CSCI 4061: Signals and Signal Handlers

Chris Kauffman

Last Updated:
Thu Oct 19 12:16:40 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
- `select()`: Multiplexing I/O

Exam 1 Scores Posted

- Exams returned Monday
- Bulk stats on Piazza

Lab07: select(), signals

Covers `select()` system call and signals for control flow

Project 2

- Under development
- Will discuss on Tue

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 to 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

     `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_hanlder);
  ```

  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`

Exercise: Non-Reentrant Function Example

- Program calls non-reentrant function `f()` in `main()` and `handle_signal()`
- With no interrupts, would expect to see 7 printed, with interrupts see `19,7` in either order
- Show a control flow involving signals that prints 19 twice
- Why is f() not reentrant?

```c
int z;
int f(int x, int y){
  int tmp = x + y;
  z = tmp * 2 + 1;
  return z;
}

void handle_signal(int sig){
  int t = f(4,5);
  printf("%d\n",t);
  return;
}

int main(){
  signal(SIGINT,handle_signal);
  int v = f(1,2);
  printf("%d\n",v);
}
```

Answer: Non-Reentrant Function Example

- Program below calls non-reentrant function `f()` in `main()` and `handle_signal()`
- With no interrupts, would expect to see 7 printed, with interrupts see 19 and 7
- Right hand shows one possible flow through the code which produces 19 then 19 again

```
 1 int z;
 2 int f(int x, int y){
 3   int tmp = x + y;
 4   z = tmp * 2 + 1;
 5   return z;
 6 }
 7
 8 void handle_signal(int sig){
 9   int t = f(4,5);
10   printf("%d\n",t);
11   return;
12 }
13
14 int main(){
15   signal(SIGINT,handle_signal);
16   int v = f(1,2);
17   printf("%d\n",v);
18 }
```

```
EXECUTION STARTS IN main()
15: signal(SIGINT,handle_signal);
16: int v = f(1,2); // main(), Expect: (1+2)*2+1 = 7
 3: tmp = x + y;    // f(1,2): tmp = 1+2 = 3
 4: z = tmp*2 + 1;  // z is 7
SIGINT delivered, run handler
    9: int t = f(4,5);   // handle_signal(2)
    3: tmp = x + y;      // f(4,5): tmp = 4+5 = 9
    4: z = tmp*2 + 1;    // z is now 19
    5: return z;         // back to handle_signal()
    9: int t = f(4,5);   // finished, t is 19
   10: printf("%d\n",t); // puts 19 on screen
   11: return;           // back to normal control
 5: return z;         // back to main(), but z is 19
16: int v = f(1,2);    // v is Actually 19
17: printf("%d\n",v);  // 19 Actually printed
                       //  7 Expected
```

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

```
#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 succes, -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 block signals, disable receiving them
- Signal is still there, just awaiting delivery
- Blocking is different from Ignoring a signal
  - Ignored signals are received and discarded
  - Blocked signals will be delivered after unblocking
- Can protect Critical Sections of code with by blocking if signals would screw it up

Process Signal Mask

Example: block all signals that can be blocked

```
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 Call

Examine the code for `non-reeentrant.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

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

Note: Be *very careful* where you unblock signal handling in `main()` to avoid errors: protect the Critical Section

Portable Signal Functions: sigaction()

- The sigaction() function is more portable than signal() to register signal handlers.
- Makes use of struct sigaction which specifies properties of signal handler registrations

Type	Field	Purpose
void(*) (int)	sa_handler	Pointer to a signal-catching function or one of the macros SIG_IGN or SIG_DFL.
sigset_t	sa_mask	Additional set of signals to be blocked during execution of signal-catching function.
int	sa_flags	Special flags to affect behavior of signal.

```
int main(){                              // SAMPLE HANDLER SETUP USING sigaction()
  struct sigaction my_sa = {};           // portable signal handling setup with sigaction()
  my_sa.sa_handler = handle_signals;     // run function handle_signals
  sigemptyset(&my_sa.sa_mask);           // don’t block any other signals during handling
  my_sa.sa_flags = SA_RESTART;           // restart system calls on signals if possible
  sigaction(SIGTERM, &my_sa, NULL);      // register SIGTERM with given action
  sigaction(SIGINT,  &my_sa, NULL);      // register SIGINT with given action
  ...;
}
```