1. Introduction

You will be primarily using priority queues for this programming assignment. Among priority queues, you are free to pick and choose the most appropriate/efficient heap implementation. Your design can also use any other data structures that we have discussed in class, as long as the use of such a data structure could be justified from a design perspective. However, you are required to choose a simple data structure wherever possible. An unnecessarily overcomplicated implementation (example: using a heap where actually a simple vector would suffice) will lead to reduction in points. The idea is to let you make the design choice, and identify the most suitable data structure for each component in the system in order to effectively address performance and functional tradeoffs. In your report, please discuss your design and your design choices.

Use of any appropriate STL containers is highly encouraged whenever possible (priority queue, set, map, list, vector, etc.). Recall that STL's "priority queue" implements a binary heap in the form of a *MaxHeap*. You can easily change it into a *MinHeap* (as shown in the book, Figure 6.57). You are also free to re-use/reproduce any source code that is provided in the Weiss textbook. Most of the source codes are available from the following website:

https://users.cs.fiu.edu/~weiss/dsaa_c++4/code/

2. Problem Statement

Your organization has purchased a new parallel computer (or "cluster") which has '*p*' processors. Your task is to design and implement a simple shortest-job-first scheduler that allows multiple users to access the cluster at the same time, as per the specifications in Section 3 of this document. Figure 1 is a conceptual illustration of the different components in the system, how they interact, and the necessary data containers you will have to implement. Refer to this figure while reading the rest of this document.

3. Problem Specification

The scheduler performs all required functions at regular timesteps, called "*ticks*". The *tick* in other words, is the pulse of the scheduler. During each tick, various events could happen and the scheduler's responsibility is to address all these events before moving on to the next tick. Because of this, each tick could take variable amount of time to complete, and for design simplicity you can assume that there is sufficient real time within each tick to complete all required tasks. The various events and the respective action that is required to be carried out by the scheduler are outlined below. (Refer Figures 1 & 2 as you read this.)

Init condition: When the algorithm starts, the system is initialized with *p* processors in the free pool. Figure 2 shows the sequence of steps that the scheduler should perform during each tick as a pseudocode.

1. Event: The user inputs at most one job at the start of any tick. Each job arrives to the scheduler in the following format: <*job_description, n_procs, n_ticks*>. If the *job_description* field is set to *"NULL"* then that means that the user does not have any job to insert during this clock tick.

Action: For each job submission, the system assigns the job a new, unused integer id (referred to as "*job_id*") and then calls the ***InsertJob()*** function with the following parameters:

<*job_id, job_description, n_procs, n_ticks*>, where:

a. "*job_description*" contains details on what command to execute to run the job (i.e., "<*program name*> <*its arguments*>"). It is a string data type. If the *job_description* is set to *"NULL"* then it means that there are no jobs to be inserted in this clock tick;
b. "*n_procs*" is the number of processors that this job needs to run on;

c. "*n_ticks*" is the number of ticks that this job is going to need on each of the *n_procs* processors. Note that this number is the equivalent of the job's estimated running time, from start to finish, as the job will be launched on all *n_procs* processors at the same time to run in parallel for *n_ticks* number of ticks.

Note: The notation ***InsertJob()*** does not mean there are no arguments to this function. You need to decide what arguments each function signature should have. I have purposefully left that choice to you as part of design.
Same applies to all the other functions.

The ***InsertJob()*** function first checks if ($0 < n\_procs \leq p$) and ($n\_ticks > 0$). If so, it inserts the job into a "wait queue". Otherwise, a job submission error is raised with an appropriate message.

2. Event: The job wait queue is not empty.

Action: Among all the jobs in the wait queue, find the job that is expected to take the least time to complete (***FindShortest()***) – i.e., a job with the minimum value for *n_ticks*. Let J_i denote the selected job, and p_i denote its *n_procs*. Now, check if there are at least pi processors currently available in the processor "free pool" (***CheckAvailability()***). If not, then leave the wait queue intact and return. Otherwise:

a. remove J_i from the wait queue (***DeleteShortest()***), and
b. remove p_i number of arbitrary processors from the free pool and "assign" them to that job (***RunJob()***), so that they can start executing the job from the next tick. When a job goes to the running state, along with it a countdown "timer" is started; initialize this to *n_ticks*.

If after the first job allocation, there are still more jobs in the wait queue that have sufficient available processors to run on, then iteratively keep removing all the next shortest jobs until no more processors are available for the next shortest job, or the wait queue becomes empty.

Note: remember your code does not have to concern itself about running the actual job. You can safely assume that will be internally taken care of by the individual processors that you assigned to run the job. Your responsibility is to make sure that the scheduler is aware of how many processors are now available in the free pool after the allocation and how many processors this new job that got moved to the run queue is using

3. Housekeeping Event: The running job queue is not empty.

Action: The job timer for ALL running jobs is decremented by one (***DecrementTimer()***).

4. Housekeeping Event: A job that was in the running state completed during this tick – i.e., its timer has reached a value of zero after decrement.

Action: The scheduler should release all the processors attached to this job, back into the free pool so that they become available for other jobs during later ticks (***ReleaseProcs()***).

4. API and Testing

Follow the below instructions to test your code.

- Write a separate TestDriver.cpp program which first reads an input file, in which a series of job inserts are mentioned, one line per tick. An example input is shown below:

  ```
  J1    8     10
  J2    2     1
  J3    12    12
  J4    10    2
  J5    5     8
  J6    4     2
  J7    4     6
  J8    2     5
  J9    4     3
  J10   6     2
  ```

  Your code should iterate through these lines, one tick per each line, and adding any more iterations (i.e., ticks) as necessary until the Wait Queue becomes empty. Within each iteration, the current tick number should be displayed (starting at serial number '1'). Then the code should call the ***Tick()*** function (as specified in the pseudocode in Figure 2. Define the ***Tick()*** function as a member function of your Scheduler class.

- At the end of the ***Tick()*** function, print a system-defined job_id number unique to any job submitted during this tick or an error message upon failure to insert. Also print all jobs which got allocated to the running queue in this tick, and also any job or jobs that completed during this tick. Each print message should include the job description, number of processors it needs, and the number of ticks it needed.

- The test program should terminate if the user inputs “exit” for the job descriptor in the standard input.