P10 SJF Process Scheduler

Overview

Your computer performs multiple processes, ranging from mundane stuff like disk checkup to more important matters such as installing Eclipse. The **process scheduler** is an important component of the operating system that is responsible for deciding which process will run next. In this assignment, we are going to develop a simple process scheduler to schedule processes that are ready to be run. This scheduler consists of only one processor. This means that only one process can be executed at a time. Our scheduler will run on a heap-based priority queue, and will operate conforming to the Shortest Job First (SJF), or shortest job next scheduling policy. In particular, this scheduling policy selects the ready process with the smallest execution time (aka burst time) to execute next. [Burst Time: Time required by a process to be executed in the Central Processing Unit (CPU)]. This programming assignment represents an interesting use case of the priority queue abstract data type implemented using a min-heap.

Grading Rubric

5 points	**Pre-Assignment Quiz:** The P10 pre-assignment quiz is accessible through Canvas before having access to this specification by 9:59PM on Sunday 12/01/2019. Access to the pre-assignment quiz will be unavailable passing its deadline.
20 points	**Immediate Automated Tests:** Upon submission of your assignment to Gradescope, you will receive feedback from automated grading tests about whether specific parts of your submission conform to this write-up specification. If these tests detect problems in your code, they will attempt to give you some feedback about the kind of defect that they noticed. Note that passing all of these tests does NOT mean your program is otherwise correct. To become more confident of this, you should run additional tests of your own.
20 points	**Additional Automated Tests:** When your manual grading feedback appears on Gradescope, you will also see the feedback from these additional automated grading tests. These tests are similar to the Immediate Automated Tests, but may test different parts of your submission in different ways.
5 points	**Manual Grading Feedback:** After the deadline for an assignment has passed, the course staff will begin manually grading your submission. We will focus on looking at your algorithms, use of programming constructs, and the style and readability of your code. This grading usually takes about a week from the hard deadline, after which you will find feedback on Gradescope.

Learning Objectives

The goals of this assignment include developing, using, and testing an implementation of a Priority Queue as a min-heap. You are going to gain experience implementing several fundamental operations on this data structure including insertion, removal, and peek operations with respect to the problem specification.

Additional Assignment Requirements and Notes

- You MUST NOT add any fields either instance or static, and any public methods either static or instance to all your submitted classes, other than those defined in this write-up.
- DO NOT make any change to the provided `WaitingQueueADT` interface. In addition, **DO NOT** submit the `WaitingQueueADT.java` file on Gradescope.
- All your test methods should be defined and implemented in your `ProcessSchedulerTester.java`.
- You CAN define local variables that you may need to implement the methods defined in this program.
- You CAN define private methods to help implement the different public methods defined in this write-up.
- In addition to the required test methods, we HIGHLY recommend that you develop your additional own unit tests (public static methods that return a boolean) to convince yourself of the correctness of every behavior implemented in your `CustomProcess` and `WaitingProcessQueue` classes with respect to the specification provided in this write-up. Make sure to design the test scenarios for every method before starting its implementation. Make sure also to check for all the special cases.
- All implemented methods including the overridden ones MUST have their own javadoc-style method headers, according to the CS300 Course Style Guide.

1 Application Demo

The following illustrates a demo (sample of run) of the driver method of this programming assignment. Note that the way how your operating system manage processes is far away more complicated.

```
==========   Welcome to the SJF Process Scheduler App   ========

Enter command:
[schedule <burstTime>] or [s <burstTime>]
[run] or [r]
[quit] or [q]

sch 5
WARNING: Please enter a valid command!

Enter command:
[schedule <burstTime>] or [s <burstTime>]
[run] or [r]
[quit] or [q]

s 5
Process ID 1 scheduled. Burst Time = 5

Enter command:
[schedule <burstTime>] or [s <burstTime>]
[run] or [r]
[quit] or [q]

s 10
Process ID 2 scheduled. Burst Time = 10

Enter command:
[schedule <burstTime>] or [s <burstTime>]
[run] or [r]
[quit] or [q]

s 3
Process ID 3 scheduled. Burst Time = 3

Enter command:
[schedule <burstTime>] or [s <burstTime>]
[run] or [r]
[quit] or [q]
```

```
r
Starting 3 processes

Time 0 : Process ID 3 Starting.
Time 3 : Process ID 3 Completed.
Time 3 : Process ID 1 Starting.
Time 8 : Process ID 1 Completed.
Time 8 : Process ID 2 Starting.
Time 18 : Process ID 2 Completed.

Time 18 : All scheduled processes completed.

Enter command:
[schedule <burstTime>] or [s <burstTime>]
[run] or [r]
[quit] or [q]

s 1
Process ID 4 scheduled. Burst Time = 1

Enter command:
[schedule <burstTime>] or [s <burstTime>]
[run] or [r]
[quit] or [q]

r
Starting 1 process

Time 18 : Process ID 4 Starting.
Time 19 : Process ID 4 Completed.

Time 19 : All scheduled processes completed.

Enter command:
[schedule <burstTime>] or [s <burstTime>]
[run] or [r]
[quit] or [q]

q
4 processes run in 19 units of time!
Thank you for using our scheduler!
Goodbye!
```

2 Getting Started

Start by creating a new Java Project in eclipse called P10 SJF Process Scheduler, for instance. You have to ensure that your new project uses Java 8, by setting the "Use an execution environment JRE:" drop down setting to "JavaSE-1.8" within the new Java Project dialog box. Then, add the following source file to it WaitingQueueADT.java. The provided `WaitingQueueADT` interface represents the generic abstract data type for a priority queue, which we are going to implement in this programming assignment.

3 Create and implement the CustomProcess Class

Now, create a class called `CustomProcess`. This class represents the data type for the processes used in our application. Your CustomProcess class must implement the java.lang.Comparable interface over itself. We note that CustomProcess class IS NOT a generic class.

Each process is identified by a UNIQUE PROCESS_ID and has a predefined burstTime. The state of this class is defined by the following private fields.

```
private static int nextProcessId = 1; // stores the id to be assigned to the next process
                                      // to be created
private final int PROCESS_ID; // unique identifier for this process
private int burstTime; // time required by this process for CPU execution
```

This class defines also ONE constructor that takes one input argument, as follows. This constructor should create a new instance of CustomProcess with the given burstTime. The burstTime of this process must be a non-zero, positive integer. Otherwise, the constructor MUST throw an `IllegalArgumentException` with a descriptive error message.

```
public CustomProcess(int burstTime) { }
```

Your `CustomProcess` class MUST also override `toString()` method defined in java.lang.Object class as well as the `compareTo()` method defined in the java.lang.Comparable interface, as follows.

```
  /**
   * Returns a String representation of this CustomProcess
   * @return a string representation of this CustomProcess
   */
  public String toString() {
    return "p" + this.PROCESS_ID + "(" + this.burstTime + ")";
  }
  public int compareTo(CustomProcess other);
```

The `CustomProcess.compareTo()` method operates as follows. Suppose that we have two instances of CustomProcess objects referred by p1 and p2.

- $p1.compareTo(p2) < 0$ means that the `p1` is **smaller** than `p2`.
- $p1.compareTo(p2) == 0$ means that `p1` and `p2` are **equals** with respect to the comparison criteria.
- $p1.compareTo(p2) > 0$ means that `p1` is **larger** than `p2`.

If `p1` and `p2` have **different burst times**, the customProcess `p1` is **smaller** than `p2` if `p1` has a **smaller burst time** than `p2`. By the same way, `p1` is **larger** than `p2` if `p1` has a **larger burst time** than `p2`. If `p1` and `p2` have the **same burst time**, the **smaller** CustomProcess is the one having the **lower** `PROCESS_ID`. Given that, the customProcesses `p1` and `p2` will be **equals** if they have the same `burstTime` and the same `PROCESS_ID`.

Finally, you have to implement accessors for both `burstTime` and `PROCESS_ID` fields. These accessors MUST have the following signatures.

```
public int getProcessId() {}
public int getBurstTime() {}
```

4 Create the WaitingProcessQueue Class

Now create a new class called `WaitingProcessQueue` and add it to your project. This class implements the WaitingQueueADT interface of CustomProcesses. Note that the `WaitingProcessQueue` class IS NOT a generic class and MUST contain the following private fields.

```
private static final int INITIAL_CAPACITY = 20; // the initial capacity of this
                                               // waiting process queue
private CustomProcess[] data; // min heap-array storing the CustomProcesses
                              // inserted in this WaitingProcessQueue.
                              // data is an oversize array
private int size; // number of CustomProcesses stored in this WaitingProcessQueue
```

- Note that the `INITIAL_CAPACITY` must be a positive (and non-zero) integer value. You can change the suggested value of `INITIAL_CAPACITY` if you would like.
- Note that your `WaitingProcessQueue` MUST be an array-based implementation of priority queue as a min-heap from the scratch. You are **NOT allowed** to import or use java.util.ArrayList class or any other ArrayList-based implementation.

- The `data` instance field array MUST be an **array-based MIN-heap**. It will be implemented such that the ROOT node is ALWAYS the entry at **index 0** in the array.
- The `CustomProcess.compareTo()` method should be used to compare the different processes stored in the `data` array.
- Unused indexes (meaning any past the current size of the `data` array) should contain null.
- The array `data` should expand to fit all the processes being inserted. You can devise your dynamic array by simply doubling its size whenever it is full. One simple way to do so is to copy all elements into a new 2×-sized array when you run out of space. We note that implementing a shadow array to expand the size of your data array is not required by this assignment.
- Your `WaitingProcessQueue` class contains ONLY ONE no-argument constructor that creates an empty waiting process queue whose initial capacity is `INITIAL_CAPACITY`.
- In addition to implementing ALL the methods defined in the `WaitingQueueADT` interface, your `WaitingProcessQueue` must have the following two private helper methods.

```java
private void minHeapPercolateUp(int index) { }
private void minHeapPercolateDown(int index) { }
```

These private methods should respectively percolate up and percolate down the element at the given index in the `data` min-heap array. You would use these methods in the implementation of your `insert` and `removeBest` methods respectively.

- you can add as many private helper methods as you judge necessary to implement the defined behavior of your `WaitingProcessQueue` class.
- You have to override the `toString()` method in your WaitingProcessQueue class. This method should return a String representation of all the non-null elements (custom processes) stored in the array `data` starting from index 0 to index size-1. If the waiting process queue is empty, your `WaitingProcessQueue.toString()` method should return a single space string " ". Otherwise, it should return the string representations of the different custom processes stored in the `WaitingProcessQueue.data` array separated by a space.
- NO further public method not defined in this write-up should be added to your `WaitingProcessQueue` class.

5 About your test methods

All your test methods must be implemented in your `ProcessSchedulerTester`. Make sure to create such class and add it to your project if not yet done. This class should include at least

4 public test methods to check the functionality of your code. Make sure to run all your tests in your `ProcessSchedulerTester`'s main method. In particular you HAVE TO implement the following TWO test methods having EXACTLY the two following signatures.

```
// checks the correctness of the insert operation
// implemented in the WaitingProcessQueue class
public static boolean testInsertWaitingProcessQueue(){}

// checks the correctness of the removeBest operation
// implemented in the WaitingProcessQueue class
public static boolean testRemoveBestWaitingProcessQueue(){}
```

6 Create the ProcessScheduler class

The `ProcessScheduler` class represents the data type for the main scheduler for our processes. Your ProcessScheduler class should define the following private instance fields.

```
private int currentTime; // stores the current time after the last run
private int numProcessesRun; // stores the number of processes run so far
private WaitingProcessQueue queue; // this processing unit's queue
```

It defines also a **no-argument constructor** that sets up the WaitingProcessQueue queue to an empty one, and initializes both `currentTime` and `numProcessesRun` to zero.

We note that this version process scheduler is very simplified. As illustrated in the demo presented in Demo. 1, first a set of processes are scheduled according to the SJF policy. Then, they will be run one by one according to the same policy. This process may be repeated many times.

To do so, in addition to the constructor, this class MUST implement the following public methods.

```
// This method inserts the given process in the WaitingProcessQueue queue.
public void scheduleProcess(CustomProcess process) {}
```

```
// runs the ready processes already scheduled in this processScheduler's queue
public String run() {}
```

- When it is called, the `run()` method starts running the ready processes already scheduled or inserted in the waiting process queue according to the SJF scheduling policy. The process that has the highest priority (i.e. the smallest process with respect to the `CustomProcess.compareTo()` method) should be run first. This should be simply the

CustomProcess returned by the `WaitingProcessQueue.removeBest()` method.

- The `ProcessScheduler.run()` method returns when all the scheduled processes are run and the queue is empty. It returns a String that represents the log of one run operation. The format of this log is as follows.

- The log returned by the run() method should begin with one of the following messages, depending on the `size of the queue`, if it contains only one process, or zero or many processes.

```
"Starting " + <size of the queue> + " process\n\n"
or
"Starting " + <size of the queue> + " processes\n\n"
```

- Each time a ready process is removed to be run, the following message must be added to this log:

```
"Time " + <currentTime> + " : Process ID " + <processId> + " Starting.\n"
```

- Then, the following message will be added to the log:

```
"Time " + <currentTime> + ": Process ID " + <processId> + " Completed.\n"
```

- When ALL processes are run, the following message will be added to the log:

```
"\nTime " + <currentTime> + ": All scheduled processes completed.\n"
```

- Do not forget to update the number of processes run so far. NOTE that your application starts at `currentTime = 0`. When your scheduler runs, the currentTime will be incremented with respect to the burst time of the process being run. The time taken to insert or remove processes is negligible.

7 Implement the Driver method of the SJF Process Scheduler

The final step in this assignment is to implement the main method of the ProcessScheduler class. This method serves as the driver of the application. An example of the output of this main method is provided at the top of this write-up. NOTE that this method creates and uses only one instance of the Scanner class, and also only one instance of the ProcessScheduler class. You are free to organize your solution to the best of your ability, using whatever combination of fields and methods that seems best to you. But, make sure to define at least TWO private static helper methods to help implement the behavior of this driver application. Running the main method in the `ProcessScheduler` class must result in an interactive session like the one demonstrated in Demo. 1. We provide in the following some specific requirements for how this interactive session should proceed.

- When taking input from the user, you have to check the validity of the input and print a warning if it is invalid.
- If the user enters a wrong command or if the format of the command is not valid, the exact following message should be printed out to the console.

```
"WARNING: Please enter a valid command!\n"
```

- If the burst time of a process is not an integer, the following message should be printed out to the console.

```
"WARNING: burst time MUST be an integer!\n"
```

- If the burst time of a process is not a non-zero, positive integer, meaning if the constructor of the `CustomProcess` class throws an `IllegalArgumentException`, the error message of the thrown exception should be printed out to the console.
- Each time a process is scheduled after a valid s or schedule command is entered, the following message will be printed out to the console.

```
"Process ID " + <processID> + " scheduled. Burst Time = " + <burstTime> + "\n"
```

printed out to the console:

- Each time a valid r or run command is entered, the log message returned by run() method must be printed out to the console.
- When the program terminates after the user enters a valid q or quit command, the following goodbye message must be printed out to the console.

```
<numProcessesRun> + " processes run in " + <currentTime> + " units of time!\n" +
"Thank you for using our scheduler!\n" + "Goodbye!\n"
```

8 Assignment Submission

Congratulations on finishing this CS300 assignment! After verifying that your work is correct, and written clearly in a style that is consistent with the CS300 Course Style Guide, you should submit your final work through gradescope.com. The only 4 files that you must submit include: `CustomProcess.java`,`WaitingProcessQueue.java`,`ProcessScheduler.java`, and `ProcessSchedulerTester.java`. Your score for this assignment will be based on your **"active"** submission made prior to the hard deadline of Due: **9:59PM on December 4th**. The second portion of your grade for this assignment will be determined by running that same submission against additional offline automated grading tests after the submission deadline. Finally, the third portion of your grade for your submission will be determined by humans looking for organization, clarity, commenting, and adherence to the CS300 Course Style Guide.

Extra Challenges

Here are some suggestions for interesting ways to extend this memory game, after you have completed, backed up, and submitted the graded portion of this assignment. **No extra credit will be awarded for implementing these features**, but they should provide you with some valuable practice and experience. DO NOT submit such extensions via gradescope.

1. **Suggestion 1** – You can update your program to provide a log with the following statistics (metrics) for every process being run.
   - **Arrival Time:** Time at which the process arrives in the ready queue (is scheduled).
   - **Completion Time:** Time at which process completes its execution.
   - **Burst Time:** Time required by a process for CPU execution.
   - **Turn Around Time:** Time Difference between completion time and arrival time. $(turnAroundTime = completionTime - arrivalTime)$
   - **Waiting Time:** Time Difference between turn around time and burst time. $(waitingTime = turnAroundTime - burstTime)$
   - **ResponseRatio:** $responseRatio = (waitingTime + burstTime)/burstTime$
2. **Suggestion 2** – You can expand the heap array in your WaitingProcessQueue class using a shadow array. Make sure to implement appropriately `swap` and `set` operations for a shadow array.