Assignment 2 Report

1.0 Overview

My program runs a multi-server single queue simulation based on two structs: ‘Server’ and ‘Customer’. The Customer struct contains attributes detailing their time of arrival, required service duration, and their priority.

The Server struct contains attributes detailing their status (busy or idle), the number of customers they served, the server’s total time spent idle, and the time at which their current service will end.

Based on these two structs, the program has two arrays; one contains all of the servers in the simulation, *Server servers[4]*, and another containing all the customers in the queue, *Customer queue[100]*.

The program uses these two arrays by treating them as heaps .The *servers* array acts as a min-heap, which sorted by their current service-end-at time, and the *queue* array acts as a max-heap sorted by each customer’s priority.

By sorting the servers into a min heap based on their service-end-at time, the server who will next be free is always sorted into the first position in the array. This means when you need to check for any free servers, you only need to check the idle status of *servers[0]*.

Using these data structures, the program then:

1. Reads in a new customer from the data file, storing the values in temporary variables.
2. Assesses whether any servers are free:
    a. If yes
        i. Immediately processes their arrival by either serving them or entering them into the queue.
    b. If no servers are idle
        i. Check whether their current service will end before the arrival of the next customer
            1. If yes
                a. Continue serving and processing customers until the next arrival
                b. Then processes the arrival by either serving them or entering them into the queue.
            2. If their current service time ends after the ‘next arrival’ time read from the data file
                a. This means that we have read the terminating variable in the file, and there are no more new customers left, only the ones in the queue. The program then loops ‘processing service’ until the queue length is empty.
3. Loop steps 1 – 2 until the end of the file and queue, then we heapsort the *servers[0]* and process their final customer.
4. Calculates various output variables based on the collected variables during the simulation

2.0 Pseudo Code

```
Server {
    Idle;
    cusomtersServed;
    endOfCurrentService;
    totalTimeIdle;
}
Customer{
    arrivalTime;
    serviceDuration;
    priority;
}

Customer queue[100];
Server servers[4];

simTime = 0;
nextArrival = 0.0;
nextService = 0.0;
nextPriority = 0;
queueLength = -1;
totalCustomers = 0;

processArrival();
processService();
serverSiftDown();
queueSiftUp();
queueSiftDown();
customerSwap();
serverSwap();


main(){
    open file
    prompt for number of servers in the simulation
    initialise servers

    if(file is open){
    while(!end of file){
        read >> nextArrival >> nextService >> nextPrioirty;

        if(server !idle){
            if(nextArrival!= 0){
                while(!server[0].idle && servers[0].serviceEndAt < nextArrival){
                    processService();
                }
                processArrival();
            }else{
                while(queueLength > 0){
                    processService();
                }
        }else{
            processArrival();
        }

        For(Server s : servers){
            If(!s.idle){
                processService();
            }
            simTime = time of s last service completed
            swap first and last server
            siftDown servers[0] with heapSize -
        }

        print << relevant statistics;

   }
```

```
processArrival(){
 simTime = nextArrival;
 totalCustomers ++;

     if(servers[0] idle){
         //serve immediately
         servers[0].idleTime += (simTime – servers[0].serviceEndAt);
         servers[0].idle = false;
         servers[0].serviceEndAt = simTime + nextService;
         servers[0].customersServed ++;

         //restore min heap property
         serverSiftDown(servers[0]);

         }
         else {
             //enter queue
             queueLength ++;
             queue[queueLength].arrivalTime = nextArrival;
             queue[queueLength].serviceDuration = nextService;
             queue[queueLength].priority = nextPriority;

             //restore heap property of queue
             queueSiftUp();

             //check for max queue length
             If(queueLength > maxQueueLength){
                 maxQueueLength = queueLength;
             }
         }
      }
}



processService(){
 simTime = servers[0].serviceEndAt;

 if(queue empty){
     servers[0].idle = true
 }
 else {
     //serve next person in queue
     //update total wait time of customers
     totalWaitTime += (simTime – queue[0].arrivalTime);

     servers[0].idle = false;
     servers[0].serviceEndAt = simTime + queue[0].serviceDuration;
     servers[0].customersServed ++;

     //restore heap property of servers array
     serverSiftDown();
     //remove customer from queue and restore heap property
     Swap first and last customer in queue
     queueLength --;
     queueSiftDown(queue[0]);
 }
}
```

```
serverSiftDown(current, heapSize){
 int child = 2 * current + 1;

 if (child > heapSize) {    //if at leaf, return
     return;
 }

 if (child + 1 < heapSize && servers[child].serviceEndAt >= servers[child +
 1].serviceEndAt) {
     child++;
 }

 if (servers[current].serviceEndAt >= servers[child].serviceEndAt) {
     //swap
     serverSwap(current, child);
     //siftDown
     serverSiftDown(child, heapSize);
 }

 return;
}

queueSiftUp(current, heapSize){
 int child = current / 2;

 if (child < 0 || child == current) {    //if at root, return
     return;
 }

 if (child + 1 < heapSize && queue[child].priority > queue[child + 1].priority)
{
     child++;
 }

 if (queue[current].priority > queue[child].priority) {
     //swap
     custSwap(current, child);
     //siftDown
     queueSiftUp(child, heapSize);
 }
 return;

}


queueSiftDown(current, heapSize) {
 int child = current * 2 + 1;


 if (child > heapSize) {    //if at root, return
     return;
 }

 if (child + 1 < heapSize && queue[child].priority < queue[child + 1].priority)
 {
     child++;
 }

 if (queue[current].priority <= queue[child].priority) {
     //swap
     custSwap(current, child);
     //siftDown
     queueSiftDown(child, heapSize);
 }

 return;
}
```

3.0 Data Structures & Reasoning

3.1 Structs

For my program, I chose to run the simulation using two structs. I chose to do this as I would need to store multiple variables relevant to one person/server at any given time. By using structs, I could easily access and alter specific variables without any complex sorting or methodologies.

3.2 Heaps

My program uses a min-heap to store its servers, and a max-heap to store it's queue. I chose to do this, as during the simulation, only ever want to know which server is free, and who is next in the queue.

By sorting the servers into a min-heap based on the end time of their current/last service, the server who is free or is next to be free will always be the root of the heap. This means there is no searching or tracking which servers are idle separately, as you only have to check the root of the server heap to determine if any servers are free.

Likewise, by sorting the queue into a max-heap based on priority, the customer who is next to be served will always be the root.

4.0 Complexity Analysis

My program runs on a basic loop, which contains:

1. Processing Arrivals
    a. Where a customer will be immediately served
        i. Updating all 4 server attributes and the simulation time
        ii. Server is sifted down
    b. Or put into the queue
        i. Writing into queue
            1. Updating 3 customer attributes
        ii. Sifted up into queue
2. Processing Services
    a. Simulation time is updated
    b. Server is made idle
    c. Next customer is served
        i. Updating all 4 server attributes
        ii. Server is sifted down
        iii. First and last customer swapped
        iv. Customer sifted down

In a worst case scenario, every customer (except the first) will have to be put into a queue (requiring, siftUp()), and then served (requiring serverSiftDown(), customer swap and custSiftDown()).

With siftUp() normal complexity of $log_2(n)$, when a customer is placed into the queue, and additional 3 variables must be written to fill out the struct attributes, so, if every customer but the first enters the queue, a complexity of:

$\sum_{k=1}^{n} \log(k)\, x\, 3$ operations must occur, where *n* is the number of customers

In addition, at least one customer will be immediately served (the first to arrive), meaning all server's attributes (4) must be updated, and then they must be siftedDown() inside the server array to restore the min-heap property.

This means the complexity of the first customer's arrival and service will be:

$log_2(m)$ x 4 operations, where *m* is the number of servers.

In addition, in worst case scenario if every customer but the first entered the queue and is then served, each of these services will require the server to be sifted down, as well as the first and last customer to be swapped, and then the first customer to be sifted down.

The servers will be sifted down *(n-1)* times:

$(n-1) * (log_2(m) * 4)$

Then each customer served will be swapped:

O(4)

Then sifted down:

***

$\sum_{k=1}^{n} \log(k)$ *x* 3

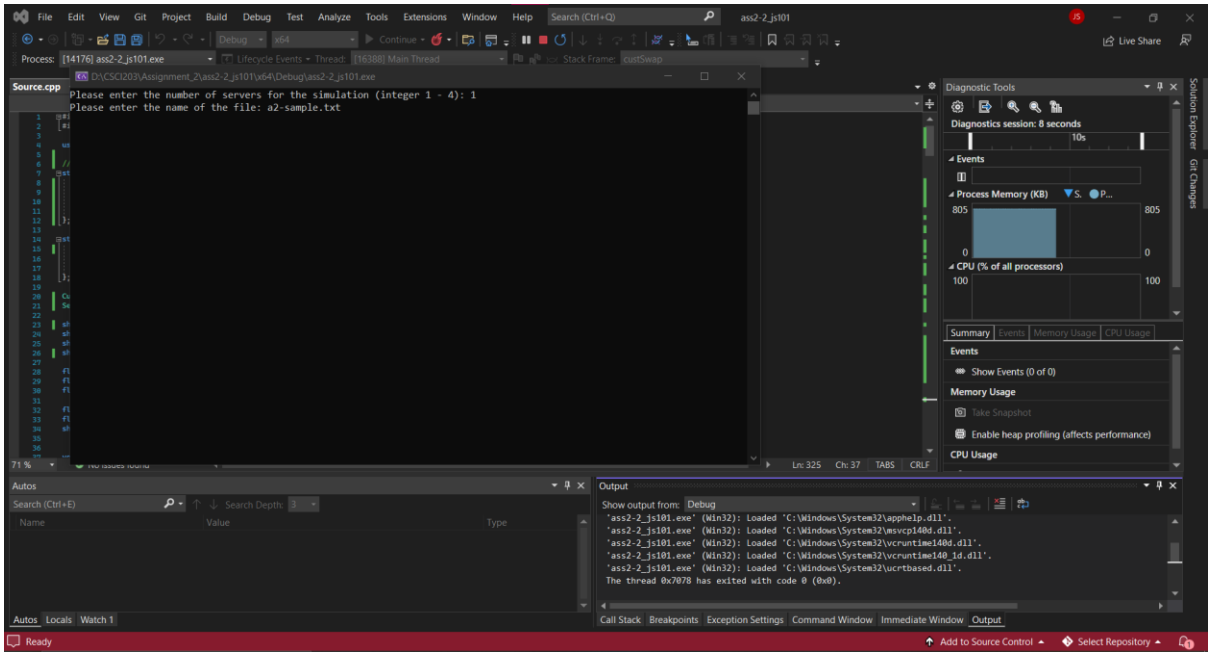
In total:

$O(((n -1) 4 log_2(m)) + 4 + (***above sum x 2))$

5.0 Screenshots

5.1 1 Server

Compilation



Execution

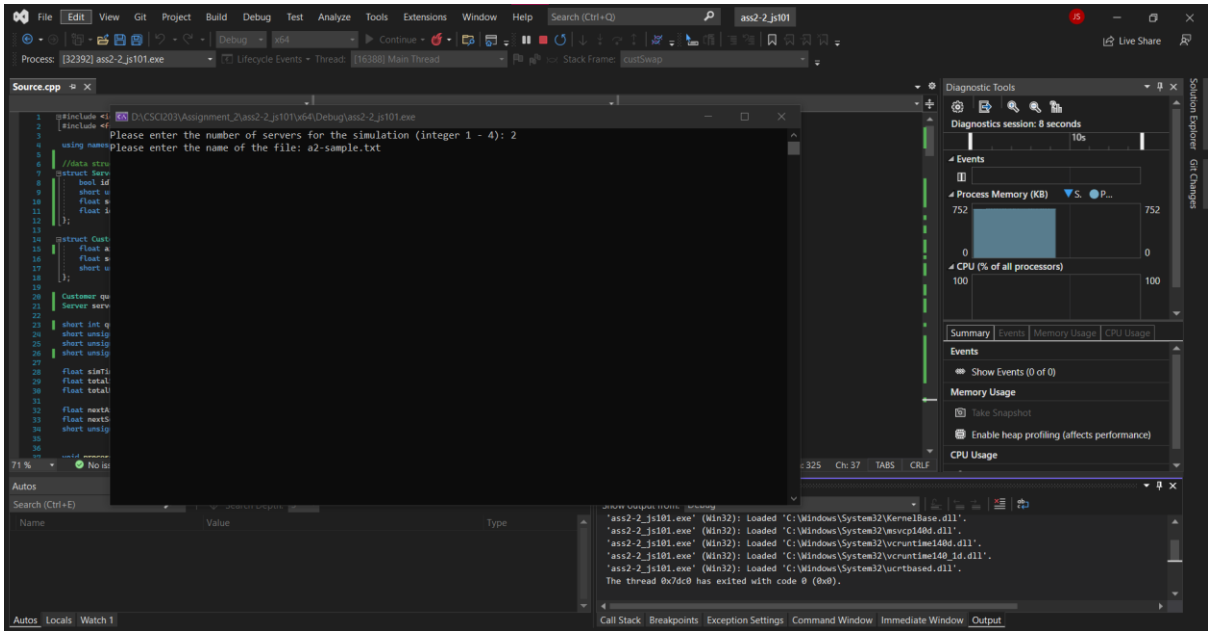
```
------------------------------------------------------------
Server #1
Customers Served: 100
Idle time: 2.7388
Idle rate (total idle time / total simulation time): 0.0021354
------------------------------------------------------------
Total simulation time: 1282.57
Total customers: 100
Average service time per customer: 12.7983
Average wait time per customer: 367.692
Maximum length of queue: 58
Average length of queue (total queuing time / time last service completed): 28.9551

D:\CSCI203\Assignment_2\ass2-2_js101\x64\Debug\ass2-2_js101.exe (process 14176) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the consc
le when debugging stops.
Press any key to close this window . . .
```

As we can see, the simulation took over 1000 time units, when only one server is available. The maximum length of the queue got to 58 when the total number of customers was only 100.

5.2 2 Servers

Compilation



Execution

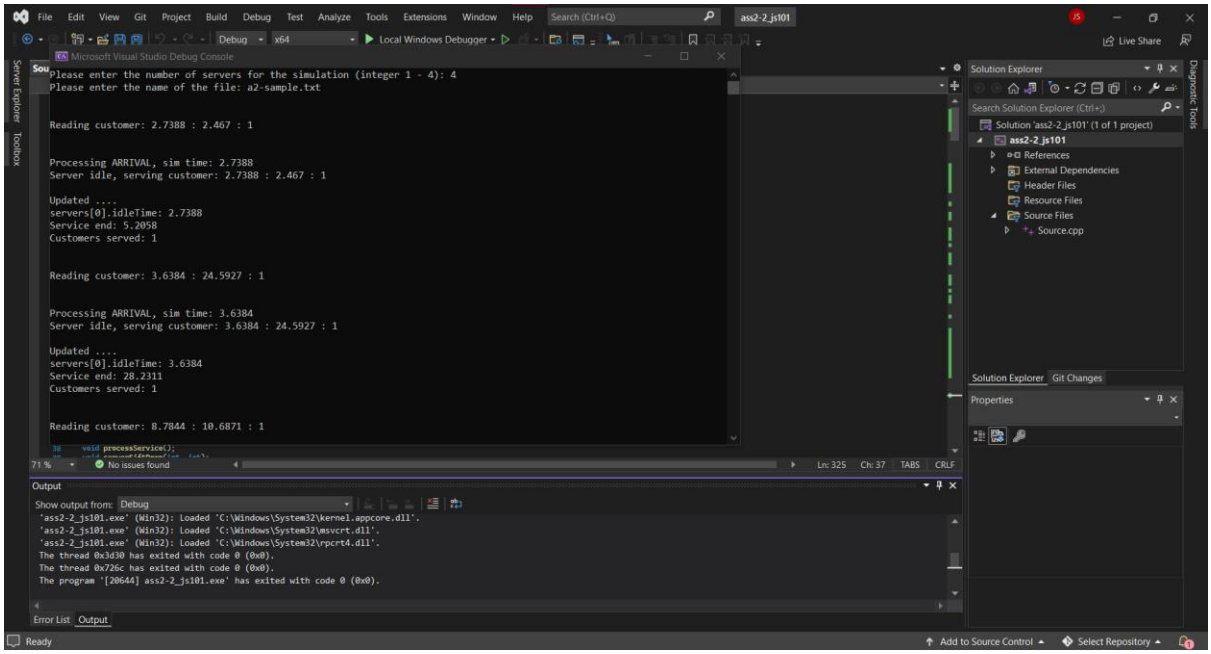
```
---------------------------------------------------------
Server #1
Customers Served: 54
Idle time: 10.7239
Idle rate (total idle time / total simulation time): 0.0163848
---------------------------------------------------------
---------------------------------------------------------
Server #2
Customers Served: 46
Idle time: 10.6362
Idle rate (total idle time / total simulation time): 0.0162508
---------------------------------------------------------
Total simulation time: 654.503
Total customers: 100
Average service time per customer: 12.7983
Average wait time per customer: 56.5691
Maximum length of queue: 20
Average length of queue (total queuing time / time last service completed): 8.72949

D:\CSCI203\Assignment_2\ass2-2_js101\x64\Debug\ass2-2_js101.exe (process 32392) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

As we can see, with even 2 servers, the total simulation time is halved, and the maximum queue length is drastically reduced.

5.3 4 Servers

Compilation



Execution

```
------------------------------------------------------------
Server #1
Customers Served: 27
Idle time: 217.263
Idle rate (total idle time / total simulation time): 0.409034
------------------------------------------------------------
------------------------------------------------------------
Server #2
Customers Served: 25
Idle time: 185.205
Idle rate (total idle time / total simulation time): 0.34868
------------------------------------------------------------
------------------------------------------------------------
Server #3
Customers Served: 28
Idle time: 215.835
Idle rate (total idle time / total simulation time): 0.406345
------------------------------------------------------------
------------------------------------------------------------
Server #4
Customers Served: 20
Idle time: 154.919
Idle rate (total idle time / total simulation time): 0.291661
------------------------------------------------------------
Total simulation time: 531.161
Total customers: 100
Average service time per customer: 12.7983
Average wait time per customer: 0.249317
Maximum length of queue: 1
Average length of queue (total queuing time / time last service completed): 0.0474074

D:\CSCI203\Assignment_2\ass2-2_js101\x64\Debug\ass2-2_js101.exe (process 20644) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
```

As we can see, with 4 servers, the simulation time is reduced to it’s minimum, and the maximum queue length is 1. We can derive from this in combination that 4 servers is probably a bit overkill.

NOTE: It is possible I did not update the servers idle time at the end of the simulation, i.e. when a server finished their last service but another is still going. I am too tired to check. Thank you.