CMP SCI 4760 **Operating Systems**
Project # 3 Due Date: March 18th, 2025

Important: Please do all projects on `opsys`

Adding functionality to our system

Purpose

In the previous project we had our oss create a simulated lock and then launch user processes who would then the check the clock. The previous project did not coordinate the user processes at all after running and it actually was possible for a race condition on the clock. In this project we are adding tight coordination between the user projects and oss. That is, they will only go through an internal loop and check the clock when oss tells them they should.

Task

As in the previous project, you will have two executables, oss and worker. The oss executable will be launching workers and oss will be maintaining a system clock. However, we have a few more details.

In particular, oss will now alternate sending messages to the users that it launches. That is, suppose we had three user processes running. oss would first second a message to p0, then after receiving a message back, would send a message to p1, then after getting a message back, send a message to p2 and so on, repeating the sequence as needed.

To keep track of necessary information, oss will 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
    int messagesSent    // total times oss sent a message to it
};

struct PCB processTable[20];
```

Note that your process table is limited in size and so should reuse old entries. That is, if the process occupying entry 0 finishes, then the next time you go to launch a process, it should go in that spot. When oss detects that a process has terminated, it can clear out that entry and set its occupied to 0, indicating that it is now a free slot.

worker, the children

The worker takes in two command line arguments, this time corresponding to how many seconds and nanoseconds it should decide to stay around in the system. For example, if you were running it directly you might call it like (note that oss launches this when you are actually running the project):

```
./worker 5 500000
```

The worker will start by attaching to shared memory. However, it will not examine it. It will then go into a loop.

```
do {
   msgrcv(from oss);

   check the clock

   determine if it is time to terminate

   msgsnd(to oss, saying if we are done or not)

} while (not done);
```

It determines the termination time by adding up the system clock time and the time passed to it in the initial call through command line arguments (in our simulated system clock, not actual time). This is when the process should decide to leave the system and terminate.

For example, if the system clock was showing 6 seconds and 100 nanoseconds and the worker was passed 5 and 500000 as above, the target time to terminate in the system would be 11 seconds and 500100 nanoseconds. The worker would check this termination time to see if it has passed each iteration of the loop. If it ever looks at the system clock and sees values over the ones when it should terminate, it should output some information, send a message to oss and then terminate.

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

```
WORKER PID:6577 PPID:6576 SysClockS: 5 SysclockNano: 1000 TermTimeS: 11 TermTimeNano: 500100
--Just Starting
```

The worker should then go into a loop, waiting for a message from oss, checking the clcok and then sending a message back. It should also do some periodic output.

In this project, unlike previous projects, you should output a message every iteration of the loop. Something like the following:

```
WORKER PID:6577 PPID:6576 SysClockS: 6 SysclockNano: 45000000 TermTimeS: 11 TermTimeNano: 500
--1 iteration has passed since it started
```

and then the next iteration it would output:

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

Once its time has elapsed, supposing it ran for 10 iterations, it would send out one final message:

```
WORKER PID:6577 PPID:6576 SysClockS: 11 SysclockNano: 700000 TermTimeS: 11 TermTimeNano: 5001
--Terminating after sending message back to oss after 10 iterations.
```

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 timelimitForChildren]
    [-i intervalInMsToLaunchChildren] [-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 -i parameter is new. This specifies how often you should launch a child (based on the system clock). For example, if -i 100 is passed, then your system would launch a child no more than once every 100 milliseconds. This is set to allow user processes to slowly go into the system, rather than flood in initially.

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, initialize and set up a message queue 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, incrementing the clock and then constantly checking to see if a child has terminated. Rough pseudocode for this loop is below:

```
while (stillChildrenToLaunch || someChildrenStillRunning) {

        incrementClock();

        calculateNextChildToSendAMessageTo

        printf("Sending message to child %d\n",nextChild);

        msgsnd(sendMessageToNextChildProcess);

        msgrcv(msg back from child process)

        printf("Rcved message from child %d\n",nextChild);
        if (child has decided to terminate) {
           printf("Child %d has decided to terminate\n",nextChild);
           wait(0);  // to give it time to clear out of the system
           updatePCBOfTerminatedChild
        }

        possiblyLaunchNewChild(obeying process limits and time bound limits)

}
```

You should have oss output the process table immediately after it launches a new child process, doing so in a readable format. It should also output the process table every half a second in our simulated system. For example:

```
OSS PID:6576 SysClockS: 7 SysclockNano: 500000
Process Table:
Entry Occupied PID  StartS StartN MessagesSent
0     1        6577 5      500000 489
1     0        0    0      0
```

```
2     0         0     0       0
...
19    0         0     0       0
```

Incrementing the clock

We increment the clock by a different amount in this project. As we are tightly coordinating the clock and sending out more output, in this project I want oss to increment the clock by 250ms divided by the number of current children. So for example, if we have 5 children running, you would increment the clock by 50ms every iteration. If you had just one child running, each iteration the clock would be incremented by 250ms (a quarter second).

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

Our message queue

The message queue for this project can consist of just one integer, with a 1 indicating that the user process is still running and a 0 indicating it intends to terminate. You can do it in other ways, but in later projects we will want to use at least an integer.

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
```

Ending Output

After all child processes have finished, your oss should output a summary of what happened in your system. For this project, this should consist of the total amount of processes that were launched, as well as the total number of times messages were sent from oss.

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 and message queues 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 message 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.

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 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.