1 Project

For the programming project, please make sure that you turn in a complete project. This means that you should include a

- Makefile
- Readme File: tells me how to run your program.
- Summary: includes:
  - what you did
  - an overview of the design and the data structures used
  - include the results of your sample runs here
  - if you consult outside sources, include a bibliography or citation.
  - you can work in groups (as usual): only submit one result from each group. Include all names on your work. Submit only 1 file (archive it!)

Your programs will be critiqued based on correctness, design, and imlementation (both style and efficiency). Your source code documentation and the summary will also form a part of the evaluation.

This assignment is aimed at creating an application for priority queues. You can choose any implementation of a priority queue as presented in Weiss' book. You may also use other data structures that we've discussed. Don't use a more complicated data structure when a simpler one can suffice. Justify why you chose the data structures you did.

You are to design and implement a simple shortest-job-first scheduler for a p processor parallel computer (cluster) that allows multiple users to access the cluster at the same time.

The scheduler performs all required functions at regular timestops (ticks). At each tick, various events can happen and the scheduler must address all these events before moving on to the next tick. So (in your simulation) each tick could take a variable amount of time to complete. The various events are:

- *Event*: The user inputs one or more jobs at the start of any tick. Each job arrives to the scheduler in the follwing format: (job_description, n_procs, n_ticks).
- *Action*: For each job submitted, the systems assigns the job a new, unused integer id (the job_id) and then calls InsertJob(...) with the following parameters: (job-id, job-description, n-procs, n-ticks):
  - job-description: contains the details as to which program to run, such as "pgm-name pgm-args". It is a string type.
  - n-procs is the number of processors that this job needs to run (eg these are the processor resources).
  - n-ticks is the number of ticks that this process will need on each of the n-processors. This is equivalent to the total amount of time (running time). The job will be launched simultaneously on all n-processors to run in parallel for n-ticks.

  Insert-Job checks that the number of processors required is less than or equal to the total number of processors and the number of ticks is non zero. Then it should be inserted in the the wait queue.

- *Event*: the wait queue is not empty:
- *Action*: Implement the shortest time first scheduling algorithm, modulo processor availability. If there are not enough processors available, you can return and leave the queue untouched (waiting for the currently scheduled process to finish and release the processors.)

  If the job can be scheduled:

  - Remove the job from the wait queue .
  - Remove p_i processors from the free pool of processor resources and assign them to the job. The job goes into the running state, and will begin running at the next tick. Start a count_down time for the job (initially n-ticks).
- *Event*: Tick
- *Action*: At each "tick" when the run queue is not empty, decrement the count-down timer for each job.
- *Event*: Tick
- *Action*: At each "tick" if a count-down time has reached 0, release all processors back into the free pool

2 The program

(See Figure 1 for the flow diagram and Figure 2 for the pseudocode.)

You should write a driver program and a scheduler class. Tick() should be a member of the scheduler class. The driver program runs forever to calling the Tick function at each iteraiton. At each iteration you should:

- call the Tick() function.
- Display at each tick the waitqueue, the run queue, the number of elapsed ticks and free proccessor pool.
- At each tick, accept new job requests from the user.
- At each tick, perform the tick functions as specified above.
- At the end of each tick, display the job-id's as assigned by the tick functions for the jobs inserted at that tick.
- If any job completed during the tick, print a job report (how many ticks used, processors used)

Terminate when the user enters "exit".

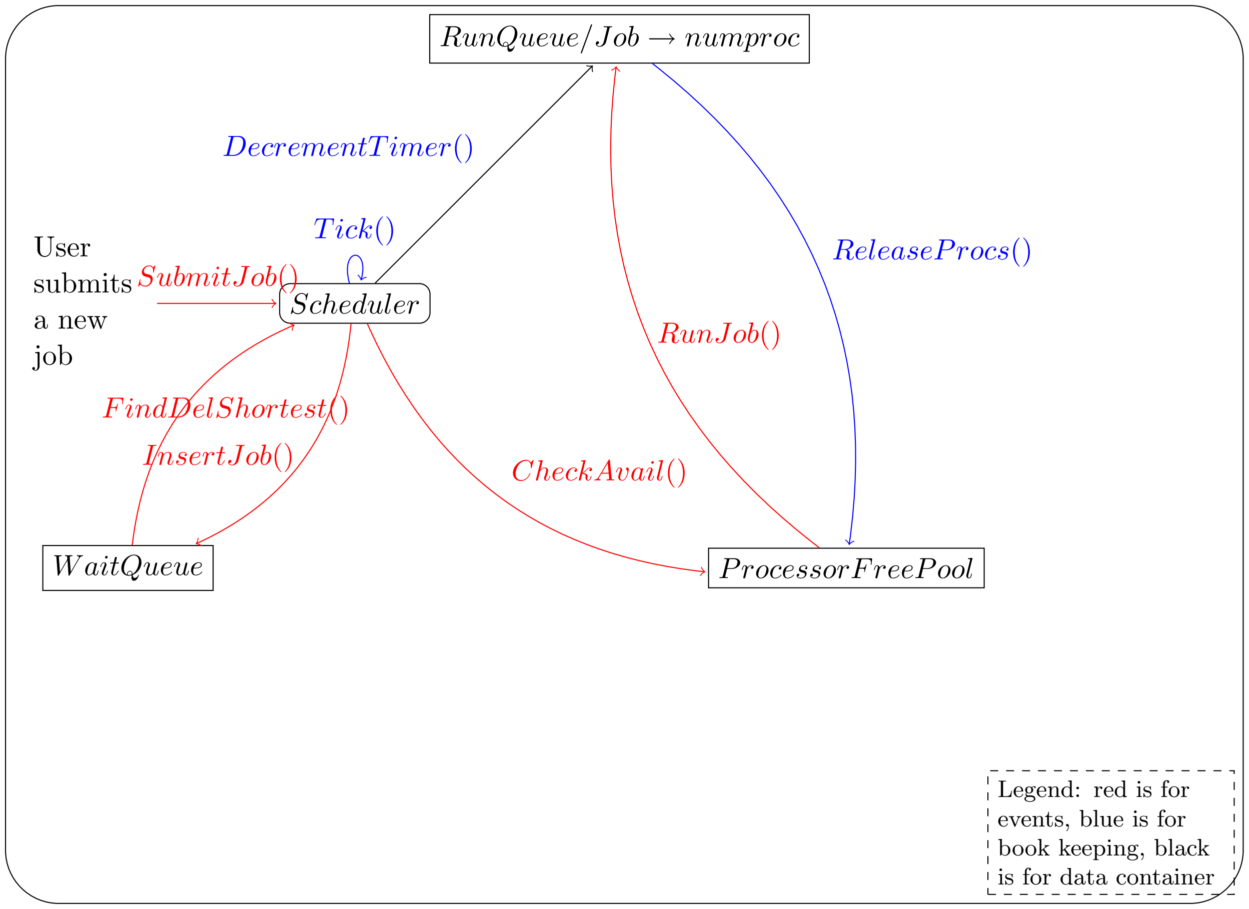

Figure 1: Scheduler

3 Summary

Your summary should include the design, and summary of the results, notations on your implementation (as mentioned above.) Make sure you describe your three main data structures: (wait queue, run queue, free pool) .

Your summary should also include an analysis of the runtime behavior. Make a table giving the worst-case run-time complexities in your implementation for each of the event functions: InsertJob, FindShortest, DeleteShortest, Check- Availability, RunJob, DecrementTimer, ReleaseProcs. You can use n to denote the number of maximum number of jobs (waiting or running) at any given point of time, and p to denote the total number of processors (free or busy).

Lastly, your summary should include a critique of your design and implementation. what are the main shortcomings/bottlenecks of this shortest-job-first strategy, from both a performance and functionality point of view. Support your claims with an example.

4 Extensions

Optional extension: How would you alter the design to handle priority aging? Adding time quantum? Altering the time quantum to allocate different amounts to different priority classes?

```
1. Prompt the user for submitting new jobs, one at a time.
2. Insert each of those jobs into the wait queue.
3. Decrement timer for all running jobs in the run queue.
4. Release all the processor corresponding to the completion of a running job back to
   the free pool. (A job has completed if its timer has gone to 0.) Remove from the run
   queue.
5. Find the next shorted job ($\langle j_i, p_i, t_i ramngle$) from the waitqueue, check if the number
   of processors are available in the free pool. If so:
     • remove job $j_i$ from the wait queue.
     • insert job $j_i$ into the run queue, and assign $p_i$ processors from the free pool, and
       initialize its timer.
```

Figure 2: Tick() Pseudocode

Optional extension: include resource scheduling: besides the number of processors available include other resources required. Implement one of the deadlock detection/prevention algorithms.