CS 201R
Fall 2020
Program 6

This is it! The last program of the course. This is where you put together everything you've learned into a larger, more significant program. Along the way we'll be looking at an issue to give you a preview of some of what you'll be dealing with in CS 303, Data Structures.

For this program we'll be doing a *discrete-time simulation.* What this means is that for purposes of the simulation, time advances in separate discrete 'ticks;' in this case, each tick represents one minute. The basic pattern is simple: At each tick of the clock, we deal with any events that happen during that tick—in this case, arrival of new jobs and completion of old jobs for a print shop. These events are considered to happen 'all at once,' because all we're interested in is the state of things at the end of the tick compared to the beginning. If one job arriving and another being completed both happen during the same clock tick, we don't worry about whether one of them arrived 30 seconds before the other or which one was 'really' first, as long as both happened during the same tick. We just compare the state of things after the tick with the state of things before it.

A university print shop has a high-speed high-quality printer, capable of 150 pages per minute, along with stapling, binding, and various other features. Jobs arrive at random, either submitted online or from walk-in customers. Under most circumstances the staff can keep up with demand, but at other times jobs arrive faster than they can be processed and a backlog results. We're going to look at various strategies for managing the backlog.

While *queuing theory* is a large field of study within computer science, we're going to compare 3 simple strategies:

- First-in, first out (FIFO), otherwise known as first-come, first-served. Jobs are handled in the order of arrival, regardless of size or category.
- Shortest-job first (SJF). In this strategy, when we're able to start a new job, we look at all jobs that are waiting and select the smallest. You've seen this at the checkout line of the grocery store, where someone with a full cart lets someone with just a few items go ahead of them.
- A multi-level priority queue. Each job is classified on arrival as coming from either Administration, Faculty, or a Student. Each classification has its own FIFO queue. Administration jobs are handled first; if there are no Administration jobs pending, then the next Faculty job is selected; if there are no Administration or Faculty jobs pending, Student jobs are handled. (This option was included at the suggestion of several members of the Administration. Oddly, students were not consulted.)

The print shop is open from 9 AM to 5 PM—eight hours, or 480 minutes. Once jobs stop arriving, all pending jobs still need to be done before people can leave for the day.

You have a data file showing one day's job submissions. Your task is to write a program that will compare each of these strategies. You will find the total waiting time, shortest and longest individual waiting times, and the length of the average wait, for each category, under each of the queuing strategies.

Each job is on its own line, and consists of 3 entries: The arrival time (as a clock tick, an integer in the range 0-480, sorted), the category ('A', 'F', or 'S'), and the size of the job in pages. The printer can handle 150 pages per minute but there is also a certain amount of time needed to set things up, load

paper, etc. To find how long a job will take, take the number of pages, divide by 150 (keeping fractions), add 1 minute for setup and so forth, and *round* to the nearest whole number.

The basic strategy for each run of the simulation is:

- Read the data file, which is already in chronological order, with time in ticks. This data should go into a queue or list. Your program will keep 2 queues active at any one time: One of input data (all jobs, including those not yet arrived) and one of jobs that have already arrived but have not yet been handled—this will be your FIFO, SJF, or Multi-level queue.
- Each minute:
  - Check the front of the input data to see if any new jobs are arriving this minute. If so, add them to the 'jobs pending' queue and remove them from the data queue.
  - If the printer is available:
    - Take the next job from the jobs-pending queue. Note how long it has been waiting (current time – arrival time; yes, if the job arrived during this same tick, then the waiting time is 0), and update statistics as appropriate (you will need the number of jobs handled, the longest wait, and the total waiting time, for each category of job.)
    - Based on the size of the job and the current time, compute the clock tick when the printer will next be available (current time + service time).
    - Update the status of the printer as either busy or not
  - Increment the clock.
- Report the summary statistics.

Programming notes: This is your chance to show off how much you've learned this semester about object-oriented programming. For example, you may want to define a PrintQueue abstract class and then derive 3 child classes from it, one for each type of queuing strategy; your main code then becomes very simple. You can write one function that takes the input data (either already in a queue, or just a reference to the stream), an output stream, and the PrintQueue to be used for that run.

You may find the standard template library's queue or list classes useful. Or you can build data structures of your own, if those don't do quite what you need, or use them as building blocks. For the SJF queue, you may want to store data in a vector that you can keep sorted by size.

Compare the results for the 3 methods. What are the advantages and drawbacks of each? Are there considerations other than raw efficiency we might look at (such as 'fairness,' however that's defined)? Which is more important, the average wait, or the maximum wait? (Also remember that in the results, total waiting time is in job-minutes. If there are 100 jobs pending for 1 minute, that's 100 job-minutes of waiting.) In addition to your code, write up a short document discussing your results, and making a recommendation about which strategy would be best to follow, and why. (There's not necessarily a single right or 'best' answer here; I'm looking for evidence that you can look at some data, think about it for a bit, and come to some conclusions, including thinking about the context of what you're doing.) Don't worry about references or formatting (within reason—you should still use complete sentences, arranged into paragraphs).