Concrete Architecture of the Linux Kernel

Ivan Bowman	Saheem Siddiqi	Meyer C. Tanuan
(ibowman@sybase.com)	(s4siddiqi@neumann)	(mtanuan@descartes.com)

Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1

CS 746G, Winter 1998
12-Feb-98

Available at: http://plg.uwaterloo.ca/~itbowman/CS746G/a2

Keywords: Reverse Engineering, concrete architecture, Linux.

Abstract

The objective of this report is to describe the concrete (as-built) architecture of the Linux kernel. A concrete architecture description of the Linux kernel serves as a high-level specification for developers to modify and extend the existing kernel source code.

We used a reverse engineering tool (Portable Bookshelf) to extract the detailed design information out of the Linux kernel source. The biggest challenge we had to overcome was to cluster the enormous volume of extracted information into subsystems. We clustered the extracted information based on our domain-specific knowledge about modern operating systems; the Software Landscape visualization tool was used in

an iterative process to view and refine this clustering.

Although the Portable Bookshelf tool provided detailed design information and some initial clustering based on file names, we found that this was not sufficient to form a concrete architecture description. The extraction tool didn't accurately extract dependencies. It missed some dependencies, and asserted some that did not exist. The tool didn't automatically provide a structure that helped understand the system. We verified the extracted design facts and refined the clustering using our conceptual model of the Linux kernel.

We concluded that the Portable Bookshelf tool by itself cannot accurately describe the concrete architecture of the Linux kernel. We remedied this limitation by using domain-specific knowledge to describe the concrete architecture.

Contents

Figures

---

1 Introduction

1.1 Purpose

The goal of this report is to describe the concrete architecture of the Linux kernel. Concrete architecture refers to the architecture of the system as it is built. We intend to develop the concrete architecture to provide high-level documentation of the existing Linux kernel.

1.2 Introduction to Linux

Linus B. Torvalds wrote the first Linux kernel in 1991. Linux gained in popularity because it has always been distributed as free software. Since the source code is readily available, users can freely change the kernel to suit their needs. However, it is important to understand how the Linux kernel has evolved and how it currently works before new system programs are written.

A concrete architecture based on the Linux kernel source code can provide a reliable and up-to-date reference for Linux kernel hackers and developers. Linux has been revised several times since 1991 by a group of volunteers who communicate through the USENET newsgroups on the Internet. To date, Torvalds has acted as the as the main kernel developer. In the event that Linus Torvalds is no longer part of the Linux kernel project, we can reasonably expect that the Linux kernel can be enhanced and modified if an accurate and up-to-date concrete architecture is maintained.

Linux is a Unix-compatible system. Most of the common Unix tools and programs now run under Linux. Linux was originally developed to run on the Intel 80386 microprocessor. The original version was not readily portable to other platforms because it uses Intel’s specific interrupt handling routines. When Linux was ported to other hardware platforms such as the DEC Alpha and Sun SPARC, much of the platform-dependent code was moved into platform-specific modules that support a common interface.

The Linux user base is large. In 1994, Ed Chi estimated Linux has approximately 40,000 users ([Chi 1994]). The Linux Documentation Project [LDP] is working on developing useful and reliable documentation for the Linux kernel; both for use by Linux users and by Linux developers. To our knowledge, LDP does not maintain an up-to-date concrete

architecture using reverse engineering practices. There are many books and documents about the Linux kernel [CS746G Bibliography]. However, no available documentation adequately describes both the conceptual and concrete architecture of Linux. Publications (such as [Rusling 1997]) talk about how the Linux kernel works. However, these books do not thoroughly explain the subsystems and the interdependencies of the subsystems.

1.3 Background on Software Architecture

The study of software architecture has gained recent popularity in both the industrial and academic community. Software architecture involves the study of large software systems. Recent studies have shown that software architecture is important because it enhances communication among stakeholders of the system. Software architecture can be used to support earlier design decisions; also, it can be used as a transferable abstraction of a system ([Bass 1998]).

Software architecture is related to the study of software maintenance. Maintaining existing, or legacy systems is often problematic. The status of these existing systems can be described in a spectrum ranging from well designed and well documented to poorly designed and undocumented. In many cases, some or all of the original architects and developers may no longer be involved with the existing system. This lack of live architecture knowledge tremendously complicates the software maintenance task. In order to change, extend, modify, or remove functionality of an existing system, the implementation of the system needs to be understood. This problem reinforces the need to develop techniques to extract architectural and design information from existing system. The process of extracting high-level models from the source code is often referred to as reverse engineering.

The discipline of reverse engineering has two main areas [Bass 1998]:

1. Technical approaches: These extraction approaches derive information about a system based on existing artifacts. In particular, source code, comments, user documentation, executable modules, and system descriptions are often extracted.
2. Human knowledge and inference: These approaches focus on how humans understand software. Typically, human investigators use the following strategies:

    - Top-down strategies: start at the highest level of abstraction and recursively fill in understanding of the sub-parts.
    - Bottom-up strategies: understand the lowest level components and how these work together to accomplish the system’s goals
    - Model-based strategies: understand the mental model of how the system works and tries to deepen the understanding of selected areas
    - Opportunistic strategies: use some combination of the above approaches.

In this report, we used both the technical and human knowledge approaches to describe the concrete architecture of the Linux kernel. The opportunistic strategy follows the same strategy as the hybrid approach of [Tzerpos 1996]. Instead of using "live" information from the developers of the Linux kernel, we used our domain-specific knowledge of modern operating systems (i.e., the conceptual architecture from assignment 1) to repeatedly refine the concrete architecture of the Linux kernel.

1.4 Methodology/Approach

In this report, we used the opportunistic strategy to develop the concrete architecture of the Linux kernel. We used a modified version of the hybrid approach of [Tzerpos 1996] to determine the structure of the Linux kernel. The steps, which are not necessarily in sequential order, are:

- Define Conceptual Architecture. Since we have no direct access to "live" information from the developers, we used our modern operating system domain knowledge to create the Conceptual Architecture of the Linux kernel. This step was done in Assignment 1 ([Bowman 1998], [Siddiqi 1998], and [Tanuan 1998]).
- Extract facts from source code. We used the Portable Bookshelf's C Fact Extractor (cfx) and Fact Base Generator (fbgen) (described in [Holt 1997]) to extract dependency facts from the source code.
- Cluster into subsystems. We used the Fact Manipulator (i.e., grok and grok scripts) to cluster facts into subsystems. This clustering was performed partially by the tool

(using file names and directories), and partially using our conceptual model of the Linux kernel.
- Review the generated software structure. We used the Landscape Layouter, Adjuster, Editor and Viewer ([Holt 1997]) to visualize the extracted design information. Based on these diagrams, we could visually see the dependencies between the subsystems. The landscape diagrams confirmed our understanding of the concrete architecture. In cases where the extracted architecture disagreed with the conceptual architecture, we inspected the source code and documentation manually.
- Refine clustering using Conceptual Architecture. We used the Conceptual Architecture of the Linux kernel to verify the clustering of the components and the dependencies of these components.
- Refine layout using Conceptual Architecture. In conjunction with the previous step, we drew the layout of the Linux kernel structure manually using Visio drawing tool.

There are many views of the Linux kernel, depending on the motive and viewpoint. In this report, we described the concrete architecture using a software structure ([Mancoridis Slides]). We used the software structure to do the following:

- specify the decomposition of the Linux kernel into five major subsystems
- describe the interfaces of the components (i.e., subsystems and modules)
- describe the dependencies between components.

We describe dependencies between resources, where a resources can be subsystems, modules, procedures, or variable. The dependency relation is quite broad; usually we do not distinguish between function calls, variable reference, and type usage.

The software structure says little about the run-time structure of the Linux kernel. However, we believe that the software structure together with the detailed specifications will give enough information for a potential Linux developer to modify or extend the Linux without reviewing all the source code. We are not mainly concerned with the process view [Kruchten 1995] of the Linux kernel because we treat the Linux kernel as one single execution process.

1.5 Reader Profile

We assume that the reader has a sufficient background in computer science and operating systems to thoroughly understand the discussion on the major components and interactions of the components of the Linux kernel in this report. We do not assume a detailed knowledge of the Linux operating system.

1.6 Organization of this report

This remainder of this report is organized as follows:

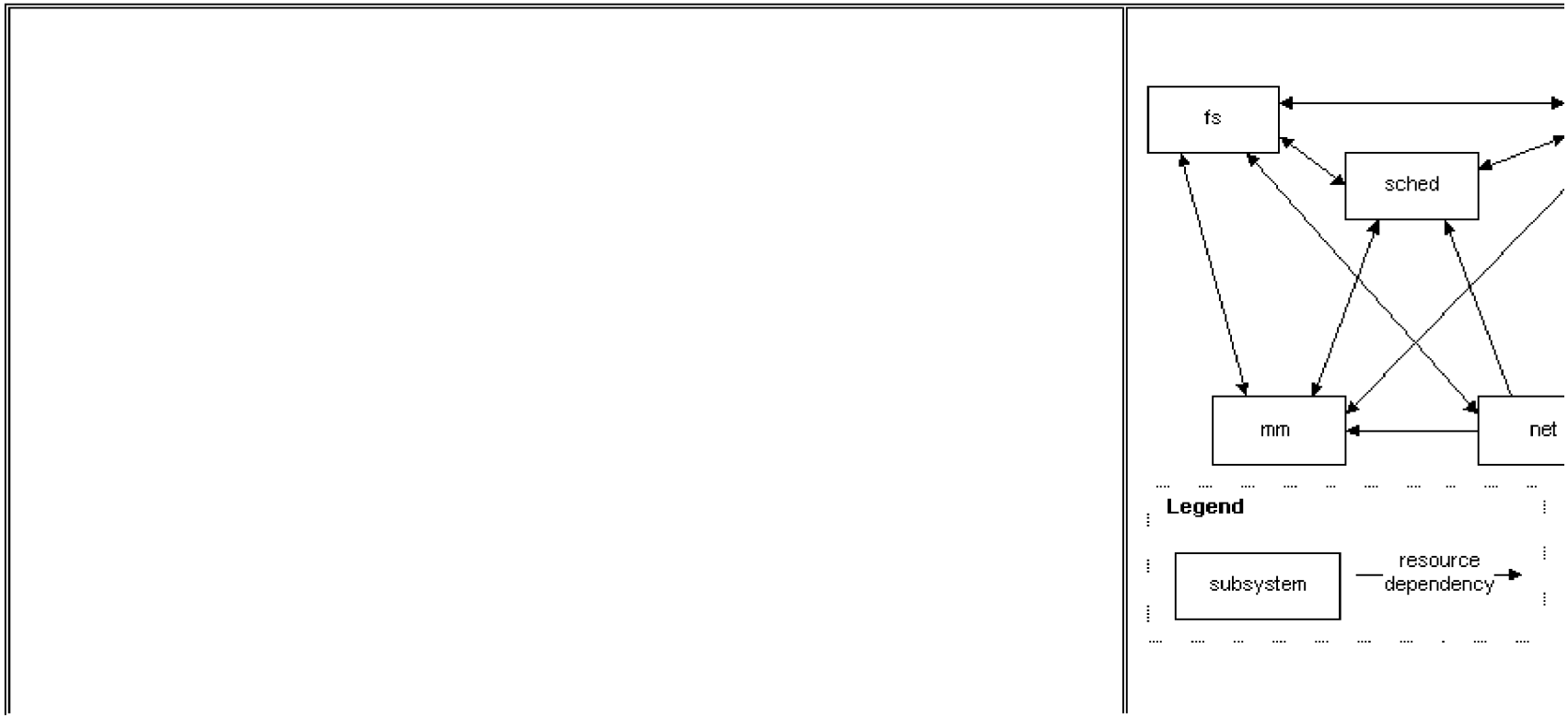
- Section 2 describes the overall system architecture. It describes the system architecture showing the five major subsystems and the interdependencies between them.
- Section 3 describes the subsystem architecture of the major subsystems: Process Scheduler, Memory Manager, Virtual File System, Inter-Process Communication, Network Interface. Each subsystem description will be supported with a diagram to show the subsystem in context with lines showing dependencies. It also includes an interpretation of the system abstraction and design information extracted from the Linux kernel source code.
- Section 4 describes the problems that we encountered in the report, and describe the findings and conclusion that we arrived at.

2 System Structure

The Linux Kernel is useless by itself; it participates as one layer in the overall system ([Bowman 1998]).

Within the kernel layer, Linux is composed of five major subsystems: the process scheduler (sched), the memory manager (mm), the virtual file system (vfs), the network interface (net), and the inter-process communication (ipc). The as-built architecture decomposition is similar to the conceptual architecture decomposition of [Siddiqi 1998], [Tanuan 1998], and [Bowman 1998]. This correspondence is not too surprising given that the conceptual architecture was derived from the as-built architecture. Our decomposition does not exactly follow the directory structure of the source code, as we believe that this doesn't perfectly match the subsystem grouping; however, our clustering is quite close to this directory structure.

One difference that became clear after visualizing the extracted design details is that the subsystem dependencies differ quite radically from the conceptual dependencies. The conceptual architecture suggested very few inter-system dependencies, as shown by Figure 1(a) (derived from [Bowman 1998]).



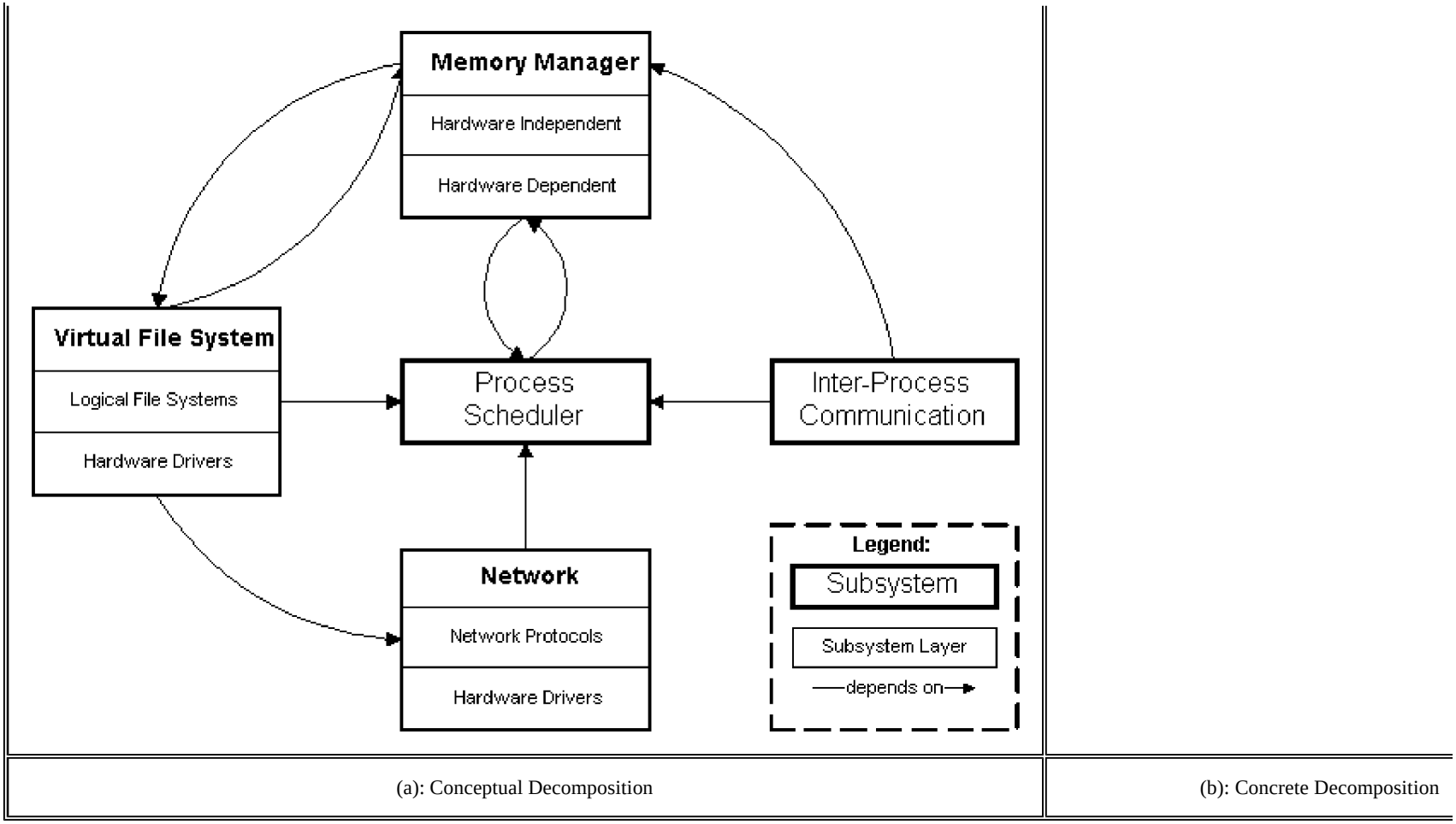


Figure 1: Conceptual versus Concrete System Decomposition

Although the conceptual architecture has few dependencies, the concrete architecture shows that the five major subsystems of the Linux kernel are highly interdependent. Figure 1(b) shows that the connection is only missing two edges from a complete graph ([PBS] gives the details of which modules interact across subsystems). This interdependency is a striking disagreement with the conceptual architecture. It seems that any reverse-engineering technique based on interconnection properties (such as the Rigi system described in [M|ller 1993]) would fail to extract any relevant structure from such a system. This validates the hybrid approach discussed by Tzerpos ([Tzerpos 1996]).

The differences at the system level are characteristic of differences at the subsystem level. The subsystem structure corresponds largely to the conceptual structure. However, we found many dependencies in the concrete architecture that weren't in the conceptual architecture (divergences in the sense of Murphy [Murphy 1995]). The reasons for these additional dependencies are discussed in the following section, where we examine the detailed design of each of the major subsystems.

---

3 Subsystem Structure

3.1 Process Scheduler

3.1.1 Goals

Process scheduling is the heart of the Linux operating system. The process scheduler has the following responsibilities:

- allow processes to create new copies of themselves
- determine which process will have access to the CPU and effect the transfer between running processes
- receive interrupts and route them to the appropriate kernel subsystem
- send signals to user processes
- manage the timer hardware
- clean up process resources when a processes finishes executing

The process scheduler also provides support for dynamically loaded modules; these modules represent kernel functionality that can be loaded after the kernel has started executing. This loadable module functionality is used by the virtual file system and the network interface.

3.1.2 External Interface

The process scheduler provides two interfaces: first, it provides a limited system call interface that user processes may call; secondly, it provides a rich interface to the rest of the kernel system.

Processes can only create other processes by copying the existing process. At boot time, the Linux system has only one running process: init. This process then spawns others, which can also spawn off copies of themselves, through the fork() system call. The fork() call generates a new child process that is a copy of its parent. Upon termination, a user process (implicitly or explicitly) calls the _exit() system call.

Several routines are provided to handle loadable modules. A create_module() system call will allocate enough memory to load a module. The call will initialize the module structure, described below, with the name, size, starting address, and initial status for the allocated module. The init_module() system call loads the module from disk and

activates it. Finally, delete_module() unloads a running module.

Timer management can be done through the setitimer() and getitimer() routines. The former sets a timer while the latter gets a timer's value.

Among the most important signal functions is signal(). This routine allows a user process to associate a function handler with a particular signal.

3.1.3 Subsystem Description

The process scheduler subsystem is primarily responsible for the loading, execution, and proper termination of user processes. The scheduling algorithm is called at two different points during the execution of a user process. First, there are system calls that call the scheduler directly, such as sleep(). Second, after every system call, and after every slow system interrupt (described in a moment), the schedule algorithm is called.

Signals can be considered an IPC mechanism, thus are discussed in the inter-process communication section.

Interrupts allow hardware to communicate with the operating system. Linux distinguishes between slow and fast interrupts. A slow interrupt is a typical interrupt. Other interrupts are legal while they are being processed, and once processing has completed on a slow interrupt, Linux conducts business as usual, such as calling the scheduling algorithm. A timer interrupt is exemplary of a slow interrupt. A fast interrupt is one that is used for much less complex tasks, such as processing keyboard input. Other interrupts are disabled as they are being processed, unless explicitly enabled by the fast interrupt handler.

The Linux OS uses a timer interrupt to fire off once every 10ms. Thus, according to our scheduler description given above, task rescheduling should occur at lease once every 10ms.

3.1.4 Data Structures

The structure task_struct represents a Linux task. There is a field that represents the process state; this may have the following values:

- running
- returning from system call
- processing an interrupt routine
- processing a system call
- ready
- waiting

In addition, there is a field that indicates the processes priority, and field which holds the number of clock ticks (10ms intervals) which the process can continue executing without forced rescheduling. There is also a field that holds the error number of the last faulting system call.

In order to keep track of all executing processes, a doubly linked list is maintained, (through two fields that point to task_struct). Since every process is related to some other process, there are fields which describe a processes: original parent, parent, youngest child, younger sibling, and finally older sibling.

There is a nested structure, mm_struct, which contains a process's memory management information, (such as start and end address of the code segment).

Process ID information is also kept within the task_struct. The process and group id are stored. An array of group id's is provided so that a process can be associated with more than one group.

File specific process data is located in a fs_struct substructure. This will hold a pointer to the inode corresponding to a processors root directory, and it's current working directory.

All files opened by a process will be kept track of through a files_struct substructure of the task_struct.

Finally, there are fields that hold timing information; for example, the amount of time the process has spent in user mode.

All executing processes have an entry in the process table. The process table is implemented as an array of pointers to task structures. The first entry in the process table is the special init process, which is the first process executed by the Linux system.

Finally, a module structure is implemented to represent the loaded modules. This structure contains fields that are used to implement a list of module structure: a field which points to the modules symbol table, and another field that holds the name of the module. The module size (in pages), and a pointer to the starting memory for the module are also fields within the module structure.

3.1.5 Subsystem Structure

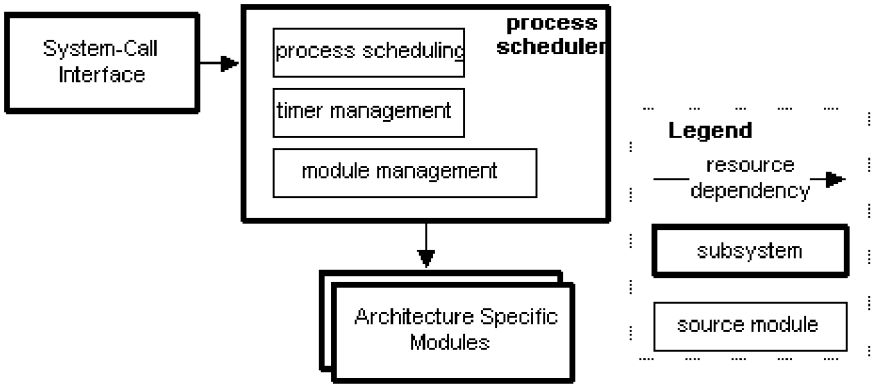


Figure 2: Process Scheduler Structure

Figure 2 shows the Process Scheduler subsystem. It is used to represent, collectively, process scheduling and management (i.e. loading and unloading), as well as timer management and module management functionality.

3.1.6 Subsystem Dependencies

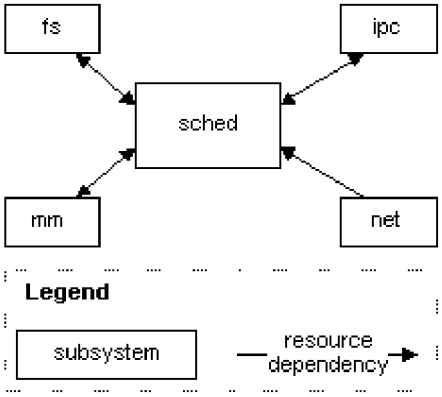


Figure 3: Process Scheduler Dependencies

Figure 3 shows how the process scheduler depends on other kernel subsystems. The process scheduler requires the memory manager to set up the memory mapping when a process is scheduled. Further, the process scheduler depends on the IPC subsystem for the semaphore queues that are used in bottom-half-handling (discussed in Section 3.3). Finally, the process scheduler depends on the file system to load loadable modules from the persistent device. All subsystems depend on the process scheduler, since they need to suspend user processes while hardware operations complete. For more details about the specific dependencies between subsystem modules, please see [PBS].

3.2 Memory Manager

3.2.1 Goals

As discussed in [Rusling 1997] pp 13-30, the memory manager provides the following capabilities to its clients:

- Large address space - user programs can reference more memory than physically exists
- Protection - the memory for a process is private and cannot be read or modified by another process; also, the memory manager prevents processes from overwriting code and read-only-data.
- Memory Mapping - clients can map a file into an area of virtual memory and access the file as memory
- Fair Access to Physical Memory - the memory manager ensures that processes all have fair access to the machine's memory resources, thus ensuring reasonable system performance
- Shared Memory - the memory manager allows processes to share some portion of their memory. For example, executable code is usually shared amongst processes.

3.2.2 External Interface

The memory manager provides two interfaces to its functionality: a system-call interface that is used by user processes, and an interface that is used by other kernel subsystems to accomplish their tasks.

- System Call Interface
  - malloc() / free() - allocate or free a region of memory for the process's use
  - mmap() / munmap() / msync() / mremap() - map files into virtual memory regions
  - mprotect - change the protection on a region of virtual memory
  - mlock() / mlockall() / munlock() / munlockall() - super-user routines to prevent memory being swapped
  - swapon() / swapoff() - super-user routines to add and remove swap files for the system
- Intra-Kernel Interface
  - kmalloc() / kfree() - allocate and free memory for use by the kernel’s data structures
  - verify_area() - verify that a region of user memory is mapped with required permissions
  - get_free_page() / free_page() - allocate and free physical memory pages

In addition to the above interfaces, the memory manager makes all of its data structures and most of its routines available within the kernel. Many kernel modules interface with the memory manager through access to the data structures and implementation details of the subsystem.

3.2.3 Subsystem Description

Since Linux supports several hardware platforms, there is a platform-specific part of the memory manager that abstracts the details of all hardware platforms into one common interface. All access to the hardware memory manager is through this abstract interface.

The memory manager uses the hardware memory manager to map virtual addresses (used by user processes) to physical memory addresses. When a user process accesses a memory location, the hardware memory manager translates this virtual memory address to a physical address, then uses the physical address to perform the access. Because of this mapping, user processes are not aware of what physical address is associated with a particular virtual memory address. This allows the memory manager subsystem to move the process's memory around in physical memory. In addition, this mapping permits two user processes to share physical memory if regions of their virtual memory address space map to the same physical address space.

In addition, the memory manager swaps process memory out to a paging file when it is not in use. This allows the system to execute processes that use more physical memory than is available on the system. The memory manager contains a daemon (kswapd). Linux uses the term daemon to refer to kernel threads; a daemon is scheduled by the process scheduler in the same way that user processes are, but daemons can directly access kernel data structures. Thus, the concept of a daemon is closer to a thread than a process.

The kswapd daemon periodically checks to see if there are any physical memory pages that haven't been referenced recently. These pages are evicted from physical memory; if necessary, they are stored on disk. The memory manager subsystem takes special care to minimize the amount of disk activity that is required. The memory manager avoids writing pages to disk if they could be retrieved another way.

The hardware memory manager detects when a user process accesses a memory address that is not currently mapped to a physical memory location. The hardware memory manager notifies the Linux kernel of this page fault, and it is up to the memory manager subsystem to resolve the fault. There are two possibilities: either the page is currently swapped out to disk, and must be swapped back in, or else the user process is making an invalid reference to a memory address outside of its mapped memory. The hardware memory manager also detects invalid references to memory addresses, such as writing to executable code or executing data. These references also result in page faults that are reported to the memory manager subsystem. If the memory manager detects an invalid memory access, it notifies the user process with a signal; if the process doesn't handle this signal, it is terminated.

3.2.4 Data Structures

The following data structures are architecturally relevant:

- vm_area - the memory manager stores a data structure with each process that records what regions of virtual memory are mapped to which physical pages. This data structure also stores a set of function pointers that allow it to perform actions on a particular region of the process's virtual memory. For example, the executable code region of the process does not need to be swapped to the system paging file since it can use the executable file as a backing store. When regions of a process's virtual memory are mapped (for example when an executable is loaded), a vm_area_struct is set up for each contiguous region in the virtual address space. Since speed is critical when looking up a vm_area_struct for a page fault, the structures are stored in an AVL tree.
- mem_map - the memory manager maintains a data structure for each page of physical memory on a system. This data structure contains flags that indicate the status of the page (for example, whether it is currently in use). All page data structures are available in a vector (mem_map), which is initialized at kernel boot time. As page status changes, the attributes in this data structure are updated.
- free_area - the free_area vector is used to store unallocated physical memory pages; pages are removed from the free_area when allocated, and returned when freed. The Buddy system [Knuth 1973; Knowlton 1965; Tanenbaum 1992] is used when allocating pages from the free_area.

3.2.5 Subsystem Structure

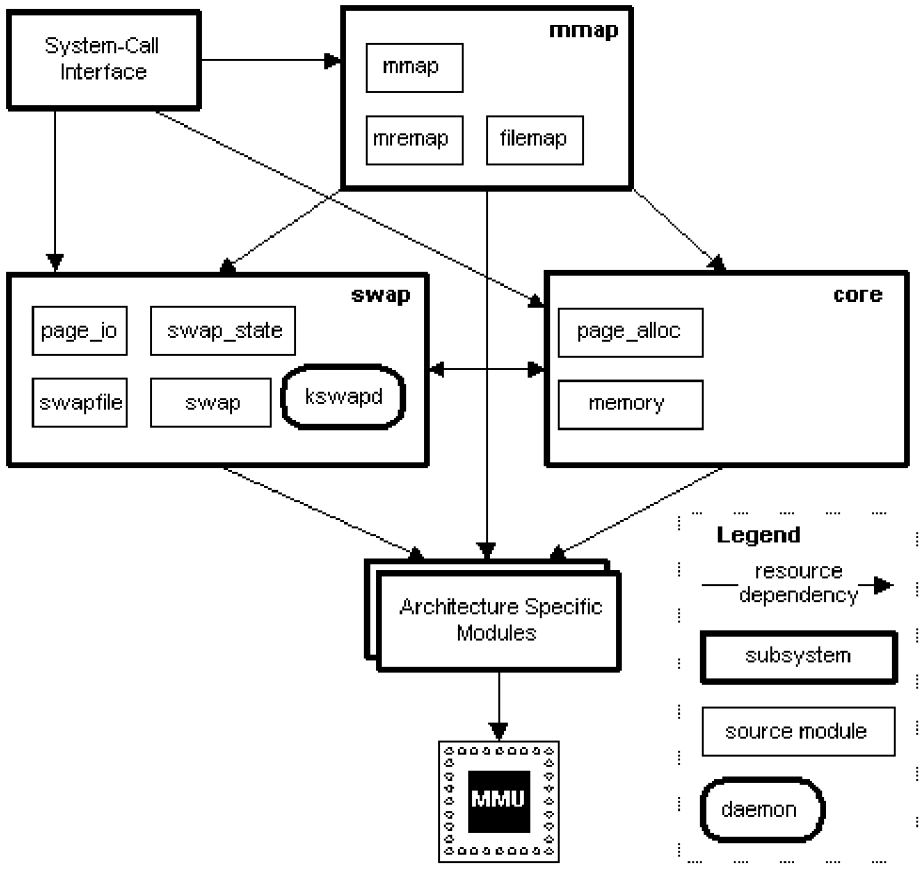


Figure 4: Memory Manager Structure

The memory manager subsystem is composed of several source code modules; these can be decomposed by areas of responsibility into the following groups (shown in Figure 4):

- System Call Interface - this group of modules is responsible for presenting the services of the memory manager to user processes through a well-defined interface (discussed earlier).
- Memory-Mapped Files (mmap) - this group of modules is responsible for supported memory-mapped file I/O.
- Swapfile Access (swap) - this group of modules controls memory swapping. These modules initiate page-in and page-out operations.
- Core Memory Manager (core) - these modules are responsible for the core memory manager functionality that is used by other kernel subsystems.
- Architecture Specific Modules- these modules provide a common interface to all supported hardware platforms. These modules execute commands to change the hardware MMU's virtual memory map, and provide a common means of notifying the rest of the memory-manager subsystem when a page

fault occurs.

One interesting aspect of the memory manager structure is the use of kswapd, a daemon that determines which pages of memory should be swapped out. kswapd executes as a kernel thread, and periodically scans the physical pages in use to see if any are candidates to be swapped. This daemon executes concurrently with the rest of the memory manager subsystem.

3.2.6 Subsystem Dependencies

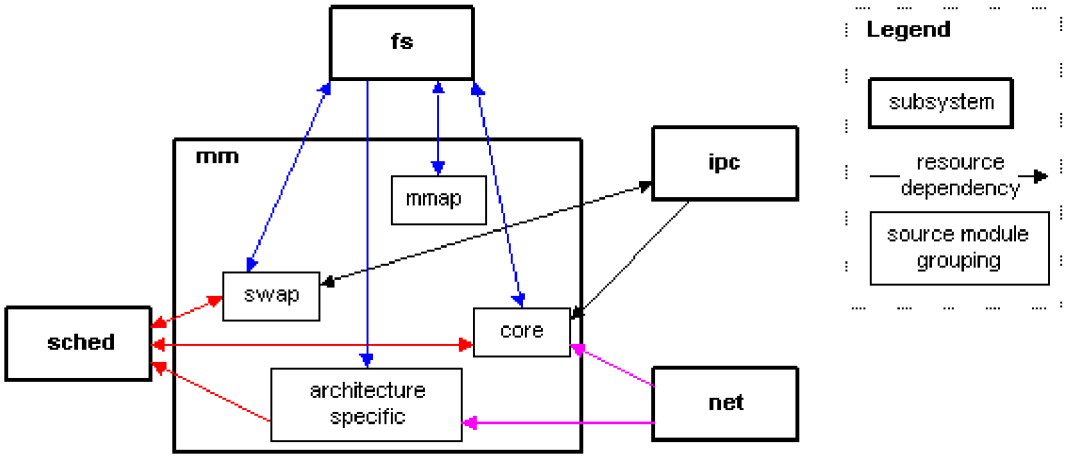


Figure 5: Memory Manager Dependencies

The memory manager is used directly (via data structures and implementation functions) by each of sched, fs, ipc, and net. This dependency is difficult to describe succinctly; please refer to [PBS] for a detailed view of the subsystem dependencies. Figure 5 shows the high-level dependencies between the memory manager and other subsystems. Internal dependencies are elided for clarity.

3.3 Virtual File System

3.3.1 Goals

Linux is designed to support many different physical devices. Even for one specific type of device, such as hard drives, there are many interface differences between different hardware vendors. In addition to the physical devices that Linux supports, Linux supports a number of logical file systems. By supporting many logical file systems, Linux can inter-operate easily with other operating systems. The Linux file system supports the following goals:

- Multiple hardware devices - provide access to many different hardware devices
- Multiple logical file systems - support many different logical file systems
- Multiple executable formats - support several different executable file formats (like a.out, ELF, java)
- Homogeneity - present a common interface to all of the logical file systems and all hardware devices
- Performance - provide high-speed access to files
- Safety - do not lose or corrupt data
- Security - restrict user access to access files; restrict user total file size with quotas

3.3.2 External Interface

The file system provides two levels of interface: a system-call interface that is available to user processes, and an internal interface that is used by other kernel subsystems. The system-call interface deals with files and directories. Operations on files include the usual open/close/read/write/seek/tell that are provided by POSIX compliant systems; operations on directories include readdir/creat/unlink/chmod/stat as usual for POSIX systems.

The interface that the file subsystem supports for other kernel subsystems is much richer. The file subsystem exposes data structures and implementation function for direct manipulation by other kernel subsystems. In particular, two interfaces are exposed to the rest of the kernel -- inodes and files. Other implementation details of the file subsystem are also used by other kernel subsystems, but this use is less common.

Inode Interface:

- create(): create a file in a directory
- lookup(): find a file by name within a directory
- link() / symlink() / unlink() / readlink() / follow_link(): manage file system links
- mkdir() / rmdir(): create or remove sub-directories
- mknod(): create a directory, special file, or regular file
- readpage() / writepage(): read or write a page of physical memory to a backing store
- truncate(): set the length of a file to zero
- permission(): check to see if a user process has permission to execute an operation
- smap(): map a logical file block to a physical device sector
- bmap(): map a logical file block to a physical device block
- rename(): rename a file or directory

In addition to the methods you can call with an inode, the namei() function is provided to allow other kernel subsystems to find the inode associated with a file or directory.

File Interface:

- open() / release(): open or close the file
- read() / write(): read or write to the file
- select(): wait until the file is in a particular state (readable or writeable)

- lseek(): if supported, move to a particular offset in the file
- mmap(): map a region of the file into the virtual memory of a user process
- fsync() / fasync(): synchronize any memory buffers with the physical device
- readdir: read the files that are pointed to by a directory file
- ioctl: set file attributes
- check_media_change: check to see if a removable media has been removed (such as a floppy)
- revalidate: verify that all cached information is valid

3.3.3 Subsystem Description

The file subsystem needs to support many different logical file systems and many different hardware devices. It does this by having two conceptual layers that are easily extended. The device driver layer represents all physical devices with a common interface. The virtual file system layer (VFS) represents all logical file systems with a common interface. The conceptual architecture of the Linux kernel ([Bowman 1998], [Siddiqi 1998]) shows how this decomposition is conceptually arranged.

Device Drivers

The device driver layer is responsible for presenting a common interface to all physical devices. The Linux kernel has three types of device driver: character, block, and network. The two types relevant to the file subsystem are character and block devices. Character devices must be accessed sequentially; typical examples are tape drives, modems, and mice. Block devices can be accessed in any order, but can only be read and written to in multiples of the block size.

All device drivers support the file operations interface described earlier. Therefore, each device can be accessed as though it was a file in the file system (this file is referred to as a device special file). Since most of the kernel deals with devices through this file interface, it is relatively easy to add a new device driver by implementing the hardware-specific code to support this abstract file interface. It is important that it is easy to write new device drivers since there is a large number of different hardware devices.

The Linux kernel uses a buffer cache to improve performance when accessing block devices. All access to block devices occurs through a buffer cache subsystem. The buffer cache greatly increases system performance by minimizing reads and writes to hardware devices. Each hardware device has a request queue; when the buffer cache cannot fulfill a request from in-memory buffers, it adds a request to the device's request queue and sleeps until this request has been satisfied. The buffer cache uses a separate kernel thread, kflushd, to write buffer pages out to the devices and remove them from the cache.

When a device driver needs to satisfy a request, it begins by initiating the operation with the hardware device manipulating the device's control and status registers (CSR's). There are three general mechanisms for moving data from the main computer to the peripheral device: polling, direct memory access (DMA), and interrupts. In the polling case, the device driver periodically checks the CSR's of the peripheral to see if the current request has been completed. If so, the driver initiates the next request and continues. Polling is appropriate for low-speed hardware devices such as floppy drives and modems. Another mechanism for transfer is DMA. In this case, the device driver initiates a DMA transfer between the computer's main memory and the peripheral. This transfer operates concurrently with the main CPU, and allows the CPU to process other tasks while the operation is continuing. When the DMA operation is complete, the CPU receives an interrupt. Interrupt handling is very common in the Linux kernel, and it is more complicated than the other two approaches.

When a hardware device wants to report a change in condition (mouse button pushed, key pressed) or to report the completion of an operation, it sends an interrupt to the CPU. If interrupts are enabled, the CPU stops executing the current instruction and begins executing the Linux kernel's interrupt handling code. The kernel finds the appropriate interrupt handler to invoke (each device driver registers handlers for the interrupts the device generates). While an interrupt is being handled, the CPU executes in a special context; other interrupts may be delayed until the interrupt is handled. Because of this restriction, interrupt handlers need to be quite efficient so that other interrupts won't be lost. Sometimes an interrupt handler cannot complete all required work within the time constraints; in this case, the interrupt handler schedules the remainder of the work in a bottom-half handler. A bottom-half handler is code that will be executed by the scheduler the next time that a system call has been completed. By deferring non-critical work to

a bottom half handler, device drivers can reduce interrupt latency and promote concurrency.

In summary, device drivers hide the details of manipulating a peripheral's CSR's and the data transfer mechanism for each device. The buffer cache helps improve system performance by attempting to satisfy file system requests from in-memory buffers for block devices.

Logical File Systems

Although it is possible to access physical devices through their device special file, it is more common to access block devices through a logical file system. A logical file system can be mounted at a mount point in the virtual file system. This means that the associated block device contains files and structure information that allow the logical file system to access the device. At any one time, a physical device can only support one logical file system; however, the device can be reformatted to support a different logical file system. At the time of writing, Linux supports fifteen logical file systems; this promotes interoperability with other operating systems.

When a file system is mounted as a subdirectory, all of the directories and files available on the device are made visible as subdirectories of the mount point. Users of the virtual file system do not need to be aware what logical file system is implementing which parts of the directory tree, nor which physical devices are containing those logical file systems. This abstraction provides a great deal of flexibility in both choice of physical devices and logical file systems, and this flexibility is one of the essential factors in the success of the Linux operating system.

To support the virtual file system, Linux uses the concept of inodes. Linux uses an inode to represent a file on a block device. The inode is virtual in the sense that it contains operations that are implemented differently depending both on the logical system and physical system where the file resides. The inode interface makes all files appear the same to other Linux subsystems. The inode is used as a storage location for all of the information related to an open file on disk. The inode stores associated buffers, the total length of the file in blocks, and the mapping between file offsets and device blocks.

Modules

Most of the functionality of the virtual file system is available in the form of dynamically loaded modules (described in section 3.1). This dynamic configuration allows Linux users to compile a kernel that is as small as possible, while still allowing it to load required device driver and file system modules if necessary during a single session. For example, a Linux system might optionally have a printer attached to its parallel port. If the printer driver were always linked in to the kernel, then memory would be wasted when the printer isn't available. By making the printer driver be a loadable module, Linux allows the user to load the driver if the hardware is available.

3.3.4 Data Structures

The following data structures are architecturally relevant to the file subsystem:

- super_block: each logical file system has an associated superblock that is used to represent it to the rest of the Linux kernel. This superblock contains information about the entire mounted file system -- what blocks are in use, what the block size is, etc. The superblock is similar to inodes in that they behave as a virtual interface to the logical file system.
- inode: an inode is an in-memory data structure that represents all of the information that the kernel needs to know about a file on disk. A single inode might be used by several processes that all have the file open. The inode stores all of the information that the kernel needs to associate with a single file. Accounting, buffering, and memory mapping information are all stored in the inode. Some logical file systems also have an inode structure on disk that maintains this information persistently, but this is distinct from the inode data structure used within the rest of the kernel.
- file: the file structure represents a file that is opened by a particular process. All open files are stored in a doubly-linked list (pointed to by first_file); the file descriptor that is used in POSIX style routines (open, read, write) is the index of a particular open file in this linked list.

3.3.5 Subsystem Structure

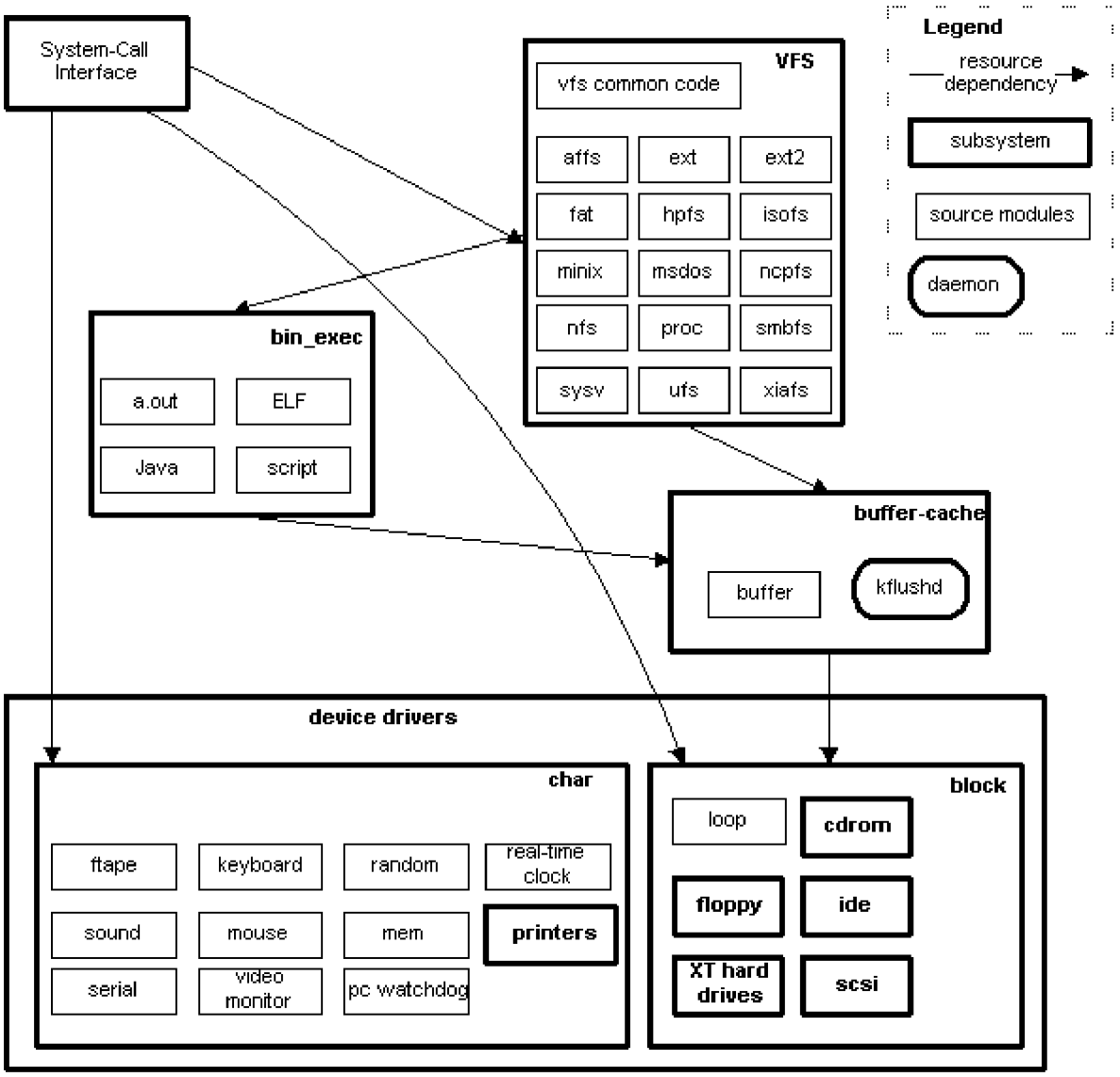


Figure 6: File Subsystem Structure

3.3.6 Subsystem Dependencies

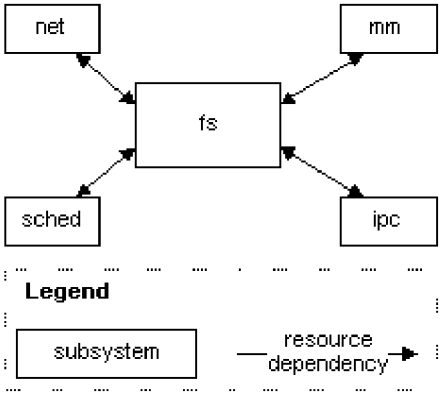


Figure 7: File Subsystem Dependencies

Figure 7 shows how the file system is dependent on other kernel subsystems. Again, the file system depends on all other kernel subsystems, and all other kernel subsystems depend on the file subsystem. In particular, the network subsystem depends on the file system because network sockets are presented to user processes as file descriptors. The memory manager depends on the file system to support swapping. The IPC subsystem depends on the file system for implementing pipes and FIFO's. The process scheduler depends on the file system to load loadable modules.

The file system uses the network interface to support NFS; it uses the memory manager to implement the buffer cache and for a ramdisk device; it uses the IPC subsystem to help support loadable modules, and it uses the process scheduler to put user processes to sleep while hardware requests are completed. For more details, see [PBS].

3.4 Inter-Process Communication

3.4.1 Goals

The Linux IPC mechanism is provided so that concurrently executing processes have a means to share resources, synchronize and exchange data with one another. Linux implements all forms of IPC between processes executing on the same system through shared resources, kernel data structures, and wait queues.

Linux provides the following forms of IPC:

- Signals – perhaps the oldest form of Unix IPC, signals are asynchronous messages sent to a process.
- Wait queues – provides a mechanism to put processes to sleep while they are waiting for an operation to complete. This mechanism is used by the process scheduler to implement bottom-half handling as described in section 3.3.3.
- File locks – provides a mechanism to allow processes to declare either regions of a file, or the entire file itself, as read-only to all processes except the one which holds the file lock.
- Pipes and Named Pipes – allows connection-oriented, bi-directional data transfer between two processes either by explicitly setting up the pipe connection, or communicating through a named pipe residing in the file-system.
- System V IPC
    - Semaphores – an implementation of a classical semaphore model. The model also allows for the creation of arrays of semaphores.
    - Message queues – a connectionless data-transfer model. A message is a sequence of bytes, with an associated type. Messages are written to message queues, and messages can be obtained by reading from the message queue, possibly restricting which messages are read in by type.
    - Shared memory – a mechanism by which several processes have access to the same region of physical memory.
- Unix Domain sockets – another connection-oriented data-transfer mechanism that provides the same communication model as the INET sockets, discussed in the next section.

3.4.2 External Interface

A signal is a notification sent to a process by the kernel or another process. Signals are sent with the send_sig() function. The signal number is provided as a parameter, as well as the destination process. Processes may register to handle signals by using the signal() function.

File locks are supported directly by the Linux file system. To lock an entire file, the open() system call can be used, or the sys_fcntl() system-call can be used. Locking areas within a file is done through the sys_fcntl() system call.

Pipes are created by using the pipe() system call. The file-systems read() and write() calls are then used to transfer data on the pipe. Named pipes are opened using the open() system-call.

The System V IPC mechanisms have a common interface, which is the ipc() system call. The various IPC operations are specified using parameters to the system call.

The Unix domain socket functionality is also encapsulated by a single system call, socketcall().

Each of the system-calls mentioned above are well documented, and the reader is encouraged to consult the corresponding man-page.

The IPC subsystem exposes wait calls to other kernel subsystems. Since wait queues are not used by user processes, they do not have a system-call interface. Wait queues are used in implementing semaphores, pipes, and bottom-half handlers (see section 3.3.3). The procedure add_wait_queue() inserts a task into a wait queue. The procedure remove_wait_queue() removes a task from the wait queue.

3.4.3 Subsystem Description

The following is a brief description of the low-level functioning of each IPC mechanism identified in section 3.4.1.

Signals are used to notify a process of an event. A signal has the effect of altering the state of the recipient process, depending on the semantics of the particular signal. The kernel can send signals to any executing process. A user process may only send a signal to a process or process group if it possesses the associated PID or GID. Signals are not handled immediately for dormant processes. Rather, before the scheduler sets a process running in user mode again, it checks if a signal was sent to the process. If so, then the scheduler calls the do_signal() function, which handles the signal appropriately.

Wait queues are simply linked lists of pointers to task structures that correspond to processes that are waiting for a kernel event, such as the conclusion of a DMA transfer. A process can enter itself on the wait queue by either calling the sleep_on() or interruptable_sleep_on() functions. Similarly, the functions wake_up() and wake_up_interruptable() remove the process from the wait queue. Interrupt routines also use wait-queues to avoid race conditions.

Linux allows user process to prevent other processes from access a file. This exclusion can be based on a whole file, or a region of a file. File-locks are used to implement this exclusion. The file-system implementation contains appropriate data fields in it’s data structures to allow the kernel to determine if a lock has been placed on a file, or a region inside a file. In the former case, a lock attempt on a locked file will fail. In the latter case, an attempt to lock a region already locked will fail. In either case, the requesting process is not permitted to access the file since the lock has not been granted by the kernel.

Pipes and named pipes have a similar implementation, as their functionality is almost the same. The creation process is different. However, in either case a file descriptor is returned which refers to the pipe. Upon creation, one page of memory is associated with the opened pipe. This memory is treated like a circular buffer, to which write operations are done atomically. When the buffer is full, the writing processes block. If a read request is made for more data than what is available, the reading processes block. Thus, each pipe has a wait queue associated with it. Processes are added and removed from the queue during the read and writes.

Semaphores are implemented using wait queues, and follow the classical semaphore model. Each semaphore has an associated value. Two operations, up() and down() are implemented on the semaphore. When the value of the semaphore is zero, the process performing the decrement on the semaphore is blocked on the wait queue. Semaphore arrays are simply a contiguous set of semaphores. Each process also maintains a list of semaphore operations it has performed, so that if the process exits prematurely, these operations can be undone.

The message queue is a linear linked-list, to which processes read or write a sequence of bytes. Messages are received in the same order that they are written. Two wait queues are associated with the message queues, one for processes that are writing to a full message queue, and another for serializing the message writes. The actual size of the message is set when the message queue is created.

Shared memory is the fastest form of IPC. This mechanism allows processes to share a region of their memory. Creation of shared memory areas is handled by the memory management system. Shared pages are attached to the user processes virtual memory space by the system call sys_shmat(). A shared page can be removed from the user segment of a process by calling the sys_shmdt() call.

The Unix domain sockets are implemented in a similar fashion to pipes, in the sense that both are based on a circular buffer based on a page of memory. However, sockets provide a separate buffer for each communication direction.

3.4.4 Data Structures

In this section, the important data structures needed to implement the above IPC mechanisms are described.

Signals are implemented through the signal field in the task_struct structure. Each signal is represented by a bit in this field. Thus, the number of signals a version of Linux can

support is limited to the number of bits in a word. The field blocked holds the signals that are being blocked by a process.

There is only one data structure associated with the wait queues, the wait_queue structure. These structures contain a pointer to the associated task_struct, and are linked into a list.

File locks have an associated file_lock structure. This structure contains a pointer to a task_struct for the owning process, the file descriptor of the locked file, a wait queue for processes which are waiting for the cancellation of the file lock, and which region of the file is locked. The file_lock structures are linked into a list for each open file.

Pipes, both nameless and named, are represented by a file system inode. This inode stores extra pipe-specific information in the pipe_inode_info structure. This structure contains a wait queue for processes which are blocking on a read or write, a pointer to the page of memory used as the circular buffer for the pipe, the amount of data in the pipe, and the number of processes which are currently reading and writing from/to the pipe.

All system V IPC objects are created in the kernel, and each have associated access permissions. These access permissions are held in the ipc_perm structure. Semaphores are represented with the sem structure, which holds the value of the semaphore and the pid of the process that performed the last operation on the semaphore. Semaphore arrays are represented by the semid_ds structure, which holds the access permissions, the time of the last semaphore operation, a pointer to the first semaphore in the array, and queues on which processes block when performing semaphore operations. The structure sem_undo is used to create a list of semaphore operations performed by a process, so that they can all be undone when the process is killed.

Message queues are based on the msquid_ds structure, which holds management and control information. This structure stores the following fields:

- access permissions
- link fields to implement the message queue (i.e. pointers to msquid_ds)
- times for the last send, receipt and change
- queues on which processes block, as described in the previous section
- the current number of bytes in the queue
- the number of messages
- the size of the queue (in bytes)
- the process number of the last sender
- the process number of the last receiver.

A message itself is stored in the kernel with a msg structure. This structure holds a link field, to implement a link list of messages, the type of message, the address of the message data, and the length of the message.

The shared memory implementation is based on the shmid_ds structure, which, like the msquid_ds structure, holds management and control information. The structure contains access control permissions, last attach, detach and change times, pids of the creator and last process to call an operation for the shared segment, number of processes to which the shared memory region is attached to, the number of pages which make up the shared memory region, and a field for page table entries.

The Unix domain sockets are based on the socket data structure, described in the Network Interface section (3.5).

3.4.5 Subsystem Structure

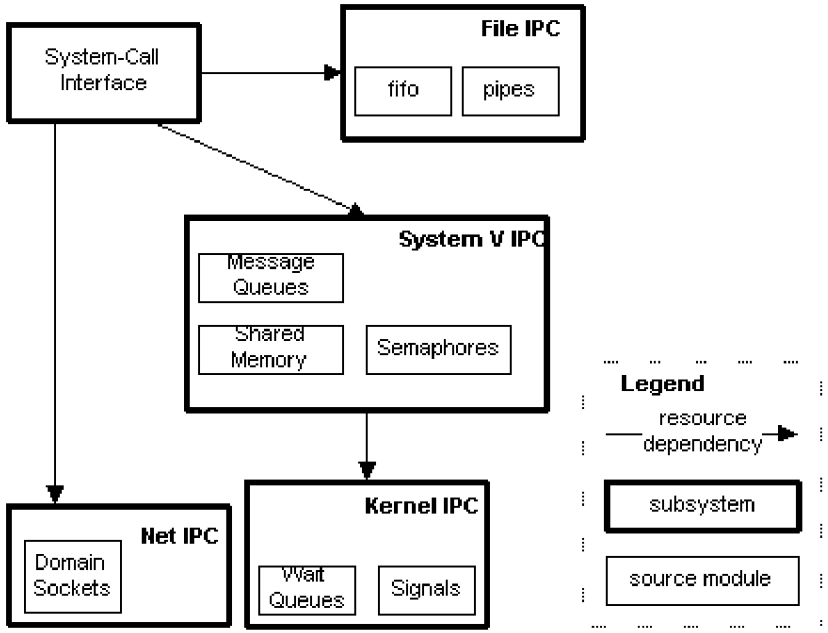


Figure 8: IPC Subsystem Structure

Figure 8 shows the IPC subsystem resource dependencies. Control flows from the system call layer down into each module. The System V IPC facilities are implemented in the ipc directory of the kernel source. The kernel IPC module refers to IPC facilities implemented within the kernel directory. Similar conventions hold for the File and Net IPC facilities.

The System V IPC module is dependant on the Kernel IPC mechanism. In particular, semaphores are implemented with wait queues. All other IPC facilities are implemented independently of each other.

3.4.6 Subsystem Dependencies

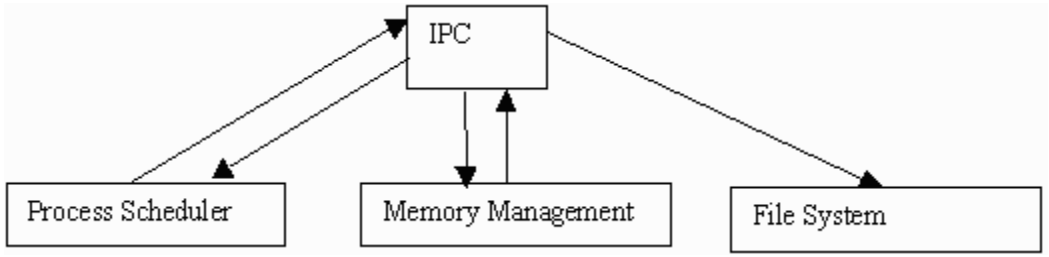


Figure 9: IPC Subsystem Dependencies

Figure 9 shows the resource dependencies between the IPC subsystem and other kernel subsystems.

The IPC subsystem depends on the file system for sockets. Sockets use file descriptors, and once they are opened, they are assigned to an inode. Memory management depends on IPC as the page swapping routine calls the IPC subsystem to perform swapping of shared memory. IPC depends on memory management primarily for the allocation of buffers and the implementation of shared memory.

Some IPC mechanisms use timers, which are implemented in the process scheduler subsystem. Process scheduling relies on signals. For these two reasons, the IPC and Process Scheduler modules depend on each other. For more details about the dependencies between the IPC subsystem modules and other kernel subsystems, see [PBS].

3.5 Network Interface

3.5.1 Goals

The Linux network system provides network connectivity between machines, and a socket communication model. Two types of socket implementations are provided: BSD sockets and INET sockets. BSD sockets are implemented using INET sockets.

The Linux network system provides two transport protocols with differing communication models and quality of service. These are the unreliable, message-based UDP protocol and the reliable, streamed TCP protocol. These are implemented on top of the IP networking protocol. The INET sockets are implemented on top of both transport protocols and the IP protocol.

Finally, the IP protocol sits on top of the underlying device drivers. Device drivers are provided for three different types of connections: serial line connections (SLIP), parallel line connections (PLIP), and ethernet connections. An address resolution protocol mediates between the IP and ethernet drivers. The address resolver’s role is to translate

between the logical IP addresses and the physical ethernet addresses.

3.5.2 External Interface

The network services are used by other subsystems and the user through the socket interface. Sockets are created and manipulated through the socketcall() system call. Data is sent and received using read() and write() calls on the socket file descriptor.

No other network mechanism/functionality is exported from the network sub-system.

3.5.3 Subsystem Description

The BSD socket model is presented to the user processes. The model is that of a connection-oriented, streamed and buffered communication service. The BSD socket is implemented on top of the INET socket model.

The BSD socket model handles tasks similar to that of the VFS, and administers a general data structure for socket connections. The purpose of the BSD socket model is to provide greater portability by abstracting communication details to a common interface. The BSD interface is widely used in modern operating systems such as Unix and Microsoft Windows. The INET socket model manages the actual communication end points for the IP-based protocols TCP and UDP.

Network I/O begins with a read or write to a socket. This invokes a read/write system call, which is handled by a component of the virtual file system, (the chain of read/write calls down the network subsystem layers are symmetric, thus from this point forward, only writes are considered). From there, it is determined that the BSD socket sock_write () is what implements the actual file system write call; thus, it is called. This routine handles administrative details, and control is then passed to inet_write() function. This in-turn calls a transport layer write call (such as tcp_write()).

The transport layer write routines are responsible for splitting the incoming data into transport packets. These routines pass control to the ip_build_header() routine, which builds an ip protocol header to be inserted into the packet to be sent, and then tcp_build_header() is called to build a tcp protocol header. Once this is done, the underlying device drivers are used to actually send the data.

The network system provides two different transport services, each with a different communication model and quality of service. UDP provides a connectionless, unreliable data transmission service. It is responsible for receiving packets from the IP layer, and finding the destination socket to which the packet data should be sent. If the destination socket is not present, an error is reported. Otherwise, if there is sufficient buffer memory, the packet data is entered into a list of packets received for a socket. Any sockets sleeping on a read operation are notified, and awoken.

The TCP transport protocol offers a much more complicated scheme. In addition to handling data transfer between sending and receiving processes, the TCP protocol also performs complicated connection management. TCP sends data up to the socket layer as a stream, rather than as a sequence of packets, and guarantees a reliable transport service.

The IP protocol provides a packet transfer service. Given a packet, and a destination of the packet, the IP communication layer is responsible for the routing of the packet to the correct host. For an outgoing data stream, the IP is responsible for the following:

- partitioning the stream into IP packets
- routing the IP packets to the destination address

- generating a header to be used by the hardware device drivers
- selecting the appropriate network device to send out on

For an incoming packet stream, the IP must do the following:

- check the header for validity
- compare the destination address with the local address and forwarding it along if the packet is not at it’s correct destination
- defragment the IP packet
- send the packets up to the TCP or UDP layer to be further processed.

The ARP (address resolution protocol) is responsible for converting between the IP and the real hardware address. The ARP supports a variety of hardware devices such as ethernet, FDDI, etc. This function is necessary as sockets deal with IP addresses, which cannot be used directly by the underlying hardware devices. Since a neutral addressing scheme is used, the same communication protocols can be implemented across a variety of hardware devices.

The network subsystem provides its own device drivers for serial connections, parallel connections, and ethernet connections. An abstract interface to the various hardware devices is provided to hide the differences between communication mediums from the upper layers of the network subsystem.

3.5.4 Data Structures

The BSD socket implementation is represented by the socket structure. It contains a field which identifies the type of socket (streamed or datagram), and the state of the socket (connected or unconnected). A field that holds flags which modify the operation of the socket, and a pointer to a structure that describes the operations that can be performed on the socket are also provided. A pointer to the associated INET socket implementation is provided, as well as a reference to an inode. Each BSD socket is associated with an inode.

The structure sk_buff is used to manage individual communication packets. The buffer points to the socket to which it belongs, contains the time it was last transferred, and link fields so that all packets associated with a given socket can be linked together (in a doubly linked list). The source and destination addresses, header information, and packet data are also contained within the buffer. This structure encapsulates all packets used by the networking system (i.e. tcp packet, udp packets, ip packets, etc.).

The sock structure refers to the INET socket-specific information. The members of this structure include counts of the read and write memory requested by the socket, sequence numbers required by the TCP protocol, flags which can be set to alter the behavior of the socket, buffer management fields, (for example, to maintain a list of all packets received for the given socket), and a wait queue for blocking reads and writes. A pointer to a structure that maintains a list of function pointers that handle protocol-specific routines, the proto structure, is also provided. The proto structure is rather large and complex, but essentially, it provides an abstract interface to the TCP and UDP protocols. The source and destination addresses, and more TCP-specific data fields are provided. TCP uses timers extensively to handle time-outs, etc. thus the sock structure contains data fields pertaining to timer operations, as well as function pointers which are used as callbacks for timer alarms.

Finally, the device structure is used to define a network device driver. This is the same structure used to represent file system device drivers.

3.5.5 Subsystem Structure

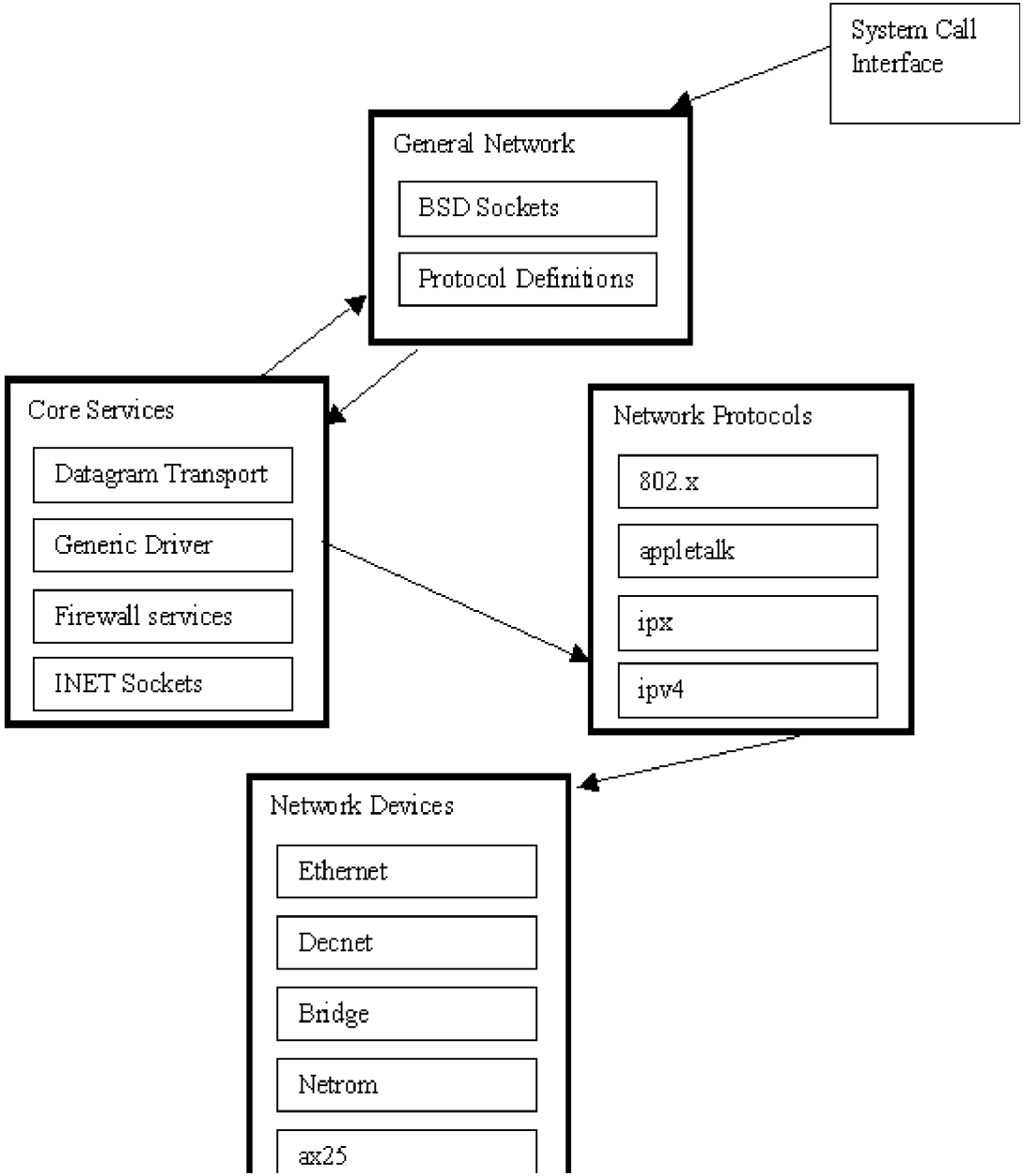

System Call Interface
General Network
BSD Sockets
Protocol Definitions
Core Services
Datagram Transport
Generic Driver
Firewall services
INET Sockets
Network Protocols
802.x
appletalk
ipx
ipv4
Network Devices
Ethernet
Decnet
Bridge
Netrom
ax25

Figure 10: Network Subsystem Structure

The General Network contains those modules that provide the highest level interface to user processes. This is essentially the BSD Socket interface, as well as a definition of the protocols supported by the network layer. Included here are the MAC protocols of 802.x, ip, ipx, and AppleTalk.

The Core services correspond to high-level implementation facilities, such as INET sockets, support for firewalls, a common network device driver interface, and facilities for datagram and TCP transport services.

The system call interface interacts with the BSD socket interface. The BSD socket layer provides a general abstraction for socket communication, and this abstraction is implemented using the INET sockets. This is the reason for the dependency between the General Network module and the Core services module.

The Protocol modules contain the code that takes user data and formats them as required by a particular protocol. The protocol layer finally sends the data to the proper hardware device, hence the dependency between the Network Protocols and Network Devices modules.

The Network devices module contains high-level device-type specific routines. The actual device drivers reside with the regular device drivers under the directory drivers/net.

3.5.6 Subsystem Dependencies

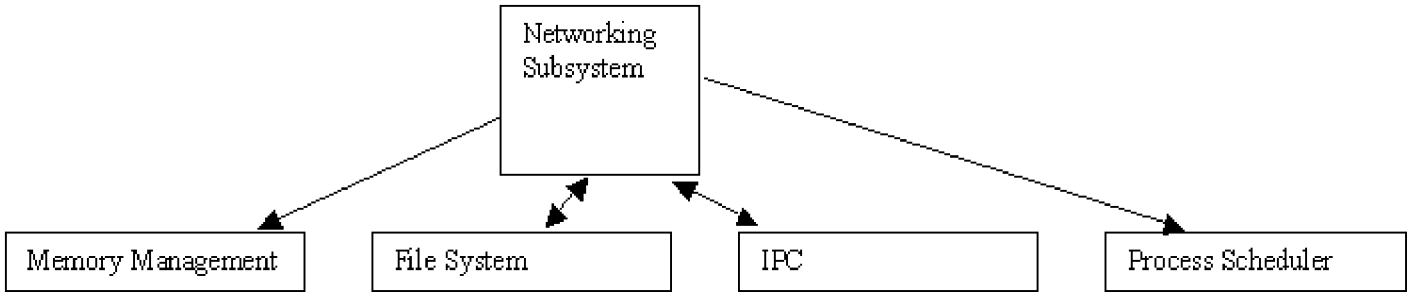


Figure 11: Network Subsystem Dependencies

Figure 11 shows the dependencies between the networking subsystem and other subsystems.

The network subsystem depends on the Memory Management system for buffers. The File System is used to provide an inode interface to sockets. The file system uses the network system to implement NFS. The network subsystem uses the kerneld daemon, and thus it depends on IPC. The network subsystem uses the Process Scheduler for timer functions and to suspend processes while network transmissions are executed. For more details about the specific dependencies between the network subsystem modules and other kernel subsystems, see [PBS].

4 Findings

In summary, we reviewed existing documentation, extracted source file facts, clustered modules into subsystems, and refined these clusters using domain knowledge about modern operating systems. We found that the concrete architecture did not match the conceptual architecture we formed earlier. The initial extracted structure showed missing dependencies due to limitations in the extraction tool. After manually verifying the dependencies, we found that all absences were caused by these tool limitations. On the other hand, the final concrete architecture exhibited several dependencies that were not accounted for in the conceptual architecture. These dependencies were due to design decisions such as having the IPC subsystem perform some of the memory manager swapping functions. These unexpected dependencies, or divergences as described in [Murphy 1995], could indicate that the conceptual architecture wasn't properly presented, or that the concrete architecture didn't use an appropriate clustering mechanism. However, we believe that these divergences are the result of the Linux developers ignoring architectural considerations.

We reviewed the documentation available for the Linux kernel. There is a large quantity of system documentation, such as [Rusling 1997]. There was even some limited documentation available with the source code. However, this documentation was at too high a level to serve as a detailed concrete architecture.

We used the cfx tool to extract detailed design information from the Linux kernel source. The cfx extractor tool produced one hundred thousand facts, which were combined using grok into fifteen thousand dependencies between one sixteen hundred source modules. After examining the output of the fact extractor, we noticed that there were two problems with the results. First, some dependencies were not detected. This occurred in particular in the case of implicit invocation. In the implicit invocation case, a subsystem registers its capabilities with another subsystem; since the access to the registered subsystem is through function pointers, the cfx tool was not able to detect these dependencies. Secondly, the cfx tool reported dependencies that did not exist. One example of this occurred in the IPC subsystem. Three source modules in the IPC subsystem have a function (findkey) that has the same name. Since the cfx tool uses a name-equivalency assumption, it reported that all three modules depended on each other when a manual verification showed they did not.

In addition to the artifacts introduced by the cfx tool, we encountered some problems because of the assumption by the grok scripts that each source module (*.c) had an associated header file (*.h). This is generally true in the Linux kernel, but there are some header files that are implemented in many source modules. For example, mm.h is implemented in several source modules in the memory manager subsystem. Since all calls to functions are abstracted to calls to the header file where the function is declared, we had difficulty separating source modules into subsystems. All of the source modules were shown depending on mm.h rather than the specific source module that implemented the resources being used. This problem could be corrected by adjusting the grok scripts; perhaps some combination that recognized the special nature of header files such as mm.h would be best.

Because of these problems (missed, spurious, and misplaced dependencies), we manually verified the extracted facts where they differed from our expectations. Unfortunately, the volume of source code and our relative inexperience with Linux means that our extracted dependencies cannot be viewed as completely accurate.

4.1.1 Future Work

The PBS tools should be adjusted to handle the Linux source structure. The conceptual and concrete architectures we have presented should be refined through discussions with the Linux developer community. After refinement, the two models can be compared using the Reflexion model [Murphy 1995].

5 Definitions:

BSD Berkeley System Distribution:	A version of Unix based on the AT&T System V Unix developed by the Computer System Research Group of the University of California at Berkeley.
device special file:	a file on the file system that represents a physical device. Reading or writing to this file will result in direct access to the physical device. These files are rarely used directly, with the exception of character mode devices.
fs:	the file system subsystem of the Linux kernel
interrupt latency:	the time between when an interrupt is reported to the CPU and the time that it is handled. If interrupt latency is too large, then high-speed peripherals will fail because their interrupts are not processed (by reading data from their CSR's) before the next interrupt is asserted. The subsequent interrupt overwrites the data in the CSR's.
IPC:	Inter-process Communication. Linux supports signals, pipes and the System V IPC mechanisms named after the Unix release in which they first appeared.
kernel thread (daemon):	a kernel thread is a process that runs in kernel mode. These processes are properly part of the kernel since they have access to all of the kernel's data structures and functions, but they are treated as separate processes that can be suspended and resumed. Kernel threads do not have virtual memory, but access the same physical memory that the rest of the kernel does.
logical file system:	a system that present blocks of data as files and directories; these files and directories store attributes that are specific to the logical file system, but can include permissions, access and modification time, and other accounting information.
mm:	the memory manager subsystem of the Linux kernel
mount point:	the directory within the virtual file system where a logical file system is mounted. All files on the mounted logical file system appear as subdirectories of the mount point. The root file system is mounted at '/'.
Net:	the network interface subsystem of the Linux kernel
Reverse Engineering :	The process of extracting high-level models from the source code.
Sched:	the process scheduler subsystem of the Linux kernel
Software Architecture:	The structure(s) of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.
System V:	A version of Unix developed by the Bell Laboratories of AT&T..
Unix:	An operating system originally designed at the Bell Laboratories of AT&T in the 1970s

6 References:

[Balasubramanian 1993]
Balasubramanian, K. and Johnson D.: "Linux Memory Management Overview," The Linux Kernel Hacker's Guide, http://www.redhat.com:8080/HyperNews/get/memory/memory.html.

[Bass 1998]
Bass, Len, Clements Paul, Kazman, Rick, "Software Architecture in Practice", ISBN 0-201-19930-0, Addison Wesley, 1998

[Bowman 1998]
Bowman, I.: "Conceptual Architecture of the Linux Kernel", http://www.grad.math.uwaterloo.ca/~itbowman/CS746G/a1/, 1998.

[Chi 1994]
Chi, E.: "Introduction and History of Linux", http://lenti.med.umn.edu/~chi/technolog.html, 1994.

[CS746G Bibliography]
http://plg.uwaterloo.ca/~holt/cs/746/98/linuxbib.html

[Holt 1997]
Holt, R.: "Portable Bookshelf Tools", http://www.turing.toronto.edu/homes/holt/pbs/tools.html.

[Knowlton 1965]
Knowlton, K.C.: "A Fast Storage Allocator," Communications of the ACM, vol. 8, pp. 623-625, Oct. 1965.

[Knuth 1973]
Knuth, D.E.: The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition, Reading, MA: Addison-Wesley, 1973.

[Kruchten 1995]
Kruchten, P.B., The 4+1 Views Model of Architecture, IEEE Software, Nov 95, pp 42-50.

[LDP]
Linux Documentation Project, http://sunsite.unc.edu/mdw/linux.html.

[Mancoridis Slides]
Mancoridis, S.: MCS680 Slides, Drexel University.

[Muller 1993]
Muller, Hausi A., Mehmet, O.A., Tilley, S.R., and Uhl, J.S.: "A Reverse Engineering Approach to Subsystem Identification", Software Maintenance and Practice, Vol. 5, 181-204, 1993.

[Murphy 1995]
Murphy, G.C, Notkin, D., and Sullivan, K., Software Reflexion Models: Bridging the Gap between Source and High-Level Models, Proceedings of the Third ACM Symposium on the Foundations of Software Engineering (FSE '95)

[PBS]
Our Extracted Linux Landscapes, http://plg.uwaterloo.ca/~itbowman/pbs/.

[Rusling 1997]
Rusling, D.A: The Linux Kernel, ftp://sunsite.unc.edu/pub/Linux/docs/linux-doc-project/linux-kernel/, 1997.

[Siddiqi 1998]
Siddiqi,S.: "A Conceptual Architecture for the Linux Kernel", http://se.math.uwaterloo.ca/~s4siddiq/CS746G/LA.html, 1998.

[Tanenbaum 1992]
Tanenbaum, A.S: Modern Operating Systems, Englewood Cliffs, NJ: Prentice-Hall, 1992.

[Tanuan 1998]
Tanuan,M.: "An Introduction to the Linux Operating System Architecture", http://www.grad.math.uwaterloo.ca/~mctanuan/cs746g/LinuxCA.html, 1998.

[Tzerpos 1996]
Tzerpos,V. and Holt,R.: "A Hybrid Process for Recovering Software Architecture", http://www.turing.toronto.edu/homes/vtzer/papers/hybrid.ps, 1996.