

10.2 OVERVIEW OF LINUX

In this section we will provide a general introduction to Linux and how it is used, for the benefit of readers not already familiar with it. Nearly all of this material applies to just about all UNIX variants with only small deviations. Although Linux has several graphical interfaces, the focus here is on how Linux appears to a programmer working in a shell window on X. Subsequent sections will focus on system calls and how it works inside.

10.2.1 Linux Goals

UNIX was always an interactive system designed to handle multiple processes and multiple users at the same time. It was designed by programmers, for programmers, to use in an environment in which the majority of the users are relatively sophisticated and are engaged in (often quite complex) software development projects. In many cases, a large number of programmers are actively cooperating to produce a single system, so UNIX has extensive facilities to allow people to work together and share information in controlled ways. The model of a group of experienced programmers working together closely to produce advanced software is obviously very different from the personal-computer model of a single beginner working alone with a word processor, and this difference is reflected throughout UNIX from start to finish. It is only natural that Linux inherited many of these goals, even though the first version was for a personal computer.

What is it that good programmers really want in a system? To start with, most like their systems to be simple, elegant, and consistent. For example, at the lowest level, a file should just be a collection of bytes. Having different classes of files for sequential access, random access, keyed access, remote access, and so on (as mainframes do) just gets in the way. Similarly, if the command

```
ls A*
```

means list all the files beginning with “A”, then the command

```
rm A*
```

should mean remove all the files beginning with “A” and not remove the one file whose name consists of an “A” and an asterisk. This characteristic is sometimes called the *principle of least surprise*.

Another thing that experienced programmers generally want is power and flexibility. This means that a system should have a small number of basic elements that can be combined in an infinite variety of ways to suit the application. One of the basic guidelines behind Linux is that every program should do just one thing and do it well. Thus compilers do not produce listings, because other programs can do that better.

Finally, most programmers have a strong dislike for useless redundancy. Why type *copy* when *cp* is clearly enough to make it abundantly clear what you want? It

is a complete waste of valuable hacking time. To extract all the lines containing the string “ard” from the file *f*, the Linux programmer merely types

```
grep ard f
```

The opposite approach is to have the programmer first select the *grep* program (with no arguments), and then have *grep* announce itself by saying: “Hi, I’m *grep*, I look for patterns in files. Please enter your pattern.” After getting the pattern, *grep* prompts for a file name. Then it asks if there are any more file names. Finally, it summarizes what it is going to do and asks if that is correct. While this kind of user interface may be suitable for rank novices, it drives skilled programmers up the wall. What they want is a servant, not a nanny.

10.2.2 Interfaces to Linux

A Linux system can be regarded as a kind of pyramid, as illustrated in Fig. 10-1. At the bottom is the hardware, consisting of the CPU, memory, disks, a monitor and keyboard, and other devices. Running on the bare hardware is the operating system. Its function is to control the hardware and provide a system call interface to all the programs. These system calls allow user programs to create and manage processes, files, and other resources.

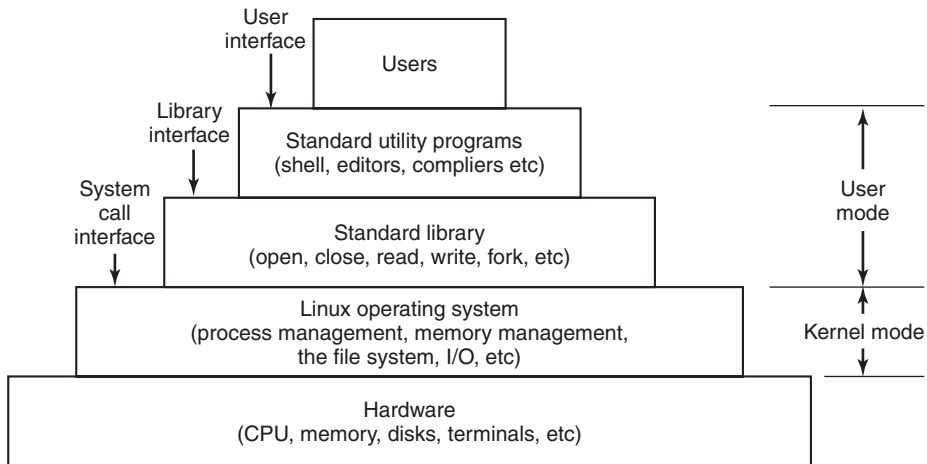


Figure 10-1. The layers in a Linux system.

Programs make system calls by putting the arguments in registers (or sometimes, on the stack), and issuing trap instructions to switch from user mode to kernel mode. Since there is no way to write a trap instruction in C, a library is provided, with one procedure per system call. These procedures are written in assembly language but can be called from C. Each one first puts its arguments in the

proper place, then executes the trap instruction. Thus to execute the read system call, a C program can call the `read` library procedure. As an aside, it is the library interface, and not the system call interface, that is specified by POSIX. In other words, POSIX tells which library procedures a conformant system must supply, what their parameters are, what they must do, and what results they must return. It does not even mention the actual system calls.

In addition to the operating system and system call library, all versions of Linux supply a large number of standard programs, some of which are specified by the POSIX 1003.2 standard, and some of which differ between Linux versions. These include the command processor (shell), compilers, editors, text-processing programs, and file-manipulation utilities. It is these programs that a user at the keyboard invokes. Thus, we can speak of three different interfaces to Linux: the true system call interface, the library interface, and the interface formed by the set of standard utility programs.

Most of the common personal computer distributions of Linux have replaced this keyboard-oriented user interface with a mouse-oriented graphical user interface, without changing the operating system itself at all. It is precisely this flexibility that makes Linux so popular and has allowed it to survive numerous changes in the underlying technology so well.

The GUI for Linux is similar to the first GUIs developed for UNIX systems in the 1970s, and popularized by Macintosh and later Windows for PC platforms. The GUI creates a desktop environment, a familiar metaphor with windows, icons, folders, toolbars, and drag-and-drop capabilities. A full desktop environment contains a window manager, which controls the placement and appearance of windows, as well as various applications, and provides a consistent graphical interface. Popular desktop environments for Linux include GNOME (GNU Network Object Model Environment) and KDE (K Desktop Environment).

GUIs on Linux are supported by the X Windowing System, or commonly X11 or just X, which defines communication and display protocols for manipulating windows on bitmap displays for UNIX and UNIX-like systems. The X server is the main component which controls devices such as the keyboard, mouse, and screen and is responsible for redirecting input to or accepting output from client programs. The actual GUI environment is typically built on top of a low-level library, *xlib*, which contains the functionality to interact with the X server. The graphical interface extends the basic functionality of X11 by enriching the window view, providing buttons, menus, icons, and other options. The X server can be started manually, from a command line, but is typically started during the boot process by a display manager, which displays the graphical login screen for the user.

When working on Linux systems through a graphical interface, users may use mouse clicks to run applications or open files, drag and drop to copy files from one location to another, and so on. In addition, users may invoke a terminal emulator program, or *xterm*, which provides them with the basic command-line interface to the operating system. Its description is given in the following section.

10.2.3 The Shell

Although Linux systems have a graphical user interface, most programmers and sophisticated users still prefer a command-line interface, called the **shell**. Often they start one or more shell windows from the graphical user interface and just work in them. The shell command-line interface is much faster to use, more powerful, easily extensible, and does not give the user RSI from having to use a mouse all the time. Below we will briefly describe the bash shell (*bash*). It is heavily based on the original UNIX shell, *Bourne shell* (written by Steve Bourne, then at Bell Labs). Its name is an acronym for *Bourne Again SHell*. Many other shells are also in use (*ksh*, *csh*, etc.), but *bash* is the default shell in most Linux systems.

When the shell starts up, it initializes itself, then types a **prompt** character, often a percent or dollar sign, on the screen and waits for the user to type a command line.

When the user types a command line, the shell extracts the first word from it, where word here means a run of characters delimited by a space or tab. It then assumes this word is the name of a program to be run, searches for this program, and if it finds it, runs the program. The shell then suspends itself until the program terminates, at which time it tries to read the next command. What is important here is simply the observation that the shell is an ordinary user program. All it needs is the ability to read from the keyboard and write to the monitor and the power to execute other programs.

Commands may take arguments, which are passed to the called program as character strings. For example, the command line

```
cp src dest
```

invokes the *cp* program with two arguments, *src* and *dest*. This program interprets the first one to be the name of an existing file. It makes a copy of this file and calls the copy *dest*.

Not all arguments are file names. In

```
head -20 file
```

the first argument, *-20*, tells *head* to print the first 20 lines of *file*, instead of the default number of lines, 10. Arguments that control the operation of a command or specify an optional value are called **flags**, and by convention are indicated with a dash. The dash is required to avoid ambiguity, because the command

```
head 20 file
```

is perfectly legal, and tells *head* to first print the initial 10 lines of a file called *20*, and then print the initial 10 lines of a second file called *file*. Most Linux commands accept multiple flags and arguments.

To make it easy to specify multiple file names, the shell accepts **magic characters**, sometimes called **wild cards**. An asterisk, for example, matches all possible strings, so

```
ls *.c
```

tells *ls* to list all the files whose name ends in *.c*. If files named *x.c*, *y.c*, and *z.c* all exist, the above command is equivalent to typing

```
ls x.c y.c z.c
```

Another wild card is the question mark, which matches any one character. A list of characters inside square brackets selects any of them, so

```
ls [ape]*
```

lists all files beginning with “a”, “p”, or “e”.

A program like the shell does not have to open the terminal (keyboard and monitor) in order to read from it or write to it. Instead, when it (or any other program) starts up, it automatically has access to a file called **standard input** (for reading), a file called **standard output** (for writing normal output), and a file called **standard error** (for writing error messages). Normally, all three default to the terminal, so that reads from standard input come from the keyboard and writes to standard output or standard error go to the screen. Many Linux programs read from standard input and write to standard output as the default. For example,

```
sort
```

invokes the *sort* program, which reads lines from the terminal (until the user types a CTRL-D, to indicate end of file), sorts them alphabetically, and writes the result to the screen.

It is also possible to redirect standard input and standard output, as that is often useful. The syntax for redirecting standard input uses a less-than symbol (<) followed by the input file name. Similarly, standard output is redirected using a greater-than symbol (>). It is permitted to redirect both in the same command. For example, the command

```
sort <in >out
```

causes *sort* to take its input from the file *in* and write its output to the file *out*. Since standard error has not been redirected, any error messages go to the screen. A program that reads its input from standard input, does some processing on it, and writes its output to standard output is called a **filter**.

Consider the following command line consisting of three separate commands:

```
sort <in >temp; head -30 <temp; rm temp
```

It first runs *sort*, taking the input from *in* and writing the output to *temp*. When that has been completed, the shell runs *head*, telling it to print the first 30 lines of

temp and print them on standard output, which defaults to the terminal. Finally, the temporary file is removed. It is not recycled. It is gone with the wind, forever.

It frequently occurs that the first program in a command line produces output that is used as input to the next program. In the above example, we used the file *temp* to hold this output. However, Linux provides a simpler construction to do the same thing. In

```
sort <in | head -30
```

the vertical bar, called the **pipe symbol**, says to take the output from *sort* and use it as the input to *head*, eliminating the need for creating, using, and removing the temporary file. A collection of commands connected by pipe symbols, called a **pipeline**, may contain arbitrarily many commands. A four-component pipeline is shown by the following example:

```
grep ter *.t | sort | head -20 | tail -5 >foo
```

Here all the lines containing the string “ter” in all the files ending in *.t* are written to standard output, where they are sorted. The first 20 of these are selected out by *head*, which passes them to *tail*, which writes the last five (i.e., lines 16 to 20 in the sorted list) to *foo*. This is an example of how Linux provides basic building blocks (numerous filters), each of which does one job, along with a mechanism for them to be put together in almost limitless ways.

Linux is a general-purpose multiprogramming system. A single user can run several programs at once, each as a separate process. The shell syntax for running a process in the background is to follow its command with an ampersand. Thus

```
wc -l <a >b &
```

runs the word-count program, *wc*, to count the number of lines (*-l* flag) in its input, *a*, writing the result to *b*, but does it in the background. As soon as the command has been typed, the shell types the prompt and is ready to accept and handle the next command. Pipelines can also be put in the background, for example, by

```
sort <x | head &
```

Multiple pipelines can run in the background simultaneously.

It is possible to put a list of shell commands in a file and then start a shell with this file as standard input. The (second) shell just processes them in order, the same as it would with commands typed on the keyboard. Files containing shell commands are called **shell scripts**. Shell scripts may assign values to shell variables and then read them later. They may also have parameters, and use *if*, *for*, *while*, and *case* constructs. Thus a shell script is really a program written in shell language. The Berkeley C shell is an alternative shell designed to make shell scripts (and the command language in general) look like C programs in many respects. Since the shell is just another user program, other people have written and distributed a variety of other shells. Users are free to choose whatever shells they like.

10.2.4 Linux Utility Programs

The command-line (shell) user interface to Linux consists of a large number of standard utility programs. Roughly speaking, these programs can be divided into six categories, as follows:

1. File and directory manipulation commands.
2. Filters.
3. Program development tools, such as editors and compilers.
4. Text processing.
5. System administration.
6. Miscellaneous.

The POSIX 1003.1-2008 standard specifies the syntax and semantics of about 150 of these, primarily in the first three categories. The idea of standardizing them is to make it possible for anyone to write shell scripts that use these programs and work on all Linux systems.

In addition to these standard utilities, there are many application programs as well, of course, such as Web browsers, media players, image viewers, office suites, games, and so on.

Let us consider some examples of these programs, starting with file and directory manipulation.

`cp a b`

copies file *a* to *b*, leaving the original file intact. In contrast,

`mv a b`

copies *a* to *b* but removes the original. In effect, it moves the file rather than really making a copy in the usual sense. Several files can be concatenated using *cat*, which reads each of its input files and copies them all to standard output, one after another. Files can be removed by the *rm* command. The *chmod* command allows the owner to change the rights bits to modify access permissions. Directories can be created with *mkdir* and removed with *rmdir*. To see a list of the files in a directory, *ls* can be used. It has a vast number of flags to control how much detail about each file is shown (e.g., size, owner, group, creation date), to determine the sort order (e.g., alphabetical, by time of last modification, reversed), to specify the layout on the screen, and much more.

We have already seen several filters: *grep* extracts lines containing a given pattern from standard input or one or more input files; *sort* sorts its input and writes it on standard output; *head* extracts the initial lines of its input; *tail* extracts the final lines of its input. Other filters defined by 1003.2 are *cut* and *paste*, which allow

columns of text to be cut and pasted into files; *od*, which converts its (usually binary) input to ASCII text, in octal, decimal, or hexadecimal; *tr*, which does character translation (e.g., lowercase to uppercase), and *pr*, which formats output for the printer, including options to include running heads, page numbers, and so on.

Compilers and programming tools include *gcc*, which calls the C compiler, and *ar*, which collects library procedures into archive files.

Another important tool is *make*, which is used to maintain large programs whose source code consists of multiple files. Typically, some of these are **header files**, which contain type, variable, macro, and other declarations. Source files often include these using a special *include* directive. This way, two or more source files can share the same declarations. However, if a header file is modified, it is necessary to find all the source files that depend on it and recompile them. The function of *make* is to keep track of which file depends on which header, and similar things, and arrange for all the necessary compilations to occur automatically. Nearly all Linux programs, except the smallest ones, are set up to be compiled with *make*.

A selection of the POSIX utility programs is listed in Fig. 10-2, along with a short description of each. All Linux systems have them and many more.

Program	Typical use
cat	Concatenate multiple files to standard output
chmod	Change file protection mode
cp	Copy one or more files
cut	Cut columns of text from a file
grep	Search a file for some pattern
head	Extract the first lines of a file
ls	List directory
make	Compile files to build a binary
mkdir	Make a directory
od	Octal dump a file
paste	Paste columns of text into a file
pr	Format a file for printing
ps	List running processes
rm	Remove one or more files
rmdir	Remove a directory
sort	Sort a file of lines alphabetically
tail	Extract the last lines of a file
tr	Translate between character sets

Figure 10-2. A few of the common Linux utility programs required by POSIX.

10.2.5 Kernel Structure

In Fig. 10-1 we saw the overall structure of a Linux system. Now let us zoom in and look more closely at the kernel as a whole before examining the various parts, such as process scheduling and the file system.

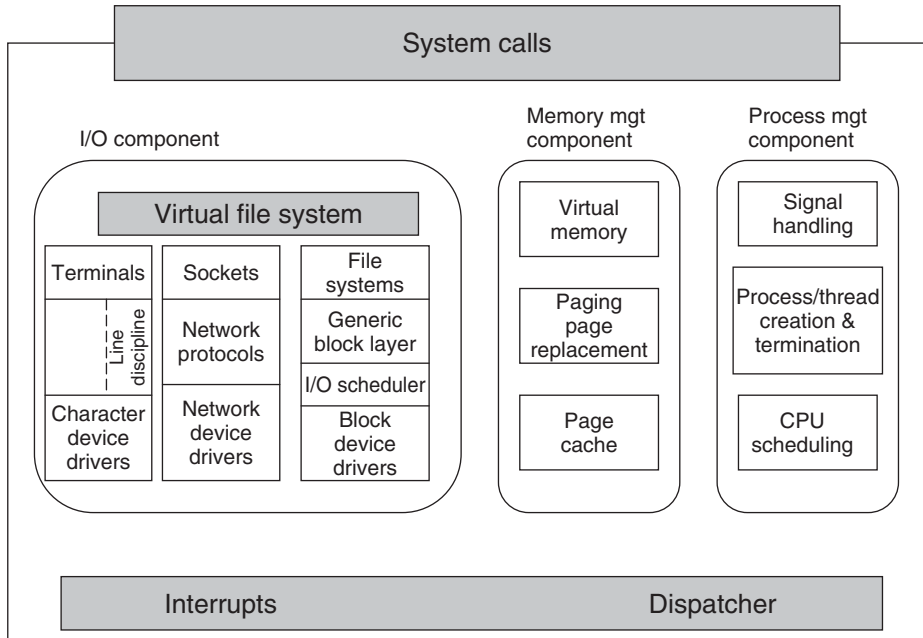


Figure 10-3. Structure of the Linux kernel

The kernel sits directly on the hardware and enables interactions with I/O devices and the memory management unit and controls CPU access to them. At the lowest level, as shown in Fig. 10-3 it contains interrupt handlers, which are the primary way for interacting with devices, and the low-level dispatching mechanism. This dispatching occurs when an interrupt happens. The low-level code here stops the running process, saves its state in the kernel process structures, and starts the appropriate driver. Process dispatching also happens when the kernel completes some operations and it is time to start up a user process again. The dispatching code is in assembler and is quite distinct from scheduling.

Next, we divide the various kernel subsystems into three main components. The I/O component in Fig. 10-3 contains all kernel pieces responsible for interacting with devices and performing network and storage I/O operations. At the highest level, the I/O operations are all integrated under a **VFS (Virtual File System)** layer. That is, at the top level, performing a read operation on a file, whether it is in

memory or on disk, is the same as performing a read operation to retrieve a character from a terminal input. At the lowest level, all I/O operations pass through some device driver. All Linux drivers are classified as either character-device drivers or block-device drivers, the main difference being that seeks and random accesses are allowed on block devices and not on character devices. Technically, network devices are really character devices, but they are handled somewhat differently, so that it is probably clearer to separate them, as has been done in the figure.

Above the device-driver level, the kernel code is different for each device type. Character devices may be used in two different ways. Some programs, such as visual editors like *vi* and *emacs*, want every keystroke as it is hit. Raw terminal (tty) I/O makes this possible. Other software, such as the shell, is line oriented, allowing users to edit the whole line before hitting ENTER to send it to the program. In this case the character stream from the terminal device is passed through a so-called line discipline, and appropriate formatting is applied.

Networking software is often modular, with different devices and protocols supported. The layer above the network drivers handles a kind of routing function, making sure that the right packet goes to the right device or protocol handler. Most Linux systems contain the full functionality of a hardware router within the kernel, although the performance is less than that of a hardware router. Above the router code is the actual protocol stack, including IP and TCP, but also many additional protocols. Overlaying all the network is the socket interface, which allows programs to create sockets for particular networks and protocols, getting back a file descriptor for each socket to use later.

On top of the disk drivers is the I/O scheduler, which is responsible for ordering and issuing disk-operation requests in a way that tries to conserve wasteful disk head movement or to meet some other system policy.

At the very top of the block-device column are the file systems. Linux may, and in fact does, have multiple file systems coexisting concurrently. In order to hide the gruesome architectural differences of various hardware devices from the file system implementation, a generic block-device layer provides an abstraction used by all file systems.

To the right in Fig. 10-3 are the other two key components of the Linux kernel. These are responsible for the memory and process management tasks. Memory-management tasks include maintaining the virtual to physical-memory mappings, maintaining a cache of recently accessed pages and implementing a good page-replacement policy, and on-demand bringing in new pages of needed code and data into memory.

The key responsibility of the process-management component is the creation and termination of processes. It also includes the process scheduler, which chooses which process or, rather, thread to run next. As we shall see in the next section, the Linux kernel treats both processes and threads simply as executable entities, and will schedule them based on a global scheduling policy. Finally, code for signal handling also belongs to this component.

While the three components are represented separately in the figure, they are highly interdependent. File systems typically access files through the block devices. However, in order to hide the large latencies of disk accesses, files are copied into the page cache in main memory. Some files may even be dynamically created and may have only an in-memory representation, such as files providing some run-time resource usage information. In addition, the virtual memory system may rely on a disk partition or in-file swap area to back up parts of the main memory when it needs to free up certain pages, and therefore relies on the I/O component. Numerous other interdependencies exist.

In addition to the static in-kernel components, Linux supports dynamically loadable modules. These modules can be used to add or replace the default device drivers, file system, networking, or other kernel codes. The modules are not shown in Fig. 10-3.

Finally, at the very top is the system call interface into the kernel. All system calls come here, causing a trap which switches the execution from user mode into protected kernel mode and passes control to one of the kernel components described above.

10.3 PROCESSES IN LINUX

In the previous sections, we started out by looking at Linux as viewed from the keyboard, that is, what the user sees in an *xterm* window. We gave examples of shell commands and utility programs that are frequently used. We ended with a brief overview of the system structure. Now it is time to dig deeply into the kernel and look more closely at the basic concepts Linux supports, namely, processes, memory, the file system, and input/output. These notions are important because the system calls—the interface to the operating system itself—manipulate them. For example, system calls exist to create processes and threads, allocate memory, open files, and do I/O.

Unfortunately, with so many versions of Linux in existence, there are some differences between them. In this chapter, we will emphasize the features common to all of them rather than focus on any one specific version. Thus in certain sections (especially implementation sections), the discussion may not apply equally to every version.

10.3.1 Fundamental Concepts

The main active entities in a Linux system are the processes. Linux processes are very similar to the classical sequential processes that we studied in Chap 2. Each process runs a single program and initially has a single thread of control. In other words, it has one program counter, which keeps track of the next instruction to be executed. Linux allows a process to create additional threads once it starts.

Linux is a multiprogramming system, so multiple, independent processes may be running at the same time. Furthermore, each user may have several active processes at once, so on a large system, there may be hundreds or even thousands of processes running. In fact, on most single-user workstations, even when the user is absent, dozens of background processes, called **daemons**, are running. These are started by a shell script when the system is booted. (“Daemon” is a variant spelling of “demon,” which is a self-employed evil spirit.)

A typical daemon is the *cron daemon*. It wakes up once a minute to check if there is any work for it to do. If so, it does the work. Then it goes back to sleep until it is time for the next check.

This daemon is needed because it is possible in Linux to schedule activities minutes, hours, days, or even months in the future. For example, suppose a user has a dentist appointment at 3 o’clock next Tuesday. He can make an entry in the cron daemon’s database telling the daemon to beep at him at, say, 2:30. When the appointed day and time arrives, the cron daemon sees that it has work to do, and starts up the beeping program as a new process.

The cron daemon is also used to start up periodic activities, such as making daily disk backups at 4 A.M., or reminding forgetful users every year on October 31 to stock up on trick-or-treat goodies for Halloween. Other daemons handle incoming and outgoing electronic mail, manage the line printer queue, check if there are enough free pages in memory, and so forth. Daemons are straightforward to implement in Linux because each one is a separate process, independent of all other processes.

Processes are created in Linux in an especially simple manner. The fork system call creates an exact copy of the original process. The forking process is called the **parent process**. The new process is called the **child process**. The parent and child each have their own, private memory images. If the parent subsequently changes any of its variables, the changes are not visible to the child, and vice versa.

Open files are shared between parent and child. That is, if a certain file was open in the parent before the fork, it will continue to be open in both the parent and the child afterward. Changes made to the file by either one will be visible to the other. This behavior is only reasonable, because these changes are also visible to any unrelated process that opens the file.

The fact that the memory images, variables, registers, and everything else are identical in the parent and child leads to a small difficulty: How do the processes know which one should run the parent code and which one should run the child code? The secret is that the fork system call returns a 0 to the child and a nonzero value, the child’s **PID (Process Identifier)**, to the parent. Both processes normally check the return value and act accordingly, as shown in Fig. 10-4.

Processes are named by their PIDs. When a process is created, the parent is given the child’s PID, as mentioned above. If the child wants to know its own PID, there is a system call, `getpid`, that provides it. PIDs are used in a variety of ways. For example, when a child terminates, the parent is given the PID of the child that

```

pid = fork( );           /* if the fork succeeds, pid > 0 in the parent */
if (pid < 0) {
    handle_error( );     /* fork failed (e.g., memory or some table is full) */
} else if (pid > 0) {
    /* parent code goes here. */
} else {
    /* child code goes here. */
}

```

Figure 10-4. Process creation in Linux.

just finished. This can be important because a parent may have many children. Since children may also have children, an original process can build up an entire tree of children, grandchildren, and further descendants.

Processes in Linux can communicate with each other using a form of message passing. It is possible to create a channel between two processes into which one process can write a stream of bytes for the other to read. These channels are called **pipes**. Synchronization is possible because when a process tries to read from an empty pipe it is blocked until data are available.

Shell pipelines are implemented with pipes. When the shell sees a line like

```
sort <f | head
```

it creates two processes, *sort* and *head*, and sets up a pipe between them in such a way that *sort*'s standard output is connected to *head*'s standard input. In this way, all the data that *sort* writes go directly to *head*, instead of going to a file. If the pipe fills, the system stops running *sort* until *head* has removed some data from it.

Processes can also communicate in another way besides pipes: software interrupts. A process can send what is called a **signal** to another process. Processes can tell the system what they want to happen when an incoming signal arrives. The choices available are to ignore it, to catch it, or to let the signal kill the process. Terminating the process is the default for most signals. If a process elects to catch signals sent to it, it must specify a signal-handling procedure. When a signal arrives, control will abruptly switch to the handler. When the handler is finished and returns, control goes back to where it came from, analogous to hardware I/O interrupts. A process can send signals only to members of its **process group**, which consists of its parent (and further ancestors), siblings, and children (and further descendants). A process may also send a signal to all members of its process group with a single system call.

Signals are also used for other purposes. For example, if a process is doing floating-point arithmetic, and inadvertently divides by 0 (something that mathematicians tend to frown upon), it gets a SIGFPE (floating-point exception) signal. Some of the signals that are required by POSIX are listed in Fig. 10-5. Many Linux systems have additional signals as well, but programs using them may not be portable to other versions of Linux and UNIX in general.

Signal	Cause
SIGABRT	Sent to abort a process and force a core dump
SIGALRM	The alarm clock has gone off
SIGFPE	A floating-point error has occurred (e.g., division by 0)
SIGHUP	The phone line the process was using has been hung up
SIGILL	The user has hit the DEL key to interrupt the process
SIGQUIT	The user has hit the key requesting a core dump
SIGKILL	Sent to kill a process (cannot be caught or ignored)
SIGPIPE	The process has written to a pipe which has no readers
SIGSEGV	The process has referenced an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purposes
SIGUSR2	Available for application-defined purposes

Figure 10-5. Some of the signals required by POSIX.

10.3.2 Process-Management System Calls in Linux

Let us now look at the Linux system calls dealing with process management. The main ones are listed in Fig. 10-6. Fork is a good place to start the discussion. The Fork system call, supported also by other traditional UNIX systems, is the main way to create a new process in Linux systems. (We will discuss another alternative in the following section.) It creates an exact duplicate of the original process, including all the file descriptors, registers, and everything else. After the fork, the original process and the copy (the parent and child) go their separate ways. All the variables have identical values at the time of the fork, but since the entire parent address space is copied to create the child, subsequent changes in one of them do not affect the other. The fork call returns a value, which is zero in the child, and equal to the child's PID in the parent. Using the returned PID, the two processes can see which is the parent and which is the child.

In most cases, after a fork, the child will need to execute different code from the parent. Consider the case of the shell. It reads a command from the terminal, forks off a child process, waits for the child to execute the command, and then reads the next command when the child terminates. To wait for the child to finish, the parent executes a `waitpid` system call, which just waits until the child terminates (any child if more than one exists). `Waitpid` has three parameters. The first one allows the caller to wait for a specific child. If it is `-1`, any old child (i.e., the first child to terminate) will do. The second parameter is the address of a variable that will be set to the child's exit status (normal or abnormal termination and exit value). This allows the parent to know the fate of its child. The third parameter determines whether the caller blocks or returns if no child is already terminated.

System call	Description
<code>pid = fork()</code>	Create a child process identical to the parent
<code>pid = waitpid(pid, &statloc, opts)</code>	Wait for a child to terminate
<code>s = execve(name, argv, envp)</code>	Replace a process' core image
<code>exit(status)</code>	Terminate process execution and return status
<code>s = sigaction(sig, &act, &oldact)</code>	Define action to take on signals
<code>s = sigreturn(&context)</code>	Return from a signal
<code>s = sigprocmask(how, &set, &old)</code>	Examine or change the signal mask
<code>s = sigpending(set)</code>	Get the set of blocked signals
<code>s = sigsuspend(sigmask)</code>	Replace the signal mask and suspend the process
<code>s = kill(pid, sig)</code>	Send a signal to a process
<code>residual = alarm(seconds)</code>	Set the alarm clock
<code>s = pause()</code>	Suspend the caller until the next signal

Figure 10-6. Some system calls relating to processes. The return code *s* is `-1` if an error has occurred, *pid* is a process ID, and *residual* is the remaining time in the previous alarm. The parameters are what the names suggest.

In the case of the shell, the child process must execute the command typed by the user. It does this by using the `exec` system call, which causes its entire core image to be replaced by the file named in its first parameter. A highly simplified shell illustrating the use of `fork`, `waitpid`, and `exec` is shown in Fig. 10-7.

In the most general case, `exec` has three parameters: the name of the file to be executed, a pointer to the argument array, and a pointer to the environment array. These will be described shortly. Various library procedures, such as `execl`, `execv`, `execle`, and `execve`, are provided to allow the parameters to be omitted or specified in various ways. All of these procedures invoke the same underlying system call. Although the system call is `exec`, there is no library procedure with this name; one of the others must be used.

Let us consider the case of a command typed to the shell, such as

```
cp file1 file2
```

used to copy *file1* to *file2*. After the shell has forked, the child locates and executes the file *cp* and passes it information about the files to be copied.

The main program of *cp* (and many other programs) contains the function declaration

```
main(argc, argv, envp)
```

where *argc* is a count of the number of items on the command line, including the program name. For the example above, *argc* is 3.

The second parameter, *argv*, is a pointer to an array. Element *i* of that array is a pointer to the *i*th string on the command line. In our example, *argv*[0] would point

```

while (TRUE) {                                /* repeat forever */
    type_prompt( );                          /* display prompt on the screen */
    read_command(command, params);           /* read input line from keyboard */

    pid = fork( );                          /* fork off a child process */
    if (pid < 0) {
        printf("Unable to fork0);          /* error condition */
        continue;                          /* repeat the loop */
    }

    if (pid != 0) {
        waitpid (-1, &status, 0);          /* parent waits for child */
    } else {
        execve(command, params, 0);         /* child does the work */
    }
}

```

Figure 10-7. A highly simplified shell.

to the two-character string “cp”. Similarly, *argv*[1] would point to the five-character string “file1” and *argv*[2] would point to the five-character string “file2”.

The third parameter of *main*, *envp*, is a pointer to the environment, an array of strings containing assignments of the form *name = value* used to pass information such as the terminal type and home directory name to a program. In Fig. 10-7, no environment is passed to the child, so that the third parameter of *execve* is a zero in this case.

If *exec* seems complicated, do not despair; it is the most complex system call. All the rest are much simpler. As an example of a simple one, consider *exit*, which processes should use when they are finished executing. It has one parameter, the exit status (0 to 255), which is returned to the parent in the variable *status* of the *waitpid* system call. The low-order byte of *status* contains the termination status, with 0 being normal termination and the other values being various error conditions. The high-order byte contains the child’s exit status (0 to 255), as specified in the child’s call to *exit*. For example, if a parent process executes the statement

```
n = waitpid(-1, &status, 0);
```

it will be suspended until some child process terminates. If the child exits with, say, 4 as the parameter to *exit*, the parent will be awakened with *n* set to the child’s PID and *status* set to 0x0400 (0x as a prefix means hexadecimal in C). The low-order byte of *status* relates to signals; the next one is the value the child returned in its call to *exit*.

If a process exits and its parent has not yet waited for it, the process enters a kind of suspended animation called the **zombie state**—the living dead. When the parent finally waits for it, the process terminates.

Several system calls relate to signals, which are used in a variety of ways. For example, if a user accidentally tells a text editor to display the entire contents of a very long file, and then realizes the error, some way is needed to interrupt the editor. The usual choice is for the user to hit some special key (e.g., DEL or CTRL-C), which sends a signal to the editor. The editor catches the signal and stops the print-out.

To announce its willingness to catch this (or any other) signal, the process can use the `sigaction` system call. The first parameter is the signal to be caught (see Fig. 10-5). The second is a pointer to a structure giving a pointer to the signal-handling procedure, as well as some other bits and flags. The third one points to a structure where the system returns information about signal handling currently in effect, in case it must be restored later.

The signal handler may run for as long as it wants to. In practice, though, signal handlers are usually fairly short. When the signal-handling procedure is done, it returns to the point from which it was interrupted.

The `sigaction` system call can also be used to cause a signal to be ignored, or to restore the default action, which is killing the process.

Hitting the DEL key is not the only way to send a signal. The `kill` system call allows a process to signal another related process. The choice of the name “kill” for this system call is not an especially good one, since most processes send signals to other ones with the intention that they be caught. However, a signal that is not caught, does, indeed, kill the recipient.

For many real-time applications, a process needs to be interrupted after a specific time interval to do something, such as to retransmit a potentially lost packet over an unreliable communication line. To handle this situation, the `alarm` system call has been provided. The parameter specifies an interval, in seconds, after which a `SIGALRM` signal is sent to the process. A process may have only one alarm outstanding at any instant. If an alarm call is made with a parameter of 10 seconds, and then 3 seconds later another alarm call is made with a parameter of 20 seconds, only one signal will be generated, 20 seconds after the second call. The first signal is canceled by the second call to alarm. If the parameter to alarm is zero, any pending alarm signal is canceled. If an alarm signal is not caught, the default action is taken and the signaled process is killed. Technically, alarm signals may be ignored, but that is a pointless thing to do. Why would a program ask to be signaled later on and then ignore the signal?

It sometimes occurs that a process has nothing to do until a signal arrives. For example, consider a computer-aided instruction program that is testing reading speed and comprehension. It displays some text on the screen and then calls `alarm` to signal it after 30 seconds. While the student is reading the text, the program has nothing to do. It could sit in a tight loop doing nothing, but that would waste CPU time that a background process or other user might need. A better solution is to use the `pause` system call, which tells Linux to suspend the process until the next signal arrives. Woe be it to the program that calls `pause` with no alarm pending.

10.3.3 Implementation of Processes and Threads in Linux

A process in Linux is like an iceberg: you only see the part above the water, but there is also an important part underneath. Every process has a user part that runs the user program. However, when one of its threads makes a system call, it traps to kernel mode and begins running in kernel context, with a different memory map and full access to all machine resources. It is still the same thread, but now with more power and also its own kernel mode stack and kernel mode program counter. These are important because a system call can block partway through, for example, waiting for a disk operation to complete. The program counter and registers are then saved so the thread can be restarted in kernel mode later.

The Linux kernel internally represents processes as **tasks**, via the structure *task_struct*. Unlike other OS approaches (which make a distinction between a process, lightweight process, and thread), Linux uses the task structure to represent any execution context. Therefore, a single-threaded process will be represented with one task structure and a multithreaded process will have one task structure for each of the user-level threads. Finally, the kernel itself is multithreaded, and has kernel-level threads which are not associated with any user process and are executing kernel code. We will return to the treatment of multithreaded processes (and threads in general) later in this section.

For each process, a process descriptor of type *task_struct* is resident in memory at all times. It contains vital information needed for the kernel's management of all processes, including scheduling parameters, lists of open-file descriptors, and so on. The process descriptor along with memory for the kernel-mode stack for the process are created upon process creation.

For compatibility with other UNIX systems, Linux identifies processes via the PID. The kernel organizes all processes in a doubly linked list of task structures. In addition to accessing process descriptors by traversing the linked lists, the PID can be mapped to the address of the task structure, and the process information can be accessed immediately.

The task structure contains a variety of fields. Some of these fields contain pointers to other data structures or segments, such as those containing information about open files. Some of these segments are related to the user-level structure of the process, which is not of interest when the user process is not runnable. Therefore, these may be swapped or paged out, in order not to waste memory on information that is not needed. For example, although it is possible for a process to be sent a signal while it is swapped out, it is not possible for it to read a file. For this reason, information about signals must be in memory all the time, even when the process is not present in memory. On the other hand, information about file descriptors can be kept in the user structure and brought in only when the process is in memory and runnable.

The information in the process descriptor falls into a number of broad categories that can be roughly described as follows:

1. **Scheduling parameters.** Process priority, amount of CPU time consumed recently, amount of time spent sleeping recently. Together, these are used to determine which process to run next.
2. **Memory image.** Pointers to the text, data, and stack segments, or page tables. If the text segment is shared, the text pointer points to the shared text table. When the process is not in memory, information about how to find its parts on disk is here too.
3. **Signals.** Masks showing which signals are being ignored, which are being caught, which are being temporarily blocked, and which are in the process of being delivered.
4. **Machine registers.** When a trap to the kernel occurs, the machine registers (including the floating-point ones, if used) are saved here.
5. **System call state.** Information about the current system call, including the parameters, and results.
6. **File descriptor table.** When a system call involving a file descriptor is invoked, the file descriptor is used as an index into this table to locate the in-core data structure (i-node) corresponding to this file.
7. **Accounting.** Pointer to a table that keeps track of the user and system CPU time used by the process. Some systems also maintain limits here on the amount of CPU time a process may use, the maximum size of its stack, the number of page frames it may consume, and other items.
8. **Kernel stack.** A fixed stack for use by the kernel part of the process.
9. **Miscellaneous.** Current process state, event being waited for, if any, time until alarm clock goes off, PID, PID of the parent process, and user and group identification.

Keeping this information in mind, it is now easy to explain how processes are created in Linux. The mechanism for creating a new process is actually fairly straightforward. A new process descriptor and user area are created for the child process and filled in largely from the parent. The child is given a PID, its memory map is set up, and it is given shared access to its parent's files. Then its registers are set up and it is ready to run.

When a fork system call is executed, the calling process traps to the kernel and creates a task structure and few other accompanying data structures, such as the kernel-mode stack and a *thread_info* structure. This structure is allocated at a fixed offset from the process' end-of-stack, and contains few process parameters, along

with the address of the process descriptor. By storing the process descriptor's address at a fixed location, Linux needs only few efficient operations to locate the task structure for a running process.

The majority of the process-descriptor contents are filled out based on the parent's descriptor values. Linux then looks for an available PID, that is, not one currently in use by any process, and updates the PID hash-table entry to point to the new task structure. In case of collisions in the hash table, process descriptors may be chained. It also sets the fields in the *task_struct* to point to the corresponding previous/next process on the task array.

In principle, it should now allocate memory for the child's data and stack segments, and to make exact copies of the parent's segments, since the semantics of fork say that no memory is shared between parent and child. The text segment may be either copied or shared since it is read only. At this point, the child is ready to run.

However, copying memory is expensive, so all modern Linux systems cheat. They give the child its own page tables, but have them point to the parent's pages, only marked read only. Whenever either process (the child or the parent) tries to write on a page, it gets a protection fault. The kernel sees this and then allocates a new copy of the page to the faulting process and marks it read/write. In this way, only pages that are actually written have to be copied. This mechanism is called **copy on write**. It has the additional benefit of not requiring two copies of the program in memory, thus saving RAM.

After the child process starts running, the code running there (a copy of the shell in our example) does an `exec` system call giving the command name as a parameter. The kernel now finds and verifies the executable file, copies the arguments and environment strings to the kernel, and releases the old address space and its page tables.

Now the new address space must be created and filled in. If the system supports mapped files, as Linux and virtually all other UNIX-based systems do, the new page tables are set up to indicate that no pages are in memory, except perhaps one stack page, but that the address space is backed by the executable file on disk. When the new process starts running, it will immediately get a page fault, which will cause the first page of code to be paged in from the executable file. In this way, nothing has to be loaded in advance, so programs can start quickly and fault in just those pages they need and no more. (This strategy is really just demand paging in its most pure form, as we discussed in Chap. 3.) Finally, the arguments and environment strings are copied to the new stack, the signals are reset, and the registers are initialized to all zeros. At this point, the new command can start running.

Figure 10-8 illustrates the steps described above through the following example: A user types a command, `ls`, on the terminal, the shell creates a new process by forking off a clone of itself. The new shell then calls `exec` to overlay its memory with the contents of the executable file `ls`. After that, `ls` can start.

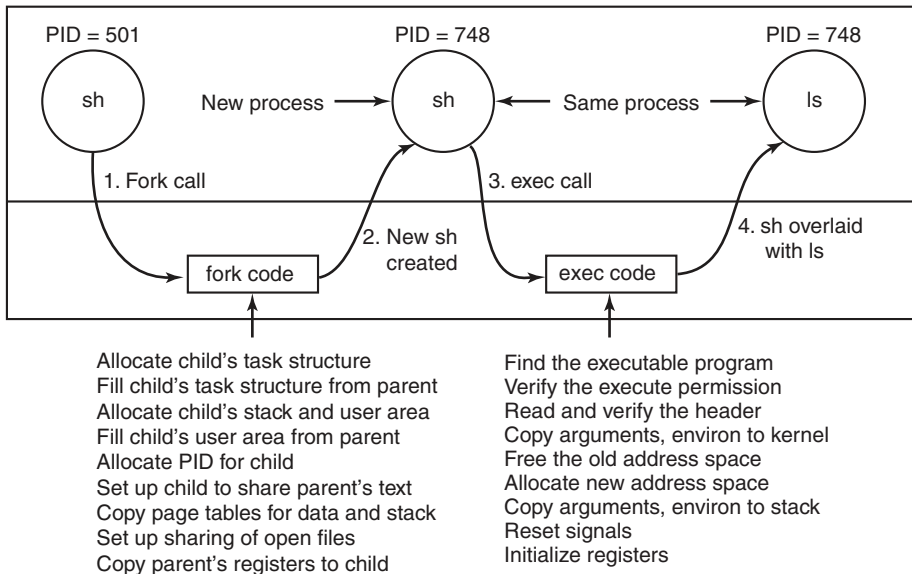


Figure 10-8. The steps in executing the command `ls` typed to the shell.

Threads in Linux

We discussed threads in a general way in Chap. 2. Here we will focus on kernel threads in Linux, particularly on the differences among the Linux thread model and other UNIX systems. In order to better understand the unique capabilities provided by the Linux model, we start with a discussion of some of the challenging decisions present in multithreaded systems.

The main issue in introducing threads is maintaining the correct traditional UNIX semantics. First consider `fork`. Suppose that a process with multiple (kernel) threads does a `fork` system call. Should all the other threads be created in the new process? For the moment, let us answer that question with yes. Suppose that one of the other threads was blocked reading from the keyboard. Should the corresponding thread in the new process also be blocked reading from the keyboard? If so, which one gets the next line typed? If not, what should that thread be doing in the new process?

The same problem holds for many other things threads can do. In a single-threaded process, the problem does not arise because the one and only thread cannot be blocked when calling `fork`. Now consider the case that the other threads are not created in the child process. Suppose that one of the not-created threads holds a mutex that the one-and-only thread in the new process tries to acquire after doing the `fork`. The mutex will never be released and the one thread will hang forever. Numerous other problems exist, too. There is no simple solution.

File I/O is another problem area. Suppose that one thread is blocked reading from a file and another thread closes the file or does an `lseek` to change the current file pointer. What happens next? Who knows?

Signal handling is another thorny issue. Should signals be directed at a specific thread or just at the process? A `SIGFPE` (floating-point exception) should probably be caught by the thread that caused it. What if it does not catch it? Should just that thread be killed, or all threads? Now consider the `SIGINT` signal, generated by the user at the keyboard. Which thread should catch that? Should all threads share a common set of signal masks? All solutions to these and other problems usually cause something to break somewhere. Getting the semantics of threads right (not to mention the code) is a nontrivial business.

Linux supports kernel threads in an interesting way that is worth looking at. The implementation is based on ideas from 4.4BSD, but kernel threads were not enabled in that distribution because Berkeley ran out of money before the C library could be rewritten to solve the problems discussed above.

Historically, processes were resource containers and threads were the units of execution. A process contained one or more threads that shared the address space, open files, signal handlers, alarms, and everything else. Everything was clear and simple as described above.

In 2000, Linux introduced a powerful new system call, `clone`, that blurred the distinction between processes and threads and possibly even inverted the primacy of the two concepts. `Clone` is not present in any other version of UNIX. Classically, when a new thread was created, the original thread(s) and the new one shared everything but their registers. In particular, file descriptors for open files, signal handlers, alarms, and other global properties were per process, not per thread. What `clone` did was make it possible for each of these aspects and others to be process specific or thread specific. It is called as follows:

```
pid = clone(function, stack_ptr, sharing_flags, arg);
```

The call creates a new thread, either in the current process or in a new process, depending on *sharing_flags*. If the new thread is in the current process, it shares the address space with the existing threads, and every subsequent write to any byte in the address space by any thread is immediately visible to all the other threads in the process. On the other hand, if the address space is not shared, then the new thread gets an exact copy of the address space, but subsequent writes by the new thread are not visible to the old ones. These semantics are the same as POSIX `fork`.

In both cases, the new thread begins executing at *function*, which is called with *arg* as its only parameter. Also in both cases, the new thread gets its own private stack, with the stack pointer initialized to *stack_ptr*.

The *sharing_flags* parameter is a bitmap that allows a finer grain of sharing than traditional UNIX systems. Each of the bits can be set independently of the other ones, and each of them determines whether the new thread copies some data

structure or shares it with the calling thread. Fig. 10-9 shows some of the items that can be shared or copied according to bits in *sharing_flags*.

Flag	Meaning when set	Meaning when cleared
CLONE_VM	Create a new thread	Create a new process
CLONE_FS	Share umask, root, and working dirs	Do not share them
CLONE_FILES	Share the file descriptors	Copy the file descriptors
CLONE_SIGHAND	Share the signal handler table	Copy the table
CLONE_PARENT	New thread has same parent as the caller	New thread's parent is caller

Figure 10-9. Bits in the *sharing_flags* bitmap.

The *CLONE_VM* bit determines whether the virtual memory (i.e., address space) is shared with the old threads or copied. If it is set, the new thread just moves in with the existing ones, so the clone call effectively creates a new thread in an existing process. If the bit is cleared, the new thread gets its own private address space. Having its own address space means that the effect of its STORE instructions is not visible to the existing threads. This behavior is similar to fork, except as noted below. Creating a new address space is effectively the definition of a new process.

The *CLONE_FS* bit controls sharing of the root and working directories and of the umask flag. Even if the new thread has its own address space, if this bit is set, the old and new threads share working directories. This means that a call to *chdir* by one thread changes the working directory of the other thread, even though the other thread may have its own address space. In UNIX, a call to *chdir* by a thread always changes the working directory for other threads in its process, but never for threads in another process. Thus this bit enables a kind of sharing not possible in traditional UNIX versions.

The *CLONE_FILES* bit is analogous to the *CLONE_FS* bit. If set, the new thread shares its file descriptors with the old ones, so calls to *lseek* by one thread are visible to the other ones, again as normally holds for threads within the same process but not for threads in different processes. Similarly, *CLONE_SIGHAND* enables or disables the sharing of the signal handler table between the old and new threads. If the table is shared, even among threads in different address spaces, then changing a handler in one thread affects the handlers in the others.

Finally, every process has a parent. The *CLONE_PARENT* bit controls who the parent of the new thread is. It can either be the same as the calling thread (in which case the new thread is a sibling of the caller) or it can be the calling thread itself, in which case the new thread is a child of the caller. There are a few other bits that control other items, but they are less important.

This fine-grained sharing is possible because Linux maintains separate data structures for the various items listed in Sec. 10.3.3 (scheduling parameters, memory image, and so on). The task structure just points to these data structures, so it

is easy to make a new task structure for each cloned thread and have it point either to the old thread's scheduling, memory, and other data structures or to copies of them. The fact that such fine-grained sharing is possible does not mean that it is useful, however, especially since traditional UNIX versions do not offer this functionality. A Linux program that takes advantage of it is then no longer portable to UNIX.

The Linux thread model raises another difficulty. UNIX systems associate a single PID with a process, independent of whether it is single- or multithreaded. In order to be compatible with other UNIX systems, Linux distinguishes between a process identifier (PID) and a task identifier (TID). Both fields are stored in the task structure. When `clone` is used to create a new process that shares nothing with its creator, PID is set to a new value; otherwise, the task receives a new TID, but inherits the PID. In this manner all threads in a process will receive the same PID as the first thread in the process.

10.3.4 Scheduling in Linux

We will now look at the Linux scheduling algorithm. To start with, Linux threads are kernel threads, so scheduling is based on threads, not processes.

Linux distinguishes three classes of threads for scheduling purposes:

1. Real-time FIFO.
2. Real-time round robin.
3. Timesharing.

Real-time FIFO threads are the highest priority and are not preemptable except by a newly readied real-time FIFO thread with even higher priority. Real-time round-robin threads are the same as real-time FIFO threads except that they have time quanta associated with them, and are preemptable by the clock. If multiple real-time round-robin threads are ready, each one is run for its quantum, after which it goes to the end of the list of real-time round-robin threads. Neither of these classes is actually real time in any sense. Deadlines cannot be specified and guarantees are not given. These classes are simply higher priority than threads in the standard timesharing class. The reason Linux calls them real time is that Linux is conformant to the P1003.4 standard ("real-time" extensions to UNIX) which uses those names. The real-time threads are internally represented with priority levels from 0 to 99, 0 being the highest and 99 the lowest real-time priority level.

The conventional, non-real-time threads form a separate class and are scheduled by a separate algorithm so they do not compete with the real-time threads. Internally, these threads are associated with priority levels from 100 to 139, that is, Linux internally distinguishes among 140 priority levels (for real-time and non-real-time tasks). As for the real-time round-robin threads, Linux allocates CPU time to the non-real-time tasks based on their requirements and their priority levels.

In Linux, time is measured as the number of clock ticks. In older Linux versions, the clock ran at 1000Hz and each tick was 1ms, called a **jiffy**. In newer versions, the tick frequency can be configured to 500, 250 or even 1Hz. In order to avoid wasting CPU cycles for servicing the timer interrupt, the kernel can even be configured in “tickless” mode. This is useful when there is only one process running in the system, or when the CPU is idle and needs to go into power-saving mode. Finally, on newer systems, **high-resolution timers** allow the kernel to keep track of time in sub-jiffy granularity.

Like most UNIX systems, Linux associates a nice value with each thread. The default is 0, but this can be changed using the `nice(value)` system call, where value ranges from -20 to +19. This value determines the static priority of each thread. A user computing π to a billion places in the background might put this call in his program to be nice to the other users. Only the system administrator may ask for *better* than normal service (meaning values from -20 to -1). Deducing the reason for this rule is left as an exercise for the reader.

Next, we will describe in more detail two of the Linux scheduling algorithms. Their internals are closely related to the design of the **runqueue**, a key data structure used by the scheduler to track all runnable tasks in the system and select the next one to run. A runqueue is associated with each CPU in the system.

Historically, a popular Linux scheduler was the Linux **O(1) scheduler**. It received its name because it was able to perform task-management operations, such as selecting a task or enqueueing a task on the runqueue, in constant time, independent of the total number of tasks in the system. In the O(1) scheduler, the runqueue is organized in two arrays, *active* and *expired*. As shown in Fig. 10-10(a), each of these is an array of 140 list heads, each corresponding to a different priority. Each list head points to a doubly linked list of processes at a given priority. The basic operation of the scheduler can be described as follows.

The scheduler selects a task from the highest-priority list in the active array. If that task’s timeslice (quantum) expires, it is moved to the expired list (potentially at a different priority level). If the task blocks, for instance to wait on an I/O event, before its timeslice expires, once the event occurs and its execution can resume, it is placed back on the original active array, and its timeslice is decremented to reflect the CPU time it already used. Once its timeslice is fully exhausted, it, too, will be placed on the expired array. When there are no more tasks in the active array, the scheduler simply swaps the pointers, so the expired arrays now become active, and vice versa. This method ensures that low-priority tasks will not starve (except when real-time FIFO threads completely hog the CPU, which is unlikely).

Here, different priority levels are assigned different timeslice values, with higher quanta assigned to higher-priority processes. For instance, tasks running at priority level 100 will receive time quanta of 800 msec, whereas tasks at priority level of 139 will receive 5 msec.

The idea behind this scheme is to get processes out of the kernel fast. If a process is trying to read a disk file, making it wait a second between read calls will

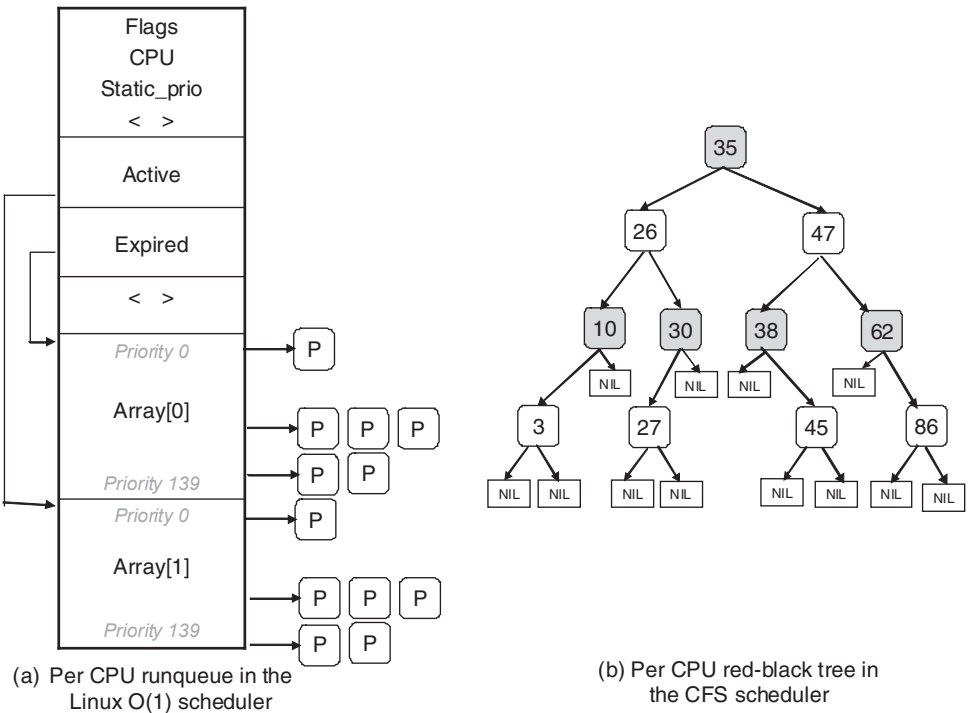


Figure 10-10. Illustration of Linux runqueue data structures for (a) the Linux O(1) scheduler, and (b) the Completely Fair Scheduler.

slow it down enormously. It is far better to let it run immediately after each request is completed, so that it can make the next one quickly. Similarly, if a process was blocked waiting for keyboard input, it is clearly an interactive process, and as such should be given a high priority as soon as it is ready in order to ensure that interactive processes get good service. In this light, CPU-bound processes basically get any service that is left over when all the I/O bound and interactive processes are blocked.

Since Linux (or any other OS) does not know a priori whether a task is I/O- or CPU-bound, it relies on continuously maintaining interactivity heuristics. In this manner, Linux distinguishes between static and dynamic priority. The threads' dynamic priority is continuously recalculated, so as to (1) reward interactive threads, and (2) punish CPU-hogging threads. In the O(1) scheduler, the maximum priority bonus is -5 , since lower-priority values correspond to higher priority received by the scheduler. The maximum priority penalty is $+5$. The scheduler maintains a *sleep_avg* variable associated with each task. Whenever a task is awakened, this variable is incremented. Whenever a task is preempted or when its quantum expires, this variable is decremented by the corresponding value. This value is used

to dynamically map the task's bonus to values from -5 to $+5$. The scheduler recalculates the new priority level as a thread is moved from the active to the expired list.

The $O(1)$ scheduling algorithm refers to the scheduler made popular in the early versions of the 2.6 kernel, and was first introduced in the unstable 2.5 kernel. Prior algorithms exhibited poor performance in multiprocessor settings and did not scale well with an increased number of tasks. Since the description presented in the above paragraphs indicates that a scheduling decision can be made through access to the appropriate active list, it can be done in constant $O(1)$ time, independent of the number of processes in the system. However, in spite of the desirable property of constant-time operation, the $O(1)$ scheduler had significant shortcomings. Most notably, the heuristics used to determine the interactivity of a task, and therefore its priority level, were complex and imperfect, and resulted in poor performance for interactive tasks.

To address this issue, Ingo Molnar, who also created the $O(1)$ scheduler, proposed a new scheduler called **Completely Fair Scheduler** or **CFS**. CFS was based on ideas originally developed by Con Kolivas for an earlier scheduler, and was first integrated into the 2.6.23 release of the kernel. It is still the default scheduler for the non-real-time tasks.

The main idea behind CFS is to use a *red-black tree* as the runqueue data structure. Tasks are ordered in the tree based on the amount of time they spend running on the CPU, called *vruntime*. CFS accounts for the tasks' running time with nanosecond granularity. As shown in Fig. 10-10(b), each internal node in the tree corresponds to a task. The children to the left correspond to tasks which had less time on the CPU, and therefore will be scheduled sooner, and the children to the right on the node are those that have consumed more CPU time thus far. The leaves in the tree do not play any role in the scheduler.

The scheduling algorithm can be summarized as follows. CFS always schedules the task which has had least amount of time on the CPU, typically the leftmost node in the tree. Periodically, CFS increments the task's *vruntime* value based on the time it has already run, and compares this to the current leftmost node in the tree. If the running task still has smaller *vruntime*, it will continue to run. Otherwise, it will be inserted at the appropriate place in the red-black tree, and the CPU will be given to task corresponding to the new leftmost node.

To account for differences in task priorities and "niceness," CFS changes the effective rate at which a task's virtual time passes when it is running on the CPU. For lower-priority tasks, time passes more quickly, their *vruntime* value will increase more rapidly, and, depending on other tasks in the system, they will lose the CPU and be reinserted in the tree sooner than if they had a higher priority value. In this manner, CFS avoids using separate runqueue structures for different priority levels.

In summary, selecting a node to run can be done in constant time, whereas inserting a task in the runqueue is done in $O(\log(N))$ time, where N is the number

of tasks in the system. Given the levels of load in current systems, this continues to be acceptable, but as the compute capacity of the nodes, and the number of tasks they can run, increase, particularly in the server space, it is possible that new scheduling algorithms will be proposed in the future.

Besides the basic scheduling algorithm, the Linux scheduler includes special features particularly useful for multiprocessor or multicore platforms. First, the runqueue structure is associated with each CPU in the multiprocessing platform. The scheduler tries to maintain benefits from affinity scheduling, and to schedule tasks on the CPU on which they were previously executing. Second, a set of system calls is available to further specify or modify the affinity requirements of a select thread. Finally, the scheduler performs periodic load balancing across runqueues of different CPUs to ensure that the system load is well balanced, while still meeting certain performance or affinity requirements.

The scheduler considers only runnable tasks, which are placed on the appropriate runqueue. Tasks which are not runnable and are waiting on various I/O operations or other kernel events are placed on another data structure, **waitqueue**. A waitqueue is associated with each event that tasks may wait on. The head of the waitqueue includes a pointer to a linked list of tasks and a spinlock. The spinlock is necessary so as to ensure that the waitqueue can be concurrently manipulated through both the main kernel code and interrupt handlers or other asynchronous invocations.

Synchronization in Linux

In the previous section we mentioned that Linux uses spinlocks to prevent concurrent modifications to data structures like the waitqueues. In fact, the kernel code contains synchronization variables in numerous locations. We will next briefly summarize the synchronization constructs available in Linux.

Earlier Linux kernels had just one **big kernel lock**. This proved highly inefficient, particularly on multiprocessor platforms, since it prevented processes on different CPUs from executing kernel code concurrently. Hence, many new synchronization points were introduced at much finer granularity.

Linux provides several types of synchronization variables, both used internally in the kernel, and available to user-level applications and libraries. At the lowest level, Linux provides wrappers around the hardware-supported atomic instructions, via operations such as `atomic_set` and `atomic_read`. In addition, since modern hardware reorders memory operations, Linux provides memory barriers. Using operations like `rmb` and `wmb` guarantees that all read/write memory operations preceding the barrier call have completed before any subsequent accesses take place.

More commonly used synchronization constructs are the higher-level ones. Threads that do not wish to block (for performance or correctness reasons) use spinlocks and spin read/write locks. The current Linux version implements the so-called “ticket-based” spinlock, which has excellent performance on SMP and

multicore systems. Threads that are allowed to or need to block use constructs like mutexes and semaphores. Linux supports nonblocking calls like `mutex_trylock` and `sem_trywait` to determine the status of the synchronization variable without blocking. Other types of synchronization variables, like `futexes`, completions, “read-copy-update” (RCU) locks, etc., are also supported. Finally, synchronization between the kernel and the code executed by interrupt-handling routines can also be achieved by dynamically disabling and enabling the corresponding interrupts.

10.3.5 Booting Linux

Details vary from platform to platform, but in general the following steps represent the boot process. When the computer starts, the BIOS performs Power-On-Self-Test (POST) and initial device discovery and initialization, since the OS’ boot process may rely on access to disks, screens, keyboards, and so on. Next, the first sector of the boot disk, the **MBR (Master Boot Record)**, is read into a fixed memory location and executed. This sector contains a small (512-byte) program that loads a standalone program called **boot** from the boot device, such as a SATA or SCSI disk. The *boot* program first copies itself to a fixed high-memory address to free up low memory for the operating system.

Once moved, *boot* reads the root directory of the boot device. To do this, it must understand the file system and directory format, which is the case with some bootloaders such as **GRUB (GRand Unified Bootloader)**. Other popular bootloaders, such as Intel’s LILO, do not rely on any specific file system. Instead, they need a block map and low-level addresses, which describe physical sectors, heads, and cylinders, to find the relevant sectors to be loaded.

Then *boot* reads in the operating system kernel and jumps to it. At this point, it has finished its job and the kernel is running.

The kernel start-up code is written in assembly language and is highly machine dependent. Typical work includes setting up the kernel stack, identifying the CPU type, calculating the amount of RAM present, disabling interrupts, enabling the MMU, and finally calling the C-language *main* procedure to start the main part of the operating system.

The C code also has considerable initialization to do, but this is more logical than physical. It starts out by allocating a message buffer to help debug boot problems. As initialization proceeds, messages are written here about what is happening, so that they can be fished out after a boot failure by a special diagnostic program. Think of this as the operating system’s cockpit flight recorder (the black box investigators look for after a plane crash).

Next the kernel data structures are allocated. Most are of fixed size, but a few, such as the page cache and certain page table structures, depend on the amount of RAM available.

At this point the system begins autoconfiguration. Using configuration files telling what kinds of I/O devices might be present, it begins probing the devices to

see which ones actually are present. If a probed device responds to the probe, it is added to a table of attached devices. If it fails to respond, it is assumed to be absent and ignored henceforth. Unlike traditional UNIX versions, Linux device drivers do not need to be statically linked and may be loaded dynamically (as can be done in all versions of MS-DOS and Windows, incidentally).

The arguments for and against dynamically loading drivers are interesting and worth stating explicitly. The main argument for dynamic loading is that a single binary can be shipped to customers with divergent configurations and have it automatically load the drivers it needs, possibly even over a network. The main argument against dynamic loading is security. If you are running a secure site, such as a bank's database or a corporate Web server, you probably want to make it impossible for anyone to insert random code into the kernel. The system administrator may keep the operating system sources and object files on a secure machine, do all system builds there, and ship the kernel binary to other machines over a local area network. If drivers cannot be loaded dynamically, this scenario prevents machine operators and others who know the superuser password from injecting malicious or buggy code into the kernel. Furthermore, at large sites, the hardware configuration is known exactly at the time the system is compiled and linked. Changes are sufficiently rare that having to relink the system when a new hardware device is added is not an issue.

Once all the hardware has been configured, the next thing to do is to carefully handcraft process 0, set up its stack, and run it. Process 0 continues initialization, doing things like programming the real-time clock, mounting the root file system, and creating *init* (process 1) and the page daemon (process 2).

Init checks its flags to see if it is supposed to come up single user or multiuser. In the former case, it forks off a process that executes the shell and waits for this process to exit. In the latter case, it forks off a process that executes the system initialization shell script, */etc/rc*, which can do file system consistency checks, mount additional file systems, start daemon processes, and so on. Then it reads */etc/tty*s, which lists the terminals and some of their properties. For each enabled terminal, it forks off a copy of itself, which does some housekeeping and then executes a program called *getty*.

Getty sets the line speed and other properties for each line (some of which may be modems, for example), and then displays

login:

on the terminal's screen and tries to read the user's name from the keyboard. When someone sits down at the terminal and provides a login name, *getty* terminates by executing */bin/login*, the login program. *Login* then asks for a password, encrypts it, and verifies it against the encrypted password stored in the password file, */etc/passwd*. If it is correct, *login* replaces itself with the user's shell, which then waits for the first command. If it is incorrect, *login* just asks for another user name. This mechanism is shown in Fig. 10-11 for a system with three terminals.

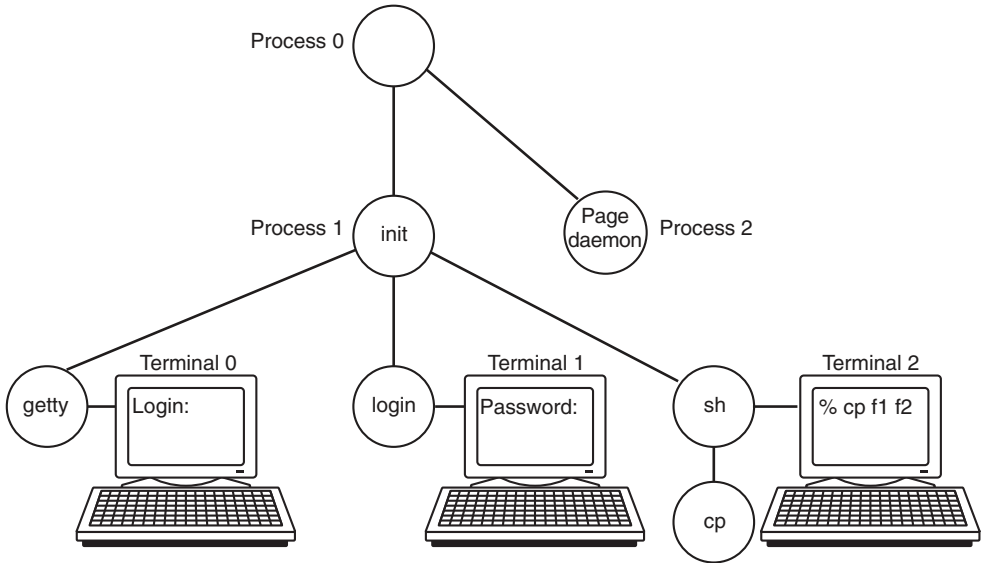


Figure 10-11. The sequence of processes used to boot some Linux systems.

In the figure, the *getty* process running for terminal 0 is still waiting for input. On terminal 1, a user has typed a login name, so *getty* has overwritten itself with *login*, which is asking for the password. A successful login has already occurred on terminal 2, causing the shell to type the prompt (%). The user then typed

`cp f1 f2`

which has caused the shell to fork off a child process and have that process execute the *cp* program. The shell is blocked, waiting for the child to terminate, at which time the shell will type another prompt and read from the keyboard. If the user at terminal 2 had typed *cc* instead of *cp*, the main program of the C compiler would have been started, which in turn would have forked off more processes to run the various compiler passes.

10.4 MEMORY MANAGEMENT IN LINUX

The Linux memory model is straightforward, to make programs portable and to make it possible to implement Linux on machines with widely differing memory management units, ranging from essentially nothing (e.g., the original IBM PC) to sophisticated paging hardware. This is an area of the design that has barely changed in decades. It has worked well so it has not needed much revision. We will now examine the model and how it is implemented.

10.4.1 Fundamental Concepts

Every Linux process has an address space that logically consists of three segments: text, data, and stack. An example process' address space is illustrated in Fig. 10-12(a) as process A. The **text segment** contains the machine instructions that form the program's executable code. It is produced by the compiler and assembler by translating the C, C++, or other program into machine code. The text segment is normally read-only. Self-modifying programs went out of style in about 1950 because they were too difficult to understand and debug. Thus the text segment neither grows nor shrinks nor changes in any other way.

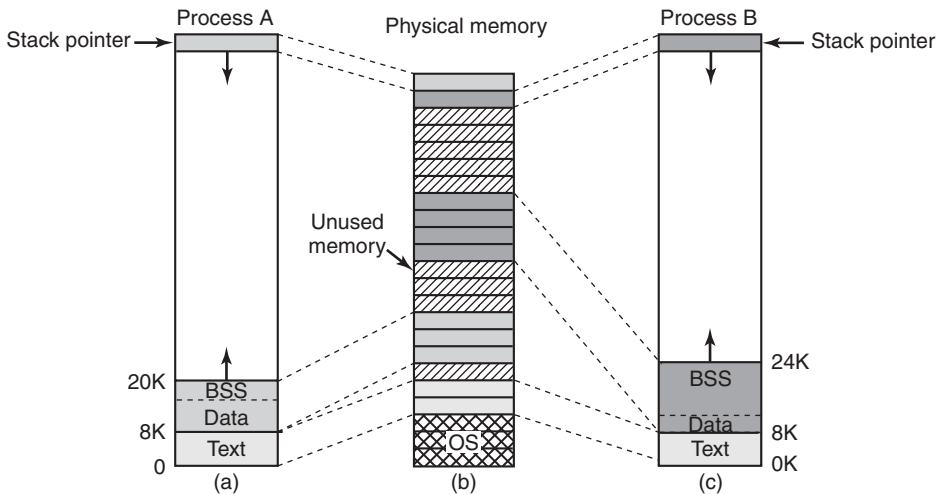


Figure 10-12. (a) Process A's virtual address space. (b) Physical memory. (c) Process B's virtual address space.

The **data segment** contains storage for all the program's variables, strings, arrays, and other data. It has two parts, the initialized data and the uninitialized data. For historical reasons, the latter is known as the **BSS** (historically called **Block Started by Symbol**). The initialized part of the data segment contains variables and compiler constants that need an initial value when the program is started. All the variables in the BSS part are initialized to zero after loading.

For example, in C it is possible to declare a character string and initialize it at the same time. When the program starts up, it expects that the string has its initial value. To implement this construction, the compiler assigns the string a location in the address space, and ensures that when the program is started up, this location contains the proper string. From the operating system's point of view, initialized data are not all that different from program text—both contain bit patterns produced by the compiler that must be loaded into memory when the program starts.

The existence of uninitialized data is actually just an optimization. When a global variable is not explicitly initialized, the semantics of the C language say that its

initial value is 0. In practice, most global variables are not initialized explicitly, and are thus 0. This could be implemented by simply having a section of the executable binary file exactly equal to the number of bytes of data, and initializing all of them, including the ones that have defaulted to 0.

However, to save space in the executable file, this is not done. Instead, the file contains all the explicitly initialized variables following the program text. The uninitialized variables are all gathered together after the initialized ones, so all the compiler has to do is put a word in the header telling how many bytes to allocate.

To make this point more explicit, consider Fig. 10-12(a) again. Here the program text is 8 KB and the initialized data is also 8 KB. The uninitialized data (BSS) is 4 KB. The executable file is only 16 KB (text + initialized data), plus a short header that tells the system to allocate another 4 KB after the initialized data and zero it before starting the program. This trick avoids storing 4 KB of zeros in the executable file.

In order to avoid allocating a physical page frame full of zeros, during initialization Linux allocates a static *zero page*, a write-protected page full of zeros. When a process is loaded, its uninitialized data region is set to point to the zero page. Whenever a process actually attempts to write in this area, the copy-on-write mechanism kicks in, and an actual page frame is allocated to the process.

Unlike the text segment, which cannot change, the data segment can change. Programs modify their variables all the time. Furthermore, many programs need to allocate space dynamically, during execution. Linux handles this by permitting the data segment to grow and shrink as memory is allocated and deallocated. A system call, `brk`, is available to allow a program to set the size of its data segment. Thus to allocate more memory, a program can increase the size of its data segment. The C library procedure `malloc`, commonly used to allocate memory, makes heavy use of it. The process address-space descriptor contains information on the range of dynamically allocated memory areas in the process, typically called the **heap**.

The third segment is the stack segment. On most machines, it starts at or near the top of the virtual address space and grows down toward 0. For instance, on 32bit x86 platforms, the stack starts at address `0xC0000000`, which is the 3-GB virtual address limit visible to the process in user mode. If the stack grows below the bottom of the stack segment, a hardware fault occurs and the operating system lowers the bottom of the stack segment by one page. Programs do not explicitly manage the size of the stack segment.

When a program starts up, its stack is not empty. Instead, it contains all the environment (shell) variables as well as the command line typed to the shell to invoke it. In this way, a program can discover its arguments. For example, when

```
cp src dest
```

is typed, the `cp` program is run with the string “`cp src dest`” on the stack, so it can find out the names of the source and destination files. The string is represented as an array of pointers to the symbols in the string, to make parsing easier.

When two users are running the same program, such as the editor, it would be possible, but inefficient, to keep two copies of the editor's program text in memory at once. Instead, Linux systems support **shared text segments**. In Fig. 10-12(a) and Fig. 10-12(c) we see two processes, *A* and *B*, that have the same text segment. In Fig. 10-12(b) we see a possible layout of physical memory, in which both processes share the same piece of text. The mapping is done by the virtual-memory hardware.

Data and stack segments are never shared except after a fork, and then only those pages that are not modified. If either one needs to grow and there is no room adjacent to it to grow into, there is no problem since adjacent virtual pages do not have to map onto adjacent physical pages.

On some computers, the hardware supports separate address spaces for instructions and data. When this feature is available, Linux can use it. For example, on a computer with 32-bit addresses, if this feature is available, there would be 2^{32} bits of address space for instructions and an additional 2^{32} bits of address space for the data and stack segments to share. A jump or branch to 0 goes to address 0 of text space, whereas a move from 0 uses address 0 in data space. This feature doubles the address space available.

In addition to dynamically allocating more memory, processes in Linux can access file data through **memory-mapped files**. This feature makes it possible to map a file onto a portion of a process' address space so that the file can be read and written as if it were a byte array in memory. Mapping a file in makes random access to it much easier than using I/O system calls such as `read` and `write`. Shared libraries are accessed by mapping them in using this mechanism. In Fig. 10-13 we see a file that is mapped into two processes at the same time, at different virtual addresses.

An additional advantage of mapping a file in is that two or more processes can map in the same file at the same time. Writes to the file by any one of them are then instantly visible to the others. In fact, by mapping in a scratch file (which will be discarded after all the processes exit), this mechanism provides a high-bandwidth way for multiple processes to share memory. In the most extreme case, two (or more) processes could map in a file that covers the entire address space, giving a form of sharing that is partway between separate processes and threads. Here the address space is shared (like threads), but each process maintains its own open files and signals, for example, which is not like threads. In practice, however, making two address spaces exactly correspond is never done.

10.4.2 Memory Management System Calls in Linux

POSIX does not specify any system calls for memory management. This topic was considered too machine dependent for standardization. Instead, the problem was swept under the rug by saying that programs needing dynamic memory management can use the *malloc* library procedure (defined by the ANSI C standard).

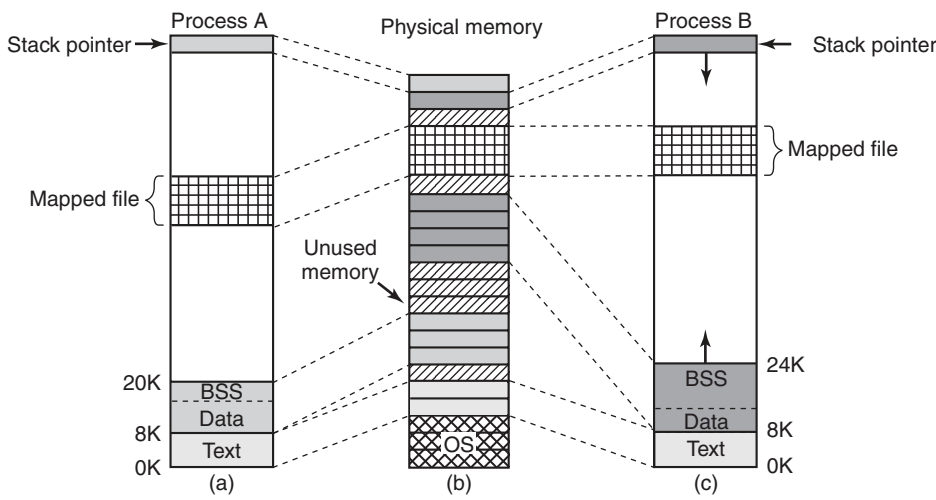


Figure 10-13. Two processes can share a mapped file.

How *malloc* is implemented is thus moved outside the scope of the POSIX standard. In some circles, this approach is known as passing the buck.

In practice, most Linux systems have system calls for managing memory. The most common ones are listed in Fig. 10-14. *Brk* specifies the size of the data segment by giving the address of the first byte beyond it. If the new value is greater than the old one, the data segment becomes larger; otherwise it shrinks.

System call	Description
<code>s = brk(addr)</code>	Change data segment size
<code>a = mmap(addr, len, prot, flags, fd, offset)</code>	Map a file in
<code>s = unmap(addr, len)</code>	Unmap a file

Figure 10-14. Some system calls relating to memory management. The return code *s* is `-1` if an error has occurred; *a* and *addr* are memory addresses, *len* is a length, *prot* controls protection, *flags* are miscellaneous bits, *fd* is a file descriptor, and *offset* is a file offset.

The *mmap* and *munmap* system calls control memory-mapped files. The first parameter to *mmap*, *addr*, determines the address at which the file (or portion thereof) is mapped. It must be a multiple of the page size. If this parameter is 0, the system determines the address itself and returns it in *a*. The second parameter, *len*, tells how many bytes to map. It, too, must be a multiple of the page size. The third parameter, *prot*, determines the protection for the mapped file. It can be marked readable, writable, executable, or some combination of these. The fourth parameter, *flags*, controls whether the file is private or sharable, and whether *addr* is a requirement or merely a hint. The fifth parameter, *fd*, is the file descriptor for

the file to be mapped. Only open files can be mapped, so to map a file in, it must first be opened. Finally, *offset* tells where in the file to begin the mapping. It is not necessary to start the mapping at byte 0; any page boundary will do.

The other call, `unmap`, removes a mapped file. If only a portion of the file is unmapped, the rest remains mapped.

10.4.3 Implementation of Memory Management in Linux

Each Linux process on a 32-bit machine typically gets 3 GB of virtual address space for itself, with the remaining 1 GB reserved for its page tables and other kernel data. The kernel's 1 GB is not visible when running in user mode, but becomes accessible when the process traps into the kernel. The kernel memory typically resides in low physical memory but it is mapped in the top 1 GB of each process virtual address space, between addresses `0xC0000000` and `0xFFFFFFFF` (3–4 GB). On current 64-bit x86 machines, only up to 48 bits are used for addressing, implying a theoretical limit of 256 TB for the size of the addressable memory. Linux splits this memory between the kernel and user space, resulting in a maximum 128 TB per-process virtual address space per process. The address space is created when the process is created and is overwritten on an `exec` system call.

In order to allow multiple processes to share the underlying physical memory, Linux monitors the use of the physical memory, allocates more memory as needed by user processes or kernel components, dynamically maps portions of the physical memory into the address space of different processes, and dynamically brings in and out of memory program executables, files, and other state information as necessary to utilize the platform resources efficiently and to ensure execution progress. The remainder of this section describes the implementation of various mechanisms in the Linux kernel which are responsible for these operations.

Physical Memory Management

Due to idiosyncratic hardware limitations on many systems, not all physical memory can be treated identically, especially with respect to I/O and virtual memory. Linux distinguishes between the following memory zones:

1. **ZONE_DMA** and **ZONE_DMA32**: pages that can be used for DMA.
2. **ZONE_NORMAL**: normal, regularly mapped pages.
3. **ZONE_HIGHMEM**: pages with high-memory addresses, which are not permanently mapped.

The exact boundaries and layout of the memory zones is architecture dependent. On x86 hardware, certain devices can perform DMA operations only in the first 16 MB of address space, hence **ZONE_DMA** is in the range 0–16 MB. On 64-bit machines there is additional support for those devices that can perform 32-bit DMA

operations, and `ZONE_DMA32` marks this region. In addition, if the hardware, like older-generation i386, cannot directly map memory addresses above 896 MB, `ZONE_HIGHMEM` corresponds to anything above this mark. `ZONE_NORMAL` is anything in between them. Therefore, on 32-bit x86 platforms, the first 896 MB of the Linux address space are directly mapped, whereas the remaining 128 MB of the kernel address space are used to access high memory regions. On x86_64 `ZONE_HIGHMEM` is not defined. The kernel maintains a *zone* structure for each of the three zones, and can perform memory allocations for the three zones separately.

Main memory in Linux consists of three parts. The first two parts, the kernel and memory map, are **pinned** in memory (i.e., never paged out). The rest of memory is divided into page frames, each of which can contain a text, data, or stack page, a page-table page, or be on the free list.

The kernel maintains a map of the main memory which contains all information about the use of the physical memory in the system, such as its zones, free page frames, and so forth. The information, illustrated in Fig. 10-15, is organized as follows.

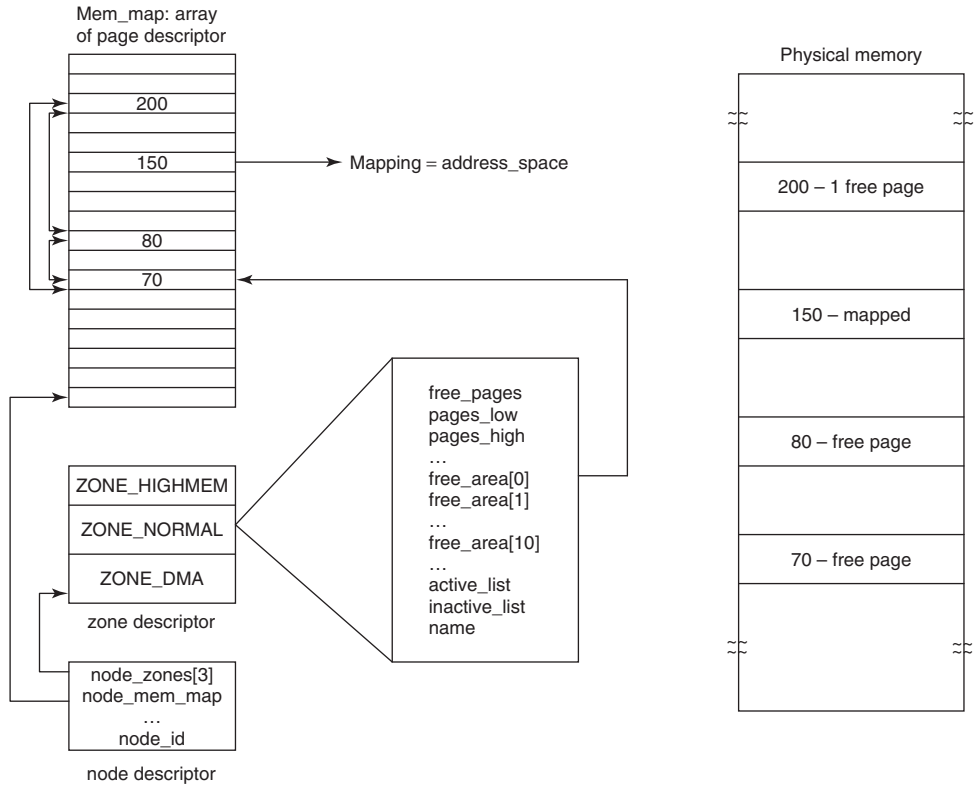


Figure 10-15. Linux main memory representation.

First of all, Linux maintains an array of **page descriptors**, of type *page* one for each physical page frame in the system, called *mem_map*. Each page descriptor contains a pointer to the address space that it belongs to, in case the page is not free, a pair of pointers which allow it to form doubly linked lists with other descriptors, for instance to keep together all free page frames, and a few other fields. In Fig. 10-15 the page descriptor for page 150 contains a mapping to the address space the page belongs to. Pages 70, 80, and 200 are free, and they are linked together. The size of the page descriptor is 32 bytes, therefore the entire *mem_map* can consume less than 1% of the physical memory (for a page frame of 4 KB).

Since the physical memory is divided into zones, for each zone Linux maintains a *zone descriptor*. The zone descriptor contains information about the memory utilization within each zone, such as number of active or inactive pages, low and high watermarks to be used by the page-replacement algorithm described later in this chapter, as well as many other fields.

In addition, a zone descriptor contains an array of free areas. The i th element in this array identifies the first page descriptor of the first block of 2^i free pages. Since there may be more than one blocks of 2^i free pages, Linux uses the pair of page-descriptor pointers in each *page* element to link these together. This information is used in the memory-allocation operations. In Fig. 10-15 *free_area[0]*, which identifies all free areas of main memory consisting of only one page frame (since 2^0 is one), points to page 70, the first of the three free areas. The other free blocks of size one can be reached through the links in each of the page descriptors.

Finally, since Linux is portable to NUMA architectures (where different memory addresses have different access times), in order to differentiate between physical memory on different nodes (and avoid allocating data structures across nodes), a *node descriptor* is used. Each node descriptor contains information about the memory usage and zones on that particular node. On UMA platforms, Linux describes all memory via one node descriptor. The first few bits within each page descriptor are used to identify the node and the zone that the page frame belongs to.

In order for the paging mechanism to be efficient on both 32- and 64-bit architectures, Linux makes use of a four-level paging scheme. A three-level paging scheme, originally put into the system for the Alpha, was expanded after Linux 2.6.10, and as of version 2.6.11 a four-level paging scheme is used. Each virtual address is broken up into five fields, as shown in Fig. 10-16. The directory fields are used as an index into the appropriate page directory, of which there is a private one for each process. The value found is a pointer to one of the next-level directories, which are again indexed by a field from the virtual address. The selected entry in the middle page directory points to the final page table, which is indexed by the page field of the virtual address. The entry found here points to the page needed. On the Pentium, which uses two-level paging, each page's upper and middle directories have only one entry, so the global directory entry effectively chooses the page table to use. Similarly, three-level paging can be used when needed, by setting the size of the upper page directory field to zero.

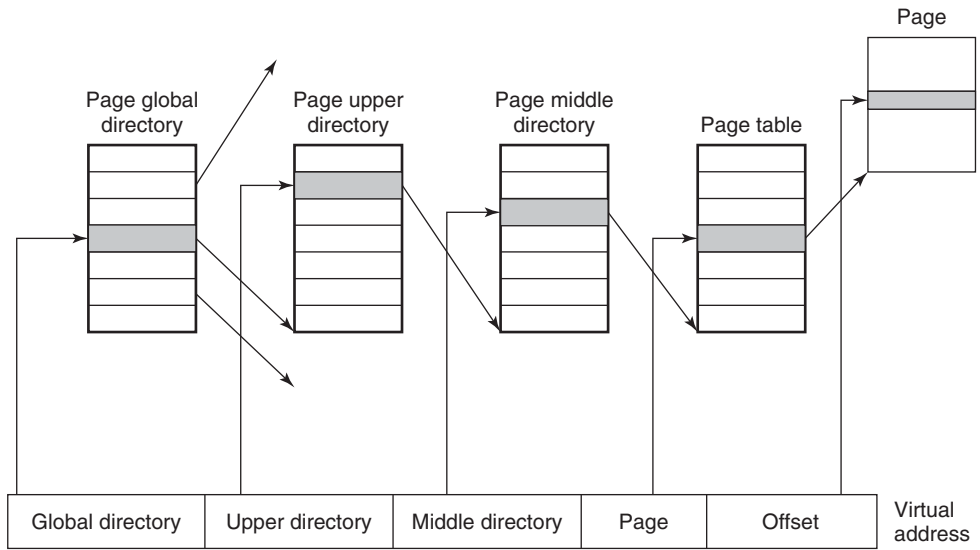


Figure 10-16. Linux uses four-level page tables.

Physical memory is used for various purposes. The kernel itself is fully hard-wired; no part of it is ever paged out. The rest of memory is available for user pages, the paging cache, and other purposes. The page cache holds pages containing file blocks that have recently been read or have been read in advance in expectation of being used in the near future, or pages of file blocks which need to be written to disk, such as those which have been created from user-mode processes which have been swapped out to disk. It is dynamic in size and competes for the same pool of pages as the user processes. The paging cache is not really a separate cache, but simply the set of user pages that are no longer needed and are waiting around to be paged out. If a page in the paging cache is reused before it is evicted from memory, it can be reclaimed quickly.

In addition, Linux supports dynamically loaded modules, most commonly device drivers. These can be of arbitrary size and each one must be allocated a contiguous piece of kernel memory. As a direct consequence of these requirements, Linux manages physical memory in such a way that it can acquire an arbitrary-sized piece of memory at will. The algorithm it uses is known as the buddy algorithm and is described below.

Memory-Allocation Mechanisms

Linux supports several mechanisms for memory allocation. The main mechanism for allocating new page frames of physical memory is the **page allocator**, which operates using the well-known **buddy algorithm**.

The basic idea for managing a chunk of memory is as follows. Initially memory consists of a single contiguous piece, 64 pages in the simple example of Fig. 10-17(a). When a request for memory comes in, it is first rounded up to a power of 2, say eight pages. The full memory chunk is then divided in half, as shown in (b). Since each of these pieces is still too large, the lower piece is divided in half again (c) and again (d). Now we have a chunk of the correct size, so it is allocated to the caller, as shown shaded in (d).

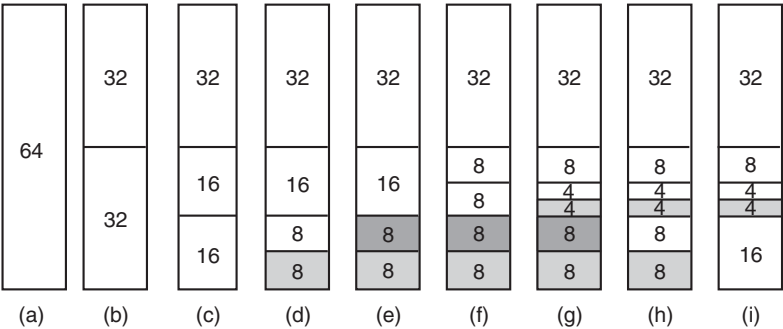


Figure 10-17. Operation of the buddy algorithm.

Now suppose that a second request comes in for eight pages. This can be satisfied directly now (e). At this point a third request comes in for four pages. The smallest available chunk is split (f) and half of it is claimed (g). Next, the second of the 8-page chunks is released (h). Finally, the other eight-page chunk is released. Since the two adjacent just-freed eight-page chunks came from the same 16-page chunk, they are merged to get the 16-page chunk back (i).

Linux manages memory using the buddy algorithm, with the additional feature of having an array in which the first element is the head of a list of blocks of size 1 unit, the second element is the head of a list of blocks of size 2 units, the next element points to the 4-unit blocks, and so on. In this way, any power-of-2 block can be found quickly.

This algorithm leads to considerable internal fragmentation because if you want a 65-page chunk, you have to ask for and get a 128-page chunk.

To alleviate this problem, Linux has a second memory allocation, the **slab allocator**, which takes chunks using the buddy algorithm but then carves slabs (smaller units) from them and manages the smaller units separately.

Since the kernel frequently creates and destroys objects of certain type (e.g., *task_struct*), it relies on so-called **object caches**. These caches consist of pointers to one or more slab which can store a number of objects of the same type. Each of the slabs may be full, partially full, or empty.

For instance, when the kernel needs to allocate a new process descriptor, that is, a new *task_struct*, it looks in the object cache for task structures, and first tries to find a partially full slab and allocate a new *task_struct* object there. If no such

slab is available, it looks through the list of empty slabs. Finally, if necessary, it will allocate a new slab, place the new task structure there, and link this slab with the task-structure object cache. The `kmalloc` kernel service, which allocates physically contiguous memory regions in the kernel address space, is in fact built on top of the slab and object cache interface described here.

A third memory allocator, `vmalloc`, is also available and is used when the requested memory need be contiguous only in virtual space, not in physical memory. In practice, this is true for most of the requested memory. One exception consists of devices, which live on the other side of the memory bus and the memory management unit, and therefore do not understand virtual addresses. However, the use of `vmalloc` results in some performance degradation, and it is used primarily for allocating large amounts of contiguous virtual address space, such as for dynamically inserting kernel modules. All these memory allocators are derived from those in System V.

Virtual Address-Space Representation

The virtual address space is divided into homogeneous, contiguous, page-aligned areas or regions. That is to say, each area consists of a run of consecutive pages with the same protection and paging properties. The text segment and mapped files are examples of areas (see Fig. 10-13). There can be holes in the virtual address space between the areas. Any memory reference to a hole results in a fatal page fault. The page size is fixed, for example, 4 KB for the Pentium and 8 KB for the Alpha. Starting with the Pentium, support for page frames of 4 MB was added. On recent 64-bit architectures, Linux can support **huge pages** of 2 MB or 1 GB each. In addition, in a **PAE (Physical Address Extension)** mode, which is used on certain 32-bit architectures to increase the process address space beyond 4 GB, page sizes of 2 MB are supported.

Each area is described in the kernel by a `vm_area_struct` entry. All the `vm_area_structs` for a process are linked together in a list sorted on virtual address so that all the pages can be found. When the list gets too long (more than 32 entries), a tree is created to speed up searching it. The `vm_area_struct` entry lists the area's properties. These properties include the protection mode (e.g., read only or read/write), whether it is pinned in memory (not pageable), and which direction it grows in (up for data segments, down for stacks).

The `vm_area_struct` also records whether the area is private to the process or shared with one or more other processes. After a fork, Linux makes a copy of the area list for the child process, but sets up the parent and child to point to the same page tables. The areas are marked as read/write, but the pages themselves are marked as read only. If either process tries to write on a page, a protection fault occurs and the kernel sees that the area is logically writable but the page is not writeable, so it gives the process a copy of the page and marks it read/write. This mechanism is how copy on write is implemented.

The *vm_area_struct* also records whether the area has backing storage on disk assigned, and if so, where. Text segments use the executable binary as backing storage and memory-mapped files use the disk file as backing storage. Other areas, such as the stack, do not have backing storage assigned until they have to be paged out.

A top-level memory descriptor, *mm_struct*, gathers information about all virtual-memory areas belonging to an address space, information about the different segments (text, data, stack), about users sharing this address space, and so on. All *vm_area_struct* elements of an address space can be accessed through their memory descriptor in two ways. First, they are organized in linked lists ordered by virtual-memory addresses. This way is useful when all virtual-memory areas need to be accessed, or when the kernel is searching to allocate a virtual-memory region of a specific size. In addition, the *vm_area_struct* entries are organized in a binary “red-black” tree, a data structure optimized for fast lookups. This method is used when a specific virtual memory needs to be accessed. By enabling access to elements of the process address space via these two methods, Linux uses more state per process, but allows different kernel operations to use the access method which is more efficient for the task at hand.

10.4.4 Paging in Linux

Early UNIX systems relied on a **swapper process** to move entire processes between memory and disk whenever not all active processes could fit in the physical memory. Linux, like other modern UNIX versions, no longer moves entire processes. The main memory management unit is a page, and almost all memory-management components operate on a page granularity. The swapping subsystem also operates on page granularity and is tightly coupled with the **page frame reclaiming algorithm**, described later in this section.

The basic idea behind paging in Linux is simple: a process need not be entirely in memory in order to run. All that is actually required is the user structure and the page tables. If these are swapped in, the process is deemed “in memory” and can be scheduled to run. The pages of the text, data, and stack segments are brought in dynamically, one at a time, as they are referenced. If the user structure and page table are not in memory, the process cannot be run until the swapper brings them in.

Paging is implemented partly by the kernel and partly by a new process called the **page daemon**. The page daemon is process 2 (process 0 is the idle process—traditionally called the swapper—and process 1 is *init*, as shown in Fig. 10-11). Like all daemons, the page daemon runs periodically. Once awake, it looks around to see if there is any work to do. If it sees that the number of pages on the list of free memory pages is too low, it starts freeing up more pages.

Linux is a fully demand-paged system with no prepaging and no working-set concept (although there is a call in which a user can give a hint that a certain page

may be needed soon, in the hope it will be there when needed). Text segments and mapped files are paged to their respective files on disk. Everything else is paged to either the paging partition (if present) or one of the fixed-length paging files, called the **swap area**. Paging files can be added and removed dynamically and each one has a priority. Paging to a separate partition, accessed as a raw device, is more efficient than paging to a file for several reasons. First, the mapping between file blocks and disk blocks is not needed (saves disk I/O reading indirect blocks). Second, the physical writes can be of any size, not just the file block size. Third, a page is always written contiguously to disk; with a paging file, it may or may not be.

Pages are not allocated on the paging device or partition until they are needed. Each device and file starts with a bitmap telling which pages are free. When a page without backing store has to be tossed out of memory, the highest-priority paging partition or file that still has space is chosen and a page allocated on it. Normally, the paging partition, if present, has higher priority than any paging file. The page table is updated to reflect that the page is no longer present in memory (e.g., the page-not-present bit is set) and the disk location is written into the page-table entry.

The Page Replacement Algorithm

Page replacement works as follows. Linux tries to keep some pages free so that they can be claimed as needed. Of course, this pool must be continually replenished. The **PFRA (Page Frame Reclaiming Algorithm)** algorithm is how this happens.

First of all, Linux distinguishes between four different types of pages: *unreclaimable*, *swappable*, *syncable*, and *discardable*. Unreclaimable pages, which include reserved or locked pages, kernel mode stacks, and the like, may not be paged out. Swappable pages must be written back to the swap area or the paging disk partition before the page can be reclaimed. Syncable pages must be written back to disk if they have been marked as dirty. Finally, discardable pages can be reclaimed immediately.

At boot time, *init* starts up a page daemon, *kswapd*, for each memory node, and configures them to run periodically. Each time *kswapd* awakens, it checks to see if there are enough free pages available, by comparing the low and high watermarks with the current memory usage for each memory zone. If there is enough memory, it goes back to sleep, although it can be awakened early if more pages are suddenly needed. If the available memory for any of the zones ever falls below a threshold, *kswapd* initiates the page frame reclaiming algorithm. During each run, only a certain target number of pages is reclaimed, typically a maximum of 32. This number is limited to control the I/O pressure (the number of disk writes created during the PFRA operations). Both the number of reclaimed pages and the total number of scanned pages are configurable parameters.

Each time PFRA executes, it first tries to reclaim easy pages, then proceeds with the harder ones. Many people also grab the low-hanging fruit first. Discardable and unreferenced pages can be reclaimed immediately by moving them onto the zone's freelist. Next it looks for pages with backing store which have not been referenced recently, using a clock-like algorithm. Following are shared pages that none of the users seems to be using much. The challenge with shared pages is that, if a page entry is reclaimed, the page tables of all address spaces originally sharing that page must be updated in a synchronous manner. Linux maintains efficient tree-like data structures to easily find all users of a shared page. Ordinary user pages are searched next, and if chosen to be evicted, they must be scheduled for write in the swap area. The **swappiness** of the system, that is, the ratio of pages with backing store vs. pages which need to be swapped out selected during PFRA, is a tunable parameter of the algorithm. Finally, if a page is invalid, absent from memory, shared, locked in memory, or being used for DMA, it is skipped.

PFRA uses a clock-like algorithm to select old pages for eviction within a certain category. At the core of this algorithm is a loop which scans through each zone's active and inactive lists, trying to reclaim different kinds of pages, with different urgencies. The urgency value is passed as a parameter telling the procedure how much effort to expend to reclaim some pages. Usually, this means how many pages to inspect before giving up.

During PFRA, pages are moved between the active and inactive list in the manner described in Fig. 10-18. To maintain some heuristics and try to find pages which have not been referenced and are unlikely to be needed in the near future, PFRA maintains two flags per page: active/inactive, and referenced or not. These two flags encode four states, as shown in Fig. 10-18. During the first scan of a set of pages, PFRA first clears their reference bits. If during the second run over the page it is determined that it has been referenced, it is advanced to another state, from which it is less likely to be reclaimed. Otherwise, the page is moved to a state from where it is more likely to be evicted.

Pages on the inactive list, which have not been referenced since the last time they were inspected, are the best candidates for eviction. They are pages with both *PG_active* and *PG_referenced* set to zero in Fig. 10-18. However, if necessary, pages may be reclaimed even if they are in some of the other states. The *refill* arrows in Fig. 10-18 illustrate this fact.

The reason PRFA maintains pages in the inactive list although they might have been referenced is to prevent situations such as the following. Consider a process which makes periodic accesses to different pages, with a 1-hour period. A page accessed since the last loop will have its reference flag set. However, since it will not be needed again for the next hour, there is no reason not to consider it as a candidate for reclamation.

One aspect of the memory-management system that we have not yet mentioned is a second daemon, *pdflush*, actually a set of background daemon threads. The *pdflush* threads either (1) wake up periodically, typically every 500 msec, to write

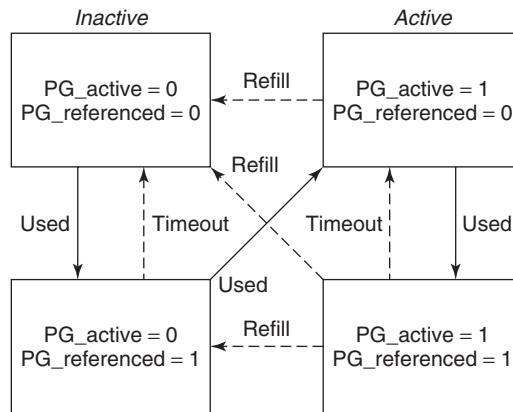


Figure 10-18. Page states considered in the page-frame replacement algorithm.

back to disk very old dirty pages, or (2) are explicitly awakened by the kernel when available memory levels fall below a certain threshold, to write back dirty pages from the page cache to disk. In **laptop mode**, in order to conserve battery life, dirty pages are written to disk whenever *pdflush* threads wake up. Dirty pages may also be written out to disk on explicit requests for synchronization, via systems calls such as *sync*, *fsync*, or *fdatasync*. Older Linux versions used two separate daemons: *kupdate*, for old-page write back, and *bdflush*, for page write back under low memory conditions. In the 2.4 kernel this functionality was integrated in the *pdflush* threads. The choice of multiple threads was made in order to hide long disk latencies.

10.5 INPUT/OUTPUT IN LINUX

The I/O system in Linux is fairly straightforward and the same as in other UNICES. Basically, all I/O devices are made to look like files and are accessed as such with the same *read* and *write* system calls that are used to access all ordinary files. In some cases, device parameters must be set, and this is done using a special system call. We will study these issues in the following sections.

10.5.1 Fundamental Concepts

Like all computers, those running Linux have I/O devices such as disks, printers, and networks connected to them. Some way is needed to allow programs to access these devices. Although various solutions are possible, the Linux one is to integrate the devices into the file system as what are called **special files**. Each I/O

device is assigned a path name, usually in */dev*. For example, a disk might be */dev/hd1*, a printer might be */dev/lp*, and the network might be */dev/net*.

These special files can be accessed the same way as any other files. No special commands or system calls are needed. The usual *open*, *read*, and *write* system calls will do just fine. For example, the command

```
cp file /dev/lp
```

copies the *file* to printer, causing it to be printed (assuming that the user has permission to access */dev/lp*). Programs can open, read, and write special files exactly the same way as they do regular files. In fact, *cp* in the above example is not even aware that it is printing. In this way, no special mechanism is needed for doing I/O.

Special files are divided into two categories, block and character. A **block special file** is one consisting of a sequence of numbered blocks. The key property of the block special file is that each block can be individually addressed and accessed. In other words, a program can open a block special file and read, say, block 124 without first having to read blocks 0 to 123. Block special files are typically used for disks.

Character special files are normally used for devices that input or output a character stream. Keyboards, printers, networks, mice, plotters, and most other I/O devices that accept or produce data for people use character special files. It is not possible (or even meaningful) to seek to block 124 on a mouse.

Associated with each special file is a device driver that handles the corresponding device. Each driver has what is called a **major device** number that serves to identify it. If a driver supports multiple devices, say, two disks of the same type, each disk has a **minor device** number that identifies it. Together, the major and minor device numbers uniquely specify every I/O device. In few cases, a single driver handles two closely related devices. For example, the driver corresponding to */dev/tty* controls both the keyboard and the screen, often thought of as a single device, the terminal.

Although most character special files cannot be randomly accessed, they often need to be controlled in ways that block special files do not. Consider, for example, input typed on the keyboard and displayed on the screen. When a user makes a typing error and wants to erase the last character typed, he presses some key. Some people prefer to use backspace, and others prefer DEL. Similarly, to erase the entire line just typed, many conventions abound. Traditionally @ was used, but with the spread of e-mail (which uses @ within e-mail address), many systems have adopted CTRL-U or some other character. Likewise, to interrupt the running program, some special key must be hit. Here, too, different people have different preferences. CTRL-C is a common choice, but it is not universal.

Rather than making a choice and forcing everyone to use it, Linux allows all these special functions and many others to be customized by the user. A special system call is generally provided for setting these options. This system call also

handles tab expansion, enabling and disabling of character echoing, conversion between carriage return and line feed, and similar items. The system call is not permitted on regular files or block special files.

10.5.2 Networking

Another example of I/O is networking, as pioneered by Berkeley UNIX and taken over by Linux more or less verbatim. The key concept in the Berkeley design is the **socket**. Sockets are analogous to mailboxes and telephone wall sockets in that they allow users to interface to the network, just as mailboxes allow people to interface to the postal system and telephone wall sockets allow them to plug in telephones and connect to the telephone system. The sockets' position is shown in Fig. 10-19.

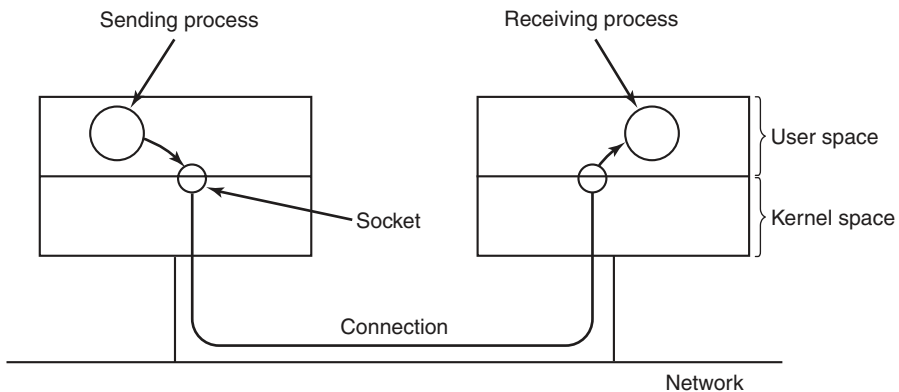


Figure 10-19. The uses of sockets for networking.

Sockets can be created and destroyed dynamically. Creating a socket returns a file descriptor, which is needed for establishing a connection, reading data, writing data, and releasing the connection.

Each socket supports a particular type of networking, specified when the socket is created. The most common types are

1. Reliable connection-oriented byte stream.
2. Reliable connection-oriented packet stream.
3. Unreliable packet transmission.

The first socket type allows two processes on different machines to establish the equivalent of a pipe between them. Bytes are pumped in at one end and they come out in the same order at the other. The system guarantees that all bytes that are sent correctly arrive and in the same order they were sent.

The second type is rather similar to the first one, except that it preserves packet boundaries. If the sender makes five separate calls to `write`, each for 512 bytes, and the receiver asks for 2560 bytes, with a type 1 socket all 2560 bytes will be returned at once. With a type 2 socket, only 512 bytes will be returned. Four more calls are needed to get the rest. The third type of socket is used to give the user access to the raw network. This type is especially useful for real-time applications, and for those situations in which the user wants to implement a specialized error-handling scheme. Packets may be lost or reordered by the network. There are no guarantees, as in the first two cases. The advantage of this mode is higher performance, which sometimes outweighs reliability (e.g., for multimedia delivery, in which being fast counts for more than being right).

When a socket is created, one of the parameters specifies the protocol to be used for it. For reliable byte streams, the most popular protocol is **TCP (Transmission Control Protocol)**. For unreliable packet-oriented transmission, **UDP (User Datagram Protocol)** is the usual choice. Both of these are layered on top of **IP (Internet Protocol)**. All of these protocols originated with the U.S. Dept. of Defense's ARPANET, and now form the basis of the Internet. There is no common protocol for reliable packet streams.

Before a socket can be used for networking, it must have an address bound to it. This address can be in one of several naming domains. The most common one is the Internet naming domain, which uses 32-bit integers for naming endpoints in Version 4 and 128-bit integers in Version 6 (Version 5 was an experimental system that never made it to the major leagues).

Once sockets have been created on both the source and destination computers, a connection can be established between them (for connection-oriented communication). One party makes a `listen` system call on a local socket, which creates a buffer and blocks until data arrive. The other makes a `connect` system call, giving as parameters the file descriptor for a local socket and the address of a remote socket. If the remote party accepts the call, the system then establishes a connection between the sockets.

Once a connection has been established, it functions analogously to a pipe. A process can read and write from it using the file descriptor for its local socket. When the connection is no longer needed, it can be closed in the usual way, via the `close` system call.

10.5.3 Input/Output System Calls in Linux

Each I/O device in a Linux system generally has a special file associated with it. Most I/O can be done by just using the proper file, eliminating the need for special system calls. Nevertheless, sometimes there is a need for something that is device specific. Prior to POSIX most UNIX systems had a system call `ioctl` that performed a large number of device-specific actions on special files. Over the course of the years, it had gotten to be quite a mess. POSIX cleaned it up by splitting its

functions into separate function calls primarily for terminal devices. In Linux and modern UNIX systems, whether each one is a separate system call or they share a single system call or something else is implementation dependent.

The first four calls listed in Fig. 10-20 are used to set and get the terminal speed. Different calls are provided for input and output because some modems operate at split speed. For example, old videotex systems allowed people to access public databases with short requests from the home to the server at 75 bits/sec with replies coming back at 1200 bits/sec. This standard was adopted at a time when 1200 bits/sec both ways was too expensive for home use. Times change in the networking world. This asymmetry still persists, with some telephone companies offering inbound service at 20 Mbps and outbound service at 2 Mbps, often under the name of **ADSL (Asymmetric Digital Subscriber Line)**.

Function call	Description
<code>s = cfsetospeed(&termios, speed)</code>	Set the output speed
<code>s = cfsetispeed(&termios, speed)</code>	Set the input speed
<code>s = cfgetospeed(&termios, speed)</code>	Get the output speed
<code>s = cfgetispeed(&termios, speed)</code>	Get the input speed
<code>s = tcsetattr(fd, opt, &termios)</code>	Set the attributes
<code>s = tcgetattr(fd, &termios)</code>	Get the attributes

Figure 10-20. The main POSIX calls for managing the terminal.

The last two calls in the list are for setting and reading back all the special characters used for erasing characters and lines, interrupting processes, and so on. In addition, they enable and disable echoing, handle flow control, and perform other related functions. Additional I/O function calls also exist, but they are somewhat specialized, so we will not discuss them further. In addition, `ioctl` is still available.

10.5.4 Implementation of Input/Output in Linux

I/O in Linux is implemented by a collection of device drivers, one per device type. The function of the drivers is to isolate the rest of the system from the idiosyncracies of the hardware. By providing standard interfaces between the drivers and the rest of the operating system, most of the I/O system can be put into the machine-independent part of the kernel.

When the user accesses a special file, the file system determines the major and minor device numbers belonging to it and whether it is a block special file or a character special file. The major device number is used to index into one of two internal hash tables containing data structures for character or block devices. The structure thus located contains pointers to the procedures to call to open the device, read the device, write the device, and so on. The minor device number is passed as

a parameter. Adding a new device type to Linux means adding a new entry to one of these tables and supplying the corresponding procedures to handle the various operations on the device.

Some of the operations which may be associated with different character devices are shown in Fig. 10-21. Each row refers to a single I/O device (i.e., a single driver). The columns represent the functions that all character drivers must support. Several other functions also exist. When an operation is performed on a character special file, the system indexes into the hash table of character devices to select the proper structure, then calls the corresponding function to have the work performed. Thus each of the file operations contains a pointer to a function contained in the corresponding driver.

Device	Open	Close	Read	Write	ioctl	Other
Null	null	null	null	null	null	...
Memory	null	null	mem_read	mem_write	null	...
Keyboard	k_open	k_close	k_read	error	k_ioctl	...
Tty	tty_open	tty_close	tty_read	tty_write	tty_ioctl	...
Printer	lp_open	lp_close	error	lp_write	lp_ioctl	...

Figure 10-21. Some of the file operations supported for typical character devices.

Each driver is split into two parts, both of which are part of the Linux kernel and both of which run in kernel mode. The top half runs in the context of the caller and interfaces to the rest of Linux. The bottom half runs in kernel context and interacts with the device. Drivers are allowed to make calls to kernel procedures for memory allocation, timer management, DMA control, and other things. The set of kernel functions that may be called is defined in a document called the **Driver-Kernel Interface**. Writing device drivers for Linux is covered in detail in Cooperstein (2009) and Corbet et al. (2009).

The I/O system is split into two major components: the handling of block special files and the handling of character special files. We will now look at each of these components in turn.

The goal of the part of the system that does I/O on block special files (e.g., disks) is to minimize the number of transfers that must be done. To accomplish this goal, Linux has a **cache** between the disk drivers and the file system, as illustrated in Fig. 10-22. Prior to the 2.2 kernel, Linux maintained completely separate page and buffer caches, so a file residing in a disk block could be cached in both caches. Newer versions of Linux have a unified cache. A *generic block layer* holds these components together, performs the necessary translations between disk sectors, blocks, buffers and pages of data, and enables the operations on them.

The cache is a table in the kernel for holding thousands of the most recently used blocks. When a block is needed from a disk for whatever reason (i-node, directory, or data), a check is first made to see if it is in the cache. If it is present in

the cache, the block is taken from there and a disk access is avoided, thereby resulting in great improvements in system performance.

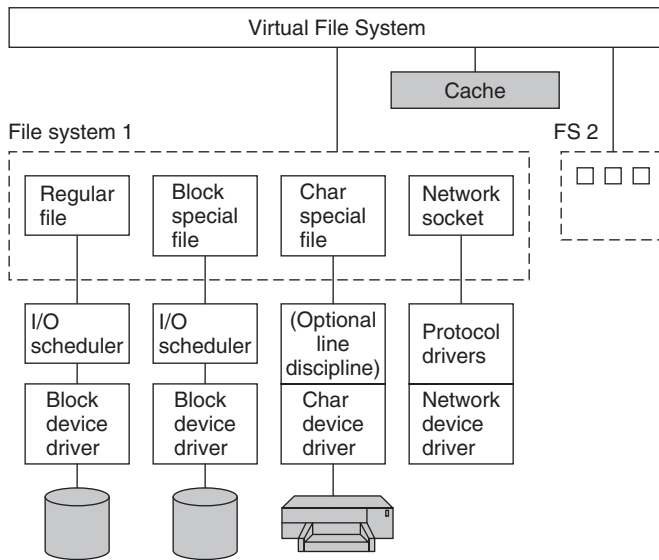


Figure 10-22. The Linux I/O system showing one file system in detail.

If the block is not in the page cache, it is read from the disk into the cache and from there copied to where it is needed. Since the page cache has room for only a fixed number of blocks, the page-replacement algorithm described in the previous section is invoked.

The page cache works for writes as well as for reads. When a program writes a block, it goes to the cache, not to the disk. The *pdflush* daemon will flush the block to disk in the event the cache grows above a specified value. In addition, to avoid having blocks stay too long in the cache before being written to the disk, all dirty blocks are written to the disk every 30 seconds.

In order to reduce the latency of repetitive disk-head movements, Linux relies on an **I/O scheduler**. Its purpose is to reorder or bundle read/write requests to block devices. There are many scheduler variants, optimized for different types of workloads. The basic Linux scheduler is based on the original **Linux elevator scheduler**. The operations of the elevator scheduler can be summarized as follows: Disk operations are sorted in a doubly linked list, ordered by the address of the sector of the disk request. New requests are inserted in this list in a sorted manner. This prevents repeated costly disk-head movements. The request list is subsequently *merged* so that adjacent operations are issued via a single disk request. The basic elevator scheduler can lead to starvation. Therefore, the revised version of the Linux disk scheduler includes two additional lists, maintaining read or write operations ordered by their deadlines. The default deadlines are 0.5 sec for reads

and 5 sec for writes. If a system-defined deadline for the oldest write operation is about to expire, that write request will be serviced before any of the requests on the main doubly linked list.

In addition to regular disk files, there are also block special files, also called **raw block files**. These files allow programs to access the disk using absolute block numbers, without regard to the file system. They are most often used for things like paging and system maintenance.

The interaction with character devices is simple. Since character devices produce or consume streams of characters, or bytes of data, support for random access makes little sense. One exception is the use of **line disciplines**. A line discipline can be associated with a terminal device, represented via the structure *tty_struct*, and it represents an interpreter for the data exchanged with the terminal device. For instance, local line editing can be done (i.e., erased characters and lines can be removed), carriage returns can be mapped onto line feeds, and other special processing can be completed. However, if a process wants to interact on every character, it can put the line in raw mode, in which case the line discipline will be bypassed. Not all devices have line disciplines.

Output works in a similar way, expanding tabs to spaces, converting line feeds to carriage returns + line feeds, adding filler characters following carriage returns on slow mechanical terminals, and so on. Like input, output can go through the line discipline (cooked mode) or bypass it (raw mode). Raw mode is especially useful when sending binary data to other computers over a serial line and for GUIs. Here, no conversions are desired.

The interaction with **network devices** is different. While network devices also produce/consume streams of characters, their asynchronous nature makes them less suitable for easy integration under the same interface as other character devices. The networking device driver produces packets consisting of multiple bytes of data, along with network headers. These packets are then routed through a series of network protocol drivers, and ultimately are passed to the user-space application. A key data structure is the socket buffer structure, *skbuff*, which is used to represent portions of memory filled with packet data. The data in an *skbuff* buffer do not always start at the start of the buffer. As they are being processed by various protocols in the networking stack, protocol headers may be removed, or added. The user processes interact with networking devices via **sockets**, which in Linux support the original BSD socket API. The protocol drivers can be bypassed and direct access to the underlying network device is enabled via *raw_sockets*. Only the superuser is allowed to create raw sockets.

10.5.5 Modules in Linux

For decades, UNIX device drivers were statically linked into the kernel so they were all present in memory whenever the system was booted. Given the environment in which UNIX grew up, commonly departmental minicomputers and then

high-end workstations, with their small and unchanging sets of I/O devices, this scheme worked well. Basically, a computer center built a kernel containing drivers for the I/O devices and that was it. If next year the center bought a new disk, it relinked the kernel. No big deal.

With the arrival of Linux on the PC platform, suddenly all that changed. The number of I/O devices available on the PC is orders of magnitude larger than on any minicomputer. In addition, although all Linux users have (or can easily get) the full source code, probably the vast majority would have considerable difficulty adding a driver, updating all the device-driver related data structures, relinking the kernel, and then installing it as the bootable system (not to mention dealing with the aftermath of building a kernel that does not boot).

Linux solved this problem with the concept of **loadable modules**. These are chunks of code that can be loaded into the kernel while the system is running. Most commonly these are character or block device drivers, but they can also be entire file systems, network protocols, performance monitoring tools, or anything else desired.

When a module is loaded, several things have to happen. First, the module has to be relocated on the fly, during loading. Second, the system has to check to see if the resources the driver needs are available (e.g., interrupt request levels) and if so, mark them as in use. Third, any interrupt vectors that are needed must be set up. Fourth, the appropriate driver switch table has to be updated to handle the new major device type. Finally, the driver is allowed to run to perform any device-specific initialization it may need. Once all these steps are completed, the driver is fully installed, the same as any statically installed driver. Other modern UNIX systems now also support loadable modules.

10.6 THE LINUX FILE SYSTEM

The most visible part of any operating system, including Linux, is the file system. In the following sections we will examine the basic ideas behind the Linux file system, the system calls, and how the file system is implemented. Some of these ideas derive from MULTICS, and many of them have been copied by MS-DOS, Windows, and other systems, but others are unique to UNIX-based systems. The Linux design is especially interesting because it clearly illustrates the principle of *Small is Beautiful*. With minimal mechanism and a very limited number of system calls, Linux nevertheless provides a powerful and elegant file system.

10.6.1 Fundamental Concepts

The initial Linux file system was the MINIX 1 file system. However, because it limited file names to 14 characters (in order to be compatible with UNIX Version 7) and its maximum file size was 64 MB (which was overkill on the 10-MB hard

disks of its era), there was interest in better file systems almost from the beginning of the Linux development, which began about 5 years after MINIX 1 was released. The first improvement was the ext file system, which allowed file names of 255 characters and files of 2 GB, but it was slower than the MINIX 1 file system, so the search continued for a while. Eventually, the ext2 file system was invented, with long file names, long files, and better performance, and it has become the main file system. However, Linux supports several dozen file systems using the Virtual File System (VFS) layer (described in the next section). When Linux is linked, a choice is offered of which file systems should be built into the kernel. Others can be dynamically loaded as modules during execution, if need be.

A Linux file is a sequence of 0 or more bytes containing arbitrary information. No distinction is made between ASCII files, binary files, or any other kinds of files. The meaning of the bits in a file is entirely up to the file's owner. The system does not care. File names are limited to 255 characters, and all the ASCII characters except NUL are allowed in file names, so a file name consisting of three carriage returns is a legal file name (but not an especially convenient one).

By convention, many programs expect file names to consist of a base name and an extension, separated by a dot (which counts as a character). Thus *prog.c* is typically a C program, *prog.py* is typically a Python program, and *prog.o* is usually an object file (compiler output). These conventions are not enforced by the operating system but some compilers and other programs expect them. Extensions may be of any length, and files may have multiple extensions, as in *prog.java.gz*, which is probably a *gzip* compressed Java program.

Files can be grouped together in directories for convenience. Directories are stored as files and to a large extent can be treated like files. Directories can contain subdirectories, leading to a hierarchical file system. The root directory is called / and always contains several subdirectories. The / character is also used to separate directory names, so that the name */usr/ast/x* denotes the file *x* located in the directory *ast*, which itself is in the */usr* directory. Some of the major directories near the top of the tree are shown in Fig. 10-23.

Directory	Contents
bin	Binary (executable) programs
dev	Special files for I/O devices
etc	Miscellaneous system files
lib	Libraries
usr	User directories

Figure 10-23. Some important directories found in most Linux systems.

There are two ways to specify file names in Linux, both to the shell and when opening a file from inside a program. The first way is by means of an **absolute path**, which means telling how to get to the file starting at the root directory. An

example of an absolute path is `/usr/ast/books/mos4/chap-10`. This tells the system to look in the root directory for a directory called `usr`, then look there for another directory, `ast`. In turn, this directory contains a directory `books`, which contains the directory `mos4`, which contains the file `chap-10`.

Absolute path names are often long and inconvenient. For this reason, Linux allows users and processes to designate the directory in which they are currently working as the **working directory**. Path names can also be specified relative to the working directory. A path name specified relative to the working directory is a **relative path**. For example, if `/usr/ast/books/mos4` is the working directory, then the shell command

```
cp chap-10 backup-10
```

has exactly the same effect as the longer command

```
cp /usr/ast/books/mos4/chap-10 /usr/ast/books/mos4/backup-10
```

It frequently occurs that a user needs to refer to a file that belongs to another user, or at least is located elsewhere in the file tree. For example, if two users are sharing a file, it will be located in a directory belonging to one of them, so the other will have to use an absolute path name to refer to it (or change the working directory). If this is long enough, it may become irritating to have to keep typing it. Linux provides a solution by allowing users to make a new directory entry that points to an existing file. Such an entry is called a **link**.

As an example, consider the situation of Fig. 10-24(a). Fred and Lisa are working together on a project, and each of them needs access to the other's files. If Fred has `/usr/fred` as his working directory, he can refer to the file `x` in Lisa's directory as `/usr/lisa/x`. Alternatively, Fred can create a new entry in his directory, as shown in Fig. 10-24(b), after which he can use `x` to mean `/usr/lisa/x`.

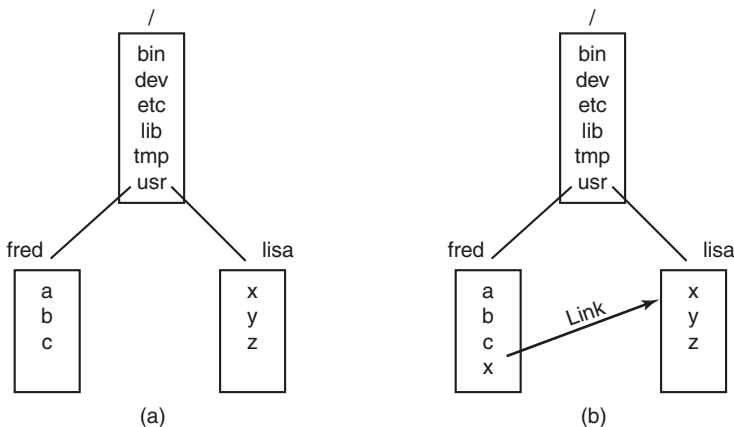


Figure 10-24. (a) Before linking. (b) After linking.

In the example just discussed, we suggested that before linking, the only way for Fred to refer to Lisa's file *x* was by using its absolute path. Actually, this is not really true. When a directory is created, two entries, `.` and `..`, are automatically made in it. The former refers to the working directory itself. The latter refers to the directory's parent, that is, the directory in which it itself is listed. Thus from `/usr/fred`, another path to Lisa's file *x* is `../lisa/x`.

In addition to regular files, Linux also supports character special files and block special files. Character special files are used to model serial I/O devices, such as keyboards and printers. Opening and reading from `/dev/tty` reads from the keyboard; opening and writing to `/dev/lp` writes to the printer. Block special files, often with names like `/dev/hd1`, can be used to read and write raw disk partitions without regard to the file system. Thus a seek to byte *k* followed by a read will begin reading from the *k*th byte on the corresponding partition, completely ignoring the i-node and file structure. Raw block devices are used for paging and swapping by programs that lay down file systems (e.g., *mkfs*), and by programs that fix sick file systems (e.g., *fsck*), for example.

Many computers have two or more disks. On mainframes at banks, for example, it is frequently necessary to have 100 or more disks on a single machine, in order to hold the huge databases required. Even personal computers often have at least two disks—a hard disk and an optical (e.g., DVD) drive. When there are multiple disk drives, the question arises of how to handle them.

One solution is to put a self-contained file system on each one and just keep them separate. Consider, for example, the situation shown in Fig. 10-25(a). Here we have a hard disk, which we call *C:*, and a DVD, which we call *D:*. Each has its own root directory and files. With this solution, the user has to specify both the device and the file when anything other than the default is needed. For instance, to copy a file *x* to a directory *d* (assuming *C:* is the default), one would type

```
cp D:/x /a/d/x
```

This is the approach taken by a number of systems, including Windows 8, which it inherited from MS-DOS in a century long ago.

The Linux solution is to allow one disk to be mounted in another disk's file tree. In our example, we could mount the DVD on the directory `/b`, yielding the file system of Fig. 10-25(b). The user now sees a single file tree, and no longer has to be aware of which file resides on which device. The above copy command now becomes

```
cp /b/x /a/d/x
```

exactly the same as it would have been if everything had been on the hard disk in the first place.

Another interesting property of the Linux file system is **locking**. In some applications, two or more processes may be using the same file at the same time, which may lead to race conditions. One solution is to program the application with

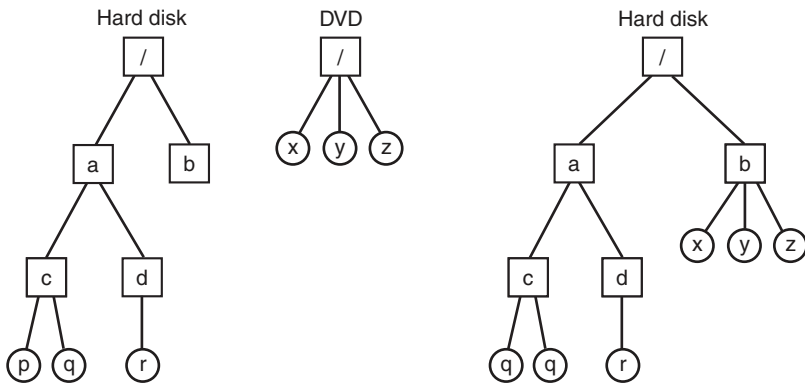


Figure 10-25. (a) Separate file systems. (b) After mounting.

critical regions. However, if the processes belong to independent users who do not even know each other, this kind of coordination is generally inconvenient.

Consider, for example, a database consisting of many files in one or more directories that are accessed by unrelated users. It is certainly possible to associate a semaphore with each directory or file and achieve mutual exclusion by having processes do a down operation on the appropriate semaphore before accessing the data. The disadvantage, however, is that a whole directory or file is then made inaccessible, even though only one record may be needed.

For this reason, POSIX provides a flexible and fine-grained mechanism for processes to lock as little as a single byte and as much as an entire file in one indivisible operation. The locking mechanism requires the caller to specify the file to be locked, the starting byte, and the number of bytes. If the operation succeeds, the system makes a table entry noting that the bytes in question (e.g., a database record) are locked.

Two kinds of locks are provided, **shared locks** and **exclusive locks**. If a portion of a file already contains a shared lock, a second attempt to place a shared lock on it is permitted, but an attempt to put an exclusive lock on it will fail. If a portion of a file contains an exclusive lock, all attempts to lock any part of that portion will fail until the lock has been released. In order to successfully place a lock, every byte in the region to be locked must be available.

When placing a lock, a process must specify whether it wants to block or not in the event that the lock cannot be placed. If it chooses to block, when the existing lock has been removed, the process is unblocked and the lock is placed. If the process chooses not to block when it cannot place a lock, the system call returns immediately, with the status code telling whether the lock succeeded or not. If it did not, the caller has to decide what to do next (e.g., wait and try again).

Locked regions may overlap. In Fig. 10-26(a) we see that process *A* has placed a shared lock on bytes 4 through 7 of some file. Later, process *B* places a shared

lock on bytes 6 through 9, as shown in Fig. 10-26(b). Finally, *C* locks bytes 2 through 11. As long as all these locks are shared, they can coexist.

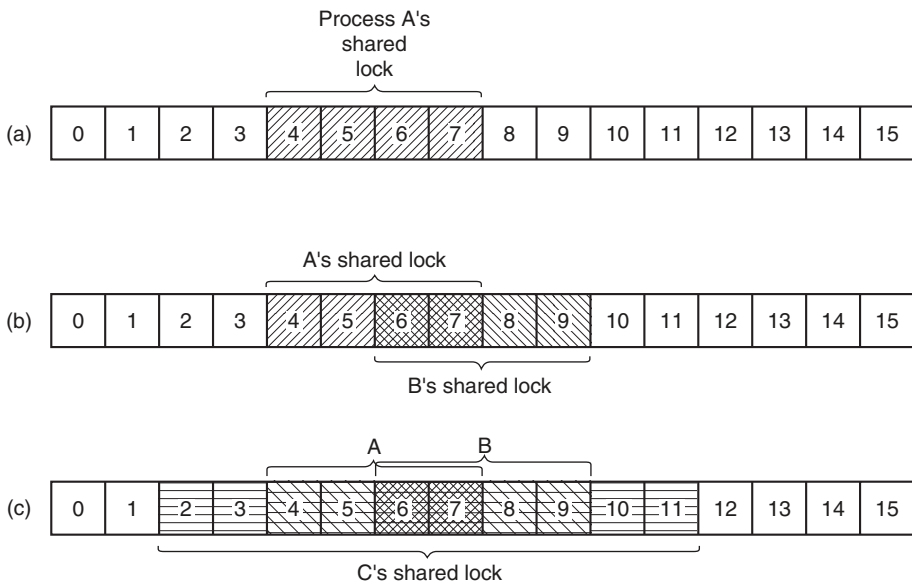


Figure 10-26. (a) A file with one lock. (b) Adding a second lock. (c) A third one.

Now consider what happens if a process tries to acquire an exclusive lock to byte 9 of the file of Fig. 10-26(c), with a request to block if the lock fails. Since two previous locks cover this block, the caller will block and will remain blocked until both *B* and *C* release their locks.

10.6.2 File-System Calls in Linux

Many system calls relate to files and the file system. First we will look at the system calls that operate on individual files. Later we will examine those that involve directories or the file system as a whole. To create a new file, the `creat` call can be used. (When Ken Thompson was once asked what he would do differently if he had the chance to reinvent UNIX, he replied that he would spell `creat` as `create` this time.) The parameters provide the name of the file and the protection mode. Thus

```
fd = creat("abc", mode);
```

creates a file called *abc* with the protection bits taken from *mode*. These bits determine which users may access the file and how. They will be described later.

The `creat` call not only creates a new file, but also opens it for writing. To allow subsequent system calls to access the file, a successful `creat` returns a small

nonnegative integer called a **file descriptor**, *fd* in the example above. If a `creat` is done on an existing file, that file is truncated to length 0 and its contents are discarded. Files can also be created using the `open` call with appropriate arguments.

Now let us continue looking at the main file-system calls, which are listed in Fig. 10-27. To read or write an existing file, the file must first be opened by calling `open` or `creat`. This call specifies the file name to be opened and how it is to be opened: for reading, writing, or both. Various options can be specified as well. Like `creat`, the call to `open` returns a file descriptor that can be used for reading or writing. Afterward, the file can be closed by `close`, which makes the file descriptor available for reuse on a subsequent `creat` or `open`. Both the `creat` and `open` calls always return the lowest-numbered file descriptor not currently in use.

When a program starts executing in the standard way, file descriptors 0, 1, and 2 are already opened for standard input, standard output, and standard error, respectively. In this way, a filter, such as the `sort` program, can just read its input from file descriptor 0 and write its output to file descriptor 1, without having to know what files they are. This mechanism works because the shell arranges for these values to refer to the correct (redirected) files before the program is started.

System call	Description
<code>fd = creat(name, mode)</code>	One way to create a new file
<code>fd = open(file, how, ...)</code>	Open a file for reading, writing, or both
<code>s = close(fd)</code>	Close an open file
<code>n = read(fd, buffer, nbytes)</code>	Read data from a file into a buffer
<code>n = write(fd, buffer, nbytes)</code>	Write data from a buffer into a file
<code>position = lseek(fd, offset, whence)</code>	Move the file pointer
<code>s = stat(name, &buf)</code>	Get a file's status information
<code>s = fstat(fd, &buf)</code>	Get a file's status information
<code>s = pipe(&fd[0])</code>	Create a pipe
<code>s = fcntl(fd, cmd, ...)</code>	File locking and other operations

Figure 10-27. Some system calls relating to files. The return code *s* is `-1` if an error has occurred; *fd* is a file descriptor, and *position* is a file offset. The parameters should be self explanatory.

The most heavily used calls are undoubtedly `read` and `write`. Each one has three parameters: a file descriptor (telling which open file to read or write), a buffer address (telling where to put the data or get the data from), and a count (telling how many bytes to transfer). That is all there is. It is a very simple design. A typical call is

```
n = read(fd, buffer, nbytes);
```

Although nearly all programs read and write files sequentially, some programs need to be able to access any part of a file at random. Associated with each file is a

pointer that indicates the current position in the file. When reading (or writing) sequentially, it normally points to the next byte to be read (written). If the pointer is at, say, 4096, before 1024 bytes are read, it will automatically be moved to 5120 after a successful `read` system call. The `lseek` call changes the value of the position pointer, so that subsequent calls to `read` or `write` can begin anywhere in the file, or even beyond the end of it. It is called `lseek` to avoid conflicting with `seek`, a now-obsolete call that was formerly used on 16-bit computers for seeking.

`Lseek` has three parameters: the first one is the file descriptor for the file; the second is a file position; the third tells whether the file position is relative to the beginning of the file, the current position, or the end of the file. The value returned by `lseek` is the absolute position in the file after the file pointer is changed. Slightly ironically, `lseek` is the only file system call that never causes a real disk seek because all it does is update the current file position, which is a number in memory.

For each file, Linux keeps track of the file mode (regular, directory, special file), size, time of last modification, and other information. Programs can ask to see this information via the `stat` system call. The first parameter is the file name. The second is a pointer to a structure where the information requested is to be put. The fields in the structure are shown in Fig. 10-28. The `fstat` call is the same as `stat` except that it operates on an open file (whose name may not be known) rather than on a path name.

Device the file is on
I-node number (which file on the device)
File mode (includes protection information)
Number of links to the file
Identity of the file's owner
Group the file belongs to
File size (in bytes)
Creation time
Time of last access
Time of last modification

Figure 10-28. The fields returned by the `stat` system call.

The pipe system call is used to create shell pipelines. It creates a kind of pseudofile, which buffers the data between the pipeline components, and returns file descriptors for both reading and writing the buffer. In a pipeline such as

```
sort <in | head -30
```

file descriptor 1 (standard output) in the process running `sort` would be set (by the shell) to write to the pipe, and file descriptor 0 (standard input) in the process running `head` would be set to read from the pipe. In this way, `sort` just reads from file descriptor 0 (set to the file *in*) and writes to file descriptor 1 (the pipe) without even

being aware that these have been redirected. If they have not been redirected, *sort* will automatically read from the keyboard and write to the screen (the default devices). Similarly, when *head* reads from file descriptor 0, it is reading the data *sort* put into the pipe buffer without even knowing that a pipe is in use. This is a clear example of how a simple concept (redirection) with a simple implementation (file descriptors 0 and 1) can lead to a powerful tool (connecting programs in arbitrary ways without having to modify them at all).

The last system call in Fig. 10-27 is `fcntl`. It is used to lock and unlock files, apply shared or exclusive locks, and perform a few other file-specific operations.

Now let us look at some system calls that relate more to directories or the file system as a whole, rather than just to one specific file. Some common ones are listed in Fig. 10-29. Directories are created and destroyed using `mkdir` and `rmdir`, respectively. A directory can be removed only if it is empty.

System call	Description
<code>s = mkdir(path, mode)</code>	Create a new directory
<code>s = rmdir(path)</code>	Remove a directory
<code>s = link(oldpath, newpath)</code>	Create a link to an existing file
<code>s = unlink(path)</code>	Unlink a file
<code>s = chdir(path)</code>	Change the working directory
<code>dir = opendir(path)</code>	Open a directory for reading
<code>s = closedir(dir)</code>	Close a directory
<code>dirent = readdir(dir)</code>	Read one directory entry
<code>rewinddir(dir)</code>	Rewind a directory so it can be reread

Figure 10-29. Some system calls relating to directories. The return code *s* is `-1` if an error has occurred; *dir* identifies a directory stream, and *dirent* is a directory entry. The parameters should be self explanatory.

As we saw in Fig. 10-24, linking to a file creates a new directory entry that points to an existing file. The `link` system call creates the link. The parameters specify the original and new names, respectively. Directory entries are removed with `unlink`. When the last link to a file is removed, the file is automatically deleted. For a file that has never been linked, the first `unlink` causes it to disappear.

The working directory is changed by the `chdir` system call. Doing so has the effect of changing the interpretation of relative path names.

The last four calls of Fig. 10-29 are for reading directories. They can be opened, closed, and read, analogous to ordinary files. Each call to `readdir` returns exactly one directory entry in a fixed format. There is no way for users to write in a directory (in order to maintain the integrity of the file system). Files can be added to a directory using `creat` or `link` and removed using `unlink`. There is also no way to seek to a specific file in a directory, but `rewinddir` allows an open directory to be read again from the beginning.

10.6.3 Implementation of the Linux File System

In this section we will first look at the abstractions supported by the Virtual File System layer. The VFS hides from higher-level processes and applications the differences among many types of file systems supported by Linux, whether they are residing on local devices or are stored remotely and need to be accessed over the network. Devices and other special files are also accessed through the VFS layer. Next, we will describe the implementation of the first widespread Linux file system, ext2, or the second extended file system. Afterward, we will discuss the improvements in the ext4 file system. A wide variety of other file systems are also in use. All Linux systems can handle multiple disk partitions, each with a different file system on it.

The Linux Virtual File System

In order to enable applications to interact with different file systems, implemented on different types of local or remote devices, Linux takes an approach used in other UNIX systems: the Virtual File System (VFS). VFS defines a set of basic file-system abstractions and the operations which are allowed on these abstractions. Invocations of the system calls described in the previous section access the VFS data structures, determine the exact file system where the accessed file belongs, and via function pointers stored in the VFS data structures invoke the corresponding operation in the specified file system.

Figure 10-30 summarizes the four main file-system structures supported by VFS. The **superblock** contains critical information about the layout of the file system. Destruction of the superblock will render the file system unreadable. The **i-nodes** (short for index-nodes, but never called that, although some lazy people drop the hyphen and call them **inodes**) each describe exactly one file. Note that in Linux, directories and devices are also represented as files, thus they will have corresponding i-nodes. Both superblocks and i-nodes have a corresponding structure maintained on the physical disk where the file system resides.

Object	Description	Operation
Superblock	specific file-system	read_inode, sync_fs
Dentry	directory entry, single component of a path	create, link
I-node	specific file	d_compare, d_delete
File	open file associated with a process	read, write

Figure 10-30. File-system abstractions supported by the VFS.

In order to facilitate certain directory operations and traversals of paths, such as `/usr/ast/bin`, VFS supports a **dentry** data structure which represents a directory entry. This data structure is created by the file system on the fly. Directory entries

are cached in what is called the *dentry_cache*. For instance, the *dentry_cache* would contain entries for */*, */usr*, */usr/ast*, and the like. If multiple processes access the same file through the same hard link (i.e., same path), their file object will point to the same entry in this cache.

Finally, the **file** data structure is an in-memory representation of an open file, and is created in response to the *open* system call. It supports operations such as read, write, *sendfile*, lock, and other system calls described in the previous section.

The actual file systems implemented underneath the VFS need not use the exact same abstractions and operations internally. They must, however, implement file-system operations semantically equivalent to those specified with the VFS objects. The elements of the *operations* data structures for each of the four VFS objects are pointers to functions in the underlying file system.

The Linux Ext2 File System

We next describe one of the most popular on-disk file systems used in Linux: **ext2**. The first Linux release used the MINIX 1 file system and was limited by short file names and 64-MB file sizes. The MINIX 1 file system was eventually replaced by the first extended file system, **ext**, which permitted both longer file names and larger file sizes. Due to its performance inefficiencies, *ext* was replaced by its successor, *ext2*, which is still in widespread use.

An *ext2* Linux disk partition contains a file system with the layout shown in Fig. 10-31. Block 0 is not used by Linux and contains code to boot the computer. Following block 0, the disk partition is divided into groups of blocks, irrespective of where the disk cylinder boundaries fall. Each group is organized as follows.

The first block is the **superblock**. It contains information about the layout of the file system, including the number of i-nodes, the number of disk blocks, and the start of the list of free disk blocks (typically a few hundred entries). Next comes the group descriptor, which contains information about the location of the bitmaps, the number of free blocks and i-nodes in the group, and the number of directories in the group. This information is important since *ext2* attempts to spread directories evenly over the disk.

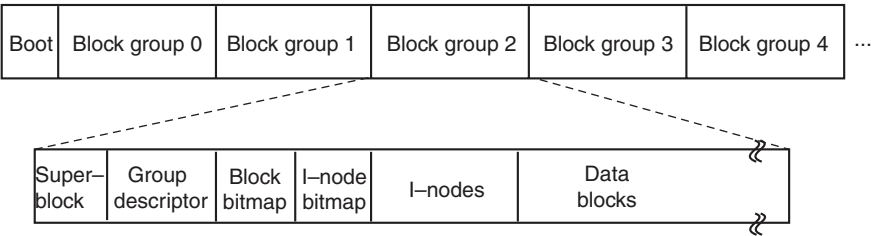


Figure 10-31. Disk layout of the Linux *ext2* file system.

Two bitmaps are used to keep track of the free blocks and free i-nodes, respectively, a choice inherited from the MINIX 1 file system (and in contrast to most UNIX file systems, which use a free list). Each map is one block long. With a 1-KB block, this design limits a block group to 8192 blocks and 8192 i-nodes. The former is a real restriction but, in practice, the latter is not. With 4-KB blocks, the numbers are four times larger.

Following the superblock are the i-nodes themselves. They are numbered from 1 up to some maximum. Each i-node is 128 bytes long and describes exactly one file. An i-node contains accounting information (including all the information returned by `stat`, which simply takes it from the i-node), as well as enough information to locate all the disk blocks that hold the file's data.

Following the i-nodes are the data blocks. All the files and directories are stored here. If a file or directory consists of more than one block, the blocks need not be contiguous on the disk. In fact, the blocks of a large file are likely to be spread all over the disk.

I-nodes corresponding to directories are dispersed throughout the disk block groups. Ext2 makes an effort to collocate ordinary files in the same block group as the parent directory, and data files in the same block as the original file i-node, provided that there is sufficient space. This idea was borrowed from the Berkeley Fast File System (McKusick et al., 1984). The bitmaps are used to make quick decisions regarding where to allocate new file-system data. When new file blocks are allocated, ext2 also *preallocates* a number (eight) of additional blocks for that file, so as to minimize the file fragmentation due to future write operations. This scheme balances the file-system load across the entire disk. It also performs well due to its tendencies for collocation and reduced fragmentation.

To access a file, it must first use one of the Linux system calls, such as `open`, which requires the file's path name. The path name is parsed to extract individual directories. If a relative path is specified, the lookup starts from the process' current directory, otherwise it starts from the root directory. In either case, the i-node for the first directory can easily be located: there is a pointer to it in the process descriptor, or, in the case of a root directory, it is typically stored in a predetermined block on disk.

The directory file allows file names up to 255 characters and is illustrated in Fig. 10-32. Each directory consists of some integral number of disk blocks so that directories can be written atomically to the disk. Within a directory, entries for files and directories are in unsorted order, with each entry directly following the one before it. Entries may not span disk blocks, so often there are some number of unused bytes at the end of each disk block.

Each directory entry in Fig. 10-32 consists of four fixed-length fields and one variable-length field. The first field is the i-node number, 19 for the file *colossal*, 42 for the file *voluminous*, and 88 for the directory *bigdir*. Next comes a field `rec_len`, telling how big the entry is (in bytes), possibly including some padding after the name. This field is needed to find the next entry for the case that the file

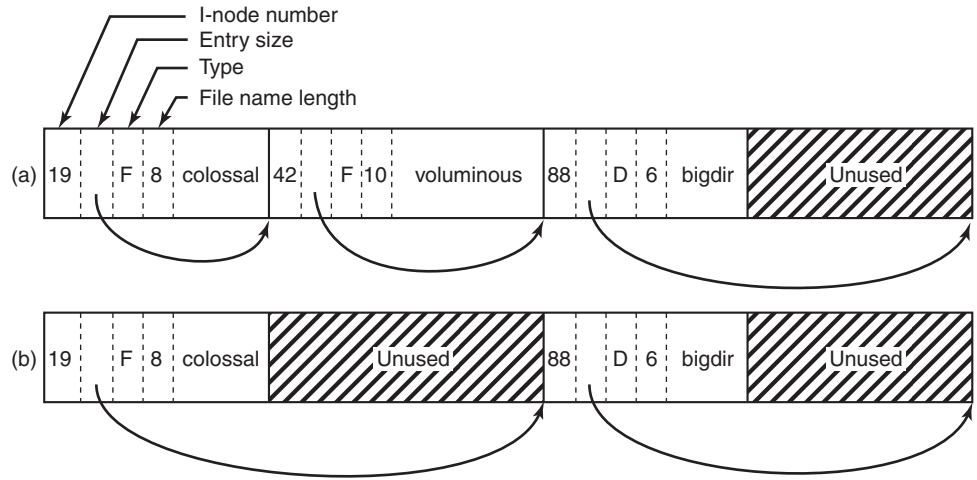


Figure 10-32. (a) A Linux directory with three files. (b) The same directory after the file *voluminous* has been removed.

name is padded by an unknown length. That is the meaning of the arrow in Fig. 10-32. Then comes the type field: file, directory, and so on. The last fixed field is the length of the actual file name in bytes, 8, 10, and 6 in this example. Finally, comes the file name itself, terminated by a 0 byte and padded out to a 32-bit boundary. Additional padding may follow that.

In Fig. 10-32(b) we see the same directory after the entry for *voluminous* has been removed. All the removal has done is increase the size of the total entry field for *colossal*, turning the former field for *voluminous* into padding for the first entry. This padding can be used for a subsequent entry, of course.

Since directories are searched linearly, it can take a long time to find an entry at the end of a large directory. Therefore, the system maintains a cache of recently accessed directories. This cache is searched using the name of the file, and if a hit occurs, the costly linear search is avoided. A *dentry* object is entered in the dentry cache for each of the path components, and, through its i-node, the directory is searched for the subsequent path element entry, until the actual file i-node is reached.

For instance, to look up a file specified with an absolute path name, such as */usr/ast/file*, the following steps are required. First, the system locates the root directory, which generally uses i-node 2, especially when i-node 1 is reserved for bad-block handling. It places an entry in the dentry cache for future lookups of the root directory. Then it looks up the string “usr” in the root directory, to get the i-node number of the */usr* directory, which is also entered in the dentry cache. This i-node is then fetched, and the disk blocks are extracted from it, so the */usr* directory can be read and searched for the string “ast”. Once this entry is found, the i-node

number for the `/usr/ast` directory can be taken from it. Armed with the i-node number of the `/usr/ast` directory, this i-node can be read and the directory blocks located. Finally, “file” is looked up and its i-node number found. Thus, the use of a relative path name is not only more convenient for the user, but it also saves a substantial amount of work for the system.

If the file is present, the system extracts the i-node number and uses it as an index into the i-node table (on disk) to locate the corresponding i-node and bring it into memory. The i-node is put in the **i-node table**, a kernel data structure that holds all the i-nodes for currently open files and directories. The format of the i-node entries, as a bare minimum, must contain all the fields returned by the `stat` system call so as to make `stat` work (see Fig. 10-28). In Fig. 10-33 we show some of the fields included in the i-node structure supported by the Linux file-system layer. The actual i-node structure contains many more fields, since the same structure is also used to represent directories, devices, and other special files. The i-node structure also contains fields reserved for future use. History has shown that unused bits do not remain that way for long.

Field	Bytes	Description
Mode	2	File type, protection bits, setuid, setgid bits
Nlinks	2	Number of directory entries pointing to this i-node
Uid	2	UID of the file owner
Gid	2	GID of the file owner
Size	4	File size in bytes
Addr	60	Address of first 12 disk blocks, then 3 indirect blocks
Gen	1	Generation number (incremented every time i-node is reused)
Atime	4	Time the file was last accessed
Mtime	4	Time the file was last modified
Ctime	4	Time the i-node was last changed (except the other times)

Figure 10-33. Some fields in the i-node structure in Linux.

Let us now see how the system reads a file. Remember that a typical call to the library procedure for invoking the `read` system call looks like this:

```
n = read(fd, buffer, nbytes);
```

When the kernel gets control, all it has to start with are these three parameters and the information in its internal tables relating to the user. One of the items in the internal tables is the file-descriptor array. It is indexed by a file descriptor and contains one entry for each open file (up to the maximum number, usually defaults to 32).

The idea is to start with this file descriptor and end up with the corresponding i-node. Let us consider one possible design: just put a pointer to the i-node in the file-descriptor table. Although simple, unfortunately this method does not work.

The problem is as follows. Associated with every file descriptor is a file position that tells at which byte the next read (or write) will start. Where should it go? One possibility is to put it in the i-node table. However, this approach fails if two or more unrelated processes happen to open the same file at the same time because each one has its own file position.

A second possibility is to put the file position in the file-descriptor table. In that way, every process that opens a file gets its own private file position. Unfortunately this scheme fails too, but the reasoning is more subtle and has to do with the nature of file sharing in Linux. Consider a shell script, *s*, consisting of two commands, *p1* and *p2*, to be run in order. If the shell script is called by the command

```
s >x
```

it is expected that *p1* will write its output to *x*, and then *p2* will write its output to *x* also, starting at the place where *p1* stopped.

When the shell forks off *p1*, *x* is initially empty, so *p1* just starts writing at file position 0. However, when *p1* finishes, some mechanism is needed to make sure that the initial file position that *p2* sees is not 0 (which it would be if the file position were kept in the file-descriptor table), but the value *p1* ended with.

The way this is achieved is shown in Fig. 10-34. The trick is to introduce a new table, the **open-file-description table**, between the file descriptor table and the i-node table, and put the file position (and read/write bit) there. In this figure, the parent is the shell and the child is first *p1* and later *p2*. When the shell forks off *p1*, its user structure (including the file-descriptor table) is an exact copy of the shell's, so both of them point to the same open-file-description table entry. When *p1* finishes, the shell's file descriptor is still pointing to the open-file description containing *p1*'s file position. When the shell now forks off *p2*, the new child automatically inherits the file position, without either it or the shell even having to know what that position is.

However, if an unrelated process opens the file, it gets its own open-file-description entry, with its own file position, which is precisely what is needed. Thus the whole point of the open-file-description table is to allow a parent and child to share a file position, but to provide unrelated processes with their own values.

Getting back to the problem of doing the read, we have now shown how the file position and i-node are located. The i-node contains the disk addresses of the first 12 blocks of the file. If the file position falls in the first 12 blocks, the block is read and the data are copied to the user. For files longer than 12 blocks, a field in the i-node contains the disk address of a **single indirect block**, as shown in Fig. 10-34. This block contains the disk addresses of more disk blocks. For example, if a block is 1 KB and a disk address is 4 bytes, the single indirect block can hold 256 disk addresses. Thus this scheme works for files of up to 268 KB.

Beyond that, a **double indirect block** is used. It contains the addresses of 256 single indirect blocks, each of which holds the addresses of 256 data blocks. This mechanism is sufficient to handle files up to $10 + 2^{16}$ blocks (67,119,104 bytes). If

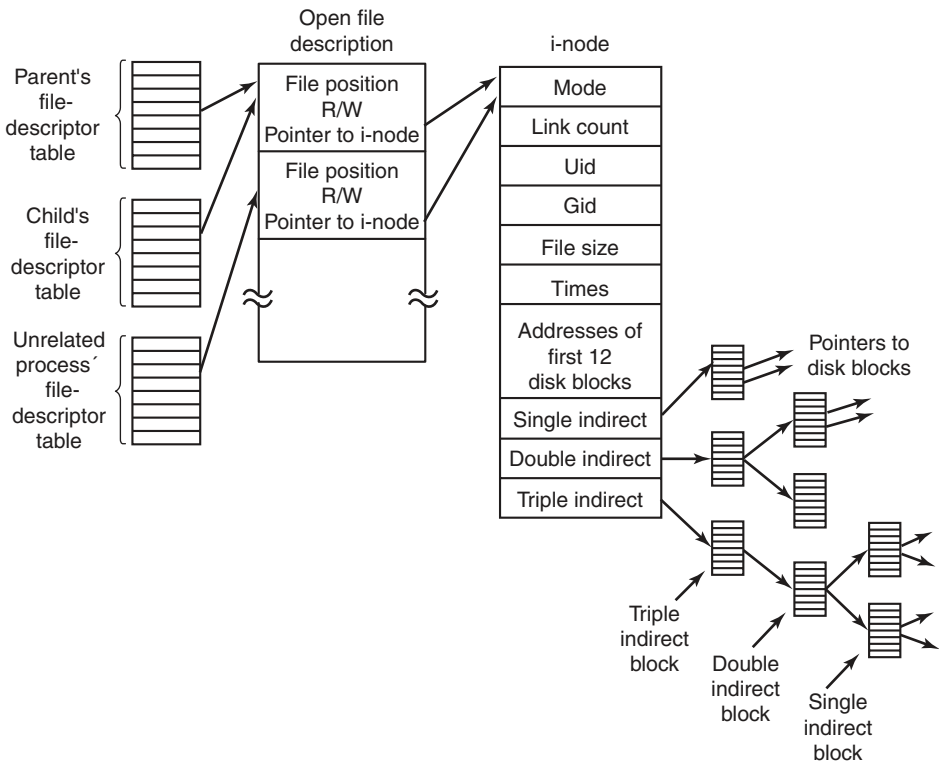


Figure 10-34. The relation between the file-descriptor table, the open-file-description-table, and the i-node table.

even this is not enough, the i-node has space for a **triple indirect block**. Its pointers point to many double indirect blocks. This addressing scheme can handle file sizes of 2^{24} 1-KB blocks (16 GB). For 8-KB block sizes, the addressing scheme can support file sizes up to 64 TB.

The Linux Ext4 File System

In order to prevent all data loss after system crashes and power failures, the ext2 file system would have to write out each data block to disk as soon as it was created. The latency incurred during the required disk-head seek operation would be so high that the performance would be intolerable. Therefore, writes are delayed, and changes may not be committed to disk for up to 30 sec, which is a very long time interval in the context of modern computer hardware.

To improve the robustness of the file system, Linux relies on **journaling file systems**. **Ext3**, a successor of the ext2 file system, is an example of a journaling file system. **Ext4**, a follow-on of ext3, is also a journaling file system, but unlike

ext3, it changes the block addressing scheme used by its predecessors, thereby supporting both larger files and larger overall file-system sizes. We will describe some of its features next.

The basic idea behind a journaling file system is to maintain a *journal*, which describes all file-system operations in sequential order. By sequentially writing out changes to the file-system data or metadata (i-nodes, superblock, etc.), the operations do not suffer from the overheads of disk-head movement during random disk accesses. Eventually, the changes will be written out, committed, to the appropriate disk location, and the corresponding journal entries can be discarded. If a system crash or power failure occurs before the changes are committed, during restart the system will detect that the file system was not unmounted properly, traverse the journal, and apply the file-system changes described in the journal log.

Ext4 is designed to be highly compatible with ext2 and ext3, although its core data structures and disk layout are modified. Regardless, a file system which has been unmounted as an ext2 system can be subsequently mounted as an ext4 system and offer the journaling capability.

The journal is a file managed as a circular buffer. The journal may be stored on the same or a separate device from the main file system. Since the journal operations are not "journaled" themselves, these are not handled by the same ext4 file system. Instead, a separate **JBD (Journaling Block Device)** is used to perform the journal read/write operations.

JBD supports three main data structures: *log record*, *atomic operation handle*, and *transaction*. A log record describes a low-level file-system operation, typically resulting in changes within a block. Since a system call such as write includes changes at multiple places—i-nodes, existing file blocks, new file blocks, list of free blocks, etc.—related log records are grouped in atomic operations. Ext4 notifies JBD of the start and end of system-call processing, so that JBD can ensure that either all log records in an atomic operation are applied, or none of them. Finally, primarily for efficiency reasons, JBD treats collections of atomic operations as transactions. Log records are stored consecutively within a transaction. JBD will allow portions of the journal file to be discarded only after all log records belonging to a transaction are safely committed to disk.

Since writing out a log entry for each disk change may be costly, ext4 may be configured to keep a journal of all disk changes, or only of changes related to the file-system metadata (the i-nodes, superblocks, etc.). Journaling only metadata gives less system overhead and results in better performance but does not make any guarantees against corruption of file data. Several other journaling file systems maintain logs of only metadata operations (e.g., SGI's XFS). In addition, the reliability of the journal can be further improved via checksumming.

Key modification in ext4 compared to its predecessors is the use of **extents**. Extents represent contiguous blocks of storage, for instance 128 MB of contiguous 4-KB blocks vs. individual storage blocks, as referenced in ext2. Unlike its predecessors, ext4 does not require metadata operations for each block of storage. This

scheme also reduces fragmentation for large files. As a result, ext4 can provide faster file system operations and support larger files and file system sizes. For instance, for a block size of 1 KB, ext4 increases the maximum file size from 16 GB to 16 TB, and the maximum file system size to 1 EB (Exabyte).

The /proc File System

Another Linux file system is the **/proc** (process) file system, an idea originally devised in the 8th edition of UNIX from Bell Labs and later copied in 4.4BSD and System V. However, Linux extends the idea in several ways. The basic concept is that for every process in the system, a directory is created in */proc*. The name of the directory is the process PID expressed as a decimal number. For example, */proc/619* is the directory corresponding to the process with PID 619. In this directory are files that appear to contain information about the process, such as its command line, environment strings, and signal masks. In fact, these files do not exist on the disk. When they are read, the system retrieves the information from the actual process as needed and returns it in a standard format.

Many of the Linux extensions relate to other files and directories located in */proc*. They contain a wide variety of information about the CPU, disk partitions, devices, interrupt vectors, kernel counters, file systems, loaded modules, and much more. Unprivileged user programs may read much of this information to learn about system behavior in a safe way. Some of these files may be written to in order to change system parameters.

10.6.4 NFS: The Network File System

Networking has played a major role in Linux, and UNIX in general, right from the beginning (the first UNIX network was built to move new kernels from the PDP-11/70 to the Interdata 8/32 during the port to the latter). In this section we will examine Sun Microsystem's **NFS (Network File System)**, which is used on all modern Linux systems to join the file systems on separate computers into one logical whole. Currently, the dominant NSF implementation is version 3, introduced in 1994. NFSv4 was introduced in 2000 and provides several enhancements over the previous NFS architecture. Three aspects of NFS are of interest: the architecture, the protocol, and the implementation. We will now examine these in turn, first in the context of the simpler NFS version 3, then we will turn to the enhancements included in v4.

NFS Architecture

The basic idea behind NFS is to allow an arbitrary collection of clients and servers to share a common file system. In many cases, all the clients and servers are on the same LAN, but this is not required. It is also possible to run NFS over a

wide area network if the server is far from the client. For simplicity we will speak of clients and servers as though they were on distinct machines, but in fact, NFS allows every machine to be both a client and a server at the same time.

Each NFS server exports one or more of its directories for access by remote clients. When a directory is made available, so are all of its subdirectories, so actually entire directory trees are normally exported as a unit. The list of directories a server exports is maintained in a file, often */etc/exports*, so these directories can be exported automatically whenever the server is booted. Clients access exported directories by mounting them. When a client mounts a (remote) directory, it becomes part of its directory hierarchy, as shown in Fig. 10-35.

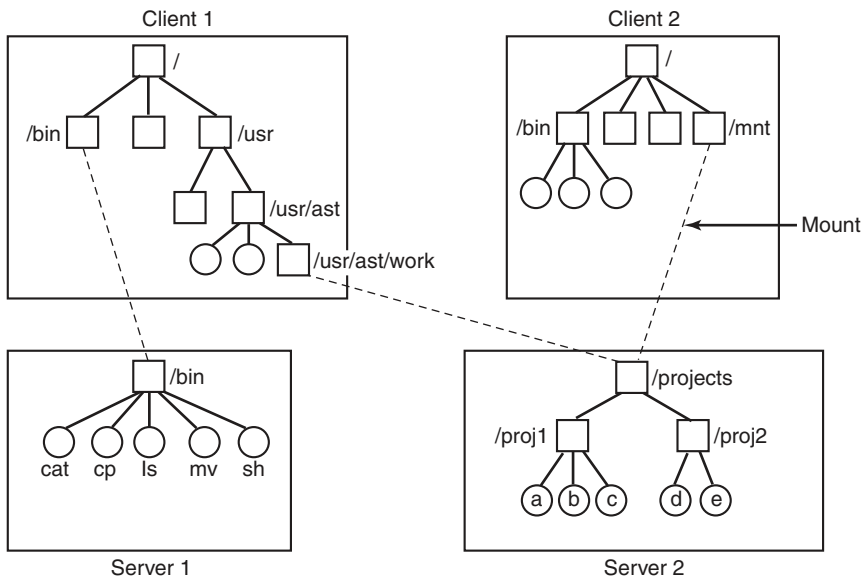


Figure 10-35. Examples of remote mounted file systems. Directories are shown as squares and files as circles.

In this example, client 1 has mounted the *bin* directory of server 1 on its own *bin* directory, so it can now refer to the shell as */bin/sh* and get the shell on server 1. Diskless workstations often have only a skeleton file system (in RAM) and get all their files from remote servers like this. Similarly, client 1 has mounted server 2's directory */projects* on its directory */usr/ast/work* so it can now access file *a* as */usr/ast/work/proj1/a*. Finally, client 2 has also mounted the *projects* directory and can also access file *a*, only as */mnt/proj1/a*. As seen here, the same file can have different names on different clients due to its being mounted in a different place in the respective trees. The mount point is entirely local to the clients; the server does not know where it is mounted on any of its clients.

NFS Protocols

Since one of the goals of NFS is to support a heterogeneous system, with clients and servers possibly running different operating systems on different hardware, it is essential that the interface between the clients and servers be well defined. Only then is anyone able to write a new client implementation and expect it to work correctly with existing servers, and vice versa.

NFS accomplishes this goal by defining two client-server protocols. A **protocol** is a set of requests sent by clients to servers, along with the corresponding replies sent by the servers back to the clients.

The first NFS protocol handles mounting. A client can send a path name to a server and request permission to mount that directory somewhere in its directory hierarchy. The place where it is to be mounted is not contained in the message, as the server does not care where it is to be mounted. If the path name is legal and the directory specified has been exported, the server returns a **file handle** to the client. The file handle contains fields uniquely identifying the file-system type, the disk, the i-node number of the directory, and security information. Subsequent calls to read and write files in the mounted directory or any of its subdirectories use the file handle.

When Linux boots, it runs the */etc/rc* shell script before going multiuser. Commands to mount remote file systems can be placed in this script, thus automatically mounting the necessary remote file systems before allowing any logins. Alternatively, most versions of Linux also support **automounting**. This feature allows a set of remote directories to be associated with a local directory. None of these remote directories are mounted (or their servers even contacted) when the client is booted. Instead, the first time a remote file is opened, the operating system sends a message to each of the servers. The first one to reply wins, and its directory is mounted.

Automounting has two principal advantages over static mounting via the */etc/rc* file. First, if one of the NFS servers named in */etc/rc* happens to be down, it is impossible to bring the client up, at least not without some difficulty, delay, and quite a few error messages. If the user does not even need that server at the moment, all that work is wasted. Second, by allowing the client to try a set of servers in parallel, a degree of fault tolerance can be achieved (because only one of them needs to be up), and the performance can be improved (by choosing the first one to reply—presumably the least heavily loaded).

On the other hand, it is tacitly assumed that all the file systems specified as alternatives for the automount are identical. Since NFS provides no support for file or directory replication, it is up to the user to arrange for all the file systems to be the same. Consequently, automounting is most often used for read-only file systems containing system binaries and other files that rarely change.

The second NFS protocol is for directory and file access. Clients can send messages to servers to manipulate directories and read and write files. They can

also access file attributes, such as file mode, size, and time of last modification. Most Linux system calls are supported by NFS, with the perhaps surprising exceptions of open and close.

The omission of open and close is not an accident. It is fully intentional. It is not necessary to open a file before reading it, nor to close it when done. Instead, to read a file, a client sends the server a lookup message containing the file name, with a request to look it up and return a file handle, which is a structure that identifies the file (i.e., contains a file system identifier and i-node number, among other data). Unlike an open call, this lookup operation does not copy any information into internal system tables. The read call contains the file handle of the file to read, the offset in the file to begin reading, and the number of bytes desired. Each such message is self-contained. The advantage of this scheme is that the server does not have to remember anything about open connections in between calls to it. Thus if a server crashes and then recovers, no information about open files is lost, because there is none. A server like this that does not maintain state information about open files is said to be **stateless**.

Unfortunately, the NFS method makes it difficult to achieve the exact Linux file semantics. For example, in Linux a file can be opened and locked so that other processes cannot access it. When the file is closed, the locks are released. In a stateless server such as NFS, locks cannot be associated with open files, because the server does not know which files are open. NFS therefore needs a separate, additional mechanism to handle locking.

NFS uses the standard UNIX protection mechanism, with the *rw**x* bits for the owner, group, and others (mentioned in Chap. 1 and discussed in detail below). Originally, each request message simply contained the user and group IDs of the caller, which the NFS server used to validate the access. In effect, it trusted the clients not to cheat. Several years' experience abundantly demonstrated that such an assumption was—how shall we put it?—rather naive. Currently, public key cryptography can be used to establish a secure key for validating the client and server on each request and reply. When this option is used, a malicious client cannot impersonate another client because it does not know that client's secret key.

NFS Implementation

Although the implementation of the client and server code is independent of the NFS protocols, most Linux systems use a three-layer implementation similar to that of Fig. 10-36. The top layer is the system-call layer. This handles calls like open, read, and close. After parsing the call and checking the parameters, it invokes the second layer, the Virtual File System (VFS) layer.

The task of the VFS layer is to maintain a table with one entry for each open file. The VFS layer additionally has an entry, a **virtual i-node**, or **v-node**, for every open file. V-nodes are used to tell whether the file is local or remote. For remote files, enough information is provided to be able to access them. For local files, the

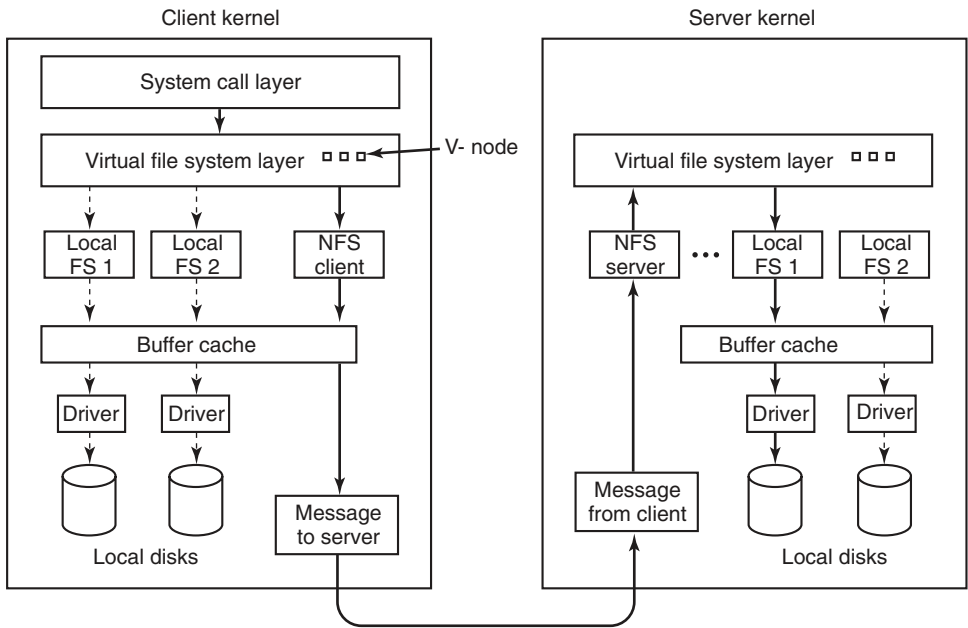


Figure 10-36. The NFS layer structure

file system and i-node are recorded because modern Linux systems can support multiple file systems (e.g., ext2fs, /proc, FAT, etc.). Although VFS was invented to support NFS, most modern Linux systems now support it as an integral part of the operating system, even if NFS is not used.

To see how v-nodes are used, let us trace a sequence of *mount*, *open*, and *read* system calls. To mount a remote file system, the system administrator (or */etc/rc*) calls the *mount* program specifying the remote directory, the local directory on which it is to be mounted, and other information. The *mount* program parses the name of the remote directory to be mounted and discovers the name of the NFS server on which the remote directory is located. It then contacts that machine, asking for a file handle for the remote directory. If the directory exists and is available for remote mounting, the server returns a file handle for the directory. Finally, it makes a mount system call, passing the handle to the kernel.

The kernel then constructs a v-node for the remote directory and asks the NFS client code in Fig. 10-36 to create an **r-node (remote i-node)** in its internal tables to hold the file handle. The v-node points to the r-node. Each v-node in the VFS layer will ultimately contain either a pointer to an r-node in the NFS client code, or a pointer to an i-node in one of the local file systems (shown as dashed lines in Fig. 10-36). Thus, from the v-node it is possible to see if a file or directory is local or remote. If it is local, the correct file system and i-node can be located. If it is remote, the remote host and file handle can be located.

When a remote file is opened on the client, at some point during the parsing of the path name, the kernel hits the directory on which the remote file system is mounted. It sees that this directory is remote and in the directory's v-node finds the pointer to the r-node. It then asks the NFS client code to open the file. The NFS client code looks up the remaining portion of the path name on the remote server associated with the mounted directory and gets back a file handle for it. It makes an r-node for the remote file in its tables and reports back to the VFS layer, which puts in its tables a v-node for the file that points to the r-node. Again here we see that every open file or directory has a v-node that points to either an r-node or an i-node.

The caller is given a file descriptor for the remote file. This file descriptor is mapped onto the v-node by tables in the VFS layer. Note that no table entries are made on the server side. Although the server is prepared to provide file handles upon request, it does not keep track of which files happen to have file handles outstanding and which do not. When a file handle is sent to it for file access, it checks the handle, and if it is valid, uses it. Validation can include verifying an authentication key contained in the RPC headers, if security is enabled.

When the file descriptor is used in a subsequent system call, for example, `read`, the VFS layer locates the corresponding v-node, and from that determines whether it is local or remote and also which i-node or r-node describes it. It then sends a message to the server containing the handle, the file offset (which is maintained on the client side, not the server side), and the byte count. For efficiency reasons, transfers between client and server are done in large chunks, normally 8192 bytes, even if fewer bytes are requested.

When the request message arrives at the server, it is passed to the VFS layer there, which determines which local file system holds the requested file. The VFS layer then makes a call to that local file system to read and return the bytes. These data are then passed back to the client. After the client's VFS layer has gotten the 8-KB chunk it asked for, it automatically issues a request for the next chunk, so it will have it should it be needed shortly. This feature, known as **read ahead**, improves performance considerably.

For writes an analogous path is followed from client to server. Also, transfers are done in 8-KB chunks here, too. If a write system call supplies fewer than 8 KB of data, the data are just accumulated locally. Only when the entire 8-KB chunk is full is it sent to the server. However, when a file is closed, all of its data are sent to the server immediately.

Another technique used to improve performance is caching, as in ordinary UNIX. Servers cache data to avoid disk accesses, but this is invisible to the clients. Clients maintain two caches, one for file attributes (i-nodes) and one for file data. When either an i-node or a file block is needed, a check is made to see if it can be satisfied out of the cache. If so, network traffic can be avoided.

While client caching helps performance enormously, it also introduces some nasty problems. Suppose that two clients are both caching the same file block and

one of them modifies it. When the other one reads the block, it gets the old (stale) value. The cache is not coherent.

Given the potential severity of this problem, the NFS implementation does several things to mitigate it. For one, associated with each cache block is a timer. When the timer expires, the entry is discarded. Normally, the timer is 3 sec for data blocks and 30 sec for directory blocks. Doing this reduces the risk somewhat. In addition, whenever a cached file is opened, a message is sent to the server to find out when the file was last modified. If the last modification occurred after the local copy was cached, the cache copy is discarded and the new copy fetched from the server. Finally, once every 30 sec a cache timer expires, and all the dirty (i.e., modified) blocks in the cache are sent to the server. While not perfect, these patches make the system highly usable in most practical circumstances.

NFS Version 4

Version 4 of the Network File System was designed to simplify certain operations from its predecessor. In contrast to NFSv3, which is described above, NFSv4 is a **stateful** file system. This permits open operations to be invoked on remote files, since the remote NFS server will maintain all file-system-related structures, including the file pointer. Read operations then need not include absolute read ranges, but can be incrementally applied from the previous file-pointer position. This results in shorter messages, and also in the ability to bundle multiple NFSv3 operations in one network transaction.

The stateful nature of NFSv4 makes it easy to integrate the variety of NFSv3 protocols described earlier in this section into one coherent protocol. There is no need to support separate protocols for mounting, caching, locking, or secure operations. NFSv4 also works better with both Linux (and UNIX in general) and Windows file-system semantics.

10.7 SECURITY IN LINUX

Linux, as a clone of MINIX and UNIX, has been a multiuser system almost from the beginning. This history means that security and control of information was built in very early on. In the following sections, we will look at some of the security aspects of Linux.

10.7.1 Fundamental Concepts

The user community for a Linux system consists of some number of registered users, each of whom has a unique **UID (User ID)**. A UID is an integer between 0 and 65,535. Files (but also processes and other resources) are marked with the

UID of their owner. By default, the owner of a file is the person who created the file, although there is a way to change ownership.

Users can be organized into groups, which are also numbered with 16-bit integers called **GIDs (Group IDs)**. Assigning users to groups is done manually (by the system administrator) and consists of making entries in a system database telling which user is in which group. A user could be in one or more groups at the same time. For simplicity, we will not discuss this feature further.

The basic security mechanism in Linux is simple. Each process carries the UID and GID of its owner. When a file is created, it gets the UID and GID of the creating process. The file also gets a set of permissions determined by the creating process. These permissions specify what access the owner, the other members of the owner's group, and the rest of the users have to the file. For each of these three categories, potential accesses are read, write, and execute, designated by the letters *r*, *w*, and *x*, respectively. The ability to execute a file makes sense only if that file is an executable binary program, of course. An attempt to execute a file that has execute permission but which is not executable (i.e., does not start with a valid header) will fail with an error. Since there are three categories of users and 3 bits per category, 9 bits are sufficient to represent the access rights. Some examples of these 9-bit numbers and their meanings are given in Fig. 10-37.

Binary	Symbolic	Allowed file accesses
111000000	rwX-----	Owner can read, write, and execute
111111000	rwXrwX---	Owner and group can read, write, and execute
110100000	rw-r-----	Owner can read and write; group can read
110100100	rw-r--r--	Owner can read and write; all others can read
111101101	rwXr-Xr-X	Owner can do everything, rest can read and execute
000000000	-----	Nobody has any access
000000111	-----rwx	Only outsiders have access (strange, but legal)

Figure 10-37. Some example file-protection modes.

The first two entries in Fig. 10-37 allow the owner and the owner's group full access, respectively. The next one allows the owner's group to read the file but not to change it, and prevents outsiders from any access. The fourth entry is common for a data file the owner wants to make public. Similarly, the fifth entry is the usual one for a publicly available program. The sixth entry denies all access to all users. This mode is sometimes used for dummy files used for mutual exclusion because an attempt to create such a file will fail if one already exists. Thus if multiple processes simultaneously attempt to create such a file as a lock, only one of them will succeed. The last example is strange indeed, since it gives the rest of the world more access than the owner. However, its existence follows from the protection rules. Fortunately, there is a way for the owner to subsequently change the protection mode, even without having any access to the file itself.

The user with UID 0 is special and is called the **superuser** (or **root**). The superuser has the power to read and write all files in the system, no matter who owns them and no matter how they are protected. Processes with UID 0 also have the ability to make a small number of protected system calls denied to ordinary users. Normally, only the system administrator knows the superuser's password, although many undergraduates consider it a great sport to try to look for security flaws in the system so they can log in as the superuser without knowing the password. Management tends to frown on such activity.

Directories are files and have the same protection modes that ordinary files do except that the *x* bits refer to search permission instead of execute permission. Thus a directory with mode *rw~~x~~rx-r~~x~~-x* allows its owner to read, modify, and search the directory, but allows others only to read and search it, but not add or remove files from it.

Special files corresponding to the I/O devices have the same protection bits as regular files. This mechanism can be used to limit access to I/O devices. For example, the printer special file, */dev/lp*, could be owned by the root or by a special user, daemon, and have mode *rw-----* to keep everyone else from directly accessing the printer. After all, if everyone could just print at will, chaos would result.

Of course, having */dev/lp* owned by, say, daemon with protection mode *rw-----* means that nobody else can use the printer. While this would save many innocent trees from an early death, sometimes users do have a legitimate need to print something. In fact, there is a more general problem of allowing controlled access to all I/O devices and other system resources.

This problem was solved by adding a new protection bit, the **SETUID bit**, to the 9 protection bits discussed above. When a program with the SETUID bit on is executed, the **effective UID** for that process becomes the UID of the executable file's owner instead of the UID of the user who invoked it. When a process attempts to open a file, it is the effective UID that is checked, not the underlying real UID. By making the program that accesses the printer be owned by daemon but with the SETUID bit on, any user could execute it, and have the power of daemon (e.g., access to */dev/lp*) but only to run that program (which might queue print jobs for printing in an orderly fashion).

Many sensitive Linux programs are owned by the root but with the SETUID bit on. For example, the program that allows users to change their passwords, *passwd*, needs to write in the password file. Making the password file publicly writable would not be a good idea. Instead, there is a program that is owned by the root and which has the SETUID bit on. Although the program has complete access to the password file, it will change only the caller's password and not permit any other access to the password file.

In addition to the SETUID bit there is also a SETGID bit that works analogously, temporarily giving the user the effective GID of the program. In practice, this bit is rarely used, however.

10.7.2 Security System Calls in Linux

There are only a small number of system calls relating to security. The most important ones are listed in Fig. 10-38. The most heavily used security system call is `chmod`. It is used to change the protection mode. For example,

```
s = chmod("/usr/ast/newgame", 0755);
```

sets *newgame* to *rwxr-xr-x* so that everyone can run it (note that 0755 is an octal constant, which is convenient, since the protection bits come in groups of 3 bits). Only the owner of a file and the superuser can change its protection bits.

System call	Description
<code>s = chmod(path, mode)</code>	Change a file's protection mode
<code>s = access(path, mode)</code>	Check access using the real UID and GID
<code>uid = getuid()</code>	Get the real UID
<code>uid = geteuid()</code>	Get the effective UID
<code>gid = getgid()</code>	Get the real GID
<code>gid = getegid()</code>	Get the effective GID
<code>s = chown(path, owner, group)</code>	Change owner and group
<code>s = setuid(uid)</code>	Set the UID
<code>s = setgid(gid)</code>	Set the GID

Figure 10-38. Some system calls relating to security. The return code *s* is `-1` if an error has occurred; *uid* and *gid* are the UID and GID, respectively. The parameters should be self explanatory.

The `access` call tests to see if a particular access would be allowed using the real UID and GID. This system call is needed to avoid security breaches in programs that are SETUID and owned by the root. Such a program can do anything, and it is sometimes needed for the program to figure out if the user is allowed to perform a certain access. The program cannot just try it, because the access will always succeed. With the `access` call the program can find out if the access is allowed by the real UID and real GID.

The next four system calls return the real and effective UIDs and GIDs. The last three are allowed only for the superuser. They change a file's owner, and a process' UID and GID.

10.7.3 Implementation of Security in Linux

When a user logs in, the login program, *login* (which is SETUID root) asks for a login name and a password. It hashes the password and then looks in the password file, */etc/passwd*, to see if the hash matches the one there (networked systems work slightly differently). The reason for using hashes is to prevent the password

from being stored in unencrypted form anywhere in the system. If the password is correct, the login program looks in */etc/passwd* to see the name of the user's preferred shell, possibly *bash*, but possibly some other shell such as *csh* or *ksh*. The login program then uses *setuid* and *setgid* to give itself the user's UID and GID (remember, it started out as SETUID root). Then it opens the keyboard for standard input (file descriptor 0), the screen for standard output (file descriptor 1), and the screen for standard error (file descriptor 2). Finally, it executes the preferred shell, thus terminating itself.

At this point the preferred shell is running with the correct UID and GID and standard input, output, and error all set to their default devices. All processes that it forks off (i.e., commands typed by the user) automatically inherit the shell's UID and GID, so they also will have the correct owner and group. All files they create also get these values.

When any process attempts to open a file, the system first checks the protection bits in the file's i-node against the caller's effective UID and effective GID to see if the access is permitted. If so, the file is opened and a file descriptor returned. If not, the file is not opened and *-1* is returned. No checks are made on subsequent read or write calls. As a consequence, if the protection mode changes after a file is already open, the new mode will not affect processes that already have the file open.

The Linux security model and its implementation are essentially the same as in most other traditional UNIX systems.

10.8 ANDROID

Android is a relatively new operating system designed to run on mobile devices. It is based on the Linux kernel—Android introduces only a few new concepts to the Linux kernel itself, using most of the Linux facilities you are already familiar with (processes, user IDs, virtual memory, file systems, scheduling, etc.) in sometimes very different ways than they were originally intended.

In the five years since its introduction, Android has grown to be one of the most widely used smartphone operating systems. Its popularity has ridden the explosion of smartphones, and it is freely available for manufacturers of mobile devices to use in their products. It is also an open-source platform, making it customizable to a diverse variety of devices. It is popular not only for consumer-centric devices where its third-party application ecosystem is advantageous (such as tablets, televisions, game systems, and media players), but is increasingly used as the embedded OS for dedicated devices that need a **graphical user interface (GUI)** such as VOIP phones, smart watches, automotive dashboards, medical devices, and home appliances.

A large amount of the Android operating system is written in a high-level language, the Java programming language. The kernel and a large number of low-