CS 4760 **Operating Systems**
Assignment 2 Due Date: February 26, 2020

Concurrent UNIX Processes and shared memory

The goal of this homework is to become familiar with using shared memory and creating multiple processes. In this project we will be using multiple processes to determine if a set of numbers are prime or not. We will be using `getopt` and `perror` as well as fork().

Problem: You will write a program that will test a number given as its command line argument to check whether the number is a prime number. The code will run as a child process to another *controller* process.

You will create two executable files. The first executable file, `oss`, will be in charge of launching a specific number of child processes at various times using a `fork` followed by an `exec`. `oss` should then keep track of how many children have finished executing and terminate itself only when all of its children have finished.

When you run `oss`, it should take in several command line options as follows:

`-h`	Describe how the project should be run and then, terminate.
`-n x`	Indicate the maximum total of child processes `oss` will ever create. (Default 4)
`-s x`	Indicate the number of children allowed to exist in the system at the same time. (Default 2)
`-b B`	Start of the sequence of numbers to be tested for primality
`-i I`	Increment between numbers that we test
`-o filename`	Output file.

Note that all of these options should have some sensible default values, which should be described if `-h` is specified. For example, if the project is called with: `oss -n 5 -s 2 -b 101 -i 4 -o output.log` then `oss` will fork/exec 2 child processes but then not create any more until one of them terminated. Once a child terminates, it'll create another, continuing this until it has created a total of 5 children. At that point it will wait until all of them have terminated. When done, it would output appropriate information to the file `output.log`. The parameters `-b` and `-i` mean that our child processes would check the primality of the numbers 101, 105, 109, 113, and 117 (5 different numbers, one for each child launched).

Since checking primality may take a long time, we want to set a maximum amount of time for a child to try and determine if a number is prime and we also want a way for a child to tell the master process that it has finished. We will accomplish this synchronization with shared memory. `oss` will also be responsible for creating shared memory to store two integers and then initializing those integers to zero. This shared memory should be accessible to the children and will act as a *clock* – one integer represents seconds, the other one represents nanoseconds. Note that this is our *simulated* clock and will not have any correlation to real time. `oss` will be responsible for advancing this clock by incrementing it at various times as described later. Note that child processes do not ever increment the clock, they only look at it (provide read-only access to children). In addition to this shared clock, it should also allocate in shared memory an array equal in size to the maximum number of child processes that it will ever launch and it should initialize all the locations in this array to 0.

Implementation: `oss` starts by parsing command line options and then allocating shared memory. It should then launch a number of children equal to the maximum number it will simultaneously have in the system at any one time. When it does so, it should pass to each child a logical identification (for example, child 3) and the number that the child will test for primality using command line argument.

`oss` will then enter a loop. In this loop, it should start by incrementing the clock by 10000 (you can vary this number for testing in order to speed or slow down your clock). It should then see if any of the child processes have terminated. If so, it should first check to see the result of that child process (which it can look for in the shared memory array) and print the result to the output file. It should then fork and exec another child if it has not exceeded the maximum number of child processes to launch. If it already has enough child processes launched, then it should just not launch a child process and do another iteration of the loop. When logging completed children, it should note what time (based on our simulated clock) that it completed.

The children: The task in the child executable (let us call it `prime`) is fairly straightforward. It will be given two command line arguments when it is execed, which is its index/id and the number it should test for primality. In addition, I do not want any child process to run for any more than 1 millisecond of simulated time. So as soon as a child launches, it should look in the shared memory clock and store this time. It should then start trying to determine primality. Periodically it should check the time again and if more than a millisecond has passed in this time, it should finish by putting -1 in its associated location in the shared memory array. If it is able to determine if the number is prime, it should place that number in the array. If it determines that the number is not prime, it should put the negative of that number in the array. In either case, it should then detach shared memory and terminate.

The output file: The output file should have a line indicating when it launched each child process (showing the value of our simulated system clock when it was launched). It should also ouput a line when it detects that a child process has terminated (again based on the time in our system clock). Lastly, it should display a list of the numbers that it determined were prime and the numbers that it determined were not prime, followed by a list of numbers that it did not have the time to make a determination.

Termination Criteria: There are several termination criteria. First, if all the children have finished, the parent process should deallocate shared memory and terminate.

In addition, I expect your program to terminate after 2 seconds of real clock. This can be done using a timeout signal, at which point it should kill all currently running child processes and terminate. It should also catch the `ctrl-c` signal, free up shared memory and then terminate all children. No matter how it terminates, `oss` should also output the value of the shared clock to the output file. For an example of a periodic timer interrupt, you can look at p318 in the text, in the program `periodicasterik.c`.

I suggest you implement these requirements in the following order:

1. Get a `Makefile` that compiles the two source files.
2. Have your main executable read in the command line arguments and output the values to your screen (and ensure `-h` displays useful results).
3. Have master allocate shared memory, use it, then deallocate it. Make sure to check all possible error returns.
4. Get Master to fork and exec one child and have that child attach to shared memory and read the clock. Have the child output any information it was passed from master and then the value of the clock to test for correct launch. Master should wait for it to terminate and then output the value of the clock at that time.
5. Put in the signal handling to terminate after 2 seconds. A good idea to test this is to simply have the child go into an infinite loop so master will not ever terminate normally. Once this is working have it catch `Ctrl-c` and free up the shared memory, send a kill signal to the child and then terminate itself.
6. Set up the code to fork child processes until the specific limits.
7. Make each child process test the primality of their numbers.

If you do it in this order, incrementally, you help make sure that the basic fundamentals are working before getting to the point of launching many processes.

Make sure you never have more than 20 processes in the system at any time, even if the program is invoked with n being more than 20.

Hints

You will need to set up shared memory in this project to allow the processes to communicate with each other. Please check the man pages for `shmget`, `shmctl`, `shmat`, and `shmdt` to work with shared memory.

You will also need to set up signal processing and to do that, you will check on the functions for `signal` and `abort`. If you abort a process, make sure that the parent cleans up any allocated shared memory before dying.

In case you face problems, please use the shell command `ipcs` to find out any shared memory allocated to you and free it by using `ipcrm`.

Please make any error messages meaningful. The format for error messages should be:

```
oss: Error: Detailed error message
```

where `oss` is actually the name of the executable (`argv[0]`) that you are trying to execute. These error messages should be sent to `stderr` using `perror`.

What to handin

Handin an electronic copy of all the sources, `README`, `Makefile`(s), and results. Create your programs in a directory called *username*.2 where *username* is your login name on hoare. Once you are done with everything, *remove the executables and object files as well as log files*, and issue the following commands:

```
% cd
% chmod 755 ~
% ~sanjiv/bin/handin cs4760 2
% chmod 700 ~
```

Do not forget `Makefile` (with suffix or pattern rules), `RCS` (or some other version control like Git), and `README` for the assignment. If you do not use version control, you will lose 10 points. I want to see the log of how the file was modified. Omission of a `Makefile` will result in a loss of another 10 points, while `README` will cost you 5 points. Make sure that there is no shared memory left after your process finishes (normal or interrupted termination). Also, use relative path to execute the child.

You do not have to hand in a hard copy of the project. Assignment is due by 11:59pm on the due date.