Signals

Signals are another area in UNIX where the initial implementation was inadequate, and multiple implementations have developed in the course of the time. If you try to port software which assumes the presence of one implementation, and your system doesn't support this implementation, you could be in for a significant amount of rewriting. The situation isn't improved by the fact that there are a number of subtle differences between the various implementations and even between different systems with the same implementation. In this chapter, we'll look at those aspects of signal implementation which are of interest to porting.

There have been four different implementations in the course of UNIX history:

- The Seventh Edition had so-called *unreliable signals*. and handled them with the *signal* system call. System V still supplies them with the `signal` system call. As we will see on page 188, the use of the `signal` function does not automatically imply unreliable signals.
- 4.2BSD introduced the first implementation of *reliable signals*. It uses the functions `signal`, `sigvec`, `sigblock`, `sigsetmask` and `sigpause`.
- System V introduced an alternative implementation of reliable signals. It uses the functions `sigset`, `sighold`, `sigrelse`, `sigignore` and `sigpause`.
- Finally, POSIX.1 defined a third implementation of reliable signals. These are based on the BSD signals and use the functions `sigaction`, `sigprocmask`, `sigpending` and `sigsuspend`.

Most people think of signals as the way the operating system or an outside user stops a program that is misbehaving. More generally, they are a means to cause execution of functions out of sequence, and have thus been called *software interrupts*. Hardware interrupts cause the system to interrupt normal processing and perform a specific sequence of instructions. Signals do the same thing in software: when a process receives a signal, the kernel simulates a call to a pre-defined routine.* The routine, called a *signal handler*, handles the signal and possibly returns to the "caller". It would be a significant overhead for every program to supply a

* This is not a real call: when the kernel delivers the signal, it modifies the process stack and registers so that it looks as if the signal handler has just been called. When the process continues executing, it is in the signal handler. Nobody ever really calls the signal handler.

signal handler for every conceivable signal, so the kernel supplies two default methods of handling the signal. The choice of a signal handler or one of the two defaults is called the *disposition* of the signal. Initially, each signal's disposition is set either to ignore the signal or to terminate the process if the signal occurs. In some cases, the system writes a *core* file, a copy of the state of the process, when the process is terminated.

Signals may come from a number of different sources:

- External events. For example, pressing `CTRL-C` or `DEL` on most systems causes the terminal driver to send a `SIGINT` signal to the foreground process group of the terminal.
- Internal events. For example, `alarm` causes a `SIGALRM` signal after the specified timeout.
- Hardware interrupts. For example, if a process attempts to access a page that is not part of its address space, it will receive a `SIGSEGV` or `SIGBUS` signal.
- As the result of another process calling `kill`.

In this chapter, we'll consider which signals are supported by which operating systems, and how signals are implemented in different operating systems.

Supported signals

The Seventh Edition had 15 signals, and current implementations allow up to 31, though not all are used. In the course of time, the meanings have also diverged somewhat. Table 13-1 gives an overview of which signals are present in which implementations.

Table 13−1: Signal usage

Signal	V7	SVR3	SVR4	BSD	POSIX	action	purpose
SIGABRT			•	•	•	core	abort call[2]
SIGALRM	•	•	•	•	•	kill	real-time timer expired
SIGBUS	•	•	•	•		core	bus error
SIGCHLD			•	•	•	ignore	child status has changed
SIGCLD		•	•			ignore	child status has changed
SIGCONT		•	•	•	•	ignore	continue after stop
SIGEMT	•	•	•	•		core	emulate instruction executed
SIGFPE	•	•	•	•	•	core	floating-point exception
SIGHUP	•	•	•	•	•	kill	line hangup
SIGILL	•	•	•	•	•	core	illegal instruction
SIGINFO			•	•		ignore	status request from keyboard

Table 13−1: Signal usage (continued)

Signal	V7	SVR3	SVR4	BSD	POSIX	action	purpose
SIGINT	•	•	•	•	•	kill	interrupt program (usually from terminal driver)
SIGIO			•	•		ignore	I/O completion outstanding[1]
SIGIOT	•	•				core	IOT instruction[2]
SIGKILL	•	•	•	•	•	kill	kill program[4]
SIGPIPE	•	•	•	•	•	kill	write on a pipe with no reader
SIGPROF			•	•		kill	profiling timer alarm
SIGPWR		•	•	•		ignore	power fail/restart
SIGQUIT	•	•	•	•	•	core	quit program (usually from terminal driver)
SIGSEGV	•	•	•	•	•	core	segmentation violation
SIGSTOP		•	•	•	•	stop	stop[4]
SIGSYS	•	•	•	•		core	invalid system call
SIGTERM	•	•	•	•	•	kill	software termination signal
SIGTRAP	•	•	•	•		core	trace trap
SIGTSTP		•	•	•	•	stop	stop signal generated from keyboard
SIGTTIN		•	•	•	•	stop	background read from control terminal
SIGTTOU		•	•	•	•	stop	background write to control terminal
SIGURG			•	•		ignore	urgent condition present on socket
SIGUSR1		•	•	•	•	kill	User defined signal 1
SIGUSR2		•	•	•	•	kill	User defined signal 2
SIGVTALRM			•			kill	virtual time alarm
SIGWINCH		•[3]	•	•		ignore	Window size change
SIGXCPU		•	•	•		core	cpu time limit exceeded
SIGXFSZ		•	•	•		core	file size limit exceeded

[1] Sometimes called SIGPOLL in System V.
[2] SIGIOT and SIGABRT usually have the same signal number.
[3] Not available in all versions.
[4] This signal cannot be caught or ignored.

Unreliable and reliable signals

The terms *unreliable signals* and *reliable signals* need explaining. The problem relates to what happens when a signal handler is active: if another signal occurs during this time, and it is allowed to be delivered to the process, the signal handler will be entered again. Now it's not difficult to write reentrant* signal handlers—in fact, it's a very good idea, because it

* A *reentrant function* is one which can be called by functions which it has called—in other words, it

means that you can use one signal handler to handle multiple signals—but if the same signal reoccurs before the signal handler has finished handling the previous instance, it could happen again and again, and the result can be a stack overflow with repeated signal handler calls.

The original signal implementation, which we call *unreliable signals*, had a simplistic attitude to this problem: it reset the signal dispostion to the default, which meant that if another signal occurred while the previous one was being processed, the system would either ignore the signal (so it would lose the signal) or terminate the process (which is probably not what you want). It was up to the signal handler to reinstate the signal disposition, and this couldn't be done immediately without running the risk of stack overflow.

All newer signal implementations provide so-called *reliable signals*. The signal disposition is not changed on entry to the signal handler, but a new signal will not be delivered until the signal handler returns. This concept is called *blocking* the signal: the system notes that the signal is pending, but doesn't deliver it until it is unblocked.

There are a number of things that the term *reliable signal* does *not* mean:

- It doesn't imply that the underlying kernel implementation is bug-free. Depending on the implementation, there is still a slight chance that the kernel will lose the signal.
- It doesn't imply that a signal cannot get lost. The method used to queue signals is to set a bit in a bit mask. If multiple signals of the same kind occur while the signal is blocked, only one will be delivered.
- It doesn't imply that you don't need reentrant signal handlers. The system blocks only the signal that is currently being handled. If you have a single handler for multiple signals, it will need to be reentrant. In particular, this means that you should at least be very careful with static variables and preferably use few local variables (since they take up stack space). You should also be careful with the functions you call—we'll take another look at this on page 187.

The semantics of each implementation differ in subtle ways, so changing to a different set of signal calls involves more than just changing the function calls and parameters. Here's a brief overview of the differences you might encounter:

- With unreliable signals, after a signal occurs, the signal disposition is reset to default, so the signal handler must reinstate itself before returning. If a second signal occurs before the disposition is reinstated, the process may be terminated (if the default disposition is *terminate*) or the signal may be completely forgotten (if the default disposition is *ignore*).
- The names and purposes of the signals differ significantly from one implementation to the next. See Table 13-2 for an overview.
- In modern implementations, the function call `signal` varies in its meaning. In System V, it uses the old, unreliable Seventh Edition signal semantics, while in BSD it is an interface to the `sigaction` system call, which provides reliable signals. If you're porting BSD `signal` to System V, you should modify the code use `sigaction` instead.

can be entered again before it has returned. This places a number of restrictions on the function. In particular, it cannot rely on external values, and may not use static storage.

- The first parameter to a signal handler is always the number of the signal. Both System V.4 and BSD can supply additional parameters to the signal handlers. We'll look at the additional parameters in more detail on page 183.
- The handling of *interrupted system calls* varies from one system to the next. We'll look into this topic in more detail on page 186.
- The difference between the signals `SIGBUS` and `SIGSEGV` is purely historical: it relates to the PDP-11 hardware interrupt that detected the problem. In modern systems, it depends on the whim of the implementor when you get which signal. POSIX.1 defines only `SIGSEGV`, but this doesn't help much if the processor generates `SIGBUS` anyway. It's best to treat them as being equivalent.
- `SIGCLD` is a System V version of `SIGCHLD`. A number of hairy problems can arise with `SIGCLD`; we'll look at them in more detail on page 186.
- `SIGILL` was generated by the `abort` function in early BSD implementations. Early System V used `SIGIOT` instead. All modern implementations generate `SIGABRT`. Frequently you'll find that these two signals are in fact defined to have the same number; if you run into troubles where one or the other is undefined, you could possibly do just this:

```
#define SIGIOT SIGABRT
```

Signal handlers

Modern versions of UNIX define signal handlers to be of type

```
void (*signal (int signum, void (*handler))) (int hsignum)
```

This is probably one of the most confusing definitions you are likely to come across. To understand it, it helps to remember that we are talking about two functions:

- The *signal handler*, called `handler` in this declaration, takes an int parameter `hsignum` and returns a `void` pointer to the old signal handler function, which is of the same type as itself.
- The function `signal`, which takes two parameters. The first is `signum`, the number of the signal to be handled, of type `int`, and the second is a pointer to a signal handler function `handler`. It also returns a `void` pointer to a signal handler function.

In fact, in many implementations the signal handler function takes additional parameters, and you may find that your program takes advantage of them. We'll look at these in the following sections.

System V.4 signal handlers

The System V.4 signal handler interface offers additional functionality in certain circumstances: if you use the `sigaction` interface and you set the flag `SA_SIGINFO` in `sa_flags`, the signal handler is invoked as if it were defined

```
void handler (int signum,
              struct siginfo *info,
              struct ucontext *context);
```

siginfo is an enormous structure, defined in */usr/include/siginfo.h*, which starts with

```
struct siginfo
  {
  int si_signo;                          /* signal from signal.h */
  int si_code;                           /* code from above */
  int si_errno;                          /* error from errno.h */
...  more stuff, including space for further growth
  }
```

ucontext is defined in */usr/include/ucontext.h* and contains information about the user context at the time of the signal application. It includes the following fields:

- uc_sigmask is the blocked signal mask.
- us_stack points to the top of stack at the time the signal was delivered.
- uc_mcontext contains the processor registers and any implementation specific context data.

For example, assume you had set the signal handler for SIGFPE with the call in Example 13-1.

Example 13−1:

```
void bombout_handler (int signum,
                      struct siginfo *info,
                      struct ucontext *context);

sigset_t bombout_mask;
struct sigaction bad_error = {&bombout_handler, handler for the signal
                              &bombout_mask,    signals to mask
                              SA_SIGINFO};      we want additional info

sigemptyset (&bombout_mask);                    no signals in mask
sigaction (SIGFPE, &bad_error, NULL);
```

On receipt of a SIGFPE,

- signal will be set to SIGFPE.
- info->si_signo will also be set to SIGFPE.
- On an i386 machine, info->si_code might be, for example, FPE_INTDIV (indicating an integer divide by zero) or FPE_FLTUND (indicating fbating point underfbw).
- The value of info->si_errno can't be relied on to have any particular value.
- context->uc_sigmask contains the current signal mask.
- context->uc_stack will point to the stack in use at the time the signal was delivered.

- `context->uc_mcontext` will contain the contents of the processor registers at the time of the interrupt. This can be useful for debugging.

BSD signal handlers

BSD signal handlers do not use the flag `SA_SIGINFO` for `sa_flags`. Signal handlers always receive three parameters:

```
void handler (int signum, int code, struct sigcontext *context);
```

`code` gives additional information about certain signals—you can find this information in the header file */usr/include/machine/trap.h*. This file also contains information about how hardware interrupts are mapped to signals. `context` is hardware-dependent context information that can be used to restore process state under some circumstances. For example, for a Sparc architecture it is defined as

```
struct sigcontext
  {
  int   sc_onstack;                         /* sigstack state to restore */
  int   sc_mask;                            /* signal mask to restore */
  /* begin machine dependent portion */
  int   sc_sp;                              /* %sp to restore */
  int   sc_pc;                              /* pc to restore */
  int   sc_npc;                             /* npc to restore */
  int   sc_psr;                             /* psr to restore */
  int   sc_g1;                              /* %g1 to restore */
  int   sc_o0;                              /* %o0 to restore */
  };
```

The program of Example 13-1 won't compile under BSD, since BSD doesn't define `SA_SIGINFO`, and the parameters for `bombout_handler` are different. We need to modify it a little:

```
void bombout_handler (int signum,
                      int code,
                      struct sigcontext *context);

sigset_t bombout_mask;
struct sigaction bad_error = {&bombout_handler, handler for the signal
                              &bombout_mask,    signals to mask
                              0};
... the rest stays the same
```

If you enter this signal handler because of a `SIGFPE`, you might find:

- `signum` will be set to `SIGFPE`.
- On an i386 machine, `code` might be, for example, `FPE_INTOVF_TRAP` (indicating an integer divide by zero) or `FPE_FLTUND_TRAP` (indicating floating point underflow).
- The value of `sc_onstack` would be the previous *sigstack* state.
- `context->sc_mask` contains the current blocked signal mask, like `context->uc_sigmask` in the System V.4 example.

- The rest of the `context` structure shows the same kind of register information that System V.4 stores in `context->uc_mcontext`.

SIGCLD and SIGCHLD

System V treats the death of a child differently from other implementations: The System V signal `SIGCLD` differs from the BSD and POSIX.1 signal `SIGCHLD` and from all other signals by remaining active until you call `wait`. This can cause infinite recursion in the signal handler if you reinstate the signal via `signal` or `sigset` before calling `wait`. If you use the POSIX.1 `sigaction` call, you don't have to worry about this problem.

When a child dies, it becomes a *zombie*. As all voodoo fans know, a zombie is one of the Living Dead, neither alive nor dead. In UNIX terminology, when a child process dies it becomes a zombie: the text and data segments are freed, and the files are closed, but the process table entry and some other information remain until it is exorcized by the parent process, which is done by calling `wait`. By default, System V ignores `SIGCLD` and `SIGCHLD`, but the system creates zombies, so you can find out about child status by calling *wait*. If, however, you change the default to *explicitly* ignore the signal, the system ignores `SIGCHLD` and `SIGCLD`, but it also no longer creates zombie processes. If you set the disposition of `SIGCHLD` and `SIGCLD` to *ignore*, but you call `wait` anyway, it waits until *all* child processes have terminated, and then returns -1 (error), with `errno` set to `ECHILD`. You can achieve the same effect with `sigaction` by specifying the `SA_NOCLDWAIT` flag in `sa_flags`. There is no way to achieve this behaviour in other versions of UNIX: if you find your ported program is collecting zombies (which you will see with the *ps* program), it might be that the program uses this feature to avoid having to call `wait`. If you experience this problem, you can solve it by adding a signal handler for `SIGCLD` that just calls `wait` and returns.

The signal number for `SIGCLD` is the same as for `SIGCHLD`. The semantics depend on how you enable it: if you enable it with `signal`, you get `SIGCLD` semantics (and unreliable signals), and if you enable it with `sigaction` you get `SIGCHLD` and reliable signals. Don't rely on this, however. Some versions of System V have special coding to ensure that a separate `SIGCLD` signal is delivered for each child that dies.

Interrupted system calls

Traditional UNIX kernels differentiate between *fast* and *slow* system calls. Fast calls are handled directly by the kernel, while slow calls require the cooperation of other processes or device drivers. While the call is being executed, the calling process is suspended.

If a signal for a process occurs while the process is suspended, the behaviour depends both on whether the call is fast or slow, and on the signal implementation. On traditional systems, if the priority is numerically less than (of a higher priority than) the constant `PZERO`, the signal is slow and remains pending until the priority rises above `PZERO`. Otherwise it is fast, and the system call is interrupted. Typically, this means that disk and network operations are not interrupted, since they run at a priority below `PZERO`, whereas terminal and serial line operations can be interrupted. Some newer systems treat the relationship between priority and delivering signals more flexibly.

In the Seventh Edition, if a system call was interrupted, it returned an error, and `errno` was sent to `EINTR`. It was up to the process to decide whether to repeat the call or not. This added a significant coding overhead to just about every program; the result was that programs usually did not provide for interrupted system calls, and died when it happened.

Later signal implementations improved on this state of affairs:

- In 4.2BSD, signals automatically restarted the system calls `ioctl`, `read`, `readv`, `wait`, `waitpid`, `write` and `writev`.
- In 4.3BSD, the 4.2BSD signal implementation was modified so that the user could elect not to restart specific system calls after interruption. The default remained to restart the system call.
- In POSIX.1, when you call `sigaction` you can specify that system calls interrupted by specific signals should be restarted. This is done with the `SA_RESTART` flag in the field `sa_flags`. If this flag is not set, the calls will not be restarted.
- SunOS 4 does not have `SA_RESTART`, but it has `SA_INTERRUPT` instead, which is effectively the reverse of `SA_RESTART`: system calls will be restarted unless `SA_INTERRUPT` is set,

On modern systems, the action taken depends on the system calls you have used and the system you are using:

- With System V, you have the choice of no restart (unreliable `signal` or System V `sigset` and friends) or POSIX.1 selective restart based on the signal (`SA_RESTART` with `sigaction`).
- With BSD, you have the choice of no restart (reliable `signal` based on `sigaction`), default restart based on system calls (`sigvec` and friends) or again the POSIX.1 selective restart based on the signal (`SA_RESTART` with `sigaction`).

Calling functions from signal handlers

By definition, signals interrupt the normal flow of program execution. This can cause problems if they call a function that has already been invoked, and which has saved some local state. The function needs to be written specially to avoid such problems—it should block either all signals during execution, or, preferably, it should be written reentrantly. Either solution is difficult, and typically system libraries do not support this kind of reentrancy. On the other hand, there's not much you can do without calling some library routine. POSIX.1 defines "safe" routines that you can call from a signal handler. They are:

`_exit`	`access`	`alarm`	`cfgetispeed`	`cfgetospeed`
`cfsetispeed`	`cfsetospeed`	`chdir`	`chmod`	`chown`
`close`	`creat`	`dup`	`dup2`	`execle`
`execve`	`fcntl`	`fork`	`fstat`	`getegid`
`geteuid`	`getgid`	`getgroups`	`getpgrp`	`getpid`

getppid	getuid	kill	link	lseek
mkdir	mkfifo	open	pathconf	pause
pipe	read	rename	rmdir	setgid
setpgid	setsid	setuid	sigaction	sigaddset
sigdelset	sigemptyset	sigfillset	sigismember	sigpending
sigprocmask	sigsuspend	sleep	stat	sysconf
tcdrain	tcflow	tcflush	tcgetattr	tcgetpgrp
tcsendbreak	tcsetattr	tcsetpgrp	time	times
umask	uname	unlink	utime	wait
waitpid	write			

In addition, System V.4 allows `abort`, `exit`, `longjmp`, and `signal`.

Current signal implementations

In this section, we'll look at the differences between individual signal implementations. We'll concentrate on what you need to do to convert from one to another. If you *do* need to convert signal code, you should use the POSIX.1 signal implementation whenever practical.

Seventh Edition signal function

The Seventh Edition provided only one signal function, `signal`, which is the granddaddy of them all. All systems supply `signal`, though on some systems, such as newer BSD systems, it is a library function that calls `sigaction`. This also means that you can't rely on specific semantics if you use `signal`—avoid it if at all possible. Older UNIX systems (specifically, those that did not expect function prototypes to be used) implicitly defined the return type of `signal` to be an `int`. This does not change the meaning of the return value, but it can confuse more pedantic compilers. About the only system still on the market that returns an `int` from `signal` is XENIX.

BSD signal functions

The BSD signal functions were the first attempt at reliable signals, and they form the basis of the POSIX.1 implementation. All modern systems offer the POSIX.1 implementation as well, and on many BSD systems the functions described in this section are just an interface to the POSIX.1 functions.

Signal sets

A central difference between the Seventh Edition and System V implementations, on the one side, and the BSD and POSIX.1 implementations, on the other side, is the way signals can be specified. The Seventh Edition functions treat individual signals, which are specified by their number. The BSD routines introduced the concept of the *signal set*, a bit map of type *sigset_t*, that specifies any number of signals, as illustrated in Figure 13-1:

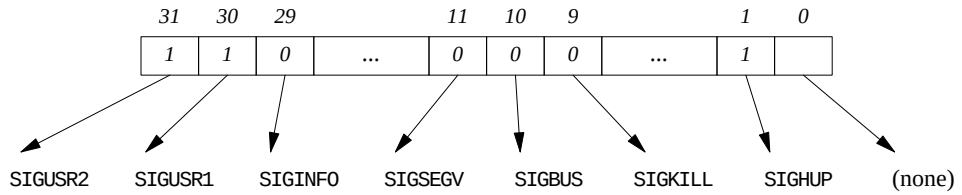

Figure 13−1. BSD and POSIX.1 signal sets

For each signal, if the corresponding bit in the bit map is set, the signal is said to be included in the set. In this example, the signals specified are `SIGUSR2`, `SIGUSR1` and `SIGHUP`. This method enables any number of signals to be specified as the parameter of one call.

The kernel maintains two special signal sets for each process: the *signal mask* and the *pending signal set*. The signal mask specifies which signals should currently not be delivered to the process—these signals are said to be *blocked*. This does not mean that they will be ignored: if a signal occurs while it is blocked, the kernel notes that it has occurred and sets its bit in the pending signal set. When a subesequent call to `sigsetmask` resets the bit for this signal in the signal mask, the kernel delivers the signal to the process and clears the bit in the pending signal set.

sigsetmask

`sigsetmask` sets the process signal mask:

```
#include <sys/signal.h>
int sigsetmask (int mask);
```

`sigsetmask` can be defined in terms of the POSIX.1 function `sigprocmask` using the `SIG_SETMASK` flag—see page 194 for more details.

sigblock

`sigblock` modifies the process signal mask. Unlike `sigsetmask`, it performs a logical OR of the specified mask with the current signal mask, so it can only block signals and not enable them.

```
#include <sys/signal.h>
int sigblock (int mask);
```

`sigblock` can be defined in terms of the POSIX.1 function `sigprocmask` using the `SIG_BLOCK` flag—see page 194 for more details.

sigvec

`sigvec` corresponds to the Seventh Edition *signal*: it sets the disposition of a signal. In addition, it can block other signals during the processing of a signal.

```
#include <signal.h>
... in signal.h is the definition
struct sigvec
  {
  void     (*sv_handler) ();
  sigset_t sv_mask;
  int      sv_flags;
  };

sigvec (int signum, struct sigvec *vec, struct sigvec *ovec);
```

`signum` is the signal whose disposition is to be changed. `vec` specifies the new disposition of the signal, and the function returns the old disposition to `ovec`.

If `vec->sv_mask` is non-zero, it specifies the signals to block while the signal handler is running. This is logically *or*ed with the current signal mask, so it works like an implicit `sigblock` on entering the signal handler. On exit from the signal handler, the kernel reinstates the previous signal mask.

`flags` can consist of:

- `SV_ONSTACK` specifies to take the signal on alternate signal stack, if one has been defined.
- `SV_INTERRUPT` specifies that system calls should not be restarted after the signal handler has completed.

`sigvec` is almost identical to the POSIX.1 function `sigaction` described on page 193—only the names of the `sigvec` structure and its members are different. Note, however, that the flag `SV_INTERRUPT` has the opposite meaning from the POSIX.1 flag `SA_RESTART`, which frequently has the same numeric value.

sigpause

`sigpause` combines the functionality of `sigmask` and `pause`: it first sets the signal mask and then calls *pause* to wait for a signal to occur.

```
#include <sys/signal.h>
int sigpause (sigset_t sigmask);
```

Typical use of BSD signal functions

Most signal coding consists of initialization. Typical programs set the disposition of the signals in which they are interested during program initialization, and don't change them much after that. For example, with BSD signals you might see code like that in Example 13-2.

Example 13−2:

Example 13−2: (continued)

```
struct sigvec hupvec = {&hup_handler, 0, 0}; /* disposition of SIGHUP */
struct sigvec iovec = {&io_handler, 1 << SIGHUP, 0}; /* disposition of SIGIO */
sigvec (SIGHUP, &hupvec, NULL);              /* instate handlers for SIGHUP, */
sigvec (SIGIO, &iovec, NULL);                /* SIGIO, */
sigvec (SIGURG, &iovec, NULL);               /* and SIGURG */
```

Occasionally a process will use `sigpause`, usually to wait for I/O. In Example 13-3, it blocks the signals `SIGINT` and `SIGQUIT`:

Example 13−3:

```
sigpause ((1 << SIGINT) | (1 << SIGQUIT));   /* wait for a signal */
```

System V signal functions

The following signal functions were implemented in System V and are effectively obsolete: the POSIX.1 functions have replaced them even in System V.3. The syntax of the function calls is more like the Seventh Edition than POSIX.1. In particular, they do not support the concept of a signal set. If you do find it necessary to replace System V signals with POSIX.1 signals, there is considerable scope for simplification by merging multiple System V calls (one per signal) into a single POSIX.1 call.

sigset

`sigset` is the System V reliable equivalent of `signal`:

```
#include <signal.h>
void (*sigset (int sig, void (*disp) (int))) (int);
```

Unlike `signal`, the signal is not disabled when the signal handler is executing—instead it is blocked until the signal handler terminates.

sighold

`sighold` blocks the delivery of signal `sig` by setting the corresponding bit in the process signal mask. Semantically this corresponds to the POSIX.1 function `sigprocmask` with the `SIG_BLOCK` flag, but it can block only one signal per call.

```
#include <signal.h>
int sighold (int sig);
```

sigrelse

`sigrelse` allows the delivery of signal `sig` by resetting the corresponding bit in the process signal mask. Semantically this corresponds to the POSIX.1 function `sigprocmask` with the `SIG_UNBLOCK` flag, but it can release only one signal per call.

```
#include <signal.h>
int sigrelse (int sig);
```

sigignore

`sigignore` sets the disposition of signal `sig` to `SIG_IGN`—the kernel ignores the signal.

```
#include <signal.h>
int sigignore (int sig);
```

sigpause

```
#include <signal.h>
int sigpause (int sig);
```

`sigpause` enables the delivery of signal `sig` and then waits for delivery of any signal.

CAUTION This is *not* the same as the BSD function `sigpause` described on page 190. BSD `sigpause` takes a signal mask as an argument, System V `sigpause` takes a single signal number. In addition, BSD `sigpause` only resets the mask temporarily—until the function return—whereas System V `sigpause` leaves it in this condition.

Example of System V signal functions

On page 190, we looked at what typical BSD code might look like. The System V equivalent of this program might perform the initialization in Example 13-4. System V doesn't supply the functionality associated with `SIGIO` and `SIGURG`—it uses `SIGPOLL` instead. See Chapter 14, *File systems*, pages 209 and 225, for more details of `SIGIO` and `SIGPOLL` respectively.

Example 13−4:

```
sigset (SIGHUP, &hup_handler);              /* instate handlers for SIGHUP */
sigset (SIGPOLL, &io_handler);              /* and SIGPOLL */
```

System V `sigpause` has a different syntax, so we need to set the signal mask explicitly with calls to `sighold`, and also to release them explicitly with `sigrelse`

Example 13−5:

```
sighold (SIGINT);                           /* block SIGINT */
sighold (SIGQUIT);                          /* and SIGQUIT */
sigpause (0);                               /* wait for something to happen */
sigrelse (SIGINT);                          /* unblock SIGINT */
sigrelse (SIGQUIT);                         /* and SIGQUIT */
```

POSIX.1 signal functions

All modern UNIX implementations claim to support POSIX.1 signals. These are the functions to use if you need to rewrite signal code. They are similar enough to the BSD functions to be confusing. In particular, the BSD functions pass signal masks as `long`s, whereas the POSIX.1 functions pass pointers to the signal mask—this enables the number of signals to exceed the number of bits in a `long`.

sigaction

`sigaction` is the POSIX.1 equivalent of `signal`. It specifies the disposition of a signal. In addition, it can specify a mask of signals to be blocked during the processing of a signal, and a number of flags whose meaning varies significantly from system to system.

```
#include <signal.h>
struct sigaction
  {
  void     (*sa_handler)();                /* handler */
  sigset_t sa_mask;                         /* signals to block during processing */
  int      sa_flags;
  };

void sigaction (int sig,
                const struct sigaction *act,
                struct sigaction *oact);
```

`signum` is the signal whose disposition is to be changed. `act` specifies the new disposition of the signal, and the function returns the old disposition to `oact`.

If `act->sa_mask` is non-zero, it specifies which signals to block while the signal handler is running. This is logically *or*ed with the current signal mask, so it works like an implicit `sigblock` on entering the signal handler.

Here's an overview of the flags:

Table 13−2: `sigaction` flags

Parameter	supported by	meaning
SA_ONSTACK	BSD, System V	Take the signal on the alternate signal stack, if one has been defined. POSIX.1 does not define the concept of an alternate signal stack—see page 196 for more details. Linux plans similar functionality with the SA_STACK flag, but at the time of writing it has not been implemented.
SA_RESETHAND	System V	Reset the disposition of this signal to SIG_DFL when the handler is entered (simulating Seventh Edition semantics). This is the same as the Linux SA_ONESHOT flag.
SA_ONESHOT	Linux	Reset the disposition of this signal to SIG_DFL when the handler is entered (simulating Seventh Edition semantics). This is the same as the System V SA_RESETHAND flag.

Table 13−2: `sigaction` flags (continued)

Parameter	supported by	meaning
SA_RESTART	BSD, Linux, System V	Restart system calls after the signal handler has completed (see page 186).
SA_SIGINFO	System V	Provide additional parameters to signal handler (see page 183).
SA_NODEFER	System V	Don't block this signal while its signal handler is active. This means that the signal handler can be called from a function which it calls, and thus needs to be reentrant.
SA_NOCLDWAIT	System V	Don't create zombie children on SIGCLD (see page 186).
SA_NOCLDSTOP	Linux, System V	Don't generate SIGCHLD when a child stops, only when it terminates.
SA_NOMASK	Linux	Disable the signal mask (allow all signals) during the execution of the signal handler.
SA_INTERRUPT	Linux	Disable automatic restart of signals. This corresponds to the SunOS 4 flag SV_INTERRUPT to sigvec (see page 190). Currently not implemented.

sigprocmask

sigprocmask manipulates the process signal mask. It includes functional modes that correspond to both of the BSD functions sigblock and sigsetmask:

```
#include <signal.h>
int sigprocmask (int how, const sigset_t *set, sigset_t *oset)
```

The parameter how determines how the mask is to be manipulated. It can have the following values:

Table 13−3: sigprocmask functional modes

Parameter	meaning
SIG_BLOCK	Create a new signal mask by logically *or*ing the current mask with the specified set.
SIG_UNBLOCK	Reset the bits in the current signal mask specified in set.
SIG_SETMASK	Replace the current signal mask by set.

sigpending

```
#include <signal.h>
int sigpending (sigset_t *set);
```

`sigpending` returns the pending signal mask to `set`. These are the signals pending delivery but currently blocked, which will be delivered as soon as the signal mask allows. The return value is an error indication and *not* the signal mask. This function does not have an equivalent in any other signal implementation

sigsuspend

```
#include <sys/signal.h>
int sigsuspend (const sigset_t *sigmask);
```

`sigsuspend` temporarily sets the process signal mask to `sigmask`, and then waits for a signal. When the signal is received, the previous signal mask is restored on exit from `sigsuspend`. It always returns -1 (error), with `errno` set to `EINTR` (interrupted system call).

Example of POSIX.1 signal functions

On page 190, we looked at a simple example of signal setup, and on page 192 we changed it for System V. Changing it from BSD to POSIX.1 is mainly a matter of changing the names. We change the calls to `sigvec` to `sigaction`, and their parameters are now also of type `struct sigaction` instead of `struct sigvec`.

Unfortunately, there is a problem with this example: POSIX.1 does not define any of the I/O signals to which this example refers. This is not as bad as it sounds, since there are no pure POSIX.1 systems, and all systems offer either `SIGIO`/`SIGURG` or `SIGPOLL`. In Example 13-6, we'll stick with the BSD signals `SIGIO` and `SIGURG`:

Example 13−6:

```
struct sigaction hupvec = {&hup_handler, 0, 0}; /* disposition of SIGHUP */
struct sigaction iovec = {&io_handler, 1 << SIGHUP, 0}; /* disposition of SIGIO */
sigaction (SIGHUP, &hupvec, NULL);          /* instate handlers for SIGHUP, */
sigaction (SIGIO, &iovec, NULL);            /* SIGIO, */
sigaction (SIGURG, &iovec, NULL);           /* and SIGURG */
sigset_t blockmask;                         /* create a mask */
sigemptyset (&blockmask);                   /* clear signal mask */
sigaddset (&blockmask, SIGINT);             /* add SIGINT to the mask */
sigaddset (&blockmask, SIGQUIT);            /* add SIGQUIT to the mask */
```

Example 13-7 shows the corresponding call to sigsuspend:

Example 13−7:

```
sigsuspend (&blockmask);                    /* let the action begin */
```

We'll look at `sigemptyset` and `sigaddset` in the next section. It's unfortunate that this part of the initialization looks so complicated—it's just part of the explicit programming style that POSIX.1 desires. On most systems, you could get the same effect without the calls to `sigemptyset` and `sigaddset` by just defining

```
int blockmask = (1 << SIGINT) | (1 << SIGQUIT);
sigpause ((sigset_t *) &blockmask);          /* let the action begin */
```

The only problem with this approach (and it's a showstopper) is that it's not portable: on a different system, `sigset_t` might not map to `int`.

Signals under Linux

Linux signals are an implementation of POSIX.1 signals, and we discussed some of the details in the previous section. In addition, it's good to know that:

- For compatibility, `SIGIOT` is defined as `SIGABRT`. POSIX.1 does not define `SIGIOT`.
- As we saw, POSIX.1 does not supply the signals `SIOPOLL`, `SIGIO` and `SIGURG`. Linux does, but they it maps all three signals to the same numerical value.
- If you really want to, you can simulate unreliable signals under Linux with `sigaction` and the `SA_ONESHOT` flag.

Other signal-related functions

A significant advantage of the BSD and POSIX.1 signal functions over the Seventh Edition and System V versions is that they have signal set parameters. The down side of signal sets is that you need to calculate the values of the bits. The following functions are intended to make manipulating these structures easier. They are usually implemented as macros:

- `sigemptyset (sigset_t *set)` sets `set` to the "empty" signal set—in other words, it excludes all signals.
- `sigfillset (sigset_t *set)` sets all valid signals in `set`.
- `sigaddset (sigset_t *set, int signum)` adds signal `signum` to `set`.
- `sigdelset (sigset_t *set, int signum)` removes signal `signum` from `set`.
- `sigismember (sigset_t *set, int signum)` returns 1 if `signum` is set in `set` and 0 otherwise.

sigstack and sigaltstack

As we have already discussed, a signal is like a forced function call. On modern processors with stack-oriented hardware, the call uses stack space. In some cases, a signal that arrives at the wrong time could cause a stack overflow. To avoid this problem, both System V and BSD (but not POSIX.1) allow you to define a specific *signal stack*. On receipt of a signal, the stack is switched to the alternate stack, and on return the original stack is reinstated. This can also occasionally be of interest in debugging: if a program gets a signal because of a reference beyond the top of the stack, it's not much help if the signal destroys the evidence.

BSD supplies the `sigstack` system call:

```
#include <sys/signal.h>
struct sigstack
  {
  caddr_t ss_sp;                          /* Stack address */
  int     ss_onstack;                     /* Flag, set if currently
                                * executing on this stack */
  };
int sigstack (const struct sigstack *ss, struct sigstack *oss);
```

- `ss` may be `NULL`. If it is not, the process signal stack is set to `ss->ss_sp`.
- `ss->ss_onstack` tells `sigstack` whether the process is currently executing on the stack.
- `oss` may also be `NULL`. If it is not, information about the current signal stack is returned to it.

System V supplies the function `sigaltstack`:

```
#include <signal.h>
typedef struct
  {
  char *ss_sp;                            /* Stack address */
  int   ss_size;                          /* Stack size */
  int   ss_flags;                         /* Flags, see below */
  }
stack_t;
int sigaltstack (const stack_t *ss, stack_t *oss);
```

- `ss` may be `NULL`. If it is not, the process signal stack is set to `ss->ss_sp`, and its size is set to `ss->ss_size`.
- `oss` may also be `NULL`. If it is not, information about the current signal stack is returned to it.
- The structure element `ss_flags` may contain the following flags:
  - `SS_DISABLE` specifies that the alternate stack is to be disabled. `ss_sp` and `ss_size` are ignored. This flag is also returned in `oss` when the alternate stack is disabled.
  - `SS_ONSTACK` (returned) indicates that the process is currently executing on the alternate stack. If this is the case, a modification of the stack is not possible.

setjmp and longjmp

When you return from a function, C language syntax does not give you a choice of where to return to: you return to the instruction after the call. Occasionally, deep in a series of nested function calls, you will discover you need to return several levels down the stack—effectively, you want to perform multiple returns. Standard “structured programming” techniques do not handle this requirement well, and you can’t just perform a `goto` to the location, because that would leave the stack in a mess. The library functions `setjmp` and `longjmp` provide this *non-local return*.

What does this have to do with signals? Nothing, really, except that the receipt of a signal is one of the most common reasons to want to perform a non-local return: a signal can interrupt processing anywhere where the process signal mask allows it. In many cases, the result of the signal processing is not related to the processing that was interrupted, and it may be necessary to abort the processing and perform a non-local return. For example, if you are redisplaying data in an X window and the size of the window changes, you will get a SIGWINCH signal. This requires a complete recalculation of what needs to be displayed, so there is no point in continuing the current redisplay operation.

Non-local returns are implemented with the functions setjmp, longjmp, and friends. setjmp saves the process context and longjmp restores it—in other words, it returns to the point in the program where setjmp was called. Unlike a normal function return, a longjmp return may involve discarding a significant part of the stack. There are a number of related functions:

```
#include <setjmp.h>

int setjmp (jmp_buf env);
void longjmp (jmp_buf env, int val);
int _setjmp (jmp_buf env);
void _longjmp (jmp_buf env, int val);
void longjmperror (void);
int sigsetjmp (sigjmp_buf env, int savemask);
void siglongjmp (sigjmp_buf env, int val);
```

The definitions of jmp_buf and sigjmp_buf are less than illuminating: they are just defined as an array of ints long enough to contain the information that the system saves. In fact, they contain the contents of the registers that define the process context—stack pointer, frame pointer, program counter, and usually a number of other registers.

From the user point of view, setjmp is unusual in that it can return more often than you call it. Initially, you call setjmp and it returns the value 0. If it returns again, it's because the program called longjmp, and this time it returns the value parameter passed to longjmp, which normally should not be 0. The caller can then use this value to determine whether this is a direct return from setjmp, or whether it returned via longjmp:

```
int return_code = setjmp (env);
if (return_code)
  {                                          /* non-0 return code: return from longjmp */
  printf ("longjmp returned %d\n", return_code);
  }
```

These functions are confusing enough in their own right, but they also have less obvious features:

- It doesn't make any sense for longjmp to return 0, and System V.4 longjmp will never return 0, even if you tell it to—it will return 1 instead. BSD longjmp will return whatever you tell it to.
- The setjmp functions save information about the state of the function that called them. Once this function returns, this information is no longer valid. For example, the

following code will not work:

```
jmp_buf env;                                /* save area for setjmp */

int mysetjmp ()
  {
  int a = 0;
  if (a = setjmp (env))
    printf ("Bombed out\n");
  return a;
  }

foo ()
  {
  ...
  mysetjmp ();                              /* catch bad errors */
  ...
  }
```

The return instruction from `mysetjmp` to `foo` frees its local environment. The memory which it occupies, and which the call to `setjump` saved, will be overwritten by the next function call, so a `longjmp` cannot restore it.

- BSD attempts to determine whether the parameter `env` to the `longjmp` functions is invalid (such as in the example above). If it detects such an error, it will call `longjmperror`, which is intended to inform that the `longjmp` has failed. If `longjmperror` returns, the process is aborted.

  If `longjmp` does not recognize the error, or if the system is not BSD, the resulting process state is indeterminate. To quote the System V.4 man page: *If longjmp is called even though env was never primed by a call to setjmp, or when the last such call was in a function that has since returned, absolute chaos is guaranteed.* In fact, the system will probably generate a `SIGSEGV` or a `SIGBUS`, but the core dump will probably show nothing recognizable.

- When `longjmp` returns to the calling function, automatic variables reflect the last modifications made to them in the function. For example:

```
int foo ()
  {
  int a = 3;
  if (setjmp (env))
    {
    printf ("a: %d\n", a);
    return a;
    }
  a = 2;
  longjmp (env, 4);
  }
```

At the point where `longjmp` is called, the variable `a` has the value 2, so this function will print `a:2`.

- When `longjmp` returns to the calling function, register variables will normally have the values they had at the time of the call to `setjmp`, since they have been saved in the jump buffer. Since optimizers may reassign automatic variables to registers, this can have confusing results. If you compile the example above with *gcc* and optimize it, it will print `a: 3`. This is clearly an unsuitable situation: the solution is to declare `a` to be *volatile* (see Chapter 20, *Compilers*, page 340 for more information). If we do this, `a` will always have the value 2 after the `longjmp`.
- BSD `setjmp` includes the signal mask in the state information it saves, but System V.4 `setjmp` does not save the signal mask. If you want to simulate System V.4 semantics under BSD, you need to use `_setjmp` and `_longjmp`, which do not save the signal mask. In either system, you can use `sigsetjmp`, which saves the signal mask only if `save` is non-zero. Except for the type of its first parameter, the corresponding `siglongjmp` is used in exactly the same manner as `longjmp`.
- The functions must be paired correctly: if you `_setjmp`, you must `_longjmp`, and if you `setjmp` you must `longjmp`.

kill

`kill` is one of the most badly named system calls in the UNIX system. Its function is to send a signal:

```
#include <signal.h>
int kill (pid_t pid, int sig);
```

Normally, `pid` is the process ID of the process that should receive the signal `sig`. There are a couple of additional tricks, however:

- If `pid` is 0, the kernel sends `sig` to all processes whose process group ID is the same as the group ID of the calling process.
- If `pid` is -1, most implementations broadcast the signal to all user processes if the signal is sent by *root*. Otherwise the signal is sent to all processes with the same effective user ID. BSD does not broadcast the signal to the calling process, System V does. POSIX.1 does not define this case.
- If `pid` is < -1, System V and BSD broadcast the signal to all processes whose process group ID is `abs (pid)` (`abs` is the absolute value function). Again, non-*root* processes are limited to sending signals to processes with the same effective user ID. BSD can also perform this function with the call `killpg`.

Another frequent use of `kill` is to check whether a process exists: `kill (pid, 0)` will not actually send a signal, but it will return success if the process exists and an error indication otherwise.

killpg

`killpg` broadcasts a signal to all processes whose process group ID is `abs (pid)`. It is supplied with BSD systems:

```
#include <sys/signal.h>

int killpg (pid_t pgrp, int sig);
```

This function sends the signal to the process group of the specified process, assuming that you have the same effective user ID as the recipient process, or you are super-user. You can use pid 0 to indicate your own process group. If you don't have this function, you can possibly replace it with `kill(-pgid)`—see the section on *kill* above.

raise

`raise` is an ANSI C function that enables a process to send a signal to itself. It is defined as

```
int raise (int signum);
```

Older systems don't have `raise`. You can fake it in terms of `kill` and `getpid`:

```
kill (getpid (), signum);
```

sys_siglist and psignal

At the name implies, `sys_siglist` is a list and not a function. More exactly, it is an array of signal names, indexed by signal number, and is typically supplied with BSD-derived systems. For example,

```
printf ("Signal %d (%s)\n", SIGSEGV, sys_siglist [SIGSEGV]);
```

returns

```
Signal 11 (Segmentation fault)
```

Some systems supply the function `psignal` instead of `sys_siglist`. It prints the text corresponding to a signal. You can get almost the same effect as the `printf` above by writing

```
char msg [80];
sprintf (msg, "Signal %d", SIGSEGV);
psignal (SIGSEGV, msg);
```

This gives the output:

```
Signal 11: Segmentation fault
```