Programming Assignment #2

COEN 346

Christopher Simpson (27588445)

Marc Bass (26488994)

I certify that this submission is my original work and meets the Faculty's Expectations of Originality

**INTRODUCTION**

On first glance at the assignment, the decision as a group was to create two separate schedulers. With one scheduler to deliver the processes and another to accept and manage the expired and active queues, this would accommodate the specifications. Once this initial decision was reached, the rest of the assignment was based around that concept. Initial concerns were made for the timings, and a deep look into the use of *chrono* in the standard library was considered.

This scheduler was also designed with the idea of possibly expanding the first assignment to inject new processes, in a de facto batch job system. This consideration coincided with the two scheduler concept.

The process scheduler can handle an input file with any ordering of arrival times, as long as the form and structure remain the same. The scheduler is expandable and scalable, with linked lists implementing the queues, allowing for many more processes. Multiple long term and short term schedulers were considered, and the possibility is there.

Some assumptions were made to complete the assignment. The priority, from the example in the assignment, updates more than once. We have set it so the priority updates each time the process receives two turns with the CPU. Secondly, the algorithm of the scheduler increases priority and reduces the time slice given, the longer it runs, if it is the sole process. This might not have been considered when the algorithm was designed. We assume this is as intended, as the scheduler should be most efficient when more than one process is in the queue.

**DISCUSSION**

Short Term Scheduler:

*void shortTermScheduler();*

The short-term scheduler is modeled after a round robin scheduler. Some changes to this scheduler were made based on the assignment’s requirements. A priority-queue, which changes the order in which the scheduler activates processes, and a dynamic time quantum were the two requirements for the short-term scheduler. As a result of the short-term scheduler running concurrently with the long-term scheduler, some synchronization tools had to be used to control access to the queues.

The short-term scheduler initially busy-waits, waiting for the long-term scheduler to indicate that it has parsed, and built its job queue. Once signaled, the short-term scheduler begins its main loop, ending only when the job queue, active queue, and expired queue are empty. In this loop, the scheduler checks whether the currently active queue is empty, and swaps the two queues if it is. Otherwise, the process at the top of the queue is made the active process, and resumed. The scheduler then sleeps until either the time quantum or burst time has elapsed. At this point, the process thread is suspended and has its priority updated if it has been run two times previously without being updated. Finally, the scheduler pops the active process off the active queue, and pushes it into the expired queue;

Long Term Scheduler:

*void longTermScheduler();*

The long term scheduler is designed to prepare the processes from an input file. The function instantiates the PCBs from the information parsed from the input file. The number of processes in the input file will truncate the list, likewise if the list is shorter than the number of processes, the list will terminate at fewer processes. The long term scheduler then requests and sends the PCB to the expired queue, which will sort based on priority. This function will be run on a separate thread than the short term scheduler.

The long term scheduler starts by calling the function *createJobQueue*, which uses *parseProcesses* to create a priority queue from the input.txt file in the working directory. Once the queue is created, a flag is flipped to inform the short term scheduler to start accepting jobs. The function then loops the entire queue, sleeping for the arrival time before sending the job to the expired queue. The function checks the mutexes, then then queues for the expired one, and sets the start time and last run time for the process.

A trigger to the output log function and a display message for clarity are sent as each process is pushed into the expired queue. The PCB is then popped from the job queue and allowed to continue in the expired queue. This is repeated until the job queue is empty. The possibility of adding processes while the long term scheduler is running is a possibility. Modifications would include a busy wait, a wait for more jobs if empty section and the relevant semaphores.

Input Parser:

*list<string \*> parseProcesses();*

The parsing is done by taking an entire line using *getline* and examining one character at a time. The trigger characters are an empty space and the tab character, ‘\t’. Each character is added to a string one by one, and once one of the trigger characters appears, the current string is pushed into the argument list. A check to see if the string is empty when a leading character of the line is a trigger, allows empty strings to be avoided as arguments.

The final argument is added at the end, as the loop terminates after examining the last character, without pushing it. The mutex is released and an error is set up in case the input file fails to open. The argument list is returned to be interpreted.

Process PCB Creation

*void createJobQueue();*

The job queue is created by first calling on *parseProcesses* to parse the input file in the working directory. This is assigned to a list<string \*>, and the PCBs for the job queue are created using this. The front of the list is the number of processes to be expected. The consecutive sets of 4 after the number of processes consist of the process name, scheduled arrival time, total burst time and initial priority. After a mutex check, these 4 pieces of information are collected as a group and a new PCB is created and pushed into the job queue.

The PCB is given a new thread created with *CreateThread* for windows. The thread is created suspended, allowed possible by the *CreateThread* parameters. The strings are converted to integers using the *stoi* function, and are then used to initialize *chrono::duration*s for the PCB. The durations help maintain for higher precision on timings.

The possibility to reduce the critical section to only the pushing of the PCB into the job queue is possible. This requires an array to store the needed values, strings and threads in preparation for the critical section. This was not optimized as the current implementation does not expect to share the job queue. This was left for possible expansibility with the first assignment.

**CONCLUSION**

Our implementation of the scheduler for assignment two was based on real-time measurements using the C++11 chrono libraries. It was built with modularity in mind, allowing it to be applied beyond simply simulating a process and instead actually managing CPU-time resources. Unfortunately, this precise use of real time measurements led to some difficulties. The first was the lack of precision in a real-time environment. Processes intended to run for 100ms would often encounter an error of ~0.5ms. In initial testing, this appeared unimportant, resulting in only a 12ms drift at most. However, upon further testing it became apparent that after several hundred context switches the drift would continue to increase linearly.

These problems underline the importance of speed during context switching. In our simulations, the issue became most apparent when dealing with a CPU bound process with a 5ms time slice. In these situations, even though the overhead of the context switch was only 0.5ms, that overhead would represent 10% of the overall CPU time of the system. Dropping the complexity of the context switch would not only decrease the time drift of the system mentioned above, but also the efficiency of the scheduler.

Despite the shortcomings, the scheduler works very well. It correctly reads an input and outputs a scheduler log file, and is easily expandable beyond the scope of the assignment. The assignment could have simulated timings, and ignored context-switch timing. However, doing so would have resulted in a less expandable implementation and would have been less representative of a real-world scheduling system.