Post office queuing system

February 2021

Abstract

This project is designed to study the efficiency of the post office queuing system. The simulation was programmed in C, and aims to answer the question: given a certain rate of new customers in a post office branch, what is the optimal number of service points?

1 Program design

1.1 Design decisions and assumptions

The format of input file that I chose to use, is each parameter on its own line consisting of a key value pair separated by whitespace. Commented lines beginning with a hash(#) are ignored, even when containing parameter names. If a parameter is not specified in the file, default values are used rather than throwing an error. If this happens, it is printed to the output file to make you aware that the program has used a default value. Any errors that occur during the reading of the input file are handled according to section 1.2

The queue entity is modelled with a doubly linked list. This allows the program to keep track of the front and the back of the queue, and traverse in either direction. This decision was made due to the requirement of new people joining the back of the queue, as well as people from the front leaving to get served. The two main methods for the queue are push which adds a person to the back, and shift which removes the person in the front.

Throughout the programs operation, various statistics are collected at different stages for the purpose of reporting averages at the end of a simulation. To facilitate this, there are two structs which contain individual statistics. One to keep track of an individual post office simulation, and another one to keep track of all of the simulations (when $numSims > 1$).

The aspects of the program determined by random number generation are the number of customers who arrive at each time interval, the time it takes to serve a particular customer, and each customers tolerance/patience to waiting. Each of these is given an average by the parameter file. In order to achieve this average, a uniform distribution is used. This was chosen for a few reasons,

including making the parameter file easier to specify and better modelling the flow of customers in a post office.

1.2 Coding practices

Basic code style standards are adhered to, such as a 3 space indent, curly braces on a new line and spaces inside if statement parenthesis. Logical chunks of code are split between multiple files, and imported when necessary. This allows for easy maintenance of the code base as all the code relating to the queue is found in *queue.c/h* and so on. It also means that certain aspects of the code base could be reused in another project in future. All header files include *ifndef* guards to prevent duplicate definitions.

Memory is allocated using the *malloc* function, and freed as soon as it is no longer required.

Error handling is managed using the status code method of returning 0 when functions successfully executed and a negative number when a fatal error occurred, and a positive number for warnings. It should be noted that this is not done for every single function as some functions don't have any error-prone operations. All errors are printed to *stderr*. I chose to use the -1 status code to indicate a file error and -2 to indicate a memory fault. By assigning different types of errors different status codes, the program can deal with them differently. If an error occurs in a function, rather than exiting immediately, the function returns its status code to the function which called it, and so on until the error bubbles up to the main function where it is dealt with (exit for fatal errors). This allows the reaction to an error to be easily changed in the future.

All of the major functions are documented with ANSI standard compliant comments. This would allow a third party tool to generate documentation from my source code.

1.3 Limitations

There are certain technological limitations to this program. Large numbers of iterations may take a long time to run due to the program being single threaded, preventing it from benefiting from a multi-cored system. In addition to this, my program has limitations in the number of iterations it can safely run before the tracking statistics exceed the maximum integer size (usually $2^{31} - 1$).

2 Experimentation

2.1 Aims

This experiment aims to answer the question of what is the optimum number of service points to have at a post office. For the purpose of this experiment, I will consider optimum to mean the fewest number of service points required to ensure that on average, less than 10% of customers either timeout and leave,

or arrive when the queue is full. This is equivelent to saying 90% of customers who arrive, eventually get served (are fulfilled).

2.2 Methods

To run this experiment, I am using parameters that I think would be typical for a relatively small British post office. Assuming the post office is open for about 8 hours, and each interval is equivalent to 5 minutes, there will be approximately 100 intervals in a day.

Accordingly, I am using the following input parameters in the input file:

```
maxQueueLength    10
numServicePoints 3
closingTime       100
# additional parameters
averageNewCustomersPerInterval  3
averageServeTime                2
averageToleranceToWaiting       15
```

I've started off with 3 service points. The command to run the simulation is:

simQ inputFile.txt 1000 outputFile.txt

The resultant output in outputFile.txt is:

```
Parameters read from inputFile.txt:
Max queue length:          10
Number of service points:  3
Closing time:              100
Average new customers per interval: 3
Average tolerance to waiting:       15
Average serve time:                 2


Average across all 1000 simulations
Average number of fulfilled customers:    157.32
Average number of unfulfilled customers:  91.31
Average number of timed out customers:    1.08
Average wait time per customer:           4.13
Average time from closing to completion:  9.20
```

Running the same command multiple times results in slightly different results each time due to the use of random numbers, but nothing substantial changes.

I ran the simulation a few more times with different number of service points each time to see what the optimum number is, and the results are listed below under conclusions.

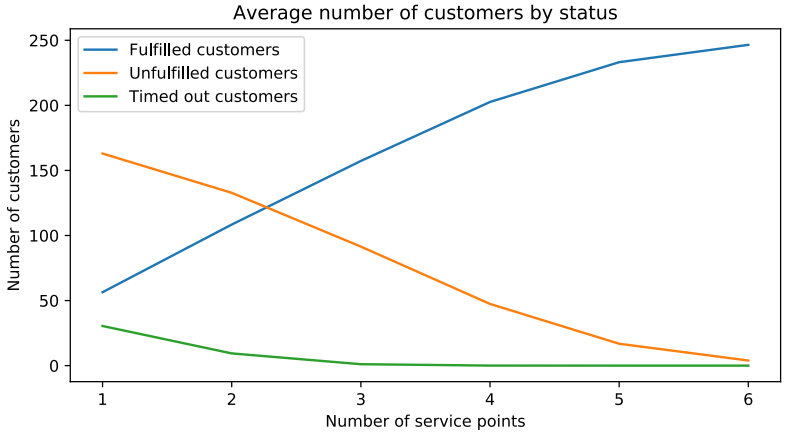

Figure 1: The effect of changing the number of service points

2.3 Conclusions

I kept all other parameters the same, changing the number of service points to observe the number of customers who were fulfilled, unfulfilled and timed out. As shown in Figure 1, when there are 4 service points, approximately 81% of customers get served, and when there are 5 service points, approximately 93.3% of customers get served. As stated in the aims section, I am assuming that serving less than 90% of customers is unacceptable to a post office, so in this scenario, they would be best suited to having 5 service points. Alternatively, they could train their staff to reduce the average serve time, or put out newspapers for customers to read in the queue to make them more willing to join a long queue and have a higher tolerance to waiting.

2.4 Repeatability

This experiment is highly repeatable, as subsequent runs of the same parameter set are likely to produce very similar results, only differing by a few decimal places.

2.5 Limitations

The limitations of this experiment are that the random number generator probably doesn't represent real life accurately, as on certain days such as the week before Christmas, a post office is likely to be significantly busier than at other times, yet my random number generator is constant. Furthermore, this simulation is only useful if the parameters are set to realistic values, which would require monitoring an actual post office.