Unix System Programming - Chapter 8

Neal Nelson

The Evergreen State College

Apr 2010

USP Chapter 8 - Signals

- Section 8.1 - Basic Signal Concepts
- Section 8.2 - Generating Signals
- Section 8.3 - Signal Masks and Signal Sets
- Section 8.4 - Catching and Ignoring Signals - sigaction
- Section 8.5 - Waiting for Signals - pause, sigsuspend, sigwait
- Section 8.6 - Async-safety, Errors, and Restart

Basic Signal Concepts

- A *signal* is a software notification of an event to a process
- A signal is a kind of software generated interrupt, asynchronous to the thread of execution of the receiving process
- A signal is *generated* when the event that causes the signal occurs.
- A signal is *delivered* when the process takes action based on that signal
- A signal is *pending* between generation and delivery

Basic Signal Concepts

- A signal's *lifetime* is the time it is *pending* between generation and delivery
- Signals can be set as *blocked* and then later *unblocked*, so signal lifetimes can be long
- Blocked signals are not the same as ignored signals; blocking is not a signal action
- A process must be running on a processor at the time of signal delivery
- A process can ultimately take one of three kinds actions for most signals
  - Ignore (throw the signal away)
  - Catch (execute a signal handler installed by the user)
  - Default (usually this kills the receiving process)

Signal Actions

- POSIX signals are listed by name in USP p257 Table 8.1
- A process *catches* a signal if it executes a *signal handler* when a signal is delivered
- A program installs a signal handler by calling `sigaction` with the name of the user-written signal handler function
- The `sigaction` function may also be called with SIG_IGN or SIG_DFL to ignore a signal or re-install the default signal action.
- The default signal action is usually to kill the process
- An ignore signal action throws the signal away when delivered and has no effect on the process.
- SIGKILL and SIGSTOP cannot be caught or ignored (or blocked, see below).
- The old way of installing a signal handler using `signal` should not be used because it is unreliable (USP p273).

Generating Signals from the Shell Command Line

- Signals are generated using the kill command from the shell or the kill system call from a program
- First let's look at the command line kill
- The kill command takes a process id and a signal name as arguments
- The command line shell command `kill -l` lists the signals on your system
- Certain keystrokes cause specific signals to be sent from the keyboard to whatever process is in control of the keyboard
- The shell command `stty -a` can be used to see what keystrokes send what signals (eg, cntl-C usually sends SIGKILL to terminate a process)

Generating Signals with the `kill` system call

- A kill system call takes a process id and a signal name
- #include <signal.h>
- Usually obtain process id for kill via `getpid` or `getppid` or `getgpid` or from a pid saved from a `fork`.
- A 0 process id in the kill system call sends the signal to members of the caller's process group.
- a -1 process id in the kill system call sends the signal to all processes that the sender has permission to kill.
- a negative process id in the kill system call sends the signal to all processes in the process group with group id equal to the absolute value of the negative process id.

Generating a signal with `raise` system call

- The `raise` system call sends a signal to self process.
- Of course the raise only needs the signal name as a parameter.
- Well, `raise(sig)` is just a shortcut for `kill(getpid(), sig)`.
- #include <signal.h>
- Maybe you use this as part of a design for exception handling using signals. So you'd use this to raise an exception.
- Using signals for exception handling would only work in an environment that was not multi-threaded.

Generating Alarm Signals

- The `alarm` system call causes a SIGALRM to be sent to the calling process after a specified number of seconds has elapsed
- A call to `alarm` before an alarm signal has been generated causes the alarm time to be reset to a new value
- A call to `alarm` with a 0 seconds of time cancels the previous alarm request
- The default signal action for an alarm is to kill the process.
- #include <unistd.h>

Blocking Signals and Signal Masks

- Every process has a *signal mask* that refines the specific signal action upon delivery
- A signal mask specifies a set of currently *blocked signals*; that's the refinement
- Blocked signals are not thrown away like ignored signals, but are instead delivered when the process unblocks the signal
- A process blocks or unblocks a collections of signals by changing its signal mask using `sigprocmask` function call

Signal Sets

- Signal sets are used when setting the signal mask to establish what signals are blocked or unblocked
- Signal sets are an abstract set type used in designating sets of signals
- Signal sets have type sigset_t and are used as parameters in the `sigprocmask` function for setting the signal mask
- Signal sets have are manipulated by 5 set operations: `sigaddset`, `sigdelset`, `sigemptyset`, `sigfillset` `sigismember`
- #include <signal.h>

Setting the Signal Mask - `sigprocmask`

- Use `sigprocmask` to modify the signal mask
- #include <signal.h>
- First parameter is how to modify the mask
  - SIG_BLOCK - add a collection of signals to those currently blocked
  - SIG_UNBLOCK - delete a collection of signals from those currently blocked
  - SIG_SETMASK - set the collection of signals being blocked to the specified set
- The second parameter is the set of signals to which the modifications apply
- The third parameter is the signal set before modifications
- If the third parameter is NULL, then no modifications are made and just the oldmask is returned in the third parameter.

Setting the Signal Mask - `sigprocmask`

- *Only use sigprocmask in a single thread*. Use `pthread_sigmask` in multi-threaded code.
- **General Rule** - Use signals only in a single-threaded environment or application.
- Some signals, such as SIGSTOP, SIGKILL cannot be blocked, caught, or ignored. Blocking requests are ignored without notification for these signals.
- Blocking and unblocking signals to solve timing problems seems fraught with unseen problems to me - either problems created or problems to prevent. See USP exercises 8.11 to 8.13.
- **Consider:** - Move to pthreads if you are really building a multi-threaded solution to a problem and ditch signals altogether.

Catching and Ignoring signals - `sigaction`

- `sigaction` is used to establish a signal handler action
  - ignore signals
  - set the default signal handler
  - explicitly associate a user-defined signal handler function with the signal.
- #include <signal.h>
- `sigaction` takes three parameters; the first is the signal name for this new action
- The second parameter is a pointer to a `struct sigaction` that specifies the action to be taken. If this parameter is NULL, no change is made to the current action for the signal
- The third parameter is a pointer to a struct sigaction that receives the previous actions associated with the signal. If this parameter is NULL, no previous action is returned.
- Consult the man page for `sigaction` for details and purpose of the various sigaction fields the `struct sigaction`
- Study examples 8.15 to 8.20 as a guide to how to set the fields of the `struct sigaction`

Setting up the `struct sigaction`

- the `struct sigaction` is complicated because it is set up to work in two different modes
- A legacy mode uses the `sa_handler` field in which to pass your sighandler function
- A newer, more flexible mode uses the `sa_sigaction` field to set up a handler that includes an extra parameter used to pass in information to the sighandler function.
- the newer mode using `sa_sigaction` comes with the POSIX RTS real time and POSIX AIO asynchronous IO extensions.
- Use only one of the two signal handler function fields

Setting up the `struct sigaction`

- This USP Chapter 8 discusses only the legacy `sa_handler` field
- The `sa_flags` field is cleared for this mode.
- USP Chapter 9.4 discusses the real time `sa_sigaction` mode and the `sa_flags` field is set for this case.
- the `int` argument to both forms of the signal handler function is used to pass the number of the signal that was delivered so that a single signal handler can handle multiple types of signals. USP says this parameter is not usually used in signal handlers.

- Done for now. Here's what's left
- Section 8.4 - Sigaction Examples
- Section 8.5 - Waiting for Signals - pause, sigsuspend, sigwait
- Section 8.6 - Async-safety, Errors, and Restart

Sigaction Examples

- All examples use just the legacy `sa_handler`
- Example 8.16 p269 sets a signal handler
- You must use async-signal safe functions (later in Section 8.6)
- Try this example

Sigaction Examples - `sig_atomic_t` and volatile qualifier

- doneflag is a signal-thread shared variable
- Program 8.5 p270 uses a sig_atomic_t for mutual exclusion
- Notice the volatile qualifier - keeps the compiler from optimizing away the need for this.
- Volatile qualifier tells compiler the variable may be modified asynchronous to this program. Handy!

Sigaction Examples - shared string buffer

- Program 8.6 pp271-272 shares an output string buffer
- Uses write for async-sig safety in signal handler p271
- Saves errno across write call
- Blocks signals when accessing signal buffer (SIG_BLOCK) p272 top half

Waiting for Signals - pause

- Signals can be used to avoid a busy wait for an event
- pause suspends the calling thread until delivery of a some signal
- Programmer has to determine whether the pause was released by the desired signal by communicating with the handler
- The pause design in early Unix has a major synchronization problem
- Example 8.21, 8.22 illustrate the problem
- A signal unblock and a pause must be together atomic to solve the problem and apparently there is not reliable way to do that.

Waiting for Signals - `sigsuspend`

- `sigsuspend` allows block and pause to be together atomic
- Examples 8.24, 8.25, 8.26 pp275-277 show how to use `sigsuspend` properly
- These examples require study to see how they work to supply a correct solution.
- Program 8.7, 8.8 pp278-279 encapsulate the correct use of suspend to provide a simple way to wait for a signal
- Program 8.9, 8.10 pp 280-281 give another encapsulation to correctly use a two-signal control for turning on or off a service.

Waiting for Signals - `sigwait`

- `sigwait` blocks until any of a user-supplied set `sigmask` of signals is pending and then removes that signal from the set of pending signals and unblocks.
- All signals in `sigmask` should be blocked before calling `sigwait`
- It is simple to use because no signal handler is really necessary.
- In my Linux system the function is `sigwaitinfo` that waits for a queue of pending signals and has a handy information parameter.

Cautions about signals interacting with function calls

- Three difficulties to consider
  - Should functions interrupted by signals be restarted (does the restart library function need to be used)
  - What if signal handlers call non-reentrant functions?
  - How do you handle `errno`?
- Two kinds of system calls to consider
  - Slow system calls like terminal IO that can block for a long time and can be interrupted by signals
  - Not slow system calls that either don't block (eg, getpid) or block only for a very short time (eg, disk IO).

Slow system calls are non-async-safe functions

- Slow system calls are the ones interrupted by signals.
- The interruptible calls return -1 and set errno to EINTR
- Look in the ERRORS section of the man page for the function to see if it can return EINTR and consequently be interrupted.
- Your program must handle EINTR errors explicitly.
- The restart library is a big help here for the most common functions.
- Only old system calls have this problem - new library functions like the pthreads library never set errno to EINTR

Async-safe functions

- *Async-safe functions* are those that can be safely called from within a signal handler.
- Not async-safe can come from
  - Using global data structures in a non-reentrant way
  - Using static data structures (remember, static doesn't mean global).
  - Using malloc or free
- When in doubt, use the restart library
- Use only POSIX guaranteed async-safe functions listed in Table 8.2 p285 in signal handlers if possible
- Signal handlers should save and restore errno if they call functions that might change errno
- Carefully analyze potential interactions between signal handler that changes an external variable and other program code that accesses the variable. Block signals to prevent unwanted interactions.

Done

- Done.