Signal Lab

Attribution

This is a modification of an assignment from Ron Zarcharski[1] Ron attributes the assignment to John Montelius[2]. The theme of the lab is the same as that created by Ron and John; however, I modified the text extensively and the code some.

Introduction

In this lab you explore signals in a Linux environment by compiling and executing sample code. Signals are an asynchronous event sent to a process. Signals are similar to interrupts, which are asynchronous hardware events. The OS can signal a process, and one process sends a signal to another. Sometimes a hardware interrupt is processed by the OS and sent as a signal to a process. For example, a floating point exception (hardware interrupt) occurs when your code contains a divide by zero. Your process can catch a `SIGFPE` (Signal Floating Point Exception)

The Linux `kill` command and the library function `kill()` send signals to processes. The formats of `kill` and kill() are

```
$ kill <signal> <pid>
int status = kill(pid, signal);
```

`man kill` and `kill -l` lists all possible signals. This lab uses a few.

```
No    Name        Default Action       Description
1     SIGHUP      terminate process    terminal line hangup
2     SIGINT      terminate process    interrupt program
3     SIGQUIT     create core image    quit program
4     SIGILL      create core image    illegal instruction
5     SIGTRAP     create core image    trace trap
6     SIGABRT     create core image    abort program (formerly SIGIOT)
7     SIGEMT      create core image    emulate instruction executed
8     SIGFPE      create core image    floating-point exception
9     SIGKILL     terminate process    kill program
10    SIGBUS      create core image    bus error
11    SIGSEGV     create core image    segmentation violation
12    SIGSYS      create core image    non-existent system call invoked
13    SIGPIPE     terminate process    write on a pipe with no reader
14    SIGALRM     terminate process    real-time timer expired
15    SIGTERM     terminate process    software termination signal
16    SIGURG      discard signal       urgent condition present on socket
17    SIGSTOP     stop process         stop (cannot be caught or ignored)
```

[1] https://github.com/zacharski/cpsc405/blob/master/signalLab.md
[2] https://people.kth.se/%7Ejohanmon/ose/assignments/signals.pdf

```
18    SIGTSTP      stop process         stop signal generated from keyboard
19    SIGCONT      discard signal       continue after stop
20    SIGCHLD      discard signal       child status has changed
21    SIGTTIN      stop process         background read attempted from control terminal
22    SIGTTOU      stop process         background write attempted to control terminal
23    SIGIO        discard signal       I/O is possible on a descriptor (see fcntl(2))
24    SIGXCPU      terminate process    cpu time limit exceeded (see setrlimit(2))
25    SIGXFSZ      terminate process    file size limit exceeded (see setrlimit(2))
26    SIGVTALRM    terminate process    virtual time alarm (see setitimer(2))
27    SIGPROF      terminate process    profiling timer alarm (see setitimer(2))
28    SIGWINCH     discard signal       Window size change
29    SIGINFO      discard signal       status request from keyboard
30    SIGUSR1      terminate process    User defined signal 1
31    SIGUSR2      terminate process    User defined signal 2
```

`SIGKILL` and `SIGTERM` both terminate a process. `kill SIGKILL` (equivalent to `kill -9`) immediately terminates a process. `kill SIGTERM` terminates a process, but the process is allowed to close files, etc. `kill` is equivalent to `kill SIGTERM`.

A program runs in the foreground of a terminal window when the command does not contain the `&` suffix. The following is an example of running a program in the foreground.
`$ ./killme_with_a_signal`

The following is an example of running a program in the background.
`$ ./killme_with_a_signal &`

Entering `ctrl-c` in a terminal window sends `SIGINT` to the foreground processes. Notice how the `ctrl-c` terminates a foreground process, but not the shell.

Catching a Signal

A process inherits the signal handlers of its creator. A process can create its own signal handlers. For example, if a program has various files open, it can create a `SIGTERM` handler to close files. The following program (`catch_signals.c`) establishes a signal handler that catches and counts `SIGINT` signals until 4 have been caught.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

int volatile count;

void handler (int sig){
      printf("Signal %d  ouch, that hurt!\n", sig);
      count++;
}

int main(){
```

```
    struct sigaction sa;
    int pid = getpid();
    printf("Ok, let's go, kill me (%d) if you can!\n", pid);
    sa.sa_handler = handler; // 1.
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask); // 2.

    if (sigaction(SIGINT, &sa, NULL) != 0){ // 3.
          return(1);
    }

    while (count != 4){
     /* do nothing */
    }
    printf("I've had enough!\n");
    return 0;
}
```

The `sigaction` `struct` is used to establish the function `handler` as a signal handler.

```
struct sigaction {
           void     (*sa_handler)(int); // handler
           void     (*sa_sigaction)(int, siginfo_t *, void *);
           sigset_t   sa_mask; // defines signals for handler
           int        sa_flags;
           void     (*sa_restorer)(void);
};
```

1. `sa.sa_handler = handler;` establishes our signal handler, which is the function called when the signal is passed to the process.
2. `sigemptyset(&sa.sa_mask)` clears the signals so they are not blocked when we're in the handler.
3. `sigaction(SIGINT, &sa, NULL)` establishes the signal handler. `sigaction` is a library call that changes the signal table such that when a `SIGINT` signal is sent the function `handler` is called.
    - `SIGINT` - the signal we want to handle.
    - `&sa` - pointer to the sigaction structure.
    - `NULL` - null pointer (that we don't have to bother about now).

Notice `handler` prints a clever message to indicate it is invoked. You should not use a system library call such as `printf` inside a handler since this might be in conflict with an ongoing library call. We do so in this exercise to keep things as simple as possible.

Experiments

Perform the following experiments.

1. Run the program in the foreground enter `ctrl-c` four times.
   ```
   $ ./catch_signals
   Ok, let's go, kill me (3950) if you can!
   ctrl-c
   ```
2. Run the program in the background and enter the `kill -2 <pid>` command four times in the current terminal window.
   ```
   $ ./catch_signals &
   Ok, let's go, kill me (3950) if you can!
   $ kill -2 3950
   ```
3. Run the program in the background and enter the `kill -2 <pid>` command four times in another terminal window.
   ```
   $ ./catch_signals &
   Another terminal window
   Ok, let's go, kill me (3950) if you can!
   $ kill -2 3950
   ```
4. Run the program in the background and use your `my_kill` program four times in another terminal window.
   ```
   $ ./catch_signals &
   Another terminal window
   Ok, let's go, kill me (3950) if you can!
   $ ./my_kill 3950
   ```
5. Look up `kill` and `sigaction` using the `man` command.

Catch and Throw - Java and C - Divide by Zero

Java defines exceptions that can be caught and thrown. The C language does not define exceptions, catch, and throw; but signals can be used. The following code catches a divide by zero exception. A divide by zero is first detected by the hardware - a divide by zero exception/interrupt is generated. The CPU’s interrupt vector table provides control to the OS interrupt handler, which is part of the kernel. By default, the OS kills a process with a divide by zero exception. The following code (FPE_signal.c) demonstrates default and registering a `SIGFPE` handler.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

void handler (int sig){
      printf("Floating point exception (signal): %d was caught!\n", sig);
      exit(1);
```

```
        return ;
}

int not_so_good(){
        int x = 0;
        return 1 % x;
}

int main(int argc, char *argv[]){
    if (argc > 1) { // catch FPE
        struct sigaction sa;
        printf("Catch a FPE.\n");
        sa.sa_handler = handler;
        sa.sa_flags = 0;
        sigemptyset(&sa.sa_mask);

        /* now catch FPE signals */
        sigaction(SIGFPE, &sa, NULL);
    }
    else {
        printf("Default action for FPE.\n");
    }
    not_so_good();
    printf("Will probably not write this.\n");
    return 0;
}
```

Experiments

Perform the following experiments.

6. Run the program with the default action.
7. Run the program with the signal handler that catches the divide by 0.

Catching a Signal and Context

A signal handler can catch a signal and information about the signal by using a flag in the `sigaction` `structure`. The following code (`killme_with_a_signal.c`) catches a `SIGINT` signal and prints information about the process that sent the signal.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

int volatile done = 0;
```

```
void handler(int sig, siginfo_t *siginfo, void *context){
      printf("signal %d was caught.\n", sig);
      printf("your UID is %d\n", siginfo->si_uid);
      printf("your PID is %d\n", siginfo->si_pid);
      done = 1;
}


int main(){
      struct sigaction sa;
      int pid = getpid();
      printf("Ok, kill me (%d) and I'll tell you who you are.\n",  pid);
      sa.sa_flags = SA_SIGINFO; // set flag for more info to handler
      sa.sa_sigaction = handler;
      sigemptyset(&sa.sa_mask);
      if(sigaction(SIGINT, &sa, NULL) != 0) {
            return (1);
      }
      while (! done) {
      }
      printf("Told you so!\n");
      return (0);
}
```

Three arguments are passed to the handler in this program: the signal number, a pointer to a `siginfo_t structure` and a pointer to a context that we can ignore for a while.
The `siginfo_t structure` contains information about the process that sent the signal.

Experiments

8. Run the program in one shell and kill it from another shell using `kill -2`. Observe which process killed you. Is the PID printed the PID of the shell issuing the kill command.
9. Run the program in one shell and kill it from another shell using your `my_kill` program. Observe which process killed you. Is the PID printed the PID of the shell issuing the kill command.

Summary

Signals are asynchronous events. The kernel uses signals to asynchronously inform processes about exceptions. If your program does not establish signal handlers, the kernel processes exceptions. A user process can register a function as a signal handler, and the kernel passes control to the function. Signals can also be used in between processes, to send notifications to, or control processes. Signals themselves contain no information, but the kernel can provide more information about who sent the signal and why.

Questions

10. Describe what a signal is in your own words.
11. Pick several specific signals and describe them.
12. Describe how to catch a signal.
13. When catching a signal, can a process determine who sent it? If so how?
14. How is a signal related to interrupts?

Submissions

1. Submit a copy/paste run log showing you ran the code to verify the experiment questions and also did some other experiments.
2. Submit answers to the questions.