p6: Signal Handling

Start Assignment

Due Friday by 11:59pm **Points** 90 **Submitting** a file upload
Available Apr 21 at 12am - May 5 at 11:59pm

GOALS ALARM ZERO REQUIREMENTS SUBMITTING

Announcements:

- **4/22 - CAUTION: There is no LATE day or OOPS (https://canvas.wisc.edu/courses/330348/assignments/1852099) point for this programming project. All work must be submitted to Canvas on or before the last day of classes.**
- Released
  - Complete this **p6 Getting Started Activity (https://canvas.wisc.edu/courses/330348/pages/p6-getting-started-activity)** to see how to create and work in multiple terminal windows connected to the same machine.
  - Start this assignment by learning more about the "man pages" the online user manual.
  - Lectures L23-L26 (Week 12 and 13) will each help with different parts of p6. Write comments for where to put each type of code.

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. You will also gain experience using the built-in Linux manual (**man** pages) which are the best source for how to use the various commands on any given machine. This is because the available man pages are specific to each machine.

A Periodic ALARM

For the ALARM , you'll be writing two programs: one called **mySigHandler.c** and another called **sendsig.c**.

The **mySigHandler** will be a signal handler with code to handle three different types of signals:

1. a periodic signal from an alarm
2. a keyboard interrupt signal
3. a user defined signal

The second program will be used to send signals to other programs including sending signals to your running **mySigHandler**.

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

Setting up the Alarm

Write a program, **mySigHandler.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 a few things. First, it sets an alarm that will go off 4 seconds later, causing the kernel to trigger a **SIGALRM** signal to be sent to the program. Next, it registers a signal handler to handle the **SIGALRM** signal so that process can do something other than default behavior when it receives the signal.

A signal handler is function you write to handle a specific signal should it occur. You will not call this function rather the kernel will call it if you properly register it for the desired signal.

Your alarm handler function should get and print the **pid** (*process id*) of the program and the current time (in the same format as the Linux **date** command). This function must also re-arm or restart a new alarm to go off again 4 seconds later, and then return back to the main function, which continues its infinite loop. Finally, the main function must provide some instruction to user before starting the infinite loop.

At this stage, the output will look like this:

```
[deppeler@liederkranz] (44)$ ls
mySigHandler*  mySigHandler.c
[deppeler@liederkranz] (45)$ ./mySigHandler
PID and time print every 4 seconds.
Type Ctrl-C to end the program.
PID: 6280 CURRENT TIME: Tue Nov 24 21:44:01 2020
PID: 6280 CURRENT TIME: Tue Nov 24 21:44:05 2020
PID: 6280 CURRENT TIME: Tue Nov 24 21:44:09 2020
PID: 6280 CURRENT TIME: Tue Nov 24 21:44:13 2020
PID: 6280 CURRENT TIME: Tue Nov 24 21:44:17 2020
^C
[deppeler@liederkranz] (46)$
```

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

Since both **main()** and the alarm handler need to know and use the number of seconds in order to arm the alarm, make the number of seconds a ***global*** variable. Signal handlers are not called directly in the program. They cannot receive arguments from other functions in the program and so 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 function calls, 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.

Use the man pages from your CS machine:

There are different sections of the online manual. By typing the section number, you can learn how to use the command from different places. The manual section numbers include:

- 1 how to Linux commands from the command line
- 2 how to make system calls
- 3 how to call C library functions in C programs.

Read more on how to use man pages **http://www.linfo.org/man.html (http://www.linfo.org/man.html)**

Type **man <section-number> <function-or-command-name>** at the Linux prompt.

time() and **ctime()** and **localtime()** are C library functions that can be used 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()**: is C library function used to set a **SIGALRM** signal to occur in a specified number of seconds.
- **sigaction()**: is a C library function used to register a *handler* function to be called if and when a specific type of signal is sent to the program. You will create multiple handlers for your program where each is registered its particular signal function. 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; you must 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));
... rest of code to set signal handler and check return values.
```

User Defined Signals

Linux has two basic user defined signals, **SIGUSR1** and **SIGUSR2**. As the name suggests the purpose and use of 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 **mySigHandler.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**. To achieve this, you will need to write another signal handler and register it to handle the **SIGUSR1** signal using sigaction() one more time.

Test your implementation by using the `kill` command at the Linux prompt to send a **SIGUSR1** signal to **mySigHandler.c**. Later in this assignment, you will implement a small program that can be used to send signals to other programs. See the user manual command `man 1 kill` to see how to use the `kill` command to send user signals and interrupts from the Linux prompt.

Note: You will be working in multiple different remote sessions, you will need connect each remote session to the same Linux machine (e.g., rockhopper-04). Otherwise, sending signals from a program in one terminal session to a program (process) in another session is going to fail. Repeat the [p6 Getting Started Activity (https://canvas.wisc.edu/courses/330348/pages/p6-getting-started-activity)](https://canvas.wisc.edu/courses/330348/pages/p6-getting-started-activity) to see how to do this if you are not sure.

Handling Ctrl-C, the Keyboard Interrupt Signal

The last modification you'll make to your **mySigHandler.c** program will change the default behavior that occurs when Ctrl+C (read as "control C") is typed. Ctrl-C is the keyboard ***int***errupt signal. When Ctrl-C is signaled and your program receives this signal, it must print the number of times it received the **SIGUSR1** signal (from the **sendsig** program explained later) and then call **exit(0)**. To handle the keyboard interrupt signal, 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:

```
[deppeler@liederkranz] (66)$ ./mySigHandler
PID and time print every 4 seconds.
Type Ctrl-C to end the program.
PID: 10985 CURRENT TIME: Tue Nov 24 22:00:12 2020
PID: 10985 CURRENT TIME: Tue Nov 24 22:00:16 2020
PID: 10985 CURRENT TIME: Tue Nov 24 22:00:20 2020
SIGUSR1 handled and counted!
PID: 10985 CURRENT TIME: Tue Nov 24 22:00:24 2020
SIGUSR1 handled and counted!
PID: 10985 CURRENT TIME: Tue Nov 24 22:00:28 2020
PID: 10985 CURRENT TIME: Tue Nov 24 22:00:32 2020
SIGUSR1 handled and counted!
SIGUSR1 handled and counted!
PID: 10985 CURRENT TIME: Tue Nov 24 22:00:36 2020
SIGUSR1 handled and counted!
PID: 10985 CURRENT TIME: Tue Nov 24 22:00:40 2020
```

```
PID: 10985 CURRENT TIME: Tue Nov 24 22:00:44 2020
^C
SIGINT handled.
SIGUSR1 was handled 5 times. Exiting now.
[deppeler@liederkranz] (67)$
```

Sending Signals

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

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

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

Run your **sendsig** program with the process id of a running **mySigHandler** program to see how your program handles the user signals (-u and -i) that you send. Make sure that both programs work as required (shown in examples). For example:

1. Start **mySigHandler** in a terminal window.
2. Note the **pid** for the running process and let it run.
3. Open a new terminal window connected to the same machine.
4. Run **sendsig** with **-u** and the **pid** of **mySigHandler**
5. Check that **mySigHandler** handles the user signal as required.
6. Try **sendsig -u pid** a few more times.
7. Then, try **sendsig -i pid** to end your **mySigHandler** program.

Note: When the SIGINT is sent to your signal handler program from another terminal or program call, it will behave slightly different than when you type Ctrl-C at the keyboard. At the keyboard, you see ***^C*** *that is echoed to the terminal window when you typed it. You will not see this* ***^C*** *character when the signal is sent from your sendsig program. Your handler code should still produce the desired output as before.*

Divide by ZERO

For the third program, you will add an exception handler that handles *divide by 0* exceptions.

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

- 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 as shown in the examples that follow.
- Keep a total count of how many division operations were successfully completed.

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

Normally, we'd design our programs to check the user input for validity. For this 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() or strtol()** return.

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

```
[deppeler@liederkranz] (121)$ ls
division*  division.c
[deppeler@liederkranz] (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
[deppeler@liederkranz] (123)$
```

Please note the behavior of the program example 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 arithmetic error that occurs typically causes a program to crash, 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:

```
[deppeler@liederkranz] (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.
[deppeler@liederkranz] (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 Ctrl+C just like the **mySigHandler.c** program did. Except in this program on the first Ctrl+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.

```
[deppeler@liederkranz] (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 completed successfully: 2
The program will be terminated.
[deppeler@liederkranz] (144)$
```

Note: Implement two independent handlers (functions); do not combine the signal handler code into one signal handler instance. Do not place the calls to **sigaction()** within the loop! These calls should be completed once before entering the loop that requests and does division on the two integers.

Why the name "kill"?

https://softwareengineering.stackexchange.com/questions/150800/why-is-the-kill-command-called-so (https://softwareengineering.stackexchange.com/questions/150800/why-is-the-kill-command-called-so)

REQUIREMENTS

- Your program must follow style guidelines as given in the **Style Guide (https://canvas.wisc.edu/courses/330348/pages/programming-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 (https://canvas.wisc.edu/courses/330348/pages/program-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 -O0
```

on the Linux lab machines. The `-O0` option sets `gcc` optimization level to `0`. So, your programs must compile there, and without warnings or errors. If you wish to also rename the output file that is generated, that is still done with two additional command line arguments: `-o output`

- Your program must print messages as described and shown in the sample runs above .

SUBMITTING Your Work

Leave plenty of time before the deadline to complete the two steps for submission found below. Submit your work before the due date. There is NO LATE period or opportunity to use **Oops** point since all work is due on the last class period date. No submissions or updates to submissions are accepted after the LATE period.

1.) Submit only the files listed below under Project p6 in Assignments on Canvas as a single submission. Do not zip, compress, submit your files in a folder, or submit each file individually.

- **mySigHandler.c**
- **sendsig.c**
- **division.c**

Repeated Submission: You may resubmit your work repeatedly so we strongly encourage you to use Canvas to store a backup of your current work. If you resubmit, submit all files. Canvas will modify your file names by appending a hyphen and a number (e.g., mySigHandler-1.c).

2.) Verify your submission to ensure it is complete and correct. If not, resubmit all of your work rather than updating just some of the files.

- **Make sure you have submitted all the files listed above.** Forgetting to submit or not submitting one or more of the listed files will result in you losing credit for the assignment.
- **Make sure the files that you have submitted have the correct contents.** Submitting the wrong version of your files, empty files, skeleton files, executable files, corrupted files, or other wrong files will result in you losing credit for the assignment.
- **Make sure your file names exactly match those listed above.** If you resubmit your work, Canvas will modify your file names as mentioned in **Repeated Submission** above. These Canvas modified names are accepted for grading.

Project p6 f20 (1)

Criteria	Ratings				Pts
1a. mySigHandler.c: Compilation gcc mySigHandler.c -Wall -m32 -std=gnu99 -o mySigHandler	**5 pts** **No Errors or Warnings**	**3 pts** **Compiler Warnings (no errors)**	**0 pts** **Compiler Errors**		5 pts
1b. sendsig.c: Compilation gcc sendsig.c -Wall -m32 -std=gnu99 -o sendsig	**5 pts** **No Errors or Warnings**	**3 pts** **Compiler Warnings (no errors)**	**0 pts** **Compiler Errors**		5 pts
1c. division.c: Compilation gcc division.c -Wall -m32 -std=gnu99 -o division	**5 pts** **No Errors or Warnings**	**3 pts** **Compiler Warnings (no errors)**	**0 pts** **Compiler Errors**		5 pts
2a. mySigHandler.c: Style and Commenting, and Good Programming Practices	**6 to >5.0 pts** **Excellent**	**5 to >0.0 pts** **Incomplete**	**0 pts** **Doesn't Follow Guides**		6 pts
2b. sendsig.c: Style and Commenting, and Good Programming Practices	**6 to >5.0 pts** **Excellent**	**5 to >0.0 pts** **Incomplete**	**0 pts** **Doesn't Follow Guides**		6 pts
2c. division.c: Style and Commenting, and Good Programming Practices	**6 to >5.0 pts** **Excellent**	**5 to >0.0 pts** **Incomplete**	**0 pts** **Doesn't Follow Guides**		6 pts
3a. mySigHandler.c & division.c: sigaction Return Value	**3 pts** **Always Checked**	**2 pts** **Mostly Checked**	**1 pts** **Sometimes Checked**	**0 pts** **Never Checked**	3 pts
3b. mySigHandler.c: time Return Value	**1 pts** **Always Checked**	**0 pts** **Not Always Checked**			1 pts
3c. mySigHandler.c: ctime Return Value	**1 pts** **Always Checked**	**0 pts** **Not Always Checked**			1 pts
3d. sendsig.c: kill Return Value	**1 pts** **Always Checked**	**0 pts** **Not Always Checked**			1 pts

Criteria	Ratings			Pts
3e. division.c: fgets Return Value	**1 pts** **Always Checked**	**0 pts** **Not Always Checked**		1 pts
4a. mySigHandler.c & division.c: sigaction Setup and Use struct sigaction setup properly; sigaction() called with proper args (signal() not used); freed if struct sigaction on heap	**5 to >4.0 pts** **Correct**	**4 to >0.0 pts** **Some Issues**	**0 pts** **Used signal Instead**	5 pts
4b. mySigHandler.c & division.c: sigaction() Call	**4 pts** **In main() Before Loop**	**0 pts** **In Loop or Signal Handlers**		4 pts
5a. mySigHandler.c: main() Infinite Loop	**4 pts** **Is Empty**	**2 pts** **Has Either printf() or alarm()**	**0 pts** **Has BOTH printf() and alarm()**	4 pts
5b. mySigHandler.c: SIGALRM Handler prints the pid and time, also sets another alarm	**5 pts** **Works as Specified**	**3 pts** **Partially Works**	**0 pts** **Doesn't Work**	5 pts
5c. mySigHandler.c: SIGUSR1 Handler prints a message and increments a counter	**4 pts** **Works as Specified**	**2 pts** **Partially Works**	**0 pts** **Doesn't Work**	4 pts
5d. mySigHandler.c: SIGINT Handler prints the counter and exits	**4 pts** **Works as Specified**	**2 pts** **Patially Works**	**0 pts** **Doesn't Work**	4 pts
6a. sendsig.c: Arguments checked argc for correct number; parsed argv correctly	**5 pts** **Works as Specified**	**3 pts** **Partially Works**	**0 pts** **Doesn't Work**	5 pts
6b. sendsig.c: Sends Signal	**4 pts** **Sent to Correct Process**	**0 pts** **Doesn't Work**		4 pts
7. division.c: main() Loop reads 2 integers and displays quotient and remainder	**5 pts** **Works as Specified**	**3 pts** **Partially Works**	**0 pts** **Doesn't Work**	5 pts

Criteria	Ratings			Pts
8a. division.c: SIGFPE Handler prints number of successful divisions and exits	**4 pts** **Works as Specified**	**2 pts** **Partially Works**	**0 pts** **Doesn't Work**	4 pts
8b. division.c: SIGFPE Handler Error Message	**2 pts** **Printed**	**0 pts** **Not Printed**		2 pts
8c. division.c: SIGINT Handler prints number of successful divisions and exits	**4 pts** **Works as Specified**	**2 pts** **Partially Works**	**0 pts** **Doesn't Work**	4 pts
				Total Points: 90