Unix Internals

Dean Brock
Department of Computer Science
University of North Carolina at Asheville

Most users of Unix believe that the word “Unix” refers to a collection of powerful, mysterious, and sometime frustrating commands like `ls`, `grep`, and `vi`. However, to the Unix geek, the term “Unix” refers to the Unix operating system, or kernel. The Unix operating system is the foundation of all the utilities of Unix. Its synonym, “kernel,” suggests the central role the operating system plays in Unix. The most reverent Unix users call it simply “the system.”

The kernel is both the slave and master of all Unix applications. For example, the `more` command lists a file by ordering the kernel to provide the file's contents. However, it is the kernel which determines when `more` can run and decides if `more`, or more precisely the user running `more`, is really allowed to read the contents of the file.

The kernel is positioned between the user applications and the hardware as shown in Figure 1. In certain instances, applications have direct access to the hardware. For example, the application can multiply two integers with a single machine instruction. However, other facilities of the computer are protected from direct manipulation by user applications. For example, a user application cannot directly command the disk to write blocks of data. If this were possible, applications written by malicious *or* careless users could damage the structure of the Unix file system. An application is not even allowed complete control over when it runs. If it were, it could “hog” the resources of the processor and prevent other jobs for performing their work.

The top layer of the operating system is the *system call interface*. To the programmer, this is a collection of about 100 special functions. These are the functions described in Section 2 of the Unix manual. They are named “system calls” because they are invocations of the Unix operating system. One system call is `rmdir`, which removes a directory. A simple program using `rmdir` to remove a directory called `remove.me` is shown in Figure 2. You are probably familiar with the Unix command `rmdir`, documented in Section 1 of the manual. This command is nothing more than a C program that performs the system call `rmdir` and then determines if the call succeeds.

Oddly enough, Unix programmers rarely make direct use of system calls. Generally they use friendlier C library routines, such as `fprintf`, which do the dirty work of speaking to the kernel via system calls, such as `write`. System calls are Unix's lowest-level API, or Application Programmer's Interface. Unfortunately, there are minor variations of this API between the various flavors of Unix. This differences are especially noticeable in programming tarpits like controlling a terminal.

POSIX, IEEE's *Portable Operating System Interface for Computing Environments*, is one attempt to create a universal Unix-like system call interface sanctioned by the American National Standard Institute and similar organization. POSIX is sufficiently portable that even non-Unix operating systems such as OS/2, Windows NT, and VMS can hide their considerable blemishes beneath a POSIX interface. If you must make system calls, it's a good idea to stick to the POSIX interface. Read Lewine's book, *POSIX Programmer's Guide*, for more information about programming with POSIX.

The kernel is a large program weighing in at several megabytes of compiled code. The operating system on your Unix computer is probably stored in a file called `/vmunix` or `/unix`. Almost all of the kernel is written in ordinary C. A few routines are written in low-level assembly or machine code. In general, assembly-coded routines form a thin veneer that covers the upper and lower surfaces of the kernel. The purpose of this chapter is to discuss the structure of the kernel. As you might suspect, Unix kernels do vary from vendor to vender; however, almost all versions incorporate structure, concepts, and even C statements derived from early versions of Unix written at Bell Labs and enhancements provided by the Computer Systems Research Group of the University of California at Berkeley. Consequently there are many similarities among widely-used Unix operating system kernels. Significant vendor-written additions, such as Sun's Network File System, find their way into the products of other vendors by cross licensing agreements.

The kernel has complete control of the hardware. It can execute privileged machine instructions that are forbidden in user-written applications. The operating system achieves this power by executing in a special mode we'll call *kernel* mode but which also goes by the names *monitor*, *supervisor*, or *system* mode. Modern computers are designed to support at least two modes of execution in order to run multi-user operating systems such as Unix. We'll call the other mode of operation *user* mode, because it is the mode in which all user-written applications run. Almost all the important control programs, such as the line printer spooler or *daemon*, are executed in user mode. The power of these daemons (and yes, that is the correct spelling) is derived from their "*user id*," which is usually 0, the number identifying *superuser* or *root*. The kernel has some very special rules for dealing with system calls made by applications running under the user id of the superuser. We'll see a few of these later in this chapter. There are a couple of daemons which actually do execute in kernel mode, but because these daemons execute code that is compiled into the kernel, it's best to consider them an integral part of the operating system. We'll return to these oddities in a little while.

One of the privileges of system mode is the ability to switch into user mode. The kernel can schedule an application, or process, to run by switching into user mode and then branching to compiled code of that application. The application switches into kernel mode when it makes a system call. At the machine level, this is accomplished by executing a special *trap* instruction, which may be called a *monitor call* or *supervisor call* by some computer manufacturers. Because the trap instruction is also an automatic branch to compiled code of the kernel, the user application loses control of the machine when it executes the trap and thus the user code is denied execution in the privileged state.

Underneath the operating system is the hardware. We've already mentioned that the operating system has privileged access to the instructions needed to schedule user applications. The kernel also has direct access to devices attached to the processor. One of the important services provided by the kernel is hiding all the nasty little details of handling devices from user-written programs. For example, an application doesn't need to "know" how to position the heads of a disk in order to read data. It only needs to open a file. We'll discuss device handling in detail in this chapter. It isn't easy. The Unix kernel, unlike the MS/DOS "kernel," doesn't just order the disk to read a block of data and wait for the disk controller to respond. Rather, the system directs an I/O request to the controller and then finds some other useful work to perform. When the disk has completed its assignment, it *interrupts* the kernel which then forwards the disk's data to the appropriate user application.

We'll look at the components of the average Unix kernel in the remainder of this chapter.

First, we study the control of running applications, or *processes*. We'll pay particular attention to the problems of executing several processes "simultaneously." Then we'll see how the kernel manages memory and actually provides applications with more memory than is physically present in the computer. Devices and files are next. Here, the Unix file system is dissected. Unix's networking capability is a significant reason for its popularity. The implementation of the popular TCP/IP protocol suite is the next-to-last topic of the chapter. The chapter ends with a discussion of the steps required to get a Unix system up and running when it is powered on.

You will not find the details of the Unix kernel in this chapter. You'll have to buy a source code license or get your hands on one of those public domain versions of Unix to make a really close inspection. I've tried to cover those aspects of the kernel that you're most likely to encounter as user or system administrator. Many of the important data structures of the kernel are presented in this chapter. If you have a working Unix system, you can view many of these structures using system "debugging" commands such as `pstat`, `ps`, or `crash`. Most Unix systems have a directory, usually `/usr/include/sys`, which contains C definitions of the data structures used by the kernel. If you want a more interactive reading of this chapter, you should log into your Unix system and check out your possibilities for examining the kernel as you read.

Processes

The universal definition of a process is a *program in execution*. A Unix application, loaded and running, is a Unix process. There are many components to the process. Many of these parts go by special names in the Unix system. The compiled code of the process is the *text*. The local variables, those declared within a function, are stored on the *stack*. The remaining storage, external variables and dynamic data obtained by calling `malloc()`, is simply called *data*. The programmer is well aware of these three components of the process. In fact, data and compiled code are often the only characteristics of a process that concern your average C programmer.

However, the kernel must keep up with many other features of the process. For example, the kernel must "know" the PC (program counter) of a process, so that the process can be resumed after it is stopped. The kernel must also know the identity of the user running the process, the files opened by the process, and many other details. As a result, the process is divided into two halves. The user half consists of memory that can be directly accessed and/or changed by the user's program. This includes the previously discussed text, stack, and data areas and, in some versions of Unix, shared memory segments attached to several processes. The other half consists of a couple of data structures used by the kernel to manage the process. These data structures may only be accessed by the kernel and are the focus of our discussion of the internals of process management.

In the "traditional" implementation of Unix, there are two kernel data structures for each process. The first is the *process table entry*, also known as the *proc structure* or *proc area*. The process table is an array with one entry for each possible process. Each entry contains fields that hold the values of certain crucial features of the process. Typically these include:

Process id

Process credentials

- User id -- real, effective, and saved
- Group ids

Related process
    Parent process id
    Pointer to list of child processes
Scheduling information
    Priority
    System usage
    “Nice” value
Signal information
    Blocked and handled signals
    Pending signals
State of the process -- running, waiting, *etc*.

along with many, many other values. The file `/usr/include/sys/proc.h` gives a complete definition of the proc structure on most versions of Unix.

The second kernel data structure for managing the process is the *user area*. A user area is allocated for each newly created process. The user area contains process data the kernel must “know” when the process is active. An inactive process may be completely unloaded from memory and *swapped* to disk. The user area may be swapped along with all information -- text, data, and stack -- used by the process when it executes in user mode. The separation of process-related kernel structure into process table entry and user area allows the kernel to store some of its knowledge of inactive processes on disk. In general, the process table entry contains the minimal information needed to determine which process gets the next chance to run. Our previous list of process table entry fields may seem a bit excessive for these purposes, but many of these fields qualify by unexpected means. For example, a signal can start a process; consequently, the signal information must be contained in the process table entry.

The user area contains many fields. Typically it contains copies of arguments passed to the kernel by system calls, memory management information, and statistics related to system resource utilization. One of its most important components is the *file descriptor table*, an array used to record files opened by the process. In some versions of Unix, the user area ends with the process' *kernel stack*. When the process makes a system call and executes in kernel mode, it uses its own personal kernel stack to hold the local variables of the kernel's C routines. You should be able to find a definition of the C data structure for your computer's user area in the file `/usr/include/sys/user.h`. When executing a system call, the kernel variable `u` is set to refer to the user area of the calling process, so that `u.u_ofile` (or something similar) refers to the file descriptions of the current process. Consequently, the user area is sometimes called the *u area* or *u dot area*.

Some versions of Unix have made radical changes to the traditional Unix process. One change is kernel support for *threads* or *light weight processes*. A thread is similar to a process in that it has a program counter and stack, but several threads may execute within a single process and share certain resources of the process, such as open files. OSF/1 is an example of a Unix operating system which supports multi-threaded processes. The OSF/1 kernel is a marriage of the Mach micro-kernel developed at Carnegie Mellon University and a traditional Unix kernel. In OSF/1, the user area has been divided into two separate structures: a *utask* structure for the entire process and a *uthread* structure for each thread. Additionally, the Mach-inspired routines of OSF/1 use their own task (process) and thread structures to schedule threads and manage virtual memory. Consequently, the proc and user areas of OSF/1 are quite a bit different than those you'll find in many other versions of Unix.

Many Unix system calls can be satisfied merely by accessing fields of the proc and user areas. For example, `get_pid` (get process id), requires nothing more than getting into kernel mode and retrieving the `p_pid` field from the process table entry of the executing process. A process invokes a system call via a *wrapper* routine. The wrapper is a machine-specific procedure that (1) packs the system call arguments for shipment into the kernel, (2) transfers into kernel mode by executing a special hardware trap instruction, and (3) unpacks the results returned by the kernel into the system call return value and, when appropriate, an error variable `errno`.

When a process enters kernel mode to execute a system call, the kernel executes that call until (1) the call completes, (2) the kernel decides to *block* to wait for an event, such as the completion of a disk operation, or (3) an interrupt occurs. In case 1, system call completion, the kernel will check to see if the running process has exceeded its *time slice* or *quantum*. If it hasn't, the process re-enters user mode. The kernel will return arguments to the user process by placing return values in machine registers, or perhaps on the user stack. To the process, the system call appears pretty much like any other procedure call. However, if the process has exceeded its quantum and there are other processes which are ready-to-run, one of those processes will be given the CPU for its own quantum.

In case 2, blocking, the kernel inserts the process into a queue of *sleeping* processes. Process “sleep on” a specific event, represented by a *wait channel*. Usually the wait channel is the address of the data structure associated with the expected event. For example, if a process is waiting for a data block to be read from a disk, the wait channel should be the address of the buffer that will eventually contain the block. The `ps` commands lists the wait channels on which processes are napping. You'll have to read the `man` page for your system to see exactly how to do this.

Not all events are associated with external I/O. For example, one process may wait for another to terminate or to write a character to a pipe. The kernel must be carefully written to control process sleep. It must make sure that sleeping processes really do receive the appropriate wakeup calls. Also, some forms of sleep are *interruptible* in that the process may be awakened before the event really occurs. In general, a process enters an interruptible sleep when there is no guarantee that the awaited event will occur in a timely fashion. Waiting for terminal input is an example of an interruptible sleep. It may be necessary to awaken the process before the event really occurs. A process entering interruptible sleep must check the reasons it was awakened when it is rescheduled. Other sorts of sleep, *e. g.*, waiting for disk I/O, are non-interruptible. The use of the word “interruptible” is a bit misleading here. It does not mean that the hardware is not allowed to interrupt the system but only that the process sleep may be terminated early. Using `ps` on your system, you should be able to recognize non-interruptible sleepers by examining either the `PRI` field or the `STAT` field.

In case 3, the *real* hardware interrupt, the kernel is diverted from the task of completing the system call. The interrupt is hardware's way of getting the attention of the operating system. The kernel executes a handler appropriate for the interrupt. Handlers run briefly. Usually, they set a bit or two, and occasionally, they initiate new I/O operations. For example, assume a disk interrupt handler has been invoked to oversee the completion of a read operation. The handler will locate processes waiting for the newly arrived data by examining the wait channel of sleeping processes. Processes waiting for the data are marked as ready-to-run. They will be restarted at a later moment. Now that one disk operation is finished, the handler may be able to start another.

The handler executes on its own stack, independent of the process that was executing when the interrupt was delivered. Because processes may be interrupted, the kernel must be carefully written to prevent processes from interfering with each other. Suppose the following C statement occurs within the kernel:

```
num_procs++ ;
```

The intent of the statement is to count the number of processes that have been created by the kernel. Translated into machine language, the single C statement expands to several machine instructions similar to:

```
LOAD        ACC,num_procs
INCREMENT   ACC
STORE       ACC,num_procs
```

There is a danger that between the time one process loads the value of `num_procs` into the accumulator `ACC`, increments the accumulator, and stores the result back in `num_procs`, an interrupt will occur. Should a second process be scheduled after the interrupt, it could charge through this same section of code with the eventual result that `num_procs` will be incremented once rather than twice. The consequences of this race ending in a "tie" aren't serious, but the Unix kernel must be designed to avoid situations where two processes or a process and an interrupt handler interfere with each other while executing within the kernel.

Within the traditional Unix kernel, there are three ways of arbitrating ties. First, the kernel can disable specific interrupts for brief periods to ensure that a crucial section of code is executed as an atomic unit. Second, data structures can be protected by *busy* fields. The first process to use the structure sets the busy field. Late arrivals will test the busy field and, seeing it set, sleep upon the event of the data structure being freed. The temporary owner of the structure will awaken the sleepers when it is done with the structure. Finally, most Unix kernels are *non-preemptible* in the sense that if an interrupt occurs while the CPU is performing a system call in kernel mode, the interrupt handler returns the CPU to the process executing the system call. Non-preemptibility severely restricts the opportunities in which two processes may race within the kernel. In fact, for practical purposes, processes race only with interrupt handlers.

However, non-preemptibility has its costs. It can delay important processes from useful work in user mode while some other process completes a time-consuming system call such as `fork` or `exit`. In some kernels, this cost is reduced by building *preemption points* in the kernel where processes performing a system call may volunteer to yield the CPU. True multi-processing, where the computer contains several processors, is the ultimate foe of the non-preemptible kernel. The "benefits" of non-preemptibility are lost if more than one processor can simultaneously execute in kernel mode. Preemptible Unix kernels, such SMP (symmetric multi-processing) versions of System V and OSF/1, have required additional mechanisms for controlling concurrency. OSF/1 has borrowed *locks* from Mach. Kernel data structures are protected by locks which processes hold while manipulating the data structure. Locks resemble the previously discussed fields but are designed for use within SMP kernels.

Signals

Let's take a look at how signals are delivered and handled in Unix to illustrate how the kernel controls processes. Signals are an interrupt mechanism for user processes. One process can send a signal to another using the `kill` system call. Signals are also generated for machine exceptions, *e. g.* the process dividing by zero, and terminal interrupts, *e. g.* the user typing a `^C`.

A user process installs a signal handler, a function executed when the signal occurs, by calling the Posix library routine `signal`, which in turns calls the appropriate system call, most likely `sigvec`, for the running version of Unix. Executing within the kernel, the process modifies its process table entry to show that the signal is being handled and modifies its user area to contain the address of the handler. The process table field that keeps up with handled signals is usually called `p_sigcaught`. It is modified by executing a C statement similar to:

`p->p_sigcaught |= 1 <<` *signum*

Because of the possibility that an interrupt handler may also want to modify this same field at the same time, this statement should be executed with either interrupts off or locks on. The address of the signal handler is usually stored at location `u_signal[`*signum*`]` within the user area. Note that the process table entry contains only the signal information necessary to determine if the process should be scheduled. All other information needed when the process is running is stored in the user area.

Invoking the signal handler is divided into two phases: posting and delivery. Posting consists of checking the proc structure to see if the signal is being handled and, if so, setting the appropriate bit of field `p_sig` of the process table entry to show that the signal is pending. If the target process is in an interruptible sleep, it will be awakened.

In kernel mode, processes can post a signal to other target process; however, processes must deliver their own signals. Just before a process makes a transition from kernel to user mode, it checks its process table entry. If a signal is pending, it arranges for the delivery of the signal. The mechanism for actually invoking the system handler from user mode is very machine dependent. It is accomplished by modifying the process' user stack so that it will execute a small section of “trampoline code” when it returns to user mode. The trampoline code is responsible for executing the signal handler and then bouncing back into the kernel. Many fields of the process table entry and user area will be modified to handle the signal. Signal delivery is one of the kernel's most challenging tasks.

Virtual Memory

Today's computers allow programs to address more memory than is physically available in the machine. This is accomplished with a mechanism called *virtual memory*, in which *virtual* addresses generated by the program are translated into *physical* addresses that point to *real* memory locations. The hardware used to accomplish this translation varies. In general, the virtual address spaces addressed by processes is divided into fixed sized *pages*. The physical address space is also divided into pages, although many people reserve the term “page” for the divisions of virtual memory and the term “frame” for the divisions of physical memory. For clarity, we'll follow that convention in our discussion. Page sizes range from 512 to 16,384 bytes on modern CPUs. Each virtual address is divided into two parts, a page number and an offset within the page. A hardware address translation facility transforms the virtual page number into a physical frame number. These interactions of the various components of virtual memory are illustrated in Figure 3.

Because there are more virtual addresses than physical addresses, occasionally a process will attempt to address a virtual page that has no corresponding physical frame. When this occurs a *page fault* generates an interrupt and causes the processes to enter kernel mode. In the Unix operating system, pages that are not in core, *i. e.* in physical frames, are stored in an area of the disk called the *swap space*. When a page fault occurs, the kernel must move the needed page

from the swap space into a physical frame. The kernel must also run a *page replacement* algorithm that locates a victim frame to *page out* to disk to free up frames needed to *page in* recently addressed data.

The goal of good memory management is to minimize page faults. This requires a page replacement algorithm that has a knack for choosing victim frames unlikely to be accessed in the near future. The details of virtual memory management are best left to the classic operating system texts such as Silberschatz and Galvin's *Operating Systems Concepts* or Deitel's *Operating Systems*. The details of hardware address translation vary widely (and wildly) among various computer designs. You'll have to consult the hardware reference manual for your computer to extract information about your own system's peculiarities. Most Unix users and system managers are blissfully ignorant of these details.

Early Unix operating systems did not support virtual memory. Instead they divided memory into two parts, one part for the kernel and the rest for user programs. User programs would run only when they were completely loaded into physical memory. When it was necessary to run unloaded user programs, entire programs, with the possible exception of shared text or compiled code, were swapped to and from disks. Process 0 was the *swapper* or *swap daemon*, a kernel-only process in charge of this loading and unloading of user processes.

Release 4.0 of the Berkeley Standard Distribution, 4BSD, was the first version of Unix to support true virtual memory. A new kernel-only process, the *page daemon*, now ran as process 2. The page daemon is awakened at regular intervals, one-fourth of a second in early BSD implementations, to check the number of free frames in the system. If there are lots of free frames (in particular, if the number of free frames is greater than a threshold variable `lotsfree`), the page daemon goes back to sleep. Otherwise, the page daemon scans several physical frames and places inactive frames on the *free list*. The identification of inactive frames is a machine-dependent operation. Some computers provide significant hardware support to identify inactive frames. Some provide none; on these computers, the operating system must be creative.

When page faults occur, frames are taken from the free list and given to the process needing the memory. Thus one process *steals* pages from another. *Dirty* pages have been modified since they were loaded into memory. Before a dirty frame can be turned over to another process, the kernel must make sure that the contents of the frame are written to the swap area.

Under certain circumstances a system may be trying to run more processes than its limited memory can handle. When this happens, the system *thrashes* as processes continually steal pages from each other. When the page daemon notices that it is having a hard time finding inactive pages, it will ask its fellow kernel-only process, the swapper, to forcibly move entire processes out to disk.

Many ideas of 4BSD memory management, such as the threshold-driven page daemon, survive in recent versions of Unix. However, several vendors have modified these algorithms to improve performance. Most of these modifications improve the ability of processes to share pages and will be described in a couple of paragraphs. SunOS was the first Unix to allow different processes to share libraries of commonly used routines. Such sharing significantly reduces the space used by executable problems on the disk and in memory.

The OSF/I kernel seems to be the most radical departure from 4BSD. It uses a memory management strategy adopted from Mach. There is a threshold-driven page daemon, but now it maintains three pools of pages: active, inactive, and free. It also supports *external pagers*, user

tasks which assume the responsibility of moving pages between memory and their more permanent location, which could be a disk drive or some exotic networked storage device.

In the System V flavor of Unix, the virtual memory address space of a process is divided into *regions* of contiguous pages. The data structure used to manage regions is usually defined in the file `/usr/include/sys/region.h`. The previously mentioned text, data, and stack are each stored in their own regions. Shared memory segments also form regions. Each process has its own *per process region table*, or *pregion*, which points to its regions. Under certain circumstances, *e. g.*, two processes executing the same text or accessing the same shared memory segment, two or more pregion entries point to one region. In OSF/1, the kernel uses *virtual memory objects* for a similar purpose.

The challenge of Unix memory management is allowing different processes to efficiently share pages. The `fork` system call is the biggest creator of sharable memory in Unix. The `fork` creates a new child process by "copying" the virtual memory of the parent process. Because `fork` is the only way to create a process, it is used frequently by Unix programs. Every time you type a new Unix command using a shell, the shell calls `fork` to create a new copy of itself. However, all this copied data is only used for a very short time. Soon the newly created process will make the system call `execve` and completely replace itself with the code and data of program required to execute the command you typed.

All Unix implementations easily handle sharing of read-only text regions. Efficient sharing of writable data and stack segments is more difficult. This requires the use of a technique called *copy-on-write*. In copy-on-write, two or more processes share a page in a read-only fashion until one process attempts to modify the page. At that time, the writing process causes a memory protection fault. The kernel diagnoses the protection fault as an attempt to modify a copy-on-write page and gives the writing process its own writable copy of the page which it may modify however it pleases. Today most commercial versions of Unix support some form of copy-on-write.

In addition to allocating physical frames to processes, the kernel must also allocate sections of the swap area to provide the backing store needed for pages that can not be stored in core. Usually, the swap area is stored on a contiguous section of a disk. It is also possible to spread the swap area over several disks so that system performance can be improved by scheduling multiple concurrent page swaps. Many versions of Unix also support paging over a network connection for diskless workstations. In general, the location of the primary swap area will be built into the kernel while the locations of secondary swap areas are specified in system files.

It is possible for a Unix system running many large processes to run out of swap space, resulting in "insufficient memory" error. This error is often misinterpreted by the neophyte Unix user to mean that more RAM is needed when, in fact, it is the swap space that is exhausted. The problem can be eliminated by allocating additional secondary swap areas.
Most Unixes have a `pstat` command which prints how much of the swap space is free. This command is useful for anticipating shortfalls in swap space. The `vmstat` command also prints all sorts of useful statistics about paging activity. Consult your `man` page for the details.

Devices

Physical devices such as disks and tapes attach to our computers in a modular fashion. You'd hope that our operating systems support similar plug-in attachment for the software

needed to control these devices, and indeed they do. All contemporary operating systems (yes, even MS/DOS) provide a means of controlling devices through modules called *device drivers*.

The Unix device driver is a collection of procedures. Each procedure is responsible for performing some operation on the device. All kernel code specific to the peculiarities of the device are hidden within the driver. There is only one part of the Unix operating system that "knows" what it takes to read a sector from a SCSI disk: the SCSI disk device driver.

There are routines, or *entry points*, in the device driver for the I/O operations, like read and write, which are available at the user level. The entry points of these routines are stored in a *switch table*. Whenever the kernel must perform a particular operation on the device, it simply looks up the appropriate routine in the device's switch table and calls it. In addition, the device driver contains routines for probing the system to see if the device is present, for initializing the device, and for handling device interrupts.

In Unix, there are two types of devices, *character* and *block*, and consequently two types of device drivers and corresponding switch tables. The definitions of the structures which hold the switch table are usually found in `/usr/include/sys/conf.h`. Within the kernel, device drivers are known by numbers, actually indices into the global device switch tables. Because there are separate switch tables for character and block devices, it is possible for different character and block device drivers to share the same number.

Individual devices are known by two numbers. A *major* number identifies the device driver. A *minor* number identifies the specific device *(e. g.*, the tape drive with SCSI ID 2), or a specific part of a device (*e. g.*, the fourth partition of the third disk), or sometimes even a specific way of accessing the device (*e. g.*, the second tape drive set to its highest output density). The device driver is expected to handle all devices of its type. User programs may access devices through device *special* files contained in the directory `/dev` and its subdirectories. By typing "`ls -l /dev`" you can obtain a listing of the many device special files of your computer. On my workstation, the entry

```
brw-------   1 root     system     8,3074 Jun 30 15:45 /dev/rz3c
```

identifies a block device (note the `b' in the first column) with major number 8 and minor number 3074. The entry

```
crw-rw----   1 root     system     0,  1 Jun 30 15:00 /dev/mouse
```

is for as character device ('c' in the first column) with major number 0 and minor number 1. Device special files are created by the kernel in response to the `mknod` system call. See this book's chapter on System Administration for more information.

Block devices are usually reserved for disk partitions. The kernel reduces the number of disk I/O operations by saving recently read and written data blocks in an area of memory called the *buffer cache* or *buffer pool*. Whenever possible, disk I/O requests are satisfied using the buffer cache rather than the disk. Device drives for block devices transfer data to and from the buffer pool. Before actually initiating a I/O operation for a block device, the kernel checks the buffer pool to see if the desired data is already present in memory.

Disk partitions are actually accessible through both character and block device special files. The character, or *raw*, device is needed for low-level operations such as formatting the disk or writing a new Unix file system onto a disk partition. These devices are often given names starting with an 'r', such as `/dev/rsd0c`, to hint at their raw nature. The block device is needed for relatively high-level operations such as file system backup.

Device drivers are not just for reading and writing. The device driver for a CD-ROM reader has an `ioctl` (I/O control) entry point that can be reached via the system call of the same

name. Given the right CD and the right `ioctl` arguments, the device driver will order the CD-ROM to start playing MacArthur Park through your ear phones. (And *please*, use the ear phones!)

There are also many useful *software* devices, sometimes also called `pseudo` devices. The kernel memory itself can be accessed via a software device known to users as `/dev/kmem`. Reads and writes of `/dev/kmem` are transformed into reads and writes of kernel memory by the software device driver. Programs such as `ps` and `pstat` open `/dev/kmem` to read the kernel variables needed to generate their reports. If you want to take an intimate look at a running kernel to test your knowledge of Unix internals, run `dbx` using `/dev/kmem` as ``core.'' The ultimate example of a do-nothing pseudo device is `/dev/null`, which looks like a empty file when read and like a trash can when written.

This flexibility of the device driver interface has resulted in a few feature-laden devices. Terminal drivers are immense. They have been given the responsibility for echoing and editing user input and implementing the concept of a *control terminal*. When a process without a control terminal opens a terminal, the terminal driver assigns the opened terminal to be the control terminal of the process. The terminal driver also provides `ioctl`s that allow the process to change its control terminal. When special interrupt characters like control-C are typed on a terminal, the terminal driver posts the appropriate signal to all processes under that terminal's control. The terminal driver is also able to make distinctions between foreground and background processes. For example, the terminal controller can be asked to post a signal to any background process attempting to write to the terminal.

Obviously, there are many applications that don't really want the terminal driver to provide these services. Text editors really want to do their own editing and processing of ``interrupt'' characters without any interference from a terminal driver. Consequently, the terminal driver provides different levels of services ranging from a *canonical* or *cooked mode*, with full editing, to a *raw mode*, with little processing. To further complicate matters, there are several different flavors of cooked terminals. Some applications prefer the “standard” and others prefer the “new.” Often other choices, such as the System V and Posix interfaces, are available. These choices are usually implemented in the form of *line disciplines*, collections of routines that are invoked by the terminal driver.

Because so many Unix applications are dependent on services provided by the complex Unix terminal driver, the kernel must include special pseudo terminal device drivers so that applications connected to a networked or windowed “terminal” session will behave properly.

In System V, *streams* may be used to extend the capabilities of ordinary device drivers. Streams are sequences of modules starting with a *stream head*, providing the user-level interface, and terminating in a driver, providing the hardware-level interface. Between the head and the driver, there may be *stream modules* which provide special services. Data written by the user application passes through the stream head and *downstream* through the stream modules to the driver. Data read by user applications is generated by the driver and passes *upstream*. Stream modules provide services by manipulating the passing data. For example, terminal line editing can be implemented as a stream module. BSD-based versions of Unix have been slow to embrace streams; consequently, few applications use them in spite of the nifty modularization they allow. If you are interested in learning more about streams (or *STREAMS* as the System V Release 3 version is capitalized), take a look at the *STREAMS Primer* or *STREAMS Programmer's Guide*.

If you're the sort of person who attaches unusual few-of-a-kind devices to your Unix

computer, someday you may need to write a device driver. Your Unix vendor will produce the documentation needed to accomplish this job; however, you better read one of those many how-to books on the subject, such as Egan and Teixeira's *Writing a UNIX Device Driver*, before sitting down to write any C code.

Open File Management

Unix has a rather simple view of files: a file is nothing more than a sequence of bytes. Consequently, the Unix kernel doesn't have to worry about record formats. Those concerns are left to mainframe operating systems like IBM's MVS. However, Unix does allow random access to any byte within a file. That's something the designers of MVS didn't worry about.

Unix vendors have done so much experimenting with the implementation of virtual memory that it's difficult to name any kernel data structures for memory management that are common to all the Unix internals we encounter these days. In contrast, the Unix file system always revolves around three kernel data structures - the file descriptor tables, the file table, and the vnode table - which contain a fairly consistent collection of fields.

The user area of every Unix process contains that process' file descriptor table. The file descriptor table is really quite simple. It's nothing more than an array of pointers to entries of the open file table. Each time a file is opened, the kernel will allocate a unused file descriptor for the file and return to the process the index of the newly allocated descriptor. System programmers refer to the returned index itself as the file descriptor, a double use of the word that can result in some confusion. Anyway, once the user process has obtained the file descriptor, it is used in subsequent I/O operations to identify the file.

An entry within the file table is also allocated with every successful `open` system call. File table entries contain the following fields:

`f_flag`	how the file was opened, *e. g.*, read only, read-write
`f_offset`	present position within the file, *i. e.*, how far into the file have the reads and writes progressed
`f_count`	reference count of the number of file descriptor tables that point to this file table entry
`f_type`	type of the file, *e. g.* regular file or socket
`f_data`	the address of the appropriate vnode table entry or socket structure

Sockets are used in interprocess communication and will be discussed in a succeeding section. The reference count is needed because when the kernel performs a `fork` system call, it duplicates the file descriptors of parent processes for the newly created child processes, thus increasing the number of references to the file table entry. Each time a file is closed, the corresponding file descriptor is freed and the reference count of its file table entry is decremented. When the reference count reaches zero, the file table entry is freed. When child and parent process share one file table entry, they share a common offset into the file. Reads performed by one advance the file offset for both.

The information needed to manage a Unix file on disk is contained in its *inode* or *index node*. Originally, all the inodes of a disk were contained on one contiguous collection of sectors. The “identity” of the file was the index of its inode which, by the way, can be obtained using the `-i` option of the `ls` command. Information contained within the disk inode includes

user id of file's owner
group id of file's group

file *modes* or protection
size of file
type of the file, *e. g.*, regular file, directory, block device special file
major and minor device numbers for device special files
map of the file's data blocks

When a local file is opened for the first time, an incore *vnode* or *virtual node* is allocated that will hold the disk inode plus additional information such as the inode number and a reference count. The vnode is held in memory until its reference count reaches zero. Generally, that means the vnode remains allocated until all processes that have opened the file close it. When a remote file is opened via the Network File System, the vnode will contain an *nfsnode* rather than an inode. Vnodes are a relatively recent addition to Unix. What we're calling vnodes were incore inodes before Sun Microsystems introduced the Virtual File System. You'll still find a few holdouts, so whenever you see the term "inode table," think "vnode table." In Ultrix internals, you may even hear about *gnodes* or *generic nodes*.

On each I/O operation, the kernel follows pointers from the file descriptor down to appropriate vnode. It acquires the present file offset as it passes through the file table entry. When the kernel finds the file table entry, it can also check if the attempted I/O operation is consistent with the manner in which the file was opened. For example, an attempt to write to a file opened for reading will fail when the kernel looks at the file table entry and notes that the file is opened read-only. The relationship of these three table structures is given in Figure 4.

Your system's definition of file table entries should be found in `/usr/include/sys/file.h`. The file most likely to contain your system's definition of a vnode is `/usr/include/sys/vnode.h`. If you type the command "`pstat -f`", you should get a listing of the file table entries presently in use within your system. If you use the `-i` (for inode, of course) option of `pstat`, you'll be rewarded with a view of the vnode table. As a test of your understanding of the role of the file table entries and vnodes within the kernel, you might start a program that reads a very long file and try using the "`ls -i`", "`pstat -i`", and "`pstat -f`" to monitor how far into the file the program has read.

Virtual File Systems

In Unix, the file system is the data structures and routines that transform the data blocks of disk partitions into the Unix directory hierarchy. Early versions of Unix supported only one file system, one that is now called the System V File System. At Berkeley this file system was enhanced to yield a file structure very similar to those found on the disks of Unix computers today. When Sun Microsystems implemented the Network File System, the kernel had to support two types of files systems: local and remote. This was accomplished by implementing the Virtual File System.

The Virtual File System, or VFS, is the abstraction of all Unix file systems. It provides a template that can be used to plug new file systems into Unix. The file system is represented by a collection of about fifty procedures. Every vnode contains a pointer to a switch table of procedures used to perform operations on the file associated with the vnode. These modules perform operations such as creating and deleting files, looking up files within a directory, and reading and writing data to a file.

Now that Unix has VFS, the task of adding new types of file systems is much easier. Consequently, in recent years we have seen Unix kernels that can access files of a floppy disk

formatted under MS/DOS file system and a CD-ROM pressed according to the ISO 9660 specification. In a while, you'll be introduced to a couple of very unconventional file systems, such as the *process* file system containing “files” consisting of images of the virtual memory of executing processes. If you can make any collection of information look like a Unix file system (that is, have a root and a directory hierarchy), someone can write the procedures that transform it into a virtual file system.

The roots of file systems are attached to the larger Unix directory hierarchy at *mount points*. The chapter on system administration explains how the system manager constructs an ASCII file, usually `/etc/fstab`, containing lines similar to

```
/dev/vg1/tryon  /backup/tryon  ufs rw 1 2
```

which specifies that a virtual file system of type `ufs` (Unix file system), stored on device `/dev/vg1/tryon` is attached to the system's directory hierarchy at the mount point `/backup/tryon`. When the system is booted, the `mount` program makes the `mount` system call to order the kernel to add this file system to the directory hierarchy.

The kernel maintains an in-memory data structure called the *mount table* for all accessible file systems. The mechanism for maintaining mounted file systems is both simple and elegant. Let's study it using our example from the previous paragraph. Before the `mount` system call is made, a directory `/backup/tryon` must exist within the smaller directory hierarchy. This directory will be the *mount point*. The file system `/backup/tryon` has a root directory, or at least the illusion of a root directory created by the routines of the virtual file system. Mounting is the joining of the mount point directory and the file system's root directory. The system's directory hierarchy grows a new branch consisting of the newly mounted file system.

When the kernel executes the `mount` system call, it allocates two in-core vnodes, one for the mount point and one for the file system root, along with a mount table entry, or *mount structure*, which points to the two vnodes. The kernel is able to distinguish the vnodes for mount points and file system roots from their mundane counterparts. When the kernel is moving down the directory hierarchy (away from the root) and encounters a mount point directory, it uses the mount structure to jump from the mount point vnode to the file system root vnode. When the kernel is moving toward the root following a trail of `..`'s similar to `../../../..` and encounters a file system root, it crosses over to the mount point vnode. Figure 5 shows how the kernel ties these three structures together in versions of Unix supporting virtual file systems. The mount structure also contains pointers to vital information needed to maintain the file system.

In the good ole days, things were a bit different. To conserve valuable memory, the in-core inodes didn't contain pointers back to the mount structure. Crossing a mount point involved searching the mount table looking for pointers aimed back at the mount point or root inodes. If you look hard enough, you can still find a couple of systems where they still save memory the old-fashioned way.

Real File Systems

We now turn our attention to the problem of mapping a file system consisting of Unix directories onto physical devices such as disks. First, let's review the structure of a disk as illustrated in Figure 6. The disk consists of several vertically aligned *platters* rotating on a common axis. An assembly of disk *heads* moves between the platters. The circle “traced” by the head on a surface is called a *track*. A *cylinder* is a collection of vertically aligned tracks.

When the head is still, data can be read from or written to the surfaces of the platters. Data is written in *sectors*, which are 512 bytes on most recent disks.

It takes a long time for the disk heads to *seek* from one cylinder to the next. If a file is spread throughout the disk, user programs do nothing while the heads ramble from track to track. The smart operating system tries to keeps a file on a single cylinder or at least confine the file to a small neighborhood of cylinders. Mainframe operating systems like MVS tend to be smart when it comes to allocating data sectors for a file. However, they often rely on advice, such as estimates of file size from users in making these decisions. MS/DOS can be pretty dumb. It works fine when the disk is new, but in time it starts scattering files all over the disk. When it comes to file allocation, Unix operating system kernels aren't dumb, but this certainly isn't an arena in which they shine. Fortunately, they are getting smarter.

In our exposition of the disk-based Unix file system, we will concentrate on the “new” or “fast” file system of the Berkeley Software Distribution. This file system format seems to be the most common one found on Unix systems today. The Unix file system is built using the data blocks of the entire disk or a *partition* of the disks. Partitions are large collections of adjacent cylinders. The Unix operating system treats the partition as if it was a disk, just a bit smaller. The data blocks of the partition can be accessed directly though the appropriate device files. The `mkfs` command creates a new file system by writing to a partition's character device file.

The Unix file system is really a varied collection of data indices. For each file, there is an *inode* that locates the data blocks of the file. For each directory, there is a *directory file* that locates the files of the directory. And for the file system, there is a *superblock* and several *cylinder group blocks* that keep up with other useful information, such as the location of unused data blocks. We're going to take a bottom-up look at the layout of the Unix file system starting with the data blocks of the file.

To be written physically onto a disk, a file must be broken into sectors. However, because sectors are only about 512 bytes long, the kernel does not allocate disk space in units of sectors. Instead it breaks the file into data blocks of a significantly larger size, typically 8192 bytes. In addition to reducing bookkeeping costs, the larger allocation units allow the kernel to package several disk I/O operations into one. Of course, storing a 1000 byte, or even a 9000 byte, file into 8192 byte data blocks leads to certain inefficiencies, in particular about 7000 wasted bytes in the last block. To avoid this potential loss, the last “block” of small files can be broken into *fragments* of a much smaller size, usually 1024 bytes. So a file of 20000 bytes would be stored in two 8192-byte data blocks and four 1024-byte fragments with only 480 precious bytes wasted in the last fragment. The last fragments of a file are allocated so that they are all contained within consecutive fragments of one of the larger data blocks. This rule might seem odd, but it simplifies the job of kernel. It doesn't have to keep up with the location of four separate fragments, just the location of the first of the four. By the way, the sizes of blocks and fragments are not fixed by the operating system but can be specified by the system administration when he or she creates a new file system.

We introduced the disk inode two sections ago. We mentioned that every disk file has an inode which contains several pieces of information, such as its access permissions and the identity of its owner. In this section, we concentrate on only one aspect of the inode, its maps of the file's data blocks. The inode itself isn't very large. On most versions of Unix, they are only 256 bytes long and can store the locations of only a few data blocks. For larger files, a tree-structured data block directory is used. Let's use some real numbers to show how this is done. Assume that the block size is 8192 bytes and that the disk inode is big enough to hold the

locations of the first twelve data blocks. We don't have to worry about the fragment size because all the fragments of the file are stored in one data block. As long as a file is less that 98,304 (12*8192) bytes long, the location of all its data blocks can be contained within the disk inode.

For longer files, the inode points to the location of a *single indirect* block. The indirect block points to additional data blocks of a file. If four bytes are required to address a disk block, then the indirect block could hold the address of the next 2048 (8192/4) blocks of the file. With one indirect block, the kernel can manage files up to 16,875,520 (2060*8192) bytes long. However, there are a few disk hogs out there who create files that exceed the capacities of the single indirect block. For them, there is the *double indirect* block. The double indirect block is the root of a two-level data directory. It is a disk block that contains the address of several single indirect blocks which, in turn, contain the address of data blocks. The structure of the inode and its indirect blocks is illustrated in Figure 7. With double indirect blocks, files may now contain 4,188,172 (2048*2048 + 2048 + 12) blocks or 34,309,505,024 bytes. Unfortunately, the Unix requirement that files be byte-addressable and the 32-bit words of most computers restrict our files to a mere 2,147,483,647 bytes.

Originally, the inodes of the file system were stored in contiguous sectors near the beginning of the partition. This led to rather serious performance problems. For example, executing the command:

```
grep fun *
```

which searches all the files of a directory would result in a great deal of disk head motion. For each file, the heads would move to the front of the partition to read the file's inode and then back into the center of the partition to read the file's data. In today's “fast” file systems, this problem is lessening by the use of *cylinder groups* of sixteen contiguous cylinders. Each cylinder group contains its own set of inodes and data blocks. For small files, the kernel attempts to allocate both the inode and the data blocks of the file within one cylinder group. For directories, the kernel tries to allocate all the inodes of the regular files, but not subdirectories, of the directory into one cylinder group. The result of these two allocation strategies is that, in general, all the inodes and data blocks associated with a directory and its files are contained in a small area of a disk.

Each cylinder group also contains a *cylinder group block* that records the free inodes and the free fragments of the group. These records are stored as bit maps. Checking if a particular inode or fragment is free is simply a matter of going to the appropriate bit and testing its value. Testing if an entire data block is free is accomplished by testing all the bits corresponding to the fragments of the data block. The kernel tries to allocate data blocks in such a way that when successive blocks of a file are read or written, the next block is just a short rotation of the platter away.

If you've ever created a file system using `newfs`, you've probably seen the messages `newfs` prints out giving the size of the cylinder groups and the number of groups within the partition before it starts writing to the disk. After the file system is created, you can use the `dumpfs` to see this same information.

Directories are nothing more than specially marked files which have one entry for each file of the directory. The entry contains the *component name* and inode number of the file. The kernel locates the file `/etc/passwd` by searching the directory file of the root for the component `etc` and its associated inode number. It then searches the directory file for `/etc` to find the component `passwd`. The inode number of the root directory is always 2. That simple convention gives the kernel a starting place for its inode search.

There is one more special disk: the *superblock*. The superblock contains all sorts of information that is needed to manage the file system. This includes:

number of blocks in the file system
size of data blocks
size of fragments
number of cylinder groups
size of cylinder groups
number of inodes within a cylinder group

Because the information contained within the superblock is so crucial to the management of the file system, the superblock is replicated within each cylinder in such a way that it is stored on different platters throughout the partition. Your best bet for determining what is contained within the superblock and cylinder group blocks of your Unix computer is to take a short break to log in as root and try out the `dumpfs` command on a few systems. This will reinforce your knowledge of structures used to maintain the file system.

In recent versions of Unix, several vendors have made some significant changes to the Unix file system. One of these is the *journalled* file system used in AIX, the Unix used in IBM's RS/6000 workstations. In the journalled file system, the kernel maintains a transaction log of important changes made to a file system. The transaction log is actually written to a second *log* file system. For example, when a new directory is created, a record summarizing file system changes is written to the transaction log. At first glance, it's not at all obvious that the journalled file system is such a great idea. After all, aren't you now writing to the disk twice, once when you write the log record and once more when you *really* modify the file system? The advantage is that the transaction log can be written sequentially, with very little head motion to the log disk and that once the transaction record is written, the real disk updates, which are scattered all over the actual file system disk, can be written at leisure. Should the system crash before the actual updates are written, it's straightforward to update the file system to a consistent state using the information stored in the log.

AIX also introduced *logical volumes* to Unix. In most versions of Unix, a file system is confined to one disk or partition. Consequently, when designing the logical Unix directory hierarchy, the system administrator must be sure to match file systems to available physical disks. Logical volumes can be thought of as logical disks. Several physical disks are joined into a *volume group*. The volume group can be partitioned into several logical volumes, each of which can hold a Unix file system. The underlying physical data block of a logical volume may actually be spread across several disks. In fact, it is possible to *mirror* a logical volume so that each of its data blocks is stored on two different disks.

With the “standard” Unix file system, it is possible for files, especially large files, to get scattered across the disk a bit too much. To lessen this problem, some Unix vendors have introduced *extent-based* file systems, similar to those used in those proprietary mainframe operating systems. An extent is a large block of consecutive disk sectors. The file system attempts to allocate the file in a few large extents. In the best case, the file will be allocated in a single block of consecutive sectors. Because the Unix operating system provides no system calls that set the maximum size of a file when the file it created, the implementation of extent-base file systems is a challenge. One extent-based file system for Unix is Digital's recently introduced Polycenter Advanced File System for DEC OSF/1.

As we mentioned earlier in this section, not all file systems get their data from local disks. The most famous non-disk file system is NFS, the Network File System designed by Sun

Microsystems. A great deal of the administration and a bit of the implementation of NFS is discussed in the chapter on Network Administration. NFS follows the client-server paradigm. Servers *export* files. Clients *import* files. The server machine runs an NFS daemon which receives packets from clients requesting that I/O operations be performed on the server's local disk using one of the server's local file systems. Although the NFS daemon could be implemented as a user process, it is invariably implemented as a special kernel-only process just like the swapper and page daemons mentioned in our section on virtual memory.

The client side of NFS is exercised when a process makes a system call involving an imported file. Using the Sun Remote Procedure Call (RPC) protocol, the client kernel makes a request to the server. Because Sun RPC is inherently synchronous, that is, the client process makes an RPC and waits for the result, the client side of NFS is usually implemented with a gang of block I/O daemons which run as kernel-only processes on the client machine. Suppose a process has entered kernel mode making a system call involving an NFS file system and wants the service of a remote NFS daemon *but* does not want to hang around waiting for the response. In this case, the process gets one of the block I/O daemons to do the actual calling and waiting while the process goes back to its own important task of executing the user program.

Optimizing File Access

At this time, you might be wondering just why or how the kernel might issue a request for some data that isn't immediately needed by a user process. Well, it turns out that in addition to providing a mechanism for managing virtual file systems, the kernel has a few tricks to optimize the transfer of data from virtual file system to user process. First, the kernel maintains a *buffer cache* of recently read and written data blocks. I hope you remember from our discussion of virtual memory how the kernel tries to keep the active pages of a process in core and the inactive pages on disk. The kernel does something similar for the data blocks of the process using the buffer cache. When a user process writes some data, instead of blocking the user process until the data is actually written to a block of the local or perhaps remote disk, the kernel remembers the modified data in its in-core copy of the data block, stored in the buffer cache. At some later time, the data will really be written to disk. If the process modifies the data block twice before it is written to disk, the kernel saves an I/O operation. This technique may be called either *write-behind* or *delayed write* by Unix gurus.

Write-behind has a cost. If the system crashes before the new data reaches the disk, the file update may be lost, even though a *real* user has been assured that the new data was saved. To lessen this possibility, the kernel supports a system call named `sync` which causes all modified, but unwritten, data blocks in the cache to be written through the appropriate virtual file system. The kernel also can also detect when a file is being written sequentially. Once a data block is completely filled with new data, the kernel realizes that the block may not be used for a very long time and schedules the writing of data blocks to free up some room in the buffer cache.

Important file system *meta-data*, such as inodes and indirect blocks, are also stored in the data cache. The kernel must treat blocks containing meta-data carefully. If the wrong piece of meta-data is lost, files and entire directories may be lost. Usually, updated meta-data is written immediately to disk. Even then, the kernel must be careful about the order in which meta-data is written to the disk. Suppose a user deletes a file, thus freeing an inode. What should be written to the disk first, the updated inode or the updated directory entry?
No matter which is written first, the file system will be inconsistent if the computer loses power

between the two writes. An excellent discussion of how Unix maintains file system consistency can be found in Bach's *The Design of the Unix Operating System*. No Unix file system, except possibly the journalled file system, is completely safe from unexpected crashes. Consequently, when Unix boots it runs a program called `fsck`, for file system check, on its local disks. This program is able to repair minor file system inconsistences. Although `fsck` is not part of the kernel, it has a rather intimate relationship with it. `fsck` knows the order in which the kernel updates meta-data, and it uses this information quite well when it must reconstruct a file system.

File system reads are optimizing using *read-ahead*. The kernel can be an obsequious piece of code when it comes to serving user processes. It never wants to keep a user process waiting. When it notices that a file is being read sequentially, it endeavors to get that next block of data into memory before the process even asks for it by reading ahead. This is why the kernel needed all those block I/O daemons for the Network File System.

Most Unix kernels have a few other data structures that are used to keep file information in memory. There is a *directory name lookup* cache that speeds up the translation of a pathname into an inode number. The cache is indexed by a pair consisting of the inode number of a directory and a character string of a component within the directory to yield the inode number of the component. If you're lucky, you'll find the inode number of `/usr/local/bin/emacs` by four quick trips to the directory name lookup cache instead of a half dozen or more disk reads.

There is also a *file attribute cache* which remembers characteristics of recently accessed files. Unix programs frequently look up file attributes, and consequently this cache has proven to make a big difference in the performance of the Network File System. Although caches make remote file access faster, they can lead to annoying inconsistences. For example, if you use an editor to change a file's data or `chmod` to change a file's access or `rm` to disavow the file's existence while logged into an NFS server, clients may continue to hold on to their old view of the file. To reduce the occurrences of these problems, the Network File System uses timers to expire information.

The Network File System has some other imaginative solutions to the problems of maintaining traditional Unix file semantics in a networked environment. For example, in Unix when you unlink (remove) a file that is open, the kernel removes the file's name from the directory hierarchy but retains the file's data until the file is closed. If a user application running on an NFS client unlinks an open file, the client kernel instructs the server to rename the file to some rather odd name that is unlikely to be used by any user program. When the user application finally closes the file, the client kernel tells the server to really remove the recently renamed file. For a more detailed discussion of this and other subterfuges practiced by NFS, consult the article by Sandberg *et al* in the Summer 1985 USENIX proceedings.

Networking

We're going to take a rather abstract look at the implementation of networking within the Unix kernel. There are two reasons for this. First, the details of networking are found in the protocol definitions. These definitions specify the format and meaning of the packets transmitted between machines. There are many excellent *and lengthy* sources of information about protocols, and we do not need or wish to include yet another description of the seven layer OSI model in this chapter. The popular protocols for Unix operating systems follow the ARPANET Reference Model. While this protocol stack is not as deep as the OSI model, a remote login between two Unix computers connected by an Ethernet still involve four levels of protocols:

(1) application layer: data exchanged between the user programs,
(2) transport layer: the Transmission Control Protocol,
(3) network layer: the Internet Protocol, and
(4) physical layer: the Ethernet hardware and cabling.

The operating system is responsible for implementing the middle two layers of the stack. TCP, the Transmission Control Protocol, manages the sessions that connect user programs on different machines. IP, the Internet Protocol, is able to route packets across many diverse physical networks. If you want to find out more about these protocols, take a look at the recently-released book by Stevens entitled *TCP/IP Illustrated*.

The operating system must also provide an interface between the network layer and the physical layer. This interface is more or less a souped-up device driver and is sometimes referred to as the *link layer* in the BSD dialect. Finally the operating system provides the interface between the application layer and the transport layer. In BSD-based Unix, this is implemented as a collection of system calls using *sockets*. In System V Unix, a special Transport Layer Interface, or TLI, is defined.

The existence of both the TLI and the socket interface is the second reason why we are avoiding the details in our discussion of networking. This is one area in which BSD and System V based implementations of Unix can differ greatly. The layering that can be obtained using System V streams (or STREAMS, if you wish to shout) is very useful for implementing network protocols. Write a streams module for each protocol and just stack them up. The streams device driver interfaces with the hardware. The TLI is actually a programmer-friendly collection of routines that manipulate the streams head at the top of the stack.

Although the System V implementation should be commended for its use of an elegant kernel facility to implement networking, by the time it appeared many important network applications were already written using the BSD socket interface. The socket interface continues to be the most popular choice for Unix network programming and, for this reason, we will concentrate on it for the remainder of this section.

A *socket* is an endpoint of communication. When applications are connected by a TCP session or *stream* (an overused word in Unix as you are beginning to notice), applications exchange messages by reading and writing sockets. To the programmer, the socket looks very similar to a file. The `socket` system call creates a new socket and returns a file descriptor that refers to the socket, similar to how the open call returns a file descriptor that refers to the file. Reads and writes to a socket have the same syntax as reads and writes to a file. Inside the kernel, the socket's file descriptor points to a file table entry, but now the file table entry points to a socket structure (`struct socket`) rather than a vnode.

The kernel's socket structure is where the applications and protocols meet. You should find the socket structure defined in `/usr/include/sys/socketvar.h`. The socket structure has many fields, such as pointers to queues for data waiting to be read by the applications, for data waiting to be transmitted by the protocols, and even for connections waiting to be accepted. The socket structure is also the beginning of a chain of kernel data structures used to implement the relevant network protocols.

Protocols are controlled through two major data structures, the *protocol switch table* and the *protocol control blocks*. The protocol switch table points to a dozen or so modules that implement the protocol. The kernel calls the appropriate modules as data passes up and down the protocol stack. The protocol control block, or PCB, points to all the information required to manage the protocol. The format of the protocol control block varies according to the needs of

the underlying protocol. For example, the TCP control block contains information required to generate acknowledgments for received data, while the IP control block points to information required to route messages to the appropriate hardware interfaces. Many protocols actually use two different kinds of protocol control blocks, a single *per-protocol* PCB for global management of the protocol and several PCB's with specialized information for a single connection or remote machine.

At the bottom of the kernel's protocol chain is the network interface structure. This structure contains a switch table of procedures that are called to get data in and out of the hardware interface. The network interface structure also points to an output queue of messages waiting for their chance to appear on the external network.

Most kernel data structures, such as file table entries or disk data blocks, have a fixed size. This greatly simplifies memory allocation. However, the messages generated in networking have unpredictable sizes. Even worse, while going down the protocol stack, messages grow as each protocol places its own header on the front of the message, and, while going up the stack, messages shrink. The efficient implementation of networking within the kernel required a new method of memory allocation. In BSD Unix, the *mbuf*, or memory buffer, was introduced for this purpose. The mbuf is a rather intricate data structure which kernel programmers access through a well-designed set of routines. A single mbuf is small, less than 128 bytes longs, but an mbuf can point to a much larger data area and several mbufs be chained together to hold large messages. The mbuf is used throughout the kernel implementation of networking. Messages are passed between protocols in mbufs and most data structures, such as routing tables and protocol control blocks, are stored in mbufs.

Booting the Kernel

We end our survey of the Unix kernel with a look at how the kernel starts. Booting the kernel starts with some very machine specific actions required to load an initial *bootstrap* program. Generally, a computer has a special console monitor which knows how to load a few thousand bytes of data by reading special *boot blocks* from a disk or by downloading a file from a network server. This bootstrap may be able to directly load the kernel into memory, or it may only load yet another bootstrap, such as the one stored in `/osf_boot` on systems running DEC OSF/1, which loads the kernel.

Once loaded, the kernel executes a special *cold start* routine that constructs the foundation needed to call the kernel's main C routine. The kernel then performs various initialization activities such as allocating memory, setting values, and threading pointers between its major data structures. Many other tasks must also be completed. The root file system must be mounted and the two initial kernel-only processes, the swapper and the page daemon, are created.

About this time, the kernel will also try to determine the status of the devices attached to the computer. This is done by executing special *probe* routines included with device drivers. As each device is discovered, the associated device driver is ordered to configure itself. As your Unix computer boots, you should see the kernel print a brief message each time it discovers and configures a device.

The kernel probes only for devices whose drivers have been linked into the kernel by the system administrator. There are a few versions of Unix, such as NeXTSTEP, that actually support the loading of additional modules into a running kernel. On these systems, the loading and initialization of a device driver can be deferred to long after the kernel boots.

The first true user process run by the kernel is process number 1, `init`. The `init` process is ultimately responsible for all other aspects of system installation, such as mounting file systems and starting network daemons. Unlike those other two initial processes, the swapper and the page daemon, `init` does have a user side. Most Unix kernel have a clever way to starting init. Hidden inside the kernel is code for a program that makes a single system call:

```
execve("/etc/init", init, boot arguments ... );
```

which, *if* executed in user mode, replaces the code of that user program with the code stored in `/etc/init`. The kernel loads the one system call program into a text area allocated for process 1 and marks process 1 as runnable. Once `/etc/init` is loaded, it finishes the job of bringing the system into multi-user mode.

There are some significant differences between the BSD and System V versions of `init`. The BSD `init` executes the shell script stored in `/etc/rc` to complete the boot while the System V `init` executes a series of programs specified in `/etc/inittab`. However, this isn't our concern. This is an *internals* chapter, and now we're talking about a user-mode process. Turn to the System Administration and Network Administration chapters to follow the further adventures of our number one process.

Learning More

AT&T. *STREAMS Primer*. Englewood Cliffs, N.J.: Prentice-Hall, 1987.

AT&T. *STREAMS Programmer's Reference Guide*. Englewood Cliffs, N.J.: Prentice-Hall, 1989.

Bach, Maurice J. *The Design of the Unix Operating System*. Englewood Cliffs, N.J.: Prentice-Hall, 1986.

Deitel, Harvey M. *An Introduction to Operating Systems*. 2nd ed. Reading, MA: Addison-Wesley, 1990.

Egan, Janet I., Thomas J. Teixeira. *Writing a UNIX device driver*. New York: Wiley, 1988.

Gingell, Robert A., Joseph P. Moran, William A. Shannon. “Virtual Memory Architecture in SunOS”. *USENIX Conference Proceedings*, Summer 1987.

Leffler, Samuel J., Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD Unix Operating System*. Reading, MA: Addison-Wesley, 1989.

Lewine, Donald. *POSIX Programmer's Guide*. Sebastopol, CA: O'Reilly & Assts, 1991.

Sandberg, R., D. Goldman, S. Kleiman, D. Walsh, B. Lyon. “Design and implementation of the Sun Network File System”. *USENIX Conference Proceedings*, Summer 1985.

Silberschatz, Abraham, and Peter B. Galvin. *Operating System Concepts*. Reading, MA: Addison-Wesley, 1994.

Stevens, W. Richard. *TCP/IP Illustrated, Volume 1: The Protocols*. Reading, MA: Addison-Wesley, 1993.