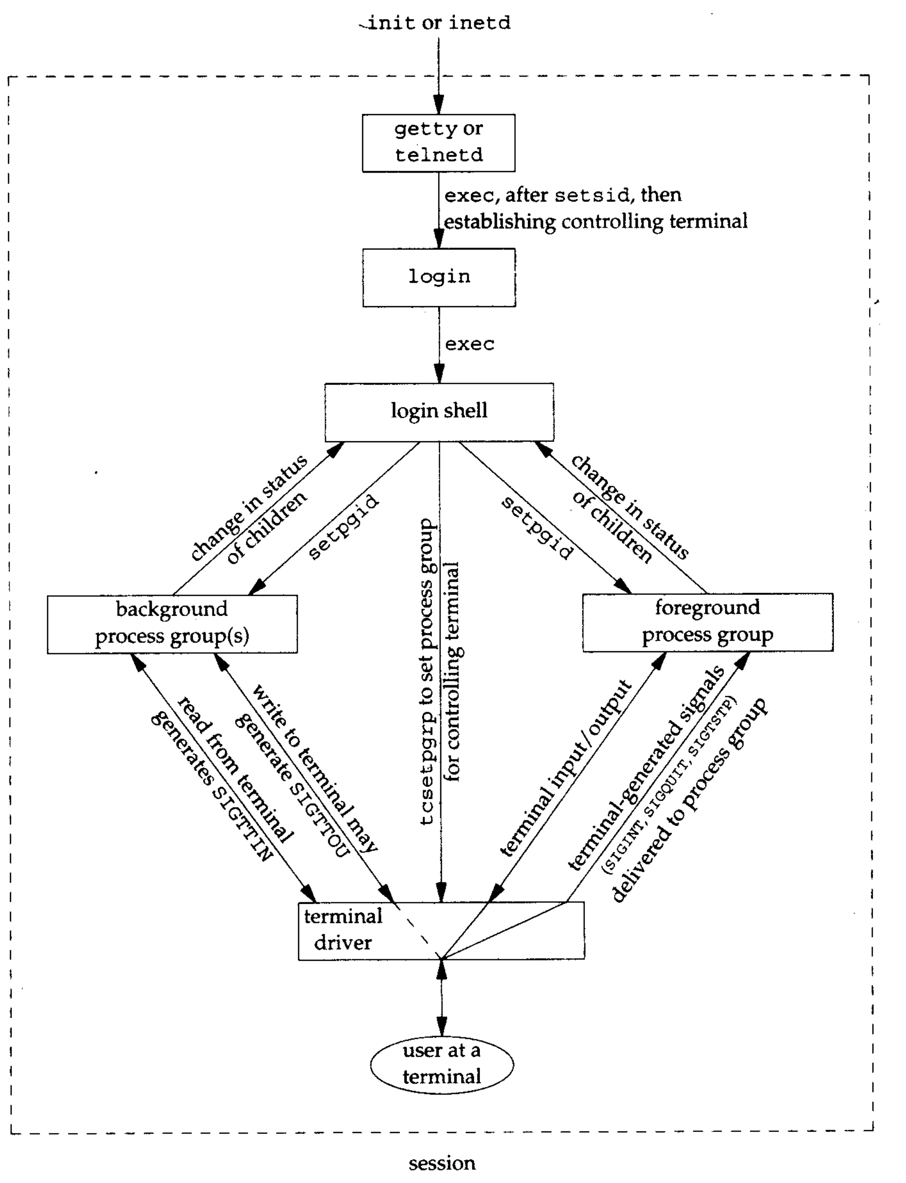
init or inetd
getty or telnetd
exec, after setsid, then establishing controlling terminal
login
exec
login shell
change in status of children
setpgid
tcsetpgrp to set process group for controlling terminal
setpgid
change in status of children
background process group(s)
foreground process group
read from terminal generates SIGTTIN
write to terminal may generate SIGTTOU
terminal input/output
terminal-generated signals (SIGINT, SIGQUIT, SIGTSTP) delivered to process group
terminal driver
user at a terminal
session

Figure 9.8 Summary of job control features with foreground and background jobs, and terminal driver

group to the actual terminal. The dashed line corresponding to the SIGTTOU signal means that whether the output from a process in the background process group appears on the terminal is an option.

Is job control necessary or desirable? Job control was originally designed and implemented before windowing terminals were widespread. Some people claim that a well-designed windowing system removes any need for job control. Some complain that the implementation of job control—requiring support from the kernel, the terminal driver, the shell, and some applications—is a hack. Some use job control with a windowing system, claiming a need for both. Regardless of your opinion, job control is a required feature of POSIX.1.

9.9 Shell Execution of Programs

Let's examine how the shells execute programs and how this relates to the concepts of process groups, controlling terminals, and sessions. To do this, we'll use the ps command again.

First, we'll use a shell that doesn't support job control—the classic Bourne shell running on Solaris. If we execute

```
ps -o pid,ppid,pgid,sid,comm
```

the output is

```
  PID  PPID  PGID   SID COMMAND
  949   947   949   949 sh
 1774   949   949   949 ps
```

The parent of the ps command is the shell, which we would expect. Both the shell and the ps command are in the same session and foreground process group (949). We say that 949 is the foreground process group because that is what you get when you execute a command with a shell that doesn't support job control.

Some platforms support an option to have the ps(1) command print the process group ID associated with the session's controlling terminal. This value would be shown under the TPGID column. Unfortunately, the output of the ps command often differs among versions of the UNIX System. For example, Solaris 9 doesn't support this option. Under FreeBSD 5.2.1 and Mac OS X 10.3, the command

```
ps -o pid,ppid,pgid,sess,tpgid,command
```

and under Linux 2.4.22, the command

```
ps -o pid,ppid,pgrp,session,tpgid,comm
```

print exactly the information we want.

Note that it is a misnomer to associate a process with a terminal process group ID (the TPGID column). A process does not have a terminal process control group. A process belongs to a process group, and the process group belongs to a session. The session may or may not have a controlling terminal. If the session does have a controlling terminal, then the terminal device knows the process group ID of the foreground process. This value can be set in the terminal driver with the tcsetpgrp function, as we show in Figure 9.8. The foreground process group ID is an attribute of the terminal, not the process. This value from the terminal device driver is what ps prints as the TPGID. If it finds that the session doesn't have a controlling terminal, ps prints −1.

If we execute the command in the background,

```
ps -o pid,ppid,pgid,sid,comm &
```

the only value that changes is the process ID of the command:

```
  PID  PPID  PGID   SID COMMAND
  949   947   949   949 sh
 1812   949   949   949 ps
```

This shell doesn't know about job control, so the background job is not put into its own process group and the controlling terminal isn't taken away from the background job.

Let's now look at how the Bourne shell handles a pipeline. When we execute

```
ps -o pid,ppid,pgid,sid,comm | cat1
```

the output is

```
  PID  PPID  PGID   SID COMMAND
  949   947   949   949 sh
 1823   949   949   949 cat1
 1824  1823   949   949 ps
```

(The program cat1 is just a copy of the standard cat program, with a different name. We have another copy of cat with the name cat2, which we'll use later in this section. When we have two copies of cat in a pipeline, the different names let us differentiate between the two programs.) Note that the last process in the pipeline is the child of the shell and that the first process in the pipeline is a child of the last process. It appears that the shell forks a copy of itself and that this copy then forks to make each of the previous processes in the pipeline.

If we execute the pipeline in the background,

```
ps -o pid,ppid,pgid,sid,comm | cat1 &
```

only the process IDs change. Since the shell doesn't handle job control, the process group ID of the background processes remains 949, as does the process group ID of the session.

What happens in this case if a background process tries to read from its controlling terminal? For example, suppose that we execute

```
cat > temp.foo &
```

With job control, this is handled by placing the background job into a background process group, which causes the signal SIGTTIN to be generated if the background job tries to read from the controlling terminal. The way this is handled without job control is that the shell automatically redirects the standard input of a background process to /dev/null, if the process doesn't redirect standard input itself. A read from /dev/null generates an end of file. This means that our background cat process immediately reads an end of file and terminates normally.

The previous paragraph adequately handles the case of a background process accessing the controlling terminal through its standard input, but what happens if a background process specifically opens /dev/tty and reads from the controlling terminal? The answer is "it depends," but it's probably not what we want. For example,

```
crypt < salaries | lpr &
```

is such a pipeline. We run it in the background, but the crypt program opens /dev/tty, changes the terminal characteristics (to disable echoing), reads from the device, and resets the terminal characteristics. When we execute this background pipeline, the prompt `Password:` from `crypt` is printed on the terminal, but what we enter (the encryption password) is read by the shell, which tries to execute a command of that name. The next line we enter to the shell is taken as the password, and the file is not encrypted correctly, sending junk to the printer. Here we have two processes trying to read from the same device at the same time, and the result depends on the system. Job control, as we described earlier, handles this multiplexing of a single terminal between multiple processes in a better fashion.

Returning to our Bourne shell example, if we execute three processes in the pipeline, we can examine the process control used by this shell:

```
ps -o pid,ppid,pgid,sid,comm | cat1 | cat2
```

generates the following output

```
  PID  PPID  PGID   SID COMMAND
  949   947   949   949 sh
 1988   949   949   949 cat2
 1989  1988   949   949 ps
 1990  1988   949   949 cat1
```

Don't be alarmed if the output on your system doesn't show the proper command names. Sometimes you might get results such as

```
  PID  PPID  PGID   SID COMMAND
  949   947   949   949 sh
 1831   949   949   949 sh
 1832  1831   949   949 ps
 1833  1831   949   949 sh
```

What's happening here is that the ps process is racing with the shell, which is forking and executing the `cat` commands. In this case, the shell hasn't yet completed the call to `exec` when ps has obtained the list of processes to print.

Again, the last process in the pipeline is the child of the shell, and all previous processes in the pipeline are children of the last process. Figure 9.9 shows what is happening. Since the last process in the pipeline is the child of the login shell, the shell is notified when that process (`cat2`) terminates.

Now let's examine the same examples using a job-control shell running on Linux. This shows the way these shells handle background jobs. We'll use the Bourne-again shell in this example; the results with other job-control shells are almost identical.

```
ps -o pid,ppid,pgrp,session,tpgid,comm
```

gives us

```
  PID  PPID  PGRP  SESS TPGID COMMAND
 2837  2818  2837  2837  5796 bash
 5796  2837  5796  2837  5796 ps
```

(Starting with this example, we show the foreground process group in a **bolder font**.) We immediately have a difference from our Bourne shell example. The Bourne-again shell places the foreground job (ps) into its own process group (5796). The ps command is the process group leader and the only process in this process group.

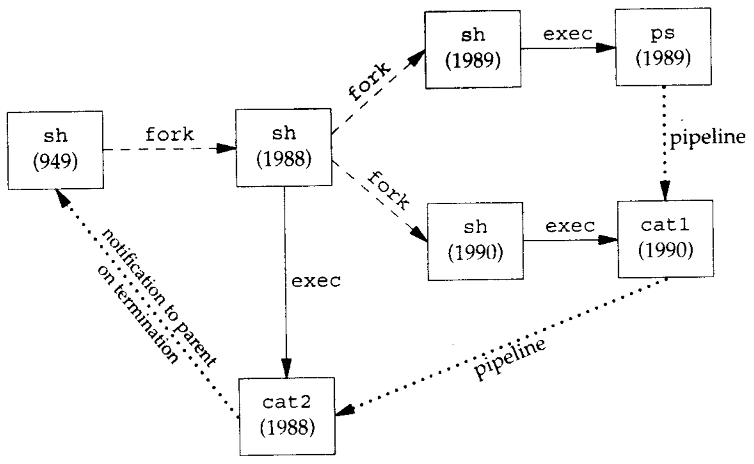

Figure 9.9 Processes in the pipeline ps | cat1 | cat2 when invoked by Bourne shell

Furthermore, this process group is the foreground process group, since it has the controlling terminal. Our login shell is a background process group while the ps command executes. Note, however, that both process groups, 2837 and 5796, are members of the same session. Indeed, we'll see that the session never changes through our examples in this section.

Executing this process in the background,

```
ps -o pid,ppid,pgrp,session,tpgid,comm &
```

gives us

```
  PID  PPID  PGRP  SESS TPGID COMMAND
 2837  2818  2837  2837  2837 bash
 5797  2837  5797  2837  2837 ps
```

Again, the ps command is placed into its own process group, but this time the process group (5797) is no longer the foreground process group. It is a background process group. The TPGID of 2837 indicates that the foreground process group is our login shell.

Executing two processes in a pipeline, as in

```
ps -o pid,ppid,pgrp,session,tpgid,comm | cat1
```

gives us

```
  PID  PPID  PGRP  SESS TPGID COMMAND
 2837  2818  2837  2837  5799 bash
 5799  2837  5799  2837  5799 ps
 5800  2837  5799  2837  5799 cat1
```

Both processes, ps and cat1, are placed into a new process group (5799), and this is the foreground process group. We can also see another difference between this example and the similar Bourne shell example. The Bourne shell created the last process in the pipeline first, and this final process was the parent of the first process. Here, the Bourne-again shell is the parent of both processes. If we execute this pipeline in the background,

```
ps -o pid,ppid,pgrp,session,tpgid,comm | cat1 &
```

the results are similar, but now ps and cat1 are placed in the same background process group:

```
 PID  PPID  PGRP  SESS TPGID COMMAND
2837  2818  2837  2837  2837 bash
5801  2837  5801  2837  2837 ps
5802  2837  5801  2837  2837 cat1
```

Note that the order in which a shell creates processes can differ depending on the particular shell in use.

9.10 Orphaned Process Groups

We've mentioned that a process whose parent terminates is called an orphan and is inherited by the init process. We now look at entire process groups that can be orphaned and how POSIX.1 handles this situation.

Example

Consider a process that forks a child and then terminates. Although this is nothing abnormal (it happens all the time), what happens if the child is stopped (using job control) when the parent terminates? How will the child ever be continued, and does the child know that it has been orphaned? Figure 9.10 shows this situation: the parent process has forked a child that stops, and the parent is about to exit.

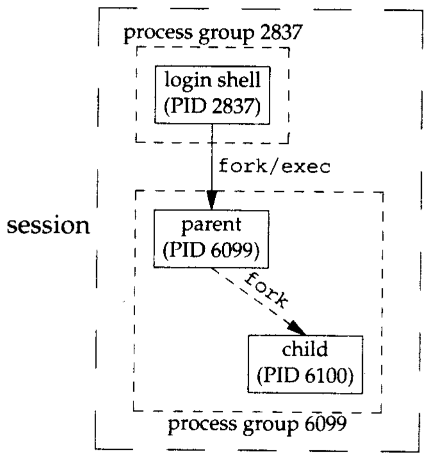

Figure 9.10 Example of a process group about to be orphaned

The program that creates this situation is shown in Figure 9.11. This program has some new features. Here, we are assuming a job-control shell. Recall from the previous section that the shell places the foreground process into its own process group (6099 in this example) and that the shell stays in its own process group (2837). The child inherits the process group of its parent (6099). After the `fork`,

- The parent sleeps for 5 seconds. This is our (imperfect) way of letting the child execute before the parent terminates.
- The child establishes a signal handler for the hang-up signal (`SIGHUP`). This is so we can see whether `SIGHUP` is sent to the child. (We discuss signal handlers in Chapter 10.)
- The child sends itself the stop signal (`SIGTSTP`) with the `kill` function. This stops the child, similar to our stopping a foreground job with our terminal's suspend character (Control-Z).
- When the parent terminates, the child is orphaned, so the child's parent process ID becomes 1, the `init` process ID.
- At this point, the child is now a member of an *orphaned process group*. The POSIX.1 definition of an orphaned process group is one in which the parent of every member is either itself a member of the group or is not a member of the group's session. Another way of wording this is that the process group is not orphaned as long as a process in the group has a parent in a different process group but in the same session. If the process group is not orphaned, there is a chance that one of those parents in a different process group but in the same session will restart a stopped process in the process group that is not orphaned. Here, the parent of every process in the group (e.g., process 1 is the parent of process 6100) belongs to another session.
- Since the process group is orphaned when the parent terminates, POSIX.1 requires that every process in the newly orphaned process group that is stopped (as our child is) be sent the hang-up signal (`SIGHUP`) followed by the continue signal (`SIGCONT`).
- This causes the child to be continued, after processing the hang-up signal. The default action for the hang-up signal is to terminate the process, so we have to provide a signal handler to catch the signal. We therefore expect the `printf` in the `sig_hup` function to appear before the `printf` in the `pr_ids` function.

Here is the output from the program shown in Figure 9.11:

```
$ ./a.out
parent: pid = 6099, ppid = 2837, pgrp = 6099, tpgrp = 6099
child: pid = 6100, ppid = 6099, pgrp = 6099, tpgrp = 6099
$ SIGHUP received, pid = 6100
child: pid = 6100, ppid = 1, pgrp = 6099, tpgrp = 2837
read error from controlling TTY, errno = 5
```

Note that our shell prompt appears with the output from the child, since two processes—our login shell and the child—are writing to the terminal. As we expect, the parent process ID of the child has become 1.

```
#include "apue.h"
#include <errno.h>

static void
sig_hup(int signo)
{
    printf("SIGHUP received, pid = %d\n", getpid());
}

static void
pr_ids(char *name)
{
    printf("%s: pid = %d, ppid = %d, pgrp = %d, tpgrp = %d\n",
        name, getpid(), getppid(), getpgrp(), tcgetpgrp(STDIN_FILENO));
    fflush(stdout);
}

int
main(void)
{
    char    c;
    pid_t   pid;

    pr_ids("parent");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {   /* parent */
        sleep(5);       /* sleep to let child stop itself */
        exit(0);        /* then parent exits */
    } else {            /* child */
        pr_ids("child");
        signal(SIGHUP, sig_hup);    /* establish signal handler */
        kill(getpid(), SIGTSTP);    /* stop ourself */
        pr_ids("child");    /* prints only if we're continued */
        if (read(STDIN_FILENO, &c, 1) != 1)
            printf("read error from controlling TTY, errno = %d\n",
                errno);
        exit(0);
    }
}
```

Figure 9.11 Creating an orphaned process group

After calling pr_ids in the child, the program tries to read from standard input. As we saw earlier in this chapter, when a background process group tries to read from its controlling terminal, SIGTTIN is generated for the background process group. But here we have an orphaned process group; if the kernel were to stop it with this signal, the processes in the process group would probably never be continued. POSIX.1 specifies that the read is to return an error with errno set to EIO (whose value is 5 on this system) in this situation.

Finally, note that our child was placed in a background process group when the parent terminated, since the parent was executed as a foreground job by the shell. □

We'll see another example of orphaned process groups in Section 19.5 with the `pty` program.

9.11 FreeBSD Implementation

Having talked about the various attributes of a process, process group, session, and controlling terminal, it's worth looking at how all this can be implemented. We'll look briefly at the implementation used by FreeBSD. Some details of the SVR4 implementation of these features can be found in Williams [1989]. Figure 9.12 shows the various data structures used by FreeBSD.

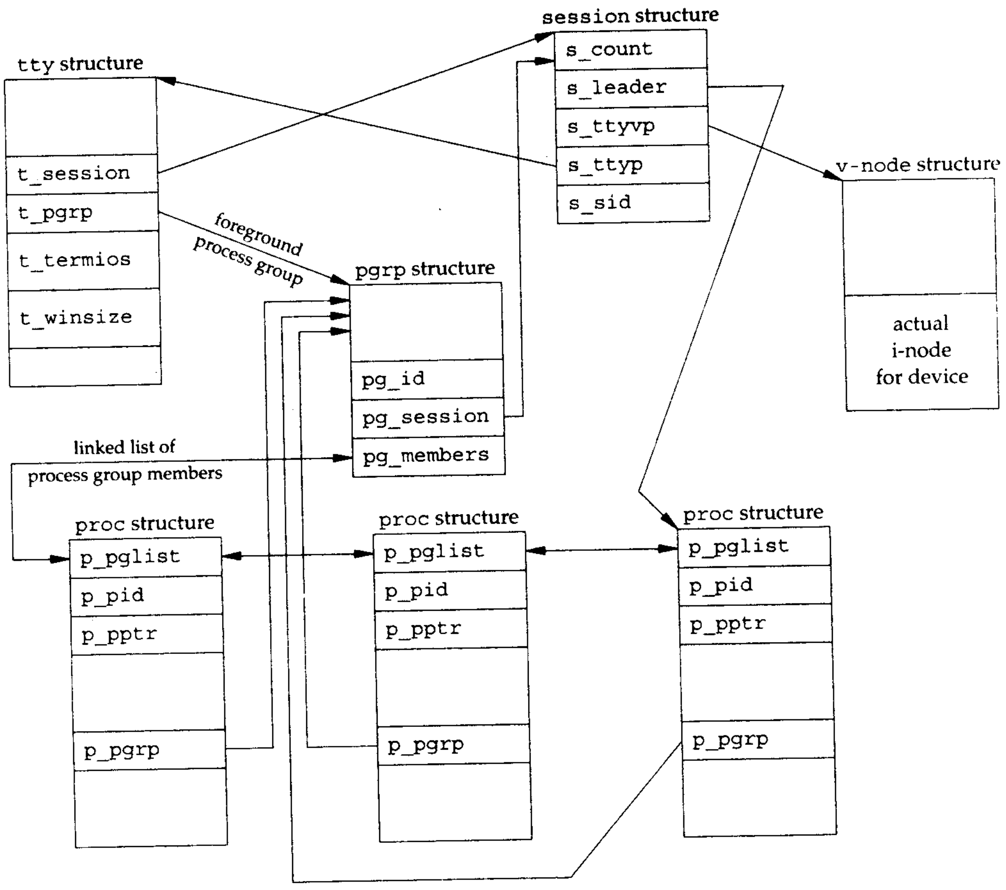

Figure 9.12 FreeBSD implementation of sessions and process groups

Let's look at all the fields that we've labeled, starting with the session structure. One of these structures is allocated for each session (e.g., each time setsid is called).

- s_count is the number of process groups in the session. When this counter is decremented to 0, the structure can be freed.
- s_leader is a pointer to the proc structure of the session leader.
- s_ttyvp is a pointer to the vnode structure of the controlling terminal.
- s_ttyp is a pointer to the tty structure of the controlling terminal.
- s_sid is the session ID. Recall that the concept of a session ID is not part of the Single UNIX Specification.

When setsid is called, a new session structure is allocated within the kernel. Now s_count is set to 1, s_leader is set to point to the proc structure of the calling process, s_sid is set to the process ID, and s_ttyvp and s_ttyp are set to null pointers, since the new session doesn't have a controlling terminal.

Let's move to the tty structure. The kernel contains one of these structures for each terminal device and each pseudo-terminal device. (We talk more about pseudo terminals in Chapter 19.)

- t_session points to the session structure that has this terminal as its controlling terminal. (Note that the tty structure points to the session structure and vice versa.) This pointer is used by the terminal to send a hang-up signal to the session leader if the terminal loses carrier (Figure 9.7).
- t_pgrp points to the pgrp structure of the foreground process group. This field is used by the terminal driver to send signals to the foreground process group. The three signals generated by entering special characters (interrupt, quit, and suspend) are sent to the foreground process group.
- t_termios is a structure containing all the special characters and related information for this terminal, such as baud rate, is echo on or off, and so on. We'll return to this structure in Chapter 18.
- t_winsize is a winsize structure that contains the current size of the terminal window. When the size of the terminal window changes, the SIGWINCH signal is sent to the foreground process group. We show how to set and fetch the terminal's current window size in Section 18.12.

Note that to find the foreground process group of a particular session, the kernel has to start with the session structure, follow s_ttyp to get to the controlling terminal's tty structure, and then follow t_pgrp to get to the foreground process group's pgrp structure. The pgrp structure contains the information for a particular process group.

- pg_id is the process group ID.
- pg_session points to the session structure for the session to which this process group belongs.
- pg_members is a pointer to the list of proc structures that are members of this process group. The p_pglist structure in that proc structure is a

doubly-linked list entry that points to both the next process and the previous process in the group, and so on, until a null pointer is encountered in the proc structure of the last process in the group.

The proc structure contains all the information for a single process.

- p_pid contains the process ID.
- p_pptr is a pointer to the proc structure of the parent process.
- p_pgrp points to the pgrp structure of the process group to which this process belongs.
- p_pglist is a structure containing pointers to the next and previous processes in the process group, as we mentioned earlier.

Finally, we have the vnode structure. This structure is allocated when the controlling terminal device is opened. All references to /dev/tty in a process go through this vnode structure. We show the actual i-node as being part of the v-node.

9.12 Summary

This chapter has described the relationships between groups of processes: sessions, which are made up of process groups. Job control is a feature supported by most UNIX systems today, and we've described how it's implemented by a shell that supports job control. The controlling terminal for a process, /dev/tty, is also involved in these process relationships.

We've made numerous references to the signals that are used in all these process relationships. The next chapter continues the discussion of signals, looking at all the UNIX System signals in detail.

Exercises

9.1 Refer back to our discussion of the utmp and wtmp files in Section 6.8. Why are the logout records written by the init process? Is this handled the same way for a network login?

9.2 Write a small program that calls fork and has the child create a new session. Verify that the child becomes a process group leader and that the child no longer has a controlling terminal.

10

Signals

10.1 Introduction

Signals are software interrupts. Most nontrivial application programs need to deal with signals. Signals provide a way of handling asynchronous events: a user at a terminal typing the interrupt key to stop a program or the next program in a pipeline terminating prematurely.

Signals have been provided since the early versions of the UNIX System, but the signal model provided with systems such as Version 7 was not reliable. Signals could get lost, and it was difficult for a process to turn off selected signals when executing critical regions of code. Both 4.3BSD and SVR3 made changes to the signal model, adding what are called *reliable signals*. But the changes made by Berkeley and AT&T were incompatible. Fortunately, POSIX.1 standardized the reliable-signal routines, and that is what we describe here.

In this chapter, we start with an overview of signals and a description of what each signal is normally used for. Then we look at the problems with earlier implementations. It is often important to understand what is wrong with an implementation before seeing how to do things correctly. This chapter contains numerous examples that are not entirely correct and a discussion of the defects.

10.2 Signal Concepts

First, every signal has a name. These names all begin with the three characters `SIG`. For example, `SIGABRT` is the abort signal that is generated when a process calls the `abort` function. `SIGALRM` is the alarm signal that is generated when the timer set by the `alarm` function goes off. Version 7 had 15 different signals; SVR4 and 4.4BSD both have 31 different signals. FreeBSD 5.2.1, Mac OS X 10.3, and Linux 2.4.22 support 31 different signals, whereas Solaris 9 supports 38 different signals. Both Linux and Solaris, however, support additional application-defined signals as real-time extensions (the real-time extensions in POSIX aren't covered in this book; refer to Gallmeister [1995] for more information).

These names are all defined by positive integer constants (the signal number) in the header `<signal.h>`.

> Implementations actually define the individual signals in an alternate header file, but this header file is included by `<signal.h>`. It is considered bad form for the kernel to include header files meant for user-level applications, so if the applications and the kernel both need the same definitions, the information is placed in a kernel header file that is then included by the user-level header file. Thus, both FreeBSD 5.2.1 and Mac OS X 10.3 define the signals in `<sys/signal.h>`. Linux 2.4.22 defines the signals in `<bits/signum.h>`, and Solaris 9 defines them in `<sys/iso/signal_iso.h>`.

No signal has a signal number of 0. We'll see in Section 10.9 that the `kill` function uses the signal number of 0 for a special case. POSIX.1 calls this value the *null signal*.

Numerous conditions can generate a signal.

- The terminal-generated signals occur when users press certain terminal keys. Pressing the DELETE key on the terminal (or Control-C on many systems) normally causes the interrupt signal (`SIGINT`) to be generated. This is how to stop a runaway program. (We'll see in Chapter 18 how this signal can be mapped to any character on the terminal.)
- Hardware exceptions generate signals: divide by 0, invalid memory reference, and the like. These conditions are usually detected by the hardware, and the kernel is notified. The kernel then generates the appropriate signal for the process that was running at the time the condition occurred. For example, `SIGSEGV` is generated for a process that executes an invalid memory reference.
- The `kill`(2) function allows a process to send any signal to another process or process group. Naturally, there are limitations: we have to be the owner of the process that we're sending the signal to, or we have to be the superuser.
- The `kill`(1) command allows us to send signals to other processes. This program is just an interface to the `kill` function. This command is often used to terminate a runaway background process.
- Software conditions can generate signals when something happens about which the process should be notified. These aren't hardware-generated conditions (as is the divide-by-0 condition), but software conditions. Examples are `SIGURG` (generated when out-of-band data arrives over a network connection), `SIGPIPE` (generated when a process writes to a pipe after the reader of the pipe has terminated), and `SIGALRM` (generated when an alarm clock set by the process expires).

Signals are classic examples of asynchronous events. Signals occur at what appear to be random times to the process. The process can't simply test a variable (such as `errno`) to see whether a signal has occurred; instead, the process has to tell the kernel "if and when this signal occurs, do the following."

We can tell the kernel to do one of three things when a signal occurs. We call this the *disposition* of the signal, or the *action* associated with a signal.

1. Ignore the signal. This works for most signals, but two signals can never be ignored: `SIGKILL` and `SIGSTOP`. The reason these two signals can't be ignored is to provide the kernel and the superuser with a surefire way of either killing or stopping any process. Also, if we ignore some of the signals that are generated by a hardware exception (such as illegal memory reference or divide by 0), the behavior of the process is undefined.

2. Catch the signal. To do this, we tell the kernel to call a function of ours whenever the signal occurs. In our function, we can do whatever we want to handle the condition. If we're writing a command interpreter, for example, when the user generates the interrupt signal at the keyboard, we probably want to return to the main loop of the program, terminating whatever command we were executing for the user. If the `SIGCHLD` signal is caught, it means that a child process has terminated, so the signal-catching function can call `waitpid` to fetch the child's process ID and termination status. As another example, if the process has created temporary files, we may want to write a signal-catching function for the `SIGTERM` signal (the termination signal that is the default signal sent by the `kill` command) to clean up the temporary files. Note that the two signals `SIGKILL` and `SIGSTOP` can't be caught.

3. Let the default action apply. Every signal has a default action, shown in Figure 10.1. Note that the default action for most signals is to terminate the process.

Figure 10.1 lists the names of all the signals, an indication of which systems support the signal, and the default action for the signal. The SUS column contains • if the signal is defined as part of the base POSIX.1 specification and **XSI** if it is defined as an XSI extension to the base.

When the default action is labeled "terminate+core," it means that a memory image of the process is left in the file named `core` of the current working directory of the process. (Because the file is named `core`, it shows how long this feature has been part of the UNIX System.) This file can be used with most UNIX System debuggers to examine the state of the process at the time it terminated.

The generation of the `core` file is an implementation feature of most versions of the UNIX System. Although this feature is not part of POSIX.1, it is mentioned as a potential implementation-specific action in the Single UNIX Specification's XSI extension.

The name of the core file varies among implementations. On FreeBSD 5.2.1, for example, the core file is named *cmdname*.core, where *cmdname* is the name of the command corresponding to the process that received the signal. On Mac OS X 10.3, the core file is named core.*pid*, where *pid* is the ID of the process that received the signal. (These systems allow the core filename to be configured via a `sysctl` parameter.)

Most implementations leave the core file in the current working directory of the corresponding process; Mac OS X places all core files in `/cores` instead.

Name	Description	ISO C	SUS	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	Default action
SIGABRT	abnormal termination (abort)	•	•	•	•	•	•	terminate+core
SIGALRM	timer expired (alarm)		•	•	•	•	•	terminate
SIGBUS	hardware fault		•	•	•	•	•	terminate+core
SIGCANCEL	threads library internal use						•	ignore
SIGCHLD	change in status of child		•	•	•	•	•	ignore
SIGCONT	continue stopped process		•	•	•	•	•	continue/ignore
SIGEMT	hardware fault			•	•	•	•	terminate+core
SIGFPE	arithmetic exception	•	•	•	•	•	•	terminate+core
SIGFREEZE	checkpoint freeze						•	ignore
SIGHUP	hangup		•	•	•	•	•	terminate
SIGILL	illegal instruction	•	•	•	•	•	•	terminate+core
SIGINFO	status request from keyboard			•		•		ignore
SIGINT	terminal interrupt character	•	•	•	•	•	•	terminate
SIGIO	asynchronous I/O			•	•	•	•	terminate/ignore
SIGIOT	hardware fault			•	•	•	•	terminate+core
SIGKILL	termination		•	•	•	•	•	terminate
SIGLWP	threads library internal use						•	ignore
SIGPIPE	write to pipe with no readers		•	•	•	•	•	terminate
SIGPOLL	pollable event (poll)		XSI		•		•	terminate
SIGPROF	profiling time alarm (setitimer)		XSI	•	•	•	•	terminate
SIGPWR	power fail/restart				•		•	terminate/ignore
SIGQUIT	terminal quit character		•	•	•	•	•	terminate+core
SIGSEGV	invalid memory reference	•	•	•	•	•	•	terminate+core
SIGSTKFLT	coprocessor stack fault				•			terminate
SIGSTOP	stop		•	•	•	•	•	stop process
SIGSYS	invalid system call		XSI	•	•	•	•	terminate+core
SIGTERM	termination	•	•	•	•	•	•	terminate
SIGTHAW	checkpoint thaw						•	ignore
SIGTRAP	hardware fault		XSI	•	•	•	•	terminate+core
SIGTSTP	terminal stop character		•	•	•	•	•	stop process
SIGTTIN	background read from control tty		•	•	•	•	•	stop process
SIGTTOU	background write to control tty		•	•	•	•	•	stop process
SIGURG	urgent condition (sockets)		•	•	•	•	•	ignore
SIGUSR1	user-defined signal		•	•	•	•	•	terminate
SIGUSR2	user-defined signal		•	•	•	•	•	terminate
SIGVTALRM	virtual time alarm (setitimer)		XSI	•	•	•	•	terminate
SIGWAITING	threads library internal use						•	ignore
SIGWINCH	terminal window size change			•	•	•	•	ignore
SIGXCPU	CPU limit exceeded (setrlimit)		XSI	•	•	•	•	terminate+core/ ignore
SIGXFSZ	file size limit exceeded (setrlimit)		XSI	•	•	•	•	terminate+core/ ignore
SIGXRES	resource control exceeded						•	ignore

Figure 10.1 UNIX System signals

The core file will not be generated if (a) the process was set-user-ID and the current user is not the owner of the program file, or (b) the process was set-group-ID and the current user is not the group owner of the file, (c) the user does not have permission to write in the current working directory, (d) the file already exists and the user does not have permission to write to it, or (e) the file is too big (recall the `RLIMIT_CORE` limit in Section 7.11). The permissions of the `core` file (assuming that the file doesn't already exist) are usually user-read and user-write, although Mac OS X sets only user-read.

In Figure 10.1, the signals with a description "hardware fault" correspond to implementation-defined hardware faults. Many of these names are taken from the original PDP-11 implementation of the UNIX System. Check your system's manuals to determine exactly what type of error these signals correspond to.

We now describe each of these signals in more detail.

`SIGABRT` This signal is generated by calling the `abort` function (Section 10.17). The process terminates abnormally.

`SIGALRM` This signal is generated when a timer set with the `alarm` function expires (see Section 10.10 for more details). This signal is also generated when an interval timer set by the `setitimer(2)` function expires.

`SIGBUS` This indicates an implementation-defined hardware fault. Implementations usually generate this signal on certain types of memory faults, as we describe in Section 14.9.

`SIGCANCEL` This signal is used internally by the Solaris threads library. It is not meant for general use.

`SIGCHLD` Whenever a process terminates or stops, the `SIGCHLD` signal is sent to the parent. By default, this signal is ignored, so the parent must catch this signal if it wants to be notified whenever a child's status changes. The normal action in the signal-catching function is to call one of the `wait` functions to fetch the child's process ID and termination status.

Earlier releases of System V had a similar signal named `SIGCLD` (without the `H`). The semantics of this signal were different from those of other signals, and as far back as SVR2, the manual page strongly discouraged its use in new programs. (Strangely enough, this warning disappeared in the SVR3 and SVR4 versions of the manual page.) Applications should use the standard `SIGCHLD` signal, but be aware that many systems define `SIGCLD` to be the same as `SIGCHLD` for backward compatibility. If you maintain software that uses `SIGCLD`, you need to check your system's manual page to see what semantics it follows. We discuss these two signals in Section 10.7.

`SIGCONT` This job-control signal is sent to a stopped process when it is continued. The default action is to continue a stopped process, but to ignore the signal if the process wasn't stopped. A full-screen editor, for example, might catch this signal and use the signal handler to make a note to redraw the terminal screen. See Section 10.20 for additional details.

`SIGEMT` This indicates an implementation-defined hardware fault.

The name EMT comes from the PDP-11 "emulator trap" instruction. Not all platforms support this signal. On Linux, for example, `SIGEMT` is supported only for selected architectures, such as SPARC, MIPS, and PA-RISC.

`SIGFPE` This signals an arithmetic exception, such as divide by 0, floating-point overflow, and so on.

`SIGFREEZE` This signal is defined only by Solaris. It is used to notify processes that need to take special action before freezing the system state, such as might happen when a system goes into hibernation or suspended mode.

`SIGHUP` This signal is sent to the controlling process (session leader) associated with a controlling terminal if a disconnect is detected by the terminal interface. Referring to Figure 9.12, we see that the signal is sent to the process pointed to by the `s_leader` field in the `session` structure. This signal is generated for this condition only if the terminal's `CLOCAL` flag is not set. (The `CLOCAL` flag for a terminal is set if the attached terminal is local. The flag tells the terminal driver to ignore all modem status lines. We describe how to set this flag in Chapter 18.)

Note that the session leader that receives this signal may be in the background; see Figure 9.7 for an example. This differs from the normal terminal-generated signals (interrupt, quit, and suspend), which are always delivered to the foreground process group.

This signal is also generated if the session leader terminates. In this case, the signal is sent to each process in the foreground process group.

This signal is commonly used to notify daemon processes (Chapter 13) to reread their configuration files. The reason `SIGHUP` is chosen for this is that a daemon should not have a controlling terminal and would normally never receive this signal.

`SIGILL` This signal indicates that the process has executed an illegal hardware instruction.

4.3BSD generated this signal from the `abort` function. `SIGABRT` is now used for this.

`SIGINFO` This BSD signal is generated by the terminal driver when we type the status key (often Control-T). This signal is sent to all processes in the foreground process group (refer to Figure 9.8). This signal normally causes status information on processes in the foreground process group to be displayed on the terminal.

Linux doesn't provide support for `SIGINFO` except on the Alpha platform, where it is defined to be the same value as `SIGPWR`.

`SIGINT` This signal is generated by the terminal driver when we type the interrupt key (often DELETE or Control-C). This signal is sent to all processes in the foreground process group (refer to Figure 9.8). This signal is often used to terminate a runaway program, especially when it's generating a lot of unwanted output on the screen.

`SIGIO` This signal indicates an asynchronous I/O event. We discuss it in Section 14.6.2.

In Figure 10.1, we labeled the default action for `SIGIO` as either "terminate" or "ignore." Unfortunately, the default depends on the system. Under System V, `SIGIO` is identical to `SIGPOLL`, so its default action is to terminate the process. Under BSD, the default is to ignore the signal.

Linux 2.4.22 and Solaris 9 define `SIGIO` to be the same value as `SIGPOLL`, so the default behavior is to terminate the process. On FreeBSD 5.2.1 and Mac OS X 10.3, the default is to ignore the signal.

`SIGIOT` This indicates an implementation-defined hardware fault.

The name IOT comes from the PDP-11 mnemonic for the "input/output TRAP" instruction. Earlier versions of System V generated this signal from the `abort` function. `SIGABRT` is now used for this.

On FreeBSD 5.2.1, Linux 2.4.22, Mac OS X 10.3, and Solaris 9, `SIGIOT` is defined to be the same value as `SIGABRT`.

`SIGKILL` This signal is one of the two that can't be caught or ignored. It provides the system administrator with a sure way to kill any process.

`SIGLWP` This signal is used internally by the Solaris threads library, and is not available for general use.

`SIGPIPE` If we write to a pipeline but the reader has terminated, `SIGPIPE` is generated. We describe pipes in Section 15.2. This signal is also generated when a process writes to a socket of type `SOCK_STREAM` that is no longer connected. We describe sockets in Chapter 16.

`SIGPOLL` This signal can be generated when a specific event occurs on a pollable device. We describe this signal with the `poll` function in Section 14.5.2. `SIGPOLL` originated with SVR3, and loosely corresponds to the BSD `SIGIO` and `SIGURG` signals.

On Linux and Solaris, `SIGPOLL` is defined to have the same value as `SIGIO`.

`SIGPROF` This signal is generated when a profiling interval timer set by the `setitimer`(2) function expires.

`SIGPWR` This signal is system dependent. Its main use is on a system that has an uninterruptible power supply (UPS). If power fails, the UPS takes over and the software can usually be notified. Nothing needs to be done at this point, as the system continues running on battery power. But if the battery gets low (if the power is off for an extended period), the software is usually notified again; at this point, it behooves the system to shut everything down within about 15–30 seconds. This is when `SIGPWR` should be sent. Most systems have the process that is notified of the low-battery condition send the `SIGPWR` signal to the `init` process, and `init` handles the shutdown.

Linux 2.4.22 and Solaris 9 have entries in the `inittab` file for this purpose: `powerfail` and `powerwait` (or `powerokwait`).

In Figure 10.1, we labeled the default action for `SIGPWR` as either "terminate" or "ignore." Unfortunately, the default depends on the system. The default on Linux is to terminate the process. On Solaris, the signal is ignored by default.

`SIGQUIT`	This signal is generated by the terminal driver when we type the terminal quit key (often Control-backslash). This signal is sent to all processes in the foreground process group (refer to Figure 9.8). This signal not only terminates the foreground process group (as does `SIGINT`), but also generates a `core` file.
`SIGSEGV`	This signal indicates that the process has made an invalid memory reference. The name SEGV stands for "segmentation violation."
`SIGSTKFLT`	This signal is defined only by Linux. This signal showed up in the earliest versions of Linux, intended to be used for stack faults taken by the math coprocessor. This signal is not generated by the kernel, but remains for backward compatibility.
`SIGSTOP`	This job-control signal stops a process. It is like the interactive stop signal (`SIGTSTP`), but `SIGSTOP` cannot be caught or ignored.
`SIGSYS`	This signals an invalid system call. Somehow, the process executed a machine instruction that the kernel thought was a system call, but the parameter with the instruction that indicates the type of system call was invalid. This might happen if you build a program that uses a new system call and you then try to run the same binary on an older version of the operating system where the system call doesn't exist.
`SIGTERM`	This is the termination signal sent by the `kill`(1) command by default.
`SIGTHAW`	This signal is defined only by Solaris and is used to notify processes that need to take special action when the system resumes operation after being suspended.
`SIGTRAP`	This indicates an implementation-defined hardware fault. The signal name comes from the PDP-11 TRAP instruction. Implementations often use this signal to transfer control to a debugger when a breakpoint instruction is executed.
`SIGTSTP`	This interactive stop signal is generated by the terminal driver when we type the terminal suspend key (often Control-Z). This signal is sent to all processes in the foreground process group (refer to Figure 9.8). Unfortunately, the term *stop* has different meanings. When discussing job control and signals, we talk about stopping and continuing jobs. The terminal driver, however, has historically used the term *stop* to refer to stopping and starting the terminal output using the Control-S and Control-Q characters. Therefore, the terminal driver calls the character that generates the interactive stop signal the suspend character, not the stop character.
`SIGTTIN`	This signal is generated by the terminal driver when a process in a background process group tries to read from its controlling terminal. (Refer to the discussion of this topic in Section 9.8.) As special cases, if

either (a) the reading process is ignoring or blocking this signal or (b) the process group of the reading process is orphaned, then the signal is not generated; instead, the read operation returns an error with `errno` set to `EIO`.

`SIGTTOU` This signal is generated by the terminal driver when a process in a background process group tries to write to its controlling terminal. (Refer to the discussion of this topic in Section 9.8.) Unlike the `SIGTTIN` signal just described, a process has a choice of allowing background writes to the controlling terminal. We describe how to change this option in Chapter 18.

If background writes are not allowed, then like the `SIGTTIN` signal, there are two special cases: if either (a) the writing process is ignoring or blocking this signal or (b) the process group of the writing process is orphaned, then the signal is not generated; instead, the write operation returns an error with `errno` set to `EIO`.

Regardless of whether background writes are allowed, certain terminal operations (other than writing) can also generate the `SIGTTOU` signal: `tcsetattr`, `tcsendbreak`, `tcdrain`, `tcflush`, `tcflow`, and `tcsetpgrp`. We describe these terminal operations in Chapter 18.

`SIGURG` This signal notifies the process that an urgent condition has occurred. This signal is optionally generated when out-of-band data is received on a network connection.

`SIGUSR1` This is a user-defined signal, for use in application programs.

`SIGUSR2` This is another user-defined signal, similar to `SIGUSR1`, for use in application programs.

`SIGVTALRM` This signal is generated when a virtual interval timer set by the `setitimer`(2) function expires.

`SIGWAITING` This signal is used internally by the Solaris threads library, and is not available for general use.

`SIGWINCH` The kernel maintains the size of the window associated with each terminal and pseudo terminal. A process can get and set the window size with the `ioctl` function, which we describe in Section 18.12. If a process changes the window size from its previous value using the `ioctl` set-window-size command, the kernel generates the `SIGWINCH` signal for the foreground process group.

`SIGXCPU` The Single UNIX Specification supports the concept of resource limits as an XSI extension; refer to Section 7.11. If the process exceeds its soft CPU time limit, the `SIGXCPU` signal is generated.

In Figure 10.1, we labeled the default action for `SIGXCPU` as either "terminate with a core file" or "ignore." Unfortunately, the default depends on the operating system. Linux 2.4.22 and Solaris 9 support a default action of terminate with a core file, whereas FreeBSD 5.2.1 and Mac OS X 10.3 support a default action of ignore. The Single UNIX Specification requires that the default action be to terminate the process abnormally. Whether a core file is generated is left up to the implementation.

`SIGXFSZ` This signal is generated if the process exceeds its soft file size limit; refer to Section 7.11.

Just as with `SIGXCPU`, the default action taken with `SIGXFSZ` depends on the operating system. On Linux 2.4.22 and Solaris 9, the default is to terminate the process and create a core file. On FreeBSD 5.2.1 and Mac OS X 10.3, the default is to be ignored. The Single UNIX Specification requires that the default action be to terminate the process abnormally. Whether a core file is generated is left up to the implementation.

`SIGXRES` This signal is defined only by Solaris. This signal is optionally used to notify processes that have exceeded a preconfigured resource value. The Solaris resource control mechanism is a general facility for controlling the use of shared resources among independent application sets.

10.3 `signal` Function

The simplest interface to the signal features of the UNIX System is the `signal` function.

```
#include <signal.h>

void (*signal(int signo, void (*func)(int)))(int);
```

Returns: previous disposition of signal (see following) if OK, `SIG_ERR` on error

The `signal` function is defined by ISO C, which doesn't involve multiple processes, process groups, terminal I/O, and the like. Therefore, its definition of signals is vague enough to be almost useless for UNIX systems.

Implementations derived from UNIX System V support the `signal` function, but it provides the old unreliable-signal semantics. (We describe these older semantics in Section 10.4.) This function provides backward compatibility for applications that require the older semantics. New applications should not use these unreliable signals.

4.4BSD also provides the `signal` function, but it is defined in terms of the `sigaction` function (which we describe in Section 10.14), so using it under 4.4BSD provides the newer reliable-signal semantics. FreeBSD 5.2.1 and Mac OS X 10.3 follow this strategy.

Solaris 9 has roots in both System V and BSD, but it chooses to follow the System V semantics for the `signal` function.

On Linux 2.4.22, the semantic of `signal` can follow either the BSD or System V semantics, depending on the version of the C library and how you compile your application.

Because the semantics of `signal` differ among implementations, it is better to use the `sigaction` function instead. When we describe the `sigaction` function in Section 10.14, we provide an implementation of `signal` that uses it. All the examples in this text use the `signal` function that we show in Figure 10.18.

The *signo* argument is just the name of the signal from Figure 10.1. The value of *func* is (a) the constant SIG_IGN, (b) the constant SIG_DFL, or (c) the address of a function to be called when the signal occurs. If we specify SIG_IGN, we are telling the system to ignore the signal. (Remember that we cannot ignore the two signals SIGKILL and SIGSTOP.) When we specify SIG_DFL, we are setting the action associated with the signal to its default value (see the final column in Figure 10.1). When we specify the address of a function to be called when the signal occurs, we are arranging to "catch" the signal. We call the function either the *signal handler* or the *signal-catching function*.

The prototype for the signal function states that the function requires two arguments and returns a pointer to a function that returns nothing (void). The signal function's first argument, *signo*, is an integer. The second argument is a pointer to a function that takes a single integer argument and returns nothing. The function whose address is returned as the value of signal takes a single integer argument (the final (int)). In plain English, this declaration says that the signal handler is passed a single integer argument (the signal number) and that it returns nothing. When we call signal to establish the signal handler, the second argument is a pointer to the function. The return value from signal is the pointer to the previous signal handler.

Many systems call the signal handler with additional, implementation-dependent arguments. We discuss this further in Section 10.14.

The perplexing signal function prototype shown at the beginning of this section can be made much simpler through the use of the following typedef [Plauger 1992]:

```
typedef void Sigfunc(int);
```

Then the prototype becomes

```
Sigfunc *signal(int, Sigfunc *);
```

We've included this typedef in apue.h (Appendix B) and use it with the functions in this chapter.

If we examine the system's header <signal.h>, we probably find declarations of the form

```
#define SIG_ERR    (void (*)())-1
#define SIG_DFL    (void (*)())0
#define SIG_IGN    (void (*)())1
```

These constants can be used in place of the "pointer to a function that takes an integer argument and returns nothing," the second argument to signal, and the return value from signal. The three values used for these constants need not be –1, 0, and 1. They must be three values that can never be the address of any declarable function. Most UNIX systems use the values shown.

Example

Figure 10.2 shows a simple signal handler that catches either of the two user-defined signals and prints the signal number. In Section 10.10, we describe the pause function, which simply suspends the calling process until a signal is received.

```
#include "apue.h"

static void sig_usr(int);   /* one handler for both signals */

int
main(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("can't catch SIGUSR1");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("can't catch SIGUSR2");
    for ( ; ; )
        pause();
}

static void
sig_usr(int signo)      /* argument is signal number */
{
    if (signo == SIGUSR1)
        printf("received SIGUSR1\n");
    else if (signo == SIGUSR2)
        printf("received SIGUSR2\n");
    else
        err_dump("received signal %d\n", signo);
}
```

Figure 10.2 Simple program to catch SIGUSR1 and SIGUSR2

We invoke the program in the background and use the kill(1) command to send it signals. Note that the term *kill* in the UNIX System is a misnomer. The kill(1) command and the kill(2) function just send a signal to a process or process group. Whether or not that signal terminates the process depends on which signal is sent and whether the process has arranged to catch the signal.

```
$ ./a.out &                          start process in background
[1]      7216                        job-control shell prints job number and process ID
$ kill -USR1 7216                    send it SIGUSR1
received SIGUSR1
$ kill -USR2 7216                    send it SIGUSR2
received SIGUSR2
$ kill 7216                          now send it SIGTERM
[1]+  Terminated          ./a.out
```

When we send the SIGTERM signal, the process is terminated, since it doesn't catch the signal, and the default action for the signal is termination. □

Program Start-Up

When a program is executed, the status of all signals is either default or ignore. Normally, all signals are set to their default action, unless the process that calls exec is ignoring the signal. Specifically, the exec functions change the disposition of any signals being caught to their default action and leave the status of all other signals alone. (Naturally, a signal that is being caught by a process that calls exec cannot be caught by the same function in the new program, since the address of the signal-catching function in the caller probably has no meaning in the new program file that is executed.)

One specific example is how an interactive shell treats the interrupt and quit signals for a background process. With a shell that doesn't support job control, when we execute a process in the background, as in

```
cc main.c &
```

the shell automatically sets the disposition of the interrupt and quit signals in the background process to be ignored. This is so that if we type the interrupt character, it doesn't affect the background process. If this weren't done and we typed the interrupt character, it would terminate not only the foreground process, but also all the background processes.

Many interactive programs that catch these two signals have code that looks like

```
void sig_int(int), sig_quit(int);

if (signal(SIGINT, SIG_IGN) != SIG_IGN)
    signal(SIGINT, sig_int);
if (signal(SIGQUIT, SIG_IGN) != SIG_IGN)
    signal(SIGQUIT, sig_quit);
```

Doing this, the process catches the signal only if the signal is not currently being ignored.

These two calls to signal also show a limitation of the signal function: we are not able to determine the current disposition of a signal without changing the disposition. We'll see later in this chapter how the sigaction function allows us to determine a signal's disposition without changing it.

Process Creation

When a process calls fork, the child inherits the parent's signal dispositions. Here, since the child starts off with a copy of the parent's memory image, the address of a signal-catching function has meaning in the child.

10.4 Unreliable Signals

In earlier versions of the UNIX System (such as Version 7), signals were unreliable. By this we mean that signals could get lost: a signal could occur and the process would never know about it. Also, a process had little control over a signal: a process could catch the signal or ignore it. Sometimes, we would like to tell the kernel to block a signal: don't ignore it, just remember if it occurs, and tell us later when we're ready.

Changes were made with 4.2BSD to provide what are called *reliable signals*. A different set of changes was then made in SVR3 to provide reliable signals under System V. POSIX.1 chose the BSD model to standardize.

One problem with these early versions is that the action for a signal was reset to its default each time the signal occurred. (In the previous example, when we ran the program in Figure 10.2, we avoided this detail by catching each signal only once.) The classic example from programming books that described these earlier systems concerns how to handle the interrupt signal. The code that was described usually looked like

```
    int    sig_int();          /* my signal handling function */

    ...
    signal(SIGINT, sig_int); /* establish handler */
    ...

sig_int()
{
    signal(SIGINT, sig_int); /* reestablish handler for next time */
    ...                      /* process the signal ... */
}
```

(The reason the signal handler is declared as returning an integer is that these early systems didn't support the ISO C `void` data type.)

The problem with this code fragment is that there is a window of time—after the signal has occurred, but before the call to `signal` in the signal handler—when the interrupt signal could occur another time. This second signal would cause the default action to occur, which for this signal terminates the process. This is one of those conditions that works correctly most of the time, causing us to think that it is correct, when it isn't.

Another problem with these earlier systems is that the process was unable to turn a signal off when it didn't want the signal to occur. All the process could do was ignore the signal. There are times when we would like to tell the system "prevent the following signals from occurring, but remember if they do occur." The classic example that demonstrates this flaw is shown by a piece of code that catches a signal and sets a flag for the process that indicates that the signal occurred:

```
int    sig_int_flag;           /* set nonzero when signal occurs */

main()
{
    int    sig_int();          /* my signal handling function */
    ...
    signal(SIGINT, sig_int); /* establish handler */
    ...
    while (sig_int_flag == 0)
        pause();               /* go to sleep, waiting for signal */
    ...
}

sig_int()
{
    signal(SIGINT, sig_int); /* reestablish handler for next time */
    sig_int_flag = 1;          /* set flag for main loop to examine */
}
```

Here, the process is calling the pause function to put it to sleep until a signal is caught. When the signal is caught, the signal handler just sets the flag sig_int_flag to a nonzero value. The process is automatically awakened by the kernel after the signal handler returns, notices that the flag is nonzero, and does whatever it needs to do. But there is a window of time when things can go wrong. If the signal occurs after the test of sig_int_flag, but before the call to pause, the process could go to sleep forever (assuming that the signal is never generated again). This occurrence of the signal is lost. This is another example of some code that isn't right, yet it works most of the time. Debugging this type of problem can be difficult.

10.5 Interrupted System Calls

A characteristic of earlier UNIX systems is that if a process caught a signal while the process was blocked in a "slow" system call, the system call was interrupted. The system call returned an error and errno was set to EINTR. This was done under the assumption that since a signal occurred and the process caught it, there is a good chance that something has happened that should wake up the blocked system call.

Here, we have to differentiate between a system call and a function. It is a system call within the kernel that is interrupted when a signal is caught.

To support this feature, the system calls are divided into two categories: the "slow" system calls and all the others. The slow system calls are those that can block forever. Included in this category are

- Reads that can block the caller forever if data isn't present with certain file types (pipes, terminal devices, and network devices)
- Writes that can block the caller forever if the data can't be accepted immediately by these same file types
- Opens that block until some condition occurs on certain file types (such as an open of a terminal device that waits until an attached modem answers the phone)
- The pause function (which by definition puts the calling process to sleep until a signal is caught) and the wait function
- Certain ioctl operations
- Some of the interprocess communication functions (Chapter 15)

The notable exception to these slow system calls is anything related to disk I/O. Although a read or a write of a disk file can block the caller temporarily (while the disk driver queues the request and then the request is executed), unless a hardware error occurs, the I/O operation always returns and unblocks the caller quickly.

One condition that is handled by interrupted system calls, for example, is when a process initiates a read from a terminal device and the user at the terminal walks away from the terminal for an extended period. In this example, the process could be blocked for hours or days and would remain so unless the system was taken down.

POSIX.1 semantics for interrupted `reads` and `writes` changed with the 2001 version of the standard. Earlier versions gave implementations a choice for how to deal with `reads` and `writes` that have processed partial amounts of data. If `read` has received and transferred data to an application's buffer, but has not yet received all that the application requested and is then interrupted, the operating system could either fail the system call with `errno` set to `EINTR` or allow the system call to succeed, returning the partial amount of data received. Similarly, if `write` is interrupted after transferring some of the data in an application's buffer, the operation system could either fail the system call with `errno` set to `EINTR` or allow the system call to succeed, returning the partial amount of data written. Historically, implementations derived from System V fail the system call, whereas BSD-derived implementations return partial success. With the 2001 version of the POSIX.1 standard, the BSD-style semantics are required.

The problem with interrupted system calls is that we now have to handle the error return explicitly. The typical code sequence (assuming a read operation and assuming that we want to restart the read even if it's interrupted) would be

```
again:
    if ((n = read(fd, buf, BUFFSIZE)) < 0) {
        if (errno == EINTR)
            goto again;      /* just an interrupted system call */
        /* handle other errors */
    }
```

To prevent applications from having to handle interrupted system calls, 4.2BSD introduced the automatic restarting of certain interrupted system calls. The system calls that were automatically restarted are `ioctl`, `read`, `readv`, `write`, `writev`, `wait`, and `waitpid`. As we've mentioned, the first five of these functions are interrupted by a signal only if they are operating on a slow device; `wait` and `waitpid` are always interrupted when a signal is caught. Since this caused a problem for some applications that didn't want the operation restarted if it was interrupted, 4.3BSD allowed the process to disable this feature on a per signal basis.

POSIX.1 allows an implementation to restart system calls, but it is not required. The Single UNIX Specification defines the `SA_RESTART` flag as an XSI extension to `sigaction` to allow applications to request that interrupted system calls be restarted.

System V has never restarted system calls by default. BSD, on the other hand, restarts them if interrupted by signals. By default, FreeBSD 5.2.1, Linux 2.4.22, and Mac OS X 10.3 restart system calls interrupted by signals. The default on Solaris 9, however, is to return an error (`EINTR`) instead.

One of the reasons 4.2BSD introduced the automatic restart feature is that sometimes we don't know that the input or output device is a slow device. If the program we write can be used interactively, then it might be reading or writing a slow device, since terminals fall into this category. If we catch signals in this program, and if the system doesn't provide the restart capability, then we have to test every read or write for the interrupted error return and reissue the read or write.

Figure 10.3 summarizes the signal functions and their semantics provided by the various implementations.

We don't discuss the older `sigset` and `sigvec` functions. Their use has been superceded by the `sigaction` function; they are included only for completeness. In contrast, some implementations promote the `signal` function as a simplified interface to `sigaction`.

Functions	System	Signal handler remains installed	Ability to block signals	Automatic restart of interrupted system calls?
signal	ISO C, POSIX.1	unspecified	unspecified	unspecified
	V7, SVR2, SVR3, SVR4, Solaris			never
	4.2BSD	•	•	always
	4.3BSD, 4.4BSD, FreeBSD, Linux, Mac OS X	•	•	default
sigset	XSI	•	•	unspecified
	SVR3, SVR4, Linux, Solaris	•	•	never
sigvec	4.2BSD	•	•	always
	4.3BSD, 4.4BSD, FreeBSD, Mac OS X	•	•	default
sigaction	POSIX.1	•	•	unspecified
	XSI, 4.4BSD, SVR4, FreeBSD, Mac OS X, Linux, Solaris	•	•	optional

Figure 10.3 Features provided by various signal implementations

Be aware that UNIX systems from other vendors can have values different from those shown in this figure. For example, `sigaction` under SunOS 4.1.2 restarts an interrupted system call by default, different from the platforms listed in Figure 10.3.

In Figure 10.18, we provide our own version of the `signal` function that automatically tries to restart interrupted system calls (other than for the `SIGALRM` signal). In Figure 10.19, we provide another function, `signal_intr`, that tries to never do the restart.

We talk more about interrupted system calls in Section 14.5 with regard to the `select` and `poll` functions.

10.6 Reentrant Functions

When a signal that is being caught is handled by a process, the normal sequence of instructions being executed by the process is temporarily interrupted by the signal handler. The process then continues executing, but the instructions in the signal handler are now executed. If the signal handler returns (instead of calling `exit` or `longjmp`, for example), then the normal sequence of instructions that the process was executing when the signal was caught continues executing. (This is similar to what happens when a hardware interrupt occurs.) But in the signal handler, we can't tell where the process was executing when the signal was caught. What if the process was in the middle of allocating additional memory on its heap using `malloc`, and we call malloc from the signal handler? Or, what if the process was in the middle of a call to a function, such as getpwnam (Section 6.2), that stores its result in a static location, and we call the same function from the signal handler? In the malloc example, havoc can result for the process, since malloc usually maintains a linked list of all its allocated areas, and it may have been in the middle of changing this list. In the case of getpwnam, the information returned to the normal caller can get overwritten with the information returned to the signal handler.

The Single UNIX Specification specifies the functions that are guaranteed to be reentrant. Figure 10.4 lists these reentrant functions.

accept	fchmod	lseek	sendto	stat
access	fchown	lstat	setgid	symlink
aio_error	fcntl	mkdir	setpgid	sysconf
aio_return	fdatasync	mkfifo	setsid	tcdrain
aio_suspend	fork	open	setsockopt	tcflow
alarm	fpathconf	pathconf	setuid	tcflush
bind	fstat	pause	shutdown	tcgetattr
cfgetispeed	fsync	pipe	sigaction	tcgetpgrp
cfgetospeed	ftruncate	poll	sigaddset	tcsendbreak
cfsetispeed	getegid	posix_trace_event	sigdelset	tcsetattr
cfsetospeed	geteuid	pselect	sigemptyset	tcsetpgrp
chdir	getgid	raise	sigfillset	time
chmod	getgroups	read	sigismember	timer_getoverrun
chown	getpeername	readlink	signal	timer_gettime
clock_gettime	getpgrp	recv	sigpause	timer_settime
close	getpid	recvfrom	sigpending	times
connect	getppid	recvmsg	sigprocmask	umask
creat	getsockname	rename	sigqueue	uname
dup	getsockopt	rmdir	sigset	unlink
dup2	getuid	select	sigsuspend	utime
execle	kill	sem_post	sleep	wait
execve	link	send	socket	waitpid
_Exit & _exit	listen	sendmsg	socketpair	write

Figure 10.4 Reentrant functions that may be called from a signal handler

Most functions that are not in Figure 10.4 are missing because (a) they are known to use static data structures, (b) they call malloc or free, or (c) they are part of the standard I/O library. Most implementations of the standard I/O library use global data structures in a nonreentrant way. Note that even though we call printf from signal handlers in some of our examples, it is not guaranteed to produce the expected results, since the signal hander can interrupt a call to printf from our main program.

Be aware that even if we call a function listed in Figure 10.4 from a signal handler, there is only one errno variable per thread (recall the discussion of errno and threads in Section 1.7), and we might modify its value. Consider a signal handler that is invoked right after main has set errno. If the signal handler calls read, for example, this call can change the value of errno, wiping out the value that was just stored in main. Therefore, as a general rule, when calling the functions listed in Figure 10.4 from a signal handler, we should save and restore errno. (Be aware that a commonly caught signal is SIGCHLD, and its signal handler usually calls one of the wait functions. All the wait functions can change errno.)

Note that longjmp (Section 7.10) and siglongjmp (Section 10.15) are missing from Figure 10.4, because the signal may have occurred while the main routine was updating a data structure in a nonreentrant way. This data structure could be left half updated if we call siglongjmp instead of returning from the signal handler. If it is going to do such things as update global data structures, as we describe here, while catching signals that cause sigsetjmp to be executed, an application needs to block the signals while updating the data structures.

Example

Figure 10.5 shows a program that calls the nonreentrant function getpwnam from a signal handler that is called every second. We describe the alarm function in Section 10.10. We use it here to generate a SIGALRM signal every second.

```
#include "apue.h"
#include <pwd.h>

static void
my_alarm(int signo)
{
    struct passwd   *rootptr;

    printf("in signal handler\n");
    if ((rootptr = getpwnam("root")) == NULL)
            err_sys("getpwnam(root) error");
    alarm(1);
}

int
main(void)
{
    struct passwd   *ptr;

    signal(SIGALRM, my_alarm);
    alarm(1);
    for ( ; ; ) {
        if ((ptr = getpwnam("sar")) == NULL)
            err_sys("getpwnam error");
        if (strcmp(ptr->pw_name, "sar") != 0)
            printf("return value corrupted!, pw_name = %s\n",
                    ptr->pw_name);
    }
}
```

Figure 10.5 Call a nonreentrant function from a signal handler

When this program was run, the results were random. Usually, the program would be terminated by a SIGSEGV signal when the signal handler returned after several iterations. An examination of the core file showed that the main function had called getpwnam, but that some internal pointers had been corrupted when the signal handler called the same function. Occasionally, the program would run for several seconds before crashing with a SIGSEGV error. When the main function did run correctly after the signal had been caught, the return value was sometimes corrupted and sometimes fine. Once (on Mac OS X), messages were printed from the malloc library routine warning about freeing pointers not allocated through malloc.

As shown by this example, if we call a nonreentrant function from a signal handler, the results are unpredictable. □

10.7 SIGCLD Semantics

Two signals that continually generate confusion are SIGCLD and SIGCHLD. First, SIGCLD (without the H) is the System V name, and this signal has different semantics from the BSD signal, named SIGCHLD. The POSIX.1 signal is also named SIGCHLD.

The semantics of the BSD SIGCHLD signal are normal, in that its semantics are similar to those of all other signals. When the signal occurs, the status of a child has changed, and we need to call one of the wait functions to determine what has happened.

System V, however, has traditionally handled the SIGCLD signal differently from other signals. SVR4-based systems continue this questionable tradition (i.e., compatibility constraint) if we set its disposition using either signal or sigset (the older, SVR3-compatible functions to set the disposition of a signal). This older handling of SIGCLD consists of the following.

1. If the process specifically sets its disposition to SIG_IGN, children of the calling process will not generate zombie processes. Note that this is different from its default action (SIG_DFL), which from Figure 10.1 is to be ignored. Instead, on termination, the status of these child processes is discarded. If it subsequently calls one of the wait functions, the calling process will block until all its children have terminated, and then wait returns −1 with errno set to ECHILD. (The default disposition of this signal is to be ignored, but this default will not cause the preceding semantics to occur. Instead, we specifically have to set its disposition to SIG_IGN.)

    POSIX.1 does not specify what happens when SIGCHLD is ignored, so this behavior is allowed. The Single UNIX Specification includes an XSI extension specifying that this behavior be supported for SIGCHLD.

    4.4BSD always generates zombies if SIGCHLD is ignored. If we want to avoid zombies, we have to wait for our children. FreeBSD 5.2.1 works like 4.4BSD. Mac OS X 10.3, however, doesn't create zombies when SIGCHLD is ignored.

    With SVR4, if either signal or sigset is called to set the disposition of SIGCHLD to be ignored, zombies are never generated. Solaris 9 and Linux 2.4.22 follow SVR4 in this behavior.

    With sigaction, we can set the SA_NOCLDWAIT flag (Figure 10.16) to avoid zombies. This action is supported on all four platforms: FreeBSD 5.2.1, Linux 2.4.22, Mac OS X 10.3, and Solaris 9.

2. If we set the disposition of SIGCLD to be caught, the kernel immediately checks whether any child processes are ready to be waited for and, if so, calls the SIGCLD handler.

Item 2 changes the way we have to write a signal handler for this signal, as illustrated in the following example.

Example

Recall from Section 10.4 that the first thing to do on entry to a signal handler is to call signal again, to reestablish the handler. (This action was to minimize the window of time when the signal is reset back to its default and could get lost.) We show this in Figure 10.6. This program doesn't work on some platforms. If we compile and run it under a traditional System V platform, such as OpenServer 5 or UnixWare 7, the output

```
#include    "apue.h"
#include    <sys/wait.h>

static void sig_cld(int);

int
main()
{
    pid_t   pid;

    if (signal(SIGCLD, sig_cld) == SIG_ERR)
        perror("signal error");
    if ((pid = fork()) < 0) {
        perror("fork error");
    } else if (pid == 0) {      /* child */
        sleep(2);
        _exit(0);
    }
    pause();    /* parent */
    exit(0);
}

static void
sig_cld(int signo)  /* interrupts pause() */
{
    pid_t   pid;
    int     status;

    printf("SIGCLD received\n");
    if (signal(SIGCLD, sig_cld) == SIG_ERR) /* reestablish handler */
        perror("signal error");
    if ((pid = wait(&status)) < 0)      /* fetch child status */
        perror("wait error");
    printf("pid = %d\n", pid);
}
```

Figure 10.6 System V SIGCLD handler that doesn't work

is a continual string of `SIGCLD received` lines. Eventually, the process runs out of stack space and terminates abnormally.

> FreeBSD 5.2.1 and Mac OS X 10.3 don't exhibit this problem, because BSD-based systems generally don't support historic System V semantics for `SIGCLD`. Linux 2.4.22 also doesn't exhibit this problem, because it doesn't call the `SIGCHLD` signal handler when a process arranges to catch `SIGCHLD` and child processes are ready to be `wait`ed for, even though `SIGCLD` and `SIGCHLD` are defined to be the same value. Solaris 9, on the other hand, does call the signal handler in this situation, but includes extra code in the kernel to avoid this problem.
>
> Although the four platforms described in this book solve this problem, realize that platforms (such as UnixWare) still exist that haven't addressed it.

The problem with this program is that the call to `signal` at the beginning of the signal handler invokes item 2 from the preceding discussion—the kernel checks whether a child needs to be `wait`ed for (which there is, since we're processing a `SIGCLD` signal), so it generates another call to the signal handler. The signal handler calls `signal`, and the whole process starts over again.

To fix this program, we have to move the call to `signal` after the call to `wait`. By doing this, we call `signal` after fetching the child's termination status; the signal is generated again by the kernel only if some other child has since terminated.

> POSIX.1 states that when we establish a signal handler for `SIGCHLD` and there exists a terminated child we have not yet `wait`ed for, it is unspecified whether the signal is generated. This allows the behavior described previously. But since POSIX.1 does not reset a signal's disposition to its default when the signal occurs (assuming that we're using the POSIX.1 `sigaction` function to set its disposition), there is no need for us to ever establish a signal handler for `SIGCHLD` within that handler.

□

Be cognizant of the `SIGCHLD` semantics for your implementation. Be especially aware of some systems that `#define SIGCHLD` to be `SIGCLD` or vice versa. Changing the name may allow you to compile a program that was written for another system, but if that program depends on the other semantics, it may not work.

> On the four platforms described in this text, `SIGCLD` is equivalent to `SIGCHLD`.

10.8 Reliable-Signal Terminology and Semantics

We need to define some of the terms used throughout our discussion of signals. First, a signal is *generated* for a process (or sent to a process) when the event that causes the signal occurs. The event could be a hardware exception (e.g., divide by 0), a software condition (e.g., an `alarm` timer expiring), a terminal-generated signal, or a call to the `kill` function. When the signal is generated, the kernel usually sets a flag of some form in the process table.

We say that a signal is *delivered* to a process when the action for a signal is taken. During the time between the generation of a signal and its delivery, the signal is said to be *pending*.

A process has the option of *blocking* the delivery of a signal. If a signal that is blocked is generated for a process, and if the action for that signal is either the default action or to catch the signal, then the signal remains pending for the process until the process either (a) unblocks the signal or (b) changes the action to ignore the signal. The system determines what to do with a blocked signal when the signal is delivered, not when it's generated. This allows the process to change the action for the signal before it's delivered. The `sigpending` function (Section 10.13) can be called by a process to determine which signals are blocked and pending.

What happens if a blocked signal is generated more than once before the process unblocks the signal? POSIX.1 allows the system to deliver the signal either once or more than once. If the system delivers the signal more than once, we say that the signals are queued. Most UNIX systems, however, do *not* queue signals unless they support the real-time extensions to POSIX.1. Instead, the UNIX kernel simply delivers the signal once.

> The manual pages for SVR2 claimed that the `SIGCLD` signal was queued while the process was executing its `SIGCLD` signal handler. Although this might have been true on a conceptual level, the actual implementation was different. Instead, the signal was regenerated by the kernel as we described in Section 10.7. In SVR3, the manual was changed to indicate that the `SIGCLD` signal was ignored while the process was executing its signal handler for `SIGCLD`. The SVR4 manual removed any mention of what happens to `SIGCLD` signals that are generated while a process is executing its `SIGCLD` signal handler.

> The SVR4 `sigaction(2)` manual page in AT&T [1990e] claims that the `SA_SIGINFO` flag (Figure 10.16) causes signals to be reliably queued. This is wrong. Apparently, this feature was partially implemented within the kernel, but it is not enabled in SVR4. Curiously, the SVID doesn't make the same claims of reliable queuing.

What happens if more than one signal is ready to be delivered to a process? POSIX.1 does not specify the order in which the signals are delivered to the process. The Rationale for POSIX.1 does suggest, however, that signals related to the current state of the process be delivered before other signals. (`SIGSEGV` is one such signal.)

Each process has a *signal mask* that defines the set of signals currently blocked from delivery to that process. We can think of this mask as having one bit for each possible signal. If the bit is on for a given signal, that signal is currently blocked. A process can examine and change its current signal mask by calling `sigprocmask`, which we describe in Section 10.12.

Since it is possible for the number of signals to exceed the number of bits in an integer, POSIX.1 defines a data type, called `sigset_t`, that holds a *signal set*. The signal mask, for example, is stored in one of these signal sets. We describe five functions that operate on signal sets in Section 10.11.

10.9 `kill` and `raise` Functions

The `kill` function sends a signal to a process or a group of processes. The `raise` function allows a process to send a signal to itself.

> `raise` was originally defined by ISO C. POSIX.1 includes it to align itself with the ISO C standard, but POSIX.1 extends the specification of `raise` to deal with threads (we discuss how threads interact with signals in Section 12.8). Since ISO C does not deal with multiple processes, it could not define a function, such as `kill`, that requires a process ID argument.

```
#include <signal.h>

int kill(pid_t pid, int signo);

int raise(int signo);

                                        Both return: 0 if OK, -1 on error
```

The call

```
raise(signo);
```

is equivalent to the call

```
kill(getpid(), signo);
```

There are four different conditions for the *pid* argument to `kill`.

pid > 0	The signal is sent to the process whose process ID is *pid*.
pid == 0	The signal is sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send the signal. Note that the term *all processes* excludes an implementation-defined set of system processes. For most UNIX systems, this set of system processes includes the kernel processes and `init` (pid 1).
pid < 0	The signal is sent to all processes whose process group ID equals the absolute value of *pid* and for which the sender has permission to send the signal. Again, the set of all processes excludes certain system processes, as described earlier.
pid == −1	The signal is sent to all processes on the system for which the sender has permission to send the signal. As before, the set of processes excludes certain system processes.

As we've mentioned, a process needs permission to send a signal to another process. The superuser can send a signal to any process. For other users, the basic rule is that the real or effective user ID of the sender has to equal the real or effective user ID of the receiver. If the implementation supports `_POSIX_SAVED_IDS` (as POSIX.1 now requires), the saved set-user-ID of the receiver is checked instead of its effective user ID. There is also one special case for the permission testing: if the signal being sent is `SIGCONT`, a process can send it to any other process in the same session.

POSIX.1 defines signal number 0 as the null signal. If the *signo* argument is 0, then the normal error checking is performed by `kill`, but no signal is sent. This is often used to determine if a specific process still exists. If we send the process the null signal and it doesn't exist, `kill` returns −1 and `errno` is set to `ESRCH`. Be aware, however, that UNIX systems recycle process IDs after some amount of time, so the existence of a process with a given process ID does not mean that it's the process that you think it is.

Also understand that the test for process existence is not atomic. By the time that `kill` returns the answer to the caller, the process in question might have exited, so the answer is of limited value.

If the call to kill causes the signal to be generated for the calling process and if the signal is not blocked, either *signo* or some other pending, unblocked signal is delivered to the process before kill returns. (Additional conditions occur with threads; see Section 12.8 for more information.)

10.10 alarm and pause Functions

The alarm function allows us to set a timer that will expire at a specified time in the future. When the timer expires, the SIGALRM signal is generated. If we ignore or don't catch this signal, its default action is to terminate the process.

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);

                    Returns: 0 or number of seconds until previously set alarm
```

The *seconds* value is the number of clock seconds in the future when the signal should be generated. Be aware that when that time occurs, the signal is generated by the kernel, but there could be additional time before the process gets control to handle the signal, because of processor scheduling delays.

Earlier UNIX System implementations warned that the signal could also be sent up to 1 second early. POSIX.1 does not allow this.

There is only one of these alarm clocks per process. If, when we call alarm, a previously registered alarm clock for the process has not yet expired, the number of seconds left for that alarm clock is returned as the value of this function. That previously registered alarm clock is replaced by the new value.

If a previously registered alarm clock for the process has not yet expired and if the *seconds* value is 0, the previous alarm clock is canceled. The number of seconds left for that previous alarm clock is still returned as the value of the function.

Although the default action for SIGALRM is to terminate the process, most processes that use an alarm clock catch this signal. If the process then wants to terminate, it can perform whatever cleanup is required before terminating. If we intend to catch SIGALRM, we need to be careful to install its signal handler before calling alarm. If we call alarm first and are sent SIGALRM before we can install the signal handler, our process will terminate.

The pause function suspends the calling process until a signal is caught.

```
#include <unistd.h>

int pause(void);

                                        Returns: -1 with errno set to EINTR
```

The only time pause returns is if a signal handler is executed and that handler returns. In that case, pause returns −1 with errno set to EINTR.

Example

Using alarm and pause, we can put a process to sleep for a specified amount of time. The sleep1 function in Figure 10.7 appears to do this (but it has problems, as we shall see shortly).

```
#include    <signal.h>
#include    <unistd.h>

static void
sig_alrm(int signo)
{
    /* nothing to do, just return to wake up the pause */
}

unsigned int
sleep1(unsigned int nsecs)
{
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        return(nsecs);
    alarm(nsecs);         /* start the timer */
    pause();              /* next caught signal wakes us up */
    return(alarm(0));     /* turn off timer, return unslept time */
}
```

Figure 10.7 Simple, incomplete implementation of sleep

This function looks like the sleep function, which we describe in Section 10.19, but this simple implementation has three problems.

1. If the caller already has an alarm set, that alarm is erased by the first call to alarm. We can correct this by looking at the return value from the first call to alarm. If the number of seconds until some previously set alarm is less than the argument, then we should wait only until the previously set alarm expires. If the previously set alarm will go off after ours, then before returning we should reset this alarm to occur at its designated time in the future.

2. We have modified the disposition for SIGALRM. If we're writing a function for others to call, we should save the disposition when we're called and restore it when we're done. We can correct this by saving the return value from signal and resetting the disposition before we return.

3. There is a race condition between the first call to alarm and the call to pause. On a busy system, it's possible for the alarm to go off and the signal handler to be called before we call pause. If that happens, the caller is suspended forever in the call to pause (assuming that some other signal isn't caught).

Earlier implementations of sleep looked like our program, with problems 1 and 2 corrected as described. There are two ways to correct problem 3. The first uses setjmp, which we show in the next example. The other uses sigprocmask and sigsuspend, and we describe it in Section 10.19. □

Example

The SVR2 implementation of sleep used setjmp and longjmp (Section 7.10) to avoid the race condition described in problem 3 of the previous example. A simple version of this function, called sleep2, is shown in Figure 10.8. (To reduce the size of this example, we don't handle problems 1 and 2 described earlier.)

```
#include    <setjmp.h>
#include    <signal.h>
#include    <unistd.h>

static jmp_buf  env_alrm;

static void
sig_alrm(int signo)
{
    longjmp(env_alrm, 1);
}

unsigned int
sleep2(unsigned int nsecs)
{
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        return(nsecs);
    if (setjmp(env_alrm) == 0) {
        alarm(nsecs);       /* start the timer */
        pause();            /* next caught signal wakes us up */
    }
    return(alarm(0));       /* turn off timer, return unslept time */
}
```

Figure 10.8 Another (imperfect) implementation of sleep

The sleep2 function avoids the race condition from Figure 10.7. Even if the pause is never executed, the sleep2 function returns when the SIGALRM occurs.

There is, however, another subtle problem with the sleep2 function that involves its interaction with other signals. If the SIGALRM interrupts some other signal handler, when we call longjmp, we abort the other signal handler. Figure 10.9 shows this scenario. The loop in the SIGINT handler was written so that it executes for longer than 5 seconds on one of the systems used by the author. We simply want it to execute longer than the argument to sleep2. The integer k is declared volatile to prevent an optimizing compiler from discarding the loop. Executing the program shown in Figure 10.9 gives us

```
$ ./a.out
^?                          we type the interrupt character
sig_int starting
sleep2 returned: 0
```

We can see that the longjmp from the sleep2 function aborted the other signal handler, sig_int, even though it wasn't finished. This is what you'll encounter if you mix the SVR2 sleep function with other signal handling. See Exercise 10.3. □

```
#include "apue.h"

unsigned int    sleep2(unsigned int);
static void     sig_int(int);

int
main(void)
{
    unsigned int    unslept;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    unslept = sleep2(5);
    printf("sleep2 returned: %u\n", unslept);
    exit(0);
}

static void
sig_int(int signo)
{
    int             i, j;
    volatile int    k;

    /*
     * Tune these loops to run for more than 5 seconds
     * on whatever system this test program is run.
     */
    printf("\nsig_int starting\n");
    for (i = 0; i < 300000; i++)
        for (j = 0; j < 4000; j++)
            k += i * j;
    printf("sig_int finished\n");
}
```

Figure 10.9 Calling `sleep2` from a program that catches other signals

The purpose of these two examples, the `sleep1` and `sleep2` functions, is to show the pitfalls in dealing naively with signals. The following sections will show ways around all these problems, so we can handle signals reliably, without interfering with other pieces of code.

Example

A common use for `alarm`, in addition to implementing the `sleep` function, is to put an upper time limit on operations that can block. For example, if we have a `read` operation on a device that can block (a "slow" device, as described in Section 10.5), we might want the `read` to time out after some amount of time. The program in Figure 10.10 does this, reading one line from standard input and writing it to standard output.

```
#include "apue.h"

static void sig_alrm(int);

int
main(void)
{
    int     n;
    char    line[MAXLINE];

    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("signal(SIGALRM) error");

    alarm(10);
    if ((n = read(STDIN_FILENO, line, MAXLINE)) < 0)
        err_sys("read error");
    alarm(0);

    write(STDOUT_FILENO, line, n);
    exit(0);
}

static void
sig_alrm(int signo)
{
    /* nothing to do, just return to interrupt the read */
}
```

Figure 10.10 Calling read with a timeout

This sequence of code is common in UNIX applications, but this program has two problems.

1. The program in Figure 10.10 has one of the same flaws that we described in Figure 10.7: a race condition between the first call to `alarm` and the call to `read`. If the kernel blocks the process between these two function calls for longer than the alarm period, the `read` could block forever. Most operations of this type use a long alarm period, such as a minute or more, making this unlikely; nevertheless, it is a race condition.
2. If system calls are automatically restarted, the `read` is not interrupted when the `SIGALRM` signal handler returns. In this case, the timeout does nothing.

Here, we specifically want a slow system call to be interrupted. POSIX.1 does not give us a portable way to do this; however, the XSI extension in the Single UNIX Specification does. We'll discuss this more in Section 10.14. □

Example

Let's redo the preceding example using `longjmp`. This way, we don't need to worry about whether a slow system call is interrupted.

```
#include "apue.h"
#include <setjmp.h>

static void     sig_alrm(int);
static jmp_buf  env_alrm;

int
main(void)
{
    int     n;
    char    line[MAXLINE];

    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("signal(SIGALRM) error");
    if (setjmp(env_alrm) != 0)
        err_quit("read timeout");

    alarm(10);
    if ((n = read(STDIN_FILENO, line, MAXLINE)) < 0)
        err_sys("read error");
    alarm(0);

    write(STDOUT_FILENO, line, n);
    exit(0);
}

static void
sig_alrm(int signo)
{
    longjmp(env_alrm, 1);
}
```

Figure 10.11 Calling `read` with a timeout, using `longjmp`

This version works as expected, regardless of whether the system restarts interrupted system calls. Realize, however, that we still have the problem of interactions with other signal handlers, as in Figure 10.8. □

If we want to set a time limit on an I/O operation, we need to use `longjmp`, as shown previously, realizing its possible interaction with other signal handlers. Another option is to use the `select` or `poll` functions, described in Sections 14.5.1 and 14.5.2.

10.11 Signal Sets

We need a data type to represent multiple signals—a *signal set*. We'll use this with such functions as `sigprocmask` (in the next section) to tell the kernel not to allow any of the signals in the set to occur. As we mentioned earlier, the number of different signals can exceed the number of bits in an integer, so in general, we can't use an integer to represent the set with one bit per signal. POSIX.1 defines the data type `sigset_t` to contain a signal set and the following five functions to manipulate signal sets.

```
#include <signal.h>

int sigemptyset(sigset_t *set);

int sigfillset(sigset_t *set);

int sigaddset(sigset_t *set, int signo);

int sigdelset(sigset_t *set, int signo);

                              All four return: 0 if OK, -1 on error

int sigismember(const sigset_t *set, int signo);

                        Returns: 1 if true, 0 if false, -1 on error
```

The function `sigemptyset` initializes the signal set pointed to by *set* so that all signals are excluded. The function `sigfillset` initializes the signal set so that all signals are included. All applications have to call either `sigemptyset` or `sigfillset` once for each signal set, before using the signal set, because we cannot assume that the C initialization for external and static variables (0) corresponds to the implementation of signal sets on a given system.

Once we have initialized a signal set, we can add and delete specific signals in the set. The function `sigaddset` adds a single signal to an existing set, and `sigdelset` removes a single signal from a set. In all the functions that take a signal set as an argument, we always pass the address of the signal set as the argument.

Implementation

If the implementation has fewer signals than bits in an integer, a signal set can be implemented using one bit per signal. For the remainder of this section, assume that an implementation has 31 signals and 32-bit integers. The `sigemptyset` function zeros the integer, and the `sigfillset` function turns on all the bits in the integer. These two functions can be implemented as macros in the `<signal.h>` header:

```
#define sigemptyset(ptr)   (*(ptr) = 0)
#define sigfillset(ptr)    (*(ptr) = ~(sigset_t)0, 0)
```

Note that `sigfillset` must return 0, in addition to setting all the bits on in the signal set, so we use C's comma operator, which returns the value after the comma as the value of the expression.

Using this implementation, `sigaddset` turns on a single bit and `sigdelset` turns off a single bit; `sigismember` tests a certain bit. Since no signal is ever numbered 0, we subtract 1 from the signal number to obtain the bit to manipulate. Figure 10.12 shows implementations of these functions.

```
#include    <signal.h>
#include    <errno.h>

/* <signal.h> usually defines NSIG to include signal number 0 */
#define SIGBAD(signo)   ((signo) <= 0 || (signo) >= NSIG)

int
sigaddset(sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

    *set |= 1 << (signo - 1);       /* turn bit on */
    return(0);
}

int
sigdelset(sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

    *set &= ~(1 << (signo - 1));    /* turn bit off */
    return(0);
}

int
sigismember(const sigset_t *set, int signo)
{
    if (SIGBAD(signo)) { errno = EINVAL; return(-1); }

    return((*set & (1 << (signo - 1))) != 0);
}
```

Figure 10.12 An implementation of `sigaddset`, `sigdelset`, and `sigismember`

We might be tempted to implement these three functions as one-line macros in the `<signal.h>` header, but POSIX.1 requires us to check the signal number argument for validity and to set `errno` if it is invalid. This is more difficult to do in a macro than in a function.

10.12 `sigprocmask` Function

Recall from Section 10.8 that the signal mask of a process is the set of signals currently blocked from delivery to that process. A process can examine its signal mask, change its signal mask, or perform both operations in one step by calling the following function.

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *restrict set,
                sigset_t *restrict oset);
```

Returns: 0 if OK, −1 on error

First, if *oset* is a non-null pointer, the current signal mask for the process is returned through *oset*.

Second, if *set* is a non-null pointer, the *how* argument indicates how the current signal mask is modified. Figure 10.13 describes the possible values for *how*. SIG_BLOCK is an inclusive-OR operation, whereas SIG_SETMASK is an assignment. Note that SIGKILL and SIGSTOP can't be blocked.

how	Description
SIG_BLOCK	The new signal mask for the process is the union of its current signal mask and the signal set pointed to by *set*. That is, *set* contains the additional signals that we want to block.
SIG_UNBLOCK	The new signal mask for the process is the intersection of its current signal mask and the complement of the signal set pointed to by *set*. That is, *set* contains the signals that we want to unblock.
SIG_SETMASK	The new signal mask for the process is replaced by the value of the signal set pointed to by *set*.

Figure 10.13 Ways to change current signal mask using sigprocmask

If *set* is a null pointer, the signal mask of the process is not changed, and *how* is ignored.

After calling sigprocmask, if any unblocked signals are pending, at least one of these signals is delivered to the process before sigprocmask returns.

The sigprocmask function is defined only for single-threaded processes. A separate function is provided to manipulate a thread's signal mask in a multithreaded process. We'll discuss this in Section 12.8.

Example

Figure 10.14 shows a function that prints the names of the signals in the signal mask of the calling process. We call this function from the programs shown in Figure 10.20 and Figure 10.22.

```
#include "apue.h"
#include <errno.h>

void
pr_mask(const char *str)
{
    sigset_t    sigset;
    int         errno_save;

    errno_save = errno;     /* we can be called by signal handlers */
    if (sigprocmask(0, NULL, &sigset) < 0)
        err_sys("sigprocmask error");

    printf("%s", str);
    if (sigismember(&sigset, SIGINT))   printf("SIGINT ");
    if (sigismember(&sigset, SIGQUIT))  printf("SIGQUIT ");
    if (sigismember(&sigset, SIGUSR1))  printf("SIGUSR1 ");
    if (sigismember(&sigset, SIGALRM))  printf("SIGALRM ");

    /* remaining signals can go here  */

    printf("\n");
    errno = errno_save;
}
```

Figure 10.14 Print the signal mask for the process

To save space, we don't test the signal mask for every signal that we listed in Figure 10.1. (See Exercise 10.9.) □

10.13 `sigpending` Function

The `sigpending` function returns the set of signals that are blocked from delivery and currently pending for the calling process. The set of signals is returned through the *set* argument.

```
#include <signal.h>

int sigpending(sigset_t *set);
```

Returns: 0 if OK, −1 on error

Example

Figure 10.15 shows many of the signal features that we've been describing.

```
#include "apue.h"

static void sig_quit(int);

int
main(void)
{
    sigset_t    newmask, oldmask, pendmask;

    if (signal(SIGQUIT, sig_quit) == SIG_ERR)
        err_sys("can't catch SIGQUIT");

    /*
     * Block SIGQUIT and save current signal mask.
     */
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

    sleep(5);    /* SIGQUIT here will remain pending */

    if (sigpending(&pendmask) < 0)
        err_sys("sigpending error");
    if (sigismember(&pendmask, SIGQUIT))
        printf("\nSIGQUIT pending\n");

    /*
     * Reset signal mask which unblocks SIGQUIT.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
    printf("SIGQUIT unblocked\n");

    sleep(5);    /* SIGQUIT here will terminate with core file */
    exit(0);
}

static void
sig_quit(int signo)
{
    printf("caught SIGQUIT\n");
    if (signal(SIGQUIT, SIG_DFL) == SIG_ERR)
        err_sys("can't reset SIGQUIT");
}
```

Figure 10.15 Example of signal sets and sigprocmask

The process blocks SIGQUIT, saving its current signal mask (to reset later), and then goes to sleep for 5 seconds. Any occurrence of the quit signal during this period is blocked and won't be delivered until the signal is unblocked. At the end of the 5-second sleep, we check whether the signal is pending and unblock the signal.

Note that we saved the old mask when we blocked the signal. To unblock the signal, we did a SIG_SETMASK of the old mask. Alternatively, we could SIG_UNBLOCK only the signal that we had blocked. Be aware, however, if we write a function that can be called by others and if we need to block a signal in our function, we can't use SIG_UNBLOCK to unblock the signal. In this case, we have to use SIG_SETMASK and reset the signal mask to its prior value, because it's possible that the caller had specifically blocked this signal before calling our function. We'll see an example of this in the system function in Section 10.18.

If we generate the quit signal during this sleep period, the signal is now pending and unblocked, so it is delivered before sigprocmask returns. We'll see this occur because the printf in the signal handler is output before the printf that follows the call to sigprocmask.

The process then goes to sleep for another 5 seconds. If we generate the quit signal during this sleep period, the signal should terminate the process, since we reset the handling of the signal to its default when we caught it. In the following output, the terminal prints ^\ when we input Control-backslash, the terminal quit character:

```
$ ./a.out
^\                          generate signal once (before 5 seconds are up)
SIGQUIT pending             after return from sleep
caught SIGQUIT              in signal handler
SIGQUIT unblocked           after return from sigprocmask
^\Quit(coredump)            generate signal again
$ ./a.out
^\^\^\^\^\^\^\^\^\^\        generate signal 10 times (before 5 seconds are up)
SIGQUIT pending
caught SIGQUIT              signal is generated only once
SIGQUIT unblocked
^\Quit(coredump)            generate signal again
```

The message `Quit(coredump)` is printed by the shell when it sees that its child terminated abnormally. Note that when we run the program the second time, we generate the quit signal ten times while the process is asleep, yet the signal is delivered only once to the process when it's unblocked. This demonstrates that signals are not queued on this system. □

10.14 `sigaction` Function

The `sigaction` function allows us to examine or modify (or both) the action associated with a particular signal. This function supersedes the `signal` function from earlier releases of the UNIX System. Indeed, at the end of this section, we show an implementation of `signal` using `sigaction`.

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *restrict act,
              struct sigaction *restrict oact);

                                        Returns: 0 if OK, -1 on error
```

The argument *signo* is the signal number whose action we are examining or modifying. If the *act* pointer is non-null, we are modifying the action. If the *oact* pointer is non-null, the system returns the previous action for the signal through the *oact* pointer. This function uses the following structure:

```
struct sigaction {
  void      (*sa_handler)(int);   /* addr of signal handler, */
                                  /* or SIG_IGN, or SIG_DFL */
  sigset_t  sa_mask;              /* additional signals to block */
  int       sa_flags;             /* signal options, Figure 10.16 */

  /* alternate handler */
  void      (*sa_sigaction)(int, siginfo_t *, void *);
};
```

When changing the action for a signal, if the sa_handler field contains the address of a signal-catching function (as opposed to the constants SIG_IGN or SIG_DFL), then the sa_mask field specifies a set of signals that are added to the signal mask of the process before the signal-catching function is called. If and when the signal-catching function returns, the signal mask of the process is reset to its previous value. This way, we are able to block certain signals whenever a signal handler is invoked. The operating system includes the signal being delivered in the signal mask when the handler is invoked. Hence, we are guaranteed that whenever we are processing a given signal, another occurrence of that same signal is blocked until we're finished processing the first occurrence. Recall from Section 10.8 that additional occurrences of the same signal are usually not queued. If the signal occurs five times while it is blocked, when we unblock the signal, the signal-handling function for that signal will usually be invoked only one time.

Once we install an action for a given signal, that action remains installed until we explicitly change it by calling sigaction. Unlike earlier systems with their unreliable signals, POSIX.1 requires that a signal handler remain installed until explicitly changed.

The sa_flags field of the *act* structure specifies various options for the handling of this signal. Figure 10.16 details the meaning of these options when set. The SUS column contains • if the flag is defined as part of the base POSIX.1 specification, and **XSI** if it is defined as an XSI extension to the base.

The sa_sigaction field is an alternate signal handler used when the SA_SIGINFO flag is used with sigaction. Implementations might use the same storage for both the sa_sigaction field and the sa_handler field, so applications can use only one of these fields at a time.

Normally, the signal handler is called as

```
void handler(int signo);
```

but if the SA_SIGINFO flag is set, the signal handler is called as

```
void handler(int signo, siginfo_t *info, void *context);
```

The siginfo_t structure contains information about why the signal was generated. An example of what it might look like is shown below. All POSIX.1-compliant implementations must include at least the si_signo and si_code members. Additionally, implementations that are XSI compliant contain at least the following fields:

```
struct siginfo {
  int    si_signo;  /* signal number */
  int    si_errno;  /* if nonzero, errno value from <errno.h> */
  int    si_code;   /* additional info (depends on signal) */
  pid_t  si_pid;    /* sending process ID */
  uid_t  si_uid;    /* sending process real user ID */
  void  *si_addr;   /* address that caused the fault */
  int    si_status; /* exit value or signal number */
  long   si_band;   /* band number for SIGPOLL */
  /* possibly other fields also */
};
```

Option	SUS	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	Description
`SA_INTERRUPT`			•			System calls interrupted by this signal are not automatically restarted (the XSI default for `sigaction`). See Section 10.5 for more information.
`SA_NOCLDSTOP`	•	•	•	•	•	If *signo* is `SIGCHLD`, do not generate this signal when a child process stops (job control). This signal is still generated, of course, when a child terminates (but see the `SA_NOCLDWAIT` option below). As an XSI extension, `SIGCHLD` won't be sent when a stopped child continues if this flag is set.
`SA_NOCLDWAIT`	XSI	•	•	•	•	If *signo* is `SIGCHLD`, this option prevents the system from creating zombie processes when children of the calling process terminate. If it subsequently calls `wait`, the calling process blocks until all its child processes have terminated and then returns −1 with `errno` set to `ECHILD`. (Recall Section 10.7.)
`SA_NODEFER`	XSI	•	•	•	•	When this signal is caught, the signal is not automatically blocked by the system while the signal-catching function executes (unless the signal is also included in `sa_mask`). Note that this type of operation corresponds to the earlier unreliable signals.
`SA_ONSTACK`	XSI	•	•	•	•	If an alternate stack has been declared with `sigaltstack`(2), this signal is delivered to the process on the alternate stack.
`SA_RESETHAND`	XSI	•	•	•	•	The disposition for this signal is reset to `SIG_DFL`, and the `SA_SIGINFO` flag is cleared on entry to the signal-catching function. Note that this type of operation corresponds to the earlier unreliable signals. The disposition for the two signals `SIGILL` and `SIGTRAP` can't be reset automatically, however. Setting this flag causes `sigaction` to behave as if `SA_NODEFER` is also set.
`SA_RESTART`	XSI	•	•	•	•	System calls interrupted by this signal are automatically restarted. (Refer to Section 10.5.)
`SA_SIGINFO`	•	•	•	•	•	This option provides additional information to a signal handler: a pointer to a `siginfo` structure and a pointer to an identifier for the process context.

Figure 10.16 Option flags (`sa_flags`) for the handling of each signal

Figure 10.17 shows values of si_code for various signals, as defined by the Single UNIX Specification. Note that implementations may define additional code values.

Signal	Code	Reason
SIGILL	ILL_ILLOPC	illegal opcode
	ILL_ILLOPN	illegal operand
	ILL_ILLADR	illegal addressing mode
	ILL_ILLTRP	illegal trap
	ILL_PRVOPC	privileged opcode
	ILL_PRVREG	privileged register
	ILL_COPROC	coprocessor error
	ILL_BADSTK	internal stack error
SIGFPE	FPE_INTDIV	integer divide by zero
	FPE_INTOVF	integer overflow
	FPE_FLTDIV	floating-point divide by zero
	FPE_FLTOVF	floating-point overflow
	FPE_FLTUND	floating-point underflow
	FPE_FLTRES	floating-point inexact result
	FPE_FLTINV	invalid floating-point operation
	FPE_FLTSUB	subscript out of range
SIGSEGV	SEGV_MAPERR	address not mapped to object
	SEGV_ACCERR	invalid permissions for mapped object
SIGBUS	BUS_ADRALN	invalid address alignment
	BUS_ADRERR	nonexistent physical address
	BUS_OBJERR	object-specific hardware error
SIGTRAP	TRAP_BRKPT	process breakpoint trap
	TRAP_TRACE	process trace trap
SIGCHLD	CLD_EXITED	child has exited
	CLD_KILLED	child has terminated abnormally (no core)
	CLD_DUMPED	child has terminated abnormally with core
	CLD_TRAPPED	traced child has trapped
	CLD_STOPPED	child has stopped
	CLD_CONTINUED	stopped child has continued
SIGPOLL	POLL_IN	data can be read
	POLL_OUT	data can be written
	POLL_MSG	input message available
	POLL_ERR	I/O error
	POLL_PRI	high-priority message available
	POLL_HUP	device disconnected
Any	SI_USER	signal sent by kill
	SI_QUEUE	signal sent by sigqueue (real-time extension)
	SI_TIMER	expiration of a timer set by timer_settime (real-time extension)
	SI_ASYNCIO	completion of asynchronous I/O request (real-time extension)
	SI_MESGQ	arrival of a message on a message queue (real-time extension)

Figure 10.17 siginfo_t code values

If the signal is SIGCHLD, then the si_pid, si_status, and si_uid field will be set. If the signal is SIGILL or SIGSEGV, then the si_addr contains the address responsible for the fault, although the address might not be accurate. If the signal is SIGPOLL, then the si_band field will contain the priority band for STREAMS messages that generate the POLL_IN, POLL_OUT, or POLL_MSG events. (For a complete discussion of priority bands, see Rago [1993].) The si_errno field contains the error number corresponding to the condition that caused the signal to be generated, although its use is implementation defined.

The *context* argument to the signal handler is a typeless pointer that can be cast to a ucontext_t structure identifying the process context at the time of signal delivery.

When an implementation supports the real-time signal extensions, signal handlers established with the SA_SIGINFO flag will result in signals being queued reliably. A separate range of reserved signals is available for real-time application use. The siginfo structure can contain application-specific data if the signal is generated by sigqueue. We do not discuss the real-time extensions further. Refer to Gallmeister [1995] for more details.

Example—signal Function

Let's now implement the signal function using sigaction. This is what many platforms do (and what a note in the POSIX.1 Rationale states was the intent of POSIX). Systems with binary compatibility constraints, on the other hand, might provide a signal function that supports the older, unreliable-signal semantics. Unless you specifically require these older, unreliable semantics (for backward compatibility), you should use the following implementation of signal or call sigaction directly. (As you might guess, an implementation of signal with the old semantics could call sigaction specifying SA_RESETHAND and SA_NODEFER.) All the examples in this text that call signal call the function shown in Figure 10.18.

```
#include "apue.h"

/* Reliable version of signal(), using POSIX sigaction().  */
Sigfunc *
signal(int signo, Sigfunc *func)
{
    struct sigaction    act, oact;

    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if (signo == SIGALRM) {
#ifdef  SA_INTERRUPT
        act.sa_flags |= SA_INTERRUPT;
#endif
    } else {
#ifdef  SA_RESTART
        act.sa_flags |= SA_RESTART;
#endif
    }
    if (sigaction(signo, &act, &oact) < 0)
        return(SIG_ERR);
    return(oact.sa_handler);
}
```

Figure 10.18 An implementation of signal using sigaction

Note that we must use sigemptyset to initialize the sa_mask member of the structure. We're not guaranteed that

```
act.sa_mask = 0;
```

does the same thing.

We intentionally try to set the SA_RESTART flag for all signals other than SIGALRM, so that any system call interrupted by these other signals is automatically restarted. The reason we don't want SIGALRM restarted is to allow us to set a timeout for I/O operations. (Recall the discussion of Figure 10.10.)

Some older systems, such as SunOS, define the SA_INTERRUPT flag. These systems restart interrupted system calls by default, so specifying this flag causes system calls to be interrupted. Linux defines the SA_INTERRUPT flag for compatibility with applications that use it, but the default is to not restart system calls when the signal handler is installed with sigaction. The XSI extension of the Single UNIX Specification specifies that the sigaction function not restart interrupted system calls unless the SA_RESTART flag is specified. □

Example—signal_intr Function

Figure 10.19 shows a version of the signal function that tries to prevent any interrupted system calls from being restarted.

```
#include "apue.h"

Sigfunc *
signal_intr(int signo, Sigfunc *func)
{
    struct sigaction    act, oact;

    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
#ifdef  SA_INTERRUPT
    act.sa_flags |= SA_INTERRUPT;
#endif
    if (sigaction(signo, &act, &oact) < 0)
        return(SIG_ERR);
    return(oact.sa_handler);
}
```

Figure 10.19 The signal_intr function

For improved portability, we specify the SA_INTERRUPT flag, if defined by the system, to prevent interrupted system calls from being restarted. □

10.15 sigsetjmp and siglongjmp Functions

In Section 7.10, we described the setjmp and longjmp functions, which can be used for nonlocal branching. The longjmp function is often called from a signal handler to return to the main loop of a program, instead of returning from the handler. We saw this in Figures 10.8 and 10.11.

There is a problem in calling `longjmp`, however. When a signal is caught, the signal-catching function is entered with the current signal automatically being added to the signal mask of the process. This prevents subsequent occurrences of that signal from interrupting the signal handler. If we `longjmp` out of the signal handler, what happens to the signal mask for the process?

Under FreeBSD 5.2.1 and Mac OS X 10.3, `setjmp` and `longjmp` save and restore the signal mask. Linux 2.4.22 and Solaris 9, however, do not do this. FreeBSD and Mac OS X provide the functions `_setjmp` and `_longjmp`, which do not save and restore the signal mask.

To allow either form of behavior, POSIX.1 does not specify the effect of `setjmp` and `longjmp` on signal masks. Instead, two new functions, `sigsetjmp` and `siglongjmp`, are defined by POSIX.1. These two functions should always be used when branching from a signal handler.

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);
```

Returns: 0 if called directly, nonzero if returning from a call to `siglongjmp`

```
void siglongjmp(sigjmp_buf env, int val);
```

The only difference between these functions and the `setjmp` and `longjmp` functions is that `sigsetjmp` has an additional argument. If *savemask* is nonzero, then `sigsetjmp` also saves the current signal mask of the process in *env*. When `siglongjmp` is called, if the *env* argument was saved by a call to `sigsetjmp` with a nonzero *savemask*, then `siglongjmp` restores the saved signal mask.

Example

The program in Figure 10.20 demonstrates how the signal mask that is installed by the system when a signal handler is invoked automatically includes the signal being caught. The program also illustrates the use of the `sigsetjmp` and `siglongjmp` functions.

```
#include "apue.h"
#include <setjmp.h>
#include <time.h>

static void                         sig_usr1(int), sig_alrm(int);
static sigjmp_buf                   jmpbuf;
static volatile sig_atomic_t        canjump;

int
main(void)
{
    if (signal(SIGUSR1, sig_usr1) == SIG_ERR)
        err_sys("signal(SIGUSR1) error");
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        err_sys("signal(SIGALRM) error");
    pr_mask("starting main: ");     /* Figure 10.14 */

    if (sigsetjmp(jmpbuf, 1)) {
        pr_mask("ending main: ");
        exit(0);
    }
    canjump = 1;     /* now sigsetjmp() is OK */

    for ( ; ; )
        pause();
}

static void
sig_usr1(int signo)
{
    time_t  starttime;

    if (canjump == 0)
        return;     /* unexpected signal, ignore */

    pr_mask("starting sig_usr1: ");
    alarm(3);               /* SIGALRM in 3 seconds */
    starttime = time(NULL);
    for ( ; ; )             /* busy wait for 5 seconds */
        if (time(NULL) > starttime + 5)
            break;
    pr_mask("finishing sig_usr1: ");

    canjump = 0;
    siglongjmp(jmpbuf, 1);  /* jump back to main, don't return */
}

static void
sig_alrm(int signo)
{
    pr_mask("in sig_alrm: ");
}
```

Figure 10.20 Example of signal masks, `sigsetjmp`, and `siglongjmp`

This program demonstrates another technique that should be used whenever `siglongjmp` is called from a signal handler. We set the variable `canjump` to a nonzero value only after we've called `sigsetjmp`. This variable is also examined in the signal handler, and `siglongjmp` is called only if the flag `canjump` is nonzero. This provides protection against the signal handler being called at some earlier or later time, when the jump buffer isn't initialized by `sigsetjmp`. (In this trivial program, we terminate quickly after the `siglongjmp`, but in larger programs, the signal handler may remain installed long after the `siglongjmp`.) Providing this type of protection usually isn't required with `longjmp` in normal C code (as opposed to a signal handler). Since a signal can occur at *any* time, however, we need the added protection in a signal handler.

Here, we use the data type `sig_atomic_t`, which is defined by the ISO C standard to be the type of variable that can be written without being interrupted. By this we mean that a variable of this type should not extend across page boundaries on a system with virtual memory and can be accessed with a single machine instruction, for example. We always include the ISO type qualifier `volatile` for these data types too, since the variable is being accessed by two different threads of control: the `main` function and the asynchronously executing signal handler. Figure 10.21 shows a time line for this program.

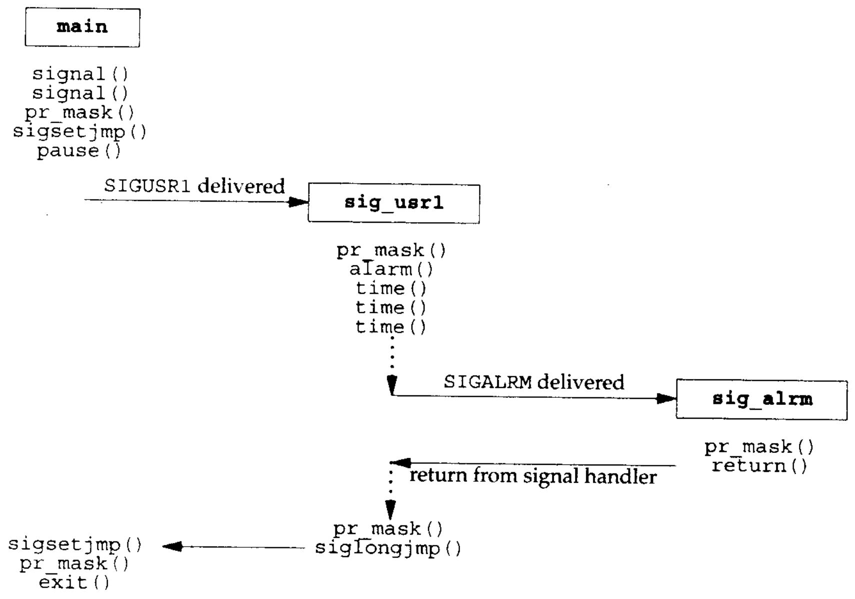

Figure 10.21 Time line for example program handling two signals

We can divide Figure 10.21 into three parts: the left part (corresponding to `main`), the center part (`sig_usr1`), and the right part (`sig_alrm`). While the process is executing in the left part, its signal mask is 0 (no signals are blocked). While executing in the center part, its signal mask is `SIGUSR1`. While executing in the right part, its signal mask is `SIGUSR1 | SIGALRM`.

Let's examine the output when the program in Figure 10.20 is executed:

```
$ ./a.out &                          start process in background
starting main:
[1]    531                           the job-control shell prints its process ID
$ kill -USR1 531                     send the process SIGUSR1
starting sig_usr1: SIGUSR1
$ in sig_alrm: SIGUSR1 SIGALRM
finishing sig_usr1: SIGUSR1
ending main:
                                     just press RETURN
[1] +  Done           ./a.out &
```

The output is as we expect: when a signal handler is invoked, the signal being caught is added to the current signal mask of the process. The original mask is restored when the signal handler returns. Also, siglongjmp restores the signal mask that was saved by sigsetjmp.

If we change the program in Figure 10.20 so that the calls to sigsetjmp and siglongjmp are replaced with calls to setjmp and longjmp on Linux (or _setjmp and _longjmp on FreeBSD), the final line of output becomes

```
ending main: SIGUSR1
```

This means that the main function is executing with the SIGUSR1 signal blocked, after the call to setjmp. This probably isn't what we want. □

10.16 sigsuspend Function

We have seen how we can change the signal mask for a process to block and unblock selected signals. We can use this technique to protect critical regions of code that we don't want interrupted by a signal. What if we want to unblock a signal and then pause, waiting for the previously blocked signal to occur? Assuming that the signal is SIGINT, the incorrect way to do this is

```
sigset_t    newmask, oldmask;

sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);

/* block SIGINT and save current signal mask */
if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
    err_sys("SIG_BLOCK error");

/* critical region of code */

/* reset signal mask, which unblocks SIGINT */
if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
    err_sys("SIG_SETMASK error");

/* window is open */
pause();  /* wait for signal to occur */

/* continue processing */
```

If the signal is sent to the process while it is blocked, the signal delivery will be deferred until the signal is unblocked. To the application, this can look as if the signal occurs between the unblocking and the pause (depending on how the kernel implements signals). If this happens, or if the signal does occur between the unblocking and the pause, we have a problem. Any occurrence of the signal in this window of time is lost in the sense that we might not see the signal again, in which case the pause will block indefinitely. This is another problem with the earlier unreliable signals.

To correct this problem, we need a way to both reset the signal mask and put the process to sleep in a single atomic operation. This feature is provided by the sigsuspend function.

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);

                                        Returns: -1 with errno set to EINTR
```

The signal mask of the process is set to the value pointed to by *sigmask*. Then the process is suspended until a signal is caught or until a signal occurs that terminates the process. If a signal is caught and if the signal handler returns, then `sigsuspend` returns, and the signal mask of the process is set to its value before the call to `sigsuspend`.

Note that there is no successful return from this function. If it returns to the caller, it always returns −1 with `errno` set to `EINTR` (indicating an interrupted system call).

Example

Figure 10.22 shows the correct way to protect a critical region of code from a specific signal.

```
#include "apue.h"

static void sig_int(int);

int
main(void)
{
    sigset_t    newmask, oldmask, waitmask;

    pr_mask("program start: ");

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    sigemptyset(&waitmask);
    sigaddset(&waitmask, SIGUSR1);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGINT);

    /*
     * Block SIGINT and save current signal mask.
     */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

    /*
     * Critical region of code.
     */
    pr_mask("in critical region: ");

    /*
     * Pause, allowing all signals except SIGUSR1.
     */
    if (sigsuspend(&waitmask) != -1)
        err_sys("sigsuspend error");

    pr_mask("after return from sigsuspend: ");

    /*
     * Reset signal mask which unblocks SIGINT.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");

    /*
     * And continue processing ...
     */
    pr_mask("program exit: ");

    exit(0);
}

static void
sig_int(int signo)
{
    pr_mask("\nin sig_int: ");
}
```

Figure 10.22 Protecting a critical region from a signal

Note that when sigsuspend returns, it sets the signal mask to its value before the call. In this example, the SIGINT signal will be blocked. We therefore reset the signal mask to the value that we saved earlier (oldmask).

Running the program from Figure 10.22 produces the following output:

```
$ ./a.out
program start:
in critical region: SIGINT
^?                                        type the interrupt character
in sig_int: SIGINT SIGUSR1
after return from sigsuspend: SIGINT
program exit:
```

We added SIGUSR1 to the mask installed when we called sigsuspend so that when the signal handler ran, we could tell that the mask had actually changed. We can see that when sigsuspend returns, it restores the signal mask to its value before the call. □

Example

Another use of sigsuspend is to wait for a signal handler to set a global variable. In the program shown in Figure 10.23, we catch both the interrupt signal and the quit signal, but want to wake up the main routine only when the quit signal is caught.

```
#include "apue.h"

volatile sig_atomic_t   quitflag;    /* set nonzero by signal handler */

static void
sig_int(int signo)  /* one signal handler for SIGINT and SIGQUIT */
{
    if (signo == SIGINT)
        printf("\ninterrupt\n");
    else if (signo == SIGQUIT)
        quitflag = 1;   /* set flag for main loop */
}

int
main(void)
{
    sigset_t    newmask, oldmask, zeromask;

    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    if (signal(SIGQUIT, sig_int) == SIG_ERR)
        err_sys("signal(SIGQUIT) error");

    sigemptyset(&zeromask);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);

    /*
     * Block SIGQUIT and save current signal mask.
     */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

    while (quitflag == 0)
        sigsuspend(&zeromask);

    /*
     * SIGQUIT has been caught and is now blocked; do whatever.
     */
    quitflag = 0;

    /*
     * Reset signal mask which unblocks SIGQUIT.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");

    exit(0);
}
```

Figure 10.23 Using sigsuspend to wait for a global variable to be set

Sample output from this program is

```
$ ./a.out
^?                      type the interrupt character
interrupt
^?                      type the interrupt character again
interrupt
^?                      and again
interrupt
^?                      and again
interrupt
^?                      and again
interrupt
^?                      and again
interrupt
^?                      and again
interrupt
^\ $                    now terminate with quit character
```

□

For portability between non-POSIX systems that support ISO C, and POSIX.1 systems, the only thing we should do within a signal handler is assign a value to a variable of type `sig_atomic_t`, and nothing else. POSIX.1 goes further and specifies a list of functions that are safe to call from within a signal handler (Figure 10.4), but if we do this, our code may not run correctly on non-POSIX systems.

Example

As another example of signals, we show how signals can be used to synchronize a parent and child. Figure 10.24 shows implementations of the five routines `TELL_WAIT`, `TELL_PARENT`, `TELL_CHILD`, `WAIT_PARENT`, and `WAIT_CHILD` from Section 8.9.

```
#include "apue.h"

static volatile sig_atomic_t sigflag; /* set nonzero by sig handler */
static sigset_t newmask, oldmask, zeromask;

static void
sig_usr(int signo)   /* one signal handler for SIGUSR1 and SIGUSR2 */
{
    sigflag = 1;
}

void
TELL_WAIT(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("signal(SIGUSR1) error");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("signal(SIGUSR2) error");
    sigemptyset(&zeromask);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGUSR1);
    sigaddset(&newmask, SIGUSR2);

    /*
     * Block SIGUSR1 and SIGUSR2, and save current signal mask.
     */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");
}

void
TELL_PARENT(pid_t pid)
{
    kill(pid, SIGUSR2);     /* tell parent we're done */
}

void
WAIT_PARENT(void)
{
    while (sigflag == 0)
        sigsuspend(&zeromask);  /* and wait for parent */
    sigflag = 0;

    /*
     * Reset signal mask to original value.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
}

void
TELL_CHILD(pid_t pid)
{
    kill(pid, SIGUSR1);         /* tell child we're done */
}

void
WAIT_CHILD(void)
{
    while (sigflag == 0)
        sigsuspend(&zeromask);  /* and wait for child */
    sigflag = 0;

    /*
     * Reset signal mask to original value.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
}
```

Figure 10.24 Routines to allow a parent and child to synchronize

We use the two user-defined signals: SIGUSR1 is sent by the parent to the child, and SIGUSR2 is sent by the child to the parent. In Figure 15.7, we show another implementation of these five functions using pipes. □

The sigsuspend function is fine if we want to go to sleep while waiting for a signal to occur (as we've shown in the previous two examples), but what if we want to call other system functions while we're waiting? Unfortunately, this problem has no bulletproof solution unless we use multiple threads and dedicate a separate thread to handling signals, as we discuss in Section 12.8.

Without using threads, the best we can do is to set a global variable in the signal handler when the signal occurs. For example, if we catch both SIGINT and SIGALRM and install the signal handlers using the signal_intr function, the signals will interrupt any slow system call that is blocked. The signals are most likely to occur when we're blocked in a call to the select function (Section 14.5.1), waiting for input from a slow device. (This is especially true for SIGALRM, since we set the alarm clock to prevent us from waiting forever for input.) The code to handle this looks similar to the following:

```
if (intr_flag)        /* flag set by our SIGINT handler */
    handle_intr();
if (alrm_flag)        /* flag set by our SIGALRM handler */
    handle_alrm();

/* signals occurring in here are lost */

while (select( ... ) < 0) {
    if (errno == EINTR) {
        if (alrm_flag)
            handle_alrm();
        else if (intr_flag)
            handle_intr();
    } else {
        /* some other error */
    }
}
```

We test each of the global flags before calling select and again if select returns an interrupted system call error. The problem occurs if either signal is caught between the first two if statements and the subsequent call to select. Signals occurring in here are lost, as indicated by the code comment. The signal handlers are called, and they set the appropriate global variable, but the select never returns (unless some data is ready to be read).

What we would like to be able to do is the following sequence of steps, in order.

1. Block SIGINT and SIGALRM.
2. Test the two global variables to see whether either signal has occurred and, if so, handle the condition.

3. Call `select` (or any other system function, such as `read`) and unblock the two signals, as an atomic operation.

The `sigsuspend` function helps us only if step 3 is a `pause` operation.

10.17 `abort` Function

We mentioned earlier that the `abort` function causes abnormal program termination.

```
#include <stdlib.h>

void abort(void);
```

This function never returns

This function sends the `SIGABRT` signal to the caller. (Processes should not ignore this signal.) ISO C states that calling `abort` will deliver an unsuccessful termination notification to the host environment by calling `raise(SIGABRT)`.

ISO C requires that if the signal is caught and the signal handler returns, `abort` still doesn't return to its caller. If this signal is caught, the only way the signal handler can't return is if it calls `exit`, `_exit`, `_Exit`, `longjmp`, or `siglongjmp`. (Section 10.15 discusses the differences between `longjmp` and `siglongjmp`.) POSIX.1 also specifies that `abort` overrides the blocking or ignoring of the signal by the process.

The intent of letting the process catch the `SIGABRT` is to allow it to perform any cleanup that it wants to do before the process terminates. If the process doesn't terminate itself from this signal handler, POSIX.1 states that, when the signal handler returns, `abort` terminates the process.

The ISO C specification of this function leaves it up to the implementation as to whether output streams are flushed and whether temporary files (Section 5.13) are deleted. POSIX.1 goes further and requires that if the call to `abort` terminates the process, then the effect on the open standard I/O streams in the process will be the same as if the process had called `fclose` on each stream before terminating.

Earlier versions of System V generated the `SIGIOT` signal from the `abort` function. Furthermore, it was possible for a process to ignore this signal or to catch it and return from the signal handler, in which case `abort` returned to its caller.

4.3BSD generated the `SIGILL` signal. Before doing this, the 4.3BSD function unblocked the signal and reset its disposition to `SIG_DFL` (terminate with `core` file). This prevented a process from either ignoring the signal or catching it.

Historically, implementations of `abort` differ in how they deal with standard I/O streams. For defensive programming and improved portability, if we want standard I/O streams to be flushed, we specifically do it before calling `abort`. We do this in the `err_dump` function (Appendix B).

Since most UNIX System implementations of `tmpfile` call `unlink` immediately after creating the file, the ISO C warning about temporary files does not usually concern us.

Example

Figure 10.25 shows an implementation of the abort function as specified by POSIX.1.

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void
abort(void)         /* POSIX-style abort() function */
{
    sigset_t            mask;
    struct sigaction    action;

    /*
     * Caller can't ignore SIGABRT, if so reset to default.
     */
    sigaction(SIGABRT, NULL, &action);
    if (action.sa_handler == SIG_IGN) {
        action.sa_handler = SIG_DFL;
        sigaction(SIGABRT, &action, NULL);
    }
    if (action.sa_handler == SIG_DFL)
        fflush(NULL);           /* flush all open stdio streams */

    /*
     * Caller can't block SIGABRT; make sure it's unblocked.
     */
    sigfillset(&mask);
    sigdelset(&mask, SIGABRT);  /* mask has only SIGABRT turned off */
    sigprocmask(SIG_SETMASK, &mask, NULL);
    kill(getpid(), SIGABRT);    /* send the signal */

    /*
     * If we're here, process caught SIGABRT and returned.
     */
    fflush(NULL);               /* flush all open stdio streams */
    action.sa_handler = SIG_DFL;
    sigaction(SIGABRT, &action, NULL);  /* reset to default */
    sigprocmask(SIG_SETMASK, &mask, NULL);  /* just in case ... */
    kill(getpid(), SIGABRT);                /* and one more time */
    exit(1);    /* this should never be executed ... */
}
```

Figure 10.25 Implementation of POSIX.1 abort

We first see whether the default action will occur; if so, we flush all the standard I/O streams. This is not equivalent to an fclose on all the open streams (since it just flushes them and doesn't close them), but when the process terminates, the system closes all open files. If the process catches the signal and returns, we flush all the streams again, since the process could have generated more output. The only condition we don't handle is if the process catches the signal and calls `_exit` or `_Exit`. In this case, any unflushed standard I/O buffers in memory are discarded. We assume that a caller that does this doesn't want the buffers flushed.

Recall from Section 10.9 that if calling `kill` causes the signal to be generated for the caller, and if the signal is not blocked (which we guarantee in Figure 10.25), then the signal (or some other pending, unlocked signal) is delivered to the process before `kill` returns. We block all signals except `SIGABRT`, so we know that if the call to `kill` returns, the process caught the signal and the signal handler returned. □

10.18 system Function

In Section 8.13, we showed an implementation of the `system` function. That version, however, did not do any signal handling. POSIX.1 requires that `system` ignore `SIGINT` and `SIGQUIT` and block `SIGCHLD`. Before showing a version that correctly handles these signals, let's see why we need to worry about signal handling.

Example

The program shown in Figure 10.26 uses the version of `system` from Section 8.13 to invoke the `ed`(1) editor. (This editor has been part of UNIX systems for a long time. We use it here because it is an interactive program that catches the interrupt and quit signals. If we invoke `ed` from a shell and type the interrupt character, it catches the interrupt signal and prints a question mark. The `ed` program also sets the disposition of the quit signal so that it is ignored.) The program in Figure 10.26 catches both `SIGINT` and `SIGCHLD`. If we invoke the program, we get

```
$ ./a.out
a                           append text to the editor's buffer
Here is one line of text
.                           period on a line by itself stops append mode
1,$p                        print first through last lines of buffer to see what's there
Here is one line of text
w temp.foo                  write the buffer to a file
25                          editor says it wrote 25 bytes
q                           and leave the editor
caught SIGCHLD
```

When the editor terminates, the system sends the `SIGCHLD` signal to the parent (the `a.out` process). We catch it and return from the signal handler. But if it is catching the `SIGCHLD` signal, the parent should be doing so because it has created its own children, so that it knows when its children have terminated. The delivery of this signal in the parent should be blocked while the `system` function is executing. Indeed, this is what POSIX.1 specifies. Otherwise, when the child created by `system` terminates, it would fool the caller of `system` into thinking that one of its own children terminated. The

```
#include "apue.h"

static void
sig_int(int signo)
{
    printf("caught SIGINT\n");
}

static void
sig_chld(int signo)
{
    printf("caught SIGCHLD\n");
}

int
main(void)
{
    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    if (signal(SIGCHLD, sig_chld) == SIG_ERR)
        err_sys("signal(SIGCHLD) error");
    if (system("/bin/ed") < 0)
        err_sys("system() error");
    exit(0);
}
```

Figure 10.26 Using system to invoke the ed editor

caller would then use one of the wait functions to get the termination status of the child, thus preventing the system function from being able to obtain the child's termination status for its return value.

If we run the program again, this time sending the editor an interrupt signal, we get

```
$ ./a.out
a                       append text to the editor's buffer
hello, world
.                       period on a line by itself stops append mode
1,$p                    print first through last lines to see what's there
hello, world
w temp.foo              write the buffer to a file
13                      editor says it wrote 13 bytes
^?                      type the interrupt character
?                       editor catches signal, prints question mark
caught SIGINT           and so does the parent process
q                       leave editor
caught SIGCHLD
```

Recall from Section 9.6 that typing the interrupt character causes the interrupt signal to be sent to all the processes in the foreground process group. Figure 10.27 shows the arrangement of the processes when the editor is running.

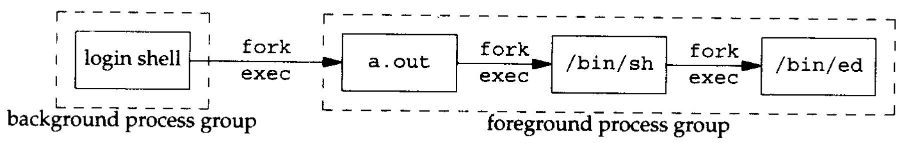

Figure 10.27 Foreground and background process groups for Figure 10.26

In this example, SIGINT is sent to all three foreground processes. (The shell ignores it.) As we can see from the output, both the a.out process and the editor catch the signal. But when we're running another program with the system function, we shouldn't have both the parent and the child catching the two terminal-generated signals: interrupt and quit. These two signals should really be sent to the program that is running: the child. Since the command that is executed by system can be an interactive command (as is the ed program in this example) and since the caller of system gives up control while the program executes, waiting for it to finish, the caller of system should not be receiving these two terminal-generated signals. This is why POSIX.1 specifies that the system function should ignore these two signals while waiting for the command to complete. □

Example

Figure 10.28 shows an implementation of the system function with the required signal handling.

```
#include    <sys/wait.h>
#include    <errno.h>
#include    <signal.h>
#include    <unistd.h>

int
system(const char *cmdstring)   /* with appropriate signal handling */
{
    pid_t               pid;
    int                 status;
    struct sigaction    ignore, saveintr, savequit;
    sigset_t            chldmask, savemask;

    if (cmdstring == NULL)
        return(1);      /* always a command processor with UNIX */

    ignore.sa_handler = SIG_IGN;    /* ignore SIGINT and SIGQUIT */
    sigemptyset(&ignore.sa_mask);
    ignore.sa_flags = 0;
    if (sigaction(SIGINT, &ignore, &saveintr) < 0)
        return(-1);
    if (sigaction(SIGQUIT, &ignore, &savequit) < 0)
        return(-1);
    sigemptyset(&chldmask);         /* now block SIGCHLD */
    sigaddset(&chldmask, SIGCHLD);
    if (sigprocmask(SIG_BLOCK, &chldmask, &savemask) < 0)
        return(-1);

    if ((pid = fork()) < 0) {
        status = -1;    /* probably out of processes */
    } else if (pid == 0) {          /* child */
        /* restore previous signal actions & reset signal mask */
        sigaction(SIGINT, &saveintr, NULL);
        sigaction(SIGQUIT, &savequit, NULL);
        sigprocmask(SIG_SETMASK, &savemask, NULL);

        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);     /* exec error */
    } else {                        /* parent */
        while (waitpid(pid, &status, 0) < 0)
            if (errno != EINTR) {
                status = -1; /* error other than EINTR from waitpid() */
                break;
            }
    }

    /* restore previous signal actions & reset signal mask */
    if (sigaction(SIGINT, &saveintr, NULL) < 0)
        return(-1);
    if (sigaction(SIGQUIT, &savequit, NULL) < 0)
        return(-1);
    if (sigprocmask(SIG_SETMASK, &savemask, NULL) < 0)
        return(-1);

    return(status);
}
```

Figure 10.28 Correct POSIX.1 implementation of system function

If we link the program in Figure 10.26 with this implementation of the system function, the resulting binary differs from the last (flawed) one in the following ways.

1. No signal is sent to the calling process when we type the interrupt or quit character.
2. When the ed command exits, SIGCHLD is not sent to the calling process. Instead, it is blocked until we unblock it in the last call to sigprocmask, after the system function retrieves the child's termination status by calling waitpid.

POSIX.1 states that if wait or waitpid returns the status of a child process while SIGCHLD is pending, then SIGCHLD should not be delivered to the process unless the status of another child process is also available. None of the four implementations discussed in this book implements this semantic. Instead, SIGCHLD remains pending after the system function calls waitpid; when the signal is unblocked, it is delivered to the caller. If we called wait in the sig_chld function in Figure 10.26, it would return −1 with errno set to ECHILD, since the system function already retrieved the termination status of the child.

Many older texts show the ignoring of the interrupt and quit signals as follows:

```
if ((pid = fork()) < 0) {
    err_sys("fork error");
} else if (pid == 0) {
    /* child */
    execl(...);
    _exit(127);
}

/* parent */
old_intr = signal(SIGINT, SIG_IGN);
old_quit = signal(SIGQUIT, SIG_IGN);
waitpid(pid, &status, 0)
signal(SIGINT, old_intr);
signal(SIGQUIT, old_quit);
```

The problem with this sequence of code is that we have no guarantee after the fork whether the parent or child runs first. If the child runs first and the parent doesn't run for some time after, it's possible for an interrupt signal to be generated before the parent is able to change its disposition to be ignored. For this reason, in Figure 10.28, we change the disposition of the signals before the fork.

Note that we have to reset the dispositions of these two signals in the child before the call to execl. This allows execl to change their dispositions to the default, based on the caller's dispositions, as we described in Section 8.10. □

Return Value from system

Beware of the return value from system. It is the termination status of the shell, which isn't always the termination status of the command string. We saw some examples in Figure 8.23, and the results were as we expected: if we execute a simple command, such as date, the termination status is 0. Executing the shell command exit 44 gave us a termination status of 44. What happens with signals?

Let's run the program in Figure 8.24 and send some signals to the command that's executing:

```
$ tsys "sleep 30"
^?normal termination, exit status = 130    we type the interrupt key
$ tsys "sleep 30"
^\sh: 946 Quit                             we type the quit key
normal termination, exit status = 131
```

When we terminate the sleep with the interrupt signal, the pr_exit function (Figure 8.5) thinks that it terminated normally. The same thing happens when we kill the sleep with the quit key. What is happening here is that the Bourne shell has a poorly documented feature that its termination status is 128 plus the signal number, when the command it was executing is terminated by a signal. We can see this with the shell interactively.

```
$ sh                                  make sure we're running the Bourne shell
$ sh -c "sleep 30"
^?                                    type the interrupt key
$ echo $?                             print termination status of last command
130
$ sh -c "sleep 30"
^\sh: 962 Quit - core dumped          type the quit key
$ echo $?                             print termination status of last command
131
$ exit                                leave Bourne shell
```

On the system being used, SIGINT has a value of 2 and SIGQUIT has a value of 3, giving us the shell's termination statuses of 130 and 131.

Let's try a similar example, but this time we'll send a signal directly to the shell and see what gets returned by system:

```
$ tsys "sleep 30" &                        start it in background this time
9257
$ ps -f                                    look at the process IDs
     UID   PID  PPID  TTY        TIME CMD
     sar  9260   949  pts/5      0:00 ps -f
     sar  9258  9257  pts/5      0:00 sh -c sleep 60
     sar   949   947  pts/5      0:01 /bin/sh
     sar  9257   949  pts/5      0:00 tsys sleep 60
     sar  9259  9258  pts/5      0:00 sleep 60
$ kill -KILL 9258                          kill the shell itself
abnormal termination, signal number = 9
```

Here, we can see that the return value from system reports an abnormal termination only when the shell itself abnormally terminates.

When writing programs that use the system function, be sure to interpret the return value correctly. If you call fork, exec, and wait yourself, the termination status is not the same as if you call system.

10.19 sleep Function

We've used the sleep function in numerous examples throughout the text, and we showed two flawed implementations of it in Figures 10.7 and 10.8.

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

Returns: 0 or number of unslept seconds

This function causes the calling process to be suspended until either

1. The amount of wall clock time specified by *seconds* has elapsed.
2. A signal is caught by the process and the signal handler returns.

As with an alarm signal, the actual return may be at a time later than requested, because of other system activity.

In case 1, the return value is 0. When sleep returns early, because of some signal being caught (case 2), the return value is the number of unslept seconds (the requested time minus the actual time slept).

Although sleep can be implemented with the alarm function (Section 10.10), this isn't required. If alarm is used, however, there can be interactions between the two functions. The POSIX.1 standard leaves all these interactions unspecified. For example, if we do an alarm(10) and 3 wall clock seconds later do a sleep(5), what happens? The sleep will return in 5 seconds (assuming that some other signal isn't caught in that time), but will another SIGALRM be generated 2 seconds later? These details depend on the implementation.

Solaris 9 implements sleep using alarm. The Solaris sleep(3) manual page says that a previously scheduled alarm is properly handled. For example, in the preceding scenario, before sleep returns, it will reschedule the alarm to happen 2 seconds later; sleep returns 0 in this case. (Obviously, sleep must save the address of the signal handler for SIGALRM and reset it before returning.) Also, if we do an alarm(6) and 3 wall clock seconds later do a sleep(5), the sleep returns in 3 seconds (when the alarm goes off), not in 5 seconds. Here, the return value from sleep is 2 (the number of unslept seconds).

FreeBSD 5.2.1, Linux 2.4.22, and Mac OS X 10.3, on the other hand, use another technique: the delay is provided by nanosleep(2). This function is specified to be a high-resolution delay by the real-time extensions in the Single UNIX Specification. This function allows the implementation of sleep to be independent of signals.

For portability, you shouldn't make any assumptions about the implementation of sleep, but if you have any intentions of mixing calls to sleep with any other timing functions, you need to be aware of possible interactions.

Example

Figure 10.29 shows an implementation of the POSIX.1 sleep function. This function is a modification of Figure 10.7, which handles signals reliably, avoiding the race condition in the earlier implementation. We still do not handle any interactions with previously set alarms. (As we mentioned, these interactions are explicitly undefined by POSIX.1.)

```
#include "apue.h"

static void
sig_alrm(int signo)
{
    /* nothing to do, just returning wakes up sigsuspend() */
}

unsigned int
sleep(unsigned int nsecs)
{
    struct sigaction    newact, oldact;
    sigset_t            newmask, oldmask, suspmask;
    unsigned int        unslept;

    /* set our handler, save previous information */
    newact.sa_handler = sig_alrm;
    sigemptyset(&newact.sa_mask);
    newact.sa_flags = 0;
    sigaction(SIGALRM, &newact, &oldact);

    /* block SIGALRM and save current signal mask */
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGALRM);
    sigprocmask(SIG_BLOCK, &newmask, &oldmask);

    alarm(nsecs);

    suspmask = oldmask;
    sigdelset(&suspmask, SIGALRM);  /* make sure SIGALRM isn't blocked */
    sigsuspend(&suspmask);          /* wait for any signal to be caught */

    /* some signal has been caught, SIGALRM is now blocked */

    unslept = alarm(0);
    sigaction(SIGALRM, &oldact, NULL);  /* reset previous action */

    /* reset signal mask, which unblocks SIGALRM */
    sigprocmask(SIG_SETMASK, &oldmask, NULL);
    return(unslept);
}
```

Figure 10.29 Reliable implementation of `sleep`

It takes more code to write this reliable implementation than what is shown in Figure 10.7. We don't use any form of nonlocal branching (as we did in Figure 10.8 to avoid the race condition between the `alarm` and `pause`), so there is no effect on other signal handlers that may be executing when the `SIGALRM` is handled. □

10.20 Job-Control Signals

Of the signals shown in Figure 10.1, POSIX.1 considers six to be job-control signals:

`SIGCHLD`	Child process has stopped or terminated.
`SIGCONT`	Continue process, if stopped.
`SIGSTOP`	Stop signal (can't be caught or ignored).
`SIGTSTP`	Interactive stop signal.
`SIGTTIN`	Read from controlling terminal by member of a background process group.
`SIGTTOU`	Write to controlling terminal by member of a background process group.

Except for SIGCHLD, most application programs don't handle these signals: interactive shells usually do all the work required to handle these signals. When we type the suspend character (usually Control-Z), SIGTSTP is sent to all processes in the foreground process group. When we tell the shell to resume a job in the foreground or background, the shell sends all the processes in the job the SIGCONT signal. Similarly, if SIGTTIN or SIGTTOU is delivered to a process, the process is stopped by default, and the job-control shell recognizes this and notifies us.

An exception is a process that is managing the terminal: the vi(1) editor, for example. It needs to know when the user wants to suspend it, so that it can restore the terminal's state to the way it was when vi was started. Also, when it resumes in the foreground, the vi editor needs to set the terminal state back to the way it wants it, and it needs to redraw the terminal screen. We see how a program such as vi handles this in the example that follows.

There are some interactions between the job-control signals. When any of the four stop signals (SIGTSTP, SIGSTOP, SIGTTIN, or SIGTTOU) is generated for a process, any pending SIGCONT signal for that process is discarded. Similarly, when the SIGCONT signal is generated for a process, any pending stop signals for that same process are discarded.

Note that the default action for SIGCONT is to continue the process, if it is stopped; otherwise, the signal is ignored. Normally, we don't have to do anything with this signal. When SIGCONT is generated for a process that is stopped, the process is continued, even if the signal is blocked or ignored.

Example

The program in Figure 10.30 demonstrates the normal sequence of code used when a program handles job control. This program simply copies its standard input to its standard output, but comments are given in the signal handler for typical actions performed by a program that manages a screen. When the program in Figure 10.30 starts, it arranges to catch the SIGTSTP signal only if the signal's disposition is SIG_DFL. The reason is that when the program is started by a shell that doesn't support job control (/bin/sh, for example), the signal's disposition should be set to SIG_IGN. In fact, the shell doesn't explicitly ignore this signal; init sets the disposition of the three job-control signals (SIGTSTP, SIGTTIN, and SIGTTOU) to SIG_IGN. This disposition is then inherited by all login shells. Only a job-control shell should reset the disposition of these three signals to SIG_DFL.

When we type the suspend character, the process receives the SIGTSTP signal, and the signal handler is invoked. At this point, we would do any terminal-related processing: move the cursor to the lower-left corner, restore the terminal mode, and so on. We then send ourself the same signal, SIGTSTP, after resetting its disposition to its default (stop the process) and unblocking the signal. We have to unblock it since we're currently handling that same signal, and the system blocks it automatically while it's being caught. At this point, the system stops the process. It is continued only when it receives (usually from the job-control shell, in response to an interactive fg command) a

```
#include "apue.h"

#define BUFFSIZE    1024

static void sig_tstp(int);

int
main(void)
{
    int     n;
    char    buf[BUFFSIZE];

    /*
     * Only catch SIGTSTP if we're running with a job-control shell.
     */
    if (signal(SIGTSTP, SIG_IGN) == SIG_DFL)
        signal(SIGTSTP, sig_tstp);

    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");

    if (n < 0)
        err_sys("read error");

    exit(0);
}

static void
sig_tstp(int signo) /* signal handler for SIGTSTP */
{
    sigset_t    mask;

    /* ... move cursor to lower left corner, reset tty mode ... */

    /*
     * Unblock SIGTSTP, since it's blocked while we're handling it.
     */
    sigemptyset(&mask);
    sigaddset(&mask, SIGTSTP);
    sigprocmask(SIG_UNBLOCK, &mask, NULL);

    signal(SIGTSTP, SIG_DFL);    /* reset disposition to default */

    kill(getpid(), SIGTSTP);     /* and send the signal to ourself */

    /* we won't return from the kill until we're continued */

    signal(SIGTSTP, sig_tstp);   /* reestablish signal handler */

    /* ... reset tty mode, redraw screen ... */
}
```

Figure 10.30 How to handle SIGTSTP

SIGCONT signal. We don't catch SIGCONT. Its default disposition is to continue the stopped process; when this happens, the program continues as though it returned from the kill function. When the program is continued, we reset the disposition for the SIGTSTP signal and do whatever terminal processing we want (we could redraw the screen, for example). □

10.21 Additional Features

In this section, we describe some additional implementation-dependent features of signals.

Signal Names

Some systems provide the array

```
extern char *sys_siglist[];
```

The array index is the signal number, giving a pointer to the character string name of the signal.

> FreeBSD 5.2.1, Linux 2.4.22, and Mac OS X 10.3 all provide this array of signal names. Solaris 9 does, too, but it uses the name _sys_siglist instead.

These systems normally provide the function psignal also.

```
#include <signal.h>

void psignal(int signo, const char *msg);
```

The string *msg* (which is normally the name of the program) is output to the standard error, followed by a colon and a space, followed by a description of the signal, followed by a newline. This function is similar to perror (Section 1.7).

Another common function is strsignal. This function is similar to strerror (also described in Section 1.7).

```
#include <string.h>

char *strsignal(int signo);

                    Returns: a pointer to a string describing the signal
```

Given a signal number, strsignal will return a string that describes the signal. This string can be used by applications to print error messages about signals received.

> All the platforms discussed in this book provide the psignal and strsignal functions, but differences do occur. On Solaris 9, strsignal will return a null pointer if the signal number is invalid, whereas FreeBSD 5.2.1, Linux 2.4.22, and Mac OS X 10.3 return a string indicating that the signal number is unrecognized. Also, to get the function prototype for psignal on Solaris, you need to include <siginfo.h>.

Signal Mappings

Solaris provides a couple of functions to map a signal number to a signal name and vice versa.

```
#include <signal.h>

int sig2str(int signo, char *str);

int str2sig(const char *str, int *signop);
```

Both return: 0 if OK, −1 on error

These functions are useful when writing interactive programs that need to accept and print signal names and numbers.

The `sig2str` function translates the given signal number into a string and stores the result in the memory pointed to by *str*. The caller must ensure that the memory is large enough to hold the longest string, including the terminating null byte. Solaris provides the constant `SIG2STR_MAX` in `<signal.h>` to define the maximum string length. The string consists of the signal name without the "SIG" prefix. For example, translating `SIGKILL` would result in the string "KILL" being stored in the *str* memory buffer.

The `str2sig` function translates the given name into a signal number. The signal number is stored in the integer pointed to by *signop*. The name can be either the signal name without the "SIG" prefix or a string representation of the decimal signal number (i.e., "9").

Note that `sig2str` and `str2sig` depart from common practice and don't set `errno` when they fail.

10.22 Summary

Signals are used in most nontrivial applications. An understanding of the hows and whys of signal handling is essential to advanced UNIX System programming. This chapter has been a long and thorough look at UNIX System signals. We started by looking at the warts in previous implementations of signals and how they manifest themselves. We then proceeded to the POSIX.1 reliable-signal concept and all the related functions. Once we covered all these details, we were able to provide implementations of the POSIX.1 `abort`, `system`, and `sleep` functions. We finished with a look at the job-control signals and the ways that we can convert between signal names and signal numbers.

Exercises

10.1 In Figure 10.2, remove the `for (;;)` statement. What happens and why?

10.2 Implement the `sig2str` function described in Section 10.21.

10.3 Draw pictures of the stack frames when we run the program from Figure 10.9.

10.4 In Figure 10.11, we showed a technique that's often used to set a timeout on an I/O operation using `setjmp` and `longjmp`. The following code has also been seen:

```
signal(SIGALRM, sig_alrm);
alarm(60);
if (setjmp(env_alrm) != 0) {
    /* handle timeout */
    ...
}
...
```

What else is wrong with this sequence of code?

10.5 Using only a single timer (either `alarm` or the higher-precision `setitimer`), provide a set of functions that allows a process to set any number of timers.

10.6 Write the following program to test the parent–child synchronization functions in Figure 10.24. The process creates a file and writes the integer 0 to the file. The process then calls `fork`, and the parent and child alternate incrementing the counter in the file. Each time the counter is incremented, print which process (parent or child) is doing the increment.

10.7 In the function shown in Figure 10.25, if the caller catches `SIGABRT` and returns from the signal handler, why do we go to the trouble of resetting the disposition to its default and call `kill` the second time, instead of simply calling `_exit`?

10.8 Why do you think the `siginfo` structure (Section 10.14) includes the real user ID, instead of the effective user ID, in the `si_uid` field?

10.9 Rewrite the function in Figure 10.14 to handle all the signals from Figure 10.1. The function should consist of a single loop that iterates once for every signal in the current signal mask (not once for every possible signal).

10.10 Write a program that calls `sleep(60)` in an infinite loop. Every five times through the loop (every 5 minutes), fetch the current time of day and print the `tm_sec` field. Run the program overnight and explain the results. How would a program such as the BSD `cron` daemon, which runs every minute on the minute, handle this?

10.11 Modify Figure 3.4 as follows: (a) change `BUFFSIZE` to 100; (b) catch the `SIGXFSZ` signal using the `signal_intr` function, printing a message when it's caught, and returning from the signal handler; and (c) print the return value from `write` if the requested number of bytes weren't written. Modify the soft `RLIMIT_FSIZE` resource limit (Section 7.11) to 1,024 bytes and run your new program, copying a file that is larger than 1,024 bytes. (Try to set the soft resource limit from your shell. If you can't do this from your shell, call `setrlimit` directly from the program.) Run this program on the different systems that you have access to. What happens and why?

10.12 Write a program that calls `fwrite` with a large buffer (a few hundred megabytes). Before calling `fwrite`, call `alarm` to schedule a signal in 1 second. In your signal handler, print that the signal was caught and return. Does the call to `fwrite` complete? What's happening?

11

Threads

11.1 Introduction

We discussed processes in earlier chapters. We learned about the environment of a UNIX process, the relationships between processes, and ways to control processes. We saw that a limited amount of sharing can occur between related processes.

In this chapter, we'll look inside a process further to see how we can use multiple *threads of control* (or simply *threads*) to perform multiple tasks within the environment of a single process. All threads within a single process have access to the same process components, such as file descriptors and memory.

Any time you try to share a single resource among multiple users, you have to deal with consistency. We'll conclude the chapter with a look at the synchronization mechanisms available to prevent multiple threads from viewing inconsistencies in their shared resources.

11.2 Thread Concepts

A typical UNIX process can be thought of as having a single thread of control: each process is doing only one thing at a time. With multiple threads of control, we can design our programs to do more than one thing at a time within a single process, with each thread handling a separate task. This approach can have several benefits.

- We can simplify code that deals with asynchronous events by assigning a separate thread to handle each event type. Each thread can then handle its event using a synchronous programming model. A synchronous programming model is much simpler than an asynchronous one.

- Multiple processes have to use complex mechanisms provided by the operating system to share memory and file descriptors, as we will see in Chapters 15 and 17. Threads, on the other hand, automatically have access to the same memory address space and file descriptors.
- Some problems can be partitioned so that overall program throughput can be improved. A single process that has multiple tasks to perform implicitly serializes those tasks, because there is only one thread of control. With multiple threads of control, the processing of independent tasks can be interleaved by assigning a separate thread per task. Two tasks can be interleaved only if they don't depend on the processing performed by each other.
- Similarly, interactive programs can realize improved response time by using multiple threads to separate the portions of the program that deal with user input and output from the other parts of the program.

Some people associate multithreaded programming with multiprocessor systems. The benefits of a multithreaded programming model can be realized even if your program is running on a uniprocessor. A program can be simplified using threads regardless of the number of processors, because the number of processors doesn't affect the program structure. Furthermore, as long as your program has to block when serializing tasks, you can still see improvements in response time and throughput when running on a uniprocessor, because some threads might be able to run while others are blocked.

A thread consists of the information necessary to represent an execution context within a process. This includes a *thread ID* that identifies the thread within a process, a set of register values, a stack, a scheduling priority and policy, a signal mask, an `errno` variable (recall Section 1.7), and thread-specific data (Section 12.6). Everything within a process is sharable among the threads in a process, including the text of the executable program, the program's global and heap memory, the stacks, and the file descriptors.

The threads interface we're about to see is from POSIX.1-2001. The threads interface, also known as "pthreads" for "POSIX threads," is an optional feature in POSIX.1-2001. The feature test macro for POSIX threads is `_POSIX_THREADS`. Applications can either use this in an `#ifdef` test to determine at compile time whether threads are supported or call `sysconf` with the `_SC_THREADS` constant to determine at runtime whether threads are supported.

11.3 Thread Identification

Just as every process has a process ID, every thread has a thread ID. Unlike the process ID, which is unique in the system, the thread ID has significance only within the context of the process to which it belongs.

Recall that a process ID, represented by the `pid_t` data type, is a non-negative integer. A thread ID is represented by the `pthread_t` data type. Implementations are allowed to use a structure to represent the `pthread_t` data type, so portable implementations can't treat them as integers. Therefore, a function must be used to compare two thread IDs.

```
#include <pthread.h>

int pthread_equal(pthread_t tid1, pthread_t tid2);

                              Returns: nonzero if equal, 0 otherwise
```

Linux 2.4.22 uses an unsigned long integer for the `pthread_t` data type. Solaris 9 represents the `pthread_t` data type as an unsigned integer. FreeBSD 5.2.1 and Mac OS X 10.3 use a pointer to the `pthread` structure for the `pthread_t` data type.

A consequence of allowing the `pthread_t` data type to be a structure is that there is no portable way to print its value. Sometimes, it is useful to print thread IDs during program debugging, but there is usually no need to do so otherwise. At worst, this results in nonportable debug code, so it is not much of a limitation.

A thread can obtain its own thread ID by calling the `pthread_self` function.

```
#include <pthread.h>

pthread_t pthread_self(void);

                         Returns: the thread ID of the calling thread
```

This function can be used with `pthread_equal` when a thread needs to identify data structures that are tagged with its thread ID. For example, a master thread might place work assignments on a queue and use the thread ID to control which jobs go to each worker thread. This is illustrated in Figure 11.1. A single master thread places new jobs on a work queue. A pool of three worker threads removes jobs from the queue. Instead of allowing each thread to process whichever job is at the head of the queue, the master thread controls job assignment by placing the ID of the thread that should process the job in each job structure. Each worker thread then removes only jobs that are tagged with its own thread ID.

11.4 Thread Creation

The traditional UNIX process model supports only one thread of control per process. Conceptually, this is the same as a threads-based model whereby each process is made up of only one thread. With pthreads, when a program runs, it also starts out as a single process with a single thread of control. As the program runs, its behavior should be indistinguishable from the traditional process, until it creates more threads of control. Additional threads can be created by calling the `pthread_create` function.

```
#include <pthread.h>

int pthread_create(pthread_t *restrict tidp,
                   const pthread_attr_t *restrict attr,
                   void *(*start_rtn)(void), void *restrict arg);

                              Returns: 0 if OK, error number on failure
```

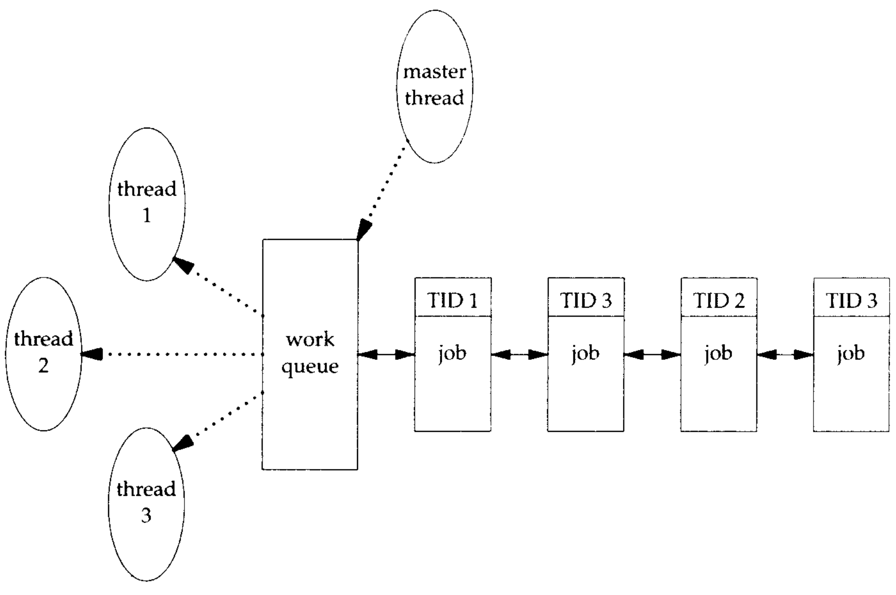

Figure 11.1 Work queue example

The memory location pointed to by *tidp* is set to the thread ID of the newly created thread when `pthread_create` returns successfully. The *attr* argument is used to customize various thread attributes. We'll cover thread attributes in Section 12.3, but for now, we'll set this to `NULL` to create a thread with the default attributes.

The newly created thread starts running at the address of the *start_rtn* function. This function takes a single argument, *arg*, which is a typeless pointer. If you need to pass more than one argument to the *start_rtn* function, then you need to store them in a structure and pass the address of the structure in *arg*.

When a thread is created, there is no guarantee which runs first: the newly created thread or the calling thread. The newly created thread has access to the process address space and inherits the calling thread's floating-point environment and signal mask; however, the set of pending signals for the thread is cleared.

Note that the pthread functions usually return an error code when they fail. They don't set `errno` like the other POSIX functions. The per thread copy of `errno` is provided only for compatibility with existing functions that use it. With threads, it is cleaner to return the error code from the function, thereby restricting the scope of the error to the function that caused it, instead of relying on some global state that is changed as a side effect of the function.

Example

Although there is no portable way to print the thread ID, we can write a small test program that does, to gain some insight into how threads work. The program in

Figure 11.2 creates one thread and prints the process and thread IDs of the new thread and the initial thread.

```
#include "apue.h"
#include <pthread.h>

pthread_t ntid;

void
printids(const char *s)
{
    pid_t       pid;
    pthread_t   tid;

    pid = getpid();
    tid = pthread_self();
    printf("%s pid %u tid %u (0x%x)\n", s, (unsigned int)pid,
      (unsigned int)tid, (unsigned int)tid);
}

void *
thr_fn(void *arg)
{
    printids("new thread: ");
    return((void *)0);
}

int
main(void)
{
    int     err;

    err = pthread_create(&ntid, NULL, thr_fn, NULL);
    if (err != 0)
        err_quit("can't create thread: %s\n", strerror(err));
    printids("main thread:");
    sleep(1);
    exit(0);
}
```

Figure 11.2 Printing thread IDs

This example has two oddities, necessary to handle races between the main thread and the new thread. (We'll learn better ways to deal with these later in this chapter.) The first is the need to sleep in the main thread. If it doesn't sleep, the main thread might exit, thereby terminating the entire process before the new thread gets a chance to run. This behavior is dependent on the operating system's threads implementation and scheduling algorithms.

The second oddity is that the new thread obtains its thread ID by calling `pthread_self` instead of reading it out of shared memory or receiving it as an argument to its thread-start routine. Recall that `pthread_create` will return the thread ID of the newly created thread through the first parameter (*tidp*). In our example, the main thread stores this in ntid, but the new thread can't safely use it. If the new thread runs before the main thread returns from calling pthread_create, then the new thread will see the uninitialized contents of ntid instead of the thread ID.

Running the program in Figure 11.2 on Solaris gives us

```
$ ./a.out
main thread: pid 7225 tid 1 (0x1)
new thread:  pid 7225 tid 4 (0x4)
```

As we expect, both threads have the same process ID, but different thread IDs. Running the program in Figure 11.2 on FreeBSD gives us

```
$ ./a.out
main thread: pid 14954 tid 134529024 (0x804c000)
new thread:  pid 14954 tid 134530048 (0x804c400)
```

As we expect, both threads have the same process ID. If we look at the thread IDs as decimal integers, the values look strange, but if we look at them in hexadecimal, they make more sense. As we noted earlier, FreeBSD uses a pointer to the thread data structure for its thread ID.

We would expect Mac OS X to be similar to FreeBSD; however, the thread ID for the main thread is from a different address range than the thread IDs for threads created with pthread_create:

```
$ ./a.out
main thread: pid 779 tid 2684396012 (0xa000a1ec)
new thread:  pid 779 tid 25166336 (0x1800200)
```

Running the same program on Linux gives us slightly different results:

```
$ ./a.out
new thread:  pid 6628 tid 1026 (0x402)
main thread: pid 6626 tid 1024 (0x400)
```

The Linux thread IDs look more reasonable, but the process IDs don't match. This is an artifact of the Linux threads implementation, where the clone system call is used to implement pthread_create. The clone system call creates a child process that can share a configurable amount of its parent's execution context, such as file descriptors and memory.

Note also that the output from the main thread appears before the output from the thread we create, except on Linux. This illustrates that we can't make any assumptions about how threads will be scheduled. □

11.5 Thread Termination

If any thread within a process calls exit, _Exit, or _exit, then the entire process terminates. Similarly, when the default action is to terminate the process, a signal sent to a thread will terminate the entire process (we'll talk more about the interactions between signals and threads in Section 12.8).

A single thread can exit in three ways, thereby stopping its flow of control, without terminating the entire process.

1. The thread can simply return from the start routine. The return value is the thread's exit code.
2. The thread can be canceled by another thread in the same process.
3. The thread can call `pthread_exit`.

```
#include <pthread.h>

void pthread_exit(void *rval_ptr);
```

The *rval_ptr* is a typeless pointer, similar to the single argument passed to the start routine. This pointer is available to other threads in the process by calling the `pthread_join` function.

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **rval_ptr);

                              Returns: 0 if OK, error number on failure
```

The calling thread will block until the specified thread calls `pthread_exit`, returns from its start routine, or is canceled. If the thread simply returned from its start routine, *rval_ptr* will contain the return code. If the thread was canceled, the memory location specified by *rval_ptr* is set to `PTHREAD_CANCELED`.

By calling `pthread_join`, we automatically place a thread in the detached state (discussed shortly) so that its resources can be recovered. If the thread was already in the detached state, calling `pthread_join` fails, returning `EINVAL`.

If we're not interested in a thread's return value, we can set *rval_ptr* to `NULL`. In this case, calling `pthread_join` allows us to wait for the specified thread, but does not retrieve the thread's termination status.

Example

Figure 11.3 shows how to fetch the exit code from a thread that has terminated.

```
#include "apue.h"
#include <pthread.h>

void *
thr_fn1(void *arg)
{
    printf("thread 1 returning\n");
    return((void *)1);
}

void *
thr_fn2(void *arg)
{
    printf("thread 2 exiting\n");
    pthread_exit((void *)2);
}

int
main(void)
{
    int         err;
    pthread_t   tid1, tid2;
    void        *tret;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
        err_quit("can't create thread 1: %s\n", strerror(err));
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
    if (err != 0)
        err_quit("can't create thread 2: %s\n", strerror(err));
    err = pthread_join(tid1, &tret);
    if (err != 0)
        err_quit("can't join with thread 1: %s\n", strerror(err));
    printf("thread 1 exit code %d\n", (int)tret);
    err = pthread_join(tid2, &tret);
    if (err != 0)
        err_quit("can't join with thread 2: %s\n", strerror(err));
    printf("thread 2 exit code %d\n", (int)tret);
    exit(0);
}
```

Figure 11.3 Fetching the thread exit status

Running the program in Figure 11.3 gives us

```
$ ./a.out
thread 1 returning
thread 2 exiting
thread 1 exit code 1
thread 2 exit code 2
```

As we can see, when a thread exits by calling `pthread_exit` or by simply returning from the start routine, the exit status can be obtained by another thread by calling `pthread_join`. □

The typeless pointer passed to `pthread_create` and `pthread_exit` can be used to pass more than a single value. The pointer can be used to pass the address of a structure containing more complex information. Be careful that the memory used for the structure is still valid when the caller has completed. If the structure was allocated on the caller's stack, for example, the memory contents might have changed by the time the structure is used. For example, if a thread allocates a structure on its stack and passes a pointer to this structure to `pthread_exit`, then the stack might be destroyed and its memory reused for something else by the time the caller of `pthread_join` tries to use it.

Example

The program in Figure 11.4 shows the problem with using an automatic variable (allocated on the stack) as the argument to pthread_exit.

```
#include "apue.h"
#include <pthread.h>

struct foo {
    int a, b, c, d;
};

void
printfoo(const char *s, const struct foo *fp)
{
    printf(s);
    printf("  structure at 0x%x\n", (unsigned)fp);
    printf("  foo.a = %d\n", fp->a);
    printf("  foo.b = %d\n", fp->b);
    printf("  foo.c = %d\n", fp->c);
    printf("  foo.d = %d\n", fp->d);
}

void *
thr_fn1(void *arg)
{
    struct foo  foo = {1, 2, 3, 4};

    printfoo("thread 1:\n", &foo);
    pthread_exit((void *)&foo);
}

void *
thr_fn2(void *arg)
{
    printf("thread 2: ID is %d\n", pthread_self());
    pthread_exit((void *)0);
}

int
main(void)
{
    int         err;
    pthread_t   tid1, tid2;
    struct foo  *fp;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
        err_quit("can't create thread 1: %s\n", strerror(err));
    err = pthread_join(tid1, (void *)&fp);
    if (err != 0)
        err_quit("can't join with thread 1: %s\n", strerror(err));
    sleep(1);
    printf("parent starting second thread\n");
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
    if (err != 0)
        err_quit("can't create thread 2: %s\n", strerror(err));
    sleep(1);
    printfoo("parent:\n", fp);
    exit(0);
}
```

Figure 11.4 Incorrect use of `pthread_exit` argument

When we run this program on Linux, we get

```
$ ./a.out
thread 1:
  structure at 0x409a2abc
  foo.a = 1
  foo.b = 2
  foo.c = 3
  foo.d = 4
parent starting second thread
thread 2: ID is 32770
parent:
  structure at 0x409a2abc
  foo.a = 0
  foo.b = 32770
  foo.c = 1075430560
  foo.d = 1073937284
```

Of course, the results vary, depending on the memory architecture, the compiler, and the implementation of the threads library. The results on FreeBSD are similar:

```
$ ./a.out
thread 1:
  structure at 0xbfafefc0
  foo.a = 1
  foo.b = 2
  foo.c = 3
  foo.d = 4
parent starting second thread
thread 2: ID is 134534144
parent:
  structure at 0xbfafefc0
  foo.a = 0
  foo.b = 134534144
  foo.c = 3
  foo.d = 671642590
```

As we can see, the contents of the structure (allocated on the stack of thread *tid1*) have changed by the time the main thread can access the structure. Note how the stack of the second thread (*tid2*) has overwritten the first thread's stack. To solve this problem, we can either use a global structure or allocate the structure using `malloc`. □

One thread can request that another in the same process be canceled by calling the `pthread_cancel` function.

```
#include <pthread.h>

int pthread_cancel(pthread_t tid);

                                   Returns: 0 if OK, error number on failure
```

In the default circumstances, `pthread_cancel` will cause the thread specified by *tid* to behave as if it had called `pthread_exit` with an argument of `PTHREAD_CANCELED`. However, a thread can elect to ignore or otherwise control how it is canceled. We will discuss this in detail in Section 12.7. Note that `pthread_cancel` doesn't wait for the thread to terminate. It merely makes the request.

A thread can arrange for functions to be called when it exits, similar to the way that the `atexit` function (Section 7.3) can be used by a process to arrange that functions can be called when the process exits. The functions are known as *thread cleanup handlers*. More than one cleanup handler can be established for a thread. The handlers are recorded in a stack, which means that they are executed in the reverse order from that with which they were registered.

```
#include <pthread.h>

void pthread_cleanup_push(void (*rtn)(void *), void *arg);

void pthread_cleanup_pop(int execute);
```

The `pthread_cleanup_push` function schedules the cleanup function, *rtn*, to be called with the single argument, *arg*, when the thread performs one of the following actions:

- Makes a call to `pthread_exit`
- Responds to a cancellation request
- Makes a call to `pthread_cleanup_pop` with a nonzero *execute* argument

If the *execute* argument is set to zero, the cleanup function is not called. In either case, `pthread_cleanup_pop` removes the cleanup handler established by the last call to `pthread_cleanup_push`.

A restriction with these functions is that, because they can be implemented as macros, they must be used in matched pairs within the same scope in a thread. The macro definition of `pthread_cleanup_push` can include a { character, in which case the matching } character is in the `pthread_cleanup_pop` definition.

Example

Figure 11.5 shows how to use thread cleanup handlers. Although the example is somewhat contrived, it illustrates the mechanics involved. Note that although we never intend to pass a nonzero argument to the thread start-up routines, we still need to match calls to `pthread_cleanup_pop` with the calls to `pthread_cleanup_push`; otherwise, the program might not compile.

```
#include "apue.h"
#include <pthread.h>

void
cleanup(void *arg)
{
    printf("cleanup: %s\n", (char *)arg);
}

void *
thr_fn1(void *arg)
{
    printf("thread 1 start\n");
    pthread_cleanup_push(cleanup, "thread 1 first handler");
    pthread_cleanup_push(cleanup, "thread 1 second handler");
    printf("thread 1 push complete\n");
    if (arg)
        return((void *)1);
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
    return((void *)1);
}

void *
thr_fn2(void *arg)
{
    printf("thread 2 start\n");
    pthread_cleanup_push(cleanup, "thread 2 first handler");
    pthread_cleanup_push(cleanup, "thread 2 second handler");
    printf("thread 2 push complete\n");
    if (arg)
        pthread_exit((void *)2);
    pthread_cleanup_pop(0);
    pthread_cleanup_pop(0);
    pthread_exit((void *)2);
}

int
main(void)
{
    int         err;
    pthread_t   tid1, tid2;
    void        *tret;

    err = pthread_create(&tid1, NULL, thr_fn1, (void *)1);
    if (err != 0)
        err_quit("can't create thread 1: %s\n", strerror(err));
    err = pthread_create(&tid2, NULL, thr_fn2, (void *)1);
    if (err != 0)
        err_quit("can't create thread 2: %s\n", strerror(err));
    err = pthread_join(tid1, &tret);
    if (err != 0)
        err_quit("can't join with thread 1: %s\n", strerror(err));
    printf("thread 1 exit code %d\n", (int)tret);
    err = pthread_join(tid2, &tret);
    if (err != 0)
        err_quit("can't join with thread 2: %s\n", strerror(err));
    printf("thread 2 exit code %d\n", (int)tret);
    exit(0);
}
```

Figure 11.5 Thread cleanup handler

Running the program in Figure 11.5 gives us

```
$ ./a.out
thread 1 start
thread 1 push complete
thread 2 start
thread 2 push complete
cleanup: thread 2 second handler
cleanup: thread 2 first handler
thread 1 exit code 1
thread 2 exit code 2
```

From the output, we can see that both threads start properly and exit, but that only the second thread's cleanup handlers are called. Thus, if the thread terminates by returning from its start routine, its cleanup handlers are not called. Also note that the cleanup handlers are called in the reverse order from which they were installed. □

By now, you should begin to see similarities between the thread functions and the process functions. Figure 11.6 summarizes the similar functions.

Process primitive	Thread primitive	Description
`fork`	`pthread_create`	create a new flow of control
`exit`	`pthread_exit`	exit from an existing flow of control
`waitpid`	`pthread_join`	get exit status from flow of control
`atexit`	`pthread_cancel_push`	register function to be called at exit from flow of control
`getpid`	`pthread_self`	get ID for flow of control
`abort`	`pthread_cancel`	request abnormal termination of flow of control

Figure 11.6 Comparison of process and thread primitives

By default, a thread's termination status is retained until `pthread_join` is called for that thread. A thread's underlying storage can be reclaimed immediately on termination if that thread has been *detached*. When a thread is detached, the `pthread_join` function can't be used to wait for its termination status. A call to `pthread_join` for a detached thread will fail, returning `EINVAL`. We can detach a thread by calling `pthread_detach`.

```
#include <pthread.h>

int pthread_detach(pthread_t tid);

                        Returns: 0 if OK, error number on failure
```

As we will see in the next chapter, we can create a thread that is already in the detached state by modifying the thread attributes we pass to `pthread_create`.

11.6 Thread Synchronization

When multiple threads of control share the same memory, we need to make sure that each thread sees a consistent view of its data. If each thread uses variables that other threads don't read or modify, no consistency problems exist. Similarly, if a variable is read-only, there is no consistency problem with more than one thread reading its value at the same time. However, when one thread can modify a variable that other threads can read or modify, we need to synchronize the threads to ensure that they don't use an invalid value when accessing the variable's memory contents.

When one thread modifies a variable, other threads can potentially see inconsistencies when reading the value of the variable. On processor architectures in which the modification takes more than one memory cycle, this can happen when the memory read is interleaved between the memory write cycles. Of course, this behavior is architecture dependent, but portable programs can't make any assumptions about what type of processor architecture is being used.

Figure 11.7 shows a hypothetical example of two threads reading and writing the same variable. In this example, thread A reads the variable and then writes a new value to it, but the write operation takes two memory cycles. If thread B reads the same variable between the two write cycles, it will see an inconsistent value.

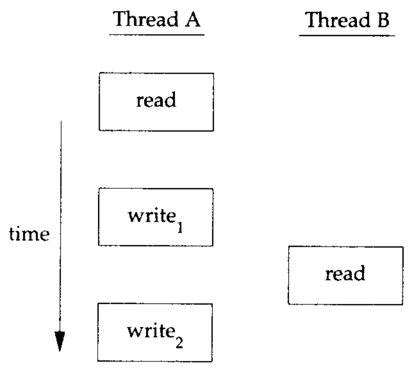

Figure 11.7 Interleaved memory cycles with two threads

To solve this problem, the threads have to use a lock that will allow only one thread to access the variable at a time. Figure 11.8 shows this synchronization. If it wants to read the variable, thread B acquires a lock. Similarly, when thread A updates the variable, it acquires the same lock. Thus, thread B will be unable to read the variable until thread A releases the lock.

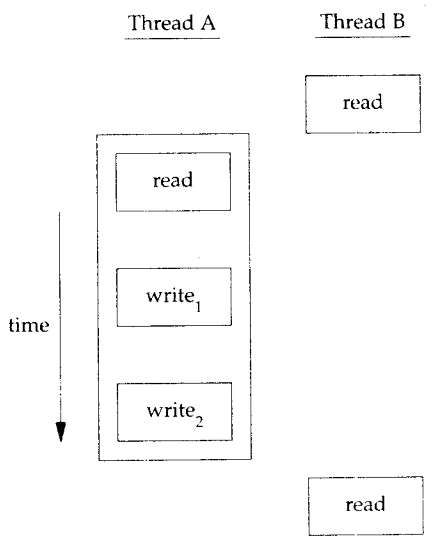

Figure 11.8 Two threads synchronizing memory access

You also need to synchronize two or more threads that might try to modify the same variable at the same time. Consider the case in which you increment a variable (Figure 11.9). The increment operation is usually broken down into three steps.

1. Read the memory location into a register.
2. Increment the value in the register.
3. Write the new value back to the memory location.

If two threads try to increment the same variable at almost the same time without synchronizing with each other, the results can be inconsistent. You end up with a value that is either one or two greater than before, depending on the value observed when the second thread starts its operation. If the second thread performs step 1 before the first thread performs step 3, the second thread will read the same initial value as the first thread, increment it, and write it back, with no net effect.

If the modification is atomic, then there isn't a race. In the previous example, if the increment takes only one memory cycle, then no race exists. If our data always appears to be *sequentially consistent*, then we need no additional synchronization. Our operations are sequentially consistent when multiple threads can't observe inconsistencies in our data. In modern computer systems, memory accesses take multiple bus cycles, and multiprocessors generally interleave bus cycles among multiple processors, so we aren't guaranteed that our data is sequentially consistent.

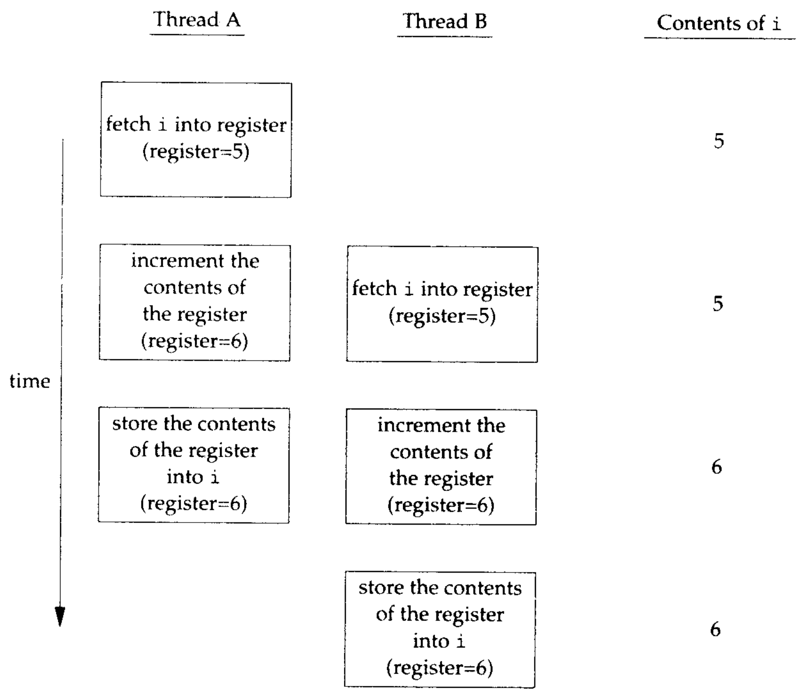
Thread A
Thread B
Contents of i
fetch i into register (register=5)
5
increment the contents of the register (register=6)
fetch i into register (register=5)
5
time
store the contents of the register into i (register=6)
increment the contents of the register (register=6)
6
store the contents of the register into i (register=6)
6

Figure 11.9 Two unsynchronized threads incrementing the same variable

In a sequentially consistent environment, we can explain modifications to our data as a sequential step of operations taken by the running threads. We can say such things as "Thread A incremented the variable, then thread B incremented the variable, so its value is two greater than before" or "Thread B incremented the variable, then thread A incremented the variable, so its value is two greater than before." No possible ordering of the two threads can result in any other value of the variable.

Besides the computer architecture, races can arise from the ways in which our programs use variables, creating places where it is possible to view inconsistencies. For example, we might increment a variable and then make a decision based on its value. The combination of the increment step and the decision-making step aren't atomic, so this opens a window where inconsistencies can arise.

Mutexes

We can protect our data and ensure access by only one thread at a time by using the pthreads mutual-exclusion interfaces. A *mutex* is basically a lock that we set (lock) before accessing a shared resource and release (unlock) when we're done. While it is set, any other thread that tries to set it will block until we release it. If more than one thread is blocked when we unlock the mutex, then all threads blocked on the lock will be made runnable, and the first one to run will be able to set the lock. The others will see that the mutex is still locked and go back to waiting for it to become available again. In this way, only one thread will proceed at a time.

This mutual-exclusion mechanism works only if we design our threads to follow the same data-access rules. The operating system doesn't serialize access to data for us. If we allow one thread to access a shared resource without first acquiring a lock, then inconsistencies can occur even though the rest of our threads do acquire the lock before attempting to access the shared resource.

A mutex variable is represented by the `pthread_mutex_t` data type. Before we can use a mutex variable, we must first initialize it by either setting it to the constant `PTHREAD_MUTEX_INITIALIZER` (for statically-allocated mutexes only) or calling `pthread_mutex_init`. If we allocate the mutex dynamically (by calling `malloc`, for example), then we need to call `pthread_mutex_destroy` before freeing the memory.

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);

int pthread_mutex_destroy(pthread_mutex_t *mutex);

                          Both return: 0 if OK, error number on failure
```

To initialize a mutex with the default attributes, we set *attr* to `NULL`. We will discuss nondefault mutex attributes in Section 12.4.

To lock a mutex, we call `pthread_mutex_lock`. If the mutex is already locked, the calling thread will block until the mutex is unlocked. To unlock a mutex, we call `pthread_mutex_unlock`.

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);

                           All return: 0 if OK, error number on failure
```

If a thread can't afford to block, it can use `pthread_mutex_trylock` to lock the mutex conditionally. If the mutex is unlocked at the time `pthread_mutex_trylock` is called, then `pthread_mutex_trylock` will lock the mutex without blocking and return 0. Otherwise, `pthread_mutex_trylock` will fail, returning `EBUSY` without locking the mutex.

Example

Figure 11.10 illustrates a mutex used to protect a data structure. When more than one thread needs to access a dynamically-allocated object, we can embed a reference count in the object to ensure that we don't free its memory before all threads are done using it.

```
#include <stdlib.h>
#include <pthread.h>

struct foo {
    int             f_count;
    pthread_mutex_t f_lock;
    /* ... more stuff here ... */
};

struct foo *
foo_alloc(void) /* allocate the object */
{
    struct foo *fp;

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        /* ... continue initialization ... */
    }
    return(fp);
}

void
foo_hold(struct foo *fp) /* add a reference to the object */
{
    pthread_mutex_lock(&fp->f_lock);
    fp->f_count++;
    pthread_mutex_unlock(&fp->f_lock);
}

void
foo_rele(struct foo *fp) /* release a reference to the object */
{
    pthread_mutex_lock(&fp->f_lock);
    if (--fp->f_count == 0) { /* last reference */
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        pthread_mutex_unlock(&fp->f_lock);
    }
}
```

Figure 11.10 Using a mutex to protect a data structure

We lock the mutex before incrementing the reference count, decrementing the reference count, and checking whether the reference count reaches zero. No locking is necessary when we initialize the reference count to 1 in the foo_alloc function, because the allocating thread is the only reference to it so far. If we were to place the structure on a list at this point, it could be found by other threads, so we would need to lock it first.

Before using the object, threads are expected to add a reference count to it. When they are done, they must release the reference. When the last reference is released, the object's memory is freed. □

Deadlock Avoidance

A thread will deadlock itself if it tries to lock the same mutex twice, but there are less obvious ways to create deadlocks with mutexes. For example, when we use more than one mutex in our programs, a deadlock can occur if we allow one thread to hold a mutex and block while trying to lock a second mutex at the same time that another thread holding the second mutex tries to lock the first mutex. Neither thread can proceed, because each needs a resource that is held by the other, so we have a deadlock.

Deadlocks can be avoided by carefully controlling the order in which mutexes are locked. For example, assume that you have two mutexes, A and B, that you need to lock at the same time. If all threads always lock mutex A before mutex B, no deadlock can occur from the use of the two mutexes (but you can still deadlock on other resources). Similarly, if all threads always lock mutex B before mutex A, no deadlock will occur. You'll have the potential for a deadlock only when one thread attempts to lock the mutexes in the opposite order from another thread.

Sometimes, an application's architecture makes it difficult to apply a lock ordering. If enough locks and data structures are involved that the functions you have available can't be molded to fit a simple hierarchy, then you'll have to try some other approach. In this case, you might be able to release your locks and try again at a later time. You can use the `pthread_mutex_trylock` interface to avoid deadlocking in this case. If you are already holding locks and `pthread_mutex_trylock` is successful, then you can proceed. If it can't acquire the lock, however, you can release the locks you already hold, clean up, and try again later.

Example

In this example, we update Figure 11.10 to show the use of two mutexes. We avoid deadlocks by ensuring that when we need to acquire two mutexes at the same time, we always lock them in the same order. The second mutex protects a hash list that we use to keep track of the `foo` data structures. Thus, the `hashlock` mutex protects both the `fh` hash table and the `f_next` hash link field in the `foo` structure. The `f_lock` mutex in the `foo` structure protects access to the remainder of the `foo` structure's fields.

```
#include <stdlib.h>
#include <pthread.h>

#define NHASH 29
#define HASH(fp) (((unsigned long)fp)%NHASH)
```

```
struct foo *fh[NHASH];

pthread_mutex_t hashlock = PTHREAD_MUTEX_INITIALIZER;

struct foo {
    int             f_count;
    pthread_mutex_t f_lock;
    struct foo     *f_next; /* protected by hashlock */
    int             f_id;
    /* ... more stuff here ... */
};

struct foo *
foo_alloc(void) /* allocate the object */
{
    struct foo  *fp;
    int         idx;

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        idx = HASH(fp);
        pthread_mutex_lock(&hashlock);
        fp->f_next = fh[idx];
        fh[idx] = fp->f_next;
        pthread_mutex_lock(&fp->f_lock);
        pthread_mutex_unlock(&hashlock);
        /* ... continue initialization ... */
        pthread_mutex_unlock(&fp->f_lock);
    }
    return(fp);
}

void
foo_hold(struct foo *fp) /* add a reference to the object */
{
    pthread_mutex_lock(&fp->f_lock);
    fp->f_count++;
    pthread_mutex_unlock(&fp->f_lock);
}

struct foo *
foo_find(int id) /* find an existing object */
{
    struct foo  *fp;
    int         idx;

    idx = HASH(fp);
    pthread_mutex_lock(&hashlock);
    for (fp = fh[idx]; fp != NULL; fp = fp->f_next) {
        if (fp->f_id == id) {
            foo_hold(fp);
            break;
        }
    }
    pthread_mutex_unlock(&hashlock);
    return(fp);
}

void
foo_rele(struct foo *fp) /* release a reference to the object */
{
    struct foo  *tfp;
    int         idx;

    pthread_mutex_lock(&fp->f_lock);
    if (fp->f_count == 1) { /* last reference */
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_lock(&hashlock);
        pthread_mutex_lock(&fp->f_lock);
        /* need to recheck the condition */
        if (fp->f_count != 1) {
            fp->f_count--;
            pthread_mutex_unlock(&fp->f_lock);
            pthread_mutex_unlock(&hashlock);
            return;
        }
        /* remove from list */
        idx = HASH(fp);
        tfp = fh[idx];
        if (tfp == fp) {
            fh[idx] = fp->f_next;
        } else {
            while (tfp->f_next != fp)
                tfp = tfp->f_next;
            tfp->f_next = fp->f_next;
        }
        pthread_mutex_unlock(&hashlock);
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        fp->f_count--;
        pthread_mutex_unlock(&fp->f_lock);
    }
}
```

Figure 11.11 Using two mutexes

Comparing Figure 11.11 with Figure 11.10, we see that our allocation function now locks the hash list lock, adds the new structure to a hash bucket, and before unlocking the hash list lock, locks the mutex in the new structure. Since the new structure is placed on a global list, other threads can find it, so we need to block them if they try to access the new structure, until we are done initializing it.

The `foo_find` function locks the hash list lock and searches for the requested structure. If it is found, we increase the reference count and return a pointer to the structure. Note that we honor the lock ordering by locking the hash list lock in `foo_find` before `foo_hold` locks the `foo` structure's `f_lock` mutex.

Now with two locks, the `foo_rele` function is more complicated. If this is the last reference, we need to unlock the structure mutex so that we can acquire the hash list lock, since we'll need to remove the structure from the hash list. Then we reacquire the structure mutex. Because we could have blocked since the last time we held the structure mutex, we need to recheck the condition to see whether we still need to free the structure. If another thread found the structure and added a reference to it while we blocked to honor the lock ordering, we simply need to decrement the reference count, unlock everything, and return.

This locking is complex, so we need to revisit our design. We can simplify things considerably by using the hash list lock to protect the structure reference count, too. The structure mutex can be used to protect everything else in the `foo` structure. Figure 11.12 reflects this change.

```
#include <stdlib.h>
#include <pthread.h>

#define NHASH 29
#define HASH(fp) (((unsigned long)fp)%NHASH)

struct foo *fh[NHASH];
pthread_mutex_t hashlock = PTHREAD_MUTEX_INITIALIZER;

struct foo {
    int             f_count; /* protected by hashlock */
    pthread_mutex_t f_lock;
    struct foo     *f_next; /* protected by hashlock */
    int             f_id;
    /* ... more stuff here ... */
};

struct foo *
foo_alloc(void) /* allocate the object */
{
    struct foo  *fp;
    int         idx;

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
```