

CSE 3320

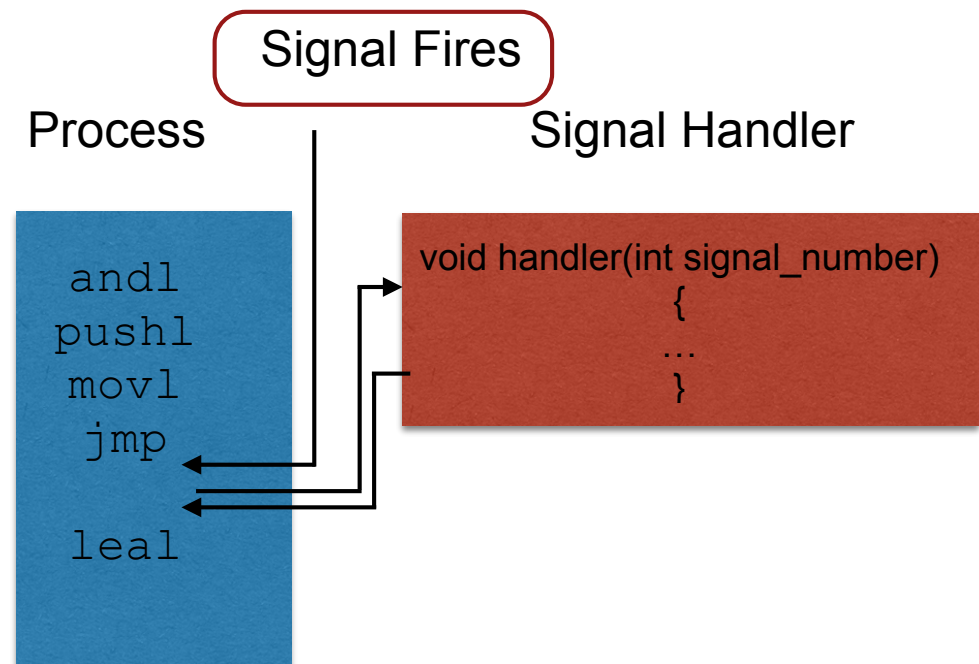
Signals

Trevor Bakker

The University of Texas at Arlington

Signals

- A signal is a asynchronous notification of an event
- Signals are how the operating system communicates with an application process.
- When a signal is received the process stops running and the assigned signal handler is run.
- When the signal has been handled the process resumes where it left off.



Some examples

- SIGSEGV - Known as a segmentation fault. It occurs when a process makes an illegal memory reference. The process halts and the default signal handler exits the process.
- SIGINT - The interrupt signal occurs when the user types ctrl-c. The default signal handler will exit the process.

Sending Signals

- By keyboard
 - ctrl-c sends the SIGINT signal and the receiving process exits
 - ctrl-z sends the SIGTSTP signal and the receiving process is suspended
 - ctrl-\ sends the SIGQUIT signal and the receiving process exits

Sending signals via shell commands

`kill -signal pid`

- Send a signal, specified by `-signal`, to the process with the process id `pid`.
- If no signal is specified then the signal sent is `SIGTERM`

Example:

`kill -9 3412`

`kill -SIGQUIT 3412`

Send signal via a function

```
int kill( pid_t pid, int sig)
```

- Send a signal, specified by -signal, to the process with the process id pid.
- If no signal is specified then the signal sent is SIGTERM

Example:

```
pid_t pid = getpid();    // process gets its own pid  
kill(pid, SIGINT);       // and sends itself a SIGINT
```

Send signal via a function

```
int raise( int sig )
```

- Send a signal to the current process

Example:

```
raise(SIGINT); // process sends itself a SIGINT
```

Signal Numbers

```
[bakker@crystal ~]$ kill -l
```

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL
5) SIGTRAP	6) SIGABRT	7) SIGBUS	8) SIGFPE
9) SIGKILL	10) SIGUSR1	11) SIGSEGV	12) SIGUSR2
13) SIGPIPE	14) SIGALRM	15) SIGTERM	16) SIGSTKFLT
17) SIGCHLD	18) SIGCONT	19) SIGSTOP	20) SIGTSTP
21) SIGTTIN	22) SIGTTOU	23) SIGURG	24) SIGXCPU
25) SIGXFSZ	26) SIGVTALRM	27) SIGPROF	28) SIGWINCH
29) SIGIO	30) SIGPWR	31) SIGSYS	34) SIGRTMIN
35) SIGRTMIN+1	36) SIGRTMIN+2	37) SIGRTMIN+3	38) SIGRTMIN+4
39) SIGRTMIN+5	40) SIGRTMIN+6	41) SIGRTMIN+7	42) SIGRTMIN+8
43) SIGRTMIN+9	44) SIGRTMIN+10	45) SIGRTMIN+11	46) SIGRTMIN+12
47) SIGRTMIN+13	48) SIGRTMIN+14	49) SIGRTMIN+15	50) SIGRTMAX-14
51) SIGRTMAX-13	52) SIGRTMAX-12	53) SIGRTMAX-11	54) SIGRTMAX-10
55) SIGRTMAX-9	56) SIGRTMAX-8	57) SIGRTMAX-7	58) SIGRTMAX-6
59) SIGRTMAX-5	60) SIGRTMAX-4	61) SIGRTMAX-3	62) SIGRTMAX-2
63) SIGRTMAX-1	64) SIGRTMAX		

Signal Terminology

- A signal is *generated* for a process when the event that causes the signal occurs. The event can be a hardware exception, a software condition, or a call to the kill function
- When a signal is generated the kernel usually sets a flag in the process table

Signal Terminology

- A signal is *delivered* to a process when the action for a signal is taken.
- During the time between the generation of the signal and the delivery of the signal the signal is said to be *pending*.

Signal Handling

- Every signal is assigned a default handler. Usually they just exit the process.
- Programs can install their own signal handlers for most signals.
- Can't install handlers for:
 - SIGKILL
 - SIGSTOP

Installing a signal handler

SIGACTION(2)

Linux Programmer's Manual

SIGACTION(2)

NAME

`sigaction` – examine and change a signal action

SYNOPSIS

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

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

- The function *handler* specified in the `sigaction` struct is installed as the new handler for the signal specified by `signum`.
- The new handler is called every time the process receives a signal of type `signum`.

Signal Handler Example

- See `sigint.c` and `multiple_signal_handlers.c` on course website under Code Samples

Blocking Signals

- We can block signals, but why would we want to?
 - Race conditions. What happens when a signal is received while we are in the middle of handling the same type of signal?
- The POSIX standard provides a way to block signals.

Blocking Signals

- Each process has a signal mask
 - The operating system uses the mask to determine which signals to deliver.
- The function `sigprocmask()` provides a way for programs to modify their signal mask to block and unblock signals.

Blocking Signals

- If a signal is generated for a process but blocked then the signal remains pending for the process until the process either:
 - unblocks the signal
 - changes the action to ignore the signal

Multiple Blocked Signals

- The POSIX standard allows the OS to deliver one or multiple copies of the signal.
- Most UNIX implementations do not queue signals and just deliver a single one.

sigprocmask()

SIGPROCMASK(2)

BSD System Calls Manual

SIGPROCMASK(2)

NAME

sigprocmask -- manipulate current signal mask

SYNOPSIS

#include <signal.h>

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

- First parameter is *how*. It can be:
 - SIG_BLOCK - adds the provided set to the current signal mask
 - SIG_UNBLOCK - removes the provided set from the current signal mask
 - SIG_SETMASK - the current mask is replaced by the provided set

sigprocmask()

SIGPROCMASK(2)

BSD System Calls Manual

SIGPROCMASK(2)

NAME

sigprocmask -- manipulate current signal mask

SYNOPSIS

#include <signal.h>

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

- Second parameter is a pointer to the a signal mask called *set*.
- Third parameter is a pointer to which the old set can be returned.

Signal Blocking Example

- See `sig_set_example.c` on course website under Code Samples