- Signals originating from a process in user mode, such as when a process wishes to receive an *alarm* signal after a period of time, or when processes send arbitrary signals to each other with the *kill* system call;
- Signals related to terminal interaction such as when a user hangs up a terminal (or the "carrier" signal drops on such a line for any reason), or when a user presses the "break" or "delete" keys on a terminal keyboard;
- Signals for tracing execution of a process.

The discussion in this and in following chapters explains the circumstances under which signals of the various classes are used.

The treatment of signals has several facets, namely how the kernel sends a signal to a process, how the process handles a signal, and how a process controls its reaction to signals. To send a signal to a process, the kernel sets a bit in the signal field of the process table entry, corresponding to the type of signal received. If the process is asleep at an interruptible priority, the kernel awakens it. The job of the sender (process or kernel) is complete. A process can remember different types of signals, but it has no memory of how many signals it receives of a particular type. For example, if a process receives a hangup signal and a kill signal, it sets the appropriate bits in the process table signal field, but it cannot tell how many instances of the signals it receives.

The kernel checks for receipt of a signal when a process is about to return from kernel mode to user mode and when it enters or leaves the sleep state at a suitably low scheduling priority (see Figure 7.6). The kernel handles signals only when a process returns from kernel mode to user mode. Thus, a signal does not have an instant effect on a process running in kernel mode. If a process is running in user mode, and the kernel handles an interrupt that causes a signal to be sent to the process, the kernel will recognize and handle the signal when it returns from the interrupt. Thus, a process never executes in user mode before handling outstanding signals.

Figure 7.7 shows the algorithm the kernel executes to determine if a process received a signal. The case for "death of child" signals will be treated later in the chapter. As will be seen, a process can choose to ignore signals with the *signal* system call. In the algorithm *issig*, the kernel simply turns off the signal indication for signals the process wants to ignore but notes the existence of signals it does not ignore.

---

1. The use of signals in some circumstances uncovers errors in programs that do not check for failure of system calls (private communication from D. Ritchie).

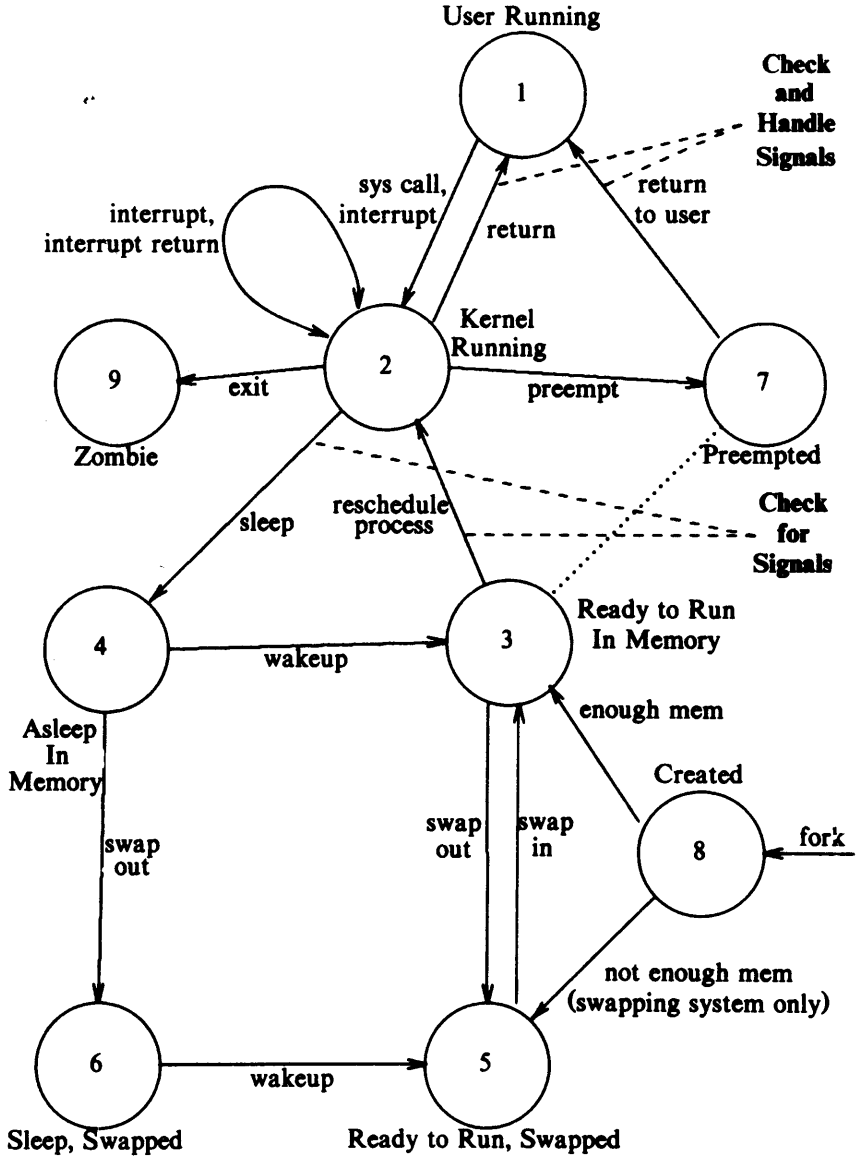

Figure 7.6. Checking and Handling Signals in the Process State Diagram

```
algorithm issig                          /* test for receipt of signals */
input:  none
output: true, if process received signals that it does not ignore
        false otherwise
{
    while (received signal field in process table entry not 0)
    {
        find a signal number sent to the process;
        if (signal is death of child)
        {
            if (ignoring death of child signals)
                free process table entries of zombie children;
            else if (catching death of child signals)
                return(true);
        }
        else if (not ignoring signal)
            return(true);
        turn off signal bit in received signal field in process table;
    }
    return(false);
}
```

Figure 7.7. Algorithm for Recognizing Signals

7.2.1 Handling Signals

The kernel handles signals in the context of the process that receives them so a process must run to handle signals. There are three cases for handling signals: the process *exit*s on receipt of the signal, it ignores the signal, or it executes a particular (user) function on receipt of the signal. The default action is to call *exit* in kernel mode, but a process can specify special action to take on receipt of certain signals with the *signal* system call.

The syntax for the *signal* system call is

```
oldfunction = signal(signum, function);
```

where *signum* is the signal number the process is specifying the action for, *function* is the address of the (user) function the process wants to invoke on receipt of the signal, and the return value *oldfunction* was the value of *function* in the most recently specified call to *signal* for *signum*. The process can pass the values 1 or 0 instead of a function address: The process will ignore future occurrences of the signal if the parameter value is 1 (Section 7.4 deals with the special case for ignoring the "death of child" signal) and *exit* in the kernel on receipt of the signal if its value is 0 (the default value). The *u area* contains an array of signal-handler fields, one for each signal defined in the system. The kernel stores the address of the user function in the field that corresponds to the signal number. Specification

```
algorithm psig   /* handle signals after recognizing their existence */
input:  none
output: none
{
        get signal number set in process table entry;
        clear signal number in process table entry;
        if (user had called signal sys call to ignore this signal)
                return;                 /* done */
        if (user specified function to handle the signal)
        {
                get user virtual address of signal catcher stored in u area;
                /* the next statement has undesirable side-effects */
                clear u area entry that stored address of signal catcher,
                modify user level context:
                                artificially create user stack frame to mimic
                                call to signal catcher function;
                modify system level context:
                                write address of signal catcher into program
                                counter field of user saved register context;
                return;
        }
        if (signal is type that system should dump core image of process)
        {
                create file named "core" in current directory;
                write contents of user level context to file "core";
        }
        invoke exit algorithm immediately;
}
```

Figure 7.8. Algorithm for Handling Signals

to handle signals of one type has no effect on handling signals of other types.

When handling a signal (Figure 7.8) the kernel determines the signal type and turns off the appropriate signal bit in the process table entry, set when the process received the signal. If the signal handling function is set to its default value, the kernel will dump a "core" image of the process (see exercise 7.7) for certain types of signals before *exit*ing. The dump is a convenience to programmers, allowing them to ascertain its causes and, thereby, to debug their programs. The kernel dumps core for signals that imply something is wrong with a process, such as when a process executes an illegal instruction or when it accesses an address outside its virtual address space. But the kernel does not dump core for signals that do not imply a program error. For instance, receipt of an interrupt signal, sent when a user hits the "delete" or "break" key on a terminal, implies that the user wants to terminate a process prematurely, and receipt of a hangup signal implies that the login terminal is no longer "connected." These signals do not imply that anything

is wrong with the process. The *quit* signal, however, induces a core dump even though it is initiated outside the running process. Usually sent by typing the control-vertical-bar character at the terminal, it allows the programmer to obtain a core dump of a running process, useful for one that is in an infinite loop.

When a process receives a signal that it had previously decided to ignore, it continues as if the signal had never occurred. Because the kernel does not reset the field in the *u area* that shows the signal is ignored, the process will ignore the signal if it happens again, too. If a process receives a signal that it had previously decided to catch, it executes the user specified signal handling function immediately when it returns to user mode, after the kernel does the following steps.

1. The kernel accesses the user saved register context, finding the program counter and stack pointer that it had saved for return to the user process.
2. It clears the signal handler field in the *u area*, setting it to the default state.
3. The kernel creates a new stack frame on the user stack, writing in the values of the program counter and stack pointer it had retrieved from the user saved register context and allocating new space, if necessary. The user stack looks as if the process had called a user-level function (the signal catcher) at the point where it had made the system call or where the kernel had interrupted it (before recognition of the signal).
4. The kernel changes the user saved register context: It resets the value for the program counter to the address of the signal catcher function and sets the value for the stack pointer to account for the growth of the user stack.

After returning from the kernel to user mode, the process will thus execute the signal handling function; when it returns from the signal handling function, it returns to the place in the user code where the system call or interrupt originally occurred, mimicking a return from the system call or interrupt.

For example, Figure 7.9 contains a program that catches interrupt signals (*SIGINT*) and sends itself an interrupt signal (the result of the *kill* call here), and Figure 7.10 contains relevant parts of a disassembly of the load module on a VAX 11/780. When the system executes the process, the call to the *kill* library routine comes from address (hexadecimal) ee, and the library routine executes the *chmk* (change mode to kernel) instruction at address 10a to call the *kill* system call. The return address from the system call is 10c. In executing the system call, the kernel sends an interrupt signal to the process. The kernel notices the interrupt signal when it is about to return to user mode, removes the address 10c from the user saved register context, and places it on the user stack. The kernel takes the address of the function *catcher*, 104, and puts it into the user saved register context. Figure 7.11 illustrates the states of the user stack and saved register context.

Several anomalies exist in the algorithm described here for the treatment of signals. First and most important, when a process handles a signal but before it returns to user mode, the kernel clears the field in the *u area* that contains the address of the user signal handling function. If the process wants to handle the signal again, it must call the *signal* system call again. This has unfortunate

```
#include <signal.h>
main()
{
        extern catcher();
        signal(SIGINT, catcher);
        kill(0, SIGINT);
}

catcher()
{
}
```

Figure 7.9. Source Code for a Program that Catches Signals

```
****    VAX    DISASSEMBLER    ****

_main()
                e4:
                e6:    pushab    0x18(pc)
                ec:    pushl     $0x2
    # next line calls signal
                ee:    calls     $0x2,0x23(pc)
                f5:    pushl     $0x2
                f7:    clrl      -(sp)
    # next line calls kill library routine
                f9:    calls     $0x2,0x8(pc)
               100:    ret
               101:    halt
               102:    halt
               103:    halt
_catcher()
               104:
               106:    ret
               107:    halt
_kill()
               108:
    # next line traps into kernel
               10a:    chmk      $0x25
               10c:    bgequ     0x6 <0x114>
               10e:    jmp       0x14(pc)
               114:    clrl      r0
               116:    ret
```

Figure 7.10. Disassembly of Program that Catches Signals

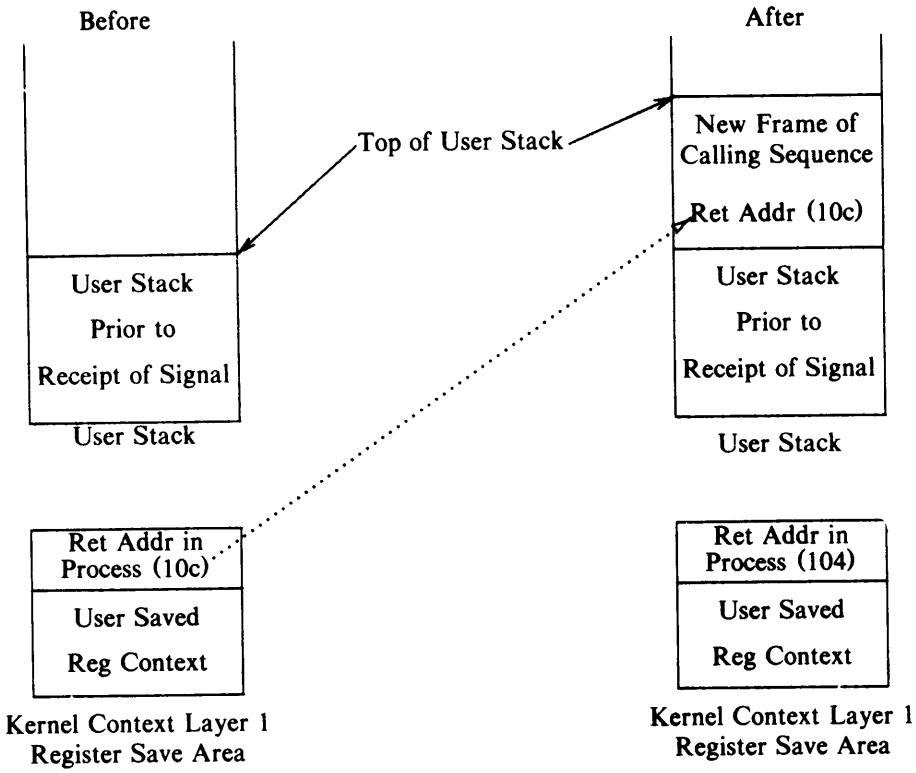

Figure 7.11. User Stack and Kernel Save Area Before and After Receipt of Signal

ramifications: A race condition results because a second instance of the signal may arrive before the process has a chance to invoke the system call. Since the process is executing in user mode, the kernel could do a context switch, increasing the chance that the process will receive the signal before resetting the signal catcher.

The program in Figure 7.12 illustrates the race condition. The process calls the *signal* system call to arrange to catch interrupt signals and execute the function *sigcatcher*. It then creates a child process, invokes the *nice* system call to lower its scheduling priority relative to the child process (see Chapter 8), and goes into an infinite loop. The child process suspends execution for 5 seconds to give the parent process time to execute the *nice* system call and lower its priority. The child process then goes into a loop, sending an interrupt signal (via *kill*) to the parent process during each iteration. If the *kill* returns because of an error, probably because the parent process no longer exists, the child process *exit*s. The idea is that the parent process should invoke the signal catcher every time it receives an interrupt signal. The signal catcher prints a message and calls *signal* again to

```
#include <signal.h>
sigcatcher()
{
        printf("PID %d caught one\n", getpid());      /* print proc id */
        signal(SIGINT, sigcatcher);
}

main()
{
        int ppid;

        signal(SIGINT, sigcatcher);

        if (fork() == 0)
        {
                /* give enough time for both procs to set up */
                sleep(5);               /* lib function to delay 5 secs */
                ppid = getppid();       /* get parent id */
                for (;;)
                        if (kill(ppid, SIGINT) == -1)
                                exit();
        }

        /* lower priority, greater chance of exhibiting race */
        nice(10);
        for (;;)

}
```

Figure 7.12. Program Demonstrating Race Condition in Catching Signals

catch the next occurrence of an interrupt signal, and the parent continues to execute in the infinite loop.

It is possible for the following sequence of events to occur, however.

1. The child process sends an interrupt signal to the parent process.
2. The parent process catches the signal and calls the signal catcher, but the kernel preempts the process and switches context before it executes the *signal* system call again.
3. The child process executes again and sends another interrupt signal to the parent process.
4. The parent process receives the second interrupt signal, but it has not made arrangements to catch the signal. When it resumes execution, it *exit*s.

The program was written to encourage such behavior, since invocation of the *nice* system call by the parent process induces the kernel to schedule the child process

more frequently. However, it is indeterminate when this result will occur.

According to Ritchie (private communication), signals were designed as events that are fatal or ignored, not necessarily handled, and hence the race condition was not fixed in early releases. However, it poses a serious problem to programs that want to catch signals. The problem would be solved if the signal field were not cleared on receipt of the signal. But such a solution could result in a new problem: If signals keep arriving and are caught, the user stack could grow out of bounds because of the nested calls to the signal catcher. Alternatively, the kernel could reset the value of the signal-handling function to ignore signals of that type until the user again specifies what to do for such signals. Such a solution implies a loss of information, because the process has no way of knowing how many signals it receives. However, the loss of information is no more severe than it is for the case where the process receives many signals of one type before it has a chance to handle them. Finally, the BSD system allows a process to block and unblock receipt of signals with a new system call; when a process unblocks signals, the kernel sends pending signals that had been blocked to the process. When a process receives a signal, the kernel automatically blocks further receipt of the signal until the signal handler completes. This is analogous to how the kernel reacts to hardware interrupts: it blocks report of new interrupts while it handles previous interrupts.

A second anomaly in the treatment of signals concerns catching signals that occur while the process is in a system call, sleeping at an interruptible priority. The signal causes the process to take a *longjmp* out of its sleep, return to user mode, and call the signal handler. When the signal handler returns, the process appears to return from the system call with an error indicating that the system call was interrupted. The user can check for the error return and restart the system call, but it would sometimes be more convenient if the kernel automatically restarted the system call, as is done in the BSD system.

A third anomaly exists for the case where the process ignores a signal. If the signal arrives while the process is asleep at an interruptible sleep priority level, the process will wake up but will not do a *longjmp*. That is, the kernel realizes that the process ignores the signal only after waking it up and running it. A more consistent policy would be to leave the process asleep. However, the kernel stores the signal function address in the *u area*, and the *u area* may not be accessible when the signal is sent to the process. A solution to this problem would be to store the signal function address in the process table entry, where the kernel could check whether it should awaken the process on receipt of the signal. Alternatively, the process could immediately go back to sleep in the *sleep* algorithm, if it discovers that it should not have awakened. Nevertheless, user processes never realize that the process woke up, because the kernel encloses entry to the *sleep* algorithm in a "while" loop (recall from Chapter 2), putting the process back to sleep if the sleep event did not really occur.

Finally, the kernel does not treat "death of child" signals the same as other signals. In particular, when the process recognizes that it has received a "death of

child" signal, it turns off the notification of the signal in the process table entry signal field and in the default case, it acts as if no signal had been sent. The effect of a "death of child" signal is to wake up a process sleeping at interruptible priority. If the process catches "death of child" signals, it invokes the user handler as it does for other signals. The operations that the kernel does if the process ignores "death of child" signals will be discussed in Section 7.4. Finally, if a process invokes the *signal* system call with "death of child" parameter, the kernel sends the calling process a "death of child" signal if it has child processes in the zombie state. Section 7.4 discusses the rationale for calling *signal* with the "death of child" parameter.

7.2.2 Process Groups

Although processes on a UNIX system are identified by a unique ID number, the system must sometimes identify processes by "group." For instance, processes with a common ancestor process that is a login shell are generally related, and therefore all such processes receive signals when a user hits the "delete" or "break" key or when the terminal line hangs up. The kernel uses the *process group ID* to identify groups of related processes that should receive a common signal for certain events. It saves the group ID in the process table; processes in the same process group have identical group ID's.

The *setpgrp* system call initializes the process group number of a process and sets it equal to the value of its process ID. The syntax for the system call is

```
grp = setpgrp();
```

where *grp* is the new process group number. A child retains the process group number of its parent during *fork*. *Setpgrp* also has important ramifications for setting up the control terminal of a process (see Section 10.3.5).

7.2.3 Sending Signals from Processes

Processes use the *kill* system call to send signals. The syntax for the system call is

```
kill(pid, signum)
```

where *pid* identifies the set of processes to receive the signal, and *signum* is the signal number being sent. The following list shows the correspondence between values of *pid* and sets of processes.

- If *pid* is a positive integer, the kernel sends the signal to the process with process ID *pid*.
- If *pid* is 0, the kernel sends the signal to all processes in the sender's process group.
- If *pid* is −1, the kernel sends the signal to all processes whose real user ID equals the effective user ID of the sender (Section 7.6 will define real and

effective user ID's). If the sending process has effective user ID of superuser, the kernel sends the signal to all processes except processes 0 and 1.

- If *pid* is a negative integer but not −1, the kernel sends the signal to all processes in the process group equal to the absolute value of *pid*.

In all cases, if the sending process does not have effective user ID of superuser, or its real or effective user ID do not match the real or effective user ID of the receiving process, *kill* fails.

```
#include <signal.h>
main()
{
        register int i;

        setpgrp();
        for (i = 0;  i < 10;  i++)
        {
                if (fork() == 0)
                {
                        /* child proc */
                        if (i & 1)
                                setpgrp();
                        printf("pid = %d pgrp = %d\n", getpid(), getpgrp());
                        pause();        /* sys call to suspend execution */
                }
        }
        kill(0, SIGINT);
}
```

Figure 7.13. Sample Use of Setpgrp

In the program in Figure 7.13, the process resets its process group number and creates 10 child processes. When created, each child process has the same process group number as the parent process, but processes created during odd iterations of the loop reset their process group number. The system calls *getpid* and *getpgrp* return the process ID and the group ID of the executing process, and the *pause* system call suspends execution of the process until it receives a signal. Finally, the parent executes the *kill* system call and sends an interrupt signal to all processes in its process group. The kernel sends the signal to the 5 "even" processes that did not reset their process group, but the 5 "odd" processes continue to loop.

7.3 PROCESS TERMINATION

Processes on a UNIX system terminate by executing the *exit* system call. An *exit*ing process enters the zombie state (recall Figure 6.1), relinquishes its resources, and dismantles its context except for its slot in the process table. The syntax for the call is

```
exit(status);
```

where the value of *status* is returned to the parent process for its examination. Processes may call *exit* explicitly or implicitly at the end of a program: the startup routine linked with all C programs calls *exit* when the program returns from the *main* function, the entry point of all programs. Alternatively, the kernel may invoke *exit* internally for a process on receipt of uncaught signals as discussed above. If so, the value of *status* is the signal number.

The system imposes no time limit on the execution of a process, and processes frequently exist for a long time. For instance, processes 0 (the swapper) and 1 (*init*) exist throughout the lifetime of a system. Other examples are *getty* processes, which monitor a terminal line, waiting for a user to log in, and special-purpose administrative processes.

```
algorithm exit
input:  return code for parent process
output: none
{
        ignore all signals;
        if (process group leader with associated control terminal)
        {
                send hangup signal to all members of process group;
                reset process group for all members to 0;
        }
        close all open files (internal version of algorithm close);
        release current directory (algorithm iput);
        release current (changed) root, if exists (algorithm iput);
        free regions, memory associated with process (algorithm freereg);
        write accounting record;
        make process state zombie
        assign parent process ID of all child processes to be init process (1);
                if any children were zombie, send death of child signal to init;
        send death of child signal to parent process;
        context switch;
}
```

Figure 7.14. Algorithm for Exit

Figure 7.14 shows the algorithm for *exit*. The kernel first disables signal handling for the process, because it no longer makes any sense to handle signals. If the *exit*ing process is a *process group leader* associated with a control terminal (see Section 10.3.5), the kernel assumes the user is not doing any useful work and sends a "hangup" signal to all processes in the process group. Thus, if a user types "end of file" (control-d character) in the login shell while some processes associated with the terminal are still alive, the *exit*ing process will send them a hangup signal. The kernel also resets the process group number to 0 for processes in the process group, because it is possible that another process will later get the process ID of the process that just *exit*ed and that it too will be a process group leader. Processes that belonged to the old process group will not belong to the later process group. The kernel then goes through the open file descriptors, *clos*ing each one internally with algorithm *close*, and releases the inodes it had accessed for the current directory and changed root (if it exists) via algorithm *iput*.

The kernel now releases all user memory by freeing the appropriate regions with algorithm *detachreg* and changes the process state to zombie. It saves the *exit* status code and the accumulated user and kernel execution time of the process and its descendants in the process table. The description of *wait* in Section 7.4 shows how a process gets the timing data for descendant processes. The kernel also *write*s an accounting record to a global accounting file, containing various run-time statistics such as user ID, CPU and memory usage, and amount of I/O for the process. User-level programs can later read the accounting file to gather various statistics, useful for performance monitoring and customer billing. Finally, the kernel disconnects the process from the process tree by making process 1 (*init*) adopt all its child processes. That is, process 1 becomes the legal parent of all live children that the *exit*ing process had created. If any of the children are zombie, the exiting process sends *init* a "death of child" signal so that *init* can remove them from the process table (see Section 7.9); the *exit*ing process sends its parent a "death of child" signal, too. In the typical scenario, the parent process executes a *wait* system call to synchronize with the *exit*ing child. The now-zombie process does a context switch so that the kernel can schedule another process to execute; the kernel never schedules a zombie process to execute.

In the program in Figure 7.15, a process creates a child process, which prints its PID and executes the *pause* system call, suspending itself until it receives a signal. The parent prints the child's PID and *exit*s, returning the child's PID as its status code. If the *exit* call were not present, the startup routine calls *exit* when the process returns from *main*. The child process spawned by the parent lives on until it receives a signal, even though the parent process is gone.

7.4 AWAITING PROCESS TERMINATION

A process can synchronize its execution with the termination of a child process by executing the *wait* system call. The syntax for the system call is

```
main()
{
        int child;

        if ((child = fork()) == 0)
        {
                printf("child PID %d\n", getpid());
                pause();        /* suspend execution until signal */
        }
        /* parent */
        printf("child PID %d\n", child);
        exit(child);
}
```

Figure 7.15. Example of Exit

pid = wait(stat_addr);

where *pid* is the process ID of the zombie child, and *stat addr* is the address in user space of an integer that will contain the *exit* status code of the child.

Figure 7.16 shows the algorithm for *wait*. The kernel searches for a zombie child of the process and, if there are no children, returns an error. If it finds a zombie child, it extracts the PID number and the parameter supplied to the child's *exit* call and returns those values from the system call. An *exit*ing process can thus specify various return codes to give the reason it *exit*ed, but many programs do not consistently set it in practice. The kernel adds the accumulated time the child process executed in user and in kernel mode to the appropriate fields in the parent process *u area* and, finally, releases the process table slot formerly occupied by the zombie process. The slot is now available for a new process.

If the process executing *wait* has child processes but none are zombie, it sleeps at an interruptible priority until the arrival of a signal. The kernel does not contain an explicit wake up call for a process sleeping in *wait*: such processes only wake up on receipt of signals. For any signal except "death of child," the process will react as described above. However, if the signal is "death of child," the process may respond differently.

- In the default case, it will wake up from its sleep in *wait*, and *sleep* invokes algorithm *issig* to check for signals. *Issig* (Figure 7.7) recognizes the special case of "death of child" signals and returns "false." Consequently, the kernel does not "long jump" from *sleep*, but returns to *wait*. The kernel will restart the *wait* loop, find a zombie child — at least one is guaranteed to exist, release the child's process table slot, and return from the *wait* system call.
- If the process catches "death of child" signals, the kernel arranges to call the user signal-handler routine, as it does for other signals.

```
algorithm wait
input:  address of variable to store status of exiting process
output: child ID, child exit code
{
        if (waiting process has no child processes)
                return(error);

        for (;;)        /* loop until return from inside loop */
        {
                if (waiting process has zombie child)
                {
                        pick arbitrary zombie child;
                        add child CPU usage to parent;
                        free child process table entry;
                        return(child ID, child exit code);
                }
                if (process has no children)
                        return error;
                sleep at interruptible priority (event child process exits);
        }
}
```

Figure 7.16. Algorithm for Wait

- If the process ignores "death of child" signals, the kernel restarts the *wait* loop, frees the process table slots of zombie children, and searches for more children.

For example, a user gets different results when invoking the program in Figure 7.17 with or without a parameter. Consider first the case where a user invokes the program without a parameter (*argc* is 1, the program name). The (parent) process creates 15 child processes that eventually *exit* with return code *i*, the value of the loop variable when the child was created. The kernel, executing *wait* for the parent, finds a zombie child process and returns its process ID and *exit* code. It is indeterminate which child process it finds. The C library code for the *exit* system call stores the *exit* code in bits 8 to 15 of *ret_code* and returns the child process ID for the *wait* call. Thus *ret_code* equals 256**i*, depending on the value of *i* for the child process, and *ret_val* equals the value of the child process ID.

If a user invokes the above program with a parameter (*argc* > 1), the (parent) process calls *signal* to ignore "death of child" signals. Assume the parent process sleeps in *wait* before any child processes *exit*: When a child process *exit*s, it sends a "death of child" signal to the parent process; the parent process wakes up because its sleep in *wait* is at an interruptible priority. When the parent process eventually runs, it finds that the outstanding signal was for "death of child"; but because it ignores "death of child" signals, the kernel removes the entry of the zombie child from the process table and continues executing *wait* as if no signal had happened.

```
#include <signal.h>
main(argc, argv)
        int argc;
        char *argv[];
{
        int i, ret_val, ret_code;

        if (argc >= 1)
                signal(SIGCLD, SIG_IGN);        /* ignore death of children */
        for (i = 0;  i < 15;  i++)
                if (fork() == 0)
                {
                        /* child proc here */
                        printf("child proc %x\n", getpid());
                        exit(i);
                }
        ret_val = wait(&ret_code);
        printf("wait ret_val %x ret_code %x\n", ret_val, ret_code);
}
```

Figure 7.17. Example of Wait and Ignoring Death of Child Signal

The kernel does the above procedure each time the parent receives a "death of child" signal, until it finally goes through the *wait* loop and finds that the parent has no children. The *wait* system call then returns a −1. The difference between the two invocations of the program is that the parent process *wait*s for the termination of *any* child process in the first case but *wait*s for the termination of *all* child processes in the second case.

Older versions of the UNIX system implemented the *exit* and *wait* system calls without the "death of child" signal. Instead of sending a "death of child" signal, *exit* would wake up the parent process. If the parent process was sleeping in the *wait* system call, it would wake up, find a zombie child, and return. If it was not sleeping in the *wait* system call, the wake up would have no effect; it would find a zombie child on its next *wait* call. Similarly, the *init* process would sleep in *wait*, and *exit*ing processes would wake it up if it were to adopt new zombie processes.

The problem with that implementation is that it is impossible to clean up zombie processes unless the parent executes *wait*. If a process creates many children but never executes *wait*, the process table will become cluttered with zombie children when the children *exit*. For example, consider the dispatcher program in Figure 7.18. The process *read*s its standard input file until it encounters the end of file, creating a child process for each *read*. However, the parent process does not *wait* for the termination of the child process, because it wants to dispatch processes as fast as possible and the child process may take too long until it *exit*s. If the parent makes the *signal* call to ignore "death of child"

```
#include <signal.h>
main(argc, argv)
{
        char buf[256];

        if (argc != 1)
                signal(SIGCLD, SIG_IGN);        /* ignore death of children */
        while (read(0, buf, 256))
                if (fork() == 0)
                {
                        /* child proc here typically does something with buf */
                        exit(0);
                }
}
```

Figure 7.18. Example Depicting the Reason for Death of Child Signal

signals, the kernel will release the entries for the zombie processes automatically. Otherwise, zombie processes would eventually fill the maximum allowed slots of the process table.

7.5 INVOKING OTHER PROGRAMS

The *exec* system call invokes another program, overlaying the memory space of a process with a copy of an executable file. The contents of the user-level context that existed before the *exec* call are no longer accessible afterward except for *exec*'s parameters, which the kernel copies from the old address space to the new address space. The syntax for the system call is

```
execve(filename, argv, envp)
```

where *filename* is the name of the executable file being invoked, *argv* is a pointer to an array of character pointers that are parameters to the executable program, and *envp* is a pointer to an array of character pointers that are the *environment* of the executed program. There are several library functions that call the *exec* system call such as *execl*, *execv*, *execle*, and so on. All call *execve* eventually, hence it is used here to specify the *exec* system call. When a program uses command line parameters, as in

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

the array *argv* is a copy of the *argv* parameter to *exec*. The character strings in the environment are of the form "name=value" and may contain useful information for programs, such as the user's home directory and a path of directories to search for executable programs. Processes can access their environment via the global

```
algorithm exec
input:  (1) file name
        (2) parameter list
        (3) environment variables list
output: none
{
        get file inode (algorithm namei);
        verify file executable, user has permission to execute;
        read file headers, check that it is a load module;
        copy exec parameters from old address space to system space;
        for (every region attached to process)
                detach all old regions (algorithm detach);
        for (every region specified in load module)
        {
                allocate new regions (algorithm allocreg);
                attach the regions (algorithm attachreg);
                load region into memory if appropriate (algorithm loadreg);
        }
        copy exec parameters into new user stack region;
        special processing for setuid programs, tracing;
        initialize user register save area for return to user mode;
        release inode of file (algorithm iput);
}
```

Figure 7.19. Algorithm for Exec

variable *environ*, initialized by the C startup routine.

Figure 7.19 shows the algorithm for the *exec* system call. *Exec* first accesses the file via algorithm *namei* to determine if it is an executable, regular (nondirectory) file and to determine if the user has permission to execute the program. The kernel then reads the file header to determine the layout of the executable file.

Figure 7.20 shows the logical format of an executable file as it exists in the file system, typically generated by the assembler or loader. It consists of four parts:

1. The primary header describes how many sections are in the file, the start address for process execution, and the *magic number*, which gives the type of the executable file.
2. Section headers describe each section in the file, giving the section size, the virtual addresses the section should occupy when running in the system, and other information.
3. The sections contain the "data," such as text, that are initially loaded in the process address space.
4. Miscellaneous sections may contain symbol tables and other data, useful for debugging.

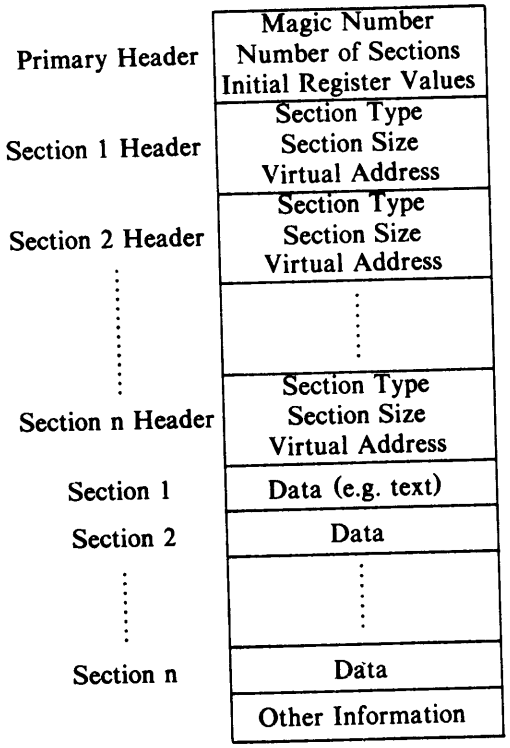

Figure 7.20. Image of an Executable File

Specific formats have evolved through the years, but all executable files have contained a primary header with a magic number.

The magic number is a short integer, which identifies the file as a load module and enables the kernel to distinguish run-time characteristics about it. For example, use of particular magic numbers on a PDP 11/70 informed the kernel that processes could use up to 128K bytes of memory instead of 64K bytes,[2] but the magic number still plays an important role in paging systems, as will be seen in Chapter 9.

2. The values of the magic numbers were the values of PDP 11 jump instructions; original versions of the system *executed* the instructions, and the program counter jumped to various locations depending on the size of the header and on the type of executable file being executed! This feature was no longer in use by the time the system was written in C.

At this point, the kernel has accessed the inode for the executable file and has verified that it can execute it. It is about to free the memory resources that currently form the user-level context of the process. But since the parameters to the new program are contained in the memory space about to be freed, the kernel first copies the arguments from the old memory space to a temporary buffer until it attaches the regions for the new memory space.

Because the parameters to *exec* are user addresses of arrays of character strings, the kernel copies the address of the character string and then the character string to kernel space for each character string. It may choose several places to store the character strings, dependent on the implementation. The more popular places are the kernel stack (a local array in a kernel routine), unallocated areas (such as pages) of memory that can be borrowed temporarily, or secondary memory such as a swapping device.

The simplest implementation for copying parameters to the new user-level context is to use the kernel stack. But because system configurations usually impose a limit on the size of the kernel stack and because the *exec* parameters can have arbitrary length, the scheme must be combined with another. Of the other choices, implementations use the fastest method. If it is easy to allocate pages of memory, such a method is preferable since access to primary memory is faster than access to secondary memory (such as a swapping device).

After copying the *exec* parameters to a holding place in the kernel, the kernel detaches the old regions of the process using algorithm *detachreg*. Special treatment for text regions will be discussed later in this section. At this point the process has no user-level context, so any errors that it incurs from now on result in its termination, caused by a signal. Such errors include running out of space in the kernel region table, attempting to load a program whose size exceeds the system limit, attempting to load a program whose region addresses overlap, and others. The kernel allocates and attaches regions for text and data, loading the contents of the executable file into main memory (algorithms *allocreg*, *attachreg*, and *loadreg*). The data region of a process is (initially) divided into two parts: data initialized at compile time and data not initialized at compile time ("bss"). The initial allocation and attachment of the data region is for the initialized data. The kernel then increases the size of the data region using algorithm *growreg* for the "bss" data, and initializes the value of the memory to 0. Finally, it allocates a region for the process stack, attaches it to the process, and allocates memory to store the *exec* parameters. If the kernel has saved the *exec* parameters in memory pages, it can use those pages for the stack. Otherwise, it copies the *exec* parameters to the user stack.

The kernel clears the addresses of user signal catchers from the *u area*, because those addresses are meaningless in the new user-level context. Signals that are ignored remain ignored in the new context. Then the kernel sets the saved register context for user mode, specifically setting the initial user stack pointer and program counter: The loader had written the initial program counter in the file header. The kernel takes special action for *setuid* programs and for process tracing, covered in

the next section and in Chapter 11, respectively. Finally, it invokes algorithm *iput*, releasing the inode that was originally allocated in the *namei* algorithm at the beginning of *exec*. The use of *namei* and *iput* in *exec* corresponds to their use in *open*ing and *clos*ing a file; the state of a file during the *exec* call resembles that of an open file except for the absence of a file table entry. When the process "returns" from the *exec* system call, it executes the code of the new program. However, it is the same process it was before the *exec*; its process ID number does not change, nor does its position in the process hierarchy. Only the user-level context changes.

```
main()
{
        int status;
        if (fork() == 0)
                execl("/bin/date", "date", 0);
        wait(&status);
}
```

Figure 7.21. Use of Exec

For example, the program in Figure 7.21 creates a child process that invokes the *exec* system call. Immediately after the parent and child processes return from *fork*, they execute independent copies of the program. When the child process is about to invoke the *exec* call, its text region consists of the instructions for the program, its data region consists of the strings "/bin/date" and "date", and its stack contains the stack frames the process pushed to get to the *exec* call. The kernel finds the file "/bin/date" in the file system, finds that all users can execute it, and determines that it is an executable load module. By convention, the first parameter of the argument list *argv* to *exec* is the (last component of the) path name of the executable file. The process thus has access to the program name at user-level, sometimes a useful feature.[3] The kernel then copies the strings "/bin/date" and "date" to an internal holding area and frees the text, data, and stack regions occupied by the process. It allocates new text, data, and stack regions for the process, copies the instruction section of the file "/bin/date" into the text region, and copies the data section of the file into the data region. The kernel reconstructs the original parameter list (here, the character string "date") and puts it in the stack region. After the *exec* call, the child process no longer executes the

3. On System V for instance, the standard programs for renaming a file (*mv*), copying a file (*cp*), and linking a file (*ln*) are one executable file because they execute similar code. The process looks at the name the user used to invoke it to determine what it should do.

old program but executes the program "date": When the "date" program completes, the parent process receives its exit status from the *wait* call.

Until now, we have assumed that process text and data occupy separate sections of an executable program and, hence, separate regions of a running process. There are two advantages for keeping text and data separate: protection and sharing. If text and data were in the same region, the system could not prevent a process from overwriting its instructions, because it would not know which addresses contain instructions and which contain data. But if text and data are in separate regions, the kernel can set up hardware protection mechanisms to prevent processes from overwriting their text space. If a process mistakenly attempts to overwrite its text space, it incurs a protection fault that typically results in termination of the process.

```
#include <signal.h>
main()
{
        int i, *ip;
        extern f(), sigcatch();

        ip = (int *)f;        /* assign ip to address of function f */
        for (i = 0;  i < 20;  i++)
                signal(i, sigcatch);
        *ip = 1;              /* attempt to overwrite address of f */
        printf("after assign to ip\n");
        f();
}

f()
{
}

sigcatch(n)
        int n;
{
        printf("caught sig %d\n", n);
        exit(1);
}
```

Figure 7.22. Example of Program Overwriting its Text

For example, the program in Figure 7.22 assigns the pointer *ip* to the address of the function *f* and then arranges to catch all signals. If the program is compiled so that text and data are in separate regions, the process executing the program incurs a protection fault when it attempts to write the contents of *ip*, because it is writing its write-protected text region. The kernel sends a *SIGBUS* signal to the process on

an AT&T 3B20 computer, although other implementations may send other signals. The process catches the signal and *exit*s without executing the print statement in *main*. However, if the program were compiled so that the program text and data were part of one region (the data region), the kernel would not realize that a process was overwriting the address of the function *f*. The address of *f* contains the value 1! The process executes the print statement in *main* but executes an illegal instruction when it calls *f*. The kernel sends it a *SIGILL* signal, and the process *exit*s.

Having instructions and data in separate regions makes it easier to protect against addressing errors. Early versions of the UNIX system allowed text and data to be in the same region, however, because of process size limitations imposed by PDP machines: Programs were smaller and required fewer "segmentation" registers if text and data occupied the same region. Current versions of the system do not have such stringent size limitations on processes, and future compilers will not support the option to load text and data in one region.

The second advantage of having separate regions for text and data is to allow sharing of regions. If a process cannot write its text region, its text does not change from the time the kernel loads it from the executable file. If several processes execute a file they can, therefore, share one text region, saving memory. Thus, when the kernel allocates a text region for a process in *exec*, it checks if the executable file allows its text to be shared, indicated by its magic number. If so, it follows algorithm *xalloc* to find an existing region for the file text or to assign a new one (see Figure 7.23).

In *xalloc*, the kernel searches the active region list for the file's text region, identifying it as the one whose inode pointer matches the inode of the executable file. If no such region exists, the kernel allocates a new region (algorithm *allocreg*), attaches it to the process (algorithm *attachreg*), loads it into memory (algorithm *loadreg*), and changes its protection to read-only. The latter step causes a memory protection fault if a process attempts to write the text region. If, in searching the active region list, the kernel locates a region that contains the file text, it makes sure that the region is loaded into memory (it sleeps otherwise) and attaches it to the process. The kernel unlocks the region at the conclusion of *xalloc* and decrements the region count later, when it executes *detachreg* during *exit* or *exec*. Traditional implementations of the system contain a *text table* that the kernel manipulates in the way just described for text regions. The set of text regions can thus be viewed as a modern version of the old text table.

Recall that when allocating a region for the first time in *allocreg* (Section 6.5.2), the kernel increments the reference count of the inode associated with the region, after it had incremented the reference count in *namei* (invoking *iget*) at the beginning of *exec*. Because the kernel decrements the reference count once in *iput* at the end of *exec*, the inode reference count of a (shared text) file being executed is at least 1: Therefore, if a process *unlink*s the file, its contents remain intact. The kernel no longer needs the file after loading it into memory, but it needs the pointer to the in-core inode in the region table to identify the file that corresponds

```
algorithm xalloc                /* allocate and initialize text region */
input:  inode of executable file
output: none
{
        if (executable file does not have separate text region)
                return;
        if (text region associated with text of inode)
        {
                /* text region already exists...attach to it */
                lock region;
                while (contents of region not ready yet)
                {
                        /* manipulation of reference count prevents total
                         * removal of the region.
                         */
                        increment region reference count;
                        unlock region;
                        sleep (event contents of region ready);
                        lock region;
                        decrement region reference count;
                }
                attach region to process (algorithm attachreg);
                unlock region;
                return;
        }
        /* no such text region exists---create one */
        allocate text region (algorithm allocreg); /* region is locked */
        if (inode mode has sticky bit set)
                turn on region sticky flag;
        attach region to virtual address indicated by inode file header
                                (algorithm attachreg);
        if (file specially formatted for paging system)
                /* Chapter 9 discusses this case */
        else        /* not formatted for paging system */
                read file text into region (algorithm loadreg);
        change region protection in per process region table to read only;
        unlock region;
}
```

Figure 7.23. Algorithm for Allocation of Text Regions

to the region. If the reference count were to drop to 0, the kernel could reallocate the in-core inode to another file, compromising the meaning of the inode pointer in the region table: If a user were to *exec* the new file, the kernel would find the text region of the old file by mistake. The kernel avoids this problem by incrementing the inode reference count in *allocreg*, preventing reassignment of the in-core inode.

When the process detaches the text region during *exit* or *exec*, the kernel decrements the inode reference count an extra time in *freereg*, unless the inode has the sticky-bit mode set, as will be seen.

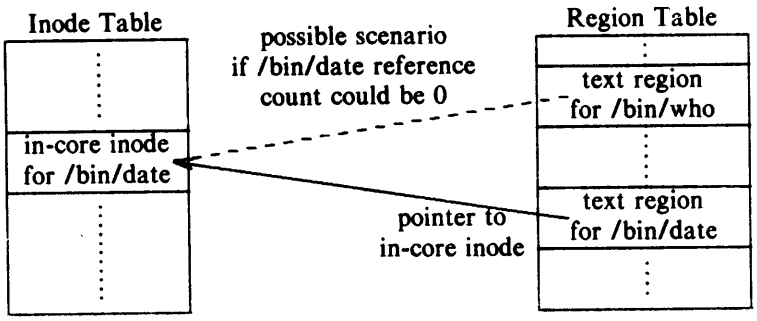

Figure 7.24. Relationship of Inode Table and Region Table for Shared Text

For example, reconsider the *exec* of "/bin/date" in Figure 7.21, and assume that the file has separate text and data sections. The first time a process executes "/bin/date", the kernel allocates a region table entry for the text (Figure 7.24) and leaves the inode reference count at 1 (after the *exec* completes). When "/bin/date" *exit*s, the kernel invokes *detachreg* and *freereg*, decrementing the inode reference count to 0. However, if the kernel had not incremented the inode reference count for "/bin/date" the first time it was *exec*ed, its reference count would be 0 and the inode would be on the free list while the process was running. Suppose another process *exec*s the file "/bin/who", and the kernel allocates the in-core inode previously used for "/bin/date" to "/bin/who". The kernel would search the region table for the inode for "/bin/who" but find the inode for "/bin/date" instead. Thinking that the region contains the text for "/bin/who", it would execute the wrong program. Consequently, the inode reference count for running, shared text files is at least 1, so that the kernel cannot reallocate the inode.

The capability to share text regions allows the kernel to decrease the startup time of an *exec*ed program by using the *sticky-bit*. System administrators can set the sticky-bit file mode with the *chmod* system call (and command) for frequently used executable files. When a process executes a file that has its sticky-bit set, the kernel does not release the memory allocated for text when it later detaches the region during *exit* or *exec*, even if the region reference count drops to 0. The kernel leaves the text region intact with *inode* reference count 1, even though it is no longer attached to any processes. When another process *exec*s the file, it finds the region table entry for the file text. The process startup time is small, because it does not have to read the text from the file system: If the text is still in memory, the kernel does not do any I/O for the text; if the kernel has swapped the text to a

swap device, it is faster to load the text from a swap device than from the file system, as will be seen in Chapter 9.

The kernel removes the entries for sticky-bit text regions in the following cases:

1. If a process *open*s the file for writing, the *write* operations will change the contents of the file, invalidating the contents of the region.
2. If a process changes the permission modes of the file (*chmod*) such that the sticky-bit is no longer set, the file should not remain in the region table.
3. If a process *unlink*s the file, no process will be able to *exec* it any more because the file has no entry in the file system; hence no new processes will access the file's region table entry. Because there is no need for the text region, the kernel can remove it to free some resources.
4. If a process unmounts the file system, the file is no longer accessible and no processes can *exec* it, so the logic of the previous case applies.
5. If the kernel runs out of space on the swap device, it attempts to free available space by freeing sticky-bit regions that are currently unused. Although other processes may need the text region soon, the kernel has more immediate needs.

The sticky text region must be removed in the first two cases because it no longer reflects the current state of the file. The kernel removes the sticky entries in the last three cases because it is pragmatic to do so. Of course, the kernel frees the region *only* if no processes currently use it (its reference count is 0); otherwise, the system calls *open*, *unlink*, and *umount* (cases 1, 3 and 4) fail.

The scenario for *exec* is slightly more complicated if a process *exec*s itself. If a user types

```
sh script
```

the shell *fork*s and the child process *exec*s the shell and executes the commands in the file "script". If a process *exec*s itself and allows sharing of its text region, the kernel must avoid deadlocks over the inode and region locks. That is, the kernel cannot lock the "old" text region, hold the lock, and then attempt to lock the "new" text region, because the old and new regions are one region. Instead, the kernel simply leaves the old text region attached to the process, since it will be reused anyway.

Processes usually invoke *exec* after *fork*; the child process thus copies the parent address space during the *fork*, discards it during the *exec*, and executes a different program image than the parent process. Would it not be more natural to combine the two system calls into one to invoke a program and run it as a new process? Ritchie surmises that *fork* and *exec* exist as separate system calls because, when designing the UNIX system, he and Thompson were able to add the *fork* system call without having to change much code in the existing kernel (see page 1584 of [Ritchie 84a]). But separation of the *fork* and *exec* system calls is functionally important too, because the processes can manipulate their standard input and standard output file descriptors independently to set up pipes more elegantly than if

the two system calls were combined. The example of the shell in Section 7.8 highlights this feature.

7.6 THE USER ID OF A PROCESS

The kernel associates two user IDs with a process, independent of the process ID the *real user ID* and the *effective user ID* or *setuid* (set user ID). The real user ID identifies the user who is responsible for the running process. The effective user ID is used to assign ownership of newly created files, to check file access permissions, and to check permission to send signals to processes via the *kill* system call. The kernel allows a process to change its effective user ID when it *execs* a *setuid* program or when it invokes the *setuid* system call explicitly.

A *setuid* program is an executable file that has the *setuid* bit set in its permission mode field. When a process *execs* a *setuid* program, the kernel sets the effective user ID fields in the process table and *u area* to the owner ID of the file. To distinguish the two fields, let us call the field in the process table the *saved* user ID. An example illustrates the difference between the two fields.

The syntax for the *setuid* system call is

```
setuid(uid)
```

where *uid* is the new user ID, and its result depends on the current value of the effective user ID. If the effective user ID of the calling process is superuser, the kernel resets the real and effective user ID fields in the process table and *u area* to *uid*. If the effective user ID of the calling process is not superuser, the kernel resets the effective user ID in the *u area* to *uid* if *uid* has the value of the real user ID or if it has the value of the saved user ID. Otherwise, the system call returns an error. Generally, a process inherits its real and effective user IDs from its parent during the *fork* system call and maintains their values across *exec* system calls.

The program in Figure 7.25 demonstrates the *setuid* system call. Suppose the executable file produced by compiling the program has owner "maury" (user ID 8319), its *setuid* bit is on, and all users have permission to execute it. Further, assume that users "mjb" (user ID 5088) and "maury" own the files of their respective names, and that both files have read-only permission for their owners. User "mjb" sees the following output when executing the program:

```
uid 5088 euid 8319
fdmjb −1 fdmaury 3
after setuid(5088): uid 5088 euid 5088
fdmjb 4 fdmaury −1
after setuid(8319): uid 5088 euid 8319
```

The system calls *getuid* and *geteuid* return the real and effective user IDs of the process, 5088 and 8319 respectively for user "mjb". Therefore, the process cannot *open* file "mjb", because its effective user ID (8319) does not have read permission

```
#include <fcntl.h>
main()
{
        int uid, euid, fdmjb, fdmaury;

        uid = getuid();                 /* get real UID */
        euid = geteuid();       /* get effective UID */
        printf("uid %d euid %d\n", uid, euid);

        fdmjb = open("mjb", O_RDONLY);
        fdmaury = open("maury", O_RDONLY);
        printf("fdmjb %d fdmaury %d\n", fdmjb, fdmaury);

        setuid(uid);
        printf("after setuid(%d): uid %d euid %d\n", uid, getuid(), geteuid());

        fdmjb = open("mjb", O_RDONLY);
        fdmaury = open("maury", O_RDONLY);
        printf("fdmjb %d fdmaury %d\n", fdmjb, fdmaury);

        setuid(euid);
        printf("after setuid(%d): uid %d euid %d\n", euid, getuid(), geteuid());
}
```

Figure 7.25. Example of Execution of Setuid Program

for the file, but the process can *open* file "maury". After calling *setuid* to reset the effective user ID of the process to the real user ID ("mjb"), the second print statement prints values 5088 and 5088, the user ID of "mjb". Now the process can *open* the file "mjb", because its effective user ID has read permission on the file, but the process cannot open file "maury". Finally, after calling *setuid* to reset the effective user ID to the saved *setuid* value of the program (8319), the third print statement prints values 5088 and 8319 again. The last case shows that a process can *exec* a *setuid* program and toggle its effective user ID between its real user ID and its *exec*ed *setuid*.

User "maury" sees the following output when executing the program:

```
uid 8319 euid 8319
fdmjb -1 fdmaury 3
after setuid(8319): uid 8319 euid 8319
fdmjb -1 fdmaury 4
after setuid(8319): uid 8319 euid 8319
```

The real and effective user IDs are always 8319: the process can never *open* file "mjb", but it can *open* file "maury". The effective user ID stored in the *u area* is

the result of the most recent *setuid* system call or the *exec* of a *setuid* program; it is solely responsible for determining file access permissions. The *saved* user ID in the process table allows a process to reset its effective user ID to it by executing the *setuid* system call, thus recalling its original, effective user ID.

The *login* program executed by users when logging into the system is a typical program that calls the *setuid* system call. *Login* is *setuid* to root (superuser) and therefore runs with *effective user ID* root. It queries the user for various information such as name and password and, when satisfied, invokes the *setuid* system call to set its real and effective user ID to that of the user trying to log in (found in fields in the file "/etc/passwd"). *Login* finally *exec*s the shell, which runs with its real and effective user IDs set for the appropriate user.

The *mkdir* command is a typical *setuid* program. Recall from Section 5.8 that only a process with effective user ID superuser can create a directory. To allow ordinary users the capability to create directories, the *mkdir* command is a *setuid* program owned by root (superuser permission). When executing *mkdir*, the process runs with superuser access rights, creates the directory for the user via *mknod*, and then changes the owner and access permissions of the directory to that of the real user.

7.7 CHANGING THE SIZE OF A PROCESS

A process may increase or decrease the size of its data region by using the *brk* system call. The syntax for the *brk* system call is

```
brk(endds);
```

where *endds* becomes the value of the highest virtual address of the data region of the process (called its *break* value). Alternatively, a user can call

```
oldendds = sbrk(increment);
```

where *increment* changes the current break value by the specified number of bytes, and *oldendds* is the break value before the call. *Sbrk* is a C library routine that calls *brk*. If the data space of the process increases as a result of the call, the newly allocated data space is virtually contiguous to the old data space; that is, the virtual address space of the process extends continuously into the newly allocated data space. The kernel checks that the new process size is less than the system maximum and that the new data region does not overlap previously assigned virtual address space (Figure 7.26). If all checks pass, the kernel invokes *growreg* to allocate auxiliary memory (e.g., page tables) for the data region and increments the process size field. On a swapping system, it also attempts to allocate memory for the new space and clear its contents to zero; if there is no room in memory, it swaps the process out to get the new space (explained in detail in Chapter 9). If the process is calling *brk* to free previously allocated space, the kernel releases the memory; if the process accesses virtual addresses in pages that it had released, it incurs a memory fault.

```
algorithm brk
input:   new break value
output: old break value
{
        lock process data region;
        if (region size increasing)
                if (new region size is illegal)
                {
                        unlock data region;
                        return(error);
                }
        change region size (algorithm growreg);
        zero out addresses in new data space;
        unlock process data region;
}
```

Figure 7.26. Algorithm for Brk

Figure 7.27 shows a program that uses *brk* and sample output when run on an AT&T 3B20 computer. After arranging to catch *segmentation violation* signals by calling *signal*, the process calls *sbrk* and prints out its initial *break* value. Then it loops, incrementing a character pointer and writing its contents, until it attempts to write an address beyond its data region, causing a *segmentation violation* signal. Catching the signal, *catcher* calls *sbrk* to allocate another 256 bytes in the data region; the process continues from where it was interrupted in the loop, writing into the newly acquired data space. When it loops beyond the data region again, the entire procedure repeats. An interesting phenomenon occurs on machines whose memory is allocated by pages, as on the 3B20. A page is the smallest unit of memory that is protected by the hardware and so the hardware cannot detect when a process writes addresses that are beyond its *break* value but still on a "semilegal" page. This is shown by the output in Figure 7.27: the first *sbrk* call returns 140924, meaning that there are 388 bytes left on the page, which contain 2K bytes on a 3B20. But the process will fault only when it addresses the next page, at address 141312. *Catcher* adds 256 to the *break* value, making it 141180, still below the address of the next page. Hence, the process immediately faults again, printing the same address, 141312. After the next *sbrk*, the kernel allocates a new page of memory, so the process can address another 2K bytes, to 143360, even though the *break* value is not that high. When it faults, it will call *sbrk* 8 times until it can continue. Thus, a process can sometimes cheat beyond its official *break* value, although it is poor programming style.

The kernel automatically extends the size of the user stack when it overflows, following an algorithm similar to that for *brk*. A process originally contains enough (user) stack space to hold the *exec* parameters, but it overflows its initial stack area as it pushes data onto the stack during execution. When it overflows its

```
#include <signal.h>
char *cp;
int callno;

main()
{
        char *sbrk();
        extern catcher();

        signal(SIGSEGV, catcher);
        cp = sbrk(0);
        printf("original brk value %u\n", cp);
        for (;;)
                *cp++ = 1;
}

catcher(signo)
        int signo;
{
        callno++;
        printf("caught sig %d %dth call at addr %u\n", signo, callno, cp);
        sbrk(256);
        signal(SIGSEGV, catcher);
}
```

```
original brk value 140924
caught sig 11 1th call at addr 141312
caught sig 11 2th call at addr 141312
caught sig 11 3th call at addr 143360
        . . . (same address printed out to 10th call)
caught sig 11 10th call at addr 143360
caught sig 11 11th call at addr 145408
        . . . (same address printed out to 18th call)
caught sig 11 18th call at addr 145408
caught sig 11 19th call at addr 145408
```

Figure 7.27. Use of Brk and Sample Output

stack, the machine incurs a memory fault, because the process is attempting to access a location outside its address space. The kernel determines that the reason for the memory fault was because of stack overflow by comparing the value of the (faulted) stack pointer to the size of the stack region. The kernel allocates new space for the stack region exactly as it allocates space for *brk*, above. When it

```
/* read command line until "end of file" */
while (read(stdin, buffer, numchars))
{
        /* parse command line */
        if (/* command line contains & */)
                amper = 1;
        else
                amper = 0;
        /* for commands not part of the shell command language */
        if (fork() == 0)
        {
                /* redirection of IO? */
                if (/* redirect output */)
                {
                        fd = creat(newfile, fmask);
                        close(stdout);
                        dup(fd);
                        close(fd);
                        /* stdout is now redirected */
                }
                if (/* piping */ )
                {
                        pipe(fildes);
```

Figure 7.28. Main Loop of the Shell

returns from the interrupt, the process has the necessary stack space to continue.

7.8 THE SHELL

This chapter has covered enough material to explain how the *shell* works. The shell is more complex than described here, but the process relationships are illustrative of the real program. Figure 7.28 shows the main loop of the shell and demonstrates asynchronous execution, redirection of output, and pipes.

The shell *reads* a command line from its standard input and interprets it according to a fixed set of rules. The standard input and standard output file descriptors for the login shell are usually the terminal on which the user logged in, as will be seen in Chapter 10. If the shell recognizes the input string as a built-in command (for example, commands *cd*, *for*, *while* and others), it executes the command internally without creating new processes; otherwise, it assumes the command is the name of an executable file.

```
                if (fork() == 0)
                {
                        /* first component of command line */
                        close(stdout);
                        dup(fildes[1]);
                        close(fildes[1]);
                        close(fildes[0]);
                        /* stdout now goes to pipe */
                        /* child process does command */
                        execlp(command1, command1, 0);
                }
                /* 2nd command component of command line */
                close(stdin);
                dup(fildes[0]);
                close(fildes[0]);
                close(fildes[1]);
                /* standard input now comes from pipe */
        }
        execve(command2, command2, 0);
    }
    /* parent continues over here...
     * waits for child to exit if required
     */
    if (amper == 0)
        retid = wait(&status);
}
```

Figure 7.28. Main Loop of the Shell (continued)

The simplest command lines contain a program name and some parameters, such as

```
who
grep -n include *.c
ls -l
```

The shell *fork*s and creates a child process, which *exec*s the program that the user specified on the command line. The parent process, the shell that the user is using, *wait*s until the child process *exit*s from the command and then loops back to *read* the next command.

To run a process asynchronously (in the background), as in

```
nroff -mm bigdocument &
```

the shell sets an internal variable *amper* when it parses the ampersand character. If it finds the variable set at the end of the loop, it does not execute *wait* but immediately restarts the loop and *read*s the next command line.

The figure shows that the child process has access to a copy of the shell command line after the *fork*. To redirect standard output to a file, as in

```
nroff -mm bigdocument > output
```

the child *creat*s the output file specified on the command line; if the *creat* fails (for creating a file in a directory with wrong permissions, for example), the child would *exit* immediately. But if the *creat* succeeds, the child *close*s its previous standard output file and *dup*s the file descriptor of the new output file. The standard output file descriptor now refers to the redirected output file. The child process *close*s the file descriptor obtained from *creat* to conserve file descriptors for the *exec*ed program. The shell redirects standard input and standard error files in a similar way.

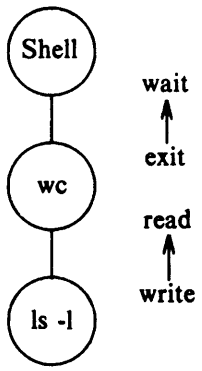

Figure 7.29. Relationship of Processes for ls −l | wc

The code shows how the shell could handle a command line with a single *pipe*, as in

```
ls -l | wc
```

After the parent process *fork*s and creates a child process, the child creates a *pipe*. The child process then *fork*s; it and its child each handle one component of the command line. The grandchild process created by the second *fork* executes the first command component (*ls*): It writes to the *pipe*, so it closes its standard output file descriptor, *dup*s the pipe write descriptor, and *close*s the original pipe write descriptor since it is unnecessary. The parent (*wc*) of the last child process (*ls*) is the child of the original shell process (see Figure 7.29). This process (*wc*) *close*s its standard input file and *dup*s the pipe read descriptor, causing it to become the standard input file descriptor. It then *close*s the original pipe read descriptor since it no longer needs it, and *exec*s the second command component of the original command line. The two processes that execute the command line execute

asynchronously, and the output of one process goes to the input of the other process. The parent shell meanwhile *wait*s for its child process (*wc*) to *exit*, then proceeds as usual: The entire command line completes when *wc exit*s. The shell loops and *read*s the next command.

7.9 SYSTEM BOOT AND THE INIT PROCESS

To initialize a system from an inactive state, an administrator goes through a "bootstrap" sequence: The administrator "boots" the system. Boot procedures vary according to machine type, but the goal is common to all machines: to get a copy of the operating system into machine memory and to start executing it. This is usually done in a series of stages; hence the name bootstrap. The administrator may set switches on the computer console to specify the address of a special hard-coded bootstrap program or just push a single button that instructs the machine to load a bootstrap program from its microcode. This program may consist of only a few instructions that instruct the machine to execute another program. On UNIX systems, the bootstrap procedure eventually reads the boot block (block 0) of a disk, and loads it into memory. The program contained in the boot block loads the kernel from the file system (from the file "/unix", for example, or another name specified by an administrator). After the kernel is loaded in memory, the boot program transfers control to the start address of the kernel, and the kernel starts running (algorithm *start*, Figure 7.30).

The kernel initializes its internal data structures. For instance, it constructs the linked lists of free buffers and inodes, constructs hash queues for buffers and inodes, initializes region structures, page table entries, and so on. After completing the initialization phase, it *mount*s the root file system onto root ("/") and fashions the environment for process 0, creating a *u area*, initializing slot 0 in the process table and making root the current directory of process 0, among other things.

When the environment of process 0 is set up, the system is running as process 0. Process 0 *fork*s, invoking the *fork* algorithm directly from the kernel, because it is executing in kernel mode. The new process, process 1, running in kernel mode, creates its user-level context by allocating a data region and attaching it to its address space. It grows the region to its proper size and copies code (described shortly) from the kernel address space to the new region: This code now forms the user-level context of process 1. Process 1 then sets up the saved user register context, "returns" from kernel to user mode, and executes the code it had just copied from the kernel. Process 1 is a user-level process as opposed to process 0, which is a kernel-level process that executes in kernel mode. The text for process 1, copied from the kernel, consists of a call to the *exec* system call to execute the program "/etc/init". Process 1 calls *exec* and executes the program in the normal fashion. Process 1 is commonly called *init* because it is responsible for initialization of new processes.

Why does the kernel copy the code for the *exec* system call to the user address space of process 1? It could invoke an internal version of *exec* directly from the

```
algorithm start              /* system startup procedure */
input:  none
output: none
{
        initialize all kernel data structures;
        pseudo-mount of root;
        hand-craft environment of process 0;
        fork process 1:
        {
                /* process 1 in here */
                allocate region;
                attach region to init address space;
                grow region to accommodate code about to copy in;
                copy code from kernel space to init user space to exec init;
                change mode: return from kernel to user mode;
                /* init never gets here---as result of above change mode,
                 * init exec's /etc/init and becomes a "normal" user process
                 * with respect to invocation of system calls
                 */
        }
        /* proc 0 continues here */
        fork kernel processes;
        /* process 0 invokes the swapper to manage the allocation of
         * process address space to main memory and the swap devices.
         * This is an infinite loop;  process 0 usually sleeps in the
         * loop unless there is work for it to do.
         */
        execute code for swapper algorithm;
}
```

Figure 7.30. Algorithm for Booting the System

kernel, but that would be more complicated than the implementation just described. To follow the latter procedure, *exec* would have to parse file names in kernel space, not just in user space, as in the current implementation. Such generality, needed only for *init*, would complicate the *exec* code and slow its performance in more common cases.

The *init* process (Figure 7.31) is a process dispatcher, spawning processes that allow users to log in to the system, among others. *Init* reads the file "/etc/inittab" for instructions about which processes to spawn. The file "/etc/inittab" contains lines that contain an "id," a state identifier (single user, multi-user, etc.), an "action" (see exercise 7.43), and a program specification (see Figure 7.32). *Init* reads the file and, if the *state* in which it was invoked matches the state identifier of a line, creates a process that executes the given program specification. For example, when invoking *init* for the multi-user state (state 2), *init* typically spawns

```
algorithm init      /* init process, process 1 of the system */
input:  none
output: none
{
      fd = open("/etc/inittab", O_RDONLY);
      while (line_read(fd, buffer))
      {
            /* read every line of file */
            if (invoked state != buffer state)
                  continue;      /* loop back to while */
            /* state matched */
            if (fork() == 0)
            {
                  execl("process specified in buffer");
                  exit();
            }
            /* init process does not wait */
            /* loop back to while */
      }

      while ((id = wait((int *) 0)) != -1)
      {
            /* check here if a spawned child died;
             * consider respawning it */
            /* otherwise, just continue */
      }
}
```

Figure 7.31. Algorithm for Init

```
Format: identifier, state, action, process specification
Fields separated by colons.
Comment at end of line preceded by '#'

co::respawn:/etc/getty console console          # Console in machine room
46:2:respawn:/etc/getty -t 60 tty46 4800H       # comments here
```

Figure 7.32. Sample Inittab File

getty processes to monitor the terminal lines configured on a system. When a user successfully logs in, *getty* goes through a *login* procedure and *exec*s a login shell, described in Chapter 10. Meanwhile, *init* executes the *wait* system call, monitoring the death of its child processes and the death of processes "orphaned" by *exit*ing parents.

Processes in the UNIX system are either user processes, daemon processes, or kernel processes. Most processes on typical systems are user processes, associated with users at a terminal. *Daemon* processes are not associated with any users but do system-wide functions, such as administration and control of networks, execution of time-dependent activities, line printer spooling, and so on. *Init* may spawn daemon processes that exist throughout the lifetime of the system or, on occasion, users may spawn them. They are like user processes in that they run at user mode and make system calls to access system services.

Kernel processes execute only in kernel mode. Process 0 spawns kernel processes, such as the page-reclaiming process *vhand*, and then becomes the *swapper* process. Kernel processes are similar to daemon processes in that they provide system-wide services, but they have greater control over their execution priorities since their code is part of the kernel. They can access kernel algorithms and data structures directly without the use of system calls, so they are extremely powerful. However, they are not as flexible as daemon processes, because the kernel must be recompiled to change them.

7.10 SUMMARY

This chapter has discussed the system calls that manipulate the process context and control its execution. The *fork* system call creates a new process by duplicating all the regions attached to the parent process. The tricky part of the *fork* implementation is to initialize the saved register context of the child process, so that it starts executing inside the *fork* system call and recognizes that it is the child process. All processes terminate in a call to the *exit* system call, which detaches the regions of a process and sends a "death of child" signal to its parent. A parent process can synchronize execution with the termination of a child process with the *wait* system call. The *exec* system call allows a process to invoke other programs, overlaying its address space with the contents of an executable file. The kernel detaches the old process regions and allocates new regions, corresponding to the executable file. Shared-text files and use of the sticky-bit mode improve memory utilization and the startup time of *exec*ed programs. The system allows ordinary users to execute with the privileges of other users, possibly superuser, with *setuid* programs and use of the *setuid* system call. The *brk* system call allows a process to change the size of its data region. Processes control their reaction to signals with the *signal* system call. When they catch a signal, the kernel changes the user stack and the user saved register context to set up the call to the signal handler. Processes can send signals with the *kill* system call, and they can control receipt of signals designated for particular process groups through the *setpgrp* system call.

The shell and *init* use standard system calls to provide sophisticated functions normally found in the kernel of other systems. The shell uses the system calls to interpret user commands, redirecting standard input, standard output and standard error, spawning processes, setting up pipes between spawned processes, synchronizing execution with child processes, and recording the exit status of commands. Similarly, *init* spawns various processes, particularly to control terminal execution. When such a process *exits*, *init* can respawn a new process for the same function, if so specified in the file "/etc/inittab".

7.11 EXERCISES

1. Run the program in Figure 7.33 at the terminal. Redirect its standard output to a file and compare the results.

```
main()
{
        printf("hello\n");
        if (fork() == 0)
                printf("world\n");
}
```

Figure 7.33. Fork and the Standard I/O Package

2. Describe what happens in the program in Figure 7.34 and compare to the results of Figure 7.4.
3. Reconsider the program in Figure 7.5, where two processes exchange messages through a pair of pipes. What happens if they try to exchange messages through one pipe?
4. In general, could there be any loss of information if a process receives several instances of a signal before it has a chance to react? (Consider a process that counts the number of interrupt signals it receives.) Should this problem be fixed?
5. Describe an implementation of the *kill* system call.
6. The program in Figure 7.35 catches "death of child" signals, and like many signal-catcher functions, resets the signal catcher. What happens in the program?
7. When a process receives certain signals and does not handle them, the kernel dumps an image of the process as it existed when it received the signal. The kernel creates a file called "core" in the current directory of the process and copies the *u area*, text, data, and stack regions into the file. A user can subsequently investigate the dumped image of the process with standard debugging tools. Describe an algorithm the kernel could follow to create a core file. What should the algorithm do if a file "core" already exists in the current directory? What should the kernel do if multiple processes dump "core" files in one directory?
8. Reconsider the program in Figure 7.12 where a process bombards another process with signals that the second process catches. Discuss what would happen if the signal-handling algorithm were changed in either of the following two ways:

```
#include  <fcntl.h>
int fdrd, fdwt;
char c;

main(argc, argv)
        int argc;
        char *argv[];
{
        if (argc != 3)
                exit(1);
        fork();


        if ((fdrd = open(argv[1], O_RDONLY)) == -1)
                exit(1);
        if (((fdwt = creat(argv[2], 0666)) == -1) &&
                        ((fdwt = open(argv[2], 0_WRONLY)) == -1))
                exit(1);
        rdwrt();
}
rdwrt()
{
        for (;;)
        {
                if (read(fdrd, &c, 1) != 1)
                        return;
                write(fdwt, &c, 1);
        }
}
```

Figure 7.34. Program where Parent and Child Do Not Share File Access

- The kernel does not change the signal-handling function until the user explicitly requests to do so;
- The kernel causes the process to ignore the signal until the user calls *signal* again.

9. Redesign the algorithm for handling signals such that the kernel automatically arranges for a process to ignore further instances of a signal it is handling until the signal handler returns. How can the kernel find out when the signal handler, running in user mode, returns? This specification is closer to the treatment of signals on BSD systems.

* 10. If a process receives a signal while sleeping at an interruptible priority in a system call, it *longjmp*s out of the system call. The kernel arranges for the process to execute its signal handler, if specified; when the process returns from the signal handler, it appears to have returned from the system call with an error indication (interrupted) on System V. The BSD system automatically restarts the system call for the process. How can this feature be implemented?

```
#include <signal.h>
main()
{
        extern catcher();

        signal(SIGCLD, catcher);
        if (fork() == 0)
                exit();
        /* pause suspends execution until receipt of a signal */
        pause();
}

catcher()
{
        printf("parent caught sig\n");
        signal(SIGCLD, catcher);
}
```

Figure 7.35. Catching Death of Child Signals

11. The conventional implementation of the *mkdir* command invokes the *mknod* system call to create the directory node, then calls the *link* system call twice to link the directory entries "." and ".." to the directory node and its parent directory. Without the three operations, the directory will not be in the correct format. What happens if *mkdir* receives a signal while executing? What if the signal is *SIGKILL*, which cannot be caught? Reconsider this problem if the system were to implement a *mkdir* system call.
12. A process checks for signals when it enters or leaves the sleep state (if it sleeps at an interruptible priority) and when it returns to user mode from the kernel after completion of a system call or after handling an interrupt. Why does the process not have to check for signals when entering the system for execution of a system call?
* 13. Suppose a process is about to return to user mode after executing a system call and it finds that it has no outstanding signals. Immediately after checking, the kernel handles an interrupt and sends the process a signal. (For instance, a user hits the "break" key.) What does the process do when the kernel returns from the interrupt?
* 14. If several signals are sent to a process simultaneously, the kernel handles them in the order that they are listed in the manual. Given the three possibilities for responding to receipt of a signal — catching the signals, *exit*ing after dumping a core image of the process, and *exit*ing without dumping a core image of the process — is there a better order for handling simultaneous signals? For example, if a process receives a quit signal (causes a core dump) and an interrupt signal (no core dump), does it make more sense to handle the quit signal or the interrup signal first?
15. Implement a new system call

    newpgrp(pid, ngrp);

    that resets the process group of another process, identified by process ID *pid* to *ngrp*. Discuss possible uses and dangers of such a system call.

16. Comment on the following statement: A process can sleep on any event in the *wait* algorithm, and the system would work correctly.
17. Consider implementation of a new system call,

    nowait(pid);

    where the process ID *pid* identifies a child of the process issuing the call. When issuing the call, the process informs the kernel that it will never *wait* for the child process to *exit*, so that the kernel can immediately clean up the child process slot when the child dies. How could the kernel implement such a solution? Discuss the merits of such a system call and compare it to the use of "death of child" signals.
18. The C loader automatically includes a startup routine that calls the function *main* in the user program. If the user program does not call *exit* internally, the startup routine calls *exit* for the user after the return from *main*. What would happen if the call to *exit* were missing from the startup routine (because of a bug in the loader) when the process returns from *main*?
19. What information does *wait* find when the child process invokes *exit* without a parameter? That is, the child process calls *exit()* instead of *exit(n)*. If a programmer consistently invokes *exit* without a parameter, how predictable is the value that *wait* examines? Demonstrate and prove your claim.
20. Describe what happens when a process executing the program in Figure 7.36 *execs* itself. How does the kernel avoid deadlocks over locked inodes?

```
main(argc, argv)
        int argc;
        char *argv[];
{
        execl(argv[0], argv[0], 0);
}
```

Figure 7.36. An Interesting Program

21. By convention, the first argument to *exec* is the (last component of the) file name that the process executes. What happens when a user executes the program in Figure 7.37. What happens if "a.out" is the load module produced by compiling the program in Figure 7.36?
22. Suppose the C language supported a new data type "read-only," such that a process incurs a protection fault whenever it attempts to write "read-only" data. Describe an implementation. (Hint: Compare to shared text.) What algorithms in the kernel change? What other objects could one consider for implementation as regions?
23. Describe how the algorithms for *open*, *chmod*, *unlink*, and *unmount* change for sticky-bit files. For example, what should the kernel do with a sticky-bit file when the file is *unlink*ed?
24. The superuser is the only user who has permission to *write* the password file "/etc/passwd", preventing malicious or errant users from corrupting its contents. The *passwd* program allows users to change their password entry, but it must make sure that they do not change other people's entries. How should it work?

```
main()
{
        if (fork() == 0)
        {
                execl("a.out", 0);
                printf("exec failed\n");
        }
}
```

Figure 7.37. An Unconventional Program

* 25. Explain the security problem that exists if a *setuid* program is not write-protected.
26. Execute the following sequence of shell commands, where the file "a.out" is an executable file.

```
chmod 4777 a.out
chown root a.out
```

The *chmod* command turns on the *setuid* bit (the 4 in 4777), and the owner "root" is conventionally the superuser. Can execution of such a sequence allow a simple breach of security?

27. What happens if you run the program in Figure 7.38? Why?

```
main()
{
        char *endpt;
        char *sbrk();
        int brk();

        endpt = sbrk(0);
        printf("endpt = %ud after sbrk\n", (int) endpt);

        while (endpt--)
        {
                if (brk(endpt) == -1)
                {
                        printf("brk of %ud failed\n", endpt);
                        exit();
                }
        }
}
```

Figure 7.38. A Tight Squeeze

28. The library routine *malloc* allocates more data space to a process by invoking the *brk* system call, and the library routine *free* releases memory previously allocated by malloc. The syntax for the calls is

```
ptr = malloc(size);
free(ptr);
```

where *size* is an unsigned integer representing the number of bytes to allocate, and *ptr* is a character pointer that points to the newly acquired space. When used as a parameter for free, *ptr* must have been previously returned by malloc. Implement the library routines.

29. What happens when running the program in Figure 7.39? Compare to the results predicted by the system manual.

```
main()
{
        int i;
        char *cp;
        extern char *sbrk();

        cp = sbrk(10);
        for (i = 0;  i < 10;  i++)
                *cp++ = 'a' + i;
        sbrk(-10);
        cp = sbrk(10);
        for (i = 0;  i < 10;  i++)
                printf("char %d = '%c'\n", i, *cp++);
}
```

Figure 7.39. A Simple Sbrk Example

30. When the shell creates a new process to execute a command, how does it know that the file is executable? If it is executable, how does it distinguish between a shell script and a file produced by a compilation? What is the correct sequence for checking the above cases?
31. The shell symbol ">>" appends output to the specified file: for example,

    run >> outfile

    creats the file "outfile" if it does not already exist and *writes* the file, or it *opens* the file and *writes* after the existing data. Write code to implement this.

```
main()
{
        exit(0);
}
```

Figure 7.40. Truth Program

32. The shell tests the *exit* return from a process, treating a 0 value as true and a non-0 value as false (note the inconsistency with C). Suppose the name of the executable file corresponding to the program in Figure 7.40 is *truth*. Describe what happens

when the shell executes the following loop. Enhance the sample shell code to handle this case.

```
while truth
do
truth &
done
```

33. Why must the shell create the processes to handle the two command components of a pipeline in the indicated order (Figure 7.29)?
34. Make the sample code for the shell loop more general in how it handles pipes. That is, allow it to handle an arbitrary number of pipes on the command line.
35. The environment variable PATH describes the ordered set of directories that the shell should search for executable files. The library functions *execlp* and *execvp* prepend directories listed in PATH to file name arguments that do not begin with a slash character. Implement these functions.
* 36. A superuser should set up the PATH environment variable so that the shell does *not* search for executable files in the current directory. What security problem exists if it attempts to execute files in the current directory?
37. How does the shell handle the *cd* (change directory) command? For the command line

```
cd pathname &
```

what does the shell do?
38. When the user types a "delete" or "break" key at the terminal, the terminal driver sends an interrupt signal to all processes in the process group of the login shell. The user intends to stop processes spawned by the shell but probably does not want to log off. How should the shell loop in Figure 7.28 be enhanced?
39. The user can type the command

```
nohup command_line
```

to disallow receipt of hangup signals and quit signals in the processes generated for "command_line." How should the shell loop in Figure 7.28 handle this?
40. Consider the sequence of shell commands

```
nroff -mm bigfile1 > big1out &
nroff -mm bigfile2 > big2out
```

and reexamine the shell loop shown in Figure 7.28. What would happen if the first *nroff* finished executing before the second one? How should the code for the shell loop be modified to handle this case correctly?
41. When executing untested programs from the shell, a common error message printed by the shell is "Bus error — core dumped." The program apparently did something illegal; how does the shell know that it should print an error message?
42. Only one *init* process can execute as process 1 on a system. However, a system administrator can change the state of the system by invoking *init*. For example, the system comes up in single user state when it is booted, meaning that the system console is active but user terminals are not. A system administrator types the command

```
init 2
```

at the console to change the state of *init* to state 2 (multi-user). The console shell *fork*s and *exec*s *init*. What should happen in the system, given that only one *init* process should be active?

43. The format of entries in the file "/etc/inittab" allows specification of an action associated with each generated process. For example, the action typically associated with *getty* is *respawn*, meaning that *init* should recreate the process if it dies. Practically, this means that *init* will spawn another *getty* process when a user logs off, allowing another user to access the now inoperative terminal line. How can *init* implement the respawn action?

44. Several kernel algorithms require a search of the process table. The search time can be improved by use of parent, child, and sibling pointers: The parent pointer points to the parent of the process, the child pointer points to any child process, and the sibling pointer points to another process with the same parent. A process finds all its children by following its child pointer and then following the sibling pointers (loops are illegal). What algorithms benefit from this implementation? What algorithms must remain the same?

8

PROCESS SCHEDULING AND TIME

On a time sharing system, the kernel allocates the CPU to a process for a period of time called a time slice or time quantum, preempts the process and schedules another one when the time slice expires, and reschedules the process to continue execution at a later time. The scheduler function on the UNIX system uses relative time of execution as a parameter to determine which process to schedule next. Every active process has a scheduling priority; the kernel switches context to that of the process with the highest priority when it does a context switch. The kernel recalculates the priority of the running process when it returns from kernel mode to user mode, and it periodically readjusts the priority of every "ready-to-run" process in user mode.

Some user processes also have a need to know about time: For example, the *time* command prints the time it took for another command to execute, and the *date* command prints the date and time of day. Various time-related system calls allow processes to set or retrieve kernel time values or to ascertain the amount of process CPU usage. The system keeps time with a hardware clock that interrupts the CPU at a fixed, hardware-dependent rate, typically between 50 and 100 times a second. Each occurrence of a clock interrupt is called a *clock tick*. This chapter explores time related activities on the UNIX system, considering process scheduling, system calls for time, and the functions of the clock interrupt handler.

8.1 PROCESS SCHEDULING

The scheduler on the UNIX system belongs to the general class of operating system schedulers known as *round robin with multilevel feedback*, meaning that the kernel allocates the CPU to a process for a time quantum, preempts a process that exceeds its time quantum, and feeds it back into one of several priority queues. A process may need many iterations through the "feedback loop" before it finishes. When the kernel does a context switch and restores the context of a process, the process resumes execution from the point where it had been suspended.

```
algorithm schedule_process
input:  none
output: none
{
        while (no process picked to execute)
        {
                for (every process on run queue)
                        pick highest priority process that is loaded in memory;
                if (no process eligible to execute)
                        idle the machine;
                        /* interrupt takes machine out of idle state */
        }
        remove chosen process from run queue;
        switch context to that of chosen process, resume its execution;
}
```

Figure 8.1. Algorithm for Process Scheduling

8.1.1 Algorithm

At the conclusion of a context switch, the kernel executes the algorithm to schedule a process (Figure 8.1), selecting the highest priority process from those in the states "ready to run and loaded in memory" and "preempted." It makes no sense to select a process if it is not loaded in memory, since it cannot execute until it is swapped in. If several processes tie for highest priority, the kernel picks the one that has been "ready to run" for the longest time, following a round robin scheduling policy. If there are no processes eligible for execution, the processor idles until the next interrupt, which will happen in at most one clock tick; after handling that interrupt, the kernel again attempts to schedule a process to run.

8.1.2 Scheduling Parameters

Each process table entry contains a priority field for process scheduling. The priority of a process in user mode is a function of its recent CPU usage, with processes getting a lower priority if they have recently used the CPU. The range of process priorities can be partitioned into two classes (see Figure 8.2): user priorities and kernel priorities. Each class contains several priority values, and each priority has a queue of processes logically associated with it. Processes with user-level priorities were preempted on their return from the kernel to user mode, and processes with kernel-level priorities achieved them in the *sleep* algorithm. User-level priorities are below a threshold value, and kernel-level priorities are above the threshold value. Kernel-level priorities are further subdivided: Processes with low kernel priority wake up on receipt of a signal, but processes with high kernel priority continue to sleep (see Section 7.2.1).

Figure 8.2 shows the threshold priority between user priorities and kernel priorities as the double line between priorities "waiting for child exit" and "user level 0." The priorities called "swapper," "waiting for disk I/O," "waiting for buffer," and "waiting for inode" are high, noninterruptible system priorities, with 1, 3, 2, and 1 processes queued on the respective priority level, and the priorities called "waiting for tty input," "waiting for tty output," and "waiting for child exit" are low, interruptible system priorities with 4, 0, and 2 processes queued, respectively. The figure distinguishes user priorities, calling them "user level 0," "user level 1," to "user level *n*,"[1] containing 0, 4, and 1 processes, respectively.

The kernel calculates the priority of a process in specific process states.

- It assigns priority to a process about to go to sleep, correlating a fixed, priority value with the reason for sleeping. The priority does not depend on the run-time characteristics of the process (I/O bound or CPU bound), but instead is a constant value that is hard-coded for each call to sleep, dependent on the reason the process is sleeping. Processes that sleep in lower-level algorithms tend to cause more system bottlenecks the longer they are inactive; hence they receive a higher priority than processes that would cause fewer system bottlenecks. For instance, a process sleeping and waiting for the completion of disk I/O has a higher priority than a process waiting for a free buffer for several reasons: First, the process waiting for completion of disk I/O already has a buffer; when it wakes up, there is a chance that it will do enough processing to release the buffer and, possibly, other resources. The more resources it frees, the better the chances are that other processes will not block waiting for resources. The system will have fewer context switches and, consequently, process response

1. The highest priority value on the system is 0. Thus, user level 0 has higher priority than user level 1, and so on.

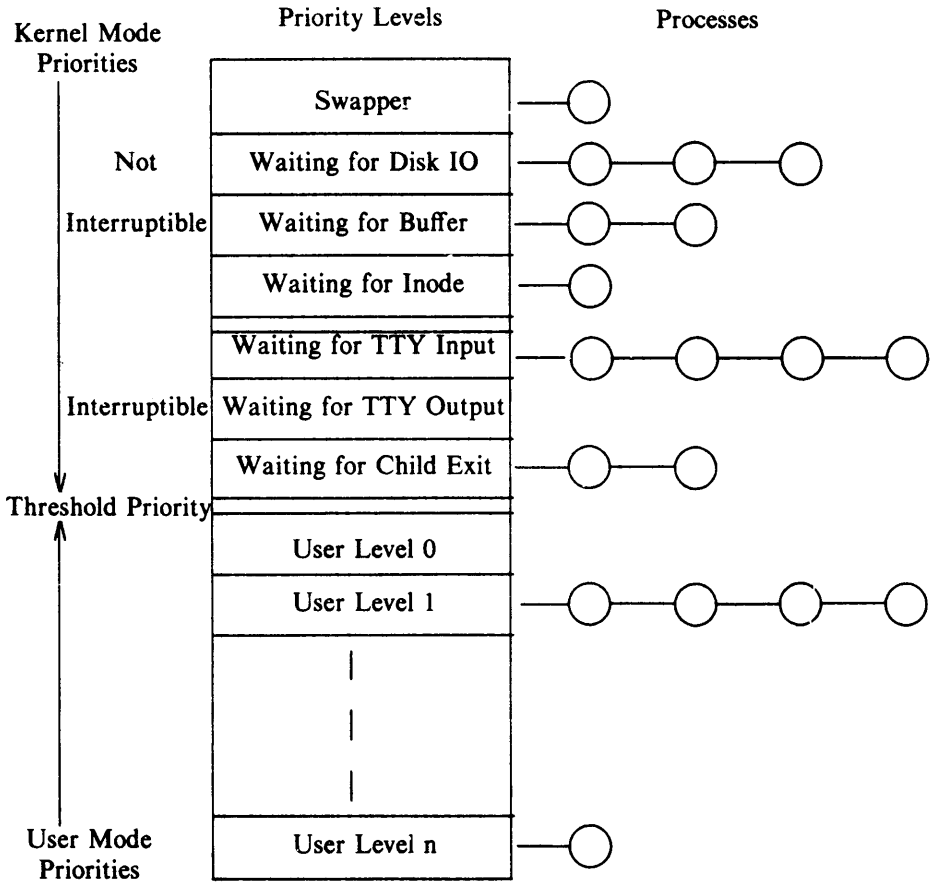

Figure 8.2. Range of Process Priorities

time and system throughput are better. Second, a process waiting for a free buffer may be waiting for a buffer held by the process waiting for completion of I/O. When the I/O completes, both processes wake up because they sleep on the same address. If the process waiting for the buffer were to run first, it would sleep again anyway until the other process frees the buffer; hence its priority is lower.

- The kernel adjusts the priority of a process that returns from kernel mode to user mode. The process may have previously entered the sleep state, changing its priority to a kernel-level priority that must be lowered to a user-level priority when returning to user mode. Also, the kernel penalizes the executing process in fairness to other processes, since it had just used valuable kernel resources.