Lab#11: Discrete Event Simulation using Queues – Part1

Today we will do a discrete event simulation of an airline check-in hall. An airline has decided to use automated check-in kiosks to check in passengers. Here are some specific details about the situation:

- It takes 5 to 8 minutes to service a customer using an automated kiosk.
- Up to 3 customers can arrive in any given minute.
- Customers form a single queue from where they are directed to available kiosks. That is, it is a single queue, multiple server situation.
- The airline wants to know, what would be the optimum number of kiosks that will need to be installed so that, on average, there are no more than three customers waiting in the queue.

We will create a simulation of the above that will allow us to explore the question. We will be deliberate about developing the simulation, focusing on design first, and then building the program. The program will be built in stages- in modules/units, that are tested individually, and will incrementally result in a final product. Please, pay close attention to the design and how the implementation proceeds. This exercise requires us to model two entities: a *customer*, and a *kiosk*. Further, arrival, and waiting of customers will require a *queue*.

Customer: First lets us model the customer.

A customer will have the following attributes:

- A unique customer id. It will be a number of the form C# (e.g., C1, C98, etc. with <= 3 digits in the ID)
- We will need to record the arrival time (in simulation minutes)
- Service time of the customer will vary between 5 and 8 minutes

The code shown below implements the above requirements for modeling customers:

```
typedef struct {
   char *id;      // id is for form 'C'+# (e.g, C23, C9, etc.)
   int arrivalTime;
   int serviceTime;
} Customer;
```

For the `Customer` type, we will define the following functions:

```
// Given arrival time, create a new customer
Customer *newCustomer(int arrTime);
   // Creates a new customer, with an ID (up to 4 chars), whose
   // arrival time is arrTime, and service time is between 5-8 minutes (random)

// Accessors
char *getCustId(const Customer *c);
int getArrivalTime(const Customer *c);
int getServiceTime(const Customer *c);

// Print Representation
char *custToString(const Customer *c);
   // Print form: <id: arrivalTime, serviceTime>
   // e.g. <C23: 12, 7>, <C19, 42, 8>, etc.
```

Task#1: Implement & Test Customer Type

Create a file **Customer.h** and place the definition of the `Customer` type in it. Also add the prototypes of all the functions in it. Next, create a file **Customer.c** and implement all the functions defined in **Customer.h**

In **Customer.h** you will need to additionally define a static variable (at the top of the file):

```
static int count = 0;
```

This variable will be used to generate a unique customer id using the function shown below:

```
char *genCustId() {     // creates a unique id as 'C'+count (e.g. C23, C9, etc.)
  count++;
  char *id = malloc(5*sizeof(char)); // Max 4-letters for ID +1 for '\0'
  sprintf(id, "C%d", count);
  return id;
} // genCustId()
```

Note that this function is a helper function (will be used by `newCustomer()`) and therefore there is no need for it to be listed in the **Customer.h** file. It is a "private" function for the `Customer` type.

Write a test program (shown below), **testCustomer.c** that creates an array of 5 customers and prints it out:

```
#include <stdio.h>
#include "Customer.h"

int main(void) {

   Customer *cs[5];     // We will create 5 customers
   int t = 23;
   for (int i=0; i < 5; i++) {
      cs[i] = newCustomer(t);
      t++;
   }

   for (int i=0; i < 5; i++) {
      printf("<%s: %d, %d>\n", cs[i]->id, cs[i]->arrivalTime, cs[i]->serviceTime);
      printf("Customer id: %s, Arr: %d, Ser: %d\n", getCustId(cs[i]),
                     getArrivalTime(cs[i]), getServiceTime(cs[i]));
      printf("%s\n", custToString(cs[i]));
      printf("--------------------\n");
   }

   return 0;
} // main()
```

Closely study the three print statements in the program above. Try to see how the functionality of the `Customer` type is being tested. Why?

Test the program for 10 customers and observe that you are getting correct customer IDs, that their arrival time is being recorded (starting from 23) and their service time ranges between 5 and 8 minutes.

Kiosk: Next, let us identify the salient features for modeling a check-in kiosk:

- We will give each kiosk a unique id (for example, K1, K42, etc. with <= 3 digits)
- Each kiosk keeps track of the customer being served by it, unless it is idle.
- If serving a customer, it keeps track of service time remaining.
- We will also will keep track of the total number of customers served.

This is implemented in the type, **Kiosk** below.

```
typedef struct {
   char *id;
   Customer *nowServing;  // Customer this kiosk is serving (or NULL)
   int timeRemaining;     // How much time still remains to serve the customer being served
} Kiosk;
```

For the `Kiosk` Type, we will define the following functions:

```
// Create a new Kiosk: K#
Kiosk *newKiosk();

// Accessors
char *getKioskId(const Kiosk *k);
Customer *getCustomer(const Kiosk *k);
   // Returns the customer being served (or NULL)
void assignCustomer(Kiosk *k, Customer *c);
   // Assigns customer, c to kiosk, k
void tick(Kiosk *k);
   // Performs 1 minute of service to customer (if serving one)
   // If customer is completely served, it resets this kiosk (to no customer)
int busy(const Kiosk *k);
   // Is Kiosk, k serving a customer?
int getNServed();   // returns total# Customers served so far in simulation
   // How many total customers have been served so far

// Print representation of a Kiosk is K12: <C19: 42, 8> [3]
char *kioskToString(const Kiosk *k);
```

Task#2: Implement & Test the Kiosk Type

Create a file, **Kiosk.h** and place the definition of the `Kiosk` type and its associated function prototypes in it (as above). Next, create a file **Kiosk.c** that implements the functions defined in **Kiosk.h**.

As you did for the `Customer` type, you will need to define a static variable (call it **`count`**) and define a function `genKioskId()` to generate `Kiosk` id numbers. Additionally, along with `count`, you should define another static variable, **`nServed`**, that keeps track of the number of customers completely served so far. This number should be incremented in the `tick()` function whenever a customer has been completely served (i.e. when it sets `timeRemaining = 0`). Also, note in the functions above, we are defining an accessor to get the value of this variable (using the function, `getNServed()`).

Finally, you should write a program, **testKiosk.c** as below:

```
#include <stdio.h>
#include "Kiosk.h"
#include "Customer.h"

int main(void) {

   Kiosk *ks[5];
   for (int i=0; i < 5; i++) {
      ks[i] = newKiosk();
   }

   for (int i=0; i < 5; i++) {
      printf("<%s: %d>\n", ks[i]->id, ks[i]->timeRemaining);
      printf("%s\n", kioskToString(ks[i]));

   }

   return 0;
} // main()
```

The program above is testing the basic creation and printing of kiosks. It is not testing the functionality of all the functions in the Kiosk type. Experiment with statements in the program above to test some of the functionality you have implemented.

The Queue of Customers

The simulation we are building requires a Queue. We can implement the Queue using the linked list data type (`List`) we created earlier. Here is a definition of `Queue`:

```
typedef struct {
   List *L;
} Queue;    // A Queue is just a linked list
```

That is, a Queue is essentially a List (linked List). Here are the functions we will define on a Queue:

```
// Creates a new Queue
Queue *newQueue();

// Basic necesseties...
int sizeQ(const Queue *q);
int emptyQ(const Queue *q);

// Core functions for a Queue
void insertQ(Queue *q, void *item);
void *removeQ(Queue *q);
```

Task#3: Implement and Test Queue Type

Create a file, **Queue.h** and place the definition of the `Queue` type and its associated function prototypes in it (as above). Next, create a file **Queue.c** that implements the functions defined in **Queue.h**. You will need to decide which end of the `List` serves as the rear of the queue and the other end will be the front of the `Queue`. Recall, a queue is a first-in-first-out (FIFO) data structure.

As you did for the two earlier types (Customer and Kiosk), write a test program to test the Queue type:

```
int main(void) {
   Queue *Q;
   Q = newQueue();

   printf("Q Size = %d\n", sizeQ(Q));
   if (emptyQ(Q))
      printf("The queue is empty.\n");
   else
      printf("The queue is not empty.\n");

   int *n;
   printf("Adding: ");
   for (int i=0; i < 10; i++) {     // Isert 10 #'s to Q
      n = malloc(sizeof(int));
      *n = rand()%100;
      printf("%d ", *n);
      insertQ(Q, n);
   }
   printf("\n");

   printf("Q Size = %d\n", sizeQ(Q));
   if (emptyQ(Q))
      printf("The queue is empty.\n");
   else
      printf("The queue is not empty.\n");

   for (int i = 0; i < 8; i++) {    // Remove 8 #'s from Q
      n = removeQ(Q);
      printf("Removed: %d\n", *n);
   }

   printf("Q Size = %d\n", sizeQ(Q));
   if (emptyQ(Q))
      printf("The queue is empty.\n");
   else
      printf("the queue is not empty.\n");

   return 0;
} // main()
```

As before, study the program above, and its output. Make sure that Queue is behaving as it is supposed to.

Once completed, let your professor know by e-mail.