Program #4 (Cpt S 223)

Fall 2011

Due: November 14, 2011, 11:59pm on Angel

General Guidelines

- All source code must be in C++. You can use Windows or Unix environments at your discretion.
- Each program project should be accompanied by a COVER SHEET (see details below). Assignments without cover page will NOT be graded.
- For this programming assignment, you have the option of either work as a TEAM if you prefer, or as an individual. A TEAM should contain at most 2 students including yourself.
- In the rest of this document, the term "YOU" is used either to refer to you (if you are working individually) or your team (if you work as a team).
- The final material for submission should be entirely written by you. If you decide to consult with others or refer materials online, you MUST give due credits to these sources (people, books, webpages, etc.) by listing them on the cover sheet mentioned below. Note that no points will be deducted for referencing these sources. However, your discussion/consultation should be limited to the initial design level. Sharing or even showing your source code/assignment verbiage to anyone else in the class, or direct reproduction of source code/verbiage from online resources, will all be considered plagiarism, and therefore will be awarded ZERO points and subject to the WSU Academic Dishonesty policy. (Reproducing from the Weiss textbook is an exception to this rule, and such reproduction is encouraged wherever possible.)
- Grading will be based on correctness, design and implementation efficiency, coding style, exception handling, source code documentation, and the written report. Try to also follow the good coding practices discussed in class during the C++ review lecture.
- If working as a team, the same grade will be given to both members of a team.
- If working as a team, submit only one copy of your assignment into the Drop box and indicate in the text box of the drop-box while submission, both names in the project. If you specify only one name then only that person will get the grade and the other will not. So please make sure you fill in both names.

- If working as a team, remember to include both your last names in the zip file of your submission.

- **Angel DropBox submission:** We will use DropBox. All assignments with cover sheet should be zipped or tarred into one archive folder (named after your last name). Instructions for DropBox submission: I have set up a have set up a new DropBox called “Program project 4” inside Angel. The way to submit to the dropbox is as follows:

  1) Once you login to Angel, click on the “Lessons” tab.

  2) You will see “Program project 4”. Click on that and you will see the submission website. Under Title, just write your first and last name and say Program 4 - e.g., “Ananth Kalyanaraman, Program 4”. If you are submitting as a team, type in both your names in the textbox. Otherwise leave it empty. Then click on the ”Attachments” button and attach your ZIP file there. And then press the SUBMIT button.

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 structure that we have discussed so far in class, as long as the use of such a data structure could be justified from a design perspective. In fact, go for a simpler 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.

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 @ page 252). You are also free to re-use/reproduce any source code that is provided in the Weiss textbook. Most of its source code are available from the following website: `http://www.cs.fiu.edu/~weiss/dsaa_c++/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. 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.)

<u>Init condition:</u> 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 one or more jobs at the start of any tick. Each job arrives to the scheduler in the following format: <job_description, n_procs, n_ticks>.

   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()**[1] function with the following parameters:

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

   (i) "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;

   (ii) "n_procs" is the number of processors that this job needs to run on;

[1] <u>Note:</u> 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.

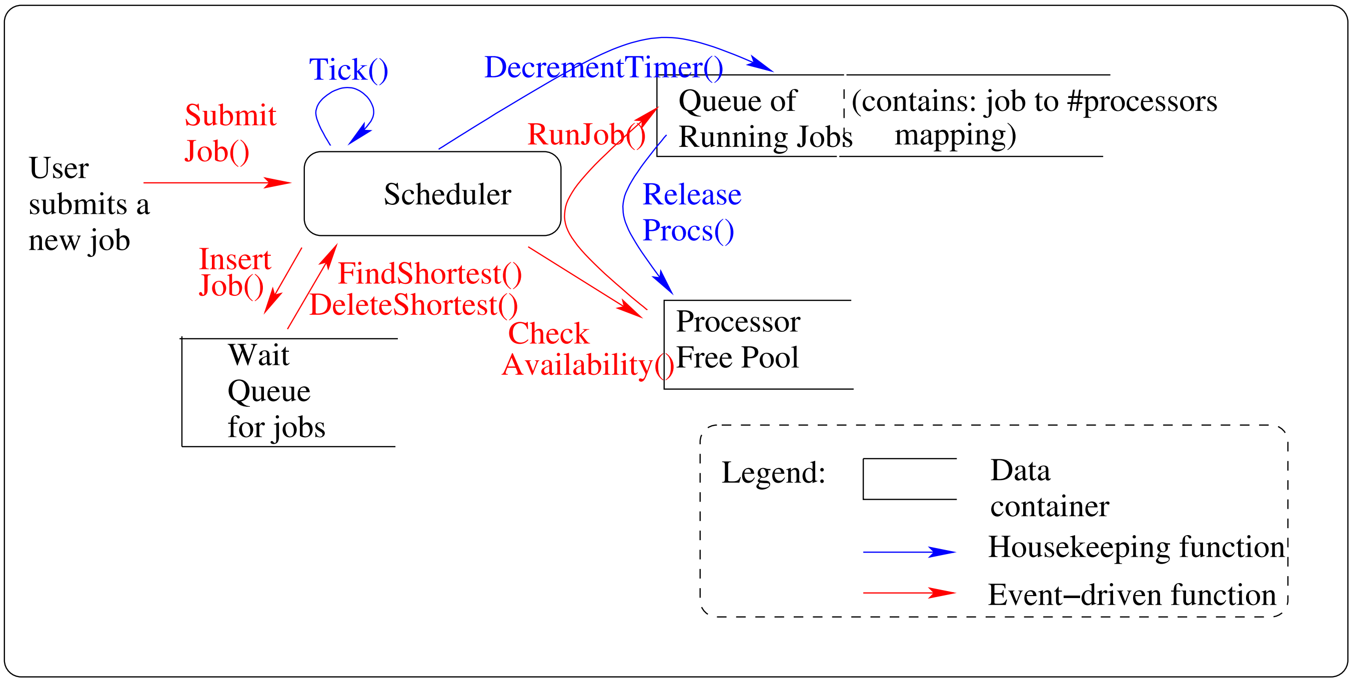

Figure 1: Organization of the scheduler system for a parallel cluster that has a total of p processors.

Algorithm 1 *Pseudocode for the scheduler' Tick procedure*

Tick()

0. Prompt user for submitting new jobs, one at a time.
1. Insert each of those jobs into the wait queue;
2. Decrement timer for all running jobs in the running queue;
3. Release all the processors corresponding to the completed/expired jobs from the running queue to the free pool. A job is said to have "expired" if its timer becomes zero during this tick. Also, any expired job should be automatically removed from the running job queue;
4. Find the next shortest job ($< J_i, p_i, t_i >$) from the wait queue and check if sufficient number of processors are available in the free pool to run it. If so:
   - Remove job J_i from the wait queue;
   - Insert job J_i into the running queue and assign p_i processors (from free pool) to that job;

Figure 2: Sequence of steps to be carried out during each Tick() of the scheduler.

(iii) "n_ticks" is the number of ticks that this job is going to need on each of the n_proc 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.

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. If there are more than one job to choose from this category, then pick any one of them. Let J_i denote the selected job, and p_i denote its n_procs. Now, check if there are at least p_i 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[2]. When a job goes to the running state, along with it a countdown "timer" is started; initialize this to n_ticks.

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()**).

[2] 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.

4 API and Testing

Follow the below instructions to test your code.

- Write a separate main.cpp program which runs an indefinite number of iterations. Each iteration corresponds to one tick. In each iteration, the current tick number should be displayed. Then the code should call the Tick() function. Define the Tick() function as a member function of your Scheduler class.

- At the start of each tick, the program should prompt the user to input an arbitrary number of jobs through the standard input, one job at a time. Basically prompt the user for three entries: (i) Job descriptor (a string), (ii) n_procs (an integer), and (iii) n_ticks (an integer). After each entry the user will press enter to signal the end of that entry. If the user has no more new jobs to submit, then the user will signal that by entering the string "null" for job description.

- Then the tick procedure should carry out the remaining steps outlined in the Tick() pseudocode in Figure 2.

- At the end of the Tick() function, display back a system-defined Job_id number unique to each job submitted during this tick or an error message upon failure to insert.

- If there was any job that completed during this tick, then *print* its job_id along with information on how many processors got released. Also, at the end of the tick, print separately the following items:

  i) the contents of the wait queue (just the job ids),
  ii) list of all currently running jobs (job ids, present timer value, number of processors being used), and
  iii) the number of processors in the free pool.

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

5 Design Document - (2 page limit)

Along with your source code, also submit a one-page design document (as Word Doc or PDF - no other formats allowed).

In the first part of this document, provide a figure that best describes your design for the scheduler system. It should basically be a modified version of Figure 1, in which you should state what data structures/STL containers you used to implement the three data containers (wait queue, running job queue, free pool) .

In the second part, make a table to simply state the worst-case run-time complexities in your implementation for each of the event functions: InsertJob, FindShortest, DeleteShortest, CheckAvailability, RunJob, DecrementTimer, ReleaseProcs. No need for justifications. Just write the answers using the Theta (or Big-Oh if Theta is not possible) notation. 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).

In the third part, write a short paragraph about what you think are the main shortcomings/bottlenecks of this shortest-job-first strategy, from both a performance and functionality point of view. Support your claims with brief but compelling rationale.

6 FINAL CHECKLIST

___ Cover sheet (just one per team)

___ separate folder containing all your sourcecode (including the main function you used for testing)

___ 1-page design document

___ All the above zipped into another folder archive called Program4¡YourLastName(s)¿.zip