Lab 07 - Signals

Due: November 8, 2015

1 Introduction

At some point in your computer science career you have probably terminated a malfunctioning program by pressing CTRL-C on the keyboard. The ability to do so is useful, since it allows you to recover your session, even if the currently-running process has entered an infinite loop or will not terminate in a reasonable amount of time. It also means that dangerous programs can be terminated without resorting to a samurai sword.

To provide the functionality of CTRL-C, the shell makes use of *signals* — tiny messages that indicate that some event has occurred. Signals may occur as a result of system events, such as segmentation faults or illegal instructions, and can also be sent by processes to other processes. When you type CTRL-C in a terminal, the shell sends a signal called `SIGINT` to the current foreground process group. By default, a process will immediately terminate upon receiving a `SIGINT` signal, returning control of the shell to the user. A similar sequence of events occurs when CTRL-\ or CTRL-Z are typed. CTRL-\ is used to send the `SIGQUIT` signal, which instructs a process to exit gracefully. CTRL-Z is used to send the `SIGTSTP` signal, which instructs a process to temporarily suspend its execution.

In this lab, you will learn how to override the default behavior for signals, and how to send signals of your own.

2 Signals

2.1 Catching Signals

The process of responding to a signal is called *catching* the signal.[1] The usual way to specify the program's response to a signal is to write a function called a *signal handler* to perform the desired actions. The signal handler must then be *installed* (tied to a particular signal) so that it will be called when the signal is received. Each signal can have at most one signal handler, but a single signal handler may be used for multiple signals.

To install a signal handler in C, you need to use the `sigaction()` function, which has the following signature:

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

The first argument specifies the signal id. Constants representing different signals are defined in `<signal.h>`. The ones you will be using in this lab are `SIGINT`, `SIGTSTP` and `SIGQUIT`. The second argument to `sigaction`() is a `struct sigaction` that describes the action to associate to that signal (in this case, calling a signal handler). This struct has several members that you must initialize before passing it to the function:

[1] While most signals can be caught by a process, there are two that cannot be: `SIGSTOP` and `SIGKILL`. These signals force the process to pause and terminate, respectively.

- `sa_handler`, a function pointer to your signal handler
- `sa_mask`, the signals to mask off when executing the handler
- `sa_flags`, a set of flags which modify the behavior of the signal

The last argument to `sigaction()` is a pointer in which to store the previous action (or NULL to discard the old value).

As usual, more information can be found on the appropriate man page.

2.2 Asynchronous Signal Safety

Signals may either be *synchronous* or *asynchronous*. Synchronous signals pertain to specific actions in the program, like explicit requests by a process to generate a signal and most errors. Asynchronous signals are generated by events outside of the control of a process and can happen at any time. These include I/O-related signals and signals sent by other processes.

Signal handling can lead to problems if the handler interferes with what the program was doing when the signal occurred. This is especially problematic for asynchronous signals, because you cannot control when they may occur. For instance, if a signal occurs in the middle of a buffered I/O function like `printf()`, and the signal handler also calls a buffered I/O function, the buffer may be corrupted or left in an inconsistent state, leading to undesirable behavior. Functions which are immune to this type of problem are called *async-signal-safe*. A list of async-signal-safe functions can be found on the signal (7) man page. All other syscalls and library functions should be considered unsafe with respect to signals. In addition, modifying a global variable of type other than `volatile sig_atomic_t` should be considered an unsafe operation.[2]

A more thorough understanding of concurrency issues is required to fully ensure asynchronous signal safety. For this lab, the following safeguards should be sufficient:

- When installing the signal handlers, include all signals you will be catching in the `struct sigaction`'s `sa_mask` field. This will prevent signal handlers from interrupting each other.
- If you call any async-signal-unsafe functions in your main code after installing the handlers, mask off all relevant signals during the unsafe call. You should use the `sigprocmask()` function to accomplish this. (See the `main()` function in the *siglab.c* stencil for an example.)

2.3 Signals and Blocking Functions

Some blocking syscalls and library functions are affected by signals. The signal(7) man page states the following:

> If a signal handler is invoked while a system call or library function call is blocked, then either:

[2] `sig_atomic_t` specifies an integer type that can be read or written with a single instruction, so a signal cannot occur "in the middle" of an access. The `volatile` modifier indicates to the compiler that the value of the variable may be changed by things beyond a given section of code. This prevents the compiler from optimizing out operations that may locally appear to have no effect, and instructs it to read the value from memory for each use, rather than caching it in a register.

- the call is automatically restarted after the signal handler returns; or
- the call fails with the error `EINTR`.

Different functions exhibit different behavior when interrupted. You can also modify this behavior when installing a signal handler by including the `SA_RESTART` flag in the `sa_flags` field of the `struct sigaction` passed to `sigaction()`. More detailed information can be found on the signal(7) man page.

3 Assignment

This lab has two parts: first you will implement a short program that catches signals to override the default behavior; then you will implement another program that sends signals to any process.

To install the stencil for this assignment, run

```
cs033_install lab07
```

3.1 Part 1: Catching Signals

In the first part of the lab, you will write a program that catches and handles `SIGINT`, `SIGTSTP`, and `SIGQUIT`. Stencil for this part is in *siglab.c*.

3.1.1 Signal Handlers

The first step in this lab is to write signal handlers that deal with the `SIGINT`, `SIGQUIT` and `SIGTSTP` signals. These are functions that take an integer argument (representing the signal) and have return type `void`. Each handler should print out a message to the user to indicate the signal was caught. Each signal should produce a different message, for the sake of testing.

3.1.2 Installing Handlers

After writing the signal handlers, you must install them. This is normally done with the `sigaction()` function, which has the following signature:

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

In this lab, you will implement the following function to simplify the process of installing a handler:

```
int install_handler(int signum, void (*handler)(int))
```

This function takes a signal number and a pointer to a signal handler function and installs the handler for that signal, returning 0 on success and -1 on failure. It should set up a `struct sigaction` and use the `sigaction()` function to install the given handler.

3.1.3 Additional Functionality

In addition to handling signals, the program should read input from stdin and echo it to stdout. This should be done in an asynchronous-signal-safe way, using the `read()` and `write()` syscalls. The program should terminate on EOF (when `read` returns 0). This functionality will be implemented in the `read_and_echo()` function, which returns an `int` indicating success or failure when done.

3.1.4 Terminating Your Program

If you make a mistake, you might produce a program that is difficult to terminate. If you find yourself in such a situation, you can kill the process from another terminal using the command

```
pkill -SIGKILL siglab
```

This will send the uncatchable `SIGKILL` signal to all processes named "`siglab`", terminating them immediately.

3.2 Part 2: Sending and Responding to Signals

Once you have signal handlers installed, you can no longer easily terminate or stop your program from the terminal. Now, your job is to implement a "weakness" in the impenetrable wall of signal handlement. Specifically, you will modify `siglab` to terminate upon receipt of a particular sequence of signals, `SIGINT SIGTSTP SIGQUIT`, delivered within an appropriate timespan. If any other sequence of signals is sent, your program should not react. You'll also be writing another short program, `knocker` (whose source is in *knocker.c*), to send this signal sequence. We refer to this process as "signal knocking", as it resembles the process of "port knocking" which is used to control access to protected ports on a firewall.

3.2.1 Implementing Signal Knocking

Modify your signal handlers so that `siglab` terminates when the correct sequence of signals is received in the span of one or two seconds. You will probably want to use global variables for this. Note that since signals are masked off during the signal handlers, these do not have to have signal-safe data types.

You can use the `time()` function to get a count of the number of seconds since January 1, 1970. This can be used to determine whether there has been too much of a delay between signals.

3.2.2 A Simple "Signal Knocker"

Your signal knocker will take two arguments: an integer representing the PID (process ID) of the process to send the signals to, and a string of characters indicating the sequence of signals to be sent. In the second argument, the character '`c`' indicates a `SIGINT`, '`z`' a `SIGTSTP` and '`q`' a `SIGQUIT`. For example, the following should terminate your `siglab` program:

```
./knocker <PID> czq
```

3.2.3 Sending a Signal

To send a signal to another process in C, you can use the `kill()` system call, which has the signature:

```
int kill(pid_t pid, int sig);
```

This function sends signal `sig` to process `pid`. Use this function to send the correct sequence of signals to terminate your signal-handling program.

3.2.4 Getting the PID of a Process

In order to send a signal to a process, you need that process's PID. When using the `kill` command in a terminal, it's simple enough to obtain the PID for a given process — `pgrep -l <program name>` will print out a list of all processes whose names contain a match of the regular expression `<program name>`, and the corresponding PIDs. For instance, `pgrep -l foo` prints all current processes whose names contain the string "`foo`".

A C program can access the current process's PID with the system call `getpid()`, which returns the process's PID as a integer of type `pid_t`. We recommend adding a line at the beginning of your signal-handling program to print out its PID, making it easier to set up your signal knocker.

4 Getting Checked Off

Once you've finished both parts of the lab, call a TA over and have them inspect your work. If you do not complete the lab during lab hours, you may get credit for finishing it on your own time and bringing it to **lab hours** to be checked off before next week's lab. The last lab hours to get this lab checked-off on time are Sunday, November 8th from 2-4pm, so please plan your work accordingly. Remember to read the course missive for information about course requirements and policies regarding labs and assignments.