Introduction to POSIX Signals

Michael Jantz
Prasad Kulkarni
Douglas Niehaus

Edited By: Ishrak Hayet

Introduction

- This lab is an introduction to signals in Unix systems.
  - In it, you will learn about some common uses for signals.
  - You will also construct a small program that uses signals.

Signals

- A *signal* is a short message that may be sent to a process or a group of processes.
- The only information given to a process is usually the number identifying the signal; there is no room in standard signals for arguments, a message or other accompanying information.
- Signals serve two main purposes:
  - To make a process aware that a specific event has occurred.
  - To cause a process to execute a *signal handler* function included in its code.

Interrupts vs. Signals

- Signals and interrupts are very similar in their behavior
- Important *difference*: interrupts are sent to the operating system by the hardware, signals are sent to the process by the operating system, or other processes through the OS
- Important *similarity*: both signals and interrupts associate handlers with asynchronous events which interrupt current processing, thus inserting the handler into current code path
- Signals can thus be thought of as an interrupt in software:
  - However, note that signals have nothing to do with Soft-IRQs. The name seems related, but these are a method for deferring much of the processing associated with a hardware-interrupt into a less restrictive execution context inside the OS.

Signal Disposition

- Each signal that can be delivered in Linux has a current disposition, which determines how the process behaves when it receives the signal

- A process can change the disposition of a signal using various system calls. Using these system calls, a process can elect one of the following behaviors to occur on delivery of the signal:
  - **mask** the signal or
  - **catch** the signal:
    - **handle** it with either the default or user-defined signal handler (a function).
    - **Ignore** the caught signal

Signal Disposition

- Signals have standard *names* under POSIX, but signal *numbers* can be different across platforms
  - See Signal (7) manual page and /usr/include/bits/signum.h
  - bash> kill –l

- The entries in the “Action” column of the table on the following slide specify the default disposition for each signal as follows:

Term	Default action is to terminate the process.
Ign	Default action is to ignore the signal.
Core	Default action is to terminate the process and dump core (see core(5)).
Stop	Default action is to stop the process.
Cont	Default action is to continue the process if it is currently stopped.

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard (Ctl-C)
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User- defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty (Ctl-Z)
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

Using Signals with the Shell

- Sometimes a user may start a long job and then want to do something else
  - bash> find /usr -print
  - OOPS! No & at the end for background execution
  - Ctl-Z sends SIGTSTP
    - Returns control to command line “suspending” child process, which depends on context a bit
  - While Ctl-C sends SIGINT, generally terminating

- Signals can be used in other ways as well

Multiple Jobs in One Shell

- You can stop the current process (without losing what you've already done) by issuing the SIGTSTP signal. This is done with Ctl-Z:

```
bash> find /usr -print
^Z
[1]+  Stopped                 find /usr -print
bash>
```

- Before window based systems were common programmers did more control of multiple jobs from the command line. For example you could start another job from the same command line:

```
bash> find /lib -print
```

- And stop this process (Ctl-Z): (act fast before this one finishes)

```
[2]+  Stopped                 find /lib -print
```

- We now are managing a shell session with multiple jobs. Type jobs:

```
bash> jobs
[1]-  Stopped                 find /usr -print
[2]+  Stopped                 find /lib -print
```

Foreground and Background

- Note that the jobs are given numbers. To bring the one numbered [1] back to the foreground do:

```
bash> fg %1
```

  - **fg %n** brings the nth process (as listed when you type jobs) to the foreground.
  - **fg** with no arguments will bring the current process (the one with the + next to it when you type **jobs**), to the foreground.

- Since you brought the **find** to the foreground you see its output continue from where it was when you stopped it

- You can bring the **make** process to the foreground if you like

- **bg** takes job numbers as arguments as does **fg**, except that it sends the selected job to the background, as if you had started it with & at the end of the command line, as opposed to the foreground

Killing Jobs

- Now, let's finish this example up. You can kill jobs using the kill command:

```
bash> kill %1
[1]-  Stopped              find /usr -print
bash>  jobs
[1]-  Terminated           find /usr -print
[2]+  Stopped              find /lib  -print
```

- By default, kill delivers a SIGINT signal, the same as Ctl-C, to the job you specify
  - You can also use the PID of a process instead of %n, see **ps** command

- Some commands do not wish to be terminated by Ctl-C, and to prevent this they can *catch* the SIGINT signal

- However, it is obviously prudent to be able to have a **SIGKILL** that cannot be caught

```
bash> kill -KILL <pid>
```

Signal Handlers

- How can a process ignore the SIGINT signal?
- For most signals, user processes are allowed to define signal handlers to override the signal's default disposition.
- The SIGKILL and SIGSTOP signals are the exceptions, as they are caught by the operating system
  - This is why they cannot be handled or masked by user
- In this lab, you are going to see how to set up signal handlers and thus be able to choose how your programs will respond to signals if they are sent

signals.c

- This file contains two functions you will use as signal handlers:
  - **catch_int** keeps a count of ctrl-c
  - **catch_pipe** catches SIGPIPE signal and exits after printing a message
- The counter used to keep track of ctrl-c presses is “ctrl_c_counter”
- You should keep track of the “ctrl_c_counter”
- If the “ctrl_c_counter” exceeds the threshold, then prompt the user whether the user wants to close the pipe p’s read end p[0] or not? [Y/n?]
- If no, the program will continue with “ctrl_c_counter” reset to 0
- If yes, pipe p’s read end p[0] should be closed
- During the prompt, if the user doesn’t respond within 3 seconds, the program should exit (use SIGALRM – you should implement the handler for SIGALRM)

Overriding Signal Disposition

We will use the following functions to set a custom disposition (including custom handler) of a signal

sigfillset
sigdelset
sigaction (system call)

Letting the program run to catch a signal

A program should be in execution in order to catch a signal

We will use the following system call inside a loop to keep the program running and receptive to signals

Declaration: int pause(void)
Usage: pause()

Sigaction and Sigaction

struct sigaction

A structure defining a signal’s disposition

```
struct sigaction {
        void     (*sa_handler)(int);
        void     (*sa_sigaction)(int, siginfo_t *, void *);
        sigset_t   sa_mask;
        int        sa_flags;
        void     (*sa_restorer)(void);
};
```

function sigaction

A system call that assigns a disposition (struct sigaction reference) to a signal

```
int sigaction (int signum,
        const struct sigaction *act,
        struct sigaction *old_act)
```

struct sigaction

- The **struct sigaction** has the following structure:

```
struct sigaction {
    void      (*sa_handler)(int);
    void      (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
    void      (*sa_restorer)(void);
};
```

- **sa_handler** is a pointer to the handler function that will be called when the signal is received. Use this when the handler does not need additional information other than the signal number. You will use this in this lab to store a pointer to each handler function.

- **sa_sigaction** is also a pointer to a handler function. Use this when your handler needs more information than just a signal number; see **sigaction** man page for more details. You will not need this for this lab.

- **sa_mask** is the set of signals you wish to mask (i.e. block) during execution of the signal handler. You will need to use this for this lab.

- **sa_flags** allows you to set various options when handling the signal. We will not change the default flags for this lab.

- **sa_restorer** is obsolete. You will not need this for this lab.

function sigaction

- **int sigaction (int signum, const struct sigaction *act, struct sigaction *old_act)**
  - Assigns a handler for a signal according to contents of a **struct sigaction**
  - **signum** is the number of the signal for which you would like to assign a handler; SIGINT, SIGPIPE, and SIGALRM are used for this lab
  - **act** is a pointer to the **struct sigaction** specifying how the signal should be handled - see the following slide for a more in-depth explanation
  - **old_act** is a pointer to the **struct sigaction** that was associated with this signal before this call. For this lab, we don't care about this information. Simply pass in NULL to ignore it.

sigfillset

- We will use the following function to create a signal mask set:
  - **Declaration:** int sigfillset(sigset_t *set)
  - **Usage:** sigset_t sa_mask;

    sigfillset(&sa_mask);
- We will assign this set to the sa_mask attribute of the struct sigaction variable, the reference of which we will pass to the function sigaction
- **sigfillset** initializes the set pointed to by **set** to be full
  - i.e. it includes all signals
- Passing this signal masking set to the custom disposition ensures that signals from this set are blocked while inside the handler of this disposition

Adding a Timeout

- After 5 consecutive ctrl-c presses, the user is prompted whether to close pipe p’s read p[0] or not?

- Sometimes, the user might forget to reply
  - Think of how logging out of the machines here at the lab works

- In this situation, it might make sense to add a timeout which performs the exit if there is no response from the user for some time.

- As the final part of this lab, we will add a timeout to exit our program if the user forgets to type a response when prompted.

Alarm

- The alarm system call is a convenient way to implement timeouts.
  - **unsigned int alarm(unsigned int <u>seconds</u>)**
  - **alarm( )** arranges for a SIGALRM signal to be delivered to the calling process in **<u>seconds</u>** seconds.
- You should initiate an alarm to go off after so many seconds after the user has been prompted to exit
- You will have to unmask the SIGALRM signal when initializing the SIGINT signal handler (so SIGALRM will be handled when the timeout occurs). Use sigdelset to remove SIGALRM from the masked set (in slide 18):
  - **Declaration:** int sigdelset(sigset_t *set, int signum)
  - **Usage:** sigdelset(&sa_mask, SIGALRM)
- You should define the SIGALRM custom signal handler that will exit the program if the user doesn’t respond within 3 seconds. Otherwise, it will continue

Quitting inside the Alarm Handler

- Inside the alarm handler function, you will have to quit the program if the user has not responded
- You can quit using the exit(-1) function call inside the alarm handler
- **However, if the user responds during the prompt you should ensure that the program continues**

Final Output

SIGINT and SIGPIPE

```
bash> ./signals
^C^C^C^C^C
Close pipe’s read end? [Y/n]: n

Pipe p’s read end is still open. Continuing

Writing to the pipe p’s write end

^C^C^C^C^C
Close pipe’s read end? [Y/n]: y

Closing pipe p’s read end p[0] …

Writing to the pipe p’s write end

SIGPIPE received – exiting…
```

SIGALRM

```
bash> ./signals
^C^C^C^C^C
Close pipe’s read end? [Y/n]:

No response from the user! Exiting …
```