Lab#12/Assignment 5: Discrete Event Simulation Using Queues – Part 2

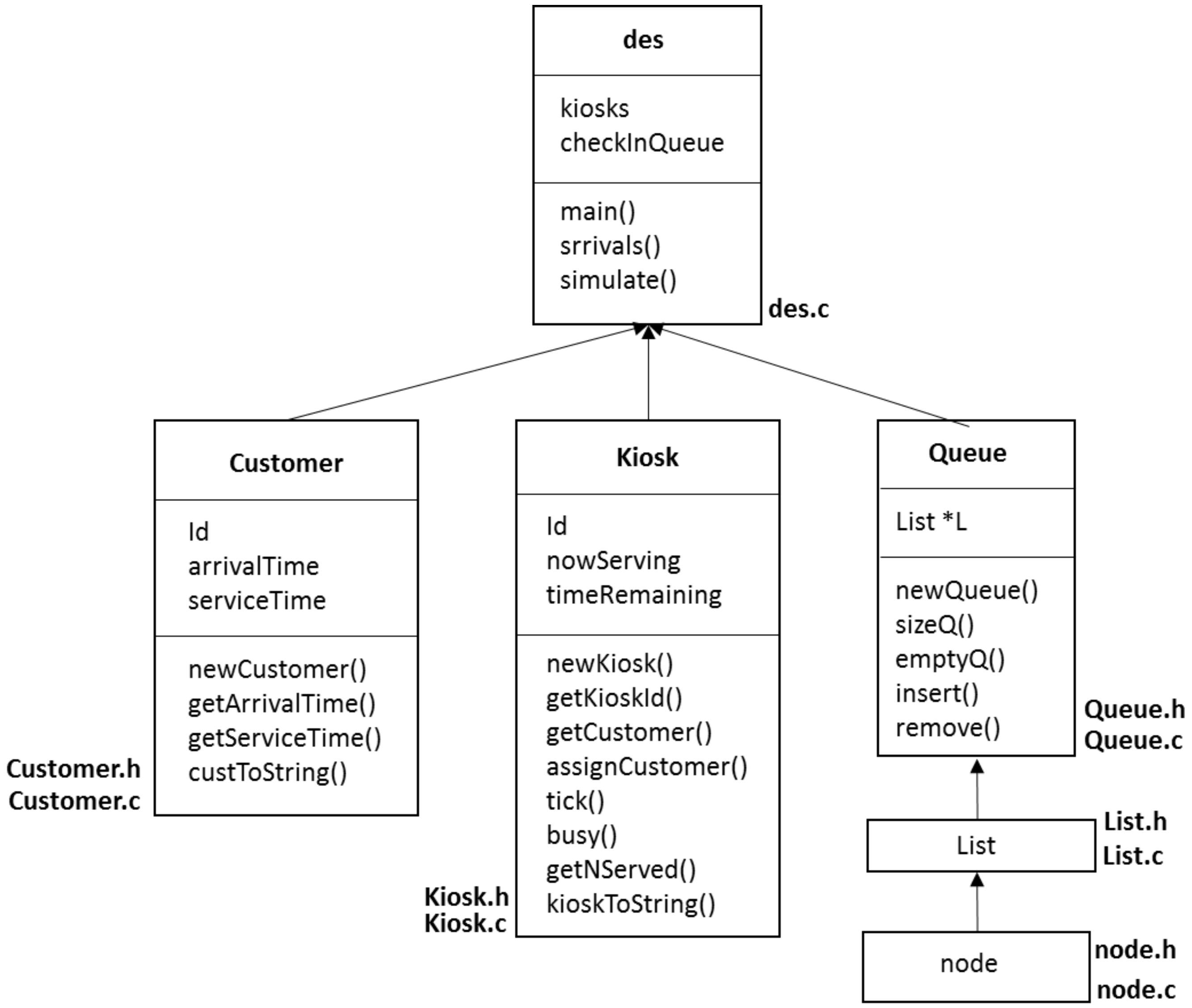
Putting it all together!

[Note: This Lab/Assignment requires you to have completed Labs 10 and 11.]

So far in Lab 11, we have designed, implemented, and tested three modules/units: `Customer`, `Kiosk`, and `Queue`. Now, it is time to put all the pieces together into a simulation program. Let's call it **des.c** (for discrete event simulation). This program will simulate what happens in an airport check-in hall: that is, in each minute up to 3 customers can arrive and queue up. We will be able to vary the number of kiosks to explore what may be the optimum number so that, say in an hour, the average queue length is no more than three customers.

The `des.c` program, again we will be implementing it incrementally, uses a queue of customers and assigns them to available kiosks as and when they are available. By simulating what happens in each minute, our program will be able to control all aspects of the simulation.

The diagram below shows the design of our program, and its associated files:



It also summarizes, for each data type/module, what data each module holds, and summarizes all the functions we can use. We will write our simulation so that we can run it repeatedly with two varying parameters: how long (in minutes) should the simulation run, and also how many kiosks to make available for that duration. For example, here is a command to run the simulation for 10 minutes, using 4 kiosks:

```
[Xena@CodeWarrior] $ ./des 10 4
```

As you can see in the figure above, the `des.c` program uses a number of kiosks and a `checkInQueue`. Further, it defines three functions that are explained below:

1. **arrival()** – returns the number of customers that arrived in any given minute.
2. **simulate(m, k)** – runs the simulation for `m` minutes on k kiosks
3. **main()** - the main program which essentially sets up the simulation parameters.

Of the above, `simulate()` is the key function. It actually carries out the simulation. Here is an algorithm for it:

```
simulate(m, k) - runs the simulation for m minutes on k kiosks

Create k kiosks and store them in a variable named, kiosks
Create checkInqueue, a queue of customers

for each minute in the simulation
   let n = # of customers that arrived in this minute
   create n customers and add them to the queue
   for each kiosk that is not busy
      assign a customer from the queue to a kiosk
      each kiosk serves a customer assigned to them for 1 minute
```

We will begin by implementing pieces of the above algorithm. The description above is missing some aspects: keeping track of how many customers were served, or the average queue length, etc. We will add this functionality later after we have completed the implementation of the algorithm above. With that in mind, here is an initial design of the simulate() function:

```
void simulate(int m, int k) { // simulate for m minutes on k kiosks
   // create nk kiosks
   Kiosk *kiosks[k];
   for (int i=0; i < nk; i++)
      kiosks[i] = newKiosk();
   printf("There are %d kiosks.", nk);

   printf("[ ");  // Let us print out each kiosk…
   for (int i=0; i < nk; i++)
      printf("%s ", kioskToString(kiosks[i]));
   printf("]\n");

   // Create a queue of customers
   Queue *checkInQueue;
   checkInQueue = newQueue();
   int nc = 0;    // Total customers arrived
   // for each minute…
   for (int t=1; t <= minutes; t++) {
      // n Customers arrive
      int n = arrivals();
      // create and add each customer to the queue
      for (int i = 1; i <= n; i++) {
         Customer *c = newCustomer(t);
         insertQ(checkInQueue, c);
         nc++;
      }

      printf("At time %d there are %d customers on Q.\n", t, sizeQ(checkInQueue));
   }
} // simulate()
```

Compare the above program to the algorithm for `simulate()` presented above. We have only addressed the first few steps. This is a good way to start, put the program together, and start to employ the units/modules we developed earlier.

Notice that we are using the `arrival()` function to retrieve the number of customers that arrived. This function's prototype is:

```
int arrival(); // Returns the number of customers that arrived (any value [0..3])
```

Use the random number generator (remember to seed it in the `main()` function) to complete this function. And finally, the starter version of the `main()` function:

```
int main(void) {
   srand(time(NULL));

   simulate(10, 4);

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

Task#1: Implement and test the program. Make sure, if you haven't already done so, to create a **Makefile** for use in recompiling and building the project.

Task#2: Once you have completed and tested the program in Task#1, modify the `main()` function so that it can accept command line parameters and is able to pass them to the `simulate()` function.

Assigning customers to kiosks: The next step, dispatching customers to kiosks, requires checking for idle kiosks, and assigning customers to them (if there are any in the queue). You will need to add just a few lines of code to accomplish this. First, think about what you need to do here and see if you can accomplish this on your own. You should be able to. Once done, compare your answer to the one provided below:

```
      // Assign customer(s) to kiosk(s)
      for (int k = 0; k < nk; k++) {
         if (!busy(kiosks[k]) && !emptyQ(checkInQueue)) {
            Customer *c = removeQ(checkInQueue);
            assignCustomer(kiosks[k], c);
         }
      }
      printf("[ ");     // print out status of all kiosks
      for (int i=0; i < nk; i++)
         printf("%s ", kioskToString(kiosks[i]));
      printf("]\n");
```

Again, at the end, we print out the kiosks just to be able to peek in on customer assignments. At this stage, here is an example output from the des.c program we have so far:

```
[Xena@CodeWarrior] $ ./des 10 4
There are 4 kiosks.[ K1: [] K2: [] K3: [] K4: [] ]

At time 1 there are 1 customers on Q [1].
[ K1: <C1: 1, 7> K2: [] K3: [] K4: [] ]

At time 2 there are 0 customers on Q [1].
[ K1: <C1: 1, 7> K2: [] K3: [] K4: [] ]

At time 3 there are 3 customers on Q [4].
[ K1: <C1: 1, 7> K2: <C2: 3, 7> K3: <C3: 3, 5> K4: <C4: 3, 8> ]

At time 4 there are 0 customers on Q [4].
[ K1: <C1: 1, 7> K2: <C2: 3, 7> K3: <C3: 3, 5> K4: <C4: 3, 8> ]

At time 5 there are 3 customers on Q [7].
[ K1: <C1: 1, 7> K2: <C2: 3, 7> K3: <C3: 3, 5> K4: <C4: 3, 8> ]

At time 6 there are 4 customers on Q [8].
[ K1: <C1: 1, 7> K2: <C2: 3, 7> K3: <C3: 3, 5> K4: <C4: 3, 8> ]

At time 7 there are 5 customers on Q [9].
[ K1: <C1: 1, 7> K2: <C2: 3, 7> K3: <C3: 3, 5> K4: <C4: 3, 8> ]

At time 8 there are 5 customers on Q [9].
[ K1: <C1: 1, 7> K2: <C2: 3, 7> K3: <C3: 3, 5> K4: <C4: 3, 8> ]

At time 9 there are 6 customers on Q [10].
[ K1: <C1: 1, 7> K2: <C2: 3, 7> K3: <C3: 3, 5> K4: <C4: 3, 8> ]

At time 10 there are 6 customers on Q [10].
[ K1: <C1: 1, 7> K2: <C2: 3, 7> K3: <C3: 3, 5> K4: <C4: 3, 8> ]
```

Task#3: Implement the above commands in your program (or use your own) and make sure you are able to get a similar output from your program as shown above.

Study the output above (and that of your program) carefully as it is showing what happened in every minute of the simulation we have implemented so far. You will notice that once four customers were assigned a kiosk, the remainder of them just have to wait on the queue. This is because we haven't yet added the commands to service each customer at a kiosk. That is our next task.

Kiosks run for 1 minute: Next, we add the following code to run each kiosk for 1 minute:

```
// Kiosks run for 1 minute
for (int k=0; k < nk; k++)
   tick(kiosks[k]);

printf("[ ");// print out status of all kiosks
for (int i=0; i < nk; i++)
   printf("%s ", kioskToString(kiosks[i]));
printf("]\n");
```

Now you will be able to see the customers that have been served by any kiosks. Study the **tick()** method in the **Kiosk** module (in Part 1 from last week) to understand what needs to be done for 1 minute of servicing. Review it and make sure that customers are being served and removed from the queue based on their service time.

Again, run the program several times. Vary the number of minutes, and kiosks to see how the queue grows/shrinks.

Recording data: Finally, we can add some code to measure the pertinent information for this simulation:

- How many total customers arrived during the simulation?
- How many were served during the simulation?
- What was the average queue length during the simulation?

Task#4: In the **des.c** file, add the variable:

```
int queueLength = 0;    // total of all queue lengths during simulation
```

Add the commands:

```
queueLength += sizeQ(checkInQueue);
```

at appropriate places in the `simulate()` function. And, at the end, add commands to printout the following metrics:

```
Simulation: 10 minutes, with 4 kiosks
24 customers arrived.
17 customers served.
Average queue length: 3.2
```

Complete the program and **run the simulation for the duration of 60 minutes**. Use different number of kiosks. And see if you can determine the optimum number of kiosks needed to keep no more than three customers waiting in queue, on average.

Fill the table below. In each box write down n1/n2/x where n1 is the number of customers arrived, n2 is the number of customers served, and x is the average queue length.

#Kiosks->	**1**	**2**	**4**	**8**	**9**	**10**
Run#1						
2						
3						
4						
5						
6						
7						
8						
9						
10						

#Kiosks->	**11**	**12**	**13**	**14**	**15**	**16**
Run#1						
2						
3						
4						
5						
6						
7						
8						
9						
10						

#Kiosks->	**17**	**18**	**19**	**20**	**21**	**22**
Run#1						
2						
3						
4						
5						
6						
7						
8						
9						
10						

With a little more effort, the program can be easily extended to generate the tables above. The data can then be plotted so that one can see the performance of the system with different number of kiosks, the number of customers arrived, and served.

Using the DEBUG flag: You will notice that when you run the simulation for longer times (e.g., 60 minutes) the program produces too much output. Once you have ensured, during testing, that the program runs correctly. Make use of the DEBUG macro to suppress the printing when it is not in DEBUG mode. This way, the print statements remain in the program for later use, in case you need to modify, enhance, or debug the program.

Some other things to think about:
What will happen if the customer arrivals double (that is up to 6 customers arrive in every minute)?
What about if the time taken at a kiosk could be reduced by half (say between 3 to 4 min)?
What if, at any given time, 15% of the kiosks are out of service?
The simulation can easily be extended to answer such questions.