ECE198KL: Introduction to Computer Engineering II **Spring 2013**
Program 8 **due: 10 p.m. on Monday 25 March**

Testing Heap Code for a Discrete Event Simulator

Your task this week is to write a good set of test cases for a heap implementation and to perform and understand some queueing experiments that make use of the heap to implement a discrete event simulation model of a store.

The objective for this week is to give you experience with testing as well as some simple metrics and strategies for testing. You will also be exposed to some interesting mathematical models, which you may want to learn in more detail later.

As always, routine details such as how to obtain the code and how to hand in your program can be found in the specification for Program 1. I did not embed the code in the WALY library, so I'll include some brief instructions on how to build the package later in this document.

The Pieces

Including file headers, comments, and so forth, this week's package is just over 1,000 lines of code. Your primary focus is on the `heap.c` file that implements the heap, but you may want to look through the rest of the package so as to understand how a simple discrete event simulator works.

The files provided to you (in decreasing order of relevance) are as follows:

`prog8.h`	A header file including function declarations for all components.
`heap.c`	A heap used for tracking the motion of people through a store (the discrete event system being simulated).
`testing.c`	The file into which you will place your heap tests. Wrappers for the heap functions are already written for you in this file, so you need merely design an appropriate sequence of calls.
`prog8.c`	The main program used to run an experiment.
`hist.c`	A simple histogram package for tracking and printing probability distributions produced by the simulation experiments.
`store.c`	Implementation of two store models.
`util.c`	A utility function for sampling an exponential distribution.
`Makefile`	Directions for building programs, creating graphs, and so forth.
`steady`	Gnuplot script for creating one comparison graph.
`onequeue`	Gnuplot script for creating a second comparison graph.

In programming studio, we will take a look at the results of the experimental runs, and will talk about how you can set up experiments of your own if you're interested.

The assignment must be completed on your own.

The Models

There are two sets of questions for you to answer in this section. Your answers should be typed into the ANSWERS file and committed to your repository.

The code given to you models a simple store with multiple checkout counters. Customers who have finished shopping and are ready to check out arrive at some specified rate (the *arrival rate*, using an *exponential distribution* for interarrival times—the expected number of customers per time unit is simply the rate). These customers proceed to one of the checkout counters and form a line (a *queue*) to check out. Customers in each line are served at a second rate (the *service rate*). An exponential distribution is used here, too: some customers buy more than others, and take longer to check out.

The code supports two models: distributed and centralized. In the distributed model, each customer looks over the number of other customers in each queue, then picks one of the queues with the shortest line. The customer is then locked in to that queue—they don't move around afterward. In the centralized model, the store has customers form another (centralized) queue, and a customer only moves to a checkout counter when a counter is free. This decision process is equivalent to checking the next counter to become free when the customer shows up and routing the customer to that counter (in reality, the centralized queue is needed because the future knowledge implied by checking the next counter to become free is not possible...simulations are amazing!).

You can build the code, run some experiments, and produce some graphs by simply typing: `make steady.pdf` and `make onequeue.pdf`.

The `onequeue.pdf` graph (make it and look at it) illustrates the difference between the two store models in terms of the delay experienced by customers. In the experiments, we use 8 queues, an arrival rate of 0.9, and a service rate of 1.0. The service rate is distributed among the 8 queues, so the rate in any given queue is only 1/8, and a customer expects to spend an average of 8 time units waiting if they find an empty counter (see the "no queueing delay" line in the graph). The horizontal axis in the graph represents the total delay experienced, and the vertical axis represents the probability of experiencing that delay.

Based on the `onequeue.pdf` graph, answer the following questions. Why do both models show higher average delay than that expected according to the service rate? Comment on and explain the difference between the results for the centralized and distributed models.

When we run an experiment, we call the starting time 0. The store is empty at time 0. As part of each experiment, you specify the time period of measurement, which need not start at time 0 (and, as you'll see, it probably should not!). The delay experienced by any customer arriving between the start and end times of the experiment is recorded in the histogram and printed out at the end of the program. To get enough data, we can also repeat the experiment many times using different sequences of random numbers.

The `steady.pdf` graph shows delay distributions (using the same format) as a function of time. The service rate line is repeated. **Based on the `steady.pdf` graph, write your answers to the following.** In the first 100 time units, the service seems pretty good, but over time it seems to deteriorate. Explain why. Also explain the difference between measuring the first and second 10,000 simulated time units. Do you think the third 10,000 time units look any different than the second 10,000?

Testing the Heap Code

The main part of this assignment involves your writing a set of tests for a heap implementation. Your tests must exercise every statement in each of the four functions provided by the heap. The fraction of statements exercised by a set of tests is known as *statement coverage*, and you should try to achieve 100% statement coverage for the heap code. Declarations for the four functions appear below.

```
int32_t heap_init ();
int32_t heap_insert (double time, int32_t queue_num, int32_t type, double arrival);
int32_t heap_peek_min (double* time_ptr);
int32_t heap_get_min (double* time_ptr, int32_t* queue_num_ptr, int32_t* type_ptr,
                      double* arrival_ptr);
```

You will need to look through the `heap.c` file carefully to understand the heap code. Although the implementation is almost identical to the one that we developed in class (for which notes are on the class web page), the heap in this code is ordered using simulation time (a `double`), and each event includes additional information (stored in separate arrays, to avoid having you need to make use of structures before you have had a chance to digest them a bit more).

The `heap_init` call simply empties the heap, setting the length to 0.

The `heap_insert` call inserts a new event at the end of the arrays and percolates it upward in the tree to fix the heap property.

The `heap_peek_min` gives the time of the next event, if any.

The `heap_get_min` extracts the next event, recording the event's properties into the addresses passed, and ensures that the heap property is restored after the event is removed.

A set of wrapper functions and a couple of simple examples have been provided to you in the `testing.c` file to get you started. The `testing` program built by default when you type `make` will execute your tests. You must create a set of tests that exercise every statement in every function and add those tests to the `testing.c` file. You may find it helpful to use a printout to mark off sections of code as you cover them, and to use `gdb` to ensure that the program is indeed behaving as you expect it to behave.

Challenges for Program 8

Here are some challenges for this week.

(2 points) Run the onequeue experiment at an arrival rate of 0.3 instead of 0.9. Turn in the new graph (you will need to add it to your repository—choose a new name). Give us the graph PDF file name and explain the results in the ANSWERS file.

(2 points) Add a line for the third 10,000 time units to the `steady.pdf` graph. Turn in the new graph (you will need to add it to your repository—choose a new name). Give us the graph PDF file name and explain the results in the ANSWERS file.

(6 points) Statement coverage is necessary for a good set of tests, but having statement coverage is rarely sufficient to guarantee a good test set. To strengthen your tests, add sanity checks on the other fields of the events (queue number, type, and arrival). For example, you can use the random number generator (seeded with bits from the event time) to produce a reproducible but hard-to-predict set of bits for the each event, then check that these bits are moved around correctly by the heap code. Similarly, you can insert a specific set of types and/or queue numbers with random event times, then check that the same set comes out in some order.

(10 points) Add a new queue model to explore the benefit of miaZa's ordering strategy. For those who haven't been there, here's an explanation. Four types of food are available (sandwiches, pizza, pasta, salad). People can customize their order by selecting toppings, dressings, sauces, and so forth. Some people are fast, while others take a long time. After their order form is ready, they pay, then wait for their food to appear. The restaurant separates out those people who are filling out their order forms so as to avoid having a slow customer block the paying line. In your model, use an exponential distribution to model the time needed to order. Use a deterministic time (fixed amount for each person in the queue) to pay. And use four queues with deterministic service time for the four types of food. Note that the customers can order in parallel in both cases, but a customer cannot start paying until they have finished their order. In the miaZa's model, a slow customer does not block the payment queue. In a traditional store model, they do. Build both options to allow comparison of the results. You'll need to allow all parameter values to be specified on the command line, and to produce and explain some interesting graphs if you want full credit for this part. Note that the larger number of parameters can be detected in the code, so don't break the operation of the original program and models. You can also, of course, use the type argument to differentiate them.

Specifics

- Write your answers to the questions about the experiments into the `ANSWERS` file. No other file will be graded for this portion of the assignment.
- Your testing code must be written in C and and must be contained in the `testing.c` file provided to you — we will NOT grade files with any other name. (If you take the challenge of creating a new model, you may create a `miazas.c` file to implement that model, and may change `prog8.c` and the `Makefile` to make use of it.)
- Your tests must execute every C statement in each of the four heap functions for full credit.
- The heap code has no known bugs, so your tests should pass.
- Your tests must be well-commented. Explain what you are doing with each portion of the sequence. You may use either C-style (`/* can span multiple lines */`) or C++-style (`// comment to end of line`) comments, as you prefer. Follow the commenting style of the code examples provided in class and in the textbook.

Grading Rubric

Questions on Experiments (20%)
- 10% - good explanations for `onequeue.pdf` graph
- 10% - good explanations for `steady.pdf` graph

Testing (45%)
- 5% - `heap_init` statements covered by tests
- 15% - `heap_insert` statements covered by tests
- 10% - `heap_peek_min` statements covered by tests
- 15% - `heap_get_min` statements covered by tests

Style (15%)
- 5% - compilation generates no warnings (note: any warning means 0 points here)
- 5% - does not use global variables
- 5% - indentation and variable names are appropriate and reasonably meaningful

Comments, clarity, and write-up (20%)
- 20% - testing strategy described clearly for each of the heap functions

Note that some point categories in the rubric may depend on other categories. If your code does not compile, you may receive a score close to 0 points.