NITPulse

NAVODAYA INSTITUTE OF TECHNOLOGY, RAICHUR

DEPARMENT OF COMPUTER SCIENCE & ENGINEERING

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—for example, 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 SignalConcepts

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 had 31 different signals. FreeBSD 8.0 supports 32 different signals. Mac OS X 10.6.8 and

Linux 3.2.0 each support 31 different signals, whereas Solaris 10 supports 40 different signals. FreeBSD, Linux, and Solaris, however, support additional application-defined signals introduced to support real-time applications. Although the POSIX real-time extensions aren't covered in this book (refer to Gallmeister [1995] for more information), as of SUSv4 the real-time signal interfaces have moved to the base specification.

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

> Implementations actually define the individual signals in a different 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 8.0 and Mac OS X 10.6.8 define the signals in <sys/signal.h>. Linux 3.2.0 defines the signals in <bits/signum.h>, and Solaris 10 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 a process should be notified of various events. These aren't hardware-generated conditions (as is the divideby-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 that

NTPulse

has no reader), and SIGALRM (generated when an alarm clock set by the process expires).

Signals are classic examples of asynchronous events. They 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 beignored: 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 part of the XSI option.

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 option.
>
> The name of the core file varies among implementations. On FreeBSD 8.0, 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 10.6.8, 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. On Linux 3.2.0, the name is configured through /proc/sys/kernel/core_pattern.)

Not to be republished

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 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	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
SIGJVM1	Java virtual machine internal use						•	ignore
SIGJVM2	Java virtual machine internal use						•	ignore
SIGKILL	termination		•	•	•	•	•	terminate
SIGLOST	resource lost						•	terminate
SIGLWP	threads library internal use			•			•	terminate/ignore
SIGPIPE	write to pipe with no readers		•	•	•	•	•	terminate
SIGPOLL	pollable event (`poll`)				•		•	terminate
SIGPROF	profiling time alarm (`setitimer`)			•	•	•	•	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
SIGTHR	threads library internal use			•				terminate
SIGTRAP	hardware fault		XSI	•	•	•	•	terminate+core
SIGTSTP	terminal stop character		•	•	•	•	•	stop process

NTPulse

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 or terminate+core
SIGXFSZ	file size limit exceeded (setrlimit)	XSI	•	•	•	•	terminate or terminate+core
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, (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 of ''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 which 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 signal indicates an implementation-defined hardware fault. Implementations usually generate this signal on certain types of memory faults, as we describe in Section 14.8.

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.

NIPTSE

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 which 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 10.21 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 9.13, 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 task is that a daemon should not have a controlling terminal and would normally never receive this signal.

NTPulse

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 purpose.

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 9.9). 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, although the symbol is defined to be the same value as SIGPWR on the Alpha platform. This is most likely to provide some level of compatibility with software developed for OSF/1.

SIGINT This signal is generated by the terminal driver when we press the interrupt key (often DELETE or Control-C). This signal is sent to all processes in the foreground process group (refer to Figure 9.9). 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 14.5.2.

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 3.2.0 and Solaris 10 define SIGIO to be the same value as SIGPOLL, so the default behavior is to terminate the process. On FreeBSD 8.0 and Mac OS X 10.6.8, 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 purpose.

On FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10, SIGIOT is defined to be the same value as SIGABRT.

SIGJVM1 A signal reserved for use by the Java virtual machine on Solaris.

SIGJVM2 Another signal reserved for use by the Java virtual machine on Solaris.

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.

SIGLOST This signal is used to notify a process running on a Solaris NFSv4 client system that a lock could not be reacquired during recovery.

SIGLWP This signal is used internally by the Solaris threads library; it is not available for general use. On FreeBSD, SIGLWP is defined to be an alias for SIGTHR.

NIIT Pilso

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 is marked obsolescent in SUSv4, so it might be removed in a future version of the standard. It can be generated when a specific event occurs on a pollable device. We describe this signal with the poll function in Section 14.4.2. 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 marked obsolescent in SUSv4, so it might be removed in a future version of the standard. 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 (for example, 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. This is when SIGPWR should be sent. On most systems, the process that is notified of the low-battery condition sends the SIGPWR signal to the init process, and init handles the system shutdown.

Solaris 10 and some Linux distributions 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 press the terminal quit key (often Control-backslash). This signal is sent to all processes in the foreground process group (refer to Figure 9.9). 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 (which is usually a sign that the program has a bug, such as dereferencing an uninitialized pointer).

The name SEGV stands for ''segmentation violation.''

SIGSTKFLT This signal is defined only by Linux. It showed up in the earliest versions of Linux, where it was 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.

NIIT Use

`SIGSTOP` This job-control signal stops a process. It is similar to 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. Because it can be caught by applications, using `SIGTERM` gives programs a chance to terminate gracefully by cleaning up before exiting (in contrast to `SIGKILL`, which can't be caught or ignored).

`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.

`SIGTHR` This is a signal reserved for use by the thread library on FreeBSD. It is defined to have the same value as `SIGLWP`.

`SIGTRAP` This signal 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 press the terminal suspend key (often Control-Z). This signal is sent to all processes in the foreground process group (refer to Figure 9.9).

> 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 fails 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. (This is discussed in Section 9.8.) Unlike the case with background reads, a process can choose to allow background writes to the controlling terminal. We describe how to modify this option in Chapter 18.

NIIPulse

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. These include 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. It 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 part of the XSI option; 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'' or ''terminate with a core file.'' The default depends on the operating system. Linux 3.2.0 and Solaris 10 support a default action of terminate with a core file, whereas FreeBSD 8.0 and Mac OS X 10.6.8 support a default action of terminate without generating a core file. 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 3.2.0 and Solaris 10, the default is to terminate the process and create a core file. On FreeBSD 8.0 and Mac OS X 10.6.8, the default is to terminate the process without generating a core file. 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.

NTPulse

`SIGXRES` This signal is defined only by Solaris. It 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.

NITPulse

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.) The signal 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. Most current systems follow this strategy, but Solaris 10 follows the System V semantics for the signal function.

Because the semantics of signal differ among implementations, we must use the sigaction function instead. We provide an implementation of signal that uses sigaction in Section 10.14. All the examples in this text use the signal function from Figure 10.18 to give us consistent semantics regardless of which particular platform we use.

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

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 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 signalcatching function in the caller probably has no meaning in the new program file that is executed.)

One specific example of this signal status behavior 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 done 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);`

Following this approach, 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 was 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 was 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 interrupting me, 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();   /* my signal handling function */ int
sig_int_flag;    /* set nonzero when signal occurs */ main()
{ 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 was 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 on certain file types that block the caller until some condition occurs (such as a terminal device open waiting 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 `read`s and `write`s changed with the 2001 version of the standard. Earlier versions gave implementations a choice of how to deal with `read`s and `write`s 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.

NTPulse

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 requires an implementation to restart system calls only when the `SA_RESTART` flag is in effect for the interrupting signal. As we will see in Section 10.14, this flag is used with the `sigaction` function to allow applications to request that interrupted system calls be restarted.
>
> Historically, when using the `signal` function to establish a signal handler, implementations varied with respect to how interrupted system calls were handled. System V never restarted system calls by default. BSD, in contrast, restarted them if the calls were interrupted by signals. On FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8, when signal handlers are installed with the `signal` function, interrupted system calls will be restarted. The default on Solaris 10, however, is to return an error (`EINTR`) instead when system calls are interrupted by signal handlers installed with the `signal` function. By using our own implementation of the `signal` function (shown in Figure 10.18), we avoid having to deal with these differences.

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.

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			never

	SVR4, Solaris			never
	4.2BSD	•	•	always
	4.3BSD, 4.4BSD, FreeBSD, Linux, Mac OS X	•	•	default
sigaction	POSIX.1, 4.4BSD, SVR4, FreeBSD, Linux, Mac OS X, 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, unlike 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 14.4 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 safe to call from within a signal handler. These functions are reentrant and are called *async-signal safe* by the Single UNIX Specification. Besides being reentrant, they block any signals during operation if delivery of a signal might cause inconsistencies. Figure 10.4 lists these async-signal safe functions. Most of the functions that are not included 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 handler 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 potentially 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

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

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

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.

NITPulse

```
my_alarm(int signo)


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

static void

{
    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 when `getpwnam` called `free`, the signal handler interrupted it and called `getpwnam`, which in turn called `free`. The data structures maintained by `malloc` and `free` had been corrupted when the signal handler (indirectly) called `free` while the `main` function was also calling `free`. 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.

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`. The name `SIGCLD` (without the `H`) is from System V, and this signal has different semantics from the BSD signal, named `SIGCHLD`. The POSIX.1 signal is also named `SIGCHLD`.

VTUPulse

The semantics of the BSD SIGCHLD signal are normal, in the sense 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 behavior:

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 XSI option requires this behavior to 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. With SVR4, if either signal or sigset is called to set the disposition of SIGCHLD to be ignored, zombies are never generated. All four platforms described in this book follow SVR4 in this behavior.

   With sigaction, we can set the SA_NOCLDWAIT flag (Figure 10.16) to avoid zombies. This action is also supported on all four platforms.

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 is intended 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

NIPRISE

doesn't work on traditional System V platforms. The output is a continual string of `SIGCLD received` lines. Eventually, the process runs out of stack space and terminates abnormally.

NITPulse

Although the four platforms described in this book solve this problem, realize that platforms (such as AIX) still exist that haven't addressed it.

```
#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

FreeBSD 8.0 and Mac OS X 10.6.8 don't exhibit this problem, because BSD-based systems generally don't support historical System V semantics for `SIGCLD`. Linux 3.2.0 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 10, on the other hand, does call the signal handler in this situation, but includes extra code in the kernel to avoid this problem.

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 is the case, 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.

> Of the four platforms described in this text, only Linux 3.2.0 and Solaris 10 define `SIGCLD`. On these platforms, `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

NTPulse

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.

> With SUSv4, the real-time signal functionality moved from the real-time extensions to the base specification. As time goes on, more systems will support queueing signals even if they don't support the real-time extensions. We discuss queueing signals further in Section 10.20.
>
> 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 didn'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.

> The `raise` function 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.

NOT FOR USE

```
#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. One special case for the permission testing also exists: 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 technique 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 necessarily mean that it's the process that you think it is.

NIIPulse

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. When that time occurs, the signal is generated by the kernel, although additional time could elapse 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 behavior.

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 seconds)
{ if (signal(SIGALRM, sig_alrm) == SIG_ERR)
    return(seconds);
    alarm(seconds);        /* 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 alarm's return value. If the number of seconds until some previously set alarm is less than the argument, then we should wait only until the existing 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 our function is called and restore it when we're done. We can correct this by saving the return value from signal and resetting the disposition before our function returns.

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).

NKPulse

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_bufenv_alrm;

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

unsigned int
sleep2(unsigned int seconds)
{
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        return(seconds);
    if (setjmp(env_alrm) == 0) {
        alarm(seconds);      /*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, then 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 as `volatile` to prevent an optimizing compiler from discarding the loop.

NTPulse

```
#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
     *onwhatever 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

When we execute the program shown in Figure 10.9 and interrupt the sleep by typing the interrupt character, we get the following output:

```
$ ./a.out
^C                              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. □

The purpose of the `sleep1` and `sleep2` examples 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.

NTUPulse

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.

NIPulse

Here, we specifically want a slow system call to be interrupted. We'll see a portable

If we want to set a time limit on an I/O operation, we need to use longjmp, as shown

way to do this 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_bufenv_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. □

previously, while recognizing its possible interaction with other signal handlers. Another option is to use the select or poll functions, described in Sections 14.4.1 and 14.4.2.

NTPulse

10.11 Signal Sets

We need a data type to represent multiple signals—a *signal set*. We'll use this data type 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.

NTUPulse

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

```
#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`

from the signal number to obtain the bit to manipulate. Figure 10.12 shows implementations of these functions.

NTPulse

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

```
#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_ret("sigprocmask error");
    } else {
        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;     /* restore errno */
}
```

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.

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

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.

NTPulse

–

NITPulse

```
#include <signal.h> int

sigpending(sigset_t *set);
```

Returns: 0 if OK, −1 on error

NITPulse

Example

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

Figure 10.15 Example of signal sets and `sigprocmask`

```
#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");

    /*
     *Restore 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");
}
```

NTPulse

The process blocks SIGQUIT, saving its current signal mask (to restore 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 restore 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 either of 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. (This characteristic was illustrated in the previous example.)

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 part of the XSI option.

The `sa_sigaction` field is an alternative 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.

NEU Pulse

Option	SUS	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10	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). When the XSI option is supported, SIGCHLD won't be sent when a stopped child continues if this flag is set.
SA_NOCLDWAIT	•	•	•	•	•	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	•	•	•	•	•	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 alternative stack has been declared with sigaltstack(2), this signal is delivered to the process on the alternative stack.
SA_RESETHAND	•	•	•	•	•	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 can optionally cause sigaction to behave as if SA_NODEFER is also set.
SA_RESTART	•	•	•	•	•	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

NTPulse

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 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 */ union sigval si_value; /* application-specific
  value */
  /* possibly other fields also */
};
```

The sigval union contains the following fields:

```
int sival_int; void
*sival_ptr;
```

Applications pass an integer value in si_value.sival_int or pass a pointer value in si_value.sival_ptr when delivering signals.

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.

If the signal is SIGCHLD, then the si_pid, si_status, and si_uid fields will be set. If the signal is SIGBUS, SIGILL, SIGFPE, or SIGSEGV, then the si_addr contains the address responsible for the fault, although the address might not be accurate. 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. This structure contains at least the following fields:

```
ucontext_t *uc_link;     /* pointer to context resumed when */
                         /* this context returns */
sigset_t uc_sigmask; /* signals blocked when this context */
                         /* is active */
stack_t   uc_stack;     /* stack used by this context */ mcontext_t
uc_mcontext; /* machine-specific representation of */
```

VTUPulse

```
                                /* saved context */
```

The `uc_stack` field describes the stack used by the current context. It contains at least the

```
void *ss_sp;     /* stack base or pointer */
size_t ss_size;/* stack size */
int ss_flags;/* flags */
```

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 signal numbers is available for real-time application use. Applications can pass information along with the signal by using the `sigqueue` function (Section 10.20).

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
Any	SI_USER	signal sent by `kill`
	SI_QUEUE	signal sent by `sigqueue`
	SI_TIMER	expiration of a timer set by `timer_settime`
	SI_ASYNCIO	completion of asynchronous I/O request
	SI_MESGQ	arrival of a message on a message queue (real-time extension)

Figure 10.17 `siginfo_t` code values

following members:

NIIT Use

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 { act.sa_flags |=
        SA_RESTART;
    }
    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 set the SA_RESTART flag for all signals other than SIGALRM, so that any system call interrupted by these other signals will be 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 by default does not restart system calls when the signal handler is installed with sigaction. The Single UNIX Specification specifies that the sigaction function not restart interrupted system calls unless the SA_RESTART flag is specified. □

NTPulse

Example—signal_intr Function

```
#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 approach 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 8.0 and Mac OS X 10.6.8, setjmp and longjmp save and restore the signal mask. Linux 3.2.0 and Solaris 10, however, do not do this, although Linux supports an option to provide BSD behavior. FreeBSD and Mac OS X provide the functions _setjmp and _longjmp, which do not save and restore the signal mask.

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

NKPulse

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.

```
sig_usr1(int signo)
```

NITPulse

```
#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. This 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);
static void                 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
```

```
{ 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 examined in the signal handler, and `siglongjmp` is called only if the flag `canjump` is nonzero. This technique provides protection against the signal handler being called at some earlier or later time, when the jump buffer hasn't been 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 as well, 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 timeline for this program. 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`.

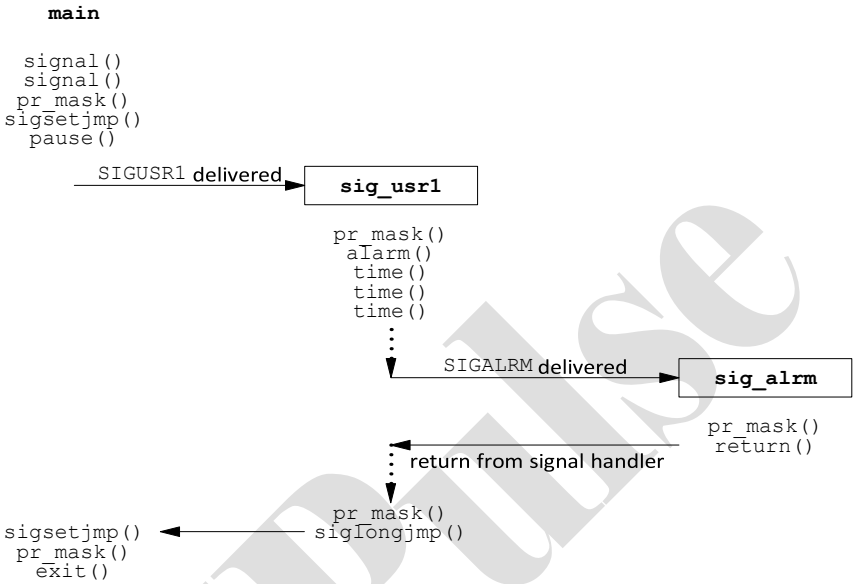

Figure 10.21 Timeline for example program handling two signals

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
$insig_alrm: SIGUSR1 SIGALRM
finishing sig_usr1: SIGUSR1
ending main:
                                     just press RETURN
[1] +  Done          ./a.out&
```

The output is what 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. □

NTPulse

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. But 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 */

/* restore 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 restore 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).

NIIT Pulse

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);
}
```

NTPulse

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

Figure 10.22 Protecting a critical region from a signal

When sigsuspend returns, it sets the signal mask to its value before the call. In this example, the SIGINT signal will be blocked, so we restore 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
^C                                    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_tquitflag; /*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)
```

NITPulse

```
        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
^C                      type the interrupt character
interrupt
^C                      type the interrupt character again
interrupt
^C                      and again
interrupt
^\ $                    now terminate with the 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`—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.

NTPulse

```
#include "apue.h"

static volatile sig_atomic_t sigflag; /* set nonzero by sig handler */

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)
{
static sigset_t newmask, oldmask, zeromask;
```

NITPulse

```
    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 we're 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 `read` function 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 (read( ... ) < 0) { if (errno
== EINTR) { if (alrm_flag)
handle_alrm(); else if (intr_flag)
handle_intr();
    } else {
        /* some other error */
    }
} else if (n == 0) {
    /* end of file */
} else {
```

```
        /* process input */
    }
```

NITPulse

We test each of the global flags before calling `read` and again if `read` 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 `read`. 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 `read` 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 `read` (or any other system function) 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 allows an implementation to call `fclose` on open standard I/O streams before terminating if the call to `abort` terminates the process.

NIPulse

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

Historically, implementations of abort have differed 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 ... */
}
```

ignoring the signal or catching it.

NTPulse

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 calling 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 the case where 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 handles these signals correctly, 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

```
    caught SIGCHLD
```

```
#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

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 caller would then use one of the `wait` functions to get the termination status of the child, thereby 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
wtemp.foo               write the buffer to a file
13                      editor says it wrote 13 bytes
^C                      type the interrupt character
?                       editor catches signal, prints question mark
caught SIGINT           and so does the parent process
q                       leave editor
```

MTPulse

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.

```
        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.

```
sigemptyset(&ignore.sa_mask);
ignore.sa_flags = 0;
if (sigaction(SIGINT, &ignore, &saveintr) < 0) return(-
    1);
if (sigaction(SIGQUIT, &ignore, &savequit) < 0)
```

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. FreeBSD 8.0, Mac OS X 10.6.8, and Solaris 10 all implement this semantic. Linux 3.2.0, however, doesn't —`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, a Linux system 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` regarding whether the parent or child runs first. If the child runs first and the parent doesn't run for some time after, an interrupt signal might 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`

The return value from `system` 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"
^Cnormal termination, exit status = 130  we press the interrupt key
```

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

When we terminate the `sleep` call 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` call with the quit key. As this example demonstrates, the Bourne shell has a poorly documented feature in which 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"
^C                                      press the interrupt key
$ echo $?                               print termination status of last command
130
$ sh -c "sleep 30"
^\sh: 962 Quit - core dumped            press 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 is 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 30 sar 949 947 pts/5
  0:01 /bin/sh sar 9257 949 pts/5  0:00 tsys
sleep 30 sar 9259 9258 pts/5  0:00 sleep 30 $ 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 terminates abnormally.

> Other shells behave differently when handling terminal-generated signals, such as `SIGINT` and `SIGQUIT`. With `bash` and `dash`, for example, pressing the interrupt or quit key will result in an exit status indicating abnormal termination with the corresponding signal number. However, if we find our process executing `sleep` and send it a signal directly, so that the signal goes only to the individual process instead of the entire foreground process group, we will find that these shells behave like the Bourne shell and exit with a normal termination status of 128 plus the signal number.

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`.

NTPulse

10.19 sleep, nanosleep, and clock_nanosleep Functions

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 occur 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 the interim), but will another SIGALRM be generated 2 seconds later? These details depend on the implementation.

> FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10 implement sleep using the nanosleep function, which allows the implementation to be independent of signals and the alarm timer. 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)
{
```

NTPulse

```
    /* nothing to do, just returning wakes up sigsuspend() */
} unsigned
sleep(unsigned int seconds)
{
    struct sigaction    newact, oldact;
    sigset_t
    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(seconds);
    suspmask = oldmask;

    /* make sure SIGALRM isn't blocked */
    sigdelset(&suspmask, SIGALRM);

    /* wait for any signal to be caught */
    sigsuspend(&suspmask);

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

    unslept = alarm(0);

    /* reset previous action */
    sigaction(SIGALRM, &oldact, NULL);

    /* 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 `alarm` and `pause`), so there is no effect on other signal handlers that may be executing when the `SIGALRM` is handled. □

The `nanosleep` function is similar to the `sleep` function, but provides nanosecond-level granularity.

```
int
                        newmask, oldmask, suspmask;
```

NTPulse

```
#include <time.h>

int nanosleep(const struct timespec *reqtp, struct timespec *remtp);
```

Returns: 0 if slept for requested time or −1 on error

This function suspends the calling process until either the requested time has elapsed or the function is interrupted by a signal. The *reqtp* parameter specifies the amount of time

to sleep in seconds and nanoseconds. If the sleep interval is interrupted by a signal and the process doesn't terminate, the `timespec` structure pointed to by the *remtp* parameter will be set to the amount of time left in the sleep interval. We can set this parameter to `NULL` if we are uninterested in the time unslept.

If the system doesn't support nanosecond granularity, the requested time is rounded up. Because the `nanosleep` function doesn't involve the generation of any signals, we can use it without worrying about interactions with other functions.

> The `nanosleep` function used to belong to the Timers option in the Single UNIX Specification, but was moved to the base in SUSv4.

With the introduction of multiple system clocks (recall Section 6.10), we need a way to suspend the calling thread using a delay time relative to a particular clock. The `clock_nanosleep` function provides us with this capability.

```
#include <time.h>

int clock_nanosleep(clockid_t clock_id, int flags,
                    const struct timespec *reqtp, struct timespec *remtp);
```

Returns: 0 if slept for requested time or error number on failure

The *clock_id* argument specifies the clock against which the time delay is evaluated. Identifiers for clocks are listed in Figure 6.8. The *flags* argument is used to control whether the delay is absolute or relative. When *flags* is set to 0, the sleep time is relative (i.e., how long we want to sleep). When it is set to `TIMER_ABSTIME`, the sleep time is absolute (i.e., we want to sleep until the clock reaches the specified time).

The other arguments, *reqtp* and *remtp*, are the same as in the `nanosleep` function. However, when we use an absolute time, the *remtp* argument is unused, because it isn't needed; we can reuse the same value for the *reqtp* argument for additional calls to `clock_nanosleep` until the clock reaches the specified absolute time value. Note that except for error returns, the call

```
clock_nanosleep(CLOCK_REALTIME, 0, reqtp, remtp);
```

has the same effect as the call

```
nanosleep(reqtp, remtp);
```

APUE

The problem with using a relative sleep is that some applications require precision with how long they sleep, and a relative sleep time can lead to sleeping longer than desired. For example, if an application wants to perform a task at regular intervals, it would have to get the current time, calculate the amount of time until the next time to execute the task, and then call `nanosleep`. Between the time that the current time is obtained and the call to `nanosleep` is made, processor scheduling and preemption can result in the relative sleep time extending past the desired interval. Using an absolute time improves the precision, even though a time-sharing process scheduler makes no guarantee that our task will execute immediately after our sleep time has ended.

> In older versions of the Single UNIX Specification, the `clock_nanosleep` function belonged to the Clock Selection option. In SUSv4, it was moved to the base.

10.20 `sigqueue` Function

In Section 10.8 we said that most UNIX systems don't queue signals. With the real-time extensions to POSIX.1, some systems began adding support for queueing signals. With SUSv4, the queued signal functionality has moved from the real-time extensions to the base specification.

Generally a signal carries one bit of information: the signal itself. In addition to queueing signals, these extensions allow applications to pass more information along with the delivery (recall Section 10.14). This information is embedded in a `siginfo` structure. Along with system-provided information, applications can pass an integer or a pointer to a buffer containing more information to the signal handler. To use queued signals we have to do the following:

1. Specify the `SA_SIGINFO` flag when we install a signal handler using the `sigaction` function. If we don't specify this flag, the signal will be posted, but it is left up to the implementation whether the signal is queued.
2. Provide a signal handler in the `sa_sigaction` member of the `sigaction` structure instead of using the usual `sa_handler` field. Implementations might allow us to use the `sa_handler` field, but we won't be able to obtain the extra information sent with the `sigqueue` function.
3. Use the `sigqueue` function to send signals.

```
#include <signal.h> int sigqueue(pid_t pid, int signo, const union
sigval value)
                                                    Returns: 0 if OK, −1 on error
```

The `sigqueue` function is similar to the `kill` function, except that we can only direct signals to a single process with `sigqueue`, and we can use the *value* argument to transmit either an integer or a pointer value to the signal handler.

Signals can't be queued infinitely. Recall the `SIGQUEUE_MAX` limit from Figure 2.9 and Figure 2.11. When this limit is reached, `sigqueue` can fail with `errno` set to `EAGAIN`.

With the real-time signal enhancements, a separate set of signals was introduced for application use. These are the signal numbers between `SIGRTMIN` and `SIGRTMAX`, inclusive. Be aware that the default action for these signals is to terminate the process.

Figure 10.30 summarizes the way queued signals differ in behavior among the implementations covered in this text.

Behavior	SUS	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
supports `sigqueue`	•	•	•		•
queues other signals besides `SIGRTMIN` to `SIGRTMAX`	optional	•			•
queues signals even if the caller doesn't use the `SA_SIGINFO` flag	optional	•	•		

Figure 10.30 Behavior of queued signals on various platforms

NITPulse

Mac OS X 10.6.8 doesn't support `sigqueue` or real-time signals. On Solaris 10, `sigqueue` is in the real-time library, `librt`.

10.21 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 background process group member.

`SIGTTOU` Write to controlling terminal by a background process group member.

Except for `SIGCHLD`, most application programs don't handle these signals: interactive shells usually do all the work required to handle them. 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 10.31 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

NTPulse

comments are given in the signal handler for typical actions performed by a program that manages a screen.

NITPulse

When the program in Figure 10.31 starts, it arranges to catch the SIGTSTP signal only if the

```
#include "apue.h"

#define BUFFSIZE    1024

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 ... */
}

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);
}
```

Figure 10.31 How to handle SIGTSTP

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

NTPulse

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 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). □

10.22 Signal Names and Numbers

In this section, we describe how to map between signal numbers and 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 8.0, Linux 3.2.0, and Mac OS X 10.6.8 all provide this array of signal names. Solaris 10 does, too, but it uses the name _sys_siglist instead.

To print the character string corresponding to a signal number in a portable manner, we can use the psignal function.

```
#include <signal.h>

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

The string *msg* (which normally includes 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. If *msg* is NULL, then only the description is written to the standard error. This function is similar to perror (Section 1.7).

If you have a siginfo structure from an alternative sigaction signal handler, you can print the signal information with the psiginfo function.

```
#include <signal.h>

void psiginfo(const siginfo_t *info, const char *msg);
```

It operates in a similar manner to the `psignal` function. Although this function has access to more information than just the signal number, platforms vary in exactly what additional information is printed.

If you only need the string description of the signal and don't necessarily want to write it to standard error (you might want to write it to a log file, for example), you can use the `strsignal` function. 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 10, `strsignal` will return a null pointer if the signal number is invalid, whereas FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8 return a string indicating that the signal number is unrecognized.
>
> Only Linux 3.2.0 and Solaris 10 support the `psiginfo` function.

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.

NTPulse

13

Under System V–based systems, a similar command is `ps -efj`. (In an attempt to improve security, some UNIX systems don't allow us to use `ps` to look at any processes other than our own.) The output from `ps` looks like

Daemon Processes

13.1 Introduction

Daemons are processes that live for a long time. They are often started when the system is bootstrapped and terminate only when the system is shut down. Because they don't have a controlling terminal, we say that they run in the background. UNIX systems have numerous daemons that perform day-to-day activities.

In this chapter, we look at the process structure of daemons and explore how to write a daemon. Since a daemon does not have a controlling terminal, we need to see how a daemon can report error conditions when something goes wrong.

For a discussion of the historical background of the term *daemon* as it applies to computer systems, see Raymond [1996].

13.2 DaemonCharacteristics

Let's look at some common system daemons and how they relate to the concepts of process groups, controlling terminals, and sessions that we described in Chapter 9. The `ps`(1) command prints the status of various processes in the system. There are a multitude of options—consult your system's manual for all the details. We'll execute

```
ps -axj
```

under BSD-based systems to see the information we need for this discussion. The `-a` option shows the status of processes owned by others, and `-x` shows processes that don't have a controlling terminal. The `-j` option displays the job-related information: the session ID, process group ID, controlling terminal, and terminal process group ID.

NIIPulse

The system processes you see will depend on the operating system implementation. Anything with a parent process ID of 0 is usually a kernel process started as part of the system bootstrap procedure. (An exception is `init`, which is a user-level command started by the kernel at boot time.) Kernel processes are special and generally exist for the entire lifetime of the system. They run with superuser privileges and have no controlling terminal and no command line.

In the sample `ps` output, kernel daemons appear with their names in square brackets. This version of Linux uses a special kernel process, `kthreadd`, to create other kernel processes, so `kthreadd` appears as the parent of the other kernel daemons. Each kernel component that needs to perform work in a process context, but that isn't invoked from the context of a user-level process, will usually have its own kernel daemon. For example, on Linux

- The `kswapd` daemon is also known as the pageout daemon. It supports the virtual memory subsystem by writing dirty pages to disk slowly over time, so the pages can be reclaimed.
- The `flush` daemon flushes dirty pages to disk when available memory reaches a configured minimum threshold. It also flushes dirty pages back to disk at regular intervals to decrease data loss in the event of a system failure. Several flush daemons can exist—one for each backing device. The sample output shows one flush daemon named `flush-8:0`. In the name, the backing device is identified by its major device number (8) and its minor device number (0).
- The `sync_supers` daemon periodically flushes file system metadata to disk.
- The `jbd` daemon helps implement the journal in the `ext4` file system.

Process 1 is usually `init` (`launchd` on Mac OS X), as we described in Section 8.2. It is a system daemon responsible for, among other things, starting system services specific to various run levels. These services are usually implemented with the help of their own daemons.

The `rpcbind` daemon provides the service of mapping RPC (Remote Procedure Call) program numbers to network port numbers. The `rsyslogd` daemon is available to any program to log system messages for an administrator. The messages may be printed on a console device and also written to a file. (We describe the `syslog` facility in Section 13.4.)

We talked about the `inetd` daemon in Section 9.3. It listens on the system's network interfaces for incoming requests for various network servers. The `nfsd`, `nfsiod`, `lockd`, `rpciod`, `rpc.idmapd`, `rpc.statd`, and `rpc.mountd` daemons provide support for the Network File System (NFS). Note that the first four are kernel daemons, while the last three are user-level daemons.

The `cron` daemon executes commands at regularly scheduled dates and times. Numerous system administration tasks are handled by `cron` running programs at regularly intervals. The `atd` daemon is similar to `cron`; it allows users to execute jobs at specified times, but it executes each job once only, instead of repeatedly at regularly scheduled times. The `cupsd` daemon is a

print spooler; it handles print requests on the system. The `sshd` daemon provides secure remote login and execution facilities.

Note that most of the daemons run with superuser (root) privileges. None of the daemons has a controlling terminal: the terminal name is set to a question mark. The kernel daemons are started without a controlling terminal. The lack of a controlling terminal in the user-level daemons is probably the result of the daemons having called `setsid`. Most of the user-level daemons are process group leaders and session leaders, and are the only processes in their process group and session. (The one exception is `rsyslogd`.) Finally, note that the parent of the user-level daemons is the `init` process.

13.3 Coding Rules

Some basic rules to coding a daemon prevent unwanted interactions from happening. We state these rules here and then show a function, `daemonize`, that implements them.

1. Call `umask` to set the file mode creation mask to a known value, usually 0. The inherited file mode creation mask could be set to deny certain permissions. If the daemon process creates files, it may want to set specific permissions. For example, if it creates files with group-read and group-write enabled, a file mode creation mask that turns off either of these permissions would undo its efforts. On the other hand, if the daemon calls library functions that result in files being created, then it might make sense to set the file mode create mask to a more restrictive value (such as 007), since the library functions might not allow the caller to specify the permissions through an explicit argument.
2. Call `fork` and have the parent `exit`. This does several things. First, if the daemon was started as a simple shell command, having the parent terminate makes the shell think that the command is done. Second, the child inherits the process group ID of the parent but gets a new process ID, so we're guaranteed that the child is not a process group leader. This is a prerequisite for the call to `setsid` that is done next.
3. Call `setsid` to create a new session. The three steps listed in Section 9.5 occur. The process (a) becomes the leader of a new session, (b) becomes the leader of a new process group, and (c) is disassociated from its controlling terminal.

   > Under System V–based systems, some people recommend calling `fork` again at this point, terminating the parent, and continuing the daemon in the child. This guarantees that the daemon is not a session leader, which prevents it from acquiring a controlling terminal under the System V rules (Section 9.6). Alternatively, to avoid acquiring a controlling terminal, be sure to specify `O_NOCTTY` whenever opening a terminal device.

4. Change the current working directory to the root directory. The current working directory inherited from the parent could be on a mounted file system. Since daemons normally exist until the system is rebooted, if the daemon stays on a mounted file system, that file system cannot be unmounted.

   Alternatively, some daemons might change the current working directory to a specific location where they will do all their work. For example, a line printer spooling daemon might change its working directory to its spool directory.

NITPUNE

5. Unneeded file descriptors should be closed. This prevents the daemon fromholding open any descriptors that it may have inherited from its parent (which could be a shell or some other process). We can use our `open_max` function (Figure 2.17) or the `getrlimit` function (Section 7.11) to determine the highest descriptor and close all descriptors up to that value.

NTPulse

6. Some daemons open file descriptors 0, 1, and 2 to `/dev/null` so that any library

terminal device, there is nowhere for output to be displayed, nor is there anywhere to receive input from an interactive user. Even if the daemon was started from an interactive session, the daemon runs in the background, and the login session can terminate without affecting the daemon. If other users log in on the same terminal device, we wouldn't want output from the daemon showing up on the terminal, and the users wouldn't expect their input to be read by the daemon.

Example

Figure 13.1 shows a function that can be called from a program that wants to initialize itself as a daemon.

```
#include "apue.h"
#include <syslog.h>
#include <fcntl.h>
#include <sys/resource.h>

void
daemonize(const char *cmd)
{
    int                 i,fd0, fd1, fd2;
    pid_t               pid;
    struct rlimit       rl;
    struct sigaction    sa;

    /*
     *Clear file creation mask.
     */
    umask(0);

    /*
     *Get maximum number of file descriptors.
     */
    if (getrlimit(RLIMIT_NOFILE, &rl) < 0)
        err_quit("%s: can't get file limit", cmd);

    /*
     *Become a session leader to lose controlling TTY.
     */
    if ((pid = fork()) < 0)
        err_quit("%s: can't fork", cmd);
    else if (pid != 0) /* parent */
        exit(0);
    setsid();

    /*
```

routines that try to read from standard input or write to standard output or standard error will have no effect. Since the daemon is not associated with a

NITPulse

```
     * Ensure future opens won't allocate controlling TTYs.
     */ sa.sa_handler =
    SIG_IGN;
    if (sigaction(SIGHUP, &sa, NULL) < 0)
        err_quit("%s: can't ignore SIGHUP", cmd);
    if ((pid = fork()) < 0)
        err_quit("%s: can't fork", cmd);
    else if (pid != 0) /* parent */
        exit(0);

    /*
     *Change the current working directory to the root so
     *wewon't prevent file systems from being unmounted.
     */
    if (chdir("/") < 0)
        err_quit("%s: can't change directory to /", cmd);

    /*
     *Close all open file descriptors.
     */
    if (rl.rlim_max == RLIM_INFINITY)
        rl.rlim_max = 1024;
    for (i = 0; i < rl.rlim_max; i++)
        close(i);

    /*
     *Attach file descriptors 0, 1, and 2 to /dev/null.
     */
    fd0 = open("/dev/null", O_RDWR);
    fd1 = dup(0);
    fd2 = dup(0);

    /*
     *Initialize the log file.
     */
    openlog(cmd, LOG_CONS, LOG_DAEMON);
    if (fd0 != 0 || fd1 != 1 || fd2 != 2) {
        syslog(LOG_ERR, "unexpected file descriptors %d %d %d",
          fd0, fd1, fd2);
        exit(1);
    }
}
```

Figure 13.1 Initialize a daemon process

If the `daemonize` function is called from a `main` program that then goes to sleep, we can check the status of the daemon with the `ps` command:

```
$ ./a.out
$ ps -efj
UID     PIDPPID PGIDSID TTY CMD
 sigemptyset(&sa.sa_mask)
 ;
 sa.sa_flags = 0;
```

```
        sar 13800         1 13799 13799?    ./a.out
        $ ps -efj | grep 13799
        sar 13800         1 13799 13799?    ./a.out
```


We can also use `ps` to verify that no active process exists with ID 13799. This means that our daemon is in an orphaned process group (Section 9.10) and is not a session leader and, therefore, has no chance of allocating a controlling terminal. This is a result of performing the second `fork` in the `daemonize` function. We can see that our daemon has been initialized correctly.

NITPulse

Figure 13.2 The BSD `syslog` facility

There are three ways to generate log messages:

1. Kernel routines can call the `log` function. These messages can be read by any user process that `open`s and `read`s the `/dev/klog` device. We won't describe this function any further, since we're not interested in writing kernel routines.
2. Most user processes (daemons) call the `syslog`(3) function to generate log messages. We describe its calling sequence later. This causes the message to be sent to the UNIX domain datagram socket `/dev/log`.
3. A user process on this host, or on some other host that is connected to this hostby a TCP/IP network, can send log messages to UDP port 514. Note that the `syslog` function never generates these UDP datagrams: they require explicit network programming by the process generating the log message.

Refer to Stevens, Fenner, and Rudoff [2004] for details on UNIX domain sockets and UDP sockets.

Normally, the `syslogd` daemon reads all three forms of log messages. On start-up, this daemon reads a configuration file, usually `/etc/syslog.conf`, which determines where different classes of messages are to be sent. For example, urgent messages can be sent to the system administrator (if logged in) and printed on the console, whereas warnings may be logged to a file.

Our interface to this facility is through the `syslog` function.

```
#include <syslog.h>

void openlog(const char *ident, int option, int facility);

void syslog(int priority, const char *format, ...);

void closelog(void); int setlogmask(int maskpri);
```

Returns: previous log priority mask value

Calling `openlog` is optional. If it's not called, the first time `syslog` is called, `openlog` is called automatically. Calling `closelog` is also optional—it just closes the descriptor that was being used to communicate with the `syslogd` daemon.

Calling `openlog` lets us specify an *ident* that is added to each log message. This is normally the name of the program (e.g. `cron`, `inetd`). The *option* argument is a bitmask specifying various options. Figure 13.3 describes the available options, including a bullet in the XSI column if the option is included in the `openlog` definition in the Single UNIX Specification.

The *facility* argument for `openlog` is taken from Figure 13.4. Note that the Single UNIX Specification defines only a subset of the facility codes typically available on a given platform. The reason for the *facility* argument is to let the configuration file specify that messages from different facilities are to be handled differently. If we don't call `openlog`, or if we call it with a *facility* of 0, we can still specify the facility as part of the *priority* argument to `syslog`.

We call `syslog` to generate a log message. The *priority* argument is a combination of the *facility*, shown in Figure 13.4, and a *level*, shown in Figure 13.5. These *level*s are ordered by priority, from highest to lowest.

option	XSI	Description
LOG_CONS	•	If the log message can't be sent to syslogd via the UNIX domain datagram, the message is written to the console instead.
LOG_NDELAY	•	Open the UNIX domain datagram socket to the syslogd daemon immediately; don't wait until the first message is logged. Normally, the socket is not opened until the first message is logged.
LOG_NOWAIT	•	Do not wait for child processes that might have been created in the process of logging the message. This prevents conflicts with applications that catch SIGCHLD, since the application might have retrieved the child's status by the time that syslog calls wait.
LOG_ODELAY	•	Delay the opening of the connection to the syslogd daemon until the first message is logged.
LOG_PERROR		Write the log message to standard error in addition to sending it to syslogd. (Unavailable on Solaris.)
LOG_PID	•	Log the process ID with each message. This is intended for daemons that fork a child process to handle different requests (as compared to daemons, such as syslogd, that never call fork).

Figure 13.3 The *option* argument for openlog

The *format* argument and any remaining arguments are passed to the vsprintf function for formatting. Any occurrences of the characters %m in *format* are first replaced with the error message string (strerror) corresponding to the value of errno.

The setlogmask function can be used to set the log priority mask for the process. This function returns the previous mask. When the log priority mask is set, messages are not logged unless their priority is set in the log priority mask. Note that attempts to set the log priority mask to 0 will have no effect.

The logger(1) program is also provided by many systems as a way to send log messages to the syslog facility. Some implementations allow optional arguments to this program, specifying the *facility*, *level*, and *ident*, although the Single UNIX Specification doesn't define any options. The logger command is intended for a shell script running noninteractively that needs to generate log messages.

Example

In a (hypothetical) line printer spooler daemon, you might encounter the sequence

```
openlog("lpd", LOG_PID, LOG_LPR);
syslog(LOG_ERR, "open error for %s: %m", filename);
```

The first call sets the *ident* string to the program name, specifies that the process ID should always be printed, and sets the default *facility* to the line printer system. The call to syslog specifies an error condition and a message string. If we had not called openlog, the second call could

have been `syslog(LOG_ERR | LOG_LPR, "open error for %s: %m", filename);` Here, we specify the *priority* argument as a combination of a *level* and a *facility*. □

facility	XSI	Description
LOG_AUDIT		the audit facility
LOG_AUTH		authorization programs: login, su, getty, ...
LOG_AUTHPRIV		same as LOG_AUTH, but logged to file with restricted permissions
LOG_CONSOLE		messages written to /dev/console
LOG_CRON		cron and at
LOG_DAEMON		system daemons: inetd, routed, ...
LOG_FTP		the FTP daemon (ftpd)
LOG_KERN		messages generated by the kernel
LOG_LOCAL0	•	reserved for local use
LOG_LOCAL1	•	reserved for local use
LOG_LOCAL2	•	reserved for local use
LOG_LOCAL3	•	reserved for local use
LOG_LOCAL4	•	reserved for local use
LOG_LOCAL5	•	reserved for local use
LOG_LOCAL6	•	reserved for local use
LOG_LOCAL7	•	reserved for local use
LOG_LPR		line printer system: lpd, lpc, ...
LOG_MAIL		the mail system
LOG_NEWS		the Usenet network news system
LOG_NTP		the network time protocol system
LOG_SECURITY		the security subsystem
LOG_SYSLOG		the syslogd daemon itself
LOG_USER	•	messages from other user processes (default)
LOG_UUCP		the UUCP system

Figure 13.4 The *facility* argument for `openlog`

level	Description
LOG_EMERG	emergency (system is unusable) (highest priority)
LOG_ALERT	condition that must be fixed immediately
LOG_CRIT	critical condition (e.g., hard device error)
LOG_ERR	error condition
LOG_WARNING	warning condition
LOG_NOTICE	normal, but significant condition
LOG_INFO	informational message
LOG_DEBUG	debug message (lowest priority)

Figure 13.5 The `syslog` *level*s (ordered)

In addition to `syslog`, many platforms provide a variant that handles variable argument lists.

```
#include <syslog.h>
#include <stdarg.h>

void vsyslog(int priority, const char *format, va_list arg);
```

All four platforms described in this book provide `vsyslog`, but this function is not included in the Single UNIX Specification. Note that to make its declaration visible to your application,

you might need to define an additional symbol, such as __BSD_VISIBLE on FreeBSD or __USE_BSD on Linux.

Most syslogd implementations will queue messages for a short time. If a duplicate message arrives during this period, the syslog daemon will not write it to the log. Instead, the daemon prints a message similar to ''last message repeated *N* times.''

13.5 Single-Instance Daemons

Some daemons are implemented so that only a single copy of the daemon should be running at a time for proper operation. Such a daemon might need exclusive access to a device, for example. In the case of the cron daemon, if multiple instances were running, each copy might try to start a single scheduled operation, resulting in duplicate operations and probably an error.

If the daemon needs to access a device, the device driver will sometimes prevent multiple attempts to open the corresponding device node in /dev. This restricts us to one copy of the daemon running at a time. If no such device is available, however, we need to do the work ourselves.

The file- and record-locking mechanism provides the basis for one way to ensure that only one copy of a daemon is running. (We discuss file and record locking in Section 14.3.) If each daemon creates a file with a fixed name and places a write lock on the entire file, only one such write lock will be allowed to be created. Successive attempts to create write locks will fail, serving as an indication to successive copies of the daemon that another instance is already running.

File and record locking provides a convenient mutual-exclusion mechanism. If the daemon obtains a write-lock on an entire file, the lock will be removed automatically if the daemon exits. This simplifies recovery, eliminating the need for us to clean up from the previous instance of the daemon.

Example

The function shown in Figure 13.6 illustrates the use of file and record locking to ensure that only one copy of a daemon is running.

```
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <syslog.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>
#include <sys/stat.h>

#define LOCKFILE "/var/run/daemon.pid"
#define LOCKMODE (S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)

extern int lockfile(int);
```

```
int


already_running(void)
{
    int     fd;
    char buf[16];

    fd = open(LOCKFILE, O_RDWR|O_CREAT, LOCKMODE);
    if (fd < 0) {
        syslog(LOG_ERR, "can't open %s: %s", LOCKFILE, strerror(errno));
        exit(1);
    }
    if (lockfile(fd) < 0) {
        if (errno == EACCES || errno == EAGAIN) {
            close(fd);
            return(1);
        }
        syslog(LOG_ERR, "can't lock %s: %s", LOCKFILE, strerror(errno));
        exit(1);
    }
    ftruncate(fd, 0);
    sprintf(buf, "%ld", (long)getpid());
    write(fd, buf, strlen(buf)+1);
    return(0);
}
```

Figure 13.6 Ensure that only one copy of a daemon is running

Each copy of the daemon will try to create a file and write its process ID in the file. This will allow administrators to identify the process easily. If the file is already locked, the `lockfile` function will fail with `errno` set to `EACCES` or `EAGAIN`, so we return 1, indicating that the daemon is already running. Otherwise, we truncate the file, write our process ID to it, and return 0.

We need to truncate the file, because the previous instance of the daemon might have had a process ID larger than ours, with a larger string length. For example, if the previous instance of the daemon was process ID 12345, and the new instance is process ID 9999, when we write the process ID to the file, we will be left with 99995 in the file. Truncating the file prevents data from the previous daemon appearing as if it applies to the current daemon. □

13.6 DaemonConventions

•

Several common conventions are followed by daemons in the UNIX System.

If the daemon uses a lock file, the file is usually stored in `/var/run`. Note, however, that the daemon might need superuser permissions to create a file here. The name of

the file is usually *name*`.pid`, where *name* is the name of the daemon or the service. For example, on Linux, the name of the `cron` daemon's lock file is `/var/run/crond.pid`.

- If the daemon supports configuration options, they are usually stored in `/etc`. The configuration file is named *name*`.conf`, where *name* is the name of the daemon or the name of the service. For example, the configuration for the `syslogd` daemon is usually `/etc/syslog.conf`.
- Daemons can be started from the command line, but they are usually started from one of the system initialization scripts (`/etc/rc*` or `/etc/init.d/*`). If the daemon should be restarted automatically when it exits, we can arrange for `init` to restart it if we include a `respawn` entry for it in `/etc/inittab` (assuming the system uses a System V style `init` command).
- If a daemon has a configuration file, the daemon reads the file when it starts, but usually won't look at it again. If an administrator changes the configuration, the daemon would need to be stopped and restarted to account for the configuration changes. To avoid this, some daemons will catch `SIGHUP` and reread their configuration files when they receive the signal. Since they aren't associated with terminals and are either session leaders without controlling terminals or members of orphaned process groups, daemons have no reason to expect to receive `SIGHUP`. Thus they can safely reuse it.

Example

The program shown in Figure 13.7 shows one way a daemon can reread its configuration file. The program uses `sigwait` and multiple threads, as discussed in Section 12.8.

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

sigset_t mask;

extern int already_running(void);

void
reread(void)
{
    /* ... */
}

void *
thr_fn(void *arg)
{
    int err, signo;

    for (;;) {
        err = sigwait(&mask, &signo);
        if (err != 0) {
```

NIPUSE

```
                syslog(LOG_ERR, "sigwait failed");
                syslog(LOG_INFO, "Re-reading configuration file");
                reread();
                break;

            case SIGTERM:
                syslog(LOG_INFO, "got SIGTERM; exiting");
                exit(0);

            default:
                syslog(LOG_INFO, "unexpected signal %d\n", signo);
            }
        }
        return(0);
    }

    int
    main(int argc, char *argv[])
    {
        int                 err;
        pthread_t           tid;
        char                *cmd;
        struct sigaction    sa;

        if ((cmd = strrchr(argv[0], '/')) == NULL)
            cmd = argv[0];
        else
            cmd++;

        /*
         *Become a daemon.
         */
        daemonize(cmd);

        /*
         *Make sure only one copy of the daemon is running.
         */
        if (already_running()) {
            syslog(LOG_ERR, "daemon already running");
            exit(1);
        }

        /*
         *Restore SIGHUP default and block all signals.
         */
                exit(1);
            }

            switch (signo) {
            case SIGHUP:
```

NTPulse

```
        sa.sa_handler = SIG_DFL;
        sigemptyset(&sa.sa_mask);
        sa.sa_flags = 0;
        if (sigaction(SIGHUP, &sa, NULL) < 0)
            err_quit("%s: can't restore SIGHUP default");
```


_

NTPulse

```
    sigfillset(&mask);
    if ((err = pthread_sigmask(SIG_BLOCK, &mask, NULL)) != 0)
        err_exit(err, "SIG_BLOCK error");

reread(void)
{
    /*
     *Create a thread to handle SIGHUP and SIGTERM.
     */
    err = pthread_create(&tid, NULL, thr_fn, 0);
    if (err != 0)
        err_exit(err, "can't create thread");

    /*
     *Proceed with the rest of the daemon.
     */
    /* ... */
    exit(0);
}
```

Figure 13.7 Daemon rereading configuration files

We call `daemonize` from Figure 13.1 to initialize the daemon. When it returns, we call `already_running` from Figure 13.6 to ensure that only one copy of the daemon is running. At this point, `SIGHUP` is still ignored, so we need to reset the disposition to the default behavior; otherwise, the thread calling `sigwait` may never see the signal.

We block all signals, as is recommended for multithreaded programs, and create a thread to handle signals. The thread's only job is to wait for `SIGHUP` and `SIGTERM`. When it receives `SIGHUP`, the thread calls `reread` to reread its configuration file. When it receives `SIGTERM`, the thread logs a message and exits.

Recall from Figure 10.1 that the default action for `SIGHUP` and `SIGTERM` is to terminate the process. Because we block these signals, the daemon will not die when one of them is sent to the process. Instead, the thread calling `sigwait` will return with an indication that the signal has been received. □

Example

Not all daemons are multithreaded. The program in Figure 13.8 shows how a single-threaded daemon can catch `SIGHUP` and reread its configuration file.

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

extern int lockfile(int);
extern int already_running(void);

void
    /* ... */
}
```

```
void sigterm(int
signo)




{
    syslog(LOG_INFO, "got SIGTERM; exiting");
    exit(0);
}

void
sighup(int signo)
{
    syslog(LOG_INFO, "Re-reading configuration file");
    reread();
}

int
main(int argc, char *argv[])
{
    char                *cmd;
    struct sigaction    sa;

    if ((cmd = strrchr(argv[0], '/')) == NULL)
        cmd = argv[0];
    else
        cmd++;

    /*
     *Become a daemon.
     */
    daemonize(cmd);

    /*
     *Make sure only one copy of the daemon is running.
     */
    if (already_running()) {
        syslog(LOG_ERR, "daemon already running");
        exit(1);
    }

    /*
     *Handle signals of interest.
     */
    sa.sa_handler = sigterm;
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask, SIGHUP);
    sa.sa_flags = 0;
    if (sigaction(SIGTERM, &sa, NULL) < 0) { syslog(LOG_ERR, "can't
        catch SIGTERM: %s", strerror(errno));
        exit(1);
    }
    sa.sa_handler = sighup;
    sigemptyset(&sa.sa_mask);
```

NTPulse

```
    sigaddset(&sa.sa_mask, SIGTERM);
    sa.sa_flags = 0;
    if (sigaction(SIGHUP, &sa, NULL) < 0) { syslog(LOG_ERR, "can't
        catch SIGHUP: %s", strerror(errno));
        exit(1);
    }

    /*
     * Proceed with the rest of the daemon.
     */
    /* ... */
    exit(0);
}
```

Figure 13.8 Alternative implementation of daemon rereading configuration files

After initializing the daemon, we install signal handlers for `SIGHUP` and `SIGTERM`. We can either place the reread logic in the signal handler or just set a flag in the handler and have the main thread of the daemon do all the work instead.

□

13.7 Client–Server Model

A common use for a daemon process is as a server process. Indeed, in Figure 13.2, we can call the `syslogd` process a server that has messages sent to it by user processes (clients) using a UNIX domain datagram socket.

In general, a *server* is a process that waits for a *client* to contact it, requesting some type of service. In Figure 13.2, the service being provided by the `syslogd` server is the logging of an error message.

In Figure 13.2, the communication between the client and the server is one way. The client sends its service request to the server; the server sends nothing back to the client. In the upcoming chapters, we'll see numerous examples of two-way communication between a client and a server—the client sends a request to the server, and the server sends a reply back to the client.

It is common to find servers that `fork` and `exec` another program to provide service to a client. These servers often manage multiple file descriptors: communication endpoints, configuration files, log files, and the like. At best, it would be careless to leave these file descriptors open in the child process, because they probably won't be used in the program executed by the child, especially if the program is unrelated to the server. At worst, leaving them open could pose a security problem — the program executed could do something malicious, such as change the server's configuration file or trick the client into thinking it is communicating with the server, thereby gaining access to unauthorized information.

NTUPulse

An easy solution to this problem is to set the close-on-exec flag for all file descriptors that the executed program won't need. Figure 13.9 shows a function that we can use in a server process to do just this.

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

int
set_cloexec(int fd)
{
    int     val;

    if ((val = fcntl(fd, F_GETFD, 0)) < 0)
        return(-1);

    val |= FD_CLOEXEC;      /* enable close-on-exec */

    return(fcntl(fd, F_SETFD, val));
}
```

Figure 13.9 Set close-on-exec flag

13.8 Summary

Daemon processes are running all the time on most UNIX systems. Initializing our own process to run as a daemon takes some care and an understanding of the process relationships described in Chapter 9. In this chapter, we developed a function that can be called by a daemon process to initialize itself correctly.

We also discussed the ways a daemon can log error messages, since a daemon normally doesn't have a controlling terminal. We discussed several conventions that daemons follow on most UNIX systems and showed examples of how to implement some of these conventions.

Exercises

13.1 As we might guess from Figure 13.2, when the `syslog` facility is initialized, either by calling `openlog` directly or on the first call to `syslog`, the special device file for the UNIX domain datagram socket, `/dev/log`, has to be opened. What happens if the user process (the daemon) calls `chroot` before calling `openlog`?

13.2 Recall the sample `ps` output from Section 13.2. The only user-level daemon that isn't a session leader is the `rsyslogd` process. Explain why the `syslogd` daemon isn't a session leader.

13.3 List all the daemons active on your system, and identify the function of each one.

13.4 Write a program that calls the `daemonize` function in Figure 13.1. After calling this function, call `getlogin` (Section 8.15) to see whether the process has a login name now that it has become a daemon. Print the results to a file.

NIPulse