CSC190: Computer Algorithms and Data Structures
Assignment 2
Assigned: Feb 16, 2015; Due: Mar 15, 2015 @ 10:00 a.m.

1 Objectives

The queue data structure can be used for modelling and simulating many interesting systems. In many practical systems, some kind of buffers/queues are used to temporarily place tasks or items prior to processing these. Entities arriving first at the buffer are processed first. Since the FIFO service paradigm is quite natural in a broad range of practical systems, queues can be naturally used to model the behaviour of these. In this assignment, you will be modelling a simple data network.

2 Introduction to Data Networks

Transmitting data through the internet (emails, video/voice streaming, etc.) is a complex process. The internet is essentially a large network composed of intermediate devices that forward transmitted data originating from a source device to a destination device through links as illustrated in Figure 1. These intermediate devices include routers and switches that forward data from incoming links to outgoing links based on their destination.

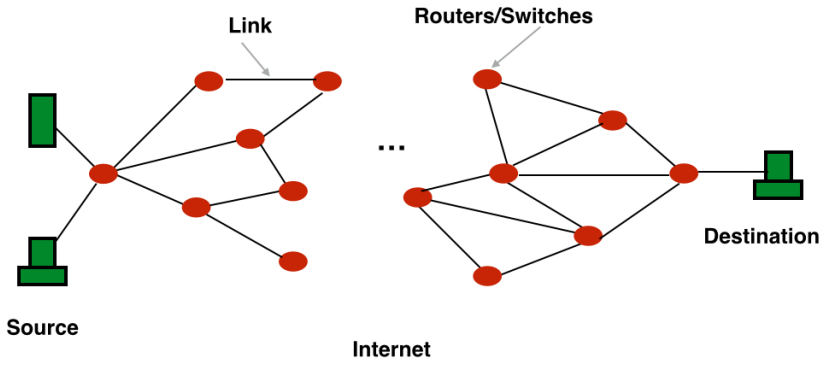

Figure 1: A Simplified Visualization

2.1 A Transportation of People Analogy

In order to understand how data is transmitted across the internet from a source to a destination, consider the following analogy. Suppose that 20 people are to be transported from Point A to Point B via cars only. Everyone is leaving from the same place and intend to arrive at the same place. Cars have fixed space (i.e. can only seat a maximum of 5 people). Assume that all 20 people manage to fit into 4 cars. These cars then will use various roads to travel from Point A to Point B. Since the roads are public, these will be shared by other cars as well. Depending on the time of the day, there will be various degrees of traffic on the roads. Certain roads such as the highways allow cars to travel at a faster speed than others. The cross point of roads will be managed by traffic signals. Based on all these factors, the drivers of the four cars may choose different paths (composed of various roads) to reach the final destination (i.e. Point B). All four cars will arrive at Point B but possibly at different times depending on the conditions of the roads travelled and speed limits.

2.2 Similarities between Data Network and Transportation

This transportation example has surprisingly many similarities with the process incurred during the transmission of data from a source device to a destination device. Suppose that you would like to send an email from your computer (source) to your friend (destination) who resides in another country. In the transmission process, your email is first broken down into smaller entities called packets (analogous to dividing 20 people into 4 cars). These packets then traverse links (analogous to roads) in order to reach the final destination. Different links have different speeds/bandwidths (analogous to speed limits on roads). Since links in the internet are mostly public, your email packets will share the link with other packets from other sources (roads are shared with other people who are driving from other places). Multiple links intersect at a single point (i.e. junction at a road intersection). Intermediate devices such as switches or routers decide on where to forward packets arriving at these junctions. Packets may arrive at different times at a router. When more than one packet arrives at the router, these are stored in a buffer/queue (i.e. cars waiting at a junction). In this assignment, the following assumptions are made: data packets are all of the same size, time taken for a router to forward packets to appropriate links is constant and the queue is infinite in length. Based on the congestion properties and conditions of the network, routers may forward packets to different links (i.e. drivers taking different routes). Even though the packets will arrive at the destination, these might arrive at different times based on the conditions of the paths.

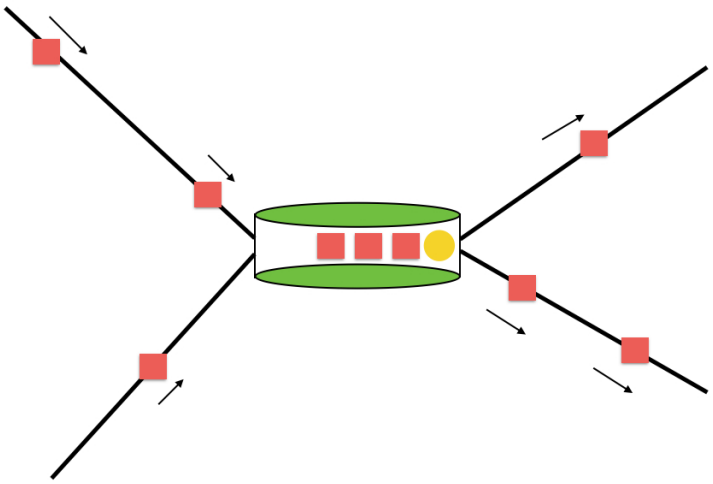

Figure 2: A Simple Representation of a Router Queue

2.3 Your Task for this Assignment

In this assignment, you will model the behaviour of the queue in a single router when packets into arrive into the queue at a particular rate and depart from the queue at a particular rate. As depicted in Figure 2, multiple incoming links can converge at the router. Packets arriving through these links are forwarded to appropriate outgoing links by the router. Service time (i.e. the forwarding of packets to appropriate links) is assumed to be constant. Intuitively, it is pretty reasonable to postulate that if the arrival rate of packets into a queue is less than the service rate, then the queue will be not be congested and the average wait time of a packet in the queue before being processed will not be excessive. On the other hand, when the arrival rate of packets into the queue is greater than the service rate, then the queue will get filled up quickly and each packet in the queue will have to wait for very large periods of time before being served. In this assignment, you will compute the average waiting times of packets departing a router queue by simulating the router queue for various packet arrival rates. You will notice that as the arrival rate gets closer to the service rate, the average waiting times of packets in the queue will increase exponentially and explode. This trend has been modelled via a well known law called the Little's Law. The average waiting time $E(S)$ or sojourn time of a packet in a queue can be related to the arrival rate λ of packets into the queue and

service rate μ of the queue which denotes the departure rate of packets from the queue via this equation: $E(S) = \frac{1}{\lambda} * (\frac{\lambda}{\mu} + \frac{1}{2} * \frac{(\frac{\lambda}{\mu})^2}{1-\frac{\lambda}{\mu}})$. You can use your simulations to verify whether the router queue does obey this derivation or not.

2.4 Simulation of a Router Queue

A router queue can have no packets at a time or some packets waiting to be serviced. Packets arrive at random times. These arrival times depend on the arrival rates of packets. The function `double getRandTime(double arrivalRate)` is provided to you in the `main.c` file. This function returns the duration (seconds) within which the next packet will arrive into the queue for a particular arrival rate λ which is passed as an argument to the function. For instance, suppose $\lambda = 0.1$ *packets/sec*, the function call `getRandTime(0.1)` may result in 1.344 *sec*. This means that the next packet will arrive into the queue within 1.344 *sec*. Hence, packets can be queued into the router according to the times returned by `getRandTime`. Every packet queued into the router will be processed by the router according to the first come first serve policy. Time taken to serve each packet is assumed to be constant. Suppose the service rate is $\mu = 10$ *packets/sec* then since the service time is $\frac{1}{10} = 0.1$ *sec*. This means that the time required for the router to process every packet at the front of the queue is 0.1 *sec*.

In a simulation, certain metrics about the system are measured throughout the simulation period. We are particularly interested in the average waiting time of departing packets for various combinations of λ and μ. The simulation will run for ρ *sec*. Time progression `currTime` in a simulation is based on events occurring in the simulation. Simulation will start at `0` sec. In a router queue, an event will occur if a packet is scheduled to arrive into or depart from the queue. Suppose that a packet is scheduled to arrive into the queue at `timeForNextArrival`. Suppose that the number of packets in the queue is at least one. Then the first packet in the queue is scheduled to depart from the queue at `timeForNextDeparture`. The next event that will occur in the simulation is an `ARRIVAL` if `timeForNextArrival < timeForNextDeparture`. Otherwise, the next event is a packet `DEPARTURE`. If there are no packers in the queue, then the only event that can occur next is an `ARRIVAL`. When an event occurs, the simulator updates the current time variable `currTime` to the time at which the event occurs. The simulator operates a loop conditioned upon the current simulation time. The loop will terminate when `currTime` is greater than or equal to the total simulation time `simTime` allocated to the simulation. At each iteration of the while loop, the simulator will perform operations on two queue data structures. If the current event is an `ARRIVAL` then the simulator will enqueue a node into the `buffer` queue (which mimics the router queue) and compute `timeForNextArrival`. On the other hand, if the current event is a `DEPARTURE`, then the first node in the `buffer` queue is dequeued and this node is enqueued into `eventQueue`. The simulator then computes `timeForNextDeparture`. All nodes in `eventQueue` contain information about the arrival time and departure time of a packet into and from the `buffer` queue. At the end of the simulation, the simulator uses the `calcAverageWaitingTime` function to dequeue all nodes from `eventQueue` and compute the average waiting time or sojourn time of packets that have departed the `buffer` queue. Packets still remaining in the `buffer` queue are not accounted for in this calculation. All dynamically allocated elements are freed and the average packet waiting time computed is returned by the function `runSimulation` which is running the simulation.

3 Materials Provided

You will download contents in the folder `Assignment2` which contains two folders (`code` and `expOutput`) onto your ECF workspace. Folder `code` contains a skeleton of function implementations and declarations. Your task is to expand these functions appropriately in `assignment2.c` and `queue.c` files and include necessary libraries in `assignment2.h` and `queue.h` files as required for implementation. `main.c` evokes all the functions you will have implemented and is similar to the file that will be used to test your implementations. Use `main.c` file to test all your implementations. Folder `expOutput` contains outputs expected for function calls in `main.c`. Note that we will **NOT** use function calls with the same parameters for grading your assignment. Do **NOT** change the name of these files or functions.

4 Grading: Final Mark Composition

It is **IMPORTANT** that you follow all instructions provided in this assignment very closely. Otherwise, you will lose a *significant* amount of marks as this assignment is auto-marked and relies heavily on you accurately following the provided instructions. Following is the mark composition for this assignment (total of 40 points):

- Successful compilation of all program files i.e. the following command results in no errors (3 points):
  `gcc assignment2.c queue.c main.c -o run -lm`
- Successful execution of the following commands: (2 points)
  `valgrind --quiet --leak-check=full --track-origins=yes ./run 1`
  `valgrind --quiet --leak-check=full --track-origins=yes ./run 2`
- Output from Part 1 exactly matches expected output (7 points)
- Output from Part 2 exactly matches expected output (18 points)
- Code content (10 points)

Sample expected outputs are provided in folder `expOutput`. We will test your program with a set of completely different data files. Late submissions will not be accepted. All late submissions will be assigned a grade of 0/40.

5 Implementation of the Simulation Program

You will be using queue data structure that you will have implemented in Lab 5 as a basic building block for this assignment. Your simulation program will make use of the queue data structure to model a queue in a router. This queue is assumed to be infinite in length (i.e. no restriction on the size of the queue). Packets will arrive into the queue at rate λ. Time taken to serve these packets (i.e. forward to the appropriate link) is fixed as dictated by the service rate μ.

Part 1: Defining Interfaces and Structures of a Queue

In this part, function implementations of `enqueue`, `dequeue` and `freeQueue` will be tested. These functions are to be defined in the `queue.c` file. Prototypes of these functions and structures associated with the queue data structure reside in `queue.h`. The underlying implementation of the queue data structure will be based on linked lists. Three structures to be defined first are:

- `Data`
- `Node`
- `Queue`

The `Data` structure will consist of two `double` members: `arrivalTime` and `departureTime`. The arrival and departure times of a packet into and from the queue are recorded in these fields. The structure `Node` has two members: `data` and `next`. `data` is of type `struct Data` and `next` is a pointer of type `struct Node`. The third structure to be defined is `Queue` which has three member variables: `currSize`, `front` and `rear`. `currSize` stores the number of nodes currently residing in the queue. `front` points to the first node in the queue and `rear` points to the last node in the queue. You will implement the following four functions that will operate on the queue data structure:

- `struct Queue initQueue();`
- `void enqueue(struct Queue *qPtr, struct Data d);`
- `struct Data dequeue(struct Queue *qPtr);`
- `void freeQueue(struct Queue *qPtr);`

The function `initQueue` will initialize a queue structure by setting the members `currSize` to 0 and pointer members `front` and `rear` to NULL. The `enqueue` function will insert the node `d` into the back of the queue represented by the pointer `qPtr`. The `dequeue` function will remove a node from the front of the queue represented by the pointer `qPtr` and return the data contained in this node. `freeQueue` will use the `dequeue` function to free all nodes in the queue pointed to by `qPtr`. This part of the assignment will be tested via the following commands:

- `./run 1`
- `valgrind --quiet --leak-check=full --track-origins=yes ./run 1`

Outputs from these tests must match the contents of `Part1.txt` which is the result of the parameters passed from function calls in `main.c`.

Part 2: Implementing the Router Queue Simulator

You will build a queue simulator that computes the average waiting time of data packets departing from a router queue. You will analyze the impact on average waiting times for various packet arrival rates. In this implementation, you will first define a structure called `simulation` and three functions. The `simulation` structure stores all parameters and data structures associated with a simulation. This structure consists of the following members: `double currTime`, `double arrivalRate`, `double serviceTime`, `double timeForNextArrival`, `double timeForNextDeparture`, `double totalSimTime`, `struct Queue buffer, eventQueue;` and `Event e;`. `currTime` keeps tracks of the time progression of a simulation. `arrivalRate` and `serviceTime` store λ and $\frac{1}{\mu}$ respectively. `timeForNextArrival` stores the time at which the next packet will arrive into the `buffer` queue. `timeForNextDeparture` will keep track of the time at which a packet will depart from the `buffer` queue. `totalSimTime` denotes the total time duration for which the simulation is run for. It is assumed that the `buffer` is initially empty and packets start entering the queue only when the simulation begins. `buffer` is a `Queue` data structure that will mimic an actual queue in a router. `eventQueue` is another queue that is used to store information about packets entering and exiting the `buffer` queue. `Event` is an enum with two constants: `ARRIVAL` and `DEPARTURE`. `e` is used to store information about the next event that will occur in the simulation. Three functions which are to be implemented for this part are:

- `struct Simulation initSimulation(double arrivalRate, double serviceTime, double simTime);`
- `double runSimulation(double arrivalRate, double serviceTime, double simTime);`
- `double calcAverageWaitingTime(struct Simulation * S);`

The service rate is denoted by μ is fixed and denotes the rate at which packets depart from the queue. The arrival rate of packets into the queue can vary depending on the congestion in the network. Suppose that the service rate is $\mu = 10$ *packets/sec* (i.e. the service time for a packet is fixed and therefore if $\mu = 10$ *packets/sec* then it takes the router $\frac{1}{10} = 0.1$ *sec* to process a packet (also known as service time)). Packets will arrive into a queue at random times. However, on average, these packets arrive at a rate λ. A simulation is evoked via the `runSimulation` function. Three arguments `arrivalRate`, `serviceTime` and `simTime` are passed into this function. `arrivalRate` is λ, `serviceTime` is $\frac{1}{\mu}$ and `simTime` is ρ. In this function, a `simulation` structure variable is initialized via the `initSimulation` function. The `initSimulation` function returns the initialized `simulation` data structure after assigning to the members of the `simulation` data structure the values passed as arguments to the function. The arrival time of the first packet into the queue is computed via the `getRandTime` function and this value is stored in the `timeForNextArrival` member. The departure time of the first packet from the queue is computed by adding the fixed service time to the previously computed packet arrival time and this value is stored into the `timeForNextDeparture` member. After the `initSimulation` function returns the initialized data structure, the `runSimulation` function will launch a while loop and progress through simulation by performing appropriate enqueueing or dequeueing operations based on the current event.

Packets are enqueued into `buffer` as `struct Node`. The arrival time of the packet into `buffer` is recorded in the `arrivalTime` member of the `Data` variable which is a member of the `Node` struct. When a packet is dequeued from `buffer`, its departure time is recorded in the `departureTime` member of the `Data` variable and this node is then enqueued into the `eventQueue` queue which records all packets departing the `buffer` queue. When the simulation runs for ρ *seconds*, it exits the while loop and evokes `calcAverageWaitingTime` to compute the average waiting time of packets that have already departed from the `buffer` queue. This value is printed to the console. The function then frees all dynamically allocated memory. This part of the assignment will be tested via the following commands:

- `./run 2`
- `valgrind --quiet --leak-check=full --track-origins=yes ./run 2`

Outputs from these tests must match the content in `Part2.txt` which is the result of the parameters passed from function calls in `main.c`. You can test if your implementation is correct by printing out the arrival time and departure time of packets when these are removed from `buffer` queue. Output that should be printed to the console for the specific function call `runSimulation(10,0.1,10)` is available in `test.txt` located in the `expOutput` folder.

6 Code Submission

For this assignment, if your submit via `git`, you will receive **bonus** points. Otherwise you can submit through the `submitcsc190s` command on your ECF machine (no bonus points). Ensure that you submit through only one venue.

6.1 Submission through `Git`

Once you have completed this assignment, you will submit your work by:

- Log onto your ECF account
- Browse into the directory you had cloned in Lab 0 (i.e. `cd ~/UTORID/`)
- Create a folder named `Assignment2` (i.e. `mkdir Assignment2`) in that cloned directory
- Ensure that your code compiles in the ECF environment
- Copy all your completed code (`assignment2.h`, `assignment2.c` into the `Assignment2` folder
- Browse into the `~/UTORID/` directory
- Add all files in the Assignment2 folder (i.e. `git add *`)
- Commit all files that have been modified in the `Assignment2` folder (i.e. `git commit -m "adding assignment files"`)
- Push all changes committed to the `git` server (i.e. `git push origin master`)

ENSURE that your work satisfies the following checklist:

- You submit before the deadline
- All files and functions retain the same original names
- Your code compiles without error in the ECF environment (if it does not compile then your maximum grade will be 3/40)
- Do not resubmit any files in `Assignment2` after the deadline (otherwise we will consider your work to be a late submission)

6.2 Submission through `submitcsc190s`

- Log onto your ECF account
- Ensure that your completed code compiles
- Browse into the directory that contains your completed code (`assignment2.h`, `assignment2.c`)
- Submit by issuing the command:
  `submitcsc190s 7 assignment2.h assignment2.c`

ENSURE that your work satisfies the following checklist:

- You submit before the deadline
- All files and functions retain the same original names
- Your code compiles without error in the ECF environment (if it does not compile then your maximum grade will be 3/40)
- Do not resubmit any files in `Assignment2` after the deadline (otherwise we will consider your work to be a late submission)