。特别要注意的是这甚至会引起实时进程把CPU释放给非实时进程。

1. schedule

26689: prev和next会被设置为schedule最感兴趣的两个进程：其中一个是在调用schedule时正在运行的进程（prev），另外一个应该是接着就给予CPU的进程（next）。记住prev和next可能是相同的——schedule可以重新调度已经获得CPU的进程。

26706: 如同第6章中介绍的一样，这就是中断处理程序的“下半部分”运行的地方。

26715: 内核实时系统部分的实现，循环调度程序（SCHEM_RR）通过移动“耗尽的”RR进程——已经用完其时间片的进程——到队列末尾，这样具有相同优先级的其他RR进程就可以获得时间片了。同时这补充了耗尽进程的时间片。重要的是它为SCHEM_FIFO并不是这样处理的，这样和预计的一样，后面的进程在其时间片偶然用完时就无须释放CPU。

26720: 由于代码的其他部分已经决定了进程必须被移进或移出TASK_RUNNING状态，所以会经常使用schedule——例如，如果进程正在等待的硬件条件已经发生了——所以如果必要，这个switch会改变进程的状态。如果进程已经处于TASK_RUNNING状态，它就无须处理了。如果它是可以中断的（等待信号），并且信号到达了进程，就返回TASK_RUNNING状态。在所有其他情况下（例如，进程已经处于TASK_UNINTERRUPTIBLE状态了），应该从运行队列中将进程移走。

26735: 将p初始化为运行队列中的第一个任务；p会遍历队列中的所有任务。

26736: c记录了运行队列中所有进程的最好“goodness”——具有最好“goodness”的进程是最易获得CPU的进程（我们很快就会讨论goodness）。goodness值越高越好，一个进程的goodness值永远不会为负——这是Unix用户经常见到的一种奇异情况，其中较高的优先级（通常称为完美“niceness”级）意味着进程会较少地获得CPU时间（至少这在内核中是有意义的）。

26757: 开始遍历执行任务列表，跟踪具有最好goodness的进程。注意只有在当前记录被破坏而不是当它简单地被约束时它才会改变最好进程的概念。因此，由于对队列中第一个进程的原因，这种约束就会被打破了。

26758: 这个循环中只考虑了唯一一个可以调度的进程。can_schedule宏的SMP版本在26568行定义；其定义使SMP内核只有任务尚未在CPU上运行，才会把调度作为该CPU上的一个任务（这样具有完美的意义——在几乎不必要的任务中造成混淆完全是一种浪费）。UP版本在26573行，它总是真值——换言之，在UP的情况下，运行队列中的每一个进程都需要竞争CPU。

26767: 值为0的goodness意味着进程已经用完它的时间片或者它已经明确说明要释放CPU。如果所有运行队列中的所有进程都具有0值的goodness，在循环结束后c的值就是0。在这种情况下，schedule要重新计算进程计数器；新计数器的值是原来

值的一半加上进程的静态优先级——除非进程已经释放CPU，否则原来计数器的值都是0，`schedule`通常只是把计数器重新初始化为静态优先级。（中断处理程序和由另外一个处理器引起的分支在`schedule`搜寻`goodness`最大值时都将增加此循环中的计数器，因此由于这个原因计数器可能不会为0。虽然这有些罕见。）

调度程序不必麻烦地重新计算现在哪一个进程具有最高的`goodness`值；它只是调度前面循环中遇到的第一个进程。此时，这个进程是它发现的第一个具有次高`goodness`值（0）的进程，因此`schedule`就能够计算出自己现在和以后所应该运行的任务（记住，这就是“quick-and-dirty”的思想）。

26801: 如果`schedule`已经选择了一个不同于前面正在运行的进程来调度，那么它就必须挂起原来的进程并允许新的进程运行。这是通过后面我们将介绍的`switch_to`处理的。`switch_to`的一个重要结果对于应用程序开发者来说可能显得有些奇怪：对于`schedule`的调用并不返回。也就是它不是立即返回的；在系统条件判断语句返回到当前任务时调用就会返回。作为一个特殊情况，当任务退出而调用`schedule`时，对于`schedule`的调用从不会返回——因为内核不会返回已经退出的任务。还有另外一种特殊情况，如果`schedule`不会调度其他进程——也就是说，如果在`schedule`结束时`next`和`prev`是相同的——那么上下文中的跳转不会执行，`schedule`实际上不会立即返回。

26809: `schedule`末尾的`__schedule_tail`和`reacquire_kernel_lock`函数在UP平台上不执行任何操作，现在我们就已经看完了调度程序的内核。顺便说一下，为了确保你已经正确地理解了这些代码，自己证明以下的特性：如果运行队列为空，那么下面就会调用`idle`任务。

2. `switch_to`

`switch_to`处理从一个进程到下一个进程的跳转，称为上下文跳转（context-switching）；这是在不同处理器上会有不同处理的低级特性。有趣的是，在x86平台上内核开发人员使用软件处理大多数的上下文跳转，这样就忽略了一些硬件的支持。这种机制背后的原因在`__switch_to`函数（2638行）上面的标题注释中有所说明，这个函数和`switch_to`宏（12939行）一起处理上下文跳转。

由于很多上下文跳转要依赖于对内核处理内存方式的正确理解，这在下一章中才会详细介绍，本章只是稍微涉及一点。上下文跳转背后的基本思想是记忆当前位置和将要到达的位置——这是我们必须保存的当前上下文——接着跳转到另外一个前面已经存储过的上下文。通过使用很少的汇编代码，`switch_to`宏保存了后面将要介绍的上下文两个重要的部分。

12945: 首先，`switch_to`宏保存ESP寄存器的内容，它指向进程的当前堆栈。堆栈在下一章中将深入介绍；现在你只需简单了解堆栈中保存的局部变量和函数调用信息。`switch_to`宏也保存EIP寄存器的内容，这是进程的当前指令指针——如果允许继续运行时所执行的下一条指令的地址。

12948: 把`next->tss.eip`——保存指令的指针——压入返回堆栈，记录当后面紧跟的跳转到`__switch_to`的`jmp`返回时的返回地址。这样做的最终结果是当`__switch_to`返回时，我们又回到了新的进程。

12949: 调用`__switch_to`（2638行），它完成段寄存器和页表的保存和恢复工作。在你阅

读完第8章以后这些特征字就更有意义了。

12955: tss代表任务状态(task-state)段,这是Intel使用的支持硬件上下文跳转的CPU特性的术语。虽然内核代码使用软件实现上下文跳转,但是开发人员仍然会使用TSS来记录进程的状态。struct task_struct结构的tss成员的类型是struct thread_struct结构,本书由于篇幅所限,忽略了它的定义。其成员仅仅对应于x86的TSS——成员是为EIP和ESP而存在的,如此而已。

7.11.2 计算goodness值

进程的goodness值通过goodness函数(26388行)计算。goodness返回下面两类中的一个值:1 000以下或者1 000以上。1 000和1 000以上的值只能赋给“实时”进程,从0到999的值只能赋给“普通”进程。实际上普通进程的goodness值只使用了这个范围底部的一部分,从0到41(或者对于SMP来说是0到56,因为SMP模式会优先照顾等待同一个处理器的进程)。无论是在SMP还是在UP上,实时进程的goodness值的范围都是从1 001到1 099。

有关这两类goodness结果的重要的一点是:该值在实时系统中的范围肯定会比非实时系统的范围要高(因此偏移量(offset)是100而不是1000)。POSIX.1b规定内核要确保在实时进程和非实时进程同时竞争CPU时,实时进程要优先于非实时进程。由于调度程序总是选择具有最大goodness值的进程,又由于任何尚未释放CPU的实时进程的goodness值总是比非实时进程的goodness大,Linux对这一点的遵守是很容易得到证明的。

尽管在goodness上面的标题注释中有所说明,该函数是从不会返回-1 000的,也不会返回其他的负值。由于idle进程的counter值为负,所以如果使用idle进程作为参数调用goodness,就会返回负值,但这是不会发生的。

goodness只是一个简单的函数,但是它是Linux调度程序必不可少的部分。运行队列中的每个进程每次执行schedule时都可能调用它,因此其执行速度必须很快。但是,如果一旦它调度失误,那么整个系统都要遭殃了。考虑到这些冲突压力,我想改进现有的系统是相当困难的。

goodness

26394: 如果进程已经释放了CPU,就返回0(在清除SCHED_YIELD位之后,这是因为进程只可能有一次想释放CPU,现在它已经的确把CPU释放了)。

26402: 如果这是一个实时进程,goodness返回的值就属于数值较高的一类;这要精确地依赖于rt_priority的值。

26411: 此处,代码识别出这是一个非实时进程,它把goodness(在这个函数中被称为weight)初始化为其当前的counter值,这样如果进程已经占用CPU一段时间了,或者进程开始的优先级比较低,那么进程就不太可能获得CPU。

26412: 如果权值weight的值为0,那么进程的计数器就已经被用完了,因此goodness就不会再增加加权因素。其他进程就可以有机会运行。

26418: 尽力优先考虑等待同一个处理器的进程(只在SMP系统中是这样——顺便说一下,考虑一下运行在一个双处理器的系统中的三个进程的实现情况)。

26423: 给相关的当前进程或者当前线程增加了一些优点;这有助于合理使用缓存以避免使用昂贵的MMU上下文跳转。

26425: 增加进程的priority。这样, goodness (和其他类似的调度程序) 就对较高优先级的进程比对较低优先级的进程更感兴趣, 即使在前面进程已经部分用完了它们的时间片也是这样。

26428: 返回计算出来的goodness值。

7.11.3 非实时优先级

每个Linux进程都有一个优先级, 这是从1到40的一个整数, 其值存储在struct task_struct结构的priority成员中 (对于实时进程, 在struct task_struct结构中还会使用一个成员——rt_priority成员。随后很快就会对它进行更详细的讨论)。它的范围使用PRIO_MIN (在16094行宏定义为-20) 和PRIO_MAX (在16095行宏定义为20) 限定——理论上来说的确是这样。但是非常令人气愤的是, 控制优先级的函数——sys_setpriority和sys_nice——并没有注意到这些明显的常量, 却相反宁愿使用一些固定的值 (它们也使用最大的完美值19, 而不是20)。基于这个原因, PRIO_MIN和PRIO_MAX两个常量并没有广泛使用。不过这又是一个热心读者改进代码的机会。

由于已经在文档中说明sys_nice (27562行) 要废弃不用了——可能会使用sys_setpriority来重新实现——我们就忽略前面一个函数, 只讨论后面一个。

1. sys_setpriority

29213: sys_setpriority使用三个参数——which、who和niceval。which和who参数提供了一种可以用来指定一个给定用户所拥有的单个进程、一组进程或者所有进程的方法。who要根据which的值做出不同的解释; 它会作为一个进程ID、进程组ID或者用户ID读取。

29220: 这是确保which有效地进行稳健性检测。我认为这里的模糊不清是不必要的。如果我们不使用

```
if ( which > 2 || which > 0 )
```

而使用如下语句

```
if ( which != PRIO_PROCESS && which != PRIO_PGRP && which != PRIO_USER )
```

或者至少是

```
if ( which > PRIO_USER || which < PRIO_PGRP )
```

另外, 在29270行也可以使用同样的方法。

29226: niceval是使用用户术语定义的——也就是说, 它是在从-20到19的范围中, 而不是像内核中使用的一样, 是在从1到40的范围中。如同变量名说明的一样, 这是一个完美的值, 但不是一个优先级。因此, 为了实现这种转化, sys_setpriority应该跳过一些循环, 同时要截断niceval超出允许范围的值。

我承认自己被这段代码的复杂性所困扰着。使用实际上使用的DEF_PRIORITY的值——20, 以下的简化代码显然可以实现相同的效果:

```
if ( niceval < -19 )
    priority = 40;
else if ( niceval > 19 )
    priority = 1;
else
    priority = 20 - niceval;
```

在保持比`sys_setpriority`中的代码更简单的同时，我的实现方法中当然也要考虑`DEF_PRIORITY`。因此，或者我严重误解了一些内容，或者就像我提出的代码本身，它根本就不需要这么复杂。

29241: 循环遍历系统的任务列表中的所有任务，执行它可以允许的修改。`proc_sel` (29190行) 说明了给定的进程是否对所提供的`which`和`who`值满意，可以用它来选择进程；由于`sys_getpriority`也要使用这个函数，所以它也是`sys_setpriority`应该考虑的一个因素。

对于读取和设置单个进程优先级的普通情况（如果没有其他问题，就通过提早退出`for_each_task`循环），`sys_setpriority`和`sys_getpriority` (29274行开始的代码和此处有相似的内部循环) 都对它有一点加速作用。`sys_setpriority`可能不会被很频繁地调用，但是`sys_getpriority`却可能被很频繁地调用，因而这样的努力是值得的。

2. `update_process_times`

`sys_setpriority`只会影响进程的`priority`成员——也就是其静态优先级。回忆一下，进程也是具有动态优先级的，这由`counter`成员表示，这一点我们在对`schedule`和`goodness`的讨论中就已经清楚地看到了。我们已经可以看出在调度程序发现`counter`值为0时，`schedule`会周期性地根据其静态优先级重新计算每一个进程的动态优先级。但是我们仍然还没有看到另外一部分困扰我们的问题：`counter`是在哪里被递减的？它是怎样达到0的？

对于UP，答案与`update_process_times` (27382行) 有关（和前面一样，我们把对于SMP问题的讨论延迟到第10章）。`update_process_times`是作为`update_times` (27412行) 的一部分被调用的，它还是第6章中讨论的定时器中断的一部分。作为一个结果，它被相当频繁地调用——每秒钟100次（当然，这只是相对于人类的能力来说是相当频繁的，对于CPU来说却是很慢的）。在每一次调用的时候，它都会把当前进程的`counter`值减少从上次以来经过的“滴答”的数目（百分之一秒——请参看第6章）。通常，这只是一次滴答，但是如果内核正忙于处理中断，那么内核就可能忽略定时器的滴答。当计数器减小到0以下时，`update_process_times`就增加`need_resched`标志，说明这个进程需要重新调度。

现在，由于进程缺省的优先级（使用内核优先级的术语，而不使用用户空间的完美值）是20，缺省情况下进程得到一个21次滴答的时间片（的确这是21次滴答，而不是20次滴答，因为进程直到其动态优先级减少到0以下时才会为重新调度做出标记）。一次滴答是百分之一秒，或者是10 μ s，因此缺省的时间片就是210 μ s——大约是五分之一秒，这在16466行有确切的描述。

我发现这个结果十分奇怪，因为原来以为理想的反应迅速的系统应该具有小很多的时间片——实际上我对这一点的认识是如此强烈以至于开始的时候我还以为文档的说明发生了错误。但是，回顾一下，我觉得自己也不应该奇怪。毕竟，进程不会频繁地耗尽其整个时间片，因为它们经常都会因为I/O的原因而阻塞。在几个进程都绑定在CPU上时，在它们之间太频繁地跳转是没有必要的（特别是在诸如x86之类的CPU上，这里的上下文跳转的代价是相当高的）。最后，我必须承认我从来没有注意到自己留意Linux逻辑单元的响应的迟缓特性，因此我觉得210 μ s的时间片是个不错的选择——即使这在最初的时候看起来是太长了。

如果由于某些原因你需要时间片比当前最大值还长（410 μ s，优先级上长到了40），你可以简单地使用`SCHED_FIFO`调度策略，在你准备好以后就可以释放CPU（或者重新编写`sys_setpriority`和`sys_nice`）。

7.11.4 实时优先级

Linux的实时进程增加了一级优先级。实时优先级保存在struct task_struct结构的rt_priority成员中，它是一个从0到99的整数（值0意味着进程不是实时进程，在这种情况下其policy成员必须是SCHED_OTHER）。

实时任务仍然使用相同的counter成员作为它们的非实时的计数器部分，因此，它们的动态优先级处理方式是一样的。实时任务为了相同目的甚至使用非实时任务使用的priority成员，作为当时间片用完时用来补充counter值使用的值。为了清晰起见，rt_priority只是用来对实时进程划分等级以对它们进行区分——否则它们的处理方式就和非实时进程相同了。

进程的rt_priority被设定为使用POSIX.1b规定的函数sched_setscheduler和sched_setparam（通常只有root才可以调用这两个函数，这一点我们在讨论权能时会看到）设置其调度策略。注意，这意味着如果具有修改权限，进程的调度策略在进程生命期结束以后就可以改变。

实现这些POSIX函数的系统调用sys_sched_setscheduler（27688行）和sys_sched_setparam（27694行）都会把实际的工作交给setscheduler（27618行）处理，这个函数我们下面就介绍。

setscheduler

27618: 这个函数的三个参数是目标进程pid（0意味着当前进程）、新的调度策略policy和包含附加信息的一个结构param——它记录了rt_priority的新值。

27630: 在一些稳健性检测之后，setscheduler从用户空间中得到提供的struct sched_param结构的备份。在16204行定义的struct sched_param结构只有一个成员sched_priority，它就是调用者为目标进程设计的rt_priority。

27639: 使用find_process_by_pid（27608行）找到目标进程，如果pid是0，这个函数就返回一个指向当前任务的指针；如果存在指向具有给定PID进程，就返回指向该进程的指针；或者如果不存在具有这个PID的进程，就返回NULL。

27645: 如果policy参数为负时，就保留当前的调度策略。否则，如果这是个有效值，那么现在就可以将其接收。

27657: 确保优先级没有越界。这是通过使用一点小技巧来加强的。该行只是第一步，它被用来确保所提供的值没有大得超出了范围。

27659: 现在已经确知新的实时优先级位于0到99的范围之内。如果policy是SCHED_OTHER，但是新的实时优先级不是0，那么这个测试就失败了。如果policy指明了一个实时调度程序，但是新的实时优先级是0（如果这里它不是0，就应该是从1到99），测试也会失败。否则，测试就能成功。这虽然并不是很易读，但它确实是正确的、最小的，（我想）速度也很快。我不确定这里我们是否对速度有所苛求，但是——到底一个进程需要多长时间来设置它的调度程序？下面的代码就应该具有更好的可读性，而且当然也不会太慢：

```
if (policy == SCHED_OTHER) {
    if (lp.sched_priority != 0)
        goto out_unlock;
} else { /* SCHED_FIFO or SCHED_RR */
    if ((lp.sched_priority < 1) ||
        (lp.sched_priority > 99))
        goto out_unlock;
}
```

- 27663: 不是每一个进程都可以设置自己的调度策略和其他进程的调度策略。如果所有进程都可以设置自己的调度策略, 那么任何进程都可以简单地设置自己的调度策略为 `SCHED_FIFO`, 并进入一个无限循环来抢占 CPU, 这样必然会锁定系统。显然, 这种做法是不能够允许的。因此, 只有进程拥有这样处理的权能时, `setscheduler` 才会允许进程设置自己的调度策略。权能在下一节中将比较详细地介绍。
- 27666: 在相同的行中, 我们不希望别人可以修改其他用户进程的调度策略; 普通情况下, 只允许你修改你自己所有的进程的调度策略。因此, `setscheduler` 要确保, 用户或者是设置自己所有的进程的处理程序, 或者具有修改其他进程的调度策略的权能。
- 27672: 这里才是 `setscheduler` 实际工作的地方, 它在目标进程的 `struct task_struct` 结构中设置 `policy` 和 `rt_priority`。如果该进程在运行队列中 (这通过检测其 `next_run` 成员为非空来测试), 就将它移到运行队列的顶部——这比较容易令人感到迷惑; 可能这有助于 `SCHED_FIFO` 进程实现对 CPU 的抢占。进程为重新调度做出标记, `setscheduler` 清空并退出。

7.12 遵守限制

内核经常需要决定是否允许进程执行某个操作。进程可能被简单地禁止执行某些操作, 但却被允许在受限的环境中执行一些别的操作; 这些操作基本上可以由权能表示, 并且/或者可以从用户 ID 和组 ID 中推导出来。在其他时间, 允许进程处理一些操作, 但只是在受限的环境中——例如, 它对 CPU 的使用必须受到限制。

7.12.1 权能

在前面一节中, 你已经看到了一个检测权能的例子——实际上是有两次相同的权能。这是 `CAP_SYS_NICE` 权能 (14104 行), 它决定是否应该允许进程设置优先级 (完美级别) 或调度策略。由于这比仅仅的完美级别要更适用, `CAP_SYS_NICE` 是一个误用的位——虽然很容易就可以看出设置调度策略和相关的概念是紧密相关的, 而且你一般也不会要一个权能而不要另外一个权能。

每一个进程都有三个权能, 它们被存储在进程的 `struct task_struct` 结构中 (在 16400 行到 16401 行中):

- `cap_effective`——有效位集合。
- `cap_permitted`——允许位集合。
- `cap_inheritable`——继承位集合。

进程权能的有效位集合是当前可以处理的内容的集合; 这是通过广泛使用的 `capable` 函数检测的集合, 这个函数在 16738 行定义。

允许位集合规定进程正常可以被赋予的权能。这个集合通常不会增加——只有一种情况例外: 如果一个进程具有 `CAP_SETPCAP` 权能, 那么它就可以将自己的允许位集合中的任何权能赋给其他进程, 即使目标进程还没有拥有这个权能。

如果一个权能在允许位集合中, 但是并不在有效位集合中, 那么进程现在还没有马上拥有权能, 但是它可以通过请求权能而获得。为什么要麻烦地区别它们呢? 在本章开始我们第

一次讨论权能的时候，我们简单地考虑了一个简单的例子：一个长期运行的进程只是偶然需要权能，而不是所有情况下都需要。为了保证进程不会偶然缺少权能，进程可以一直等待，直到它需要权能，接着请求权能，执行有权限的操作，并再次取消权能。这种方法比较安全。

继承位集合不像你想象的那么简单。它不是祖先继承在执行fork的同时传递的权能集合——实际上，在创建的那一刻（也就是紧随着fork），子孙进程的权能的三个集合和其祖先的三个权能集合都是相同的。相反，继承位集合在exec运行期间才会起作用。进程在调用exec之前的继承位集合有助于决定它的允许位集合和继承位集合，它们在exec执行结束以后也会保留下来——详细的介绍请参看compute_creds（9948行）。注意在exec之后权能是否保留要部分依赖于进程的继承位集合；它还要部分依赖于文件本身中的权能位集合（或者不管怎样，这至少是一个计划——虽然这种特性还没有完全实现）。

顺便提一下，注意到允许位集合必须总是有效位集合和继承位集合的超集（superset）（或者和有效位集合相同）。（只有对于有效位集合这才是严格正确的。一个进程可能会扩展另外一个进程的继承位集合从而它不再是其允许位集合的子集，但是就我知道的来说，这是无意义的，因此我们从现在就开始忽略这种可能性。）然而，可能和你希望相反，有效位集合不一定要是继承位集合的超集（或者和继承位集合相同）。也就是说，在exec结束以后，进程可能会拥有一个以前不曾有过的权能（虽然这个权能必须在其允许位集合中——也就是说，这是一个原来进程自己可能已经得到了的权能）。我认为这种需要只是局部的，这样进程就不需要暂时获得不需要的权能，而能够获得足以执行exec程序的权能。

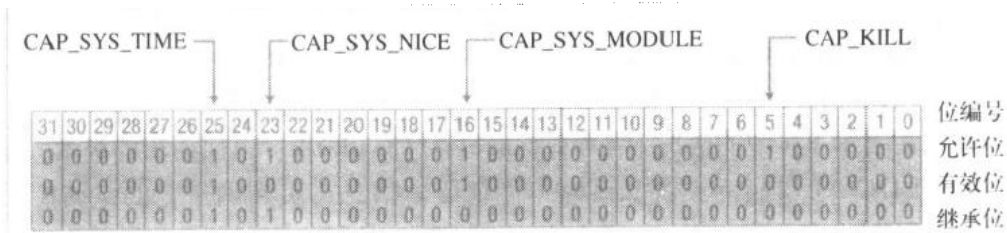


图7-4 权能集

图7-4说明了各种可能性。它显示了一个理想进程的三种权能集合，位从左到右计数。允许进程可以获得CAP_KILL权能，这样就允许它不考虑其他属主而杀掉别的进程，但是它还没有立即拥有权限，而且也不会exec执行过程中自动获得。目前它具有增加和删除内核模块的权能（使用CAP_SYS_MODULE），但是同样也不会exec执行过程中自动获得。它可以获得CAP_SYS_NICE权能，但是直到exec执行完后才会获得（假定文件权能位允许）。最后，它可以立即修改系统时间（CAP_SYS_TIME），但是也是只有通过exec才能获得这个权能。除非其他具有CAP_SETPCAP权能的进程提供了这个权能，否则这个进程不能获得这个权能，它可能执行的其他进程也不可能获得这个权能。

保证这些不同性质的代码主要是在kernel/capability.c中，从22460行开始。两个主要的函数是读取权能的函数sys_capget（22480行）和设置权能的函数sys_capset（22592行）；它们在下一节中讨论。通过exec继承的权能使用fs/exec.c的compute_creds（9948行）处理，这一点已经介绍过了。

当然，root肯定拥有所有的权能。内核权能特性给root提供了一种规则的方法来有所选择地只把需要的权能赋给特定的进程，而不用考虑该进程是否作为root用户运行。

权能一个有趣的特性是它们可以用来改变系统的“风格”。例如，为所有的进程设置CAP_SYS_NICE权能会使所有进程都增加自己的优先级（并设置它们的调度规则，等等）。如果修改了系统中每一个进程的运行方式，那么就改变了系统本身。自己设想一下发明一种新的、可以通过更令人兴奋的方式修补系统的内核权能。

权能的尚未为人所知的优点是它们使源程序代码非常清晰。当检测当前进程是否允许设置系统时间时，却反而要检测当前进程是否以root运行，这种方式看起来似乎有些不好。权能使我们了解它们的意思。权能的存在甚至还能够使查询进程的用户ID或组ID的代码更为清晰，这是因为，这样的处理代码对问题的答案比较感兴趣，而不是对从其中可以推导出的结论更感兴趣。否则，代码应该已经使用权能查询它需要了解的内容了。由于权能更加一致地和Linux内核代码结合起来，这种特性就变得更加可靠了。

13916: 内核可以识别的权能从这里开始。因为这些宏定义的解释已非常详细了，我们就不再详细介绍其中每一个的内容了。

14153: 赋给每一个权能的数字是简单的连续整数，但是由于要使用无符号整数中的位来编址，所以就使用CAP_TO_MASK宏把它们转化为2的幂。

14154: 设置和检测权能的核心只是一系列位操作；从这里到include/linux/capability.h的末尾定义了用来使位操作更为清晰的宏和内联函数。

1. sys_capget

22480: sys_capget有两个参数：header和dataptr。header是cap_user_header_t类型（13878行）的，它是一个指向定义权能使用的版本和目标进程的PID的指针；dataptr是cap_user_data_t类型（13884行）的，它也是一个指向结构类型的指针——这个结构包含有效位、允许位和继承位集合。sys_capget通过第二个指针返回信息。

22492: 在版本不匹配的情况下，sys_capget通过header指针返回使用的版本，接着返回EINVAL错误（或者如果它不能把版本信息拷贝到调用者的空间中，就返回EFAULT）。

22509: 定义调用者希望了解其权能的进程；如果pid不是0，也不是当前进程的PID，sys_capget就要查询它。

22520: 如果它能装载目标进程，它就把自己的权能拷贝到临时变量data中。

22530: 如果所有工作到目前为止都运行良好，它就把权能拷贝回用户空间中由dataptr参数提供的地址中。然后，它返回error变量——通常如果一切运行良好，这就是0；否则就是一个错误号。

2. sys_capset

22592: sys_capset的参数几乎和sys_capget的参数类似。不同之处是data（不再称为dataptr了）是常量。

22600: 和sys_capget一样，sys_capset确保内核和调用进程使用一致的权能系统的版本。如果版本不一致，就拒绝尝试请求。

22613: 如果pid不是0，就说明调用者希望设置其他进程的权能，在大多数情况下这种尝试都会遭到拒绝。如果调用者具有CAP_SETPCAP权能，这意味着允许它设置任何进程的权能，sys_capset就允许这种尝试。这种测试的前面部分有些太受限制了：如果它和当前进程的pid相等，就接收这个pid。

- 22616: 从用户空间中拷贝新的权能, 如果失败就返回错误。
- 22627: 和22509行开始的`sys_capget`代码类似, `sys_capset`定义了调用者希望了解其权能的进程。这就是两者的区别所在, `sys_capset`为了说明进程组(或者是-1指明是所有进程), 也允许其pid值为负。在这种情况下, `target`仍然设置为`current`, 因此当前进程的权能要在后面的计算中使用。
- 22642: 现在它必须保证合法地使用新的权能位集合, 而且在内部保持一致。除非这种新特性在调用者的允许位集合中, 否则这种测试会验证出新进程的继承位集合没有包含任何新鲜的东西。因此, 它不会放弃调用者尚未拥有的任何权能。
- 22650: 类似地, `sys_capset`也要确保除非调用者的允许位中包含新的特性, 否则目标进程的允许位集合也不会包含尚未具有的特性。因此, 它也不会放弃调用者尚未拥有的任何权能。
- 22658: 回想一下进程的有效位集合必须是其允许位集合的一个子集。这种性质在这里得到了加强。
- 22666: `sys_capset`现在已经准备对请求做出修改。负的pid值意味着它正在给不止一个进程修改权能——如果pid是-1, 就是所有的进程; 如果pid是其他的负值, 就是一个进程组中的所有进程。在这些情况下, 实际工作分别由`cap_set_all`(22561行)和`cap_set_pg`(22539行)完成; 这只是通过一些适当的进程集合简单地循环, 按照和单个进程相同的方法覆盖掉集合中的每一个进程的权能位集合。
- 22676: 如果pid是正数(或者是0, 表示当前进程), 权能位集合只赋给目标进程。

7.12.2 用户ID和组ID

尽管权能功能强大、十分有用, 但它并不是你实现访问控制的唯一武器。在一些情况中, 我们需要了解哪个用户正在运行一个进程, 或者进程是作为哪个用户来运行。用户使用整型的用户ID来区别, 一个用户可以属于一个组或者多个组, 每一个都有自己特有的整型ID。

有两种风格的用户ID和组ID: 实际的ID和有效的ID。一般说来, 实际用户(或组)ID为你说明了哪个用户创建了进程, 有效用户(或组)ID为你说明在情况改变时进程作为哪个用户运行。由于访问控制的决定要更多依赖于进程作为哪儿用户运行, 而不是哪个用户创建了这个进程, 因此内核会比检测实际用户(和组)ID更加频繁地检测有效用户(或)ID——在我们现在关心的代码中就是这样处理的。`struct task_struct`结构中的相关成员是`uid`、`euid`、`gid`和`legid`(16396行到16397行)。注意用户ID和用户名不同, 前者是一个整数, 而后者是一个字符串。`/etc/passwd`文件把这两者关联起来。

让我们再回到`sys_setpriority`, 并看一下前面我们忽略了从29244行和29245行的一些代码。`sys_setpriority`通常执行的操作都是让用户降低自己进程的优先级, 但是不能降低其他用户进程的优先级——除非用户具有`CAP_SYS_NICE`权能。因此, `if`表达式的前面两个术语要检测目标进程的用户ID是否和`sys_setpriority`的调用者的实际用户ID或者有效用户ID匹配。如果两个都不匹配, 并且`SYS_CAP_NICE`没有设置, `sys_setpriority`就正确地拒绝这种尝试。

如果允许, 进程可以使用`sys_setuid`和`sys_setgid`(29578行和29445行)和其他一些函数修改它们的用户ID和组ID。用户ID和组ID也可以通过执行可执行的`setuid`或`setgid`可执行程序进行修改。

7.12.3 资源限制

可以要求内核限制一个进程使用系统中的各种资源，包括内存和CPU时间。这可以通过 `sys_setrlimit` 实现（30057行）。通过浏览 `struct rusage` 结构（16068行）可对支持限制有一个基本的概念。进程特有的限制在 `struct task_struct` 结构中记录——还可能在什么地方？请参看 16404行的 `rlim` 数组成员。

违反限制的结果根据限制的不同也会有所不同。例如，对于 `RLIMIT_NPROC`（在本书的源程序代码中没有包括）——有关一个用户可以拥有的进程数目的限制——和你在 23974 行中看到的一样，结果仅仅和 `fork` 失败一样。超出其他限制的后果对于一些进程可能比较严重，这样进程会被杀死（请参看 27333 行）。进程可以使用 `sys_getrlimit`（30046 行）请求特殊限制，或者使用 `sys_getrusage`（30143 行）请求资源使用限制。

在 30067 行中，注意进程可以随意减少自己的资源限制，但是它增加自己的资源限制时只能增加到一个最大值，这个值可以根据每一个资源限制进行具体设置。因此，当前的资源限制和所有的资源限制是分别记录的（使用在 16089 行定义的 `struct rlimit` 结构的 `rlim_cur` 成员和 `rlim_max` 成员）。然而具有 `CAP_SYS_RESOURCE` 权能的进程可以覆盖这个最大值。

这和优先级的规则不同：可以允许进程减小自己的优先级，但是为增加其优先级需要特殊许可，即使是它减少了自己的优先级接着又要马上增加它也是如此。当前资源限制和最大资源限制这两个相互关联的概念并没有反映在内核优先级的调度中。还有，注意到一个进程可以改变另一个进程的优先级（当然是假定它有权这样处理），但是一个进程只能修改自己的资源限制。

7.13 进程的结束

我们已经看到进程是如何生成的，怎样给它们赋予各自的生存周期。现在我们应该看一下它们是如何消亡的。

7.13.1 `exit`

同第 6 章中介绍的一样，你可以通过给进程发送信号 9 强行杀掉进程，但是更普通的情况是进程自动退出。进程通过调用系统调用 `exit` 自动退出，它在内核中是由 `sys_exit` 实现的（23322 行）（顺便说一下，当 C 程序从它的 `main` 部分返回时，就会潜在调用 `exit`）。当进程退出时，内核释放所有分配给这个进程的资源——内存、文件，等等——当然，还要停止给它继续使用 CPU 的机会。

然而内核不能立即回收代表进程的 `struct task_struct` 结构，这是因为该进程的祖先必须能够使用 `wait` 系统调用查询其子孙进程的退出状态。`wait` 返回它检测出的死亡状态的进程的 PID，因此，如果死亡的子孙进程在祖先进程仍在等待时就已经重新分配了，那么应用程序就会被搞乱（和其他问题一样，同一个祖先结束时可以有两个具有相同 PID 的子孙进程——一个进程是活动的，另一个进程是死亡的，祖先进程也不知道哪一个已经退出了）。因此，内核必须保留死亡子孙进程的 PID 直到 `wait` 发生为止——这通过完整地保持其 `struct task_struct` 结构来自动实现；分配 PID 的代码就不用再查询它在任务列表中发现的进程是否是活动的。

处于这种在两种状态之间的进程——它既不是活动的，也没有真正死亡——被称为僵进程（zombies）。那么 `sys_exit` 的任务就是把活动进程转化为僵进程。

`sys_exit`本身的工作很少；它只是简单地把退出代码转化为`do_exit`希望的格式，接着就会调用`do_exit`，由它来处理实际的工作（`do_exit`也会作为发送信号的一部分来调用，这一点我们在第6章中已经讨论过了）。

- 23267: `do_exit`把退出代码作为参数处理，在其返回类型之前使用特殊符号 `NORET_TYPE`。虽然现在`NORET_TYPE`（14955行）定义为空——因此它也就不起作用——但是原来它经常被定义为`__volatile__`，用来提示gcc该函数不会返回。了解了这点知识，gcc就执行一些额外的优化工作并取消有关函数不能成功返回的警告信息。使用其新的定义，`NORET_TYPE`对于编译器就没有用处了，但是它仍然给我们传递了很多有用的信息。
- 23285: 释放它的信号量和其他System V IPC结构，这一点我们将在第9章中介绍。
- 23286: 释放分配给它的内存，这一点我们在第8章中介绍。
- 23290: 释放分配给它的文件，很快就会讨论。
- 23291: 释放它的文件系统数据，它超出了本书的范围。
- 23292: 释放它的信号量处理程序表，这一点我们在第6章中介绍过了。
- 23294: 剩下的任务是进入`TASK_ZOMBIE`状态，其退出代码被记录下来以供将来祖先进程使用。
- 23296: 调用`exit_notify`（23198行），它会警告当前退出任务的祖先进程和其进程组中的所有成员，该进程正在退出。
- 23304: 调用`schedule`（26686行）释放CPU。这个对于`schedule`的调用从来不会返回，这是因为它跳转到下一个进程的上下文，从不会再跳转回来，因此这是现在退出的进程的最后一次拥有CPU的机会。

`__exit_files`

进程如何和文件交互不是本书的主题。但是我们应该快速浏览一下`__exit_files`（23109行），因为这有助于我们理解`__clone`函数，这个函数使祖先进程和子孙进程可以共享特定的信息。祖先进程和子孙进程可以共享的一种信息是它们打开的文件列表。和当时说明的一样，Linux使用引用计数器规则来保证进程退出之后可以正确地处理扫尾工作。这里就有个扫尾工作的很好的例子。

- 23115: 假设进程已经打开了文件（几乎总会是这样的），`__exit_files`会递减原来存储在`tsk->files->count`中的引用计数器。诸如`atomic_dec_and_test`之类的原子操作将在第10章详细介绍；知道`atomic_dec_and_test`（10249行）递减其参数值并当参数新值是0时返回真值就足够了。因此，如果`tsk`的对于目标`struct files_struct`结构的引用是最后一次时，这就是正确的（如果这是一个私有拷贝，没有和其他任何进程共享，那么引用计数器的初始值就是1，当然它被减小为0）。
- 23116: 在释放记录进程的打开文件的内存之前，必须把这些文件都关闭，这是通过调用`close_files`（23081行）实现的。
- 23118: 释放保留进程的文件描述符数组`fd`的内存，这个数组是`files`的一个子域。打开文件（`NR_OPEN`，在15067行中定义为1 024）的最大数量要加以选择，这样本行中的if测试就能正确——`fd`数组必须刚好适合一个内存页的大小。这样做可以使得内存的分配（或释放）速度快许多；否则，`__exit_files`只好使用更通用但是速度

却慢得多的内核的内存函数了。下一章会加深你对这种决策的理解。

23122: 最后, `__exit_files`释放`files`本身。

其他`__exit_xxx`函数背后的概念是类似的: 它们减少了任务自有的对于潜在共享信息的引用计数器, 如果这是最后一次引用, 它们要负责执行所有必须的工作将其清除。

7.13.2 wait

和`exec`一样, `wait`是一组函数, 而不是一个函数 (但是和`exec`不同, `wait`家族的函数实际包含一个名为`wait`的函数)。`wait`家族中的其他函数最终都是使用一个系统调用`sys_wait4` (23327行) 实现的, 这个系统调用的名字反映出它实现了`wait`家族中最通用的函数`wait4`。标准C库`libc`的实现必须重新组织对于其他`wait`函数调用的参数, 并调用`sys_wait4` (这还不是问题的全部: 由于历史的原因, 内核到Alpha的移植也会提供`sys_waitpid`。但是即使是`sys_waitpid`也会反过来调用`sys_wait4`)。

除了处理一些其他内容, `sys_wait4`——也只有`sys_wait4`——最终把僵进程送进坟墓。然而从应用程序的观点来看, `wait`和相关函数要检测子孙进程的状态: 检测是否有进程死亡了, 如果有, 到底是哪一个进程, 这个进程是怎样死亡的。

1. `sys_wait4`

23327: 作为相当通用的一个函数, `sys_wait4`有很多参数, 其中一些是可选的。和通常情况一样, `pid`是目标进程的PID; 和你看到的一样, 0和负值是特殊的。如果`stat_addr`非空, 那么它就是所得子孙进程的退出状态应该拷贝到的地址。`options`是一些可能定义`sys_wait4`的操作的标志的集合。如果`ru`非空, 那么它就是所获得的子孙进程资源使用信息所应该拷贝到的地址。

23335: 如果提供了无效选项, `sys_wait4`就返回错误代码。这种决定看起来有点荒唐; 我们可以简单忽略一些无关选项。当然, 这样处理所需要的参数, 如果调用者设置了自己不想设置的位, 那么希望的操作是不要执行——在任何情况下, 这都意味着调用者不能正确理解, 在这种情况下发送一个失败信号量要比简单地忽略调用者的这种困惑要更多。

23342: 循环遍历该进程的直接子进程 (但不包括其孙进程、曾孙进程, 等等)。如同本章中前面说明的一样, 进程的最年轻 (最近创建的) 子孙进程通过`struct task_struct`结构的`p_cptr`成员是可访问的, 这个最年轻进程原来的兄弟进程通过其`p_osptr`成员也是可以访问的; 因此, `sys_wait4`从这个最年轻子孙进程开始遍历其祖先的所有子孙进程, 并逐渐遍历其原来的兄弟进程。

23343: 根据`pid`参数的值筛选出不匹配的PID。注意值为-1的`pid`参数是如何潜在地对进程进行选择的, 正如我们所期望的: `pid`值在23343、23346和23349行中的测试没有成功, 因此它就不会遭到拒绝。这样, 系统需要对每一个子孙进程进行考虑。

23376: 这就是我们现在感兴趣的情况——祖先进程正在等待一个已经结束了的进程。这是最后实际上得到僵进程的地方。它通过更新子孙进程使用的进程的用户时间和系统时间开始 (这通过29772行的`sys_times`系统调用实现), 因为子孙进程不会再参与计算了。

23382: 其他的资源使用信息被收集起来 (如果要求这样处理), 子孙进程的退出状态被

传递到特定的地址中（同样，如果要求这样处理）。

- 23387: 设置retval为当前得到的死亡子孙进程的PID。这就是最后的结果；retval不会再改变了。
- 23388: 如果这个垂死进程的当前祖先进程不是原来的祖先进程，那么进程就会离开进程图表中的当前位置（通过REMOVE_LINKS, 16876行），在其原始祖先的控制下重新安装自己（通过SET_LINKS, 16887行），接着给其祖先进程发送SIGCHLD信号量，这样祖先进程就知道其子孙进程已经退出了。这种通知是通过notify_parent（28548行，在第6章中介绍）传递的。
- 23396: 否则——正常情况——最后可以调用release（22951行）释放所得子孙进程的struct task_struct结构（在看完sys_wait4以后，我们马上就会看release）。
- 23400: 现在已经成功地获取了子孙进程，因此sys_wait4只需要返回成功信息就完美地完成了工作；它跳转到23418行，从这儿返回retval（所获得子孙进程的PID）。
- 23401: 注意特殊的流程控制；default的情况需要继续执行从23342行开始的for循环。因为只有既没有停止运行也不是僵进程的进程才会执行到default的情况，所以这种流程控制是正确的，但是初次阅读时比较容易误解。而且，无论如何这也有些多余；没有它循环也一样能处理。
- 23406: 如果代码能运行到此处，for循环就可以完整地运行下来——正在调用的进程遍历执行其子孙进程没有发现匹配的整个列表——计算的结果是三种状态中的一种。或者由于该任务没有和所提供的pid参数匹配的子孙进程，因而还没有进程退出，或者（是前面情况的一个特例）该任务根本就没有子孙进程。
- 23408: 如果flag不为0，在for循环中就可以执行到23358行，这说明至少有一个进程和所提供的pid参数匹配——它不是僵进程，也没有被终止，因此它就不能被获取。在这种情况下，如果提供了WNOHANG选项——这意味着如果不能获取子孙进程，那么调用者就不会等待——它向前跳转到最后，返回0。
- 23411: 如果有信号量被接收，就退出并返回一个错误。这个信号量不是SIGCHLD——如果它是SIGCHLD，就应该已经发现了死亡的进程，因此就不可能执行到此处。
- 23413: 否则，一切都没有问题；调用者只需要等待一个子孙进程退出。因此，进程的状态被设置为TASK_INTERRUPTIBLE并调用schedule释放CPU给另一个进程使用。正在等待的进程直到再次获得CPU时才会返回，同时要再次检测死亡子孙进程（通过向回跳转到23339行的repeat标号）。回想一下处于TASK_INTERRUPTIBLE状态的进程要等待信号量将其唤醒——在这种情况下，它特别希望SIGCHLD来指明子孙进程已经退出了，但是任何信号量都可以到达。
- 23417: flag是0，因为，或者进程没有子孙进程，或者所提供的pid参数不能和它的任何子孙进程匹配——不管怎样，sys_wait4都给调用者返回一个ECHILD错误。

2. release

- 22951: release的唯一一个参数是指向要释放的struct task_struct结构的指针。
- 22953: 确保该任务没有试图释放自身——这是会在内核中引起逻辑错误的一种无意义的情况。
- 22969: UP代码实际上是通过调用free_uid（23532行）开始的，它用来释放潜在共享的

struct user_struct结构，这个结构除了其他功能以外，还要帮助fork确保不会出现单个用户影响所有进程的情况。

- 22970: 减小系统关于正在运行的任务总数的计数并释放tarray_freelist中的僵死进程的时间片。
- 22974: 僵死进程的PID也会释放，并且使用REMOVE_LINKS (16876行) 解除它同进程表和任务列表的关联。注意，由于内核数据结构在此处正在做出修正，task数组中的进程项并不需要被设置为NULL；把它的空槽增加到自由列表中就足够了。
- 22979: 僵死进程有关次要页表错误、主要页表错误的总数以及向外交换所使用的时间的数量被增加到当前进程对应的“子孙进程计数”中——这是正确的；release只能通过sys_wait4调用，这样只允许进程释放自己的子孙进程。因此，当前进程必须是僵死进程的祖先。
- 22982: 最后，应该回收垂死进程的struct task_struct结构，这可以通过对free_task_struct的调用 (2391行) 来实现。这个函数简单地回收存储在这个结构中的内存页。现在，进程最终功德圆满地寿终正寝了。

第8章 内存

内存是内核所管理的最重要的资源之一。某进程区别于其他进程的一个特征是两个进程存在于逻辑上相互独立的内存空间（与之相反，线程共享内存）。即使进程都是同一程序的实例，比如，两个xterm或两个Emacse，内核都会为每个进程安排内存空间，使得它们看起来像是在系统之上运行的唯一进程。当一个进程不可能偶然或恶意地修改其他进程的执行空间时，系统的安全性和稳定性就会得到增强。

内核也生存在它自己的内存空间之中，即内核空间（kernel space）。与之对应的是用户空间（user space），它是所有非内核任务所处的内存空间的一个通用术语。

8.1 虚拟内存

计算机系统包括不同级别的存储器。图8-1说明了这些存储器中最重要的几项，并且以我自己原有的Linux机器（Linux box）为例标注了一些参数的估计值。当你从左向右观察该图时，会发现存储器容量越来越大而速度却越来越慢。尤其令人注意的是，访问速度跨越了3个数量级（乘数因子为1000），而容量竟跨越了超过8个数量级（乘数因子为312 500 000）（实际上有时速度的差异是可以被掩盖的，不过这些数字足以很好地说明这一部分讨论的目的）。最大的差距体现在最后两个：RAM和磁盘上，它们又分别可被称作主存和辅存。



图8-1 具有不同速度和容量的不同级别的存储器

额外附加的存储器空间总是十分诱人的，即使它们也很慢。如果在RAM被用完时，通过暂时把不用的代码和数据转移到磁盘上以腾出更多空间的方法来使用磁盘代替RAM的话，那将是很好的事情。正如读者可能已经知道的，Linux恰好能够做到这一点，这被称之为虚拟内存。

虚拟内存是一种对RAM和磁盘（或称之为：主存和辅存）进行无缝混合访问的技术。所有这些（虚拟）内存对于应用程序来说就好像它真的存在一样。当然我们知道它并非真的内存，这正是为什么它被称为是“虚拟的”，但是多亏了内核，使得应用程序无法分辨出它们的区别。对于应用程序来说，就好像真的有很大数量的RAM，只不过有时候比较慢而已。

术语“虚拟内存”还有另外一层意思，从严格意义上来讲是与前述的第一种意思没有关系的。这里的虚拟内存指的是对进程驻留地址进行欺骗的方法。每个进程都会有这样一种错觉，认为它的地址是从0开始并由此连续向上发展的。很明显，这一点同时对所有进程都成立是不可能的，但是在生成代码的时候，这个假定却能够带来很大方便，这是由于进程不必知

道它们是否真正从0地址开始驻留，而且它们也不必去关心此事。

这两种意思也不必相关，因为一个操作系统从理论上可以给每个进程分配一个独有的逻辑地址空间，而不用混合使用主存和辅存。然而在所有我已经知道的系统中，对这两种虚拟内存的实现方式要么都采纳，要么都不采纳，这一点可能会在开始时令人感到困惑。

为了避免意义上的分歧，有人倾向于术语“虚拟内存”代表“逻辑地址空间”的意义，同时使用“分页 (paging)”或“交换”表示“磁盘作为内存”使用的含义。尽管这种严格的区分具有充足的理由，但是我更喜欢普通的用法。除非上下文要求，否则我很少花费精力对它们进行区分。

8.1.1 交换和分页

早期的虚拟内存 (VM) 系统仅能够把整个应用程序代码和数据，即完整的进程，从磁盘上移出或移入磁盘。这种技术被称为交换 (swapping)，因为它是把一个进程同另一个进程进行了对调。由于这个原因，磁盘上为VM所保留的区域通常被称为“交换空间”，或简称为“交换区”，尽管如我们所见，现代的系统已不再使用这种最初意义上的交换技术。与此类似，读者通常会见到的术语是“交换设备”和“交换分区”，它是磁盘分区的同义词，但是被专门作为交换空间使用，术语“交换文件”是一个用于交换的、规则的、有固定长度的文件。

交换是很有用的，当然要比根本没有VM好得多，但是它也有一定局限性。首先，交换需要把整个进程同时调入内存，所以当运行一个需要比系统所有RAM还要大的存储空间的进程时，交换便于事无补了，即使磁盘有大量空间可供补充。

其次，交换可能会很低效。交换就必须把整个进程同时调出，这就意味着为了2K的空间不得不把一个8MB的进程整个调出。同理，即使仅仅需要执行被调进的应用程序代码的一小部分，也必须把整个进程同时调进。

分页是把系统的内存划分成很小的块，即页面，每个页面可以独立地从磁盘调入或调出。分页与交换技术相似，但它使用更加细小的粒度 (granularity)。分页比交换有更多的登记 (book-keeping) 开销，这是因为页面数远比进程数要多，然而通过分页可以获得更多的灵活性。而且分页也更快一些，原因之一就是不再需要把整个进程调进调出，而只需要交换必要的页面就足够了。要记住前述的1000倍的速度差异，所以我们应该尽可能避免磁盘的I/O操作。

传统上特定平台上页面的大小是固定的，比如x86平台为4K，这可以简化分页操作。不过，大多数CPU为可变大小的页面提供硬件支持，通常能够达到4M或者更大。可变大小页面可以使分页操作执行得更快和更有效，不过要以复杂性为代价。标准发行的Linux内核不支持可变大小页面，所以我们仍然假定页面大小是4K (已经有支持Cyrix可变大小页面机制的补丁程序，但它们不是本书中官方发行版本的部分。而且据说由此获得的性能增益也并不非常显著)。

因为分页可以完成交换所能完成的所有工作，而且更加有效，所以类似于Linux一样的现代操作系统已不再使用交换，严格地说是只使用分页技术。但是术语“交换”已得到了广泛使用，以至于实际应用中术语“交换”和“分页”已经几乎可以通用；由于内核使用分页技术，所以本书就遵从这种用法。

Linux能够交换到一个专用磁盘分区、一个文件或是分区和文件的不同组合。Linux甚至允许在系统运行时增加和移去交换空间，当你暂时需要额外大量的交换空间，或者假如你发

现需要额外交换空间而又不想重启系统的时候，这就会很有用了。另外，与一些Unix的风格不同，Linux即使没有任何交换空间也能运行得很好。

8.1.2 地址空间

地址空间是一段表示内存位置的地址范围。地址空间有三种：

- 物理地址空间。
- 线性地址空间。
- 逻辑地址空间，也被称为虚拟地址空间。

(需要指出的是，I/O地址能够被看作是第四种地址空间，但是本书中对其不作讨论。)

物理地址是一个系统中可用的真实的硬件地址。假如一个系统有64MB内存，它的合法地址范围是从0到0x4000000 (以十六进制表示)。每个地址都对应于所安装的内存组件 (SIMM) 中的一组晶体管，而且对应于处理器地址总线上的一组特定信号。

分页可以在一个进程的生存期里，把它或它的片段移入或者移出不同的物理内存区域 (或不同物理地址)。这正是进程被分配一个逻辑地址空间的原因之一。就任何特定的进程来说，从0开始扩展到十六进制地址0xc000000共3GB的地址空间是绰绰有余的。即使每个进程有相同的逻辑地址空间，相应进程的物理地址也都是不同的，因此它们不会彼此重叠。

从内核的角度来看，逻辑和物理地址都被划分成页面。因此，就像我们所说的逻辑和物理地址一样，可以称它们为逻辑和物理页面：每个合法的逻辑地址恰好处于一个逻辑页面中，物理地址也是这样的。

与之相反，线性地址通常不被认为是分页的。CPU (实际是下文中的MMU) 会以一种体系结构特有的方式把进程使用的逻辑地址转换成线性地址。在x86平台上，这种转换是简单地把虚拟地址与另一地址，即进程的段基址相加；因为每个任务的基址都被设置为0，所以在这种体系结构中，逻辑地址和线性地址是相同的。得到的线性地址接着被转换成物理地址，并与系统的RAM直接作用。

8.1.3 内存管理单元

在逻辑地址和物理地址之间相互转换的工作是由内核和硬件内存管理单元 (MMU—memory management unit) 共同完成的。MMU是被集成进现代的CPU里的，它们都是同一块CPU芯片内的一个部分，但是把MMU当作一个独立的部分仍然非常有益。内核告诉MMU如何为每个进程把某逻辑页面映射到某特定物理页面，而MMU在进程提出内存请求时完成实际的转换工作。

当地址转换无法完成时，比如，由于给定的逻辑地址不合法或者由于逻辑页面没有对应的物理页面的时候，MMU就给内核发出信号。这种情况称为“页面错误”，本章后面会对此进行详细论述。

MMU也负责增强内存保护，比如当一个应用程序试图在它的内存中对一个已标明是只读的页面进行写操作时，MMU就会通知OS。

MMU的主要好处在于速度。为了获得同样的效果，缺少MMU时OS将不得不使用软件为每个进程的每一次内存引用进行校验，这种校验同时包括数据和指令在内，而这可能还包括要为进程创建其生存所需的虚拟机 (Java所进行的一些工作与此类似)。这样做的结果将使系

统慢得令人无法忍受。但是，一个以这种内存访问合法性方式集成在计算机硬件里的MMU却根本不会使系统变慢。在MMU建立起一个进程以后，内核就只是偶尔参与工作，例如在发生页面错误时，而这与全部内存引用数量相比是非常少的。

除此而外，MMU还可以协助保护内存自身。没有MMU，内核可能不能防止一个进程非法侵入它自己的内存空间或者是其他进程的内存空间。但是如何避免内核也会作同样的操作呢？在Intel的80486或更新的芯片上（不是80386），MMU的内存保护特性也适用于内核进程。

8.1.4 页目录和页表

在x86体系结构上，把线性地址（或者逻辑地址——记住在Linux上，这二者具有相同的值）解析到物理地址分为两个步骤，整个过程如图8-2所示。提供给进程的线性地址被分为三个部分：一个页目录索引，一个页表索引和一个偏移量。“页目录”（page directory）是一个指向页表的指针数组，“页表”（page table）是一个指向页面的指针数组，因此地址解析就是一个跟踪指针链的过程。一个页目录使你能够确定一个页表，继而得到一个页面，然后页面中的“偏移量”（offset）能够指出该页面里的一个地址。

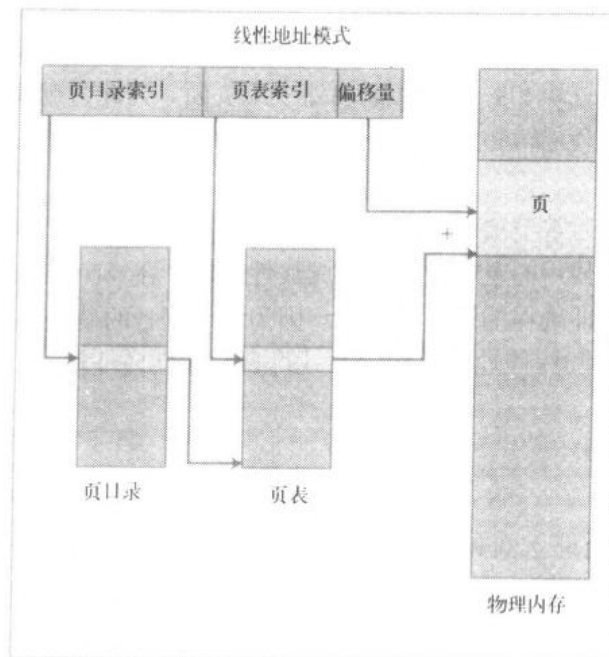


图8-2 x86结构上的分页

为了进行更详细、更准确的描述：给定页目录索引中的页目录项保存着贮存在物理内存上的一个页表地址；给定页表索引中的页表项保存着物理内存上相应物理页面的基地址；然后线性地址的偏移量加到这个物理地址上形成最终物理页面内的目的地址。

其他CPU使用三级转换方法，如图8-3所示。这在64位体系中尤其有用，以Alpha为例，其更大的64位的地址空间意味着类似于x86体系的地址转换将要求大量的页目录、大量页表、大量偏移量或三者兼有。对于这种情况，Alpha的设计者们向线性地址模式中引入了另一层次，即Linux所称的“页面中间目录”（page middle directory），它位于页目录和页表之间。

这个方案与以前实际是一样的，只不过多增加了一级。这种三级转换方法同样具有页目

录，页目录的每一项包含一个页面中间目录的入口地址，页面中间目录的每一项包含一个页表的入口地址，而页表也同以前一样每一项包含物理内存中一个页面的地址，这个地址再加上偏移量就得到了最终的地址。

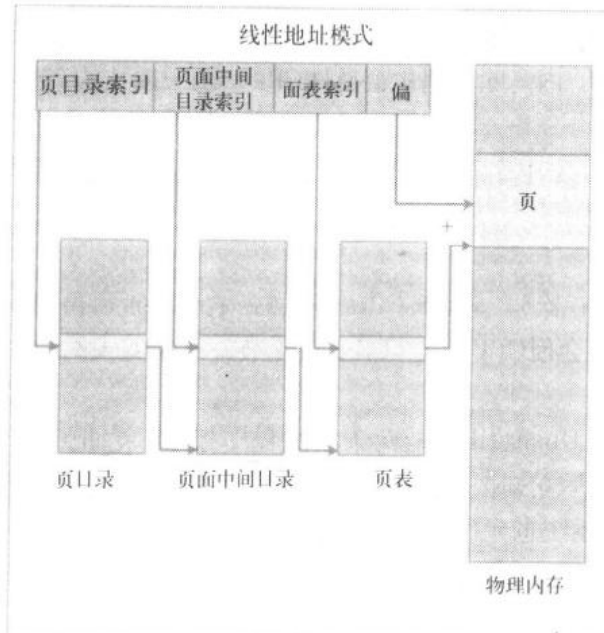


图8-3 一般情况下内核的分页

通过进一步观察可知，使情况更为复杂的是，三部分地址模式与两级地址转换是相关联，而四部分地址模式则与三级地址转换相关联，这是由于我们通常所说的“级（或层次levels）”不包括索引到页目录的第一步（我想是因为这一步没有进行转换的缘故）。

令人奇怪的是内核开发者们决定只用其中一种模式来处理问题。绝大部分的内核代码对MMU一视同仁，就如同MMU都使用三级转换方法（也就是四部分地址模式）一样。在x86平台上，通过将页面中间目录定义为1，页面相关的宏可以把三级分解过程完美地转换到二级分解过程上去。这些宏认为页面中间目录和页目录是几乎可以进行相互替换的等价品，以至于内核的主要代码可以认为其地址就是由四个部分组成的。

在x86系统中，32位地址中10位是页目录索引，接下来10位是页表索引，剩下的12位用作偏移量，这就构成了4K大小的页面（ 2^{12} 等于4 096个偏移量）。

用于创建和操作每一级项的函数和宏定义在头文件include/asm-i386/page.h（10786行）和include/asm-i386/pgtable.h（10876行）之中。在读者浏览这些函数和宏的时候，记住PGD通常代表“页目录项”（不只是“页目录”），PMD通常代表“页面中间目录项”（不只是“页面中间目录”），同样PTE也通常代表“页表项”。而且，正如上面解释中限定词“通常”所暗示的那样，例外是存在的，例如下文将要提到的pte_alloc就分配页表而不是（如你所可能会认为的）页表项。非常遗憾的是，由于篇幅所限，我们不能对全部例程进行讨论，我们将在后面对其中的一部分进行讨论。

页表项不仅记录了一个页面的基地址，而且记录了它的保护信息（protections），也就是一组指定该页为可读、可写、和/或可执行的标志（这容易让人联想到文件的保护位）。随着我们对页面保护信息的进一步剖析，读者会看到页表项所包括的其他页面特有的标志。

8.1.5 转换后备缓存

如果简单地执行从线性地址到物理地址的转换过程，在跟踪指针链时将会需要几个内存引用。RAM虽然不像磁盘那么慢，但是仍然比CPU要慢得多，这样就容易形成性能的瓶颈。为了减少这种开销，最近被执行过的地址转换结果将被存储在MMU的转换后备缓存（TLB, translation lookaside buffer）内。除了偶尔会通知CPU，由于内核的某操作致使TLB无效之外，Linux不用明确管理TLB。

在作用于TLB的函数和宏中，我们只研究一下__flush_tlb，在x86平台上，它是其他大部分函数和宏的基础。

`__flush_tlb`

10917: CR3（控制寄存器3）是x86 CPU寄存器，它保存页目录的基地址。往这个寄存器送入一个值将会使CPU认为TLB变成无效，甚至写入与CR3已有值相同的值也是这样。

因此，__flush_tlb仅是两条汇编程序指令：它把CR3的值保存在临时变量tmpreg里，然后立刻把tmpreg的值拷贝回CR3中，整个过程就这么简单！

注意x86系统也允许使某一个单独的TLB项无效，而并不一定非要使全部项，这种方法使用invlpg指令——参见10926行它的使用信息。

8.1.6 段

由于段不是在所有CPU中均可用，所以Linux内核中与体系结构无关的部分不能对段进行辨识。在不同的CPU体系中，段的（如果段在体系中是可用的）处理方式大相径庭，这一点是非常重要的。因此，我们在这个问题上不会花费太多时间，不过x86系统上内核使用段的方式还是值得讨论一下的。

段可以被简单地看作是定义内存区域的另一种机制，有些类似于页。这两种机制可以重叠：地址总是在页面之内，也可能处于段内。与页不同，段的大小可以变化，甚至在其生存期里能够增长和收缩。与页相同的是，段可以被保护，而且其保护可由硬件实施；当x86的段保护和同一地址的页保护发生冲突时，段保护优先。

x86系统使用一些寄存器和全局描述符表（GDT）与局部描述符表（LDT）来对段进行跟踪。描述符是段的描述信息，它是用来保存段的大小和基址以及段的保护信息的8字节的对象。GDT在系统中只有一个，而Linux可以为每个任务建立一个LDT。

接下来我们将简单解释内核是如何使用这些表来建立段的。内核本身拥有分离的代码和数据段，它们被记述在GDT的第2和第3行项里。每个任务也有分离的代码和数据段。当前任务的代码段和数据段不仅在它自己的LDT的第0和第1行项被说明，而且还被记述在GDT的第4和第5行项里。

在GDT里，每个任务占两行项，一个用来定位它的LDT，一个用来定位它的TSS（前面章节曾简要提及的任务状态段）。因为x86 CPU限制GDT的大小为8 192个项，而且Linux为每个任务占用两行GDT项，因此，显而易见的是我们不能拥有超过4096个任务，这也正是在第7章里提到的限制。事实上，任务的最大数目要稍小一点儿，不过仍有4090个，这是由于GDT的前12行项被保留用于其他目的。

富有经验的x86程序员可能已经注意到Linux所使用的x86分段机制是采用最低限度方式

的；段的主要使用目的仅是为了避免用户代码出现在内核段中。Linux更倾向于分页机制。从大的方面来看，对于处理器来说，分页或多或少都是相同的，或者说总的事实就是这样，因此内核越是以分页方式工作，它的可移植性就越好。

最后要提及的是，如果读者对于x86的分段机制很感兴趣的话，不妨阅读一下“Intel体系结构下的软件开发手册第3卷（Intel Architecture Software Developer's Manual Volume 3）”，该书可以从Intel站点上免费得到（developer.intel.com/design/pentiumii/manuals/243192.htm）。

8.2 进程的内存组织

有三个重要的数据结构用于表示进程的内存使用：`struct vm_area_struct`（15113行）、`struct vm_operations_struct`（15171行）和`struct mm_struct`（16270行）。我们随后将对这三个数据结构进行逐一介绍。

8.2.1 `struct vm_area_struct`

内核使用一个或更多的`struct vm_area_struct`来跟踪进程使用的内存区域，该结构体通常缩写为VMA。每个VMA代表进程地址空间的一块单独连续的区间。对于一个给定的进程，两个VMA决不会重叠，一个地址最多被一个VMA所覆盖；进程从未访问过的一个地址将不会在任何一个VMA中。

两个VMA之间的区别有两个特征：

- 两个VMA可以不连续：换句话说，一个VMA的末尾不一定是另一个的开头。
- 两个VMA的保护模式可以不同：例如，一个是可写的，而另一个可能是不可写的。即使两个这样的VMA是连在一起的，它们也必须被分开管理，因为其保护信息不同。

应注意的一个重点是，一个地址可以被一个VMA所覆盖，即使内核并没有分配一个页面来存储这个地址。VMA的一个主要应用就是在页面错误时决定如何作出反应。我们可以将VMA看作是一个进程所占用的内存空间及这些空间的保护模式的总体视图。内核能够反复重新计算从页表而来的VMA中的大部分信息，不过那样速度会相当慢。

进程的所有VMA是以一个排序的双向链表方式存储的，并且它们使用自己的指针来管理该列表。当一个进程有多于`AVL_MIN_MAP_COUNT`数目（在16286行定义为32）的VMA时，内核也会创建一个AVL树来存储它们，此时仍然是使用VMA自己的指针对该树进行管理。AVL树是一个平衡二叉树结构，因此这种方法在VMA数量巨大时查找效率十分高。不过，即使在AVL树被创建后，线性列表也会被保留以便内核即使不使用递归也能轻松的遍历一个进程的所有VMA。

`struct vm_area_struct`的两个最重要的元素是它的`vm_start`和`vm_end`成员（分别在15115行和15116行），它们定义了VMA所覆盖的起止范围，其中`vm_start`是VMA中的最小地址，而`vm_end`是比VMA最大地址大一位的地址。在本章后面我们会反复提及这些成员。

注意，`vm_start`和`vm_end`的类型是`unsigned long`，而不是读者可能会认为的`void*`。由于这个原因，内核在所有表示地址的地方都使用`unsigned long`类型，而不用`void*`类型。采用这种方法的部分原因是可以避免因内核对诸如比特一级的地址进行计算操作时引起的编译警告，还可能避免由于它们而偶然引起的间接错误。在引用内核空间的一个数据结构的地址时，内核代码使用指针变量；在对用户空间的地址进行操作时，内核却频繁地使用`unsigned long`——

实际上，几乎只有本章中所涉及的代码才是这样。

这样就给用来编译内核的编译器提出了要求。使用unsigned long作为地址类型就意味着编译器必须使unsigned long的类型长度和void*的一样，尽管实践中对这一点的要求不是十分严格。对于x86寄存器上的gcc来说，两种类型很自然的都是32位长。在64位指针长度的体系中，比如Alpha，gcc的unsigned long类型长度通常也是64位。尽管如此，在将来的体系结构上，gcc的一个端口可能提供与void*不同的unsigned long类型长度，这是需要内核的移植版本开发人员注意的一点。

还要说明的是，除了gcc之外你不需要对编译器的性能有太多担心，因为其他大部分与gcc相关的特性都已经包括在代码之中了。假如读者试图用某个另外的编译器来编译内核的话，我想有关unsigned long和void*长度的错误将会占编译错误列表的绝大多数。

8.2.2 struct vm_operations_struct

一个VMA可能代表一段平常的内存区间，就像是malloc函数所返回的那样。但是它也可以是对应于一个文件、共享内存、交换设备或是其他特别的对象而建立的一块内存区域；这种对应关系是由本章后面将要涉及的称为mmap的系统调用所确定的。

我们不想讨论太多关于VMA可以被映射的每一种对象的专门知识，这会使对内核代码的剖析变得凌乱不堪，因为那样就不得不反复决定是否要关闭一个文件、分离共享内存等等令人非常头疼的事情。与此不同，对象类型struct vm_operations_struct抽象了各种可能提供给被映射对象的操作，比如打开、关闭之类。一个struct vm_operations_struct结构体就是一组函数指针，它们之中可能会是NULL用来表示一个操作对某个被映射对象是不可用的。举例来说，在一个共享内存没有被映射的情况下，把该共享内存对象的页面与磁盘进行同步是没有意义的，表示共享内存操作的struct vm_operations_struct里的sync成员就是NULL。

总之，一旦VMA映射为一个对象，那么它的vm_ops成员就会是一个非空的指针，指向一个表示被映射对象所提供操作的struct vm_operations_struct结构体。对于VMA可以映射的每一种对象类型，都有一个该VMA可能会在某处指向的静态static struct vm_operations_struct结构体。参见第21809行的例子。

8.2.3 struct mm_struct

一个进程所保留的所有VMA都是由struct mm_struct结构体来管理的。指向这种结构类型的指针在struct task_struct中，确切地说，它就是后者的mm成员。这个成员被前一章中所讨论的goodness（26388行）应用，来判断是否两个任务是在同一个线程组中。两个具有相同mm成员（正如我们所见到的）的任务管理同一块全局内存区域，这也是线程的一个特点。

struct mm_struct结构体的mmap成员（16271行）就是前述的VMA的链接列表，而它的mmap_avl成员，如果非空就是VMA的AVL树。读者可以浏览struct mm_struct的定义，会发现它还包括相当多的其他成员，它们中的几个会在本章中涉及到。

8.2.4 VMA的操作

本小节介绍后面要用到的find_vma函数，并捎带简介它的同类函数find_vma_prev。这将阐明VMA处理操作的一些方面，也为读者将要接触的代码做准备。

1. find_vma

33460: 简单说来, find_vma函数的工作就是找到包含某特定地址的第一个VMA。更准确地说, 它的工作是找到其vm_end比给定地址大的第一个VMA, 这个地址可能会在该VMA之外, 因为它可以比VMA的vm_start要小。这个函数返回指向VMA的指针, 如果没有满足要求的VMA就返回NULL。

33468: 首先, 通过使用mmap_cache的mm成员, 满足进程最近一次请求的同一VMA会被检查, 而mmap_cache正是为此目的而设。我没有亲自测试过, 不过这个函数的文档中说高速缓存的命中率可以达到35%, 考虑到高速缓存只由一个VMA组成, 那么这个数字就相当好了。当然, 著名的、被称之为“引用的局部性(locality of reference)”的特性一直在其中提供了很大帮助, 这也是软件访问数据(和指令)时的一条原则, 即访问最近使用过的数据(和指令)。由于VMA包含一块连续的地址区间, 引用的局部性就使得所需的地址都在同一个VMA中变为可能, 而这样的VMA就会满足前面的要求。

当修改VMA列表时的其他几个地方, 这个高速缓存的值被设为空, 表明对VMA列表所做的修改可能会使它失效。至少在一个这种情况中, 第33953行, 使该高速缓存为空不总是必要的; 这段代码如果能够再聪明一些的话, 就可能从本质上改善高速缓存的命中率。

33471: 高速缓存没有命中。假如没有AVL树, find_vma只是搜索列表上的所有VMA, 然后返回第一个符合条件的VMA。回想一下VMA的列表是保持顺序的, 所以符合条件的VMA也就是所有符合条件的VMA中地址最小的一个。假如搜索到列表的末尾都没有一个匹配, vma就被置为NULL, 并被返回。

33476: 若有大量VMA, 沿树遍历就比沿链表遍历要快; 由于AVL树是平衡的, 这就是一种对数时间操作, 而不是线性时间操作。

树的迭代遍历并不是十分罕见的现象, 不过一些特征也并不非常明显。首先注意第33484行的赋值; 这个操作一直跟踪当前找到的最好节点, 当不能找到更好的时, 它就会被返回。接着的下一行中的if语句是一个最优测试, 检测addr是否处于VMA中(我们已知的一点是addr小于VMA的vm_end)。因为VMA绝对不会彼此覆盖, 没有其他VMA将是一个较贴近的匹配结果, 所以树的遍历可以早些结束。

33492: 如果在树的遍历或列表搜索过程中找到一个VMA, 找到的值就被保存在高速缓存里以便下一次查找。

33496: 在任何情况下, vma都被返回; 它的值或者是NULL, 或者是满足查找条件的第一个VMA。

2. find_vma_prev

如前所述, 这个函数(从第33501行开始)和find_vma函数是一样的, 不过它还会额外地返回一个指向前一个限定的addr(如果有)的VMA的指针。这个函数不仅是因为它本身的缘故而令人感兴趣, 更主要是由于它的出现会告诉我们一些关于内核程序设计, 特别是关于Linux内核程序设计的信息。

应用程序员很可能已经在更加通用的find_vma_prev函数之上写出了find_vma函数, 这只

需简单地把指向VMA的指针去掉即可，代码如下：

```
struct vm_area_struct *
find_vma(struct mm_struct * mm,
         unsigned long addr)
{
    struct vm_area_struct * notused;
    return find_vma_prev(mm, addr, &notused);
}
```

应用程序员这样做的原因是具有代表性的应用程序并不太拘于速度因素。这并非纯粹是在为铺张浪费找借口，而是由于CPU速度的不断增加使得应用能够更关注于其他方面，我们现在可以出于可维护性的充分理由而提供一个可以到处使用的额外函数调用。

与之相反，一个内核程序员可能不会随便增加多余的函数调用；试着减少几个CPU时钟周期会被认为比负责维护某个近乎是副本的函数要更胜一筹。即使没有其他原因，我们也可以说内核开发者所持有的这种态度就是为了让应用程序员能相对自由一些。

相对于源代码相对封闭的操作系统而言，为什么这种重复在Linux内核中不那么重要，这里是否有更深层次原因呢？尽管Linux内核必须限制它占用的CPU时间，Linux内核的开发工作却不受程序员时间的限制（明确地说，我必须指出Linux的开发者不必把他们的时间浪费在会议上的，他们也不必被人制订的时间表所拘束）。正是由于这众多的队伍，众人的智慧，才改变了软件开发的规则。

Linux内核的源代码对任何人都是公开的，Linus本人曾说过的一句名言是“...只要眼睛够多，所有的臭虫（程序错误）都是浅薄的（...give enough eyeballs, all bugs are shallow）”。就算函数find_vma和find_vma_prev的执行会产生重大差异，在你能想到“重编译”之前，不知什么地方的某个Linux内核开发者就已经迅速发现并修复了这个问题。实际上，Linux内核开发者比它的商业对手动作快得多，所得到的代码运行更快，而且错误更少，尽管有时偶然出现的结构会被认为在任何其他环境中都不可维护。

当然，如果没有人负责对这些函数的改进进行维护的话，我认为这也是非常愚蠢的。内核的下一个发布版本就把它合并了。但是我仍然对此持怀疑态度，而且即使我在这个具体问题上的态度并不正确，我依然在总体上保持原有态度。不同的事还会继续不同，而不同正是Linux之所以为Linux的一方面。

8.3 分页

本章前面对分页已做了概要描述，现在我们进一步来研究Linux是如何处理分页的。

8.3.1 页面保护详述

正如早先提及的，页表项不仅保存了一个页面的基地址，还有其他一些标志信息，这些标志指出了该页面上所能进行的操作。

如果页表项只保存一个页面的基地址，并且页面是页对准的（page-aligned），这个地址的低12位（x86系统），即偏移量部分通常总是为0。取代这些位置0的作法是把它们用于编码与页面有关的标志，在获取页地址时只需简单地把它们屏蔽掉就行了。以下就是这12位编码中的标志位：

- `_PAGE_PRESENT`位（11092行），若置位，当前页面物理存在于RAM中。

- `_PAGE_RW`位 (11093行), 置为0表示该页面是只读的, 置为1表示可读可写, 没有只写的页面。
- `_PAGE_USER`位 (11094行), 置位表示某页面是用户空间页面, 清空表示为内核空间页面。
- `_PAGE_WT`位 (11095行), 置为1表示页面高速缓存管理策略是透写 (writethrough), 置为0表示管理策略是回写 (缺省)。透写会立刻把写入高速缓存的数据拷贝到主存储器内, 即使保存在高速缓存的数据仍是读访问。与之相反, 回写 (writeback) 具有更高的效率, 写入高速缓存的数据仅当其必须为其他数据腾出空间, 而必须移出时才被拷贝到主存储器内 (这是由硬件, 而不是Linux完成的)。尽管直到本书写作之时, 这个标志位在内核中的使用还并不非常普遍, 不过这种情形有望很快改变。有时候, 在Intel公司的处理器资料中把WT位更多的称为PWT。
- `_PAGE_PCD`位 (11096行), 关闭页面高速缓存; 本书中的代码不总是使用这个标志位 (缩写“CD”表示“caching disabled”)。如果我们恰好知道一个不经常使用的页面, 那么就不必为它设置高速缓存, 这可能会更有效率。这个标志位好像对于映射内存的I/O设备来说更有用处, 尽管我们想确保对表示设备的内存进行的写操作不被高速缓存缓冲, 但是取而代之的作法是立刻把数据直接拷贝到设备之中。
- `_PAGE_ACCESSED`位 (11097行), 若置位表示该页面最近曾被访问过。Linux可以设置或清除这个标志, 不过通常这是由硬件完成的。因为清除了该标志的页面已很久未被使用过, 所以它们会在交换时被优先调出主存。
- `_PAGE_DIRTY`位 (11098行), 若置位, 表明该页面的内容自从上次该位被清除后已发生改变。这就意味着它是一个内容没有保存的页面, 就不能简单地交换而被删除。当一个页面第一次写入内存时, 该标志位由MMU或Linux设置; 当这个页面调出内存时, Linux要读取它的值。
- `_PAGE_PROTNONE`位 (11103行), 是一个以前的页表项没有使用过的标志位, 用来跟踪当前页面。

`_PAGE_4M`位和`_PAGE_GLOBAL`位在同一个`#define`定义块中出现, 但是由于它们不像其他标志位那样用于页面级的保护, 所以我们在此不予讨论。

随后的文件中, 上述这些标志位被组合在一个高级宏内。

8.3.2 写拷贝

提高效率的一条路就是偷懒——只做最少量的必要工作, 而且只在不得不做的时候才做。现实生活中这可能是个坏习惯, 至少它会导致拖拖拉拉。而在计算机的世界里, 它可能更是一种优点。写拷贝 (copy-on-write) 就是Linux内核一种通过懒惰来获得效率的方法。其基本思想是把一个页面标记为只读, 却把它所含的VMA标识为可写。任何对页面的写操作都会与页级保护相冲突, 然后触发一个页面错误。页面错误处理程序会注意到这是由页级保护和VMA的保护不一致而导致的, 然后它就会创建一个该页的可写拷贝作为代替。

写拷贝十分有用。进程经常fork并立刻exec, 这样为fork而拷贝进程页面会造成浪费, 因为exec之后它们会不得被抛弃。正如读者所见, 进程分配大量内存时也使用同样的机制。所有被分配的页面都与一个单独的空白页面相映射, 这就是写拷贝的原意。向某页面的第一

次写操作会触发页面错误，然后空白页面执行拷贝。用这种办法，只有页面分配不能再延期时，它才会被分配。

8.3.3 页面错误

到现在为止，本章已几次提到一个页面可以不在RAM里的可能性——毕竟，如果页面总是在内存里，虚拟内存就没什么必要了。但是我们还没有详细介绍过当某页面不在RAM中会怎样。当处理器试图访问一个当前不在RAM中的页面时，MMU就会产生一次页面错误，而内核会尽力解决它。在进程违反页级保护时，页面错误也会产生，例如进程试图向只读内存区域写入。

因为任何无效内存访问都会导致页面错误，同样的机制支持请求分页。请求分页 (demand paging) 的意思是只有在页面被引用的时候才从磁盘上读取它们——即按需分配——这是另一种通过懒惰来获得效率的方法。

特别是请求分页用于实现被请求页面的可执行化。为了达到这个目的，应用程序第一次被装载时，只有一小部分可执行映象 (image) 被读入物理内存，然后内核就依靠页面错误来调入需要的 (比如，进程首次跳转到一个子例程时) 可执行页面。除了一些意外的情况，这样做总是要比把所有部分一次读入要快，这是因为磁盘较慢，而且并不是所有的程序都会用到的。事实上，因为一个大程序运行一次时，大部分功能特性都不会再用到，所以通常根本不需要全部都读入 (这一点大多数中小规模的程序也是成立的)。这对于按需分页 (demand-paged) 的可执行程序稍有不同——如果你对这种情况进行考虑的话，你就可以知道按需分页还需要二进制处理程序的支持，而且它是一个具有决定意义的部分。

1. do_page_fault

6980: do_page_fault是内核函数，产生页面错误时 (在363行设置) 被调用。当页面错误产生时，CPU调整进程的寄存器，当解决页面错误时，进程再从引起错误的指令处开始执行。通过这种方法，在内核使冲突地内存访问操作完成后，会自动重试该操作。相反，如果页面错误仍然无法解决，内核就通知引起冲突的进程。当页面错误是由内核本身导致的时候，所采用的措施是近似的，但并不完全相同。

6992: 控制寄存器2 (CR2) 是Intel CPU的寄存器，保存引起页面错误的线性地址。该寄存器内的地址会被直接读入局部变量address。

7004: 函数find_vma (33460行) 返回地址范围末尾在address之后的第一个VMA。大家知道，这并不能够保证该地址位于VMA的范围内，而仅保证该地址比VMA的结束地址小，这样它就可能比VMA的初始地址还要小。因此这一点要被检查。假如通过判断，则address在VMA之内，控制就会向前跳转到标号good_area处 (7023行)；我们随后会对这一点进行讨论。

7005: 如果find_vma返回空值NULL，那么address就位于进程的所有VMA之后——换句话说就是超出了由进程引用的所有内存。

7009: vma的开头和结尾都确实超过了address；因此，address在VMA低端地址以下。但是这并不会失去什么。如果VMA是向下扩展的类型——也可以说它是堆栈——这个堆栈可以简单地向下扩展来适应那个地址。

7011: CPU提供的error_code的测试位2。与监控 (内核) 模式相比，更多是在用户模式

发生页面错误时设置此位。如果是在用户模式下，`do_page_fault`函数会保证给定的地址在为进程建立的堆栈区域内，正如ESP寄存器所定义的那样（例如，在代码溢出了被分配的堆栈矩阵时，就会产生这种情况）。如果是在监控（内核）模式下，就会跳过后一种判断，而简单地假定内核运行正常。

7019: 如果可能，使用`expand_stack`（15480行）将VMA扩展到包含新的地址。如果成功，VMA的`vm_start`成员将调整到包含`address`。

7023: 到达`good_area`标号时，就意味着VMA包含`address`，或者说要么它已经包括了`address`，要么就是堆栈扩展后包括了该地址。

不管哪一种方法，包括错误产生原因信息的`error_code`最低两位现在都可以被测试了。第0位是“当前/保护”位：0表示该页不是当前；1表示该页是当前，但试图的访问操作与页级保护位冲突。第1位是“读/写”位：0表示读，1表示写。

7025: `switch`条件判断语句对于上述两个测试位所组合出的四种可能情况作出相应处理：

- 情况 2或3——检查包括的VMA是否可写。若可写，就是向一个写拷贝页面执行一次写操作；变量`write`被增加（设置到1）以便接下来对`handle_mm_fault`的调用能够完成写拷贝过程。
- 情况 1——这意味着页面错误是由试图从一个存在但不可读的页面中读数据而导致的；这个尝试会被拒绝。
- 情况 0——表示页面错误是由试图从一个不存在的页面中读数据而导致的。如果涉及的VMA保护指出该区间既不可读也不可写，读页面只不过是浪费时间——如果再次尝试，将引起另一个页面错误，这样`do_page_fault`函数会以情况1的结果告终，即拒绝尝试。否则`do_page_fault`函数继续执行，并从磁盘上读入页面。

7047: 请求`handle_mm_fault`函数（下面讨论），使该页面变为当前页面。如果失败，则发出一个SIGBUS错误。

7062: 大多数内核函数的清除代码都不太显著。`do_page_fault`函数是一个例外；我们会比较详尽地研究它的清除代码。下列任何情况发生都会跳到`bad_area`标号处：

- 被引用的地址超过了为进程分配的（或保留的）所有内存。
- 被引用的地址位于所有VMA之外，而且可能由于比该地址小的VMA不是堆栈而无法扩展到该地址。
- 违反了页面的读/写保护。

7066: 如果用户代码引起以上任何错误，那将发送致命的SIGSEGV信号——一个分段违例（注意术语“分段”在这里是历史上的说法，而不是字面所表达的意思——对CPU来说，从技术角度看它是分页违例，不一定是分段违例）。这个信号通常会像第6章中讨论的那样杀死一个进程。

7075: Intel Pentium CPU（以及它的一些兼容产品）具有一个所谓的f00f缺陷，它允许任何进程用非法的`0xf00fc7c8`指令来冻结CPU。Intel所提议的弥补工作就是在这里实现的：中断描述符表（见第6章）的一部分以前是被标记为只读的，因为这样会使非法指令执行时产生页面错误代替冻结CPU。在这里，`do_page_fault`函数检查导致页面错误的地址是否位于IDT中由非法指令执行而产生的位置上。如果是这样

的，处理器会试着执行“Invalid Opcode”服务中断——CPU的缺陷会使得正确完成这一步失败，而代码却会通过直接调用do_invalid_op函数而产生正确的结果。否则，CPU决不会对IDT进行写操作（即使没有标记为只读时也是如此），所以即使第7080行的检测失败，非法指令也是根本不会被执行的。

7086: 下列情况发生时，标号no_context会被执行：

- 在内核（不是用户）模式里到达bad_area，而且CPU不执行触发f00f缺陷的非法指令。
- 在一个中断中或没有用户环境（用户任务没有处于正在执行状态）时发生的页面错误。
- handle_mm_fault函数错误，并且系统处于内核模式中（我还从未遇到过这种情况）。

这里的任何一种情形都是内核错误（经常由驱动程序所导致），它不是因为任何用户代码而造成的页面错误。如果内核（或驱动程序）事先为这种可能准备了错误处理代码，那么这些错误处理代码一定能在错误发生时可以通过某种特殊技术跳转过去，这不在本书讨论范围之内。

7097: 否则，内核试图访问一个坏页面，do_page_fault函数将不知如何处理它。这也可能被考虑到。内核启动代码检查是否MMU写保护工作正常；如果正常，那就不是一个真正的错误，do_page_fault函数就可以简单地返回了。

7109: 内核访问了一个坏页面，并且do_page_fault函数无法修复这个错误。do_page_fault函数会在第7129行显示出一些描述错误的信息，然后中止内核本身。这样整个系统就会被停止，很明显没有任何操作会被执行。不过，如果系统运行到了这一步，内核也别无选择了。

7134: 最后一个标号是do_sigbus，只有当handle_mm_fault函数无法处理错误时才会执行到这里。这种情况相对简单；大体上是给违例的进程发送一个SIGBUS错误信号，如果这是在内核模式里发生的就再跳回到no_context标号处。

2. handle_mm_fault

32725: 调用者已经检测到了需要一个可用的页面。该页面正是包含address的页面，这个地址应归入vma中。handle_mm_fault函数本身相当简洁，但是它建立在其他几个处理冗长细节问题的函数和宏之上。我们介绍完此函数后将逐一研究那些底层函数。

32732: 查找关联的页目录和页面中间目录项（如前所述，这两者在x86平台上实际是一样的）。

32735: 从页面中间目录项得到或定位（如果可能的话）页表。

32737: 调用handle_pte_fault函数把页面读入页表项；如果成功，就调用update_mmu_cache函数更新MMU的高速缓存。控制流程到此为止，一切顺利，handle_mm_fault函数就可以返回一个非零值（1）表示成功了。如果此过程任何一步出错，控制就转向第32744行，函数返回0值表示失败。

3. pgd_offset

11284: 这个宏将address除以 $2^{\text{PGDIR_SHIFT}}$ （第11052行定义为22），并对结果向下舍入，然后

把最终结果（移位之前的高端10位）作为提供的struct mm_struct的pgd数组的一个索引。因此，它的值就是页目录项，相应的页表address地址就位于该项中。

这等价于代码

```
&((mm)->pgd[(address) >> PGDIR_SHIFT]);
```

而且可能会更高效。

4. pmd_alloc

11454: 因为x86平台上没有定义页面中间目录，这样就极其简单：它只需返回给定的pgd指针，并映射为一种不同类型。在其他平台上，该函数与pte_alloc类似，还要实现更多的工作。

5. pte_alloc

11422: pte_alloc函数有两个参数：一个是指针，指向目标地址所位于的页面中间目录项，另一个是地址本身。如果我们暂时跳过一部分内容，那么对该函数经过变形的逻辑的理解就会更容易，所以让我们看一下随后的若干行代码。

11425: 用一种几乎无法理解的方式把address转换成PMD内的一个偏移量。

这一行需要详细进行解释。首先，回忆一下PMD中的每项都是一个指针，在x86平台上它的长度是4个字节（这里的代码是特定体系结构的一部分，所以我们可以做出这样的假定）。用C语言来定义就是：

```
&pmd[middle_10_bits(address)]
```

（为清晰起见，我在这里引入了假定的pmd数组和middle_10_bits函数）该代码等价于：

```
pmd + middle_10_bits(address)
```

这又与如下代码指向的地址相同：

```
((char *) pmd)
+ middle_10_bits(address) * sizeof(pte_t*)
```

其技巧是在最后的公式中——或者更准确地说是+号后边的部分——最接近于第11425行所要计算的 actual 值。

为了使这一点更为明确，首先可知：

```
4 * (PTRS_PER_PTE - 1)
```

就是4092（第11059行PTRS_PER_PTE被定义为1024）。用二进制表示，4092只用占最低12位，甚至最后2位也用不上。它和只占最低10位的1023左移2位后的值相同。这样就有

```
(address >> (PAGE_SHIFT-2))
```

把address右移10位（第10790行PAGE_SHIFT被定义为12）。这两个表达式结果再逐位进行“与”（AND）操作。最终的结果类似于：

```
((address >> PAGE_SHIFT) & (PTRS_PER_PTE-1))
<< 2
```

尽管这仍很复杂，不过它更简单明了：它把address右移12位（为了去掉页面偏移量部分），屏蔽掉除最低10位的其他位（去掉页目录索引部分，只保留最低10位的页面中间目录索引），接着把结果左移2位（相当于乘以4，即指针长度的字节数sizeof(pte_t*)）。更直接的方法可能会稍慢一些，但在内核里，我们终究是要尽力节省时间的（虽然更直接的方法看来并非明显偏慢：同样版本内核进行两次移位、两次减法及按位“与”的操作，和进行两次移位、两次

按位“与”的操作，就我的测试看来实际上是一样快)。

不管采用哪一种方法，经过计算之后，把address和无符号长整型的PMD的基地址相加（在第11432行和别的地方执行），就得到了指向与address初值关联的PTE的项指针。

11428: 如果PMD项不指向任一个页表，函数向前跳到getnew处分配一个页表。

11435: 通过调用get_pte_fast (11357行) 尝试从pte_quicklist中申请一个页表。这个页表是页表的一个高速缓存，其思想是分配页表（它们本身就是独立的页面）慢，而从一系列近期释放的页表中指定一个却会稍快一些。所以，代码经常用free_pte_fast (11369行) 来释放页表，这会把它们放在pte_quicklist里而不是确实把它们消除掉。

11439: pte_quicklist能够提供一个页表页面。页表可以被送入页面中间目录，并且函数返回页表中这个页面的偏移值。

11438: pte_quicklist缓存里没有剩下页面，因此pte_alloc需要调用get_pte_slow函数 (7216行) 来缓慢分配一个页面。该函数用__get_free_page来分配页面，执行过程和一个页面被找到时相似。

11430: 如果PMD项不是0，但是无效的，pte_alloc显示一个警告（通过调用7187行的bad_pte），并放弃尝试。

11432: 所期待的正常情况：pte_alloc函数返回一个指向包括address地址的PTE的指针。

6. handle_pte_fault

32690: handle_pte_fault函数试图取回或者创建一个缺少的PTE。

32702: 给定的项与物理内存中的任何一个页面都无关联（32700行），而且确实没有被设置（32701行）。这样，do_no_page（32633行）将被调用以创建一个新的页面映射。

32704: 页面在内存中不存在，但是它有一个映射，所以它一定在交换空间里。函数do_swap_page（32569行）将被调用以把该页面读回内存。

32708: 页面在内存里，所以情况可能是内核正在处理一个页面保护冲突。handle_pte_fault首先要用pte_mkyoung（11252行）来把该页面标记为已被访问。

32713: 如果是一个写访问操作而页面又不是可写的，handle_pte_fault就调用do_wp_page函数（32401行）。这个函数完成真正的写拷贝功能。

32715: 这是一次对可写页面的写访问。handle_pte_fault设置该页面的“dirty”位，表示在它被丢弃之前必须被复制到交换空间。

32720: 所需的页面现在可被调用者使用，所以handle_pte_fault函数返回非零值（确定为1）以示成功。

7. update_mmu_cache

11506: 在x86平台上，update_mmu_cache函数是一个空操作。它是一种所谓的“挂钩（hook）”函数——这种函数要在内核的平台无关部分中适当地点保证被调用，以便不同的移植版本都能够在必要的情况下对它进行定义。

8. do_wp_page

32401: 如前所述，真正的写拷贝操作是在这里实现的，所以我们有必要介绍一下。tsk试图写入address，这个地址在给定的vma里并由所提供的page_table来控制。

- 32410: 调用`__get_free_page` (15364行, 简单地转向第34696行调用`__get_free_pages`函数) 为进程提供一个新页面, 此页面是写保护页面的一个新拷贝。注意这里可以允许一个任务转换。有趣的是, 这里的代码不检查`__get_free_page`分配新页面时是否成功——它实际上可能不需要新的页面, 因此到必要时才会去进行检查。
- 32422: 增加“次要 (minor)”页面错误, 这些错误无需访问磁盘就可被满足。
- 32438: 只有两个页面用户存在, 其中一个是交换高速缓存 (swap cache), 它是已被交换出但还未被回收的页面的临时缓冲池。该页面被移出交换高速缓存后 (利用37686行的函数`delete_from_swap_cache`), 现在它就只有一个用户了。
- 32445: 要么从交换高速缓存里回收该页面, 要么它只有一个开始用户。这个页面会被标记为可写和“脏” (dirty, 因为它已被写过)。
- 32448: 如果已分配了一个新页面, 它就没有用了: 由于该页面只有一个用户, 所以没有必要进行拷贝。`do_wp_page`函数释放这个新页面, 并返回非零值表示成功。
- 32454: 页面拥有不止一个用户, 不能简单地从交换高速缓存里被收回。因此`do_wp_page`函数将需要拷贝一个新页面。如果先前的页面分配失败, 现在就是该结果产生作用的时候了, `do_wp_page`函数将不得不返回错误。
- 32459: 利用`copy_cow_page` (31814行) 拷贝页面内容。这通常是调用`copy_page`宏 (10801行), 它是一个`memcpy`。
- 32460: 利用`flush_page_to_ram` (10900行) 使RAM新旧页面拷贝同步。像`update_mmu_cache`函数一样, 在x86平台上这是一个空操作。
- 32463: 像以前一样, 使得页面可写和“脏”, 同时保留从封装的VMA而来的其他页面保护 (比如可执行)。
- 32466: 对函数`free_page` (在15386行, 它只是调用34633行的`free_pages`函数) 的调用不会真正释放旧的页面, 因为该页面拥有多个用户——它只会减少旧页面被引用的次数。由于满足了调用者的请求, `do_wp_page`就返回非零值表示成功。

8.3.4 页面调出

现在读者已经对交换页面调入有所了解, 接下来看一看另一方面, 交换页面的调出。

`try_to_swap_out`

- 38863: `try_to_swap_out`函数是最低一级交换调出函数, 它由内核任务`kswapd` (见39272行`kswapd`函数) 周期性地调用 (通过一系列其他函数调用)。这个函数用来写一个页面, 该页面是由位于给定任务的给定VMA中的一个单独页表项来控制的。
- 38873: 如果内存中缺少该页面, 它就不能从内存写回到磁盘, 这样`try_to_swap_out`函数就返回失败。如果给定的地址明显是不合法的 (`max_mapnr`是当前系统中物理内存的页面数目; 参见7546行), 它也会丢弃尝试操作。
- 38880: 如果页面被保留、锁住, 或者被一个外设用于直接内存访问时, 它就不能被调出。
- 38885: 如果页面最近被访问过, 把它调出可能是不明智的, 因为引用的局部性可能会使该页面不久将再被引用。把该页面标记成“旧的”, 这样将来的再一次尝试就可能把它调出内存——如果内核不顾一切要这么做的话, 这可能很快会发生。但

事实是，页面还没有被调出。

这一行之后的代码注释本身就含有大量信息，所以我们将不失完整性地跳过几段代码。

38965: 减少任务的驻留段长度（注意vma->vm_mm是指向含有vma的struct mm_struct的指针）。驻留段长度是物理内存中的任务所占页面数目，而且很明显，这些页面中的一个现在已经不存在了。

38968: 因为页面无效，所以try_to_swap_out函数必须通知所有TLB以无效化它们对该页面的引用。TLB不应该再把地址解析到一个已经不存在的页面。try_to_swap_out函数接着把这个页面放入交换缓存。

38977: 最后，try_to_swap_out函数通过使用rw_swap_cache（35186行）把旧的页面写回磁盘。写操作是异步的，以便等待磁盘处理时系统也可以做其他工作。

38979: 用__free_page（34621行）来释放页面，并返回非零值表示成功。

8.4 交换设备

Linux拥有一个按优先级排序的有效交换设备列表（及文件，不过为了简单起见，这一部分通常用“设备”来代替这两者）。当需要分配一个交换页面时，Linux会在仍然拥有空间的优先级最高的交换设备上分配它。

Linux也会在所有优先级相同的未满足交换设备之间轮转使用，采用的是循环方式，通过在多个磁盘上分布分页请求的方法可以提高交换的性能。在等待第一个请求被满足时，另一个请求就可以分派到下一个磁盘上。最快的设置是把交换分区分布在几个相似的磁盘上，并给它们同样的优先级设置；而较慢的磁盘则有稍低一些的优先级。

不过，循环也可能造成交换速度的降低。如果同一磁盘上的多个交换设备有同样的优先级，那么磁盘的读/写头将不得不在磁盘上来回反复访问它们；在这种情况下，臭名昭著的1000倍的速度差异就不容忽视了。幸运的是，系统管理员会合理安排优先级以避免这种情况。Linux继承了Unix的传统特性，既能让你陷入绝境，也能使你达成非常良好的目标。最简单的方案是给每个交换设备分配不同的优先级；这会有助于避免最坏的情况，但可能也不会最好。尽管如此，由于该方法简单且有助于避免最坏的情况，因此，如果你不指定优先级设置，它将是缺省设置。

交换设备用struct swap_info_struct（17554行）结构体类型来表示。在37834行定义了这些结构体的一个数组swap_info。好几个文件里的函数都操作和使用swap_info数组来进行交换管理；很快我们就会对它们进行分析。先来分析一下struct swap_info_struct的成员，这会让我们更清楚地了解这些函数。

- swap_device——发生交换的设备号；如果struct代表一个文件而不是分区，值是0。
- swap_file——struct代表的交换文件或分区
- swap_map——对交换空间里每个交换页面的用户数进行计数的数组；为0则表示页面空闲。
- swap_lockmap——用来跟踪基于磁盘的页面当前是否正被读出或写入磁盘，数组里的每一位对应一个页面。在I/O过程中，页面将被锁定以防止内核同时对同一页面执行两次I/O操作或其他类似的愚蠢操作——需要记住的是，一旦有可能，其他进程就会与I/O操

作相重叠，所以发生这种情况并非难事。

- `lowest_bit`和`highest_bit`——跟踪交换设备里第一个和最后一个可用的页面的位置。这可以帮助于加快寻找空闲页面的循环。设备的第一个页面是一个不允许用于交换的头部，因此`lowest_bit`不会是0。
- `cluster_next`和`cluster_nr`——用来对磁盘上的交换页面进行分组以获得更高的效率。
- `prio`——交换设备的优先级。
- `pages`——设备上可用的页面数目。
- `max`——内核在此设备中所允许的最大页面数目。
- `next`——把`swap_info`数组中的所有`struct`形成一个单独的链接列表（并保持优先级顺序）。这样，数组就被逻辑排序，而不是物理排序了。`next`的值就是列表中逻辑指向下一个元素的索引，如果到达列表末尾它就是-1。

`swap_list`在37832行定义，包括列表头（即`head`成员——参见17627行`struct swap_list_t`的定义）的索引；如果列表为空则此索引为-1。它还包括名字很令人迷惑的`next`成员，这个成员能够跟踪我们将要在其上尝试页面分配的下一个交换设备。因此`next`是一个迭代指针。如果列表为空或者当前没有交换，它的值就是-1。

1. `get_swap_page`

- 37879: `get_swap_page`函数从最高优先级的拥有空间的可用交换设备里获得一个页面；如果找到一个，它就返回一个非零代码描述该项，如果系统没有交换就返回0。
- 37885: 从上一次停止的地方继续进行迭代。如果列表是空的或没有剩余交换设备，函数即刻返回。
- 37891: 否则的话，有理由确信存在交换空间，`get_swap_page`函数恰恰需要找到它。这个循环过程一直迭代，直到函数找到一项（很可能的情形），或者扫描了每一个交换设备但没有一个还有剩余空间（不太可能的情形）为止。
- 37894: 利用`scan_swap_map`（37838行）扫描当前交换设备的`swap_map`以寻找一个空闲单元，如果找到了一项，`lowest_bit`和`highest_bit`成员也会被更新。要返回的`offset`是0，或者是该项。
- 37897: 当前的交换设备能够分配一个页面。`get_swap_page`函数现在把`swap_list`的迭代游标向前推进以便请求能正确地分布在交换设备上。

如果已经到达交换设备列表的末尾，或是下一个交换设备的优先级低于当前设备，迭代过程就会从列表的头部重新开始。这产生两个重要作用：

- 如果较高优先级设备的交换空间又变得可用，`get_swap_page`就会在下一次迭代时从那个设备开始分配交换。如果孤立地观察这些代码，读者会认为当高优先级设备可用的时候，这个函数仍可以从低优先级设备分配少数页面。然而事实并非如此，在我们介绍交换页面是如何被释放的时候，读者就会看到这一点。
- 如果优先级高的交换设备不可用，那么在下次内核分配一个交换项时，`get_swap_page`函数将沿列表进行迭代，直到它找到当前优先级的第一个设备为止，并试着从那个设备分配交换。因此，在内核转向优先级较低的设备之前，内核会继续考虑优先级较高的设备直至它们全部耗尽。这就是先前讨论过的循环执行过程。

- 37910: 当前设备没有可用的交换空间, 或者当前设备是不可写的 (这与我们所说的是同样的)。跳到下一个设备, 这样, 如果它已经到达末尾, 但还未曾循环一整圈时, 它就会再从头开始循环。
- 37916: 如果 `get_swap_page` 函数到达列表的末尾, 而且已经循环了一遍, 它就已经考虑了所有交换设备, 但是没有一个拥有空余的空间。因此, 结论是再也没有可用的交换空间了, 函数返回 0。

2. swap_free

- 37923: `swap_free` 函数是与 `get_swap_page` 函数相对的, 它释放一个单独的交换项。
- 37939: 通过许多简单而又周密的测试后, `swap_free` 函数检查是否正在释放交换页面的设备比随后将被考虑的设备具有更高的优先级。如果是, 它就把此作为一个将 `swap_list` 的迭代器重新设置在列表头部的线索。这样, 对 `get_swap_page` 函数的下一次调用就会从列表头部开始并能够检测到新近被释放的高优先级空间。
- 37944: 假如最新被释放的页面处于 `lowest_bit` 和 `highest_bit` 成员所定义的范围之外, 就要相应地对它们进行调整。你可以看到如果 `swap_free` 函数在一个以前已经耗尽了了的设备中释放页面, 这通常会引起对 `lowest_bit` 或者 `highest_bit` 的调整, 但并非都要调整。这会使该区间比所需要的大, 交换页面分配也会因此比所需要的慢。不过这种情况很少发生。无论如何, 交换范围都将调整自己, 以使更多的交换页面能够被分配和释放。
- 37950: 对 `swap_map` 每一元素的使用计数只维护到一个最大值, 即 `SWAP_MAP_MAX` (17551 行定义为 32 767)。达到这个最大值之后, 内核将无法知道真正的计数值有多大; 由此它也无法安全地减少该值。否则的话, `swap_free` 函数将减少使用计数, 并增加空闲页面的总数。

3. sys_swapoff

- 38161: `sys_swapoff` 函数在可能的情况下从交换设备列表中移去被指明的交换设备。
- 38178: 搜索 `swap_info_struct` 的列表以查找匹配的项, 设置 `p` 指向这个数据项, `type` 指向该数据项的索引, `prev` 指向前一项的索引。如果第一个元素被删除, `prev` 将是 -1。
- 38195: 如果 `sys_swapoff` 函数搜索了整个列表, 但没有找到匹配项, 那显然是给定了一个无效的名字。函数返回错误。
- 38198: 如果 `prev` 是负值, `sys_swapoff` 函数将删除列表的第一个元素; 它相应地适当更新 `swap_list.head`。可以证明, 这等价于
- ```
swap_list.head =
 swap_info[swap_list.head].next
```
- 不过速度更快, 因为其中所牵扯的间接转换更少。
- 38203: 如果正被移去的设备是内核进行交换尝试的下一个设备, 迭代游标会被重新设置在列表头部。这样下一次分配可能要稍慢一点儿, 不过并不显著; 无论如何, 实际中这样的情况是相当少见的。
- 38209: 由于设备仍在使用中而不能被释放时, 它会被恢复到列表的适当地方。如果这是数个拥有同样优先级交换设备之中的一个, 它可能不会回到同以前一样的相对位置了——它将是具有同样优先级的设备的第一个, 而不是最后一个——不过列表

仍然是按照优先级进行排序的。

从交换设备列表上删除一个仍有可能被我们放回原处的设备，这看起来就像是在做无用功——为什么不等到确信它可以被删除时再删除它呢？

答案在于经由一系列函数调用遍历`swap_list`后，在前面代码行对`try_to_unuse` (38105行)的调用能够结束。如果正被删除的交换设备那时仍在`swap_list`里，那么终止这一切的代码将会给系统造成极大的混乱。

38223: 若在一个分区上进行交换，`sys_swapoff`函数将解除对它的引用。

38244: `sys_swapoff`函数以使所有数据域无效和释放已分配的内存而告终。特别是这行代码清除`SWP_USED`位，这样，内核就会在它再次利用该交换设备时，知道它已经是不可用的了。接下来，`sys_swapoff`函数清除`err`指示符并返回成功。

#### 4. `sys_swapon`

38300: `sys_swapon`函数是`sys_swapoff`的对应函数，它向系统列表里增加交换设备或交换文件。

38321: 查找未用的一个项。这里有一些微妙之处。读者可能会从`nr_swapfiles`的名字推断出它就是交换文件（或者设备）的数目，但实际它不是。它是曾被使用过的`swap_info`的最大索引值，而且从不会被降低（它记录着这个数组被使用的最高峰值）。因此，把`swap_info`中的这些项循环一遍的结果是，要么发现未用的一个项，要么在最后一次循环增量后让`p`指向第`nr_swapfiles`项之后。在后一种情况下，若`nr_swapfiles`比`MAX_SWAPFILES`小，那么所有用过的项恰好会排在数组的左边，而循环就使得`p`指向它们右边的一个空位。这样，`nr_swapfiles`就会被更新。

有趣的是，即使`nr_swapfiles`不是最高峰值，而是活动交换设备的计数值，循环也能正确执行。不过若我们改变了`nr_swapfiles`的原意，文件里的其他代码就会有问题了。

38328: 在`swap_info`里找到了一个未用的项；`sys_swapon`函数开始对其进行填充。这里所提供的一些值将会发生变化。

38338: 若`SWAP_FLAGS_PREFER`被置位，`swap_flags`的低端15位就被编码为所需的优先级（这里使用的常量和接着的几行代码在17510行进行定义）。否则，就不指定优先级。如前所述，在此情况下的缺省作法是给每一个设备分派一个逐渐降低的优先级，其目的是在无须人工干预时也能得到令人满意的交换性能。

38344: 保证内核允许交换的文件或设备可以被打开。

38352: 检查提供给`sys_swapon`函数的是一个文件还是一个分区。若`S_ISBLK`返回为真，它就是一个块设备，即磁盘分区。在此情形下，`sys_swapon`函数继续确保能够打开该块设备，而且内核此时没有同它进行交换。

38375: 同样，若给定的不是分区，`sys_swapon`函数必须确保它是一个普通文件。若是文件，函数还要确保内核此时没有同该文件进行交换。

38384: 如果两项测试均告失败，`sys_swapon`函数就不会再被请求在磁盘分区或文件上进行交换；它已经拒绝了该尝试。

38396: 从交换设备里把头页面读入`swap_header`；这是一个在17516行定义的`union swap_header`联合体类型。

- 38400: 检查一串特征字节序列, 该序列记述了交换头部的版本信息, 它是由mkswap程序给出的。
- 38412: 交换类型版本1。此时, 该头页面被当作一个大的位映射, 每一位代表设备中剩下的一个可用页面。同其他页面一样, 头页面也是4K字节, 即32K比特。由于每一位表示一个页面, 设备就可以拥有32 768个页面, 也就是每个设备总计128MB (实际上要稍小一些, 因为头页面的最后10位用于签名, 这样我们就不能假定它们对应的80个页面也是可用的; 另外头页面本身也不能用于交换)。如果实际设备比这个值小, 那么头页面中的一些位就不起作用。在38417行, 函数进入循环来检查哪些页面是可用的, 并对它正在创建的swap\_info\_struct的lowest\_bit、highest\_bit以及max成员进行设置。
- 注意这个头页面位映射不会永远被保持——当sys\_swapon函数结束时, 它就会被释放。内核利用交换映射表来跟踪正在使用的页面; 该头页面位映射仅被用来设置lowest\_bit和其他swap\_info\_struct结构体的成员。
- 38427: 分配交换映射表并把所有使用计数值设置为0。
- 38440: 交换类型版本2的交换并没有减轻交换区容量的限制, 不过它以一种更自然和有效的方式贮存头部的信息。在此情形之下, swap\_header的info成员就包含了sys\_swapon函数所需的信息。
- 38451: 新的交换头部版本不需要sys\_swapon函数把头页面当作一个位映射来计算lowest\_bit、highest\_bit和max的值——lowest\_bit总是1, 另外两个值可以从明确存储在头部的信息在定长时间内计算出来。这要比执行32 768次位测试的循环快得多, 也简单得多, 而且后者的定义语句甚至比前者要多出两倍以上! 尽管如此, 这部分及余下的工作, 从概念上讲还是与以前十分相似的; sys\_swapon函数只不过是交换头部直接获取了它所需要的大部分信息, 而无须再计算它们而已。
- 读者现在可以看出我刚才撒的一个小谎; 交换类型版本2实际上真正克服了交换区容量的限制。在这个版本中, 文件末尾的80个页面不会由于交换头部签名而不可利用, 因此, 单独一个设备可以有320K用于交换。不过上限仍然是128MB左右。
- 38491: sys\_swapon函数忽略读取头部。它把设备交换映射表的第一个元素设置为SWAP\_MAP\_BAD (17552行) 以避免内核在头页面上进行交换。
- 38492: 分配加锁映射表并清零。
- 38499: 更新可用的交换页面总数, 并对此结果显示一个消息 (在38502行, 从移位计数器里减去10以便输出结果是千字节表示, 2<sup>10</sup>就是1K)。
- 38505: 在交换设备的逻辑列表中插入新元素, 仍遵循优先级排序的顺序。这里的代码功能上与sys\_swapoff函数中相应的代码是一样的, 所以没理由把它们分离开来。一个能代替两者的内联函数就能简单地解决问题。
- 38519: 进行清理工作, 然后结束。

## 8.5 内存映射mmap

mmap是一个重要的系统调用, 它允许为不同目的而设置专用的独享内存区域。该内存可



能是一个文件或其他特别对象的代理，在这种情形中，内核将保持内存区域和潜在对象的一致，或者该内存可能是为一个应用程序所需要的简单的无格式内存（应用程序通常不使用mmap来分配无格式内存区，因为此时malloc更符合其目的）。

mmap最普遍的使用方法之一是为内核本身通过内存映射（memory-map）形成一个可执行文件（参见8323行的一个例子）。这是关于二进制处理程序如何同分页机制协同工作，以提供按需分页的可执行体，这正如本章早些时候所暗示的。可执行体通过mmap被映射为进程内存空间中的适当区域，然后do\_page\_fault函数调入执行体所需的剩余页面。

被mmap分配的内存可能被标记为可执行，其中充满了指令代码，随后系统跳入其中开始执行；这正是Java Just-In-Time（JIT）编译器的工作方式。更简单地说，可执行文件能够被直接映射成一个正在运行的进程的内存空间；这项技术用于动态连接库的实现中。

执行mmap功能的内核函数是do\_mmap。

#### 1. do\_mmap

33240: do\_mmap函数具有几个参数；它们共同定义要在内存中映射的文件或设备，并决定所创建的内存区域的首选地址及其他特性。

33252: TASK\_SIZE和在10867行定义的PAGE\_OFFSET值相同——即是0xc0000000或3GB。这是用户进程所能拥有的最大内存，在此基础上代码才有意义：显然，如果要求do\_mmap函数分配大于3GB的内存，或者在addr之后的3GB内存空间没有足够的空间，分配请求就必须被放弃。

33275: 如果file为NULL，do\_mmap函数将被请求去执行匿名映射（anonymous mapping）操作，这是一种并不与任何一个文件或其他特别对象连接的映射过程。否则，映射将被关联到一个文件，接着do\_mmap函数要继续检查为内存区域设置的标志位是否与用户在文件上允许执行的操作相兼容。举例来说，在33278行，函数要确保是否内存区可写，因为文件已经被打开并执行写操作了。省略这项判断可能会使文件打开时所做的检查发生混乱。

33307: 允许调用者强调do\_mmap函数应该或者在要求的地址上提供映射操作，或者根本什么也不做。如果提供地址，do\_mmap函数只需保证提供的地址从一个页面的边界开始。否则，它将获得在addr处或之后的第一个可用地址（通过调用始于33432行的get\_unmapped\_area函数），然后就使用这个地址。

33323: 创建一个VMA，并对其进行填写。

33333: 如果内存映射着一个可读文件，则内存区域就被设为可读、可写和可执行（do\_mmap函数可以很快取消写许可——这只是假定）。另外，如果要求共享该内存区域，那么现在就可以满足该请求。

33347: 若文件不可写，则内存区域也必须不可写。

33351: 在此情形中，没有这样的文件，使得do\_mmap函数必须与该文件的打开模式和许可权限相一致——就允许函数自由运行。因此，函数把内存区域设为可读、可写和可执行的。

33361: 在地址范围建立时，利用do\_munmap来清除任何旧的内存映射。因为新的VMA还没有插入进程列表之中（只有do\_mmap函数当前知道它的存在），所以新VMA不会被此次调用影响。

33406: 不会再有错误发生。do\_mmap函数把新VMA插入进程的VMA列表（或者它的AVL树）、合并所有新相连的片段（接下来会对merge\_segments函数进行讨论）、更新一些统计数字，并返回新映射的地址。

## 2. merge\_segments

33892: merge\_segments函数是一个有趣的函数，它把相邻的VMA合并成单独的一个大范围的VMA。换句话说，如果一个VMA所覆盖（有意这样设计）范围是从0x100到0x200，而另一个VMA的覆盖范围是从0x200到0x300，并且两者保护信息相同，那么merge\_segments函数就会用一个覆盖范围从0x100到0x300的单独VMA来代替它们（注意函数名中的“segments”并不表示此时我们采用CPU分段机制）。merge\_segments函数的参数是结构体struct mm\_struct，它包含了我们感兴趣的VMA以及可能进行合并的开始地址和终止地址。

33897: find\_vma\_prev（33501行）函数将其vm\_end定位在给定的start\_addr之后的第一个VMA上——由此，第一个VMA可能会包括start\_addr。回忆一下find\_vma\_prev函数，它也返回一个指向前一个VMA的指针prevl（如果第一个VMA满足条件则返回值是NULL）。

33911: 进入处理所有覆盖给定范围的VMA的循环。在该循环过程中，merge\_segments函数将尝试把每一个片段都与其前一个片段进行合并，而前一个片段的值可以通过prev获得。

33921: 绝大部分条件判断都是相对直截了当的，不过最后一个测试就不这么简捷了。它确保prev和mpnt是连续的——也就是在prev的结尾和mpnt的开头之间没有未被映射的内存。即使检测结果是一个的vm\_end和另一个的vm\_start相等，这两块区域在这一点上也未必一定相互覆盖——vm\_end是要比VMA拥有的最后地址还要大一位的。33926行到33932行的代码为被映射文件和共享内存坚持了同样的特性：一块区域的末尾要等于下一块的开头。

33937: merge\_segments函数找到了可以合并的VMA。它把mpnt从VMA列表（还可能是AVL树）里移去，再将它存入prev。要注意的是即使VMA的数目降到了MIN\_MAP\_COUNT以下，它都不会拆除AVL树。

33948: 如果将要消失的VMA是一个被内存映射的文件的一部分，merge\_segments函数就删除它对该文件的引用。

## 3. do\_munmap

33689: do\_munmap函数明显是do\_mmap函数的反作用函数；它从一个进程的内存空间里废除虚拟内存映射。

33695: 如果do\_munmap函数被要求取消映射的地址不是页面对准的，或者地址区域位于进程的内存空间之外，那么很明显它就是无效的，因此请求就会被拒绝。

33699: 如果连一个页面也没有被释放，就拒绝尝试。

33707: 查找包括给定地址的VMA。令人奇怪的是，do\_munmap函数返回的是0——而不是错误，如果地址不在任何一个VMA之内的话。从某种意义上讲，这是正确的；do\_munmap函数被要求用来确保一个进程不再对特定内存区域进行映射，如果一开始就没有这种映射的话，那就很容易办到。不过这仍颇为奇怪；在调用

者看来这是一个错误，而且do\_munmap函数也应该报告这个错误。然而，某些调用程序却希望它如33361行的示例那样执行工作。

33717: 如果给出的内存区域整个在单独的一个VMA中，但又不该VMA的一端，那么移去这段区域就会在封闭的VMA里生成一个空洞。内核是不会容忍这个空洞的，因为按照定义，VMA应该是连续的一段内存。因此在这种情况下，do\_munmap函数就需要创建另一个VMA，使得空洞的两边各有一个VMA。尽管如此，如果内核已经为该进程创建了所允许的所有VMA，那么函数就不能这样做了，所以此时do\_munmap函数不能满足请求。

33730: 标识所有与该区域相交或在区域里的VMA为空闲状态，同时把每一个都放在本地堆栈free里。顺着这个过程，do\_munmap函数会把VMA从它们的AVL树中删除（如果有的话）。

33743: do\_munmap函数已经建造了要释放的VMA堆栈，现在释放它们。

33748: 计算要释放的准确范围，要牢牢记住的是这个范围可能不能以完整的VMA来度量。假如为min和max的定义适当，这三行可以被写成如下代码：

```
st = max(mpnt->vm_start, addr);
end = min(mpnt->vm_end, addr + len);
```

由此，st是do\_munmap函数实际开始释放区域的开头，end是该区域的结尾。

33765: 如果VMA是共享映射的一部分，do\_munmap函数通过调用remove\_shared\_vm\_struct（33140行）来断开mpnt与共享VMA列表的连接。

33759: 更新MMU数据结构，它对应于这个VMA里当前被释放掉的子区域。

33765: 调用unmap\_fixup函数来修补映射，我们接下来就会对这个函数进行研究。

33773: do\_munmap函数已经释放了该范围内由VMA代表的所有映射；最后重要的一步就是要为同一区域释放页表，这是通过调用free\_pgtables（33645行）实现的。

#### 4. unmap\_fixup

33578: unmap\_fixup函数修补给定VMA的映射，这可以通过对一端进行调整，或者通过在中间制造一个空洞，又或者通过把VMA完全删除的方法来完成。

33590: 第一种情况比较简单：去掉整个区间的映射。do\_munmap函数仅仅需要关闭底下的文件或其他对象即可（如果它们有的话）。读者可以看到，这无须把VMA本身从current->mm里移出，它已经被调用者删除了。因为VMA的全部范围将被解除映射，没有什么要向后推移的，所以unmap\_fixup函数就此返回。

33599: 接下来的两种情况处理把VMA从开头到末尾一块区间移去的问题。这也是比较简单的：它们的主要工作是要调整VMA的vm\_start或vm\_end成员。

33608: 这是四种情况中最有意思的一种——从一个VMA的中间移去一块区域，从而会产生一个空洞。函数先开始要复制一份额外生成的VMA的本地拷贝，然后通过将\*extra设置为NULL来通知调用者该附加VMA已被使用。

33611: 图8-4表示了分裂VMA的过程。大部分信息被直接从旧VMA复制到了新VMA，在此之后，unmap\_fixup函数对两个VMA的范围都做了调整以解决空洞问题。原先的VMA、area被缩小到了表示低于空洞的子区域，而mpnt则表示高于空洞的子区域。

33626: 把全部新子区域插入current->mm。

33629: 除了第一种的其他情况下, unmap\_fixup函数保持了旧的VMA。它缩小了, 但还未消失, 因此它将被插回到VMA的current->mm集合中。

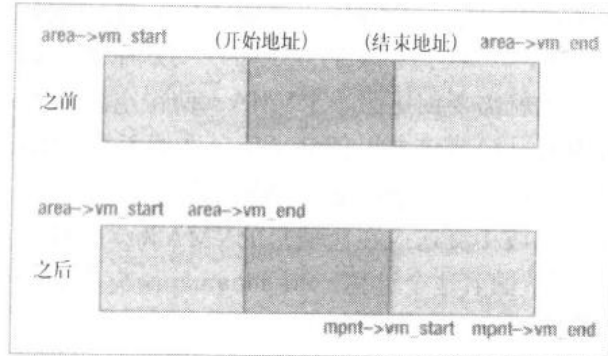


图8-4 VMA的分裂

## 8.6 用户空间和内核空间的动态内存

用户任务和内核本身都需要经常快速分配内存。C程序一般使用著名的malloc和free函数来完成这项工作; 内核也有它自己类似的机制。当然, 内核必须至少提供支持C语言的malloc和free函数的低级操作。

在Linux平台上, 就像其他的Unix变种一样, 一个进程的数据区分为两个便于使用的部分, 即“栈”和“堆”。为了避免这两个部分冲突, 栈从(准确的是接近)可用地址空间的顶端开始并向下扩展, 而堆从紧靠代码段上方开始并向上扩展。虽然可以使用mmap在堆和栈之间分配内存, 但是这部分空间通常是没有使用的内存的空白地带。

即使不去研究有关的内核代码(不过我们还是要继续这项工作), 读者也能对这些地址区间所处位置有相当好的了解。下面的短程序显示了几个挑选出来的对象的地址, 它们分处于三种不同内存区域之内。由于种种理由, 我们不能保证它可以被移植到所有平台上, 不过它可以在Linux的任何版本下工作, 而且也应该可以被移植到你所尝试的大部分其他平台上。

```
#include <stdio.h> /* printf(). */
#include <stdlib.h> /* malloc(). */
static void
one(void * p, const char * description)
{
 printf("%10p: %s\n", p, description);
}

int
main(void)
{
 int i;
 int j;

 one(&i, "A stack variable, \i");
 one(&j, "Next stack variable, \j");
 one((void *) one, "The function \one");
}
```

```

 one((void *) main, "The function \"main\"");
 one(malloc(1), "First heap allocation");
 one(malloc(1), "Second heap allocation");
return 0
}

```

在我的系统上，我得到了如下的数字。你的结果可能会稍有不同，除了所使用的编译器标志外，它还取决于你的内核及gcc的版本。即使不完全相同，它们也应该与下面结果相当接近。

```

0xbffff824: A stack variable, "i"
0xbffff820: Next stack variable, "j"
0x80484a0: The function "one"
0x80484bc: The function "main"
0x80496f8: First heap allocation
0x8049708: Second heap allocation

```

从这里你不难看出，如果使用大概的数字的话，栈从接近0xC0000000处开始并向下生长，代码从0x8000000处开始，而堆则如前所述从临近代码上部的地方开始并向上扩展。

### 8.6.1 brk

系统调用brk是一个在C库函数malloc和free底层的原语操作。进程的brk值是一个位于进程堆空间和它的堆、栈中间未映射区域之间的转折点。从另一个角度看，它就是进程的最高有效堆地址。

堆位于代码段顶端和brk之间。如果brk底下的可用自由空间不够满足请求，C库函数malloc就抬高brk；如果被释放的空间位于brk之下，就降低brk。顺便说一句，Linux是我所知道的唯一的在使用free函数时真正的减少进程内存空间的Unix变体；其他我所经历过的所有Unix商业版本实际上都是保留该进程的空间的——显然这是“以防万一”的作法。（其他Unix的自由版本可能同Linux一样，不过我没有使用过。）另外，对于大量的分配工作，GNU的C库使用mmap和munmap系统调用来执行malloc和free。

代码、数据，以及栈的关系如图8-5所示。

sys\_brk

33155: 实现brk的函数是sys\_brk。它可以修改进程的brk值，还可以返回一个新值。如果无法修改brk的值，返回的brk值就等于其原值。

33177: 如果brk的新值位于代码区域之中，它就明显偏低，从而必须被抛弃。

33179: 通过使用宏PAGE\_ALIGN（10842行）把brk参数向上取整到地址更高的下一个页面。

33180: 按页对准进程原有的brk值。这看起来有些多余，因为如果进程的brk只是在这里被设置，它就一定是按页排列的。但是在初始化一个进程的时候，进程的brk可以被设置在别的地方，代码并不会把它按页对准排列。不管进程的brk在哪里被设置，把它按页对准都可能会快一些；允许内核在这里跳过一次页对准操作，而且由于此处要比别的地方更频繁地对进程的brk进行设置，它应该不会降低执行效率，而且还会少许提高。

33185: brk被降低了，不过还没有进入代码区域，因此尝试被允许。

33192: 如果堆的大小有限制, 它就要被考虑。图8-5清楚地表明, `brk - mm->end_code`是堆的大小。

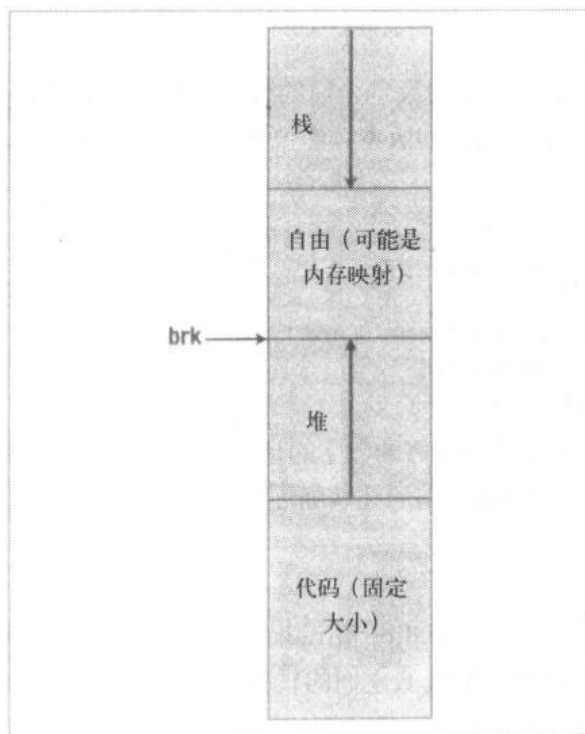


图8-5 代码、数据和堆栈

33197: 如果`brk`扩展到了已被一个VMA所内存映射的 (`mmapped`) 区域, 它就是不可利用的, 因此这个新`brk`值要被舍弃。

33201: 最后一项必要的检查是察看是否存在足够的自由页面用于空间分配。

33205: 使用`do_mmap`函数 (33240行) 为新区域分配空间。然后, `sys_brk`函数更新进程的`brk`的位置并返回新值。

## 8.6.2 `vmalloc`和`vfree`

内核编程中一个有趣的方面是并没有像应用程序编程人员通常所想当然的那样能够得到很多服务。就拿`malloc`和`free`作为例子, 它们就是建立在一个内核原语`brk`之上的C库函数。

假使内核被修订以使其可以和标准C库链接, 并使用它的函数`malloc`和`free`, 那么最终结果将是既笨拙又缓慢——这些函数被要求从用户模式调用, 所以内核将不得不切换到用户模式去调用它们, 然后它们又不得不掉转回到内核, 还必须要对整个过程进行监控, 等等。为了避免这一切, 内核有许多十分熟悉的函数自己的版本, 它们包括`malloc`和`free`在内。

的确, 内核提供了像`malloc`和`free`一样的两对独立的函数。第一对是`kmalloc`和`kfree`, 管理在内核段内分配的内存——这是真实地址已知的实际和物理内存块。第二对是`vmalloc`和`vfree`, 用于对内核使用的虚拟内存进行分配和释放。由`kmalloc`返回的内存更适合于类似设备驱动的程序来使用, 因为它在物理内存里而且是物理连续的。不过, `kmalloc`要比`vmalloc`所能使用的资源少, 因为`vmalloc`还可以处理交换空间。

`vmalloc`和`vfree`的一部分也是通过`kmalloc`和`kfree`来实现的，因为它们需要一部分不可交换的内存用于登记操作（bookkeeping）。`kmalloc`和`kfree`又依次使用`__get_free_pages`、`free_pages`及其他低级页面操作函数实现的。

在此我不对`kmalloc`和`kfree`进行解释，不过本书中提供了相关代码以供读者阅读（分别见37043和37058行）。我将要讨论的是更有意思的函数`vmalloc`和`vfree`。

### 1. `vmalloc`

38776: `vmalloc`函数拥有一个参数，即要分配的内存区域的大小。函数返回指向分配区域的指针，如果无法分配就返回`NULL`。

`vmalloc`可以分配内存的虚拟地址范围，这是由常量`VMALLOC_START`（11081行）和`VMALLOC_END`（11084行）决定的。`VMALLOC_START`从超过物理内存结束地址8MB的地方开始，以便对任何在这一区域错误的内核内存访问进行截获，`VMALLOC_END`在接近可能的最大32位地址4GB的地址处。除非你的系统拥有比我的系统多得多的物理内存，否则这就意味着几乎整个CPU地址空间都潜在地可为`vmalloc`所用。

38781: 如果不在一个页面的边界上的话。`vmalloc`函数首先把要求的区域大小向上取整到地址更高的下一个页面边界，（`PAGE_ALIGN`宏在10842行定义）。如果最终范围结果太小（0）或明显过大，则请求会被拒绝。

38784: 利用`get_vm_area`来为`size`大小的块定位一段足够大的内存区域，这个函数接下来会进行介绍。

38788: 通过调用`vmalloc_area_pages`（38701行）保证能够建立页表映射。

38792: 返回被分配的区域。

### 2. `get_vm_area`

38727: `get_vm_area`函数返回从`VMALLOC_START`到`VMALLOC_END`的一段自由内存区间。通常这就是`vmalloc`函数的工作；它还用于我未曾提及的其他少数场合。调用程序有责任确保参数`size`是一个非零的页面大小的倍数。

`vmalloc`函数采用所谓的“首次适应算法（first-fit algorithm）”，因为它返回一个指向定位区域的指针，该区域是它所能找到的第一个满足请求的区域。除此而外，还有“最佳适应算法（best-fit algorithm）”，该算法选取足够满足需求的最小的一块可用自由区域进行分配，还有“最坏适应算法（worst-fit algorithm）”，该算法总是分配最大的一块可用自由区域。每种分配方式都有优点和缺点，不过“首次适应算法”在这里对要达到的目的来讲，就已经非常简单、快捷，而且足以满足要求了。

38732: 分配一个`struct vm_struct`来代表新的区域。被分配的区域用一个有序链表，即`vm_list`（38578行）来维护，该链表是由`struct vm_structs`构成的。包括`struct vm_struct`结构体的头文件被省略以节约空间，不过结构体的定义十分简单：

```
struct vm_struct {
 unsigned long flags;
 void * addr;
 unsigned long size;
 struct vm_struct * next;
};
```

如图8-6所示，链表的每一个元素都与单独一块已分配了的内存块相关联。形象地看起来，`get_vm_area`函数的任务就是在已分配的区域之间找出足够宽的间隔。

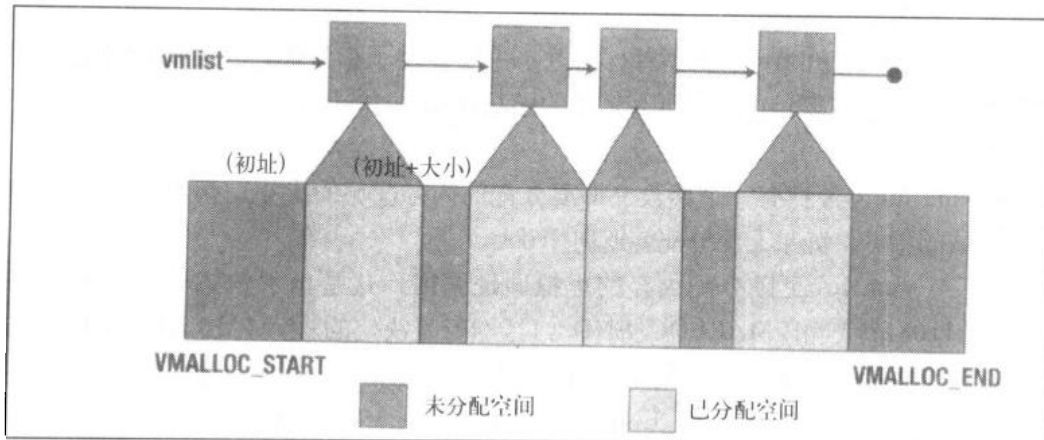


图8-6 vmist链表

- 38737: 沿着链表进行循环。循环的结果要么是找到一个足够大的自由区域，要么是证明这样的区域不存在。它会先从VMALLOC\_START开始尝试，然后挨个尝试紧随着每块被分配区域之后的地址。
- 38746: 链表为空或者循环发现了一个足够大的新块；无论哪种情形，现在`addr`都是最小可用地址。填充新的`struct vm_struct`结构体，它将会被返回。
- 38747: 给保留块增加一个页面的大小（x86平台上是4K），来捕获内核超出的内存——可能的话还包括下一个更高地址块下方的内存。因为在决定是否当前区域足够大的时候（38738行），并没有把这些额外的空间算在内，那么保留区域可能会与接下来的一个相重叠，而且内核内存中超出而进入这个“额外”区域的部分也确实可能覆盖到被分配了的内存。

事实不是这样。我们很容易证明`addr`总是页对准的，而且我们也已知道`size`总是页面大小的倍数。因此，`addr + size`要小于接下来区域的开始地址，它至少是一整页。当然超出范围多于一页的内存会进入下一个区域，不过超出范围少于一页的内存就不会这样。

因为内核不会为额外内存建立页面映射，所以对它的错误访问将造成不可解决的页面错误（这在Linux的现代版本中几乎还未听说过）。这将会带给内核一次痛苦的中断，不过那要比允许内核悄然无息地破坏自己的数据结构要好一些。至少你可以立刻知道这个折磨人的系统停机，它可以帮助你诊断问题所在；而后一种作法可能在内核已经破坏了你的磁盘之后，才能看出它的危害。

### 3. vfree

- 38753: `vfree`函数比`vmalloc`简单得多（至少，要是把`get_vm_area`加进`vmalloc`是这样的），不过为了完整起见，我们还是要对`vfree`略为讨论。当然`addr`是要被释放的已分配区域的开头地址。
- 38763: 在几项简洁而又完善的测试之后，函数沿着`vmist`进行循环，搜索要释放的区域。这个线性查找过程使我想到一件有趣的事，假如采用一个如同VMA管理所用的AVL树那样的平衡树结构，也将会提高`vmalloc`和`vfree`函数的性能。



- 38764: 当与addr相匹配的struct vm\_struct被找到时, vfree函数就把它从链表里分离出去, 并释放该结构体和它所关联的页面, 然后返回。每个struct vm\_struct不仅记录它的初址, 还记录区域的大小, 这一点对于get\_vm\_area是便利的, 在这里同样也颇为便利, 因此, vfree函数是知道应该释放多大空间的。
- 38772: 如果vfree函数在链表里找到了匹配项, 它在此之前就应该已经返回了, 所以没有找到匹配项。这是一个坏事, 不过还未糟糕到不可收拾的地步。这样, vfree函数以显示一个警告而结束。

## 8.7 主存储器信息转储

在一些情况之下, 比如一个满是“臭虫”的程序试图去访问所允许的内存空间之外的内存时, 进程可以“信息转储 (dumping core)”。进行“信息转储”就是把一个进程的内存空间的映象 (随同一些关于应用程序本身和其状态的识别信息一起) 写入一个文件以备将来使用诸如gdb之类的调试器进行分析的过程 (“磁心”是一个差不多已经过时的“内存”术语)。

当然, 或许你的代码从来不会犯这样的错误, 但这可能会发生在别的程序员身上因此在此我要对此问题进行一些讨论。

不同的二进制处理程序完成转储内核的方式不同 (第7章里论述过二进制处理程序)。最常用的Linux二进制格式是ELF, 所以我们来看看ELF二进制处理程序是如何进行转储内核的。

elf\_core\_dump

- 8748: elf\_core\_dump 函数由此开始。因为一个进程转储内存是由接受到一个信号量而引起的 (它也可能发送给自己, 例如通过对abort的调用), 该信号量编号在signr中被给出。signr对进程是否或如何执行转储内存没有影响, 但是在调试器里看内存文件的用户却想要知道是哪个信号导致内存转储的, 它就像是一个关于出了什么错的提示一样。指向struct pt\_regs (11546行)的regs参数包含一份对CPU寄存器的描述。regs的重要性除了一些其他原因外, 还在于它包含了EIP寄存器的内容, 该寄存器是指令指针, 它决定了收到信号时所执行的指令。
- 8771: 假如进程未通过一些基本检查则立即返回, 这些检查中的第一个是确保dumpable标志被设置。进程的dumpable标志 (16359行) 通常会被设置; 它的清除主要是在进程改变其用户或组ID的时候。这似乎是一项安全措施。例如, 我们不愿意创建一个被设定为root的不可读执行程序的可读内存文件——那会使得保证执行体不可读的目的 (出于安全考虑) 遭到失败。
- elf\_core\_dump 函数此时也会返回, 假如内存文件的大小限制使得连一个页面也无法转储, 或者如果有其他线程要引用将要转储的内存。转储内核是和退出进程相关的, 从用户的角度来看, 只要进程任何一个线程还存在, 它就没有消亡。
- 如果进程通过了这些测试, elf\_core\_dump 函数就继续运行, 并清除dumpable位以便它不会再次尝试转储进程的内存。(尽管这种情形不能会发生; 我认为这只是预防式的编程设计)。
- 8785: 进入一个循环, 对内存文件大小限制之内可以被转储的VMA个数进行计数。尽管elf\_core\_dump 函数把计数值保存在叫做segs的变量里, 它并不表明我们正对本章

中所使用过的“内存段”进行计数。不要认为这个变量的名字有其他特别的附加涵义。

由于elf\_core\_dump函数在转储VMA之前要向内存文件写一些头部信息，而且这些头部的大小没有进行计算，因此输出结果可能会稍微超出内存文件的大小限制。这不难解决：一个简单的策略是在写入头部时递减limit，并把循环计数移动到头部写入代码之后。实际解决方案要更麻烦一些，不过也并不是十分复杂。

- 8805: ELF内存文件格式根据正式规范进行定义；第一部分是描述文件的头部。结构体struct elfhdr类型（参见14726和14541行）定义了头部的格式，elf\_core\_dump函数填写这个类型的一个局部变量elf。
- 8827: 创建要转储到的文件名，并尝试打开这个文件。通过把8828行的#if 0改变为#if 1，我们可以让内存文件名包括生成文件的执行程序的名字（或至少是名字的前16个字符——参见在16406行定义的struct task\_struct的comm成员）。有的时候这是一个很有用的特性：一看到内存文件的名字就能够可以马上知道是什么应用程序生成的。不过，这种行为并不标准，而且还有可能破坏已有代码——例如监视器脚本程序，它周期性地检查名为“core”的文件是否存在——所以缺省行为还是为遵守标准惯例而把文件命名为普通的“core”。尽管如此，发现这么一个可以调整的内核参数还是不错的。这个可选项也对8756行局部变量corefile那看似与众不同的定义方式进行了解释。
- 8853: 设置PF\_DUMPCORE标志（16448行），发出信号表明该进程正在转储内核。这个标志不在本书所涉及的任何代码中使用，它被用于读者将要了解的记帐进程。记帐进程（process accounting）跟踪一个进程的资源使用情况和其他的一些相关信息——包括它是否在退出时转储内核——这些信息原本是用来帮助计算中心计算应向每个资源使用部门或用户收取多少费用的。
- 8855: 写入早先建立的ELF内存文件头部。这里要涉及一些隐含的流控制：定义在8707行的DUMP\_WRITE宏，使得elf\_core\_dump函数在写操作失败时关闭文件并返回。
- 8862: 跟在ELF内存文件头部之后的是一系列节点（note）；它们中的每一个都有特殊目的，记录着有关进程的特定信息。我们将逐一对其论述。一个注解（数据类型是struct memelfnote，8666行）包括一个指向辅助数据（它的data成员）的指针和该数据的长度（它的datasz成员）；填写一个注解的大部分工作就是填充辅助数据结构，然后使该注解指向它。
- 有些信息被存储在若干个注解里。代码中没有对这种重复进行解释，但是，其中至少有一部分原因是从Unix的变种中拷贝它们的行为方式。保持文件格式和其他平台一致有助于把诸如gdb这样的程序移植到Linux上来；少许重复要比延迟移植版本的进度和增加诸如此类的复杂关键工具的维护要好得多。
- 8865: 注解0在辅助数据结构体（类型struct\_elf\_prstatus；参见14774行）里记录了进程的继承关系、信号量及CPU的使用情况。我们需要特别注意8869行的elf\_core\_dump，它存储了引起进程转储内核的信号编号。所以当在一个内存文件上运行gdb而它显示“Program terminated with signal 11, Segmentation fault”的时候，你就会知道该信息是从哪里来的了。

- 8916: 注解1在辅助数据结构体psinfo (属于类型struct\_elf\_prpsinfo; 参见14813行) 里记录了进程的属主、状态、优先级等等信息。8922行有一个虽然正确、但很不寻常的指向一个文字字符串常量的数组下标; 被选择的字符是进程状态的一个记忆码。这与ps程序的STAT域报告的状态字是一样的 (除非下标溢出)。更有意思的是8945行, 代码把执行体的名字 (如前所述, 最多16个字符) 复制进了注解。gdb和程序“文件”都用这个字段来报告是哪一个程序生成的内核转储。
- 8948: 注解2记录转储进程的struct task\_struct, 这明显存储了大量关于该进程的必要信息。因为struct task\_struct内的一些信息是由当调试器检查代码时便不再有效的指针组成的, elf\_core\_dump 函数随后还会独立转储一些指针所指向的信息——最紧要的, 如进程的内存空间。
- 8954: 如果这个系统包含一个FPU (浮点计算单元), 那么就会据此而生成一个注解。否则, 8957行对所存储的注解数目进行递减。
- 8968: 对于每个被创建的注解, 都有一个描述该注解的头部; 而注解本身会紧随其后。注解头是struct elf\_phdr类型; 参见14727和14581行它的定义。
- 8992: 这是写入进程内存空间的第一步。在这里, 函数写入头部信息 (又一次是phdr), 该头部描述了它将要写入的所有VMA。
- 9016: 最后, elf\_core\_dump 函数才真正地写入它先前辛辛苦苦创建好的各个注解 (内存文件)。
- 9022: 在文件里向前跳过4K到达下一个边界, 内存文件真正的数据是从这里开始的。完成此项操作的DUMP\_SEEK宏在8710行定义, 像DUMP\_WRITE宏一样, 假如搜索失败, 它也会导致elf\_core\_dump 函数的返回。
- 9024: 在所有那些准备之后, 这里的工作简直有些虎头蛇尾。不过, 这才是转储内核的主要部分: 写入进程的每一个VMA直至先前计算出并保存在segs里的上限。接下来是少许收尾工作, 然后elf\_core\_dump 函数就完成了使命。

## 第9章 System V IPC

Unix在开发的早期就提供了管道的机制，管道在同一机器的两个进程间的双向通信方面工作的相当出色。后来，BSD (Berkeley Software Development) 的Unix版本又提供了通用的套接字socket，它用来在不同机器的（或者是同一机器的）两个进程间进行通信。

Unix System V版本增加了被视为一体的三个机制，现在它们被统称为System V IPC。像管道一样，这些机制都可以用于同一机器上的进程间通信，不过与管道和套接字不同的是，System V的IPC特性使得同一机器上的许多进程间都可以互相通信，而不是仅限于两个进程。而且，管道——不是套接字——还有一个更大的限制就是两个通信中的进程必须相关。它们必须有一个建立管道的共同祖先进程——通常情况下，一个进程是另一个的父进程，或者这两个都是为它们建立管道的父进程的子进程。System V IPC像套接字一样，使得进程间通信 (IPC) 不需要有共同的继承关系，只需要一个经过协商的协议。

组成System V IPC的三个进程间通信机制是：消息队列、信号量和共享内存。

### 9.1 消息队列

System V的消息队列是进程间互相发送消息的一种异步方式，在这种情形之下，发送方不必等待接收方检查它的消息——即在发送完消息后，发送方就可以从事其他工作了——而接收方也不必一直等待消息(原文为：“the receiver doesn't have to go to sleep if no messages are waiting.”，直译是：如果没有消息正被等待，接收方也不必进入休眠。——译者注)。对消息进行编码和解码是发送者和接受者进程的工作；消息队列的执行并不会给它们特别的帮助。这就形成了一个实现起来相对比较简单的通用机制，尽管是以增加应用程序的复杂度为代价来获得这种简明性的。

这里是一个可能发生在SMP机器上的简单的应用情景。运行在一个CPU上的调度程序把工作请求发送到一个特定的消息队列上。工作请求可能以各种形式出现：用来破译代码的一组密码、需要进行计算的在不规则图形里的像素范围、在一个原子系统里要更新的一部分空间、或者诸如此类的任务。与此同时，工作者进程在其他CPU上运行，只要它们空闲就从消息队列中检索消息，然后再把结果消息发送到另一个消息队列上去。

这种体系结构很容易实现，而且假定选择好了每个消息中被请求工作的粒度，就能极大地提高机器中CPU的利用效率（还要注意的，因为调度进程可能不用做许多工作，所以调度进程的CPU上大部分时间也可以运行一个工作者进程）。以这种方式，消息队列可以被用作是远程过程调用 (RPC) 的一种低级形式。

新消息总是加在队列的末尾，不过它们并不总是从排头移出。在某个方面，消息队列与语音邮件类似：新消息总是在末尾，不过消息接收方可以从列表的中间接收（以及删除）消息。

### 消息队列概述

首先介绍消息队列是因为它的实现最简单，不过它仍然体现出了几个所有三种System V

IPC机制都具有的共同结构特征。

给进程提供了四种与队列相关的系统调用：

- `msgget`——一个不合时宜的名字：读者可能认为这会得到一个等待的消息。但实际它不会。调用者提供一个消息队列键标（key），如果存在一个队列，`msgget`就用该键标为它返回一个标识号，如果没有队列，就用它为一个新的消息队列返回一个标识号。因此，`msgget`所得到的不是一个消息，而是唯一标识一个消息队列的标识号。
- `msgsnd`——向一个消息队列发送一条消息。
- `msgrcv`——从一个消息队列中接受一条消息。
- `msgctl`——在消息队列上执行一组管理操作，检索关于它的限制的信息（比如队列所允许的最大消息数据量）、删除一个队列，等等。

### 1. struct msg

15919: `struct msg`代表在队列中等待的一个消息。它有如下成员：

- `msg_next`——指向队列中的下一个消息，假如这是最后一个消息就为NULL。
- `msg_type`——用户指定类型编码；在本章讨论消息如何被接收时再分析它的使用。
- `msg_spot`——指向消息内容的开头。读者后面将看到，为消息分配的空间总是紧靠在`struct msg`的上边，因此`msg_spot`恰恰指向`struct msg`末尾之后的位置。
- `msg_stime`——记录消息被发送的时间。因为消息以先进先出（FIFO）顺序保存，所以队列中的消息拥有的`msg_stime`值就是单调非递减的。
- `msg_ts`——记录消息的大小容量（“ts”是“text size”的缩写，尽管消息不一定非要是人们可以读懂的文本）。一条消息的最大容量是MSGMAX，它在15902行定义为4056字节。推测一下，这应该是4K（4096字节）减去一个`struct msg`的结果。不过b只有20字节，因此，还有另外的20字节有待说明。

### 2. struct msqid\_ds

15865: `msqid_ds`代表一个消息队列。它有如下成员：

- `msg_perm`——说明哪一个进程可以读写该消息队列。
- `msg_first`和`msg_last`——指向队列中的第一个和最后一个消息。
- `msg_stime`和`msg_rtime`——分别记录消息被发送入队列的最后时间和消息从队列中读出的最后时间。（什么时候队列中最后一条消息的`msg_stime`成员不等于队列本身的`msg_stime`成员？至少有两个答案，但是你所掌握的信息现在只能得出一个，仔细阅读代码可以寻求另一个答案。）
- `msg_ctime`——上一次队列改变的时间，即它被创建的时间，或是上一次利用`msgctl`系统调用设置参数被确信的时间。
- `wwait`——等待写消息队列的进程队列。因为消息发送是异步的，通常进程把一个消息写入消息队列后就可离开。但是，为了避免拒绝服务（denial-of-service）的攻击，队列有一个最大容量——若没有这个限制，一个进程就可以不断地向一个没有读者的队列发送消息，强迫内核为每个消息分配内存，直至空间耗尽。因此，当一个队列达到其最大容量时，想要发送消息给该队列的进程必须等待，直到队列中有了空间容纳新的消息，或者发送消息的尝试被立刻拒绝为止（读者将看到，进程能够选择它所希望的执行方式）。`wwait`队列保留那些决定等待的进程。

- `rwait`——与之类似，消息通常可以从消息队列中立刻读出。但是如果没有正等待被读的消息将怎么办呢？进程再次进行选择：它们要么立刻重获控制（用一个错误代码表示读消息失败），要么进入休眠等待消息到来。
- `msg_cbytes`——当前在队列中的所有消息的总字节总数。
- `msg_qnum`——队列中消息的总数。对于能够进入队列的消息数目没有明确的限制（这也是一个问题，本章随后还要进行解释）。
- `msg_qbytes`——队列中允许存储的所有消息的最大字节数；比较`msg_cbytes`和`msg_qbytes`以确定是否还有空间容纳新消息。`msg_qbytes`缺省为`MSGMNB`，尽管这个限制可以被有适当权限的用户动态地增加。

`MSGMNB`在15904行定义为16384。有四个理由可以说明为什么这个界限被定的这样低。第一，实际上，你通常不需要把太多的信息包括在一个给定的消息中，所以这个界限并不是十分苛刻的。第二，如果消息发送方的速度远远领先于接收方，那么让消息能多包含些信息可能也没有意义——它们还将是接收方要费些时间才能得到的一大块数据。第三，这个每队列16K的界限可以与潜在的128个队列相乘，总计达2MB。但是，采用这个界限的主要原因还是为了避免先前提及的拒绝服务攻击。然而，没有什么能防止应用程序发送长度为零的（也就是空的）消息。`msg_qbytes`不会被这样的消息影响，而且仍然要给消息头分配内存，因此拒绝服务攻击仍然是可行的。解决这个问题一个方案是引入一个独立的、对允许进入队列的消息总数进行限制的界限；另一个方案是从`msg_qbytes`中减去整个消息长度——包括消息头。再一种解决方法当然是不允许有空消息，但这又将同兼容性相抵触。

- `msg_lspid`和`msg_lrpid`——最后消息发送方和最后消息接收方的PID。

### 3. `msgque`

20129: 消息队列实现中的主要数据结构是`msgque`，一个指向`struct msqid_ds`的指针数组。这些指针有一个是`MSGMNI`（15900行定义为128），它等于128个消息队列的最大值。为什么不只是用一个的数组而要用一个指针数组呢？一个原因是为了节省空间：替代一个128个56字节结构体的数组（7168字节，7K），`msgque`是一个128个4字节指针的数组（512字节）。在正常情况下，当很少的消息队列投入使用，这能够节约好几千字节的内存。在最坏的情况时，所有的消息队列都被分配了，最大的消耗也只是512字节。唯一会引发的真正缺点是附加了一层间接转换，这意味着速度要有少许损失。

主要消息队列数结构之间的关系如图9-1所示。

### 4. `msg_init`

20137: `msg_init`用于消息队列实现时变量的初始化。大部分都是不必要的，因为同样的变量已经在函数前面紧挨本段代码的声明中被初始化为同样的值了。

20142: 然而这个把`msgque`的项目设置为`IPC_UNUSED`的循环是必要的。`IPC_UNUSED`不在本书讨论之列，值为-1（能够更好的被映射为`void*`）；它代表一个没有使用的消息队列。`msgque`的项目可能接纳的其他特殊值是`IPC_NOID`（也不在本书讨论之列）——这只是暂时的，也就是在消息队列被创建的时候。

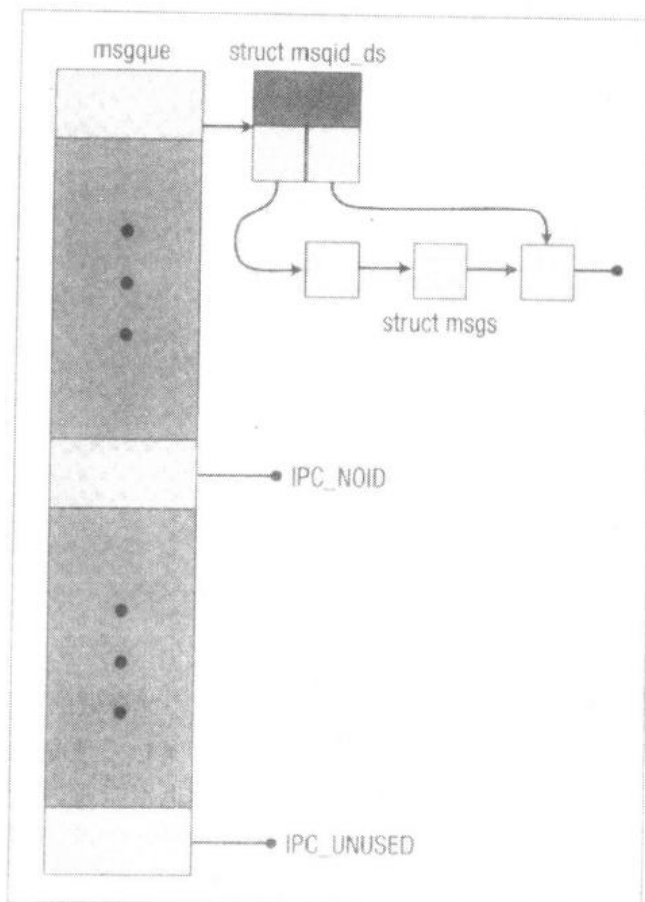


图9-1 消息队列数据结构

### 5. real\_msgsnd

20149: `real_msgsnd`实现`sys_msgsnd`的实质内容，即`msgsnd`系统调用。这里和内核的约定有一些偏差，该约定要在命名系统调用的“内脏函数”（原文为：`guts function`，通常的意思是“内脏”，这里是指本质的东西。——译者注。）时使用一个`do_`前缀。在20338行调用了`real_msgsnd`函数，在那里它处于`lock_kernel/unlock_kernel`函数对之中（那两个函数在第10章中讨论——基本上，每次只能有一个CPU对内核加锁，这与SMP机器有关）。这是一种确保`unlock_kernel`得到执行的最佳方式——否则，`real_msgsnd`复杂的流程控制将因在它退出时确保调用`unlock_kernel`而变得更加复杂。

正如读者已经熟悉的，内核大多使用返回代码变量和`goto`语句来解决这样的问题。虽然它不能很好地适应每种情况，但是`sys_msgsnd`函数的方法更加清晰。例如，当一个函数必须获得多项资源，其中一些只有在以前所有资源请求都成功地被满足时才能提出请求，考虑这样可能引发什么样的后果。简单扩展的解决方法将需要大量函数——就像下边代码段所描述的：

```
void f1(void)
{
 if (acquire_resource1()) {
```

```

 f2();
 release_resource1();
 }
}

void f2(void)
{
 if (acquire_resource2()) {
 f3();
 release_resource2();
 }
}

void f3(void)
{
 if (acquire_resource3()) {
 /* ... Real work here */
 release_resource3();
 }
}

```

很快，这样的代码将变得臃肿不堪，内核不这样做的原因就在于此。

- 20158: 开始一系列条件判断。假如有了第一个测试，本行三项测试中的第二项就是不必要的——任何不能通过第二项测试的消息同样也不能通过第一项测试。虽然以后这种说法可能会不成立，假如MSGMAX的界限增加到足够高的话（事实上，在写作本书时，完全消除MSGMAX界限的工作已在开展之中了）。
- 20164: 消息队列标识号对两段信息进行编码：与之对应的msgque元素的索引在低端7位，一个序列编号（其作用随后就将讨论到）就位于紧挨这7位之前的16位里。现在所需要的全部就是数组下标部分。
- 20166: 如果指定的数组下标处没有消息队列，或者正在创建一个，那么就没有消息可以进入队列。
- 20171: 保存在消息队列中的序列编号必须和那个msgque参数里的编码相匹配。其思想是：在正确的数组下标处有一个消息队列并不代表它就是调用者所需要的消息队列。自从调用者引用一个队列之后，原先处于那个下标的消息队列可能已经被移去了，而且在同一下标处创建了一个新的消息队列。16位序列编号被周期性增加，所以在同一下标处的新队列将有一个和旧队列不同的序列编号（除非正好先创建了65 535个其他的新队列，这是相当不可能的——或者是131 071个其他的新队列，这就更不可能了。本章随后将对其进行解释，实际情况并非这样简单）。不管怎样，只要序列编号不匹配，real\_msgsnd就返回一个EIDRM错误来指示调用者所需要的消息队列已经被移出了。
- 20174: 确保调用者有写消息队列的权限。类似的一个方法将在第11章详细介绍；在这里，可以简单地把它看作是类似于Unix文件权限应用的一个方法。
- 20177: 检查如果提供的消息被写入队列，是否会超过队列所允许的最大容量。接下来一行代码再次检查同一件事，这显然是当代码从2.0系列的内核版本被转换过来时留下的一个编辑疏漏。在两次检查之间，曾经有过一些能够释放队列中的部分空



间的代码。

- 20180: 队列中没有空间。如果在msgflg里的IPC\_NOWAIT位被设置了, 这种情况发生时, 调用者就不会等待, 这样的结果是返回EAGAIN错误。
- 20182: 进程将要进入休眠状态。real\_msgsnd首先检查是否有信号正在等待该进程。如果存在等待消息的话, 就会用进程的休眠被信号中断的方式来处理它(进程可能已经休眠, 就如随后所示的那样)。
- 20184: 假如没有正在等待进程的信号, 进程就进入休眠状态, 直到有信号到达或一条消息移出队列时它才被唤醒。当进程被唤醒之后, 它将再次向队列写入。
- 20190: 为消息队列头(struct msg)和消息体分配足够的空间——正如前面所说, 消息体将紧接在消息头后面存放。消息头的msg\_spot直接指向该头部之后消息体开始的地方。
- 20196: 从用户空间复制消息体。
- 20202: 再次检查消息队列的合法性。msgque的项目可能已经在20184行这个进程休眠时被其他进程修改过了, 因此直到通过检查之前不能认为msg是有效的指针。

即便如此, 这里看起来也有一个潜在的缺陷。如果在当前进程执行到这一步之前, 该消息队列已被删除, 而另一个消息队列被设置在同一个数组下标的地方, 那又将怎样呢? 在UP机器上是不会发生这种情况的, 因为销毁消息队列的函数freeque(20440行), 在销毁它之前将唤醒任何休眠于该队列的进程, 而且在real\_msgsnd完成之前, freeque不会继续进行(本章后面将对freeque进行分析)。然而, 在SMP机器上, 这仍然是一个小小的隐患。

假如发生这种情形, msgque[id]将不是IPC\_UNUSED或IPC\_NOID, 但是msg指向的内存已经被freeque释放了, 因此在20203行将废除无效的指针引用。

- 20209: 填写消息头, 将其入队, 并更新队列自己相应的统计值(比如消息的总共大小)。注意只要有可能就推迟填写消息头的工作, 所以, 假如在分配和当前阶段之间检测到错误时, 这样就不会浪费时间。
- 20226: 唤醒所有等待消息到达这个队列的进程, 然后返回0以示成功。

#### 6. real\_msgrcv

- 20230: 同real\_msgsnd函数一样, real\_msgrcv函数实现msgrcv系统调用。msgtyp参数含义灵活, 这可以从在20248行开始的标题注释中看出。struct msg的msg\_type域在这里发挥作用: 在该函数中它要与msgtyp参数相比较。

另一个与real\_msgsnd相同的地方是real\_msgrcv函数也是从20349行的lock\_kernel/unlock\_kernel函数对内调用的。

- 20239: 从msgid提取msgque下标, 并确保在那个下标所指的空间中有合法的一项。
- 20253: 进入一个迭代循环, 直到进程接收一个消息或放弃而返回, 尽管它的结构明显, 但该循环总是从中间退出, 并且, 它总是通过从函数返回来这样做, 因此, 它可以替代为while(1)循环; 该循环有完全一样的功效, 并且还能稍快一点。
- 20254: 确保进程已指定正确的序列编号, 且允许从该队伍读出。
- 20262: 这个if/else语句对从队列中选择一个消息。第一种情况最简单: 它只需要得到队列中的第一条消息, 使得nmsg或者为NULL或者指向队列的第一个元素。

- 20266: msgtyp为正值, 并且msgflg里的MSG\_EXCEPT位(15862行)被设置。real\_msgrcv函数沿着队列搜索第一个类型和msgtyp不匹配的项。
- 20272: msgtyp为正值, 但是MSG\_EXCEPT位未被设置。real\_msgrcv函数沿着队列搜索第一个类型和msgtyp匹配的项。
- 20279: msgtyp是负值。real\_msgrcv函数用最小的msgtyp编号来搜索消息, 如果最小值比msgtyp的绝对值还要小的话。注意20281行在比较时使用“<”而不是“<=”, 这样队列中消息的选择就不再有利于第一个消息了。这样的结果不仅令人满意——尽量遵循FIFO方式是一个好的策略——而且效率也稍有提高, 因为这种方式减轻了赋值工作。如果比较采用了“<=”, 每个连接(tie)都将意味一次赋值操作。
- 20287: 此时, 如果有消息满足给定的标准, nmsg就指向它。否则, nmsg就是NULL。
- 20288: 即使找到一个合适的消息, 它也有可能不被返回。如果调用程序的缓冲没有足够大的空间来容纳整个消息体, 调用者通常会得到E2BIG错误。然而, 假如msgflg的MSG\_NOERROR位(15860行)被设置, 那么这个错误就不会被公布(我找不出什么理由可以让一个应用程序去设置MSG\_NOERROR标志位, 我也找不出任何一个使用它的应用程序)。
- 20292: 如果msgsz指定了多于消息体的字节数, real\_msgrcv函数就把msgsz减少到消息的实际大小。当程序执行过这里之后, msgsz就是应该被复制到调用者缓冲区的字节数。

虽然此处代码更加传统的写法有时比较慢, 不过平均起来还是要更快一些:

```
if (msgsz > nmsg->msg_ts)
 msgsz = nmsg->msg_ts;
```

- 20294: 把选中的消息从队列中移出。队列是一个单向链表, 不是双向链表, 所以当队列中非第一个的消息被移出时, real\_msgrcv函数必须先要在队列中进行循环以寻找它的前趋队列节点。

通过将队列转换为双向链接, 前趋节点就能在恒定时间里被找到。这个改变将引入空间损耗(需要额外的指针), 时间损耗(用来更新附加的指针), 以及复杂度的提高(需要增加完成这些工作的代码)。尽管如此, 那些代价都是很小的, 而在被移出的消息处于队列中部的情况下, 它们可以显著地提高速度。

不过实际情况中, 大部分应用程序从队列中移出的都是第一个消息。其结果是, 额外花费在管理msg\_prev指针(假定我们这样称呼它们)上的时间通常被完全浪费了。只在从队列中间移出消息时它才会有所补偿, 但应用程序又很少这样做。结果是为了提高特殊情形时的速度而降低了普遍情况下的效率——这几乎总是一个坏主意。甚至于确实要移出队列中间节点的应用程序也不会等很长时间, 因为通常消息队列很短, 典型情况下最多也就是几十个消息而已, 而且平均在循环进行到一半时就能找到选择的消息了。

因此, 只有当消息队列有成百上千条消息, 而且应用程序又要移出队列中间的节点时, 应用程序才会经历一次明显的速度减慢过程。考虑到这种情况的罕见程度, 开发者的决定就是正确的。除此而外, 如果一个应用程序真的陷入这种困境, 而且它的开发者又非常需要这额外一点点速度——那么好吧, 这就是Linux,

他们可以自己修改内核源代码以满足需要。

- 20305: 处理移去队列中唯一节点的情况。
- 20308: 更新消息队列统计值。
- 20313: 唤醒所有等待写入这个消息队列的进程——也就是所有被`real_msgsnd`函数设置为休眠状态的进程。
- 20314: 把消息复制到用户空间并释放队列节点（头部和体部）。
- 20318: 返回正被返回的消息的容量大小——这对可变长度消息来说至关重要，因为应用程序的消息格式可能无法说明消息在哪里结束。
- 20320: 没有符合调用程序标准的消息。接下来发生的操作将取决于调用者：如果调用者设置`msgflg`的`IPC_NOWAIT`位，那么`real_msgrcv`函数可以立刻返回一个失败错误。
- 20323: 否则，调用者宁愿在没有可用的消息时进入休眠状态。如果一个信号正等待调用进程则返回`EINTR`错误；否则，调用者进入休眠状态，直到一个信号到达或者别的进程写队列为止。
- 20329: 永远不会执行到这里，但是编译器并不知道这一点。所以，这儿有一个假`return`语句，只是为了满足`gcc`的要求而已。

#### 7. `sys_msgget`

- 20412: 因为`sys_msgget`的流程控制比`sys_msgsnd`和`sys_msgrcv`的要简单，所以就没有必要把`sys_msgget`的所有实质操作转移到一个独立的辅助函数上。尽管它确实有自己的辅助函数，本章随后还将对它进行分析。
- 20414: 跟踪函数所需的返回值`ret`不必初始化成`-EPERM`。`ret`会在函数的每一分支路径上被赋值，所以这一行的赋值就是多余的。然而，`gcc`优化器的聪明程度足以发现并消除这种无效赋值，因此这一点是没有意义的。
- 20418: 特殊键值`IPC_PRIVATE`（未包括在内——它的值是0）表明调用者需要一个新队列，无论是否有其他具有相同键值的消息队列存在。在这种情况下，通过使用`newque`（20370行）能够立刻创建该队列，随后我们还将详细讨论`newque`。
- 20420: 否则，`key`唯一地标识出了调用者需要使用的消息队列。一般说来，开发人员选择键值时或多或少带有随机性（或者给用户一种办法来选择一个），而且希望它不会与任何运行中的应用程序的键值发生冲突。

这可能听起来耸人听闻，但是临时文件名也存在同样的问题——你只能期望没有其他应用程序选择了同样的命名方式。实际上，很少出现问题——`key_t`是`int`类型的`typedef`，所以在32位机上有超过40亿个可能值，而在64位机上超过了 $9 \times 10^{18}$ 个！这个键值空间容量的巨大有助于降低偶然冲突的机率。而且对于消息队列键值，或者对于文件，即使偶然发生冲突，一个授权方案也能进一步减小问题发生的可能性。

即便如此，难道我们不能做得更好吗？像标准C库函数`tmpnam`一样的函数，可以有效地帮助产生能保证在系统范围内唯一的临时文件名，但是却没有任何类似的办法能够产生一个消息队列键值，而且保证它的唯一性。

如果对这个问题进一步研究的话，这些因素看起来应该是两个不同的问题。

应用程序大体上并不关心临时文件的名称是什么，只要它不是正在使用的文件就可以。但是应用程序一般需要提前知道应把消息发送到哪一个队列中。如果一个应用程序动态地选择它的消息队列键值，那么它有时又莫名其妙地需要把被选择的键值告诉其他应用程序（等价的，它可以发送msgid来代替键值）。而且，假如被涉及的应用程序已经有办法来像那样进行彼此之间的发送消息，那么它们还要消息队列做什么？因此，这可能不是一个值得解决的问题。如果一个应用程序需要一个非专有（nonprivate）队列的唯一键值，但是它对实际键值是什么并不太关心，那么它就可以通过尝试键值1来得到一个（记住0就是IPC\_PRIVATE），并且可以从那儿逐步尝试，直到成功为止——那只需少量的工作，尽管不太可能需要。

无论如何，这一行使用findkey（20354行，后边讨论）来查找拥有给定键值的存在着的一个消息队列。

20421: 如果键值没有被使用，那么sys\_msgget就可以创建它。如果IPC\_CREAT位没有被设置，则将返回ENOENT错误；否则，newque函数（20370行）创建队列。

20425: 键值已被使用。如果调用者把IPC\_CREAT和IPC\_EXCL位都设置了，在这种情况下调用者就希望能够产生一个错误，因此它就得到了一个（这是为了故意与open的O\_CREAT和O\_EXCL位恰好能够并列）。

若不加考虑，很难分辨出原文所写的if判断和下面的等价形式相比哪个更快：

```

) else if (msgflg & (IPC_CREAT |
 IPC_EXCL) ==
 (IPC_CREAT |
 IPC_EXCL)) {

```

两种判断方式都检查那两个标志位是否都被设置了，但是，出于种种原因，你可能会期望任意一个快于对方。然而，结果是gcc为两者产生同样的代码，至少是在优化编译的时候。（如果你对此有兴趣的话，它所选择的方案是把我建议的替代品直接进行翻译的结果——它的转换方案在内核中看起来就好像是同时测试两个标志位的代码。）这是一个相当棒的优化过程，而且也是我过去所没有期望的。

20428: 否则，使用该键值，调用者将接受具有那个键值的存在着队列（这是最普遍的情况）。如果在期望的地方没有消息队列（考虑到findkey的执行，那应该是决不会出现的情况）或者调用者缺少访问它的权限许可，那么将返回一个错误。

20434: 序列编号和msgque下标被编码在返回值里。这将成为调用者要传递给sys\_msgsnd、sys\_msgrcv及sys\_msgctl的msgid参数。

这种编码方案有两个重要特征。更明显的一个特征是如何把序列编号部分和数组下标部分分离开来：因为id是一个索引msgque的数组下标，它只能具有最大到（但不包括）msgque中含有元素的数目，即MSGMNI的值。通过把这个值与序列编号相乘，就可以使低位空出来以保存id了——它很像是一种标准的MSGMNI算法。

这里还需要注意的是返回值永远不会是负值——这一点是非常重要的，因为C库执行时可能会把负的返回值当作是一个错误。因为当前值是128，所以数组下

标占据返回值的低端7位。序列编号是16位，因此只有ret的低23位可以被这次赋值设置成1，而且所有高位应是0。特别地，符号位是0，所以ret是0或正值。

20437: 不管ret被如何计算，它现在都将返回。

## 8. sys\_msgctl

20468: `sys_msgctl`函数无疑是消息队列实现中最大的一个函数。这部分由于它要完成许多不同的功能——类似于`ioctl`函数，它是一个功能联系松散的函数聚合体。（顺便要说明的是，不要因此处的混乱而责备Linux的开发者们；他们只是想要提供与System V那蹩脚的设计相一致的兼容性。）

`msqid`参数指定了一个消息队列，`cmd`指出`sys_msgctl`函数应该对它如何操作。很快读者就会看到，需不需要`buf`取决于`cmd`，而且即使当它被使用时它的含义也将随情况的不同而不同。

20477: 拒绝明显非法的参数。在非正常的，参数无效情况已经是不容置疑时，在调用`lock_kernel`函数之前执行本操作能够挽救不必要的内核锁定（当然，流程控制将不得不相应做出调整——必须跳过`unlock_kernel`函数）。

20481: 在`IPC_INFO`和`MSG_INFO`情况中，调用者需要有关消息队列实现的属性信息。它可能要用这些信息来选择消息容量，比如说，在最大消息容量较大的机器上，调用进程可以提高它自己在每个消息中发送的信息量界限。

所有清晰地消息队列实现中定义那些缺省界限的常数都是通过`struct msginfo`（15888行）类型的对象复制回来的。假如`cmd`是`MSG_INFO`，而不是`IPC_INFO`时，还要包括一些额外信息，读者可以在20495行看到这一点，不过这两种情况在其他方面是相同的。

注意一下调用者的缓存`buf`，它被定义成了指向一种不同类型`struct msqid_ds`的指针。不过没有关系。复制是由`copy_to_user`函数（13735行）完成的，它并不关心它的参数的类型，尽管当被要求向一块不可访问的内存写入时该函数也会产生错误。如果调用者提供了一个指向一块足够大空间的指针，`sys_msgctl`函数将把请求的数据复制到那里；使得类型（或至少是容量）正确是取决于调用程序的。

20505: 如果复制成功，`sys_msgctl`函数返回一个附加的信息段，即`max_msqid`。注意这种情况完全忽略了`msqid`参数。这样做有重要的意义，因为它返回了有关消息队列实现情况的总体信息，而不是某个特别的消息队列的具体信息。不过，这种情况下是否应该拒绝负的`msqid`值仍是一个各人看法不同的问题。不可否认的是，即使没有使用`msqid`时，拒绝一个无效的`msqid`值一定能够简化代码。

20508: `MSG_STAT`请求内核对给定消息队列持续做出的统计性信息——它的当前和最大容量、它的最近的读者和写者的PID，等等。

20512: 如果`msqid`参数不合法，在给定位置处没有队列存在，或者调用者缺少访问该队列的许可，则返回一个错误。因此，队列上的读许可不仅意味着是对入队消息的读许可，而且也是对关于队列本身“元数据（metadata）”的读许可。

顺便提及一下，要注意命令`MSG_STAT`假定`msqid`只是`msgque`下标，并不包括序列编号。

20521: 调用者通过了测试。`sys_msgctl`函数把请求的信息复制到一个临时变量中，然后

再把临时变量复制回调用者的缓存。

- 20533: 返回“完全的”标识符——序列编号现在已经被编码在其中了（在20520行完成）。
- 20535: 还剩下三种情况：IPC\_SET、IPC\_STAT和IPC\_RMID。与读者迄今为止所见的情况有所不同的是，那些情况都在switch语句里被完全处理了，而剩余的这三种在此仅进行部分处理。第一种情况，IPC\_SET只要确保用户提供的缓冲区非空，就将它拷贝到tbuf里以便后面函数的进一步处理（注意拷贝操作之后在20540行对err的赋值是不必要的——因为它使用之前的20550行，err将被再次赋值）。
- 20542: 剩余三种情况中的第二种，IPC\_STAT仅仅执行一次健全性检查——它的真正工作还在后边的函数体中。最后一种情况，IPC\_RMID在这个语句中不工作；它所有的工作都推迟到后边的函数中完成。
- 20548: 这段代码对所有剩余的情况都是共同的，而且大家现在都应该对它比较熟悉了：它从msqid里提取出数组下标，确保在指定的下标处存在着一个有效的消息队列，并验证序列编号的合法性。
- 20559: 处理IPC\_STAT命令的剩余部分。假如用户有从队列中读出的许可，sys\_msgctl函数就把统计信息复制进调用者的缓冲区里。这与先前MSG\_STAT的情形非常类似。这两者之间的唯一不同在于：正如读者所见，MSG\_STAT期望一个“不完全”的msqid，而IPC\_STAT却期望一个“完全”的msqid（就是说包括序列编号）。
- 20572: 复制统计数据到用户空间。如果按照如下方式重写这三行代码，那么运行速度或许稍快一些：

```
err = 0;
if (copy_to_user(buf, &tbuf,
 sizeof(*buf)))
 err = -EFAULT;
```

毕竟，对于写入用户空间来说，成功肯定比失败更为普遍。基于同样的原因，MSG\_STAT情况下（始于20530行），相应的代码被重写成以下形式可能更快：

```
if (copy_to_user(buf, &tbuf,
 sizeof(*buf))) {
 err = -EFAULT;
 goto out;
}
err = id;
```

或者，下面的一个甚至可能更快，因为没有一次多余的赋值操作：

```
if (copy_to_user(buf, &tbuf,
 sizeof(*buf)))
 err = -EFAULT;
else
 err = 0;
/* Or: */
err = copy_to_user(buf, &tbuf,
 sizeof(*buf))
 ? -EFAULT
 : 0;
```

然而，和直觉相反的是，我对所有这三种修改都做了测试，结果却发现内核

的版本执行起来更快。这必然与gcc生成目标代码的方式有关：显然，我的版本中的一条额外跳转要比内核版本的额外赋值所花费的代价高得多（从C源代码来考虑额外的跳转并不直观——你不得不考察gcc的汇编输出代码）。回想前边章节所讨论过的，跳转会带来明显的性能损失，这是因为它们会使得CPU丧失其内在的并行性所带来的好处。CPU的设计者们竭尽全力要避免分支造成的性能损失影响，不过很明显，他们并不总是成功的。

最终，对gcc优化器的进一步改善也许能消除内核版本和我的代码之间的差别。每当两种形式逻辑相同而一个较快时，假如gcc能够发现这种等价并为两者生成同样的代码，那将非常令人愉快。不过这个问题要比看上去难得多。为了生成最快的代码，gcc将需要猜测哪一次赋值最易发生——另一种情况则涉及了分支（对gcc的最近版本所做的工作已为这样的改进打下了基础）。

- 20576: 在IPC\_SET情形里，调用者需要设置消息队列的某些参数：它的最大容量、属主和模式（mode）。
- 20578: 为了操纵消息队列的参数，调用者必须拥有该队列或者拥有CAP\_SYS\_ADMIN权能（14092行）。权能已在第7章中讨论过。
- 20584: 把消息队列中最大字节数的界限提高到正常限制以上，这就类似于提高任何其他资源的硬界限一样，因此，它也需要与之相同的权能，即CAP\_SYS\_RESOURCE（14117行）。资源限制在第7章已经讨论过。
- 20587: 调用者应该被允许执行该操作，所以被选择的参数根据调用者提供的tbuf被设置。
- 20595: IPC\_RMID意味着删除特定的队列——不是队列中的消息，而是队列本身。假如调用者拥有该队列或者有CAP\_SYS\_ADMIN权能，这个队列就可以用freeque函数调用（20440行）来释放。
- 20605: cmd最终不是经过验证的命令中的一条，所以调用程序得到EINVAL错误。在这种情况下，在20548行所做的工作原本是可以避免的。假设我们要试图尽早检测无效的cmd，通过删除switch语句里的default情况，并把下列代码附加到函数第20546行的第一个switch后：

```
case IPC_RMID:
 break; /* Nothing to do yet. */
default:
 err = -EINVAL;
 goto out;
break; /* Not reached. */
```

这样就会改变函数的行为状态。当调用者提供了一个无效cmd和一个无效msqid时，它将得到一个与现在所得的不同的错误——有了这种改变之后，无效的cmd将先于无效的msqid而被检查出来。虽然有关msgctl的文档并没有许诺任何一种行为，但是这样我们就可以自由地改变它。其结果能少许提高这种无效cmd情形下的速度。

然而，不幸的是要注意这种解决方案需要在第一个switch开关处引入一个空的IPC\_RMID case。没有它，函数将错误的把IPC\_RMID也当作一种无效cmd情况而抛弃掉。这个额外的case减缓了cmd合法正常条件下的速度——虽然不很严

重，但情况的确如此。而且，正如你所知道的，用普遍情形的代价损失来换取特殊情形时速度的提高从来就不是一个良好的解决办法。因此还是原来的方式更好。

#### 9. findkey

- 20354: findkey函数为sys\_msgget系统调用（调用在第20420行）定位具有给定键值的消息队列。
- 20359: 开始对msgque里所有可能被占据的单元槽进行循环。max\_msqid跟踪msgque里被占据的最大数组元素；在这里用到了它，并且在很快就要提到的newque和freeque里将对它进行维护。若没有max\_msqid，这个循环将需要在msgque的所有MSGMNI（128）个元素里反复进行，就算是只有前5个在使用也要如此。
- 20360: 如果当前数组元素值是IPC\_NOID，那么就会在那里创建一个消息队列。这个消息队列可能具有正被搜寻的键值，所以findkey函数将等待该队列的创建工作完成（当20385行的kmallocc调用使进程休眠时就会进入这种状态）。
- 20362: 如果该msgque的项目是未被使用的，那么它明显不具有匹配的键值。
- 20364: 若匹配的键值被找到，相应的数组下标就被返回。
- 20367: 如果循环结束仍未找到匹配的键值，就返回-1以示失败。

#### 10. newque

- 20370: newque函数定位一个没有使用的msgque项，并尝试在那里创建一个新的消息队列。
- 20376: 循环msgque以查找未用的一项。如果找到了一项，就用IPC\_NOID来标记它，控制随之跳转到20383行的found标号处。
- 20381: 如果循环结束却没有发现未用的项目，msgque就是满的。newque返回ENOSPC错误，表示表里没有剩余的空间。
- 20384: 分配一个struct msqid\_ds来代表新的队列。
- 20387: 如果分配失败，该msgque项目被设置回IPC\_UNUSED标志。
- 20388: 一旦发现有IPC\_NOID就激活任何已经休眠的findkey。
- 20391: 初始化新队列。
- 20404: 如果这个队列被建立在msgque中原来最高的已用单元槽之后，newque就相应地增加max\_msqid。
- 20406: 在msgque里建立新队列。
- 20408: 唤醒每个可能一直在等待该队列初始化完成的findkey。
- 20409: 返回序列编号和msgque的数组下标（创建一组宏来处理此处的编码和随后的解码不会有什么损害）。奇怪的是，没有在这里增加序列编号——它要由接下来讨论的freeque来完成。如果读者考虑一下，这里的决定是有一定道理的。你并不需要每个队列都有一个唯一的序列编号——你只是想让每次msgque元素被重用时有个不同的序列编号，以便数组下标和序列编号二者的组合（combination）不可能重复而已。数组下标直到建立在该位置的队列被释放后才能重新使用，所以增加序列编号的操作也可以推迟到那个时候。

为了把这个含义说的更明确一些，一个序列编号是可以被两个msgque元素同



时使用的。

#### 11. freeque

- 20440: 我们将以freeque函数来结束这次内核消息队列实现的讨论，它的作用是删除一个队列，并释放相应的msgque项。
- 20449: 如果正在被释放的是最高的被使用项，freeque函数将尽可能减低max\_msqid。循环之后，max\_msqid将再次成为被使用的msgque项的最高下标值，或者在所有元素项都没有使用时变成0。要注意的是如果max\_msqid是0，则msgque要么是空，要么就只有一个元素项。
- 20452: msgque数组的元素被标记成未使用，尽管此时struct msqid\_ds还没有被释放（在msq里，freeque函数仍然有一个指向该struct msqid\_ds的指针）。
- 20454: 假如有某个进程正等待读出或写入这个队列，必须警告它们该队列即将消失。这里的循环唤醒所有那些进程。每个正等着向该队列发送消息的进程将在第20171行知道被改变了的序列编号；每个等待从该队列里读取消息的进程也将在第20254行进行同样的工作。
- 20458: 调用schedule函数（26686行，在第7章讨论过）来赋予被唤醒了了的进程运行的机会。有趣的是，被唤醒了了的进程可能还没有得到CPU使用权——当前进程仍然有最大的优先权。假如这种情况发生，新近被唤醒了了的进程将不会从各自的等待队列中被移出；而freeque又会注意到这一点，并继续重复以试图再次唤醒进程。最终，执行freeque的进程会因耗尽它的时间片而将（CPU）让出给其他进程。在考虑了这一切之后，在调用之前明确设置当前进程的SCHED\_YIELD标志（16202行）可能是更好的方法，这样可以给其他进程更好的使用CPU的机会。
- 20460: 没有被挂起的读者和写者，所以该队列和它的消息可以被安全地释放掉。

## 9.2 信号量

信号量（semaphores）是一种对资源访问进行保护的方式。信号量在通常概念上讲是指一种发送信号的标志（名称由此而来），但是，我认为更好的象征是一把钥匙（key）。不要把它与我们已经讲过的整数类型的键值（key）搞混了——在这个类比中，我所指的意思是你的前门钥匙。

在最简单的情况下，信号量只是悬挂在一扇锁着的门旁吊钩上单独的一把钥匙。为了穿过这道门，你必须把钥匙从吊钩上拿下来；当你出来时再把钥匙重新放回吊钩之上。如果你到达时钥匙不再那里，你就不得不等待它的拥有者把它放回原处——假如你已决定要通过这道门，就必须这样。而作为另一种选择，如果无法立刻得到钥匙，你也可以因没有耐心等待而就此放弃。

以上描述了某资源每次只能由一个实体来使用的情形；在这种只有一把钥匙的情况下，信号量可以被看作是一个二元信号量。对于每次可以被多个实体占用的资源而言，信号量可被看作是计数信号量。这与前边一样，只不过是吊钩上悬挂了更多的钥匙而已。如果资源同时可供四个用户使用（或者假如有四个等价的可用资源，它们基本上是相同的），那么就有四把钥匙。依此类推。

进程使用信号量来协调它们的动作。比如，假设你正在写一个程序，而且想保证每次在

给定的机器上最多只有该程序的一个实例可运行。这方面的好例子是声音文件播放器——可能你不会想让它同时播放多个声音文件，因为其结果将是令人烦恼的。另一个例子是X服务器。当然偶尔也会有充分的理由使得在同一个机器上同时运行多个X服务器，但是对于一个X服务器来说，禁止这样做也是很合理的，至少缺省的做法就是如此。

信号量提供了一种解决这个问题的方案。你的音响播放器、X服务器或是其他任何程序都可以定义一个信号量，检查该信号量是否在使用，若没有使用则继续运行。如果该信号量已被使用，则表明程序的另一个实例在运行之中——你的程序可以等待信号量被释放（像音响播放器）、放弃并退出（像X服务器），或者暂时继续其他工作，稍候再试信号量。顺便说一句，这样一种信号量的用法由于显而易见的原因通常被叫做“相互排斥（mutual exclusion）”；它的通用简称，“互斥（mutex）”将在内核源代码中反复出现。

锁文件是获得与二元信号量同样效果的更为普遍的一种方式，至少在某种情况下如此。锁文件更易使用，而且锁文件的一些实现可工作在网络上；但是信号量则不行。另一方面，锁文件在超出二元的情况时就不容易使用推广了。锁文件超出了本书的范围。

信号量和消息队列二者的代码是如此相似，以至于没有必要再讨论sem\_init（20695行）、findkey（20706行）、sys\_semget（20770行）、newary（20722行）及freeary（20978行）了，因为它们几乎同所对应的消息队列是一样的。

#### 1. struct sem

16983: struct sem结构体代表一个单独的信号量。它有两个成员：

- semval——如果是0或为正值，semval + 1就是仍然挂在这个信号量吊钩上的钥匙数目。若为负值，它的绝对值就比正等待访问它的进程数目大一。缺省的信号量是二元的，但是它们也可以通过使用sys\_semctl（21013行）变为计数型的；信号量的最大值是SEMVMX（在16971行定义为32 767）。
- sempid——存储最后一个操作该信号量的进程的PID。

#### 2. struct semid\_ds

16927: struct semid\_ds与struct msqid\_ds相对应：它跟踪所有关于单独一个信号量及在它上面所执行的一系列操作的信息。我们所感兴趣的、有别于struct msqid\_ds中的成员如下所述：

- sem\_base——指向一个struct sem数组，换句话说，指向一个信号量数组。如同单独一个struct msqid\_ds可以包含多个消息一样，一个struct semid\_ds也可以包含多个信号量——该数组中信号量总和一般被称为“信号量集合”。然而与消息队列不同的是，被一个struct semid\_ds所跟踪的信号量的数目并不在它的生存期里变化。数组的容量大小是固定的。

这些数组中一个最大的长度是SEMMSL，它在第16968行被定义为32。数组的实际长度记录在struct semid\_ds的sem\_nsems成员中。

- sem\_pending——跟踪挂起的信号量操作组的一个队列。信号量操作一有可能就立刻完成，正如读者所预期的那样，所以，只有当操作必须等待时，这个队列才会增加节点。此成员与struct msqid\_ds的rwait和wwait成员大致等价。
- sem\_pending\_last——跟踪上述同一队列的队尾。它并不直接指向最后一个节点——它指向一个指向最后节点的指针，这将有利于稍微提高后面代码的速度（尽

管这给理解增加了难度)。(需要顺便提一下的是,我不知道为什么同样的思想没有被应用于消息队列。)

- `sem_undo`——当各个进程退出时所应该执行的操作的一个队列。这将在后续章节中讨论。

### 3. struct sem\_queue

16989: `struct sem_queue`结构体是单个`struct semid_ds`之上休眠着的操作队列中的一个节点。它有如下成员:

- `next`和`prev`——队列中的下一个和前一个节点。正如`sem_pending_last`一样, `prev`是指向一个指向前面节点的指针的指针。读者将在本章后面章节中看到系统要这样做的原因。`prev`永远不会变成`NULL`; 在退化的情况里, 即当队列为空时, `prev`指向`next`。
- `sleeper`——当某进程必须等待完成一个信号量操作时使用的等待队列。等待队列在第2章中介绍过。
- `undo`——一个由`sops`所暗示的撤销操作的操作数组, 用另一种方式表示它就是`sops`的反转。
- `pid`——尝试完成这个队列节点操作的进程的PID。
- `status`——记录一个休眠进程被唤醒的过程。
- `sma`——向后指向这个结构体`struct`所存在的`sem_pending`队列的`struct semid_ds`。
- `sops`——指向这个队列节点所代表的一个或多个操作的一个数组; 它永远不为`NULL`。这个队列节点所描述的工作目的是执行`sops`里所有的操作。
- `nsops`——`sops`数组的长度。
- `alter`——说明操作是否会影响信号量集合里的任何一个信号量。这个问题的回答看起来总是肯定的, 但要记住等待信号量变成0 (即成为可用) 并不会影响信号量本身。

### 4. struct sembuf

16939: `struct sembuf`结构体表示在信号量上执行的单个操作。它有如下成员:

- `sem_num`——是`struct semid_ds`的`sem_base`数组的数组下标, 该数组由这种操作适用的信号量构成。因为`struct sembuf`是`struct sem_queue`的一部分, 而且`struct sem_queue`知道它与哪一个`struct semid_ds`相关联, 这样就不会不知道该操作应使用哪一个`struct semid_ds`的信号量数组了。在其他情况下, 一个`struct sembuf`数组与一个索引`semary`的下标组成一对, 这也蕴含了一个信号量数组。
- `sem_op`——要执行的信号量操作。通常, 它的值是-1、0或1, -1表示获得(`procure`)信号量(从吊钩上取走钥匙), 1表示交出(`vacate`)信号量(把钥匙重新放回到吊钩上), 而0表示等待该信号量变成0。除了这些值以外的值也是有用的, 不过它们只是被翻译为获得或交出更多的信号量值而已——也就是说, 取走或放回吊钩上更多的钥匙(这段文字里的“获得”和“交出”可能看起来有点儿怪——无须担心; 这只是普通的信号量术语)。
- `sem_flg`——可以修改操作执行方法的一个或多个标志位(在这个`short`里的每一个位)。

这些数据结构之间的关系如图9-2所示。

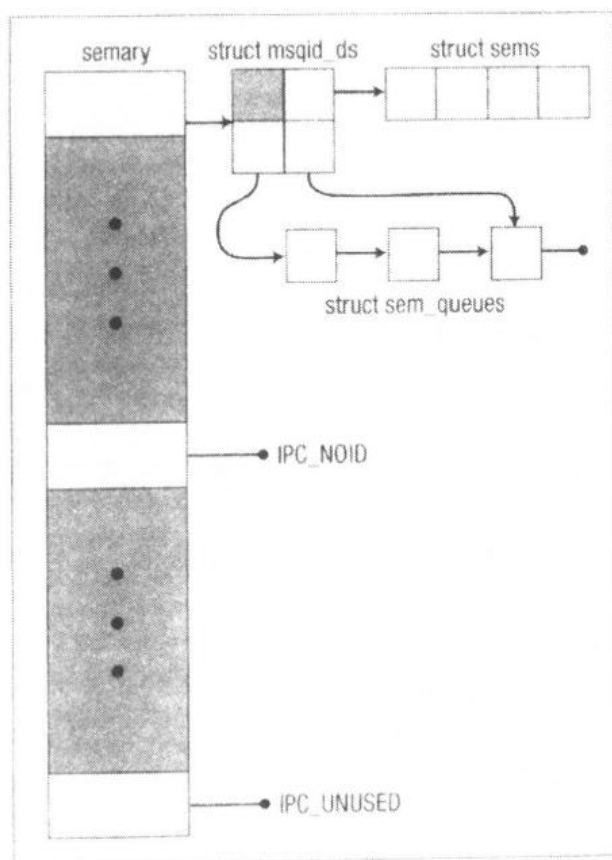


图9-2 信号量数据结构

## 5. struct sem\_undo

17014: struct sem\_undo 含有足够的撤销单个信号量的操作的信息。当一个进程执行信号量操作的同时设置了 SEM\_UNDO 标志位时，就创建一个用来撤销该操作的 struct sem\_undo。进程的 struct sem\_undo 列表所包含的所有撤销操作在该进程退出时都会执行。熟悉设计模式的读者可能发现这是命令模式的一个实例。

这个特性保证不管进程如何退出，都将自动为它执行相应的清理工作——这样一来，就不会意外地让其他进程空等一个永远也不会被释放的信号量了（除非进程获得信号量后陷入死循环之中，但是避免这个问题不是内核的工作——在这种情况下，目的是提供给进程正确的工作方法，而不是对其进行人工干预）。

struct sem\_undo 有如下成员：

- proc\_next——指向固有进程 struct sem\_undo 列表里的下一个 struct sem\_undo。
- id\_next——指向与信号量集合相关联的 struct sem\_undo 列表里的下一个 struct sem\_undo。你所看到的是正确的，同一个 struct sem\_undo 确实是同时存在于两个不同的列表之中。读者将在本章的后面看到这两者的作用。
- semid——标识出这个 struct sem\_undo 所归属的 semarray 项。
- semadj——一个调整器的数组，这些调节将用在和这个 struct sem\_undo 相关联的信号量集合中的每个信号量上。这种结构所不知道的信号量在数组中有一个 0

——并没有进行调整。

## 6. sys\_semop

- 21244: `sys_semop`函数实现了`semop`系统调用。消息队列代码中没有直接同`sys_semop`函数对等的函数——它是`sys_msgsnd`、`sys_msgrcv`，或者同时是两者的对应函数，这取决于你如何看待它。在任意一种情况下，它的工作都是在一个或多个信号量上完成一种或多种操作。它将自动尝试完成所有操作（即无需中断）。假如无法全部完成，它将不会执行其中的任何一项操作。
- 21254: 同消息队列函数非常类似，这像是在比绝对必要的时机稍微提前一些的时候就锁住内核。加锁也应该可以被推迟到第21265行左右再执行。
- 21255: 参数的健全性检查。特别注意`nsops`受到SEMOPM的限制，它是可以被立刻尝试的信号量操作的最大数目。在第16970行它被定义为32。
- 21261: 把请求的操作描述从用户空间复制到一个临时缓冲区，即`sops`中。
- 21265: 确保在指定的数组位置存在一项。正如读者所见，同消息队列代码的`msgque`对等的是`semary`（20688行）。还要注意的是数组下标和序列编号以与消息队列代码相同的方式被打包进了`semid`参数。当然，这里应用的常量稍有不同——SEMNI在第16967行定义为128（而巧合的是，MSGMNI也是一样的值）。
- 21272: 开始一个遍历所有特定操作的循环。首先检查在操作中给出的信号量数目是否超出范围，如果是的话就放弃它。但是令人奇怪的是，这里返回的失败信息是EFBIG错误（意思是“文件太大”）而不是EINVAL错误（“非法参数”）。尽管这也是符合文档规范的。
- 21275: 记录设置了SEM\_UNDO标志位的操作的数目。`undos`只是一个标志——重要的是它是否为0——因此，当条件满足时给它赋值1（或任何非零值）将产生同样的效果。不过，内核的版本更快一点。而且因为循环重复的循环次数最多是SEMOPM次，`undos`就不可能被增加多次以至于回到原点再次变为0。
- 21277: 接下来的几个测试更新两个局部标志：`decrease`和`alter`。它们分别用来跟踪集合里的任何操作是否在减少某个信号量的值以及是否在修改一个信号量的值。直到循环结束之后，`alter`才会在第21282行被计算出来——在循环里，它只是跟踪是否有操作增加信号量的值；这个结果与后边`decrease`里的信息结合起来，最终决定是否发生了改变。
- 要注意这里的代码没有检查组合在一起的操作是否将彼此抵消——可能一个操作把某个信号量减1，而另一个操作又会把它加1。如果这仅有一个操作，那么从某种意义上讲，`decrease`和`alter`标志的值将是很容易引起误解的。内核可以尝试着优化这种情况（并得到实现同样内容的更加精致的版本），不过，相比较于所花费的时间和精力，这可能并不值得：一个愚蠢到执行这样一种奇怪得空操作的应用程序就应该这么慢，而一个聪明的应用程序则不应该这么愚蠢。
- 21285: 确保进程具有在信号量上执行特定操作的许可。如果`alter`为真，那么修改信号量的进程就需要写许可；否则，它只是在等待一个或多个信号量的值变为0，这样该进程就只需要读许可。
- 21291: 包括某些撤销操作的一组操作。如果当前进程已经有了在退出时要在该信号量

上执行的一组撤销操作，那么新的撤销操作的数据就应该合并到那一组中去。这个循环查找存在的撤销操作集合，假如有，就使un指向它，若没有，则un为NULL。

- 21295: 进程还没有这个信号量集合的一个取消集 (undo set)，所以需要为它分配一个新的。在读者所见的消息队列代码中的一个实践之后，为撤销调节 (semadj数组) 分配的空间将被安排在紧靠struct sem\_undo本身之后，并作为同一分配的一部分。接着就填入struct sem\_undo。
- 21311: 在提供的操作集合里没有撤销操作，所以un被设置为NULL。
- 21313: 调用try\_atomic\_semop (20838行，后边讨论) 来尝试在单个的槽内执行所有操作。如果有任何引起变化的操作，un即为非空；若失败，就需要利用它在函数返回之前取消任何已经完成的部分操作。
- 21315: try\_atomic\_semop返回0表示成功，负值表示错误。无论何种情况，控制流程都向前跳转到第21359行。
- 21321: 否则，try\_atomic\_semop返回一个正值。这表示此刻无法执行所有操作，但是该进程希望等待，且稍后再试。一个局部struct sem\_queue将首先被填写。
- 21328: 代表修改信号量操作的节点被放在队列的末尾；代表等待信号量值归0的操作的节点位于队列的前边。在本章后边探究update\_queue函数时 (20900行)，读者将对这种做法的原因有所了解。

注意在挂起操作的队列中放置了一个局部变量——这很不寻常；这样的数据结构通常具有以堆形式分配的节点。在这种情况下这样做是安全的，因为节点在函数返回之前将被从队列中移出；而上下文转换部分将负责剩下的工作。或者进程也可先退出，此时由sem\_exit (21379行) 来负责进行收尾工作。

- 21333: 开始一个反复尝试执行这些操作的循环，仅当所有要求的操作都成功完成或者发生一个错误时该循环才会退出。
- 21336: 一直休眠到被一个信号中断或有某断点被再次尝试为止。
- 21342: 如果进程由于此刻具有成功的机会而被update\_queue唤醒，则它重新尝试进行该操作。
- 21358: 把这个进程从等待修改信号量集合的进程队列中移出。
- 21360: 假如这个操作的集合改变了队列，那么某个其他进程所等待的条件可能就已经具备。sys\_semop调用update\_queue来寻找并唤醒这样的进程。

#### 7. sys\_semctl

- 21013: 实现semctl系统调用的sys\_semctl函数与sys\_msgctl有许多共同之处。相应的，这里的讨论只涉及那些感兴趣的点，比如在sys\_msgctl里没有对应部分的sys\_semctl命令。
- 21093: GETVAL、GETPID、GETNCNT、GETZCNT及SETVAL命令对单个信号量，而不是信号量集合进行操作，所以在这些情形里提供的semnum参数必须首先进行范围检查。若semnum在范围之内，curr就指向相应的信号量。
- 21115: 几乎是同样的命令集——GETVAL、GETPID、GETNCNT及GETZCNT——涉及了对信号量的一段信息进行的阅读和计算。这里就完成那些工作。注意sempid成

- 员的高位在第21116行被屏蔽掉了——通过后面的讨论你将知道这样做的原因。
- 21121: GETALL命令请求这个信号量集合里所有信号量的值。和许多其他命令一样, 此命令的工作并非在一处全部完成; 稍后读者将见到其余的命令。
- 21126: SETVAL命令把信号量的值设置成给定的值——当然, 是在规定的限制内。同样, 此时只完成部分工作——主要是范围检查。
- 21142: SETALL是SETVAL的泛化; 它设置集合中所有的信号量值。同SETVAL类似, 在此只完成诸如范围检查一类的准备工作。
- 21173: GETALL的剩余部分由此开始。
- 21175: 确保进程有读取信号量值的许可。这里的许可检查与第21112行的相重复。
- 21177: 把信号量值复制到局部数组sem\_io里, 然后再从那里将它们复制到用户空间。
- 21183: SETVAL的剩余工作由此开始。
- 21187: 因为信号量取得新值, 所以, 任何有记录的为semnum信号量所进行的取消调节操作都将变为无效。这个循环通过把它们设置成0以使它们失去作用。
- 21189: 把信号量的值设置成调用者提供的值, 并调用update\_queue (20900行) 来唤醒那些等待该条件成立的进程。
- 21220: SETALL的主要部分由此开始。
- 21224: 所有信号量的值都被设置成了调用者提供的值。
- 21226: 与集合内各个信号量相关的所有取消调节操作都被设置成0。当信号量被设置为它已经拥有的值时, 这并没有什么特别的地方——它也不应该有什么特别之处。如果调用程序需要为除了一个信号量之外的所有信号量都赋予新值, 那么就不能通过设置那个信号量为原值的方法来欺骗它。取而代之的做法是, 必须要对集合中除不应改变其值以外的所有信号量施用SETVAL命令。

#### 8. sem\_exit

- 21379: sem\_exit函数在消息队列代码里没有对应函数。它实现进程在退出时所要求自动执行的撤销操作。所以, 它在进程退出时调用 (23285行)。
- 21389: 如果进程的semsleeping成员非空, 那么以下二者必有一成立: 要么进程正在某个sem\_queue队列上处于休眠状态, 要么它已经从该队列被移出但semsleeping还未更新。假如是前者, 进程将被从休眠队列里移出。
- 21395: 开始遍历当前进程的struct sem\_undo列表。轮流对每个项条进行分析, 然后在循环的更新部分释放它们。
- 21397: 如果对应于这个撤销结构体的信号量已被释放, 就继续循环。struct semundo的semid域可以被freeary设置成-1, 本章随后将对其进行介绍。
- 21399: 同样, 如果相应的semque项不再有效, 则继续循环。
- 21406: 与从消息队列中间移出一条消息的情形相当类似, 这个循环遍历sma的struct semundo列表以找寻将被移出的前一个节点。当找到时, sem\_exit向前跳转到第21413行的found标号处。
- 21411: 如果在sma的列表里没有找到撤销结构体, 那么就发生了错误。sem\_exit显示一条警告消息, 并停止外层循环。这种反应看来有点过激, 因为, 可能会有更多的撤销结构体能够依照这种处理方式进行释放。不应该因一个烂苹果就糟踏整整一

桶苹果。尽管这几乎是“不可能发生”的情形，仅当内核逻辑错误才会导致其发生。我的推测是这样的，若检测到这样一个错误的话，剩下的数据就不再可信了。

21414: 在sma的列表里找到了撤销结构体，unp就指向一个指向其前驱的指针。接着把un从队列里移出。

21417: 执行这个撤销结构体里对所有信号量的调节。

21427: 像往常一样，调用update\_queue以免被这个函数所执行的操作满足了唤醒某个休眠进程的条件。

21429: 所有的struct sem\_undo都已经处理过了——或者在21412行就检测到了错误，并结束了循环。不管哪一种结果，当前进程的队列被设置成为NULL，然后函数返回。

### 9. append\_to\_queue

20805: 把q附加在sma的sem\_pending队列之后。这里的实现很紧凑；通常类似如下的形式：

```
q->prev = sma->last;
if (sma->sem_pending) /* Not empty? */
 sma->sem_pending_last->next = q;
else
 sma->sem_pending = q;
sma->sem_pending_last = q;
q->next = NULL;
```

真正的优点在于内核的实现方式避免了潜在的代价昂贵的分支。通过使sem\_pending\_last成为指向一个指向队列节点的指针的指针，而不仅仅是一个指向队列节点的指针，可能会部分提高执行的效率。

### 10. prepend\_to\_queue

20812: 把q附加在sma的sem\_pending队列之前。由于sem\_pending不是一个指针的指针，这种实现就同前面考虑过的简单实现一样具有相同的形式。

### 11. remove\_from\_queue

20823: 这是struct sem\_queue队列上的最后一个原语操作，它把一个节点从队列中移出。

20826: 通过修改前一队列节点的next指针，部分解除q与队列的链接。

20828: 如果有下一个节点，还要更新下一个节点的prev指针；或者假如这已经是队列的最后一个节点，就使用sma->sem\_pending\_last。要注意的是，没有非常清晰的代码被用于移出队列中唯一的节点——假设你还没有发现原因的话，这就值得你花些时间研究一下，为什么这种情形下代码也可工作。

20831: 把已移出节点的prev指针设置成NULL，以便第21350和21390行代码能有效地发现该节点是否仍在队列之内。

### 12. try\_atomic\_semop

20838: 该函数上方的标题注释说明它被用于测试给定的操作集能否被全部执行。该注释没有说明这些操作是否能够被全部被执行，通常情况下它们是能够被执行的。

20846: 开始循环所有通过检查的操作，并依次对其进行尝试执行。

20850: sem\_op为0表示调用程序希望等待curr->semval变为0。因此，如果curr->semval不是0，调用程序不得不阻塞，这意味着操作无法自动被执行（由于这个进程被阻



塞时要完成其他工作)。

- 20853: 调用程序的PID被暂时保存在curr->sempid的低16位里; 从前的PID现在被移进高16位。
- 20854: curr->semval依照sem\_op所要求的进行调整——还是临时性的。虽然该操作结果的范围在随后的代码段中进行了核查, 但是不管是在这里, 还是在其调用者中, sem\_op都未进行范围检查。由于sem\_op的值过大或者过小都将造成semval回绕(wrap around), 这样将导致意想不到的结果。
- 20855: 如果这条操作的SEM\_UNDO标志位被设置了, 就表示在进程退出时该操作应当被自动取消, 相应的撤销结构体会被更新。要注意这里假定un是非空的——确保这一点是调用程序的责任。
- 20858: 对新的semval进行范围检查。
- 20864: 循环即将完成, 所有操作都将成功完成。如果调用程序只想知道操作能否成功, 但此刻并不想执行它们, 这些操作可以就马上被取消。否则, 操作已经被执行了, 所以try\_atomic\_semop就继续执行下去, 并返回成功。
- 20874: 当一个操作把semval增加得过大时, 就跳到out\_of\_range标号处。函数安排返回ERANGE错误, 并向前跳转到撤销代码。
- 20878: 当进程因为它必须等待信号量归0或者操作不能立刻获得信号量而不得不等待信号量时, 程序将跳至would\_block标号处。如果这种情形之下调用程序不愿等待, 就返回EAGAIN错误。否则, 函数返回1表示调用者将需要休眠。
- 20884: 在undo标号之后的代码取消所有从第20846行开始的for循环里所做的工作。
- 20888: 这一行代码显而易见的部分是用来把在第20853行暂存的值保存在curr->sempid的低16位中。其隐含的部分是没有必要被设置成0的高16位(在此假定是32位平台): C标准有意给予编译器用0或用符号位拷贝来填充空余位的自由。在实际实现中, 低级机器指令怎样能最快的工作, 编译器就如何工作, 结果有时是这些操作的一种, 而有时则是另一种(C语言标准为什么不拘限于任何一种实现的原因正在于此)。这样的结果是, 高位可以是全0, 也可以是全1, 这也正是在第21116行里只有低16位被屏蔽的原因。

### 13. update\_queue

- 20900: update\_queue函数在信号量的值发生改变时被调用。它完成那些此刻可以成功(或者将要失败)的挂起操作, 并把它们从挂起队列中移出。
- 20907: 如果这个节点的状态标志已经被前一次update\_queue调用增加过了, 那么与该节点相关的进程就还没有机会把它自己从队列中移出。为了给其他进程机会来执行它的挂起操作并从队列脱离, 函数返回。
- 20910: 检查是否此刻能够完成当前一组挂起操作。q->alter是最后一个被通过的参数, 所以即将成功的变异操作就会自动被取消。这是因为进程将要继续亲自尝试这些操作, 而它们是不应该被执行两次的。
- 20914: 假设错误或者成功状态已经能够被判定(对立于需要继续等待), 这个节点就被从队列中移出, 并且与它关联的进程也会被唤醒。否则, 节点留在队列中, 以便在将来某处被再次尝试。

- 20917: 如果该操作集包括一些变异的操作, 标志就被提高以便进程知道唤醒它是由于现在能够成功了; 进程将尝试那些操作并把自己从队列中移出。前面讨论过, 第21342行要对这个标志进行检查。
- 20920: 函数现在返回, 以便多个变异进程不会同时尝试进行它们那些可能并不互相兼容的改变。回忆一下, 非变异的操作位于队列头部, 而变异的操作是位于末尾的。其结果是, 所有的非变异进程(它们不会彼此干扰)被首先唤醒, 然后最多唤醒一个变异进程。
- 20922: 否则, 将产生一个错误。该错误代码被保存在q->status里, 接着队列节点被移出。

#### 14. count\_semncnt

- 20938: count\_semncnt函数从21117行被调用来实现sys\_semctl内的GETNCNT命令。它的工作是记录因等待获得信号量而阻塞的任务数目。
- 20949: 这个循环用于执行在sma的挂起队列中等待着的每个任务中的每个挂起操作。每当找到一个满足的操作时, 它就递增semncnt——该操作试图获得特定的以及没有设置IPC\_NOWAIT标志的信号量。

#### 15. count\_semzcnt

- 20957: count\_semzcnt函数在21119行被调用来实现sys\_semctl内的GETZCNT命令。它除了要对等待信号量归0的任务(也就是等待信号量变得可用的任务)进行计数之外, 它和count\_semncnt函数几乎一样。因此, 唯一的区别就在第20970行, 在那里它使用等于0而不是小于0来进行测试。

## 9.3 共享内存

“共享内存”顾名思义就是: 一块预留出的内存区域, 而且一组进程均可对其进行访问。因为它涉及IPC和内存管理两方面的内容, 这部分讨论将融合本章及第8章以前的材料进行分析。

截至目前为止, 共享内存是本章要介绍的三种IPC机制里最快的一种, 而且也是最简单的一种——对于进程来说, 获得共享内存后, 它和任何其他内存看起来都是一样的。由一个进程对共享内存所做出的改变对所有其他进程都是立即可见的——它们只需通过一个指向共享内存空间的指针来读取, 然后就轻松地获得了结果。然而, System V共享内存没有确保互斥的内置方案: 一个进程可以向共享内存中的给定地址写入而同时另一个进程从相同的地址读出, 这会导致读者看到的将是不一致的数据。这个问题在SMP机器上非常明显, 但是它也会发生在UP机器之上——例如, 假设正当把某个较大的结构写入共享内存空间时, 写者被转换出了上下文环境, 而读者又在写者完成操作之前读取了共享内存的时候。

这样的结果是, 使用共享内存的进程必须努力确保读操作与写操作的严格分离(考虑一下, 写操作和写操作之间也是如此)。锁和原子操作的相关概念将在下一章详细论述。但是读者已经了解了保证互斥访问共享内存区域的一种方法: 使用信号量。这种思想是一旦获得信号量就全速访问内存区域, 工作一完成后就立即释放该信号量。

共享内存存在一些用到消息队列的情况下也具有同样的帮助作用——一个调度进程可以把工作请求写入共享内存区域的一部分, 同时工作者进程可以把结果写入另一部分。这就意味着

应用程序要预先为请求和结果空间限制界限，但这样的内存分配和结果写入还是要比使用消息队列快。

对于每个进程来说，共享内存区域不必看起来具有相同的地址。如果进程A和进程B都在使用同一块共享内存区域，那么A可能看到它在一个地址，而B则可能会看它在另一个地址。当然，共享内存区域中给定的一个页面将最多被映射为一个物理页面。前一章介绍过的虚拟内存机制可以为每个进程进行不同的逻辑地址转换。

在内核代码中，共享内存区域被称为“段”，这正是有时会被误用于VMA的一个术语。为了预先防止任何混淆，这是一个该术语的非正式用法；它与第8章里讨论过的硬件增强的(MMU)段是不同的。为了避免这种说法所可能引起的迷惑，我将继续使用“区域”这个术语。

共享内存代码从设计到实现都与消息队列及信号量的代码有一些相似之处。因此，没有必要再介绍shm\_init (21482行)和findkey (21493行)函数。因此，剩下的一些函数和数据结构的讨论也会相应缩短。

#### 1. struct shmid\_ds

17042: 多少有点打破了已经建立的模式，struct shmid\_ds不是内核用来跟踪共享内存区域的数据结构。取而代之的是，struct shmid\_ds包含这种信息的绝大部分，而剩下的信息则位于下面要介绍的struct shmid\_kernel中。以下是struct shmid\_ds的那些与其对应对象所不同的成员：

- shm\_segsz——共享内存区域的大小，用字节（不是页面）度量。
- shm\_nattch——用典型的术语，是指“附属 (attached)”到这块区域的任务数目，换句话说，就是使用该共享内存区域的任务数。这个成员是一个参考计数。
- shm\_unused、shm\_unused2和shm\_unused3——从它们的名字就可推断，这些成员不再用于实现之中；它们的唯一角色看来是为了保持该结构体大小的向后兼容性。

#### 2. struct shmid\_kernel

17056: struct shmid\_kernel用于分离“专有 (private)”的共享内存相关信息和“公有 (public)”的信息。struct shmid\_ds里那些对用户应用程序可见的部分还保留在struct shmid\_ds之内，而关系到内核的专有信息则位于struct shmid\_kernel之内。用户应用程序需要通过struct shmid\_ds来进行shmctl系统调用，所以它的定义必须对它们是可见的，但是内核专有实现的细节就不应该出现在struct的定义之中。否则，改变内核的执行可能会中断应用程序。struct shmid\_kernel具有如下成员：

- u——即struct shmid\_ds，也就是数据的公共部分。
- shm\_npages——用页面数表示的共享内存区域的容量。它恰为shm\_segsz成员除以PAGE\_SIZE (10791行)的结果。
- shm\_pages——用于跟踪这块共享内存区域页面分配的一个“页表”——“页表”在这里加了引号，是因为它不是一个同前一章里一样的真正的、硬件支持的页表。不过它完成同样的工作。
- attaches——代表各自进程对这块共享内存区域进行映射的VMA的一个链表。VMA在第8章里已经介绍过。

#### 3. newseg

21511: 是与newque和newary相对应的函数。它分配并初始化一个struct shmid\_kernel，

然后把它安置在shm\_segs数组之中。

21537: 分配“页表”。它和紧随struct shmid\_kernel之后的对这块内存空间进行分配的另一个IPC代码一样，它们都是一个大的分配过程中的一部分。不过，struct shmid\_kernel是由kmalloc分配的（在不可交换的内核内存里），然而“页表”是由vmalloc分配的（在可交换内存里）。

21546: 以把页表填零为起点来初始化所有已分配的元素项。

#### 4. sys\_shmget

21573: 这个函数自然是对应于sys\_msgget和sys\_semget的。唯一新颖的特征是它对进程struct mm\_struct的信号量的获取和释放过程。这是一个内核信号量，它与System V信号量并不相同——内核信号量将在第10章介绍。

#### 5. killseg

21610: 这个函数对应于freeque和freeary。它的代码也同那些函数的非常相似，但是有几个特征值得注意。

21616: 如果用一个未被占用的shm\_segs元素的下标调用killseg函数，它就显示一条警告，并立刻返回。它的两个对应函数中都不存在相似的代码。

21629: 如果元素项的shm\_pages成员是NULL，那么就在某处有一个逻辑错误。struct shmid\_kernel要么是没有完全构建好，要么就是已经销毁但还没有被从数组中删除，又或者就是某个类似的看起来“不可能发生”的情况发生了。

21635: 释放为页表分配的页面。

21638: 如果页表没有映射这个页面，在释放这一项时就无需执行什么操作。

21640: 如果页面在物理内存里，则它将被释放回可用页面的缓冲池里，同时递减驻留页面的数目。

21643: 否则，页面位于交换空间，它将从那里被释放。

21648: 释放页表本身。

#### 6. sys\_shmctl

21654: 这个函数明显是对应于sys\_msgctl和sys\_semctl的，而且和它们有许多共同点。在此只介绍两个共享内存所特有的命令。

21733: SHM\_UNLOCK命令是SHM\_LOCK的反作用命令，case在第21742行。SHM\_LOCK允许拥有足够权能的进程锁住物理内存里的一整块区域，以防止它被交换出去。而SHM\_UNLOCK则对一块加锁区域进行解锁，使得其中的页面再次可以被用于交换。

在这两个case里的工作看来不甚相似：它只是确定调用者有合适的权能，要被解锁的区域当前是加锁的（或反之亦然），然后设置或者清除适当的模式位。但是这就是所要完成的一切了——其效果会在shm\_swap（22172行）中显现出来。

注意有一个分离的权能用于加锁和解锁共享内存，即CAP\_IPC\_LOCK（14021行）。

#### 7. insert\_attach

21823: 这个短小的函数只是把一个VMA添加到附属于给定struct shmid\_kernel的VMA列表中。注意该VMA是添加到列表头部的——顺序并不重要，而且这样处理最为简

单。否则的话，`attaches`的头和尾都将不得不分别被进行跟踪。

#### 8. `remove_attach`

21833: 这个函数自然是从附属于给定`struct shmctl`的列表中移出一个VMA。关于此函数的奇怪之处是它并不依赖于它的`shp`参数——该参数是一个指针，指向存储在VMA列表第一个VMA里的`shp`的`attaches`列表，它位于第21829行，而且用于更新列表的过程，同样不考虑该VMA是否为列表里的第一项（如果它是，相应的`attaches`也被更新）。

#### 9. `sys_shmat`

21898: 这个函数实现了`shmat`系统调用，调用进程借助它可以同一个共享内存区域建立联系。

21923: 在一些熟悉的准备工作之后，`sys_shmat`开始对共享内存区域应出现在调用进程内存空间中的地址进行计算。首先，它要检查调用者传过来的地址。如果它是`NULL`，而且`SHM_REMAP`标志位也未被设置（参见21959行），那么请求必须被抛弃——`NULL`永远不可被读和写。

21929: 调用者传入`NULL`作为目标地址，这意味着`sys_shmat`应该在该进程的内存空间里选择一个地址。`get_unmapped_area`（33432行）将提供一个候选的地址，顺便需要提及的是，该函数在前一章已经讨论过。如果它返回0（在所有内核支持的平台上都等价于`NULL`），那么就是无法找到足够大的区域。

21932: 如果候选的地址不是恰好在一个页面的边界上，它就会被向上舍入到更高的下一个页面边界，然后用调整过的地址取代原先的地址。`get_unmapped_area`返回在给定地址上或超过它的第一个可用地址，因此，假如上舍入的地址是可用的，它将被采用。

现在解释一下为什么地址要向上舍入而不是向下舍入（那样能够更快和更简单一些）：假如`sys_shmat`进行向下舍入，而所得地址不可用，那么代码将陷入死循环。下一次调用`get_unmapped_area`将从下舍入地址处向上搜索，并返回到原先未经舍入的地址处，而它将再次被向下舍入，发现不合适，又传送给`get_unmapped_area`……

要注意在这里使用的是`SHMLBA`（11777行）而不是`PAGE_SIZE`（10791行）来决定地址的适宜性。不过，正如你所见到的，`SHMLBA`恰好被定义为`PAGE_SIZE`，所以效果是相同的。

如果`SHMLBA`和`PAGE_SIZE`是一样的，那二者为什么要兼有呢？答案在于`SHMLBA`在绝大多数平台上就是`PAGE_SIZE`，但并不是在所有平台上都是如此。在MIPS上——CPU具有4K的`PAGE_SIZE`——Linux把`SHMLBA`定义为非常大的0x40000（256K），其注释说明选择这样大的值是为了遵守基于MIPS机器的SGI应用程序二进制接口（ABI，Application Binary Interface）。然而，MIPS ABI的版本2和3却明确声明了`SHMLBA`的值“在符合标准的实现上是允许有所不同的”，所以不清楚为什么内核开发人员认为256K的值是必要的。或许该值是非常早期的ABI版本所要求的，但是我向回检查ABI一直到1.2版仍没有发现任何这样的要求。还有，在SPARC-64上，`SHMLBA`是`PAGE_SIZE`的两倍；不幸的是，这个区别没

有在代码中进行解释。

- 21936: 否则, 调用者传送一个建议的地址。如果有必要而且是被允许的, 该地址就被向下取整。
- 21945: 确保从被选地址开始的大小为len的内存块在进程的允许内存空间之内 (len已在几行之前计算出来, 21913行)。当调用者提供候选地址时进行检查明显是必要的, 而且粗看起来, 当sys\_shmat用get\_unmapped\_area来选择一个地址时进行检查也是必要的。尽管区域的大小已经被传递给它, get\_unmapped\_area还是要执行一个相似的检查, struct shmid\_ds的shm\_segsz成员不必和len相同——len是PAGE\_SIZE的一个倍数, 而shm\_segsz则可以不是。

不过, 因为所有被get\_unmapped\_area使用的地址都是页对准的, 所以, 传递给它的区域大小是否是页面尺寸的倍数都不会影响它的计算。

- 21951: 正如注释中所说明的, 被选区域必须为进程的栈留出一些空间。这个缓冲区间有四个页面——这个数字并没有什么特别之处, 只要达到进程有足够的栈空间的目的即可。在上一章中曾提到过, 如果某任务耗尽了它的栈, 它将被杀死。综合起来考虑, 让单个系统调用失败可能要比让整个进程被无理地杀死更好一些——进程可以从前者中逐渐恢复, 但是后者却不行。
- 21959: SHM\_REMAP (17075行) 的主要作用在此体现: 如果SHM\_REMAP被设置了, 而且调用者提供的区域已在使用, 那么就没有错误, 这是因为SHM\_REMAP用于允许调用者把一块共享内存区域映射到它自己的内存里——比如是一个全局缓冲区。如果这个标志没有被设置, 被选的地址就一定不能和进程已经拥有的任何内存相互重叠。
- 21971: 如果调用者缺少使用这块内存区域的许可, 系统调用失败。如果SHM\_RDONLY (只读) 标志被提供, 调用者只需要读许可; 否则, 调用者需要读许可和写许可。
- 21991: 填写新的VMA。特别注意它的vm\_ops成员被初始化为指向shm\_vm\_ops (21809行), 就像在第8章里讨论过的一样。
- 22004: 增加这块区域的引用计数, 以便它不会被过早地销毁。
- 22005: 调用shm\_map (21844行) 把共享内存页面映射到进程的内存空间里。如果失败, 它就会返回, 同时递减引用计数, 如果这是第一个和唯一的引用, 那么还需要销毁该区域, 然后释放VMA, 这样整个工作就结束了。

注意即使这是最后一个引用, VMA也不必被释放; 该区域也必须要用SHM\_DEST标志 (17106行) 来进行标记。SHM\_DEST可以在由调用者来设置的标志位之中: 它也可以在后面sys\_shmctl的IPC\_RMID情况里被设置——参见21780行。以这样的方式, 一块共享内存区域可以比它所有的附属进程生存更长时间。与保留一个检查点文件原因一样, 这样的处理方式也是有用的: 你可能会有一一个每晚都要运行几个小时的耗时进程, 要把它的处理结果保存在一个即使该进程当前工作完成之后仍然继续存在的共享内存区域。只要通过附属到剩下的共享内存区域, 它就能够恰好在下一个晚上从停下的地方重新开始 (当然, 由于共享内存区域不同于文件, 在计算机关闭后就会消失, 所以这种方案不适用于不能有丢失危险的工作)。

22014: 添加到附属于这块区域的VMA列表中, 然后更新一些关于每区域统计的数据。

22019: 返回在调用者空间里真正被选择的共享内存区域地址, 然后成功地返回。

#### 10. shm\_open

22028: shm\_open函数像是sys\_shmat (21898行) 的一个简化版本。它把一个给定的VMA附加到一个共享内存区域里。提供的VMA是从一个已经附属于目标区域的VMA复制而来, 所以这个VMA本身已经被正确填写了; shm\_open函数的工作基本上只是要完成附属连结。

正如shm\_open上方的注释所陈述的, 这个函数是从do\_fork (23953行) 里被调用的, 该函数在第7章里已经介绍过。更准确地说, 这个函数是在dup\_mmap (23654行) 里第23692行被调用的。然后, dup\_mmap在copy\_mm (23774行) 里的第23801行被调用; 而copy\_mm又是在do\_fork里的第24051行被调用。

22033: 从VMA的vm\_pte成员里抽取shm\_segs下标, 然后确保该处有一合法项。注意下标无需进行范围检查, 这是因为和SHM\_ID\_MASK所进行的按位“与”操作 (11757行) 已强迫它合乎范围了。

22040: 添加VMA到区域里并更新区域的统计数字。

#### 11. shm\_close

22050: shm\_close明显是shm\_open的反作用函数, 它把一个VMA从它附属的共享内存区域里分离出来。尽管在其他地方内核也可以调用VMA的close操作, 但33821行看来是能结束调用shm\_close的唯一之处。这是exit\_mmap (33802行) 的一部分, 而它又是被mmap (23764行) 调用, mmap又被\_\_exit\_mm (23174行) 调用, 而\_\_exit\_mm又被do\_exit (23267行) 所调用, do\_exit函数在第7章就已经讨论过。要注意还有其他到达shm\_close的路径, 我们很快就会对其中之一进行介绍。

22056: 从VMA的vm\_pte成员里抽取shm\_segs下标, 然后把该VMA分离出区域。出于和shm\_open同样的原因, 下标不用进行范围检查。还要注意的, shm\_close不检查是否在指示的下标处存在一个合法的shm\_segs项。读者已经看到, remove\_attach不依赖于它的shp参数, 所以它对此并不关心。然而shm\_close剩下部分将假定其他共享内存代码被正确地使用和执行, 所以这种“不可能发生”的情形真的是不可能发生的。

22058: 从共享内存区域分离VMA, 然后更新区域的统计数字。

22061: 减少该区域的引用计数, 如果可能的话还需要将其释放。

#### 12. sys\_shmdt

22068: 与sys\_shmat相反, sys\_shmdt函数把一个进程从一块共享内存区域里分离出去。

22074: 开始对所有代表进程内存的VMA进行循环处理。

22076: 如果VMA代表一块共享内存区域 (这可以通过检查它的vm\_ops成员进行精巧的测试), 而且该VMA始于目标地址, 就应解除该VMA的映射。

22079: do\_munmap (33689行) 调用unmap\_fixup (33578行), 它又间接在33592行调用shm\_close。do\_munmap和unmap\_fixup都在第8章里介绍过。

## 第10章 对称多处理

在全书的讨论过程中，我一直在忽略对称多处理（SMP）代码，而把注意力集中在只涉及一个处理器的相对简单的情况。现在我们从一个新的角度来看：当内核必须支持多于一个CPU的机器时将发生什么？

在一般情况下，使用多于一个CPU来完成工作被称为“并行处理”，它可以被想象成是一段频谱范围，分布式计算在其中一端，而对称多处理在另一端。通常，当你沿着该频谱从分布式计算向SMP移动时，系统将变得更加紧密耦合——在CPU之间共享更多的资源——而且更加均匀。在一个典型的分布式系统中，每个CPU通常至少拥有它自己的高速缓存和RAM。每个CPU还往往拥有自己的磁盘、图形子系统、声卡、监视器等等。

在极端的情形下，分布式系统经常不外乎就是一组普通的计算机，虽然它们可能具有完全不同的体系结构，但是都共同工作在某个网络之上——它们甚至不需要在同一个LAN里。读者可能知道的一些有趣的分布式系统包括：Beowulf，它是对相当传统而又极其强大的分布式系统的一个通用术语称谓；SETI@home，它通过利用上百万台计算机来协助搜寻地外生命的证据，以及distributed.net，它是类似想法的另一个实现，主要关注于地球上产生的密码的破解。

SMP是并行处理的一个特殊情况，系统里所有CPU都是相同的。举例来说，SMP就是你共同支配两块80486或两块Pentium（具有相同的时钟速率）处理器，而不是一块80486和一块Pentium，或者一块Pentium和一块PowerPC。在通常的用法中，SMP也意味着所有CPU都是“在相同处境下的”——那就是它们都在同一个计算机里，通过特殊用途的硬件进行彼此通信。

SMP系统通常是另一种普通的单一计算机——只不过具有两个或更多的CPU。因此，SMP系统除了CPU以外，每样东西只有一个——一块图形卡、一个声卡，等等之类。诸如RAM和磁盘及类似的资源都是为系统的CPU们所共享的（尽管现在SMP系统中每个CPU都拥有自己的高速缓存的情况已经变得愈发普遍了）。

分布式配置需要很少的甚至不需要来自内核的特殊支持；节点之间的协同是依靠用户空间的应用程序或者诸如网络子系统之类未经修改的内核组件来处理的。但是SMP在计算机系统内创建了一个不同的硬件配置，并由此需要特殊用途的内核支持。比如，内核必须确保CPU在访问它们的共享资源时要相互合作——这是一个读者在UP世界中不曾遇到的问题。

SMP的逐渐普及主要是因为：要提高性能通过SMP比购买几台独立的机器再把它们组合在一起更加便宜和简单，而且还因为它与等待下一代CPU面世相比要快得多。

非对称多CPU的配置没有受到广泛支持，这是因为对称配置情况所需的硬件和软件支持通常较为简单。不过，内核代码中平台无关的部分实际上并不特别关心CPU是否相同——即，是否配置是真正对称的——尽管它也没有进行任何特殊处理以支持非对称配置。例如，在非对称多处理系统中，调度程序应该更愿意在较快的而不是较慢的CPU上运行进程，但是Linux内核没有对此进行区别。

谚语说得好，“天下没有白吃的午餐”。对于SMP，为提高性能所付出的代价就是内核复



杂度的增加和协同开销的增加。CPU必须安排的不互相干涉彼此的工作，但是它们又不能在这种协同上花费太多时间而造成显著地耗费额外的CPU能力。

代码的SMP特定部分由于UP机器存在的缘故而被单独编译，所以仅仅因为有了SMP寄存器是不会使UP寄存器慢下来的。这满足两条久经考验的原理：“为普遍情况进行优化”（UP机器远比SMP机器普遍）及“不为用不着的东西花钱”。

## 10.1 并行程序设计概念及其原语

具有两个CPU的SMP配置可能是最简单的并行配置，但就算是这最简单的配置也揭开了未知问题的新领域——即使要两块相同的CPU在一起协调工作，时常也都像赶着猫去放牧一样困难。幸运的是，至少自30年前以来，就在这个项目上做了大量和非常熟悉的研究工作（考虑到第一台电子数字计算机也只是在50年前建造的，那这就是一段令人惊讶的相当长的时间了）。在分析对SMP的支持是如何影响内核代码之前，对该支持所基于的若干理论性概念进行一番浏览将极大地简化这个问题。

注意：并非所有这些信息都是针对SMP内核的。一些要讨论的问题甚至是由UP内核上的并行程序设计所引起的，既要支持中断也要处理进程之间的交互。因此即使你对SMP问题没有特别的兴趣，这部分的讨论也值得一看。

### 10.1.1 原子操作

在一个并行的环境里，某些动作必须以一种基本的原子方式（atomically）执行——即不可中断。这种操作必须是不可分割的，就像是原子曾经被认为的那样。

例如，考虑一下引用计数。如果你想要释放你所控制的一份共享资源，并要了解是否还有其他（进程）仍在用它，你就会减少对该共享资源的计数值，并把该值与0进行对照测试。一个典型的动作顺序可能如下所示：

- 1) CPU把当前计数值（假设是2）装载进它的一个寄存器里。
- 2) CPU在它的寄存器里把这个值递减；现在它是1。
- 3) CPU把新值（1）写回内存里。

4) CPU推断出：因为该值是1，某个其他进程仍在使用着共享对象，所以它将不会释放该对象。

对于UP，应不必在此考虑过多（除了某些情况）。但是对于SMP就是另一番景象了：如果另一个CPU碰巧同时也在做同样的事情该如何处理呢？最坏的情形可能是这样的：

- 1) CPU A把当前计数值（2）装载进它的一个寄存器里。
- 2) CPU B把当前计数值（2）装载进它的一个寄存器里。
- 3) CPU A在它的寄存器里把这个值递减；现在它是1。
- 4) CPU B在它的寄存器里把这个值递减；现在它是1。
- 5) CPU A把新值（1）写回内存里。
- 6) CPU B把新值（1）写回内存里。

7) CPU A推断出：因为该值是1，某个其他进程仍在使用着共享对象，所以它将不会释放该对象。

- 8) CPU B推断出：因为该值是1，某个其他进程仍在使用着共享对象，所以它将不会释放

该对象。

内存里的引用计数值现在应该是0，然而它却是1。两个进程都去掉了它们对该共享对象的引用，但是没有一个能够释放它。

这是一个有趣的失败，因为每个CPU都做了它应该做的事情，尽管这样，错误的结果还是发生了。当然这个问题就在于CPU没有协调它们的动作行为——右手不知道左手正在干什么。

你会怎样试图在软件中解决这个问题呢？从任何一个CPU的观点来看待它——比如说是CPU A。需要通知CPU B它不应使用引用计数值，由于你想要递减该值，所以不管怎样你最好改变某些CPU B所能见到的信息——也就是更新共享内存位置。举例来说，你可以为此目的而开辟出某个内存位置，并且对此达成一致：若任何一个CPU正试图减少引用计数它就包含一个1，如果不是它就为0。使用方法如下：

1) CPU A从特殊内存位置取出该值把它装载进它的一个寄存器里。

2) CPU A检查它的寄存器里的值并发现它是0（如果不是，它再次尝试，重复直到该寄存器为0为止）。

3) CPU A把一个1写回特殊内存位置。

4) CPU A访问受保护的引用计数值。

5) CPU A把一个0写回特殊内存位置。

糟糕，令人不安的熟悉情况又出现了。以下所发生的问题仍然无法避免：

1) CPU A从特殊内存位置取出该值把它装载进它的一个寄存器里。

2) CPU B从特殊内存位置取出该值把它装载进它的一个寄存器里。

3) CPU A检查它的寄存器里的值并发现它是0。

4) CPU B检查它的寄存器里的值并发现它是0。

5) CPU A把一个1写回特殊内存位置。

6) CPU B把一个1写回特殊内存位置。

7) CPU A访问受保护的引用计数值。

8) CPU B访问受保护的引用计数值。

9) CPU A把一个0写回特殊内存位置。

10) CPU B把一个0写回特殊内存位置。

好吧，或许可以再使用一个特殊内存位置来保护被期望保护初始内存位置的那个特殊内存位置……

这种方案只会使问题向后再退一层，而不可能解决它。最后，原子性不可能由软件单独保证——必须要有硬件的特殊帮助。

在x86平台上，lock指令正好能提供这种帮助（准确地说，lock是一个前缀，而非一个单独的指令，不过这种区别和我们的目的没有利害关系）。lock指令用于在随后的指令执行期间锁住内存总线——至少是对目的内存地址。因为x86可以在内存里直接减值，而无需先把它读入一个寄存器中，这样对于执行一个减值原子操作来说就是万事俱备了：lock内存总线然后立刻对该内存位置执行decl操作。

函数atomic\_dec（10241行）正好为x86平台完成这样的工作。LOCK宏的SMP版本在第10192行定义并扩展成lock指令（在随后的两行定义的UP版本完全就是空的——单CPU不需要

保护自己以防其他CPU的干扰，所以锁住内存总线将完全是在浪费时间)。通过把LOCK宏放在内联编译指令的前面，随后的指令就会为SMP内核而被锁定。如果CPU B在CPU A发挥作用时执行了atomic\_dec函数，那么CPU B就会自动等待CPU A把锁移开。这样就能够成功了！

这样还只能说是差不多。最初的问题仍然没有被很好地解决。目标不仅是要自动递减引用计数值，而且还要知道结果值是否是0。现在可以完成原子递减了，可是，如果另一个处理器在递减和结果测试之间又“偷偷的”进行了干预，那又怎么办呢？

幸运的是，解决这个问题不需要来自CPU的特殊目的的帮助。不管加锁还是未锁，x86的decl指令总是会在结果为0时设置CPU的Zero标志位，而且这个标志位是CPU专有的，所以其他CPU的行为是不可能在此步骤和测试步骤之间影响到这个标志位的。相应地，atomic\_dec\_and\_test（10249行）如前面一样完成一次加锁的递减，接着依据CPU的Zero标志位来设置本地变量c。如果递减之后结果是0，函数就返回非零值（真）。

如同其他定义在一个文件里的函数一样，atomic\_dec和atomic\_dec\_and\_test都对一个类型为atomic\_t的（10205行）对象进行操作。就像LOCK，atomic\_t对于UP和SMP也有不同的定义方式——不同之处在于SMP情况里引入了volatile限定词，它指示gcc不要对被标记的变量做某种假定（比如，不要假定它可以被安全地保存在一个寄存器里）。

顺便提及一下，读者在这段代码里看到的垃圾代码\_\_atomic\_fool\_gcc据报告已不再需要了；它曾用于纠正在gcc的早期版本下代码生成里的一个故障。

### 10.1.2 test-and-set

经典的并行原语是test-and-set。test-and-set操作自动从一个内存位置读取一个值，然后写入一个新值，并把旧值返回。一般情况，该位置可以保存0或者1，而且test-and-set所写的新值是1——因此是“设置（set）”。与test-and-set对等的是test-and-clear，除了写入的是0而不是1外，它的操作是同样的。一些test-and-set的变体既能写入1，也可以写入0，因此test-and-set和test-and-clear就能够成为一体，只是操作数不同而已。

test-and-set原语足以实现任何其他并行安全的操作（实际上，在某些CPU上test-and-set是唯一被提供的此类原语）。比如，原本test-and-set是能够用于前边的例子之中来保护引用计数值的。相似的方法已被尝试——从一个内存位置读取一个值，检查它是否为0，如果是，则写入一个1，然后继续访问受保护的。这种尝试的失败并不是因为它在逻辑上是不健全的，而是因为没有一个可行的方法使其自动完成。假使有了一个原子的test-and-set，你就可以不通过使用lock来原子化decl而顺利通过了。

然而，test-and-set也有缺点：

- 它是一个低级的原语——所有的与它打交道时，其他原语都必须在它之上被执行。
- 它并不经济——当机器测试该值并发现它已经是1了怎么办呢？这个值在内存里不会被搞乱，因为只要用同样的值复写它即可。可事实是它已被设置，就意味着其他进程正在访问受保护的，所以还不能这样执行。额外需要的逻辑——测试并循环——会浪费CPU时钟周期，并使得程序变得更大（它还会浪费高速缓存里的空间）。

x86的lock指令使高级指令更容易执行，但是你也可以在上执行原子test-and-set操作。最直接的方式是把lock和btsl指令（位test-and-set）联合起来使用。这种方法在本章后面介绍的自旋锁（spinlock）中会用到。

另一种在x86上实现的方法是用它的xchg (exchange) 指令, 它会被x86自动处理, 就好像它的前面有一个lock指令一样——只要它的一个操作数是在内存里。xchg要比lock/ btsl组合更为普遍, 因为它可以一次交换8、16或32位, 而不仅仅是1位。除了一个在arch/i386/kernel/entry.S里的使用之外, 内核对xchg指令的使用都隐藏在xchg宏 (13052行) 之后, 而它又是在函数\_\_xchg (13061行) 之上实现的。这样是便于在平台相关的代码里, 内核代码也可以使用xchg宏; 每种平台都提供它自己对于该宏的等价的实现。

有趣的是, xchg宏是另一个宏tas (test-and-set——13054行) 的基础。然而, 内核代码的任何一个地方都没有用到这个宏。

内核有时候使用xchg宏来完成简单的test-and-set操作 (尽管不必在锁变得可用之前一直循环, 如同第22770行), 并把它用于其他目的 (如同第27427行)。

### 10.1.3 信号量

第9章中讨论了信号量的基本概念并演示了它们在进程间通信中的用法。内核为达到自己的目的有其特有的信号量实现, 它们一般称为“内核信号量” (在这一章里, 未经修饰的名词“信号量”应被理解为是“内核信号量”)。第9章里所讨论的基本信号量的概念同样适用于内核信号量: 允许一个可访问某资源用户的最大数目 (最初悬挂在吊钩上钥匙的特定数目), 然后规定每个申请资源者都必须先获得一把钥匙才能使用该资源。

到目前为止, 你大概已经发现信号量如何能够被建立在test-and-set之上, 并成为二元 (“唯一钥匙”) 信号量, 或者在像atomic\_dec\_and\_test这样的函数之上成为计数信号量的过程。内核正好就完成着这样的工作: 它用整数代表信号量, 并使用函数down (11644行) 和up (11714行) 以及其他一些函数来递减和递增该整数。读者将看到, 用于减少和增加整数的底层代码和atomic\_dec\_and\_test及其他类似函数所使用的代码是一样的。

第一位规范信号量概念的研究者, Edsger Dijkstra是荷兰人, 所以信号量的基础操作就用荷兰语命名为: Proberen和Verhogen, 常缩写成P和V。这对术语被翻译成“测试 (test)” (检查是否还有一把钥匙可用, 若是就取走) 和“递增 (increment)” (把一个钥匙放回到吊钩之上)。那些词首字母正是在前一章中所引入的术语“获得 (procure)” 和“交出 (vacate)” 的来源。Linux内核打破了这个传统, 用操作down和up的称呼取代了它们。

内核用一个非常简单的类型来代表信号量: 定义在11609行的struct semaphore。他只有三个成员:

- count——跟踪仍然可用的钥匙数目。如果是0, 钥匙就被取完了; 如果是负数, 钥匙被取完而且还有其他申请者在等待它。另外, 如果count是0或负数, 那么其他申请者的数目就等于count的绝对值。

sema\_init宏 (11637行) 允许count被初始化为任何值, 所以内核信号量可以是二元的 (初始化count为1), 也可以是计数型的 (赋予它某个更大的初始值)。所有内核信号量代码都完全支持二元和计数型信号量, 前者可作为后者的一个特例。不过在实践中, count总是被初始化为1, 这样内核信号量也总是二元类型的。尽管如此, 没有什么能够阻止一个开发者将来增加一个新的计数信号量。

要顺便提及的是, 把count初始化为正值而且用递减它来表明你需要一个信号量的方法并没有什么神秘之处。你也可以用一个负值 (或者是0) 来初始化计数值然后增加它, 或者遵循

其他的方案。使用正的数字只是内核所采用的办法，而这碰巧和我们头脑中的吊钩上的钥匙模型吻合得相当好。的确，正如你将看到的那样，内核锁采用的是另一种方式工作——它被初始化为负值，并在进程需要它时进行增加。

- waking——在up操作期间及之后被暂时使用；如果up正在释放信号量，则它被设置为1，否则是0。
- wait——因为要等待这个信号量再次变为可用而不得被挂起的进程队列。

#### 1. down

11644: down操作递减信号量计数值。你可能会认为它与概念里的实现一样简单，不过实际上远不是这样简单。

11648: 减少信号量计数值——要确保对SMP这是自动完成的。对于SMP来说（当然也适于UP），除了被访问的整数是在一个不同类型的struct之内以外，这同在atomic\_dec\_and\_test中所完成的工作本质上是相同的。

读者可能会怀疑count是否会下溢。它不会：进程总是在递减count之后进入休眠，所以一个给定的进程一次只能获得一个信号量，而且int具有的负值要比进程的数目多得多。

11652: 如果符号位被设置，信号量就是负值。这意味着甚至它在被递减之前就是0或者负值了，这样进程无法得到该信号量，并因此而应该休眠到它变成可用。接下来的几行代码十分巧妙地完成了这一点。如果符号位被设置则执行js跳转（即若decl的结果是负的它就跳转），2f标识出跳转的目的地。2f并非十六进制值——它是特殊的GNU汇编程序语法：2表示跳转到本地符号“2”，f表示向前搜索这个符号（2b将表示向后搜索最近的本地符号“2”）。这个本地符号在第11655行。

11653: 分支转移没有执行，所以进程得到了信号量。虽然看起来不是这样，但是这实际已经到达down的末尾。稍后将对此进行解释。

11654: down的技巧在于指令.section紧跟在跳转目标的前面，它表示把随后的代码汇编到内核的一个单独的段中——该段被称为.text.lock。这个段将在内存中被分配并标识为可执行的。这一点是由跟在段名之后的ax标志字符串来指定的——注意这个ax与x86的AX寄存器无关。

这样的结果是，汇编程序将把11655和11656行的指令从down所在的段里转移到可执行内核的一个不同的段里。所以这些行生成的目标代码与其前面行所生成的代码从物理上不是连续的。这就是为什么说11653行是down的结尾的原因。

11655: 当信号量无法得到时跳转到的目的行。pushl \$1b并不是要把十六进制值1b压入栈中——如果要执行那种工作，应该使用pushl \$0x1b（也可以写成不带\$的）。正确的解释是，这个1b和前面见到的2f一样，都是GNU汇编程序语法——它指向一个指令的地址；在此情形中，它是向后搜索时碰到的第一个本地标号“1”的地址。所以，这条指令是把11653行代码的地址压入栈中；这个地址将成为返回地址，以便在随后的跳转操作之后，执行过程还能返回到down的末尾。

11656: 开始跳转到\_\_down\_failed（不包括在本书之内）。这个函数在栈里保存几个寄存器，并调用后面要介绍的\_\_down（26932行）来完成等待信号量的工作。一旦\_\_down返回了，\_\_down\_failed就返回到down，而它也随之返回。一直到进程获

得了信号量\_\_down才会返回；最终结果就是只要down返回，进程就得到信号量了，而不管它是立刻还是经过等待后获得的。

11657: 伪汇编程序指令.previous的作用未在正式文档中说明，但是它的意思肯定是还原到以前的段中，结束11654行里的伪指令.section的作用效果。

## 2. down\_interruptible

11664: down\_interruptible函数被用于进程想要获得信号量、但也愿意在等待它时被信号中断的情况。这个函数与down的实现非常相似，不过有两个区别将在随后的两段里进行解释。

11666: 第一个区别是down\_interruptible函数返回一个int值来指示是否它获得了信号量或者被一个信号所打断。在前一种情况里返回值（在result里）是0，在后一种情况里它是负值。这部分上是由11675行代码完成的，如果函数未经等待获得了信号量，则该行把result设置为0。

11679: 第二个区别是down\_interruptible函数跳转到\_\_down\_failed\_interruptible（不包括在本书之内）而不是\_\_down\_failed。遵循\_\_down\_failed建立起来的模式，\_\_down\_failed\_interruptible只是调整几个寄存器，并调用将在随后进行研究的\_\_down\_interruptible函数（26942行）。要注意的是11676行为\_\_down\_failed\_interruptible设置的返回目标跟在xorl之后，xorl用于在信号量可以被立刻获得的情况下把result归0。\_\_down\_interruptible函数的返回值再被复制进result中。

## 3. down\_trylock

11687: 除了调用\_\_down\_failed\_trylock函数（当然还要调用26961行的\_\_down\_trylock函数，我们将在后面对它进行检查）之外，down\_trylock函数和down\_interruptible函数相同。因此，在这里不必对down\_trylock函数进行更多解释。

## 4. DOWN\_VAR

26900: 这是作为\_\_down和\_\_down\_interruptible共同代码析出因数的三个宏中的第一个。它只是声明了几个变量。

## 5. DOWN\_HEAD

26904: 这个宏使任务tsk（被DOWN\_VAR所声明）转移到task\_state给出的状态，然后把tsk添加到等待信号量的任务队列。最后，它开始一个无限循环，在此循环期间，当\_\_down和\_\_down\_interruptible准备退出时将使用break语句结束该循环。

## 6. DOWN\_TAIL

26926: 这个宏完成循环收尾工作，把tsk设置回task\_state的状态，为再次尝试获得信号量做准备。

26929: 循环已经退出；tsk已得到了信号量或者被一个信号中断了（仅适于\_\_down\_interruptible）。无论哪一种方式，任务已准备再次运行而不再等待该信号量了，因此，它被转移回TASK\_RUNNING，并从信号量的等待队列里被注销。

## 7. \_\_down

26932: \_\_down和\_\_down\_interruptible遵循以下模式：

- 1) 用DOWN\_VAR声明所需的本地变量，随后可能还有补充的本地变量声明。
- 2) 以DOWN\_HEAD开始进入无限循环。

3) 在循环体内完成函数特定的 (function-specific) 工作。

4) 重新调度。

5) 以DOWN\_TAIL结束。注意对schedule的调用 (26686行, 在第7章里讨论过) 可以被移进DOWN\_TAIL宏中。

6) 完成任何函数特定的收尾工作。

我将只对函数特定的步骤 (第3和第6步) 进行讨论。

26936: \_\_down的循环体调用waking\_non\_zero (未包括), 它自动检查sem->waking来判断是否进程正被up唤醒。如果是这样, 它将waking归零并返回1 (这仍然是同一个原子操作的一部分); 如果不是, 它返回0。因此, 它返回的值指示了进程是否获得了信号量。如果它获得了值, 循环就退出, 接着函数也将返回。否则, 进程将继续等待。

顺便要说明的是, 观察一下\_\_down尝试获得信号量是在调用schedule之前。如果信号量的计数值已知为负值时, 为什么不用另一种相反的方式来实现它呢? 实际上它对于第一遍循环之后的任何一遍重复都是没有影响的, 但是去掉一次没有必要的检查可以稍微加快第一遍循环的速度。如果需要为此提出什么特别的理由的话, 那可能就是因为, 自从信号量第一次被检查之后的几个微秒内它就应该可以被释放 (可能是在另一个处理器上), 而且额外获取标志要比一次额外调度所付出的代价少得多。因此\_\_down可能还可以在重新调度之前做一次快速检查。

#### 8. \_\_down\_interruptible

26942: \_\_down\_interruptible除了允许被信号中断以外, 它和\_\_down在本质上是一样的。

26948: 所以, 当获取信号量时对waking\_non\_zero\_interruptible (未包括) 进行调用。如果它没能得到信号量就返回0, 如果得到就返回1, 或者如果它被一个信号所中断就返回-EINTR。在第一种情况下, 循环继续。

26958: 否则, \_\_down\_interruptible退出, 如果它得到信号量就返回0 (不是1), 或者假如被中断则返回-EINTR。

#### 9. \_\_down\_trylock

26961: 有时在不能立刻获得信号量的情况下, 内核也需要继续运行。所以, \_\_down\_trylock不在循环之内。它仅仅调用waking\_nonzero\_trylock (未包括), 该函数夺取信号量, 如果失败就递增该信号量的count (因为内核不打算继续等待下去), 然后返回。

#### 10. up

11714: 我们已经详尽地分析了内核尝试获得信号量时的情况, 也讨论了它失败时的情况。现在考察当释放一个信号量时将发生什么。这一部分相对简单。

11721: 原子性地递增信号量的计数值。

11722: 如果结果小于等于0, 就有某个进程正在等待被唤醒。up向前跳转到11725行。

11724: up采用了down里同样的技巧: 这一行进入了内核的单独的一段, 而不是在up本身的段内。up末尾的地址被压入栈, 然后up跳转到\_\_up\_wakeup (未包括)。这里完成如同\_\_down\_failed一样的寄存器操作, 并调用下面要讨论的\_\_up函数。

#### 11. \_\_up

26877: `__up`函数负责唤醒所有等待该信号量的进程。

26879: 调用`wake_one_more`（未包括在本书中），该函数检查是否有进程在等待该信号量，如果有，就增加waking成员来通知它们可以尝试获取它了。

26880: 利用`wake_up`宏（16612行），它只是调用`__wake_up`函数（26829行）来唤醒所有等待进程。

## 12. `__wake_up`

26829: 正如在第2章中所讨论的那样，`__wake_up`函数唤醒所有传递给它的在等待队列上的进程，假如它们处于被mode所隐含的状态之一的话。当从`wake_up`被调用时，函数唤醒所有处于`TASK_UNINTERRUPTIBLE`或`TASK_INTERRUPTIBLE`状态的进程；当从`wake_up_interruptible`（16614行）被调用时，它只唤醒处于`TASK_INTERRUPTIBLE`状态的任务。

26842: 进程用`wake_up_process`（26356行）被唤醒，该函数曾在以前提到过，它将在本章随后进行详细介绍。

现在所感兴趣的是唤醒所有进程后的结果。因为`__wake_up`唤醒所有队列里的进程，而不仅仅是队列里的第一个，所以它们都要竞争信号量——在SMP里，它们可以精确地同时做这件事。通常，获胜者将首先获得CPU。这个进程将是拥有最大“goodness”的进程（回忆一下第7章中26388行对goodness的讨论）。这一点意义非常重大，因为拥有更高优先权的进程应该首先被给予继续其工作的机会（这对于实时进程尤其重要）。

这种方案的不足之处是有发生“饥饿（starvation）”的危险，这发生在一个进程永远不能得到它赖以继续运行的资源时。这里可能会发生饥饿现象：假如两个进程反复竞争同一个信号量，而第一个进程总是有比第二个更高的优先权，那么第二个进程将永远不会得到CPU。这种场景同它应该的运行方式存在一定差距——设想一个是实时进程，而另一个以20的完美级运行。我们可以通过只唤醒队列里第一个进程的方法来避免这种饥饿的危险，可是那样又将意味着有时候会耽误从各个方面来说都更有资格的进程对CPU的使用。

以前对此没有讨论过，可是Linux的调度程序在适当的环境下也能够使得CPU的一个进程被彻底饿死。这不完全是一件坏事——只是一种设计决策而已——而且至少应用于通篇内核代码的原则是一致的，这就很好。还要注意的是使用前面讨论过的其他机制，饥饿现象也同样会发生。例如说，`test-and-set`原语就是和内核信号量一样的潜在饥饿根源。

无论如何，在实际中，饥饿是非常少见的——它只是一个有趣的理论案例。

### 10.1.4 自旋锁

这一章里最后一个重要的并程序序设计原语是自旋锁（spinlock）。自旋锁的思想就是在密封的循环里坚持反复尝试夺取一个资源（一把锁），直到成功为止。这通常是通过在类似`test-and-set`操作之上进行循环来实现的——即，旋转——直到获得该锁。

如果这听起来好像是一个二元信号量，那是因为它就是一个二元信号量。自旋锁和二元信号量唯一的概念区别就是你不必循环等待一个信号量——你可以夺取信号量，也可以在不能立刻得到它时放弃申请。因此，自旋锁原本是可以通过在信号量代码外再包裹一层循环来实现的。不过，因为自旋锁是信号量的一个受限特例，它们有更高效率的实现方法。

自旋锁变量——其中的一位被测试和设置——总是`spinlock_t`类型（12787行）。只有



spinlock\_t的最低位被使用；如果锁可用，则它是0，如果被取走，则它是1。在一个声明里，自旋锁被初始化为值SPIN\_LOCK\_UNLOCKED（12789行）；它也可以用spin\_lock\_init函数（12791行）来初始化。这两者都把spinlock\_t的lock成员设置成0——也就是未锁状态。

注意12795行代码简洁地对公平性进行了考虑，并最后抛弃了它——公平是饥饿的背面，正如我们前面已经介绍过的（使得一个CPU或进程饥饿应被认为是“不公平的”）。

自旋锁的加锁和解锁宏建立在spin\_lock\_string和sping\_unlock\_string函数之上，所以这一小节只对spin\_lock\_string和sping\_unlock\_string函数进行详述。其他宏只是增加了IRQ加锁和解锁。

### 1. spin\_lock\_string

12805: 这个宏的代码对于所有自旋锁加锁的宏都是相同的。它也被用于x86专用的lock\_kernel和unlock\_kernel版本之中（它们不在本书讨论之列，不过其常规版本则是包括的——参见10174和10182行）。

12807: 尝试测试和设置自旋锁的最低位，这要把内存总线锁住，以便对于任何其他对同一个自旋锁的访问来说这个操作都是原子的。

12808: 如果成功了，控制流程就继续向下运行；否则，spin\_lock\_string函数向前跳转到第12810行（btsl把这一位的原值放入CPU的进位标志位（Carry flag），这正是这里使用jc的原因）。同样的技巧我们已经看到过三次了：跳转目标放在内核的单独一段中。

12811: 在封闭的循环里不停地检测循环锁的最低位。注意btsl和testb以不同方式解释它们第一个操作数——对于btsl，它是一个位状态，而对于testb，它是一个位屏蔽。因此，12811行在测试spin\_lock\_string曾在12807行已经试图设置（但失败了）的同一位，尽管一个使用\$0，而另一个使用\$1。

12813: 该位被清除了，所以spin\_lock\_string应该再次夺取它。函数调转回第12806行。

这个代码可以只用加上lock前缀的两条代码加以简化：

```
1: lock ; btsl $0, %0
 jc 1b
```

不过，使用这个简化版本的话，系统性能将明显受到损害，因为每次循环重复内存总线都要被加锁。内核使用的版本虽然长一些，但是它可以使其他CPU运行的更有效，这是由于该版本只有在它有充分理由相信能够获得锁的时候才会锁住内存总线。

### 2. spin\_unlock\_string

12816: 并不很重要：只是重新设置了自旋锁的锁定位（lock bit）。

#### 读/写自旋锁

自旋锁的一个特殊情况就是读/写自旋锁。这里的思想是这样的：在某些情况中，我们想要允许某个对象有多个读者，但是当有一个写者正在写入这个对象时，则不允许它再有其他读者或者写者。

遵循基于spinlock\_t的自旋锁的同样模式，读/写自旋锁是用rwlock\_t（12853行）来代表的，它可以在有RW\_LOCK\_UNLOCKED（12858行）的声明里被初始化。与rwlock\_t一起工作的最低级的宏是read\_lock、read\_unlock、write\_lock以及write\_unlock，它们在本小节中进行

行描述。很明显，那些跟随在这些宏之后并建立在它们之上的宏，自然要在你理解了最初的这四个宏之后再去接触。

正如第12860行注释中所声明的，当写锁被占有时，`rwlock_t`的`lock`成员是负值。当既没有读者也没有写者时它为0，当只有读者而没有写者时它是正值——在这种情况下，`lock`将对读者的数目进行计数。

### 3. `read_lock`

12867: 始于`rwlock_t`的`lock`成员的自动递增。这是推测性的操作——它可以被撤销。

12868: 如果它在增量之后为负，表示某个进程占用了写锁——或者至少是某个进程正试图得到它。`read_lock`向前跳到第12870行（注意，在一个不同的内核段里）。否则，没有写者退出（尽管还有可能有，或者也可能没有其他读者——这并不重要），所以可以继续执行读锁定代码。

12870: 一个写者出现了。`read_lock`取消第12867行增值操作的影响。

12871: 循环等待`rwlock_t`的`lock`变为0或正值。

12873: 跳回到第12866行再次尝试。

### 4. `read_unlock`

12878: 不太复杂：只是递减该计数值。

### 5. `write_lock`

12883: 一个进程需要写锁的信号：检测并设置`lock`的符号位，并保证`lock`的值是负的。

12884: 如果符号位已经被设置，则另外有进程占有了写锁；`write_lock`向前跳转到第12889行（同以前一样，那是在一个不同的内核段里）。

12885: 没有别的进程正试图获得该写锁，可是读者仍可以退出。因为符号位被设置了，读者不能获得读锁，但是`write_lock`仍然必须等待正在退出的读者完全离开。它通过检查低端的31位中是否任何一位被设置过开始，这可以表示`lock`以前曾是正值。如果没有，则`lock`在符号位反转之前曾是0，这意味着没有读者；因而，这对于写者的继续工作是很安全的，所以控制流程就可以继续向下运行了。不过，如果低端31位中任何一位被设置过了，也就是说有读者了，这样`write_lock`就会向前跳转到第12888行等到它们结束。

12888: 该进程是仅有的写者，但是有若干读者。`write_lock`会暂时清除符号位（这个宏稍后将再次操纵它）。有趣的是，对符号位进行这样的胡乱操作并不会影响读者操纵`lock`的正确性。考虑作为示例的下列顺序事件：

1) 两个读者增加了`lock`；`lock`用十六进制表示现在是0x00000002。

2) 一个即将成为写者的进程设置了符号位；`lock`现在是0x80000002。

3) 读者中的一个离开；`lock`现在是0x80000001。

4) 写者看到剩余的位不全部是0——仍然有读者存在。这样它根本没有写锁，因此它就清除符号位；`lock`现在是0x00000001。

这样，读和写可以任意顺序交错尝试操作而不会影响结果的正确程度。

12889: 循环等待计数值降到0——也就是等待所有读者退出。实际上，0除了表示所有读者已离开之外，它还表示着没有其他进程获得了写锁。

12891: 所有读者和写者都结束了操作；`write_lock`又从头开始，并再次获得写锁。

## 6. write\_unlock

12896: 不太重要: 只是重置符号位。

## 10.2 APIC和CPU-To-CPU通信

Intel 多处理规范的核心就是高级可编程中断控制器 (Advanced Programmable Interrupt Controllers, APIC) 的使用。CPU通过彼此发送中断来完成它们之间的通信。通过给中断附加动作, 不同的CPU可以在某种程度上彼此进行控制。每个CPU有自己的APIC (成为那个CPU的本地APIC), 并且还有一个I/O APIC来处理由I/O设备引起的中断。在普通的多处理器系统中, I/O APIC取代了第6章里提到的中断控制器芯片组的作用。

这里有几个示例性的函数来让你了解其工作的风格。

## 1. smp\_send\_reschedule

5019: 这个函数只有一行, 其作用将在本章随后进行说明, 它仅仅是给其ID以参数形式给出的目标CPU发送一个中断。函数用CPU ID和RESCHEDULE\_VECTOR向量调用send\_IPI\_single函数 (4937行)。RESCHEDULE\_VECTOR与其他CPU中断向量是一起在第1723行开始的一个定义块中被定义的。

## 2. send\_IPI\_single

4937: send\_IPI\_single函数发送一个IPI——那是Intel对处理器间中断 (interprocessor interrupt) 的称呼——给指定的目的CPU。在这一行, 内核以相当低级的方式与发送CPU的本地APIC对话。

4949: 得到中断命令寄存器 (ICR) 上半部分的内容——本地APIC就是通过这个寄存器进行编程的——不过它的目的信息段要被设置为dest。尽管\_\_prepare\_ICR2 (4885行) 里使用了“2”, CPU实际上只有一个ICR, 而不是两个。但是它是一个64位寄存器, 内核更愿意把它看作是两个32位寄存器——在内核代码里, “ICR”表示这个寄存器的低端32位, 所以“ICR2”就表示高端32位。我们想要设置的目的信息段就在高端32位, 即ICR2里。

4950: 把修改过的信息写回ICR。现在ICR知道了目的CPU。

4953: 调用\_\_prepare\_ICR (4874行) 来设置我们想要发送给目的CPU的中断向量 (注意没有什么措施能够保证目的CPU不是当前CPU——ICR完全能够发送一个IPI给它自己的CPU。尽管这样, 我还是没有找到有任何理由要这样做)。

4957: 通过往ICR里写入新的配置来发送中断。

## 10.3 SMP支持如何影响内核

既然读者已经学习了能够成功支持SMP的若干原语, 那么就让我们来浏览一下内核的SMP支持吧。本章剩余的部分将局限于对分布在内核之中的那些具有代表性的SMP代码进行讨论。

### 10.3.1 对调度的影响

schedule (26686行) 正是内核的调度函数, 它已在第7章中全面地介绍过了。schedule的SMP版本与UP的相比有两个主要区别:

- 在schedule里，从第26780开始的一段代码要计算某些其他地方所需的信息。
- 在SMP和UP上都要发生的对\_\_schedule\_tail的调用（26638行）实际上在UP上并无作用，因为\_\_schedule\_tail完全是为SMP所写的代码，所以从实用角度来说，它就是SMP所特有的。

### 1. schedule

- 26784: 获取当前时间，也就是自从机器开机后时钟流逝的周期数。这很像是检查jiffies，不过是以CPU周期而不是以时钟滴答作为计时方法的——显然，这要精确得多。
- 26785: 计算自从schedule上一次在此CPU上进行调度后过去了多长时间，并且为下一次的计算而记录下当前周期计数（schedule\_data是每个CPU aligned\_data数组的一部分，它在26628行定义）。
- 26790: 进程的avg\_slice成员（16342行）记录该进程在其生命周期里占有CPU的平均时间。可是这并不是简单的平均——它是加权平均，进程近期的活动远比很久以前的活动权值大（因为真实计算机的计算是有穷的，“很久以前”的部分在足够远以后，将逐渐趋近于0）。这将在reschedule\_idle中（26221行，下文讨论）被用来决定是否把进程调入另一个CPU中。因此，在UP的情况下，它是无需而且也不会被计算的。
- 26797: 记录哪一个CPU将运行next（它将在当前的CPU上被执行），并引发它的has\_cpu标志位。
- 26803: 如果上下文环境发生了切换，schedule记录失去CPU的进程——这将在下文的\_\_schedule\_tail中用到。

### 2. \_\_schedule\_tail

- 26654: 如果失去CPU的任务已经改变了状态（这一点在前边的注释里解释过了），它将被标记以便今后的重新调度。
- 26664: 因为内核已经调度出了这个进程，它就不再拥有CPU了——这样的事实也将被记录。

### 3. reschedule\_idle

- 26221: 当已经不在运行队列里的进程被唤醒时，wake\_up\_process将调用reschedule\_idle，进程是作为p而被传递进reschedule\_idle中的。这个函数试图把新近唤醒的进程在一个不同的CPU上进行调度——即一个空闲的CPU上。
- 26225: 这个函数的第一部分在SMP和UP场合中都是适用的。它将使高优先级的进程得到占用CPU的机会，同时它也会为那些处于饥饿状态的进程争取同样的机会。如果该进程是实时的或者它的动态优先级确实比当前占有CPU进程的动态优先级要高某个量级（强制选定的），该进程就会被标记为重新调度以便它能够争取占用CPU。
- 26263: 现在来到SMP部分，它仅仅适用于在上述测试中失败了的那些进程——虽然这种现象经常发生。reschedule\_idle必须确定是否要在另一个CPU上尝试运行该进程。正如在对schedule的讨论中所提到的那样，一个进程的avg\_slice成员是它对CPU使用的加权平均值；因此，它说明了假如该进程继续运行的话，是否它可能要控制CPU相对来说较长的一段时间。

26264: 这个if条件判断的第二个子句使用related宏（就在本函数之上的第26218行）来测试是否CPU都在控制着——或想要控制——内核锁。如果是这样，那么不管它们生存于何处，都将不大可能同时运行，这样把进程发送到另一个CPU上将不会全面提高并行的效能。因此，假如这条子句或者前一条子句被满足，函数将不会考虑使进程在另一CPU上进行调度，并简单地返回。

26267: 否则，reschedule\_idle\_slow（接下来讨论）被调用，以决定进程是否应当被删除。

#### 4. reschedule\_idle\_slow

26157: 正如注释中所说明的，reschedule\_idle\_slow试图找出一个空闲CPU来贮存p。这个算法是基于如下观察结果的，即task数组的前n项是系统的空闲进程，机器的n个CPU中每个都对应一个这样的空闲进程。这些空闲进程当（且仅当）对应CPU上没有其他进程需要处理器时才会运行。如果可能，函数通常是用hlt指令使CPU进入低功耗的“睡眠”状态。

因此，如果有空闲CPU存在的话，对任务数组的前n个进程进行循环是找出一个空闲CPU所必须的。reschedule\_idle\_slow函数只需简单地查询每个空闲进程是否此刻正在运行着；如果是这样，它所在的CPU就一定是空闲的，这就为进程p提供了一个很好的候选地点来运行。

当然，这个被选中的明显空闲的CPU完全有可能只是暂时空闲，而且必定会被一堆拥有更高优先级的、CPU绑定的进程所充满，这些进程可能在一纳秒后就会被唤醒，并在该CPU上运行。所以，这并不是完美的解决方法，可是从统计的角度来说，它已经相当好了——要记住，像这样的选择是很符合调度程序“快餐店式（quick-and-dirty）”的处理方式的。

26180: 建立本地变量。best\_cpu是此时正在运行的CPU；它是“最佳”的CPU，因为p在其上会避免缓冲区溢出或其他的开销麻烦。this\_cpu是运行reschedule\_idle\_slow的CPU。

26182: idle和tsk将沿task数组进行遍历，target\_tsk将是所找到的最后一个正在运行的空闲进程（或者假如没有空闲进程它就为NULL）。

26183: i从smp\_num\_cpus（前面被叫作n）开始，并且在每一次循环后都递减。

26189: 假如这个空闲进程的has\_cpu标志被设置，它就正在它的CPU上运行着（我们将称这样的CPU为“目标CPU”）。如果该标志没有被设置，那么目标CPU就正被某个其他进程占用着；因而，它也就不是空闲的，这样reschedule\_idle\_slow将不会把p发送到那里。刚刚提及问题的反面在这里出现了：现在仅因为CPU不空闲，并不能表示它所有的进程都不会死亡而使其空闲下来。可是reschedule\_idle\_slow无法知道这种情形，所以它最好还是假定目标CPU将要被占用一段时间。无论如何，这都是可能的，就算并非如此，某个其他的进程也将很快会被调度到另一个空闲CPU上运行。

26190: 不过假如CPU目标就是当前CPU，它就会被跳过。这看来很怪，无论怎样，这都是“不可能发生”的情况：一个空闲进程的counter是负值，在第26226行的测试将早已阻止这个函数执行到这一步了。

26192: 找到一个可用的空闲CPU; 相关的空闲进程被保存在target\_tsk中。

既然已找到了空闲CPU, 为什么现在不中断循环呢? 这是因为继续循环可能会发现p当前所在的处理器也是空闲的, 在两个CPU都空闲时, 维持在当前处理器上运行要比把它送往另一个好一些。

26193: 这一步reschedule\_idle\_slow检查是否p所在的处理器空闲。如果刚才找到的空闲CPU就是p所在的, 函数将向前跳转到send标号处(26203行)来在那个CPU上对p进行调度。

26199: 函数已经转向另一个CPU; 它要递减。

26204: 如果循环遍历了所有空闲的CPU, 该CPU的空闲任务就被标记为重新调度, 并且smp\_send\_reschedule(26205行)会给那个CPU发送一个IPI, 以便它可以重新对其进程进行调度。

正如读者所见到的, reschedule\_idle\_slow是CPU之间协调工作的典范示例, 而这些在UP系统中是无需进行的。对于UP机器来说, 询问进程应占有哪一个CPU和询问它是否应拥有系统的唯一的一个CPU或根本不应该占有CPU是等价的。SMP机器必须花费一些代价来决定系统中哪一个CPU是该进程的最佳栖身之所。当然, 换来了速度的极大提高, 这使得这些额外的努力还是相当合算的。

#### 5. release

22951: release中非SMP特有的部分在第7章中已经介绍过了——在这里, 一个僵进程将被送往坟墓, 而且其struct task\_struct将被释放。

22960: 查看是否该进程拥有一个CPU(拥有它的CPU可能还没有清除这个标志; 但是它马上就将执行这个操作)。如果没有, release退出循环, 并像往常一样接着释放struct task\_struct结构体。

22966: 否则, release等待进程的has\_cpu标志被清除。当它被清除后, release再次进行尝试。这种貌似奇特的情况——某进程正被删除, 然而它仍占有CPU——确实少见, 不过并非不可能。进程可能已经在一个CPU上被杀死, 而且这个CPU还没来得及清除has\_cpu标志, 但是它的父进程已经正在从另一个CPU对它进行释放了。

### 10.3.2 smp\_local\_timer\_interrupt

对于UP专有的update\_process\_times函数(27382行)来说, 这个函数就是它在SMP上的对应。该函数能够完成update\_process\_times所完成的所有任务——更新进程和内核在CPU使用方面的统计值以及其他的一些操作。与众不同的地方在于拥有这个特性的SMP版本并没有被添加到一个UP函数中去, 而是采用了一个具有同样功能, 但却完全分离的功能程序。在浏览了函数之后, 我们就能很容易知道这是为什么了——它与UP版本差别甚大, 以至于试图将二者融为一体将是无意义的。smp\_local\_timer\_interrupt可从两个地方进行调用:

- 从smp\_apic\_timer\_interrupt(5118行)调用, 它用于SMP的时钟中断。这是通过使用在第1856行定义的BUILD\_SMP\_TIMER\_INTERRUPT宏于第919行建立起来的。
- 从第5776行通常的UP时钟中断函数do\_timer\_interrupt里进行调用。只有当在UP机器上运行SMP内核时此种调用方式才会发生。

smp\_local\_timer\_interrupt

5059: `prof_counter` (4610行) 用于跟踪到更新进程和内核统计值之前, 内核应该等待多长时间; 如果该计数器还没有到达0, 控制流程会有效地跳转到函数的末尾。正如代码中所证明的, `prof_counter`项目从1开始递减计数, 除非由`root`来增加这个值, 因此在缺省情况下每次时钟滴答都要完成此项工作。然后, `prof_counter[cpu]`从`prof_multiplier[cpu]`处被重新初始化。

很明显, 这是一个优化的过程: 每次时钟滴答都在这个if语句块里完成, 所有工作将相当缓慢, 所以我们可能想到以牺牲一些精确度为代价将工作分批完成。因为乘法器是可调的, 所以你可以指定你所需要的速度频率来放松对准确度的要求。

然而, 关于这段代码我总感到有些困惑: 确定无疑的是, 当`prof_multiplier[cpu]`耗尽时, 统计值应该被更新, 就像`prof_multiplier[cpu]`的计数流逝一样——因为它们已经如此 (除了`prof_multiplier[cpu]`本身刚刚被改变时, 不过这已经偏离了这里讨论的主题)。与此不同的是, 这里代码表现出来的就好像只经过了一次滴答计数。或许其用意是为了以后能把记录下来的滴答数目和`prof_multiplier[cpu]`在某个地方相乘, 不过现在并没有这样实现。

5068: 当时钟中断被触发时, 假如系统正在用户模式运行, `smp_local_timer_interrupt`会假定全部滴答都是在用户模式里流逝的; 否则, 它将假定全部滴答是在系统模式里流逝的。

5073: 用`irq_enter` (1792行) 来夺取全局IRQ锁。这是我们要分批处理这项工作的另一个原因: 并不需要在每次时钟滴答时都要得到全局IRQ锁, 这有可能成为CPU之间争夺的一个重要根源, 实际中函数是以较低的频度来争取该锁的。因此, 函数不经常夺取这个锁, 可是一旦它获得了锁, 就不会再使其被锁。在此我们又一次以准确度为代价换来了效率上的提高。

5074: 不用为保存空闲进程的统计值而操心。这样做只会浪费CPU的周期。总之, 内核会跟踪系统处于空闲的总共时间, 对空闲进程的更多细节进行统计价值不大 (比如我们知道它们总是在系统模式下执行的, 所以就没有必要再明确计算它们的系统时间了)。

5075: `update_process_times`和`smp_local_timer_interrupt`在这一点上是一致的: 它们都调用`update_process_times`来完成对单进程CPU使用情况的统计的更新工作。

5077: 减少进程的`counter` (它的动态优先级), 如果它被耗尽就重新调度该进程。

5082: 更新内核的统计数字。如在`update_process_times`中一样, 用户时间既可以用内核的“最优时间”, 也可以用常规的用户时间来计算, 这要取决于进程的优先级是否低于`DEF_PRIORITY`。

5094: 重新初始化CPU的`prof_counter`, 并释放全局IRQ锁。该工作必须要以这种顺序完成, 当然——若以相反的方式, 则可能在`prof_counter`被重新初始化之前发生又一次时钟中断。

### 10.3.3 `lock_kernel`和`unlock_kernel`

这两个函数也有专门适应于x86平台的版本; 但是在这里只介绍通用版本。

### 1. lock\_kernel

10174: 这个函数相当简单、它获得全局内核锁——在任何一对lock\_kernel/unlock\_kernel函数里至多可以有一个CPU。显然这在UP机器上是一个空操作 (no-op)。

10176: 进程的lock\_depth成员初始化为-1 (参见24040行)。在它小于0时 (若小于0则恒为-1), 进程不拥有内核锁; 当大于或等于0时, 进程得到内核锁。

这样, 单个进程可以调用lock\_kernel, 然后在运行到unlock\_kernel之前可能又将调用另一个要使用lock\_kernel的函数。在这种情况下, 进程将立刻被赋予内核锁——而这正是我们所期望的。

其结果是, 一旦增加进程的lock\_depth, 就会使lock\_depth为0, 那么进程以前就是没有锁的。所以, 函数在此情形下获得kernel\_flag自旋锁 (3587行)。

### 2. unlock\_kernel

10182: 同样, 如果丢弃内核锁就会使lock\_depth低于0值, 进程退出它所进入的最后一对lock\_kernel/unlock\_kernel函数。此时, kernel\_flag锁一定要被解锁, 以便其他进程可以给内核加锁。通过测试结果的符号位 (即使用“<0”而不是“== -1”), 可以使gcc生成更高效的代码, 除此之外, 这还可能有利于内核在面对不配对的lock\_kernel/unlock\_kernel时可正确执行 (或者不能, 这取决于具体情况)。

## 10.3.4 softirq\_trylock

你也许能回忆起在第6章的讨论中, softirq\_trylock的作用是保证对于其他程序段来说下半部分代码 (bottom half) 是原子操作——也就是说, 保证在任何特定时段的整个系统范围之内至多只有一个下半部分代码在运行。对于UP来说这相当容易: 内核只不过需要检查或者还要设置一下标志位就可以了。但对于SMP来说自然没有这样简单。

### softirq\_trylock

12528: 测试并设置global\_bh\_count的第0位。尽管读者可能会从global\_bh\_count的名字上得到另外一种看法, 实际它总是0或者1——这样的考虑是适当的, 因为至多运行一个下半部分程序代码。不管怎样, 如果global\_bh\_count已经是1了, 那么就有一个下半部分代码在运行着, 因此控制流程就跳转到函数末尾。

12529: 如果还可得到global\_bh\_lock, 那么下半部分代码就能够在这个CPU上运行。这种情况与UP机器上使用的双锁系统非常类似。

12533: softirq\_trylock无法获取global\_bh\_lock, 因此它的工作失败了。

## 10.3.5 cli和sti

正如在第6章中解释过的, cli和sti分别用于禁止和启用中断。对于UP这简化为单个cli或sti指令。而在SMP情况下, 这就很不够了, 我们不仅需要禁止本地CPU, 还要暂时避免其他CPU处理IRQ。因此对于SMP, cli和sti宏就变成了对\_\_global\_cli和\_\_global\_sti函数的调用。

### 1. \_\_global\_cli

1220: 把CPU的EFLAGS寄存器复制到本地变量flags里。

1221: x86系统里的中断使能标志在EFLAGS寄存器的第9位——在第1205行解释了EFLAG\_IF\_SHIFT的定义。它被用来检测是否已经禁止了中断, 这样就不再需要



去禁止它们了。

1223: 禁止这个CPU的中断。

1224: 如果该CPU没有正在对IRQ进行处理, `__global_cli`就调用`get_irqlock` (1184行) 来获得全局IRQ锁。如果CPU已经在对IRQ进行了, 那么正如我们马上要看到的, 它已经拥有了该全局IRQ锁。

现在本CPU已经禁止了中断, 而且它也拥有了全局IRQ锁, 这样任务就完成了。

## 2. `__global_sti`

1233: 如果CPU没有正在对IRQ进行处理, `__global_sti`就调用`release_irqlock` (10752行) 来释放在`__global_cli`中获得的全局IRQ锁。如果CPU已经在对IRQ进行了, 那么它已经拥有了该全局IRQ锁, 正如在接下来的部分中将要解释的那样, 这个锁将在其他地方被释放掉。

1235: 再次允许在此CPU上进行中断。

### 10.3.6 `irq_enter`和`irq_exit`

第6章中顺便提及了这两个函数的UP版本。包含在一对`irq_enter/irq_exit`之中的代码段都是原子操作, 这不仅对于其他这样的代码区域是原子的, 而且对于`cli/sti`宏来说也是如此。

#### 1. `irq_enter`

1794: 调用`hardirq_enter` (10761行) 自动为此CPU增加全局IRQ计数和本地IRQ计数。这个函数记录了CPU正在处理一个IRQ的情况。

1795: 执行循环直到这个CPU得到全局IRQ锁为止。这就是为什么我要在前面说明如果CPU正在处理IRQ, 那么它就已经获得了全局IRQ锁的原因: 到这个函数退出时, 这两个特性都将被加强。对于内核代码来说, 把这两个特性分离出去并没有太大的意义——它可以直接调用`hardirq_enter`, 而且也不用去争夺全局IRQ锁。函数只是没有这样做而已。

#### 2. `irq_exit`

1802: 这个函数转向`hardirq_enter`的相反函数`hardirq_exit` (10767行)。顺便要提及的是, 对`irq_enter`和`irq_exit`来说, 其`irq`参数都被忽略了——至少在x86平台上是如此。

## 第11章 可调内核参数

遵循Unix的BSD 4.4版本所倡导的风格，Linux提供sysctl系统调用，以便在系统运行过程中对它所拥有的某些特性进行检查和重新配置，它并不需要编辑内核源代码、重新编译、重启机器。这是对早期Unix版本的一个十分重要的改进，在早期版本里调整系统经常是令人头痛的琐碎事务。Linux把可以被检查和重新配置的系统特性有机地组织成了几个种类：常规内核参数、虚拟内存参数、网络参数，等等。

同样的特性也可以从一个不同的接口进行访问：`/proc`文件系统（因为它真正的是系统的一个透视区（window）而不只是真实文件的一个容器，所以`/proc`是一个“伪的文件系统”，不过那是一个蹩脚的词汇，而且这个区别在此并不重要）。每种可调内核参数在`/proc/sys`下都表现为一个子目录，而每个单独的可调系统参数由某个子目录下的一个文件来代表。这些子目录可能又包含一级子目录，它们仍然含有更多的代表可调系统参数的文件和子目录等等，但是这种嵌套级数都不会很深。

`/proc/sys`绕过了通常的sysctl接口：一个可调内核参数的值可以简单地通过读取相应的文件来得到，通过写入该文件可以设置它的值。普通Unix文件系统的许可被应用于这些文件，以便控制能够对它们进行读写的用户。大多数文件对所有用户是可读的，但是只对root用户可写，不过也有例外：比如，`/proc/sys/vm`下的文件（虚拟内存参数）只能被root来读写。如果不使用`/proc/sys`，检查和调整系统将需要编写程序并使用必须的参数调用sysctl——虽然不是任务艰巨的劳动，可是也比不上使用`/proc/sys`来得方便。

struct ctl\_table

18274: 这是本章涉及的代码中所使用的一个主要数据结构。struct ctl\_table通常是由数组聚合起来的，每个这样的数组对应于`/proc/sys`下某处一个单独目录里的条目（依我之见，称它为struct ctl\_table\_entry可能更好）。root\_table（30328行）以及在它之后的数组通过struct ctl\_table的child指针连结节点而形成了一个数组树（child将在下边的列表中介绍）。注意所有这些都是ctl\_table的数组，它只是为struct ctl\_table进行typedef；18184行完成这项工作。

图11-1示意出了数组树间的关系。这幅图显示了由root\_table形成的树的一小部分及它所指向的树。

struct ctl\_table具有如下成员：

- ctl\_name——是唯一标识表项的一个整数，这个数在它所在的数组中是唯一的；这个数字在不同的数组中是可以重用的。数组的任何一项都已经存在这样一个唯一的数字了（就是它的数组下标），可是这个数字不能被用于该目的，因为我们想要维护不同内核发布版本中的二进制兼容性。与某内核版本里一个数组项相关联的可调内核参数，可能不会出现在将来的内核版本里，所以，假如参数是被它们的数组下标定义的，对数组里废弃项目位置的重新使用将使还没有在新内核版本下编译过的程序变得混乱。随着时间的推移，为了向后兼容而带上的元素项，只会浪费空间但没有作用的，还将会使数组变得

乱七八糟。相反的，这种方法只会“浪费”整数，而整数资源却无疑是非常丰富的（另一方面，查找也会更慢，因为一个简单的数组下标还不足以满足这种方法）。

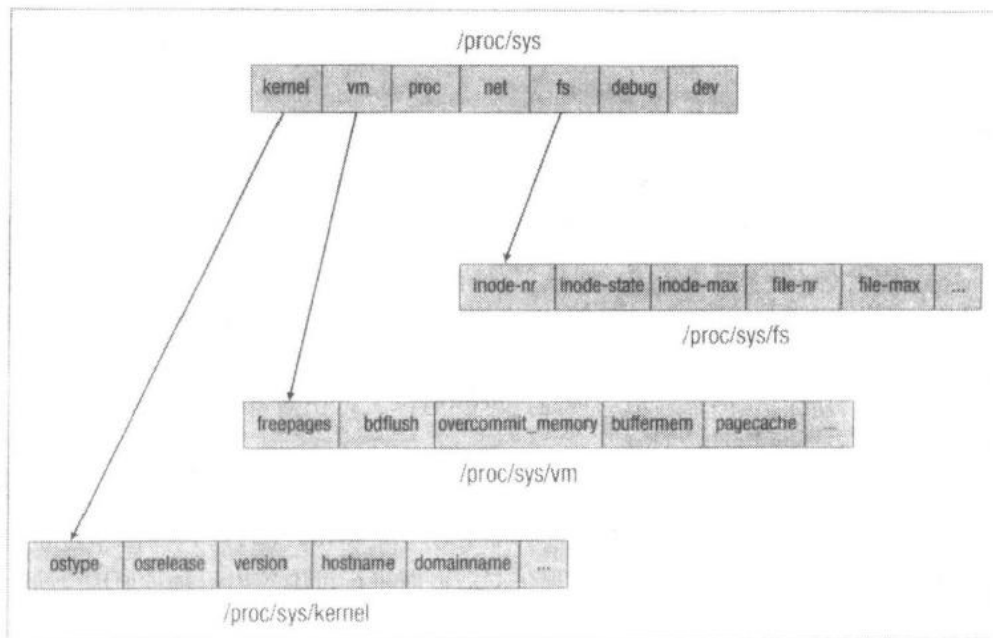


图11-1 struct ctl\_table 树的一部分

要注意的是这与有系统调用的情形相当类似：每个系统调用都与一个在系统调用表里唯一标识它位置的数字相关联。但是在这种情况下使用了一个不同的解决办法，可能由于速度在此并不重要的缘故。

- `procname`——是用于`/proc/sys`下的相应项的一个可供我们阅读的简短文件名。
- `data`——一个指向与此表项关联的数据的指针。它通常指向一个`int`或一个`char`（当然，指向`char`的指针是字符串）。
- `maxlen`——可以读取或者写入`data`的最大字节数。如果`data`指向一个单精度型的`int`，举例来说，`maxlen`就应该是`sizeof (int)`。
- `mode`——Unix类型的文件许可位，它对应于这一项的`/proc`文件（或目录）。对此的解释需要少量文件系统的内容。就像其他Unix的实现一样，Linux使用三个三位组，其中每一位都记录一个文件许可（在`ls -l`命令产生的列表里它们表现为`r`、`w`和`x`的三组字母）——参见图11-2。它们占据`mode`的低端9位。文件系统把文件的`mode`里剩余的位留作它用，比如用来跟踪文件是否是常规文件（第16位，当它如此时）、目录（第15位）、`setuid`或`setgid`执行程序（第12和11位），等等。不过就本章的目的来说，那些其他位都不是我们所关心的内容。

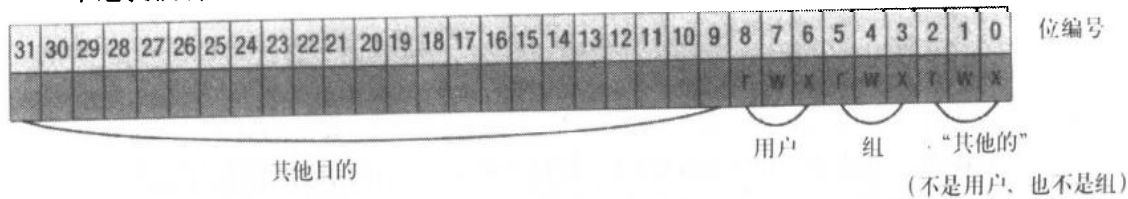


图11-2 文件的mode位

这种方式的结果是，读者将经常见到八进制的常数004、002和001与mode一起使用——它们分别是在移位mode后可能得到的适当的三位组中检测读（r）、写（w）和执行（x）位。这种移位和检查工作基本上是在30544行的test\_perm里完成的。

注意，如果一个表项的maxlen是0，那么不管它的mode是什么，从最终效果上看，它都是既不可读也不可写的。

- child——如果这是一个目录类型的条目，那么它就是指向子表的一个指针。在这样的情况下，因为没有数据与此条目相关联，data将是NULL，而maxlen则将是0。
- proc\_handler——指针，指向对data成员实际进行读取和写入操作的一个函数；它在通过/proc文件系统读写数据时被使用。以这种方法，任何类型的数据都可以通过data来指向，而且proc\_handler函数会正确处理对它的工作。proc\_handler通常指向proc\_dostring函数（30820行）或proc\_dointvec函数（30881行）；这两个及其他普遍适用的函数将在本章后面被讨论（当然，任何具有适当原型（prototype）的函数都可以使用）。对于目录类型的条目，proc\_handler是NULL。
- strategy——指针，指向对data成员实际进行读取和写入操作的另一个函数；它使用在通过sysctl系统调用进行读写的时候。它通常是sysctl\_string（31121行），不过也可以是sysctl\_intvec（31163行）；这两个函数在本章后面进行讨论。出于种种原因，大多数可调内核参数是通过/proc接口而不是sysctl系统调用进行调整的，所以这个指针是NULL会比非空更为常见。
- de——指向struct proc\_dir\_entry的一个指针，它在/proc文件系统代码中使用以追踪文件系统里的文件或目录。如果它非空，struct ctl\_table就在/proc下的某处注册过了。
- extra和extra2——指向在处理这个表元素时所需的任何补充数据。它们当前只用于指定某些整数参数的最小和最大值。

## 11.1 /proc/sys支持

不是所有实现用于可调内核参数/proc/sys接口的代码都包括在这本书中——的确，大部分代码并没有包括在内，因为它们基本上属于/proc文件系统本身。尽管如此，只要你不关心/proc剩下的部分是如何工作的，就不难理解在kernel/sysctl.c里的代码，它们与/proc文件系统一起工作，用来使/proc下的可调内核参数是可见的。

### 1. register\_proc\_table

30689: register\_proc\_table函数在/proc/sys下注册一个ctl\_table。注意这里并不要求所提供的表是根一级的节点（即ctl\_table没有父亲）——它本应该是，不过这取决于调用者是否能够进行保证。

这个表被直接建立在root之下，它应该对应于/proc/sys或者其下的一个子目录（在初次调用时，root总是指向proc\_sys\_root的，但是在递归调用时它的值改变了）。

30696: 开始在table数组的所有元素中进行循环；在当前元素的ctl\_name成员为0时循环结束，表示这是数组的末尾。

30698: 如果ctl\_table的procname域是NULL，那么即使同一数组的其他元素都可以为用户所见，它也不可以在/proc/sys下被用户所见。这样的数组元素会被跳过。

30701: 如果表项有procname，表明它应该在/proc/sys下被注册，那么它一定还有一个

`proc_handler` (如果是一个叶子, 或文件类型的节点) 或者一个`child` (如果是一个目录类型的节点)。如果它同时缺少这两者, 那么系统将显示一条警告, 然后循环继续进行。

- 30711: 若表项有一个`proc_handler`, 它被标记成常规文件。
- 30713: 否则, 正如可从第30701行推断的那样, 它一定有一个非空的`child`, 这样该条目将被看作是一个目录。注意并没有禁止`ctl_table`同时拥有非空`proc_handler`和`child`这两者——在这种情形下, 所有代码将对其一视同仁。
- 30715: 用给定的名字搜索一个存在的子目录, 如果找到就让`de`指向它, 如果没找到, 则`de`为`NULL`。为什么对文件不做类似的检查比较难于理解——这可能是我没有领会的文件系统的某个细节问题。
- 30723: 如果指定的子目录已经不存在了, 或者假如`table`对应于一个文件而不是一个目录, 新的文件或者目录就会通过调用`create_proc_entry` (未包含在本书中) 来创建。
- 30728: 如果表项是一个叶子节点, `register_proc_table`会告诉文件系统代码使用由`proc_sys_inode_operations` (30295行) 定义的文件操作。`proc_sys_inode_operations`只定义了两个操作, 读和写 (不是搜索、内存映射或者其他)。这些操作是用`proc_readsys`和`proc_writesys`函数 (30802和30808行) 来执行的, 在本章的后面章节中将对它们进行介绍。
- 30731: 到了这一行, `de`就不可能是`NULL`了——它或者已经非空, 或者在第30723行被初始化了。
- 30733: 如果增加的条目是目录类型, `register_proc_table`会被递归调用来增加这一项的所有子孙。这是内核里不多见的一次递归调用。

## 2. `unregister_proc_table`

- 30739: `unregister_proc_table`函数删除`ctl_table`数组树和`/proc`文件系统之间的关联。`ctl_table`里的条目以及它们下面所有的“子目录”里的条目也将会从`/proc/sys`消失。
- 30743: 同第30396行一样, 这一行开始在给定的表项数组上进行循环。
- 30744: 与`/proc/sys`下任意条目都不关联的表项具有一个为`NULL`的`de`成员; 显然这些表项可被忽略。
- 30748: 如果`/proc`文件系统认为这是一个目录, 但表项是一个叶子 (非目录), 这两个结构就是不一致的。`unregister_proc_table`就会显示一条警告并继续循环, 而不会移去这一项。
- 30752: 目录被逐层进行释放——内核中另一次并不多见的递归过程。
- 30756: 在递归调用结束之后, `unregister_proc_table`检查是否所有子目录和文件都被逐层删除了——如果不是, 当前元素就不能被安全移去, 接着要继续循环。
- 30762: 这里就是为什么子目录 (以及其中的文件) 可能还没有被移去的原因: 它们可能当前还正被使用着。如果这个元素正在被使用, 循环将继续, 这样该元素就不会被移走。
- 30765: 节点通过`proc_unregister` (本书不介绍) 从文件系统里被删除, 接着用于追踪该节点而分配的内存被释放。

## 3. `do_rw_proc`

- 30771: `do_rw_proc`实现`proc_readsys` (30802行)和`proc_writesys` (30806行)函数的核心部分,这两个函数被`/proc`文件系统代码用于对`ctl_table`执行读取和写入操作。
- 30782: 确保一个表与`/proc/sys`下的这一条目相关联。
- 30785: 注意这一行的第一个测试与第30782行的第二个测试是相重复的,这是因为`table`是从`de->data`初始而来。
- 30788: 确保调用进程有适当的读或写权限。
- 30795: 调用该表项的`proc_handler`来完成真正的读操作或写操作(要注意第30785行证实了`proc_handler`成员是非空的)。如前所述,`proc_handler`成员通常是`proc_dostring`或`proc_dointvec` (30820行和30792行),在随后的几段中我们将对它们进行讨论。
- 30799: `do_rw_proc`返回实际读取或写入的字节数。注意到本地变量`res`完全是多余的;它可以被参数`count`所替代。

#### 4. `proc_dostring`

- 30820: `proc_dostring`是供`/proc`文件系统代码调用以对C语言字符串型的内核参数进行读取或写入操作的函数。

注意`write`标志表示调用者正在写表元素,不过这主要是涉及从输入缓冲区里进行读取——因此,用来写入的代码是受读控制的。类似的,如果`write`未被设置,调用者正从该表项读取,这里主要涉及的是写入给定的缓冲区。

这个函数在第31085行还实现了一个存根程序;这个存根程序在`/proc`文件系统被编译出内核时使用。大多数其他函数中的类似存根程序将在这个存根程序之后被介绍。

- 30835: 从输入缓冲区内读取字符直到一个表示结束的ASCII NUL (0) 字节或发现新的一行,又或者到达了被允许从该输入缓冲区内读出数据的最大值(被`lenp`所指定)为止。(为了不引起混淆,牢记NULL是一个C指针常量,而NUL——只有一个L——是ASCII用于字符数字0的术语。)
- 30842: 如果从缓冲区读出的字符数超出了可在表项里存储的限度,该数目会被降低。在循环之前就限制最大输入长度(`lenp`)可能会更高效,因为不管怎样从`buffer`里读取大于`table->maxlen`字节的数据是无意义的。实际上,循环可能读出,假设是1024字节,然后降低计数到64,因为表项里只能存储这么多。
- 30844: 该字符串从输入缓冲区里被读出,然后以NUL结束。
- 30847: 内核为每个进程所拥有的每个文件维护一个“当前位置”变量;这就是`struct file`的`f_pos`成员。它是`tell`系统调用返回的值,并由`seek`系统调用进行设置。因此,文件的当前位置是由写入的字节数所推进的。

#### 5. `proc_doutsstring`

- 30871: 在获得`uts_sem`信号量后(29975行),`proc_doutsstring`仅是调用`proc_dostring`。这个函数被`kern_table` (30341行)里的一些条目用来设置`system_utsname`结构体的不同部分(20094行)。

#### 6. `do_proc_dointvec`

- 30881: `proc_dointvec` (30972行)把它的工作委托给了该函数。`do_proc_dointvec`读或写

一个被table的data成员所指向的int类型数组。要读写的int类型数目通过lenp传递；它通常是1，所以本函数通常只被用于读写单独一个int。

用于int的值是被buffer指定的。这些int是会被以一个未经加工的int数组传递的；相反，它们以ASCII文本给出，而这正是用户写入相关/proc文件的。

30898: 在所有要读写的int中循环。left追踪调用者想要读写int的剩余数目，而vleft追踪table->data里剩余的有效元素数目。在这二者中任何一个到达0，或它从半途退出时，该循环结束。

注意，如果从循环中去掉第30899行的if语句，可以使整个循环的效率稍微提高一些，尽管这样做的结果较难维护。取代的代码如下：

```
for (; left && vleft--; i++, first=0)
{
 if (write) {
 /* Write stuff. */
 } else {
 /* Read stuff. */
 }
}
```

它可以构造为如下这样：

```
if (write) {
 for (; left && vleft--; i++,
first=0) {
 /* Write stuff. */
 }
} else {
 for (; left && vleft--; i++,
first=0) {
 /* Read stuff. */
 }
}
```

这种方式使得并不在循环内改变的write的值将只需检查一次，而不必在每次循环重复检查。

30900: 向前搜索一个不是空格的字符，它是输入（缓冲区）里下一个数字的开头。

30913: 从用户空间把一大块数据复制到本地缓冲区buf，然后以NUL结束buf。现在buf里包含了所有输入缓冲区里剩余的ASCII文本——或者是它所能容纳的那些文本。

这种方法看起来不很有效率，原因在于它读取的可能超出了它所需要的。然而，因为buf的容量仅为20（TMPBUFLen，30885行），它就不可能读取比它所需多出许多的数据。这里的想法可能是读入稍多一些数据要比检查每个字节以确定是否应该停止读操作所付出的代价要少些。

计划使buf足够大来包括任何64位整数的ASCII表示，以便这个函数不仅可以支持32位平台，还可以支持64位平台。的确，它只能满足最大的正64位整数，它有19个位数（终结的NUL字节是第20个字节）。可是要记住这些是有符号的整数，所以最小的64位有符号整数，即-9 223 372 036 854 775 808也应是合法输入。这个数字无法被正确读取。但是幸运的是，补救方法工作量不大，而且也非常明显。

随后读者就能够看到，当这个输入出现时代码将如何对其进行处理。

- 30919: 处理打头的减号 (-)。如果发现一个减号就跳过它并设置一个标志。
- 30923: 确保从buffer读取的文本(可能是打头的减号之后的部分)至少是以一个数字开始的,这样它才能顺利转换为一个整数。若没有这次检查,就不可能分辨出第30925行调用simple\_strtoul返回的0,是因为输入就是“0”,还是因为函数无法转换任何文本。
- 30925: 把文本转换为一个整数,用conv参数换算结果。这个换算步骤对于proc\_dointvec\_jiffies这样的函数(31077行)比较有用,它用乘以常数HZ的简单手段把它的输入从秒转换为一段时间值(jiffies)。然而一般情况下,这个比例因子是1——即没有换算。
- 30927: 如果还要从缓冲区读取更多的文本,而且下一个要读的字符不是分割参数的空格,那么整个参数就无法装进buf。这样的输入是无效的,所以循环提早结束(一种可以导致函数处于这种状态的方式就是前边所描述的,输入表示的是最小的有符号64位整数)。不过,没有错误代码会被返回,因此调用者可能会错误地认为一切正常。当然这也不完全正确:一个错误代码将在第31070行被返回,不过这仅当无效参数是在第一次循环重复中被检测到的时候;如果它在后续的循环里被检测到,错误就不会被注意到。
- 30929: 参数被成功地读取。如果有前导的减号,那么现在就对它进行考虑,其他的本地变量被调整转移到下一个参数上,然后这个参数通过指针i被存储在表项中。
- 30936: 调用者从表项里读取值——由于无需对ASCII文本进行语法分析,这就是一种更为简单的情形。输出是由tab(制表符)分隔的,所以在除了第一次之外的任何一次循环里都把一个tab写入临时缓冲区里(在最后一个参数之后不用写,只需在参数之间即可)。
- 30938: 接着,当前的整数被conv因子按比例缩减并打印到临时缓冲区里。这段代码同样会受读者前面已经见到的问题的损害:临时缓冲区buf的大小可能不足以容纳打印到它里边的全部整数值。在这种情况下,实际问题还会因缓冲区的第一个位置可以是一个tab制表符而被恶化。这会使buf的可用部分减少一个字符,进一步还会降低可被正确处理的输入范围。

在这里过大或过小的整数所造成的结果要比写入情形里严重得多。在那种情形中,代码只要抛弃一些本应接受的输入即可。而在这儿,sprintf会越过buf的末尾继续写下去。

然而令人惊讶的是,这正是实际工作中可能发生的。在一次典型的执行过程中将有可能发生如下执行过程:从总体上来说,超过buf的末尾之后还要写入两个额外的字节(一个是因为它可以写入比预期更长的数字,另一个是tab制表符)。在栈里p通常是紧跟在buf之后的,所以超出buf末尾写入的部分将会覆盖p。可是由于p没有先被重新初始化时它是不会再被使用的,因此暂时覆盖它的值并没有危害。

这是一个看似有趣的故事,但是仅仅通过使buf稍微大一些就能够成为一个更好的解决方式,这样便于代码为正确的前提(reason)而工作。依照原样,对于gcc的代码生成器进行完全合法的很小的修改就能够揭示出潜在



的缺陷。

- 30939: 把当前int的文本型表示复制进输出缓冲区里——或者和它所能容纳的相等的文本——接着更新本地变量使其转移到表项的下一个数组元素。
- 30949: 如果调用者刚才在读取、输出就被新的一行结束。if条件语句也保证循环不会在其第一遍执行而且还有空间来写入新行时就结束。注意输出缓冲区不是用ASCII NUL字节（读者可能会这样猜测）来结束的，因为它无需如此：调用者能够利用lenp写入新值来减少返回字符串的长度。
- 30954: 如果调用者正向表项里写入数值，则略过从输入缓冲区读取的最后参数之后所有的空格。
- 30967: 更新文件的当前位置和lenp，然后返回0表示成功。

#### 7. proc\_dointvec\_minmax

- 30978: proc\_dointvec\_minmax函数类似于do\_proc\_dointvec，区别是这个函数还把表项的extra1和extra2成员作为可以写入该表项的限制值数组来处理。extra1里的值是最小限度，而extra2里的值则是最大限度。另一点区别是proc\_dointvec\_minmax不使用conv参数。

因为这两个函数颇为相似，所以这一段里只介绍其不同之处。

- 31033: 最大的区别在于：当写入时，超过被min和max（在extra1和extra2数组上循环得到）所定义的范围之外的值将悄无声息地被略过。这段代码的目的明显是要使min和max伴随着val一起继续。当一个数值从输入缓冲区里被读取时，它应该被下一个min和max来检查，然后才能决定被接受或被忽略。可是，这并非是实际所发生的那样。

假设从buffer而来的当前值已经进行了语法分析并存入val里，它小于最小值；为了更具体一些，再假设已是第三遍循环，以便min和max分别指向对应数组中的第三个元素。然后val将用min来检查并发现它超出了范围（太小），接着循环还要继续。可是min会作为检查的副作用被更新，而max则没有。现在，min指向它对应数组的第四个元素了，可是max仍然指向它的数组的第三个元素。这两者不再同步，而且它们还将保存这种状态，这样，在下一个从buffer里读取的值被检验时，采用的就是错误的界限。下列代码是最简单的一种修补程序：

```
if (min && val < *min++) {
 ++max; /* Keep max & min in sync. */
 continue;
}
if (max && val > *max++)
 continue;
```

正如读者将要在本章后面看到的，现在的Linux源代码永远不会暴露出这个缺陷。（未来发行的版本情况将有所不同，尽管还未曾明确写出。）

## 11.2 sysctl系统调用

用于可调内核参数的另一个接口是sysctl系统调用及相关函数。我不很喜欢这个接口。为什么？对于大部分实际工作目的来说，使用sysctl——不过，用这种方法来调整内核比用

修改源代码的旧方法能够获得更大的性能提高——只会比访问/proc文件更为笨拙。通过sysctl来进行读写需要C程序（或相似的东西），而/proc却很容易通过shell命令（或等价地通过命令解释程序脚本）来进行访问。

另一方面，如果你正在C环境下工作，调用sysctl就比打开文件、读取并（或）写入以及再关闭它要方便得多，所以，sysctl在今后也有它的用武之地。与此同时，还是让我们来看一看它的实现吧。

### 1. do\_sysctl

30471: do\_sysctl实现sys\_sysctl（30504行），即sysctl系统调用的主要内容。注意sys\_sysctl还在第31275行出现过——那个版本只是在sysctl系统调用被编译出内核时所使用的一个简单的存根程序函数。

如果oldval非空就用oldval返回内核参数原有的值，而它的新值在newval非空时从newval来进行设置。oldlenp和newlen相应的指针不是NULL的时候，分别标识出有多少字节应被写入oldval和从newval读出，当指针为NULL的时候，它们将被忽略。

要注意这里的不对称性：函数对旧值的长度使用指针，而对新的长度不使用指针。这是因为旧的长度既是输入参数，也是输出参数；它的输入值是通过oldval返回的最大字节数，而它的输出值是实际返回的字节数。而新的长度只是一个输入参数。

30482: 如果调用者需要旧的内核参数值，从oldlenp来对old\_len进行设置。

30490: 开始遍历表树的循环列表（参见本章随后对register\_sysctl\_table的讨论）。

30493: 使用parse\_table（30560行，在下一段里讨论）来定位可调内核参数，然后读和（或）写它的值。

30495: 如果parse\_table分配了所有环境信息，它就被释放。很难准确地说出这个环境信息表示着什么。它不被本书所讨论的任何代码使用——据我所知，它目前甚至没有被内核里的任何代码所使用。

30497: ENOTDIR错误表示没有在这一棵表树中找到指定的内核参数——它可能在另一棵还没有查找过的表树中。否则，error将为某个其他的错误代码，或者是代表成功的0；无论如何，函数应该返回了。

30499: 用DLIST\_NEXT宏（本书对此不做介绍）来增加循环控制变量的值。

30501: 返回ENOTDIR错误，报告出指定的内核参数在任何一个表里都未找到。

### 2. parse\_table

30560: parse\_table用于在表树里查找一个条目，其方法类同于在一个目录树里解析出一个完全合格的文件名的方法。其思想如下：沿着一个int数组（数组name）进行查找，并在一个ctl\_table数组里搜索每个int。当找到一个匹配项时，它对应的子孙表就被递归查阅（如果匹配项是目录类型的条目），或者该条目被读和（或）写（如果它是文件类型的条目）。

30566: 多少有些令人惊讶的是，这一行就开始了对于整型数组name内所有元素的循环。习惯上的方法原本是把从这一行到第30597行所有代码用一个for循环包括起来，它的开始是这样的：

```
for (; nlen; ++name, --nlen, table =
 table->child)
```

(这个循环还需要删除第30567和30568行代码，并用一个break语句来替代从30587到30590行的代码。)推测起来，可能是实际使用的版本可以生成更好的目标代码吧。

- 30570: 开始循环所有的表项，查找与当前name匹配的一项；当表已被遍历结束（table->ctl\_name为0了）或者指定的表项已被找到并处理时，本循环结束。
- 30572: 把name数组的当前项读入n里，以便它可以与当前表项的ctl\_name进行检查。因为name在内层循环中没有变化，这个读取操作可以放在循环外边（也就是移至30569行）以提高一点速度。
- 30574: 核查当前ctl\_table的名字是否与被找到的名字相匹配，或是否它有特殊的“通配符”值，即CTL\_ANY（17761行）。后者的使用目的还不清楚，因为现在并没有内核源代码的任何一处使用过CTL\_ANY。它可能用于将来的方案中——我不认为它是过去版本的一个遗留物，因为CTL\_ANY在2.0内核里也没有被用到，而且整个sysctl接口也只向后兼容到2.0以前的开发树版本。
- 30576: 如果这个表元素有一个孩子，它就是一个“目录”。
- 30577: 遵循标准Unix行为，检查目录的x（可执行）位来判断当前进程是否被允许可对它进行访问。注意到这与文件系统所实现的工作非常类似，虽然这并不是（/proc）文件系统接口。这样可以使这两种接口在施用于可调内核参数时能够得到一致的结果——如果一个用户有通过一种接口来修改某个内核参数的权限，而通过另一种却没有该权限，那么将是非常不可思议的。
- 30579: 如果表项有一个策略函数，它可能需要覆盖允许该进程进入目录的授权。这个策略函数将被访问，如果它返回一个非零值，整个查找就被中止。
- 30587: 进入目录。本行有效地继续外层循环，并转移到该名字的下一部分。
- 30592: 这个表节点是一个叶子节点，因此内核参数就被找到了。注意这并不打扰对name数组是否已到其最后元素的检查（也就是现在nlen是否为1），虽然可以证明，如果不是那样就会有某类型错误产生。不管哪一种情况，do\_sysctl\_strategy（30603行）都要负责对当前表元素进行读和（或）写操作。
- 30598: name数组非空，可是它的元素在叶子节点被找到之前均已用完。parse\_table就返回ENOTDIR错误来表示查找指定节点失败。顺便提及一点，注意前一行里的分号是多余的。

### 3. do\_sysctl\_strategy

- 30603: do\_sysctl\_strategy在单独一个ctl\_table里读和（或）写数据。计划使用该表元素里的strategy成员（如果存在的话）来完成读/写工作。如果表元素没有它自己的strategy例程，某些通用的读/写代码将被替代使用。不过读者将要看到，它并不完全按照计划工作。
- 30610: 如果oldval非空，调用者将读取旧值，这样r位就会在op里被设置。同样，如果newval非空，则w位被设置。接着，第30614行核查许可，如果当前进程缺少所需的授权就返回EPERM错误。
- 30617: 如果表项有它自己的strategy例程，这个例程就要处理读/写请求。如果它返回负

数——一个错误，这个错误就被传送给调用者。如果返回的是正数，0（成功）就会被传送给调用者。如果是0，strategy例程就拒绝由它自己来处理请求，取而代之的将是缺省行为（读者可以设想只返回0的strategy例程，如果它完成一些其他诸如收集被调用次数的这样的统计数据工作，它仍然是有用处的）。

30630: 这里是通用读取代码开始的地方。注意get\_user（13254行）的返回值不被检查（类似的缺陷发生在第9537和31186行）。

30632: 确保不会有多于与该表项的maxlen域所指定的数值相等的数据被返回。

30634: 通过oldval从表里复制所要求的数据，再将真正被写的数据总量存储在oldlenp中。

30642: 类似于oldlenp，要确保写入表项的数据不能多于它的maxlen成员所允许的值。注意，如果copy\_from\_user在中途的第30644行检测到一个错误，tabel->data可能会在仅仅被部分更新的情况下就结束。

30648: 返回0表示成功。以下三种情况都可以达到这一点：

- 调用者对这个表项既不读也不写。
- 调用者尝试读和（或）写这个表项，而且所有步骤都被成功执行。
- 表项没有关联的数据，或者因为它的maxlen是0，所以它是只读的。

三种情形中的第一种有点儿奇怪，而最后一种则更令人奇怪。第一种情况有些奇特是因为调用sysctl却要求它对指定的表项既不读也不写，这并没有多少意义，所以，可以合法地把它当作一个错误来处理。尽管如此，它要与其他系统调用的内核实现保持基本一致，那就是把一个无操作请求不看作是一个错误。比如说，在第8章中介绍的sys\_brk（33155行）在由调用者指定的新brk值与旧值相同时并不产生一个错误信号。

第三种情况要比第一种奇怪一些，因为它可能真的反映着一个错误。例如，调用代码尝试写入一个maxlen是0的参数，而且由于系统调用返回成功值而认为该尝试已被完成。看起来事情好像不是这样，因为不管怎样，为0的maxlen都会使该条目失效，不过还真的存在一个maxlen为0的表项——参见第30380行。最终，一切都归结为sysctl是怎样在文档中描述的，但是man帮助程序中却对此没有任何记载。我仍然认为do\_sysctl\_strategy在这种情况下应该返回一个EPERM错误。

#### 4. register\_sysctl\_table

30651: 把已给出根的ctl\_table新树插入到其他树所形成的循环链表里。

30655: 分配一个struct ctl\_table\_header来管理新树的信息。

30659: 把新的首部（跟踪ctl\_tables数组形成的新树）插入到首部组成的链表里。

30666: 调用register\_proc\_table（30689行，本章前面讨论过）把新的表树注册在/proc/sys目录下。如果内核在没有/proc文件系统支持的情况下进行编译时，则这一行将被编译到内核以外。

30668: 新分配的首部被返回给调用程序，以便调用程序能够在以后通过把该首部传递给unregister\_sysctl\_table（30672行）来删除相应的树。

#### 5. unregister\_sysctl\_table

30672: 如前所述，这个简单函数只是把一个ctl\_table的树从内核里这样的树所组成的循环链表里删除。如果内核是在支持/proc的情况下编译的，它也用于从/proc文件系

统里删除相应的数据。

回顾一下第30490和30500行，读者不难发现`root_table_header`（30256行）——对应于`root_table`的列表节点——是在遍历树的循环链表时被用作头和尾节点的。读者现在能够明白实际上在`unregister_sysctl_table`函数里没有什么可以避免`root_table_header`被从表头列表里删除——它只是还没有这样做而已。

## 6. sysctl\_string

31121: `sysctl_string`是`ctl_table`的策略例程之一。回忆一下，策略例程可以从第30618行（在`do_sysctl_strategy`里）被调用来选择地覆盖一个表项的缺省读/写代码（策略例程也可以从第30580行被调用，不过该例程却从不会被调用）。

31127: 如果该表没有相关数据，或者如果可访问部分的长度是0，则返回`ENOTDIR`错误。这与`do_sysctl_strategy`的做法是不一致的，在同样的情况里它返回的是成功。

31138: 当前字符串的值被复制到用户空间，然后结果以NUL来结束（这意味着比由`lenp`指定值多一个字节的数据可能会被复制——依据文档记录，这可能是一个缺陷）。因为当前值已经是NUL结束的，这四行代码可以被简化为两行：

```
if (copy_to_user(oldval, table->data,
 len + 1))
 return -EFAULT;
```

这种改变的正确性有些依赖于当写入`table->data`时代码剩余部分所遵循的三个特征：

- 代码剩余部分不能把多于`table->maxlenchar`数据复制进`table->data`里。（这也使得第31136行的测试变得没有必要。即使还需要该测试，那也只用检查“>”，而不用检查“>=”了。）
- 然后`table->data`以NUL来结束，如果必要就复写最后一个拷贝进来的字节，以便包括NUL在内的总长度不大于`table->maxlen`。
- `table->maxlen`永不发生变化。

因为所有三个特征都有效，所以在第31138行`len`将总是严格小于`table->maxlen`，而且结束NUL字节一定会在`table->data[len+1]`或之前的位置出现。

31146: 与前面的情况类似，从用户空间中复制新值，而且结果以NUL来结束。不过在这种情形下，不从用户空间复制NUL字节是一种正确的做法，因为把它从用户空间复制进来要比仅仅在`data`的适当字节安排一个NUL效率低。而且以这种方式，即使输入不是NUL结束的，`table->data`也要如此。当然，从`newval`读出的字符串可能已经是NUL结束的，在那种情况里，第31154行的赋值就是多余的。这还有另一种情况，即直接完成工作比检查是否需要执行它还要快。

31156: 返回0表示成功。相反，返回的值应该为正数，以便30618行代码认为结果是成功的。而又相反，调用代码认为`sysctl_string`想让缺省处理发生，然后它就继续从用户空间再次复制多余的数据。

## 7. sysctl\_intvec

31163: `sysctl_intvec`是在`kernel/sysctl.c`里定义的另一个策略例程。它确保假如调用程序正在写入表项，所有被写的`int`都应位于某个最小和最大值之间。（顺便提及一下，`sysctl_intvec`在这个文件里只被使用了一次——在第30414行，尽管它被广泛的用于本书所没有包括的内核的其他代码之中。）

- 31170: 如果新的欲写数据总量不符合一个int大小的边界, 它就无效, 所以尝试被抛弃。
- 31173: 假如表项没有指定一组最大或最小值, 输入的值就永远不可能超出范围, 这样调用程序里的普通写代码 (do\_sysctl\_strategy, 30603行) 就足够好了。因此在这种情况下里sysctl\_intvec返回0。
- 31184: 进行循环以确保所有来自输入数组的值都位于适当范围之内。
- 31186: 这行代码不检查get\_user的返回值——没有迫切的需要去这样做。当不能读取一个输入内存位置时, sysctl\_intvec返回0 (成功), 那么当它试图读取整个数组时, do\_sysctl\_strategy就会注意到这个问题。作为另一选择, 假如get\_user无法读取内存位置, 无用信息可能在value里结束, 并且数值可能会不正确地被抛弃。在此情况里, 调用程序将得到一个EINVAL错误, 而不是EFAULT错误, 这只是一个小缺陷。
- 31187: 注意这一行不会被折磨第31033行代码的缺陷所困扰, 该行中在最小和最大数组之上进行的并行循环会产生不同步的情况。  
这一行代码能够避免位于31033行的缺陷暴露出来。正如实际中所进行的, sysctl\_intvec和proc\_dointvec\_minmax都总是与同一个ctl\_table项相关联的。因此, 在调用处理例程proc\_dointvec\_minmax之前, 任何超出允许范围之外的数值将会被策略例程sysctl\_intvec截获。所以, 我们知道——在给定内核里所有的ctl\_table最新定义的情况下——proc\_dointvec\_minmax将永远不会遇到超出界限的数值, 而那里是唯一可以触发该缺陷的数值种类。某个调用程序或许可以注册一个使用proc\_dointvec\_minmax但没有策略例程的ctl\_table, 尽管这样, 在proc\_dointvec\_minmax里的这个缺陷迟早会造成一定损害。
- 31193: 返回0表示成功。这里不像在第31156行那样是一个错误, 因为sysctl\_intvec并不向table->data里写入。从用户空间读出的值只是被读进一个临时变量里做范围检查, 然后就被删除; do\_sysctl\_strategy将完成那项工作, 并只向table->data里进行写入。

## 附录A Linux 2.4

内核开发人员并没有因为我在写这本关于Linux的书而暂停他们的工作。说真的，内核的发展进展神速，就在这本书即将出版之际，它的最新稳定版本2.4.0，也应该面世了。尽管在本书中将对内核的成千种修改都一一涵盖到是不可能的，但是这篇附录却总结了发生在本书所涉及内核部分的最令人感兴趣的改变。这些修改中的绝大部分已经在内核的2.3.12版本里实现了，它们被包括在附赠的CD-ROM上。

### A.1 更少的“惊扰

请读者回忆一下在第2章里，函数`--wake_up`（26829行）唤醒等待在等待队列上的所有进程。可以考虑一个诸如Apache一样的Web服务器，它试图通过在同一端口派生出许多进程监听连接申请以缩短响应时间（正好与等待连接请求到达，并且与只通过fork派生一个进程来响应的方式相反）。所有这些进程都在一个等待队列里，等待连接请求到达，而当请求确实到达的时候，它们都会被唤醒。它们当中只有一个能够为请求提供服务，所以剩下的又都将返回休眠状态。

这种“惊扰（stampede，原意是指：动物受到惊吓后纷纷逃窜。作者使用该词说明：在等待队列中休眠的进程同时被唤醒的情况。——译者注）”现象会浪费CPU的时钟周期——只唤醒一个等待进程会更好，因为，不管怎样只有一个能够运行。因此，一个新的任务状态位`TASK_EXCLUSIVE`就被引进；在一次调用`--wake_up`里，最多有一个设置了`TASK_EXCLUSIVE`位的任务会被唤醒。`TASK_EXCLUSIVE`位是附加在其他任务状态位上的——它并不代表一个新的任务状态，它只是为了方便而保存在任务状态信息组里的信息而已。

现在`--wake_up`要检查它正在唤醒的进程是否设置了`TASK_EXCLUSIVE`位，并且在它唤醒该进程之后便不再唤醒其他进程（通过使用break中断循环）。`TASK_EXCLUSIVE`位现在只被用在有关网络的等待队列上，而且它在那种环境中效果也的确很好。对于大多数其他等待队列来说，你需要所有进程对等待着的资源都有机会进行占用，这样可以避免饥饿现象。不过等待在同一个端口的服务程序通常与Apache境遇相同：所有等待的任务都是一样的，而且重要的是只要它们中有一个能够处理请求就可以，即使每次都是同一个任务（也无所谓）。

### A.2 再见吧，Java

正如第7章里所预言的那样，Java二进制处理程序已经从内核里消失了，它是因被杂项二进制处理程序所替代而失效的。二元处理程序的常规用法没有变化，Java执行体仍然可被杂项二进制处理程序的适当配置所完全支持。

### A.3 ELF权能位

已有非官方的修补程序可用于（增强）ELF执行体中权能的存储。它们还没有成为正式内核版本的一部分，部分原因在于，ELF文件头是否是保存这些信息的恰当位置，这个问题还

处于讨论之中。不过，这些修补程序也有可能已经成为正式版本的一部分了。

## A.4 调度程序提速

已被高度优化了的schedule函数（26686行）又被做了进一步优化。大多数修改只是在system\_call（171行）系列调用的基础上进行了重新组织——也就是说，通过允许普通情况直接通过以及把遍布在函数里的大部分if语句体都分散开来的方式提高代码的（运行）速度。举例来说，第26706和26707行，如果又要运行的代码它们就运行底下的下半部分，现在采用的是这种形式：

```
 if (bh_mask & bh_active)
 goto handle_bh;
handle_bh_back:
 /* ... */
handle_bh:
 do_bottom_half();
 goto handle_bh_back;
```

这样一来，如果下半部分程序必须运行，控制流程就会跳转到新的handle\_bh标号处，然后在运行完下半部分之后再跳转回去。原有方式在无需运行下半部分的正常情况下也要产生一个分支转移，因为在那种情况下所产生的代码不得不跳过对handle\_bottom\_halves的调用。在新的版本里，正常情况只需直接通过，不会产生任何分支。

## A.5 更多的进程

Linux 2.4几乎消除了对同时可以运行的进程数目的固有限制。唯一剩下来的硬编码限制就是PID的最大数目了（不要忘记PID可被共享）。这个改进使得Linux能够更好的适合于高端服务器应用程序，包括经常需要同时运行大量进程的Web服务器。

在第8章中说明过，原有4090个任务的最高限度是受可以同时保存在全局描述符表（GDT）里的任务状态段（TSS）和局部描述符表（LDT）的最大数目所影响的：Linux 2.2在GDT里为每个进程存储一个TSS和一个LDT，而且GDT总共允许有8192个条目。Linux 2.4通过存储TSS和LDT本身而不是把它们存储在GDT里的方法回避了这个限制。现在，GDT只保持每CPU一个TSS和一个LDT，而且把这些条目设置为在每个CPU上的当前运行任务所需要的信息。

虽然用于追踪每个CPU的空闲进程的init\_tasks数组还存在，但是task数组（26150行）现在却没有了。

## A.6 日益进步的SMP支持

每一个锁都是一个SMP机器可以被有效地转化为UP机器的地方：正等待一个锁的CPU对系统的整体性能没有任何贡献。因此，改善SMP性能的主要方法就是通过减小锁的作用域或者完全消除它们来增加并行程序的运行机会。

Linux 2.0只有一个单独的全局内核锁，所以每次只有一个CPU可以在内核里执行。Linux 2.2把这个全局内核锁使用的大部分地方都用更小的、子系统专用的锁来替代了，其中的一些我们在第10章中已经见到过。Linux 2.4继续了这个变化，它把可能减小其作用域的锁都做了进一步的分割。比如现在每个等待队列就有一个锁，而不是所有的等待队列才有唯一的一个锁。



## 附录B GNU通用公共许可证

1991年6月, 第2版

版权所有 (C) 1989, 1991 Free Software Foundation, Inc,  
Place, Suite 330, Boston, MA 02111-1307 USA

59 Temple

任何人都可以复制和发布这一许可证原始文档的副本, 但绝对不允许对它进行任何修改。

### 序言

大多数软件许可证都有意剥夺你的共享和修改软件的自由。相对之下, GNU通用公共许可证力图保证你的共享和修改自由软件的自由——保证自由软件对所有用户都是自由的。GPL适用于大多数自由软件基金会的软件以及由愿意遵守该许可证规定的作者所开发的软件(自由软件基金会的其他一些软件受GNU库通用许可证的保护)。你也可以将它应用到你的程序中。

当我们谈到自由软件时, 指的是自由而不是价格。GNU通用公共许可证的目的是要保证你有发布自由软件的自由(如果你愿意, 你也可以对此项服务收取一定的费用); 保证你能收到源程序或者在你需要时能得到它; 保证你能修改软件或将它的一部分用于新的自由软件; 而且还保证你知道你能做这些事情。

为了保护你的权利, 我们需要作出规定: 禁止任何人不承认你的权利或者要求你放弃这些权利。如果你自行发布了软件的副本或者对其进行了修改, 那么这些规定就转化为你的责任

例如, 如果你发布这样一个程序的副本, 不管是收费的还是免费的, 你必须将你具有的一切权利给予你的接受者; 你必须保证他们能收到或得到源程序; 并且将这些条款给他们看, 使他们知道他们有这样的权利。

我们采取两项措施来保护你的权利。

一是给软件以版权保护; 二是给你提供许可证。它给你复制、发布和修改这些软件的法律许可。

同样, 为了保护每个作者和我们自己, 我们需要清楚地让每个人明白, 自由软件没有担保。如果由于其他某个人修改了软件, 并继续加以传播。我们需要它的接受者明白: 他们所得到的并不是原来的自由软件。由其他人引入的任何问题, 不应损害原作者的声誉。

最后, 任何自由软件都不断受到软件专利的威胁。我们希望能够避免这样的风险: 自由软件的再发布者以个人名义获得专利许可证, 实际上将软件变为私有。为防止这一点, 我们必须明确: 任何专利必须以允许每个人自由使用为前提, 否则就不准许有专利。

下面是有关复制、发布和修改的确切的条款和条件。

### 复制、发布和修改的条款和条件

此许可证适用于任何包含版权所有声明的程序和其他作品, 版权所有者在声明中明确

说明程序和作品可以在本GPL条款的约束下发布。下面提到的“程序”指的是任何这样的程序或作品。而“基于程序的作品”指的是程序或者任何受版权法约束的衍生作品：也就是说包含程序或程序的一部分的作品。可以是原封不动的，也可以是经过修改的和/或翻译成其他语言的（程序）。在下文中，翻译包含在修改的条款中。每个许可证接受人（licensee）用“你”来称呼。

许可证条款不适用于复制、发布和修改以外的活动。这些活动超出这些条款的范围。运行程序的活动不受条款的限制。仅当程序的输出构成基于程序作品的内容时，这一条款才适用（如果只运行程序就无关）。是否普遍适用取决于程序具体用来做什么。

1) 只要你在每一副本上明显和恰当地给出版权声明和不承担担保的声明，保持此许可证的声明和没有担保的声明完整无损，并和程序一起给每个其他的程序接受者一份许可证的副本，你就可以用任何媒体复制和发布你收到的原始的程序的源代码。

你可以为转让副本的实际行动收取一定费用。你也有权选择提供担保以换取一定的费用。

2) 你可以修改程序的一个或几个副本或程序的任何部分，以此形成基于程序的作品。只要你同时满足下面的所有条件，你就可以按前面第一款的要求复制和发布这一经过修改的程序或作品。

a) 你必须在修改的文件中附有明确的说明：你修改了这一文件及具体的修改日期。

b) 你必须使你发布或出版的作品（它包含程序的全部或部分，或包含由程序的全部或部分衍生的作品）允许第三方作为整体按许可证条款免费使用。

c) 如果修改的程序在运行时以交互方式读取命令，你必须使它在开始进入常规的交互使用方式时打印或显示声明：包括适当的版权声明和没有担保的声明（或者你提供担保的声明）；用户可以按此许可证条款重新发布程序的说明；并告诉用户如何看到这一许可证的副本（例外的情况：如果程序自身以交互方式工作，但是它并不像通常情况一样打印这样的声明，你的基于程序的作品也就不需要打印声明）。

这些要求适用于修改了的作品整体。如果能够确定作品的一部分并非程序的衍生产品，可以合理地认为这部分是独立的，是不同的作品。当你将它作为独立作品发布时，它不受此许可证和它的条款的约束。但是当你将这部分作为基于程序的作品的一部分发布时，作为整体它将受到许可证条款约束。准予其他许可证持有人的使用范围扩大到整个产品。也就是每个部分，不管它是谁写的。

因此，本条款的意图不在于索取权利；或剥夺全部由你写成的作品的权利。而是履行权利来控制基于程序的集体作品或衍生作品的发布。

此外，将与程序无关的作品和该程序或基于程序的作品一起放在存储介质或发布媒体的同一卷上，并不导致将其他作品置于此许可证的约束范围之内。

3) 你可以以目标码或可执行形式复制或发布程序（或符合第2款的基于程序的作品），只要你遵守前面的第1、2款，并同时满足下列3条中的1条。

a) 在通常用作软件交换的媒体上，和目标码一起附有机可读的完整的源码。这些源码的发布应符合上面第1、2款的要求。

b) 在通常用作软件交换的媒体上，和目标码一起附有给第三方提供相应的机器可读的源码的书面报价。有效期不少于3年，费用不超过完成源程序发布的实际成本。源码的发布应符合上面的第1、2款的要求。

c) 和目标码一起附有你收到的发布源码的报价信息（这一条款只适用于非商业性发布，而且你只收到程序的目标码或可执行代码和按b款要求提供的报价）。

作品的源码指的是对作品进行修改最优先择取的形式。对可执行的作品讲，完整的源码包括：所有模块的所有源程序，加上有关的接口的定义，加上控制可执行作品的安装和编译的script。作为特殊例外，发布的源码不必包含任何常规发布的供可执行代码在上面运行的操作系统的主要组成部分（如编译程序、内核等）。除非这些组成部分和可执行作品结合在一起。

如果采用提供对指定地点的访问和复制的方式发布可执行码或目标码，那么，提供对同一地点的访问和复制源码可以算作源码的发布，即使第三方不强求与目标码一起复制源码。

4) 除非你明确按许可证提出的要求去做，否则你不能复制、修改、转发许可证和发布程序。任何试图用其他方式复制、修改、转发许可证和发布程序是无效的。而且将自动结束许可证赋予你的权利。然而，对那些从你那里按许可证条款得到副本和权利的人们，只要他们继续全面履行条款，许可证赋予他们的权利仍然有效。

5) 因为你没有在许可证上签字，因而你没有必要一定要接受这一许可证。然而，没有任何其他东西赋予你修改和发布程序及其衍生作品的权利。如果你不接受许可证，这些行为是法律禁止的。因此，如果你修改或发布程序（或任何基于程序的作品），这就表明你接受这一许可证以及它的所有有关复制、发布和修改程序或基于程序的作品条款和条件。

6) 每当你重新发布程序（或任何基于程序的作品）时，接受者自动从原始许可证颁发者那里接收到这些条款和条件支配的复制、发布或修改程序的许可证。你不可以对接受者履行这里赋予他们的权利强加其他限制。你也没有强求第三方履行许可证条款的义务。

7) 如果由于法院判决或违反专利的指控或任何其他原因（不限于专利问题）的结果，强加于你的条件（不管是法院判决、协议或其他）和许可证的条件有冲突。他们也不能用许可证条款为你开脱。在你不能同时满足本许可证规定的义务及其他相关的义务时，你可以根本不发布程序。例如，如果某一专利许可证不允许所有那些直接或间接从你那里接受副本的人们在不付专利费的情况下重新发布程序，唯一能同时满足两方面要求的办法是停止发布程序。

如果本条款的任何部分在特定的环境下无效或无法实施，就使用条款的其余部分。并将条款作为整体用于其他环境。

本条款的目的不在于引诱你侵犯专利或提出其他财产权的要求，或争论这种要求的有效性。本条款的主要目的在于保护自由软件发布系统的完整性。它是通过通用公共许可证的应用来实现的。许多人坚持应用这一系统，已经为通过这一系统发布大量自由软件做出慷慨的贡献。作者/捐献者有权决定他/她是否通过任何其他系统发布软件。许可证持有人不能强制这种选择。

本节的目的在于明确说明许可证其余部分可能产生的结果。

8) 如果由于专利或者由于有版权的接口问题使程序在某些国家的发布和使用受到限制，将此程序置于许可证约束下的原始版权拥有者可以增加限制发布地区的条款，将这些国家明确排除在外。并在这些国家以外的地区发布程序。在这种情况下，许可证包含的限制条款和许可证正文一样有效。

9) 自由软件基金会可能随时出版通用公共许可证的修改版或新版。新版和当前的版本在原则上保持一致，但在提到新问题或有关事项时，在细节上可能存在一定差别。

每一版本都有不同的版本号。如果程序指定适用于它的许可证版本号以及“任何更新的版本”。你有权选择遵循指定的版本或自由软件基金会以后出版的新版本，如果程序未指定许可证版本，你可选择自由软件基金会已经出版的任何版本。

10) 如果你愿意将程序的一部分结合到其他自由软件中，而它们的发布条件不同。写信给作者，要求准予使用。如果是自由软件基金会加以版权保护的软件，写信给自由软件基金会，我们有时会作为例外的情况处理。我们的决定受两个主要目标的指导。这两个主要目标是：我们的自由软件的衍生作品继续保持自由状态。从整体上促进软件的共享和重复利用。

## 没有担保

11) 由于程序准予免费使用，在适用法律准许的范围内，对程序没有担保。除非另有书面说明，版权所有者和/或其他提供程序的人们“一样”不提供任何类型的担保。不论是明确的，还是隐含的。包括但不限于隐含的适销和适合特定用途的保证。全部的风险，如程序的质量和性能问题都由你来承担。如果程序出现缺陷，你承担所有必要的服务、修复和改正的费用。

12) 除非适用法律或书面协议的要求，在任何情况下，任何版权所有者或任何按许可证条款修改和发布程序的人们都不对你的损失负有任何责任。包括由于使用或不能使用程序引起的任何一般的、特殊的、偶然发生的或重大的损失（包括但不限于数据的损失，或者数据变得不精确，或者你或第三方的持续的损失，或者程序不能和其他程序协调运行等）。即使版权所有者和其他人提到这种损失的可能性也不例外。

## 如何将这些条款用到你的新程序

如果你开发了新程序，而且你需要它得到公众最大限度的利用。要做到这一点的最好办法是将它变为自由软件。使得每个人都能在遵守条款的基础上对它进行修改和重新发布。

为了做到这一点，给程序附上下列声明。最安全的方式是将它放在每个源程序的开头，以便最有效地传递拒绝担保的信息。每个文件至少应有“版权所有”行以及在什么地方能看到声明全文的说明。

<用一行空间给出程序的名称和它用来做什么的简单说明>

版权所有 (C) 19XX <作者姓名>

这一程序是自由软件，你可以遵照自由软件基金会出版的GNU通用公共许可证条款来修改和重新发布这一程序。或者用许可证的第2版，或者（根据你的选择）用任何更新的版本。

发布这一程序的目的是希望它有用，但没有任何担保。甚至没有适合特定目的的隐含的担保。更详细的情况请参阅GNU通用公共许可证。

你应该已经和程序一起收到一份GNU通用公共许可证的副本。如果还没有，写信给：

The Free Software Foundation, Inc., 59 Temple Place, Suite,  
330, Boston, MA 02111-1307 USA

还应加上如何和你保持联系的信息。

如果程序以交互方式进行工作，当它开始进入交互方式工作时，使它输出类似下面的简短声明：

Gnomovision 第69版， 版权所有 (C) 19XX， 作者姓名，

Gnomovision绝对没有担保。要知道详细情况，请输入‘show w’。

这是自由软件，欢迎你遵守一定的条件重新发布它，要知道详细情况，请输入‘show c’。

假设的命令‘show w’和‘show c’应显示通用公共许可证的相应条款。当然，你使用的命令名称可以不同于‘show w’和‘show c’。根据你的程序的具体情况，也可以用菜单或鼠标选项来显示这些条款。

如果需要，你应该取得你的上司（如果你是程序员）或你的学校签署放弃程序版权的声明。下面只是一个例子，你应该改变相应的名称：

Yoyodyne公司以此方式放弃James Hacker所写的Gnomovision程序的全部版权利益。

<Ty coon签名>，1989.4.1

Ty coon副总裁

这一许可证不允许你将程序并入专用程序。如果你的程序是一个子程序库。你可能会认为用库的方式和专用应用程序连接更有用。如果这是你想做的事，使用GNU库通用公共许可证代替本许可证。

## 附录C 光盘上的内容及系统需求

本书配套光盘上包含的内容有:

- linux-0.01: 原始的linux内核描述, 1991年的数据, 为满足了解历史的兴趣而提供。
  - linux-2.2.5: 这本书使用的全部内核代码。
  - linux-2.2.10: 到准备这张光盘时最新的稳定内核。
  - linux-2.3.12: 到准备这张光盘时最新的开发内核。
  - 所包含的2.X版本的内核描述的标志文件。
  - lckc\_code: 这本书所包含的所有代码, 它在一个单一的大文件里, 并且所有的代码行已经编了号。就像这本书的第一部分所显示的那样。
  - lckc-find-line.el: 一个简短的Emacs Lisp文件, 它可以方便地对lckc\_code进行导航。
- 以上软件受GNU通用公共许可证保护, 这意味着你可以随心所欲地使用、复制或修改它。

### 系统需求

#### 软件

- 可以使用任何能够从光盘上读取文件的操作系统, 以便查看代码。GNU/Linux就很好用。在GNU/Linux下编译代码就更容易了。前提是你有一个可用的装有gcc 2.7.2的GNU/Linux系统。
- 用Emacs、vi或其他识别标签(小箭头)的文本编辑器均可以方便地浏览源代码(光盘上没有文本编辑器)。

#### 硬件

- 查看内核代码对于CPU或内存没有特别的要求。
- 在几乎所有可以运行GNU/Linux的系统上都可以编译源代码。在编译时建议使用Intel Pentium 133MHz处理器和至少16MB内存。当然, 其他平台(例如基于Alpha或基于SPARC的工作站)也可以, 但速度差不多。
- 如果仅想查看代码, 可以直接从光盘上读取文件, 这样对于硬盘空间没有特殊的要求。但若要编译内核时, 也许想把源代码复制到硬盘上, 这样每个内核需要大约130MB的硬盘空间(对于2.X版本的内核而言)。编译之后, 这个数字则上升为150MB或更多, 这取决于内核是如何配置的。