Program 3 (Cpt S 223)

Fall 2008

Due Date: November 10, 2008, 5pm on eLearning

1 GENERAL GUIDELINES

- This submission should be accompanied by a cover sheet which is a single page document that contains all the information listed in this sample cover sheet: Word or PDF. Assignments without a cover sheet will NOT be graded. Use the same cover sheet template as in your previous two program projects.
- For this project, you may work in teams of up to two individuals (including yourself). This is not required although teamwork is encouraged. Each assignment will be graded the same way regardless of whether it was a team work or not. You can choose your partner.
- You may consult with others or refer materials online, but you MUST reference your sources (people, books, webpages, etc.). These sources must be listed on the cover sheet mentioned below (no points will be deducted). However, the final material for submission should be entirely written by you. Directly reproducing source codes or verbiage from online resources or other's people's assignments will 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 is encouraged wherever possible.)
- Grading will be based on correctness, coding style, implementation efficiency, exception handling, source code documentation, and the written report.
- **Submission:** There should be one submission per team. All assignments with cover sheet should be zipped or tarred into one archive folder (named after your last name(s) as follows: "Program3<LastName1_LastName2>.zip"), and attached by email and sent to All Instructors and All Teaching Assistants through the eLearning website. Remember to CC to your team member as well. Submissions are due by 5pm. A penalty of 10% will be assessed for late submissions that arrive within the next 24-hour.

  Note: Submissions will be accepted only through eLearning. Any submission outside the eLearning portal will be discarded. Also, we will not be able to send out acknowledgements for each submission. If there is something missing in your assignment/attachment we will contact you directly.

2 PROBLEM 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 contain any other data structure we have discussed so far in class, as long as the use of such a data structure is justified. In fact go for a simpler data structure wherever possible. An unnecessarily overcomplicated implementation (example: using a binary heap where actually a simple vector or linked list would suffice) could lead to reduction in points. The idea is to let you make the design choice, and identify the most suitable data structure for each system component that will effectively address performance and functional tradeoffs. You can re-use any source code that is provided in the Weiss textbook. All of its source code are available from the following website: `http://www.cs.fiu.edu/~weiss/dsaa_c++/Code/`.

THE 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 access the cluster at the same time, as per the specifications described below. Figure 1 is a conceptual illustration of the different components in the system, how they interact, and the necessary data containers. Refer to this figure while reading the specification.

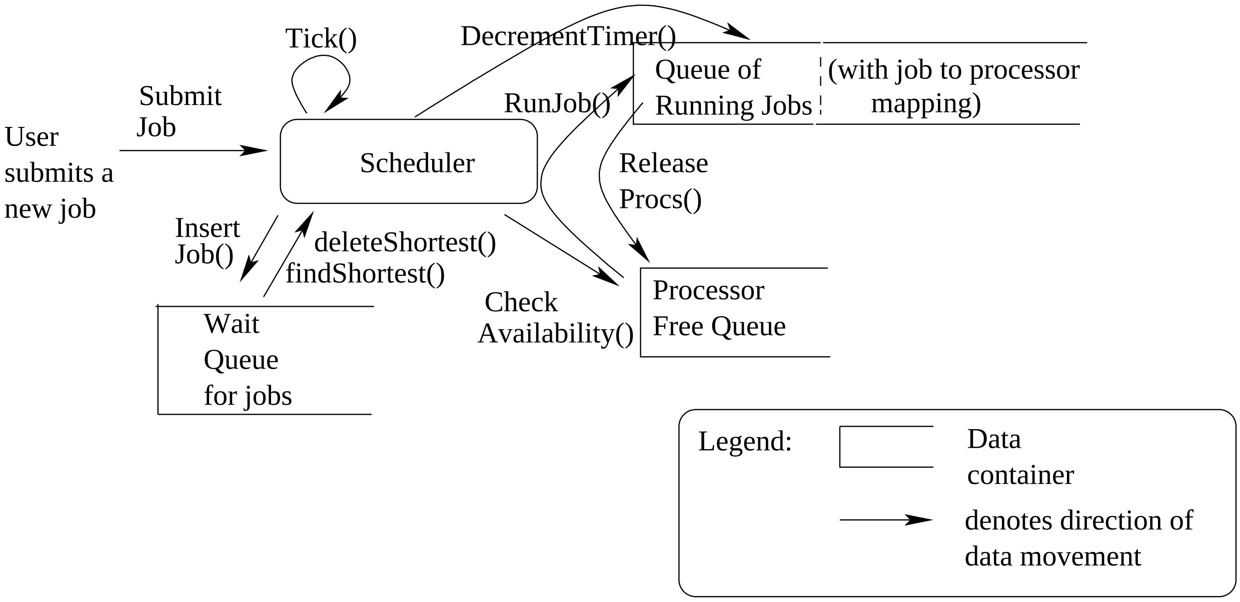

Figure 1: Organization of the scheduler system.

Algorithm 1 *Pseudocode for the scheduler' Tick procedure*

Tick()

1. Insert all user-submitted new 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 queue. A job is said to have "expired" if its timer becomes zero;

4. Find the next shortest job ($< J_i, p_i, t_i >$) from the wait queue and check if sufficient number of processors is available to run it. If there are sufficient processors (ie., $\geq p_i$):

   - Remove job J_i from the wait queue;

   - Insert job J_i into the running queue and assign the p_i processors to that job;

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

3 PROBLEM SPECIFICATION

The scheduler executes all its functions in timesteps or "ticks". This is, in other words, 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. You can assume that there is sufficient time within each tick to accomplish this. 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.)

- **Event:** A user can input 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>. Upon submission, the system assigns this job a new unused integer id called "job_id" and then calls InsertJob() with the following parameters: <job_id, job_description, n_procs, n_ticks>. Here are the meanings of these fields:

  (i) "job_description" contains details on what command to execute to run the job (i.e., "<program name> <its arguments>"). It can be of a string data type;

  (ii) "n_procs" is the number of processors that this job is going to use;

  (iii) "n_ticks" is the number of ticks that this job is going to use to complete on each processor. Note that this will be the same as the overall job run-time because the job is going to be executed concurrently on all its processors.

  Action: The scheduler first checks if ($0 <$ n_procs $\leq p$) and (n_ticks> 0). If so, it inserts the job to a " wait queue". Otherwise, a job submission error is raised.

- **Event:** The job wait queue is not empty.
  Action: Among all the jobs in the wait queue, find the job that will take the least time to complete (i.e., with the minimum n_ticks). If there are more than one to choose from, then pick any one of them. Let J_i denote the selected job, and p_i denote its n_procs. Check if there are at least p_i processors currently available in the processor "free queue". If not, then leave the wait queue intact and return. Alternatively, if sufficient processors are available, then (i) remove J_i from the wait queue, and (ii) remove p_i arbitrary processors from the free queue and assign them to that job, so that they can start executing the job from the next tick. In the process, the state of the p_i processors that got assigned to this job should be changed to a

running state so that they are not available for newer jobs. The idea is to make sure each job has exclusive access to all its processors. When a job goes to the running state, a countdown "timer" is started; initialize this to n_ticks.

- **Housekeeping Event:** At the end of each tick, the job timer for each running job is decremented by one.
- **Event:** A job that was in the running state completed during this tick — ie., its timer becomes 0.

  Action: The scheduler releases all the processors that were assigned to this job back into the free queue so that they become available for other jobs.

Note that a processor has to be in either the busy state or free state (but not both) at any given point of time. When the algorithm starts, the system is initialized such that all the p processors are in the free queue. Figure 2 shows the sequence of steps that the scheduler should perform do during each tick as a pseudocode.

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 allow the user to input one or more new new jobs through the standard input, one job at a time. Basically prompt the user for three fields: (i) Job descriptor, which is a string (with last character as end line), (ii) n_procs, which is the number of processors requested, and (iii) n_ticks, which is the number of ticks requested on each processor. Once a job is input, your code should do the necessary steps to insert the job into the wait queue and display back a system-defined Job_id number unique to that job upon success or an error message upon failure.
- Then the tick procedure should carry out all the steps outlined in the remainder of the Tick() pseudocode in Figure 2, in that order.

- 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 contents of wait queue, the free queue, and the queue of running jobs. For the wait queue, print all the job_ids. For the free queue, print only the number of processors in the queue. For the queue of running jobs, print the following information for each running job: (i) Job_id, (ii) number of assigned processors, (iii) its timer (i.e., number of ticks remaining to finish).

- The program should terminate if the user inputs "-1" for the job descriptor in the standard input. Right before it terminates, the program should report the average number of ticks that each job had to wait in the wait queue before it got moved to running status.

5 DESIGN DOCUMENT

Along with your source code, also submit a one-page Word design document.

Here, illustrate a figure that best describes the design of your scheduler system. It should basically be a modified version of Figure 1, in which you should state what data structure you used to store the three data containers. Interactions between modules should be denoted by arrows and appropriate labelled. Be as creative as possible. There is no one "correct" way to show this picture. The main iideas in your design should come out through this picture; avoid crowding it with information.

In the second part, add a couple of lines about what you think are the main performance shortcomings/bottlenecks of this shortest-job-first strategy. Support your claims with compelling rationale.

6 FINAL CHECKLIST

In one zipped file include:

___ Cover sheet

___ Source code folder with main.cpp and all project files

___ 1-page design document