ECE 223 Programming Assignment 2
Due Date in Canvas

Grocery Store Simulation Program

You will create two new ADTs for this program: a FIFO queue ADT named queue.c based on your existing linked list ADT (which must be brought up to spec if it is not already) and a priority queue ADT named priority.c. You will provide several different queues. The first is a priority queue we call the *event queue* which is used to keep up with program events. The rest of the queues will be regular FIFO queues that will represent people in line for a cashier.

The priority queue has the following interface:

```
typedef struct priority_s priority_t;

/* create and initialize a new priority queue
   must be able to hold at least size items
   return pointer to the new priority queue, NULL if error */
priority_t *priority_init(int size);

/* insert an item into the priority queue
   return 0 if successful, -1 otherwise */
int priority_insert(priority_t *q, event_t *ev);

/* remove the highest priority item from the queue
   and return it, return NULL if there is an error */
event_t *priority_remove(priority_t *q);

/* return non-zero if the priority queue us empty */
int priority_empty(priority_t *q);

/* return nono-zero if the priority queue is full
   This may be trivial using a linked implementation */
int priority_full(priority_t *q);

/* free all resourced used by the priority queue then free
   the queue itself */
void priority_finalize(priority_t *q);
```

You will implement this priority queue as a binary tree stored in an array of fixed size as discussed in class.

The second queue is a FIFO queue built in a module called queue.c with the following interface:

```
typedef struct queue_s queue_t;

/* create and initialize a new queue
   must be able to hold at least size items
   return pointer to the new queue, NULL if error */
queue_t *queue_init(int size);

/* insert an item into the queue
   return 0 if successful, -1 otherwise */
int queue_insert(queue_t *q, customer_t *c);

/* return the next item in the queue but do not remove it
   return NULL if the queue is empty or on error */
customer_t *queue_peek(queue_t *q);

/* remove the next item from the queue
   and return it, return NULL if there is an error */
customer_t *queue_remove(queue_t *q);

/* return the number of items in the queue
   You can see if queue is empty with this */
int queue_size(queue_t *q);

/* return nono-zero if the queue is full
   This may be trivial using a linked implementation */
int queue_full(queue_t *q);

/* free all resourced used by the queue then free
   the queue itself */
void queue_finalize(queue_t *q);
```

This will be implemented with your linked list module from Assignment 2. You will need an array of queues – the number can be set with a runtime command line flag -q

Simulation

You will then write a program that simulates people shopping using an event driven simulation. You will create events represented by a struct that includes an event type (an int), an event time (a float), and event data (a customer_t * and/or a queue number). These events will be placed on the event queue. The event queue will sort the events by time so that the event with the smallest time returns first. A while loop will remove events from the event queue as long as there are events in the queue, and then use a switch statement to decide what to do based on the event type. Some events will cause new events to be created and added to the event queue. Some will cause old events to be re-added to the queue, and still others will free old events. If the event queue becomes full, it is an error and the simulation must be re-run with a larger event queue. A command line argument –e should allow the size of the event queue to be set or override a default. When there are no more events in the event queue, the simulation is over.

Events and Customers

```
typedef struct customer_s
{
     double arrival_time; /* gets to the store */
     double enqueue_time; /* gets in line for cashier */
     double checkout_time;/* gets to cashier */
     double leave_time;   /* leaves the store */
} customer_t;

typedef struct event_s
{
     int event_type;      /* type of event – see below */
     double event_time;   /* sim time when event occurs */
     customer_t *customer;/* customer related to this event */
     int queue_number;    /* queue related to this event */
} event_t;
```

We will use a random variable to control when each customer arrives at the store. We will give you a function for this purpose. When a customer arrives, we first mark down the time, then the customer goes and shops for some length of time. We will use another random variable for that time. When the customer is done shopping, we will mark down that time, and then select a cashier. There can be from 1 to N cashiers. To start with the customer always picks the shortest line. The customer gets in line. When the customer gets to the cashier, we mark down the time, and then use a third random variable to determine the size of his cart – and how long it takes the cashier to check him out. Finally, the customer leaves the store and we mark down that time as well.

We will keep statistics as we go and at the end of the simulation we will produce a report showing average shopping time, average wait time, average overall time, etc.

We will simulate the shoppers using events. There will be one customer struct for each customer that comes shopping, and each customer will go through the same set of events that correspond to the points where she changes from one activity to another. The *event types* are as follows:

```
EV_ARRIVE     /* this is when shopper arrives and starts shopping */
EV_ENQUEUE    /* this is when she stops shopping and gets in line */
EV_CHECKOUT   /* this is when she gets to the cashier */
EV_LEAVE      /* this is when she leaves the store */
```

For each of these event types you will write a bit of code that carries out the needed effect – marking time, finding a queue, using a random variable, etc. These bits of code will be in a switch statement so that each time you take an event out of the event queue, you will use that event's event_type in the switch to select the right bit of code. The event queue will make sure that everything will happen in the correct order.

The main routine should be structured something like this:

```
/* initialize queues */
/* malloc new event and customer */
/* now schedule arrive event at t=0 and put in event queue */
while(!event_empty(eq))
{
    new_ev = event_remove(eq);
    time_set(new_ev.time);
    switch (new_ev.event_type)
    {
    case (EV_ARRIVE) :
        new_ev.customer.arrive_time = time_get();
        shoptime = TIME_SHOP();
        schedule enqueue for this customer
        if (MAX_CUST > num_customers++)
        {
            /* malloc new event and customer */
            next_customer = TIME_ARRIVE();
            schedule an arrive event
        }
        break;
    case (EV_ENQUEUE) :
        customer.enqueue_time = time_get();
        find the queue with the smallest line (for loop)
        if the queue is empty
        {
            schedule a checkout event for this queue t+0
        }
        insert customer onto queue
        break;
    case (EV_CHECKOUT) :
        customer.checkout_time = time_get();
        use queue_number to select a queue and remove a customer
        use TIME_CHECKOUT to find checkout time
        schedule a leave event for customer, queue_number
        break;
    case (EV_LEAVE) :
        customer.leave_time = time_get();
        if (queue is not empty)
        {
            schedule checkout event for this queue t+0
        }
        compute stats for customer
        free customer record
        break;
    default :
        break;
    }
    free event if needed
}
Print overall stats
```

NOTE: You must be very careful with events – you CAN reuse them by altering their values and reinserting them into the event queue on a schedule, BUT it is very easy to mess this up getting your pointers confused. A less error prone approach would be to always malloc a new event each time you schedule, and free every event after you take it out of the priority queue.

Your program should consist of:

```
list.c     /* from assignment 2 */
list.h
queue.c    /* implement FIFO queues on top of list.c */
queue.h
priority.c /* PQ implemented as a heap */
priority.h /* using sequential (array) storage */
sim.h      /* defs for events, customers, etc. */
main.c     /* the main simulation code */
randsim.c  /* provided to you – calls to get */
randsim.h  /* times using random variables */
```