p6: Signal Handling

1. Logistics

1. 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.

2. All work for this project is to be done on one of the CS department's instructional Linux machines. You're welcome to remotely login using ssh or MobaXterm as was shown in lecture (or any other way that you figure out on your own).
3. All projects in this course are graded on CS department's instructional Linux machines. It is your responsibility to make sure that your code runs as you expect on these machines.

2. Learning Goals

The purpose of this assignment is to gain insight into the asynchronous nature of interrupts, handling signals, and expanding C programming skills. By their very definition, signals occur at times not connected to the running program; this makes them **asynchronous**.

3. A Periodic Alarm: intdate.c & sigsend.c

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

Note: Even if you have a personal computer with a C compiler, you will not be able to work on your own computer, as the setup and handling of the variety of signals is different on different platforms. Work on the CSL machines to do this assignment!

3.1 Setting up the Alarm

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

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

Now before entering the infinite loop the `main()` function has to do two things. Firstly, you will need to set up an alarm that will go off 3 seconds later, causing a **`SIGALRM`** signal to be sent to the

program. Secondly you will need to register a signal handler to handle the **SIGALRM** signal caught 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 Unix **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 doing nothing).

At this stage, the output should look something like this:

```
[haseeb@espresso] (49)$ ls
intdate*  intdate.c
[haseeb@espresso] (50)$ ./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
[haseeb@espresso] (51)$
```

Notice that to stop the program from running, you type in a **Control+c** in the shell where the program is running. Typing `Control+c` sends an interrupt signal (called **SIGINT**) to the running 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. Signal handlers are not invoked by another function within the program, so they cannot receive parameters from another function in the program.

You will 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>** in the linux terminal. For this assignment the section number will be either 2 (system calls) or 3 (C library functions). Read more on how to use man pages here (Links to an external site.)Links to an external site..

- **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 before you use it.

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

3.2 User Defined Signals

Linux has two user defined signals, **SIGUSR1** and **SIGUSR2**. As the name suggests these signals are user defined and the program is free to chose what action it should take on catching these signals.

Extend the 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 received **SIGUSR1**. For achieving this you will need to write another signal handler and register it to handle the **SIGUSR1** signal (using `sigaction()` just like you did for **SIGALRM**)

To test your implementation you may want to use the **kill** command on your terminal to send a **SIGUSR1** signal to **intdate.c**. In section 3.5, we will implement our own toy program which 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 (example, rockhopper-04) otherwise sending signals from one session to the other is going to fail.

3.3 Handling the Keyboard Interrupt Signal

In this section, you will modify your program so that it does something different other than exiting after a `Control+c` is typed. The program should print the number of times it received the **SIGUSR1** signal before calling **exit(0)**. You will need to write another signal handler and register it to handle the **SIGINT**signal (using `sigaction()` just like you did for **SIGALRM**). With this addition the output of the program should look something like this:

```
[haseeb@espresso] (67)$ ./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.
[haseeb@espresso] (68)$
```

3.4 Sending Signals

Till now we have mostly focussed on signal handling. In this section, you will 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`** and -u for **`SIGUSR1`**) and the pid of the program to which the signal needs to be sent. The output should look like the following:

```
[haseeb@espresso] (67)$ sendsig
Usage: <signal type> <pid>
[haseeb@espresso] (68)$ sendsig -u 18163
[haseeb@espresso] (69)$ sendsig -u 18163
[haseeb@espresso] (70)$ sendsig -i 18163
```

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

4. Divide by Zero: division.c

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 these 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.

Users tend to type in bad inputs occasionally. For ease of programming, ignore error checking on 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:

```
[haseeb@espresso] (118)$ ls
division*  division.c
[haseeb@espresso] (119)$ ./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
[haseeb@espresso] (120)$
```

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. The hardware traps when this unrecoverable arithmetic error occurs, and the program crashes, because it did not (catch and) handle the **SIGFPE** signal.

To remedy this situation, modify your program to set up a handler that will be called 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 exit the program using `exit(0)` (gracefully, instead of crashing). Below is a sample output of how your program should behave:

```
[haseeb@espresso] (124)$ ./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.
[haseeb@espresso] (125)$
```

The count of the number of completed divisions needs to be a global variable so it can be used by both `main()` and your signal handler.

Lastly your program should have a separate handler to handle 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.

```
[haseeb@espresso] (128)$ ./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.
[haseeb@espresso] (129)$
```

Note: 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. Implement 2 independent handlers; do not combine the handlers.

5. Requirements

- Your program must follow style guidelines as given in Style Guidelines. The guide is from CS302 for Java, but most of it is applicable for C as well. The relevant sections are braces, whitespace and indentation.
- Include a comment at the top of each source code file with your **name and section**. You must comment every function with a header comment. See the Commenting Guide, where applicable for C.
- 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**. It is your responsibility to ensure that your programs compile on the department Linux machines, and **points will be deducted** for any warnings or errors.

- Remember to do error handling in all your programs.

6. Submitting Your Work

Submit the following **source files** under Project 6 in Assignments on Canvas on or 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 or submit your files in a folder.** Submit only the text files as listed.
- **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.