CSC139: Operating System Concepts
Spring 2018
Third Assignment: CPU Scheduling
Due Wednesday, April 11th, 2018

In this assignment, you will write code to simulate the following CPU scheduling algorithms, assuming a **single CPU**.

1. **Round Robin**
   Add the processes to a regular First-In-First-Out (FIFO) queue in the order they arrive. If multiple processes arrive at the same time, add them to the queue in the order they appear in the input file. When the time quantum expires, the process that has the CPU is placed at the end of the queue if it has not terminated yet. If a new process arrives and a time quantum expires at the same time, insert the new arrival at the end of the queue before inserting the process whose time quantum expired. After Time 0, scheduling decisions are made when either the process that has the CPU terminates or the time quantum expires. In either case, the scheduler gives the CPU to the first process in the queue. The value of the time quantum is a parameter that is specified in the input file next to the algorithm's name as shown in Example 1 below.

2. **Shortest Job First (SJF)**
   Assume **no-preemption** but take arrival times into account. To do that, you have to simulate the time (by simply defining an integer variable that keeps track of the time). At any given point in time, your scheduler considers **only the processes that have arrived**. Whenever a new process arrives, add it to a priority queue in which the key is the CPU burst length. After Time 0, scheduling decisions are made only when the process that currently has the CPU terminates. The scheduler will then give the CPU to the process with the shortest CPU burst. Ties must be broken based on arrival times, that is, if the ready queue has two or more processes with the same CPU burst, these processes get the CPU in the order they have arrived (FCFS). If multiple processes have the same CPU burst and the same arrival time, ties are broken arbitrarily.

3. **Priority Scheduling without Preemption (PR_noPREMP)**
   Take arrival times into account. Whenever a new process arrives, add it to a priority queue, in which the key is the priority value read from the input file. After Time 0, scheduling decisions are made only when the process that currently has the CPU terminates (no preemption). The scheduler will then give the CPU to the process with the highest priority (smallest priority number). Ties are broken arbitrarily.

4. **Priority Scheduling with Preemption (PR_withPREMP)**
   Take arrival times into account, and implement preemption. Whenever a new process arrives, add it to a priority queue, in which the key is the priority number read from the input file. After Time 0, scheduling decisions are made when the process that currently has the CPU terminates or when a higher priority process arrives. The schedule will then give the CPU to the process with the highest priority (smallest priority number). Ties are broken arbitrarily. If the process with the highest priority is the new process that has just arrived, the process that has the CPU must get preempted and added to the priority queue (unless it has just terminated at that point).

Your program should read an input file named “input.txt” and write the results into an output file named “output.txt”. The formats of these files are as follows:

Input File

The first line has the name of the scheduling algorithm to run (one of the four names given above).

The second line has a single integer representing the number of processes in the file.

In the rest of the file, there is one line per process with the following information:

Process number   Arrival Time   CPU burst time   Priority

If multiple processes have the same arrival time, your scheduling algorithm should assume that the processes have arrived in the order they appear in the file (there are negligibly small differences in arrival times). For priority scheduling, assume that **smaller numbers indicate higher priorities**. Non-priority algorithms should simply ignore the priority field.

Output File

Your output file will show the scheduling results for each of the algorithms listed in the input file. The first line in the output file has the name of the scheduling algorithm. The file then shows the schedule using a simple text format in which there is one line for each CPU assignment (each line corresponds to a vertical line in the Gantt chart). Each line has two numbers: one indicating the time point and one indicating the process number that got the CPU at that point. The last line in the output file shows the average waiting time.

Examples

As shown in the example below, when the algorithm is RR, there should be an integer parameter next to the algorithm’s name specifying the length of the time quantum:

Input 1 (from the book)

```
RR  4
3
1    0    24    1
2    0    3     1
3    0    3     1
```

Output 1

```
    RR  4
    0    1
    4    2
    7    3
    10   1
    14   1
    18   1
    22   1
    26   1
AVG Waiting Time: 5.67
```

Input 2 (from the book)

```
SJF
4
1    0    6    1
2    0    8    1
3    0    7    1
4    0    3    1
```

Output 2
```
    SJF
     0    4
     3   1
     9   3
    16   2
```
AVG Waiting Time: 7

Input 3 (from the book)
```
PR_noPREMP
5
1    0    10    3
2    0    1     1
3    0    2     4
4    0    1     5
5    0    5     2
```

Output 3
```
     PR_noPREMP
     0      2
     1      5
     6    1
     16   3
     18   4
```
AVG Waiting Time: 8.2

Input 4
```
PR_withPREMP
4
1    0    8     3
2    3    1     1
3    5    2     4
4    6    2     2
```

Output 4
```
    PR_withPREMP
     0      1
     3     2
     4    1
     6    4
     8    1
     11   3
```
AVG Waiting Time: 2.25

Input 5 (from the book)

```
SJF
4
1    0    4    1
2    2    5    1
3    3    5    1
4    6    3    1
```

```
Output 5
    SJF
     0    1
     4   2
     9   4
    12   3
AVG Waiting Time: 3.5
```

You can write your implementation using C, C++ or JAVA.

Priority Queue Implementation
You can implement the priority queue using the easiest and most convenient way for you (the simplest implementation is probably an unsorted array). However, you will get extra credit if you implement it as a binary heap and do a timing experiment as described in the second extra-work section below.

Extra Work
You will get up to 25% of extra credit if you do the following:
1. Implement the priority queue using both a binary heap and an unsorted array.
2. Run an experiment with a **<u>sufficiently large</u>** randomized input to compare the performance of these two priority queue implementations. To this end, you will have to include timing code as we did in the second programming assignment. The input size must be large enough to show a measurable impact of the priority queue implementation on the total time needed to process the input. Run your timing experiment using only the PR_withPREMP algorithm described above.
3. Write a report summarizing the performance results that you get (which priority queue implementation is faster and how much faster it is).

Submission
Submit your source code on Canvas. If you are doing the extra work, you will need to submit a report as well. Note that your code will be tested using many different inputs. Some sample test cases are provided for you, but we will use different test cases when we grade your program. So, make sure that before submitting your code, you test it thoroughly. Test your code on the **Athena** machine in our ECS computing system. If your code does not work on Athena, we will give you the chance to fix it and resubmit it, but you will lose points for that.