1

GENERAL OVERVIEW OF THE SYSTEM

The UNIX system has become quite popular since its inception in 1969, running on machines of varying processing power from microprocessors to mainframes and providing a common execution environment across them. The system is divided into two parts. The first part consists of programs and services that have made the UNIX system environment so popular; it is the part readily apparent to users, including such programs as the shell, mail, text processing packages, and source code control systems. The second part consists of the operating system that supports these programs and services. This book gives a detailed description of the operating system. It concentrates on a description of UNIX System V produced by AT&T but considers interesting features provided by other versions too. It examines the major data structures and algorithms used in the operating system that ultimately provide users with the standard user interface.

This chapter provides an introduction to the UNIX system. It reviews its history and outlines the overall system structure. The next chapter gives a more detailed introduction to the operating system.

1.1 HISTORY

In 1965, Bell Telephone Laboratories joined an effort with the General Electric Company and Project MAC of the Massachusetts Institute of Technology to develop a new operating system called Multics [Organick 72] The goals of the Multics system were to provide simultaneous computer access to a large community of users, to supply ample computation power and data storage, and to allow users to share their data easily, if desired. Many people who later took part in the early development of the UNIX system participated in the Multics work at Bell Laboratories. Although a primitive version of the Multics system was running on a GE 645 computer by 1969, it did not provide the general service computing for which it was intended, nor was it clear when its development goals would be met. Consequently, Bell Laboratories ended its participation in the project.

With the end of their work on the Multics project, members of the Computing Science Research Center at Bell Laboratories were left without a "convenient interactive computing service" [Ritchie 84a]. In an attempt to improve their programming environment, Ken Thompson, Dennis Ritchie, and others sketched a paper design of a file system that later evolved into an early version of the UNIX file system. Thompson wrote programs that simulated the behavior of the proposed file system and of programs in a demand-paging environment, and he even encoded a simple kernel for the GE 645 computer. At the same time, he wrote a game program, "Space Travel," in Fortran for a GECOS system (the Honeywell 635), but the program was unsatisfactory because it was difficult to control the "space ship" and the program was expensive to run. Thompson later found a little-used PDP-7 computer that provided good graphic display and cheap executing power. Programming "Space Travel" for the PDP-7 enabled Thompson to learn about the machine, but its environment for program development required cross-assembly of the program on the GECOS machine and carrying paper tape for input to the PDP-7. To create a better development environment, Thompson and Ritchie implemented their system design on the PDP-7, including an early version of the UNIX file system, the process subsystem, and a small set of utility programs. Eventually, the new system no longer needed the GECOS system as a development environment but could support itself. The new system was given the name UNIX, a pun on the name Multics coined by another member of the Computing Science Research Center, Brian Kernighan.

Although this early version of the UNIX system held much promise, it could not realize its potential until it was used in a real project. Thus, while providing a text processing system for the patent department at Bell Laboratories, the UNIX system was moved to a PDP-11 in 1971. The system was characterized by its small size: 16K bytes for the system, 8K bytes for user programs, a disk of 512K bytes, and a limit of 64K bytes per file. After its early success, Thompson set out to implement a Fortran compiler for the new system, but instead came up with the language B, influenced by BCPL [Richards 69]. B was an interpretive language with the performance drawbacks implied by such languages, so Ritchie developed it into one he called C, allowing generation of machine code, declaration of data types, and definition of data structures. In 1973, the operating system was rewritten in C, an unheard of step at the time, but one that was to have tremendous impact on its acceptance among outside users. The number of installations at Bell

Laboratories grew to about 25, and a UNIX Systems Group was formed to provide internal support.

At this time, AT&T could not market computer products because of a 1956 Consent Decree it had signed with the Federal government, but it provided the UNIX system to universities who requested it for educational purposes. AT&T neither advertised, marketed, nor supported the system, in adherence to the terms of the Consent Decree. Nevertheless, the system's popularity steadily increased. In 1974, Thompson and Ritchie published a paper describing the UNIX system in the Communications of the ACM [Thompson 74], giving further impetus to its acceptance. By 1977, the number of UNIX system sites had grown to about 500, of which 125 were in universities. UNIX systems became popular in the operating telephone companies, providing a good environment for program development, network transaction operations services, and real-time services (via MERT [Lycklama 78a]). Licenses of UNIX systems were provided to commercial institutions as well as universities. In 1977, Interactive Systems Corporation became the first Value Added Reseller (VAR)[1] of a UNIX system, enhancing it for use in office automation environments. 1977 also marked the year that the UNIX system was first "ported" to a non-PDP machine (that is, made to run on another machine with few or no changes), the Interdata 8/32.

With the growing popularity of microprocessors, other companies ported the UNIX system to new machines, but its simplicity and clarity tempted many developers to enhance it in their own way, resulting in several variants of the basic system. In the period from 1977 to 1982, Bell Laboratories combined several AT&T variants into a single system, known commercially as UNIX System III. Bell Laboratories later added several features to UNIX System III, calling the new product UNIX System V,[2] and AT&T announced official support for System V in January 1983. However, people at the University of California at Berkeley had developed a variant to the UNIX system, the most recent version of which is called 4.3 BSD for VAX machines, providing some new, interesting features. This book will concentrate on the description of UNIX System V and will occasionally talk about features provided in the BSD system.

By the beginning of 1984, there were about 100,000 UNIX system installations in the world, running on machines with a wide range of computing power from microprocessors to mainframes and on machines across different manufacturers' product lines. No other operating system can make that claim. Several reasons have been suggested for the popularity and success of the UNIX system.

1. Value Added Resellers add specific applications to a computer system to satisfy a particular market. They market the applications rather than the operating system upon which they run.
2. What happened to System IV? An internal version of the system evolved into System V.

- The system is written in a high-level language, making it easy to read, understand, change, and move to other machines. Ritchie estimates that the first system in C was 20 to 40 percent larger and slower because it was not written in assembly language, but the advantages of using a higher-level language far outweigh the disadvantages (see page 1965 of [Ritchie 78b]).
- It has a simple user interface that has the power to provide the services that users want.
- It provides primitives that permit complex programs to be built from simpler programs.
- It uses a hierarchical file system that allows easy maintenance and efficient implementation.
- It uses a consistent format for files, the byte stream, making application programs easier to write.
- It provides a simple, consistent interface to peripheral devices.
- It is a multi-user, multiprocess system; each user can execute several processes simultaneously.
- It hides the machine architecture from the user, making it easier to write programs that run on different hardware implementations.

The philosophy of simplicity and consistency underscores the UNIX system and accounts for many of the reasons cited above.

Although the operating system and many of the command programs are written in C, UNIX systems support other languages, including Fortran, Basic, Pascal, Ada, Cobol, Lisp, and Prolog. The UNIX system can support any language that has a compiler or interpreter and a system interface that maps user requests for operating system services to the standard set of requests used on UNIX systems.

1.2 SYSTEM STRUCTURE

Figure 1.1 depicts the high-level architecture of the UNIX system. The hardware at the center of the diagram provides the operating system with basic services that will be described in Section 1.5. The operating system interacts directly[3] with the hardware, providing common services to programs and insulating them from hardware idiosyncrasies. Viewing the system as a set of layers, the operating system is commonly called the *system kernel*, or just the kernel, emphasizing its

3. In some implementations of the UNIX system, the operating system interacts with a native operating system that, in turn, interacts with the underlying hardware and provides necessary services to the system. Such configurations allow installations to run other operating systems and their applications in parallel to the UNIX system. The classic example of such a configuration is the MERT system [Lycklama 78a]. More recent configurations include implementations for IBM System/370 computers [Felton 84] and for UNIVAC 1100 Series computers [Bodenstab 84].

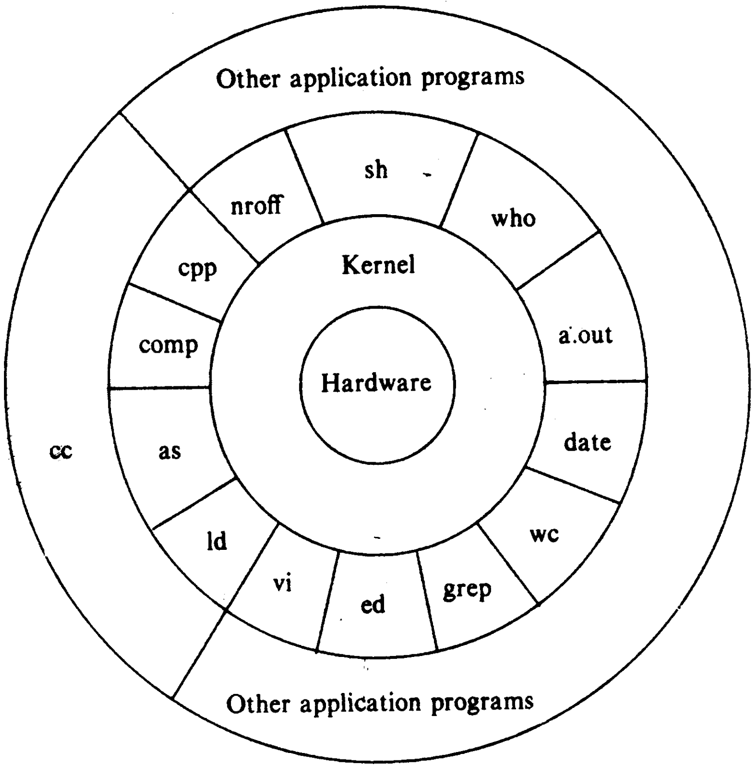


Figure 1.1. Architecture of UNIX Systems

isolation from user programs. Because programs are independent of the underlying hardware, it is easy to move them between UNIX systems running on different hardware if the programs do not make assumptions about the underlying hardware. For instance, programs that assume the size of a machine word are more difficult to move to other machines than programs that do not make this assumption.

Programs such as the shell and editors (*ed* and *vi*) shown in the outer layers interact with the kernel by invoking a well defined set of *system calls*. The system calls instruct the kernel to do various operations for the calling program and exchange data between the kernel and the program. Several programs shown in the figure are in standard system configurations and are known as *commands*, but private user programs may also exist in this layer as indicated by the program whose name is *a.out*, the standard name for executable files produced by the C compiler. Other application programs can build on top of lower-level programs, hence the existence of the outermost layer in the figure. For example, the standard C compiler, *cc*, is in the outermost layer of the figure: it invokes a C preprocessor two-pass compiler, assembler, and loader (link-editor), all separate lower-level programs. Although the figure depicts a two-level hierarchy of application programs, users can extend the hierarchy to whatever levels are appropriate. Indeed, the style of programming favored by the UNIX system encourages the combination of existing programs to accomplish a task.

Many application subsystems and programs that provide a high-level view of the system such as the shell, editors, SCCS (Source Code Control System), and document preparation packages, have gradually become synonymous with the name "UNIX system." However, they all use lower-level services ultimately provided by the kernel, and they avail themselves of these services via the set of system calls. There are about 64 system calls in System V, of which fewer than 32 are used frequently. They have simple options that make them easy to use but provide the user with a lot of power. The set of system calls and the internal algorithms that implement them form the body of the kernel, and the study of the UNIX operating system presented in this book reduces to a detailed study and analysis of the system calls and their interaction with one another. In short, the kernel provides the services upon which all application programs in the UNIX system rely, and it defines those services. This book will frequently use the terms "UNIX system," "kernel," or "system," but the intent is to refer to the kernel of the UNIX operating system and should be clear in context.

1.3 USER PERSPECTIVE

This section briefly reviews high-level features of the UNIX system such as the file system, the processing environment, and building block primitives (for example, *pipes*). Later chapters will explore kernel support of these features in detail.

1.3.1 The File System

The UNIX file system is characterized by

- a hierarchical structure,
- consistent treatment of file data,
- the ability to create and delete files,
- dynamic growth of files,
- the protection of file data,
- the treatment of peripheral devices (such as terminals and tape units) as files.

The file system is organized as a tree with a single root node called *root* (written "/"); every non-leaf node of the file system structure is a *directory* of files, and files at the leaf nodes of the tree are either directories, *regular files*, or *special* device files. The name of a file is given by a *path name* that describes how to locate the file in the file system hierarchy. A path name is a sequence of component names separated by slash characters; a component is a sequence of characters that

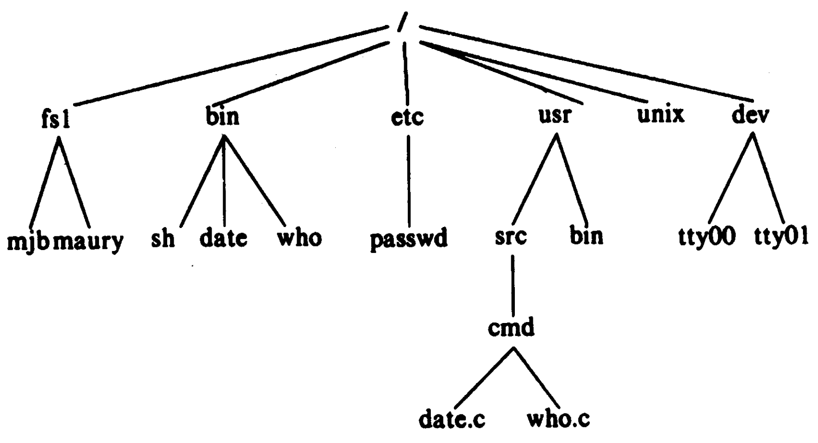


Figure 1.2. Sample File System Tree

designates a file name that is uniquely contained in the previous (directory) component. A full path name starts with a slash character and specifies a file that can be found by starting at the file system root and traversing the file tree, following the branches that lead to successive component names of the path name. Thus, the path names "/etc/passwd", "/bin/who", and "/usr/src/cmd/who.c" designate files in the tree shown in Figure 1.2, but "/bin/passwd" and "/usr/src/date.c" do not. A path name does not have to start from root but can be designated relative to the *current directory* of an executing process, by omitting the initial slash in the path name. Thus, starting from directory "/dev", the path name "tty01" designates the file whose full path name is "/dev/tty01".

Programs in the UNIX system have no knowledge of the internal format in which the kernel stores file data, treating the data as an unformatted stream of bytes. Programs may interpret the byte stream as they wish, but the interpretation has no bearing on how the operating system stores the data. Thus, the syntax of accessing the data in a file is defined by the system and is identical for all programs, but the semantics of the data are imposed by the program. For example, the text formatting program *troff* expects to find "new-line" characters at the end of each line of text, and the system accounting program *acctcom* expects to find fixed length records. Both programs use the same system services to access the data in the file as a byte stream, and internally, they parse the stream into a suitable format. If either program discovers that the format is incorrect, it is responsible for taking the appropriate action.

Directories are like regular files in this respect; the system treats the data in a directory as a byte stream, but the data contains the names of the files in the directory in a predictable format so that the operating system and programs such as *ls* (list the names and attributes of files) can discover the files in a directory.

Permission to access a file is controlled by *access permissions* associated with the file. Access permissions can be set independently to control read, write, and execute permission for three classes of users: the file owner, a file group, and everyone else. Users may create files if directory access permissions allow it. The newly created files are leaf nodes of the file system directory structure.

To the user, the UNIX system treats devices as if they were files. Devices, designated by special device files, occupy node positions in the file system directory structure. Programs access devices with the same syntax they use when accessing regular files; the semantics of reading and writing devices are to a large degree the same as reading and writing regular files. Devices are protected in the same way that regular files are protected: by proper setting of their (file) access permissions. Because device names look like the names of regular files and because the same operations work for devices and regular files, most programs do not have to know internally the types of files they manipulate.

For example, consider the C program in Figure 1.3, which makes a new copy of an existing file. Suppose the name of the executable version of the program is *copy*. A user at a terminal invokes the program by typing

```
copy oldfile newfile
```

where *oldfile* is the name of the existing file and *newfile* is the name of the new file. The system invokes *main*, supplying *argc* as the number of parameters in the list *argv*, and initializing each member of the array *argv* to point to a user-supplied parameter. In the example above, *argc* is 3, *argv[0]* points to the character string *copy* (the program name is conventionally the 0th parameter), *argv[1]* points to the character string *oldfile*, and *argv[2]* points to the character string *newfile*. The program then checks that it has been invoked with the proper number of parameters. If so, it invokes the *open* system call "read-only" for the file *oldfile*, and if the system call succeeds, invokes the *creat* system call to create *newfile*. The permission modes on the newly created file will be 0666 (octal), allowing all users access to the file for reading and writing. All system calls return −1 on failure; if the *open* or *creat* calls fail, the program prints a message and calls the *exit* system call with return status 1, terminating its execution and indicating that something went wrong.

The *open* and *creat* system calls return an integer called a *file descriptor*, which the program uses for subsequent references to the files. The program then calls the subroutine *copy*, which goes into a loop, invoking the *read* system call to read a buffer's worth of characters from the existing file, and invoking the *write* system call to write the data to the new file. The *read* system call returns the number of bytes read, returning 0 when it reaches the end of file. The program finishes the loop when it encounters the end of file, or when there is some error on the *read* system call (it does not check for *write* errors). Then it returns from *copy* and *exits* with return status 0, indicating that the program completed successfully.

```
#include <fcntl.h>
char buffer[2048];
int version = 1;        /* Chapter 2 explains this */

main(argc, argv)
        int argc;
        char *argv[];
{
        int fdold, fdnew;

        if (argc != 3)
        {
                printf("need 2 arguments for copy program\n");
                exit(1);
        }
        fdold = open(argv[1], O_RDONLY);   /* open source file read only */
        if (fdold == -1)
        {
                printf("cannot open file %s\n", argv[1]);
                exit(1);
        }
        fdnew = creat(argv[2], 0666);   /* create target file rw for all */
        if (fdnew == -1)
        {
                printf("cannot create file %s\n", argv[2]);
                exit(1);
        }
        copy(fdold, fdnew);
        exit(0);
}

copy(old, new)
        int old, new;
{
        int count;

        while ((count = read(old, buffer, sizeof(buffer))) > 0)
                write(new, buffer, count);
}
```

Figure 1.3. Program to Copy a File

The program copies any files supplied to it as arguments, provided it has permission to *open* the existing file and permission to create the new file. The file can be a file of printable characters, such as the source code for the program, or it can contain unprintable characters, even the program itself. Thus, the two invocations

```
copy copy.c newcopy.c
copy copy newcopy
```

both work. The old file can also be a directory. For instance,

```
copy    dircontents
```

copies the contents of the current directory, denoted by the name ".", to a regular file, "dircontents"; the data in the new file is identical, byte for byte, to the contents of the directory, but the file is a regular file. (The system call *mknod* creates a new directory.) Finally, either file can be a device special file. For example,

```
copy /dev/tty terminalread
```

reads the characters typed at the terminal (the special file */dev/tty* is the user's terminal) and copies them to the file *terminalread*, terminating only when the user types the character control-d. Similarly,

```
copy /dev/tty /dev/tty
```

reads characters typed at the terminal and copies them back.

1.3.2 Processing Environment

A *program* is an executable file, and a *process* is an instance of the program in execution. Many processes can execute simultaneously on UNIX systems (this feature is sometimes called multiprogramming or multitasking) with no logical limit to their number, and many instances of a program (such as *copy*) can exist simultaneously in the system. Various system calls allow processes to create new processes, terminate processes, synchronize stages of process execution, and control reaction to various events. Subject to their use of system calls, processes execute independently of each other.

For example, a process executing the program in Figure 1.4 executes the *fork* system call to create a new process. The new process, called the *child* process, gets a 0 return value from *fork* and invokes *execl* to execute the program *copy* (the program in Figure 1.3). The *execl* call overlays the address space of the child process with the file "copy", assumed to be in the current directory, and runs the program with the user-supplied parameters. If the *execl* call succeeds, it never returns because the process executes in a new address space, as will be seen in Chapter 7. Meanwhile, the process that had invoked *fork* (the parent) receives a non-0 return from the call, calls *wait*, suspending its execution until *copy* finishes, prints the message "copy done," and *exits* (every program *exits* at the end of its *main* function, as arranged by standard C program libraries that are linked during the compilation process). For example, if the name of the executable program is *run*, and a user invokes the program by

```
main(argc, argv)
        int argc;
        char *argv[];
{
        /* assume 2 args:  source file and target file */
        if (fork() == 0)
                execl("copy", "copy", argv[1], argv[2], 0);
        wait((int *) 0);
        printf("copy done\n");
}
```

Figure 1.4. Program that Creates a New Process to Copy Files

```
run oldfile newfile
```

the process copies "oldfile" to "newfile" and prints out the message. Although this program adds little to the "copy" program, it exhibits four major system calls used for process control: *fork*, *exec*, *wait*, and, discreetly, *exit*.

Generally, the system calls allow users to write programs that do sophisticated operations, and as a result, the kernel of the UNIX system does not contain many functions that are part of the "kernel" in other systems. Such functions, including compilers and editors, are user-level programs in the UNIX system. The prime example of such a program is the *shell*, the command interpreter program that users typically execute after logging into the system. The shell interprets the first word of a *command line* as a *command* name: for many commands, the shell *fork*s and the child process *exec*s the command associated with the name, treating the remaining words on the command line as parameters to the command.

The shell allows three types of commands. First, a command can be an executable file that contains object code produced by compilation of source code (a C program for example). Second, a command can be an executable file that contains a sequence of shell command lines. Finally, a command can be an internal shell command (instead of an executable file). The internal commands make the shell a programming language in addition to a command interpreter and include commands for looping (*for-in-do-done* and *while-do-done*), commands for conditional execution (*if-then-else-fi*), a "case" statement command, a command to change the current directory of a process (*cd*), and several others. The shell syntax allows for pattern matching and parameter processing. Users execute commands without having to know their types.

The shell searches for commands in a given sequence of directories, changeable by user request per invocation of the shell. The shell usually executes a command synchronously, waiting for the command to terminate before reading the next command line. However, it also allows asynchronous execution, where it reads the next command line and executes it without waiting for the prior command to terminate. Commands executed asynchronously are said to execute in the *background*. For example, typing the command

```
who
```

causes the system to execute the program stored in the file */bin/who*,[4] which prints a list of people who are currently logged in to the system. While *who* executes, the shell waits for it to finish and then prompts the user for another command. By typing

```
who &
```

the system executes the program *who* in the background, and the shell is ready to accept another command immediately.

Every process executing in the UNIX system has an execution environment that includes a current directory. The current directory of a process is the start directory used for all path names that do not begin with the slash character. The user may execute the shell command *cd*, change directory, to move around the file system tree and change the current directory. The command line

```
cd /usr/src/uts
```

changes the shell's current directory to the directory "/usr/src/uts". The command line

```
cd ../..
```

changes the shell's current directory to the directory that is two nodes "closer" to the root node: the component ".." refers to the *parent directory* of the current directory.

Because the shell is a user program and not part of the kernel, it is easy to modify it and tailor it to a particular environment. For instance, users can use the C shell to provide a history mechanism and avoid retyping recently used commands, instead of the Bourne shell (named after its inventor, Steve Bourne), provided as part of the standard System V release. Or some users may be granted use only of a restricted shell, providing a scaled down version of the regular shell. The system can execute the various shells simultaneously. Users have the capability to execute many processes simultaneously, and processes can create other processes dynamically and synchronize their execution, if desired. These features provide users with a powerful execution environment. Although much of the power of the shell derives from its capabilities as a programming language and from its capabilities for pattern matching of arguments, this section concentrates on the process environment provided by the system via the shell. Other important shell

4. The directory "/bin" contains many useful commands and is usually included in the sequence of directories the shell searches.

features are beyond the scope of this book (see [Bourne 78] for a detailed description of the shell).

1.3.3 Building Block Primitives

As described earlier, the philosophy of the UNIX system is to provide operating system primitives that enable users to write small, modular programs that can be used as building blocks to build more complex programs. One such primitive visible to shell users is the capability to *redirect I/O*. Processes conventionally have access to three files: they read from their *standard input* file, write to their *standard output* file, and write error messages to their *standard error* file. Processes executing at a terminal typically use the terminal for these three files, but each may be "redirected" independently. For instance, the command line

```
ls
```

lists all files in the current directory on the standard output, but the command line

```
ls > output
```

redirects the standard output to the file called "output" in the current directory, using the *creat* system call mentioned above. Similarly, the command line

```
mail mjb < letter
```

*open*s the file "letter" for its standard intput and *mail*s its contents to the user named "mjb." Processes can redirect input and output simultaneously, as in

```
nroff −mm < doc1 > doc1.out 2> errors
```

where the text formatter *nroff* reads the input file *doc1*, redirects its standard output to the file *doc1.out*, and redirects error messages to the file *errors* (the notation "2>" means to redirect the output for file descriptor 2, conventionally the standard error). The programs *ls*, *mail*, and *nroff* do not know what file their standard input, standard output, or standard error will be; the shell recognizes the symbols "<", ">", and "2>" and sets up the standard input, standard output, and standard error appropriately before executing the processes.

The second building block primitive is the *pipe*, a mechanism that allows a stream of data to be passed between reader and writer processes. Processes can redirect their standard output to a pipe to be read by other processes that have redirected their standard input to come from the pipe. The data that the first processes write into the pipe is the input for the second processes. The second processes could also redirect their output, and so on, depending on programming need. Again, the processes need not know what type of file their standard output is; they work regardless of whether their standard output is a regular file, a pipe, or a device. When using the smaller programs as building blocks for a larger, more complex program, the programmer uses the pipe primitive and redirection of I/O to integrate the piece parts. Indeed, the system tacitly encourages such programming style so that new programs can work with existing programs.

For example, the program *grep* searches a set of files (parameters to grep) for a given pattern:

```
grep main a.c b.c c.c
```

searches the three files a.c, b.c, and c.c for lines containing the string "main" and prints the lines that it finds onto standard output. Sample output may be:

```
a.c: main(argc, argv)
c.c: /* here is the main loop in the program */
c.c: main()
```

The program *wc* with the option −l counts the number of lines in the standard input file. The command line

```
grep main a.c b.c c.c | wc −l
```

counts the number of lines in the files that contain the string "main"; the output from *grep* is "piped" directly into the *wc* command. For the previous sample output from *grep*, the output from the piped command is

```
3
```

The use of pipes frequently makes it unnecessary to create temporary files.

1.4 OPERATING SYSTEM SERVICES

Figure 1.1 depicts the kernel layer immediately below the layer of user application programs. The kernel performs various primitive operations on behalf of user processes to support the user interface described above. Among the services provided by the kernel are

- Controlling the execution of processes by allowing their creation, termination or suspension, and communication
- Scheduling processes fairly for execution on the CPU. Processes share the CPU in a *time-shared* manner: the CPU[5] executes a process, the kernel suspends it when its time quantum elapses, and the kernel schedules another process to execute. The kernel later reschedules the suspended process.
- Allocating main memory for an executing process. The kernel allows processes to share portions of their address space under certain conditions, but protects the private address space of a process from outside tampering. If the system runs low on free memory, the kernel frees memory by writing a process

5. Chapter 12 will consider multiprocessor systems; until then, assume a single processor model.

temporarily to secondary memory, called a *swap* device. If the kernel writes entire processes to a swap device, the implementation of the UNIX system is called a *swapping* system; if it writes pages of memory to a swap device, it is called a *paging* system.
- Allocating secondary memory for efficient storage and retrieval of user data. This service constitutes the file system. The kernel allocates secondary storage for user files, reclaims unused storage, structures the file system in a well understood manner, and protects user files from illegal access.
- Allowing processes controlled access to peripheral devices such as terminals, tape drives, disk drives, and network devices.

The kernel provides its services transparently. For example, it recognizes that a given file is a regular file or a device, but hides the distinction from user processes. Similarly, it formats data in a file for internal storage, but hides the internal format from user processes, returning an unformatted byte stream. Finally, it offers necessary services so that user-level processes can support the services they must provide, while omitting services that can be implemented at the user level. For example, the kernel supports the services that the shell needs to act as a command interpreter: It allows the shell to read terminal input, to spawn processes dynamically, to synchronize process execution, to create pipes, and to redirect I/O. Users can construct private versions of the shell to tailor their environments to their specifications without affecting other users. These programs use the same kernel services as the standard shell.

1.5 ASSUMPTIONS ABOUT HARDWARE

The execution of user processes on UNIX systems is divided into two levels: user and kernel. When a process executes a system call, the *execution mode* of the process changes from *user mode* to *kernel mode*: the operating system executes and attempts to service the user request, returning an error code if it fails. Even if the user makes no explicit requests for operating system services, the operating system still does bookkeeping operations that relate to the user process, handling interrupts, scheduling processes, managing memory, and so on. Many machine architectures (and their operating systems) support more levels than the two outlined here, but the two modes, user and kernel, are sufficient for UNIX systems.

The differences between the two modes are

- Processes in user mode can access their own instructions and data but not kernel instructions and data (or those of other processes). Processes in kernel mode, however, can access kernel and user addresses. For example, the virtual address space of a process may be divided between addresses that are accessible only in kernel mode and addresses that are accessible in either mode.
- Some machine instructions are privileged and result in an error when executed in user mode. For example, a machine may contain an instruction that manipulates the processor status register; processes executing in user mode

	Processes			
	A	B	C	D
Kernel Mode	K			K
User Mode		U	U	

Figure 1.5. Multiple Processes and Modes of Execution

should not have this capability.

Put simply, the hardware views the world in terms of kernel mode and user mode and does not distinguish among the many users executing programs in those modes. The operating system keeps internal records to distinguish the many processes executing on the system. Figure 1.5 shows the distinction: the kernel distinguishes between processes A, B, C, and D on the horizontal axis, and the hardware distinguishes the mode of execution on the vertical axis.

Although the system executes in one of two modes, the kernel runs on behalf of a user process. The kernel is not a separate set of processes that run in parallel to user processes, but it is part of each user process. The ensuing text will frequently refer to "the kernel" allocating resources or "the kernel" doing various operations, but what is meant is that a process executing in kernel mode allocates the resources or does the various operations. For example, the shell reads user terminal input via a system call: The kernel, executing on behalf of the shell process, controls the operation of the terminal and returns the typed characters to the shell. The shell then executes in user mode, interprets the character stream typed by the user, and does the specified set of actions, which may require invocation of other system calls.

1.5.1 Interrupts and Exceptions

The UNIX system allows devices such as I/O peripherals or the system clock to interrupt the CPU asynchronously. On receipt of the interrupt, the kernel saves its current *context* (a frozen image of what the process was doing), determines the cause of the interrupt, and services the interrupt. After the kernel services the interrupt, it restores its interrupted context and proceeds as if nothing had happened. The hardware usually prioritizes devices according to the order that interrupts should be handled: When the kernel services an interrupt, it *blocks* out lower priority interrupts but services higher priority interrupts.

(An exception condition refers to unexpected events caused by a process, such as addressing illegal memory, executing privileged instructions, dividing by zero, and so on.) They are distinct from interrupts, which are caused by events that are external to a process. Exceptions happen "in the middle" of the execution of an instruction, and the system attempts to restart the instruction after handling the exception; interrupts are considered to happen between the execution of two instructions, and the system continues with the next instruction after servicing the interrupt. The UNIX system uses one mechanism to handle interrupts and exception conditions.

1.5.2 Processor Execution Levels

The kernel must sometimes prevent the occurrence of interrupts during critical activity, which could result in corrupt data if interrupts were allowed. For instance, the kernel may not want to receive a disk interrupt while manipulating linked lists, because handling the interrupt could corrupt the pointers, as will be seen in the next chapter. Computers typically have a set of privileged instructions that set the processor execution level in the processor status word. Setting the processor execution level to certain values masks off interrupts from that level and lower levels, allowing only higher-level interrupts. Figure 1.6 shows a sample set of execution levels. If the kernel masks out disk interrupts, all interrupts except for clock interrupts and machine error interrupts are prevented. If it masks out software interrupts, all other interrupts may occur.

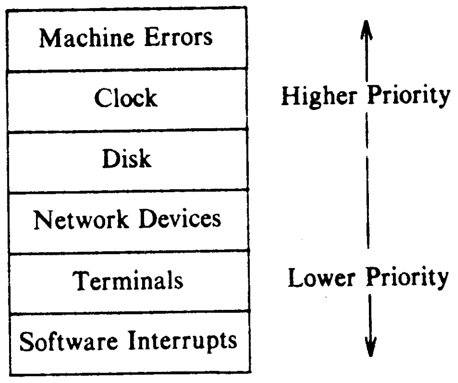


Figure 1.6. Typical Interrupt Levels

1.5.3 Memory Management

The kernel permanently resides in main memory as does the currently executing process (or parts of it, at least). When compiling a program, the compiler generates a set of addresses in the program that represent addresses of variables and data structures or the addresses of instructions such as functions. The compiler generates the addresses for a *virtual machine* as if no other program will execute simultaneously on the physical machine.

When the program is to run on the machine, the kernel allocates space in main memory for it, but the *virtual addresses* generated by the compiler need not be identical to the physical addresses that they occupy in the machine. The kernel coordinates with the machine hardware to set up a *virtual to physical* address translation that maps the compiler-generated addresses to the physical machine addresses. The mapping depends on the capabilities of the machine hardware, and the parts of UNIX systems that deal with them are therefore machine dependent. For example, some machines have special hardware to support demand paging. Chapters 6 and 9 will discuss issues of memory management and how they relate to hardware in more detail.

1.6 SUMMARY

This chapter has described the overall structure of the UNIX system, the relationship between processes running in user mode versus kernel mode, and the assumptions the kernel makes about the hardware. Processes execute in user mode or kernel mode, where they avail themselves of system services using a well-defined set of system calls. The system design encourages programmers to write small programs that do only a few operations but do them well, and then to combine the programs using *pipes* and I/O redirection to do more sophisticated processing.

The system calls allow processes to do operations that are otherwise forbidden to them. In addition to servicing system calls, the kernel does general bookkeeping for the user community, controlling process scheduling, managing the storage and protection of processes in main memory, fielding interrupts, managing files and devices, and taking care of system error conditions. The UNIX system kernel purposely omits many functions that are part of other operating systems, providing a small set of system calls that allow processes to do necessary functions at user level. The next chapter gives a more detailed introduction to the kernel, describing its architecture and some basic concepts used in its implementation.

2

INTRODUCTION TO THE KERNEL

The last chapter gave a high-level perspective of the UNIX system environment. This chapter focuses on the kernel, providing an overview of its architecture and outlining basic concepts and structures essential for understanding the rest of the book.

2.1 ARCHITECTURE OF THE UNIX OPERATING SYSTEM

It has been noted (see page 239 of [Christian 83]) that the UNIX system supports the illusions that the file system has "places" and that processes have "life." The two entities, files and processes, are the two central concepts in the UNIX system model. Figure 2.1 gives a block diagram of the kernel, showing various modules and their relationships to each other. In particular, it shows the file subsystem on the left and the process control subsystem on the right, the two major components of the kernel. The diagram serves as a useful logical view of the kernel, although in practice the kernel deviates from the model because some modules interact with the internal operations of others.

Figure 2.1 shows three levels: user, kernel, and hardware. The system call and library interface represent the border between user programs and the kernel depicted in Figure 1.1. System calls look like ordinary function calls in C programs, and libraries map these function calls to the primitives needed to enter

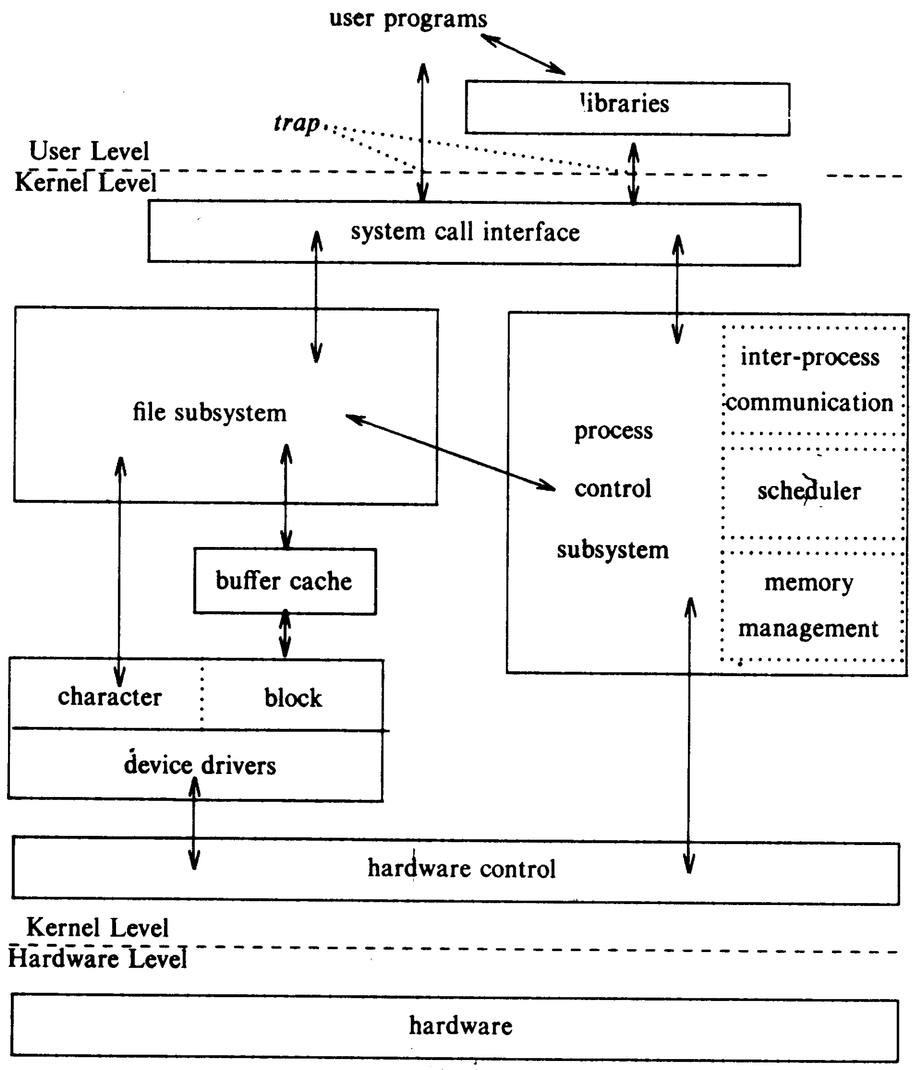


Figure 2.1. Block Diagram of the System Kernel

the operating system, as covered in more detail in Chapter 6. Assembly language programs may invoke system calls directly without a system call library, however. Programs frequently use other libraries such as the standard I/O library to provide a more sophisticated use of the system calls. The libraries are linked with the programs at compile time and are thus part of the user program for purposes of this discussion. An example later on will illustrate these points.

The figure partitions the set of system calls into those that interact with the file subsystem and those that interact with the process control subsystem. The file subsystem manages files, allocating file space, administering free space, controlling access to files, and retrieving data for users. Processes interact with the file subsystem via a specific set of system calls, such as *open* (to open a file for reading or writing), *close*, *read*, *write*, *stat* (query the attributes of a file), *chown* (change the record of who owns the file), and *chmod* (change the access permissions of a file). These and others will be examined in Chapter 5.

The file subsystem accesses file data using a buffering mechanism that regulates data flow between the kernel and secondary storage devices. The buffering mechanism interacts with block I/O device drivers to initiate data transfer to and from the kernel. Device drivers are the kernel modules that control the operation of peripheral devices. Block I/O devices are random access storage devices; alternatively, their device drivers make them appear to be random access storage devices to the rest of the system. For example, a tape driver may allow the kernel to treat a tape unit as a random access storage device. The file subsystem also interacts directly with "raw" I/O device drivers without the intervention of a buffering mechanism. Raw devices, sometimes called character devices, include all devices that are not block devices.

The process control subsystem is responsible for process synchronization, interprocess communication, memory management, and process scheduling. The file subsystem and the process control subsystem interact when loading a file into memory for execution, as will be seen in Chapter 7: the process subsystem reads executable files into memory before executing them.

Some of the system calls for controlling processes are *fork* (create a new process), *exec* (overlay the image of a program onto the running process), *exit* (finish executing a process), *wait* (synchronize process execution with the *exit* of a previously *fork*ed process), *brk* (control the size of memory allocated to a process), and *signal* (control process response to extraordinary events). Chapter 7 will examine these system calls and others.

The memory management module controls the allocation of memory. If at any time the system does not have enough physical memory for all processes, the kernel moves them between main memory and secondary memory so that all processes get a fair chance to execute. Chapter 9 will describe two policies for managing memory: swapping and demand paging. The swapper process is sometimes called the scheduler, because it "schedules" the allocation of memory for processes and influences the operation of the CPU scheduler. However, this text will refer to it as the swapper to avoid confusion with the CPU scheduler.

The *scheduler* module allocates the CPU to processes. It schedules them to run in turn until they voluntarily relinquish the CPU while awaiting a resource or until the kernel preempts them when their recent run time exceeds a time quantum. The scheduler then chooses the highest priority eligible process to run; the original process will run again when it is the highest priority eligible process available.

There are several forms of interprocess communication, ranging from asynchronous signaling of events to synchronous transmission of messages between processes.

Finally, the hardware control is responsible for handling interrupts and for communicating with the machine. Devices such as disks or terminals may interrupt the CPU while a process is executing. If so, the kernel may resume execution of the interrupted process after servicing the interrupt: Interrupts are *not* serviced by special processes but by special functions in the kernel, called in the context of the currently running process.

2.2 INTRODUCTION TO SYSTEM CONCEPTS

This section gives an overview of some major kernel data structures and describes the function of modules shown in Figure 2.1 in more detail.

2.2.1 An Overview of the File Subsystem

The internal representation of a file is given by an *inode*, which contains a description of the disk layout of the file data and other information such as the file owner, access permissions, and access times. The term inode is a contraction of the term *index node* and is commonly used in literature on the UNIX system. Every file has one inode, but it may have several names, all of which map into the inode. Each name is called a *link*. When a process refers to a file by name, the kernel parses the file name one component at a time, checks that the process has permission to search the directories in the path, and eventually retrieves the inode for the file. For example, if a process calls

```
open("/fs2/mjb/rje/sourcefile", 1);
```

the kernel retrieves the inode for "/fs2/mjb/rje/sourcefile". When a process creates a new file, the kernel assigns it an unused inode. Inodes are stored in the file system, as will be seen shortly, but the kernel reads them into an in-core[1] inode table when manipulating files.

The kernel contains two other data structures, the *file table* and the *user file descriptor table*. The file table is a global kernel structure, but the user file descriptor table is allocated per process. When a process *opens* or *creats* a file, the kernel allocates an entry from each table, corresponding to the file's inode. Entries in the three structures — user file descriptor table, file table, and inode table — maintain the state of the file and the user's access to it. The file table keeps track of the byte offset in the file where the user's next *read* or *write* will start, and the

1. The term *core* refers to primary memory of a machine, not to hardware technology.

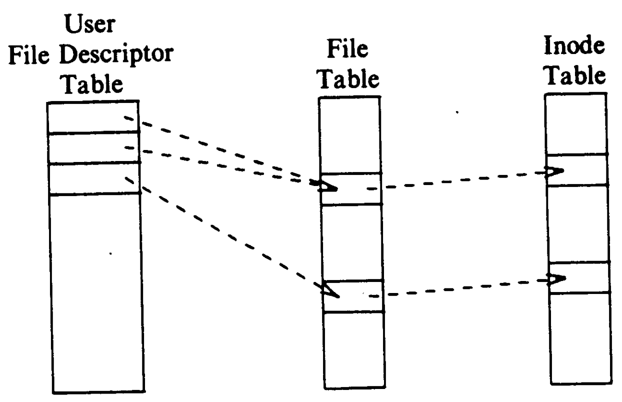


Figure 2.2. File Descriptors, File Table, and Inode Table

access rights allowed to the *open*ing process. The user file descriptor table identifies all open files for a process. Figure 2.2 shows the tables and their relationship to each other. The kernel returns a ***file descriptor*** for the *open* and *creat* system calls, which is an index into the user file descriptor table. When executing *read* and *write* system calls, the kernel uses the file descriptor to access the user file descriptor table, follows pointers to the file table and inode table entries, and, from the inode, finds the data in the file. Chapters 4 and 5 describe these data structures in great detail. For now, suffice it to say that use of three tables allows various degrees of sharing access to a file.

The UNIX system keeps regular files and directories on block devices such as tapes or disks. Because of the difference in access time between the two, few, if any, UNIX system installations use tapes for their file systems. In coming years, diskless work stations will be common, where files are located on a remote system and accessed via a network (see Chapter 13). For simplicity, however, the ensuing text assumes the use of disks. An installation may have several physical disk units, each containing one or more ***file systems***. Partitioning a disk into several file systems makes it easier for administrators to manage the data stored there. The kernel deals on a logical level with file systems rather than with disks, treating each one as a ***logical device*** identified by a logical ***device number***. The conversion between logical device (file system) addresses and physical device (disk) addresses is done by the disk driver. This book will use the term device to mean a logical device unless explicitly stated otherwise.

A file system consists of a sequence of logical blocks, each containing 512, 1024, 2048, or any convenient multiple of 512 bytes, depending on the system implementation. The size of a logical block is homogeneous within a file system but may vary between different file systems in a system configuration. Using large logical blocks increases the effective data transfer rate between disk and memory, because the kernel can transfer more data per disk operation and therefore make fewer time-consuming operations. For example, reading 1K bytes from a disk in one read operation is faster than reading 512 bytes twice. However, if a logical block is too large, effective storage capacity may drop, as will be shown in Chapter 5. For simplicity, this book will use the term "block" to mean a logical block, and it will assume that a logical block contains 1K bytes of data unless explicitly stated otherwise.

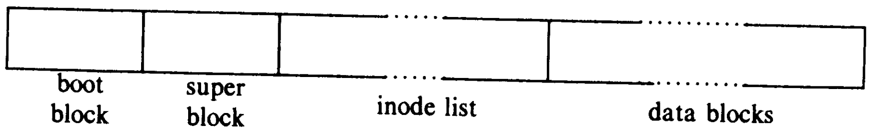


Figure 2.3. File System Layout

A file system has the following structure (Figure 2.3).

- The *boot block* occupies the beginning of a file system, typically the first sector, and may contain the *bootstrap* code that is read into the machine to *boot*, or initialize, the operating system. Although only one boot block is needed to boot the system, every file system has a (possibly empty) boot block.
- The *super block* describes the state of a file system — how large it is, how many files it can store, where to find free space on the file system, and other information.
- The *inode list* is a list of inodes that follows the super block in the file system. Administrators specify the size of the inode list when configuring a file system. The kernel references inodes by index into the inode list. One inode is the *root inode* of the file system: it is the inode by which the directory structure of the file system is accessible after execution of the *mount* system call (Section 5.14).
- The data blocks start at the end of the inode list and contain file data and administrative data. An allocated data block can belong to one and only one file in the file system.

2.2.2 Processes

This section examines the process subsystem more closely. It describes the structure of a process and some process data structures used for memory management. Then it gives a preliminary view of the process state diagram and considers various issues involved in some state transitions.

A process is the execution of a program and consists of a pattern of bytes that the CPU interprets as machine instructions (called "text"), data, and stack. Many processes appear to execute simultaneously as the kernel schedules them for execution, and several processes may be instances of one program. A process executes by following a strict sequence of instructions that is self-contained and does not jump to that of another process; it reads and writes its data and stack sections, but it cannot read or write the data and stack of other processes. Processes communicate with other processes and with the rest of the world via system calls.

In practical terms, a process on a UNIX system is the entity that is created by the *fork* system call. Every process except *process 0* is created when another process executes the *fork* system call. The process that invoked the *fork* system call is the *parent* process, and the newly created process is the *child* process. Every process has one parent process, but a process can have many child processes. The kernel identifies each process by its process number, called the *process ID* (PID). Process 0 is a special process that is created "by hand" when the system boots; after *fork*ing a child process (process 1), process 0 becomes the *swapper* process. Process 1, known as *init*, is the ancestor of every other process in the system and enjoys a special relationship with them, as explained in Chapter 7.

A user compiles the source code of a program to create an executable file, which consists of several parts:

- a set of "headers" that describe the attributes of the file,
- the program text,
- a machine language representation of data that has initial values when the program starts execution, and an indication of how much space the kernel should allocate for uninitialized data, called *bss*[2] (the kernel initializes it to 0 at run time),
- other sections, such as symbol table information.

For the program in Figure 1.3, the text of the executable file is the generated code for the functions *main* and *copy*, the initialized data is the variable *version* (put into the program just so that it should have some initialized data), and the uninitialized data is the array *buffer*. System V versions of the C compiler create a separate text section by default but support an option that allows inclusion of program instructions in the data section, used in older versions of the system.

The kernel loads an executable file into memory during an *exec* system call, and the loaded process consists of at least three parts, called *regions*: text, data, and the stack. The text and data regions correspond to the text and data-bss sections of the executable file, but the stack region is automatically created and its size is dynamically adjusted by the kernel at run time. The stack consists of logical *stack frames* that are *push*ed when calling a function and *pop*ped when returning; a special register called the *stack pointer* indicates the current stack depth. A stack

2. The name bss comes from an assembly pseudo-operator on the IBM 7090 machine, which stood for "block started by symbol."

frame contains the parameters to a function, its local variables, and the data necessary to recover the previous stack frame, including the value of the program counter and stack pointer at the time of the function call. The program code contains instruction sequences that manage stack growth, and the kernel allocates space for the stack, as needed. In the program in Figure 1.3, parameters *argc* and *argv* and variables *fdold* and *fdnew* in the function *main* appear on the stack when *main* is called (once in every program, by convention), and parameters *old* and *new* and the variable *count* in the function *copy* appear on the stack whenever *copy* is called.

Because a process in the UNIX system can execute in two modes, kernel or user, it uses a separate stack for each mode. The user stack contains the arguments, local variables, and other data for functions executing in user mode. The left side of Figure 2.4 shows the user stack for a process when it makes the *write* system call in the *copy* program. The process startup procedure (included in a library) had called the function *main* with two parameters, pushing frame 1 onto the user stack; frame 1 contains space for the two local variables of *main*. *Main* then called *copy* with two parameters, *old* and *new*, and pushed frame 2 onto the user stack; frame 2 contains space for the local variable *count*. Finally, the process invoked the system call *write* by invoking the library function *write*. Each system call has an entry point in a system call library; the system call library is encoded in assembly language and contains special *trap* instructions, which, when executed, cause an "interrupt" that results in a hardware switch to kernel mode. A process calls the library entry point for a particular system call just as it calls any function, creating a stack frame for the library function. When the process executes the special instruction, it switches mode to the kernel, executes kernel code, and uses the kernel stack.

The kernel stack contains the stack frames for functions executing in kernel mode. The function and data entries on the kernel stack refer to functions and data in the kernel, not the user program, but its construction is the same as that of the user stack. The kernel stack of a process is null when the process executes in user mode. The right side of Figure 2.4 depicts the kernel stack representation for a process executing the *write* system call in the *copy* program. The names of the algorithms are described during the detailed discussion of the *write* system call in later chapters.

Every process has an entry in the kernel *process table*, and each process is allocated a *u area*[3] that contains private data manipulated only by the kernel. The process table contains (or points to) a *per process region table*, whose entries point to entries in a *region* table. A region is a contiguous area of a process's address

3. The *u* in *u area* stands for "user." Another name for the *u area* is *u block*; this book will always refer to it as the *u area*.

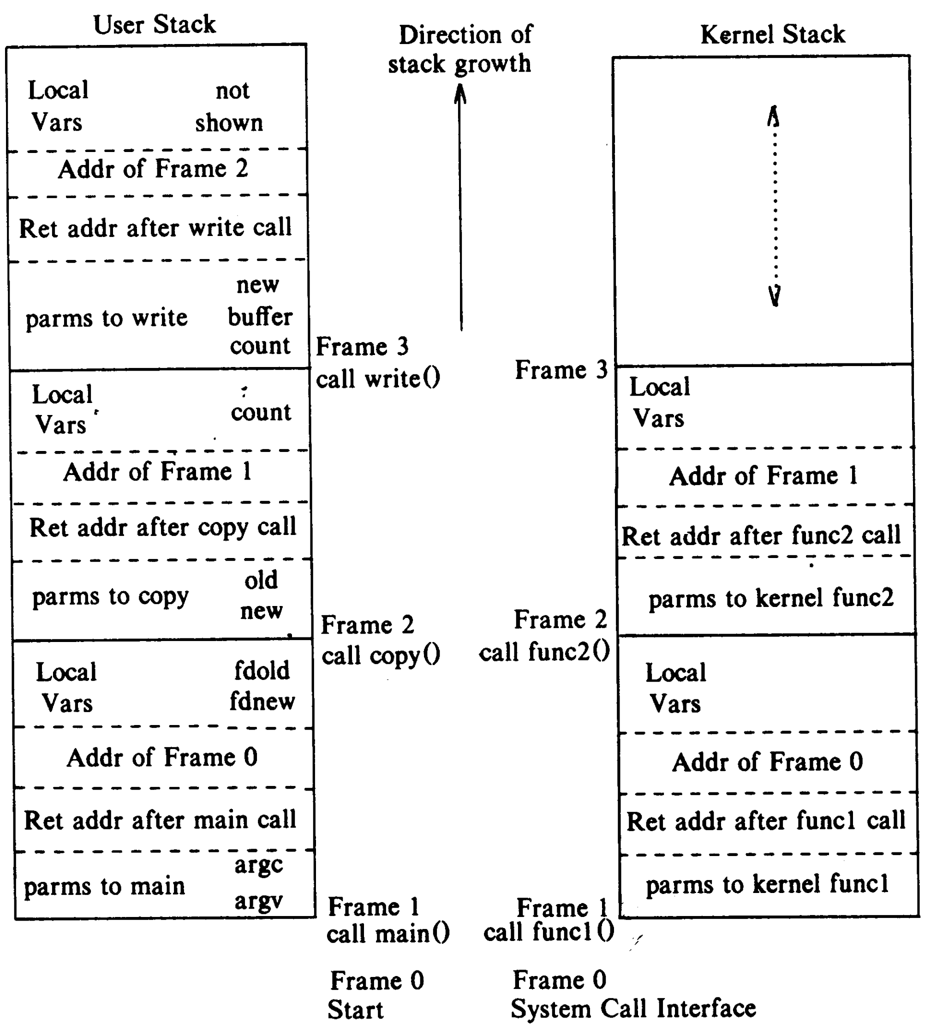


Figure 2.4. User and Kernel Stack for Copy Program

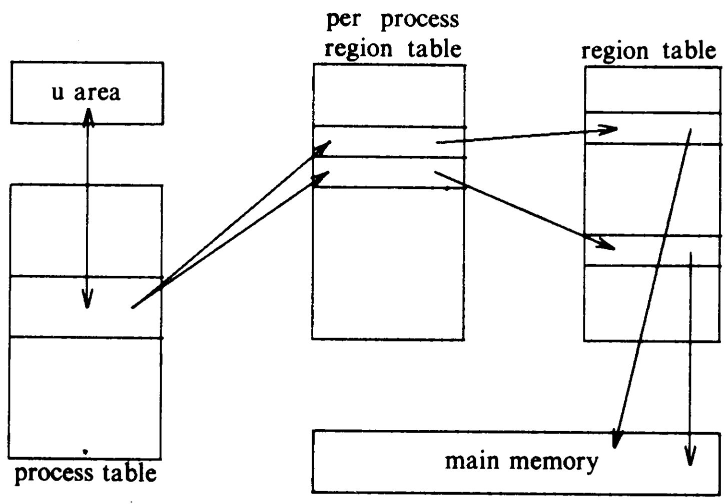


Figure 2.5. Data Structures for Processes

space, such as text, data, and stack. Region table entries describe the attributes of the region, such as whether it contains text or data, whether it is shared or private, and where the "data" of the region is located in memory. The extra level of indirection (from the per process region table to the region table) allows independent processes to share regions. When a process invokes the *exec* system call, the kernel allocates regions for its text, data, and stack after freeing the old regions the process had been using. When a process invokes *fork*, the kernel duplicates the address space of the old process, allowing processes to share regions when possible and making a physical copy otherwise. When a process invokes *exit*, the kernel frees the regions the process had used. Figure 2.5 shows the relevant data structures of a running process: The process table points to a per process region table with pointers to the region table entries for the text, data, and stack regions of the process.

The process table entry and the *u area* contain control and status information about the process. The *u area* is an extension of the process table entry, and Chapter 6 will examine the distinction between the two tables. Fields in the process table discussed in the following chapters are

- a state field,
- identifiers indicating the user who owns the process (user IDs, or UIDs),
- an event descriptor set when a process is suspended (in the *sleep* state).

The *u area* contains information describing the process that needs to be accessible only when the process is executing. The important fields are

- a pointer to the process table slot of the currently executing process,
- parameters of the current system call, return values and error codes,
- file descriptors for all open files,
- internal I/O parameters,
- current directory and current root (see Chapter 5),
- process and file size limits.

The kernel can directly access fields of the *u area* of the executing process but not of the *u area* of other processes. Internally, the kernel references the structure variable *u* to access the *u area* of the currently running process, and when another process executes, the kernel rearranges its virtual address space so that the structure *u* refers to the *u area* of the new process. The implementation gives the kernel an easy way to identify the current process by following the pointer from the *u area* to its process table entry.

2.2.2.1 Context of a process

The *context* of a process is its state, as defined by its text, the values of its global user variables and data structures, the values of machine registers it uses, the values stored in its process table slot and *u area*, and the contents of its user and kernel stacks. The text of the operating system and its global data structures are shared by all processes but do not constitute part of the context of a process.

When executing a process, the system is said to be executing in the context of the process. When the kernel decides that it should execute another process, it does a *context switch*, so that the system executes in the context of the other process. The kernel allows a context switch only under specific conditions, as will be seen. When doing a context switch, the kernel saves enough information so that it can later switch back to the first process and resume its execution. Similarly, when moving from user to kernel mode, the kernel saves enough information so that it can later return to user mode and continue execution from where it left off. Moving between user and kernel mode is a change in mode, not a context switch. Recalling Figure 1.5, the kernel does a context switch when it changes context from process A to process B; it changes execution mode from user to kernel or from kernel to user, still executing in the context of one process, such as process A.

The kernel services interrupts in the context of the interrupted process even though it may not have caused the interrupt. The interrupted process may have been executing in user mode or in kernel mode. The kernel saves enough information so that it can later resume execution of the interrupted process and services the interrupt in kernel mode. The kernel does not spawn or schedule a special process to handle interrupts.

2.2.2.2 Process states

The lifetime of a process can be divided into a set of *states*, each with certain characteristics that describe the process. Chapter 6 will describe all process states, but it is essential to understand the following states now:

1. The process is currently executing in user mode.
2. The process is currently executing in kernel mode.
3. The process is not executing, but it is ready to run as soon as the scheduler chooses it. Many processes may be in this state, and the scheduling algorithm determines which one will execute next.
4. The process is *sleeping*. A process puts itself to sleep when it can no longer continue executing, such as when it is waiting for I/O to complete.

Because a processor can execute only one process at a time, at most one process may be in states 1 and 2. The two states correspond to the two modes of execution, user and kernel.

2.2.2.3 State transitions

The process states described above give a static view of a process, but processes move continuously between the states according to well-defined rules. A *state transition* diagram is a directed graph whose *nodes* represent the states a process can enter and whose *edges* represent the events that cause a process to move from one state to another. State transitions are legal between two states if there exists an edge from the first state to the second. Several transitions may emanate from a state, but a process will follow one and only one transition depending on the system event that occurs. Figure 2.6 shows the state transition diagram for the process states defined above.

Several processes can execute simultaneously in a time-shared manner, as stated earlier, and they may all run simultaneously in kernel mode. If they were allowed to run in kernel mode without constraint, they could corrupt global kernel data structures. By prohibiting arbitrary context switches and controlling the occurrence of interrupts, the kernel protects its consistency.

The kernel allows a context switch only when a process moves from the state "kernel running" to the state "asleep in memory." Processes running in kernel mode cannot be preempted by other processes; therefore the kernel is sometimes said to be *non-preemptive*, although the system does preempt processes that are in user mode. The kernel maintains consistency of its data structures because it is non-preemptive, thereby solving the *mutual exclusion* problem — making sure that critical sections of code are executed by at most one process at a time.

For instance, consider the sample code in Figure 2.7 to put a data structure, whose address is in the pointer *bp1*, onto a doubly linked list after the structure whose address is in *bp*. If the system allowed a context switch while the kernel executed the code fragment, the following situation could occur. Suppose the

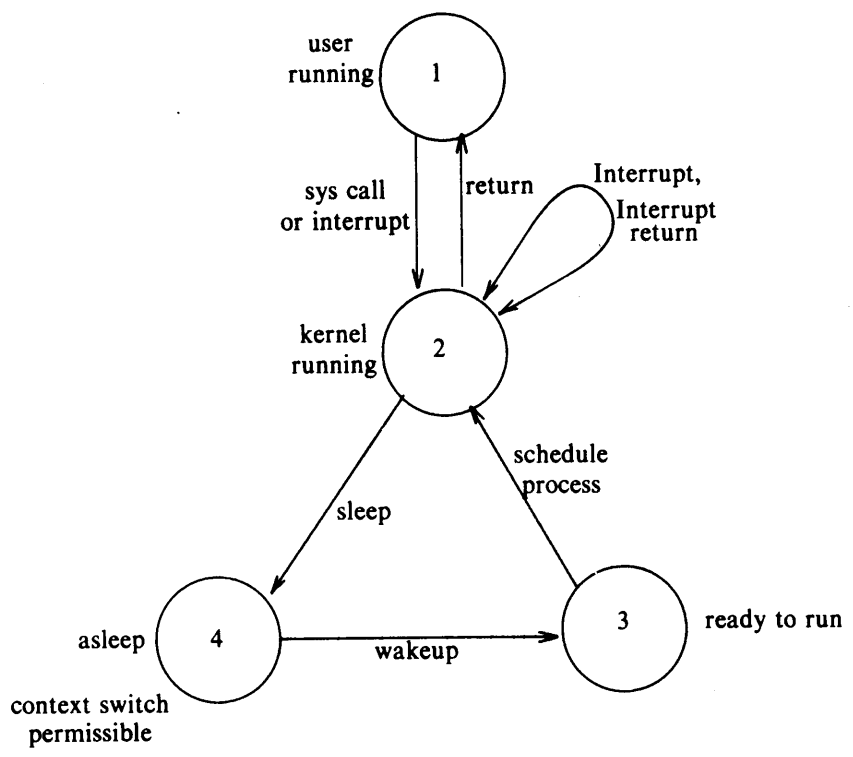


Figure 2.6. Process States and Transitions

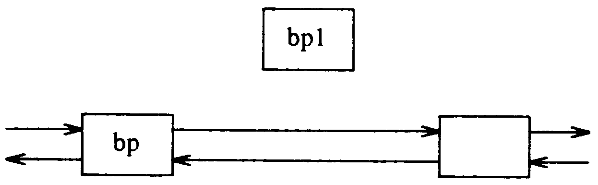
kernel executes the code until the comment and then does a context switch. The doubly linked list is in an inconsistent state: the structure *bp1* is half on and half off the linked list. If a process were to follow the forward pointers, it would find *bp1* on the linked list, but if it were to follow the back pointers, it would not find *bp1* (Figure 2.8). If other processes were to manipulate the pointers on the linked list before the original process ran again, the structure of the doubly linked list could be permanently destroyed. The UNIX system prevents such situations by disallowing context switches when a process executes in kernel mode. If a process goes to sleep, thereby permitting a context switch, kernel algorithms are encoded to make sure that system data structures are in a safe, consistent state.

A related problem that can cause inconsistency in kernel data is the handling of interrupts, which can change kernel state information. For example, if the kernel was executing the code in Figure 2.7 and received an interrupt when it reached the

```
struct queue {


} *bp, *bp1;
bp1->forp = bp->forp;
bp1->backp = bp;
bp->forp = bp1;
/* consider possible context switch here */
bp1->forp->backp = bp1;
```

Figure 2.7. Sample Code Creating Doubly Linked List



Placing bp1 on doubly linked list

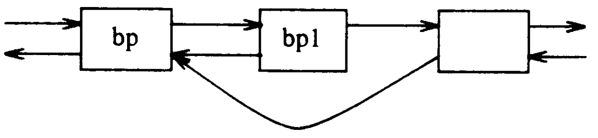


Figure 2.8. Incorrect Linked List because of Context Switch

comment, the interrupt handler could corrupt the links if it manipulates the pointers, as illustrated earlier. To solve this problem, the system could prevent all interrupts while executing in kernel mode, but that would delay servicing of the interrupt, possibly hurting system throughput. Instead, the kernel raises the processor execution level to prevent interrupts when entering *critical* regions of code. A section of code is critical if execution of arbitrary interrupt handlers could result in consistency problems. For example, if a disk interrupt handler manipulates the buffer queues in the figure, the section of code where the kernel manipulates the buffer queues is a critical region of code with respect to the disk interrupt handler. Critical regions are small and infrequent so that system throughput is largely unaffected by their existence. Other operating systems solve this problem by preventing all interrupts when executing in system states or by using elaborate locking schemes to ensure consistency. Chapter 12 will return to this issue for multiprocessor systems, where the solution outlined here is insufficient.

To review, the kernel protects its consistency by allowing a context switch only when a process puts itself to sleep and by preventing one process from changing the state of another process. It also raises the processor execution level around critical regions of code to prevent interrupts that could otherwise cause inconsistencies. The process scheduler periodically preempts processes executing in user mode so that processes cannot monopolize use of the CPU.

2.2.2.4 Sleep and wakeup

A process executing in kernel mode has great autonomy in deciding what it is going to do in reaction to system events. Processes can communicate with each other and "suggest" various alternatives, but they make the final decision by themselves. As will be seen, there is a set of rules that processes obey when confronted with various circumstances, but each process ultimately follows these rules under its own initiative. For instance, when a process must temporarily suspend its execution ("go to sleep"), it does so of its own free will. Consequently, an interrupt handler cannot go to sleep, because if it could, the interrupted process would be put to sleep by default.

Processes go to sleep because they are awaiting the occurrence of some event, such as waiting for I/O completion from a peripheral device, waiting for a process to exit, waiting for system resources to become available, and so on. Processes are said to *sleep on an event*, meaning that they are in the sleep state until the event occurs, at which time they wake up and enter the state "ready to run." Many processes can simultaneously sleep on an event; when an event occurs, *all* processes sleeping on the event wake up because the event condition is no longer true. When a process wakes up, it follows the state transition from the "sleep" state to the "ready-to-run" state, where it is eligible for later scheduling; it does *not* execute immediately. Sleeping processes do not consume CPU resources: The kernel does not constantly check to see that a process is still sleeping but waits for the event to occur and awakens the process then.

For example, a process executing in kernel mode must sometimes lock a data structure in case it goes to sleep at a later stage; processes attempting to manipulate the locked structure must check the lock and sleep if another process owns the lock. The kernel implements such locks in the following manner:

```
while (condition is true)
        sleep (event: the condition becomes false);
set condition true;
```

It unlocks the lock and awakens all processes asleep on the lock in the following manner:

```
set condition false;
wakeup (event: the condition is false);
```

Figure 2.9 depicts a scenario where three processes, A, B, and C, contend for a locked buffer. The sleep condition is that the buffer is locked. The processes execute one at a time, find the buffer locked, and sleep on the event that the buffer becomes unlocked. Eventually, the buffer is unlocked, and all processes wake up and enter the state "ready to run." The kernel eventually chooses one process, say B, to execute. Process B executes the "while" loop, finds that the buffer is unlocked, sets the buffer lock, and proceeds. If process B later goes to sleep again before unlocking the buffer (waiting for completion of an I/O operation, for example), the kernel can schedule other processes to run. If it chooses process A, process A executes the "while" loop, finds that the buffer is locked, and goes to sleep again; process C may do the same thing. Eventually, process B awakens and unlocks the buffer, allowing either process A or C to gain access to the buffer. Thus, the "while-sleep" loop insures that at most one process can gain access to a resource.

Chapter 6 will present the algorithms for sleep and wakeup in greater detail. In the meantime, they should be considered "atomic". A process enters the sleep state instantaneously and stays there until it wakes up. After it goes to sleep, the kernel schedules another process to run and switches context to it.

2.3 KERNEL DATA STRUCTURES

Most kernel data structures occupy fixed-size tables rather than dynamically allocated space. The advantage of this approach is that the kernel code is simple, but it limits the number of entries for a data structure to the number that was originally configured when generating the system: If, during operation of the system, the kernel should run out of entries for a data structure, it cannot allocate space for new entries dynamically but must report an error to the requesting user. If, on the other hand, the kernel is configured so that it it is unlikely to run out of table space, the extra table space may be wasted because it cannot be used for other purposes. Nevertheless, the simplicity of the kernel algorithms has generally been considered more important than the need to squeeze out every last byte of main memory. Algorithms typically use simple loops to find free table entries, a method that is easier to understand and sometimes more efficient than more complicated allocation schemes.

2.4 SYSTEM ADMINISTRATION

Administrative processes are loosely classified as those processes that do various functions for the general welfare of the user community. Such functions include disk formatting, creation of new file systems, repair of damaged file systems, kernel debugging, and others. Conceptually, there is no difference between administrative

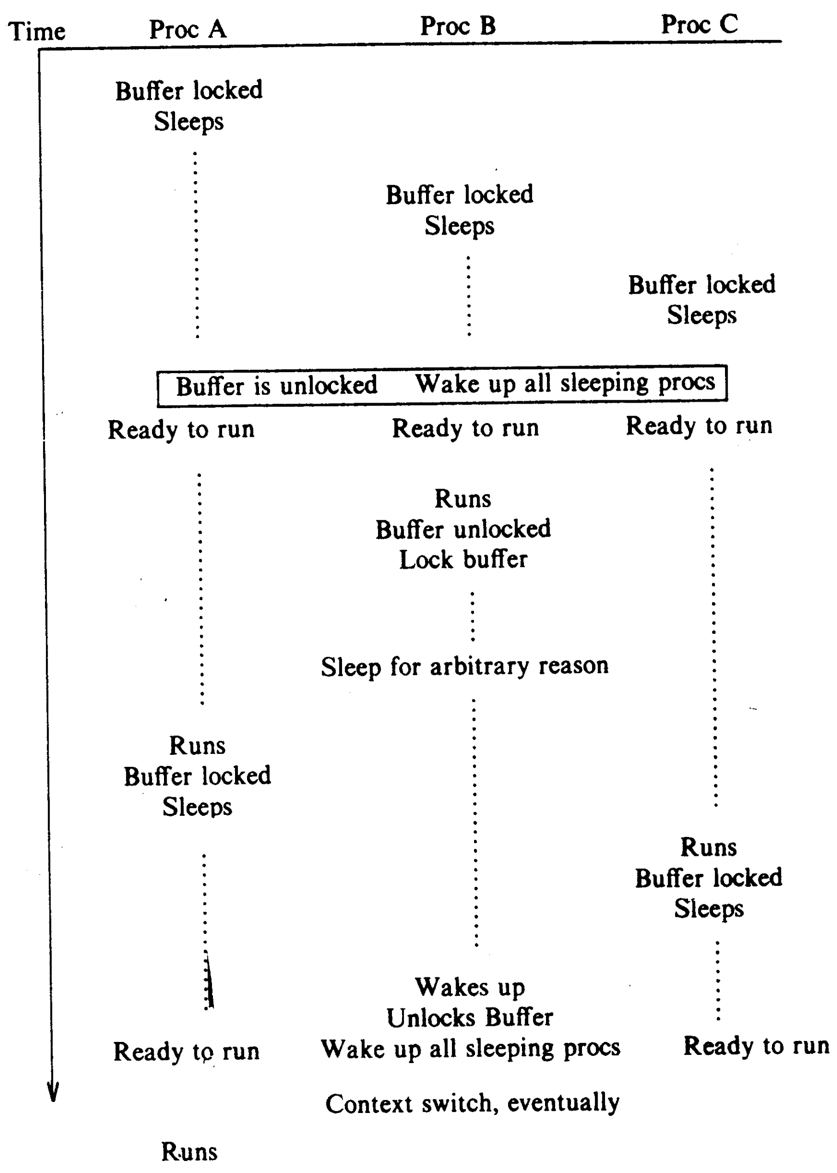


Figure 2.9. Multiple Processes Sleeping on a Lock

processes and user processes: They use the same set of system calls available to the general community. They are distinguished from general user processes only in the rights and privileges they are allowed. For example, file permission modes may allow administrative processes to manipulate files otherwise off-limits to general users. Internally. the kernel distinguishes a special user called the *superuser*, endowing it with special privileges, as will be seen. A user may become a superuser by going through a login-password sequence or by executing special programs. Other uses of superuser privileges will be studied in later chapters. In short, the kernel does not recognize a separate class of administrative processes.

2.5 SUMMARY AND PREVIEW

This chapter has described the architecture of the kernel; its two major components are the file subsystem and the process subsystem. The file subsystem controls the storage and retrieval of data in user files. Files are organized into file systems, which are treated as logical devices; a physical device such as a disk can contain several logical devices (file systems). Each file system has a super block that describes the structure and contents of the file system, and each file in a file system is described by an inode that gives the attributes of the file. System calls that manipulate files do so via inodes.

Processes exist in various states and move between them according to well-defined transition rules. In particular, processes executing in kernel mode can suspend their execution and enter the sleep state, but no process can put another process to sleep. The kernel is non-preemptive, meaning that a process executing in kernel mode will continue to execute until it enters the sleep state or until it returns to execute in user mode. The kernel maintains the consistency of its data structures by enforcing the policy of non-preemption and by blocking interrupts when executing critical regions of code.

The remainder of this text describes the subsystems shown in Figure 2.1 and their interactions in detail, starting with the file subsystem and continuing with the process subsystem. The next chapter covers the buffer cache and describes buffer allocation algorithms, used in the algorithms presented in Chapters 4, 5, and 7. Chapter 4 examines internal algorithms of the file system, including the manipulation of inodes, the structure of files, and the conversion of path names to inodes. Chapter 5 explains the system calls that use the algorithms in Chapter 4 to access the file system, such as *open*, *close*, *read*, and *write*. Chapter 6 deals with the basic ideas of the context of a process and its address space, and Chapter 7 covers system calls that deal with process management and use the algorithms in Chapter 6. Chapter 8 examines process scheduling, and Chapter 9 discusses memory management algorithms. Chapter 10 covers device drivers, postponed to this point so that the relationship between the terminal driver and process management can be explained. Chapter 11 presents several forms of interprocess communication. Finally, the last two chapters cover advanced topics, including multiprocessor systems and distributed systems.

2.6 EXERCISES

1. Consider the following sequence of commands:

```
grep main a.c b.c c.c > grepout &
wc −l < grepout &
rm grepout &
```

The ampersand ("&") at the end of each command line informs the shell to run the command in the background, and it can execute each command line in parallel. Why is this not equivalent to the following command line?

```
grep main a.c b.c c.c | wc −l
```

2. Consider the sample kernel code in Figure 2.7. Suppose a context switch happens when the code reaches the comment, and suppose another process removes a buffer from the linked list by executing the following code:

```
remove(qp)
        struct queue *qp;
{
        qp−>forp−>backp = qp−>backp;
        qp−>backp−>forp = qp−>forp;
        qp−>forp = qp−>backp = NULL;
}
```

Consider three cases:
– The process removes the structure *bp1* from the linked list.
– The process removes the structure that currently follows *bp1* on the linked list.
– The process removes the structure that originally followed *bp1* before *bp* was half placed on the linked list.

What is the status of the linked list after the original process completes executing the code after the comment?

3. What should happen if the kernel attempts to awaken all processes sleeping on an event, but no processes are asleep on the event at the time of the wakeup?

3

THE BUFFER CACHE

As mentioned in the previous chapter, the kernel maintains files on mass storage devices such as disks, and it allows processes to store new information or to recall previously stored information. When a process wants to access data from a file, the kernel brings the data into main memory where the process can examine it, alter it, and request that the data be saved in the file system again. For example, recall the *copy* program in Figure 1.3: The kernel reads the data from the first file into memory, and then writes the data into the second file. Just as it must bring file data into memory, the kernel must also bring auxiliary data into memory to manipulate it. For instance, the super block of a file system describes the free space available on the file system, among other things. The kernel reads the super block into memory to access its data and writes it back to the file system when it wishes to save its data Similarly, the inode describes the layout of a file. The kernel reads an inode into memory when it wants to access data in a file and writes the inode back to the file system when it wants to update the file layout. It manipulates this auxiliary data without the explicit knowledge or request of running processes.

The kernel could read and write directly to and from the disk for all file system accesses, but system response time and throughput would be poor because of the slow disk transfer rate. The kernel therefore attempts to minimize the frequency of disk access by keeping a pool of internal data buffers, called the buffer cache,[1] which contains the data in recently used disk blocks.

Figure 2.1 showed the position of the buffer cache module in the kernel architecture between the file subsystem and (block) device drivers. When reading data from the disk, the kernel attempts to read from the buffer cache. If the data is already in the cache, the kernel does not have to read from the disk. If the data is not in the cache, the kernel reads the data from the disk and caches it, using an algorithm that tries to save as much good data in the cache as possible. Similarly, data being written to disk is cached so that it will be there if the kernel later tries to read it. The kernel also attempts to minimize the frequency of disk write operations by determining whether the data must really be stored on disk or whether it is transient data that will soon be overwritten. Higher-level kernel algorithms instruct the buffer cache module to pre-cache data or to delay-write data to maximize the caching effect. This chapter describes the algorithms the kernel uses to manipulate buffers in the buffer cache.

3.1 BUFFER HEADERS

During system initialization, the kernel allocates space for a number of buffers, configurable according to memory size and system performance constraints. A buffer consists of two parts: a memory array that contains data from the disk and a *buffer header* that identifies the buffer. Because there is a one to one mapping of buffer headers to data arrays, the ensuing text will frequently refer to both parts as a "buffer," and the context should make clear which part is being discussed.

The data in a buffer corresponds to the data in a logical disk block on a file system, and the kernel identifies the buffer contents by examining identifier fields in the buffer header. The buffer is the in-memory copy of the disk block; the contents of the disk block map into the buffer, but the mapping is temporary until the kernel decides to map another disk block into the buffer. A disk block can never map into more than one buffer at a time. If two buffers were to contain data for one disk block, the kernel would not know which buffer contained the current data and could write incorrect data back to disk. For example, suppose a disk block maps into two buffers, A and B. If the kernel writes data first into buffer A and then into buffer B, the disk block should contain the contents of buffer B if all write operations completely fill the buffer. However, if the kernel reverses the order when it copies the buffers to disk, the disk block will contain incorrect data.

The buffer header (Figure 3.1) contains a *device number* field and a *block number* field that specify the file system and block number of the data on disk and uniquely identify the buffer. The device number is the logical file system number

1. The buffer cache is a software structure that should not be confused with hardware caches that speed memory references.

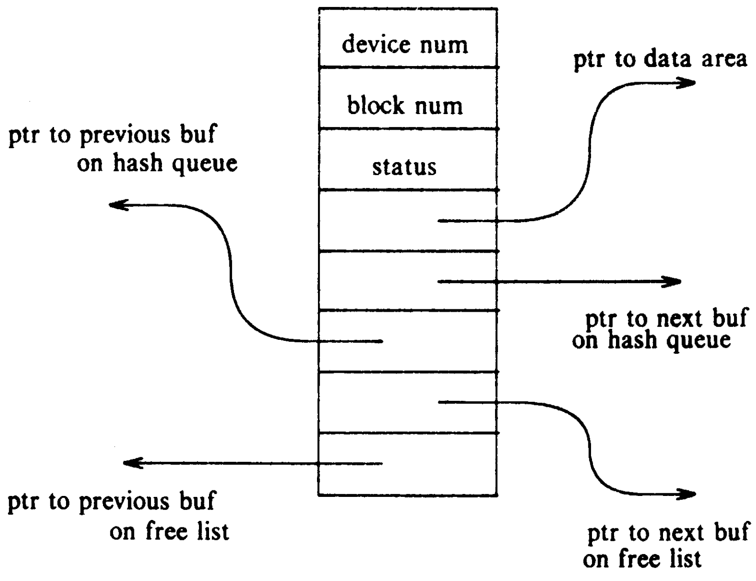

device num
block num
status
ptr to data area
ptr to previous buf on hash queue
ptr to next buf on hash queue
ptr to previous buf on free list
ptr to next buf on free list

Figure 3.1. Buffer Header

(see Section 2.2.1), not a physical device (disk) unit number. The buffer header also contains a pointer to a data array for the buffer, whose size must be at least as big as the size of a disk block, and a status field that summarizes the current status of the buffer. The status of a buffer is a combination of the following conditions:

- The buffer is currently locked (the terms "locked" and "busy" will be used interchangeably, as will "free" and "unlocked"),
- The buffer contains valid data,
- The kernel must write the buffer contents to disk before reassigning the buffer; this condition is known as "delayed-write,"
- The kernel is currently reading or writing the contents of the buffer to disk,
- A process is currently waiting for the buffer to become free.

The buffer header also contains two sets of pointers, used by the buffer allocation algorithms to maintain the overall structure of the buffer pool, as explained in the next section.

3.2 STRUCTURE OF THE BUFFER POOL

The kernel caches data in the buffer pool according to a *least recently used* algorithm: after it allocates a buffer to a disk block, it cannot use the buffer for

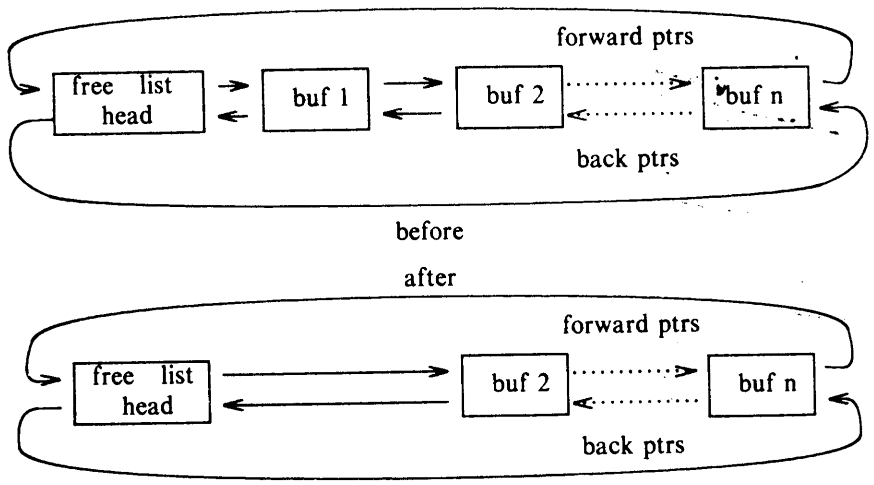


Figure 3.2. Free List of Buffers

another block until all other buffers have been used more recently. The kernel maintains a *free list* of buffers that preserves the least recently used order. The free list is a doubly linked circular list of buffers with a dummy buffer header that marks its beginning and end (Figure 3.2). Every buffer is put on the free list when the system is booted. The kernel takes a buffer from the head of the free list when it wants *any* free buffer, but it can take a buffer from the middle of the free list if it identifies a particular block in the buffer pool. In both cases, it removes the buffer from the free list. When the kernel returns a buffer to the buffer pool, it usually attaches the buffer to the tail of the free list, occasionally to the head of the free list (for error cases), but never to the middle. As the kernel removes buffers from the free list, a buffer with valid data moves closer and closer to head of the free list (Figure 3.2). Hence, the buffers that are closer to the head of the free list have not been used as recently as those that are further from the head of the free list.

When the kernel accesses a disk block, it searches for a buffer with the appropriate device-block number combination. Rather than search the entire buffer pool, it organizes the buffers into separate queues, *hashed* as a function of the device and block number. The kernel links the buffers on a hash queue into a circular, doubly linked list, similar to the structure of the free list. The number of buffers on a hash queue varies during the lifetime of the system, as will be seen. The kernel must use a hashing function that distributes the buffers uniformly across the set of hash queues, yet the hash function must be simple so that performance does not suffer. System administrators configure the number of hash queues when generating the operating system.

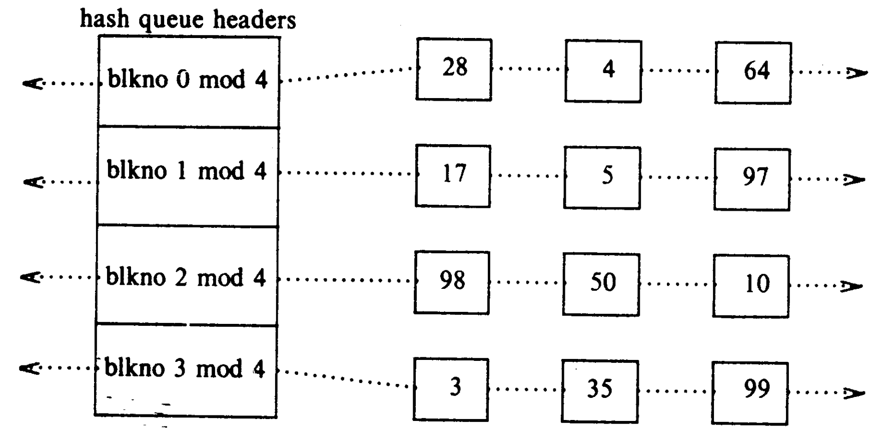


Figure 3.3. Buffers on the Hash Queues

Figure 3.3 shows buffers on their hash queues: the headers of the hash queues are on the left side of the figure, and the squares on each row are buffers on a hash queue. Thus, squares marked 28, 4, and 64 represent buffers on the hash queue for "blkno 0 mod 4" (block number 0 modulo 4). The dotted lines between the buffers represent the forward and back pointers for the hash queue; for simplicity, later figures in this chapter will not show these pointers, but their existence is implicit. Similarly, the figure identifies blocks only by their block number, and it uses a hash function dependent only on a block number; however, implementations use the device number, too.

Each buffer always exists on a hash queue, but there is no significance to its position on the queue. As stated above, no two buffers may simultaneously contain the contents of the same disk block; therefore, every disk block in the buffer pool exists on one and only one hash queue and only once on that queue. However, a buffer may be on the free list as well if its status is free. Because a buffer may be simultaneously on a hash queue and on the free list, the kernel has two ways to find it: It searches the hash queue if it is looking for a particular buffer, and it removes a buffer from the free list if it is looking for *any* free buffer. The next section will show how the kernel finds particular disk blocks in the buffer cache, and how it manipulates buffers on the hash queues and on the free list. To summarize, a buffer is always on a hash queue, but it may or may not be on the free list.

3.3 SCENARIOS FOR RETRIEVAL OF A BUFFER

As seen in Figure 2.1, high-level kernel algorithms in the file subsystem invoke the algorithms for managing the buffer cache. The high-level algorithms determine the logical device number and block number that they wish to access when they attempt to retrieve a block. For example, if a process wants to read data from a file, the kernel determines which file system contains the file and which block in the file system contains the data, as will be seen in Chapter 4. When about to read data from a particular disk block, the kernel checks whether the block is in the buffer pool and, if it is not there, assigns it a free buffer. When about to write data to a particular disk block, the kernel checks whether the block is in the buffer pool, and if not, assigns a free buffer for that block. The algorithms for reading and writing disk blocks use the algorithm *getblk* (Figure 3.4) to allocate buffers from the pool.

This section describes five typical scenarios the kernel may follow in *getblk* to allocate a buffer for a disk block.

1. The kernel finds the block on its hash queue, and its buffer is free.
2. The kernel cannot find the block on the hash queue, so it allocates a buffer from the free list.
3. The kernel cannot find the block on the hash queue and, in attempting to allocate a buffer from the free list (as in scenario 2), finds a buffer on the free list that has been marked "delayed write." The kernel must write the "delayed write" buffer to disk and allocate another buffer.
4. The kernel cannot find the block on the hash queue, and the free list of buffers is empty.
5. The kernel finds the block on the hash queue, but its buffer is currently busy.

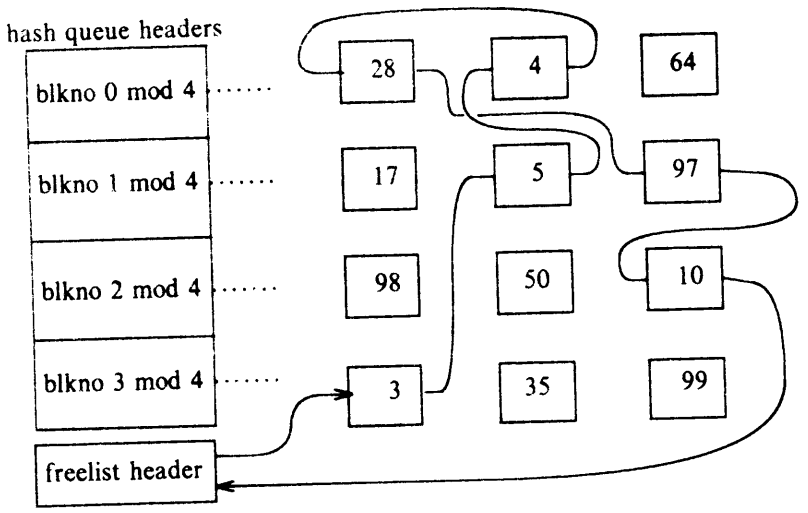
Let us now discuss each scenario in greater detail.

When searching for a block in the buffer pool by its device-block number combination, the kernel finds the hash queue that should contain the block. It searches the hash queue, following the linked list of buffers until (in the first scenario) it finds the buffer whose device and block number match those for which it is searching. The kernel checks that the buffer is free and, if so, marks the buffer "busy" so that other processes[2] cannot access it. The kernel then removes the buffer from the free list, because a buffer cannot be both busy and on the free list. If other processes attempt to access the block while the buffer is busy, they sleep until the buffer is released, as will be seen. Figure 3.5 depicts the first scenario, where the kernel searches for block 4 on the hash queue marked "blkno 0 mod 4." Finding the buffer, the kernel removes it from the free list, leaving blocks 5 and 28 adjacent on the free list.

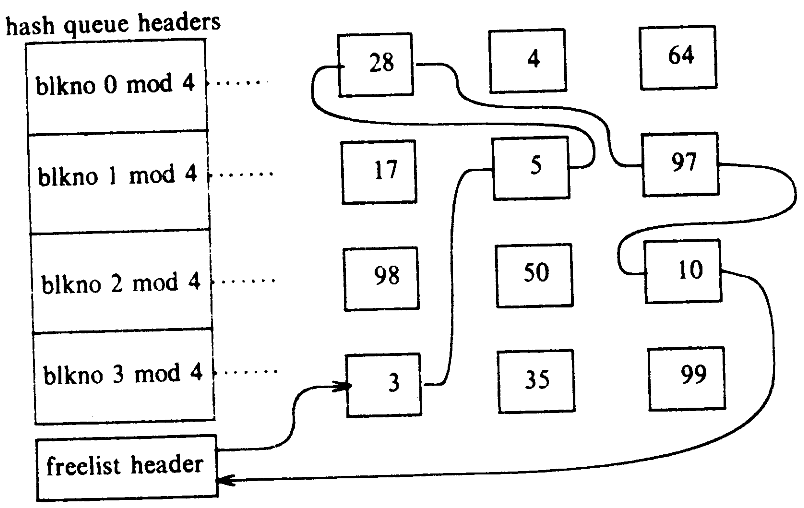
2. Recall from the last chapter that all kernel operations are done in the context of a process that is executing in kernel mode. Thus, the term "other processes" means that they are also executing in kernel mode. This term will be used when describing the interaction of several processes executing in kernel mode; if there is no interprocess interaction, the term "kernel" will be used.

```
algorithm getblk
input:  file system number
        block number
output: locked buffer that can now be used for block
{
        while (buffer not found)
        {
                if (block in hash queue)
                {
                        if (buffer busy)        /* scenario 5 */
                        {
                                sleep (event buffer becomes free);
                                continue;       /* back to while loop */
                        }
                        mark buffer busy;       /* scenario 1 */
                        remove buffer from free list;
                        return buffer;
                }
                else    /* block not on hash queue */
                {
                        if (there are no buffers on free list)  /* scenario 4 */
                        {
                                sleep (event any buffer becomes free);
                                continue;       /* back to while loop */
                        }
                        remove buffer from free list;
                        if (buffer marked for delayed write) {  /* scenario 3 */
                                asynchronous write buffer to disk;
                                continue;       /* back to while loop */
                        }
                        /* scenario 2 -- found a free buffer */
                        remove buffer from old hash queue;
                        put buffer onto new hash queue;
                        return buffer;
                }
        }
}
```

Figure 3.4. Algorithm for Buffer Allocation



(a) Search for Block 4 on First Hash Queue



(b) Remove Block 4 from Free List

Figure 3.5. Scenario 1 in Finding a Buffer: Buffer on Hash Queue

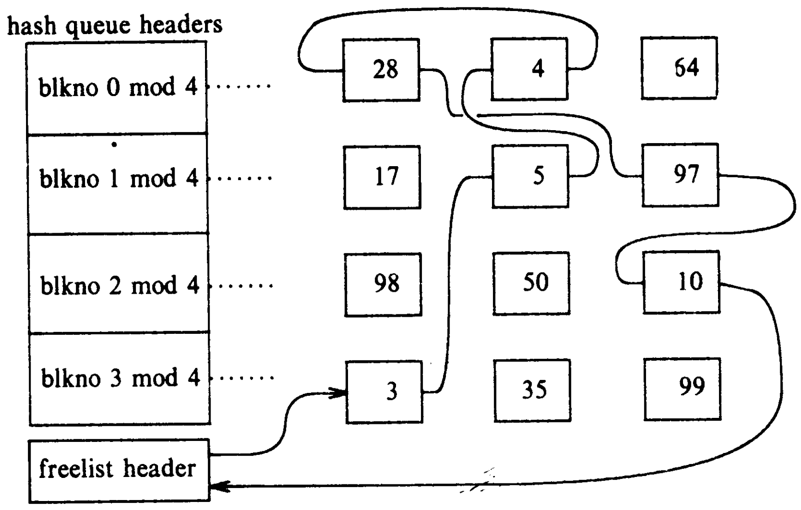
```
algorithm brelse
input:  locked buffer
output: none
{
        wakeup all procs: event, waiting for any buffer to become free;
        wakeup all procs: event, waiting for this buffer to become free;
        raise processor execution level to block interrupts;
        if (buffer contents valid and buffer not old)
                enqueue buffer at end of free list
        else
                enqueue buffer at beginning of free list
        lower processor execution level to allow interrupts;
        unlock(buffer);
}
```

Figure 3.6. Algorithm for Releasing a Buffer

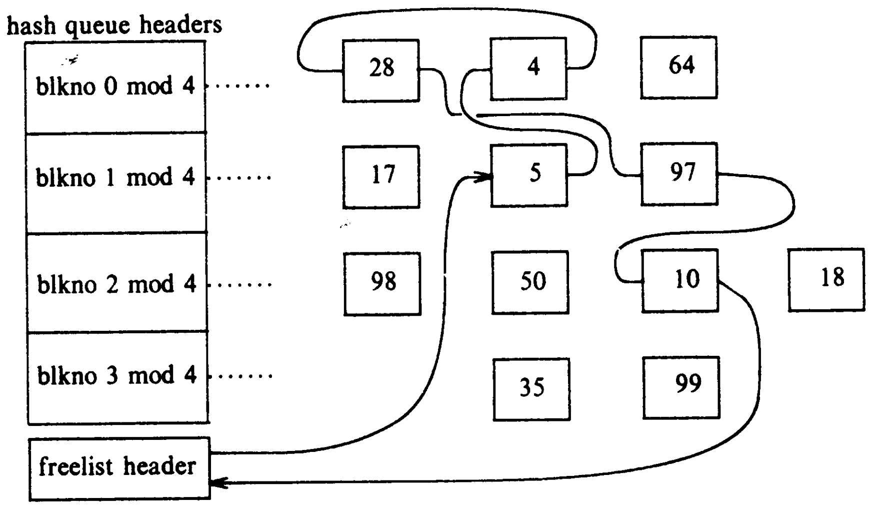
Before continuing to the other scenarios, let us consider what happens to a buffer after it is allocated. The kernel may read data from the disk to the buffer and manipulate it or write data to the buffer and possibly to the disk. The kernel leaves the buffer marked busy; no other process can access it and change its contents while it is busy, thus preserving the integrity of the data in the buffer. When the kernel finishes using the buffer, it releases the buffer according to algorithm *brelse* (Figure 3.6). It wakes up processes that had fallen asleep because the buffer was busy and processes that had fallen asleep because no buffers remained on the free list. In both cases, release of a buffer means that the buffer is available for use by the sleeping processes, although the first process that gets the buffer locks it and prevents the other processes from getting it (recall Section 2.2.2.4). The kernel places the buffer at the end of the free list, unless an I/O error occurred or unless it specifically marked the buffer "old," as will be seen later in this chapter; in the latter cases, it places the buffer at the beginning of the free list. The buffer is now free for another process to claim it.

Just as the kernel invokes algorithm *brelse* when a process has no more need for a buffer, it also invokes the algorithm when handling a disk interrupt to release buffers used for asynchronous I/O to and from the disk, as will be seen in Section 3.4. The kernel raises the processor execution level to prevent disk interrupts while manipulating the free list, thereby preventing corruption of the buffer pointers that could result from a nested call to *brelse*. Similar bad effects could happen if an interrupt handler invoked *brelse* while a process was executing *getblk*, so the kernel raises the processor execution level at strategic places in *getblk*, too. The exercises explore these cases in greater detail.

In the second scenario in algorithm *getblk*, the kernel searches the hash queue that should contain the block but fails to find it there. Since the block cannot be on another hash queue because it cannot "hash" elsewhere, it is not in the buffer



(a) Search for Block 18 - Not in Cache



(b) Remove First Block from Free List, Assign to 18

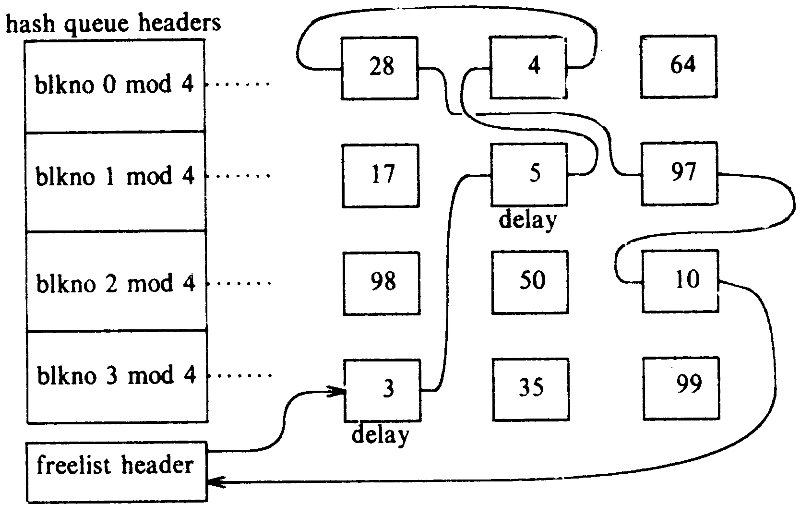
Figure 3.7. Second Scenario for Buffer Allocation

cache. So the kernel removes the first buffer from the free list instead; that buffer had been allocated to another disk block and is also on a hash queue. If the buffer has not been marked for a delayed write (as will be described later), the kernel marks the buffer busy, removes it from the hash queue where it currently resides, reassigns the buffer header's device and block number to that of the disk block for which the process is searching, and places the buffer on the correct hash queue. The kernel uses the buffer but has no record that the buffer formerly contained data for another disk block. A process searching for the old disk block will not find it in the pool and will have to allocate a new buffer for it from the free list, exactly as outlined here. When the kernel finishes with the buffer, it releases it as described above. In Figure 3.7, for example, the kernel searches for block 18 but does not find it on the hash queue marked "blkno 2 mod 4." It therefore removes the first buffer from the free list (block 3), assigns it to block 18, and places it on the appropriate hash queue.

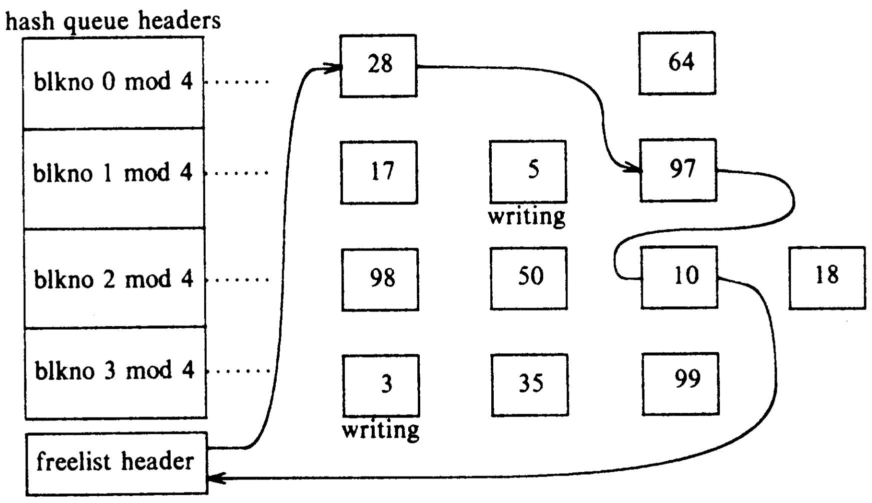
In the third scenario in algorithm *getblk*, the kernel also has to allocate a buffer from the free list. However, it discovers that the buffer it removes from the free list has been marked for "delayed write," so it must write the contents of the buffer to disk before using the buffer. The kernel starts an asynchronous write to disk and tries to allocate another buffer from the free list. When the asynchronous write completes, the kernel releases the buffer and places it at the head of the free list. The buffer had started at the end of the free list and had traveled to the head of the free list. If, after the asynchronous write, the kernel were to place the buffer at the end of the free list, the buffer would get a free trip through the free list, working against the least recently used algorithm. For example, in Figure 3.8, the kernel cannot find block 18, but when it attempts to allocate the first two buffers (one at a time) on the free list, it finds them marked for delayed write. The kernel removes them from the free list, starts write operations to disk for the blocks, and allocates the third buffer on the free list, block 4. It reassigns the buffer's device and block number fields appropriately and places the buffer, now marked block 18, on its new hash queue.

In the fourth scenario (Figure 3.9), the kernel, acting for process A, cannot find the disk block on its hash queue, so it attempts to allocate a new buffer from the free list, as in the second scenario. However, no buffers are available on the free list, so process A goes to sleep until another process executes algorithm *brelse*, freeing a buffer. When the kernel schedules process A, it must search the hash queue again for the block. It cannot allocate a buffer immediately from the free list, because it is possible that several processes were waiting for a free buffer and that one of them allocated a newly freed buffer for the target block sought by process A. Thus, searching for the block again insures that only one buffer contains the disk block. Figure 3.10 depicts the contention between two processes for a free buffer.

The final scenario (Figure 3.11) is complicated, because it involves complex relationships between several processes. Suppose the kernel, acting for process A, searches for a disk block and allocates a buffer but goes to sleep before freeing the

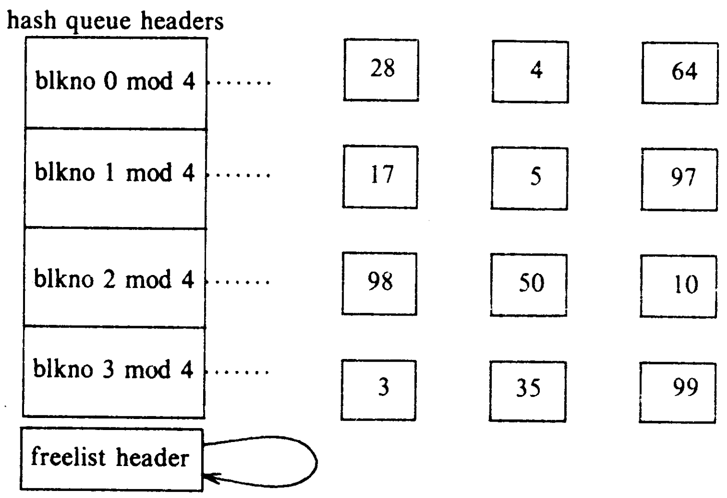


(a) Search for Block 18, Delayed Write Blocks on Free List



(b) Writing Blocks 3, 5, Reassign 4 to 18

Figure 3.8. Third Scenario for Buffer Allocation



Search for Block 18, Empty Free List

Figure 3.9. Fourth Scenario for Allocating Buffer

buffer. For example, if process A attempts to read a disk block and allocates a buffer as in scenario 2, then it will sleep while it waits for the I/O transmission from disk to complete. While process A sleeps, suppose the kernel schedules a second process, B, which tries to access the disk block whose buffer was just locked by process A. Process B (going through scenario 5) will find the locked block on the hash queue. Since it is illegal to use a locked buffer and it is illegal to allocate a second buffer for a disk block, process B marks the buffer "in demand" and then sleeps and waits for process A to release the buffer.

Process A will eventually release the buffer and notice that the buffer is in demand. It awakens all processes sleeping on the event "the buffer becomes free," including process B. When the kernel again schedules process B, process B must verify that the buffer is free. Another process, C, may have been waiting for the same buffer, and the kernel may have scheduled C to run before process B; process C may have gone to sleep leaving the buffer locked. Hence, process B must check that the block is indeed free.

Process B must also verify that the buffer contains the disk block that it originally requested, because process C may have allocated the buffer to another block, as in scenario 2. When process B executes, it may find that it had been waiting for the wrong buffer, so it must search for the block again: If it were to allocate a buffer automatically from the free list, it would miss the possibility that another process just allocated a buffer for the block.