10.3 PROCESSES IN UNIX

In the previous sections, we started out by looking at UNIX as viewed from the keyboard, that is, what the user sees at the terminal. 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 UNIX 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, allocate memory, open files, and do I/O.

Unfortunately, with so many versions of UNIX 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 only active entities in a UNIX system are the processes. UNIX 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. Most versions of UNIX allow a process to create additional threads once it starts executing.

UNIX is a multiprogramming system, so multiple, independent processes may be running at the same time. 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 automatically 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 UNIX 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 UNIX because each one is a separate process, independent of all other processes.

Processes are created in UNIX 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 as well.

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. 10-1.

```
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 10-1. Process creation in UNIX.

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 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 UNIX 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 call-

ed **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 up, the system stops running *sort* until *head* has removed some data from the pipe.

Processes can also communicate in another way: 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 a signal arrives. The choices are to ignore it, to catch it, or to let the signal kill the process (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 only send signals 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, it gets a a SIGFPE (floating-point exception) signal. The signals that are required by POSIX are listed in Fig. 10-2. Many UNIX systems have additional signals as well, but programs using them may not be portable to other versions of UNIX.

10.3.2 Process Management System Calls in UNIX

Let us now look at the UNIX system calls dealing with process management. The main ones are listed in Fig. 10-3. Fork is a good place to start the discussion. Fork is the only way to create a new process in UNIX systems. 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 core image is copied to create the child, subsequent changes in one of them do not affect the other one. 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

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 10-2. The signals required by POSIX.

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

Figure 10-3. 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 name suggests.

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). The third one determines whether the caller blocks or returns if no child is already terminated.

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. 10-4.

```
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 10-4. A highly simplified shell.

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, including *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 to the string "cp". Similarly, *argv*[1] would point to the 5-character string "file1" and *argv*[2] would point to the 5-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. 10-4, no environment is passed to the child, so 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**. 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 accidently 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. 10-2). 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.

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.

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 UNIX to suspend the process until the next signal arrives.

Thread Management System Calls

The first versions of UNIX did not have threads. That feature was added many years later. Initially there were many threads packages in use, but the proliferation of threads packages made writing portable code difficult. Eventually, the system calls used to manage threads were standardized as part of POSIX (P1003.1c).

The POSIX specification did not take a position on whether threads should be implemented in the kernel or in user space. The advantage of having user-space threads is that they can be implemented without having to change the kernel and thread switching is very efficient. The disadvantage of user-space threads is that if one thread blocks (e.g., on I/O, a semaphore, or a page fault), all the threads in the process block because the kernel thinks there is only one thread and does not schedule the process until the blocking thread is released. Thus the calls defined in P1003.1c were carefully chosen to be implementable either way. As long as user programs adhere carefully to the P1003.1c semantics, both implementations

should work correctly. The most commonly-used thread calls are listed in Fig. 10-5. When kernel threads are used, these calls are true system calls; when user threads are used, these calls are implemented entirely in a user-space run-time library.

(For the truly alert reader, note that we have a typographical problem now. If the kernel manages threads, then calls such as ''pthread_create,'' are system calls and following our convention should be set in Helvetica, like this: pthread_create. However, if they are simply user-space library calls, our convention for all procedure names is to use Times Italics, like this: *pthread_create*. Without prejudice, we will simply use Helvetica, also in the next chapter, in which it is never clear which Win32 API calls are really system calls. It could be worse: in the Algol 68 Report there was a period that changed the grammar of the language slightly when printed in the wrong font.)

Thread call	Description
pthread_create	Create a new thread in the caller's address space
pthread_exit	Terminate the calling thread
pthread_join	Wait for a thread to terminate
pthread_mutex_init	Create a new mutex
pthread_mutex_destroy	Destroy a mutex
pthread_mutex_lock	Lock a mutex
pthread_mutex_unlock	Unlock a mutex
pthread_cond_init	Create a condition variable
pthread_cond_destroy	Destroy a condition variable
pthread_cond_wait	Wait on a condition variable
pthread_cond_signal	Release one thread waiting on a condition variable

Figure 10-5. The principal POSIX thread calls.

Let us briefly examine the thread calls shown in Fig. 10-5. The first call, pthread_create, creates a new thread. It is called by

```
err = pthread_create(&tid, attr, function, arg);
```

This call creates a new thread in the current process running the code *function* with *arg* passed to it as a parameter. The new thread's ID is stored in memory at the location pointed to by the first parameters. The *attr* parameter can be used to specify certain attributes for the new thread, such as its scheduling priority. After successful completion, one more thread is running in the caller's address space than was before the call.

A thread that has done its job and wants to terminate calls pthread_exit. A thread can wait for another thread to exit by calling pthread_join. If the thread waited for has already exited, the pthread_join finishes immediately. Otherwise it

blocks.

Threads can synchronize using locks called **mutexes**. Typically a mutex guards some resource, such as a buffer shared by two threads. To make sure that only one thread at a time accesses the shared resource, threads are expected to lock the mutex before touching the resource and unlock it when they are done. As long as all threads obey this protocol, race conditions can be avoided. Mutexes are like binary semaphores, that is, semaphores that can take on only the values of 0 and 1. The name ''mutex'' comes from the fact that mutexes are used to ensure mutual exclusion on some resource.

Mutexes can be created and destroyed by the calls pthread_mutex_init and pthread_mutex_destroy, respectively. A mutex can be in one of two states: locked or unlocked. When a thread needs to set a lock on an unlocked mutex (using pthread_mutex_lock), the lock is set and the thread continues. However, when a thread tries to lock a mutex that is already locked, it blocks. When the locking thread is finished with the shared resource, it is expected to unlock the corresponding mutex by calling pthread_mutex_unlock.

Mutexes are intended for short-term locking, such as protecting a shared variable. They are not intended for long-term synchronization, such as waiting for a tape drive to become free. For long-term synchronization, **condition variables** are provided. These are created and destroyed by calls to pthread_cond_init and pthread_cond_destroy, respectively.

A condition variable is used by having one thread wait on it, and another thread signal it. For example, having discovered that the tape drive it needs is busy, a thread would do pthread_cond_wait on a condition variable that all the threads have agreed to associate with the tape drive. When the thread using the tape drive is finally done with it (possibly even hours later), it uses pthread_cond_signal to release exactly one thread waiting on that condition variable (if any). If no thread is waiting, the signal is lost. In other words, condition variables do not count like semaphores. A few other operations are also defined on threads, mutexes, and condition variables.

10.3.3 Implementation of Processes in UNIX

A process in UNIX is like an iceberg: what you see is 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 part way 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 kernel maintains two key data structures related to processes, the **process table** and the **user structure**. The process table is resident all the time and contains information needed for all processes, even those that are not currently present in memory. The user structure is swapped or paged out when its associated process is not in memory, in order not to waste memory on information that is not needed.

The information in the process table falls into the following broad categories:

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, if paging is used, to their 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. **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.

The user structure contains information that is not needed when the process is not physically in memory and runnable. 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 the process table, so they are 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 contained in the user structure includes the following items:

1. **Machine registers**. When a trap to the kernel occurs, the machine registers (including the floating-point ones, if used) are saved here.
2. **System call state**. Information about the current system call, including the parameters, and results.
3. **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.
4. **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.

5. **Kernel stack**. A fixed stack for use by the kernel part of the process.

Bearing the use of these tables in mind, it is now easy to explain how processes are created in UNIX. When a fork system call is executed, the calling process traps to the kernel and looks for a free slot in the process table for use by the child. If it finds one, it copies all the information from the parent's process table entry to the child's entry. It then allocates memory for the child's data and stack segments, and makes exact copies of the parent's data and stack segments there. The user structure (which is often kept adjacent to the stack segment), is copied along with the stack. The text segment may either be copied or shared since it is read only. At this point, the child is ready to run.

When a user types a command, say, 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*. The steps involved are shown in Fig. 10-6.

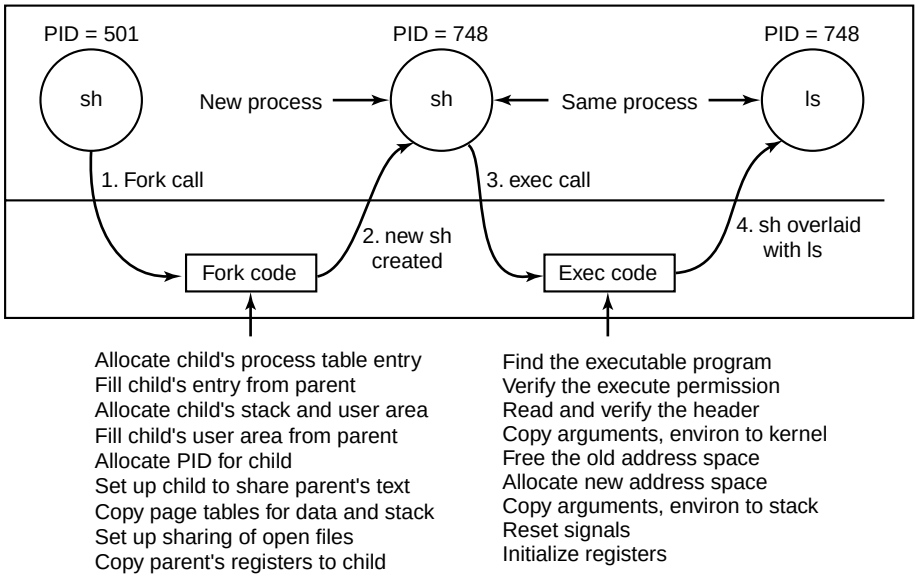

Figure 10-6. The steps in executing the command *ls* typed to the shell.

The mechanism for creating a new process is actually fairly straightforward. A new process table slot 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.

In principle, a complete copy of the address space should be made, since the

semantics of fork say that no memory is shared between parent and child. However, copying memory is expensive, so all modern UNIX systems cheat. They give the child its own page tables, but have them point to the parent's pages, only marked read only. Whenever the child 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 child 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) 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 System V, BSD, and most other UNIX 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. 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.

Threads in UNIX

The implementation of threads depends on whether they are supported in the kernel or not. If they are not, such as in 4BSD, the implementation is entirely in a user-space library. If they are, as in System V and Solaris, the kernel has some work to do. We discussed threads in a general way in Chap. 2. Here we will just make a few remarks about kernel threads in UNIX.

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 at the process in general? 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.

Threads in Linux

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.

The heart of the Linux implementation of threads is a new system call, clone, that is not present in any other version of UNIX. 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 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 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 much finer grain of sharing than traditional UNIX systems. Five bits are defined, as listed in Fig. 10-7. Each bit controls some aspect of sharing, and each of the bits can be set independently of the other ones. 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 address space. Having its own address space means that the

effect of its STORE instructions are 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.

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_PID	New thread gets old PID	New thread gets own PID

Figure 10-7. Bits in the *sharing_flags* bitmap.

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 UNIX.

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, *CLONE_PID* controls whether the new thread gets its own PID or shares its parent's PID. This feature is needed during system booting. User processes are not permitted to enable it.

This fine-grained sharing is possible because Linux maintains separate data structures for the various items listed at the start of Sec. 10.3.3 (scheduling parameters, memory image, etc.). The process table and user structure just point to these data structures, so it is easy to make a new process table entry for each cloned thread and have it either point 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 UNIX does not offer this functionality. A Linux program that takes advantage of it is then no longer portable to UNIX.

Scheduling in UNIX

Let us now examine the UNIX scheduling algorithm. Because UNIX has always been a multiprogramming system, its scheduling algorithm was designed from the beginning to provide good response to interactive processes. It is a two-level algorithm. The low-level algorithm picks the process to run next from the set of processes in memory and ready to run. The high-level algorithm moves processes between memory and disk so that all processes get a chance to be in memory and run.

Each version of UNIX has a slightly different low-level scheduling algorithm, but most of them are close to the generic one we will describe now. The low-level algorithm uses multiple queues. Each queue is associated with a range of nonoverlapping priority values. Processes executing in user mode (the top of the iceberg) have positive values. Processes executing in kernel mode (doing system calls) have negative values. Negative values have the highest priority and large positive values have the lowest, as illustrated in Fig. 10-8. Only processes that are in memory and ready to run are located on the queues, since the choice must be made from this set.

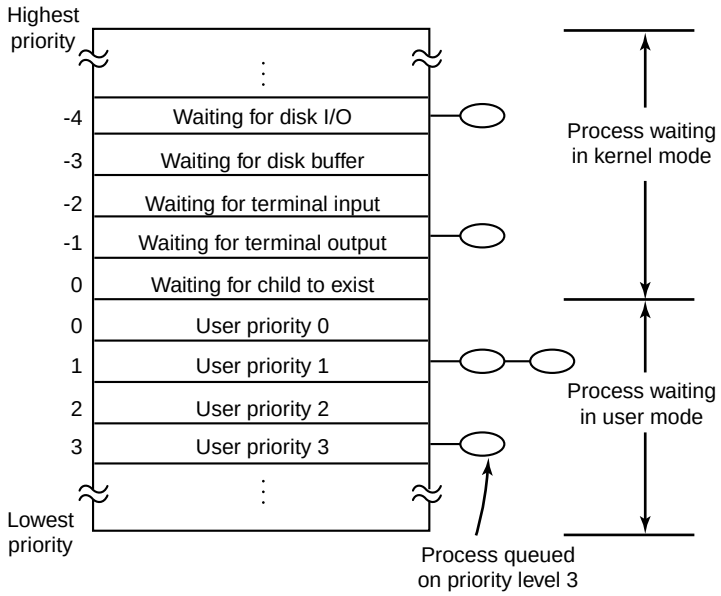

Figure 10-8. The UNIX scheduler is based on a multilevel queue structure.

When the (low-level) scheduler runs, it searches the queues starting at the highest priority (i.e., most negative value) until it finds a queue that is occupied. The first process on that queue is then chosen and started. It is allowed to run for a maximum of one quantum, typically 100 msec, or until it blocks. If a process

uses up its quantum, it is put back on the end of its queue, and the scheduling algorithm is run again. Thus processes within the same priority range share the CPU using a round-robin algorithm.

Once a second, each process' priority is recalculated according to a formula involving three components:

$$priority = CPU\_usage + nice + base$$

Based on its new priority, each process is attached to the appropriate queue of Fig. 10-8, usually by dividing the priority by a constant to get the queue number. Let us now briefly examine each of the three components of the priority formula.

CPU_usage, represents the average number of clock ticks per second that the process has had during the past few seconds. Every time the clock ticks, the CPU usage counter in the running process' process table entry is incremented by 1. This counter will ultimately be added to the process' priority giving it a higher numerical value and thus putting it on a lower-priority queue.

However, UNIX does not punish a process forever for using the CPU, so *CPU_usage* decays with time. Different versions of UNIX do the decay slightly differently. One way that has been used is to add the current value of *CPU_usage* to the number of ticks acquired in the past ΔT and divide the result by 2. This algorithm weights the most recent ΔT by ½, the one before that by ¼, and so on. This weighting algorithm is very fast because it just has one addition and one shift, but other weighting schemes have also been used.

Every process has a *nice* value associated with it. The default value is 0, but the allowed range is generally −20 to +20. A process can set *nice* to a value in the range 0 to 20 by the nice system call. 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.

When a process traps to the kernel to make a system call, it is entirely possible that the process has to block before completing the system call and returning to user mode. For example, it may have just done a waitpid system call and have to wait for one of its children to exit. It may also have to wait for terminal input or for disk I/O to complete, to mention only a few of the many possibilities. When it blocks, it is removed from the queue structure, since it is unable to run.

However, when the event it was waiting for occurs, it is put onto a queue with a negative value. The choice of queue is determined by the event it was waiting for. In Fig. 10-8, disk I/O is shown as having the highest priority, so a process that has just read or written a block from the disk will probably get the CPU within 100 msec. The relative priority of disk I/O, terminal I/O, etc. is hardwired into the operating system, and can only be modified by changing some constants in the source code and recompiling the system. These (negative) values are represented by *base* in the formula given above and are spaced far enough apart

that processes being restarted for different reasons are clearly separated into different queues.

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 slow it down enormously. It is far better to let it run immediately after each request is completed, so it can make the next one quickly. Similarly, if a process was blocked waiting for terminal 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 (i.e., those on the positive queues) basically get any service that is left over when all the I/O bound and interactive processes are blocked.

Scheduling in Linux

Scheduling is one of the few areas in which Linux uses a different algorithm from UNIX. We have just examined the UNIX scheduling algorithm, so we will now look at the Linux 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. Real-time round-robin threads are the same as real-time FIFO threads except that they 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.

Each thread has a scheduling priority. The default value is 20, but that can be altered using the nice(value) system call to a value of $20 - value$. Since *value* must be in the range −20 to +19, priorities always fall in the range: $1 \leq priority \leq 40$. The intention is that the quality of service is roughly proportional to the priority, with higher priority threads getting faster response time and a larger fraction of the CPU time than lower priority threads.

In addition to a priority, each thread has a quantum associated with it. The quantum is the number of clock ticks the thread may continue to run for. The clock runs at 100 Hz by default, so each tick is 10 msec, which is called a **jiffy**.

The scheduler uses the priority and quantum as follows. It first computes the **goodness** of each ready thread by applying the following rules:

```
if (class == real_time) goodness = 1000 + priority;
if (class == timesharing && quantum > 0) goodness = quantum + priority;
if (class == timesharing && quantum == 0) goodness = 0;
```

Both real-time classes count for the first rule. All that marking a thread as real time does is make sure it gets a higher goodness than all timesharing threads. The algorithm has one little extra feature: if the process that ran last still has some quantum left, it gets a bonus point, so that it wins any ties. The idea here is that all things being equal, it is more efficient to run the previous process since its pages and cache blocks are likely to be loaded.

Given this background, the scheduling algorithm is very simple: when a scheduling decision is made, the thread with the highest goodness is selected. As the selected thread runs, at every clock tick, its quantum is decremented by 1. The CPU is taken away from a thread if any of these conditions occur:

1. Its quantum hits 0.
2. The thread blocks on I/O, a semaphore, or something else.
3. A previously blocked thread with a higher goodness becomes ready.

Since the quanta keep counting down, sooner or later every ready thread will grind its quantum into the ground and they will all be 0. However, I/O bound threads that are currently blocked may have some quantum left. At this point the scheduler resets the quantum of *all* threads, ready and blocked, using the rule:

```
quantum = (quantum/2) + priority
```

where the new quantum is in jiffies. A thread that is highly compute bound will usually exhaust its quantum quickly and have it 0 when quanta are reset, giving it a quantum equal to its priority. An I/O-bound thread may have considerable quantum left and thus get a larger quantum next time. If `nice` is not used, the priority will be 20, so the quantum becomes 20 jiffies or 200 msec. On the other hand, for a highly I/O bound thread, it may still have a quantum of 20 left when quanta are reset, so if its priority is 20, its new quantum becomes $20/2 + 20 = 30$ jiffies. If another reset happens before it has spent 1 tick, next time it gets a quantum of $30/2 + 20 = 35$ jiffies. The asymptotic value in jiffies is twice the priority. As a consequence of this algorithm, I/O-bound threads get larger quanta and thus higher goodness than compute-bound threads. This gives I/O-bound threads preference in scheduling.

Another property of this algorithm is that when compute-bound threads are competing for the CPU, ones with higher priority get a larger fraction of it. To see this, consider two compute-bound threads, *A*, with priority 20 and *B*, with priority 5. *A* goes first and 20 ticks later has used up its quantum. Then *B* gets to

run for 5 quanta. At this point the quanta are reset. *A* gets 20 and *B* gets 5. This goes on forever, so *A* is getting 80% of the CPU and *B* is getting 20% of the CPU.

10.3.4 Booting UNIX

The exact details of how UNIX is booted vary from system to system. Below we will look briefly at how 4.4BSD is booted, but the ideas are somewhat similar for all versions. When the computer starts, the first sector of the boot disk (the master boot record) is read into memory and executed. This sector contains a small (512-byte) program that loads a standalone program called *boot* from the boot device, usually an IDE 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 it does. Then it reads in the operating system kernel and jumps to it. At this point, *boot* 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 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 fixed size, but a few, such as the buffer 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.

Once the device list has been determined, the device drivers must be located. This is one area in which UNIX systems differ somewhat. In particular, 4.4BSD cannot load device drivers dynamically, so any I/O device whose driver was not statically linked with the kernel cannot be used. In contrast, some other versions of UNIX, such as Linux, can load drivers dynamically (as can all versions of MS-DOS and Windows, incidentally).

The arguments for and against dynamically loading drivers are interesting and worth stating briefly. 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 execs 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/ttys*, 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 execs 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 types

```
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 illustrated in Fig. 10-9 for a system with three terminals.

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
```

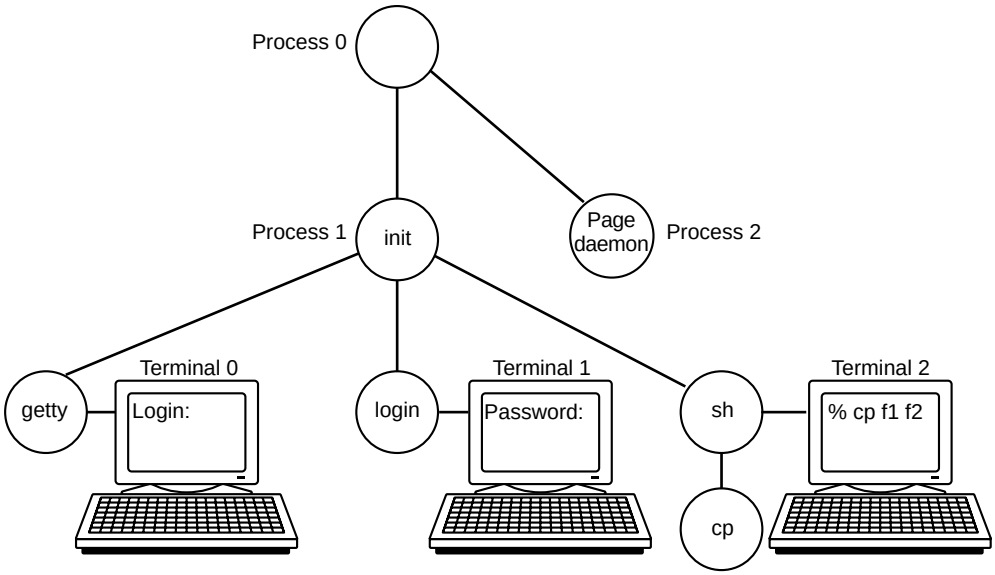

Figure 10-9. The sequence of processes used to boot some UNIX systems.

which has caused the shell to fork off a child process and have that process exec 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.