**CMP SCI 4760** **Operating Systems**
Project # 3 Due Date: Oct 12th, 2023

---

**Important:** Please do all projects on `opsys`

## Adding simple message queue

### Purpose

In the previous project added a system clock and a process table. In this project we are adding a message queue in order for oss and the workers to communicate.

### Task

As in the previous project, you will have two executables, oss and worker. The oss executable will be launching workers. However, in this project we are adding a different way to communicate and coordinate the processes. The clock will be worked with a bit differently, but the PCB remain the same as in previous projects, so the description of these are given as before.

Our main executable (oss) will now be maintaining a "simulated system clock" in shared memory. This system clock is not tied to the main clock of the system, but instead done separately. The clock consists of two separate integers (one storing seconds, the other nanoseconds) in shared memory, both of which are initialized to zero. This system clock must be accessible by the children, so it is required to be in shared memory. The children will not be modifying this clock for this assignment, but they will need to look at it.

In addition to this, oss will also maintain a process table (consisting of Process Control Blocks, one for each process). This process table does not need to be in shared memory. The first thing it should keep track of is the PID of the child process, as well as the time right before oss does a fork to launch that child process (based on our own simulated clock). It should also contain an entry for if this entry in the process table is empty (ie: not being used). I suggest making your process table an array of structs of PCBs, for example:

```
struct PCB {
    int occupied;       // either true or false
    pid_t pid;          // process id of this child
    int startSeconds;   // time when it was forked
    int startNano;      // time when it was forked
};

struct PCB processTable[20];
```

### worker, the children

In a similar fashion to the previous project, the worker will be launched with two arguments, which is its expected duration.

In a change from the previous project, the workers will doing their iterations differently. Instead of simply looking at the clock, instead they will do a loop where they do a msgrcv for a message from oss. When they get that message, then they check the clock to see if their time has expired. If it has not, then they will send a message back. They will keep doing this in a loop until their time has expired. So what should be in this message? I want the message queue to hold one integer. oss does not need to send any number to the children, but the child processes should send 0 back if they intend to terminate and a 1 back if they are not done yet.

For example, if the system clock was showing 6 seconds and 100 nanoseconds and the worker was passed 5 and 500000 as arguments, the worker process would do a loop waiting for a message from oss. When it got a message, it would check the time to see if over 11 seconds and 500100 nanoseconds is in the clock and if so send a message back with a 0.

So what output should the worker send? Upon starting up, it should output the following information:

```
WORKER PID:6577 PPID:6576 Called with oss: TermTimeS: 11 TermTimeNano: 500100
--Received message
```

The worker should then go into a loop, doing a msgrcv and then a msgsnd depending on its checks. It should also do some periodic output. Everytime it notices that the seconds have changed, it should output a message like:

```
WORKER PID:6577 PPID:6576 SysClockS: 6 SysclockNano: 45000000 TermTimeS: 11 TermTimeNano: 500
--1 seconds have passed since starting
```

and then one second later it would output:

```
WORKER PID:6577 PPID:6576 SysClockS: 7 SysclockNano: 500000 TermTimeS: 11 TermTimeNano: 50010
--2 seconds have passed since starting
```

Once its time has elapsed, it would do one final output:

```
WORKER PID:6577 PPID:6576 SysClockS: 11 SysclockNano: 700000 TermTimeS: 11 TermTimeNano: 5001
--Terminating
```

## oss, the parent

The task of oss is to launch a certain number of worker processes with particular parameters. These numbers are determined by its own command line arguments.

Your solution will be invoked using the following command:

```
oss [-h] [-n proc] [-s simul] [-t timelimit] [-f logfile]
```

While the first two parameters are similar to the previous project, the -t parameter is different. It now stands for the bound of time that a child process will be launched for. So for example, if it is called with -t 7, then when calling worker processes, it should call them with a time interval randomly between 1 second and 7 seconds (with nanoseconds also random). The -f parameter is for a log file, where you should write the output of oss (NOT the workers). The output of oss should both go to this log file, as well as the screen. This is for ease of use for this project, so yes I realize your output will be done in two places. Again, THE WORKER OUTPUT SHOULD NOT GO TO THIS FILE.

When started, oss will initialize the system clock and then go into a loop and start doing a fork() and then an exec() call to launch worker processes. However, it should only do this up to simul number of times. So if called with a -s of 3, we would launch no more than 3 initially. oss should make sure to update the process table with information as it is launching user processes.

This seems close to what we did before, however, will not be doing wait() calls as before. Instead, oss() will be going into a loop. This is different from the previous project though, as now oss is tightly controlling which order the processes get the clock. In particular, oss will be letting each worker check the clock in the order that they are in your process table. Pseudocode for this loop is below:

```
while (stillChildrenToLaunch) {

        incrementClock();
```

```
        Every half a second, output the process table to the screen and the log file

        if (childHasTerminated) {
            updatePCBOfTerminatedChild;
            possiblyLaunchNewChild(obeying process limits)
        }

        get pid of next child in pcb

        send message to that pid so that child can check the clock

        wait for a message back from that child

        if it indicates it is terminating, output that it intends to terminate
}
```

The check to see if a child has terminated should be done with a nonblocking wait() call. This can be done with code along the lines of:

```
   int pid = waitpid(-1, &status, WNOHANG);
```

waitpid will return 0 if no child processes have terminated and will return the pid of the child if one has terminated.

**Log Output**

Your oss should send enough output to a log file such that it is possible for me to determine the order that operations are being done. You should output the pid of the process that you are sending the message to, as well as the location where it is in your process table.

```
OSS: Sending message to worker 1 PID 517 at time 0:5000015
OSS: Recieving message from worker 1 PID 517 at tim 0;5000015
OSS: Sending message to worker 2 PID 519 at time 0:15000015
OSS: Recieving message from worker 2 PID 519 at tim 0;15000015
....
OSS: Sending message to worker 7 PID 528 at time 1:3000000
OSS: Recieving message from worker 7 PID 528 at time 1;3000000
OSS: Worker 7 PID 519 is planning to terminate.
OSS: Sending message to worker 1 PID 517 at time 1:4000000
OSS: Recieving message from worker 1 PID 517 at time 1;4000000
etc
```

In addition, every half a second in our simulated system, you should output the entire process table in a nice format. For example:

```
OSS PID:6576 SysClockS: 7 SysclockNano: 500000
Process Table:
Entry Occupied PID  StartS StartN
0     1        6577 5      500000
1     0        0    0      0
2     0        0    0      0
...
19    0        0    0      0
```

## Incrementing the clock

Each iteration in oss you need to increment the clock. So how much should you increment it? In this project you should increment the clock by much more than previous projects. As we are tightly controlling what children can check, set the increment to .1 second each time. This way processes will quickly cycle out through the system and we can see that they are running.

## Clock race conditions

We do not have to worry about race conditions in this project, as the children can only check the clock when oss is not incrementing it.

## Signal Handling

In addition, I expect your program to terminate after no more than 60 REAL LIFE seconds. 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.

This can be implemented by having your code send a termination signal after 60 seconds. You can find an example of how to do this in our textbook in a source file called periodicasterik.c

## Implementation details

It is required for this project that you use version control (git), a `Makefile`, and a `README`. Your `README` file should consist, at a minimum, of a description of how to compile and run your project, any outstanding problems that it still has, and any problems you encountered.

Your makefile should also compile BOTH executables every time. This requires the use of the all prefix.

As we are using shared memory, make sure to check and clear out shared memory if you have had errors. Please check the ipcrm and ipcs commands for this.

## Suggested implementation steps

1. Set up the source files and Makefiles [Day 1]
2. Write code for oss to parse options and receive the command parameters. [Day 2]
3. Implement oss initialization of shared memory and worker being able to take in arguments. At this stage, just make sure that worker can read the shared memory clock. [Day 3]
4. Implement oss to fork() and then exec() off one worker and have them just communicate back and forth using the message queue, making sure the log works. [Day 4-5]
5. Get oss to fork off workers up until the -n parameter and do their tasks [Day 6-7]
6. Implement the simultaneous restriction, as well as implement the process table and store data in it. [Day 8-9]
7. Testing and make sure your `README` file indicates how to run your project. Give a one-line example that would let me know how to run it. DO NOT SIMPLY COPY/PASTE THIS DOCUMENT INTO THE README [Day 10+]

## Criteria for success

Please follow the guidelines. Start small, implement one functionality, test. Do not wait until the last minute and contact me if you are having issues.

## Grading

1. *Overall submission: 10 pts*. Program compiles and upon reading, seems to be able to solve the assigned problem.
2. *Code readability: 10 pts*. The code must be readable, with appropriate comments. Author and date should be identified.
3. *Command line parsing: 10 pts*. Program is able to parse the command line appropriately, assigning defaults as needed; issues help if needed.
4. *Makefile: 10 pts*. Makefile works, compiles both files even if both are changed or one is changed. Also ensure your Makefile can do a clean.
5. *README: 10 pts*. Must address any special things you did, or if you missed anything.
6. *Conformance to specifications: 50 pts*. Does your application do the task.

After this project is due, I will be setting up appointments to quickly do a code review with individuals about their project. As long as you have done the project, this should not be a problem. Keep in mind that this code review can overwrite the grade guide if it is clear that you did not code your own project.

## Submission

Handin an electronic copy of all the sources, `README`, Makefile(s), and results. Create your programs in a directory called *username*.1 where *username* is your login name on opsys. Once you are done with everything, *remove the executables and object files*, and issue the following commands:

```
chmod 700 username.3

cp -p -r -v username.3 /home/hauschildm/cs4760/assignment3
```

If you have to resubmit, add a .2 to the end of your directory name and copy that over.

Do not forget `Makefile` (with suffix or pattern rules), your versioning files (`.git` subdircetory), 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 program files are modified. Therefore, you should use some logging mechanism, such as git, and let me know about it in your `README`. You must check in the files at least once a day while you are working on them. I do not like to see any extensions on `Makefile` and `README` files.

Before the final submission, perform a `make clean` and keep the latest source checked out in your directory.

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