BINUS UNIVERSITY
BINUS INTERNATIONAL

Final Project Cover Letter
(Group Work)

Student Information:

	Surname:	Given Name:	Student ID number:
1.	Sie	Jessica	2502053653
2.	Tantra	Joanne	2602127766

Course Code: COMP6048001

Course Name: Data Structures

Class: L2BC

Lecturer: NUNUNG NURUL QOMARIAH, S.Kom.,M.T.I., Ph.D.

Type of Assignment: Final Project

Submission Pattern:

Due Date: June 24 2022

Submission Date: June 24 2022

The assignment should meet the below requirements.
1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer's instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

Plagiarism/Cheating
BiNus International seriously regards all forms of plagiarism, cheating, and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity, and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

Declaration of Originality
By signing this assignment, I understand, accept, and consent to BiNus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student: Jessica Sie Joanne Tantra

COMPARISON OF THE EFFICIENCY BETWEEN QUEUEING SYSTEMS

Introduction

Problem description

While out shopping, we observed that retail stores in Jakarta, Indonesia mainly implement two kinds of queueing systems leading up to cashiers for checkout. department stores such as Sogo, Central and Metro implemented a multi-line, multiple point of sales queueing system. On the

other hand, stores like Uniqlo, H&M and even KKV implemented a single-line queueing system for their points of sales. This observation led to the research question, **How do the queueing systems implemented in retail affect the number of customers served over a period of time?**

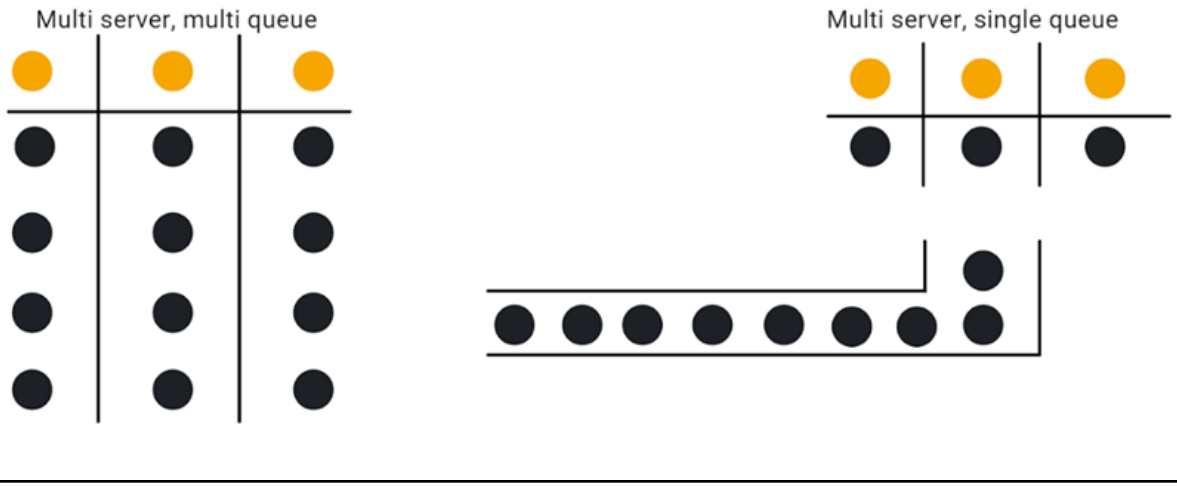

Figure 1 - Visual Representation of queueing systems implemented in retail, left - Sogo etc., right - Uniqlo etc.

This study is significant as both multi-line and single line queueing systems are implemented widely around the world in various sectors aside from retail, such as banking, healthcare systems, and service industry (Marsudi, 2013). With an efficient queueing system, businesses can maximize revenue and maintain high customer satisfaction. Therefore, results of this study can be used as a basis for retail managers and other queue management systems developers to maximize the efficiency of existing queueing systems.

As an efficient queueing system is vital in maximizing business revenue and maintaining high customer satisfaction, this poses the question of which of the existing queueing systems serve a greater number of customers over time. Henceforth, the results of this study aims to determine the relationship between queueing systems and their efficiency using queueing simulations made with queue data structure.

Hypothesis

From observing these queuing systems along with experiencing them first hand, my team and I hypothesized that the single line queueing system will be more efficient in the long term. In other words, as more and more customers arrive, the single line queue will be able to process

these customers much faster than multi-queue does. This is proven from a paper written by S Vijay Prasad, B Mahaboob, Ranadheer Donthi titled “A Comparative Study Between Multi Queue Multi Server And Single Queue Multi Server Queuing System”(Prasad et al, 2015). This research article has proved the superiority of single queue, multiple server (SQMS) over multi-queue, multiple server (MQMSM) shown by using the principle of finite mathematical induction.

Background

Queue Data Structure

Queues are a form of linear data structure which stores elements in sequential order. New elements can be appended at one end (rear of queue), and in queue elements can be removed at the other (front of queue). Queues are first-in-first-out (FIFO) structures that mimic the behavior of such systems as people in lines, cars at traffic lights, and files to be printed. Some queue operations include enqueuing to add an element to the queue, dequeuing to remove an element from the queue, and peek to return the value of the first element in the queue without deleting it. A Queue data structure is appropriate to implement for this investigation as it involves actual real life queues.

An article from the book, International Encyclopedia of the Social & Behavioral Sciences titled, Queues defines queueing systems as any time a “‘customer’ demands ‘service’ from some facility” and that their behavior is determined by “a) the arrival pattern, b) the service mechanism, and c) the queue discipline” (Smelser et Al, 2001). The arrival pattern refers to how often new elements arrive, while the service mechanism is the number of service points. The queue discipline refers to the manner in which a customer or customers are selected for service from those waiting, in other words FIFO, LIFO or by priority. Thus, the following factors will be the variables kept constant for this investigation to ensure a fair test.

Single-line, multiple POS Queueing system vs Multi-Line, multiple POS Queueing system

The difference between single-line, multiple POS queue system (shortened to single line queue) and multi-line, multiple POS queue system (shortened to as multi-line queue) can be derived from its name. As shown visually in *figure 2* single line queue system funnels customers into a single line and the multiple checkouts are designated to customers waiting in the queue on a first come, first served basis. On the other hand, a multi-line queue system requires customers to pick one of the multiple service desks and get into their respective lines.

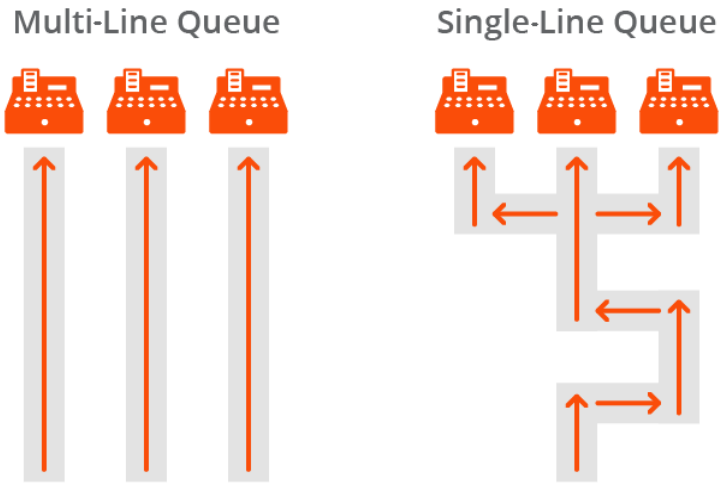

Figure 2 - Visual representation of queueing systems: Multi-line, multiple POS queue and Single line, multiple POS queue

Variables

Independent variable	Simulated Service Time /s
Dependent variable	Total number of customer served at that point in time / units
Controlled variables	Arrival pattern
	Service mechanism
	Queue discipline - FIFO
	Total simulation run time / s

Solution Implementation

Simulation structure

Since the problem statement and research question involves real life queues, we decided to program separate simulations to compare the efficiency of single and multi-line queue

systems. For each of the simulations, a linked list implementation of the queue data structure is used to represent a line in the simulation, with each node as a customer. The data field of each node stores the time needed for that customer to be served which is generated with a pseudo-random number generator. The linked list implementation is chosen over array based Queues because its dynamic sizing mirrors the size fluctuation of a real life queue.

As a representation of the individual points of sales, we created a user-defined data type called “POS” with the following member variables.

POS structure member variables	Data type	Description
active	boolean	State of POS (serving / not serving)
timeAt	integer	Countdown timer of the time a customer has left to spend at the POS

```
19  struct POS{
20      bool active; //
21      int timeAt; // 
22  };
```

Figure 3 - Points of sales structure

The individual simulations of the two queueing systems are built with the queue ADT and user defined structure of the POS. The structures of the simulations are illustrated as shown in *figure 4*. The single line queue simulation has an array with the element of the user-defined data type POS and a queue ADT to represent a line leading up to multiple cashiers. The multi-line queue simulation also has an array of the data type POS, but defines another array with the queue ADT as its elements.

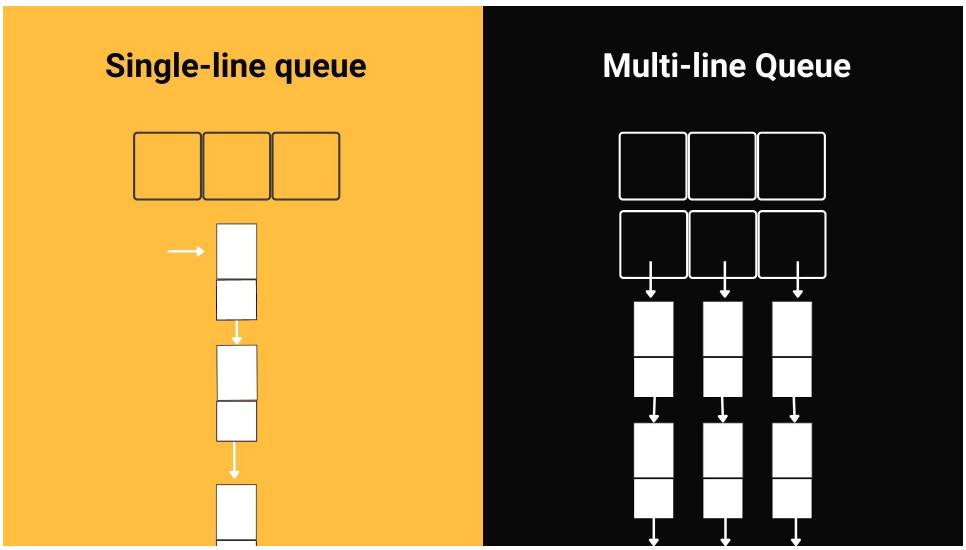

Figure 4 - Structures of single line and multi-line queue simulations

Queue hpp file

Creating Node class for individual elements of the linked list

```
// defining node class of linked list
class Node{
public:
    // PROPERTIES
    int data;
    Node* next;
    // OPERATIONS n CONSTRUCTORS
    Node():data(),next(NULL){};
    Node(int element): data (element), next(NULL){};
};
```

Declaring the Queue class with its respective properties and operations

```cpp
// defining queue class
class Queue{
    private:
        int size;
    public:
        // PROPERTIES
        Node *head; // aka queuefront
        Node * tail; // aka queuerear

        // methods
        // constructor
        Queue(): head(NULL), tail(NULL), size(0){};

        // getter and setter functions
        int getSize();
        void incrementSize();
        void decrementSize();

        // queue operations
        void enqueue(int element);
        int dequeue();
        int peek(); // returns the first in queue
        void free(); // deletes queue from memory

};
```

Implementing each of the operations of the Queue class

```cpp
// IMPLEMENTATION
int dsa::Queue::getSize(){
    return this->size;
}

void dsa::Queue::incrementSize(){
    this->size ++;
    return;

}
void dsa::Queue::decrementSize(){
    this->size --;
    return;

}
```

```cpp
void dsa::Queue::enqueue(int element){
    // creating new node
    dsa::Node *newNode = new dsa::Node(element);

    // link to linked list
    // checking if linked list is empty
    if(dsa::Queue::getSize() == 0){
        this->head = newNode;
        this->tail  = newNode;

        dsa::Queue::incrementSize();

        return;

    }
    // if not empty
    else{
        this->tail->next = newNode;
        this->tail = newNode;

        // setting new node pointer to NULL
        this->tail ->next = NULL;

        // increment size
        dsa::Queue::incrementSize();

    }
};
```

```cpp
int dsa::Queue::dequeue(){
    // temporary pointer points to head of list
    dsa::Node* temp = this->head;
    // move head pointer to next element
    this->head = this->head->next;
    // update queue size
    dsa::Queue::decrementSize();
    // remove first element in the list
    return temp->data;
}

int dsa::Queue::peek(){
    // check if queue is empty
    if(dsa::Queue::getSize() == 0){
        std::cout<< "queue is empty" <<std::endl;
        return -1;
    }
    else{
        return this->head->data;
    }
}

void dsa::Queue::free(){
    delete this->head;
    head = NULL;
}
```

Single line queue simulation

Structure

As a representation of the single line queue system, the multiple points of sales are represented by an array of the user-defined POS data type of which each of the elements is a point of sale.

Algorithm

Inside the main function

1. Instantiate a queue ADT

```
25      // instantiating Queue ADT
26      dsa::Queue myQueue;
```

2. Initializing variables

```
29      // declaring variables
30      int customerServed = 0; // number of customers served
31      int NUM_POS, range, startTime, simulationTime, arrivalTime;
32      int cTime = 0; // counter
```

3. Declare timer variables

```
81      // declaring variables for timer
82      auto start = chrono::steady_clock::now();
83      auto end = chrono::steady_clock::now();
84      int elapsedTime = int (chrono::duration_cast<chrono::seconds> (end-start).count());
```

4. Get user inputs

```
40      // getting user inputs
41      cout << " Number of servers or points of sales: ";
42      cin >> NUM_POS;
43      cout << "Start of service time: (eg.50) ";
44      cin>> startTime;
45      cout<< "Range of service time: (eg.30) ";
46      cin>> range;
47      cout<< "Arrival time: ";
48      cin>> arrivalTime;
49      cout<< "Simulation time/secs: ";
50      cin>> simulationTime;
```

5. Create, open and write to a new csv file called “singleLineQueue.csv”

```
55      // creating and opening new csv file to write
56      ofstream myFile;
57      myFile.open("singleLineQueue.csv");
58
59      myFile << "Service Time, Number of customers served\n";
```

6. Setup points of sales

```
62      // dynamically allocate an array for the POS structure
63      POS* POSArray = new POS[NUM_POS];
64      // set all POS to empty / initializing values
65      for(int i = 0; i< NUM_POS;i++){
66          POSArray[i].active = false;
67          POSArray[i].timeAt = 0;
68      }
```

7. Start simulation while elapsed time is less than simulation time from user inputs

```
82      // main loop to keep simulation running
83      while(elapsedTime < simulationTime){
```

8. Add customer to queue

```
84          if(cTime % arrivalTime == 0){
85              // adding customer to queue
86              myQueue.enqueue(rand()%range + startTime);
87          }
```

9. Serve customers by first removing a customer from the front of the queue to an unoccupied server. While the customer is being served, the state of the server changes to true and the countdown of time spent begins. After the time of the customer has elapsed, free up the POS state to false and increment the number of customers served.

```
90          // 1. get customer to cashier
91          for(int i =  0; i< NUM_POS; i++){
92              // check if POS is not serving anyone atm, and if queue is populated
93              if(POSArray[i].active == false && myQueue.getSize() != 0){
94                  
95                  POSArray[i].active = true; // occupied and serving cust
96                  POSArray[i].timeAt = myQueue.peek(); // returns the random value of customer
97                  
98                  // dequeue
99                  myQueue.dequeue();
100                 
101             }
102         }
103         
104         // 2. customer being served
105         for(int i = 0; i <NUM_POS; i++){
106             if(POSArray[i].active == true){
107                 // decrement time spent at POS
108                 POSArray[i].timeAt--;
109             }
110             
111             // 3. free up POS after customer has been served
112             // check if POS is done serving
113             if(POSArray[i].active == true && POSArray[i].timeAt ==0){
114                 // set cashier to open if the time limit is reached, increment number of customer served
115                 POSArray[i].active = false;
116                 customerServed++;
117             }
118             
119         }
```

10. Recalculate amount of time that has passed, to compare with the main loop condition

```
124         // calculating elapsed time  -> end-start
125         end = chrono::steady_clock::now();
126         elapsedTime = int(chrono::duration_cast<chrono::seconds> (end-start).count());
127         
128         if(elapsedTime != prevTime){
129             myFile<<elapsedTime<<","<<customerServed<<"\n";
130             cout<<elapsedTime<<" ";
131         }
```

11. After the main simulation loop is done running, close the csv file and show the number of customers that have been served.

```
136     myFile.flush(); // to remove an error
137     myFile.close();
138     cout<<"\nTotal number of customers served: "<< customerServed;
```

12. Delete the queue and POS array to prevent memory leaks

```
140     // prevent memory leaks
141     delete POSArray;
142     POSArray = NULL;
143     
144     myQueue.free();
```

Multi-line queue simulation

Structure

To represent the multi-line queue system, we utilized arrays with user-defined POS data structure as elements representing individual points of sales. Additionally, another with elements being a queue ADT as queues leading up to the servers.This way, every index would have its own point of sales and its corresponding line, and the array of queues can be passed into a function to determine the current shortest line for a customer to join.

algorithm

1. Creating a function to identify the shortest line of all the queues for a customer to join. The shortestQueue function takes in the array of queues as a parameter and returns the index of the shortest queue.

```
28  int shortestQueue(dsa::Queue queues[]){
29      int shortest = 0; // shortest - stores index of the shortest queue
30      // traversing each queue
31      for(int i = 0; i<3; i++){
32          if(queues[i].getSize() < queues[shortest].getSize()){
33              shortest = i;
34          }
35      }
36      return shortest;
37  }
```

2. Initialize variables

```
29      // declaring variables
30      int customerServed = 0; // number of customers served
31      int NUM_POS, range, startTime, simulationTime, arrivalTime;
32      int cTime = 0; // counter
```

3. Initializing simulation timer variables

```
101     auto start = chrono::steady_clock::now();
102     auto end = chrono::steady_clock::now();
103     int elapsedTime = int (chrono::duration_cast<chrono::seconds> (end-start).count());
```

4. Declare an array for multiple queue lines with queue ADT as elements

```
53      // declaring array with queue adt as elements
54      dsa::Queue qArr[NUM_POS];
```

5. Take-in user inputs

```
40      // getting user inputs
41      cout << " Number of servers or points of sales: ";
42      cin >> NUM_POS;
43      cout << "Start of service time: (eg.50) ";
44      cin>> startTime;
45      cout<< "Range of service time: (eg.30) ";
46      cin>> range;
47      cout<< "Arrival time: ";
48      cin>> arrivalTime;
49      cout<< "Simulation time/secs: ";
50      cin>> simulationTime;
```

6. Create, open and write to csv file

```
75      // creating and opening csv file
76      ofstream myFile;
77      myFile.open("multiLineQueue.csv");
78
79      myFile << "Service Time, Number of customers served\n";
```

7. Setup multiple POS as an array of user-defined data type, POS.

```
81      // POS SETUP
82      // allocating array for POS structure
83      POS* POSArray = new POS[NUM_POS];
84      // initializing POS values
85      for(int i = 0; i < NUM_POS; i++){
86          POSArray[i].active = false;
87          POSArray[i].timeAt = 0;
88
89      }
```

8. Start simulation while elapsed time is less than simulation time from user inputs

```
106     while(elapsedTime < simulationTime){
```

9. Checks if the interval between customers has been reached and adds a customer to one the queues. The queue is chosen by calling the shortestQueue function, previously defined, which returns the index of the shortest queue in the array of queue.

```
107         // checks if interval between each customer is reached
108         if(cTime % arrivalTime == 0 ){
109 
110             // add a customer the shortest of the queues
111             int shortestQIndex = shortestQueue(qArr);
112             qArr[shortestQIndex].enqueue(rand() % range + startTime);
113 
114         }
```

10. Serve customers by first removing a customer from the front of the queue to an unoccupied server. While the customer is being served, the state of the server changes to true and the countdown of time spent begins. After the time of the customer has elapsed, free up the POS state to false and increment the number of customers served.

```
116         // SERVING CUSTOMERS
117         // 1. getting a customer to the cashier in each line
118         for(int i = 0; i< NUM_POS; i++){
119             if(POSArray[i].active == false && qArr[i].getSize() != 0){
120                 POSArray[i].active = true; // serving customers
121                 POSArray[i].timeAt = qArr[i].peek();
122 
123                 qArr[i].dequeue();
124 
125             }
126         }
127 
128         // 2. customer spending time at POS
129         for(int i = 0; i< NUM_POS; i++){
130             if(POSArray[i].active == true){
131                 POSArray[i].timeAt --; // decrement time spent at POS
132             }
133 
134 
135             // 3. free up POS after customer is served
136             // cheking if customer is done
137             if(POSArray[i].active == true && POSArray[i].timeAt == 0){
138                 POSArray[i].active = false; // open POS to new customer
139 
140                 customerServed++; // incrementing number of customer served
141             }
```

11. Recalculate the amount of time elapsed

```
149         // calculating the time elapsed
150         end = chrono::steady_clock::now();
151         elapsedTime = int(chrono::duration_cast<chrono::seconds> (end-start).count());
```

12. Close csv file

```
161     myFile.flush();
162     myFile.close();
163     cout<<"\nTotal number of customers served: "<< customerServed;
```

13. Delete pointers and arrays to prevent memory leaks

```
165     // prevent memory leaks
166     delete POSArray;
167     POSArray = NULL;
168     for (int i = 0 ; i < NUM_POS; i++){
169         qArr[i].free();
170     }
171     delete qArr;
```

Program Manual

Inputs

Previously established as factors which determine the behavior of a queueing system above, the user inputs below are kept as constants for both single line queue and multi-line queue simulations to create a fair comparison.

Input	Variable name	Description	Value kept constant
Service mechanism	NUM_POS	Number of servers or points of sales	3
Start of service time / (0-100%)	startTime	Probability that a customer gets in queue during a single tick	50
Range of service time /ticks	range	Maximum time needed to serve a new customer	30
Arrival time / ticks	arrivalTime	Interval between each new customer	5
Simulation time / s	simulationTime	Length of time the simulation will take place	100

Outputs

- Total number of customers processed
- CSV file for simulation results

Runtime

Running the driver file (singleLineQueue.cpp) for the single line queue simulation or the multi-line queue simulation (multiQueue.cpp) , the user will be prompted to input the queue parameters previously discussed in the inputs section. The simulation then runs while printing out the seconds that have passed until the user input simulation run time. After the simulation is completed, the program outputs the total number of customers served and a csv file showing the simulation results.

```
PS C:\Users\Jessica\Desktop\Sem 2 CS\data structures 6mcs\DataStructuresFP> cd "c:\Users\Jessica\Desktop\Sem 2 CS\data structures 6mcs\DataStructuresFP\" ;
if ($?) { g++ singleLineQueue.cpp -o singleLineQueue } ; if ($?) { .\singleLineQueue }
Start of service time: (eg.50) 50
Range of service time: (eg.30) 30
Arrival time: 5
Simulation time/secs: 100
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55
56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
Total number of customers served: 15193092
PS C:\Users\Jessica\Desktop\Sem 2 CS\data structures 6mcs\DataStructuresFP>
```

Figure 5a- Terminal input and output for single line queue

```
PS C:\Users\Jessica\Desktop\Sem 2 CS\data structures 6mcs\DataStructuresFP> cd "c:\Users\Jessica\Desktop\Sem 2 CS\data structures 6mcs\DataStructuresFP\" ;
if ($?) { g++ multiQueue.cpp -o multiQueue } ; if ($?) { .\multiQueue }
Start of service time: (eg.50) 50
Range of service time: (eg.30) 30
Arrival time: 5
Simulation time/secs: 100
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
 100
Total number of customers served: 13975909
```

Figure 5b- Terminal input and output for multi-line queue

Investigation

Simulation Results

Service Time /s	Number of customers served / units							
	single line queueing system				multi-line queueing system			
	Trial 1	Trial 2	Trial 3	Average	Trial 1	Trial 2	Trial3	Average
1	309107	324212	303474	312264	297488	291798	304363	297883
2	574065	608771	554151	578996	567993	510005	561733	546577
3	870729	932780	791566	865025	872900	790317	857711	840309
4	1189167	1260064	1089256	1179496	1175571	1032747	1166945	1125088
5	1498867	1588996	1396083	1494649	1497760	1308885	1365052	1390566
6	1825911	1915934	1685095	1808980	1805078	1525324	1586009	1638804
7	2152379	2231225	1929967	2104524	2126898	1674563	1859876	1887112
8	2471952	2566097	2122235	2386761	2440878	1809811	2090107	2113599
9	2791940	2894325	2293030	2659765	2747466	1927460	2362992	2345973
10	3120530	3215366	2418666	2918187	3061423	2046991	2487532	2531982
11	3431559	3546677	2597723	3191986	3385006	2172135	2586899	2714680
:	:	:	:	:	:	:	:	:
99	29679793	19217580	15080376	21325916	28920938	18367156	13723543	20337212
100	29686462	19535758	15193092	21471771	29131886	18615154	13975909	20574316

Graphical analysis

Plotting the service time against the average number of customers served for each of the queueing systems results in the graph shown in *figure 6* .

The threadline and its equation in the form of Y=mX +C shows a linear relationship between the service time and the number of customers served, which shows that the number of customers processed increases linearly with time.

The gradient of the graph shows the efficiency, or number of customers served per period of time, of each queueing system. The gradient of the orange line for the multi-line queueing system is 198155 customers served per second of running the simulation. On the other hand, the blue line representing the single line queuing system has a gradient of 206524 customers served per second. Therefore the single line queueing system is more efficient as it

is able to service more customers over a set time period compared to the multi-line queueing system.

Figure 6- Graph of simulation service time against number of customers served

Conclusion

The obtained values of the efficiency of each queueing system are 206524 customers served per second for single-line queue and 198155 customers served per second for multi line queue. Therefore, the results of this investigation proves our initial hypothesis, that single line queue systems are more efficient than multi-line queue systems, to be true for a first-in-first-out queue discipline, an arrival time of 5 ticks and a 3 POS service mechanism.

As further extension to this investigation, a longer than 5 ticks interval between customers or a random interval (arrival time) could be input as a constant parameter in the simulation. This is to compare the queueing systems in situations closer to real life. Additionally, changing the number of the points of sales can also be another independent variable to answer the question like why the slower multi-line queueing system is still used in department stores like Sogo despite its lower efficiency. More trials with different queue parameters are needed to come to a conclusive result for other queueing system situations.

References

Marsudi, M., & Shafeek, H. (2013). Production line performance by using queuing model. *IFAC Proceedings Volumes*, *46*(9), 1152–1157. https://doi.org/10.3182/20130619-3-ru-3018.00515

Prasad, V., V.H, B., & Koka, T. A. (2015). Mathematical Analysis of Single Queue Multi Server and Multi Queue multi server queuing models: Comparison study. *Global Journal of Mathematical Analysis*, *3*(3), 97. https://doi.org/10.14419/gjma.v3i3.4689

Smelser, N. J., & Baltes, P. B. (2001). Queues. In *International Encyclopedia of the Social & Behavioral Sciences*. essay, Pergamon.