CS 4760 **Operating Systems**
Project 3 Due Date: October 13, 2023

Semaphores and Message Passing

Purpose

The goal of this homework is to become familiar with semaphores in Linux. It is a repetition of your last project with the part that was handled by the concurrency algorithm to be implemented by semaphores.

You will again use multiple concurrent processes to write into a file at random times, solving the concurrency issues using semaphores for synchronization of processes. Your job is to create the environment such that two processes cannot write into the file simultaneously and yet, every process gets its turn to write into the file.

Task

Generate twenty processes using a master program, called `master`, and make them write into a file called `cstest` in their current working directory. Needless to say that all processes will use the same working directory. Each child process will be executed by an executable called `slave`. The message to be written into the file is:

```
HH:MM:SS File modified by process number xx
```

where `HH:MM:SS` is the current system time, `xx` is the logical process number as specified by `master`. The value of `xx` is between 1 and 20. This implies that the child process will be run by the command

```
slave xx
```

The critical resource is the file `cstest` which should be updated by a child under exclusive control access. This implies that each `slave` will have a critical section that will control access to the file to write into it.

The main program `master`

Write `master` that runs up to n `slave` processes at a time. Make sure that `n` never exceeds 20. Start `master` by typing the following command:

```
master -t ss n
```

where `ss` is the maximum time in seconds (default 100 seconds) after which the process should terminate itself if not completed.

Implement `master` as follows:

1. Check for the command line argument and output a usage message if the argument is not appropriate. If `n` is more than 20, issue a warning and limit `n` to 20. It will be a good idea to `#define` the maximum value of `n` or keep it as a configurable.
2. Allocate any shared memory needed as well as semaphores, and initialize them appropriately.
3. Execute the `slave` processes and wait for all of them to terminate.
4. Start a timer for specified number of seconds (default: 100). If all children have not terminated by then, terminate the children.
5. Deallocate shared memory and semaphores and terminate.

The Application Program (`slave`)

The `slave` just writes the message into the file inside the critical section. We want to have some log messages to see that the process is behaving appropriately and it does follow the guidance required for critical section.

If a process starts to execute code to enter the critical section, it must print a message to that effect in its own log file. I'll suggest naming the log file as `logfile.xx` where `xx` is the process number for the child, passed via the command line. It will be a good idea to include the time when that happens. Also, indicate the time in log file when the process actually enters and exits the critical section. Within the critical section, wait for a random number of seconds (in the range [1,3]) before you write into the file, and then, wait for another [1,3] seconds before leaving the critical section. For each child process, tweak the code so that the process requests and enters the critical section at most five times.

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

```
for ( i = 0; i < 5; i++ )
{
    wait for semaphore;
    sleep for random amount of time (between 1 and 3 seconds);
    critical_section();
    sleep for random amount of time (between 1 and 3 seconds);
    signal on semaphore;
}
```

Unlike last project, you do not have to specify the number of processes that participate in the critical section. However, you should keep the number of processes that can be forked concurrently under 20. The number of processes actually forked will come from the number specified on command line when you start the program. This implies that the `master` will stop forking processes as the limit is reached and will fork more only after some previously forked process terminates.

Implementation

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

`master` will also allocate shared memory for synchronization purposes. It will open and close a `logfile` but will not open `cstest`. `cstest` will be opened by the child process as it enters critical section (before the `sleep`) and closed as it exits.

In addition, `master` should also print a message when an interrupt signal (`^C`) is received. The child processes just ignore the interrupt signals (no messages on screen). Make sure that the processes handle multiple interrupts correctly. As a precaution, add this feature only after your program is well debugged.

The code for `master` and `slave` processes should be compiled separately and the executables be called `master` and `slave`.

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, if needed. Use `semctl` suite of system calls to handle semaphores. Make sure that you never have more than 20 processes in the system at any time, even if I specify a larger number in the command line (issue a warning in such a case).

Invoking the solution

`master` should be invoked using the following command:

```
master -t ss n
```

Termination Criteria: There are several termination criteria. First, if all the `slave`s have finished, `master` should deallocate shared memory and semaphore, and terminate.

In addition, I expect your program to terminate after the specified amount of time as specified in `config.h`. 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, `master` should also output the time of termination to the log file.

Suggested implementation steps

I suggest you implement these requirements in the following order:

1. Use the code from project 2, `makefile` and git setup. [Day 1]
2. Update `master` to work with semaphores. [Day 2]
3. Get `master` to fork and exec one child and have that child work with semaphore to perform the assigned task. Keep signal handling functionality from last project[Day 3]
4. Implement `slave` and test it as an independent process. [Day 4]
5. Set up the code to fork multiple child processes until the specific limits in the loop. Make sure everything works correctly. [Day 5]
6. Use the semaphore system calls to ensure that concurrency is handled appropriately. [Day 6-7]
7. Test the integrated solution. [Day 8]

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. This is a hard limit specified in the configuration file.

Hints

You will need to set up semaphores in this project to allow the processes to synchronize with each other. Please check the man pages for `semget`, `semctl`, and `semop` to work with semaphores. Do not forget to use `perror` to help with any debugging.

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 and semaphore before dying.

Make sure that you observe the precautions specified in the last project. In case you face problems, please use the shell command `ipcs` to find out any shared memory and semaphores allocated to you and free those by using `ipcrm`.

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

```
master: Error: Detailed error message
```

where `master` is actually the name of the executable (`argv[0]`) that you are trying to execute. These error messages may be sent to `stderr` using `perror`. Make judicious use of `perror` with all the system calls; it will help you debug quickly in case of trouble

Criteria for success

I have tried to give you detailed steps. There is not much room for change. Make sure that you follow good programming practices including proper indentation, documentation, and `Makefile` with suffix rules. The log file should be appropriately generated. *You must clean up after yourself.* That is, after the program terminates, whether normally or by force, there should be no shared memory or semaphore left that is allocated to you.

Grading

1. *Overall submission: 15 pts.* Program compiles and upon reading, seems to be able to solve the assigned problem in the specified manner (shared memory/semaphores/fork/exec).
2. *README: 5 pts.* Must address any special things you did, or if you missed anything.
3. *Makefile: 5pts.* Must use suffix rules or pattern rules. You'll receive only 2 points for `Makefile` without those rules.
4. *Command line parsing: 5 pts.* Program is able to parse the command line appropriately, assigning defaults as needed; issues help if needed. The configuration file should be properly defined, with the comments.
5. *Use of perror: 5pts.* Program outputs appropriate error messages, making use of perror(3). Errors follw the specified format.
6. *Code readability: 10 pts.* The code must be readable, with appropriate comments. Author and date should be identified.
7. *Proper fork/exec/wait: 15 pts.* Code appropriately performs the fork/exec/wait functions. There are no zombie processes. The number of processes is limited as specified in command line options.
8. *Signals: 10 pts.* Code reacts to signals as specified. When the parent is terminated, all children are terminated as well, and shared memory deallocated.
9. *Semaphores: 20pts.* Code properly uses and deallocates semaphores. Use `ftok` to generate keys.
10. *Conformance to specifications: 10pts.* Code properly creates the log file; the log file has messages in appropriate format; and appropriate messages are displayed to screen.

Submission

Submit an electronic copy of all the sources, `README`, `Makefile`(s), and results. Create your programs in a directory called *username*.3 where *username* is your login name on opsys. 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 3
% chmod 700 ~
```

Make sure that there is no persistent object (shared memory or semaphore) 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.