**Darrell Maxwell** (n02776371)

**Operating Systems Project 1**, Fall 2015, SUNY New Paltz

**Professor:** Dr. Hanh Pham

**Due:** November 4th, 2015

**Algorithm:** Preemptive, Lowest Priority Scheduling

**Table of Contents:**

|  |  |
| --- | --- |
| **1** | Description of scheduling algorithm **(pg. 1)** |
| **2** | Description of my implementation **(pgs. 1 - 3)** |
| **3** | Experiments **(pgs. 3 - 6)** |
| **4** | Conclusion **(pg. 6)** |

1. **Preemptive, Lowest Priority Scheduling:**

In *preemptive scheduling*, a process that is running in the CPU may be swapped back in to the ready queue if the scheduling algorithm sees fit for another process to take its place. There are several criteria used by the scheduler to determine if a process should be preempted or not. In the case of *lowest priority* scheduling, a process will take the place of another process running in the CPU if its priority (a number assigned to a process indicating its importance) is lower than that of the currently executing process.

Example:

* Process #1 with a priority of 5 is executing instructions in the CPU.
* Process #2 is admitted to the ready queue with a priority of 2.
* The OS performs a context switch, saving Process #1’s state and loading Process #2 into the CPU.
* Process #2 has preempted Process #1.

1. **Implementation:**

For organizational purposes I have broken my program into four classes, each representing a component of my simulation. Below I will outline the functionality of these classes and describe their interactions and behavior:

**Simulation.java:**

The main purpose of this class is to provide an environment within which the other classes can interact. Additionally, this class processes input from the input file (input.data), and writes to the output file (output.data) when necessary. This class serves as an entry point for the entire application.

Methods:

* **main()** reads from input, initializes variables, and runs the simulation.
* **getSimTime()** returns the value of the counter variable “simTime”, which is used to keep track of the overall simulation time.
* **writeToOutput(String)** writes the given string to the output file.

**Scheduler.java:**

The Scheduler class maintains an ArrayList of Process objects (the “ready queue”) as well as having a reference to a CPU object. It can perform various actions on these processes such as increment their wait times, and schedule them to the CPU.

Methods:

* **incWaitTime()** increments the wait time for all processes sitting in the ready queue at each moment in the simulation.
* **admitProcess(Process)** admits the specified process into the ready queue, making sure that the head of the queue always points to the process having the lowest priority.
* **scheduleProcess()** schedules a process from the ready queue to the CPU unless the CPU is busy AND the currently executing process has a lower priority than the one in the ready queue.
* **readyQueueIsEmpty()** indicates if there are any processes in the ready queue.

**CPU.java:**

The CPU class contains a single Process object, which is said to be the currently executing process. The CPU decrements the remaining time for the current process as it executes. The current process can be set by the scheduler.

Methods:

* **isBusy()** returns true if there is currently a process in the CPU.
* **setProcess(Process)** sets the process to be executed.
* **getProcess()** returns the process currently executing.

**CPU.java** (continued)**:**

* **decProcTime()** decreases the time remaining for the current process and removes it from the CPU when it has finished executing (writing to output).

**Process.java:**

An object of this class represents a single process in the simulation. Several pieces of important information are held by each process, such as: process number, priority, start time, end time, remaining time, response time, and wait time.

Methods:

* **increaseWaitTime()** used to increment the wait time for waiting processes.
* **decreaseRemainingTime()** used to decrement the remaining time for running processes.

The program itself displays useful information to standard output as it runs such as: the occurrence of errors, information pertaining to I/O requests, the states of several important variables, and scheduling events in the order they occur (admission, scheduling, preemption, completion).

1. **Experiments:**

**Experiment 1:**

1. Input Data

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| input.data |  | Proc. # | Arrival | Burst | Priority |
| 5 | 1 | 3 | 12 | 4 |
| 1 1 | 2 | 0 | 4 | 5 |
| 3 12 4 | 3 | 0 | 6 | 3 |
| 0 4 5 | 4 | 7 | 8 | 4 |
| 0 6 3 | 5 | 5 | 3 | 1 |
| 7 8 4 |
| 5 3 1 |

1. Gantt Chart



1. Waiting Time and Response Time

**Avg. waiting time** = (P1(9-3) + P2(29-0) + P3((0-0)+(8-5)) + P4(21-7) + P5(5-5)) / 5 = **10.4**

**Avg. response time =** (P1(9-3) + P2(29-0) + P3(0-0) + P4(21-7) + P5(5-5)) / 5 = **9.8**

1. Output Data

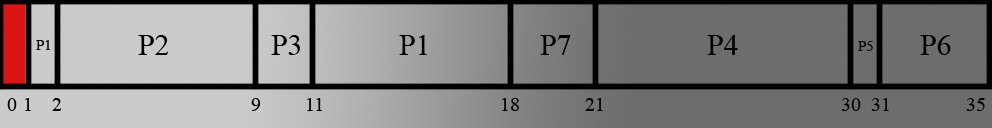
|  |  |
| --- | --- |
| **Expectation:** output.data | **Result:** output.data |
| 5 8 5 | 5 8 5 |
| 0 9 3 | 0 9 3 |
| 9 21 1 | 9 21 1 |
| 21 29 4 | 21 29 4 |
| 29 33 2 | 29 33 2 |

**Experiment 2:**

1. Input Data

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| input.data |  | Proc. # | Arrival | Burst | Priority |
| 7 | 1 | 1 | 8 | 3 |
| 1 1 | 2 | 2 | 7 | 1 |
| 1 8 3 | 3 | 3 | 2 | 1 |
| 2 7 1 | 4 | 3 | 9 | 5 |
| 3 2 1 | 5 | 4 | 1 | 6 |
| 3 9 5 |  | 6 | 5 | 4 | 6 |
| 4 1 6 |  | 7 | 6 | 3 | 4 |
| 5 4 6 |
| 6 3 4 |

1. Gantt Chart



1. Waiting Time and Response Time

**Avg. waiting time** = (P1((1-1)+(11-2)) + P2(2-2) + P3(9-3) + P4(21-3) + P5(30-4) + P6(31-5) + P7(18-6)) / 7 = **13.86**

**Avg. response time =** (P1(0-0) + P2(2-2) + P3(9-3) + P4(21-3) + P5(30-4) + P6(31-5) + P7(18-6)) / 7 = **12.57**

1. Output Data

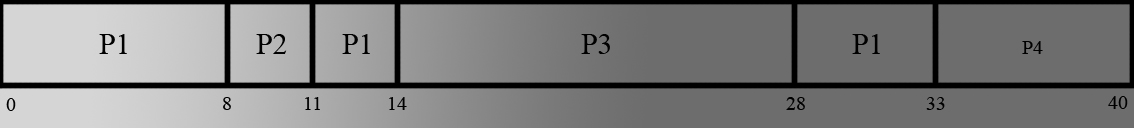
|  |  |
| --- | --- |
| **Expectation:** output.data | **Result:** output.data |
| 2 9 2 | 2 9 2 |
| 9 11 3 | 9 11 3 |
| 1 18 1 | 1 18 1 |
| 18 21 7 | 18 21 7 |
| 21 30 4 | 21 30 4 |
| 30 31 5 | 30 31 5 |
| 31 35 6 | 31 35 6 |

**Experiment 3:**

1. Input Data

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| input.data |  | Proc. # | Arrival | Burst | Priority |
| 4 | 1 | 0 | 16 | 3 |
| 1 1 | 2 | 8 | 3 | 1 |
| 0 16 3 | 3 | 14 | 14 | 2 |
| 8 3 1 | 4 | 15 | 7 | 4 |
| 14 14 2 |
| 15 7 4 |

1. Gantt Chart



1. Waiting Time and Response Time

**Avg. waiting time** = (P1((0-0)+(11-8)+(28-14)) + P2(8-8) + P3(14-14) + P4(33-15)) / 4 = **8.75**

**Avg. response time =** (P1(0-0) + P2(8-8) + P3(14-14) + P4(33-15)) / 4 = **4.5**

1. Output Data

|  |  |
| --- | --- |
| **Expectation:** output.data | **Result:** output.data |
| 18 11 2 | 18 11 2 |
| 14 28 3 | 14 28 3 |
| 0 33 1 | 0 33 1 |
| 33 40 4 | 1. 0 4 |

1. **Conclusion:**

My implementation of Preemptive, Lowest Priority Scheduling was successful for all three test cases. Priority scheduling ensures that important tasks are the first to be completed, but on the other a process that might be otherwise very quick to execute might sit in the ready queue for a relatively long time. For example, Process #5 in Experiment 2 has an execution time of just 1, yet it must wait for nearly all other processes to execute due to its high priority number.