Signals

Signals

Signals are software interrupts.

Signals are a way for a process to be notified of asynchronous events.
Some examples:

- a timer you set has gone off (SIGALRM)
- some I/O you requested has occurred (SIGIO)
- a user resized the terminal ”window” (SIGWINCH) a user disconnected from the system (SIGHUP)
- ...

See also: signal(2)/signal(3)/signal(7) (note: these man pages vary significantly across platforms!)

Signals

Besides the asynchronous events listed previously, there are many ways to generate a signal:

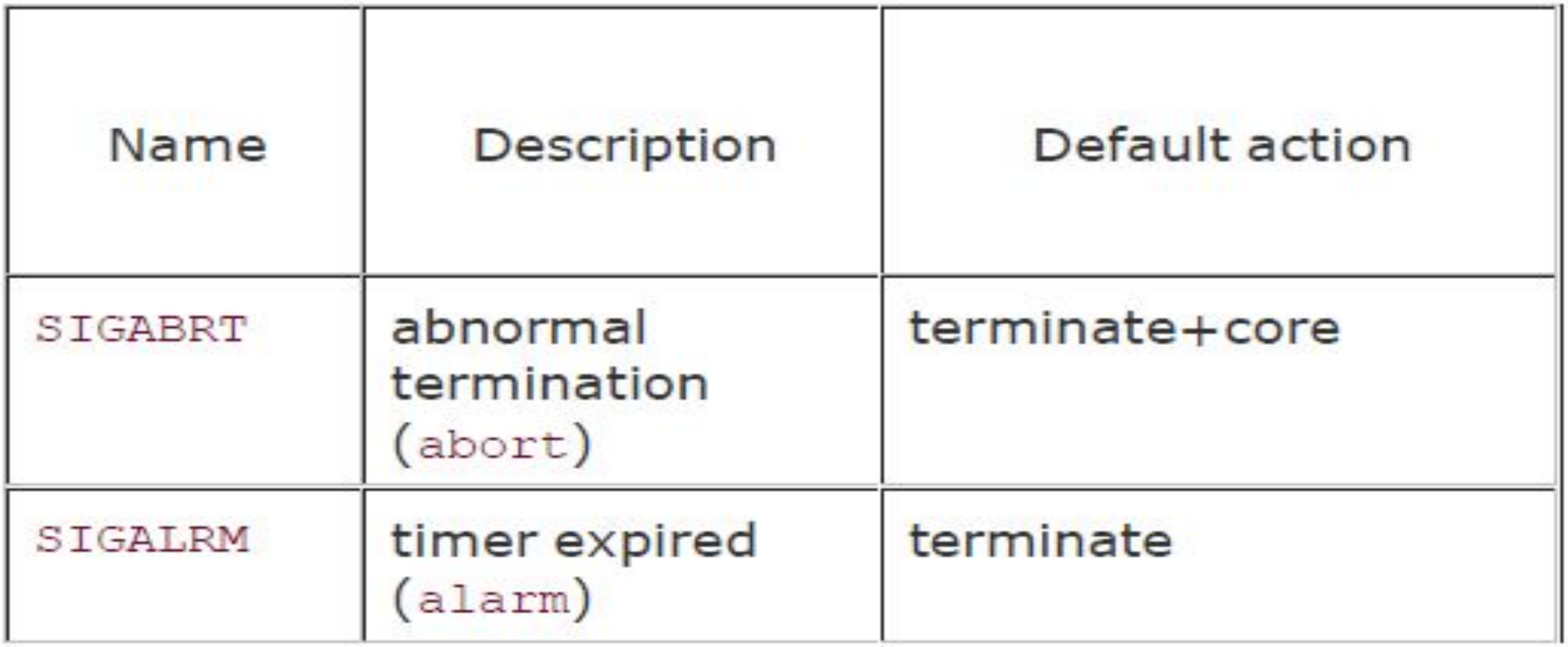
- terminal generated signals (user presses a key combination which causes the terminal driver to generate a signal)
- hardware exceptions (divide by 0, invalid memory references, etc)
- kill(1) allows a user to send any signal to any process (if the user is the owner or superuser)
- kill(2) (a system call, not the unix command) performs the same task
- software conditions (other side of a pipe no longer exists, urgent data has arrived on a network file descriptor, etc.)

Signal Concepts

Once we get a signal, we can do one of several things:

1. Ignore it. (note: there are some signals which we CANNOT or SHOULD NOT ignore)
2. Catch it. That is, have the kernel call a function which we define whenever the signal occurs.
3. Accept the default. Have the kernel do whatever is defined as the default action for this signal

Signal Concepts



Name	Description	Default action
SIGABRT	abnormal termination (abort)	terminate+core
SIGALRM	timer expired (alarm)	terminate

signal(3)

```
#include <signal.h>
void (*signal(int signo, void (*func)(int)))(int);
```

Returns: previous disposition of signal if OK, SIG_ERR otherwise

func can be:

- SIG_IGN which requests that we ignore the signal signo
- SIG_DFL which requests that we accept the default action for signal signo
- or the address of a function which should catch or handle a signal(signal handler or the signal-catching function)

signal(3)

- 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)).
- i.e., 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.

sigaction(2)

```
#include <signal.h>
int sigaction(int signo, const struct sigaction *act,
struct sigaction *oldact);
```

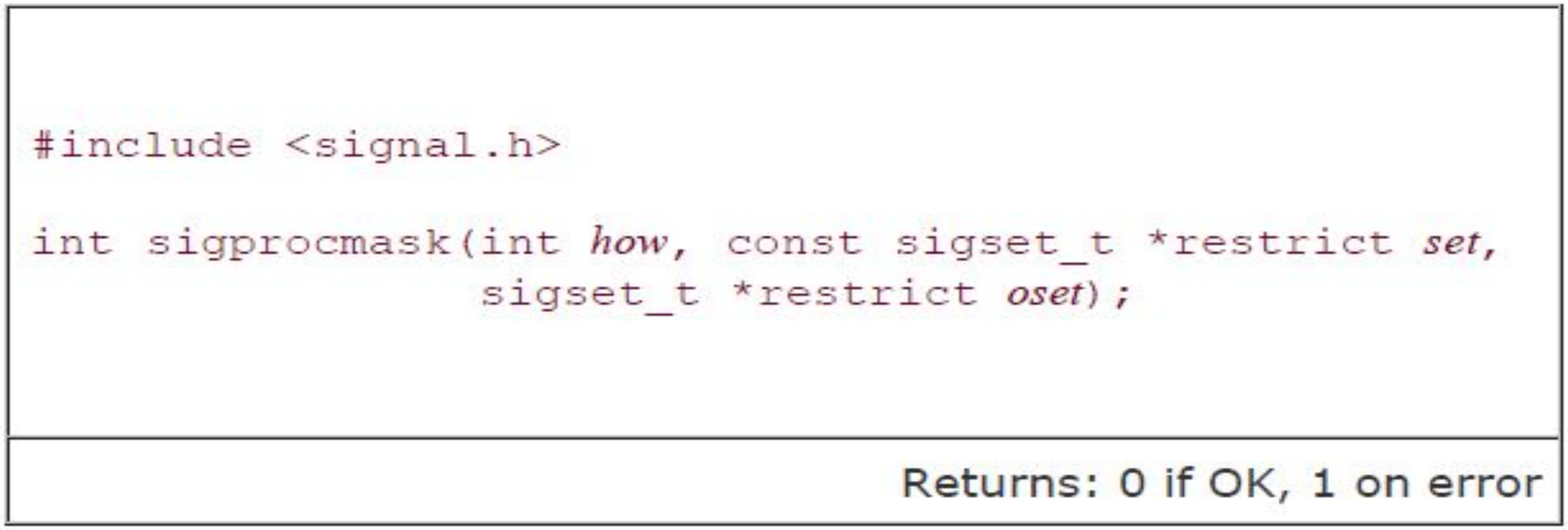
This function allows us to examine or modify the action associated with a particular signal.

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

signal(3) is (nowadays) commonly implemented via sigaction(2).

sigprocmask function

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

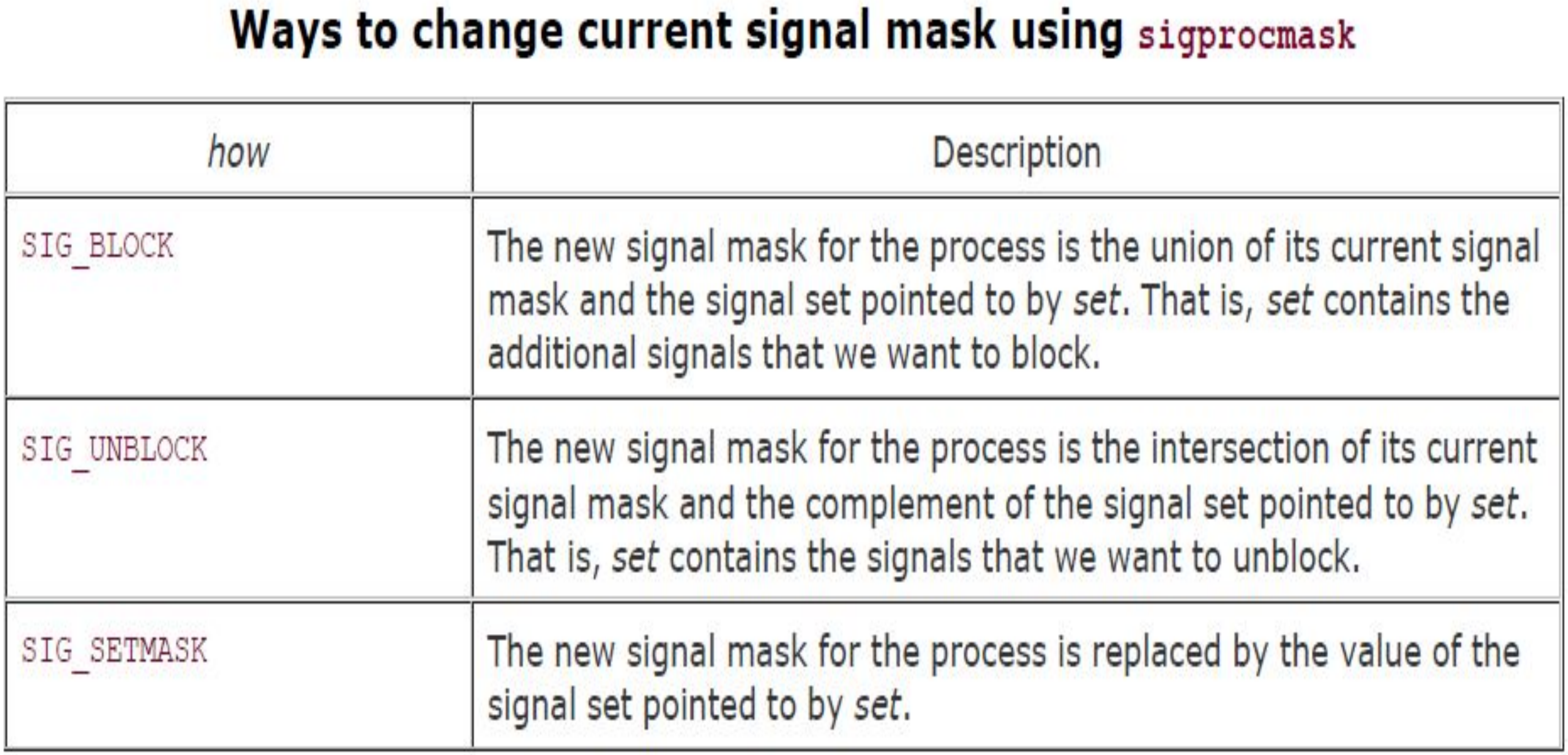


```
#include <signal.h>

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

Returns: 0 if OK, 1 on error

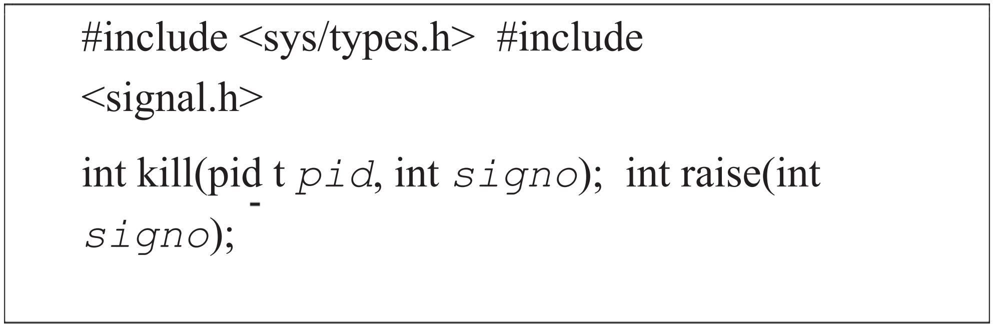
sigprocmask function



Ways to change current signal mask using `sigprocmask`

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

kill(2) and raise(3)



```
#include <sys/types.h>  #include
<signal.h>
int kill(pid_t pid, int signo);  int raise(int
signo);
```

- *pid > 0* – signal is sent to the process whose PID is pid
- *pid == 0* – signal is sent to all processes whose process group ID equals the process group ID of the sender
- *pid == -1* – POSIX.1 leaves this undefined, BSD defines it (see kill(2))

More advanced signal handling via signal sets

- int sigemptyset(sigset t *set) – intialize a signal set to be empty
- int sigfillset(sigset t *set) – initialize a signal set to contain all signals
  int sigaddset(sigset t *set, int signo) int
- sigdelset(sigset t *set, int signo) int
- sigismember(sigset t *set, int signo)

_

Interrupted System Calls

Some system calls can block for long periods of time (or forever). These include things like:

- read(2)s from files that can block (pipes, networks, terminals)
- write(2) to the same sort of files
- open(2) of a device that waits until a condition occurs (for example, a modem)
- pause(3), which purposefully puts a process to sleep until a signal occurs
- certain ioctl(3)s
- certain IPC functions

Catching a signal during execution of one of these calls traditionally led to the process being aborted with an errno return of EINTR.

Interrupted System Calls

Previously necessary code to handle

EINTR: again:

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

Nowadays, many Unix implementations automatically restart certain system calls.