Assignment 6

CS/ECE 354, Spring 2016

Due Friday, May 6th before 9 AM

1. Collaboration Policy

For this assignment, you may work in pairs (2 people). All students (whether working in a pair or not) must individually turn in the assignment. Therefore, 2 copies of the assignment will be turned in for those working as a pair. The grader will choose to grade only one copy of a pair's work, and assign the same score to both students.

If one member of a pair does something unexpected with respect to turning in the assignment (for example: bad copy of the source code, late, or perhaps not turning in portions of the assignment), then *both* of the pair get the same penalty and the same score.

If working in a pair, the names and sections of *both* students must appear as comments at the top of all files that will be turned in. Please read the handin section (section 5) of this specification carefully for details about **one extra file that should be submitted by people working in a pair**.

2. Learning Goals

The purpose of this assignment is to gain insight into the asynchronous nature of interrupts, handling interrupts and traps, and expanding C programming skills. By their very definition, interrupts occur at times not connected to the running program; this makes them **asynchronous**. You'll also learn the process of linking -- the last step in the build process.

3. Exception Handling

This section has 2 parts, you'll be writing a program for each part. These programs use specific software interrupts, requiring you to set up the capture of the interrupts to be handled, and also provide the handling code. You'll be submitting two C files (`intdate.c` and `division.c`) and a completed `p6questions` file for this section.

3.1 First Program: A Periodic Alarm

3.1.1 Periodic Alarm: Step 1

Write a C program called `intdate.c` that is composed of two parts. One part is the main function, which does absolutely nothing in an infinite loop. Before entering the infinite loop, the main function will set up what is to happen when an alarm goes off 3 seconds later. When the alarm goes off, this causes a `SIGALRM` interrupt to signal the program. This interrupt is to be caught by the program, and handled by the second part of the program, an interrupt handler. This handler function will print out the current time, in the same format as the Unix date command, re-arm the alarm to go off three seconds later, and then return back to the main function (which continues its infinite loop doing nothing).

Since the main function is to keep running forever, the `main()` function will contain an infinite loop such as

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

Before executing the infinite loop, the `main()` function needs to set up the alarm. Once it starts executing the infinite loop it does nothing useful. For this partially complete program, output will look something like

```
% ./intdate
Date will be printed every 3 seconds.
Enter ^C to end the program:
current time is Tue Mar  4 13:15:21 2014
current time is Tue Mar  4 13:15:24 2014
current time is Tue Mar  4 13:15:27 2014
^C
```

Notice that to stop the program from running, you type in a Control+c. Remember for the next part of this assignment: typing Control+c sends a software interrupt (called `SIGINT`) to the running program.

You will use library functions to help you. It is important to check the return values of these

functions, such that your program detects error conditions and acts in response to those errors. Refer the man pages of following functions that you will use:

- `time()` and `ctime()` are library calls to help your handler obtain and print the time in the correct format.
- `alarm()` activates the `SIGALRM` interrupt to occur in a specified number of seconds.
- `sigaction()` (DO NOT use `signal()`) sets up what happens when the specific type of interrupt (specified as the first parameter) causes a software interrupt. You are particularly interested in setting the `sa_handler` field of the structure that `sigaction()` needs; it specifies the handler function to run upon an interrupt.

  NOTE: Initialize the sigaction struct via memset to be clear before you use it.

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

3.1.2 Periodic Alarm: Step 2

Once the periodic printing of the time is working, make an addition(s) to the program so that it does something different other than exiting after the first time a Control+c is typed. This extension to program only exits after the user types Control+c 5 times. Set up a `SIGINT` handler (using `sigaction()` to set up the call back function). The `SIGINT` handler either prints how many more times Control+c must be typed before exiting, or it prints that it caught the 5th one and it calls `exit()`.

Output of the program looked like this on Feb 26:

```
Date will be printed every 3 seconds.
Enter ^C 5 times to end the program:
^C
Control+c caught. 4 more before program is ended.

current time is Thu Feb 26 15:15:51 2015
^C
Control+c caught. 3 more before program is ended.
^C
Control+c caught. 2 more before program is ended.
^C
Control+c caught. 1 more before program is ended.
```

```
current time is Thu Feb 26 15:15:54 2015
^C
Final Control+c caught. Exiting.
```

The alarm interrupt handler will need to re-arm the alarm each time it is called. 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. Interrupt handlers are not invoked by another function within the program, so they cannot receive parameters from another function in the program.

You will also need a global variable to keep track of the number of times a Control+c has been typed. It is only used by the `SIGINT` handler, but it needs to exist (single instantiation) the entire time that the program runs.

3.1.3 Periodic Alarm: Step 3

1.Once you are through with this program, copy /p/course/cs354-common/public/bin/demo.c to your current working folder. Compile the program using the following option:

```
gcc demo.c -o demo -Wall -m32 -std=gnu99
```

2. Run the program using:
`./demo`

3. Observe the result. It should be something like this:
```
Enter ^C to end the program:
```

4. Now Press "Control+c" to quit the program. Open the file demo.c and go through the code. Notice the printf statement after the infinite loop.
```
printf("This should not be printed\n");
```

Because the program is stuck in an infinite loop, the above printf statement is not executed. That is, the program does not go forward.

6. Now copy the file `/p/course/cs354-common/public/bin/p6questions` to your folder and answer the question in the file.

7. Once you are done answering the question, copy this file (i.e. `p6questions`) along with `intdate.c` in your handin folder for submission of the first part of this assignment.

VERY IMPORTANT: Use the following command to compile your code:

```
% gcc -o intdate intdate.c -Wall -m32 -std=gnu99
```

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 interrupts is different on different platforms. **Work on the CSL machines to do this assignment!**

3.2 Divide by zero exception handling

Write a simple program that loops (forever) to:

- prompt for and read in one integer value (followed by the newline)
- prompt for and read in a second integer value (followed by the newline)
- calculate the quotient and remainder of doing the integer division operation: int1 / int2, printing these results, and keeping track of how many division operations were successfully completed.

A Control+c will cause this program to stop running.

Use `fgets()` to read each line of input. Then, use `atoi()` to translate that C string to an integer.

Users tend to type in bad inputs occasionally. For ease of programming, mostly ignore error checking on the input. If the user enters in a bad integer value, don't check for a bad integer value, and don't worry about it. Just use whatever value `atoi()` returns. If you still don't know what this value is, look it up in the `atoi()` man page!

The count of the number of completed divisions needs to be a global variable, as it will be needed by the second program enhancement (described below).

Call the source code for this program `division.c`.

A sample run of the program might appear as

```
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
```

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

Once this program is working, enhance it in two ways.

Try your program on a divide by 0 operation. What happens?
The hardware traps when this unrecoverable arithmetic error occurs, and the program crashes, because it did not (catch and) handle the **`SIGFPE`** signal.

To make this situation a little bit better, set up a handler that will be called if the program receives the `SIGFPE` signal. In the signal handler you write, 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 (gracefully, instead of crashing). Below is a sample output of how such a program will behave.

```
Enter first integer: 1
Enter second integer: 0
A divide by zero error has occurred.
Number of successful divisions: 0
```

You needed to interrupt this program to cause it to stop running. The second enhancement to this program captures the Control+c just like the intdate program did, but on the first Control+c

interrupt, the handler prints the number of successfully completed division operations, and then exits the program (gracefully).
Set up and add a handler for the `SIGINT` signal. The handler is to print a little message stating the number of completed operations, and then exit the program (gracefully).
Here is the sample output for division.c that shows the graceful exit of the program in case of a SIGINT interrupt.

```
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
Number of successful divisions: 2
```

You'll be turning in `division.c` for this part of the assignment.

Useful functions that you will need to use:

- `fgets()` reads up to a defined maximum number of characters into a buffer, stopping before that maximum is reached if a newline or EOF is encountered.
- `atoi()` converts a C string into the integer represented by that string.
- `sigaction()` (DO NOT use `signal()`) sets up what happens when the specific type of interrupt specified as the first parameter causes a software interrupt. You will be calling `sigaction()` twice, once to set up the handling of `SIGFPE`, and another time to set up the handling of `SIGINT`. Implement 2 independent handlers; do not combine the handlers. Do not place the calls to `sigaction()` within a loop. These calls will be completed before entering the loop that requests and does division on the two integers.

4. Linking

NOTE: The answers to this part should be written in the file named **`linking_answers`**

In Assignment 0, you have already explored the build process of a simple program hello.c. Phases involved were preprocessing, compiling, assembling and linking, and they generated hello.i, hello.s, hello.o, hello respectively. Despite its complexity, you were told that a short command can do all of the above for you:

```
gcc -m32 -Wall -std=gnu99 -o hello hello.c
```

But in later assignments, you were provided with another command, **make**, and invoking gcc by yourself did not work sometimes. Actually, the above command only works when you put all your code in a single file. But as you write larger programs and some of your programs may share the same code, it is better to organize the code in separate files. In this assignment, we will focus on how programs composed of multiple files are built.

Copy the following 5 files from folder
`/p/course/cs354-common/public/bin/p6_linking/` into your private p6 folder:
`calc.h calc.c prog1.c prog2.c linking_answers`

NOTE: Always use the flags **`-m32 -Wall -std=gnu99`** during compilation.

Question 1

You are provided 2 programs `prog1.c` and `prog2.c`. Both of them call the same power function defined in `calc.c`. Try to build them following Figure 1 below.

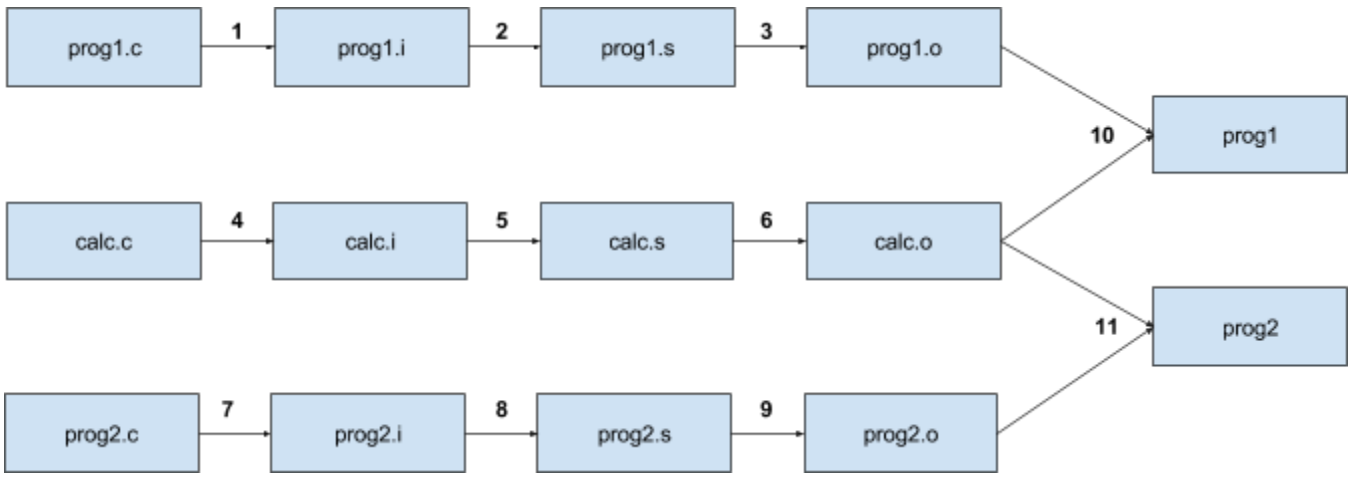

Figure 1

The command used for performing step 2 in figure 1 is:
`gcc -m32 -Wall -std=gnu99 -o prog.s -S prog1.i`

Write the commands you used for performing steps 1, 6 and 11.

Question 2

Now we find a bug in the definition of `power`. If `exp` is passed a negative number, it returns 1. Please modify the code to increment `num_err` and return `POWER_ERR` in such case. Since some changes have been made to `calc.c`, we need to repeat some steps of the build process so that the executables now use the updated `power` function. To ensure **`prog2's`** correctness, what are the **necessary steps** (among the 11 steps shown in figure 1) that you need to execute again?

NOTE: If you mention any steps that are NOT necessary, you'll lose points for this question.

Question 3

Which of the 11 steps in figure 1 do we need to execute again if we change the value of **`POWER_ERR`** in **`calc.h`**?

HINT: Try to track all the files where the value of `POWER_ERR` appears in the 11 generated files!

Question 4

`prog2` now returns `-1` when an error occurs. Can you modify `prog2.c` to make it return the value of `num_err` in `calc.c`? YOU SHOULD NOT MODIFY ANY OTHER FILES (EXCEPT prog2.c) FOR THIS QUESTION. Besides the return statement, you will need to add **only one line of cod**e. Write that one line of code that you added in `prog2.c` (excluding the return statement) and check if your program works as expected. Copy the single line of code that you added to the file `linking_answers`.

NOTE: Your code should compile and run without any errors after adding this line of code in `prog2.c`.

Question 5

Can you print the value of the variable `num_calc` (defined in the file `calc.c`) in `prog1.c` or `prog2.c`?

If your answer is YES, copy the additional line(s) of code that you used to achieve it to the file `linking_answers`.
If your answer is NO, please explain why you can't do so.

5. Handing in the assignment

You will be turning in **4 files** in total for this assignment:

1. intdate.c
2. p6questions
3. division.c
4. linking_answers

Copy all these 4 files into your handin directory. Your handin directory for this project is
/p/course/cs354-common/public/spring16.handin/login/p6
where login is your cs-login.

If you are working as part of a pair, you must turn in an **extra file**. This file will contain the names and sections of **both** students in the pair. As an example, if Kevin worked with Haseeb on this assignment, the contents of the extra file for both Kevin and Haseeb would be
Kevin Zhang section 1
Haseeb Tariq section 2

The name of this file is specialized to help the 354 automated grading tools identify who worked together. This file name is composed of the CS logins of the partners separated by a period. The file name is of the form <login1>.<login2>. Kevin's login is kzhang, and Haseeb's login is haseeb. The file name that both use will be kzhang.haseeb; please have both partners use the same file name. It does not matter which partner's name is first within this file name.

Requirements

1. Include a comment at the top of the source code of all programs with your *name* and *section* (and your partner's name and section, if working in a pair).
2. Use the instructional Linux machines for this assignment! A penalty will be imposed for any program that was obviously edited on a Windows machine and transferred to the Unix machines for turning in. It is annoying to see Windows line endings (^M) at the end of every line. Points will be taken off for such mistakes.
3. Your programs must compile on an instructional Linux machine as indicated in this specification *without warnings or errors*.

6. About Copying Code and Academic Misconduct

Don't cheat. Read this link carefully.

*Do **not** post your assignment solutions (or drafts) on any publicly accessible web sites. This specifically includes GitHub. It is academic misconduct to post your solution.*

For almost any C program that does something useful, someone has already written this program and further, has posted it for others to use. These programs do not do much that is useful, and are not likely posted anywhere. Still, it is academic misconduct for you to copy or use some or all of a program that has been written by someone else.

The penalty for academic misconduct on this assignment (and all CS/ECE 354 assignments) will be a failing grade in the course. This penalty is significantly more harsh than if you simply do not do the assignment. You will gain much more by doing the assignment than by copying, possibly modifying, and turning in someone else's effort.

*Do **not** post your assignment solution (or drafts) on any publicly accessible web sites. This specifically includes GitHub. It is academic misconduct to post your solution.*