CPU Scheduler Simulation Report:
First Come First Serve, Shortest Job First and Multilevel Feedback Queue

Gavin Wolf
Computer Operating Systems - COP 4610
Professor: Dr. Borko Furht
October 22, 2015

Table of Contents

Page

I. Introduction

One of the key responsibilities of an operating system is managing a computer's resources, including its CPU. A single process typically alternates between requiring CPU time and requiring input or output. Such a process cannot keep the CPU busy at all times. An operating system can increase CPU utilization by switching the CPU among multiple processes. The CPU scheduler is the part of the operating system that chooses which ready process will get CPU time next. The CPU scheduler is programmed to make decisions based on a particular scheduling algorithm. In this project, three such algorithms are evaluated by running simulations of their operation on a set of processes.

The three algorithms and their specifications are:

1) First Come First Serve (non-preemptive)
2) Shortest Job First (non-preemptive)
3) Multilevel Feedback Queue (preemptive - absolute priority in higher queues)
    - Queue 1 uses Round Robin scheduling with Time Quantum (Tq) of 6
    - Queue 2 uses Round Robin scheduling with Time Quantum (Tq) of 11
    - Queue 3 uses First Come First Serve
    - All processes enter Queue 1. If Tq expires before CPU burst is complete, the process is downgraded to next lower priority queue. Processes are not downgraded when preempted by a higher queue level process. Once a process has been downgraded, it will not be upgraded.

I will evaluate the algorithms according to the following metrics:

1) CPU utilization - time CPU in use / total time of simulation
2) Total time - this corresponds to throughput, which is completed processes per unit of time; for this simulation each algorithm uses the same exact processes, so total time is a proxy for throughput
3) Waiting times - the total time a process spends in the ready queue, waiting for CPU time
4) Turnaround times - the time from arrival to completion; in this simulation it is assumed that all processes arrive at time zero, so the turnaround time is equal to the time of completion
5) Response times - the time from arrival to the first response

The goal of any CPU scheduling algorithm is to maximize CPU utilization, and to minimize total time, waiting times, turnaround times and response times.

II. Logic of Simulation Program

First, I will describe the general methodology for the simulation program, then I will explain the logic for each scheduling algorithm.

General Methodology

- Increment the current execution time by one time unit until all processes are completed.
- After each increment of time, adjust all relevant data and move processes between the CPU, Ready Queue, IO List, and Completed List as appropriate.
- Output the state of the simulation at every context switch, i.e. every time the CPU changes between processes or between idle and a process. Output shows where in the system each process is at the given time.

First Come First Serve and Shortest Job First

Since there are only two additional steps in logic for Shortest Job First compared to First Come First Serve, I will include the Shortest Job First steps in *italics* in the steps below.

Steps in logic:

- Add all the processes to the ready queue
    - *For Shortest Job First, sort the ready queue by shortest next CPU burst*
- REPEAT THE BELOW STEPS UNTIL ALL PROCESSES ARE COMPLETED
- Increment current execution time
- Decrement remaining burst time of process in CPU (if any)
- Decrement remaining burst times of all processes in IO (if any)
- If there are processes in IO that just completed their bursts, move them to the ready queue
    - *For Shortest Job First, insert the Process in order based on next CPU burst*
- If there is no process is in the CPU
    - Increment idle time
    - If there is a process in the ready queue, move it to the CPU
- Else, there is a process in the CPU
    - If the process's CPU burst was just completed
        - If the process is completed, move it to the completed list
        - Else, move the process to IO
        - If there is a process in the ready queue, move it to the CPU

Multilevel Feedback Queue

Note: in the following steps in logic, I will refer to the three levels of the "multilevel" ready queue as RQ1, RQ2 and RQ3

Steps in logic:

- Add all the processes to the ready queue
- REPEAT THE BELOW STEPS UNTIL ALL PROCESSES ARE COMPLETED
- Increment current execution time

- Decrement remaining burst time of process in CPU (if any)
- Decrement remaining burst times of all processes in IO (if any)
- If there are processes in IO that just completed their bursts, move them to the RQ that they were sent to IO from
- If there is no process is in the CPU
    - Increment idle time
    - If there is a process in any level of the ready queue, move the highest-priority process to the CPU
- Else, there is a process in the CPU
    - *Case 1: Burst is complete*
        - If the process is completed, move it to the completed list
        - Else, move the process to IO
        - If there is a process in any level of the ready queue, move the highest-priority process to the CPU
    - *Case 2A: Running process came from RQ2 and another process was just added to RQ1 --> preemption!*
        - Take running process out of CPU and add it to the back of RQ2
        - Move the process that was just added to RQ1 to the CPU
    - *Case 2B: Running process came from RQ3 and another process was just added to RQ1 or RQ2 --> preemption!*
        - Take running process out of CPU and return it to the front of RQ3
        - Move the process that was just added to RQ1 or RQ2 to the CPU
    - *Case 3: Time Quantum for the running process expired*
        - If process came from RQ1 and its time slice has expired, move the process to RQ2
        - If process came from RQ2 and its time slice has expired, move the process to RQ3

III. Final Results and Discussion

This section includes the following three tables and a discussion of the data:

- Table 1: Process Data - CPU & I/O Bursts
  - This table shows the input to the simulation, the eight processes and their corresponding CPU and I/O bursts in order from left to right
- Table 2: Simulation Statistics - Overall
  - This table shows the average five key metrics for each scheduling algorithm and ranks each algorithm on each metric with 1 being the best and 3 being the worst
- Table 3: Simulation Statistics - By Process
  - This table shows the five key metrics for each individual process

Table 1: Process Data - CPU & I/O Bursts

	CPU Burst	I/O Burst	CPU Burst	I/O Burst	CPU Burst	I/O Burst	CPU Burst	I/O Burst	CPU Burst	I/O Burst	CPU Burst	I/O Burst	CPU Burst	I/O Burst	CPU Burst	I/O Burst	CPU Burst	I/O Burst	CPU Burst	Total CPU Bursts	Total I/O Bursts	CPU Bursts / I/O Bursts
P1	4	24	5	73	3	31	5	27	4	33	6	43	4	64	5	19	2			38	314	12.1%
P2	18	31	19	35	11	42	18	43	19	47	18	43	17	51	19	32	10			149	324	46.0%
P3	6	18	4	21	7	19	4	16	5	29	7	21	8	22	6	24	5			52	170	30.6%
P4	17	42	19	55	20	54	17	52	15	67	12	72	15	66	14					129	408	31.6%
P5	5	81	4	82	5	71	3	61	5	62	4	51	3	77	4	61	3	42	5	41	588	7.0%
P6	10	35	12	41	14	33	11	32	15	41	13	29	11							86	211	40.8%
P7	21	51	23	53	24	61	22	31	21	43	20									131	239	54.8%
P8	11	52	14	42	15	31	17	21	16	43	12	31	13	32	15					113	252	44.8%
Total																				**739**	**2,506**	**29.5%**

Table 2: Simulation Statistics - Overall

	Shortest Job First		First Come First Serve		Multilevel Feedback Queue	
	Value	**Rank**	**Value**	**Rank**	**Value**	**Rank**
CPU utilization	86.64%	1	82.02%	3	86.53%	2
Total Time	853	1	901	3	854	2
Average Wait Time (WT)	169.13	1	285.88	3	189.00	2
Average Turnaround Time (TT)	574.75	1	691.50	3	594.63	2
Average Response Time (RT)	79.63	3	36.25	2	18.88	1

Table 3: Simulation Statistics - By Process

	Shortest Job First *(Total Time: 853)* *(CPU utilization: 86.64%)*			**First Come First Serve** *(Total Time: 901)* *(CPU utilization: 82.02%)*			**Multilevel Feedback Queue** *(Total Time: 854)* *(CPU utilization: 86.53%)*		
	Waiting Time	**Turnaround Time**	**Response Time**	**Waiting Time**	**Turnaround Time**	**Response Time**	**Waiting Time**	**Turnaround Time**	**Response Time**
Process 1	53	405	0	318	670	0	17	369	0
Process 2	231	704	81	283	756	4	286	759	4
Process 3	51	273	9	341	563	22	78	300	10
Process 4	232	769	45	209	746	28	317	854	16
Process 5	65	694	4	272	901	45	22	651	22
Process 6	106	403	15	313	610	50	163	460	27
Process 7	483	853	458	229	599	60	328	698	33
Process 8	132	497	25	322	687	81	301	666	39
Average	**169.13**	**574.75**	**79.63**	**285.88**	**691.50**	**36.25**	**189.00**	**594.63**	**18.88**

As mentioned before, the goal of any CPU scheduling algorithm is to maximize CPU utilization, and to minimize total time, waiting times, turnaround times and response times.

The Shortest Job First algorithm ranks best on four out of five of the metrics. It has the shortest total time of 853, meaning that the algorithm completes all the jobs more quickly than the other two algorithms. With regards to average wait time, Shortest Job First is mathematically optimal and this is evidenced by this simulation. Because the wait time for a process depends on the length of the burst times of the processes ahead of it, by running the processes with the shortest burst time first, the wait time of the processes behind it is minimized. One potential drawback of Shortest Job First is starvation, which means that processes with long CPU bursts may be blocked indefinitely from getting CPU time because such processes will not run until all processes with shorter bursts have been run first. This drawback can be seen in the above tables, where Shortest Job First has the longest average response time. Process 7, in particular, has relatively long CPU burst times and is starved for 458 time units until it gets its first time in the CPU. Both the response time and the average waiting time for Process 7 are significantly higher in Shortest Job First than they are in the other scheduling algorithms.

The Multilevel Feedback Queue algorithm ranks second on four out of the five metrics. It is better than First Come First Serve on every metric. Multilevel Feedback Queue ranks number one on average response time. This makes sense intuitively because the Time Quantum in the round robin queues ensures that the maximum amount of time a process can wait in a particular queue is the Time Quantum times the number of processes ahead of the process. Processes with bursts of six (the Time Quantum of the first level of the ready queue) or less are given highest priority. Such processes get to the CPU quickly because they do not have to wait for processes with longer bursts to finish. Processes with bursts of seventeen (the Time Quantum of the first level of the ready queue (six), plus the Time Quantum of the second level of the ready queue (eleven)) or less will also get to the CPU relatively quickly. Processes with bursts longer than seventeen will fall to the third level of the ready queue and will only get to the CPU again when there are no processes in the first and second levels of the ready queue. This phenomenon can be observed with Process 7. Process 7's first CPU burst twenty-one, so Process 7 will quickly fall to the third level of the ready queue, where it will wait to run until the first and second levels of the ready queue are empty. It is therefore not surprising that Process 7 has the longest waiting time. Also, all the CPU bursts for Process 1 and Process 5 are six or less, so it is not surprising that Process 1 and Process 5 have the shortest waiting times.

The results for the Multilevel Feedback Queue algorithm very nearly approximate those of the Shortest Job First algorithm. As noted before, the Multilevel Feedback Queue algorithm gives highest priority to processes with CPU bursts of seventeen or less. In this simulation, all the CPU bursts for five of the eight processes (P1, P3, P5, P6 and P8) are seventeen or less. As a result, these five processes are given higher priority over the other three processes. That the results for the Multilevel Feedback Queue algorithm very nearly approximate those of the Shortest Job First algorithm shows that the time quanta used in the round robin queues were chosen well. In other words, the Multilevel Feedback Queue algorithm does a good job of prioritizing the processes with the shortest CPU burst times. If the time quanta were too large, then round robin would behave like First Come First Serve. In this simulation, if the time quanta were too small, then round robin would also behave like First Come First Serve because all the processes would fall to the third level of the ready queue, which is scheduled based on first come first serve.

The First Come First serve algorithm ranks last on four out of five metrics. It has by far the worst average wait time at 285.88 compared to 169.13 and 189.00 for Shortest Job First and Multilevel Feedback Queue, respectively. This result makes sense because First Come First serve does not optimize the wait time like Shortest Job First, nor does it speed through processes with relatively short bursts. First Come First Serve causes very long waiting times for I/O-bound processes, because although such processes only need a short amount of CPU time, they end up waiting behind processes that take a lot of time in the CPU. For example, Process 5 is the most I/O-bound process (in that it has the lowest ratio of total CPU burst time to total I/O burst time) and in the First Come First Serve simulation it has a substantially longer waiting time than it does in Shortest Job First or Multilevel Feedback Queue (272 vs. 65 and 22, respectively). And conversely, Process 7 is the most CPU-bound process and its waiting time in First Come First Serve is substantially lower in the First Come First Serve simulation than it is in Shortest Job First or Multilevel Feedback Queue (229 vs. 483 and 328, respectively)

IV. Sample of Dynamic Execution (Program Output)

For the sample program output for, I selected samples that highlight key events for each algorithm's execution. With each sample section, I provide some commentary to help follow these key events.

<u>*First Come First Serve - Sample of Dynamic Execution*</u>

Simulation Start + 2 Context Switches

Notice P1 finishing its CPU burst at time 4 and moving to I/O, and P2 finishing its CPU burst at time 22 and moving to I/O.

```
:::::::::::::::::::::::::::::::::::::::::::::::::
Current time: 0
Now running:  P1 (4 time units remaining)
.................................................
Ready Queue:  Process    Burst
              P2         18
              P3         6
              P4         17
              P5         5
              P6         10
              P7         21
              P8         11
.................................................
In I/O:       Process    Remaining I/O Time
              [empty]
.................................................
Completed:    [empty]
:::::::::::::::::::::::::::::::::::::::::::::::::
:::::::::::::::::::::::::::::::::::::::::::::::::
Current time: 4
Now running:  P2 (18 time units remaining)
.................................................
Ready Queue:  Process    Burst
              P3         6
              P4         17
              P5         5
              P6         10
              P7         21
              P8         11
.................................................
In I/O:       Process    Remaining I/O Time
              P1         24
.................................................
Completed:    [empty]
:::::::::::::::::::::::::::::::::::::::::::::::::
Current time: 22
Now running:  P3 (6 time units remaining)
.................................................
Ready Queue:  Process    Burst
```

```
                P4          17
                P5          5
                P6          10
                P7          21
                P8          11
..................................................
In I/O:         Process     Remaining I/O Time
                P1          6
                P2          31
..................................................
Completed:      [empty]
::::::::::::::::::::::::::::::::::::::::::::::::::
```

CPU Going from Running to Idle and Back to Running - 3 Context Switches

Notice at time 644, P1 is running and has 5 time units remaining. The Ready Queue is empty and the shortest remaining I/O burst is 7 time units. At time 649, P1 finishes and there are no ready processes so the CPU goes idle. At 651, P4 finishes its I/O burst and moves to the CPU.

```
::::::::::::::::::::::::::::::::::::::::::::::::::
Current time: 644
Now running:    P1 (5 time units remaining)
..................................................
Ready Queue:    Process     Burst
                [empty]
..................................................
In I/O:         Process     Remaining I/O Time
                P4          7
                P8          28
                P2          34
                P5          51
..................................................
Completed:      P3   P6   P7
::::::::::::::::::::::::::::::::::::::::::::::::::
Current time: 649
Now running:    [idle]
..................................................
Ready Queue:    Process     Burst
                [empty]
..................................................
In I/O:         Process     Remaining I/O Time
                P4          2
                P1          19
                P8          23
                P2          29
                P5          46
..................................................
Completed:      P3   P6   P7
::::::::::::::::::::::::::::::::::::::::::::::::::
Current time: 651
Now running:    P4 (15 time units remaining)
..................................................
Ready Queue:    Process     Burst
```

```
               [empty]
.....................................................
In I/O:        Process     Remaining I/O Time
               P1          17
               P8          21
               P2          27
               P5          44
.....................................................
Completed:     P3  P6  P7
:::::::::::::::::::::::::::::::::::::::::::::::::::::
```

Shortest Job First - Sample of Dynamic Execution

Simulation Start

Notice that the Ready Queue is ordered by shortest next CPU burst.

```
::::::::::::::::::::::::::::::::::::::::::::::::::
Current time: 0
Now running:  P1 (4 time units remaining)
..................................................
Ready Queue:  Process    Burst
              P5         5
              P3         6
              P6         10
              P8         11
              P4         17
              P2         18
              P7         21
..................................................
In I/O:       Process    Remaining I/O Time
              [empty]
..................................................
Completed:    [empty]
::::::::::::::::::::::::::::::::::::::::::::::::::
```

Process Moving from I/O to front of Ready Queue Because it has the Shortest Next CPU Burst

Let's trace P6 through two context switches. At time 248, P6 is in I/O. We can see at time 265, P6 has moved to the front of the Ready Queue and has the shortest next CPU burst of all the jobs in the Ready Queue. Notice, however, that the process that was just switched into the CPU at time 265 is P5 and has a CPU burst of 3, shorter than the next CPU burst of P6, which is 15.

```
::::::::::::::::::::::::::::::::::::::::::::::::::
Current time: 248
Now running:  P8 (17 time units remaining)
..................................................
Ready Queue:  Process    Burst
              P4         20
              P7         21
..................................................
In I/O:       Process    Remaining I/O Time
              P6         1
              P5         17
              P3         18
              P2         23
              P1         43
..................................................
Completed:    [empty]
::::::::::::::::::::::::::::::::::::::::::::::::::
Current time: 265
```

```
Now running:   P5 (3 time units remaining)
..................................................
Ready Queue:   Process     Burst
               P6          15
               P4          20
               P7          21
..................................................
In I/O:        Process     Remaining I/O Time
               P3          1
               P2          6
               P8          21
               P1          26
..................................................
Completed:     [empty]
::::::::::::::::::::::::::::::::::::::::::::::::::
```

One Process Completes, Another Moves from I/O to the Ready Queue

Notice at time 268, P3 has 5 time units remaining in CPU and P2 is in I/O with a remaining I/O burst of 3. At time 273, P3 finishes its CPU burst, which was its last burst, and moves to Completed. Also, P2 has finished its I/O and moved straight into the CPU because its next CPU burst is shorter than the next CPU bursts of the processes in the Ready Queue.

```
::::::::::::::::::::::::::::::::::::::::::::::::::
Current time: 268
Now running:   P3 (5 time units remaining)
..................................................
Ready Queue:   Process     Burst
               P6          15
               P4          20
               P7          21
..................................................
In I/O:        Process     Remaining I/O Time
               P2          3
               P8          18
               P1          23
               P5          61
..................................................
Completed:     [empty]
::::::::::::::::::::::::::::::::::::::::::::::::::
Current time: 273
Now running:   P2 (11 time units remaining)
..................................................
Ready Queue:   Process     Burst
               P6          15
               P4          20
               P7          21
..................................................
In I/O:        Process     Remaining I/O Time
               P8          13
               P1          18
               P5          56
```

```
..................................................
Completed:    P3
::::::::::::::::::::::::::::::::::::::::::::::::::
```

Multilevel Feedback Queue - Sample of Dynamic Execution

Simulation Start + Two Context Switches

Notice that P2 starts at time 4 and has a burst of 18 time units. P2's Time Quantum of 6 is over at time 10, when P2 is moved to Ready Queue 2 and has 12 time units remaining.

```
::::::::::::::::::::::::::::::::::::::::::::::::::
Current time: 0
Now running:  P1 (4 time units remaining, Level = 1)
..................................................
Ready Queue1: Process    Burst
              P2         18
              P3         6
              P4         17
              P5         5
              P6         10
              P7         21
              P8         11
..................................................
Ready Queue2: Process    Burst
              [empty]
..................................................
Ready Queue3: Process    Burst
              [empty]
..................................................
In I/O:       Process    Remaining I/O Time
              [empty]
..................................................
Completed:    [empty]
::::::::::::::::::::::::::::::::::::::::::::::::::
Current time: 4
Now running:  P2 (18 time units remaining, Level = 1)
..................................................
Ready Queue1: Process    Burst
              P3         6
              P4         17
              P5         5
              P6         10
              P7         21
              P8         11
..................................................
Ready Queue2: Process    Burst
              [empty]
..................................................
Ready Queue3: Process    Burst
              [empty]
..................................................
In I/O:       Process    Remaining I/O Time
              P1         24
..................................................
Completed:    [empty]
```

```
::::::::::::::::::::::::::::::::::::::::::::::::::::
Current time: 10
Now running:  P3 (6 time units remaining, Level = 1)
....................................................
Ready Queue1: Process    Burst
              P4         17
              P5         5
              P6         10
              P7         21
              P8         11
....................................................
Ready Queue2: Process    Burst
              P2         12
....................................................
Ready Queue3: Process    Burst
              [empty]
....................................................
In I/O:       Process    Remaining I/O Time
              P1         18
....................................................
Completed:    [empty]
::::::::::::::::::::::::::::::::::::::::::::::::::::
```

Example of Running Process Getting Preempted by a Higher Priority Process Entering The Ready Queue

Let's trace P4 through the following eight context switches. At time 54, P4 is at the front of Ready Queue 2 and has a next burst of 11. At time 65, P4 moves to the CPU, where it gets a Time Quantum of 11 times units. Also notice that P3 is in I/O with 10 time units remaining. At time 75, P3 enters Ready Queue 1 and preempts P4. P4 is sent back to Ready Queue 2 and its remaining burst is 1. At time 101, P4 moves back to the CPU. At time 102, P4 finishes its burst and moves to I/O.

```
::::::::::::::::::::::::::::::::::::::::::::::::::::
Current time: 54
Now running:  P2 (12 time units remaining, Level = 2)
....................................................
Ready Queue1: Process    Burst
              [empty]
....................................................
Ready Queue2: Process    Burst
              P4         11
              P6         4
              P7         15
              P8         5
....................................................
Ready Queue3: Process    Burst
              [empty]
....................................................
In I/O:       Process    Remaining I/O Time
              P3         21
              P5         54
```

```
               P1          69
..................................................
Completed:     [empty]
::::::::::::::::::::::::::::::::::::::::::::::::::
Current time: 65
Now running:   P4 (11 time units remaining, Level = 2)
..................................................
Ready Queue1: Process     Burst
               [empty]
..................................................
Ready Queue2: Process     Burst
               P6          4
               P7          15
               P8          5
..................................................
Ready Queue3: Process     Burst
               P2          1
..................................................
In I/O:        Process     Remaining I/O Time
               P3          10
               P5          43
               P1          58
..................................................
Completed:     [empty]
::::::::::::::::::::::::::::::::::::::::::::::::::
Current time: 75
Now running:   P3 (7 time units remaining, Level = 1)
..................................................
Ready Queue1: Process     Burst
               [empty]
..................................................
Ready Queue2: Process     Burst
               P6          4
               P7          15
               P8          5
               P4          1
..................................................
Ready Queue3: Process     Burst
               P2          1
..................................................
In I/O:        Process     Remaining I/O Time
               P5          33
               P1          48
..................................................
Completed:     [empty]
::::::::::::::::::::::::::::::::::::::::::::::::::
Current time: 81
Now running:   P6 (4 time units remaining, Level = 2)
..................................................
Ready Queue1: Process     Burst
               [empty]
..................................................
Ready Queue2: Process     Burst
```

```
              P7          15
              P8          5
              P4          1
              P3          1
..................................................
Ready Queue3: Process     Burst
              P2          1
..................................................
In I/O:       Process     Remaining I/O Time
              P5          27
              P1          42
..................................................
Completed:    [empty]
::::::::::::::::::::::::::::::::::::::::::::::::::
Current time: 85
Now running:  P7 (15 time units remaining, Level = 2)
..................................................
Ready Queue1: Process     Burst
              [empty]
..................................................
Ready Queue2: Process     Burst
              P8          5
              P4          1
              P3          1
..................................................
Ready Queue3: Process     Burst
              P2          1
..................................................
In I/O:       Process     Remaining I/O Time
              P5          23
              P6          35
              P1          38
..................................................
Completed:    [empty]
::::::::::::::::::::::::::::::::::::::::::::::::::
Current time: 96
Now running:  P8 (5 time units remaining, Level = 2)
..................................................
Ready Queue1: Process     Burst
              [empty]
..................................................
Ready Queue2: Process     Burst
              P4          1
              P3          1
..................................................
Ready Queue3: Process     Burst
              P2          1
              P7          4
..................................................
In I/O:       Process     Remaining I/O Time
              P5          12
              P6          24
              P1          27
```

```
..................................................
Completed:     [empty]
::::::::::::::::::::::::::::::::::::::::::::::::::
Current time: 101
Now running:  P4 (1 time units remaining, Level = 2)
..................................................
Ready Queue1: Process    Burst
              [empty]
..................................................
Ready Queue2: Process    Burst
              P3         1
..................................................
Ready Queue3: Process    Burst
              P2         1
              P7         4
..................................................
In I/O:       Process    Remaining I/O Time
              P5         7
              P6         19
              P1         22
              P8         52
..................................................
Completed:     [empty]
::::::::::::::::::::::::::::::::::::::::::::::::::
Current time: 102
Now running:  P3 (1 time units remaining, Level = 2)
..................................................
Ready Queue1: Process    Burst
              [empty]
..................................................
Ready Queue2: Process    Burst
              [empty]
..................................................
Ready Queue3: Process    Burst
              P2         1
              P7         4
..................................................
In I/O:       Process    Remaining I/O Time
              P5         6
              P6         18
              P1         21
              P4         42
              P8         51
..................................................
Completed:     [empty]
::::::::::::::::::::::::::::::::::::::::::::::::::
```

V. - Printed End-of-Simulation Results

First Come First Serve

```
Total Time:          901
CPU Utilization:     82.02%

Waiting Times        P1   P2   P3   P4   P5   P6   P7   P8
                     318  283  341  209  272  313  229  322
Average Wait:        285.88

Turnaround Times     P1   P2   P3   P4   P5   P6   P7   P8
                     670  756  563  746  901  610  599  687
Average Turnaround: 691.50

Response Times       P1   P2   P3   P4   P5   P6   P7   P8
                     0    4    22   28   45   50   60   81
Average Response:    36.25
```

Shortest Job First

```
Total Time:          853
CPU Utilization:     86.64%

Waiting Times        P1   P2   P3   P4   P5   P6   P7   P8
                     53   231  51   232  65   106  483  132
Average Wait:        169.13

Turnaround Times     P1   P2   P3   P4   P5   P6   P7   P8
                     405  704  273  769  694  403  853  497
Average Turnaround: 574.75

Response Times       P1   P2   P3   P4   P5   P6   P7   P8
                     0    81   9    45   4    15   458  25
Average Response:    79.63
```

Multilevel Feedback Queue

```
Total Time:          854
CPU Utilization:     86.53%

Waiting Times        P1   P2   P3   P4   P5   P6   P7   P8
                     17   286  78   317  22   163  328  301
Average Wait:        189.00

Turnaround Times     P1   P2   P3   P4   P5   P6   P7   P8
                     369  759  300  854  651  460  698  666
Average Turnaround: 594.63

Response Times       P1   P2   P3   P4   P5   P6   P7   P8
                     0    4    10   16   22   27   33   39
Average Response:    18.88
```

Appendix A - Source Code

[Starts on next page]

```cpp
/**********************************************************************************************
Name: Gavin Wolf        Z#: 15289719
Course: Computer Operating Systems - COP 4610
Professor: Dr. Borko Furht
Due Date: 10/22/2015
Programming Assignment - CPU Scheduler

Description: This program simulates CPU scheduling algorithms on a set of processes with
 specified CPU and I/O burst times.
**********************************************************************************************/

/**********************************************************************************************
**********************************************************************************************

Name: Simulation.cpp
Description: The general methodology for this simulation is as follows:
    - Run a while loop that exits when all Processes are completed
    - In the body of the while loop, increment the current execution time and make all
      necessary changes to the system, while keeping track of all relevant data
    - Output the state of the simulation at every context switch, i.e. every time the CPU
      changes between Processes or between idle and a Process. Output shows where in the
      system each Process is at the given time

**********************************************************************************************
**********************************************************************************************/

#include <iostream>
#include <vector>
#include <string>
#include <iomanip>

#include "Process.h"
#include "ReadyQueue.h"
#include "IO.h"
#include "Completed.h"

using namespace std;

//Prototypes of simulation functions
void SimulateSingleReadyQueue(bool isSJF);
void SimulateMultiReadyQueue();

/**********************************************************************************************
Name: main
Description: Prompts user to select which of the three algorithms to run and then calls the
 selected algorithm.
**********************************************************************************************/
int main()
{
    input:
    cout << "Enter the number corresponding to the scheduling algorithm you would like to run:\n\n"
         << "1 - First Come First Serve (non-preemptive)\n"
         << "2 - Shortest Job First (non-preemptive)\n"
         << "3 - Multilevel Feedback Queue (preemptive - absolute priority in higher queues)\n"
         << "    Queue 1 uses Round Robin scheduling with Time Quantum of 6\n"
         << "    Queue 2 uses Round Robin scheduling with Time Quantum of 11\n"
```

```
         << "    Queue 3 uses First Come First Serve\n\n";

    int algorithm;
    cin >> algorithm;

    if (algorithm == 1)
        SimulateSingleReadyQueue(false);
    else if (algorithm == 2)
        SimulateSingleReadyQueue(true);
    else if (algorithm == 3)
        SimulateMultiReadyQueue();
    else
        goto input;

    return 0;
}

//SETUP/OVERHEAD FOR SIMULATION FUNCTIONS:

//  Declaration of variables to track time
    int currentTime = 0; //current execution time
    int idleTime = 0; //time that no Process is in CPU

//  Prototypes of "helper" functions
    void AddProcessToCPU(Process & P, int currentTime);
    void SingleDisplay();
    void MultiDisplay();
    void MoveRQtoCPU(); //function to move a job from RQ to CPU

//  Creation of Process objects for each process with burst times ordered as follows: CPU, I/O, CPU, I/O, ... , CPU
    Process P1("P1", vector<int> { 4, 24, 5, 73, 3, 31, 5, 27, 4, 33, 6, 43, 4, 64, 5, 19, 2 });
    Process P2("P2", vector<int> { 18, 31, 19, 35, 11, 42, 18, 43, 19, 47, 18, 43, 17, 51, 19, 32, 10 });
    Process P3("P3", vector<int> { 6, 18, 4, 21, 7, 19, 4, 16, 5, 29, 7, 21, 8, 22, 6, 24, 5 });
    Process P4("P4", vector<int> { 17, 42, 19, 55, 20, 54, 17, 52, 15, 67, 12, 72, 15, 66, 14 });
    Process P5("P5", vector<int> { 5, 81, 4, 82, 5, 71, 3, 61, 5, 62, 4, 51, 3, 77, 4, 61, 3, 42, 5 });
    Process P6("P6", vector<int> { 10, 35, 12, 41, 14, 33, 11, 32, 15, 41, 13, 29, 11 });
    Process P7("P7", vector<int> { 21, 51, 23, 53, 24, 61, 22, 31, 21, 43, 20 });
    Process P8("P8", vector<int> { 11, 52, 14, 42, 15, 31, 17, 21, 16, 43, 12, 31, 13, 32, 15 });

//  Initialization of a pointer to a Process object called "ProcessInCPU". I use this variable
//   to model the Process that is currently getting CPU time
    Process * ProcessInCPU = 0;

//  Creation of Lists for First Come First Serve and Shortest Job First algorithms.
//   Because there is only one ready queue needed for these algorithms, I named the objects
//   "Single" followed by the type of list
    ReadyQueue SingleRQ; //ReadyQueue object
    IO SingleIO; //IO object
    Completed SingleCompleted; //Completed object

//  Creation of Lists for Multilevel Feedback Queue algorithm.
//   Because multiple ready queues are needed for this algorithm, I named the objects
//   "Multi" followed by the type of the list
    ReadyQueue MultiRQ1; //ReadyQueue object: level 1 of ready queue - Round Robin, Tq = 6
```

```
    ReadyQueue MultiRQ2; //ReadyQueue object: level 2 of ready queue - Round Robin, Tq = 11
    ReadyQueue MultiRQ3; //ReadyQueue object: level 3 of ready queue - First Come First Serve
    IO MultiIO; //IO object
    Completed MultiCompleted; //Completed bbject

/*********************************************************************************************
Name: SimulateSingleReadyQueue
Description: Simulates an algorithm requiring a "single" ready queue. Accepts a boolean
 parameter "isSJF". When isSJF is true, the algorithm simulated is Shortest Job First. When
 isSJF is false, the algorithm simulated is First Come First Serve
*********************************************************************************************/
void SimulateSingleReadyQueue(bool isSJF)
{
    //Add all Process objects to the ready queue
    SingleRQ.addProcess(P1);
    SingleRQ.addProcess(P2);
    SingleRQ.addProcess(P3);
    SingleRQ.addProcess(P4);
    SingleRQ.addProcess(P5);
    SingleRQ.addProcess(P6);
    SingleRQ.addProcess(P7);
    SingleRQ.addProcess(P8);

    //Sort the ready queue if Shortest Job First is selected
    if (isSJF)
        SingleRQ.sortAscending();

    //Start simulation by moving first Process to CPU
    AddProcessToCPU(SingleRQ.removeProcess(), currentTime);
    SingleDisplay(); //Context switch --> display state of simulation

    //While loop runs until all Processes have been moved to the Completed list. This happens when
    // the ready queue, IO list and CPU are all empty
    while (!(SingleRQ.isEmpty() && SingleIO.isEmpty() && ProcessInCPU == 0))
    {
        currentTime++; //Increment current execution time

        //Decrement remaining burst time of Process in CPU (if any)
        if (!(ProcessInCPU == 0))
            (*ProcessInCPU).decrementBurst();
        //Decrement remaining burst times of all Processes in IO (if any)
        if (!(SingleIO.isEmpty()))
            SingleIO.decrementBursts();

        //If there are Processes in IO that just completed their bursts --> move them to RQ
        // While loop accounts for multiple IO bursts finishing at the same time
        while (!(SingleIO.isEmpty()) && SingleIO.nextBurst() == 0)
        {
            //Remove "completed" burst
            SingleIO.removeNextBurst();

            //Move process to ready queue
            SingleRQ.addProcess(SingleIO.removeProcess());

            //If Shortest Job First is selected, sort the ready queue so that the shortest next burst is at the front
```

```
            // of the ready queue
            if (isSJF)
                SingleRQ.sortAscending();
        }

        if (ProcessInCPU == 0) //If no Process is in the CPU
        {
            idleTime++; //CPU was idle --> increment idleTime

            if (!(SingleRQ.isEmpty())) //If there is a Process in RQ --> move it to CPU
            {
                AddProcessToCPU(SingleRQ.removeProcess(), currentTime);
                SingleDisplay(); //CPU switches context --> so display state of simulation
            }
        }
        else //There is a Process in the CPU
        {
            if ((*ProcessInCPU).getNextBurst() == 0) //If the Process's CPU burst was just completed
            {
                (*ProcessInCPU).removeNextBurst();
                if ((*ProcessInCPU).isCompletedProcess()) //If Process completed, move it to Completed list
                {
                    (*ProcessInCPU).setTimeCompleted(currentTime);  //set completed time
                    SingleCompleted.addProcess((*ProcessInCPU));
                    ProcessInCPU = 0; //Removed Process from CPU so ProcessInCPU pointer = 0
                }
                else //If Process still has bursts remaining --> move to IO
                {
                    SingleIO.addProcess((*ProcessInCPU));
                    ProcessInCPU = 0; //Removed Process from CPU so ProcessInCPU pointer = 0
                }

                //If there is a Process in the ready queue --> move it to CPU
                if (!(SingleRQ.isEmpty()))
                {
                    AddProcessToCPU(SingleRQ.removeProcess(), currentTime);
                }

                //The CPU just finished a burst --> context switch --> display state of the simulation
                SingleDisplay();
            }
        }
    }

    //Calculate and display end-of-simulation statistics
    double CPUUtilization = ((currentTime - idleTime) / (double)currentTime) * 100;

    cout << "::::::::::::::::::::::::::::::::::::::::::::::::::\n"
        << "Finished\n\n"
         << "Total Time:         " << currentTime << "\n"
         << "CPU Utilization:    " << setprecision(2) << fixed << CPUUtilization << "%\n\n"
         << "Waiting Times       P1   P2   P3   P4   P5   P6   P7   P8   \n"
         << "                    ";
    SingleCompleted.displayWaitingTimes();
```

```
    cout << "Average Wait:       " << SingleCompleted.averageWaitingTime() << "\n\n";
    cout << "Turnaround Times    P1   P2   P3   P4   P5   P6   P7   P8   \n"
        << "                    ";
    SingleCompleted.displayTurnaroundTimes();
    cout << "Average Turnaround: " << SingleCompleted.averageTurnaroundTime() << "\n\n";
    cout << "Response Times      P1   P2   P3   P4   P5   P6   P7   P8   \n"
        << "                    ";
    SingleCompleted.displayResponseTimes();
    cout << "Average Response:   " << SingleCompleted.averageResponseTime() << "\n\n";
}

/*********************************************************************************************
Name: AddProcessToCPU
Description: "Helper" function to add a Process to the CPU. Accepts a reference to a Process
 object and currentTime.
*********************************************************************************************/
void AddProcessToCPU(Process & P, int currentTime)
{
    ProcessInCPU = &P;
    if ((*ProcessInCPU).getHasNotBeenInCPU()) //If first time Process has been in the CPU...

    {
        (*ProcessInCPU).setResponseTime(currentTime); //Set responseTime to currentTime
        (*ProcessInCPU).hasBeenInCPU(); //Indicate that the Process has been in the CPU
    }
}

/*********************************************************************************************
Name: SingleDisplay
Description: "Helper" function for SimulateSingleReadyQueue to display the state of the
 simulation after a context switch
*********************************************************************************************/
void SingleDisplay()
{
    cout << "::::::::::::::::::::::::::::::::::::::::::::::::::\n";

    string runningP = "Now running:  ";
    if (ProcessInCPU == 0)
        runningP += "[idle]\n";
    else
        runningP += (*ProcessInCPU).getProcessID()
        + " (" + to_string((*ProcessInCPU).getNextBurst()) + " time units remaining)\n";

    cout << "Current time: " << currentTime << "\n" << runningP
         << "..................................................\n";

    cout << "Ready Queue:  ";
    SingleRQ.display();
    SingleIO.display();
    SingleCompleted.display();
}

/*********************************************************************************************
Name: SimulateMultiReadyQueue
Description: Simulates an algorithm requiring "multi"/multiple ready queues
*********************************************************************************************/
void SimulateMultiReadyQueue()
```

```
{
    //Add all Process objects to the first ready queue
    MultiRQ1.addProcess(P1);
    MultiRQ1.addProcess(P2);
    MultiRQ1.addProcess(P3);
    MultiRQ1.addProcess(P4);
    MultiRQ1.addProcess(P5);
    MultiRQ1.addProcess(P6);
    MultiRQ1.addProcess(P7);
    MultiRQ1.addProcess(P8);

    //Set time quantum for Round Robin queue level 1
    MultiRQ1.setTimeQuantum(6);

    //Set time quantum for Round Robin queue level 2
    MultiRQ2.setTimeQuantum(11);

    //Start simulation by moving first process to CPU
    AddProcessToCPU(MultiRQ1.removeProcess(), currentTime);
    MultiDisplay(); //Context switch --> display state of simulation

    //While loop runs until all Processes have been moved to the Completed list. This happens when
    // the ready queues, IO list and CPU are all empty
    while (!(MultiRQ1.isEmpty() && MultiRQ2.isEmpty() && MultiRQ3.isEmpty() &&
             MultiIO.isEmpty() && ProcessInCPU == 0))
    {
        currentTime++; //Increment current execution time

        //Decrement remaining burst time and increment current time in CPU of Process in CPU (if any)
        if (!(ProcessInCPU == 0))
        {
            (*ProcessInCPU).decrementBurst();
            (*ProcessInCPU).incrementCurrentTimeInCPU(); //will be used to compare to Time Quantum
        }
        //Decrement remaining burst times of all Processes in IO (if any)
        if (!(MultiIO.isEmpty()))
            MultiIO.decrementBursts();

        //If there are Processes in IO that just completed their bursts --> move them to the RQ
        // they came to IO from. While loop accounts for multiple IO bursts finishing at the
        // same time
        while (!(MultiIO.isEmpty()) && MultiIO.nextBurst() == 0)
        {
            //Remove "completed" burst
            MultiIO.removeNextBurst();

            //Move Process to RQ it came from
            Process & P = MultiIO.removeProcess();
            if (P.getQueueLevel() == 1)
                MultiRQ1.addProcess(P);
            else if (P.getQueueLevel() == 2)
                MultiRQ2.addProcess(P);
            else
                MultiRQ3.addProcess(P);
```

```
        }

        if (ProcessInCPU == 0) //If no Process in CPU
        {
            idleTime++; //CPU was idle --> increment idleTime

            //If there is a Process in one of the RQ levels --> move the highest priority process to CPU
            if (!(MultiRQ1.isEmpty() && MultiRQ2.isEmpty() && MultiRQ3.isEmpty()))
            {
                MoveRQtoCPU(); //"Helper" function to move a Process from RQ to CPU
                MultiDisplay(); //Process added to CPU --> context switch --> display state of simulation
            }
        }
        else //Process is in CPU
        {
            //3 Cases to Consider:
            //  - Case 1:   Burst is complete
            //  - Case 2A:  Running Process came from RQ2 and another Process was just added to RQ1 --> preemption!
            //  - Case 2B:  Running Process came from RQ3 and another Process was just added to RQ1--> preemption!
            //  - Case 3:   Time Quantum expired

            //Case 1: Burst is complete
            if ((*ProcessInCPU).getNextBurst() == 0)
            {
                (*ProcessInCPU).removeNextBurst(); //Remove "completed" burst
                if ((*ProcessInCPU).isCompletedProcess()) //If Process is completed, move to Completed list
                {
                    (*ProcessInCPU).setTimeCompleted(currentTime); //Set time completed
                    MultiCompleted.addProcess((*ProcessInCPU));
                    ProcessInCPU = 0; //Removed Process from CPU so ProcessInCPU pointer = 0
                }
                else //If process still has bursts remaining --> move to IO
                {
                    (*ProcessInCPU).resetCurrentTimeInCPU(); //Reset counter for next compare with Time Quantum
                    MultiIO.addProcess((*ProcessInCPU));
                    ProcessInCPU = 0; //Removed Process from CPU so ProcessInCPU pointer = 0
                }

                //If there is a process in RQ --> move it to CPU
                if (!(MultiRQ1.isEmpty() && MultiRQ2.isEmpty() && MultiRQ3.isEmpty()))
                {
                    MoveRQtoCPU(); //"Helper" function to move a Process from RQ to CPU
                }

                //CPU just finished a burst --> context switch --> display state of the simulation
                MultiDisplay();
            }
            //Case 2A: Running Process came from RQ2 and another Process was just added to RQ1 --> preemption!
            else if ((*ProcessInCPU).getQueueLevel() == 2 && !(MultiRQ1.isEmpty()))
```

```
            {
                (*ProcessInCPU).resetCurrentTimeInCPU();
                MultiRQ2.addProcess((*ProcessInCPU));
                ProcessInCPU = 0; //Removed Process from CPU so ProcessInCPU pointer = 0
                MoveRQtoCPU(); //Add the higher-priority Process to CPU
                MultiDisplay(); //Context switch --> display state of simulation
            }
            //Case 2B: Running Process came from RQ3 and another Process was just added to RQ1
            // or RQ2 --> preemption!
            else if ((*ProcessInCPU).getQueueLevel() == 3 && (!(MultiRQ1.isEmpty()) || !(MultiRQ2.isEmpty())))
            {
                (*ProcessInCPU).resetCurrentTimeInCPU();
                MultiRQ3.addProcess((*ProcessInCPU)); //Take Process out of CPU, put it in front of RQ3
                ProcessInCPU = 0; //Removed Process from CPU so ProcessInCPU pointer = 0
                MoveRQtoCPU(); //Add higher-priority Process to CPU
                MultiDisplay(); //Context switch --> display state of simulation
            }
            //Case 3: Time Quantum expired
            else
            {
                //If process came from RQ1 and its time slice has expired
                if ((*ProcessInCPU).getQueueLevel() == 1 && (*ProcessInCPU).getCurrentTimeInCPU() == MultiRQ1.getTimeQuantum())
                {
                    (*ProcessInCPU).resetCurrentTimeInCPU(); //Reset current time in CPU to 0
                    (*ProcessInCPU).setQueueLevel(2); //Change its queue level to 2
                    MultiRQ2.addProcess((*ProcessInCPU)); //Move it to queue 2
                    ProcessInCPU = 0; //Removed Process from CPU so ProcessInCPU pointer = 0
                    MoveRQtoCPU(); //Load next process (if any) into CPU
                    MultiDisplay(); //Context switch --> display state of simulation
                }

                //If process came from RQ2 and its time slice has expired
                if ((*ProcessInCPU).getQueueLevel() == 2 && (*ProcessInCPU).getCurrentTimeInCPU() == MultiRQ2.getTimeQuantum())
                {
                    (*ProcessInCPU).resetCurrentTimeInCPU(); //Reset current time in CPU to 0
                    (*ProcessInCPU).setQueueLevel(3); //Change its queue level to 3
                    MultiRQ3.addProcess((*ProcessInCPU)); //Move it to queue 3
                    ProcessInCPU = 0; //Removed Process from CPU so ProcessInCPU pointer = 0
                    MoveRQtoCPU(); //Load next process (if any) into CPU
                    MultiDisplay(); //Context switch --> display state of simulation
                }
            }
        }
    }

    //Calculate and display end-of-simulation statistics
    double CPUUtilization = ((currentTime - idleTime) / (double)currentTime) * 100;

    cout << ":::::::::::::::::::::::::::::::::::::::::::::::::::::\n"
        << "Finished\n\n"
        << "Total Time:         " << currentTime << "\n"
        << "CPU Utilization:    " << setprecision(2) << fixed << CPUUtilization << "%\n\n"
        << "Waiting Times       P1   P2   P3   P4   P5   P6   P7   P8   \n"
```

```cpp
        << "                    ";
    MultiCompleted.displayWaitingTimes();
    cout << "Average Wait:       " << MultiCompleted.averageWaitingTime() << "\n\n";
    cout << "Turnaround Times    P1   P2   P3   P4   P5   P6   P7   P8   \n"
        << "                    ";
    MultiCompleted.displayTurnaroundTimes();
    cout << "Average Turnaround: " << MultiCompleted.averageTurnaroundTime() << "\n\n";
    cout << "Response Times      P1   P2   P3   P4   P5   P6   P7   P8   \n"
        << "                    ";
    MultiCompleted.displayResponseTimes();
    cout << "Average Response:   " << MultiCompleted.averageResponseTime() << "\n\n";
}

/*********************************************************************************************
Name: MoveRQtoCPU
Description: "Helper" function for SimulateMultiReadyQueue to move a a Process from one of
 the ready queues to the CPU
*********************************************************************************************/
void MoveRQtoCPU()
{
    if (!(MultiRQ1.isEmpty())) //If there is a Process in RQ1
        AddProcessToCPU(MultiRQ1.removeProcess(), currentTime); //Add Process to CPU
    else if (!(MultiRQ2.isEmpty())) //If no Process in RQ1, Process in RQ2
        AddProcessToCPU(MultiRQ2.removeProcess(), currentTime); //Add Process to CPU
    else if (!(MultiRQ3.isEmpty()))//If no Process in RQ1, no Process in RQ2, Process in RQ3
        AddProcessToCPU(MultiRQ3.removeProcess(), currentTime); //Add Process to CPU
}

/*********************************************************************************************
Name: MultiDisplay
Description: "Helper" function for SimulateMultiReadyQueue to display the state of the
simulation after a context switch
*********************************************************************************************/
void MultiDisplay()
{
    cout << "::::::::::::::::::::::::::::::::::::::::::::::::::\n";

    string runningP = "Now running:  ";
    if (ProcessInCPU == 0)
        runningP += "[idle]\n";
    else
        runningP += (*ProcessInCPU).getProcessID()
        + " (" + to_string((*ProcessInCPU).getNextBurst()) + " time units remaining, Level = "
        + to_string((*ProcessInCPU).getQueueLevel()) + ")\n";

    cout << "Current time: " << currentTime << "\n" << runningP
        << "..................................................\n";

    cout << "Ready Queue1: ";
    MultiRQ1.display();
    cout << "Ready Queue2: ";
    MultiRQ2.display();
    cout << "Ready Queue3: ";
    MultiRQ3.display();
    MultiIO.display();
    MultiCompleted.display();
}
```

```cpp
/************************************************************************************
Name: Gavin Wolf        Z#: 15289719
Course: Computer Operating Systems - COP 4610
Professor: Dr. Borko Furht
Due Date: 10/22/2015
Programming Assignment - CPU Scheduler

Description: This program simulates CPU scheduling algorithms on a set of processes with
 specified CPU and I/O burst times.
*************************************************************************************/

#include <iostream>
#include <vector>
#include <string>

using namespace std;

#ifndef Process_H
#define Process_H

/************************************************************************************
Name: Process class
Description: The Process class, similar to a process control block, keeps track of all the
 pertinent data for a Process as it moves through the system
*************************************************************************************/
class Process
{
public:
    Process(string Process, const vector<int> & bTimes); //constructor
    int getNextBurst();
    void removeNextBurst();
    void decrementBurst(); //used to decrement burst time
    int remainingBursts(); //returns the number of bursts remaining
    void hasBeenInCPU(); //mutator for isNoTimeInCPU
    void setResponseTime(const int time);
    void setQueueLevel(const int level);
    void resetCurrentTimeInCPU();
    void setCurrentTimeInCPU(const int time);
    void incrementCurrentTimeInCPU();
    void setTimeCompleted(const int time);
    bool isCompletedProcess();

    //accessors
    string getProcessID() const;
    int getTotalBurstTimes() const;
    int getResponseTime() const;
    int getHasNotBeenInCPU() const;
    int getQueueLevel() const;
    int getCurrentTimeInCPU() const;
    int getTimeCompleted() const;

private:
    string processID; //process name: P1, P2, P3, etc.
    vector<int> burstTimes; //array of burst times
    int totalBurstTimes; //sum of all burst times
    int responseTime; //time until first CPU time
    bool hasNotBeenInCPU; //flag that be used when calculating responseTime
```

```cpp
    int queueLevel; //for Multilevel Feedback Queue
    int currentTimeInCPU; //for Round Robin to compare to Time Quantum
    int timeCompleted; //current time when process completes execution
};

/********************************************************************************************
Name: shorterNextBurst struct
Description: Enables sorting based on next burst
*********************************************************************************************/
struct shorterNextBurst
{
    inline bool operator() (Process * ProcessA, Process * ProcessB)
    {
        return ((*ProcessA).getNextBurst() < (*ProcessB).getNextBurst());
    }
};

/********************************************************************************************
Name: lowerProcessNumber struct
Description: Enables sorting by ProcessID
*********************************************************************************************/
struct lowerProcessNumber
{
    inline bool operator() (Process * ProcessA, Process * ProcessB)
    {
        return ((*ProcessA).getProcessID() < (*ProcessB).getProcessID());
    }
};

#endif
```

```cpp
/*********************************************************************************************
Name: Gavin Wolf        Z#: 15289719
Course: Computer Operating Systems - COP 4610
Professor: Dr. Borko Furht
Due Date: 10/22/2015
Programming Assignment - CPU Scheduler

Description: This program simulates CPU scheduling algorithms on a set of processes with
 specified CPU and I/O burst times.
*********************************************************************************************/

#include <iostream>
#include <vector>
#include <string>

#include "Process.h"

using namespace std;

/*********************************************************************************************
Name: Process
Description: Constructor for a Process object that accepts a processName and a vector of
 burst times
*********************************************************************************************/
Process::Process(string processName, const vector<int> & bursts)
{
    processID = processName;
    responseTime = 0;
    burstTimes = bursts;
    for (unsigned int i = 0; i < burstTimes.size(); i++)
        totalBurstTimes += burstTimes[i];
    hasNotBeenInCPU = true;
    queueLevel = 1; //default to 1st queue
    currentTimeInCPU = 0;
    timeCompleted = 0;
}

/*********************************************************************************************
Name: getNextBurst
Description: Returns the first burst time from the vector burstTimes
*********************************************************************************************/
int Process::getNextBurst()
{
    return burstTimes[0];
}

/*********************************************************************************************
Name: removeNextBurst
Description: Removes the first burst time from the vector burstTimes
*********************************************************************************************/
void Process::removeNextBurst()
{
    burstTimes.erase(burstTimes.begin());
}

/*********************************************************************************************
Name: decrementBurst
```

```
Description: Decrements the first burst time in the vector burstTimes
*******************************************************************************************/
void Process::decrementBurst()
{
    burstTimes[0]--;
}

/*******************************************************************************************
Name: remainingBursts
Description: Returns the number of bursts remaining in the vector burstTimes
*******************************************************************************************/
int Process::remainingBursts()
{
    return burstTimes.size();
}

/*******************************************************************************************
Name: isCompletedProcess
Description: Returns true when no burst times are remaining in the vector burstTimes
*******************************************************************************************/
bool Process::isCompletedProcess()
{
    //checks if no bursts remaining
    return burstTimes.size() == 0;
}

/*******************************************************************************************
Name: getProcessID
Description: Accessor for processID
*******************************************************************************************/
string Process::getProcessID() const
{
    return processID;
}

/*******************************************************************************************
Name: getResponseTime
Description: Accessor for responseTime
*******************************************************************************************/
int Process::getResponseTime() const
{
    return responseTime;
}

/*******************************************************************************************
Name: getTotalBurstTimes
Description: Accessor for totalBurstTimes
*******************************************************************************************/
int Process::getTotalBurstTimes() const
{
    return totalBurstTimes;
}

/*******************************************************************************************
Name: getHasNotBeenInCPU
Description: Accessor for hasNotBeenInCPU
*******************************************************************************************/
```

```
int Process::getHasNotBeenInCPU() const
{
    return hasNotBeenInCPU;
}

/*******************************************************************************
Name: getQueueLevel
Description: Accessor for queueLevel
*******************************************************************************/
int Process::getQueueLevel() const
{
    return queueLevel;
}

/*******************************************************************************
Name: getCurrentTimeInCPU
Description: Accessor for currentTimeInCPU
*******************************************************************************/
int Process::getCurrentTimeInCPU() const
{
    return currentTimeInCPU;
}

/*******************************************************************************
Name: getTimeCompleted
Description: Accessor for timeCompleted
*******************************************************************************/
int Process::getTimeCompleted() const
{
    return timeCompleted;
}

/*******************************************************************************
Name: hasBeenInCPU
Description: Changes value of hasNotBeenInCPU to false, indicating that the process has been
 in the CPU
*******************************************************************************/
void Process::hasBeenInCPU()
{
    hasNotBeenInCPU = false;
}

/*******************************************************************************
Name: setResponseTime
Description: Sets the value of responseTime to the value of the parameter "time"
*******************************************************************************/
void Process::setResponseTime(const int time)
{
    responseTime = time;
}

/*******************************************************************************
Name: setQueueLevel
Description: Sets the value of queueLevel to the value of the parameter "level"
*******************************************************************************/
void Process::setQueueLevel(const int level)
{
```

```cpp
    queueLevel = level;
}

/********************************************************************************************
Name: resetCurrentTimeInCPU
Description: Sets the value of currentTimeInCPU to 0
*********************************************************************************************/
void Process::resetCurrentTimeInCPU()
{
    currentTimeInCPU = 0;
}

/********************************************************************************************
Name: setCurrentTimeInCPU
Description: Sets the value of currentTimeInCPU to the value of the parameter "time"
*********************************************************************************************/
void Process::setCurrentTimeInCPU(const int time)
{
    currentTimeInCPU = time;
}

/********************************************************************************************
Name: setTimeCompleted
Description: Sets the value of timeCompleted to the value of the parameter "time"
*********************************************************************************************/
void Process::setTimeCompleted(const int time)
{
    timeCompleted = time;
}

/********************************************************************************************
Name: incrementCurrentTimeInCPU
Description: Increments the value of currentTimeInCPU
*********************************************************************************************/
void Process::incrementCurrentTimeInCPU()
{
    currentTimeInCPU++;
}
```

```
/*************************************************************************************
Name: Gavin Wolf        Z#: 15289719
Course: Computer Operating Systems - COP 4610
Professor: Dr. Borko Furht
Due Date: 10/22/2015
Programming Assignment - CPU Scheduler

Description: This program simulates CPU scheduling algorithms on a set of processes with
 specified CPU and I/O burst times.
**************************************************************************************/

#include <iostream>
#include <vector>
#include <string>

using namespace std;

#ifndef List_H
#define List_H

#include "Process.h"

/*************************************************************************************
Name: List class
Description: The List class represents a list of Processes. The List class is the base class
 for the derived classes: ReadyQueue, IO, and Completed. The derived classes inherit the base
 class functionality and add additional functions.
**************************************************************************************/
class List
{
public:
    List(); //constructor
    void addProcess(Process & P); //add a Process to back of processList
    void addProcessToFront(Process & P); //add a Process to front of processList
    void sortAscending(); //sort Processes (used for Shortest Job First scheduling)
    Process & removeProcess(); //remove and return a Process from the front of processList
    int nextBurst(); //get next burst time
    void removeNextBurst(); //remove next burst (used when burst is completed)
    bool isEmpty(); //returns true when no Processes on processList
    int getTimeQuantum() const; //accessor for timeQuantum
    void setTimeQuantum(const int Tq); //setter for timeQuantum

protected:
    vector<Process*> processList; //vector of pointers to Process objects
    int timeQuantum; //for Round Robin in Multilevel Feedback Queue
};

#endif
```

```cpp
/**************************************************************************************
Name: Gavin Wolf        Z#: 15289719
Course: Computer Operating Systems - COP 4610
Professor: Dr. Borko Furht
Due Date: 10/22/2015
Programming Assignment - CPU Scheduler

Description: This program simulates CPU scheduling algorithms on a set of processes with
 specified CPU and I/O burst times.
***************************************************************************************/

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

#include "Process.h"
#include "List.h"

using namespace std;

/**************************************************************************************
Name: List
Description: Default constructor for a List object
***************************************************************************************/
List::List()
{
    processList = {};
    timeQuantum = 0;
}

/**************************************************************************************
Name: addProcess
Description: Accepts a reference to a Process object and adds it to the back of the
 processList vector
***************************************************************************************/
void List::addProcess(Process & P)
{
    processList.push_back(&P);
}

/**************************************************************************************
Name: addProcessToFront
Description: Accepts a reference to a Process object and adds it to the front of the
 processList vector
***************************************************************************************/
void List::addProcessToFront(Process & P)
{
    processList.insert(processList.begin(), &P);
}

/**************************************************************************************
Name: sortAscending
Description: Sorts the processList vector in ascending order of next burst time
***************************************************************************************/
void List::sortAscending()
{
```

```
    sort(processList.begin(), processList.end(), shorterNextBurst());
}

/************************************************************************************
Name: removeProcess
Description: Removes and returns the first Process in the processList vector
************************************************************************************/
Process & List::removeProcess()
{
    Process * P = processList[0];
    processList.erase(processList.begin());
    return *P;
}

/************************************************************************************
Name: nextBurst
Description: Returns the next burst for the first Process object in the processList vector
************************************************************************************/
int List::nextBurst()
{
    return (*processList[0]).getNextBurst();
}

/************************************************************************************
Name: removeNextBurst
Description: Removes the next burst for the first Process object in the processList vector
************************************************************************************/
void List::removeNextBurst()
{
    (*processList[0]).removeNextBurst();
}

/************************************************************************************
Name: isEmpty
Description: Returns true when there are no Processes in the processList vector
************************************************************************************/
bool List::isEmpty()
{
    return processList.size() == 0;
}

/************************************************************************************
Name: getTimeQuantum
Description: Accessor for timeQuantum
************************************************************************************/
int List::getTimeQuantum() const
{
    return timeQuantum;
}

/************************************************************************************
Name: setTimeQuantum
Description: Sets timeQuantum to the value of "Tq"
************************************************************************************/
void List::setTimeQuantum(const int Tq)
{
    timeQuantum = Tq;
```

```
}
```

```cpp
/*********************************************************************************************
Name: Gavin Wolf        Z#: 15289719
Course: Computer Operating Systems - COP 4610
Professor: Dr. Borko Furht
Due Date: 10/22/2015
Programming Assignment - CPU Scheduler

Description: This program simulates CPU scheduling algorithms on a set of processes with
 specified CPU and I/O burst times.
*********************************************************************************************/

#include <iostream>
#include <vector>
#include <string>

using namespace std;

#ifndef ReadyQueue_H
#define ReadyQueue_H

#include "Process.h"
#include "List.h"

/*********************************************************************************************
Name: ReadyQueue class
Description: The ReadyQueue class is a derived class of the List class.
*********************************************************************************************/
class ReadyQueue: public List
{
public:
    void display(); //displays the state of the ReadyQueue
};

#endif
```

```cpp
/*********************************************************************************
Name: Gavin Wolf        Z#: 15289719
Course: Computer Operating Systems - COP 4610
Professor: Dr. Borko Furht
Due Date: 10/22/2015
Programming Assignment - CPU Scheduler

Description: This program simulates CPU scheduling algorithms on a set of processes with
 specified CPU and I/O burst times.
*********************************************************************************/

#include <iostream>
#include <vector>

#include "Process.h"
#include "ReadyQueue.h"

using namespace std;

/*********************************************************************************
Name: display
Description: Displays the name and next burst time for all Processes on the ReadyQueue
object's processList
*********************************************************************************/
void ReadyQueue::display()
{
    //"Ready Queue:  Process    Burst\n"
    cout << "Process    Burst\n";

    if (isEmpty())
        cout << "              [empty]\n";

    for (unsigned int i = 0; i < processList.size(); i++)
    {
        cout << "              " << (*processList[i]).getProcessID()
             << "         " << (*processList[i]).getNextBurst() << endl;
    }

    cout << "..................................................\n";
}
```

```cpp
/*************************************************************************************
Name: Gavin Wolf        Z#: 15289719
Course: Computer Operating Systems - COP 4610
Professor: Dr. Borko Furht
Due Date: 10/22/2015
Programming Assignment - CPU Scheduler

Description: This program simulates CPU scheduling algorithms on a set of processes with
 specified CPU and I/O burst times.
*************************************************************************************/

#include <iostream>
#include <vector>
#include <string>

using namespace std;

#ifndef IO_H
#define IO_H

#include "Process.h"
#include "ReadyQueue.h"
#include "List.h"

/*************************************************************************************
Name: IO class
Description: The IO class is a derived class of the List class.
*************************************************************************************/
class IO : public List
{
public:
    void addProcess(Process & P); //add a Process, ordered by shortest next burst
    void decrementBursts(); //decrements all bursts in processList
    void display(); //displays each Process and its remaining time in I/O
};

#endif
```

```
/********************************************************************************************
Name: Gavin Wolf        Z#: 15289719
Course: Computer Operating Systems - COP 4610
Professor: Dr. Borko Furht
Due Date: 10/22/2015
Programming Assignment - CPU Scheduler

Description: This program simulates CPU scheduling algorithms on a set of processes with
 specified CPU and I/O burst times.
********************************************************************************************/

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

#include "Process.h"
#include "ReadyQueue.h"
#include "IO.h"

using namespace std;

/********************************************************************************************
Name: addProcess
Description: Accepts a reference to a Process object and adds the object to the IO object's
 processList in order, sorted by shortest next burst time
********************************************************************************************/
void IO::addProcess(Process & P)
{
    processList.push_back(&P);
    sort(processList.begin(), processList.end(), shorterNextBurst());
}

/********************************************************************************************
Name: display
Description: Displays the processID and next burst time for all Processes on the IO objects's
 processList
********************************************************************************************/
void IO::display()
{
    cout << "In I/O:       Process    Remaining I/O Time\n";

    if (isEmpty())
        cout << "              [empty]\n";

    for (unsigned int i = 0; i < processList.size(); i++)
    {
        cout << "              " << (*processList[i]).getProcessID()
            << "         " << (*processList[i]).getNextBurst() << endl;
    }

    cout << "..................................................\n";
}

/********************************************************************************************
Name: decrementBursts
Description: Decrements the next burst times for all Process on the IO object's processList
```

```
*****************************************************************************************/
void IO::decrementBursts()
{
    for (unsigned int i = 0; i < processList.size(); i++)
        (*processList[i]).decrementBurst();
}
```

```
/*************************************************************************************
Name: Gavin Wolf        Z#: 15289719
Course: Computer Operating Systems - COP 4610
Professor: Dr. Borko Furht
Due Date: 10/22/2015
Programming Assignment - CPU Scheduler

Description: This program simulates CPU scheduling algorithms on a set of processes with
 specified CPU and I/O burst times.
*************************************************************************************/

#include <iostream>
#include <vector>
#include <string>

using namespace std;

#ifndef COMPLETED_H
#define COMPLETED_H

#include "Process.h"
#include "List.h"

/*************************************************************************************
Name: Completed class
Description: The Completed class is a derived class of the List class.
*************************************************************************************/
class Completed : public List
{
public:
    void addProcess(Process & P); //add a Process to Completed, ordered by processID
    void display(); //to display Completed list at each context switch

    double averageWaitingTime(); //average waiting time of all Processes
    double averageTurnaroundTime(); //average turnaround time of all Processes
    double averageResponseTime(); //average response time of all Processes

    void displayWaitingTimes();
    void displayTurnaroundTimes();
    void displayResponseTimes();
};

#endif
```

```cpp
/*******************************************************************************
Name: Gavin Wolf        Z#: 15289719
Course: Computer Operating Systems - COP 4610
Professor: Dr. Borko Furht
Due Date: 10/22/2015
Programming Assignment - CPU Scheduler

Description: This program simulates CPU scheduling algorithms on a set of processes with
 specified CPU and I/O burst times.
********************************************************************************/

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

#include "Process.h"
#include "Completed.h"

using namespace std;

/*******************************************************************************
Name: addProcess
Description: Accepts a reference to a Process object and adds the object to the Completed
 object's processList in order, sorted by processID
********************************************************************************/
void Completed::addProcess(Process & P)
{
    processList.push_back(&P);
    sort(processList.begin(), processList.end(), lowerProcessNumber());
}

/*******************************************************************************
Name: display
Description: Displays the processID for all Processes on the Completed object's processList
********************************************************************************/
void Completed::display()
{
    cout << "Completed:    ";

    if (isEmpty())
        cout << "[empty]";

    for (unsigned int i = 0; i < processList.size(); i++)
    {
        cout << (*processList[i]).getProcessID() << "  ";
    }
    cout << "\n";
}

/*******************************************************************************
Name: averageWaitingTime
Description: Calculates and returns the average waiting time of all Processes on the
 Completed object's processList
********************************************************************************/
double Completed::averageWaitingTime()
{
```

```cpp
    int sum = 0;

    for (unsigned int i = 0; i < processList.size(); i++)
    {
        sum += ((*processList[i]).getTimeCompleted() - (*processList[i]).getTotalBurstTimes());
    }

    return sum / (double)processList.size();
}

/*****************************************************************************************
Name: displayWaitingTimes
Description: Calculates and outputs average waiting time for all Processes on the Completed
 object's processList
*****************************************************************************************/
void Completed::displayWaitingTimes()
{
    for (unsigned int i = 0; i < processList.size(); i++)
        cout << (*processList[i]).getTimeCompleted() - (*processList[i]).getTotalBurstTimes() <<
        ((*processList[i]).getTimeCompleted() - (*processList[i]).getTotalBurstTimes() < 100 ? " " : "") << "  ";

    cout << endl;
}

/*****************************************************************************************
Name: averageTurnaroundTime
Description: Calculates and returns average turnaround time for all Processes on the Completed
 object's processList
*****************************************************************************************/
double Completed::averageTurnaroundTime()
{
    int sum = 0;

    for (unsigned int i = 0; i < processList.size(); i++)
        sum += (*processList[i]).getTimeCompleted();

    return sum / (double)processList.size();
}

/*****************************************************************************************
Name: displayTurnaroundTimes
Description: Calculates and displays the turnaround time for all Processes on the Completed
 object's processList. Note: because the simulation assumes that all jobs arrive at time 0,
 the turnaround time is equal to timeCompleted
*****************************************************************************************/
void Completed::displayTurnaroundTimes()
{
    for (unsigned int i = 0; i < processList.size(); i++)
        cout << (*processList[i]).getTimeCompleted() << "  ";

    cout << endl;
}

/*****************************************************************************************
Name: averageResponseTime
Description: Calculates and returns the average response time of all Processes on the
```

```
 Completed object's processList.
*************************************************************************************/
double Completed::averageResponseTime()
{
    int sum = 0;

    for (unsigned int i = 0; i < processList.size(); i++)
        sum += (*processList[i]).getResponseTime();

    return sum / (double)processList.size();
}

/*************************************************************************************
Name: averageResponseTime
Description: Displays the response times for all Processes on the Completed object's
 processList.
*************************************************************************************/
void Completed::displayResponseTimes()
{
    for (unsigned int i = 0; i < processList.size(); i++)
        cout << (*processList[i]).getResponseTime() << ((*processList[i]).getResponseTime() < 10 ? " " : "") << "   ";

    cout << endl;
}
```