```
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

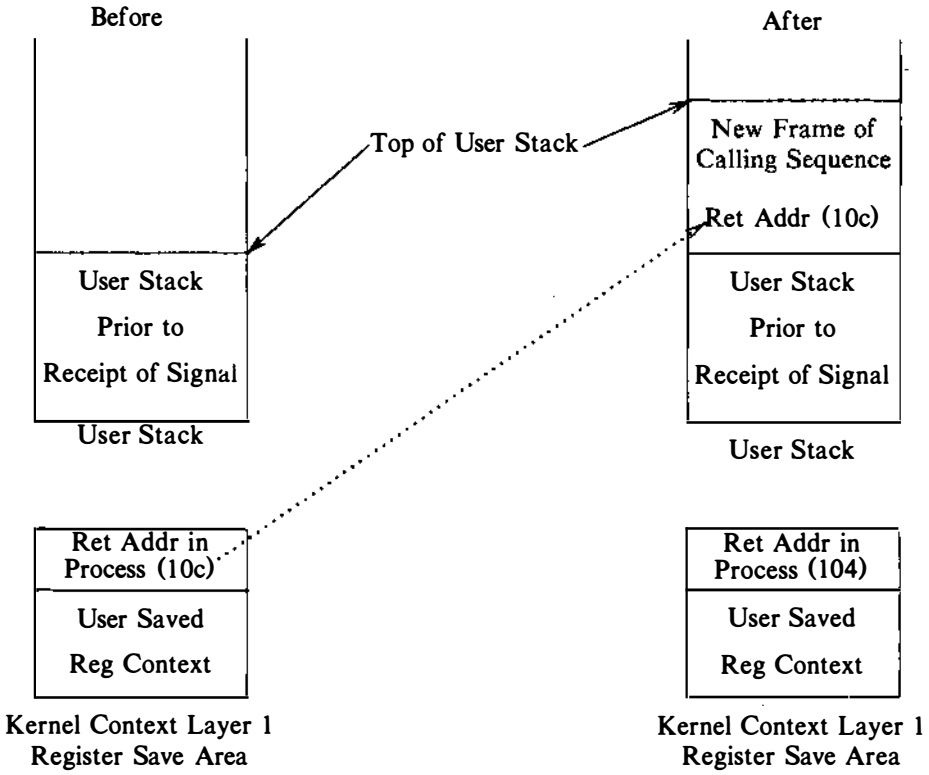

Figure 7.11. User Stack and Kernel Save Area Before and After Receipt of Signal

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