Write a program that simulates the service of jobs (programs) by the CPU of a multi-user computer.

Discussion: Before astronauts go up into space, they spend many hours in a spaceship simulator, a physical model of a space vehicle in which they can experience all the things that will happen to them in space. The spaceship simulator is a physical model of another object. The technique that is used to make the model behave as the real object is called simulation. We can use similar techniques to build computer models of objects and events rather than physical models.

A model can be thought of as a series of rules that describe the behavior of a real-world system. We change the rules and watch the effects of these changes on the behavior we are observing.

Let's look at a very useful type of simulation that uses queues as the basic data structure. In fact, the real-world system is called a queuing system. A queuing system is made up of servers and queues of objects to be served. We deal with queuing systems all the time in our daily lives. When you stand in line to check out at the grocery store or to cash a check at the bank, you are dealing with a queuing system. When you submit a batch "job" (such as a compilation) on a mainframe computer, your job must wait in line until the CPU finishes the jobs scheduled ahead of it. The operating system is a queuing system. When you make a phone call to reserve an airline ticket and get a recording that says, "Thank you for calling Air Busters; your call will be answered by the next available operator. Please wait." you are dealing with a queuing system.

Please wait. Waiting is the critical element. The objective of a queuing system is to utilize the servers (the tellers, checkers, CPU, operators, and so on) as fully as possible, while keeping the wait time within a reasonable limit. These goals usually require a compromise between cost and customer satisfaction.

To put this on a personal level, no one likes to stand in line. If there were one checkout counter for each customer in a supermarket, the customers would be delighted. The supermarket, however, would not be in business very long. So a compromise is made: the number of cashiers is kept within the limits set by the store's budget, and the average customer is not kept waiting too long.

How does a company determine the optimal compromise between the number of servers and the wait time? One way is by experience; the company tries out different numbers of servers and sees how things work out. There are two problems with this approach: it takes too long and it is too expensive. Another way of examining this problem is by using a computer simulation. To simulate a queuing system, we must know four things:

1. The number of events and how they affect the system
2. [Bonus] The number of servers
3. The distribution of arrival times
4. The expected service time

The simulation program uses these parameters to predict the average wait time. The interactions of these parameters are the rules of the model. By changing these parameters, we change the rules. The average wait times are then examined to determine what a reasonable compromise would be.

Before you start designing your computer simulation program, let's walk through a simple simulation of a real-life example. Consider the case of a drive-in bank with one teller. How long does the average car have
to wait? If business gets better and cars start to arrive more frequently, what would be the effect on the average wait time? When would the bank need to open a second drive-in window?

This problem has the characteristics of a queuing problem. We have a server (the teller) and objects being served (customers in cars), and the average wait time is what we are interested in observing.

The events in this system are the arrivals and the departures of customers. Suppose that the number of servers is 1, the average transaction takes 6 minutes, and a new customer arrives about every 5 minutes. Let's look at how we can solve this problem as a time-driven simulation. A time-driven simulation is one in which the program has a counter that represents a clock. To simulate the passing of a unit of time (a minute, for example), we increment the clock. We run the simulation for a predetermined amount of time, say, 100 minutes. (Of course, simulated time usually passes much more quickly than real time; 100 simulated minutes pass in a flash on the computer.)

From a software point of view, the simulation is a big loop that executes a set of rules for each value of the clock—from 1 to 100, in our example. Here are the rules that are processed in the loop body:

**Rule 1.** If a customer arrives, he or she gets in line.

**Rule 2.** If the teller is free and if there is anyone waiting, the first customer in line leaves the line and advances to the teller's window. The service time is set to 5 minutes.

**Rule 3.** If a customer is at the teller's window, the time remaining for that customer to be serviced is decremented.

**Rule 4.** If there are customers in line, the additional minute that they have remained in the queue is recorded.

The output from the simulation is the average wait time. We calculate this value using the following formula:

\[ \text{Average wait time} = \frac{\text{total wait time for all customers}}{\text{number of customers}} \]

Given this output, the bank can see whether their customers have an unreasonable wait in a one-teller system. If so, the bank can repeat the simulation with two tellers.

We have described this example in terms of a single teller and a specific arrival rate and transaction time. In fact, these simulation parameters should be varied to see what effect the changes have on the average wait time. Therefore these values are read as inputs to the program.

**Processing**

This is a clock-driven simulation, in which the clock is a counter that is incremented each simulated second. The clock is initially 0, and continues "ticking" until all the jobs in the input have been serviced and left the system. Unlike the simulation in the chapter, which ran for a specified number of clock ticks, this simulation runs until a particular number of jobs have been serviced. This number is input by the user.

[Bonus] When a job enters the system, it is enqueued on a wait queue. If there is room in the CPU's queue, a job is dequeued from the wait queue and enqueued onto the CPU queue.

The CPU's "queue" represents the jobs that are currently being serviced. Jobs in the CPU queue are serviced using the following scheme: although the CPU actually executes only one job at a time, there can be as many as hundred [Bonus: five] jobs in the CPU queue. To service these jobs, the CPU allocates a "time-slice" to each job in the CPU queue, depending on its class. To the observer, it appears that the CPU services several jobs simultaneously; however, most jobs are interrupted when they require I/O or other services. To avoid idleness while waiting for the required service, the CPU starts working on the next job in the CPU queue. Jobs have a time slice of 0.1 seconds.

[Bonus] We simulate this by having each job fall into one of two job classes: I/O bound and CPU bound.

- I/O-bound jobs have a time-slice of 0.1 seconds, while
- CPU-bound jobs have a time-slice of 0.2 seconds.
The job at the front of the CPU queue executes for its required time-slice; then it is dequeued and enqueued at the rear of the CPU queue. Jobs remain in the CPU queue until they complete their service time. Note that jobs may leave the CPU at non-integral times.

Your program must gather statistics, compute, and report the following information:

1. The number of jobs for each job class and CPU requirement.
2. [Bonus] The average time that a job spends in the wait queue.
3. The average elapsed time that a job spends in the CPU queue.
   [Bonus] For each of the job types, broken down by CPU time required and job class.
4. The percentage of the time that the CPU is busy. (The CPU is idle if there are no jobs in the system.)
5. The "through-put" of the system, expressed as the number of jobs handled per hour.

To summarize, in each clock cycle (a simulated second), the following tasks are performed:

1. The clock is incremented.
2. If a job (or jobs) enters the system, determine the job number, job class, and CPU time required. The job class and CPU time required for each job must be determined by a random number generator, as described in the Application section of this chapter. Time stamp the job, and enqueue it in the CPU queue [Bonus: wait queue].
3. [Bonus] If any of the CPU queues is not full, dequeue the front job(s) from the wait queue and enqueue the job(s) in the available CPU queue. Put a job into shortest length CPU queue.
4. If the CPU queue is not empty, service the jobs in the CPU queue, giving each job serviced the correct time-slice. (Do not eliminate a job until its "execution" is complete.) Otherwise, if the CPU queue is empty, increment the CPU idle time.
5. If a job (or jobs) completes its execution, remove it from the CPU queue and calculate the elapsed times for statistics.

**Input**

Your program should prompt the user to input the simulation parameters:

1. The number of jobs to be generated (at most 100).
2. The probability of one job entering the system in each simulated second. (All probabilities will be decimal numbers.)
3. The probability of two jobs entering the system in each simulated second.
4. The distribution of CPU service times for jobs (what percentage of jobs take 10, 20, 30 or 60 seconds). These must add up to 100%.
5. [Bonus] The job class distribution (what percentage of jobs are I/O bound and CPU bound). These must add up to 100%.
6. [Bonus] The number of CPUs (at most 10).

**Output**

All program output must be written to a text file, "CpuSim.out", which is printed and turned in for grading.

1. Echo print the simulation parameters supplied by the user. Be sure to label them clearly.
2. Print the job number, job class, CPU time and time each job entered the system and leaves the system.

3. [Bonus] Print out a system summary every 60 seconds. The summary should contain the time, the number of jobs in each queue, and the job number of the job at the front of the queue.

4. When the simulation is complete (the specified number of jobs have been executed), print a report of the job statistics as requested above. Be sure that your report is logically arranged and neatly labeled and formatted.

**Data Structures**

Each "job" processed by the simulation will have several characteristics:

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job number</td>
<td>Integer</td>
</tr>
<tr>
<td>[Bonus] Job class</td>
<td>I/O bound or CPU bound</td>
</tr>
<tr>
<td>CPU time required</td>
<td>Floating point</td>
</tr>
<tr>
<td>[Bonus] Time enqueued in Wait queue</td>
<td>Time stamp (simulation time)</td>
</tr>
<tr>
<td>Time enqueued in CPU queue</td>
<td>Time stamp (simulation time)</td>
</tr>
</tbody>
</table>

Your program should access the queues only through the interfaces provided by the ADT. If you need additional operations that are not specified in the FIFO Queue ADT in this chapter, they must be specified and implemented using the operations of the Queue ADT.

**Simulation Parameters for Testing**

Use the following input parameters for testing the simulation:

1. The number of jobs to execute is 100.

2. [Bonus] The number of CPUs is 4.

3. The probability of a job or jobs entering the system in any simulated second (one clock tick) is:
   - 40% for 1 job
   - [Bonus] 1% for 2 jobs entering in the same second.

4. The distribution of the CPU service times for jobs is:
   - 33% of the jobs require 10 seconds of CPU time
   - 50% of the jobs require 20 seconds of CPU time
   - 12% of the jobs require 30 seconds of CPU time
   - 5% of the jobs require 60 seconds of CPU time

5. [Bonus] The job class distribution is:
   - 65% of the jobs are I/O bound
   - 35% of the jobs are CPU bound
Discussion

The results for this simulation should be variable, as on the average three jobs enter the system each simulated minute, and each takes an average of 20 seconds of CPU time to execute. A sequence of random numbers generating more jobs than average or longer jobs than average tends to increase the size of the wait queue. There may also be some detectable "dry" times when few jobs enter the system. Try several random number seeds to explore this. You might also try changing the mix of the jobs slightly. Have fun!

Deliverables

♦ You may ignore the requirements marked as [bonus] unless you want to get the extra credit.
♦ Your design including CRC cards for each class “prog4_design”.
♦ A listing of any ADT(s) defined and used “adt4”
♦ A listing of the test driver for the ADT(s) “adt4_test”
♦ A listing of your program “prog4”
♦ A listing of the test driver for the program “test4”
♦ A listing of your test plan as input and the output from test drivers “test4”
♦ Provide all files under a compressed file (such as zip, tar, rar) where the file name is “prog4_LASTNAME” with your last name.
♦ Unless permission is obtained from the instructor, this assignment must be completed by a group of 2.
# PROGRAM SCHEDULE

<table>
<thead>
<tr>
<th>Milestone</th>
<th>Planned</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment received.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requirements understood; detailed specification recorded.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Top level of design complete.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All levels of top-down design complete; data structures determined.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coding complete (clean compile).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test plan complete.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testing complete</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program ready to turn in; all external and internal documentation complete.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assignment turned in.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>