CS 4760 **Operating Systems**
Assignment 2 Due Date: February 20, 2018

Concurrent UNIX Processes and shared memory

The goal of this homework is to become familiar with concurrent processing in Unix/Linux using shared memory.

Problem: In many applications, we have a number of processes trying to write into a single file. If any two processes open a file to append something to the file (think of logs as an example), the exact timing at which the file is closed becomes extremely important. This is the critical section problem.

For this assignment, we are going to simulate the writing of logs. In a nutshell, the problem can be specified as follows.

There is a process who reads from a single text file, picks up the next line of text, and puts it in a single buffer. Let there be five buffers and our producer process keeps on trying to fill those five buffers. Then, this process sets a flag to indicate that one of the five buffers is full. The process can write into a buffer only if the buffer is empty.

There are a number of other processes (consumers) who keep on looking for a buffer to become full. One of these processes should be able to pick up the text in the buffer and write it into the log file. The process must make sure that the file is not opened by another process for writing.

Your job is obviously to write the code to simulate the above problem. The main program will fork a producer and n consumers, with n supplied as a command line parameter. Let the default for n be 10. Thus, your master process performs just forking and signal handling. You may also want to allocate/deallocate shared memory in the master process.

The producer process will, at random times, wake up and see if a buffer is free. Obviously, buffers (and associated flags) need to be allocated in shared memory. Make sure that the buffers are writeable by producer and readable by consumers. The buffer flags (empty/full) must be readable as well as writeable by both producer and consumers. Only one consumer should be able to read a buffer at any time though five consumers can read five buffers simultaneously. You have to make sure that the synchronization between producer and consumers isvery well achieved. If you look at the code for critical section problem, you will get the template for the solution in Algorithm 4 of the lecture notes (critical section with multiple processes). Replace the `remainder_section` part of the template with a sleep for random time between 1 and 5 seconds (no zero delay).

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. Add the pid of the child to the file as comment in the log file. The preferred output format for log file is:

```
PID   Index    String
```

where `Index` is the logical number for consumer process, assigned internally by your code, and varies between 0 and $n - 1$.

The child process will be `exec`ed by the command equivalent to the shell comand

```
consumer xx
```

where `xx` is the index number of the child process. You can supply other parameters in `exec` as needed, for example shared memory addresses.

We also want to have a separate log file for producer and each consumer. This log file will be called prod.log and consxx.log where xx is the consumer number.

Producer will indicate the time (taken from system clock) at which it performed any activity, with the activity being defined as writing the buffer, waiting for a buffer to be available or checking the buffer availability, and going to sleep for r seconds. The log file format will be:

```
HH:MM:SS Started
HH:MM:SS Write   N   Message
HH:MM:SS Sleep   r
HH:MM:SS Check
HH:MM:SS Terminated  Reason
```

where `N` is the buffer number, and `Reason` is normal or killed.

The log file for consumer will be

```
HH:MM:SS Started
HH:MM:SS Read    N   Message
HH:MM:SS Sleep   r
HH:MM:SS Check
HH:MM:SS Attempt to write master log
HH:MM:SS Wrote master log
HH:MM:SS Terminated  Reason
```

The code for each child process should use the following template:

```
for ( ; ; )
{
    sleep for random amount of time (between 0 and 5 seconds);
    execute code to enter critical section;
    /* Critical section */
    execute code to exit from critical section;
}
```

At the end of it all, write a shell script to analyze your logs. I'll like some statistics on the sleep time for each process, number of text messages written, and number of attempts to enter critical section (for buffer as well as file). Also give me an average of those statistics.

Implementation

You will be required to create separate `consumer` processes from your main process. That is, the main process will just spawn the child processes and wait for them to finish. The *main* process also sets a timer at the start of computation to 100 seconds. If computation has not finished by this time, the *main* process kills all the spawned processes and then exits. Make sure that you print appropriate message(s).

In addition, the main process should print a message when an interrupt signal (`^C`) is received. All the children should be killed as a result. The children/grandchildren kill themselves upon receiving interrupt signal but print a message on `stderr` to indicate that they are dying because of an interrupt, along with the identification information. All other signals are ignored. Make sure that the processes handle multiple interrupts correctly. As a precaution, add this feature only after your programis well debugged.

The code for main and child processes should be compiled separately and the executables be called `master`, `producer`, and `consumer`.

Other points to remember: You are required to use `fork`, `exec` (or one of its variants), `wait`, and `exit` to manage multiple processes. Use `shmctl` suite of calls for shared memory allocation. Also make sure that you do not have more than twenty processes in the system at any time. You can do this by keeping a counter in `master` that gets incremented by `fork` and decremented by `wait`.

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`.

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
% ~hauschild/bin/handin cs4760 2
```

Make sure that your `$HOME` has permissions set to 0755 when you turn in the code. You can revert to 0700 after you have submitted.

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.

Your input data file should have at least 50 strings, one on each line.