CS 4760 **Operating Systems**
Assignment # 4 Due Date: 10/26/2023, 2023

Process Scheduling

In this project, you will simulate the process scheduling part of an operating system. You will implement time-based scheduling, ignoring all other parts of the OS. In this project we will be using message queues for synchronization.

Operating System Simulator

This will be your main program and serve as the master process. You will start the operating system simulator (call the executable `oss`) as one main process who will fork multiple children at random times. The randomness will be simulated by a logical clock that will also be updated by `oss`.

In the beginning, `oss` will allocate shared memory for any system data structures, which in this project only the clock needs to be in shared memory.

The process control block is a fixed size structure and contains information on managing the child process scheduling. Notice that since it is a simulator, you will not need to allocate space to save the context of child processes. But you must allocate space for scheduling-related items such as total CPU time used, total time in the system, your local simulated pid and the actual pid. The process control block resides in main memory and does not need to be accessible by the children. Since we are limiting ourselves to no more than 20 simultaneous processes in this project, you should allocate space for up to 18 process control blocks. Your process table should contain a method to keep track of what process control blocks are currently in use.

I suggest making your process table an array of structs of PCBs as below, but note you can create other fields. An example is below:

```
struct PCB {
    int occupied;       // either true or false
    pid_t pid;          // process id of this child
    int startSeconds;   // time when it was created
    int startNano;      // time when it was created
    int serviceTimeSeconds;   // total seconds it has been "scheduled"
    int serviceTimeNano;      // total nanoseconds it has been "scheduled"
    int eventWaitSec;   // when does its event happen?
    int eventWaitNano;  // when does its event happen?
    int blocked;        // is this process waiting on event?
};
```

struct PCB processTable[20];

In addition to the process table (or as part of it), oss will also have to implement a blocked queue and a ready queue.

oss, the parent

The OSS simulates time passing by using a simulated system clock. The clock is stored in two shared integers in memory, one which stores seconds and the other nanoseconds. will be in nanoseconds, so use two unsigned integers for the clock. `oss` will be creating user processes at as time goes on in the system. While the simulated clock (the two integers) are viewable by the child processes, it should only be advanced (changed) by `oss`.

Your solution will be invoked using the following command:

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

These parameters are similar but not the same as our previous projects. The -n parameter is the total number of child processes oss will ever launch. The -f is for a logfile as previously.

However, the -s is different. This project will create a new child process ever -t nanoseconds. If a time interval comes up to launch a new process, only then will the -s parameter be checked. This means WE WILL NOT be simply launching up to -s processes as we did in previous projects. Instead, processes will trickle into the system at a speed dependent on parameter -t. As we will eventually talk about, these processes will terminate gradually, so new processes will be allowed in, though we must never exceed -s processes in the system at the same time, nor exceed our overall total.

`oss` will create user processes after every time interval of -t passes. It *generates* a new process by allocating and initializing the process control block for the process and then, `fork`s and `exec`s the process.

`oss` acts as the scheduler and so will *schedule* a process by sending it a message using a message queue. When initially started, it should launch one worker process. After that, as time increments in the system, it would launch another after a -t time and so on. If the time is set to launch another process but your -s or -n parameters would be exceeded if you did so, just skip that generation.

How our clock will operate The clock in this project is incremented differently. Essentially, oss now adds time to the clock to indicate something that it had to do (if it was a real OS). In this project `oss` simulates time passing in the system by adding time to the clock and as it is the only process that would change the clock, only oss needs access to it. If a user process uses some time, `oss` should indicate this by advancing the clock. In the same fashion, if `oss` does something that should take some time if it were a real operating system, it should increment the clock by a small amount to indicate the time it spent.

Only one process will be ”running” in our simulated system at a time as we are doing scheduling.

Scheduling Algorithm. Assuming you have at least one process in your simulated system, `oss` will *select* a process to run and *schedule* it for execution. It will select the process by using a scheduling algorithm with the following features:

You will be keeping track of how long a process has been in the system in total. You are also keeping track of how long a process has been running on our simulated processor(accumulated service time). Each processes priority is based on the ratio of accumulated service time divided by total time in the system. If a worker is by itself in the system, then this ratio would be very close to 1. If a process has been in the system awhile but has not been given much scheduling time, then its ratio would be close to 0. Those processes with a smaller ratio should be selected first for scheduling. This would of course result in a higher ratio in comparison to other processes. Note that this could still result in some process being scheduled many times until it ”catches up” with another process.

Remember that when calculating this ratio that new processes in the system could have a denominator of zero. In that case, it makes sense to calculate them as if their priority was 0 (our higher priority), as that is a process that has not run at all and it would make sense to schedule it right away.

So when oss has to schedule a process, it should go through our ready queue and select the process to schedule that has the lowest of this ratio. When a process is scheduled, it should be scheduled for 50ms. The process will be *dispatched* by sending the process a message using a message queue indicating how much of a time slice it has to run. Note that this scheduling itself takes time, so before launching the process the `oss` should increment the clock for the amount of work that it did, let us say from 100 to 10000 nanoseconds.

Communication between processes

As described previously, the oss will be communicating with the user processes to ”schedule” them. These messages should not simply be put into the queue for any to read, but instead sent from oss directly to a particular user process. When the user process is finished with its task, it will send a message back to oss, and only oss, that it is now done with its time quantum. The message sent should be at least (could be more if you want to add more) a number, indicating how much time quantum they will be scheduled for. The message back can also be an integer, positive to indicate how much time they used up, negative if they indicate they are terminating after using some time. For example, of oss sent 5000 to a user process, that would indicate it was being given a time quantum of 5000 ns. If it received a -2500 back, the oss would read that as the user process used 2500ns before it terminated. If it got a 2500 back, it would indicate it used 2500 before it had an i/o interrupt. If it got 5000 back, the oss would understand that to mean the process used its entire time quantum.

User Processes

All user processes simulate the system by performing some work that would in theory take some random time (though our workers will not actually wait for that time to pass, after all we are a simulated system). The user processes will wait on receiving a message giving them a time slice and then it will simulate running. They do not do any *actual* work but instead send a message to `oss` saying how much time they used and if they had to use i/o or had to terminate. To emphasize, processes indicate they have taken up some time by sending a message back, they do not do any ”actual” running other than calculating the values below.

Processes have three things that they might need to do. One would be that they will terminate after using a percentage of the quantum they were assigned. Another would be that they are using their full time quantum. The other is that they are using part of their time quantum before having to do an i/o interrupt.

To figure out what percentage will decide to terminate, I suggest you have as a constant in your system, you should have a probability that a process will terminate when scheduled. I would suggest this probability be fairly small to start. Processes will then, using a random number, use this to determine if they should terminate. Note that when creating this random number you must be careful that the seed you use is different for all processes, so I suggest seeding off of some function of a processes pid. If it would terminate, it would of course use some random amount of its timeslice before terminating. It should indicate to `oss` that it has decided to terminate and also how much of its timeslice it used, so that `oss` can increment the clock by the appropriate amount.

Once it has decided that it will not terminate, then we have to determine if it will use its entire timeslice or if it will get blocked on an event. This should be determined by a random number. If it uses up its timeslice, this information should be conveyed to master. Otherwise, the process starts to wait for an event that will last for $r.s$ seconds where r and s are random numbers with range $[0, 5]$ and $[0, 1000]$ respectively, and 3 indicates that the process gets preempted after using p of its assigned quantum, where p is a random number in the range $[1, 99]$. As this could happen for multiple processes, this will require a blocked queue, checked by `oss` every time it makes a decision on scheduling to see if it should wake up these processes and put them back in the appropriate queues. Note that the simulated work of moving a process from a blocked queue to a ready queue would take more time than a normal scheduling decision so it would make sense to increment the system clock to indicate this.

Your simulation should end with a report on average wait time, average cpu utilization and average time a process waited in a blocked queue. Also include how long the CPU was idle with no ready processes.

Make sure that you have signal handing to terminate all processes, if needed. In case of abnormal termination, make sure to remove shared memory and message queues.

Overview of the procedure

```
while (stillChildrenToLaunch or childrenInSystem) {
       determine if we should launch a child

       check if a blocked process should be changed to ready

       calculate priorities of ready processes

       schedule a process by sending it a message

       receive a message back and update appropriate structures

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

}
```

You will want to increment your clock by a small amount everytime oss is updating one of its own data structures, as this is the work the oss is doing.

Log Output

Your program should send enough output to a log file such that it is possible for me to determine its operation. For example:

```
OSS: Generating process with PID 3 and putting it in ready queue 0 at time 0:5000015
OSS: Ready queue priorities [0.05,0.07,0.80,...]
OSS: Dispatching process with PID 2 priority 0.05 from ready queue at time 0:5000805,
OSS: total time this dispatch was 790 nanoseconds
```

```
OSS: Receiving that process with PID 2 ran for 400000 nanoseconds
OSS: Putting process with PID 2 into ready queue
OSS: Ready queue priorities [0.05,0.07,0.80,...]
OSS: Dispatching process with PID 3 priority 0.12 from ready queue at time 0:5401805,
OSS: total time this dispatch was 1000 nanoseconds
OSS: Receiving that process with PID 3 ran for 270000 nanoseconds,
OSS: not using its entire time quantum
OSS: Putting process with PID 3 into blocked queue
OSS: Ready queue priorities [0.05,0.07,0.80,...]
OSS: Dispatching process with PID 1 from ready queue priority 0.11 at time 0:5402505,
OSS: total time spent in dispatch was 7000 nanoseconds
etc
```

I suggest not simply appending to previous logs, but start a new file each time. Also be careful about infinite loops that could generate excessively long log files. So for example, keep track of total lines in the log file and terminate writing to the log if it exceeds 10000 lines.

Note that the above log was using arbitrary numbers, so your times spent in dispatch could be quite different.

I highly suggest you do this project incrementally. A suggested way to break it down...

- Have master create a process control table with one user process and create that user process, testing the message queues back and forth.
- Schedule the one user process over and over, logging the data
- Add additional user processes, now with our priority calculations.
- Add the chance for user processes to be blocked on an event, keep track of statistics on this.
- Keep track of and output statistics like throughput, wait time, etc

Do not try to do everything at once and be stuck with no idea what is failing.

Termination Criteria

`oss` should stop generating processes if it has already generated -n processes or if more than 3 real-life seconds have passed. If you stop adding new processes, the system should eventually empty of processes and then it should terminate. What is important is that you tune your parameters so that the system has processes in all the queues (at least 2 processes in the run queue) at some point and that I can see that in the log file. As discussed previously, ensure that appropriate statistics are displayed.

Criteria for success

Make sure that you implement the specified scheduling algorithm and document it appropriately. *You must clean up after yourself.* That is, after the program terminates, whether normally or by force, there should be no shared memory, semaphore, or message queue that is left allocated to you.

Grading

1. *Overall submission: 10pts*. Program compiles and upon reading, seems to solve the assigned problem in the specified manner.
2. *README/Makefile: 10pts*. Ensure that they are present and work appropriately.
3. *Code readability: 10pts*. Code should be readable with appropriate comments. Author and date should be identified.
4. *Signal handling/cleaning up after yourself: 20pts*. The code terminates appropriately and all IPC primitives are cleaned up.
5. *Proper scheduling algorithm: 20pts*. The algorithm should be correct and appropriately documented.
6. *Conformance to specifications: 30pts*. Overall proper implementation and documentation.

Deliverables

Handin an electronic copy of all the sources, `README`, Makefile(s), and results. Create your programs in a directory called *username*.4 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.4

cp -p -r username.4 /home/hauschildm/cs4760/assignment4
```

Do not forget `Makefile` (with suffix rules), version control, and `README` for the assignment. If you do not use version control, you will lose 10 points. Omission of a `Makefile` (with suffix rules) will result in a loss of another 10 points, while `README` will cost you 5 points.