p6: Signal Handling

Due Monday by 10pm **Points** 90 **Submitting** a file upload
Available until May 6 at 10:33pm

GOALS ALARM ZERO REQUIREMENTS SUBMITTING

- This project must be done individually.
  - *It is academic misconduct to share your work with others in any form including posting it on publicly accessible web sites, such as GitHub.*
  - *It is academic misconduct for you to copy or use some or all of a program that has been written by someone else.*
- All work for this project is to be done on the **CS Department's instructional Linux machines (https://csl.cs.wisc.edu/services/instructional-facilities)** . You're encouraged to remotely login using the ssh command in the terminal app on Macs, or on Windows using an ssh client such as MobaXterm.
- All projects in this course are graded on **CS Department's instructional Linux machines (https://csl.cs.wisc.edu/services/instructional-facilities)** . To receive credit, make sure your code runs as you expect on these machines.

Learning GOALS

The purpose of this assignment is to gain insight into the asynchronous nature of interrupts and signals. You'll be writing three programs to explore these concepts, which will expand your C programming skills.

A Periodic ALARM

For this part, you'll be writing two programs called **intdate.c** and **sendsig.c**. The first program will handle three signals: a periodic signal from an alarm, a keyboard interrupt signal, and a user defined signal. The second program will be used to send signals to other programs including intdate.c.

Reminder: You are to do this work on the CS instructional lab machines. The setup and handling of signals is different on different platforms and might not work the same on other machines as it does in the CS lab machines.

Setting up the Alarm

Write a program, **intdate.c**, with just a main() function that runs an infinite loop such as:

```
while (1){
}
```

Before entering the infinite loop, the main() function has to do two things. First, it sets up an alarm that will go off 3 seconds later, causing a **SIGALRM** signal to be sent to the program. Second, it registers a signal handler to handle the **SIGALRM** signal so that signal can be received by the program. The signal handler is just another function you need to write inside your program. This handler function should print the pid (process id) of the program and the current time (in the same format as the Linux **date** command). It should also re-arm the alarm to go off again three seconds later, and then return back to the main function, which continues its infinite loop.

At this stage, the output will look like this:

```
[skrentny@jimbo] (44)$ ls
intdate*  intdate.c
[skrentny@jimbo] (45)$ ./intdate
Pid and time will be printed every 3 seconds.
Enter ^C to end the program.
PID: 18238 | Current Time: Mon Apr 17 20:01:11 2017
PID: 18238 | Current Time: Mon Apr 17 20:01:14 2017
PID: 18238 | Current Time: Mon Apr 17 20:01:17 2017
PID: 18238 | Current Time: Mon Apr 17 20:01:20 2017
PID: 18238 | Current Time: Mon Apr 17 20:01:23 2017
^C
[skrentny@jimbo] (46)$
```

Notice that to stop the program from running, you type in a **Control+c** (^C in the output above) in the command shell where the program is running. Typing Control+c sends an interrupt signal (called **SIGINT**) to the running program. The default behavior of that signal is to terminate the program.

Since both main() and the alarm handler need to know/use the number of seconds in order to arm the alarm, make this value a global variable. Recall that signal handlers are not called directly in the program. They cannot receive arguments from other functions in the program and so we'll use global variables to share information with them.

You'll use library functions and system calls to write the above program. It is important to check the return values of these functions, so that your program detects error conditions and acts in response to those errors. Refer the man pages of the following functions that you will use. To use the man pages, just type **man <section-number> <function-or-command-name>** at the Linux prompt. The manual section numbers will be either 2 (system calls) or 3 (C library functions). Read more on how to use man pages here (http://www.linfo.org/man.html) .

- **time()** and **ctime()**: are library calls to help your handler function obtain and print the time in the correct format.
- **getpid()**: is a system call to help your handler function obtain the pid of the program.
- **alarm()**: activates the **SIGALRM** signal to occur in a specified number of seconds.

- **sigaction()**: registers your handler function to be called when the specific type of signal (specified as the first parameter) is sent to the program. You are particularly interested in setting the **sa_handler** field of the structure that sigaction() needs; it specifies the handler function to run upon receiving the signal. **DO NOT USE** the signal() system call; instead, use sigaction() to register your handler.

Note: Make sure to initialize the sigaction struct via memset() so that it is cleared (i.e, zeroed out) before you use it.

```
struct sigaction act;
memset (&act, 0, sizeof(act));
```

User Defined Signals

Linux has two basic user defined signals, **SIGUSR1** and **SIGUSR2**. As the name suggests these signals are defined by a user program, which is free to choose whatever action it should take on catching these signals.

Extend your implementation of **intdate.c** so that it prints a message on receiving a **SIGUSR1** signal. It should also increment a global counter to keep tally of the number of times it receives **SIGUSR1**. For achieving this you will need to write another signal handler and register it to handle the **SIGUSR1** signal using sigaction() one more time.

You can test your implementation by using the **kill** command at the Linux prompt to send a **SIGUSR1** signal to **intdate.c**. Later, we will implement our own small program that can be used to send signals to other programs.

Note: If you are working remotely, make sure that different ssh sessions are made on the same machine (e.g., rockhopper-04) otherwise sending signals from one session to the other is going to fail.

Handling the Keyboard Interrupt Signal

The last modification you'll make to your **intdate.c** program will change the default behavior resulting when a Control+c is typed. The program should first print the number of times it received the **SIGUSR1** signal and then call **exit(0)**. You will need to write another signal handler and register it to handle the **SIGINT** signal again using sigaction(). With this modification the output of the program should look something like this:

```
[skrentny@jimbo] (66)$ ./intdate
Pid and time will be printed every 3 seconds.
Enter ^C to end the program.
PID: 18163 | Current Time: Mon Apr 17 20:00:03 2017
PID: 18163 | Current Time: Mon Apr 17 20:00:06 2017
SIGUSR1 caught!
PID: 18163 | Current Time: Mon Apr 17 20:00:09 2017
PID: 18163 | Current Time: Mon Apr 17 20:00:12 2017
SIGUSR1 caught!
PID: 18163 | Current Time: Mon Apr 17 20:00:15 2017
PID: 18163 | Current Time: Mon Apr 17 20:00:18 2017
```

```
PID: 18163 | Current Time: Mon Apr 17 20:00:21 2017
^C
SIGINT received.
SIGUSR1 was received 2 times. Exiting now.
[skrentny@jimbo] (67)$
```

Sending Signals

Finally, write a simple program **sendsig.c** which can send signals (**SIGINT** and **SIGUSR1**) to other programs using their pid. For this, you will need to use the system call **kill()**.

Your program should take two command line arguments: the type of signal (-i for **SIGINT** or -u for **SIGUSR1**) and the pid of the process to which the signal is sent. The output should look like the following:

```
[skrentny@jimbo] (77)$ sendsig
Usage: <signal type> <pid>
[skrentny@jimbo] (78)$ sendsig -u 18163
[skrentny@jimbo] (79)$ sendsig -u 18163
[skrentny@jimbo] (80)$ sendsig -i 18163
```

You should use **sendsig.c** along with **intdate.c** and make sure that both programs work as expected.

Divide by ZERO

For this part, write a program, **division.c**, that does the following in an infinite loop:

- Prompt for and read in one integer value.
- Prompt for and read in a second integer value.
- Calculate the quotient and remainder of doing the integer division operation: int1 / int2
- Print the results.
- Keep a total count of how many division operations were successfully completed.

Use fgets() to read each line of input (use a buffer of 100 bytes). **DO NOT USE** fscanf() or scanf() for this assignment. Then, use atoi() to translate that C string to an integer.

Normally, we'd design our programs to check that the user input for validity. For program we'll ignore error checking the input. If the user enters a bad integer value, don't worry about it. Just use whatever value atoi() returns.

At this stage the sample run of the program would appear as:

```
[skrentny@jimbo] (121)$ ls
division*  division.c
[skrentny@jimbo] (122)$ ./division
Enter first integer: 12
Enter second integer: 2
12 / 2 is 6 with a remainder of 0
```

```
Enter first integer: 100
Enter second integer: -7
100 / -7 is -14 with a remainder of 2
Enter first integer: 10
Enter second integer: 20
10 / 20 is 0 with a remainder of 10
Enter first integer: ab17
Enter second integer: 3
0 / 3 is 0 with a remainder of 0
Enter first integer: ^C
[skrentny@jimbo] (123)$
```

Please note the behavior of the program for a non-numeric input ‘**ab17**’. Handle similar inputs in the same way.

Try giving the input for the second integer as 0. This will cause a divide by zero exception. This unrecoverable arithmetic error occurs typically causes a program to crashe, but we can implement a signal handler to override this default behavior of the **SIGFPE** signal.

Modify your program with a handler that is registered to run if the program receives the **SIGFPE** signal. In the signal handler you will print a message stating that a divide by 0 operation was attempted, print the number of successfully completed division operations, and then gracefully exit the program using exit(0) instead of crashing. Below is a sample output of how your program should behave:

```
[skrentny@jimbo] (132)$ ./division
Enter first integer: 1
Enter second integer: 2
1 / 2 is 0 with a remainder of 1
Enter first integer: 1
Enter second integer: 0
Error: a division by 0 operation was attempted.
Total number of operations completed successfully: 1
The program will be terminated.
[skrentny@jimbo] (133)$
```

As was mentioned above, a global variable is used so that the count of the number of completed divisions can be accessible by both main() and your signal handler.

Lastly, your program should have a separate handler for the keyboard interrupt Control+c just like the **intdate.c** program did. Except in this program on the first Control+c signal, the handler should print the number of successfully completed division operations, and then exit the program using exit(0).

Here is the sample output for **division.c** that shows the graceful exit of the program in case of a **SIGINT** signal.

```
[skrentny@jimbo] (143)$ ./division
Enter first integer: 1
Enter second integer: 2
1 / 2 is 0 with a remainder of 1
Enter first integer: 3
Enter second integer: 4
```

```
3 / 4 is 0 with a remainder of 3
Enter first integer: ^C
Total number of operations successfully completed: 2
The program will be terminated.
[skrentny@jimbo] (144)$
```

Note: Implement two independent handlers; do not combine the handlers. Do not place the calls to sigaction() within the loop! These calls should be completed before entering the loop that requests and does division on the two integers.

REQUIREMENTS

- Your program must follow style guidelines as given in the **Style Guide**. Since you will be creating the entire source files on your own, style will count for more points on this assignment.
- Your program must follow commenting guidelines as given in the **Commenting Guide**. Since you will be creating the entire source files on your own, commenting will count for more points on this assignment.
- Your programs should operate exactly as the sample outputs shown above.
- We will compile each of your programs with

  ```
  gcc -Wall -m32 -std=gnu99
  ```

  on the Linux lab machines. So, your programs must compile there, and without warnings or errors.
- Your program must print error messages, as described and shown in the sample runs above.

SUBMITTING Your Work

Submit the following **source files** under Project 6 in Assignments on Canvas before the deadline:

1. intdate.c
2. sendsig.c
3. division.c

It is your responsibility to ensure your submission is complete with the correct file names having the correct contents. The following points will seem obvious to most, but we've found we must explicitly state them otherwise some students will request special treatment for their carelessness:

- **You will only receive credit for the files that you submit.** You will not receive credit for files that you do not submit. Forgetting to submit, not submitting all the listed files, or submitting executable files or other wrong files will result in you losing credit for the assignment.
- **Do not zip, compress, submit your files in a folder, or submit each file individually.** Submit only the text files as listed as a single submission.
- **Make sure your file names exactly match those listed.** If you resubmit your work, Canvas will modify the file names by appending a hyphen and a number (e.g., intdate-1.c) and these Canvas modified names are accepted for grading.

Repeated Submission: You may resubmit your work repeatedly until the deadline has passed. **We strongly encourage you to use Canvas as a place to store a back up copy of your work.** If you resubmit, you must resubmit all of your work rather than updating just some of the files.

File Upload Google Doc BOX Google Drive Office 365

Upload a file, or choose a file you've already uploaded.

File: Choose File No file chosen

+ Add Another File

Click here to find a file you've already uploaded

Comments...

Cancel Submit Assignment