Unix Signals

Godmar Back

Virginia Tech

September 13, 2022

VIRGINIA TECH

Unix Signals

- Unix Signals present a uniform mechanism that allows the kernel to inform processes of events of interest from a small predefined set (< 32)
  - Traditionally represented by their integer number, sometimes associated with some optional additional information
- These events fall into 2 groups
  1. Synchronous: caused by something the process did (aka "internally generated event")
  2. Asynchronous: not related to what the process currently does (aka "externally generated event")
- Uniform API includes provisions for programs to determine actions to be taken for signals, which include
  - terminating the process, optionally with core dump
  - ignoring the signal
  - invoking a user-defined handler
  - stopping the process (in the job control sense)
  - continuing the process
- Sensible default actions support user control and fail-stop behavior when faults occur

Signals Representing Synchronous Conditions

- **SIGILL (1)** Illegal Instruction
- **SIGABRT (1)** Program called abort()
- **SIGFPE (1)** Floating Point Exception (e.g. integer division by zero, but not usually IEEE 754 division by 0.0)
- **SIGSEGV (1)** Segmentation Fault - catch all for memory and privilege violations
- **SIGPIPE (1)** Broken Pipe - attempt to write to a closed pipe
- **SIGTTIN (2)** Terminal input - attempt to read from terminal while in background
- **SIGTTOU (2)** Terminal output - attempt to write to terminal while in background

(1) Default action: terminate the process
(2) Default action: stop the process

Selected Signals Representing Asynchronous Notifications

- SIGINT (1, 3) Interrupt: user typed Ctrl-C
- SIGQUIT (1, 3) Interrupt: user typed Ctrl-\
- SIGTERM (3) User typed `kill pid` (default)
- SIGKILL (2, 3) User typed `kill -9 pid` (urgent)
- SIGALRM (1, 3) An alarm timer went off (`alarm(2)`)
- SIGCHLD (1) A child process terminated or was stopped
- SIGTSTP (1) Terminal stop: user typed Ctrl-Z
- SIGSTOP (2) User typed `kill -STOP pid`

(1) These are sent by the kernel, e.g., terminal device driver
(2) SIGKILL and SIGSTOP cannot be caught or ignored
(3) Default action: terminate the process

How Signals Work

- First, a signal is sent (via the kernel) to a target process
  - Some signals are sent internally by the kernel (e.g. SIGALRM, SIGINT, SIGCHLD)
  - User processes can use the `kill(2)` system call to send signals to each other (subject to permission)
  - The `kill(1)` command or your shell's built-in `kill` command do just that.
  - `raise(3)` sends a signal to the current process
- This action makes the signal become "pending"
- Then (possibly some time later) the target process receives the signal and performs the action (ignore, terminate, or call handler).
- Aside: the details of how processes learn about pending signals and how they react to them are complicated, but handled by the kernel
- Here we focus on what user programmers need to observe when using signals

VIRGINIA TECH

Signals Don't Queue

- Each signal represents a bit in the target process's pending mask saying whether the signal has been sent (but not yet received)
- Thus, sending a signal that's already pending has no effect
- This applies to internally triggered signals as well: notably, multiple children that terminate while SIGCHLD is pending will result in a single delivery of SIGCHLD
- More like railway signals (on/off) than individual messages

Control Flow (asynchronous notification)

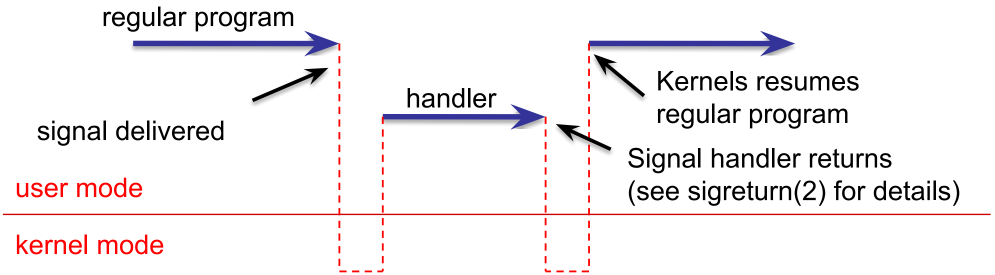

Figure 1: If a user-defined signal handler is set, it may interrupt the current program at any point. After the execution and return of the handler, the original program continues.

Control Flow Example

```c
void
list_insert (struct list_elem *before,
             struct list_elem *elem)
{
  elem->prev = before->prev;
  elem->next = before;
  before->prev->next = elem;
  before->prev = elem;
}
```

```
list_insert:
    movq    (%rdi), %rax
    movq    %rdi, 8(%rsi)
    movq    %rax, (%rsi)
    movq    %rsi, 8(%rax)
    movq    %rsi, (%rdi)
    ret
```

If a signal arrives in the middle of `list_insert()`, the manipulated list's list element are in a partially linked state. If the signal handler now takes a path where the same list is being accessed (iterated over, etc.), inconsistent behavior will result. This situation must be avoided.

Async-Signal Safety

- Is it safe to manipulate data from a signal handler while that same data is being manipulated by the program that was executing (and interrupted) when the signal was delivered?
- In general, is it safe to call a function from a signal handler while that same function was executing when the signal was delivered?
- Answer: it depends.
- POSIX defines a list of functions for which it is safe, so-called async-signal-safe functions, see signal-safety(7) for a list and the book's Web Aside: Async-signal Safety
- `printf()` is not async-signal-safe (acquires the console lock)
- Two strategies to write async-signal-safe programs:
  1. don't call async-signal-unsafe function in a signal handler
  2. block signals while calling unsafe functions in the main control flow (or when manipulating shared data)

VIRGINIA TECH

Blocking/Masking Signals

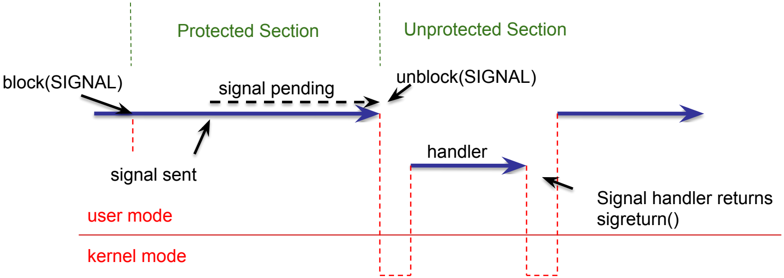

Figure 2: Programs can block signals to prevent their delivery during inopportune times. Blocked signals that become pending will be delivered when unblocked.

Trade-Off

If signals are masked/blocked most of the time in the main program, signal handlers can call most functions, but signal delivery may be delayed. If a signal is not masked most of the time, signal handlers must be very carefully implemented. In practice, coarse-grained solutions are perfectly acceptable unless there is a requirement that bounds the maximum allowed latency in which to react to a signal. Side note: OS face the same trade-off when implementing (hardware) interrupt handlers.