Hope Neels

Final Project Documentation

Cs526

**Introduction**

This documentation summarizes the data structures used and lessons learned from Cs526 Final Project. The Project is a simulation of a CPU Scheduler, in which processes enter the “ready queue” when they are waiting to be executed. Unlike a real CPU Scheduler, each process runs to completion without interruption once it has started running. However, like some real CPUs, process priorities are updated when they have been waiting to ensure fair distribution of CPU time, and this is one of the primary tasks of implementing this assignment.

**Description of Data Structures**

**D Data Structure**

This is the data structure referenced in the assignment instructions as “D” which holds processes which have not yet arrived, meaning their arrival time is later than the current system time (represented by the while-loop in main method that implements the “system clock”). I chose to design this D data structure as a priority queue. Multiple different data structures could have been chosen, but I believe a Priority Queue will be the fastest choice for removing the next process when its arrival time is reached. Because the Q data structure is also a Priority Queue (see below), I implemented two different Comparators to be passed in when the Priority Queues are instantiated. For this D data structure, the Arrival Comparator class is passed into the Priority Queue constructor, ensuring that the “minimal key” process is always the one with the lowest arrival time. Though the processes in the current “input.txt” file are already ordered by arrival time, a robust implementation of this project will ensure a fast runtime even when process arrival times are received out of order, and this is why I believe a Priority Queue is the best choice.

**Q Data Structure**

Since the assignment instructions specified that we “use the Priority Queue implemented in Java’s Priority Queue,” that’s what I chose for the Q data structure. This structure represents the container for processes whose arrival time has passed, that have been removed from the D data structure, and are now waiting to be executed. The Priority Comparator class is implemented to be passed into this Q data structure’s constructor, so that keys are ordered by process priority (instead of arrival time as in the D data structure.)

**Process Object**

The Process object encapsulates the data about the Process object, including ID, priority, duration, arrival time (all of which are read from the input file) as well as the start system time that the process begins executing. In addition to the standard constructor, setters and getters, this object has a decrementPriority() method that decreases its priority by one, for use within the updatePriority() method of ProcessScheduling.java.

One important note: I elected to define a toString() method in my Process class, so that the Process object is represented in a uniform String each time it is printed. As a result, my formatting differs slightly from the assignment suggested output. Specifically, my program always prints processes in the uniform format “id = 1, priority = 4, duration = 25, arrival time = 10” whether printing the list of all processes or printing the individual processes when they are chosen to run. In contrast, the assignment output prints the process’s *arrival* and *duration* attributes in different orders at different stages. I prefer the design choice of implementing a toString() method to reduce redundant String concatenations throughout the program.

**String[] tokens**

This String array is a temporary object used to parse the input from the “input.txt” file within the createProcessFromFile() helper method. When each line of the file is read with Scanner, the line is split into the “tokens” String array so each token can be used to create an attribute of the Process.

**ArrayList<Process> tempList**

This data structure, which is created within the updatePriority() helper method, temporarily stores processes whose priorities must be updated when they have been waiting more than 30 time units. Since Java’s Priority Queue has no way to automatically bubble an object when its key is updated, the minimal key process might not be removed after a priority was updated unless the process object is removed and reinserted. The method safely removes any processes that have been waiting longer than 30 time units using the iterator.remove() method, stores them in tempList to update their priorities, then reinserts them back into the Q Priority Queue to ensure the heap maintains its proper order.

**Explanation of Helper Methods**

More detailed descriptions of each method can be found within the code comments throughout the project files, including clear explanations of their parameters and return values. I have compartmentalized the code into segmented methods to isolate behavior and ensure good design. The ProcessScheduling.java file implements the following helper methods:

* createProcessFromFile() – reads the data from “input.txt” file, creates a new Priority Queue of Process objects based on that data (representing the “D” data structure) and returns that D data structure when complete.
* runProcess() – accepts a Process object, a currentTime int, and a totalWaitTime int to simulate “running” the process in the CPU. The method calculates and stores the process object’s wait time and execution start time, prints the required “run process” data to the output file, and returns the new total wait time for all processes.
* updatePriority() – as described above, this method ensures the heap-order property is maintained in the Q Priority Queue by removing, updating, and reinserting any Process objects that have been waiting in the D data structure for longer than the maximum wait time of 30 time units.

**Discussion**

The completion of this project clarified some of the benefits that the Adaptable Priority Queue, as described in the textbook, has over Java’s Priority Queue implementation. Specifically, the Adaptable Priority Queue’s replaceKey() method would be a great asset in a program such as this where objects need to be re-bubbled within the heap as their keys change. This functionality is what I strived to mimic with the updatePriority() helper method, but because Java’s data structure doesn’t have this ability to heapify(), my method had to remove and reinsert processes in order to guarantee the heap-order property be maintained.

The process removal/reinsertion design accomplishes the requirements for the project, because the assignment states that if two processes have equal priority, one will be chosen “arbitrarily,” and indeed this is what my program does. However, an ideal design might be a heap in which, if two processes have equal priority, the one that has existed longer (i.e. the older process) in the heap is chosen. With my design, the *older* process must be removed and reinserted to trigger up-heap bubbling, so it becomes the *newer* process. Again, this has no consequence unless two processes have an identical priority, but this is a minor detail that might be improved with an Adaptable Priority Queue.

Regarding processes with equal priorities, there is also a slight difference between my output and the suggested assignment output. Specifically, when process 4 and process 7 have equal priority in the Q Priority Queue, my program chooses process 4 first and then 7, whereas the suggested output chooses 7 first and then 4. Interestingly, this arbitrary choice makes my program slightly more efficient, with 390 total wait time for all processes instead of 391. I argue that my design is more optimal, because at the time that 4 and 7 have equal priority, process 4 has both a larger wait time and a shorter duration than 7. Thus when 4 is chosen first, the total wait time for all processes at the end of program execution is less. Another optimal design could be a data structure that, when faced with a choice between two identical priorities, *always* makes a sophisticated evaluation based on both process wait time and duration, but this kind of design is beyond the scope of my current programming expertise.

In addition to the knowledge I gained about different data structures while working on this project (as described above), I also strengthened my understanding of Boolean logic. Specifically, I chose to design my program with one while-loop to simulate the running of the CPU/system, though the assignment pseudocode suggested this could be accomplished with two separate loops instead—one to execute until D data structure becomes empty, and a second to execute until all processes are done running (that is, until Q is also empty and the last process is done). I chose the single-while-loop instead to eliminate repeated code and strive for a more elegant code. I am happy with the outcome, because I believe code repetition should be eliminated when possible, but translating the assignment logic into correct Java code forced me to think carefully about all the Boolean expressions in my *while* and *if* statements, and required some patient debugging for me to arrive at the correct design. After lots of careful thought, trial and error, and redesigns, I arrived at a correct solution which I am proud to showcase as my final project.