Aida Bumbu

Elie Kheirallah

COEN 346 Programming Project 3
Process Scheduling

To Dr. TRABELSI
Due on 23/03/2018

"We certify that this submission is our original work and meets the Faculty's expectations of Originality"

Objectives:

- Implement an O(1) Scheduler, simplified version of the Linux scheduler in Linux versions prior to 2.6.63.
- Pass and simulate processes in the scheduler regardless of the size of the input.
- Make use of threads when calling the scheduler.
- Perform a successful simulation in terms of time and functionality.

Program status:

The program runs with user-defined test cases.

High-level description of the program

Functions

Global functions:

- Min: Takes in 2 integers, returns the lower value of the 2.
- Max: Takes in 2 integers, returns the higher value of the 2.
- addToQueue(Process* (&queue)[140], vector<process> &vect, int currentTime):

Takes an array of 140 pointers, for each of the priority levels. Takes in a vector which contains the processes that are to be run. Takes in the currentTime.
First checks if all processes has been added. Then checks if the time now is when a process arrives. If the check passes, it's time to add the process. Go to the array at index = priority of the process and check if the spot is empty. If it is empty, just add the process. Otherwise, move to the next spot in that index. The way this works is like a linkedList, where the first element will point to the 2nd, the 2nd to the 3rd, etc. The reason why this data structure was used is because we know there are 140 priorities, and no more. However, we do not know how many processes there are. In order to accommodate to this, we used an array of linkedlists.

- Scheduler(process* (&q0)[140], process* (&q1)[140], bool flag, int currentTime, vector<Process> & vect):

The scheduler takes in both lists that will be running, the flag to choose which list runs, the currentTime, and the vector containing the processes.
The 2 arrays are the active and expired queues. Based on the flag (0 or 1), a queue is chosen as active and the other as expired. The very first check the scheduler does is that it checks if all processes terminated. If not, the flag is checked to see which queue is active. The scheduler checks if the time now is when a process arrives, and if that is the case, it calls the function addToQueue to add the process. Then, the scheduler goes over the queue to check for processes that are to be run, and runs them. When a process runs, it acquires a lock that way it cannot be preempted. However, that does not stop the scheduler if a new process tries to join, since the processes join in the expired queue. After the time quantum passes, the scheduler checks if the process terminated to call the termination functions. If the process didn't terminate, it pauses it since the process is not allowed to have more than its granted time quantum at a time. The

scheduler always updates a variable oldRunning to keep track of which process last ran, that way if a process ran 2 consecutive times, it will update this process's priority. After these actions are done, the flag switches and the lock unlocks.

- sortVector(vector<Process> &a): takes vector by reference, sorts it according to arrival times.
- Struct args: takes the arguments that will be passed in the thread. Here: the 2 queues, the flag, the current time, and the vector.
- void* schedule(args arr): the function that will be passed to the thread to run. Calls scheduler.

Functions inside the process class:

- Print: Takes in current time and action. Will be used to output the time, process ID and what action the process is performing.
- Process(): Default constructor
- ~Process: Destructor. Pointer points to null. Status becomes empty. Object gets destroyed.
- Process(int pid, int arrivalTime, int cpuBurst, int prior): sets the pid, arrivaltime, cpuburst, priority, and calculates the Time Quantum granted to that process based on its priority.
- Process* GetNext() const: returns the next object in the line of that priority (the next item in the linkedlist).
- setNext(process a): next pointer points to object that is put after in the queue.
- getPID(): returns the process ID.
- getBurst(): returns the process's burst time.
- GetTQ(): returns the Time Quantum.
- getStatus(): returns the current status
- getArrival(): returns arrival time
- getPrio(): returns priority
- calcTq(int prior): takes in priority and calculates the time quantum that will be granted to that process.
- sumOfWaits(int currentTime): takes in currenttime and calculates total wait time (from last run till start of run).
- calcBonus(int CurrentTime): calculates the bonus to be given when a process runs 2 consecutive times.
- SetNewPrio(int currentTime): updates the priority to the max value between 100 and the min of ((priority - bonus + 5) and 139). Updates the Time Quantum as it gets changed.
- pause(int currentTime): pauses the running process. Is called by the scheduler. Prints out the status of the process.

- start(int currentTime): Initiates the run of the process in the CPU. Updates the cpuBurst as the process was granted its time quantum. Prints out the status of the process.
- Arrive(int currentTime): prints out the status of the process.
- Resume(int currentTime): resumes the process's run. Updates the sumOfWaits. Updates the cpuBurst since the process was granted its time quantum. Prints out the status of the process.
- terminate(int currentTime): decreases the total number of processes by 1 as a process finishes its run. Prints out the status of the process.

Main: the file is read using fstream and getline. The values are stored by taking the values separated by spaces, and the strings get converted to integers. Then they are inserted in process(pid, arrival, burst, prio) to create a process with these specifications. The processes then get added to a vector, the vector gets sorted, the values get stored in the struct, then the thread is called with schedule and the struct as arguments.

Variables

Inside process class:

- PID ⇒ contains ID of process (1,2,3,etc)
- arrivalTime ⇒ time of arrival of the process
- cpuBurst ⇒ how long the process will be running for
- Tq ⇒ time quantum associated with process
- *next ⇒ pointer in the scenario where 2 processes have the same priority, next will point to the 2nd pointer that enters.
- lastFinish ⇒ saves the last finish time of the process (when paused). Used for sumOfWait
- sumOfWait ⇒ the sum of the waiting times of the process
- Status ⇒ saves the status of the process (arrived, started, paused, resumed, terminated).

Global variables:

- positionVector ⇒ used to determine the position in the vector of sorted processes.
- Mtx ⇒ mutex used for locking.
- oldRunning ⇒ used to save the ID of the last process that ran. If the process runs 2 consecutive times, it will trigger the priority update (check if oldrunning == the current process that just ran. The check is done before updating oldrunning).
- numberOfProcesses ⇒ variable that increments every time a new process is inserted, and decremented when a process terminates.
- numberOfTotalProcceses ⇒ variable that stores the total amount of processes that have been added. Does not decrement as it is used to check if all processes have been inserted (compared with positionvector).

Results

The program runs with user-defined test cases.

For the first test, we used the following code written in the imputs.txt file:

3
P 1 1000 2500 90
P 2 2000 100 120
P 3 3200 100 120

The output generated on the screen can be seen on figure 1:

```
Time 1000, P1, Arrived
Time 1000, P1, Started, Granted 1000
Time 2000, P2, Arrived
Time 2000, P1, Paused
Time 2000, P1, Resumed, Granted 1000
Time 3000, P1, Paused
Time 3000, P1, priority updated to 100
Time 3000, P2, Started, Granted 100
Time 3100, P2, Terminated
Time 3100, P1, Resumed, Granted 200
Time 3200, P3, Arrived
Time 3300, P1, Paused
Time 3300, P1, Resumed, Granted 200
Time 3500, P1, Paused
Time 3500, P1, priority updated to 105
Time 3500, P3, Started, Granted 100
Time 3600, P3, Terminated
Time 3600, P1, Resumed, Granted 175
Time 3800, P1, Terminated
Press any key to continue . . .
```

Figure 1: Output given using 3 processes

The output of the first test resembles the one provided in the document coen346-project3_docx provided with the instructions. Indeed, some mistakes were found in the document concerning the priority update. The following example taken from the document contains the mistake highlighted in red.

```
Time 1000, P1, Started, Granted 1000
Time 2000, P1, Paused
Time 2000, P2, Arrived
Time 2000, P1, Resumed, Granted 1000
Time 3000, P1, Paused
Time 3000, P1, priority updated to 105
Time 3000, P2, Started, Granted 100
Time 3100, P2, Terminated
Time 3100, P1, Resumed, Granted 200
```

```
Time 3200, P3, Arrived
Time 3300, P1, Paused
Time 3300, P1, Resumed, Granted 200
Time 3500, P1, Paused
Time 3500, P1, priority updated to 110
Time 3500, P3, Started, Granted 100
Time 3600, P3, Terminated
Time 3600, P1, Resumed, Granted 150
Time 3700, P1, Terminated
```

For the first line highlighted in red, if priority is 105, with the formula given, the Tq granted should be Tq = (140-105)*5 = 175 and 175 != 200 as shown in the second red line.
With a priority of 100 (like found in our code) Tq granted is 200 which corresponds with the code shown above.

For the second test, the following code was used:

```
5
P 1 1000 2500 90
P 2 2000 100 120
P 3 3200 100 120
P 4 3000 200 70
P 5 3800 100 100
```

The output generated is shown in Figure 2:

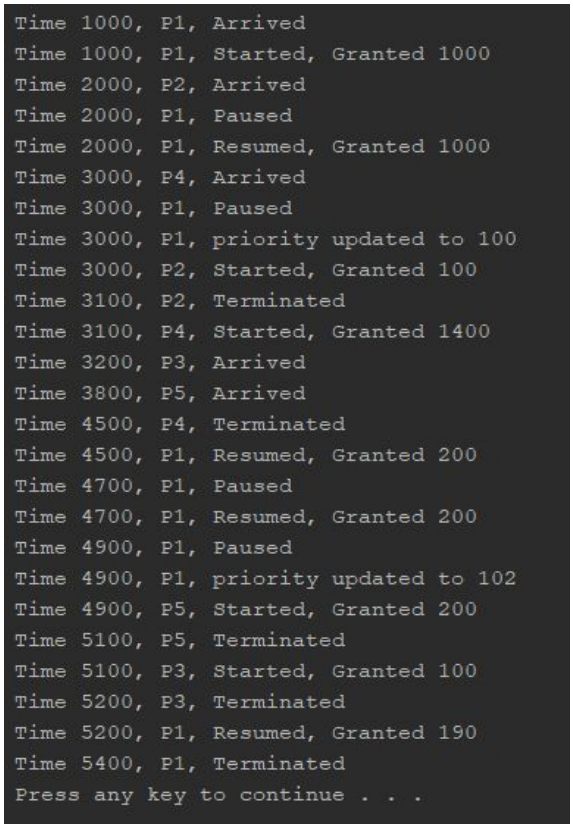

Figure 2: Output given using 5 processes

This second test proves that the program works with an arbitrary number of processes which is one of the requirements of the algorithm.

Conclusion

The implementation of the scheduler was a success. The program runs with user-defined cases as seen in the results. Any number of processes can be simulated using the scheduler. After being read from the inputs.txt file, the processes are sorted according to their arrival time inside a vector. This beforehand sorting enables the scheduler to insert the processes inside the queue according to their arrival time with O(1). Afterwards, the processes are places into a queue of size 140 (0 to 139) according to their priority. The scheduler is called using threads. The scheduler can PAUSE and RESUME processes, depending on the Tq granted. Only one process can run at a time. The use of mutex lock in the scheduler reinforce this requirement. Each process keeps track of its total execution by decreasing its private variable cpuBurst after every run. When the process terminates, the scheduler is informed using the keyword TERMINATED. That process is deleted using the destructor, to free the space allocated. After two consecutive runs, the priority of a process is updated using the formula provided. The program runs until all the processes terminate their CPU burst.The simulation is successful in terms of time and functionality.