CSCI 4061: Signals and Signal Handlers

Chris Kauffman

Last Updated:
Tue Oct 17 12:50:40 CDT 2017
Logistics

Reading
- Robbins and Robbins
  Ch 8.1-8.7, 9.1-2
- OR Stevens/Rago
  Ch 10

Goals
- Sending Signals in C
- Signal Handlers

Exam 1 Scores Posted
- Exams returned Monday
- Bulk stats on Piazza

Lab06: kill
How did it go?

Project 2
- Under development
- Will discuss on Thu
Exercise: Lab06 kill

1. What is a signal?
2. What system call is used to send a process a signal? How is it invoked?
3. What’s a simple way set up simple signal handling?
4. Which signals cannot be caught and handled?
5. What effects do these uncatchable signals have?
Answers: Lab06 kill

1. What is a signal?
   - Notification from somewhere, limited information, special effects
2. What system call is used to send a process a signal? How is it invoked?
   - `kill(pid, SIGSOMTHING);`
3. What's a simple way set up simple signal handling?
   - Use the `signal()` function as in
     ```c
     signal(SIGINT, handle_SIGINT);
     ```
     where `handle_SIGINT()` is a function taking an int
4. Which signals cannot be caught and handled? What effects to these uncatchable signals have?
   - SIGKILL terminates a process
   - SIGSTOP stops a process from running; it can be restarted with a SIGCONT
Process Signal Disposition

Every process has a default signal disposition for each signal. These can be adjusted with various system calls.

Signal dispositions
Each signal has a current disposition, which determines how the process behaves when it is delivered the signal.

The entries in the "Action" column of the tables below 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.
Ignoring Signals, Restoring Defaults

- Setting the signal handler to `SIG_IGN` will cause signals to be silently ignored.
- Setting the signal handler to `SIG_DFL` will restore default disposition.

Demo `no-interruptions-ignore.c`
Historical Notes

- Signals were an early concept but were initially "unreliable": might get lost and so were not as useful as their modern incarnation.
- Historically, required to reset signal handlers after they were called. First line of handler was always `signal(this_signal, this_handler);` though this was still buggy.
- Historically, some system calls could be interrupted by signals. Robbins & Robbins go on and on about this.

  On FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8, when signal handlers are installed with the `signal` function, interrupted system calls will be restarted. The default on Solaris 10, however, is to return an error (EINTR) instead when system calls are interrupted by signal handlers installed with the `signal` function.

  – Stevens and Rago, 10.5
Dangers in Signal Handlers

- General advice: do as little as possible in a signal handler
- Make use of only **reentrant** functions
  
  ... *reentrant if it can be interrupted in the middle of its execution, and then be safely called again ("re-entered") before its previous invocations complete execution.*
  
  – Wikipedia: Reentrancy

- Notably not reentrant
  
  printf() family
  malloc()
  free()

- Reentrant functions pertinent to thread-based programming as well (later)

- Demo non-reentrant.c
Portability Notes

- Portability of `signal()` to set up handlers is questionable:

  PORTABILITY
  The semantics when using `signal()` to establish a signal handler vary across systems (and POSIX.1 explicitly permits this variation); do not use it for this purpose.
  - `man 2 signal`

Portable Signal Functions

- `signal()` is an old function with many different implementation behaviors
- POSIX defined new functions which were designed to break from its tradition and fix problems associated with it
- Requires introduction of `signal sets`, data type for a set of signals along with associated functions
Signal Sets

- A set of signals, likely implemented as a bit vector
- Functions allow addition, removal, clearing of set and tests for membership

```c
#include <signal.h>

int sigemptyset(sigset_t *set);
// empty out the set

int sigfillset(sigset_t *set);
// fill the entire set with all signals

int sigaddset(sigset_t *set, int signo);
// add given signal to the set

int sigdelset(sigset_t *set, int signo);
// remove given signal to the set

// All of the above return 0 on success, -1 on error

int sigismember(const sigset_t *set, int signo);
// return 1 if signal is a member of set, 0 if not

Examine sigsets-demo.c
```
Blocking (Disabling) Signals

- Processes can disable receiving signals, called **blocking** the signal
- Signal is still there, just awaiting delivery
- This is different from ignoring the signal: blocked signals will still be delivered after unblocking
- Ignored signals are gone for good
- Important to protect critical sections of code

Process Signal Mask
Example: block all signals that can be blocked

```c
sigset_t block_all, defaults;
sigfillset(&block_all); // contains all
sigprocmask(SIG_SETMASK, &block_all, &defaults); // block all signals
// save defaults
```

Examine `no-interruptions-block.c`
Exercise: Protect Non-Reentrant

Examine the code for non-reentrant.c and modify it to use signal blocking to protect the critical region associated with calls to getpwnam().

- Create a mask for all signals
- Block all signals prior to function call
- Unblock after returning
- Use code like below

```c
sigset_t block_all, defaults;
sigfillset( &block_all ); // contains all
sigprocmask(SIG_SETMASK, &block_all, &defaults); // block all signals
// save defaults
```
Portable Signal Functions: sigaction()

```c
struct sigaction {
    void    (*sa_handler)(int);   /* addr of signal handler, */
                                      /* or SIG_IGN, or SIG_DFL */
    sigset_t sa_mask;              /* additional signals to block */
    int     sa_flags;              /* signal options, Figure 10.16 */
    /* alternate handler */
    void    (*sa_sigaction)(int, siginfo_t *, void *);
};
```
Ignoring Signals