CptS 360 (System Programming)
Unit 11: Signals

Bob Lewis

School of Engineering and Applied Sciences
Washington State University

Spring, 2022

Motivation

- Signals are the most basic interprocess communication mechanisms.
- If they work for you, they are comparatively simple to use.
- They're supported *everywhere*.

References

- Stevens & Rago Ch. 10
- *man* pages

Signals Overview

- signals are:
  - "software interrupts"
  - asynchronous
- signals have names
- see *signal(7)*

Signals are Generated by:

- terminal (or keyboard) (e.g., Ctrl-C)
- hardware exceptions, e.g.
  - divide by 0
  - floating point overflow/underflow
  - segmentation violation
  - bus error
- software conditions (not hardware related):
  - `SIGALRM`
  - `SIGPIPE`
  - `SIGURG`
- explicitly (see below)
  - from the command line (e.g., *kill(1)*)
  - from a program (e.g., *kill(2)*)

Possible Signal Actions

- ignore the signal
- catch the signal (call a user-defined "handler" function)
- do the default thing:
  - ignore
  - terminate
  - dump core and terminate
- stop process
- continue (stopped process)

Explicitly Sending Signals

All signal implementations use these.

- `kill(pid, sig)`
  sends a signal `sig` to a process `pid`
- `raise(sig)`
  sends a signal to the process that calls it

Signal Implementations: An Overview

- *signal(2)*
  - simplest kind of signal handling
  - sometimes all you need
  - supported by everybody, including POSIX
- *sigset(3)*
  - System V
  - non-POSIX
  - avoid use
- *sigvec(3)*
  - BSD
  - non-POSIX
  - avoid use
- *sigaction(2)*
  - "robust"
  - supported by everybody, including POSIX
  - use for detailed control

signal(2) is Unreliable (aka "Non-Robust")

- *signal(2)* (q.v.) works most of the time, but...
- handler reset to default behavior when called
  So even if you immediately restore the handler, there's a short interval when the default behavior (e.g., "`Core`") can happen.

Interrupted System Calls

- What happens to interrupts that happen during a system call? (recall system call vs. function)
- What you want to have happen may depend on the call:
  - If the call is going to complete pretty quickly, wait until the call completes before delivering the signal normally.
    - This is a "fast" system call.
    - example: *time(2)*
  - If the call might wait a long time, interrupt the call and return an error to the user.
    - This is a "slow" system call.
    - example: *wait(2)* (q.v.)
    - reads from/writes to pipes, terminal devices, network devices
    - waits on external conditions (e.g. modem)
    - Scenario: User walks away from a terminal the program is reading.
    - caller must deal with the interrupt (`errno` is `EINTR`).
    - exception: disk I/O is *never* considered slow

Automatic Restart

- system call restarted after signal is handled, if it is
- allowed in 4.2BSD for system calls on slow devices: *ioctl(2)*, *read(2)*, *readv(2)*, *write(2)*, and *writev(2)*.
- and *always* for *wait(2)* and *waitpid(2)*.
- S & R Figure 10.3 discusses various implementations of signals and automatic restart

Dealing with an Interruptable System Call

Interrupted slow calls return error (-1) and set errno to EINTR.
For example, if `fd` was an open terminal, you might use this code:

```
while ((n = read(fd, buf, ct)) < 0) {
    if (errno != EINTR) {
        // deal with other error corresponding to ‘errn
    }
}
```

Non-Reentrant Functions

- Review: What are they?
- Q: Why do they cause problems when called from signal handlers, even when you're not using multiple threads?
  - Suppose you're in the middle of a non-reentrant function when a signal handler is called.
- General rules:
  - Never call a non-reentrant function from a signal handler.
  - If your handler calls a system function, save and restore `errno`.

SIGCHLD Semantics

- On Sys V systems, it's `SIGCLD` and it behaves oddly. (That's all we'll say about that.)
- On BSD systems (and Linux), parent sets handler that will get called when child's status changes.

Reliable ("Robust") Signal Technology and Semantics

- New, improved.
- Terminology: signals are...
  - generated
    The signal has been by sent by the signalling process.
  - pending
    The signal has been generated and is placed at the end of the signal queue to be delivered.
  - blocked
    When the signal is generated, it waits in a signal queue until it is unblocked.
  - delivered
    Action taken (e.g. handler called). The action may be reset while the signal is blocked.

Multiple Signals

Dealing with multiple signals depends on whether they're real-time or not.

- non-real time signals (e.g., `SIGINT`)
  - repeats of the same signal are ignored within the handler
  - with robust signal handling (i.e., *sigaction(2)*) alternating signals are blocked, not ignored (as with *signal(2)*))
  - *signal(2)*
- real time signals (numbers between `SIGRTMIN` and `SIGRTMAX`, inclusive)
  - no predefined interpretation (cf. `SIGUSR*`)
  - allow queueing
  - may be prioritized

sigaction(2)

- the primary robust signal function
- sets
  - handler
  - mask (a `sigset_t`) of additional signals to block while handler is called
  - a variety of flags

Signal Set (`sigset_t`)

- definition:
  the set of signals currently-blocked by the process
- formerly represented by an "`unsigned int`" signal mask, they now use a `sigset_t` (which is actually an array)

Allows more signals than will fit in an `int` (or a `long`) variable, even though that's all there are now.

- *sigemptyset(3)*
- *sigfillset(3)*
- *sigaddset(3)*
- *sigdelset(3)*
- *sigismember(3)*

sigprocmask(2)

- queries and/or sets signal mask
- arguments allow adding to, subtracting from, or setting mask

sigpending(2)

- lets you see what signals, if any, are pending
- sets argument

sigsetjmp(2) and *siglongjmp(2)*

- recall *setjmp(3)* / *longjmp(3)*
- When signal caught, it's now added to signal mask and restored when handler returns.
- But if handler calls *longjmp(2)*, signal effectively blocked until?
- *siglongjmp(3)* has an argument to allow user control of whether or not to save/restore the signal mask.

sigsuspend(2)

Earlier problem:

- If signal is delivered after *sigprocmask(2)* and before a *pause(2)*, the *pause(2)* may wait forever.
- *sigsuspend(2)* temporarily sets the signal mask to its argument and then suspends (like *pause(2)*) until a signal is received that is either handled or causes termination.
- Hence, *sigsuspend(2)* combines both a reset of the mask with a *pause(2)* atomically.

abort(3)

- Actually sends `SIGABRT` to the process.
- If `SIGABRT` has handler, it is invoked but never returns to the process.

Interrupts and *system(3)*

- POSIX.2 says *system()* should ignore `SIGINT` and `SIGQUIT` and block `SIGCHLD`.
- The latter prevents signals about grandchildren being sent to the calling process.
- Instead, they are handled in the child shell.

alarm(2)

- after given number of seconds, it sends `SIGALRM` to the process that called it (equivalent to `raise(SIGALRM)`
- cancels any previous *alarm()* call
- `alarm(0)` disables alarms
- typical application: implement a timed read
- demos:
  (Run the `demos/d`*n*`_timed_read_1st_try` demo.) (S & R, Figure 10.10)
  (Run the `demos/d`*n*`_timed_read_2nd_try` demo.) (S & R, Figure 10.11)

pause(2)

- wait for signal to be caught
- resumes when handler exits
- do not confuse with *wait(2)*

sleep(3)

- suspends the calling process for an `int` number of seconds
- usually implemented with `SIGALRM`
  - (therefore) do not use with *alarm(3)*
- more precise timing: *usleep(3)*

Job-Control Signals

These are

- `SIGCHLD`
  child process change of status
- `SIGCONT`
  continue process if stopped
- `SIGSTOP`
  stop process (can't catch, can't ignore)
- `SIGTSTP`
  interactive stop (Ẑ)
- `SIGTTIN`
  background process reads from controlling terminal
- `SIGTTOU`
  background process writes to controlling terminal