CS 332/532 – 1G- Systems Programming

Lab 9

Objectives

The objective of this lab is to introduce you to signal handling in UNIX.

Description

Signals are software interrupts that provide a mechanism to deal with asynchronous events (e.g., a user pressing Control-C to interrupt a program that the shell is currently executing). A signal is a notification to a process that an event has occurred that requires the attention of the process (this typically interrupts the normal flow of execution of the interrupted process). Signals can also be used as a synchronization technique or even as a simple form of interprocess communication. Signals could be generated by hardware interrupts, the program itself, other programs, the OS kernel, or by the user.

All these signals have a unique symbolic name and starts with SIG. The standard signals (also called POSIX reliable signals) are defined in the header file *<signal.h>*. Here are some examples:

- SIGINT: Interrupt a process from keyboard (e.g., pressing Control-C). The process is terminated.
- SIGQUIT: Interrupt a process to quit from keyboard (e.g., pressing Control-/). The process is terminated, and a core file is generated.
- SIGTSTP: Interrupt a process to stop from keyboard (e.g., pressing Control-Z). The process is stopped from executing.
- SIGUSR1 and SIGUSR2: These are user-defined signals, for use in application programs.

NOTE: Please see Section 10.2 and Figure 10.1 in the textbook for a complete list of UNIX System signals. You can also find more details using *man 7 signal* on any CS UNIX system (*man signal* on Mac).

After a signal is generated, it is delivered to a process to perform some action in response to this signal. Since signals are asynchronous events, a process has to decide ahead of time how to respond when I particular signal is delivered. There are three different options possible when a signal is delivered to a process:

1. Perform the default action. Most signals have a default action associated with them, if the process does not change the default behavior, then the default action specific to that particular signal will occur. The predefined default signal is specified as SIG_DFL.

2. Ignore the signal. If a signal is ignored, then the default action is performed. The predefined ignore signal handler is specified as SIG_IGN.

3. Catch and handle the signal. It is also possible to override the default action and invoke a specific user-defined task when a signal is received. This is usually performed by invoking a signal handler using *signal()* or *sigaction()* system calls.

However, the signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.

Now, let's write a program to handle a simple signal that catches either of the two user-defined signals and prints the signal number (see Figure 10.2 in the textbook). You can download the complete example: `sigusr.c`

1. Create a user-defined function (signal handler) that will be invoked when a signal is received:

```
static void sig_usr(int signo) {
  if (signo == SIGUSR1) {
     printf("received SIGUSR1\n");
  } else if (signo == SIGUSR2) {
     printf("received SIGUSR2\n");
  } else {
     printf("received signal %d\n", signo);
  }
}
```

2. Now let's call the above user-defined signal.

```
int main(void) {
   if (signal(SIGUSR1, sig_usr) == SIG_ERR) {
     printf("can't catch SIGUSR1\n");
     exit(-1);
   }
   if (signal(SIGUSR2, sig_usr) == SIG_ERR) {
     printf("can't catch SIGUSR2\n");
     exit(-1);
   }
   for ( ; ; )
      pause();

   return 0;
```

```
}
```

Here, we use an infinite loop with the *pause* function. The pause function causes the calling process to sleep until a signal is received (see *man pause* for more details about pause). We use the printf statements in the signal handler in this example, however, printf is not a reentrant function and is not recommended to be used in signal handlers, you should use write instead. See Section 10.6 (Figure 10.4) in the textbook for the list of reentrant functions that can be called inside a signal handler.

Now compile the file and run the program using the following example. Make sure to specify the appropriate process id (highlighted below).

```
$ gcc -Wall -o sigusr sigusr.c
$ ./sigusr &
[1] 23573
$
$ jobs
[1]+ Running ./sigusr &
$ kill -USR1 %1
received SIGUSR1
$
$ kill -USR2 %1
received SIGUSR2
$
$ kill -SIGUSR1 23573
received SIGUSR1
$ kill -STOP 23573

[1]+ Stopped ./sigusr
$
$ jobs
[1]+ Stopped ./sigusr
$ kill -USR1 %1

[1]+ Stopped ./sigusr
$ kill -USR2 %1

[1]+ Stopped ./sigusr
$ kill -CONT 23573
received SIGUSR2
received SIGUSR1
$
$ jobs
[1]+ Running ./sigusr &
$ kill -TERM %1
$
[1]+ Terminated ./sigusr
```

```
$ jobs
$
```

In the examples above we used the *kill* command that we used in the previous lab to generate the signal. While we used the kill command in the previous lab to terminate a process using the TERM signal (the default signal if no signal is specified), the kill command could be used to generate any other signal that is supported by the kernel. You can list all signals that can be generated using *kill -l*. For example, on the CS Linux systems you see the following output:

```
$ kill -l
1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP
6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL 10) SIGUSR1
11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR
31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
```

We will now extend the above example to handle other signal generated from keyboard such as Control-C (SIGINT), Control-Z (SIGTSTP), and Control-\ (SIGQUIT). In this example we will use a single signal handler to handle all these signals using a switch statement (instead of separate signal handlers). The complete source code is available here: `sighandler.c`

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

static void sig_usr(int signo) {
  switch(signo) {
    case SIGINT:
        printf("received SIGINT signal %d\n", signo);
        break;
    case SIGQUIT:
        printf("received SIGQUIT signal %d\n", signo);
        break;
    case SIGUSR1:
        printf("received SIGUSR1 signal %d\n", signo);
        break;
    case SIGUSR2:
        printf("received SIGUSR2 signal %d\n", signo);
```

```
        break;
    case SIGTSTP:
        printf("received SIGTSTP signal %d\n", signo);
        break;
    default:
        printf("received signal %d\n", signo);
  }
}

int main(void) {
   if (signal(SIGINT, sig_usr) == SIG_ERR) {
     printf("can't catch SIGINT\n");
     exit(-1);
   }
   if (signal(SIGQUIT, sig_usr) == SIG_ERR) {
     printf("can't catch SIGINT\n");
     exit(-1);
   }
   if (signal(SIGUSR1, sig_usr) == SIG_ERR) {
     printf("can't catch SIGUSR1\n");
     exit(-1);
   }
   if (signal(SIGUSR2, sig_usr) == SIG_ERR) {
     printf("can't catch SIGUSR2\n");
     exit(-1);
   }
   if (signal(SIGTSTP, sig_usr) == SIG_ERR) {
     printf("can't catch SIGTSTP\n");
     exit(-1);
   }
   for ( ; ; )
      pause();

   return 0;
}
```

Compile and run this program and if you enter Control-C or Control-Z or Control-\ you will see the corresponding messages printed to the screen as shown below:

```
$ ./a.out
^Creceived SIGINT signal 2
^Zreceived SIGTSTP signal 20
^\received SIGQUIT signal 3
```

Notice that we have replaced the default signal handlers to kill this job from the keyboard and the program is in an infinite loop with pause. As a result, we cannot terminate this program from the keyboard. We have to login to the specific machine this process is running and kill this process using either *kill* or *top* commands. You can follow the examples from the first part and kill this process.

Till now we used the kill command and the keyboard to generate the signals. Now we will generate the signal in a program using the C API for *kill* or *raise* system call. In this example, we replace the SIGINT signal handler with our own, and in the signal handler, ask the user if the process should be terminated, and then based on the response, continue with either terminating the process or letting it continue to run. We use the read function instead of scanf to emphasize that scanf is not a reentrant function. The complete program is available: `sigint.c`

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

static void sig_int(int signo) {
  ssize_t n;
  char buf[2];

  signal(signo, SIG_IGN); /* ignore signal first */
  printf("Do you really want to do this: [Y/N]? ");
  fflush(stdout);
  n = read(STDIN_FILENO, buf, 2);
  if ( buf[0] == 'Y' ) {
    raise(SIGTERM); // or kill(0, SIGTERM); // or exit (-1);
  } else {
    printf("Ignoring signal, be careful next time!\n");
    fflush(stdout);
  }
  signal(signo, sig_int); /* reinstall the signal handler */
}

int main(void) {
   if (signal(SIGINT, sig_int) == SIG_ERR) {
      printf("Unable to catch SIGINT\n");
      exit(-1);
   }
   for ( ; ; )
      pause();

   return 0;
}
```

Let us now consider how signals impact child processes created using fork/exec. Consider the `forkexecvp.c` example from the earlier lab. Download this to one of the CS Linux servers, compile and run the program with hw1 1000 as the argument (`hw1.c`). Type Control-C, what happens? Here is a sample session with the above steps:

```
$ ./a.out /home/UAB/puri/CS332/shared/hw1 1000
Wait for the child process to terminate
^C
```

```
$ ps -u
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
puri    19202  0.3  0.1 125440  3912 pts/0    Ss   10:05   0:00 -bash
puri    19320  0.0  0.0 161588  1868 pts/0    R+   10:06   0:00 ps -u
$
```

Notice that when you typed Control-C both the parent and child process were terminated. Now execute the program again with the same argument and type Control-Z, what happens? Here is a sample session:

```
$ ./a.out /home/UAB/puri/CS332/shared/hw1 1000
Wait for the child process to terminate
^Z
[1]+  Stopped                 ./a.out /home/UAB/puri/CS332/shared/hw1 1000
$
$ jobs
[1]+  Stopped                 ./a.out /home/UAB/puri/CS332/shared/hw1 1000
$ ps -u
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
puri    19202  0.1  0.1 125440  3912 pts/0    Ss   10:05   0:00 -bash
puri    19351  0.0  0.0   4220   352 pts/0    T    10:08   0:00 ./a.out /home/U
puri    19352 29.0  0.6  27800 23844 pts/0    T    10:08   0:01 /home/UAB/puri/
puri    19353  0.0  0.0 161588  1864 pts/0    R+   10:08   0:00 ps -u
$ kill -CONT 19352
$
$ ps -u
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
puri    19202  0.0  0.1 125440  3912 pts/0    Ss   10:05   0:00 -bash
puri    19351  0.0  0.0   4220   352 pts/0    T    10:08   0:00 ./a.out /home/U
puri    19352 17.9  0.6  27800 23844 pts/0    R    10:08   0:04 /home/UAB/puri/
puri    19355  0.0  0.0 161588  1872 pts/0    R+   10:08   0:00 ps -u
$ Time taken for size 1000 = 32.274239 seconds
$ jobs
[1]+  Stopped                 ./a.out /home/UAB/puri/CS332/shared/hw1 1000
$ kill -CONT 19351
$ Child process exited with status = 0
[19351]: Exiting program .....
[1]+  Done                    ./a.out /home/UAB/puri/CS332/shared/hw1 1000
$ jobs
$
```

Notice that both the parent and child processes were stopped (suspended). We use *kill* command to send the signal to continue or we could have used *fg* to execute the job in the foreground.

How do we change this behavior such that when Control-C or Control-Z is entered at the keyboard only the child process is interrupted or suspended, and the parent process continues to execute?

Additional example that uses SIGCHLD to display the exit status of the child process instead of waiting for the child to complete in the parent process: `sigchild.c`

Lab 9 Assignment

Modify the program `forkexecvp.c` such that when you type Control-C or Control-Z the child process is interrupted or suspended and the parent process continues to wait until it receives a quit signal (Control-\).

Submission

You are required to submit the lab to Canvas by the deadline. No late submissions will be accepted.

Submission Checklist:

- ❑ Upload the C source file (.c file) to Canvas as part of this lab submission.
- ❑ Test your program on GitHub codespaces and include a link to your lab assignment in your shared repository on GitHub in the comments.
- ❑ Upload your README.md file which should include instructions on compiling through your Makefile, running the program and documentation.
- ❑ Screenshots of your output.
- ❑ Upload a Makefile for compiling your code.

Please do not upload executables or object files. Independent Completion Forms are not required for lab assignments.