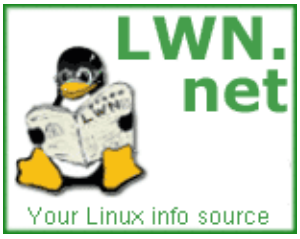


Porting device drivers to the 2.6 kernel

The 2.6 kernel contains a long list of changes which affect device driver writers. As part of the task of porting the *Linux Device Drivers* sample code to 2.6, your humble LWN Kernel Page author is producing a set of articles describing the changes which must be made. The articles are Kernel Page as they are written; they will also be collected here. With luck, this page will be a useful reference for those who must port drivers to the new kernel.

The creation of these articles is funded by LWN.net subscribers. If you find this material useful, please consider subscribing to LWN to help ensure that more of it gets written.

Except when otherwise specified, all of the articles below are written by LWN editor Jonathan Corbet. The date and kernel version attached to each article notes when the article was last updated.

Recent changes

The most recent changes to this series are:

- (November 25, 2003) The entire set of articles has been updated to reflect the 2.6.0−test10 kernel.
- (Oct. 29, 2003) Examining a kobject hierarchy added.
- (Oct. 23, 2003) kobject and sysfs added.
- (Oct. 7, 2003) kobjects and hotplug events added.
- (Oct. 1, 2003) The zen of kobjects added.

Getting started

Porting 'hello world' (February, 2003); which covers the changes required to update the simplest possible module to the 2.5 kernel.

Compiling external modules (November, 2003; 2.6.0−test9); how to build modules with the new module loader and kernel build scheme.

More module changes (November, 2003, 2.6.0−test9) covers other changes to the module loading subsystem, including module parameters, use count management, exporting symbols, and more.

Miscellaneous changes is a collection point for changes which are too small to justify their own article. Currently covered topics include `kdev_t`, designated initializers, and `min()` and `max()`. It was last updated on **November 3**, 2003 (2.6.0−test9).

Support interfaces

Char drivers and large dev_t (November 2003, 2.6.0−test9); registration and management of char drivers in the new, large `dev_t` environment.

The seq_file interface (September 2003; 2.6.0−test6); the easy way to implement virtual files correctly. A standalone example module is provided to demonstrate the use of this interface.

Low−level memory allocation (November, 2003; 2.6.0−test9); changes to functions for allocating chunks of memory and pages, and a description of the new mempool interface.

Per−CPU variables (November, 2003; 2.6.0−test9); the 2.6 interface for maintaining per−CPU data structures.

Timekeeping changes (November, 2003; 2.6.0−test9); changes to how the kernel manages time and time−related events.

The workqueue interface (November, 2003; 2.6.0−test9); a description of the new deferred execution mechanism which replaces task queues (and bottom halves in general).

Creating virtual filesystems with libfs (November, 2003; 2.6.0−test9). This article, which looks at how a kernel module can create its own virtual filesystem, predates the driver porting series but fits in well with it.

DMA Changes (November, 2003, 2.6.0−test9); changes to the DMA support layer. There is also a quick reference page for the new generic DMA API.

Sleeping and mutual exclusion

Mutual exclusion with seqlocks (November, 2003, 2.6.0−test9); a description of how to use the seqlock (formerly frlock) capability which was merged into 2.5.60.

The preemptible kernel (November, 2003; 2.6.0−test9); a look at how kernel preemption affects driver code and what can be done to work safely in the preemptible environment.

Sleeping and waking up (November, 2003; 2.6.0−test9); new ways of putting processes to sleep with better performance and without race conditions.

Completion events (November, 2003; 2.6.0−test9); documentation for the completion event mechanism.

Using read−copy−update (November, 2003; 2.6.0−test9); working with the read−copy−update mutual exclusion scheme.

Advanced driver tasks

Dealing with interrupts (November, 2003; 2.6.0−test9); interrupt handling changes which are visible to device drivers.

Supporting asynchronous I/O (November, 2003; 2.6.0−test9); how to write drivers which support the 2.6 asynchronous I/O interface.

Network drivers (November 2003, 2.6.0−test9); porting network drivers, with an emphasis on the new dynamic `net_device` allocation functions and NAPI support.

USB driver API changes (July 2003; 2.5.75); how USB drivers have changed in the 2.5 development series. This article was contributed by USB maintainer Greg Kroah−Hartman.

Block drivers

Block layer overview (November, 2003; 2.6.0−test9). The block layer has seen extensive changes in the 2.5 development series; this article gives an overview of what has been done while deferring the details for subsequent articles.

A simple block driver (November, 2003; 2.6.0−test9); this article presents the simplest possible block driver (a basic ramdisk implementation) with discussion of how the basic block interfaces have changed in 2.6. Full source to a working driver is included.

The gendisk interface (November, 2003; 2.6.0−test9); how to work with the new generic disk interface, which takes on a rather larger role in 2.6.

The BIO structure (November, 2003; 2.6.0−test9); the new low−level structure representing block I/O operations.

Request queues I (November, 2003; 2.6.0−test9); the basics of block request queues in 2.6, including request processing, request preparation control, and DMA support.

Request queues II (November, 2003, 2.6.0−test9); advanced request queue topics, including command preparation, tagged command queueing, and the "make request" mode of operation.

Memory management

Supporting mmap() (November, 2003 − 2.6.0−test9); changes in how device drivers support the `mmap()` system call.

Zero−copy user−space access (November, 2003 − 2.6.0−test9); how to get direct−access to user space to perform zero−copy I/O. If you used the `kiobuf` interface for this purpose in 2.4, you'll want to look here for the 2.6 equivalent.

Atomic kmaps (November, 2003; 2.6.0−test9); quick access to high−memory via `kmap_atomic()`.

Device model

A device model overview (November, 2003; 2.6.0−test10); an introductory look at the Linux device model and sysfs, with definitions of some commonly encountered terms.

The zen of kobjects (October, 2003; 2.6.0−test6); an attempt to demystify the kobject abstraction and its use in the kernel.

kobjects and sysfs (October, 2003; 2.6.0−test8); a description of the interaction between the kobject type and its representation in sysfs.

kobjects and hotplug events (October, 2003; 2.6.0−test6); an explanation of the kset hotplug operations and how they can be used to control how hotplug events are reported to user space. This article was written by Greg Kroah−Hartman.

Examining a kobject hierarchy (October, 2003; 2.6.0−test9); a visual exploration of the device model data structures behind `/sys/block`.

Device classes (November, 2003; 2.6.0−test10); how the device class mechanism works.

Post a comment

Porting device drivers to the 2.5 kernel

(Posted Feb 12, 2003 1:55 UTC (Wed) by **fdesloges**) (Post reply)

Wow!

This is highly valuable stuff. Are you certain you want to give this away for free after only a few days ?

This alone would be a very good reason to subscribe. And as reference stuff it will still be valuable many months down the road.

Maybe it could fit in a "available to non−subsciber 4 months later" category ? Or make only a few articles available (and the index of course to hook subscribers) ?

Whatever you do, thanks!

FD

Porting device drivers to the 2.5 kernel

(Posted Feb 12, 2003 9:58 UTC (Wed) by **KotH**) (Post reply)

I wouldnt do that. LWN has a very good reputation as information source around the open source community. A "closed for all but subscribed ppl" policy would surely destroy that reputation.

Yes, i know that LWN doesnt have as much subscribers as it should have :(but this is IMHO not the right way to get more.

Porting device drivers to the 2.5 kernel

(Posted Feb 12, 2003 11:00 UTC (Wed) by **bruno**) (Post reply)

Do you work for nothing? Do you give all your time to other in exchange of nothing? You can't pretend that other people do that, the people have childrens, wives, mortgages... and work in exchange of money to pay their bills. Sometimes you have a bussiness plan that lets you to give away your work and get money from other sources, sometimes you need to put a price or close the shop.

Porting device drivers to the 2.5 kernel

(Posted Feb 12, 2003 14:14 UTC (Wed) by **Webexcess**) (Post reply)

Do you work for nothing? Do you give all your time to other in exchange of nothing?

I think you're oversimplifying a bit. The excellent writing at lwn.net is both its product and its advertising. How will potential subscribers be enticed if they can't see what lwn.net has to offer?

Also, the target audience largely consists of enthusiasts, many of whom are students and/or live in parts of the world where american dollars are very expensive. These are the same people that helped to build Linux into what it is today −− are you suggesting that they should be excluded?

Porting device drivers to the 2.5 kernel

(Posted Feb 12, 2003 15:27 UTC (Wed) by **bruno**) (Post reply)

The excellent writing at lwn.net is only possible if there are someone that works in it full time. I think is wonderfull if someone can (and want) to work full time in something and give it away for nothing, but I don't think that you can say to someone "You MUST give your work for free", at the end of the day, with your work you have the right to make whatever you want: Give if for free, sell, rent, lease or burn it, is your work and your decision.

If KotH really thinks that is important that this information be free, he can study the linux kernel, write a book and put it on the web for free, instead of criticise the actitude of lwn.net

Porting device drivers to the 2.5 kernel

(Posted Feb 12, 2003 17:17 UTC (Wed) by **rknop**) (Post reply)

"You MUST give your work for free"... If KotH really thinks that is important that this information be free,

Be fair. That's not what KotH said. He said he thought it was in LWN.net's best interest to do what they've done, because of their good reputation in the community. He's not insisting that they must do what they've done, he's just congratulating them and saying he understands why they might think it's a good idea to do that.

−Rob

Re: Porting device drivers to the 2.5 kernel

(Posted Feb 12, 2003 19:52 UTC (Wed) by **Ross**) (Post reply)

Bruno,

I think you are mischaracterizing what other people are saying. Using the device driver documentation as an advertisement was only a suggestion. We all recognize that LWN can decide what to release and when to release it. You're point is also only a suggestion. If Jonathan felt he needed to say something, I'm sure he would. You don't need to speak for him.

Porting device drivers to the 2.5 kernel

(Posted May 8, 2003 17:05 UTC (Thu) by **LogicG8**) (Post reply)

I would just like to say that the release of this material prompted me to purchase a subscription. It is a delicate balance providing open content and providing incentive to readers to subscribe. I think that LWN has done a great job and will continue to do so in the future.

Releasing quality content for free has nabbed at least one subscriber.

On whether this stuff should be free

(Posted Feb 12, 2003 20:07 UTC (Wed) by **corbet**) (Post reply)

Just as a response to all the comments here... I appreciate the input, and certainly do not feel criticized by any of the comments.

For what it's worth, I did consider keeping this material non−free for longer than the usual period. It is different from the usual news, and it has a slightly longer useful life. In the end, I decided against such a move; I would like these articles to be generally useful, and to serve as a contribution to the kernel project. Maintaining the same access policy also lets me fold some of the articles into the Kernel Page, which could use it − development news tends to slow down a lot during feature freezes.

So the driver porting articles go free after a week. We may yet do things differently for similar material in the future.

jon

An idea that works elsewhere

(Posted Feb 27, 2003 15:20 UTC (Thu) by **materlik**) (Post reply)

You might think about adding a little, polite blurb asking for a small donation to the end of each article, similar to what tidbits (a free Macintosh newsletter) is doing with PayBits. Maybe one only visible to non−subscribers?
Their experiences so far have not been overwhelming, but there is some money being donated because of the constant reminders on the value of filtered, edited information.

Porting device drivers to the 2.5 kernel

(Posted Feb 28, 2003 20:09 UTC (Fri) by **vonbrand**) (Post reply)

I think this deserves a place in the head of the page (perhaps replacing Old Site or Weekly edition). It is not a "Recent feature" anymore...

One minor gripe is that the articles don't default to a printable page format (yes, I do promise to buy the next book of yours regardless ;−)

Driver porting: hello world

This article is part of the LWN Porting Drivers to 2.6 series.

Your editor is currently in the middle of porting the example source from Linux Device Drivers, Second Edition to the 2.5 kernel. This work is, of course, just the beginning of the rather larger job of updating the whole book. This article is the first in what will, hopefully, be a series describing what is required to make this code work again. The series will thus, with luck, be useful as a guide to how to port drivers to the new kernel API.

The obvious place to start in this sort of exercise, of course, is the classic "hello world" program, which, in this context, is implemented as a kernel module. The 2.4 version of this module looked like:

```
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void)
{
    printk(KERN_INFO "Hello, world\n");
    return 0;
}

void cleanup_module(void)
{
    printk(KERN_INFO "Goodbye cruel world\n");
}
```

One would not expect that something this simple and useless would require much in the way of changes, but, in fact, this module will not quite work in a 2.5 kernel. So what do we have to do to fix it up?

The first change is relatively insignificant; the first line:

```
#define MODULE
```

is no longer necessary, since the kernel build system (which you really should use now, see the next article) defines it for you.

The biggest problem with this module, however, is that you have to explicitly declare your initialization and cleanup functions with `module_init` and `module_exit`, which are found in `<linux/init.h>`. You really should have done that for 2.4 as well, but you could get away without it as long as you used the names `init_module` and `cleanup_module`. You can still sort of get away with it (though you may have to ignore some compiler warnings), but the new module code broke this way of doing things once, and could do so again. It's really time to bite the bullet and do things right.

With these changes, "hello world" now looks like:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int hello_init(void)
{
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}

static void hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

This module will now work − the "Hello, world" message shows up in the system log file. What also shows up there, however, is a message reading "hello: module license 'unspecified' taints kernel." "Tainting" of the kernel is (usually) a way of indicating that a proprietary module has been inserted, which is not really the case here. What's missing is a declaration of the license used by the module:

```
MODULE_LICENSE("Dual BSD/GPL");
```

`MODULE_LICENSE` is not exactly new; it was added to the 2.4.10 kernel. Some older code may still lack `MODULE_LICENSE` calls, however. They are worth adding; in addition to avoiding the "taints kernel" message, a license declaration gives your module access to GPL−only kernel symbols. Assuming, of course, that the module is GPL−licensed.

With these changes, "hello world" works as desired. At least, once you have succeeded in building it properly; that is the subject of the next article.

Post a comment

Driver porting: hello world

(Posted Feb 14, 2003 5:19 UTC (Fri) by **rusty**) (Post reply)

Hi Jon,

I appeciate the series in modernizing modules, but just FYI, I don't think the old−style init_module/cleanup_module stuff will break any time soon: there are still a large number of drivers which use it, and there's not much point making such changes.

However, this "new" scheme (introduced by Linus in 2.3 IIRC) has the advantage that your module will work correctly when built into the kernel: you want that initialization routine called, even then.

So I hope that clarifies,
Rusty.

Driver porting: compiling external modules

This article is part of the LWN Porting Drivers to 2.6 series.

The 2.5 development series saw extensive changes to the kernel build mechanism and the complete replacement of the module loading code. One result of these changes is that compiling loadable modules has gotten a bit more complicated. In the 2.4 days, a makefile for an external module could be put together in just about any old way; typically a form like the following was used:

```
KERNELDIR = /usr/src/linux
CFLAGS = -D__KERNEL__ -DMODULE -I$(KERNELDIR)/include -O

all: module.o
```

Real−world makefiles, of course, tended to be a bit more complicated, but the job of creating a loadable module was handled in a single, simple compilation step. All you really needed was a handy set of kernel headers to compile against.

With the 2.6 kernel, you still need those headers. You also, however, need a configured kernel source tree and a set of makefile rules describing how modules are built. There's a few reasons for this:

- The new module loader needs to have some extra symbols defined at compilation time. Among other things, it needs to have the `KBUILD_BASENAME` and `KBUILD_MODNAME` symbols defined.
- All loadable modules now need to go through a linking step − even those which are built from a single source file. The link brings in `init/vermagic.o` from the kernel source tree; this object creates a special section in the loadable module describing the environment in which it was built. It includes the compiler version used, whether the kernel was built for SMP, whether kernel preemption is enabled, the architecture which was compiled for, and, of course, the kernel version. A difference in any of these parameters can render a module incompatible with a given running kernel; rather than fail in mysterious ways, the new module loader opts to detect these compatibilities and refuse to load the module.

  As of this writing (2.5.59), the "vermagic" scheme is fallible in that it assumes a match between the kernel's `vermagic.o` file and the way the module is being built. That will normally be the case, but people who change compiler versions or perform some sort of compilation trickery could get burned.
- The new symbol versioning scheme ("modversions") requires a separate post−compile processing step and yet another linkable object to hold the symbol checksums.

One could certainly, with some effort, write a new, standalone makefile which would handle the above issues. But that solution, along with being a pain, is also brittle; as soon as the module build process changes again, the makefile will break. Eventually that process will stabilize, but, for a while, further changes are almost guaranteed.

So, now that you are convinced that you want to use the kernel build system for external modules, how is that

to be done? The first step is to learn how kernel makefiles work in general; makefiles.txt from a recent kernel's `Documentation/kbuild` directory is recommended reading. The makefile magic needed for a simple kernel module is minimal, however. In fact, for a single−file module, a single−line makefile will suffice:

```
obj−m := module.o
```

(where `module` is replaced with the actual name of the resulting module, of course). The kernel build system, on seeing that declaration, will compile `module.o` from `module.c`, link it with `vermagic.o`, and leave the result in `module.ko`, which can then be loaded into the kernel.

A multi−file module is almost as easy:

```
obj−m := module.o
module−objs := file1.o file2.o
```

In this case, `file1.c` and `file2.c` will be compiled, then linked into `module.ko`.

Of course, all this assumes that you can get the kernel build system to read and deal with your makefile. The magic command to make that happen is something like the following:

```
make −C /path/to/source SUBDIRS=$PWD modules
```

Where `/path/to/source` is the path to the source directory for the (configured and built) target kernel. This command causes make to head over to the kernel source to find the top−level makefile; it then moves back to the original directory to build the module of interest.

Of course, typing that command could get tiresome after a while. A trick posted by Gerd Knorr can make things a little easier, though. By looking for a symbol defined by the kernel build process, a makefile can determine whether it has been read directly, or by way of the kernel build system. So the following will build a module against the source for the currently running kernel:

```
ifneq ($(KERNELRELEASE),)
obj−m       := module.o

else
KDIR        := /lib/modules/$(shell uname −r)/build
PWD         := $(shell pwd)

default:
    $(MAKE) −C $(KDIR) SUBDIRS=$(PWD) modules
endif
```

Now a simple "make" will suffice. The makefile will be read twice; the first time it will simply invoke the kernel build system, while the actual work will get done in the second pass. A makefile written in this way is simple, and it should be robust with regard to kernel build changes.

Post a comment

Driver porting: compiling external modules

(Posted Feb 6, 2003 6:05 UTC (Thu) by **rfunk**) (Post reply)

For the make trickery in the latter part of this article, it's important to note that the original make command assumes the kernel source is in /usr/src/linux/:
make −C /usr/src/linux SUBDIRS=$PWD modules
While the makefile that allows a shorter command line assumes the kernel source is in a directory like /lib/modules/2.5.59/build/, due to the following line:
KDIR := /lib/modules/$(shell uname −r)/build

I believe there was some minor controversy on the linux−kernel mailing list recently over which location is more appropriate.

Path to the kernel source

(Posted Feb 6, 2003 14:44 UTC (Thu) by **corbet**) (Post reply)

Actually, `/usr/src/linux` is pretty much deprecated by the Prime Penguin himself; you're supposed to keep your kernel sources somewhere else. I got lazy and used it in the example, mostly because it's shorter to type than the `/lib/modules` path. The latter is the better way to go, however, especially in scripts or makefiles − it "automatically" points to the right source tree, unless you move your trees around.

Driver porting: compiling external modules

(Posted Feb 6, 2003 11:03 UTC (Thu) by **raarts**) (Post reply)

Thanks for this article.
This is why I subscribe to LWN.

Ron Arts

What about cross−compilation?

(Posted Feb 6, 2003 18:35 UTC (Thu) by **sjmadsen**) (Post reply)

The Makefile trickery at the end isn't going to work in cross−compiler enviornments. My company is building a product that uses Linux as the embedded OS, but builds typically take place on Solaris.

Even if we were building on Linux, it's unlikely that the OS on the build machine is going to match the embedded environment.

What about cross−compilation?

(Posted Feb 6, 2003 21:02 UTC (Thu) by **Peter**) (Post reply)

> The Makefile trickery at the end isn't going to work in cross−compiler enviornments. My company is building a product that uses Linux as the embedded OS, but builds typically take place on Solaris.

True enough. In fact, there is *no* automated way for the computer to read your mind and know, for a specific module build, where the matching kernel tree resides. If you don't give the computer any more information, one reasonable guess is "wherever the currently−running kernel was built, assuming it was built on this machine". In fact, I can't think of a better default guess. But if this turns out to be wrong − for various reasons, including cross−compilation − you are going to have to specify the source location.

Putting that location in the Makefile as a special macro like KERNELDIR makes it possible to override on the command line: 'make KERNELDIR=...'. The other option, of course, is to work to get your module integrated into the official kernel tree, at which time you are rid of this headache once and for all.

What about cross−compilation?

(Posted Feb 7, 2003 15:50 UTC (Fri) by **dwmw2**) (Post reply)

It works perfectly for cross−compilation for me. If you override CROSS_COMPILE (or have it set in your kernel's top−level Makefile) that works just as it always did:@
make KDIR=/local/arm−kernel CROSS_COMPILE=arm−linux−gnu−

Note also that the article is somewhat misleading −− the Makefile fragment

> KERNELDIR = /usr/src/linux
> CFLAGS = −D__KERNEL__ −DMODULE −I$(KERNELDIR)/include −O
>
> all: module.o

... was _always_ broken and nonportable −− building using the kernel makefiles was the only way to get it working portably since about the 2.0 kernel. It doesn't kernel+arch−specific CFLAGS like −mregparm= −mno−gp−opt −mno−implicit−fp right.

Hardware vendor supplied modules

(Posted Feb 6, 2003 21:02 UTC (Thu) by **pradu**) (Post reply)

I see a problem with hardware (and software) vendors modules that are supplied in source code (see for instance VMWare kernel modules). If you can't compile a kernel module without the kernel source tree and on a different machine (maybe with different compiler/processor and whatnot) from the distibutor kernel, you can't use such modules.

Or I am missing something?

Hardware vendor supplied modules

(Posted Feb 8, 2003 0:28 UTC (Sat) by **Peter**) (Post reply)

> I see a problem with hardware (and software) vendors modules that are supplied in source code (see for instance VMWare kernel modules). If you can't compile a kernel module without the kernel source tree and on a different machine (maybe with different compiler/processor and whatnot) from the distibutor kernel, you can't use such modules.

There are a few possibilities here:

- completely disallow external modules − then, VMWare and co. would have to provide a patch to integrate their modules with the kernel source proper, and each customer would have to apply this patch and rebuild the kernel. (Or, alternatively, VMWare could ship with a custom kernel, but then they'd have to follow the GPL and open−source their module.)
- have VMWare and their ilk each track all the major vendor kernels and release modules to go with each one. Many proprietary module vendors do this today. It's a horrible hack but it does work for some people.

- freeze the kernel/module interface, somehow nullify or abstract away all differences that affect it, and maintain this situation for a given length of time (say, a stable series). That would include gcc 2.95 vs. gcc 3.2, preemptible vs. non−preemptible, UP vs. SMP, 386 vs. 586 vs. K7, highmem vs. non−highmem, and a few other details, on the x86 platform. This is the Microsoft / commercial Unix solution, and the Linux people refuse to bother with it. If you think this is the way to go, please be prepared to present your patch which does all this without affecting efficiency or maintainability of the source base. Alternately, Google for 'UDI unix linux' and see what happened the last time this was proposed.
- take some sort of snapshot of the state of the build environment when you build a kernel, and make this snapshot available somehow for when you want to build an external module. This snapshot would consist of kernel headers, .config file, exact compiler and its flags, and some sort of glue to plug the external module in to all this.

At present, we have the latter option, in the form of a live source tree. Note that if you can locate the live source tree you wish to build against, and the compiler you used to build the kernel, and any other variables you passed in when you built the kernel (like a CROSS_COMPILE variable), you can use this method just fine.

If you *don't* know this information, unfortunately there is no way for the computer to divine it for you. What if I build 10 different kernels on my box, destined for 10 different machines, then I try to build an external module? How is the system supposed to know which of my 10 kernels the module is for? Barring a direct DWIM neural interface (for which Linux drivers are still sadly lacking), I'm gonna have to point my module at the proper kernel tree, and re−set−up any custom variables I set the first time.

Hardware vendor supplied modules

(Posted Feb 13, 2003 22:53 UTC (Thu) by **mmarq**) (Post reply)

>freeze the kernel/module interface, somehaw nullify or abstract away all differences...

IN OTHER WORDS, BUILD A IN KERNEL API/ABI (like in LSB)!!!

THE PROBLEM WHIT "UDI" WAS NOT THE IDEA IN ITSELF, BUT "HOW AND WHO" WAS IN CONTROL...IN THE BEGINING LINUS WAS A PROMOTER OF IT...

YOU ALREADY HAVE A "ABSTRACT INTERFACE" IN KERNEL IN THE FORM OF I2O...THE PROBLEM IS THAT YOU NEED "SPECIAL HARDWARE" TO GET ALONG WITH IT...

The current "state of the art" is, as specified in your explanation, there is no simple answer to any question... be it cross−compilation, be it building external modules, or whatever related to hardware. There's almost 100% certainty now that you run against "gcc 2.95 vs. gcc 3.2, preemptible vs. non−preemptible, UP vs. SMP, 386 vs. 586 vs. K7, highmem vs. non−highmem"....

BUILDING GOOD AND COMPLETE WIDE SUPPORTE FOR A PIECE OF HARDWARE IN EXTERNAL MODULES, "IS FOR SURE NOW" ONLY FOR A SKILLED AND WELL DOCUMENTED "COMERCIAL" TEAM...(how about that????)

If you have to have a good and extensive support of hardware in the form of in kernel loadable modules, it would mean that the source of the kernel could easely be 10 times of current size, cutting out those that dont have broadband(more than half of the world), or...

...LINUS GETS FIRED FROM TRANSMETA OR DIES OF EXAUSTION!...OR FOR STABLE KERNEL 2.8/3.0 YOU HAVE A "DEVELOPMENT CICLE OF 4 YEARS",...OR WORST THAN EVERYTHING ELSE BEFORE!!!...YOU SEE THE KERNEL HIGHJACKED IN THE FORM OF NVIDEA−KERNEL, ATI−KERNEL, OR MORE COMPLET HP/COMPAQ−KERNEL, DELL_KERNEL, GATEWAY−KERNEL, LIVING IN THE DUST THE LESS COMPLIENT RH−KERNEL, MANDRAKE_KERNEL, SUSE_KERNEL, VMWARE_KERNEL!!!!...

IS THE LINUX KERNEL HEADING FOR A "ABISM", OR IS MY IMAGINATION?????????

Hardware vendor supplied modules

(Posted Feb 8, 2003 15:12 UTC (Sat) by **jschrod**) (Post reply)

Is there any vendor who doesn't distribute source code and config/{autoconf,version}.h to its kernel? I always thought, on different machine, one installs said kernel source code, adds the config files, and there you go. At least, this worked for me all the time.

Cheers, Joachim

Driver porting: compiling external modules

(Posted Feb 13, 2003 0:17 UTC (Thu) by **pedretti**) (Post reply)

This is probably a dumb question, but what is the module that you insmod? When I compile the hello_world I get a hello_world.o and a hello_world.ko. Both of these can be insmod'ed but the hello_world.o give an error "no version magic, tainting kernel" −− so I assume the hello_world.ko is the one I want to use?

Driver porting: compiling external modules

(Posted Apr 30, 2003 20:19 UTC (Wed) by **krobidea**) (Post reply)

It seems that this 2.5 method of compiling external modules does not work on older kernel versions. I tried the example .c and Makefile, and got the following results:

− RedHat 7.3 out of the box (2.4.18−3 w/.config or any .o files). Compilation fails, thinking that module support is not compiled in.

− Version 2.4.20 I built and installed. Compilation failed, *** No rule to make target `modules'. This happens when make changed back to my working directory and did a make xxx modules. The kernel source base Makefile appears to be different in the 2.5.x versions.

So, will external modules require that the source be present, built and running? I don't believe any RedHat distributions will work.

What about the 2.4 or 2.2 target modules issue?

Driver porting: miscellaneous changes

This article is part of the LWN Porting Drivers to 2.6 series.
This article serves as a sort of final resting place for various small changes in the kernel programming API which do not fit elsewhere.

No more kdev_t

The `kdev_t` type has been removed from the kernel; everything which works with device numbers should now use the `dev_t` type. As part of this change, `dev_t` has been expanded to 32 bits; 12 bits for the major number, and 20 for the minor number.

If your driver uses the `i_rdev` field of the `inode` structure, there are a couple of new macros you can use:

```
unsigned int iminor(struct inode *inode);
unsigned int imajor(struct inode *inode);
```

Consider using these while fixing your code; the next time the type of `i_rdev` changes, you will be happier.

malloc.h

The include file `<linux/malloc.h>` has long been a synonym for `<linux/slab.h>`. In 2.5, `malloc.h` was removed, and all code should include `slab.h` directly.

Designated initializers

In the 2.3 development cycle, much effort went into converting code to the (non−standard) gcc designated initializer format for structures:

```
static struct some_structure = {
    field1:  value,
    field2:  value
};
```

In 2.5, the decision was made to go to the ANSI C standard initializer format, and all of that code was reworked again. The non−standard format still works, for now, but it is worth the effort to make the change anyway. The standard format looks like:

```
static struct some_structure = {
    .field1 = value,
    .field2 = value,
    ...
};
```

The min() and max() macros

A common step in the development of most C programmers, shortly after they learn to include `stdio.h` seems to be the definition of macros like:

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

In 2.5, it was noted that a number of kernel developers had seemingly not moved beyond that stage, and there was an unbelievable number of `min()` and `max()` definitions sprinkled throughout the kernel source. These definitions were not only redundant − they were unsafe. A `max()` macro as defined above has the usual problem with side effects; it also is not type−safe, especially when signed and unsigned values are used.

Linus initially added his own definition of `min()` and `max()` which added a third argument − an explicit type. That change upset people badly enough that some put substantial amounts of time into developing two−argument versions that are type and side−effect safe. The result now lives in `<linux/kernel.h>`, and should be used in preference to any locally−defined version.

Dependent read barriers

(Added February 24, 2003).

On most architectures, reads from memory will not be reordered across a data dependency. Consider this code fragment:

```
int *a, *b, x;

a = b;
x = *a;
```

Here, you would expect that `*a` would yield the value pointed to by `b`; it would be surprising to have the read of `*a` reordered in front of the assignment to `a`. Some architectures can do just that sort of reordering, however. One could ensure that such reordering does not happen by inserting a `rmb()` (read memory barrier) between the two assignment. But that would reduce performance needlessly on many systems. So a new barrier, `read_barrier_depends()`, has been added in this case. It should only be used in situations where data dependencies exist. On architectures where data dependencies will force ordering, `read_barrier_depends()` will do nothing. On other architectures, it expands into a regular read barrier. See this patch posting for more information.

User−mode helpers

(Added February 28, 2003).

The prototype of `call_usermodehelper()` has changed:

```
int call_usermodehelper(char *path, char **argv, char **envp,
                        int wait);
```

The new `wait` flag controls whether `call_usermodehelper()` returns before the user−mode process exits. If `wait` is set to a non−zero value, the function will wait for the process to finish its work, and the return value will be what the process itself returned. Otherwise the return is immediate, and the return value

only indicates whether the user−mode process was successfully started or not.

Note that the older `exec_usermodehelper()` function has been removed from the 2.5 kernel.

New request_module() prototype

(Added May 20, 2003).

As of 2.5.70, the prototype of `request_module()` has changed. This function now takes a printf−style argument list. As a result, code which used to look like:

```
char module_name[32];

sprintf(module_name, "foo−device−%d", number);
request_module(module_name);
```

can now be rewritten as:

```
request_module("foo−device−%d", number);
```

Most in−kernel code has already been changed to do things the new way.

devfs

We'll not go into the details here, but it is worth noting that devfs has been through extensive changes, and will likely see more changes yet before 2.6.0 is released. If your driver uses devfs, it will certainly need updating.

Note also that devfs has been officially marked as "deprecated."

Post a comment

Driver porting: miscellaneous changes, designated inits

(Posted Mar 14, 2003 23:28 UTC (Fri) by **dougg**) (Post reply)

With regard to designated initializers they were introduced in C99. Formally you are correct as C89 has been displaced as the ANSI (and ISO) C standard by C99 (ISO9899−1999) but many people may not realize that.

Driver porting: more module changes

This article is part of the LWN Porting Drivers to 2.6 series.

The first article in this series noted a couple of changes that result from the new, kernel−based module loader. In particular, explicit `module_init()` and `module_exit()` declarations are now necessary. Quite a few other things have changed as well, however; this article will summarize the most important of those changes.

Module parameters

The old `MODULE_PARM` macro, which used to specify parameters which can be passed to the module at load time, is no more. The new parameter declaration scheme add type safety and new functionality, but at the cost of breaking compatibility with older modules.

Modules with parameters should now include `<linux/moduleparam.h>` explicitly. Parameters are then declared with `module_param`:

```
module_param(name, type, perm);
```

Where `name` is the name of the parameter (and of the variable holding its value), `type` is its type, and `perm` is the permissions to be applied to that parameter's sysfs entry. The `type` parameter can be one of `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `charp`, `bool` or `invbool`. That type will be verified during compilation, so it is no longer possible to create confusion by declaring module parameters with mismatched types. The plan is for module parameters to appear automatically in sysfs, but that feature had not been implemented as of 2.6.0−test9; for now, the safest alternative is to set `perm` to zero, which means "no sysfs entry."

If the name of the parameter as seen outside the module differs from the name of the variable used to hold the parameter's value, a variant on `module` param may be used:

```
module_param_named(name, value, type, perm);
```

Where `name` is the externally−visible name and `value` is the internal variable.

String parameters will normally be declared with the `charp` type; the associated variable is a `char` pointer which will be set to the parameter's value. If you need to have a string value copied directly into a `char` array, declare it as:

```
module_param_string(name, string, len, perm);
```

Usually, `len` is best specified as `sizeof(string)`.

Finally, array parameters (supplied at module load time as a comma−separated list) may be declared with:

```
module_param_array(name, type, num, perm);
```

The one parameter not found in `module_param()` (`num`) is an output parameter; if a value for `name` is supplied when the module is loaded, `num` will be set to the number of values given. This macro uses the declared length of the array to ensure that it is not overrun if too many values are provided.

As an example of how the new module parameter code works, here is a paramaterized version of the "hello world" module shown previously:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/moduleparam.h>

MODULE_LICENSE("Dual BSD/GPL");

/*
 * A couple of parameters that can be passed in: how many times we say
 * hello, and to whom.
 */
static char *whom = "world";
module_param(whom, charp, 0);
static int howmany = 1;
module_param(howmany, int, 0);


static int hello_init(void)
{
    int i;
    for (i = 0; i <howmany; i++)
        printk(KERN_ALERT "(%d) Hello, %s\n", i, whom);
    return 0;
}

static void hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel %s\n", whom);
}

module_init(hello_init);
module_exit(hello_exit);
```

Inserting this module with a command like:

```
insmod ./hellop.ko howmany=2 whom=universe
```

causes the message "hello, universe" to show up twice in the system logfile.

Module aliases

A module alias is an alternative name by which a loadable module can be known. These aliases are typically defined in `/etc/modules.conf`, but many of them are really a feature of the module itself. In 2.6, module aliases can be embedded with a module's source. Simply add a line like:

```
MODULE_ALIAS("alias-name");
```

The module use count

In 2.4 and prior kernels, modules maintained their "use count" with macros like `MOD_INC_USE_COUNT`. The use count, of course, is intended to prevent modules from being unloaded while they are being used. This method was always somewhat error prone, especially when the use count was manipulated inside the module itself. In the 2.6 kernel, reference counting is handled differently.

The only safe way to manipulate the count of references to a module is outside of the module's code. Otherwise, there will always be times when the kernel is executing within the module, but the reference count is zero. So this work has been moved outside of the modules, and life is generally easier for module authors.

Any code which wishes to call into a module (or use some other module resource) must first attempt to increment that module's reference count:

```
int try_module_get(&module);
```

It is also necessary to look at the return value from `try_module_get()`; a zero return means that the try failed, and the module should not be used. Failure can happen, for example, when the module is in the process of being unloaded.

A reference to a module can be released with `module_put()`.

Again, modules will not normally have to manage their own reference counts. The only exception may be if a module provides a reference to an internal data structure or function that is not accounted for otherwise. In that (rare) case, a module could conceivably call `try_module_get()` on itself.

As of this writing, modules are considered "live" during initialization, meaning that a `try_module_get()` will succeed at that time. There is still talk of changing things, however, so that modules are not accessible until they have completed their initialization process. That change will help prevent a whole set of race conditions that come about when a module fails initialization, but it also creates difficulties for modules which have to be available early on. For example, block drivers should be available to read partition tables off of disks when those disks are registered, which usually happens when the module is initializing itself. If the policy changes and modules go back off−limits during initialization, a call to a function like `make_module_live()` may be required for those modules which must be available sooner. (**Update** 2.6.0−test9: this change has not happened and seems highly unlikely at this point).

Finally, it is not entirely uncommon for driver authors to put in a special `ioctl()` function which sets the module use count to zero. Sometimes, during module development, errors can leave the module reference count in a state where it will never reach zero, and there was no other way to get the kernel to unload the module. The new module code supports forced unloading of modules which appear to have outstanding references − *if* the `CONFIG_MODULE_FORCE_UNLOAD` option has been set. Needless to say, this option should only be used on development systems, and, even then, with great caution.

Exporting symbols

For the most part, the exporting of symbols to the rest of the kernel has not changed in 2.6 − except, of course, for the fact that any user of those symbols should be using `try_module_get()` first. In older kernels, however, a module which did not arrange things otherwise would implicitly export all of its symbols. In 2.6, things no longer work that way; only symbols which have explicitly been exported are visible to the rest of the kernel.

Chances are that change will cause few problems. When you get a chance, however, you can remove `EXPORT_NO_SYMBOLS` lines from your module source. Exporting no symbols is now the default, so `EXPORT_NO_SYMBOLS` is a no−op.

The 2.4 `inter_module_` functions have been deprecated as unsafe. The `symbol_get()` function exists for the cases when normal symbol linking does not work well enough. Its use requires setting up weak references at compile time, and is beyond the scope of this document; there are no users of `symbol_get()` in the 2.6.0−test9 kernel source.

Kernel version checking

2.4 and prior kernels would include, in each module, a string containing the version of the kernel that the module was compiled against. Normally, modules would not be loaded if the compile version failed to match the running kernel.

In 2.5, things still work mostly that way. The kernel version is loaded into a separate, "link−once" ELF section, however, rather than being a visible variable within the module itself. As a result, multi−file modules no longer need to define `__NO_VERSION__` before including `<linux/module.h>`.

The new "version magic" scheme also records other information, including the compiler version, SMP status, and preempt status; it is thus able to catch more incompatible situations than the old scheme did.

Module symbol versioning ("modversions") has been completely reworked for the 2.6 kernel. Module authors who use the makefiles shipped with the kernel (and that is about the only way to work now) will find that dealing with modversions has gotten easier than before. The `#define` hack which tacked checksums onto kernel symbols has gone away in favor of a scheme which stores checksum information in a separate ELF section.

Post a comment

Driver porting: more module changes

(Posted Feb 14, 2003 8:41 UTC (Fri) by **rusty**) (Post reply)

Regarding module_param(): MODULE_PARM() will certainly stay throughout the 2.6 series, so no need to change existing code just yet.

However, the real power of module_param() is that you can easily build your own types: i.e. it is extensible. One way to do this is to use the underlying module_param_call() macro, but a more convenient way goes like this:

```
#define param_check_mytype(name, p) /* optional typechecking */
int param_set_mytype(const char *val, struct kernel_param *kp) ...
int param_get_mytype(char *buffer, struct kernel_param *kp) ...
```

Then you can use it as follows:

```
module_param(myparam, mytype, 000);
```

This is extremely useful for implementing things like ranges, or any holes in the provided macros.

The other issue to note is that (as an unrelated bonus) the module_param parameters are also available as boot parameters, prefixed with the module name and a "." (− and _ can be used interchangably here). Previously __setup() was needed for boot parameters, and MODULE_PARM for module parameters.

Hope that helps!
Rusty.

Driver porting to 2.5.59: 'MOD_IN_USE' undeclared
(Posted Feb 27, 2003 8:12 UTC (Thu) by **hsieng**) (Post reply)

Hello,

While compiling a device driver in 2.5.59, I get the following error:

dmodule.c: 'MOD_IN_USE' undeclared

Please help on how to solve this problem.

Thank you.

Se−Hsieng

Driver porting: more module changes
(Posted Feb 21, 2003 6:35 UTC (Fri) by **mmarq**) (Post reply)

Well, I wonder why bother, ...perahps going for one of the kernel maintainers listed emails,...but then, if anyone can answer to this question i'l be very much appreciataded...it's very important to me!!!

Does this >"new "version magic" scheme also records other information, including the compiler version, SMP status, and preempt status; it is thus able to catch more incompatible situations than the old scheme did"<, storm, is to be lefted in production 2.6 kernels, or is it going to be somehow abstracted??

Better put!, does a "stupid trying to be a hacker" programmer like me, if by any change achieves the feat of getting a working kernel hardware driver module for the most trivial piece of hardware in the world, say, compiled with kernel 2.6.6 and gcc 3.4, "be in a versioning storm" and have to fix the module for it to recompile with preempt enable, and fix the module againg for it to recompile with SMP enabled, and for Kernel 2.6.7 the same, and for gcc3.4.1 the same,..., meaning that 20 kernel versions and 3 compiler versions ahead i would have to make 180 fixes!?????

whaw!!...without meaning any disrespect, it sure is a job for a full time payed hacker!

Well!!...If the answer is yes than is very sad "for me and for thousands like me,...because i have a very small shop that transforms old gear and also produce new linux servers and firewall appliences, and dont have the many to hire an army of programmers.

Its double sad, because after 10 years of dreams and foughts, the kernel is sleeping to the hands of IBM, HP and the like, that not only have the resources for thousands of programmers, but also control the machines on witch full working kernels can be deployed.

Any way, i respectfuly be waiting for a answer.... thank you.

Driver porting: more module changes

(Posted Feb 21, 2003 10:17 UTC (Fri) by **rrw**) (Post reply)

I think that there is one problem with this scheme of parameters. In case of numbers you cannot distinguish if the user didn't set the parameter or if he has set it to zero. Perheaps example will explain it better:

module_param(howmany, int, 0);

if you load this module with howmany=0, howmany will be zero. But if you omit howmany on modload commandline, howmany will be also zero.

One should pass the numeric arguments to the code as (int *), not the (int). Then NULL will be ``not set".

Driver porting: The seq_file interface

This article is part of the LWN Porting Drivers to 2.6 series.

There are numerous ways for a device driver (or other kernel component) to provide information to the user or system administrator. One very useful technique is the creation of virtual files, in `/proc` or elsewhere. Virtual files can provide human−readable output that is easy to get at without any special utility programs; they can also make life easier for script writers. It is not surprising that the use of virtual files has grown over the years.

Creating those files correctly has always been a bit of a challenge, however. It is not that hard to make a `/proc` file which returns a string. But life gets trickier if the output is long − anything greater than an application is likely to read in a single operation. Handling multiple reads (and seeks) requires careful attention to the reader's position within the virtual file − that position is, likely as not, in the middle of a line of output. The Linux kernel is full of `/proc` file implementations that get this wrong.

The 2.6 kernel contains a set of functions (implemented by Alexander Viro) which are designed to make it easy for virtual file creators to get it right. This interface (called "seq_file") is not strictly a 2.6 feature − it was also merged into 2.4.15. But 2.6 is where the feature is starting to see serious use, so it is worth describing here.

The seq_file interface is available via `<linux/seq_file.h>`. There are three aspects to seq_file:

- An iterator interface which lets a virtual file implementation step through the objects it is presenting.
- Some utility functions for formatting objects for output without needing to worry about things like output buffers.
- A set of canned `file_operations` which implement most operations on the virtual file.

We'll look at the seq_file interface via an extremely simple example: a loadable module which creates a file called `/proc/sequence`. The file, when read, simply produces a set of increasing integer values, one per line. The sequence will continue until the user loses patience and finds something better to do. The file is seekable, in that one can do something like the following:

```
dd if=/proc/sequence of=out1 count=1
dd if=/proc/sequence skip=1 out=out2 count=1
```

Then concatenate the output files `out1` and `out2` and get the right result. Yes, it is a thoroughly useless module, but the point is to show how the mechanism works without getting lost in other details. (Those wanting to see the full source for this module can find it here).

The iterator interface

Modules implementing a virtual file with seq_file must implement a simple iterator object that allows stepping through the data of interest. Iterators must be able to move to a specific position − like the file they

implement − but the interpretation of that position is up to the iterator itself. A seq_file implementation that is formatting firewall rules, for example, could interpret position *N* as the *N*th rule in the chain. Positioning can thus be done in whatever way makes the most sense for the generator of the data, which need not be aware of how a position translates to an offset in the virtual file. The one obvious exception is that a position of zero should indicate the beginning of the file.

The `/proc/sequence` iterator just uses the count of the next number it will output as its position.

Four functions must be implemented to make the iterator work. The first, called `start()` takes a position as an argument and returns an iterator which will start reading at that position. For our simple sequence example, the `start()` function looks like:

```
static void *ct_seq_start(struct seq_file *s, loff_t *pos)
{
        loff_t *spos = kmalloc(sizeof(loff_t), GFP_KERNEL);
        if (! spos)
                return NULL;
        *spos = *pos;
        return spos;
}
```

The entire data structure for this iterator is a single `loff_t` value holding the current position. There is no upper bound for the sequence iterator, but that will not be the case for most other seq_file implementations; in most cases the `start()` function should check for a "past end of file" condition and return `NULL` if need be.

For more complicated applications, the `private` field of the `seq_file` structure can be used. There is also a special value whch can be returned by the `start()` function called `SEQ_START_TOKEN`; it can be used if you wish to instruct your `show()` function (described below) to print a header at the top of the output. `SEQ_START_TOKEN` should only be used if the offset is zero, however.

The next function to implement is called, amazingly, `next()`; its job is to move the iterator forward to the next position in the sequence. The example module can simply increment the position by one; more useful modules will do what is needed to step through some data structure. The `next()` function returns a new iterator, or `NULL` if the sequence is complete. Here's the example version:

```
static void *ct_seq_next(struct seq_file *s, void *v, loff_t *pos)
{
        loff_t *spos = (loff_t *) v;
        *pos = ++(*spos);
        return spos;
}
```

The `stop()` function is called when iteration is complete; its job, of course, is to clean up. If dynamic memory is allocated for the iterator, `stop()` is the place to return it.

```
static void ct_seq_stop(struct seq_file *s, void *v)
{
        kfree (v);
}
```

Finally, the `show()` function should format the object currently pointed to by the iterator for output. It should return zero, or an error code if something goes wrong. The example module's `show()` function is:

```
static int ct_seq_show(struct seq_file *s, void *v)
```

```
{
        loff_t *spos = (loff_t *) v;
        seq_printf(s, "%Ld\n", *spos);
        return 0;
}
```

We will look at `seq_printf()` in a moment. But first, the definition of the seq_file iterator is finished by creating a `seq_operations` structure with the four functions we have just defined:

```
static struct seq_operations ct_seq_ops = {
        .start = ct_seq_start,
        .next  = ct_seq_next,
        .stop  = ct_seq_stop,
        .show  = ct_seq_show
};
```

This structure will be needed to tie our iterator to the `/proc` file in a little bit.

It's worth noting that the interator value returned by `start()` and manipulated by the other functions is considered to be completely opaque by the seq_file code. It can thus be anything that is useful in stepping through the data to be output. Counters can be useful, but it could also be a direct pointer into an array or linked list. Anything goes, as long as the programmer is aware that things can happen between calls to the iterator function. However, the seq_file code (by design) will not sleep between the calls to `start()` and `stop()`, so holding a lock during that time is a reasonable thing to do. The seq_file code will also avoid taking any other locks while the iterator is active.

Formatted output

The seq_file code manages positioning within the output created by the iterator and getting it into the user's buffer. But, for that to work, that output must be passed to the seq_file code. Some utility functions have been defined which make this task easy.

Most code will simply use `seq_printf()`, which works pretty much like `printk()`, but which requires the `seq_file` pointer as an argument. It is common to ignore the return value from `seq_printf()`, but a function producing complicated output may want to check that value and quit if something non−zero is returned; an error return means that the seq_file buffer has been filled and further output will be discarded.

For straight character output, the following functions may be used:

```
int seq_putc(struct seq_file *m, char c);
int seq_puts(struct seq_file *m, const char *s);
int seq_escape(struct seq_file *m, const char *s, const char *esc);
```

The first two output a single character and a string, just like one would expect. `seq_escape()` is like `seq_puts()`, except that any character in `s` which is in the string `esc` will be represented in octal form in the output.

There is also a function for printing filenames:

```
int seq_path(struct seq_file *m, struct vfsmount *mnt,
             struct dentry *dentry, char *esc);
```

Here, `mnt` and `dentry` indicate the file of interest, and `esc` is a set of characters which should be escaped in the output. This function is more suited to filesystem code than device drivers, however.

Making it all work

So far, we have a nice set of functions which can produce output within the seq_file system, but we have not yet turned them into a file that a user can see. Creating a file within the kernel requires, of course, the creation of a set of `file_operations` which implement the operations on that file. The seq_file interface provides a set of canned operations which do most of the work. The virtual file author still must implement the `open()` method, however, to hook everything up. The open function is often a single line, as in the example module:

```
static int ct_open(struct inode *inode, struct file *file)
{
        return seq_open(file, &ct_seq_ops);
};
```

Here, the call to `seq_open()` takes the `seq_operations` structure we created before, and gets set up to iterate through the virtual file.

On a successful open, `seq_open()` stores the `struct seq_file` pointer in `file->private_data`. If you have an application where the same iterator can be used for more than one file, you can store an arbitrary pointer in the `private` field of the `seq_file` structure; that value can then be retrieved by the iterator functions.

The other operations of interest − `read()`, `llseek()`, and `release()` − are all implemented by the seq_file code itself. So a virtual file's `file_operations` structure will look like:

```
static struct file_operations ct_file_ops = {
        .owner   = THIS_MODULE,
        .open    = ct_open,
        .read    = seq_read,
        .llseek  = seq_lseek,
        .release = seq_release
};
```

The final step is the creation of the `/proc` file itself. In the example code, that is done in the initialization code in the usual way:

```
static int ct_init(void)
{
        struct proc_dir_entry *entry;

        entry = create_proc_entry("sequence", 0, NULL);
        if (entry)
                entry->proc_fops = &ct_file_ops;
        return 0;
}

module_init(ct_init);
```

And that is pretty much it.

The extra−simple version

For extremely simple virtual files, there is an even easier interface. A module can define only the `show()` function, which should create all the output that the virtual file will contain. The file's `open()` method then calls:

```
int single_open(struct file *file,
                int (*show)(struct seq_file *m, void *p),
                void *data);
```

When output time comes, the `show()` function will be called once. The `data` value given to `single_open()` can be found in the `private` field of the `seq_file` structure. When using `single_open()`, the programmer should use `single_release()` instead of `seq_release()` in the `file_operations` structure to avoid a memory leak.

Post a comment

Driver porting: The seq_file interface

(Posted Nov 14, 2003 13:13 UTC (Fri) by **laf0rge**) (Post reply)

After using this article as an example to port the /proc/net/ip_conntrack interface over to seq_file, and about 5 hours of crashing/rebooting/debugging, I have to admit that there are some shortcomings in it.

Some hints for other people, so they don't fall into the same pits as I did:

1) If you allocate something in ct_seq_start(), the place to free it is _NOT_ in ct_seq_stop(). This is because ct_seq_stop() is even called if ct_seq_start() returns an error (Like ERR_PTR(−ENOMEM)). You would then end up calling kfree(ERR_PTR(−ENOMEM)) which your mm subsystem doesn't really like. I am now kfree()ing in ct_seq_next(), just before it returns with NULL at the end of the table.

2) If you take a lock in ct_seq_start(), do it unconditionally as the first thing. Even if ct_seq_start() fails, ct_seq_stop() is called. In ct_seq_stop() you have no idea of knowing if ct_seq_start() failed or not − so you will unconditionally unlock.

Driver porting: The seq_file interface

(Posted Nov 15, 2003 1:53 UTC (Sat) by **giraffedata**) (Post reply)

I am now kfree()ing in ct_seq_next(), just before it returns with NULL at the end of the table

Seems like that would be a problem if the user chooses not to read all the way to EOF.

This just sounds like a basic bug in the seq_file interface. If ct_seq_start() fails, it should be ct_seq_start's responsibility to not change any state, and thus ct_seq_stop doesn't need to be, and should not be, called. After all, does a POSIX program call close(−1) when open() fails?

Driver porting: Timekeeping changes

This article is part of the LWN Porting Drivers to 2.6 series.

One might be tempted to think that the basic task of keeping track of the time would not change that much from one kernel to the next. And, in fact, most kernel code which worries about times (and time intervals) will likely work unchanged in the 2.6 kernel. Code which gets into the details of how the kernel manages time may well need to adapt to some changes, however.

Internal clock frequency

One change which *shouldn't* be problematic for most code is the change in the internal clock rate on the x86 architecture. In previous kernels, `HZ` was 100; in 2.6 it has been bumped up to 1000. If your code makes any assumptions about what HZ really was (or, by extension, what `jiffies` really signified), you may have to make some changes now. For what it's worth, as of 2.6.0−test9, the default values of HZ in the mainline kernel source (which sometimes lags the architecture−specific trees) is as follows: **Alpha**: 1024/1200; **ARM**: 100/128/200/1000; **CRIS**: 100; **i386**: 1000; **IA−64**: 1024; **M68K**: 100; **M68K−nommu**: 50−1000; **MIPS**: 100/128/1000; **MIPS64**: 100; **PA−RISC**: 100/1000; **PowerPC32**: 100; **PowerPC64**: 1000; **S/390**: 100; **SPARC32**: 100; **SPARC64**: 100; **SuperH**: 100/1000; **UML**: 100; **v850**: 24−100; **x86−64**: 1000.

Kernel time variables

When the internal clock rate on a 32−bit system is set to 1000, the classic 32−bit `jiffies` variable will overflow in just over 49 days. Overflows could always happen on systems with a long uptime, but, when it took well over a year of uptime, it was a relatively rare occurrence − even on Linux systems. It is not uncommon at all, however, for a system to be up for more than 50 days. In most cases, having `jiffies` wrap around is not a real problem; it can be inconvenient for tasks like process accounting, however. So the 2.5 kernel has a new counter called `jiffies_64`. With 64 bits to work with, `jiffies_64` will not wrap around in a time frame that need concern most of us − at least until some future kernel starts using a gigahertz internal clock.

For what it's worth, on most architectures, the classic, 32−bit `jiffies` variable is now just the least significant half of `jiffies_64`.

Note that, on 32−bit systems, a 64−bit `jiffies` value raises concurrency issues. It is deliberately not declared as a `volatile` value (for performance reasons), so the possibility exists that code like:

```
u64 my_time = jiffies_64;
```

could get an inconsistent version of the variable, where the top and bottom halves do not match. To avoid this possibility, code accessing `jiffies_64` should use `xtime_lock`, which is the new seqlock type as of 2.5.60. In most cases, though, it will be easier to just use the convenience function provided by the kernel:

```
#include <linux/jiffies.h>
```

```
u64 my_time = get_jiffies_64();
```

Users of the internal `xtime` variable will notice a couple of similar changes. One is that `xtime`, too, is now protected by `xtime_lock` (as it is in 2.4 as of 2.4.10), so any code which plays around with disabling interrupts or such before accessing `xtime` will need to change. The best solution is probably to use:

```
struct timespec current_kernel_time(void);
```

which takes care of locking for you. `xtime` also now is a `struct timespec` rather than `struct timeval`; the difference being that the sub−second part is called `tv_nsec`, and is in nanoseconds.

Timers

The kernel timer interface is essentially unchanged since 2.4, with one exception. The new function:

```
void add_timer_on(struct timer_list *timer, int cpu);
```

will cause the timer function to run on the given CPU with the expiration time hits.

Delays

The 2.5 kernel includes a new macro `ndelay()`, which delays for a given number of nanoseconds. It can be useful for interactions with hardware which insists on very short delays between operations. On most architectures, however, `ndelay(n)` is equal to `udelay(1)` for waits of less than one microsecond.

POSIX clocks

The POSIX clocks patch (merged into 2.5.63) is beyond the scope of this article. If you are working with a device which can provide an interesting time service (high resolution or high accuracy), you may want to consider using it to drive a POSIX clock. Look into `kernel/posix−timers.c` for more information.

Post a comment

Driver porting: Timekeeping changes
(Posted Feb 27, 2003 9:13 UTC (Thu) by **ekj**) (Post reply)

With 64 bits to work with, jiffies_64 will not wrap around in a time frame that need concern most of us − at least until some future kernel starts using a gigahertz internal clock.

Actually, even with a Ghz internal clock a 64−bit counter will still need about 600 years before it ticks over. Even with a 100Ghz internal clock we will experience wrap−around only once every 6 years. Few systems stay up that long.

Driver porting: Timekeeping changes
(Posted Feb 27, 2003 19:59 UTC (Thu) by **cpeterso**) (Post reply)

Someone posted a patch on LKML a few years ago and again recently, that started jiffies at −5 minutes

instead of 0. That is, the patch would force a jiffy wraparound 5 minutes after boot, every time for all systems. Kernel and driver wraparound bugs would have to be fixed.

Does anyone know if there are plans to merge this in Linux 2.5?

Driver porting: mutual exclusion with seqlocks

This article is part of the LWN Porting Drivers to 2.6 series.

The 2.5.60 kernel added a new type of lock called a "seqlock." Seqlocks are a specialized primitive intended for the following sort of situation:

- A small amount of data is to be protected.
- That data is simple (no pointers), and is frequently accessed.
- Access to the data does not create side effects.
- It is important that writers not be starved for access.

The situation for which seqlocks were originally designed is to control access to system time variables − `jiffies_64` and `xtime`. Those variables are constantly being read, so that action should be fast. It is also important, however, that the update of those variables, which happens in the timer interrupt, not have to wait while the readers clear out.

Seqlocks consist of a regular spinlock and an integer "sequence" count. They may be declared and initialized in two ways, as follows:

```
#include <linux/seqlock.h>

seqlock_t lock1 = SEQLOCK_UNLOCKED;
seqlock_t lock2;

seqlock_init(&lock2);
```

Writers must take out exclusive access before making changes to the protected data. The usual series of events is something like:

```
seqlock_t the_lock = SEQLOCK_UNLOCKED;
/* ... */

write_seqlock(&the_lock);
/* Make changes here */
write_sequnlock(&the_lock);
```

The call to `write_seqlock()` locks the spinlock and increments the sequence number. When the work is done, `write_sequnlock()` increments the sequence number again, then releases the spinlock.

Read access to the data uses no locking at all; instead, the reader uses the lock's sequence number to detect access collisions with a writer and retry the read if necessary. The code tends to look like:

```
unsigned int seq;

do {
    seq = read_seqbegin(&the_lock);
```

```
    /* Make a copy of the data of interest */
} while read_seqretry(&the_lock, seq);
```

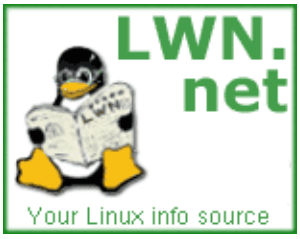
The call to `read_seqretry()` makes a couple of simple checks. If the initial sequence number obtained from `read_seqbegin()` is odd, it means that a writer was in the middle of updating the data when the reader began reading. If the initial number does not match the seqlock's sequence number at the end, then a writer showed up in the middle of the process. Either way, the data obtained could be inconsistent, and the reader must go around and try again. In the most common case, though, no collision will occur, and the reader gets very fast access with no locking or retries required.

Of course, the usual variants on the locking primitives exist for exclusion of local interrupts or bottom halves; for reference, here's the full set:

```
void write_seqlock(seqlock_t *sl);
void write_sequnlock(seqlock_t *sl);
int write_tryseqlock(seqlock_t *sl);
void write_seqlock_irqsave(seqlock_t *sl, long flags);
void write_sequnlock_irqrestore(seqlock_t *sl, long flags);
void write_seqlock_irq(seqlock_t *sl);
void write_sequnlock_irq(seqlock_t *sl);
void write_seqlock_bh(seqlock_t *sl);
void write_sequnlock_bh(seqlock_t *sl);

unsigned int read_seqbegin(seqlock_t *sl);
int read_seqretry(seqlock_t *sl, unsigned int iv);
unsigned int read_seqbegin_irqsave(seqlock_t *sl, long flags);
int read_seqretry_irqrestore(seqlock_t *sl, unsigned int iv, long flags);
```

No comments have been posted. Post one now



Driver porting: low−level memory allocation

This article is part of the LWN Porting Drivers to 2.6 series.

The 2.5 development series has brought relatively few changes to the way device drivers will allocate and manage memory. In fact, most drivers should work with no changes in this regard. There are a few improvements that have been made, however, that are worth a mention. These include some changes to page allocation, and the new "mempool" interface. Note that the allocation and management of per−CPU data is described in a separate article.

Allocation flags

The old `<linux/malloc.h>` include file is gone; it is now necessary to include `<linux/slab.h>` instead.

The `GFP_BUFFER` allocation flag is gone (it was actually removed in 2.4.6). That will bother few people, since almost nobody used it. There are two new flags which have replaced it: `GFP_NOIO` and `GFP_NOFS`. The `GFP_NOIO` flag allows sleeping, but no I/O operations will be started to help satisfy the request. `GFP_NOFS` is a bit less restrictive; some I/O operations can be started (writing to a swap area, for example), but no filesystem operations will be performed.

For reference, here is the full set of allocation flags, from the most restrictive to the least::

- `GFP_ATOMIC`: a high−priority allocation which will not sleep; this is the flag to use in interrupt handlers and other non−blocking situations.
- `GFP_NOIO`: blocking is possible, but no I/O will be performed.
- `GFP_NOFS`: no filesystem operations will be performed.
- `GFP_KERNEL`: a regular, blocking allocation.
- `GFP_USER`: a blocking allocation for user−space pages.
- `GFP_HIGHUSER`: for allocating user−space pages where high memory may be used.

The `__GFP_DMA` and `__GFP_HIGHMEM` flags still exist and may be added to the above to direct an allocation to a particular memory zone. In addition, 2.5.69 added some new modifiers:

- `__GFP_REPEAT` This flag tells the page allocater to "try harder," repeating failed allocation attempts if need be. Allocations can still fail, but failure should be less likely.
- `__GFP_NOFAIL` Try even harder; allocations with this flag must not fail. Needless to say, such an allocation could take a long time to satisfy.
- `__GFP_NORETRY` Failed allocations should not be retried; instead, a failure status will be returned to the caller immediately.

The `__GFP_NOFAIL` flag is sure to be tempting to programmers who would rather not code failure paths, but that temptation should be resisted most of the time. Only allocations which truly cannot be allowed to fail

should use this flag.

Page−level allocation

For page−level allocations, the `alloc_pages()` and `get_free_page()` functions (and variants) exist as always. They are now defined in `<linux/gfp.h>`, however, and there are a few new ones as well. On NUMA systems, the allocator will do its best to allocate pages on the same node as the caller. To explicitly allocate pages on a different NUMA node, use:

```
struct page *alloc_pages_node(int node_id,
                              unsigned int gfp_mask,
                              unsigned int order);
```

The memory allocator now distinguishes between "hot" and "cold" pages. A hot page is one that is likely to be represented in the processor's cache; cold pages, instead, must be fetched from RAM. In general, it is preferable to use hot pages whenever possible, since they are already cached. Even if the page is to be overwritten immediately (usually the case with memory allocations, after all), hot pages are better − overwriting them will not push some other, perhaps useful, data from the cache. So `alloc_pages()` and friends will return hot pages when they are available.

On occasion, however, a cold page is preferable. In particular, pages which will be overwritten via a DMA read from a device might as well be cold, since their cache data will be invalidated anyway. In this sort of situation, the `__GFP_COLD` flag should be passed into the allocation.

Of course, this whole scheme depends on the memory allocator knowing which pages are likely to be hot. Normally, order−zero allocations (i.e. single pages) are assumed to be hot. If you know the state of a page you are freeing, you can tell the allocator explicitly with one of the following:

```
void free_hot_page(struct page *page);
void free_cold_page(struct page *page);
```

These functions only work with order−zero allocations; the hot/cold status of larger blocks is not tracked.

Memory pools

Memory pools were one of the very first changes in the 2.5 series − they were added to 2.5.1 to support the new block I/O layer. The purpose of mempools is to help out in situations where a memory allocation must succeed, but sleeping is not an option. To that end, mempools pre−allocate a pool of memory and reserve it until it is needed. Mempools make life easier in some situations, but they should be used with restraint; each mempool takes a chunk of kernel memory out of circulation and raises the minimum amount of memory the kernel needs to run effectively.

To work with mempools, your code should include `<linux/mempool.h>`. A mempool is created with `mempool_create()`:

```
mempool_t *mempool_create(int min_nr,
                          mempool_alloc_t *alloc_fn,
                          mempool_free_t *free_fn,
                          void *pool_data);
```

Here, `min_nr` is the minimum number of pre−allocated objects that the mempool tries to keep around. The mempool defers the actual allocation and deallocation of objects to user−supplied routines, which have the

following prototypes:

```
typedef void *(mempool_alloc_t)(int gfp_mask, void *pool_data);
typedef void (mempool_free_t)(void *element, void *pool_data);
```

The allocation function should take care not to sleep unless `__GFP_WAIT` is set in the given `gfp_mask`. In all of the above cases, `pool_data` is a private pointer that may be used by the allocation and deallocation functions.

Creators of mempools will often want to use the slab allocator to do the actual object allocation and deallocation. To do that, create the slab, pass it in to `mempool_create()` as the `pool_data` value, and give `mempool_alloc_slab` and `mempool_free_slab` as the allocation and deallocation functions.

A mempool may be returned to the system by passing it to `mempool_destroy()`. You must have returned all items to the pool before destroying it, or the mempool code will get upset and oops the system.

Allocating and freeing objects from the mempool is done with:

```
void *mempool_alloc(mempool_t *pool, int gfp_mask);
void mempool_free(void *element, mempool_t *pool);
```

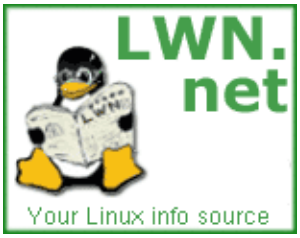
`mempool_alloc()` will first call the pool's allocation function to satisfy the request; the pre−allocated pool will only be used if the allocation function fails. The allocation may sleep if the given `gfp_mask` allows it; it can also fail if memory is tight and the preallocated pool has been exhausted.

Finally, a pool can be resized, if necessary, with:

```
int mempool_resize(mempool_t *pool, int new_min_nr, int gfp_mask);
```

This function will change the size of the pre−allocated pool, using the given `gfp_mask` to allocate more memory if need be. Note that, as of 2.5.60, `mempool_resize()` is disabled in the source, since nobody is actually using it.

No comments have been posted. Post one now



Driver porting: per−CPU variables

This article is part of the LWN Porting Drivers to 2.6 series.

The 2.6 kernel makes extensive use of per−CPU data − arrays containing one object for each processor on the system. Per−CPU variables are not suitable for every task, but, in situations where they can be used, they do offer a couple of advantages:

- Per−CPU variables have fewer locking requirements since they are (normally) only accessed by a single processor. There is nothing other than convention that keeps processors from digging around in other processors' per−CPU data, however, so the programmer must remain aware of what is going on.
- Nothing destroys cache performance as quickly as accessing the same data from multiple processors. Restricting each processor to its own area eliminates cache line bouncing and improves performance.

Examples of per−CPU data in the 2.6 kernel include lists of buffer heads, lists of hot and cold pages, various kernel and networking statistics (which are occasionally summed together into the full system values), timer queues, and so on. There are currently no drivers using per−CPU values, but some applications (i.e. networking statistics for high−bandwidth adapters) might benefit from their use.

The normal way of creating per−CPU variables at compile time is with this macro (defined in `<linux/percpu.h>`):

```
DEFINE_PER_CPU(type, name);
```

This sort of definition will create `name`, which will hold one object of the given `type` for each processor on the system. If the variables are to be exported to modules, use:

```
EXPORT_PER_CPU_SYMBOL(name);
EXPORT_PER_CPU_SYMBOL_GPL(name);
```

If you need to link to a per−CPU variable defined elsewhere, a similar macro may be used:

```
DECLARE_PER_CPU(type, name);
```

Variables defined in this way are actually an array of values. To get at a particular processor's value, the `per_cpu()` macro may be used; it works as an lvalue, so so code like the following works:

```
DEFINE_PER_CPU(int, mypcint);

per_cpu(mypcint, smp_processor_id()) = 0;
```

The above code can be dangerous, however. Accessing per−CPU variables can often be done without locking, since each processor has its own private area to work in. The 2.6 kernel is preemptible, however, and that adds a couple of challenges. Since kernel code can be preempted, it is possible to encounter race conditions with other kernel threads running on the same processor. Also, accessing a per−CPU variable requires knowing

which processor you are running on; it would not do to be preempted and moved to a different CPU between looking up the processor ID and accessing a per−CPU variable.

For both of the above reasons, kernel preemption usually must be disabled when working with per−CPU data. The usual way of doing this is with the `get_cpu_var` and `put_cpu_var` macros. `get_cpu_var` works as an lvalue, so it can be assigned to, have its address taken, etc. Perhaps the simplest example of the use of these macros can be found in `net/socket.c`:

```
get_cpu_var(sockets_in_use)++;
put_cpu_var(sockets_in_use);
```

Of course, since preemption is disabled between the calls, the code should take care not to sleep. Note that there is no version of these macros for access to another CPU's data; cross−processor access to per−CPU data requires explicit locking arrangements.

It is also possible to allocate per−CPU variables dynamically. Simply use these functions:

```
void *alloc_percpu(type);
void free_percpu(const void *);
```

`alloc_percpu()` will allocate one object (of the given type) for each CPU on the system; the allocated storage will be zeroed before being returned to the caller.

There is another set of macros which may be used to access per−CPU data obtained with `kmalloc_percpu()`. At the lowest level, you may use:

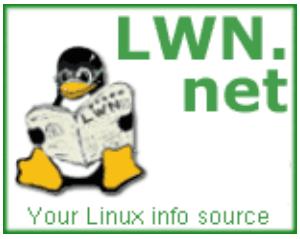
```
per_cpu_ptr(void *ptr, int cpu)
```

which returns (without any concurrency control) a pointer to the per−CPU data for the given `cpu`. For access to a local processor's data, with preemption disabled, use:

```
get_cpu_ptr(ptr)
put_cpu_ptr(ptr)
```

With the usual proviso that you do not sleep between the two.

No comments have been posted. Post one now



Driver porting: the preemptible kernel

This article is part of the LWN Porting Drivers to 2.6 series.

One significant change introduced in 2.6 is the preemptible kernel. Previously, a thread running in kernel space would run until it returned to user mode or voluntarily entered the scheduler. In 2.6, if preemption is configured in, kernel code can be interrupted at (almost) any time. As a result, the number of challenges relating to concurrency in the kernel goes up. But this is actually not that big a deal for code which was written to handle SMP properly − most of the time. If you have not yet gotten around to implementing proper locking for your 2.4 driver, kernel preemption should give you yet another reason to get that job done.

The preemptible kernel means that your driver code can be preempted whenever the scheduler decides there is something more important to do. "Something more important" could include re−entering your driver code in a different thread. There is one big, important exception, however: preemption will not happen if the currently−running code is holding a spinlock. Thus, the precautions which are taken to ensure mutual exclusion in the SMP environment also work with preemption. So most (properly written) code should work correctly under preemption with no changes.

That said, code which makes use of per−CPU variables should take extra care. A per−CPU variable may be safe from access by other processors, but preemption could create races on the same processor. Code using per−CPU variables should, if it is not already holding a spinlock, disable preemption if the possibility of concurrent access exists. Usually, macros like `get_cpu_var()` should be used for this purpose.

Should it be necessary to control preemption directly (something that should happen rarely), some macros in `<linux/preempt.h>` will come in helpful. A call to `preempt_disable()` will keep preemption from happening, while `preempt_enable()` will make it possible again. If you want to re−enable preemption, but don't want to get preempted immediately (perhaps because you are about to finish up and reschedule anyway), `preempt_enable_no_resched()` is what you need.

Normally, rescheduling by preemption takes place without any effort on the part of the code which is being scheduled out. Occasionally, however, long−running code may want to check explicitly to see whether a reschedule is pending. Code which defers rescheduling with `preempt_enable_noresched()` may want to perform such checks, for example, when it reaches a point where it can afford to sleep for a while. For such situations, a call to `preempt_check_resched()` will suffice.

One interesting side−effect of the preemption work is that it is now much easier to tell if a particular bit of kernel code is running within some sort of critical section. A single variable in the task structure now tracks the preemption, interrupt, and softirq states. A new macro, `in_atomic()`, tests all of these states and returns a nonzero value if the kernel is running code that should complete without interruption. Among other things, this macro has been used to trap calls to functions that might sleep from atomic contexts.

Post a comment

Driver porting: the preemptible kernel

(Posted Mar 30, 2003 17:27 UTC (Sun) by **goatbar**) (Post reply)

I want to start off by saying how much I appreciate these series of articles. Thank you!

I haven't even come close to using 2.5 yet, but I have been using kernels >= 2.4.18 with the RML preempt patch. Is this patch the same or pretty close to the preemptible code in the 2.5.xx kernels?

Driver porting: sleeping and waking up

This article is part of the LWN Porting Drivers to 2.6 series.

Contrary to expectations, the classic functions `sleep_on()` and `interruptible_sleep_on()` were not removed in the 2.5 series. It seems that they are still needed in a few places where (1) taking them out is quite a bit of work, and (2) they are actually used in a way that is safe. Most authors of kernel code should, however, pretend that those functions no longer exist. There are very few situations in which they can be used safely, and better alternatives exist.

`wait_event()` and friends

Most of those alternatives have been around since 2.3 or earlier. In many situations, one can use the `wait_event()` macros:

```
DECLARE_WAIT_QUEUE_HEAD(queue);

wait_event(queue, condition);
int wait_event_interruptible (queue, condition);
```

These macros work the same as in 2.4: `condition` is a boolean condition which will be tested within the macro; the wait will end when the condition evaluates true.

It is worth noting that these macros have moved from `<linux/sched.h>` to `<linux/wait.h>`, which seems a more sensible place for them. There is also a new one:

```
int wait_event_interruptible_timeout(queue, condition, timeout);
```

which will terminate the wait if the timeout expires.

`prepare_to_wait()` and friends

In many situations, `wait_event()` does not provide enough flexibility − often because tricky locking is involved. The alternative in those cases has been to do a full "manual" sleep, which involves the following steps (shown here in a sort of pseudocode, of course):

```
DECLARE_WAIT_QUEUE_HEAD(queue);
DECLARE_WAITQUEUE(wait, current);

for (;;) {
    add_wait_queue(&queue, &wait);
    set_current_state(TASK_INTERRUPTIBLE);
    if (condition)
        break;
    schedule();
    remove_wait_queue(&queue, &wait);
```

```
        if (signal_pending(current))
            return -ERESTARTSYS;
    }
    set_current_state(TASK_RUNNING);
```

A sleep coded in this manner is safe against missed wakeups. It is also a fair amount of error−prone boilerplate code for a very common situation. In 2.6, a set of helper functions has been added which makes this task easier. The modern equivalent of the above code would look like:

```
    DECLARE_WAIT_QUEUE_HEAD(queue);
    DEFINE_WAIT(wait);

    while (! condition) {
        prepare_to_wait(&queue, &wait, TASK_INTERRUPTIBLE);
        if (! condition)
            schedule();
        finish_wait(&queue, &wait)
    }
```

`prepare_to_wait_exclusive()` should be used when an exclusive wait is needed. Note that the new macro `DEFINE_WAIT()` is used here, rather than `DECLARE_WAITQUEUE()`. The former should be used when the wait queue entry is to be used with `prepare_to_wait()`, and should probably *not* be used in other situations unless you understand what it is doing (which we'll get into next).

Wait queue changes

In addition to being more concise and less error prone, `prepare_to_wait()` can yield higher performance in situations where wakeups happen frequently. This improvement is obtained by causing the process to be removed from the wait queue immediately upon wakeup; that removal keeps the process from seeing multiple wakeups if it doesn't otherwise get around to removing itself for a bit.

The automatic wait queue removal is implemented via a change in the wait queue mechanism. Each wait queue entry now includes its own "wake function," whose job it is to handle wakeups. The default wake function (which has the surprising name `default_wake_function()`), behaves in the customary way: it sets the waiting task into the `TASK_RUNNING` state and handles scheduling issues. The `DEFINE_WAIT()` macro creates a wait queue entry with a different wake function, `autoremove_wake_function()`, which automatically takes the newly−awakened task out of the queue.

And that, of course, is how `DEFINE_WAIT()` differs from `DECLARE_WAITQUEUE()` − they set different wake functions. How the semantics of the two differ is not immediately evident from their names, but that's how it goes. (The new runtime initialization function `init_wait()` differs from the older `init_waitqueue_entry()` in exactly the same way).

If need be, you can define your own wake function − though the need for that should be quite rare (about the only user, currently, is the support code for the `epoll()` system calls). The wake function has this prototype:

```
    typedef int (*wait_queue_func_t)(wait_queue_t *wait,
                                     unsigned mode, int sync);
```

A wait queue entry can be given a different wakeup function with:

```
    void init_waitqueue_func_entry(wait_queue_t *queue,
                                   wait_queue_func_t func);
```

One other change that most programmers won't notice: a bunch of wait queue cruft from 2.4 (two different kinds of wait queue lock, wait queue debugging) has been removed from 2.6.

Post a comment

Driver porting: sleeping and waking up

(Posted Feb 28, 2003 13:49 UTC (Fri) by **ortalo**) (Post reply)

As I understand it (possibly with several misunderstanding), these wait and wake up functions primarily address userspace waits.
What about kernel−internal waiting? (aka: Is it reasonable to call wake_up() from an interrupt handler?)

A practical example (the one I'm concerned with): sending out graphical display lists via DMA to a modern graphics hardware accelerator.
In this context, one process sends memory buffers to the kernel and occasionally waits when too many buffers are already submitted for execution: in this case I have to use the wait functions you describe, that's okay.
Now, later on, the displays lists get executed by the graphic hardware and the end of each display list generates an IRQ. The IRQ handler needs to acknowledge the interrupt, but also to submit the next display list for execution (if any, remember that userspace may submit display lists faster than the hardware executes them and some of them may be in queue inside the kernel).
It can work like this. But it is not very clean to directly submit the next display list to the hardware *from the interrupt handler*. For example, one would like to do a time−consuming checking step on the display list before submission; worse, if AGP is involved (sooner or later) one will want to mess up with the AGP translation tables. Doing this from an interrupt handler does not seem very reasonable (time consuming and possibly disruptive work).
In an ideal world (ie: when I'm knowledgeable enough), a sort of kernel thread (one per graphic processor) would exist for submitting work to the hardware and the interrupt handler would only signal to that kernel thread the end of execution. Would it be possible to use the wait queues you describe for synchronization between the kernel thread and the interrupt handler? How would you recommend to adress such an issue?

Rodolphe

Driver porting: sleeping and waking up

(Posted Feb 28, 2003 15:04 UTC (Fri) by **corbet**) (Post reply)

The best option, of course, would be to check the display lists at the time they are submitted by the user space process. That way you can return an immediate error if something is wrong.

If you have other stuff that needs doing, a kernel thread could certainly do it. I would recommend a look at the workqueue interface, however, as a relatively easy way to do this sort of deferred processing. You can feed a task into a workqueue from an interrupt handler and it can execute at leisure, in process context, later on.

Driver porting: sleeping and waking up

(Posted Mar 6, 2003 10:40 UTC (Thu) by **driddoch**) (Post reply)

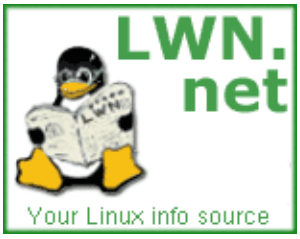
And just to illustrate how error prone the manual sleep interface is, the example has a bug: You must remove yourself from the wait queue before you return −ERESTARTSYS.

No doubt this was deliberate ;−)

Driver porting: sleeping and waking up

(Posted Mar 6, 2003 13:43 UTC (Thu) by **corbet**) (Post reply)

Of course it was deliberate. I decided that maybe it was too subtle a way of making my point, though, so I fixed it...:)



Driver porting: dealing with interrupts

This article is part of the LWN Porting Drivers to 2.6 series.

The kernel's handling of device interrupts has been massively reworked in the 2.6 series. Fortunately, very few of those changes are visible to the rest of the kernel; most well−written code should "just work" (almost) under 2.6. There are, however, two important exceptions: the return type of interrupt handlers has changed, and drivers which depend on being able to globally disable interrupts will require some changes for 2.6.

Interrupt handler return values

Prior to 2.5.69, interrupt handlers returned `void`. There is, however, one useful thing that interrupt handlers can tell the kernel: whether the interrupt was something they could handle or not. If a device starts generating spurious interrupts, the kernel would like to respond by blocking interrupts from that device. If no interrupt handler for a given IRQ has been registered, the kernel knows that any interrupt on that number is spurious. When interrupt handlers exist, however, they must tell the kernel about spurious interrupts.

So, interrupt handlers now return an `irqreturn_t` value; `void` handlers will no longer compile. If your interrupt handler recognizes and handles a given interrupt, it should return `IRQ_HANDLED`. If it knows that the interrupt was not on a device it manages, it can return `IRQ_NONE` instead. The macro:

```
IRQ_RETVAL(handled)
```

can also be used; `handled` should be nonzero if the handler could deal with the interrupt. The "safe" value to return, if, for some reason you are not sure, is `IRQ_HANDLED`.

Disabling interrupts

In the 2.6 kernel, it is no longer possible to globally disable interrupts. In particular, the `cli()`, `sti()`, `save_flags()`, and `restore_flags()` functions are no longer available. Disabling interrupts across all processors in the system is simply no longer done. This behavior has been strongly discouraged for some time, so most code *should* have been converted by now.

The proper way to do this fixing, of course, is to figure out exactly which resources were being protected by disabling interrupts. Those resources can then be explicitly protected with spinlocks instead. The change is usually fairly straightforward, but it does require an understanding of what is really going on.

It is still possible to disable all interrupts locally with `local_save_flags()` or `local_irq_disable()`. A single interrupt can be disabled globally with `disable_irq()`. Some of the spinlock operations also disable interrupts on the local processor, of course. None of these functions are changed (at least, with regard to their external interface) since 2.4.

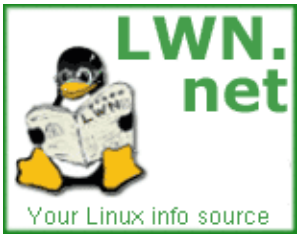
Various small changes

One function that *has* changed is `synchronize_irq()`. In 2.6, this function takes an integer IRQ number as a parameter. It spins until no interrupt handler is running for the given IRQ. If the IRQ is disabled prior to calling `synchronize_irq()`, the caller will know that no interrupt handler can be running after that call. The 2.6 version of `synchronize_irq()` only waits for handlers for the given IRQ number; it is no longer possible to wait until no interrupt handlers at all are running.

If your code has post−interrupt logic which runs as a bottom half, or out of a task queue, it will need to be changed for 2.6. Bottom halves are deprecated, and the task queue mechanism has been removed altogether. Post−interrupt processing should now be done using tasklets or work queues.

Finally, the declarations of `request_irq()` and `free_irq()` have moved from `<linux/sched.h>` to `<linux/interrupt.h>`.

No comments have been posted. Post one now



Driver porting: the workqueue interface.

This article is part of the LWN Porting Drivers to 2.6 series.

The longstanding task queue interface was removed in 2.5.41; in its place is a new "workqueue" mechanism. Workqueues are very similar to task queues, but there are some important differences. Among other things, each workqueue has one or more dedicated worker threads (one per CPU) associated with it. So all tasks running out of workqueues have a process context, and can thus sleep. Note that access to user space is not possible from code running out of a workqueue; there simply is no user space to access. Drivers can create their own work queues − with their own worker threads − but there is a default queue (for each processor) provided by the kernel that will work in most situations.

Workqueues are created with `create_workqueue`:

```
struct workqueue_struct *create_workqueue(const char *name);
```

The name of the queue is limited to ten characters; it is only used for generating the "command" for the kernel thread (which can be seen in `ps` or `top`).

Tasks to be run out of a workqueue need to be packaged in a `struct work_struct` structure. This structure may be declared and initialized at compile time as follows:

```
DECLARE_WORK(name, void (*function)(void *), void *data);
```

Here, `name` is the name of the resulting `work_struct` structure, `function` is the function to call to execute the work, and `data` is a pointer to pass to that function.

To set up a `work_struct` structure at run time, instead, use the following two macros:

```
INIT_WORK(struct work_struct *work,
          void (*function)(void *), void *data);
PREPARE_WORK(struct work_struct *work,
             void (*function)(void *), void *data);
```

The difference between the two is that `INIT_WORK` initializes the linked list pointers within the `work_struct` structure, while `PREPARE_WORK` changes only the function and data pointers. `INIT_WORK` must be used at least once before queueing the `work_struct` structure, but should *not* be used if the `work_struct` might already be in a workqueue.

Actually queueing a job to be executed is simple:

```
int queue_work(struct workqueue_struct *queue,
               struct work_struct *work);
int queue_delayed_work(struct workqueue_struct *queue,
                       struct work_struct *work,
                       unsigned long delay);
```

The second form of the call ensures that a minimum delay (in jiffies) passes before the work is actually executed. The return value from both functions is nonzero if the `work_struct` was actually added to `queue` (otherwise, it may have already been there and will not be added a second time).

Entries in workqueues are executed at some undefined time in the future, when the associated worker thread is scheduled to run (and after the delay period, if any, has passed). If it is necessary to cancel a delayed task, you can do so with:

```
int cancel_delayed_work(struct work_struct *work);
```

Note that this workqueue entry could actually be executing when `cancel_delayed_work()` returns; all this function will do is keep it from starting after the call.

To ensure that none of your workqueue entries are running, call:

```
void flush_workqueue(struct workqueue_struct *queue);
```

This would be a good thing to do, for example, in a device driver shutdown routine. Note that if the queue contains work with long delays this call could take a long time to complete. This function will *not* (as of 2.5.68) wait for any work entries submitted after the call was first made; you should ensure that, for example, any outstanding work queue entries will not resubmit themselves. You should also cancel any delayed entries (with `cancel_delayed_work()`) first if need be.

Work queues can be destroyed with:

```
void destroy_workqueue(struct workqueue_struct *queue);
```

This operation will flush the queue, then delete it.

Finally, for tasks that do not justify their own workqueue, a "default" work queue (called "`events`") is defined. `work_struct` structures can be added to this queue with:

```
int schedule_work(struct work_struct *work);
int schedule_delayed_work(struct work_struct *work, unsigned long delay);
```

Most users of workqueues can probably use the predefined queue, but one should bear in mind that it is a shared resource. Long delays in the worker function will slow down other users of the queue, and should be avoided. There is a `flush_scheduled_work()` function which will wait for everything on this queue to be executed. If your module uses the default queue, it should almost certainly call `flush_scheduled_work()` before allowing itself to be unloaded.

One final note: `schedule_work()`, `schedule_delayed_work()` and `flush_scheduled_work()` are exported to any modules which wish to use them. The other functions (for working with separate workqueues) are exported to GPL−licensed modules only.

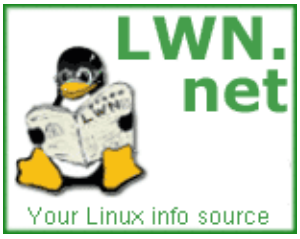
Post a comment

Driver porting: the workqueue interface.
(Posted Jun 17, 2003 21:45 UTC (Tue) by **btl**) (Post reply)

I'm trying to playing with workqueues... i've trouble on compiling
a little kernel module.

What include files do i need beside linux/workqueue.h?

I can't find where struct workqueue_struct is defined ;(



Driver porting: completion events

This article is part of the LWN Porting Drivers to 2.6 series.

Completions are a simple synchronization mechanism that is preferable to sleeping and waking up in some situations. If you have a task that must simply sleep until some process has run its course, completions can do it easily and without race conditions. They are not strictly a 2.6 feature, having been added in 2.4.7, but they merit a quick summary here.

A completion is, essentially, a one−shot flag that says "things may proceed." Working with completions requires including `<linux/completion.h>` and creating a variable of type `struct completion`. This structure may be declared and initialized statically with:

```
DECLARE_COMPLETION(my_comp);
```

A dynamic initialization would look like:

```
struct completion my_comp;

init_completion(&my_comp);
```

When your driver begins some process whose completion must be waited for, it's simply a matter of passing your completion event to `wait_for_completion()`:

```
void wait_for_completion(struct completion *comp);
```

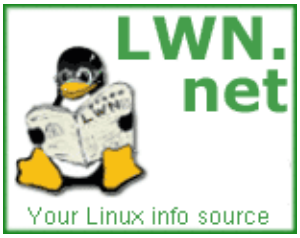
When some other part of your code has decided that the completion has happened, it can wake up anybody who is waiting with one of:

```
void complete(struct completion *comp);
void complete_all(struct completion *comp);
```

The first form will wake up exactly one waiting process, while the second will wake up all processes waiting for that event. Note that completions are implemented in such a way that they will work properly even if `complete()` is called before `wait_for_completion()`.

If you do not use `complete_all()`, you should be able to use a completion structure multiple times without problem. It does not hurt, however, to reinitialize the structure before each use − so long as you do it before initiating the process that will call `complete()`! The macro `INIT_COMPLETION()` can be used to quickly reinitialize a completion structure that has been fully initialized at least once.

No comments have been posted. Post one now



Driver porting: supporting asynchronous I/O

This article is part of the LWN Porting Drivers to 2.6 series.

One of the key "enterprise" features added to the 2.6 kernel is asynchronous I/O (AIO). The AIO facility allows user processes to initiate multiple I/O operations without waiting for any of them to complete; the status of the operations can then be retrieved at some later time. Block and network drivers are already fully asynchronous, and thus there is nothing special that needs to be done to them to support the new asynchronous operations. Character drivers, however, have a synchronous API, and will not support AIO without some additional work. For most char drivers, there is little benefit to be gained from AIO support. In a few rare cases, however, it may be beneficial to make AIO available to your users.

AIO file operations

The first step in supporting AIO (beyond including `<linux/aio.h>`) is the implementation of three new methods which have been added to the `file_operations` structure:

```
ssize_t (*aio_read) (struct kiocb *iocb, char __user *buffer,
                     size_t count, loff_t pos);
ssize_t (*aio_write) (struct kiocb *iocb, const char __user *buffer,
                      size_t count, loff_t pos);
int (*aio_fsync) (struct kiocb *, int datasync);
```

For most drivers, the real work will be in the implementation of `aio_read()` and `aio_write()`. These functions are analogous to the standard `read()` and `write()` methods, with a couple of changes: the `file` parameter has been replaced with an I/O control block (`iocb`), and they (usually) need not complete the requested operations immediately. The `iocb` argument can usually be treated as an opaque cookie used by the AIO subsystem; if you need the `struct file` pointer for this file descriptor, however, you can find it as `iocb->ki_filp`.

The `aio_` operations *can* be synchronous. One obvious example is when the requested operation can be completed without blocking. If the operation is complete before `aio_read()` or `aio_write()` returns, the return value should be the usual status or error code. So, the following `aio_read()` method, while being pointless, is entirely correct:

```
ssize_t my_aio_read(struct kiocb *iocb, char __user *buffer,
                    size_t count, loff_t pos)
{
    return my_read(iocb->ki_filp, buf, count, &pos);
}
```

In some cases, synchronous behavior may actually be required. The so-called "synchronous iocb's" allow the AIO subsystem to be used synchronously when need be. The macro:

```
is_sync_kiocb(struct kiocb *iocb)
```

will return a true value if the request must be handled synchronously.

In most cases, though, it is assumed that the I/O request will not be satisfied immediately by `aio_read()` or `aio_write()`. In this case, those functions should do whatever is required to get the operation started, then return `−EIOCBQUEUED`. Note that any work that must be done within the user process's context must be done before returning; you will not have access to that context later. In order to access the user buffer, you will probably need to either set up a DMA mapping or turn the `buffer` pointer into a series of `struct page` pointers before returning. Bear in mind also that there can be multiple asynchronous I/O requests active at any given time. A driver which implements AIO will have to include proper locking (and, probably queueing) to keep these requests from interfering with each other.

When the I/O operation completes, you must inform the AIO subsystem of the fact by calling `aio_complete()`:

```
int aio_complete(struct kiocb *iocb, long res, long res2);
```

Here, `iocb` is, of course, the IOCB you were given when the request was initiated. `res` is the usual result of an I/O operation: the number of bytes transfered, or a negative error code. `res2` is a second status value which will be returned to the user; currently (2.6.0−test9), callers of `aio_complete()` within the kernel always set `res2` to zero. `aio_complete()` can be safely called in an interrupt handler. Once you have called `aio_complete()`, you no longer own the IOCB or the user buffer, and should not touch them again.

The `aio_fsync()` method serves the same purpose as the `fsync()` method; its purpose is to ensure that all pending data are written to disk. As a general rule, device drivers will not need to implement `aio_fsync()`.

Cancellation

The design of the AIO subsystem includes the ability to cancel outstanding operations. Cancellation may occur as the result of a specific user−mode request, or during the cleanup of a process which has exited. It is worth noting that, as of 2.6.0−test9, no code in the kernel actually performs cancellation. So cancellation may not work properly, and the interface could change in the process of making it work. That said, here is how the interface looks today.

A driver which implements cancellation needs to implement a function for that purpose:

```
int my_aio_cancel(struct kiocb *iocb, struct io_event *event);
```

A pointer to this function can be stored into any IOCB which can be cancelled:

```
iocb->ki_cancel = my_aio_cancel;
```

Should the operation be cancelled, your cancellation function will be called with pointers to the IOCB and an `io_event` structure. If it is possible to cancel (or successfuly complete) the operation prior to returning from the cancellation function, the result of the operation should be stored into the `res` and `res2` fields of the `io_event` structure, and return zero. A non−zero return value from the cancellation function indicates that cancellation was not possible.

No comments have been posted. Post one now

Driver Porting: block layer overview

This article is part of the LWN Porting Drivers to 2.6 series.

The first big, disruptive changes to the 2.6 kernel came from the reworking of the block I/O layer. As one might guess, the result of all this work is a great many changes as seen by driver authors − or anybody else who works with block I/O. The transition may be painful for some, but it's worth it: the new block layer is easier to work with and offers much better performance than its predecessor.

Fully covering the changes that have been made will require a whole series of articles. So we'll start with an overview which highlights the major changes that have been made without getting into any sort of detail. Subsequent articles will fill in the rest.

Note that parts of the block layer remain volatile − this development is not yet complete. We'll keep up with further changes as they happen.

So, what has changed with the block layer?

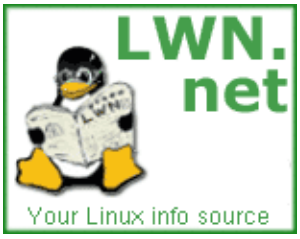
- A great deal of old cruft is gone. For example, it is no longer necessary to work with a whole set of global arrays within block drivers. These arrays (`blk_size`, `blksize_size`, `hardsect_size`, `read_ahead`, etc.) have simply vanished. The kernel still maintains much of the same information, of course, but the management of that information is much improved.
- As part of the cruft removal, most of the `<linux/blk.h>` macros (`DEVICE_NAME`, `DEVICE_NR`, `CURRENT`, `INIT_REQUEST`, etc.) have been removed; `<linux/blk.h>` is now empty. Any block driver which used these macros to implement its request loop will have to be rewritten. It is still possible to implement a simple request loop for straightforward devices where performance is not a big issue, but the mechanisms have changed.
- The `io_request_lock` is gone; locking is now done on a per−queue basis.
- Request queues have, in general, gotten more sophisticated. Quite a bit of work has been done in the area of fancy request scheduling (though drivers don't generally need to know about that). There is simple support for tagged command queueing, along with features like request barriers and queue−time device command generation. Request queues must be allocated dynamicly in 2.6.
- Buffer heads are no longer used in the block layer; they have been replaced with the new "`bio`" structure. The new representation of block I/O operations is designed for flexibility and performance; it encourages keeping large operations intact. Simple drivers can pretend that the `bio` structure does not exist, but most performance−oriented drivers − i.e. those that want to implement clustering and DMA − will need to be changed to work with `bio`s.

  One of the most significant features of the `bio` structure is that it represents I/O buffers directly with `page` structures and offsets, not in terms of kernel virtual addresses. By default, I/O buffers can be located in high memory, on the assumption that computers equipped with that much memory will also have reasonably modern I/O controllers. Support operations have been provided for tasks like `bio` splitting and the creation of DMA scatter/gather maps.
- Sector numbers can now be 64 bits wide, making it possible to support very large block devices.

- The rudimentary `gendisk` ("generic disk") structure from 2.4 has been greatly improved in 2.6; generic disks are now used extensively throughout the block layer. Among other things, each generic disk has its own `block_device_operations` structure; the operations are no longer directly associated with the driver. The most significant change for block driver authors, though, may be the fact that partition handling has been moved up into the block layer, and drivers no longer need know anything about partitions. That is, of course, the way things should always have been.

Subsequent articles will explore the above changes in depth; stay tuned.

No comments have been posted. Post one now



Driver porting: the gendisk interface

This article is part of the LWN Porting Drivers to 2.6 series.

The 2.4 kernel `gendisk` structure is used almost as an afterthought; its main purpose is to help in keeping track of disk partitions. In 2.6, the `gendisk` is at the core of the block subsystem; if you need to work with or find something out about a disk, `struct gendisk` probably has what you need. This article will cover the details of the `gendisk` structure from a disk driver's perspective. If you have not already read them, a quick look at the LWN block driver overview and simple block driver articles is probably worthwhile.

Gendisk initialization

The best way of looking at the contents of a `gendisk` structure from a block driver's point of view is to examine what that driver must do to set the structure up in the first place. If your driver makes a disk (or disk−like) device available to the system, it will have to provide an associated `gendisk` structure. (Note, however, that it is *not* necessary − or correct − to set up `gendisk` structures for disk partitions).

The first step is to create the `gendisk` structure itself; the function you need is `alloc_disk()` (which is declared in `<linux/genhd.h>`):

```
struct gendisk *alloc_disk(int minors);
```

The argument `minors` is the maximum number of minor numbers that this disk can have. Minor numbers correspond to partitions, of course (except the first, which is the "whole disk" device), so the value passed here controls the maximum number of partitions. If a single minor number is requested, the device cannot be partitioned at all. The return value is a pointer to the `gendisk` structure; the allocation can fail, so this value should always be checked against `NULL` before proceeding.

There are several fields of the `gendisk` structure which must be initialized by the block driver. They include:

`int major;`
: The major number of this device; either a static major assigned to a specific driver, or one that was obtained dynamically from `register_blkdev()`

`int first_minor;`
: The first minor device number corresponding to this disk. This number will be determined by how your driver divides up its minor number space.

`char disk_name[16];`
: The name of this disk (i.e. `hda`). This name is used in places like `/proc/partitions` and in creating a sysfs directory for the device.

`struct block_device_operations *fops;`

The device operations (open, release, ioctl, media_changed, and revalidate_disk) for this device. Each disk has its own set of operations in 2.6.

*struct request_queue *queue;*
: The request queue which will handle the list of pending operations for this disk. The queue must be created and initialized separately.

int flags;
: A set of flags controlling the management of this device. They include `GENHD_FL_REMOVABLE` for removable devices, `GENHD_FL_CD` for CDROM devices, and `GENHD_FL_DRIVERFS` which certainly means something interesting, but which is not actually used anywhere.

*void *private_data;*
: This field is reserved for the driver; the rest of the block subsystem will not touch it. Usually it holds a pointer to a driver−specific data structure describing this device.

The `gendisk` structure also holds the size of the disk, in sectors. As part of the initialization process, the driver should set that size with:

```
void set_capacity(struct gendisk *disk, sector_t size);
```

The `size` value should be in 512−byte sectors, even if the hardware sector size used by your device is different. For removable disks, setting its capacity to zero indicates to the block subsystem that there is currently no media present in the device.

Manipulating gendisks

Once you have your `gendisk` structure set up, you have to add it to the list of active disks; that is done with:

```
void add_disk(struct gendisk *disk);
```

After this call, your device is active. There are a few things worth keeping in mind about `add_disk()`:

- `add_disk()` can create I/O to the device (to read partition tables and such). You should not call `add_disk()` until your driver is sufficiently initialized to handle requests.
- If you are calling `add_disk()` in your driver initialization routine, you should not fail the initialization process after the first call.
- The call to `add_disk()` increments the disk's reference count; if the disk structure is ever to be released, the driver is responsible for decrementing that count (with `put_disk()`).

Should you need to remove a disk from the system, that is accomplished with:

```
void del_gendisk(struct gendisk *disk);
```

This function cleans up all of the information associated with the given `disk`, and generally removes it from the system. After a call to `del_gendisk()`, no more operations will be sent to the given device. Your driver's reference to the `gendisk` object remains, though; you must explicitly release it with:

```
void put_disk(struct gendisk *disk);
```

That call will cause the `gendisk` structure to be freed, as long as no other part of the kernel retains a reference to it.

Should you need to set a disk into a read−only mode, use:

```
void set_disk_ro(struct gendisk *disk, int flag);
```

If `flag` is nonzero, all partitions on the disk will be marked read−only. The kernel can track read−only status individually for each partition, but no utility function has been exported to manipulate that status for single partitions.

Partition management is handled within the block subsystem in 2.6; drivers need not worry about partitions at all. Should the need arise, the functions `add_partition()` and `delete_partition()` can be used to manipulate the (in−kernel) partition table directly. These functions are used in the generic block `ioctl()` code; there should be no need for a block driver to call them directly.

Registering block device number ranges

A call to `add_disk()` implicitly allocates the a set of minor numbers (under the given major number) from `first_minor` to `first_minor+minors−1`. If your driver must only respond to operations to disks that exist at initialization time, there is no need to worry further about number allocation. Even the traditional call to `register_blkdev()` is optional, and may be removed soon. Some drivers, however, need to be able to claim responsibility for a larger range of device numbers at initialization time.

If this is your case, the answer is to call `blk_register_region()`, which has this rather involved prototype:

```
void blk_register_region(dev_t dev,
                         unsigned long range,
                         struct module *module,
                         struct kobject *(*probe)(dev_t, int *, void *),
                         int (*lock)(dev_t, void *),
                         void *data);
```

Here, `dev` is a device number (created with `MKDEV()`) containing the major and first minor number of the region of interest; `range` is the number of minor numbers to allocate, `module` is the loadable module (if any) containing the driver, `probe` is a driver−supplied function to probe for a single disk, `lock` is a driver−supplied locking function, and `data` is a driver−private pointer which is passed to `probe()` and `lock()`.

When `blk_register_region()` is called, it simply makes a note of the desired region and returns. Note that there can be more than one registration within a specific region! At lookup time, the most "specific" registration (the one with the smallest `range`) wins.

At some point in the future, an attempt may be made to access a device number within the allocated region. At that point, there will be a call to the `lock()` function (if it was not passed as `NULL`) with the device number of interest. If `lock()` succeeds, `probe()` will be called to find the specific disk of interest. The full prototype of the probe function is:

```
struct kobject *(*probe)(dev_t dev, int *partition, void *data);
```

Here, `dev` is the device number of interest, `partition` is a pointer to a partition number (sort of), and `data` is the driver−private pointer passed to `blk_register_region()`. The partition number is actually just the offset into the allocated range; it's the minor number from `dev` with the beginning of the range subtracted.

The `probe()` function should attempt to identify a specific `gendisk` structure which corresponds to the requested number. If it is successful, it should return a pointer to the `kobject` structure contained within the `gendisk`. Kobjects are covered in a separate article; for all, all you really need to know is that you should call `get_disk()` with the `gendisk` structure as the argument, and return the value from `get_disk()` to the caller. The `probe()` function can also modify the partition number so that it corresponds to the actual partition offset in the returned device. If the function cannot handle the request at all, it can return `NULL`.

Some `probe()` functions do not, themselves, locate and initialize the device of interest. Instead, they call some other function to set in motion that whole process. For example, a number of `probe()` functions simply call `request_module()` in an attempt to load a module which can handle the device. In this mode of operation, the function should return `NULL`, which will cause the block layer to look at the device number allocations one more time. If a "better" allocation (with a smaller range) has happened in the mean time, the `probe()` function for the new driver will be called. So, for example, if a module is loaded which allocates a smaller device number range corresponding to the devices it actually implements, its `probe()` routine will be called on the next iteration.

Of course, there is the usual assocated unregister function:

```
void blk_unregister_region(dev_t dev, unsigned long range);
```

The next step

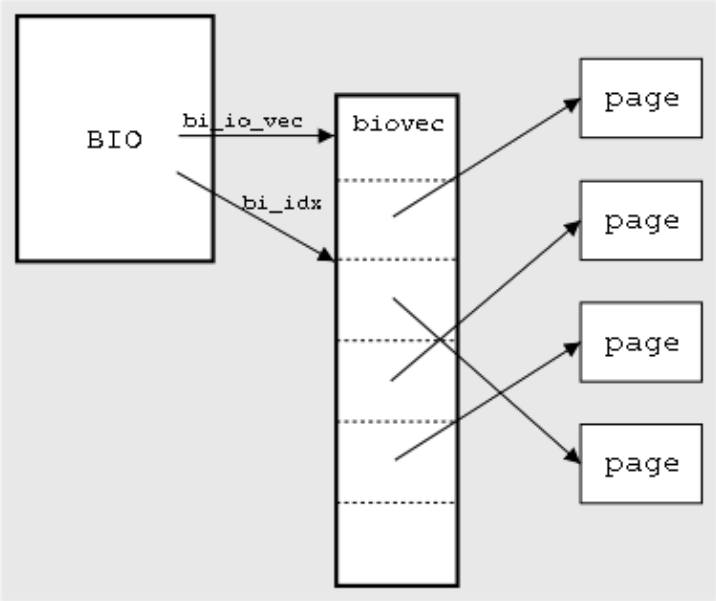
Once you have a handle on how the `gendisk` structure works, the next thing to do is to learn about BIO structures.

No comments have been posted. Post one now

Driver porting: the BIO structure

This article is part of the LWN Porting Drivers to 2.6 series.

The block layer in 2.4 (and prior) kernels was organized around the buffer head data structure. The limits of buffer heads have long been clear, however. It is hard to create a truly high−performance block I/O subsystem when the underlying buffer head structure forces each I/O request to be split into 512−byte chunks. So one of the first items on the 2.5 block "todo" list was the creation of a way to represent block I/O requests that supported higher performance and greater flexibility. The result was the new BIO structure.



BIO basics

As with most real−world code, the BIO structure incorporates a fair number of tricky details. The core of the structure (as defined in `<linux/bio.h>`) is not that complicated, however; it is as appears in the diagram to the right. The BIO structure itself contains the usual collection of housekeeping information, along with a pointer (`bi_io_vec`) pointing to an array of `bio_vec` structures. This array represents the (possibly multiple) segments which make up this I/O request. There is also an index (`bi_idx`) giving an offset into the `bi_io_vec` array; we'll get into its use shortly.

The `bio_vec` structure itself has a simple definition:

```
struct bio_vec {
    struct page     *bv_page;
    unsigned int    bv_len;
    unsigned int    bv_offset;
};
```

As is increasingly the case with internal kernel data structures, the BIO now tracks data buffers using `struct page` pointers. There are some implications of this change for driver writers:

- Data buffers for block transfers can be anywhere − kernel or user space. The driver author need not be concerned about the ultimate source or destination of the data.

- These buffers could be in high memory, unless the driver author has explicitly requested that bounce buffers be used (Request Queues I covers how to do that). The driver author cannot count on the existence of a kernel−space mapping for the buffer unless one has been created explicitly.
- More than ever, block I/O operations are scatter/gather operations, with data coming from multiple, dispersed buffers.

At first glance, the BIO structure may seem more difficult to work with than the old buffer head, which provided a nice kernel virtual address for a single chunk of data. Working with BIOs is not hard, however.

Getting request information from a BIO

A driver author could use the information above (along with the other BIO fields) to get the needed information out of the structure without too much trouble. As a general rule, however, direct access to the `bio_vec` array is discouraged. A set of accessor routines has been provided which hides the details of how the BIO structure works and eases access to that structure. Use of these routines will make the driver author's job easier, and, with luck, will enable a driver to keep working in the face of future block I/O changes.

So how does one get request information from the BIO structure? The beginning sector for the entire BIO is in the `bi_sector` field − there is no accessor function for that. The total size of the operation is in `bi_size` (in bytes). One can also get the total size in sectors with:

```
bio_sectors(struct bio *bio);
```

The function (macro, actually):

```
int bio_data_dir(struct bio *bio);
```

returns either `READ` or `WRITE`, depending on what type of operation is encapsulated by this BIO.

Almost everything else requires working through the `bio_vec` array. The encouraged way of doing that is to use the special `bio_for_each_segment` macro:

```
int segno;
struct bio_vec *bvec;

bio_for_each_segment(bvec, bio, segno) {
    /* Do something with this segment */
}
```

Within the loop, the integer variable `segno` will be the current index into the array, and `bvec` will point to the current `bio_vec` structure. Usually the driver programmer need not use either variable; instead, a new set of macros is available for use within this sort of loop:

`struct page *bio_page(struct bio *bio)`
: Returns a pointer to the current `page` structure.

`int bio_offset(struct bio *bio)`
: Returns the offset within the current page for this operation. Block I/O operations are often page−aligned, but that is not always the case.

`int bio_cur_sectors(struct bio *bio)`
: The number of sectors to transfer for this `bio_vec`.

*char *bio_data(struct bio *bio)*
: Returns the kernel virtual address for the data buffer. **Note** that this address will only exist if the buffer is not in high memory.

*char *bvec_kmap_irq(struct bio_vec *bvec, unsigned long *flags)*
: This function returns a kernel virtual address which can be used to access the data buffer pointed to by the given `bio_vec` entry; it also disables interrupts and returns an atomic kmap − so the driver should not sleep until `bvec_kunmap_irq()` has been called. Note that the `flags` argument is a pointer value, which is a departure for the usual convention for macros which disable interrupts.

*void bvec_kunmap_irq(char *buffer, unsigned long *flags);*
: Undo a mapping which was created with `bvec_kmap_irq()`.

*char *bio_kmap_irq(struct bio *bio, unsigned long *flags);*
: This function is a wrapper around `bvec_kmap_irq()`; it returns a mapping for the current `bio_vec` entry in the given `bio`. There is, of course, a corresponding `bio_kunmap_irq()`.

*char *__bio_kmap_atomic(struct bio *bio, int i, enum km_type type)*
: Use `kmap_atomic()` to obtain a kernel virtual address for the i^{th} buffer in the `bio`; the kmap slot designated by `type` will be used.

*void __bio_kunmap_atomic(char *addr, enum km_type type)*
: Return a kernel virtual address obtained with `__bio_kmap_atomic()`.

A little detail which is worth noting: all of `bio_data()`, `bvec_kmap_irq()`, and `bio_kmap_irq()` add the segment offset (`bio_offset(bio)`) to the address before returning it. It is tempting to add the offset separately, but that is an error which leads to weird problems. Trust me.

Completing I/O

Given the information from the BIO, each block driver should be able to arrange a transfer to or from its particular device. Note that a helper function (`blk_rq_map_sg()`) exists which makes it easy to set up DMA scatter/gather lists from a block request; we'll get into that when we look at request queue management.

When the operation is complete, the driver must inform the block subsystem of that fact. That is done with `bio_endio()`:

```
void bio_endio(struct bio *bio, unsigned int nbytes, int error);
```

Here, `bio` is the BIO of interest, `nbytes` is the number of bytes actually transferred, and `error` indicates the status of the operation; it should be zero for a successful transfer, and a negative error code otherwise.

Other BIO details

The `bi_private` field in the BIO structure is not used by the block subsystem, and is available for the owner of the structure to use. Drivers do *not* own BIOs passed in to their request function and should not touch `bi_private` there. If your driver creates its own BIO structures (using the functions listed below, usually), then the `bi_private` field in those BIOs is available to it.

As mentioned above, the `bi_idx` BIO field is an index into the `bi_io_vec` array. This index is maintained

for a couple of reasons. One is that it can be used to keep track of partially−complete operations. But this field (along with `bi_vcnt`, which says how many `bio_vec` entries are to be processed) can also be used to split a BIO into multiple chunks. Using this facility, a RAID or volume manager driver can "clone" a BIO into multiple structures all pointing at different parts of the `bio_vec` array. The operation is quick and efficient, and allows a large operation to be quickly dispatched across a number of physical drives.

To clone a BIO in this way, use:

```
struct bio *bio_clone(struct bio *bio, int gfp_mask);
```

`bio_clone()` creates a second BIO pointing to the same `bio_vec` array as the original. This function uses the given `gfp_mask` when allocating memory.

BIO structures contain reference counts; the structure is released when the reference count hits zero. Drivers normally need not manipulate BIO reference counts, but, should the need arise, functions exist in the usual form:

```
void bio_get(struct bio *bio);
void bio_put(struct bio *bio);
```

Numerous other functions exist for working with BIO structures; most of the functions not covered here are involved with creating BIOs. More information can be found in `<linux/bio.h>` and `block/biodoc.txt` in the kernel documentation directory.

Post a comment

bi_private

(Posted Mar 27, 2003 8:57 UTC (Thu) by **axboe**) (Post reply)

Good article, but one thing needs to be corrected concerning the use of bi_private. This field is _owned_ by whoever owns the bio, so it's definitely not for free use by the block driver (unless the block driver itself allocated the bio, of course)! In fact, this is a very important point as otherwise stacking drivers cannot work properly.

So in short, you may only look/modify bi_private if you are the owner of the bio.

bi_private

(Posted Mar 27, 2003 16:55 UTC (Thu) by **corbet**) (Post reply)

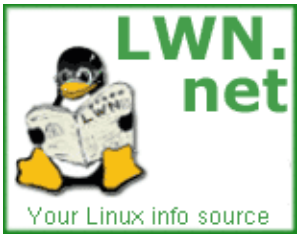
Hey, if that's the only thing I messed up, I'm happy. The article has been tweaked accordingly, thanks.

bi_private

(Posted Mar 28, 2003 22:13 UTC (Fri) by **Peter**) (Post reply)

> Good article, but one thing needs to be corrected concerning the use of bi_private.

Don't listen to this "axboe" character. He doesn't know anything about the BIO subsystem.



Driver porting: Request Queues I

This article is part of the LWN Porting Drivers to 2.5 series.

The simple block driver example earlier in this series showed how to write the simplest possible request function. Most block drivers, however, will need greater control over how requests are built and processed. This article will get into the details of how request queues work, with an emphasis on what every driver writer needs to know to process requests. A second article looks at some of the more advanced features of request queues in 2.6.

Request queues

Request queues are represented by a pointer to `struct request_queue` or to the typedef `request_queue_t`, defined in `<linux/blkdev.h>`. One request queue can be shared across multiple physical drives, but the normal usage is to create a separate queue for each drive. Request queues must be allocated and initialized by the block subsystem; this allocation (and initialization) is done by:

```
request_queue_t *blk_init_queue(request_fn_proc *request_fn,
                                spinlock_t *lock);
```

Here `request_fn` is the driver's function which will process requests, and `lock` is a spinlock which controls access to the queue. The return value is a pointer to the newly−allocated request queue if the initialization succeeded, or `NULL` otherwise. Since setting up a request queue requires memory allocation, failure is possible. A couple of other changes from 2.4 should be noted: a spinlock must be provided to control access to the queue (`io_request_lock` is no more), and there is no per−major "default" queue provided in 2.6.

When a driver is done with a request queue, it should pass it back to the system with:

```
void blk_cleanup_queue(request_queue_t *q);
```

Note that neither of these functions is normally called if a "make request" function is being used (make request functions are covered in part II).

Basic request processing

The request function prototype has not changed from 2.4; it gets the request queue as its only parameter. The queue lock will be held when the request function is called.

All request handlers, from the simplest to the most complicated, will find the next request to process with:

```
struct request *elv_next_request(request_queue_t *q);
```

The return value is the next request that should be processed, or `NULL` if the queue is empty. If you look through the kernel source, you will find references to `blk_queue_empty()` (or `elv_queue_empty()`), which tests the state of the queue. Use of this function in drivers is best avoided, however. In the future, it could be that a non−empty queue still has no requests that are ready to be processed.

In 2.4 and prior kernels, a block request contained one or more buffer heads with sectors to be transferred. In 2.6, a request contains a list of BIO structures instead. This list can be accessed via the `bio` member of the `request` structure, but the recommended way of iterating through a request's BIOs is instead:

```
struct bio *bio;

rq_for_each_bio(bio, req) {
    /* Process this BIO */
}
```

Drivers which use this macro are less likely to break in the future. Do note, however, that many drivers will never need to iterate through the list of BIOs in this way; for DMA transfers, use `bio_rq_map_sg()` (described below) instead.

As your driver performs the transfers described by the BIO structures, it will need to update the kernel on its progress. Note that drivers should *not* call `bio_endio()` as transfers complete; the block layer will take care of that. Instead, the driver should call `end_that_request_first()`, which has a different prototype in 2.6:

```
int end_that_request_first(struct request *req, int uptodate,
                           int nsectors);
```

Here, `req` is the request being handled, `uptodate` is nonzero unless an error has occurred, and `nsectors` is the number of sectors which were transferred. This function will clean up as many BIO structures as are covered by the given number of sectors, and return nonzero if any BIOs remain to be transferred.

When the request is complete (`end_that_request_first()` returns zero), the driver should clean up the request. The cleanup task involves removing the request from the queue, then passing it to `end_that_request_last()`, which is unchanged from 2.4. Note that the queue lock must be held when calling both of these functions:

```
void blkdev_dequeue_request(struct request *req);
void end_that_request_last(struct request *req);
```

Note that the driver can dequeue the request at any time (as long as it keeps track of it, of course). Drivers which keep multiple requests in flight will need to dequeue each request as it is passed to the drive.

If your device does not have predictable timing behavior, your driver should contribute its timing information to the system's entropy pool. That is done with:

```
void add_disk_randomness(struct gendisk *disk);
```

BIO walking

The "BIO walking" patch was added in 2.5.70. This patch adds some request queue fields and a new function to help complicated drivers keep track of where they are in a given request. Drivers using BIO walking will not use `rq_for_each_bio()`; instead, they rely upon the fact that the `cbio` field of the request structure

refers to the current, unprocessed BIO, `nr_cbio_segments` tells how many segments remain to be processed in that BIO, and `nr_cbio_sectors` tells how many sectors are yet to be transferred. The macro:

```
int blk_rq_idx(struct request *req)
```

returns the index of the next segment to process. If you need to access the current segment buffer directly (for programmed I/O, say), you may use:

```
char *rq_map_buffer(struct request *req, unsigned long *flags);
void rq_unmap_buffer(char *buffer, unsigned long flags);
```

These functions potentially deal with atomic kmaps, so the usual constraints apply: no sleeping while the mapping is in effect, and buffers must be mapped and unmapped in the same function.

When beginning I/O on a set of blocks from the request, your driver can update the current pointers with:

```
int process_that_request_first(struct request *req,
                               unsigned int nr_sectors);
```

This function will update the various `cbio` values in the request, but does not signal completion (you still need `end_that_request_first()` for that). Use of `process_that_request_first()` is optional; your driver may call it if you would like the block subsystem to track your current position in the request for I/O submission independently from how much of the request has actually been completed.

Barrier requests

Requests will come off the request queue sorted into an order that should give good performance. Block drivers (and the devices they drive) are free to reorder those requests within reason, however. Drives which support features like tagged command queueing and write caching will often complete operations in an order different from that in which they received the requests. Most of the time, this reordering leads to improved performance and is a good thing.

At times, however, it is necessary to inhibit this reordering. The classic example is that of journaling filesystems, which must be able to force journal entries to the disk before the operations they describe. Reordering of requests can undermine the filesystem integrity that a journaling filesystem is trying to provide.

To meet the needs of higher−level layers, the concept of a "barrier request" has been added to the 2.6 kernel. Barrier requests are marked by the `REQ_HARDBARRIER` flag in the request structure `flags` field. When your driver encounters a barrier request, it must complete that request (and all that preceded it) before beginning any requests after the barrier request. "Complete," in this case, means that the data has been physically written to the disk platter − not just transferred to the drive.

Tweaking request queue parameters

The block subsystem contains a long list of functions which control how I/O requests are created for your driver. Here's a few of them.

Bounce buffer control: in 2.4, the block code assumed that devices could not perform DMA to or from high memory addresses. When I/O buffers were located in high memory, data would be copied to or from low−memory "bounce" buffers; the driver would then operate on the low−memory buffer. Most modern devices can handle (at a minimum) full 32−bit DMA addresses, or even 64−bit addresses. For now, 2.6 will

still use bounce buffers for high−memory addresses. A driver can change that behavior with:

```
void blk_queue_bounce_limit(request_queue_t *q, u64 dma_addr);
```

After this call, any buffer whose physical address is at or above `dma_addr` will be copied through a bounce buffer. The driver can provide any reasonable address, or one of `BLK_BOUNCE_HIGH` (bounce high memory pages, the default), `BLK_BOUNCE_ANY` (do not use bounce buffers at all), or `BLK_BOUNCE_ISA` (bounce anything above the ISA DMA threshold).

Request clustering control. The block subsystem works hard to coalesce adjacent requests for better performance. Most devices have limits, however, on how large those requests can be. A few functions have been provided to instruct the block subsystem on how not to create requests which must be split apart again.

```
void blk_queue_max_sectors(request_queue_t *q, unsigned short max_sectors);
```

Sets the maximum number of sectors which may be transferred in a single request; default is 255. It is not possible to set the maximum below the number of sectors contained in one page.

```
void blk_queue_max_phys_segments(request_queue_t *q,
                                 unsigned short max_segments);
void blk_queue_max_hw_segments(request_queue_t *q,
                               unsigned short max_segments);
```

The maximum number of discontiguous physical segments in a single request; this is the maximum size of a scatter/gather list that could be presented to the device. The first functions controls the number of distinct memory segments in the request; the second does the same, but it takes into account the remapping which can be performed by the system's I/O memory management unit (if any). The default for both is 128 segments.

```
void blk_queue_max_segment_size(request_queue_t *q,
                                unsigned int max_size);
```

The maximum size that any individual segment within a request can be. The default is 65536 bytes.

```
void blk_queue_segment_boundary(request_queue_t *q,
                                unsigned long mask);
```

Some devices cannot perform transfers which cross memory boundaries of a certain size. If your device is one of these, you should call `blk_queue_segment_boundary()` with a `mask` indicating where the boundary is. If, for example, your hardware has a hard time crossing 4MB boundaries, `mask` should be set to `0x3fffff`. The default is `0xffffffff`.

Finally, some devices have more esoteric restrictions on which requests may or may not be clustered together. For situations where the above parameters are insufficient, a block driver can specify a function which can examine (and pass judgement on) each proposed merge.

```
typedef int (merge_bvec_fn) (request_queue_t *q, struct bio *bio,
                             struct bio_vec *bvec);
void blk_queue_merge_bvec(request_queue_t *q, merge_bvec_fn *fn);
```

Once the given `fn` is associated with this queue, it will be called every time a `bio_vec` entry `bvec` is being considered for addition to the given `bio`. It should return the number of bytes from `bvec` which can be added; zero should be returned if the new segment cannot be added at all. By default, there is no `merge_bvec_fn`.

Setting the hardware sector size. The old `hardsect_size` global array is gone and nobody misses it. Block drivers now inform the system of the underlying hardware's sector size with:

```
void blk_queue_hardsect_size(request_queue_t *q, unsigned short size);
```

The default is the usual 512−byte sector. There is one other important change with regard to sector sizes: your driver will always see requests expressed in terms of 512−byte sectors, regardless of the hardware sector size. The block subsystem will not generate requests which go against the hardware sector size, but sector numbers and counts in requests are always in 512−byte units. This change was required as part of the new centralized partition remapping.

DMA support

Most block I/O requests will come down to one more more DMA operations. The 2.6 block layer provides a couple of functions designed to make the task of setting up DMA operations easier.

```
void blk_queue_dma_alignment(request_queue_t *q, int mask);
```

This function sets a mask indicating what sort of memory alignment the hardware needs for DMA requests; the default is 511.

DMA operations to modern devices usually require the creation of a scatter/gather list of segments to be transferred. A block driver can create this "scatterlist" using the generic DMA support routines and the information found in the request. The block subsystem has made life a little easier, though. A simple call to:

```
int blk_rq_map_sg(request_queue_t *q, struct request *rq,
                  struct scatterlist *sg);
```

will construct a scatterlist for the given request; the return value is the number of entries in the resulting list. This scatterlist can then be passed to `pci_map_sg()` or `dma_map_sg()` in preparation for the DMA operation.

Going on

The second part of the request queue article series looks at command preparation, tagged command queueing, and writing drivers which do without a request queue altogether.

No comments have been posted. Post one now

Driver porting: Request Queues II

This article is part of the LWN Porting Drivers to 2.6 series.

This article continues the look at request queues in 2.6; if you've not read the first part in the request queue series, you may want to start there. Here we'll look at command pregeneration, tagged command queueing, and doing without a request queue altogether.

Command pregeneration

Traditionally, block drivers have prepared low−level hardware commands at the time a request is processed. There can be advantages to preparing commands at an earlier point, however. In 2.6, drivers which wish to prepare commands (or perform some other sort of processing) for requests before they hit the `request` function should set up a `prep_rq_fn` with this prototype:

```
typedef int (prep_rq_fn) (request_queue_t *q, struct request *rq);
```

This function should perform preparatory work on the given request `rq`. The 2.6 `request` structure includes a 16−byte `cmd` field where a pregenerated command can be stored; `rq->cmd_len` should be set to the length of that command. The prep function should return `BLKPREP_OK` (process the request normally), `BLKPREP_DEFER` (which defers processing of the command for now), or `BLKPREP_KILL` (which terminates the request with a failure status).

To add your prep function to a request queue, call:

```
void blk_queue_prep_rq(request_queue_t *q, prep_rq_fn *pfn);
```

The prep function is currently called out of `elv_next_request()` − immediately before the request is passed back to your driver. There is a possibility that, at some future point, the call to the prep function could happen earlier in the process.

Tagged command queueing

Tagged command queueing (TCQ) allows a block device to have multiple outstanding I/O requests, each identified by an integer "tag." TCQ can yield performance benefits; the drive generally knows best when it comes to figuring out which request should be serviced next. SCSI drivers in Linux have long supported TCQ, but each driver has included its own infrastructure for tag management. In 2.6, a simple tag management facility has been added to the block layer. The generic tag management code can make life easier, but it's important to understand how these functions interact with the request queue.

Drivers wishing to use tags should set things up with:

```
int blk_queue_init_tags(request_queue_t *q, int depth,
                        struct blk_queue_tag *tags);
```

This call should be made after the queue has been initialized. Here, `depth` is the maximum number of tagged commands which can be outstanding at any given time. The `tags` argument is a pointer to a `blk_queue_tag` structure which will be used to track the outstanding tags. Normally you can pass `tags` as `NULL`, and the block subsystem will allocate and initialize the structure for you. If you wish to share a structure (and, thus, the tags it represents) with another device, however, you can pass a pointer to the `blk_queue_tag` structure in the first queue when initializing the second. This call performs memory allocation, and will return a negative error code if that allocation failed.

A call to:

```
void blk_queue_free_tags(request_queue_t *q);
```

will clean up the TCQ infrastructure. This normally happens automatically when `blk_cleanup_queue()` is called, so drivers do not normally have to call `blk_queue_free_tags()` themselves.

To allocate a tag for a request, use:

```
int blk_queue_start_tag(request_queue_t *q, struct request *rq);
```

This function will associate a tag number with the given request `rq`, storing it in `rq->tag`. The return value will be zero on success, or a nonzero value if there are no more tags available. This function will remove the request from the queue, so your driver must take care not to lose track of it − and to not try to dequeue the request itself. It is also necessary to hold the queue lock when calling `blk_queue_start_tag()`.

`blk_queue_start_tag()` has been designed to work as the command prep function. If your driver would like to have tags automatically assigned, it can perform a call like:

```
blk_queue_prep_rq(queue, blk_queue_start_tag);
```

And every request that comes from `elv_next_request()` will already have a tag associated with it.

If you need to know if a given request has a tag associated with it, use the macro `blk_rq_tagged(rq)`. The return value will be nonzero if this request has been tagged.

When all transfers for a tagged request have been completed, the tag should be returned with:

```
void blk_queue_end_tag(request_queue_t *q, struct request *rq);
```

Timing is important here: `blk_queue_end_tag()` must be called before `end_that_request_last()`, or unpleasant things will happen. Be sure to have the queue lock held when calling this function.

If you need to know which request is associated with a given tag, call:

```
struct request *blk_queue_find_tag(request_queue_t *q, int tag);
```

The return value will be the `request` structure, or `NULL` if the given `tag` is not currently in use.

In the real world, things occasionally go wrong. If a drive (or the bus it is attached to) goes into an error state and must be reset, all outstanding tagged requests will be lost. In such a situation, the driver should call:

```
void blk_queue_invalidate_tags(request_queue_t *q);
```

This call will return all outstanding tags to the pool, and the associated I/O requests will be returned to the request queue so that they can be restarted.

Doing without a request queue

Some devices have no real need for a request queue. In particular, truly random−access devices, such as memory technology devices or ramdisks, can process requests quickly and do not benefit from sorting and merging of requests. Drivers for such devices may achieve better performance by shorting out much of the request queue structure and handling requests directly as they are generated.

As in 2.4, this sort of driver can set up a "make request" function. First, however, the request queue must still be created. The queue will not be used to handle the actual requests, but it contains other infrastructure needed by the block subsystem. If your driver will use a make request function, it should first create the queue with `blk_alloc_queue()`:

```
request_queue_t *blk_alloc_queue(int gfp_mask);
```

The `gfp_mask` argument describes how the requisite memory should be allocated, as usual. Note that this call can fail.

Once you have a request queue, you can set up the make request function; the prototype for this function has changed a bit from 2.4, however:

```
typedef int (make_request_fn) (request_queue_t *q, struct bio *bio);
```

If the make request function can arrange for the transfer(s) described in the given `bio`, it should do so and return zero. "Stacking" drivers can also redirect the bio by changing its `bi_bdev` field and returning nonzero; in this case the `bio` will then be dispatched to the new device's driver (this is as things were done in 2.4).

If the "make request" function performs the transfer itself, it is responsible for passing the BIO to `bio_endio()` when the transfer is complete. Note that the "make request" function is *not* called with the queue lock held.

To arrange for your driver's function to be called, use:

```
void blk_queue_make_request(request_queue_t *q,
                            make_request_fn *func);
```

If and when your driver shuts down, be sure to return the request queue to the system with:

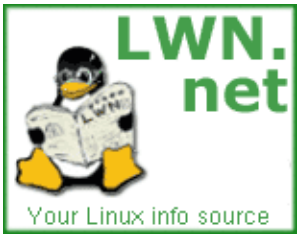
```
void blk_put_queue(request_queue_t *queue);
```

As of 2.6.0−test3, this function is just another name for `blk_cleanup_queue()`, but such things could always change in the future.

Post a comment

Driver porting: Request Queues II
(Posted May 25, 2003 14:10 UTC (Sun) by **dwmw2**) (Post reply)

Actually the drivers which make memory technology devices (i.e. flash) pretend to be a block device by some kind of 'Translation Layer' −− from the most naïve and unsafe read/erase/modify/writeback of the 'mtdblock' driver to the more complicated pseudo−filesystem of the FTL and NFTL drivers −− does benefit from request merging. You have a limited number of erase cycles to each block on the flash and it does help to combine requests which fall within the same erase block.



Driver porting: DMA changes

This article is part of the LWN Porting Drivers to 2.6 series.

The direct memory access (DMA) support layer has been extensively changed in 2.6, but, in many cases, device drivers should work unaltered. For developers working on new drivers, or for those wanting to keep their code current with the latest API, there are a fair number of changes to be aware of.

The most evident change is the creation of the new generic DMA layer. Most driver programmers will be aware of the `pci_*` DMA support functions; SPARC programmers may have also encountered the analogous set of `sbus_*` functions. Starting with 2.5.53, a new set of generic DMA functions was added which is intended to provide a DMA support API that is not specific to any particular bus. The new functions look much like the old ones; changing from one API to the other is a fairly automatic job.

The discussion below will note changes in the DMA API without looking at every new `dma_*` function. See our DMA API quick reference page for a concise summary of the mapping from the old PCI interface to the new generic functions.

Allocating DMA regions

The new and old DMA APIs both distinguish between "consistent" (or "coherent") and "streaming" memory. Consistent memory is guaranteed to look the same to the processor and to DMA−capable devices, without problems caused by caching; it is most often used for long−lasting, bidirectional I/O buffers. Streaming memory may have cache effects, and is generally used for a single transfer.

The PCI functions for allocating consistent memory are unchanged from 2.4:

```
void *pci_alloc_consistent(struct pci_dev *dev, size_t size,
                           dma_addr_t *dma_handle);
void pci_free_consistent(struct pci_dev *dev, size_t size,
                         void *cpu_addr, dma_addr_t dma_handle);
```

The generic version is a little different, adopting the term "coherent" for this type of memory, and adding an allocation flag:

```
void *dma_alloc_coherent(struct device *dev, size_t size,
                         dma_addr_t *dma_handle, int flag);
void dma_free_coherent(struct device *dev, size_t size,
                       void *cpu_addr, dma_addr_t dma_handle);
```

Here the added `flag` argument is the usual memory allocation flag. `pci_alloc_consistent()` is deemed to have an implicit `GFP_ATOMIC` flag.

For single−buffer streaming allocations, the PCI interface is, once again, unchanged, and the generic DMA interface is isomorphic to the PCI version. There is now an enumerated type for describing the direction of the

mapping:

```
enum dma_data_direction {
    DMA_BIDIRECTIONAL = 0,
    DMA_TO_DEVICE = 1,
    DMA_FROM_DEVICE = 2,
    DMA_NONE = 3,
};
```

The actual mapping and unmapping functions are:

```
dma_addr_t dma_map_single(struct device *dev, void *addr,
                          size_t size,
                          enum dma_data_direction direction);
void dma_unmap_single(struct device *dev, dma_addr_t dma_addr,
                      size_t size,
                      enum dma_data_direction direction);

dma_addr_t dma_map_page(struct device *dev, struct page *page,
                        unsigned long offset, size_t size,
                        enum dma_data_direction direction);
void dma_unmap_page(struct device *dev, dma_addr_t dma_addr,
                    size_t size,
                    enum dma_data_direction direction);
```

As is the case with the PCI versions of these functions, use of the `offset` and `size` parameters is discouraged unless you really know what you are doing.

There has been one significant change in the creation of scatter/gather streaming DMA mappings. The 2.4 version of `struct scatterlist` used a `char *` pointer (called `address`) for the buffer to be mapped, with a `struct page` pointer that would be used only for high memory addresses. In 2.6, the `address` pointer is gone, and all scatterlists must be built using `struct page` pointers.

The generic versions of the scatter/gather functions are:

```
int dma_map_sg(struct device *dev, struct scatterlist *sg,
               int nents, enum dma_data_direction direction);
void dma_unmap_sg(struct device *dev, struct scatterlist *sg,
                  int nhwentries, enum dma_data_direction direction);
```

Noncoherent DMA mappings

The generic DMA layer in 2.6 includes a set of functions for the creation of explicitly noncoherent mappings. Very few drivers will need to use this interface; it is mostly intended for code that must work on older platforms that are unable to create coherent mappings. Note that there are no PCI equivalents for these functions; you must use the generic variants.

A noncoherent mapping is created with:

```
void *dma_alloc_noncoherent(struct device *dev, size_t size,
                            dma_addr_t *dma_handle, int flag);
```

This function behaves identically to `dma_alloc_coherent()`, except that the returned mapping might not be in coherent memory. Drivers using this memory must be careful to follow the ownership rules and call the appropriate `dma_sync_*` functions when needed. An additional function:

```
void dma_sync_single_range(struct device *dev, dma_addr_t dma_handle,
                           unsigned long offset, size_t size,
                           enum dma_data_direction direction);
```

Will synchronize only a portion of a (larger) noncoherent mapping.

When your driver is done with the mapping, it should be returned to the system with:

```
void dma_free_noncoherent(struct device *dev, size_t size,
                          void *cpu_addr, dma_addr_t dma_handle);
```

Double address cycle addressing

The PCI bus is capable of a "double address cycle" (DAC) mode of operation. DAC enables the use of 64−bit DMA addresses, greatly expanding the range of memory which is reachable on systems without I/O memory mapping units. DAC is also expensive, however, and is not properly supported by all devices and buses. So the DMA support routines will normally go out of their way to avoid creating mappings that require DAC − even when the driver has set an address mask that would allow it.

There are occasions where DAC is useful, however. In particular, very large DMA mappings may not be possible in the normal, single−cycle address range. For these rare cases, the PCI layer (but not the generic DMA layer) provides a special set of functions. Note that the DAC functions can be very expensive to use; they should generally be avoided unless absolutely necessary. These functions aren't strictly a 2.6 feature; they were also added to 2.4.13.

A DAC−capable driver must begin by setting a separate address mask:

```
int pci_dac_set_dma_mask(struct pci_dev *dev, u64 mask);
```

The `mask` describes the address range that your device can support. If the function returns non−zero, DAC addressing cannot be used and should not be attempted.

A DAC mapping is created with:

```
dma64_addr_t pci_dac_page_to_dma(struct pci_dev *dev,
                                 struct page *page,
                                 unsigned long offset,
                                 int direction);
```

There's a few things to note about DAC mappings. They can only be created using `struct page` pointers and offsets; DAC mappings, by their nature, will be in high memory and thus will not have kernel virtual addresses. DAC mappings are a straight address translation requiring no external resources, so there is no need to explicitly unmap them after use. Finally, all DAC mappings are inconsistent (noncoherent) mappings, so explicit synchronization is needed to ensure that the device and CPU see the same memory. For a DAC mapping, use:

```
void pci_dac_dma_sync_single(struct pci_dev *dev,
                             dma64_addr_t dma_addr,
                             size_t len, int direction);
```

Some other details

On many architectures, no resources are consumed by DMA mappings, and thus there is no real need to unmap them. The various unmap functions are set up as no−ops on those architectures, but some programmers evidently dislike the need to remember DMA mapping addresses and lengths unnecessarily. So 2.6 (and 2.4 as of 2.4.18) has a fairly elaborate bit of preprocessor abuse which can be used to save a couple words of memory. See `Documentation/DMA−mapping.txt` in the source tree if this appeals to you.

The "PCI pool" interface is definitely not a 2.5−specific feature, since it first appeared in 2.4.4. That is new enough, however, that some references (i.e. *Linux Device Drivers, Second Edition*) do not cover them. The PCI pool interface enables the use of very small DMA buffers. In the past, such buffers would often be kept in device−specific structures. Some users ran into trouble, however, when the DMA buffer shared a cache line with other members of the same structure. The PCI pool interface was created to help move tiny DMA buffers into their own space and avoid this sort of memory corruption. Again, see `DMA−mapping.txt` for the details.

No comments have been posted. Post one now

Driver porting: DMA API quick reference

This article is part of the LWN Porting Drivers to 2.5 series.

Here is a quick listing of the new generic DMA routines alongside their 2.4 equivalents. See the associated DMA changes article for a more thorough discussion of how the DMA support layer has changed in 2.5.

2.4 PCI version

```
int pci_set_dma_mask (struct pci_dev *dev,
                      u64 mask);
```

```
void *pci_alloc_consistent(struct pci_dev *dev,
                           size_t size,
                           dma_addr_t *dma_handle);
```

```
void pci_free_consistent(struct pci_dev *dev,
                         size_t size,
                         void *cpu_addr,
                         dma_addr_t dma_handle);
```

```
PCI_DMA_NONE
PCI_DMA_BIDIRECTIONAL
PCI_DMA_TODEVICE
PCI_DMA_FROMDEVICE
```

(Integer macros).

```
dma_addr_t
pci_map_single(struct pci_device *dev,
               void *addr,
               size_t size,
               int direction);
```

```
void
pci_unmap_single(struct pci_device *dev,
                 dma_addr_t dma_handle,
                 size_t size,
                 int direction);
```

```
dma_addr_t
pci_map_page(struct pci_device *dev,
             struct page *page,
             unsigned long offset,
             size_t size,
             int direction);
```

```
void
pci_unmap_page(struct pci_device *dev,
               dma_addr_t dma_address,
               size_t size,
               int direction);
```

2.5 Generic version

```
int dma_set_mask (struct device *dev,
                  u64 mask);
```

```
void *dma_alloc_coherent(struct device *dev,
                         size_t size,
                         dma_addr_t *dma_handle,
                         int flag);
```

(`flag` is a `GFP_` allocation flag).

```
void dma_free_coherent(struct device *dev,
                       size_t size,
                       void *cpu_addr,
                       dma_addr_t dma_handle);
```

```
DMA_NONE
DMA_BIDIRECTIONAL
DMA_TODEVICE
DMA_FROMDEVICE
```

(Members of `enum dma_data_direction`).

```
dma_addr_t
dma_map_single(struct device *dev,
               void *addr,
               size_t size,
               enum dma_data_direction direction)
```

```
void
dma_unmap_single(struct device *dev,
                 dma_addr_t dma_address,
                 size_t size,
                 enum dma_data_direction directio
```

```
dma_addr_t
dma_map_page(struct device *dev,
             struct page *page,
             unsigned long offset,
             size_t size,
             enum dma_data_direction direction);
```

```
void
dma_unmap_page(struct device *dev,
               dma_addr_t dma_address,
               size_t size,
               enum dma_data_direction direction)
```

```
int
pci_map_sg(struct pci_device *dev,
           struct scatterlist *sglist,
           int nents,
           int direction);
```

```
int
dma_map_sg(struct device *dev,
           struct scatterlist *sglist,
           int nents,
           enum dma_data_direction direction);
```

(No `address` member in `scatterlist` structure; must use `page` pointers).

```
void
pci_unmap_sg(struct pci_device *dev,
             struct scatterlist *sglist,
             int nents,
             int direction);
```

```
void
dma_unmap_sg(struct device *dev,
             struct scatterlist *sglist,
             int nents,
             enum dma_data_direction direction);
```

```
void
pci_dma_sync_single(struct pci_dev *dev,
                    dma_addr_t dma_address,
                    size_t size,
                    int direction);
```

```
void
dma_sync_single(struct device *dev,
                dma_addr_t dma_address,
                size_t size,
                enum dma_data_direction direction
```

```
void
pci_dma_sync_sg(struct pci_dev *dev,
                struct scatterlist *sglist,
                int nents,
                int direction);
```

```
void
dma_sync_sg(struct device *dev,
            struct scatterlist *sglist,
            int nents,
            enum dma_data_direction direction);
```

No comments have been posted. Post one now

Driver porting: Zero−copy user−space access

This article is part of the LWN Porting Drivers to 2.6 series.

The `kiobuf` abstraction was introduced in 2.3 as a low−level way of representing I/O buffers. Its primary use, perhaps, was to represent zero−copy I/O operations going directly to or from user space. A number of problems were found with the `kiobuf` interface, however; among other things, it forced large I/O operations to be broken down into small chunks, and it was seen as a heavyweight data structure. So, in 2.5.43, kiobufs were removed from the kernel.

This article looks at how to port drivers which used the `kiobuf` interface in 2.4. We'll proceed on the assumption that the real feature of interest was direct access to user space; there wasn't much motivation to use a `kiobuf` otherwise.

Zero−copy block I/O

The 2.6 kernel has a well−developed direct I/O capability for block devices. So, in general, it will not be necessary for block driver writers to do anything to implement direct I/O themselves. It all "just works."

Should you have a need to perform zero−copy block operations, it's worth noting the presence of a useful helper function:

```
struct bio *bio_map_user(struct block_device *bdev,
                         unsigned long uaddr,
                         unsigned int len,
                         int write_to_vm);
```

This function will return a BIO describing a direct operation to the given block device `bdev`. The parameters `uaddr` and `len` describe the user−space buffer to be transferred; callers must check the returned BIO, however, since the area actually mapped might be smaller than what was requested. The `write_to_vm` flag is set if the operation will change memory − if it is a read−from−disk operation. The returned BIO (which can be `NULL` − check it) is ready for submission to the appropriate device driver.

When the operation is complete, undo the mapping with:

```
void bio_unmap_user(struct bio *bio, int write_to_vm);
```

Mapping user−space pages

If you have a char driver which needs direct user−space access (a high−performance streaming tape driver, say), then you'll want to map user−space pages yourself. The modern equivalent of `map_user_kiobuf()` is a function called `get_user_pages()`:

```
int get_user_pages(struct task_struct *task,
                   struct mm_struct *mm,
```

```
unsigned long start,
int len,
int write,
int force,
struct page **pages,
struct vm_area_struct **vmas);
```

`task` is the process performing the mapping; the primary purpose of this argument is to say who gets charged for page faults incurred while mapping the pages. This parameter is almost always passed as `current`. The memory management structure for the user's address space is passed in the `mm` parameter; it is usually `current->mm`. Note that `get_user_pages()` expects that the caller will have a read lock on `mm->mmap_sem`. The `start` and `len` parameters describe the user-buffer to be mapped; `len` is in pages. If the memory will be written to, `write` should be non-zero. The `force` flag forces read or write access, even if the current page protection would otherwise not allow that access. The `pages` array (which should be big enough to hold `len` entries) will be filled with pointers to the `page` structures for the user pages. If `vmas` is non-`NULL`, it will be filled with a pointer to the `vm_area_struct` structure containing each page.

The return value is the number of pages actually mapped, or a negative error code if something goes wrong. Assuming things worked, the user pages will be present (and locked) in memory, and can be accessed by way of the `struct page` pointers. Be aware, of course, that some or all of the pages could be in high memory.

There is no equivalent `put_user_pages()` function, so callers of `get_user_pages()` must perform the cleanup themselves. There are two things that need to be done: marking of modified pages, and releasing them from the page cache. If your device modified the user pages, the virtual memory subsystem may not know about it, and may fail to write the pages to permanent storage (or swap). That, of course, could lead to data corruption and grumpy users. The way to avoid this problem is to call:

```
int set_page_dirty_lock(struct page *page);
```

for each page in the mapping.

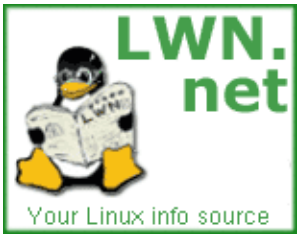
Finally, every mapped page must be released from the page cache, or it will stay there forever; simply pass each page structure to:

```
void put_page(struct page *page);
```

After you have released the page, of course, you should not access it again.

For a good example of how to use `get_user_pages()` in a char driver, see the definition of `sgl_map_user_pages()` in `drivers/scsi/st.c`.

No comments have been posted. Post one now



Driver porting: supporting mmap()

This article is part of the LWN Porting Drivers to 2.6 series.

Occasionally, a device driver will need to map an address range into a user process's space. This mapping can be done to give the process direct access to a device's I/O memory area, or to the driver's DMA buffers. 2.6 features a number of changes to the virtual memory subsystem, but, for most drivers, supporing `mmap()` will be relatively painless.

Using remap_page_range()

There are two techniques in use for implementing `mmap()`; often the simpler of the two is using `remap_page_range()`. This function creates a set of page table entries covering a given physical address range. The prototype of `remap_page_range()` changed slightly in 2.5.3; the relevant virtual memory area (VMA) pointer must be passed as the first parameter:

```
int remap_page_range(struct vm_area_struct *vma, unsigned long from,
                     unsigned long to, unsigned long size,
                     pgprot_t prot);
```

`remap_page_range()` is now explicitly documented as requiring that the memory management semaphore (usually `current->mm->mmap_sem`) be held when the function is called. Drivers will almost invariably call `remap_page_range()` from their `mmap()` method, where that semaphore is already held. So, in other words, driver writers do not normally need to worry about acquiring `mmap_sem` themselves. If you use `remap_page_range()` from somewhere other than your `mmap()` method, however, do be sure you have acquired the semaphore first.

Note that, if you are remapping into I/O space, you may want to use:

```
int io_remap_page_range(struct vm_area_struct *vma, unsigned long from,
                        unsigned long to, unsigned long size,
                        pgprot_t prot);
```

On all architectures other than SPARC, `io_remap_page_range()` is just another name for `remap_page_range()`. On SPARC systems, however, `io_remap_page_range()` uses the systems I/O mapping hardware to provide access to I/O memory.

`remap_page_range()` retains its longstanding limitation: it cannot be used to remap most system RAM. Thus, it works well for I/O memory areas, but not for internal buffers. For that case, it is necessary to define a `nopage()` method. (Yes, if you are curious, the "mark pages reserved" hack still works as a way of getting around this limitation, but its use is strongly discouraged).

Using vm_operations

The other way of implementing mmap is to override the default VMA operations to set up a driver−specific `nopage()` method. That method will be called to deal with page faults in the mapped area; it is expected to return a `struct page` pointer to satisfy the fault. The `nopage()` approach is flexible, but it cannot be used to remap I/O regions; only memory represented in the system memory map can be mapped in this way.

The good news is that the `nopage()` method really has not changed since 2.4; drivers using this approach should port forword without changes. There are a few things worth mentioning, though. One is that the `vm_operations_struct` is rather smaller than it was in 2.4.0; the `protect()`, `swapout()`, `sync()`, `unmap()`, and `wppage()` methods have all gone away (they were actually deleted in 2.4.2). Device drivers made little use of these methods, and should not be affected by their removal.

There is also one new `vm_operations_struct` method:

```
int (*populate)(struct vm_area_struct *area, unsigned long address,
                unsigned long len, pgprot_t prot, unsigned long pgoff,
                int nonblock);
```

The `populate()` method was added in 2.5.46; its purpose is to "prefault" pages within a VMA. A device driver could certainly implement this method by simply invoking its `nopage()` method for each page within the given range, then using:

```
int install_page(struct mm_struct *mm, struct vm_area_struct *vma,
                 unsigned long addr, struct page *page,
                 pgprot_t prot);
```

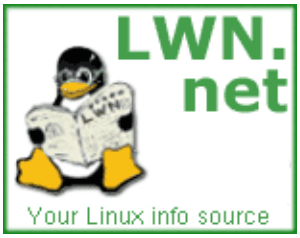
to create the page table entries. In practice, however, there is no real advantage to doing things in this way. No driver in the mainline (2.5.67) kernel tree implements the `populate()` method.

Finally, one use of `nopage()` is to allow a user process to map a kernel buffer which was created with `vmalloc()`. In the past, a driver had to walk through the page tables to find a `struct page` corresponding to a `vmalloc()` address. As of 2.5.5 (and 2.4.19), however, all that is needed is a call to:

```
struct page *vmalloc_to_page(void *address);
```

This call is not a variant of `vmalloc()` − it allocates no memory. It simply returns a pointer to the `struct page` associated with an address obtained from `vmalloc()`.

No comments have been posted. Post one now



Driver porting: Atomic kmaps

This article is part of the LWN Porting Drivers to 2.6 series.

High memory can be a pain to work with. The addressing limitations of 32−bit processors make it impossible to map all of high memory into the kernel's address space. So various workarounds must be employed to manage high memory portably; this need is one of the reasons for the increasing use of `struct page` pointers in the kernel.

When the kernel needs to access a high memory page directly, an *ad hoc* memory mapping must be set up. This is the purpose of the functions `kmap()` and `kunmap()`, which have existed since high memory support was first implemented. `kmap()` is relatively expensive to use, however; it requires global page table changes, and it can put the calling function to sleep. It is thus a poor fit to many parts of the kernel where performance is important.

To address these performance issues, a new type of kernel mapping (the "atomic kmap") has been created (they actually existed, in a slightly different form, in 2.4.1). Atomic kmaps are intended for short−term use in small, atomic sections of kernel code; it is illegal to sleep while holding an atomic kmap. Atomic kmaps are a per−CPU structure; given the constraints on their use, there is no point in sharing them across processors. They are also available in very limited numbers.

In fact, there are only about a dozen atomic kmap slots available on each processor (the actual number is architecture−dependent), and users of atomic kmaps must specify which slot to use. A new enumerated type (`km_type`) has been defined to give names to the atomic kmap slots. The slots that will be of most interest to driver writers are:

- `KM_USER0`, `KM_USER1`. These slots are to be used by code called from user space (i.e. system calls).
- `KM_IRQ0`, `KM_IRQ1`. Slots for interrupt handlers to use.
- `KM_SOFTIRQ0`, `KM_SOFTIRQ1`; for code running out of a software interrupt, such as a tasklet.

Several other slots exist, but they have been set aside for specific purposes and should not be used.

The actual interface for obtaining an atomic kmap is:

```
void *kmap_atomic(struct page *page, enum km_type type);
```

The return value is a kernel virtual address which may be used to address the given page. `kmap_atomic()` will always succeed, since the slot to use has been given to it. It will also disable preemption while the atomic kmap is held.

When you have finished with the atomic kmap, you should undo it with:

```
void kunmap_atomic(void *address, enum km_type type);
```

Users of atomic kmaps should be very aware of the fact that nothing in the kernel prevents one function from stepping on another function's mappings. Code which holds atomic kmaps thus needs to be short and simple. If you are using one of the `KM_IRQ` slots, you should have locally disabled interrupts first. As long as everybody is careful, conflicts over atomic kmap slots do not arise.

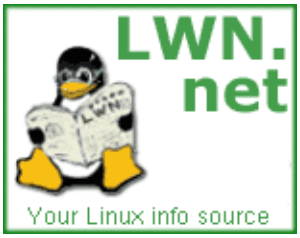
Should you need to obtain a `struct page` pointer for an address obtained from `kmap_atomic()`, you can use:

```
struct page *kmap_atomic_to_page(void *address);
```

If you are wanting to map buffers obtained from the block layer in a BIO structure, you should use the BIO−specific kmap functions (described in the BIO article) instead.

Atomic kmaps are a useful resource for performance−critical code. They should not be overused, however. For any code which might sleep, or which can afford to wait for a mapping, the old standard `kmap()` should be used instead.

No comments have been posted. Post one now



Driver porting: Network drivers

This article is part of the LWN Porting Drivers to 2.6 series.

Much of the core network driver API has not been changed between the 2.4 and 2.6 kernels. With only a relatively small amount of work, most drivers should function just fine under 2.6. If, however, you want to get the very best performance out of high−bandwidth network cards, you may have to make more extensive changes to your driver to work with the new APIs which have been made available.

Network device allocation

In 2.6, network devices are part of the wider kernel device model. There are advantages to this change, including the fact that network device information is available under `/sys/class/net/`. But hooking into the driver model poses a new set of potential race conditions which were not there before. What happens if your driver module is removed while a process has an associated sysfs file open? Network drivers are more susceptible than most to this problem because the networking subsystem does not restrict the unloading of drivers via the module use count.

The only way to properly deal with this problem is to allocate network devices in a dynamic manner, and to let the device model code figure out when to free them. To that end, all `net_device` structures must be allocated with the new `alloc_netdev()` function:

```
struct net_device *alloc_netdev(int sizeof_priv, const char *name,
                                void (*setup)(struct net_device *));
```

Here, `sizeof_priv` is the size of the structure that you would otherwise allocate and assign to the `net_device` `priv` field; `alloc_netdev()` will allocate that memory for you as well. `name` is the name of the device (a format string is acceptible, so something like `"eth%d"` works), and `setup` is a function to be called to complete the initialization of the `net_device` structure. The `setup` function can be the same function that, in older drivers, you may have assigned to the `init` field in the `net_device` structure.

For Ethernet devices, there is a simpler form:

```
struct net_device *alloc_etherdev(int sizeof_priv);
```

Calling this function is equivalent to:

```
my_dev = alloc_netdev(sizeof(my_priv), "eth%d", setup_ether);
```

Either way, when you are done with the device (i.e. after you have called `unregister_netdev()`), you must free it with:

```
void free_netdev(struct net_device *dev);
```

Note that it would be an error to free the `priv` field separately − let `free_netdev()` take care of it.

NAPI

The most significant change, perhaps, is the addition of NAPI ("New API"), which is designed to improve the performance of high−speed networking. NAPI works through:

- Interrupt mitigation. High−speed networking can create thousands of interrupts per second, all of which tell the system something it already knew: it has lots of packets to process. NAPI allows drivers to run with (some) interrupts disabled during times of high traffic, with a corresponding decrease in system load.
- Packet throttling. When the system is overwhelmed and must drop packets, it's better if those packets are disposed of before much effort goes into processing them. NAPI−compliant drivers can often cause packets to be dropped in the network adapter itself, before the kernel sees them at all.
- More careful packet treatment, with special care taken to avoid reordering packets. Out−of−order packets can be a significant performance bottleneck.

NAPI was also backported to the 2.4.20 kernel.

The following is a whirlwind tour of what must be done to create a NAPI−compliant network driver. More details can be found in `networking/NAPI_HOWTO.txt` in the kernel documentation directory, and, of course, in the source of drivers which have been converted. Note that use of NAPI is entirely optional, drivers will work just fine (though perhaps a little more slowly) without it.

The first step is to make some changes to your driver's interrupt handler. If your driver has been interrupted because a new packet is available, that packet should not be processed at the time. Instead, your driver should disable any further "packet available" interrupts and tell the networking subsystem to poll your driver shortly to pick up all available packets. Disabling interrupts, of course, is a hardware−specific matter between the driver and the adaptor. Arranging for polling is done with a call to:

```
void netif_rx_schedule(struct net_device *dev);
```

An alternative form you'll see in some drivers is:

```
if (netif_rx_schedule_prep(dev))
    __netif_rx_schedule(dev);
```

The end result is the same either way. (If `netif_rx_schedule_prep()` returns zero, it means that there was already a poll scheduled, and you should not have received another interrupt).

The next step is to create a `poll()` method for your driver; it's job is to obtain packets from the network interface and feed them into the kernel. The `poll()` prototype is:

```
int (*poll)(struct net_device *dev, int *budget);
```

The `poll()` function should process all available incoming packets, much as your interrupt handler might have done in the pre−NAPI days. There are some exceptions, however:

- Packets should not be passed to `netif_rx()`; instead, use:

  ```
  int netif_receive_skb(struct sk_buff *skb);
  ```

The return value will be `NET_RX_DROP` if the networking subsystem had to drop the packet. Network drivers could use that information to stop feeding packets for the moment, but no driver in the kernel tree does so currently.

- A new `struct net_device` field called `quota` contains the maximum number of packets that the networking subsystem is prepared to receive from your driver at this time. Once you have exhausted that quota, no further packets should be fed to the kernel in this `poll()` call.
- The `budget` parameter also places a limit on the number of packets which your driver may process. Whichever of `budget` and `quota` is lower is the real limit.
- Your driver should decrement `dev->quota` by the number of packets it processed. The value pointed to by the `budget` parameter should also be decremented by the same amount.
- If packets remain to be processed (i.e. the driver used its entire quota), `poll()` should return a value of one.
- If, instead, all packets have been processed, your driver should reenable interrupts, turn off polling, and return zero. Polling is stopped with:

```
void netif_rx_complete(struct net_device *dev);
```

The networking subsystem promises that `poll()` will not be invoked simultaneously (for the same device) on multiple processors.

The final step is to tell the networking subsystem about your `poll()` method. This, of course, is done in your initialization code when all the other `struct net_device` fields are set:

```
dev->poll = my_poll;
dev->weight = 16;
```

The `weight` field is a measure of the importance of this interface; the number stored here will turn out to be the same number your driver finds in the `quota` field when `poll()` is called. If you forget to initialize `weight` and leave it at zero, `poll()` will never be called (voice of experience here). Gigabit adaptor drivers tend to set `weight` to 64; smaller values can be used for slower media.

Receiving packets in non-interrupt mode

Network drivers tend to send packets into the kernel while running in interrupt mode. There are occasions where, instead, packets will be received by a driver running in process context. There is no problem with this mode of operation, but it is possible that the networking software interrupt which performs packet processing may be delayed, reducing performance. To avoid this problems, drivers handing packets to the kernel outside of interrupt context should use:

```
int netif_rx_ni(struct sk_buff *skb);
```

instead of `netif_rx()`.

Other 2.5 features

A number of other networking features were added in 2.5. Here is a quick summary of developments that driver developers may want to be aware of.

- **Ethtool support**. Ethtool is a utility which can perform detailed configuration of network interfaces; it can be found on the gkernel SourceForge page. This tool can be used to query network information, tweak detailed operating parameters, control message logging, and more. Supporting ethtool requires

implementing the `SIOCETHTOOL ioctl()` command, along with (parts of, at least) the lengthy set of ethtool commands. See `<linux/ethtool.h>` for a list of things that can be done.
Implementing the message logging control features requires checking the logging settings before each `printk()` call; there is a set of convenience macros in `<linux/netdevice.h>` which make that checking a little easier.

- **VLAN support**. The 2.5 kernel has support for 802.1q VLAN interfaces; this support has also been working its way into 2.4, with the core being merged in 2.4.14. See this page for information on the Linux 802.1q implementation.
- **TCP segmentation offloading**. The TSO feature can improve performance by offloading some TCP segmentation work to the adaptor and cutting back slightly on bus bandwidth. TSO is an advanced feature that can be tricky to implement with good performance; see the `tg3` or `e1000` drivers for examples of how it's done.

Post a comment

dev−>weight at zero

(Posted May 4, 2003 3:47 UTC (Sun) by **movement**) (Post reply)

> If you forget to initialize weight and leave it at zero, poll() will never be called

Any good reason there isn't "if (dev−>poll) WARN_ON(!dev−>weight);" ?

Driver porting: Device model overview

This article is part of the LWN Porting Drivers to 2.6 series.

One of the more significant changes in the 2.5 development series is the creation of the integrated device model. The device model was originally intended to make power management tasks easier through the maintenance of a representation of the host system's hardware structure. A certain amount of mission creep has occurred, however, and the device model is now closely tied into a number of device management tasks − and other kernel functions as well.

The device model presents a bit of a steep learning curve when first encountered. But the underlying concepts are not *that* hard to understand, and driver programmers will benefit from a grasp of what's going on.

The fundamental task of the driver model is to maintain a set of internal data structures which reflect the architecture and state of the underlying system. Among other things, the driver model tracks:

- Which devices exist in the system, what power state they are in, what bus they are attached to, and which driver is responsible for them.
- The bus structure of the system; which buses are connected to which others (i.e. a USB controller can be plugged into a PCI bus), and which devices each bus can potentially support (along with associated drivers), and which devices actually exist.
- The device drivers known to the system, which devices they can support, and which bus type they know about.
- What *kinds* of devices ("classes") exist, and which real devices of each class are connected. The driver model can thus answer questions like "where is the mouse (or mice) on this system?" without the need to worry about how the mouse might be physically connected.
- And many other things.

Underneath it all, the driver model works by tracking system configuration changes (hardware and software) and maintaining a complex "web woven by a spider on drugs" data structure to represent it all.

Some device model terms

The device model brings with it a whole new vocabulary to describe its data structures. A quick overview of some driver model terms appears below; much of this stuff will be looked at in detail later on.

device
: A physical or virtual object which attaches to a (possibly virtual) bus.

driver
: A software entity which may probe for and be bound to devices, and which can perform certain management functions.

bus

A device which serves as an attachment point for other devices.

class

A particular type of device which can be expected to perform in certain ways. Classes might include disks, partitions, serial ports, etc.

subsystem

A top−level view of the system's structure. Subsystems used in the kernel include `devices` (a hierarchical view of all devices on the system), `bus` (a bus−oriented view), `class` (devices by class), `net` (the networking subsystem), and others. The best way to think of a subsystem, perhaps, is as a particular view into the device model data structure rather than a physical component of the system. The same objects (devices, usually) show up in most subsystems, but they are organized differently.

Other terms will be defined as we come to them.

sysfs

Sysfs is a virtual filesystem which provides a userspace−visible representation of the device model. The device model and sysfs are sometimes confused with each other, but they are distinct entities. The device model functions just fine without sysfs (but the reverse is not true).

The sysfs filesystem is usually mounted on `/sys`; for readers without a 2.6 system at hand, an example `/sys` hierarchy from a simple system is available. The top−level directories there correspond to the known subsystems in the model. The full device model data structure can be seen by looking at the entries and links within each subsystem. Thus, for example, the first IDE disk on a particular system, being a device, would appear as:

```
/sys/devices/pci0/00:11.1/ide0/0.0
```

But that device appears (in symbolic link form) under other subsystems as:

```
/sys/block/hda/device
/sys/bus/ide/devices/0.0
```

And, additionally, the IDE controller can be found as:

```
/sys/bus/pci/devices/0.11.1
/sys/bus/pci/drivers/VIA IDE/00:11.1
```

Within the disk's own sysfs directory (under `/devices`), the link `block` points back at `/sys/block/hda`. As was said before, it is a complicated data structure.

Driver writers generally need not worry about sysfs; it is magically created and implemented by the driver model and bus driver code. The one exception comes about when it comes to exporting **attributes** via sysfs. These attributes represent some aspect of how the device and/or its driver operate; they may or may not be writeable from user space. Sysfs is now the preferred way (over `/proc` or `ioctl()`) to export these variables to user space. The next article in the series looks at how to manage attributes.

Kobjects

Even though most driver writers will never have to manipulate a kobject directly, it is hard to dig very deeply into the driver model without encountering them. A **kobject** is a simple representation of data relevant to any object found in the system; in a true object−oriented language, this would be the class that most others inherit from. Kobjects contain the attributes that, it is expected, most objects in the system will need: a name, reference count, parent, and type. Almost any object related to the device model will have a kobject buried deeply inside it somewhere.

A **kset** is a container for a set of kobjects of identical type. Ksets belong to a subsystem (but a subsystem can hold more than one kset). Among other things, ksets control how the system responds to hotplug events − the addition (or removal) of an entry to (or from) the set.

Together, kobjects and ksets make up much of the glue that holds the driver model structure together. A separate article in this series covers kobjects and ksets in detail.

No comments have been posted. Post one now

Driver porting: Device classes

This article is part of the LWN Porting Drivers to 2.6 series.

Previous articles in this series have shown how the device model maintains a data structure representing the physical structure of the host system. There is more to know about a system than how it is plugged together, however; indeed, most of the time, user space really does not care about physical connections. Users (and the applications they run) are much more interested in questions like "what disks does this system have" or "where is the mouse?"

To help with this sort of resource discovery issue, the driver model exports a "class" interface. Devices, once registered, can be associated with one or more classes which describe the function(s) performed by the device. Class memberships show up under the `/sys/class` sysfs directory, and, of course, can be decorated with all kinds of attributes. There are also mechanisms which provide notification − both within and outside of the kernel − when a device joins or leaves a class. The class interface can also be the easiest way for a driver to make arbitrary attributes available via sysfs.

For many (if not most) drivers, class membership will be handled automatically in the higher layers. Block devices, for example, are associated with the "block" class when their associated `gendisk` structures are registered. (This class currently appears in `/sys/block`, incidentally; it will likely move to `/sys/class/block` at some point). Occasionally, however, it can be necessary to explicitly associate a device with a specific class. This article describes how to do that, and − though remaining superficial − it provides more information than is really needed in order to, with luck, provide an understanding of how the class system works.

For those wishing for a hands−on example, the full source for a version of the "simple block driver" module that understands classes is available.

Creating a class

It is a rare device which exists in a unique class of its own; as a result, drivers will almost never create their own classes. Should the need arise, however, the process is simple. The first step is the creation of a `struct class` (defined in `<linux/device.h>`). There are two necessary fields, being the name and a pointer to a "release" function; the SBD driver sets up its class as:

```
static struct class sbd_class = {
     .name = "sbd",
     .release = sbd_class_release
 };
```

The name is, of course, how this class will show up under `/sys/class`. We will get to the release function shortly, after we have looked at class devices.

Beyond that, there is only one other thing that a class definition can provide: a "hotplug" function:

```
int (*hotplug)(struct class_device *dev, char **envp,
               int num_envp, char *buffer, int buffer_size);
```

The addition of a device to a class creates a hotplug event. Before `/sbin/hotplug` is called to respond to that event, the class's `hotplug()` method (if any) will be called. That method can add variables to the environment that is passed to `/sbin/hotplug`; they should be put into `buffer` (respecting the given `buffer_size`) with pointers set into `envp` (but no more than `num_envp` of them, and with a `NULL` pointer to terminate the list). The return value should be zero, or the usual negative error code.

Classes need to be registered, of course:

```
int class_register(struct class *cls);
```

The return value will be zero of all goes well. The `void` function `class_unregister()` will do exactly what one would expect.

Class devices

If your device type lacks a specific registration function of its own (such as `add_disk()` or `register_netdev()`), or if you have created your own custom class, you may find yourself adding your device(s) to a class explicitly. Membership in a class is represented by an instance of `struct class_device`. There are three fields that should normally be filled in:

```
struct class *class;
struct device *dev;
char class_id[BUS_ID_SIZE];
```

The `class` pointer, of course, should be aimed at the proper class structure. The `dev` pointer is optional; it is used to create the `device` and `driver` symbolic links in the device's class entry in sysfs. Since user−space processes looking to discover devices of a particular class probably want to have that pointer, you should make it easy for them. The `class_id` is a string which is unique within the class − it becomes, of course, the name of the device's sysfs entry.

Once the `class_device` structure has been set up, it can be added to the class with:

```
int class_device_register(struct class_device *class_dev);
```

`class_device_unregister()` can be used at module unload time.

Once you register a class device, it becomes available to the world as a whole. If your class device is allocated dynamically, you must be very careful about when you free it. Remember that user−space processes can retain references to your device via your sysfs attributes; you must not free the class device until all of those references are gone.

That, of course, is the purpose of the `release` function stored in `struct class`. This function has a simple prototype:

```
void release_fn(struct class_device *cd);
```

This function is called when the last reference to the given device goes away; it should respond by freeing the device. That call will typically happen when you call `class_device_unregister()` on the device, but it could happen later if other references persist.

Please note that, if your class device structure is dynamically allocated, or it embedded within another, dynamic structure, you *must* use a release function to free that structure or your code is buggy.

Class device attributes

Attributes are easily added to a class device entry. If the attribute is to be readable, it will need a "show" function to respond to reads; the function used to export the driver version in SBD looks like:

```
static ssize_t show_version(struct class_device *cd, char *buf)
{
    sprintf(buf, "%s\n", Version);
    return strlen(buf) + 1;
}
```

If the attribute is to be writable, you will need a store function too:

```
ssize_t (*store)(struct class_device *, const char *buf, size_t count);
```

These functions are then bundled into an attribute structure with:

```
CLASS_DEVICE_ATTR(name, mode, show, store);
```

The `name` should not be a quoted string; it is joined in the macro to create a structure called `class_device_attr_name`.

The final step is to create the actual device attribute, using:

```
int class_device_create_file(struct class_device *,
                             struct class_device_attribute *);
```

You can call `class_device_remove_file()` to get rid of an attribute, but that is also done automatically for you when a device is removed from a class.

Interfaces

The term "interface," as used within the device model, is a bit confusing. A better way to think of interfaces is as a sort of constructor and destructor mechanism for class device entries. An interface provides `add()` and `remove()` methods which are called as devices are added to (and removed from) a class; their usual purpose is to add class−specific attributes to the class device entry. They can, however, perform any other kernel function that might be useful in response to class device events.

Briefly, the creation of an interface requires the creation of a `class_interface` structure, which needs to have the following fields filled in:

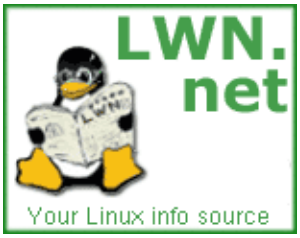
```
struct class *class;
int (*add) (struct class_device *);
void (*remove) (struct class_device *);
```

Once the interface is set up with:

```
int class_interface_register(struct class_interface *);
```

The `add()` and `remove()` functions will be called when devices are added to (or removed from) the given `class`. A call to `class_interface_unregister()` undoes the registration.

No comments have been posted. Post one now



Using read−copy−update

This article is part of the LWN Porting Drivers to 2.6 series.

Read−copy−update (RCU) is a mutual exclusion technique which can operate without locking most of the time. It can yield significant performance benefits when the data to be protected is accessed via a pointer, is read frequently, changed rarely, and references to the structure are not held while a kernel thread sleeps. The core idea behind RCU is that, when the data requires updating, a pointer to a new structure containing the new data can be stored immediately. The old structure containing the outdated data can then be freed at leisure, after it is certain that no process in the system holds a reference to that structure. For details on the ideas behind RCU, see this LWN article, or (for many details) this paper. Just don't ask SCO, even though they claim to own the technique.

The first step in using RCU within a subsystem is to define a structure containing the data to be protected. Often that structure already exists; for example, RCU has been retrofitted into the dentry cache (using `struct dentry`), the network routing cache (`struct rtable`), and several other, similar contexts. The structures need to be allocated dynamically and accessed via a pointer − RCU cannot be used with static structures.

Code which reads data structures protected by RCU need only take a couple of simple precautions:

- A call to `rcu_read_lock()` should be made before accessing the data, and `rcu_read_unlock()` should be called afterward. This call disables preemption (and does nothing else) − a fast but necessary operation for RCU to work properly. These calls (along with the rest of the RCU definitions) are found in `<linux/rcupdate.h>`.
- The code must not sleep while the "RCU read lock" is held.

Thus, code which reads an RCU−protected data structure will look something like:

```
struct my_stuff *stuff;

rcu_read_lock();
stuff = find_the_stuff(args...);
do_something_with(stuff);       /* Cannot sleep */
do_something_else_with(stuff);  /* ditto        */
rcu_read_unlock();
```

The write side of RCU is a little more complicated, but not that difficult. To update a data structure, the code starts by allocating a new copy of that structure, and filling in the new information. The code should then replace the pointer to the outdated structure with the new one, keeping a copy of the old pointer. After this operation, kernel code running on any other processor will find the new version of the structure. The old one cannot yet be freed, however, since it is possible that another processor is still using it.

The code should arrange to dispose of the old structure when it is known that it cannot be referenced anywhere else in the system. That is done through a call to `call_rcu()`:

```
void call_rcu(struct rcu_head *head,
              void (*func)(void *arg),
              void *arg);
```

The calling code must provide an `rcu_head` structure, but need not initialize it in any way. Usually, that structure is embedded within the larger structure protected by RCU. The function `func` will be called when the structure can be safely freed, with `arg` as its one argument. All that `func` need do, normally, is call something like `kfree()` to free up the structure.

The RCU algorithm works by waiting until every processor in the system has scheduled at least once. Since the rules require that references to RCU−protected structures cannot be held over sleeps, no processor can possibly hold a reference to an old structure after it has scheduled. When all processors have scheduled (after the pointer change), references to the old structure can not exist, and the structure can be freed.

For what it's worth, the RCU code exports the "wait for everybody to schedule" functionality, should it be useful elsewhere. To perform this wait, one need only make a call to `synchronize_kernel()`.

Post a comment

Using read−copy−update

(Posted Jul 11, 2003 14:41 UTC (Fri) by **goatbar**) (Post reply)

What happens if a different processor updates the stucture before before the one scheduled to be deleted is deleted. It must keep a queue of these sturctures to delete, right? RCU seems like a nice and simple algorithm.

Changes to the USB driver API for the 2.5 series kernel

[This article was contributed by Greg Kroah−Hartman]

Over the 2.5 kernel development series, the USB driver api has changed a lot. As LWN has graciously allowed me to write a kernel article this week, and I know a bit about the USB kernel code, I thought I would discuss a short summary of the major changes that have happened with it for anyone wanting to port a 2.4 USB driver to 2.5.

The main `struct usb_driver` structure has shrunk. The `fops` and `minor` variables have been removed, as the majority of USB drivers do not need to use the USB major number. If a USB driver needs to use the USB major, then the `usb_register_dev()` function should be called when a USB device has been found, and a minor number needs to be assigned to it. This function needs to have a `struct usb_interface` that the minor number should be assigned to, and a pointer to a `struct usb_class_driver structure`. This `usb_class_driver` structure is defined as:

```
struct usb_class_driver {
        char *name;
        struct file_operations *fops;
        mode_t mode;
        int minor_base;
};
```

The `name` variable is the devfs name for this driver. The `fops` variable is a pointer to the `struct file_operations` that should be called when this device is accessed. The `mode` variable defines the file permissions that devfs will use when creating the device node. Finally, the `minor_base` variable is the start of the minor range that this driver has assigned to it.

When `usb_register_dev()` is called, the devfs node will be created if devfs is enabled, and a usb class device is created in sysfs at `/sys/class/usb/`. After the device is removed from the system, the `usb_unregister_dev()` function should be called. This function will return the minor number to the USB core (to be used again later for a new device), the devfs node will be deleted if devfs is enabled in the kernel, and the usb class device will be removed from sysfs.

Because of these two functions, USB drivers no longer need to worry about managing the devfs entries on their own, like is necessary in the 2.4 kernel.

Also, USB drivers can use the usb_set_intfdata() function to save a pointer to a USB driver specific structure. This can be used instead of having to keep a static array of device pointers for every driver. `usb_set_intfdata()` should be called at the end of the USB driver probe function. Then in the `open()` function, `usb_get_intfdata()` should be called to retrieve the stored pointer.

For a good example of how to make these changes, look at how the `usb−skeleton.c` driver has changed between the 2.4 and 2.5 kernels. This driver is a framework driver that can be used to base any new USB

drivers on.

There are also a number of USB api functions that have had their parameters modified from 2.4 to 2.5. Two of the most visible examples of this is the `usb_submit_urb()` function, and the USB `probe()` callback function.

In the `usb_submit_urb()` function, the USB core and host controller drivers can need to allocate memory from the kernel to complete the USB transfer. In 2.4, the core and host controller drivers guess that it is safe to sleep when requesting memory, and would call kmalloc with the `GFP_KERNEL` flag. The USB developers quickly realized that this is not always the best thing. So the `usb_submit_urb()` function now requires that the memory flags be passed to it:

```
int usb_submit_urb(struct urb *urb, int mem_flags);
```

In the 2.5 kernel the probe callback is now:

```
int (*probe) (struct usb_interface *intf,
              const struct usb_device_id *id);
```

This was done to emphasize that USB drivers bind to a USB interface, and not to an entire USB device. If the struct usb_device structure is needed to be found, the `interface_to_usbdev()` macro should be used.

The biggest change in the USB api between the 2.4 and 2.5 kernels is much improved documentation. To build the kernel USB documentation, run:

```
make psdocs
```

By doing this, the `Documentation/DocBook/usb.ps` file will have been created. This contains a lot of details about how the USB subsystem works, and what all of the options to the USB functions are. The primary author of all of this documentation is David Brownell, who also wrote the USB gadget and USB 2.0 EHCI host controller driver.

Post a comment

improved docs

(Posted Jul 16, 2003 18:04 UTC (Wed) by **roelofs**) (Post reply)

By any chance is there a `pdfdocs` target? I can view and print PS OK, but PDF viewers tend to do a nicer job of rendering and scaling text.

Greg

There were some more changes to note...

(Posted Jul 17, 2003 15:00 UTC (Thu) by **HalfMoon**) (Post reply)

Another set of changes is *removing "automagic" resubmission* for periodic transfers (iso, interrupt). In 2.4, the host controller drivers re−issued URBs until they were unlinked, or rather they tried to do so ... they couldn't always do so, the failures got masked from device drivers. So in 2.6 kernels, device drivers do this, and they're guaranteed to see any failures. If you use periodic transfers, your driver will need updating for this.

Plus, on 2.6 kernels *every type of URB can be queued*, including control and interrupt. On 2.4 kernels interrupt transfers couldn't be queued (so high transfer rates couldn't work), and in fact there was an arbitrary one−packet limit. (That limit is now gone, you can read or write multi−packet reports without needing to de−fragment them in your driver.) And on 2.4 there were several related host controller differences: the UHCI drivers didn't queue control requests (causing problems with any composite devices, and with many user mode probrams), and wouldn't queue bulk requests unless you used an explicit QUEUE_BULK flag (now gone).

But the real USB work in 2.6 was to fix lots of bugs and remove lots of opportunities for things to break.

Changes to the USB driver API for the 2.5 series kernel

(Posted Nov 10, 2003 22:56 UTC (Mon) by **happyking**) (Post reply)

make psdocs fails on usb for 2.6.0−test5, last part is mentioned here...

jade:Documentation/DocBook/writing_usb_driver.sgml:325:2: start tag was here
make[1]: *** [Documentation/DocBook/writing_usb_driver.ps] Error 8

Driver porting: Char devices and large dev_t

This article is part of the LWN Porting Drivers to 2.6 series.

Much 2.5 kernel development work went toward increasing the size of the `dev_t` device number type. That work has necessarily forced some changes in how device drivers work with the rest of the kernel. This article describes the changes as seen from the point of view of char drivers. It is current as of the 2.6.0−test9 kernel. **Note that the interfaces describe here are still volatile** and could change significantly before 2.6.0−final is released.

Major and minor numbers

With the expanded `dev_t`, it is no longer be possible to assume that major and minor numbers fit within eight bits. To the greatest extent possible, the relevant interfaces have been changed in ways that will not break existing drivers. In particular, a driver which uses the longstanding `register_chrdev()` function to register a char device will never see minor device numbers greater then 255. Attempts to open a device node with a larger minor number will simply fail with a "no such device" error.

One change that is visible to all drivers, however, is the elimination of the `kdev_t` type. Device numbers are now a simple `dev_t` throughout the kernel. The place where this change is most apparent for most will be the change in the type of the inode `i_rdev` field. Drivers which need to get major or minor numbers from inodes should use the two new helper functions:

```
unsigned iminor(struct inode *inode);
unsigned imajor(struct inode *inode);
```

Use of these functions will help keep a driver working in the future, even if the representation within inodes changes again.

The new way

`register_chrdev()` continues to work as it always did, and drivers which use that function need not be changed. Unchanged drivers, however, will not be able to use the expanded device number range, or take advantage of the other features provided by the new code. Sooner or later, it is worthwhile to get to know the new interface.

The new way to register a char device range is with:

```
int register_chrdev_region(dev_t from, unsigned count, char *name);
```

Here, `from` is the device number of the first device in the range, `count` is the number of device numbers to register, and `name` is the base name of the device (it appears in `/proc/devices`). The return value is zero if all goes well, and a negative error number otherwise.

Note that `from` is a device number, not a major number. This interface allows the registration of an arbitrary range of device numbers, starting from anywhere. So the `from` argument specifies both the beginning major and minor number. If the `count` argument exceeds the number of minor numbers available, the allocation will continue on into the next major number; this is a design feature.

`register_chrdev_region()` works if you know which major device number you wish to use. If, instead, your driver expects to work with dynamic major number allocation, it should use:

```
int alloc_chrdev_region(dev_t *dev, unsigned baseminor,
                        unsigned count, char *name);
```

In this case, `dev` is an output−only parameter which will be set to the first device number of the allocated range. The input parameters are `baseminor`, the first minor number to use (usually zero); `count`, the number of device numbers to allocate; and `name`, the base name of the device. Once again, the return value is zero or a negative error code.

Connecting up devices

Some readers may have noticed that the above functions, unlike `register_chrdev()`, do not have a `file_operations` argument. Registering a device number range sets those numbers aside for your use, but it does not actually make any device operations available to user space. There is now a separate object (`struct cdev`) which represents char devices, and which must be set up by your driver to actually make a device available.

To work with `struct cdev`, you code should include `<linux/cdev.h>`. Then, the usual way of getting one of these structures is with:

```
struct cdev *cdev_alloc(void);
```

If all goes well, the return value will be a pointer to a newly allocated, initialized `cdev` structure. Check that value, though; there is a memory allocation involved, and things can always fail.

It is also possible to declare a static `cdev` structure, or to embed one within another structure. In this case, you should pass it to:

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

before doing anything else with it.

Your driver will need to set a couple of fields in the `cdev` structure before adding it to the system. The `owner` field should be set to the owning module, usually `THIS_MODULE`. The device's `file_operations` structure should be pointed to by the `ops` field. And, to get a directory in sysfs, you should also set the `name` field in the embedded kobject, with something like:

```
struct cdev *my_cdev = cdev_alloc();
kobject_set_name(&cdev->kobj, "my_cdev%d", devnum);
```

Note that `kobject_set_name()` takes a `printf()`−like format string and associated arguments.

Once you have the structure set up, it's time to add it to the system:

```
int cdev_add(struct cdev *cdev, dev_t dev, unsigned count);
```

`cdev` is, of course, a pointer to the `cdev` structure; `dev` is the first device number handled by this structure, and `count` is the number of devices it implements. This, one `cdev` structure can stand in for several physical devices, though you will usually not want to do things that way.

There are two important things to bear in mind when calling `cdev_add()`. The first is that this call can fail. If the return value is nonzero, the device has not been added and is not visible to user space. If, instead, the call succeeds, the device becomes immediately live. You should not call `cdev_add()` until your driver is completely ready to handle calls to the device's methods.

Adding a device also creates a directory entry under `/sys/cdev`, using the name stored in the `kobj.name` field. As of this writing, that directory is empty, but one assumes that all sorts of good things (the associated device numbers, if nothing else) will eventually show up there.

Deleting devices

If you need to get rid of a `cdev` structure, the usual way of doing things is to call:

```
void cdev_del(struct cdev *cdev);
```

This function should only be called, however, on a `cdev` structure which has been successfully added to the system with `cdev_add()`. If you need to destroy a structure which has not been added in this way (perhaps `cdev_add()` failed), you must, instead, manually decrement the reference count in the structure's kobject with a call like:

```
kobject_put(&cdev->kobj);
```

Calling `cdev_del()` on a device which is still active (if, say, a user-space process still has an open file reference to it) will cause the device to become inaccessible, but it will not actually delete the structure at that time. The reference count in the structure will keep it around until all the references have gone away. That means that your driver's methods could be called after you have deleted your `cdev` object - a possibility you should be aware of.

The reference count of a `cdev` structure can be manipulated with:

```
struct kobject *cdev_get(struct cdev *cdev);
void cdev_put(struct cdev *cdev);
```

Note that these functions change two reference counts: that of the `cdev` structure, and that of the module which owns it. It will be rare for drivers to call these functions, however.

Finding your device in file operations

Most of the methods provided by the driver in the `file_operations` structure take a `struct inode` (or a `struct file` which can be used to find the associated inode) as an argument. Traditionally, Linux drivers have looked at the device number stored in the inode's `i_rdev` field to determine which device is being operated upon. That technique still works, but, in many cases, there is a better way. In 2.6, `struct inode` contains a field called `i_cdev`, which contains a pointer to the associated `cdev` structure. If you have embedded one of those structures within your own, device-specific structure, you can use the `container_of()` macro (described in the kobject article) to obtain a pointer to that structure.

Why things were done this way

The new interface may seem rather more complex to many. Before, a single call to `register_chrdev()` was all that was necessary; now a driver has to deal with the additional hassle of managing `cdev` structures. This approach provides a great deal of flexibility, however, in how the device number space can be managed. Each device gets exactly the number range it needs, and its operations will never be invoked for device numbers outside that range. In the past, it has been noted that many drivers had incorrect range checks on minor numbers; with the new scheme, all those range checks can go away altogether.

The new method also makes it easy for each device to have its own `file_operations` structure without the need for big `switch` statements in the `open()` method. Separate `cdev` structures can also have separate entries in `/sys/cdev`. In general, char devices have become proper objects within the kernel, with all the advantages that come with that status. A little bit of extra object management is a small price to pay.

Post a comment

Driver porting: Char devices and large dev_t
(Posted Sep 29, 2003 12:31 UTC (Mon) by **cynove**) (Post reply)

Sure this interface is fine, but how can I make it available to build my out of kernel tree module ?

When using register_chrdev, everything is fine.
If I use some cdev_ function, the build process complains that cdev_del, cdev_add etc... are undefined when building module.ko. (ie stage 2)

The only difference I found is that register_chrdev is defined as extern in <linux/fs.h>, and cdev_ are just declared in cdev.h.

How can I make these functions available to my modules ?

Driver porting: Char devices and large dev_t
(Posted Sep 29, 2003 12:45 UTC (Mon) by **cynove**) (Post reply)

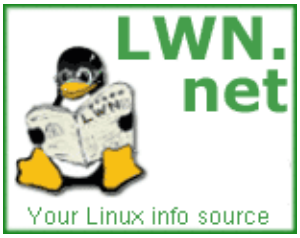
Maybe I should have done more googling !
It seems the response is there :

http://www.ussg.iu.edu/hypermail/linux/kernel/0309.2/0150.html

Exporting char dev functions
(Posted Sep 29, 2003 13:53 UTC (Mon) by **corbet**) (Post reply)

That patch went into −test6; upgrade and your module should work.

Do be aware that the char driver interface is still somewhat volatile; things will still change somewhat before it's all over.



The zen of kobjects

This article is part of the LWN Porting Drivers to 2.5 series.

The "kobject" structure first made its appearance in the 2.5.45 development kernel. It was initially meant as a simple way of unifying kernel code which manages reference counted objects. The kobject has since encountered a bit of "mission creep," however; it is now the glue that holds much of the device model and its sysfs interface together. It is rare for a driver writer to have to work with kobjects directly; they are usually hidden in structures created by higher−level code. Kobjects have a certain tendency to leak through the intervening layers, however, and make their presence known. So a familiarity with what they are and how they work is a good thing to have. This document will cover the kobject type and related topics, but will gloss over most of the interactions between kobjects and sysfs (those will be covered separately, later on).

Part of the difficulty in understanding the driver model − and the kobject abstraction upon which it is built − is that there is no obvious starting place. Dealing with kobjects requires understanding a few different types, all of which make reference to each other. In an attempt to make things easier, we'll take a multi−pass approach, starting with vague terms and adding detail as we go. To that end, here are some quick definitions of some terms we will be working with.

- A **kobject** is an object of type `struct kobject`. Kobjects have a name and a reference count. A kobject also has a parent pointer (allowing kobjects to be arranged into hierarchies), a specific type, and, perhaps, a representation in the sysfs virtual filesystem.

  Kobjects are generally not interesting on their own; instead, they are usually embedded within some other structure which contains the stuff the code is really interested in.
- A **ktype** is a type associated with a kobject. The ktype controls what happens when a kobject is no longer referenced and the kobject's default representation in sysfs.
- A **kset** is a group of kobjects all of which are embedded in structures of the same type. The kset is the basic container type for collections of kobjects. Ksets contain their own kobjects, for what it's worth. Among other things, that means that a kobject's parent is usually the kset that contains it, though things do not normally have to be that way.

  When you see a sysfs directory full of entries, generally each of those entries corresponds to a kobject in the same kset.
- A **subsystem** is a collection of ksets which, collectively, make up a major sub−part of the kernel. Subsystems normally correspond to the top−level directories in sysfs.

We'll look at how to create and manipulate all of these types. A bottom−up approach will be taken, so we'll go back to kobjects.

Embedding kobjects

It is rare (even unknown) for kernel code to create a standalone kobject; instead, kobjects are used to control

access to a larger, domain−specific object. To this end, kobjects will be found embedded in other structures. If you are used to thinking of things in object−oriented terms, kobjects can be seen as a top−level, abstract class from which other classes are derived. A kobject implements a set of capabilities which are not particularly useful by themselves, but which are nice to have in other objects. The C language does not allow for the direct expression of inheritance, so other techniques − such as structure embedding − must be used.

So, for example, the 2.6.0−test6 version of `struct cdev`, the structure describing a char device, is:

```
struct cdev {
    struct kobject kobj;
    struct module *owner;
    struct file_operations *ops;
    struct list_head list;
};
```

If you have a `struct cdev` structure, finding its embedded kobject is just a matter of using the `kobj` pointer. Code that works with kobjects will often have the opposite problem, however: given a `struct kobject` pointer, what is the pointer to the containing structure? You should avoid tricks (such as assuming that the kobject is at the beginning of the structure) and, instead, use the `container_of()` macro, found in `<linux/kernel.h>`:

```
    container_of(pointer, type, member)
```

where `pointer` is the pointer to the embedded kobject, `type` is the type of the containing structure, and `member` is the name of the structure field to which `pointer` points. The return value from `container_of()` is a pointer to the given `type`. So, for example, a pointer to a `struct kobject` embedded within a `struct cdev` called "`kp`" could be converted to a pointer to the containing structure with:

```
struct cdev *device = container_of(kp, struct cdev, kobj);
```

Programmers will often define a simple macro for "back−casting" kobject pointers to the containing type.

Initialization of kobjects

Code which creates a kobject must, of course, initialize that object. Some of the internal fields are setup with a (mandatory) call to `kobject_init()`:

```
void kobject_init(struct kobject *kobj);
```

Among other things, `kobject_init()` sets the kobject's reference count to one. Calling `kobject_init()` is not sufficient, however. Kobject users must, at a minimum, set the name of the kobject; this is the name that will be used in sysfs entries. If you dig through the kernel source, you will find code which copies a string directly into the kobject's `name` field, but that approach should be avoided. Instead, use:

```
int kobject_set_name(struct kobject *kobj, const char *format, ...);
```

This function takes a `printk`−style variable argument list. Believe it or not, it is actually possible for this operation to fail; conscientious code should check the return value and react accordingly.

The other kobject fields which should be set, directly or indirectly, by the creator are its ktype, kset, and

parent. We will get to those shortly.

Reference counts

One of the key functions of a kobject is to serve as a reference counter for the object in which it is embedded. As long as references to the object exist, the object (and the code which supports it) must continue to exist. The low−level functions for manipulating a kobject's reference counts are:

```
struct kobject *kobject_get(struct kobject *kobj);
void kobject_put(struct kobject *kobj);
```

A successful call to `kobject_get()` will increment the kobject's reference counter and return the pointer to the kobject. If, however, the kobject is already in the process of being destroyed, the operation will fail and `kobject_get()` will return `NULL`. This return value must always be tested, or no end of unpleasant race conditions could result.

When a reference is released, the call to `kobject_put()` will decrement the reference count and, possibly, free the object. Note that `kobject_init()` sets the reference count to one, so the code which sets up the kobject will need to do a `kobject_put()` eventually to release that reference.

Note that, in many cases, the reference count in the kobject itself may not be sufficient to prevent race conditions. The existence of a kobject (and its containing structure) may well, for example, require the continued existence of the module which created that kobject. It would not do to unload that module while the kobject is still being passed around. That is why the `cdev` structure we saw above contains a `struct module` pointer. The reference counting for `struct cdev` is implemented as follows:

```
struct kobject *cdev_get(struct cdev *p)
{
        struct module *owner = p->owner;
        struct kobject *kobj;

        if (owner && !try_module_get(owner))
                return NULL;
        kobj = kobject_get(&p->kobj);
        if (!kobj)
                module_put(owner);
        return kobj;
}
```

Creating a reference to a `cdev` structure requires creating a reference also to the module which owns it. So `cdev_get()` uses `try_module_get()` to attempt to increment that module's usage count. If that operation succeeds, `kobject_get()` is used to increment the kobject's reference count as well. That operation could fail, of course, so the code checks the return value from `kobject_get()` and releases its reference to the module if things don't work out.

Hooking into sysfs

An initialized kobject will perform reference counting without trouble, but it will not appear in sysfs. To create sysfs entries, kernel code must pass the object to `kobject_add()`:

```
int kobject_add(struct kobject *kobj);
```

As always, this operation can fail. The function:

```
void kobject_del(struct kobject *kobj);
```

will remove the kobject from sysfs.

There is a `kobject_register()` function, which is really just the combination of the calls to `kobject_init()` and `kobject_add()`. Similarly, `kobject_unregister()` will call `kobject_del()`, then call `kobject_put()` to release the initial reference created with `kobject_register()` (or really `kobject_init()`).

ktypes and release methods

One important thing still missing from the discussion is what happens to a kobject when its reference count reaches zero. The code which created the kobject generally does not know when that will happen; if it did, there would be little point in using a kobject in the first place. Even predicatable object lifecycles become more complicated when sysfs is brought in; user−space programs can keep a reference to a kobject (by keeping one of its associated sysfs files open) for an arbitrary period of time.

The end result is that a structure protected by a kobject cannot be freed before its reference count goes to zero. The reference count is not under the direct control of the code which created the kobject. So that code must be notified asynchronously whenever the last reference to one of its kobjects goes away.

This notification is done through a kobject's `release()` method. Usually such a method has a form like:

```
void my_object_release(struct kobject *kobj)
{
        struct my_object *mine = container_of(kobj, struct my_object, kobj);

        /* Perform any additional cleanup on this object, then... */
        kfree (mine);
}
```

One important point cannot be overstated: every kobject must have a `release()` method, and the kobject must persist (in a consistent state) until that method is called. If these constraints are not met, the code is flawed.

Interestingly, the `release()` method is not stored in the kobject itself; instead, it is associated with the ktype. So let us introduce `struct kobj_type`:

```
struct kobj_type {
        void (*release)(struct kobject *);
        struct sysfs_ops    *sysfs_ops;
        struct attribute    **default_attrs;
};
```

This structure is used to describe a particular type of kobject (or, more correctly, of containing object). Every kobject needs to have an associated `kobj_type` structure; a pointer to that structure can be placed in the kobject's `ktype` field at initialization time, or (more likely) it can be defined by the kobject's containing kset.

The `release` field in `struct kobj_type` is, of course, a pointer to the `release()` method for this type of kobject. The other two fields (`sysfs_ops` and `default_attrs`) control how objects of this type are represented in sysfs; they are beyond the scope of this document.

ksets

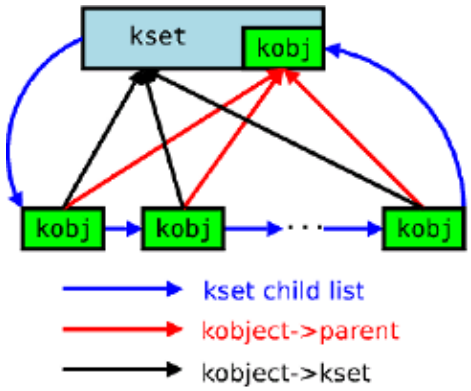
In many ways, a kset looks like an extension of the `kobj_type` structure; a kset is a collection of identical kobjects. But, while `struct kobj_type` concerns itself with the *type* of an object, `struct kset` is concerned with aggregation and collection. The two concepts have been separated so that objects of identical type can appear in distinct sets.

A kset serves these functions:

- It serves as a bag containing a group of identical objects. A kset can be used by the kernel to track "all block devices" or "all PCI device drivers."
- A kset is the directory−level glue that holds the device model (and sysfs) together. Every kset contains a kobject which can be set up to be the parent of other kobjects; in this way the device model hierarchy is constructed.
- Ksets can support the "hotplugging" of kobjects and influence how hotplug events are reported to user space.

In object−oriented terms, "kset" is the top−level container class; ksets inherit their own kobject, and can be treated as a kobject as well.

A kset keeps its children in a standard kernel linked list. Kobjects point back to their containing kset via their `kset` field. In almost all cases, the contained kobjects also have a pointer to the kset (or, strictly, its embedded kobject) in their `parent` field. So, typically, a kset and its kobjects look something like what you see in the diagram to the right. Do bear in mind that (1) all of the contained kobjects in the diagram are actually embedded within some other type, possibly even other ksets, and (2) it is not required that a kobject's parent be the containing kset.



For initialization and setup, ksets have an interface very similar to that of kobjects. The following functions exist:

```
void kset_init(struct kset *kset);
int kset_add(struct kset *kset);
int kset_register(struct kset *kset);
void kset_unregister(struct kset *kset);
```

For the most part, these functions just call the analogous `kobject_` function on the kset's embedded kobject.

For managing the reference counts of ksets, the situation is about the same:

```
struct kset *kset_get(struct kset *kset);
void kset_put(struct kset *kset);
```

A kset, too, has a name, which is stored in the embedded kobject. So, if you have a kset called `my_set`, you would set its name with:

```
kobject_set_name(my_set−>kobj, "The name");
```

Ksets also have a pointer (in the `ktype` field) to the `kobj_type` structure describing the kobjects it contains. This type will be applied to any kobject which does not contain a pointer to its own `kobj_type` structure.

Another attribute of a kset is a set of hotplug operations; these operations are invoked whenever a kobject enters or leaves the kset. They are able to determine whether a user−space hotplug event is generated for this change, and to affect how that event is presented. The hotplug operations are beyond the scope of this document; they will be discussed later with sysfs.

One might ask how, exactly, a kobject is added to a kset, given that no functions which perform that function have been presented. The answer is that this task is handled by `kobject_add()`. When a kobject is passed to `kobject_add()`, its `kset` member should point to the kset to which the kobject will belong. `kobject_add()` will handle the rest. There is currently no other way to add a kobject to a kset without directly messing with the list pointers.

Finally, a kset contains a subsystem pointer (called `subsys`). So it must be time to talk about subsystems.

Subsystems

A subsystem is a representation for a high−level portion of the kernel as a whole. It is actually a simple structure:

```
struct subsystem {
        struct kset         kset;
        struct rw_semaphore rwsem;
};
```

A subsystem, thus, is really just a wrapper around a kset. In fact, life is not quite that simple; a single subsystem can contain multiple ksets. This containment is represented by the `subsys` pointer in `struct kset`; so, if there are multiple ksets in a subsystem, it will not be possible to find all of them directly from the `subsystem` structure.

Every kset must belong to a subsystem; the subsystem's `rwsem` semaphore is used to serialize access to a kset's internal linked list.

Subsystems are often declared with a special macro:

```
decl_subsys(char *name, struct kobj_type *type,
            struct kset_hotplug_ops *hotplug_ops);
```

This macro just creates a `struct subsystem` (its name is the `name` given to the macro with `_subsys` appended) with the internal kset initialized with the given `type` and `hotplug_ops`.

Subsystems have the usual set of setup and teardown functions:

```
void subsystem_init(struct subsystem *subsys);
int subsystem_register(struct subsystem *subsys);
void subsystem_unregister(struct subsystem *subsys);
struct subsystem *subsys_get(struct subsystem *subsys)
void subsys_put(struct subsystem *subsys);
```

Most of these operations just act upon the subsystem's kset.

Kobject initialization again

Now that we have covered all of that stuff, we can talk in detail about how a kobject should be prepared for its existence in the kernel. Here are all of the `struct kobject` fields which must be initialized somehow:

- `name` and `k_name` − the name of the object. These fields should always be initialized with `kobject_set_name()`.
- `refcount` is the kobject's reference count; it is initialized by `kobject_init()`
- `parent` is the kobject's parent in whatever hierarchy it belongs to. It can be set explicitly by the creator. If `parent` is `NULL` when `kobject_add()` is called, it will be set to the kobject of the containing kset.
- `kset` is a pointer to the kset which will contain this kobject; it should be set prior to calling `kobject_add()`.
- `ktype` is the type of the kobject. If the kobject is contained within a kset, and that kset has a type set in its `ktype` field, then this field in the kobject will not be used. Otherwise it should be set to a suitable `kobj_type` structure.

Often, much of the initialization of a kobject is handled by the layer that manages the containing kset. Thus, to get back to our old example, a char driver might create a `struct cdev`, but it need not worry about setting any of the fields in the embedded kobject − except for the name. Everything else is handled by the char device layer.

Looking forward

So far, we have covered the operations used to set up and manipulate kobjects. The core concept is relatively simple: kobjects can be used to (1) maintain a reference count for an object and clean up when the object is no longer used, and (2) create a hierarchical data structure through kset membership.

What is missing so far is how kobjects represent themselves to user space. The sysfs interface to kobjects makes it easy to export information to (and to receive information from) user space. The symbolic linking features of sysfs allow the creation of pointers across distinct kobject hierarchies. Stay tuned for a description of how all that works.

Post a comment

The zen of kobjects

(Posted Oct 2, 2003 12:45 UTC (Thu) by **paulsheer**) (Post reply)

With each one of these articles i get more encouraged
to help with the kernel. *sigh* perhaps i will get
round to it at some point.

Initializing a kobject

(Posted Oct 2, 2003 17:56 UTC (Thu) by **gregkh**) (Post reply)

One minor addition, all fields in a `struct kobject` must be initialized to zero before calling `kobject_init()`. If this is not true, bad things can happen when `kobject_init()` is called.

This is easily done by calling `memset(foo, 0x00, sizeof(*foo));` if `foo` contained a `struct`

`kobject`, before calling `kobject_init()`.

The zen of kobjects

(Posted Oct 3, 2003 4:52 UTC (Fri) by **torsten**) (Post reply)

You had me, then you lost me. Somewhere just after **Embedding kobjects**.

The zen of kobjects

(Posted Oct 5, 2003 12:48 UTC (Sun) by **razalasm**) (Post reply)

This is the kind of intelligent, well written summary of Linux's ever changing technical landscape that justifies my subscription to LWN. Please keep up the good work.

minor typo???

(Posted Oct 7, 2003 14:48 UTC (Tue) by **pflugstad**) (Post reply)

In this section:

```
Subsystems are often declared with a special macro:

    decl_subsys(char *name, struct kobj_type *type,
                struct kset_hotplug_ops *hotplug_ops);

This macro just creates a struct system (its name is the name given to the
macro with _subsys appended) with the internal kset initialized with the
given type and hotplug_ops.
```

Do you mean that this creates a "struct system", or "struct subsystem", defined just above it?

minor typo???

(Posted Oct 7, 2003 14:49 UTC (Tue) by **corbet**) (Post reply)

Clearly it was meant to be a 'struct subsystem', it's fixed now.

In general, typo reports sent to lwn@lwn.net have a higher chance of being read and acted upon − we can't always keep up with the comments.

The zen of kobjects

(Posted Oct 12, 2003 23:15 UTC (Sun) by **Russell**) (Post reply)

Seems to me they could do a LOT more with the kobject idea. What about adding a hook lists to report events to whoever is interested, i.e. what gnome's glib calls signals. This could clean up a lot of things.

OO ignoramuses −− and proud of it!

(Posted Oct 20, 2003 14:55 UTC (Mon) by **guest1**) (Post reply)

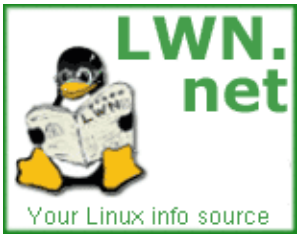
This must be one of the twistiest, dumbest, worst code designs I've ever seen!

When they speak, at every opportunity they get, most kernel hackers love to complain

loudly about how stupid and useless C++ and OO development is. And now when they code, at every opportunity they get, kernel hackers like to demonstrate how stupid and useless their OO design skills are.

This is not a clean design. They are trying to solve 2 or 3 problems with 1 lump of code. The sysfs, kset, and kobject "objects" are mashed together. 3 sets of pointers? The functionality for adding an "object" to a "container" is supposed to be contained in the container, not the object! What a joke, little early for April 1st, isn't it?

Even a quick high−level overview of any of dozens of OO libraries (especially where containers are involed) would have given the kernel hackers any number of good ideas how to create a clean, compartmentalized design. Oh yea, we don't do OO, it's dumb.



kobjects and hotplug events

October 7, 2003

This article was contributed by Greg Kroah−Hartman.

Last week, in the article about kobjects, it was mentioned that a kset has a set of hotplug operations. This week we will introduce the hotplug operations, and detail how they work.

Remember that a kset is a group of kobjects which are all embedded in the same type of structure. In the definition of a kset, a pointer to a `struct kset_hotplug_ops` is specified. If this pointer is set, whenever a kobject that is a member of that kset is created or destroyed by the kernel, the userspace program `/sbin/hotplug` will be called. If a kobject does not have a kset associated with it, the kernel will traverse up the kobject hierarchy (using the `parent` pointer) to try to find a kset to use for this test.

`struct kset_hotplug_ops` is a structure containing three function pointers and is defined as:

```
    struct kset_hotplug_ops {
        int (*filter)(struct kset *kset, struct kobject *kobj);
        char *(*name)(struct kset *kset, struct kobject *kobj);
        int (*hotplug)(struct kset *kset, struct kobject *kobj,
                       char **envp, int num_envp,
                       char *buffer, int buffer_size);
    };
```

Hotplug filters

The `filter` function will be called by the kernel before a hotplug operation happens. The kobject and the kset which are being used for the hotplug event are passed as parameters to the function. If this function returns `1` then the hotplug event will be generated; otherwise (if the function returns `0`), the hotplug event will not be generated. This function is used by the driver core and the block subsystem to filter out hotplug events for kobjects that are owned by these systems but which should not have hotplug events generated for them.

As an example, the driver core's hotplug filter is contained in the file `drivers/base/core.c` and looks like:

```
static int dev_hotplug_filter(struct kset *kset, struct kobject *kobj)
{
        struct kobj_type *ktype = get_ktype(kobj);

        if (ktype == &ktype_device) {
                struct device *dev = to_dev(kobj);
                if (dev->bus)
                        return 1;
        }
        return 0;
}
```

In this function, the first thing that happens is the type of the kobject is checked. If this really is a device type of kobject, then we know it is safe to cast this kobject to a `struct device`, which is done in the line:

```
    struct class_device *class_dev = to_class_dev(kobj);
```

If this class device has a class assigned to it (`dev->bus`), the filter function tells the kobject core that it is acceptable to generate a hotplug event for this object. If any of these tests fail, the function returns `0` stating that no hotplug event should be generated.

The filter function allows objects in the device tree to own kobjects themselves (to create subdirectories, and for other uses) and prevent hotplug events from being created for these child kobjects.

Hotplug event names

When `/sbin/hotplug` is called by the kernel, it only has one argument passed to it, the name of the subsystem creating the event. All other information about the hotplug event is passed in environment variables. For detailed examples of some of the hotplug events and environment variables, see the Linux Hotplug project website.

For the kobject core to know what kind of name to provide to this hotplug event, the `name` function callback is provided. If the kset associated with this kobject wants to override the name of the kset for the hotplug event, then this function needs to return a pointer to a string that is more suitable. If this function is not provided, or it returns NULL, then the kset's name will be used.

For example, all `struct device` objects in the kernel belong to the same device kset (the device, driver, and class model sits on top of kobjects and ksets, making it simpler for driver authors to use). This kset is called `"devices"`. It would not make much sense for every USB or IEEE1394 device that was plugged into, or removed from the system to generate a hotplug event with the name `"devices"`. Because of this, the device subsystem has a name function for its hotplug operations:

```
static char *dev_hotplug_name(struct kset *kset, struct kobject *kobj)
{
        struct device *dev = to_dev(kobj);

        return dev->bus->name;
}
```

In this function, the kobject is converted to a `struct device`, and then the name of the bus associated with this device is returned. This allows USB devices to create hotplug events with the name `"usb"` and IEEE1394 devices to create hotplug events with the name `"ieee1394"`.

One note about this function: the only way that we know it is safe to directly cast this kobject into a `struct device` is that it has passed the `filter` function first. In that function, the type of the kobject and the fact that the device had a pointer to a bus was verified. Without that filter function, that information would have to be checked before blindly casting and following two levels of pointer indirection.

Hotplug environment variables

All calls to `/sbin/hotplug` provide the majority of information within environment variables. The three variables that are always set for every hotplug calls are the following:

Variable	Value	Description
ACTION	add or remove	Describes if the kobject is being added or removed from the system.
SEQNUM	*numeric*	Provides the sequence number of the hotplug event. It is used for userspace to determine if it has received the hotplug event out of order or not. The value starts out a 0 when the kernel boots, and increments with every /sbin/hotplug call. It is a 64−bit number, so it will not roll over for a very long time.
DEVPATH	*string*	The path to the kobject that the hotplug event is happening on, within the sysfs file system. To get the true filesystem location for this kobject, add the mount point for sysfs (usually /sys) to the beginning of this string.

These variables are usually enough for userspace to determine what is happening with this hotplug event, but a lot of subsystems want to provide more information. This is especially true when a kobject is removed from the system, as the sysfs entry for the device will also be removed, preventing userspace from being able to look up any attributes about the device that was just removed. Because of this, the hotplug callback is provided for the kset to provide any additional environment variables that it wants to.

The hotplug function callback is allowed to add any additional environment variables that the kset might want added for this call to /sbin/hotplug. To review the prototype for this function:

```
    int (*hotplug)(struct kset *kset, struct kobject *kobj,
                   char **envp, int num_envp,
                   char *buffer, int buffer_size);
```

Here, kset and kobj are the objects for which the event is happening, envp is a pointer to an array of environment variables (in the usual "NAME=value" format), num_envp is the length of envp, buffer is a buffer where additional variables can be put, and buffer_size is the size of buffer. The hotplug function should create any additional environment variables that are called for, store pointers to them in envp, and terminate envp with a NULL. If the hotplug callback returns a non−zero value, the hotplug event is aborted, and /sbin/hotplug will not be called.

The driver and class subsystems pass hotplug calls down to the bus and class owners of the kobject that is being created or removed, allowing these individual subsystems to add their own environment variables. For example, for all devices located on the USB bus, the function usb_hotplug() in the drivers/usb/core/usb.c file will be called. This function is defined as (with much of the boring code removed):

```
static int usb_hotplug(struct device *dev, char **envp, int num_envp,
                       char *buffer, int buffer_size)
{
        struct usb_interface *intf;
        struct usb_device *usb_dev;
        char *scratch;
        int i = 0;
        int length = 0;

        /* ... */
        intf = to_usb_interface(dev);
        usb_dev = interface_to_usbdev(intf);

        /* ... */
        scratch = buffer;
```

```
        envp[i++] = scratch;
        length += snprintf(scratch, buffer_size − length, "PRODUCT=%x/%x/%x",
                           usb_dev−>descriptor.idVendor,
                           usb_dev−>descriptor.idProduct,
                           usb_dev−>descriptor.bcdDevice);
        if ((buffer_size − length <= 0) || (i >= num_envp))
                return −ENOMEM;
        ++length;
        scratch += length;

        /* ... */
        envp[i++] = NULL;
        return 0;
}
```

The lines:

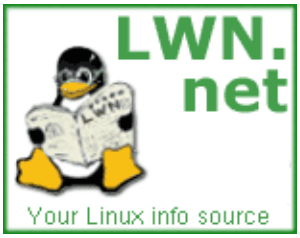
```
        scratch = buffer;
        envp[i++] = scratch;
```

set up the environment pointer to point to the next location in the buffer passed to us. Then the big call to `snprintf` creates a variable called `PRODUCT` which is assigned the value of the USB device's vendor, product and device ids separated by a `'/'` character. If `snprintf` succeeded in not overrunning the buffer provided to us, and we still have enough room for one more environment variable, then the function continues on. The last environment variable pointer is set to NULL before returning.

All that work for a simple result

With the combined effort of the kset hotplug function callbacks every kset can customize the call to `/sbin/hotplug` in whatever way it likes while still providing userspace a consistent interface from the kernel. Every kobject that is registered with `sysfs` can generate this call easily, so all parts of the kernel that use kobjects and ksets automatically get the `/sbin/hotplug` interface for free. This allows userspace projects such as the module loading scripts, devlabel, udev, and D−BUS valuable information as to what the kernel is doing whenever a change in the kobject tree occurs.

No comments have been posted. Post one now



kobjects and sysfs

This article is part of the LWN Porting Drivers to 2.5 series.

In The Zen of Kobjects, this series looked at the kobject abstraction and the various interfaces that go with it. That article, however, glossed over one important part of the kobject structure (with a promise to fill in in later): its interface to the sysfs virtual filesystem. The time has come to fulfill our promise, however, and look at how sysfs works at the lower levels.

To use the functions described below, you will need to include both `<linux/kobject.h>` and `<linux/sysfs.h>` in your source files.

How kobjects get sysfs entries

As we saw in the previous article, there are two functions which are used to set up a kobject. If you use `kobject_init()` by itself, you will get a standalone kobject with no representation in sysfs. If, instead, you use `kobject_register()` (or call `kobject_add()` separately), a sysfs directory will be created for the kobject; no other effort is required on the programmer's part.

The name of the directory will be the same as the name given to the kobject itself. The location within sysfs will reflect the kobject's position in the hierarchy you have created. In short: the kobject's directory will be found in its parent's directory, as determined by the kobject's `parent` field. If you have not explicitly set the parent field, but you have set its `kset` pointer, then the kset will become the kobject's parent. If there is no parent and no kset, the kobject's directory will become a top−level directory within sysfs, which is rarely what you really want.

Populating a kobject's directory

Getting a sysfs directory corresponding to a kobject is easy, as we have seen. That directory will be empty, however, which is not particularly useful. Most applications will want the kobject's sysfs entry to contain one or more attributes with useful information. Creating those attributes requires some additional steps, but is not all that hard.

The key to sysfs attributes is the kobject's `kobj_type` pointer. When we looked at kobject types before, we passed over a couple of sysfs−related entries. One, called `default_attrs`, describes the attributes that all kobjects of this type should have; it is a pointer to an array of pointers to `attribute` structures:

```
struct attribute {
    char                    *name;
    struct module           *owner;
    mode_t                  mode;
};
```

In this structure, `name` is the name of the attribute (as it will appear within sysfs), `owner` is a pointer to the module (if any) which is responsible for the implementation of this attribute, and `mode` is the protection bits which are to be applied to this attribute. The mode is usually `S_IRUGO` for read−only attributes; if the attribute is writable, you can toss in `S_IWUSR` to give write access to root only. The last entry in the `default_attrs` list must be `NULL`.

The `default_attrs` array says what the attributes are, but does not tell sysfs how to actually implement those attributes. That task falls to the `kobj_type->sysfs_ops` field, which points to a structure defined as:

```
struct sysfs_ops {
    ssize_t (*show)(struct kobject *kobj, struct attribute *attr,
                    char *buffer);
    ssize_t (*store)(struct kobject *kobj, struct attribute *attr,
                    const char *buffer, size_t size);
};
```

These functions will be called for each read and write operation, respectively, on an attribute of a kobject of the given type. In each case, `kobj` is the kobject whose attribute is being accessed, `attr` is the `struct attribute` for the specific attribute, and `buffer` is a one−page buffer for attribute data.

The `show()` function should encode the attribute's full value into `buffer`, being sure not to overrun `PAGE_SIZE`. Remember that the sysfs convention requires that attributes contain single values or, at most, an array of similar values, so the one−page limit should never be a problem. The return value is, of course, the number of bytes of data actually put into `buffer` or a negative error code.

The `store()` function has a similar interface; the additional `size` parameter gives the length of the data received from user space. Never forget that `buffer` contains unchecked, user−supplied data; treat it carefully and be sure that it fits whatever format you require. The return value should normally be the same as `size`, unless something has gone wrong.

As you can see, sysfs requires the use of a single set of `show()` and `store()` functions for all attributes of kobjects of the same type. Those functions will, usually, maintain their own array of attribute information to enable them to find the real function charged with implementing each attribute.

Non−default attributes

In many cases, the kobject type's `default_attrs` field describes all of the attributes that kobject will ever have. It does not need to be that way, however; attributes can be added and removed at will. If you wish to add a new attribute to a kobject's sysfs directory, simply fill in an `attribute` structure and pass it to:

```
int sysfs_create_file(struct kobject *kobj, struct attribute *attr);
```

If all goes well, the file will be created with the name given in the `attribute` structure and the return value will be zero; otherwise, the usual negative error code is returned.

Note that the same `show()` and `store()` functions will be called to implement operations on the new attribute. Before you add a new, non−default attribute to a kobject, you should take whatever steps are necessary to ensure that those functions know how to implement that attribute.

To remove an attribute, call:

```
int sysfs_remove_file(struct kobject *kobj, struct attribute *attr);
```

After the call, the attribute will no longer appear in the kobject's sysfs entry. Do be aware, however, that a user−space process could have an open file descriptor for that attribute, and that `show()` and `store()` calls are still possible after the attribute has been removed.

Symbolic links

The sysfs filesystem has the usual tree structure, reflecting the hierarchical organization of the kobjects it represents. The relationships between objects in the kernel is often more complicated than that, however. For example, one sysfs subtree (`/sys/devices`) represents all of the devices known to the system, while others represent the device drivers. These trees do not, however, represent the relationships between the drivers and the devices they implement. Showing these additional relationships requires extra pointers which, in sysfs, are implemented with symbolic links.

Creating a symbolic link within sysfs is easy:

```
int sysfs_create_link(struct kobject *kobj,
                      struct kobject *target,
                      char *name);
```

This function will create a link (called `name`) pointing to `target`'s sysfs entry as an attribute of `kobj`. It will be a relative link, so it works regardless of where sysfs is mounted on any particular system.

The link will persist even if `target` is removed from the system. If you are creating symbolic links to other kobjects, you should probably have a way of knowing about changes to those kobjects, or some sort of assurance that the target kobjects will not disappear. The consequences (dead symbolic links within sysfs) are not particularly grave, but they would not do much to create confidence in the proper functioning of the system either.

Symbolic links can be removed with:

```
void sysfs_remove_link(struct kobject *kobj, char *name);
```

Binary attributes

The sysfs conventions call for all attributes to contain a single value in a human−readable text format. That said, there is an occasional, rare need for the creation of attributes which can handle larger chunks of binary data. In the 2.6.0−test kernel, the only use of binary attributes is in the `firmware` subsystem. When a device requiring firmware is encountered in the system, a user−space program can be started (via the hotplug mechanism); that program then passes the firmware code to the kernel via binary sysfs attribute. If you are contemplating any other use of binary attributes, you should think carefully and be sure there is no other way to accomplish your objective.

Binary attributes are described with a `bin_attribute` structure:

```
struct bin_attribute {
    struct attribute attr;
    size_t size;
    ssize_t (*read)(struct kobject *kobj, char *buffer,
                    loff_t pos, size_t size);
    ssize_t (*write)(struct kobject *kobj, char *buffer,
```

```
                        loff_t pos, size_t size);
};
```

Here, `attr` is an `attribute` structure giving the name, owner, and permissions for the binary attribute, and `size` is the maximum size of the binary attribute (or zero if there is no maximum). The `read()` and `write()` functions work similarly to the normal char driver equivalents; they can be called multiple times for a single load with a maximum of one page worth of data in each call. There is no way for sysfs to signal the last of a set of write operations, so code implementing a binary attribute must be able to determine that some other way.

Binary attributes must be created explicitly; they cannot be set up as default attributes. To create a binary attribute, call:

```
int sysfs_create_bin_file(struct kobject *kobj,
                          struct bin_attribute *attr);
```

Binary attributes can be removed with:

```
int sysfs_remove_bin_file(struct kobject *kobj,
                          struct bin_attribute *attr);
```

Last notes

This article has described the low−level interface between kobjects and sysfs. Unless you are implementing a new subsystem, however, you are unlikely to work with this interface directly. Each subsystem typically implements its own set of default attributes, and, perhaps, a mechanism for interested code to add new ones. This mechanism is generally a straightforward wrapper around the low−level attribute code, however, so it should look familiar to readers of this page.

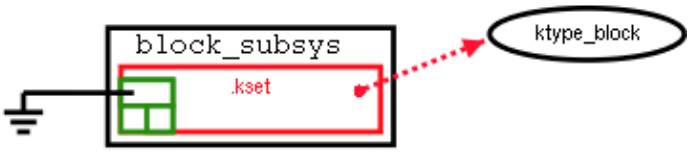
No comments have been posted. Post one now

Examining a kobject hierarchy

This article is part of the LWN Porting Drivers to 2.5 series.

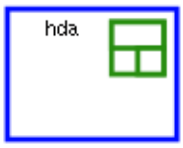
The Driver Porting Series now includes several articles on how kobjects work as a way of tieing together data structures and managing reference counts. Experience shows, however, that truly envisioning how kobject−linked data structures tie together is a difficult task. In the hope of shedding a bit more light in this direction, and as a way for your editor to exercise his minimal skills with the "dia" diagram editor, this article will show how some of the crucial data structures in the block layer are connected.

The core data structure in this investigation is the kobject. In the diagrams that follow, kobjects will be represented by the small symbol you see to the right. The upper rectangle represents the kobject's `parent` field, while the other two are its entries in the doubly−linked list that implements a kset. Not all kobjects belong to a kset, so those links will often be empty.

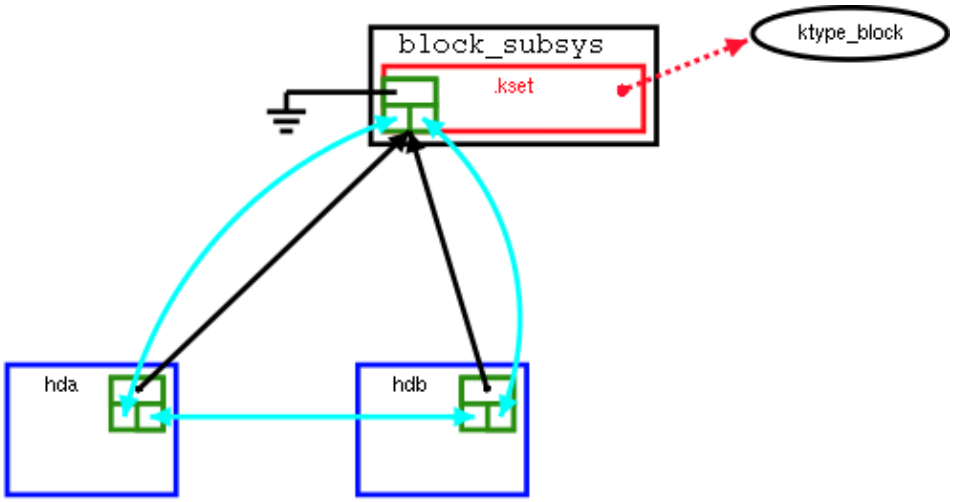


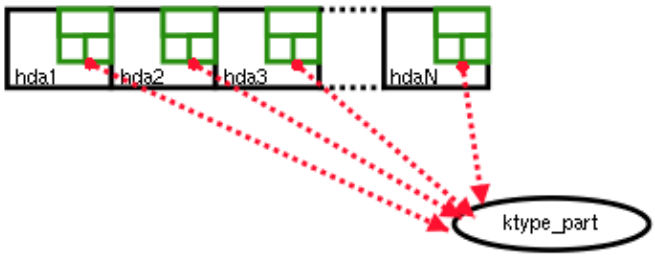
At the root of the block subsystem hierarchy is a subsystem called `block_subsys`; it is defined in `drivers/block/genhd.c`. As you'll recall from The Zen of Kobjects, a subsystem is a very simple structure, consisting of a semaphore and a kset. The kset will define, in its `ktype` field, what type of kobjects it will contain; for `block_subsys`, this field is set to `ktype_block`. Pictorially, we can show this structure as seen on the right.

Each kset contains its own kobject, and `block_subsys` is no exception. In this case, the kobject's parent field is explicitly set to `NULL` (indicated by the ground symbol in the picture). As a result, this kobject will be represented in the top level of the sysfs hierarchy; it is the kobject which lurks behind `/sys/block`.



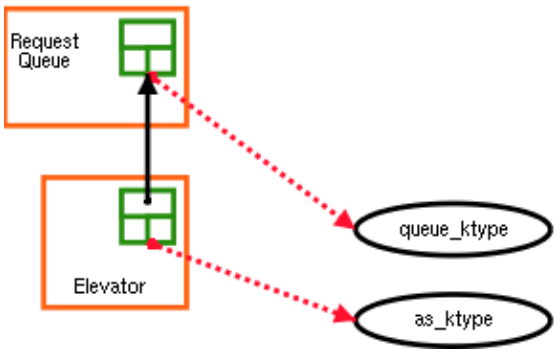
A block subsystem is not very interesting without disks. In the block hierarchy, disks are defined by a `struct gendisk`, which can be found in `<include/linux/genhd.h>`. The gendisk interface is described in this article. For our purposes, we will represent a gendisk as seen on the left; note that it has the inevitable embedded kobject inside it. A gendisk's kobject does not have an explicit type pointer; its membership in the `block_subsys` kset takes care of that. But its `parent` and `kset` pointers both point to the kobject within `block_subsys`, and the kset pointers are there too. The result, for a system with two disks, would be a structure that looks like this:





Things do not end there, however; a gendisk structure is a complicated thing. It contains, among other things, an array of partition entries (of type `struct hd_struct`), each of which has embedded within it, yes, a kobject. The parent of each partition is the disk which contains it. It would have been possible to implement the list of partitions as a kset, but things weren't done that way. Partitions are a relatively static item, and their ordering matters, so they were done as a simple array. We depict that array as seen on the right.

As you can see, the kobject type of a partition is `ktype_part`. This type implements the attributes you will see in the sysfs entries for each partition, including the starting block number and size.

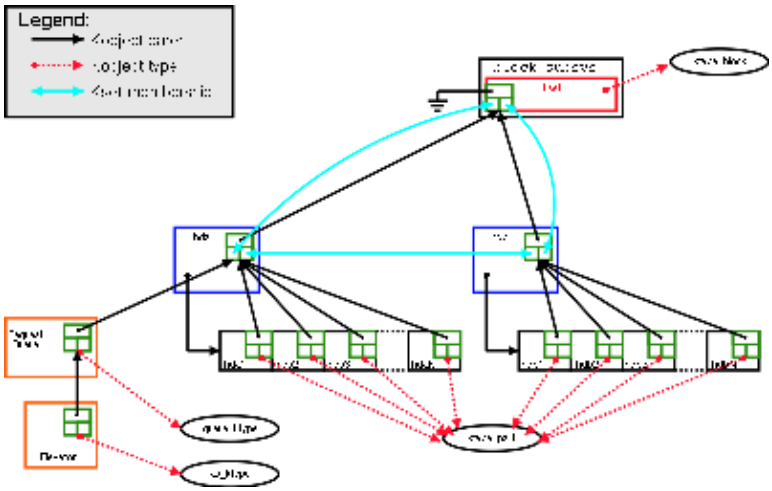


Another item associated with each gendisk is its I/O request queue. The queue, too, contains a kobject (of type `queue_ktype`) whose parent is the associated gendisk. The I/O scheduler ("elevator") in use with an I/O request queue is also represented in the hierarchy. The scheduler's kobject's type depends on which scheduler is being used; the (default) anticipatory scheduler uses `as_ktype`. The resulting piece of the puzzle looks as portrayed on the left.

The request queue and I/O scheduler information in sysfs is currently read−only. There is no reason, however, why sysfs attributes could not be used to change I/O scheduling parameters on the fly. The selectable I/O scheduler patch uses sysfs attributes to change I/O schedulers completely, for example.

Putting it all together

So far, we have seen a number of disconnected pieces. The full diagram can be found on this page; it is a bit wide to be placed inline with the text (a small, illegible version appears to the right). Also on that page, you'll find a corresponding diagram showing the sysfs names the correspond to each kobject.



The data structure as described is the full implementation of the `/sys/block` subtree of sysfs. The full sysfs tree contains rather more than this, of course. For each gendisk which shows up under `/sys/block`, there will be a separate entry under `/sys/devices` which describes the underlying hardware. Internally, the link between the two is contained in the `driverfs_dev` field of the gendisk structure. In sysfs, that link is represented as a symbolic link between the two sub−trees.

Hopefully this series of pictures helps in the visualization of a portion of the sysfs tree and the device model data structure that implements it. The device model brings a great deal of apparent complexity, but, once the underlying concepts are grasped, the whole thing is approachable.

Post a comment

legible images

(Posted Nov 1, 2003 1:45 UTC (Sat) by **roelofs**) (Post reply)

a small, illegible version appears to the right

It could be more legible than it is with proper resizing (or, more specifically, with proper resampling). I happen to be most familiar with XV, so I'll describe how to do it there, but I know other viewer/converters (and the GIMP) have similar capabilities.

In XV, make sure the mode is set to 24−bit (even for a palette image like this−−smooth resizing requires more colors), set the size to whatever you like ("S" key or Image Size −> Set Size), smooth the reduced image ("s" key or Display −> Smooth), and then save it. As a PNG this will come out in RGB mode, but you can then reopen the small image, change the mode back to 8−bit, and save it as a colormapped PNG for a file−size reduction with minimal quality loss. Not everything in this particular image will be readable, but the legend certainly will be, as will much of the other text.

Greg

Examining a kobject hierarchy

(Posted Nov 11, 2003 1:08 UTC (Tue) by **mmarq**) (Post reply)

Just a ideia.

Unfortunately i cant tell by the source (i dont think so) if the I2O Linux kernel implementation follows the V2 spec defined at (http://www.intelligent−io.com/specs_resources/V2.0_spec.pdf), but wouldn't it be grand if this, here exposed, 2.5 driver model could overlap, in the future with the I2O model at the communications level,..., that is, in a future Linux 3.0(?) spilt driver model, a "regular" driver module could "talk" with a I2O driver without any special translation layer or any other craft!

IMO whats missing in this 2.5 driver model is a communications mechanism between the various "subsystems" or " driver specific kobjects", because of multi−purposed hardware and combo peripherals...

So Why Not ?

A "special" subsystem that implements "at least" a "host MessengerInstance" type of I2O V2 made of kobjects, so that this messaging layer could be used by a kind of kobjects driver model and at the same time by the I2O model...

IMO this messaging layer rocks at the thecnical front, and also as it says in the draft spec paper:
" The architecture(MessengerInstance messaging layer) is independent of the operanting system, processor platform, and system I/O bus. This version of the specfication defines a transport interface between MessegerInstances for a shared memory environment, but does not preclude defining other transport enviroments in the future revisions."

A sure anti lock−in, and future prove model,... No wonder M$ that had backed it with Intel droped it in the recycle bin,... Could also mean a great help for stoping the lack of support from the hardware industry to the Linux project.

Creating Linux virtual filesystems

This article is part of the LWN Porting Drivers to 2.6 series.
[This article has been reworked to reflect changes in the libfs interface; those who are interested can still read the original version.]

Linus and numerous other kernel developers dislike the `ioctl()` system call, seeing it as an uncontrolled way of adding new system calls to the kernel. Putting new files into `/proc` is also discouraged, since that area is seen as being a bit of a mess. Developers who populate their code with `ioctl()` implementations or `/proc` files are often encouraged to create a standalone virtual filesystem instead. Filesystems make the interface explicit and visible in user space; they also make it easier to write scripts which perform administrative functions. But the writing of a Linux filesystem can be an intimidating task. A developer who has spent some time just getting up to speed on the driver interface can be forgiven for balking at having to learn the VFS API as well.

The 2.6 kernel contains a set of routines called "libfs" which is designed to make the task of writing virtual filesystems easier. libfs handles many of the mundane tasks of implementing the Linux filesystem API, allowing non−filesystem developers to concentrate (mostly) on the specific functionality they want to provide. What it lacks, however, is documentation. This article is an attempt to fill in that gap a little bit.

The task we will undertake is not particularly ambitious: export a simple filesystem (of type "lwnfs") full of counter files. Reading one of these files yields the current value of the counter, which is then incremented. This leads to the following sort of exciting interaction:

```
# cat /lwnfs/counter
0
# cat /lwnfs/counter
1
# ...
```

Your author was able to amuse himself well into the thousands this way; some users may tire of this game sooner, however. The impatient can get to higher values more quickly by writing to the counter file:

```
# echo 1000 > /lwnfs/counter
# cat /lwnfs/counter
1000
#
```

OK, so the Linux distributors will probably not get to excited about advertising the new "lwnfs" capability. But it works as a way of showing how to create virtual filesystems. For those who are interested, the full source is available.

Initialization and superblock setup

So let's get started. A loadable module which implements a filesystem must, at load time, register that filesystem with the VFS layer. The lwnfs module initialization code is simple:

```
static int __init lfs_init(void)
{
        return register_filesystem(&lfs_type);
}
module_init(lfs_init);
```

The `lfs_type` argument is a structure which is set up as follows:

```
static struct file_system_type lfs_type = {
        .owner      = THIS_MODULE,
        .name       = "lwnfs",
        .get_sb     = lfs_get_super,
        .kill_sb    = kill_litter_super,
};
```

This is the basic data structure which describes a filesystem type to the kernel; it is declared in `<linux/fs.h>`. The `owner` field is used to manage the module's reference count, preventing unloading of the module while the filesystem code is in use. The `name` is what eventually ends up on a `mount` command line in user space. Then there are two functions for managing the filesystem's superblock − the root of the filesystem data structure. `kill_litter_super()` is a generic function provided by the VFS; it simply cleans up all of the in−core structures when the filesystem is unmounted; authors of simple virtual filesystems need not worry about this aspect of things. (It *is* necessary to unregister the filesystem at unload time, of course; see the source for the lwnfs exit function).

In many cases, the creation of the superblock must be done by the filesystem programmer −− but see the "a simpler way" section below. This task involves a bit of boilerplate code. In this case, `lfs_get_super()` hands off the task as follows:

```
static struct super_block *lfs_get_super(struct file_system_type *fst,
                int flags, const char *devname, void *data)
{
        return get_sb_single(fst, flags, data, lfs_fill_super);
}
```

Once again, `get_sb_single()` is generic code which handles much of the superblock creation task. But it will call `lfs_fill_super()`, which performs setup specific to our particular little filesystem. It's prototype is:

```
static int lfs_fill_super (struct super_block *sb,
                           void *data, int silent);
```

The in−construction superblock is passed in, along with a couple of other arguments that we can ignore. We do have to fill in some of the superblock fields, though. The code starts out like this:

```
        sb->s_blocksize = PAGE_CACHE_SIZE;
        sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
        sb->s_magic = LFS_MAGIC;
        sb->s_op = &lfs_s_ops;
```

Most virtual filesystem implementations have something that looks like this; it's just setting up the block size of the filesystem, a "magic number" to recognize superblocks by, and the superblock operations. These operations need not be written for a simple virtual filesystem − libfs has the stuff that is needed. So `lfs_s_ops` is defined (at the top file level) as:

```
static struct super_operations lfs_s_ops = {
        .statfs         = simple_statfs,
        .drop_inode     = generic_delete_inode,
};
```

Creating the root directory

Getting back into `lfs_fill_super()`, our big remaining task is to create and populate the root directory for our new filesystem. The first step is to create the inode for the directory:

```
        root = lfs_make_inode(sb, S_IFDIR | 0755);
        if (! root)
                goto out;
        root->i_op = &simple_dir_inode_operations;
        root->i_fop = &simple_dir_operations;
```

`lfs_make_inode()` is a boilerplate function that we will look at eventually; for now, just assume that it returns a new, initialized inode that we can use. It needs the superblock and a `mode` argument, which is just like the mode value returned by the `stat()` system call. Since we passed `S_IFDIR`, the returned inode will describe a directory. The file and directory operations that we assign to this inode are, again, taken from libfs.

This directory inode must be put into the directory cache (by way of a "dentry" structure) so that the VFS can find it; that is done as follows:

```
        root_dentry = d_alloc_root(root);
        if (! root_dentry)
                goto out_iput;
        sb->s_root = root_dentry;
```

Creating files

The superblock now has a fully initialized root directory. All of the actual directory operations will be handled by libfs and the VFS layer, so life is easy. What libfs cannot do, however, is actually put anything of interest into that root directory that's our job. So the final thing that `lfs_fill_super()` does before returning is to call:

```
        lfs_create_files(sb, root_dentry);
```

In our sample module, `lfs_create_files()` creates one counter file in the root directory of the filesystem, and another in a subdirectory. We'll look mostly at the root−level file. The counters are implemented as `atomic_t` variables; our top−level counter (called, with great imagination, "`counter`") is set up as follows:

```
static atomic_t counter;

static void lfs_create_files (struct super_block *sb,
                              struct dentry *root)
{
```

```
            /* ... */
            atomic_set(&counter, 0);
            lfs_create_file(sb, root, "counter", &counter);
            /* ... */
    }
```

`lfs_create_file` does the real work of making a file in a directory. It has been made about as simple as possible, but there are still a few steps to be performed. The function starts out as:

```
static struct dentry *lfs_create_file (struct super_block *sb,
                struct dentry *dir, const char *name,
                atomic_t *counter)
{
        struct dentry *dentry;
        struct inode *inode;
        struct qstr qname;
```

Arguments include the usual superblock structure, and `dir`, the dentry for the directory that will contain this file. In this case, `dir` will be the root directory we created before, but it could be any directory within the filesystem.

Our first task is to create a directory entry for the new file:

```
        qname.name = name;
        qname.len = strlen (name);
        qname.hash = full_name_hash(name, qname.len);
        dentry = d_alloc(dir, &qname);
```

The setting up of `qname` just hashes the file name so that it can be found quickly in the dentry cache. Once that's done, we create the entry within our parent `dir`. The file also needs an inode, which we create as follows:

```
        inode = lfs_make_inode(sb, S_IFREG | 0644);
        if (! inode)
                goto out_dput;
        inode->i_fop = &lfs_file_ops;
        inode->u.generic_ip = counter;
```

Once again, we call `lfs_make_inode` (which we will look at shortly, honest), but this time we use it to create a regular file. The key to the creation of special-purpose files in virtual filesystems is to be found in the other two assignments:

- The `i_fop` field is set up with our file operations which will actually implement reads and writes on the counter.
- We use the `u.generic_ip` pointer in the inode to stash aside a pointer to the `atomic_t` counter associated with this file.

In other words, `i_fop` defines the behavior of this particular file, and `u.generic_ip` is the file-specific data. All virtual filesystems of interest will make use of these two fields to set up the required behavior.

The last step in creating a file is to add it to the dentry cache:

```
        d_add(dentry, inode);
        return dentry;
```

Putting the inode into the dentry cache allows the VFS to find the file without having to consult our filesystem's directory operations. And that, in turn, means our filesystem does not need to *have* any directory operations of interest. The entire structure of our virtual filesystem lives in the kernel's cache structure, so our module need not remember the structure of the filesystem it has set up, and it need not implement a lookup operation. Needless to say, that makes life easier.

Inode creation

Before we get into the actual implementation of the counters, it's time to look at `lfs_make_inode()`. The function is pure boilerplate; it looks like:

```
static struct inode *lfs_make_inode(struct super_block *sb, int mode)
{
        struct inode *ret = new_inode(sb);

        if (ret) {
                ret->i_mode = mode;
                ret->i_uid = ret->i_gid = 0;
                ret->i_blksize = PAGE_CACHE_SIZE;
                ret->i_blocks = 0;
                ret->i_atime = ret->i_mtime = ret->i_ctime = CURRENT_TIME;
        }
        return ret;
}
```

It simply allocates a new inode structure, and fills it in with values that make sense for a virtual file. The assignment of `mode` is of interest; the resulting inode will be a regular file or a directory (or something else) depending on how `mode` was passed in.

Implementing file operations

Up to this point, we have seen very little that actually makes the counter files work; it's all been VFS boilerplate so that we have a little filesystem to put those counters into. Now the time has come to see how the real work gets done.

The operations on the counters themselves are to be found in the `file_operations` structure that we associate with the counter file inodes:

```
    static struct file_operations lfs_file_ops = {
            .open       = lfs_open,
            .read       = lfs_read_file,
            .write      = lfs_write_file,
    };
```

A pointer to this structure, remember, was stored in the inode by `lfs_create_file()`.

The simplest operation is `open()`:

```
    static int lfs_open(struct inode *inode, struct file *filp)
    {
            filp->private_data = inode->u.generic_ip;
            return 0;
    }
```

The only thing this function need do is copy the pointer to the `atomic_t` pointer over into the `file` structure, which makes it a bit easier to get at.

The interesting work is done by the `read()` function, which must increment the counter and return its value to the user space program. It has the usual `read()` operation prototype:

```
    static ssize_t lfs_read_file(struct file *filp, char *buf,
                                 size_t count, loff_t *offset)
```

It starts by reading and incrementing the counter:

```
        atomic_t *counter = (atomic_t *) filp->private_data;
        int v = atomic_read(counter);
        atomic_inc(counter);
```

This code has been simplified a bit; see the module source for a couple of grungy, irrelevant details. Some readers will also notice a race condition here: two processes could read the counter before either increments it; the result would be the same counter value returned twice, with certain dire results. A serious module would probably serialize access to the counter with a spinlock. But this is supposed to be a simple demonstration.

So anyway, once we have the value of the counter, we have to return it to user space. That means encoding it into character form, and figuring out where and how it fits into the user-space buffer. After all, a user-space program can seek around in our virtual file.

```
        len = snprintf(tmp, TMPSIZE, "%d\n", v);
        if (*offset > len)
                return 0;
        if (count > len - *offset)
                count = len - *offset;
```

Once we've figured out how much data we can copy back, we just do it, adjust the file offset, and we're done.

```
        if (copy_to_user(buf, tmp + *offset, count))
                return -EFAULT;
        *offset += count;
        return count;
```

Then, there is `lfs_write_file()`, which allows a user to set the value of one of our counters:

```
static ssize_t lfs_write_file(struct file *filp, const char *buf,
                size_t count, loff_t *offset)
{
        atomic_t *counter = (atomic_t *) filp->private_data;
        char tmp[TMPSIZE];

        if (*offset != 0)
                return -EINVAL;
        if (count >= TMPSIZE)
                return -EINVAL;

        memset(tmp, 0, TMPSIZE);
        if (copy_from_user(tmp, buf, count))
                return -EFAULT;
        atomic_set(counter, simple_strtol(tmp, NULL, 10));
        return count;
}
```

That is just about it. The module also defines `lfs_create_dir`, which creates a directory in the filesystem; see the full source for how that works.

A simpler way

The above example contains a great deal of scary−looking boilerplate code. That boilerplate will be necessary for many applications, but there is a shortcut that will work for many others. If you know at compile time which files you wish to create, and you do not need to make subdirectories, read on for the easier way.

In this section, we'll talk about a different version of the lwnfs module − one which eliminates about 1/3 of the code. It implements a simple array of four counters, with no subdirectories. Once again, full source is available if you are interested.

Above, we looked at a function called `lfs_fill_super()`, which fills in the filesystem superblock, creates the root directory, and populates it with files. In the simpler version, the entire function becomes the following:

```
static int lfs_fill_super(struct super_block *sb, void *data, int silent)
{
        return simple_fill_super(sb, LFS_MAGIC, OurFiles);
}
```

`simple_fill_super()` is a libfs function which does almost everything we need. Its actual prototype is:

```
int simple_fill_super(struct super_block *sb, int magic,
                      struct tree_descr *files);
```

The `struct super_block` argument can be passed directly through, and `magic` is the same magic number we saw above. The `files` argument describes which files should be created in the filesystem; the relevant structure is defined as follows:

```
struct tree_descr {
        char *name;
        struct file_operations *ops;
        int mode;
};
```

The arguments should be fairly obvious by now; each structure gives the name of the file to be created, the file operations to associate with the file, and the protection bits for the file. There are, however, a couple of quirks about how the array of `tree_descr` structures should be built:

- Entries which are filled with `NULL`s (more strictly, where `name` is `NULL`) are simply ignored. Do not try to end the list with a `NULL−filled` structure, unless you like decoding oops listings.
- The list is terminated, instead, by an entry that sets `name` to the empty string.
- The entries correspond directly to the inode numbers which will be assigned to the resulting files. This knowledge can be used to figure out, in the file operations code, which file is being opened. But this feature also implies that the first entry in the list cannot be used, since the filesystem root directory will take inode zero. So, when you create your `tree_descr` list, the first entry should be `NULL`.

Having painfully learned all of the above, your author has set up the list for the four "counter" files as follows:

```
static struct tree_descr OurFiles[] = {
        { NULL, NULL, 0 },                  /* Skipped */
        { .name = "counter0",               /* Inode 1 */
          .ops = &lfs_file_ops,
          .mode = S_IWUSR|S_IRUGO },
        { .name = "counter1",               /* Inode 2 */
          .ops = &lfs_file_ops,
          .mode = S_IWUSR|S_IRUGO },
        { .name = "counter2",               /* Inode 3 */
          .ops = &lfs_file_ops,
          .mode = S_IWUSR|S_IRUGO },
        { .name = "counter3",               /* Inode 4 */
          .ops = &lfs_file_ops,
          .mode = S_IWUSR|S_IRUGO },
        { "", NULL, 0 }             /* Terminates the list */
};
```

Once the call to `simple_fill_super()` returns, the work is done and your filesystem is live. The only remaining detail might be in your `open()` method; if you have multiple files sharing the same `file_operations` structure, you will need to figure out which one is actually being acted upon. The key here is the inode number, which can be found in the `i_ino` field. The modified version of `lfs_open()` finds the right counter as follows:

```
static int lfs_open(struct inode *inode, struct file *filp)
{
        if (inode->i_ino > NCOUNTERS)
                return -ENODEV;  /* Should never happen.  */
        filp->private_data = counters + inode->i_ino - 1;
        return 0;
}
```

The `read()` and `write()` functions use the `private_data` field, and thus need not be modified from the previous version.

Conclusion

The libfs code, as demonstrated here, is sufficient for a wide variety of driver-specific virtual filesystems. Further examples can be found in the 2.5 kernel source in a few places:

- `drivers/hotplug/pci_hotplug_core.c`
- `drivers/usb/core/inode.c`
- `drivers/oprofile/oprofilefs.c`
- `fs/ramfs/inode.c`
- `fs/nfsd/nfsctl.c` (`simple_fill_super()` example)

...and in a few other spots `grep` is your friend.

Keep in mind that the 2.6 driver model code makes it easy for drivers to export information within its own virtual filesystem; for many applications, that will be the preferred way of making information available to user space. The Driver Porting Series has several articles on the driver model and sysfs. For cases where only a custom filesystem will do, however, libfs makes the task (relatively) easy.

No comments have been posted. Post one now

A simple block driver

[LWN subscriber−only content]

This article is part of the LWN Porting Drivers to 2.6 series.

Given the large number of changes to the 2.6 block layer, it is hard to know where to start describing them. We'll begin by examining the simplest possible block driver. The sbd ("simple block device") driver simulates a block device with a region of kernel memory; it is, essentially, a naive ramdisk driver implemented in less than 200 lines of code. It will allow the demonstration of some changes in how block drivers work with the rest of the system without the need for all the complexity required when one is dealing with real hardware. Code fragments will be shown below; the full driver source can be found on this page.

If you have not read the block layer overview, you might want to head over there for a moment; this article will still be here when you get back.

Initialization

In our simple driver, the module initialization function is called `sbd_init()`. Its job, of course, is to get set up for block operations and to make its disk available to the system. The first step is to set up our internal data structure; within the driver a disk (*the* disk, in this case) is represented by:

```
static struct sbd_device {
    unsigned long size;
    spinlock_t lock;
    u8 *data;
    struct gendisk *gd;
} Device;
```

Here `size` is the size of the device (in bytes), `data` is the array where the "disk" stores its data, `lock` is a spinlock for mutual exclusion, and `gd` is the kernel representation of our device.

The device initialization is pretty straightforward; it is just a matter of allocating the memory to actually store the data and initializing the spinlock:

```
Device.size = nsectors*hardsect_size;
spin_lock_init(&Device.lock);
Device.data = vmalloc(Device.size);
if (Device.data == NULL)
    return −ENOMEM;
```

(`nsectors` and `hardsect_size` are load−time parameters that control how big the device should be).

About now is where block drivers traditionally register themselves with the kernel, and `sbd` does that too:

```
major_num = register_blkdev(major_num, "sbd");
if (major_num <= 0) {
    printk(KERN_WARNING "sbd: unable to get major number\n");
```

```
        goto out;
    }
```

Note that, in 2.6, no device operations structure is passed to `register_blkdev()`. As it turns out, a block driver can happily get by without calling `register_blkdev()` at all. That function does little work, at this point, and will likely be removed sooner or later. About the only remaining tasks performed by `register_blkdev()` are the assignment of a dynamic major number (if requested), and causing the block driver to show up in `/proc/devices`.

Generic disks

If `register_blkdev()` no longer does anything, where does the real work get done? The answer lies in the much improved 2.6 "generic disk" (or "gendisk") code. The gendisk interface is covered in a separate article, so we'll look only quickly at how `sbd` does its gendisk setup.

The first step is to get a `gendisk` structure to represent the `sbd` device:

```
    Device.gd = alloc_disk(16);
    if (! Device.gd)
        goto out_unregister;
```

Note that a memory allocation is involved, so the return value should be checked. The parameter to `alloc_disk()` indicates the number of minor numbers that should be dedicated to this device. We have requested 16 minor numbers, meaning that the device will support 15 partitions.

The gendisk must be initialized; the `sbd` driver starts that task as follows:

```
    Device.gd->major = major_num;
    Device.gd->first_minor = 0;
    Device.gd->fops = &sbd_ops;
    Device.gd->private_data = &Device;
    strcpy (Device.gd->disk_name, "sbd0");
    set_capacity(Device.gd, nsectors*(hardsect_size/KERNEL_SECTOR_SIZE));
```

Most of the above should be relatively self-explanatory. The `fops` field is a pointer to the `block_device_operations` structure for this device; we'll get to that shortly. The `private_data` field can be used by the driver, so we stick a pointer to our `sbd_device` structure there. The `set_capacity()` call tells the kernel how large the device is. Note that the kernel can handle block devices which have sectors greater than 512 bytes, but it always deals with 512-byte sectors internally. So we need to normalize the sector count before passing it to the kernel.

Another thing that (usually) goes into the gendisk is the request queue to use. The `BLK_DEFAULT_QUEUE` macro from 2.4 is no more; a block driver must explicitly create and set up the request queue(s) it will use. Furthermore, request queues must be allocated dynamicly, at run time. The `sbd` driver sets up its request queue as follows:

```
    static struct request_queue *Queue;
    /* ... */
    Queue = blk_init_queue(sbd_request, &Device.lock);
    if (Queue == NULL)
            goto out;
    blk_queue_hardsect_size(Queue, hardsect_size);
    Device.gd->queue = Queue;
```

Here, `sbd_request` is the request function, which we will get to soon. Note that a spinlock must be passed into `blk_init_queue()`. The global `io_request_lock` is gone forevermore, and each block driver must manage its own locking. Typically, the lock used by the driver to serialize access to internal resources is the best choice for controlling access to the request queue as well. For that reason, the block layer expects the driver to provide a lock of its own for the queue. If a nonstandard hard sector size (i.e. not 512 bytes) is in use, the sector size should be stored into the request queue with `blk_queue_hardsect_size()`. Finally, a pointer to the queue must be stored in the `gendisk` structure.

At this point, the gendisk setup is complete. All that remains is to add the disk to the system:

```
add_disk(Device.gd);
```

Note that `add_disk()` may well generate I/O to the device before it returns − the driver must be in a state where it can handle requests before adding disks. The driver also should not fail initialization after it has successfully added a disk.

What you don't have to do

That is the end of the initialization process for the `sbd` driver. What you don't have to do is as notable as what does need to be done. For example, there are no assignments to global arrays; the whole set of global variables that used to describe block devices is gone. There is also nothing here for dealing with partition setup. Partition handling is now done in the generic block layer, and there is almost nothing that individual drivers must do at this point. "Almost" because the driver must handle one `ioctl()` call, as described below.

Open and release

The `open` and `release` methods (which are kept in the `block_device_operations` structure) actually have not changed since 2.4. The `sbd` driver has nothing to do at open or release time, so it doesn't even bother to define these methods. Drivers for real hardware may need to lock and unlock doors, check for media, etc. in these methods.

The request method

The core of a block driver, of course, is its `request` method. The `sbd` driver has the simplest possible `request` function; it does not concern itself with things like request clustering, barriers, etc. It does not understand the new `bio` structure used to represent requests at all. But it works. Real drivers will almost certainly require a more serious `request` method; see the other Driver Porting Series articles for the gory details on how to do that.

Here is the whole thing:

```
static void sbd_request(request_queue_t *q)
{
    struct request *req;

    while ((req = elv_next_request(q)) != NULL) {
        if (! blk_fs_request(req)) {
            end_request(req, 0);
            continue;
        }
        sbd_transfer(&Device, req->sector, req->current_nr_sectors,
                     req->buffer, rq_data_dir(req));
```

```
            end_request(req, 1);
        }
    }
```

The first thing to notice is that all of the old `<linux/blk.h>` cruft has been removed. Macros like `INIT_REQUEST` (with its hidden `return` statement), `CURRENT`, and `QUEUE_EMPTY` are gone. It is now necessary to deal with the request queue functions directly, but, as can be seen, that is not particularly hard.

Note that the `Device.lock` will be held on entry to the `request` function, much like `io_request_lock` is in 2.4.

The function for getting the first request in the queue is now `elv_next_request()`. A `NULL` return means that there are no more requests on the queue that are ready to process. A simple request loop like this one can simply run until the request queue is empty; drivers for real hardware will also have to take into account how many operations the device can handle, of course. Note that this function does *not* actually remove the request from the queue; it just returns a properly adjusted view of the top request.

Note also that, in 2.6, there can be multiple types of requests. Thus the test:

```
        if (! blk_fs_request(req)) {
            end_request(req, 0);
            continue;
        }
```

A nonzero return value from the `blk_fs_request()` macro says "this is a normal filesystem request." Other types of requests (i.e. packet−mode or device−specific diagnostic operations) are not something that `sbd` supports, so it simply fails any such requests.

The function `sbd_transfer()` is really just a `memcpy()` with some checking; see the full source if you are interested. The key is in the parameters: the various fields of the `request` structure (`sector`, `current_nr_sectors`, and `buffer`) look just like they did in 2.4. They also have the same meaning: they are a window looking at the first part of a (possibly larger) request. If you deal with block requests at this level, you need know nothing of the `bio` structures underlying the request. This approach only works for the simplest of drivers, however.

Note that the direction of the request is now found in the `flags` field, and can be tested with `rq_data_dir()`. A nonzero value (`WRITE`) indicates that this is a write request. Note also the absence of any code adding partition offsets; all of that is handled in the higher layers.

Finally, `end_request()` is called to finish processing of this request. This function has picked up a new parameter in 2.6, being the pointer to the `request` structure.

Other block operations

The two other `block_device_operations` methods from 2.4 − `check_media_change()` and `revalidate()` − have seen prototype changes in 2.5. They are now called `media_changed()` and `revalidate_disk()`, and both take a `gendisk` structure as their only argument. The basic task performed by these methods remains unchanged, however.

In 2.4, a block driver's `ioctl()` method would handle any commands it understood, and pass the rest on to `blk_ioctl()` for generic processing. In 2.6, the generic code gets the first crack at any `ioctl()` calls, and only invokes the driver for those it can't implement itself. As a result, `ioctl()` methods in drivers can

often be pretty small. The `sbd` driver includes an `ioctl` method which handles a single command:

```
int sbd_ioctl (struct inode *inode, struct file *filp,
               unsigned int cmd, unsigned long arg)
{
    long size;
    struct hd_geometry geo;

    switch(cmd) {
    /*
     * The only command we need to interpret is HDIO_GETGEO, since
     * we can't partition the drive otherwise.  We have no real
     * geometry, of course, so make something up.
     */
        case HDIO_GETGEO:
            size = Device.size*(hardsect_size/KERNEL_SECTOR_SIZE);
            geo.cylinders = (size & ~0x3f) >> 6;
            geo.heads = 4;
            geo.sectors = 16;
            geo.start = 4;
            if (copy_to_user((void *) arg, &geo, sizeof(geo)))
                    return -EFAULT;
            return 0;
    }
    return -ENOTTY; /* unknown command */
}
```

The notion of a regular geometry has been fiction for most devices for some years now. Tools like `fdisk` still work with cylinders, however, so a driver must make up some sort of convincing geometry story. The `sbd` implementation claims four heads and 16 sectors per cylinder, but anything else reasonable would have worked as well.

Shutting down

The last thing to look at is what happens when the module is unloaded. We must, of course, clean up our various data structures and free memory − the usual stuff. The `sbd` cleanup function looks like this:

```
static void __exit sbd_exit(void)
{
    del_gendisk(Device.gd);
    put_disk(Device.gd);
    unregister_blkdev(major_num, "sbd");
    blk_cleanup_queue(Queue);
    vfree(Device.data);
}
```

`del_gendisk()` cleans up any partitioning information, and generally makes the system forget about the gendisk passed to it. The call to `put_disk()` then releases our reference to the `gendisk` structure (obtained when we first called `alloc_disk()`) so that it can be freed. Then, of course, we must free the memory used for the device itself (only after the gendisk has been cleaned up, so we know no more operations can be requested), release the request queue, and unregister the block device.

Conclusion

That is about as simple as it gets; the above implements a true virtual block device that can support a filesystem. Real drivers, of course, will tend to be more complicated. For details on how to make them more

complicated, continue with the Driver Porting Series; the next block driver article is The Gendisk Interface.

No comments have been posted. Post one now