simQ

Simulating a Time-sharing Computer

Summary
This program will simulate a **round-robin CPU scheduler**. Computers that allow more than one user to be logged in at a time must "share" the CPU processing time among users' jobs. Each job (for example, an email job, a compile job, etc) waits in a queue for the CPU; when it is a job's turn to run, that is, it is at the *front* of the queue, the job is deQueued and granted a **time-slice** of processing time. At the end of a time-slice, either the job is finished or it is enQueued to wait another turn for another time-slice.

"No, sonny, we're not time-sharing - hoppit!"

This assignment asks you to simulate a round-robin job scheduler, that is, the part of the operating system that manages the queue of waiting jobs.

Data Structure
You will use a Queue to hold waiting Jobs (a completely new class). Each job will contain at least:

- id (integer)
- job name (a string)
- total time it will take for the job to be completed (Real, sec)
- time remaining for the job to finish (Real, in seconds)

Creating "new" jobs to enter the queue – `Job::Job` (the CTOR)
The types and length of time for jobs entering the queue will be randomly generated. For each new job to enter the queue, the *values* of the members in the Job object are to be determined as follows:

id # -- assign an integer from $1 \le n \le MAX$ (e.g., 100) and where no two jobs on the queue should ever have the same id # (*careful: how will you implement this unique-id feature? To start, don't get hung up on this point but remember to come back and address it later*)

Since this is a simulation, we will assign job **types** on a random basis:

job type	probability of occurrence	total time for job in seconds (double)
print	0.20	$1 \le t \le 5$
compile	0.30	$10 \le t \le 100$
email	0.40	$3 \le t \le 20$
other	0.10	$0 < t \le 45$

Obviously, you will have to generate random numbers (see `rand()` or see the `randGen` class in medPing. What? You mean actually *reuse* code? You betcha').

Input (prior to starting the simulation, that is, at the very beginning)
Three Real numbers from standard input (`stdin`).

(i) time slice to assign to each job (seconds)
(ii) length of time to allow new jobs to enter the queue (Part II only)
(iii) probability that a new job enters the queue while another job is in the CPU (e.g., an input of **0.05** means 5% chance that a new job joins the queue <u>*while*</u> another job is "in" the CPU using a timeslice) (Part II)

Part 0

You must get the `Job` class working. Use the suggested schedule of work at the end of this specification to help you finish on time. Yes, the `Job` class is a class that you must write!

Part I (70 points)

In main(), create a Standard Template Library (STL) queue of JOBs (`queue<JOB>`) and store (enQueue) five(5) jobs on the queue. Then, process the jobs until the queue is empty using the user-entered timeslice. For each timeslice, you must report the status of that job. Below is a sample of output. Assume five jobs have been put on the queue and the simulation is using a **timeslice=0.1sec**. Assume Job#32 (print) needed a total of 4.8s and Job#12 (email) needed a total 0.48s before it finishes.

(*assume these two jobs have entered the CPU four times already ...)*

```
-------------------------------------------------------------------------------------------------
Job #32 is DeQueued and enters the CPU with 4.40 seconds remaining ...
Job #32  --  Print -- is now in the CPU
Job #32 leaves CPU with 4.30 seconds remaining (0.50 seconds total CPU time used so far).
Job #32 is EnQueued again
-------------------------------------------------------------------------------------------------
Job #12 is DeQueued and enters the CPU with 0.08 seconds remaining ...
Job #12  --  Email -- in CPU
Job #12  --  Email  - - has finished without using all the timeslice; CPU idles for 0.02 sec
Job #12  --  Email  -- has 0 seconds remaining (0.48 seconds total CPU time used so far).
Job #12 is FINISHED
-------------------------------------------------------------------------------------------------
```

Part II (20 points) -- assumes Part I is working

Process the jobs for as long as required by the user-entered value indicating how long the simulation should accept new jobs (see second input value). For each timeslice, you must process the next job on the Queue; if there are no jobs on the queue, you should record that the machine spent this time slice on *idle*. At the end of the simulation, you will print the total amount of idle time. In addition to processing the next job in each timeslice, you must determine if (another) new job has entered the job queue *during* this time slice (see third input argument). A scenario would be: while working on Aunt Fee's email job (that is, Aunt Fee's job is getting a timeslice of the CPU processor), Uncle Foo starts a brand new print job. To ensure *unique* memory locations for new jobs, use this code:

```
if (new job arrives)
{
   JOB *newJob = new JOB( nextID );
   Q.push( *newJob );
}
```

Once the simulation has stopped accepting new jobs (second input), you should proceed as in Part I, that is, process the jobs until the queue is empty but you do *not* allow new jobs to enter the queue (after time has proceeded beyond the second input value).

Part III (10 points) -- assumes Part II is working

Using a spreadsheet (e.g., Excel), make an xy-plot of **time vs. (number of jobs on the queue)**. Thus, you'll have to modify your program to write to a file (to a comma-separated-value .csv file) the number of jobs on the queue *for each timeslice*. Once you begin work in Excel (save your file as an Excel worksheet, not .csv file or you could lose your formatting), label your axes, title your graph (see example on next page), and attached a typed(!) report explaining your graph. In particular, refer to the slopes of the graph in your explanation. Notice how I added a textbox and arrow on top of my graph for additional *annotation*. Also notice that the legend under the graph is professional done.

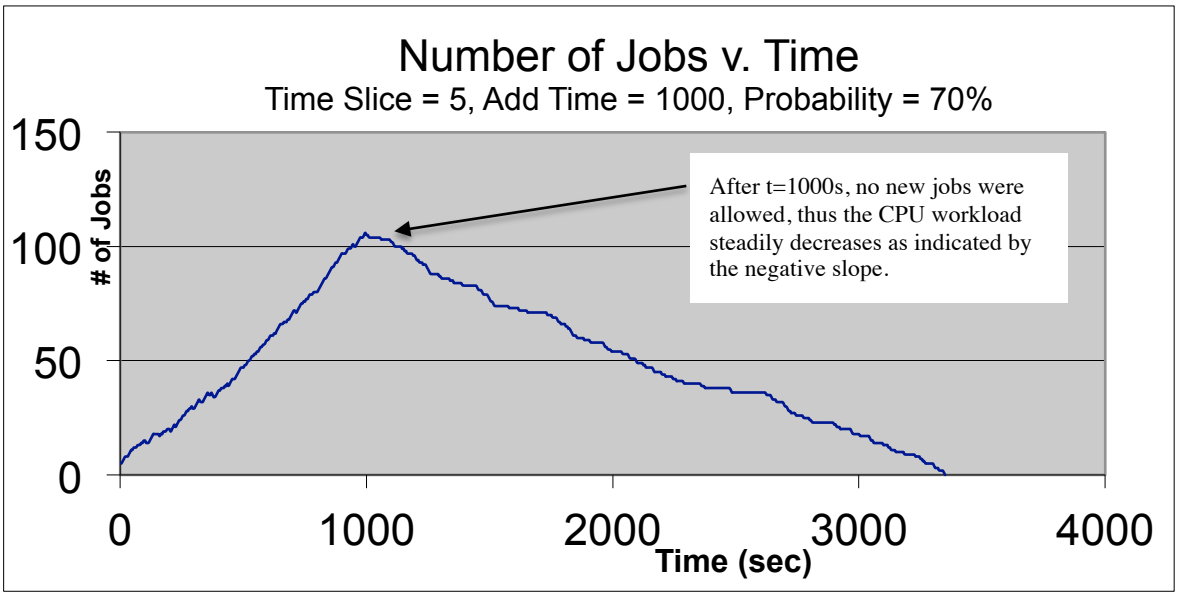

Figure 1. Graph showing an increasing number of jobs for the initial 1000 seconds of a timesharing simulation where each job has a 5 second timeslice and new jobs can emerge 70% of the time on one timeslice.

FAQ:

1) **Do I document this program?**
   You bet! In fact, insert your documentation *as you enter your software*. A good rule of thumb is that if the code is tricky for you, then you can believe that it will be (very) tricky for someone else to understand it.

   Also, write a strong SUMMARY at the beginning. Explain in clear terms what the program simulates. **Don't say *how* it does it, do say *what* it does**. Make an opening statement of 20 words or less first and then elaborate. For example, do *not* say: "*This program uses a while loop ... an STL queue ...*" Someone who does not know C++ should be able to read your opening documentation and at least understand what the simulation is doing. But of course you absolutely *must* comment where you are declaring your data structure (queue) inside your code.

2) **Should I declare constants?**
   But of course! In fact, you'll need LOTS of constants. Everytime you type in a numerical constant into your code, slap yourself on the wrist and then declare a new **`const`**, perhaps (if you deem it appropriate) put all your constants in a file called `constants.h`

3) For the third input (Part II only), does a user (at the keyboard) enter 0.05 or 5 for five percent?
   0.05

4) **What if the queue is full when I try to allow a new job to enter?**
   Well, if the queue *really* is full, you'll have to reject the new job. Print an appropriate message that says the computer is too busy and cannot handle any more work at this time.
   Note: but think about your data structure … will it *get* full?

5) Should I use arrays of characters or the (built-in) string class?
   Unless you must do otherwise, always use **string** rather than arrays of characters

6) If a job has 0.5sec remaining and the simulation is using a timeslice of 2.0sec, what happens with the remaining 1.5sec of the timeslice?
   Good question. For this simulation, we will "waste" the 1.5sec, that is, we'll assume the machine just sits and idles until the timeslice is up.

7) To what precision should I print my output?
   Print all times to the hundredths, e.g., 3.20 seconds (not 3.2000000)

8) Are we using medPing for this program?
   No need. Just use an Xcode console project or Visual Studio on Windows.

9) Is there a Starter Kit for this program?
   Yes. See our moodle (onCourse) website.

NOTE: This program is significantly more involved than any program that I have assigned to date. Begin an algorithm for the `JOB::JOB` CTOR immediately and proceed with coding and testing of that function, that is, can your `main()` just "create" a new `JOB` object and you print out the type of Job? You may want/need more than one CTOR, right? Perhaps you also want a method to explicitly ask to "reset" an object to a 'new' job, e.g., `JOB::createNewJob()`? Below is a sample schedule for you to follow:

1) define your `JOB` class (you start `Job.h` and `Job.cpp` and of course you'll need `main.cpp`)
2) test `JOB` from `main()` … by making small JOB objects, testing your getter/setter methods

Finish #1-2 by Monday, March 17

3) Design, write, and test the `createNewJob()` function.
4) Test your STL `<queue>`; enqueue five new jobs. Print out the five jobs on the queue.
5) Do #4 above and then dequeue a job, subtract the timeslice, and enqueue the job back on the queue if the job is not yet finished.

Finish #3-5 by Wednesday, March 19

6) Complete Part I.
7) Proceed to Part II.

Finish #7 by Friday, March 21

8) Save some time to test your simulation. Do you have unique IDs? Whether you do or not, make sure you document all your methods and main() clearly.

9) Create output for Excel.
10) Format your Excel table, copy into Word, and make your report to be neat and professional.
11) Save your final report in a .pdf file. Include the .pdf file in the folder that you submit to onCourse.

DUE on Tuesday, March 25