Systems modelling and simulation

Assignment 1

Author: Kemal Cikota,
Armin Rezaie Valiseh
Teacher/examiner: Mauro Caporuscio
Semester: Fall 2024
Subject: Computer Science

Abstract

Instruction for running and executing code!
The code that was written to complete this assignment can be found in the root folder. The only prerequisite needed in order to run is to have the necessary libraries installed in the VScode workspace. Theese include matplotlib, numpy, datetime and pandas. IF not already in your workspace. Use the *Pip install *desired library** to download it.

The python scripts that have names that start with 'problem' are the scripts that are directly used to solve the problems. The 'resturangPlot' and 'strip' scripts are just helper scripts for task 3.

The **raw data** for each task can be found in the 'RAW DATA' folder. This just includes all of the data which has been printed in the VScode terminal.

If for some reason, the **images** in this report can not be seen because they got corrupted, there are raw .png versions of them in the 'IMG' folder.

Contents

1 Task 1

For the first task, we wanted to create a queuing network that is more aligned with our field of studies, computer science/software technology as opposed to creating a network that is based on a manufacturing plant, grocery store, etc.

The system we chose to describe using a queueing network is a multi-threaded program with two different types of threads, I/O-bound threads and CPU-bound threads. The program runs several threads, where each thread can be of two types.

- We have I/O-bound threads which can be defined as being responsible for input/output operations, this can for example be: reading files or data from a disc, making network requests or waiting for user input.

- We also have CPU-bound threads which are purely computational and perform operations on the CPU. This could for example be CPU-related operations, calculations, or data analysis.

The start of the system, where the customers (threads) arrive, and the arrival process starts, the scheduler will submit each thread to a specific queue for that type of thread. The arrival of said threads are a part of a theoretically infinite calling population, which means threads can keep arriving without a predefined upper limit. Once the threads are sent to the CPU for computation or to an I/O resource to handle disc/network operation, they will departure from the system completely.

This is a diagram showing the queueing network:

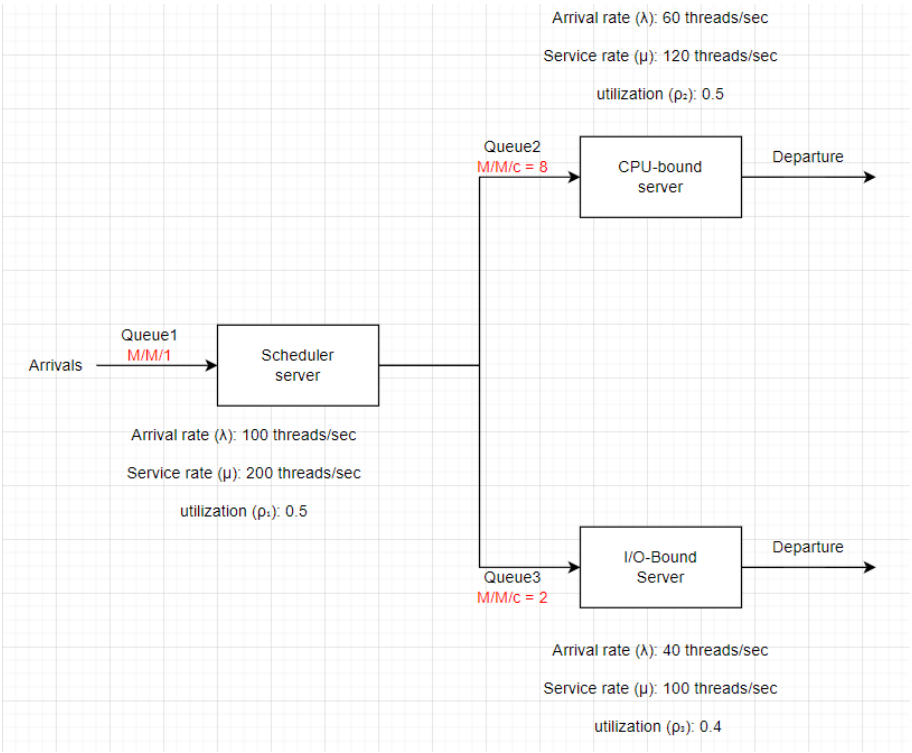

Figure 1.1: Queueing network diagram.

As we can see from Figure 1.1, we have 3 separate queues that we need to take into account. The first queue, which is the scheduler queue, is a poison process, which implies an exponential distribution for the interarrival times. This would be fitting because we have a theoretically infinite amount of threads that could be handled and they arrive at random points in time. For simplicity, we have only one scheduler without any nested scheduling processes, which implies that we have only one server for this queue. This means that the formal notation for this queue is **M/M/1**. The arrival rate for this queue (which is the arrival rate for the system itself) is λ_1 = 100 threads/s and the service rate for the scheduler is μ_1 = 200 threads/s which gives us a utilization factor of ρ_1 = 0.5. from this information we can draw conclusions for the following performance metrics:

$$L = \frac{\lambda}{\mu - \lambda} = \frac{\rho}{1 - \rho} = 1.000 \tag{1}$$

$$w = \frac{1}{\mu - \lambda} = \frac{1}{\mu(1 - \rho)} = 0.010 \tag{2}$$

$$w_Q = \frac{\lambda}{\mu(\mu - \lambda)} = \frac{\rho}{\mu(1 - \rho)} = 0.005 \tag{3}$$

$$L_Q = \frac{\lambda^2}{\mu(\mu - \lambda)} = \frac{\rho^2}{1 - \rho} = 0.500 \tag{4}$$

Since the first queue serves the threads one at a time, the arrivals for the CPU-bound queue should also follow a poison process, making the interarrival time exponential, since a given thread can have variable processing times. We chose c = 8 for this queue because we decided that the CPU has 8 CPU cores, which is supposed to mimic a mid-to-high-end CPU, similar to something that can be found in an AMD Ryzen 7, Intel i7 or something similar. This means that the formal notation for this queue **M/M/c**. 60% of the threads from the scheduler are CPU-bound threads which gives us an arrival rate of λ_2 = 60 threads/s, service rate of μ_2 = 120 threads/s and a utilization rate of ρ_2 = 0.063. From this information we can draw conclusions for the following performance metrics:

$$L = c\rho + \frac{(c\rho)^{c+1} P_0}{c!(c(c!)(1 - \rho)^2)} = c\rho + \frac{\rho P(L(\infty) \geq c)}{1 - \rho} = 0.500 \tag{5}$$

$$w = \frac{L}{\lambda} = 0.008 \tag{6}$$

$$w_Q = w - \frac{1}{\mu} = 0 \tag{7}$$

$$L_Q = \lambda w_Q = \frac{(c\rho)^{c+1} P_0}{(c(c!)(1 - \rho)^2)} = \frac{\rho P(L(\infty) \geq c)}{1 - \rho} = 0 \tag{8}$$

$$P_0 = \left[\sum_{n=0}^{c-1} \frac{\left(\frac{\lambda}{\mu}\right)^n}{n!} + \left(\frac{\left(\frac{\lambda}{\mu}\right)^c}{c!} \left(\frac{c\mu}{c\mu - \lambda} \right) \right) \right]^{-1} \tag{9}$$

$$= \left[\sum_{n=0}^{c-1} \frac{(c\rho)^n}{n!} + \left(\frac{(c\rho)^c}{c!}\frac{1}{1-\rho}\right)\right]^{-1} = 0.607 \tag{10}$$

The third queue, which is the queue for the I/O-bound threads, 40% of the threads from the scheduler arrive. This gives us an arrival rate of λ = 40, a service rate of μ = 100 and since we decided to take in to account network operations and operations where the program has to read from a disc, we will get c = 2. Similarly to the CPU-bound queue, the threads arrive from the scheduler queue which is a poisson process that handles one process at a time, the formal notation for this queue will also be **M/M/c**. From this we can draw conclusions for the following performance metrics.

$$L = c\rho + \frac{(c\rho)^{c+1}P_0}{c!(c(c!)(1-\rho)^2)} = c\rho + \frac{\rho P(L(\infty) \geq c)}{1-\rho} = 0.417 \tag{11}$$

$$w = \frac{L}{\lambda} = 0.010 \tag{12}$$

$$w_Q = w - \frac{1}{\mu} = 0 \tag{13}$$

$$L_Q = \lambda w_Q = \frac{(c\rho)^{c+1}P_0}{(c(c!)(1-\rho)^2)} = \frac{\rho P(L(\infty) \geq c)}{1-\rho} = 0.017 \tag{14}$$

$$P_0 = \left[\sum_{n=0}^{c-1} \frac{\left(\frac{\lambda}{\mu}\right)^n}{n!} + \left(\frac{\left(\frac{\lambda}{\mu}\right)^c}{c!}\left(\frac{c\mu}{c\mu-\lambda}\right)\right)\right]^{-1} \tag{15}$$

$$= \left[\sum_{n=0}^{c-1} \frac{(c\rho)^n}{n!} + \left(\frac{(c\rho)^c}{c!}\frac{1}{1-\rho}\right)\right]^{-1} = 0.667 \tag{16}$$

2 task 2

2.1 Task 2.1

In order to develop a Random Number Generator (RNG) according to the *Multiplicative Congruential Method* (MCM) and generate a sequence of 30 numbers for the different parameters we used python, the source code for this specific part of the assignment can be found in the **Problem2_1.py** file. For the algorithm we used the same formula that can be found in chapter 3.1 of the M2.1 reading material which is as following:

$$X_{i+1} = (aX_i + c) \ \ mod \ \ m, \quad i = 0, 1, 2, \ldots \tag{17}$$

The algorithm was run 30 times in a for-loop in order to get 30 generated integers. In order to make the code more modular and allow for different parameters for the MCM, the algorithm together with a print-statement was encapsulated in a function.

for subtask a) we had to use the parameters $X_0 = 8, a = 13, m = 16$ and we found that this just generated an array containing 8's. This makes sense because when we multiply $X_0 * a = 104$ and then apply the $mod16$ operation we will get a remainder which

is 8, which is the initial seed, so it makes sense that this would generate an array of just 8's.

for subtask b) we had to use the parameters $X_0 = 8, a = 11, m = 30$. We found that this created an array of the integers 8 and 28 alternating. This is because when we first multiply $X_0 * a = 88$, we get a remainder of 28 when applying the $mod30$ operation. When we then multiply $X_1 * a = 308$ and then apply the $mod16$ operation we get $X_2 = 8$ which is the same as the initial seed.

for subtask c) we had to use parameters $X_0 = 7, a = 7, m = 16$. We found that this faces the same problem as the parameters from b). Theese parameters created an array of integers 7 and 1 alternating. This is because when we first multiply $X_0 * a = 49$, we get a remainder of 1 when applying the $mod16$ operation. When we then multiply $X_1 * a = 7$ and then apply the $mod16$ operation we get $X_2 = 7$ which is the same as the initial seed.

for subtask e) we had to use the parameters $X_0 = 8, a = 7, m = 25$. Theese parameters would still generate a cycle in the array [8, 6, 17, 19, 8, 6, 17, 19, 8, ...] which is a better result than the other parameters would generate but still pretty bad considering that the cycle is short and predicable to the naked eye.

In order to get better random numbers which are less predictable and perform better in the uniformity and frequency test we would want to use a modulus which is a prime or a power of a prime. This will ensure that the the multiplier will generate numbers that spread evenly across all possible remainders.

Raw data can be found in task 2.1.txt

2.2 Task 2.2

For this task, we implemented the same RNG using the MCM from Task 2.2, again enclosed in a function. In order to test the generated numbers for uniformity we used the Kolmogorov-Smirnov (K-S) test. The K-S test works in 5 different steps. First, we had to normalize the values from the array generated by the RNG in the range [0, 1] and then sort the normalized numbers from smallest to largest. Secondly, we had to compute D+ and D- from the normalized array. Formulas for them can be found below, where i is the index of a given random number in the normalized array, N is the length of the array and R_i is the random number itself at the index i.

$$D^+ = \max_{1 \leq i \leq N} \left\{ \frac{i}{N} - R_i \right\} \tag{18}$$

$$D^- = \max_{1 \leq i \leq N} \left\{ R_i - \frac{i-1}{N} \right\} \tag{19}$$

The third step is to compute D, which is the largest number between D+ and D-. The fourth step is to locate the critical value for the K-S test. For this assignment we chose to use the significance levels 0.01 and 0.05. The critical values can be found by looking at Table A.8 in the referenced appendix on MyMoodle. The fifth step is to just compare the D-value to the critical value.

The K-S test showed that the random numbers generated from the parameters from a),

b) and c) does not pass the test for uniformity. This makes sense since the generated numbers are either arrays containing the same number or 2 numbers alternating. However, the generated numbers from parameter d) passed the K-S test, showing that the generated numbers are well spread out over the interval [0, 1] even though we can spot a relatively short cycle.

For the next test, which is the autocorrelation test, we test the independence of the individual numbers' probability of appearing in our array of generated numbers. The first step is to calculate M, which is the number of pairs of numbers that will be compared for the autocorrelation based on the chosen lag. The lag is the distance between two values in the sequence that are compared. The next step is to calculate $\hat{\rho}$, which is the estimate of the autocorrelation at the chosen lag. In the next step, we calculate $\sigma_{\hat{\rho}}$, which is the standard deviation that gives us a measure of variability in our estimate of $\hat{\rho}$. This allows us to determine if the observed correlation is significant or random. In the last step, we conduct a hypothesis test to see if Z_0, which is the ratio between $\hat{\rho}$ and $\sigma_{\hat{\rho}}$, lies in the range $-1.96 \leq Z_0 \leq 1.96$.

The autocorrelation test showed that the arrays created from all of the parameters a), b), c) and d) failed the autocorrelation which means that it is easy to predict the next generated number from a given number in the array. For a), b) and c) this doesen't come as a big surprise but even for d) we could see that it is a small cycle of numbers constantly alternating.

Raw data can be found in task 2.2.txt

2.3 Task 2.3

For this subtask we had to implement a Random Variate Generator (RVG) for the following probability distribution:

$$f(x) = \begin{cases} \frac{1}{2}(x-2), & \text{if } 2 \leq x \leq 3 \\ \frac{1}{2}\left(2 - \frac{x}{3}\right), & \text{if } 3 < x \leq 6 \\ 0, & \text{otherwise} \end{cases} \tag{20}$$

In order to do this, we will use a technique called inverse transform sampling. In short, it means that we have to find the CDF for the given probability distribution and then calculate it's inverse. This will allow us to take a random number from the uniform distribution in the range [0, 1], which is our sample, and then transform that number and map it on our custom distribution. We will start by first finding the CDF for this probability distribution function (PDF). We do this by first integrating the first part of the PDF in order to get the normalization constant and then add this constant when evaluating the CDF to make sure that the total probability sums to 1 as this is a prerequisite for a valid CDF.

$$F(x) = \int_2^3 \frac{1}{2}(x-2)\,dx = \frac{1}{2}\left[\frac{x^2}{2} - 2x\right]_2^3 = \frac{1}{4}$$

Once we have this normalization factor, we can now start computing the CDF piece-by-piece.

$$F(x)_1 = \int_2^x \frac{1}{2}(x-2)\,dx = \frac{1}{2}\left[\frac{x^2}{2} - 2x\right]_2^x = \frac{1}{2}\left(\frac{x^2}{2} - 2x + 2\right)$$

The next part:

$$F(x)_2 = \frac{1}{4} + \int_3^x \frac{1}{2}\left(2 - \frac{x}{3}\right)\, dx = \frac{1}{4} + \frac{1}{2}\left[2x - \frac{x^2}{6}\right]_3^x = -\frac{x^2}{12} + x - 2$$

This gives us the final CDF:

$$F(x) = \begin{cases} \frac{x^2}{4} - x + 1, & 2 \leq x \leq 3 \\ -\frac{x^2}{12} + x - 2, & 3 < x \leq 6 \end{cases}$$

Now we have the CDF, we now have to get the inverse of the CDF in order to get the transformed function. This is done by setting each piece of the CDF to some variable and then solving for X in the function.

$$F_1^{-1} = y = \frac{x^2}{4} - x + 1 \quad \Rightarrow \quad y = 2 + \sqrt{2}, \quad 0 < x \leq \frac{1}{4}$$

$$F_2^{-1} = y = 6 - 2\sqrt{3 - 3x}, \quad \frac{1}{4} < x \leq 1$$

Which gives us the transformed function:

$$X = \begin{cases} 2 + \sqrt{2}, & 0 < x \leq \frac{1}{4} \\ 6 - 2\sqrt{3x}, & \frac{1}{4} < x \leq 1 \end{cases}$$

It is important to note that the bounds for this transformed function are in the range [0, 1], which is important as this will make sure that we can successfully route a randomly generated number from the uniform distribution which are in the range of [0, 1] to this new distribution.

In the source code which can be found in **Problem2_3.py**, we only have to generate a random number from the uniform distribution and then insert it in one of the equations in the transformed function, depending on the range that the randomly uniform value will take, this is done using a simple if-statement. This logic was enclosed in a function so that we could run it 1000 times in a for-loop and append all of the generated values in an array. using matplotlib we could then use that array to plot a histogram showing the different values according to our custom distribution. Here is that histogram:

This is a diagram showing the queueing network:

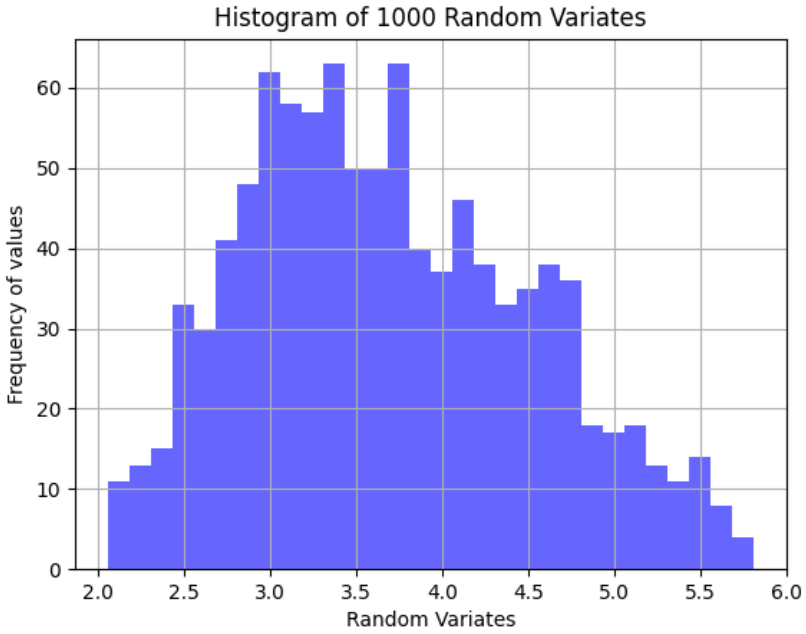

Figure 2.2: Histogram showing frequency of random variates.

As we can see here, the histogram shows the frequency of a value being generated on the y-axis and on the x-axis it shows which value the random variate actually took. As we can see this looks a lot like a triangular distribution which is reasonable given the original function which was given to us in the task.

Raw data can be found in task 2.3.txt

3 task 3

Before we present the analysis of the system and provide suggestions of how to improve the system we will have to show our assumptions for how this system currently operates as they will be important to have the correct context for our improvement suggestions:

- **Assumption 1**: Bob and Alice operate in parallel. This means that both Alice and Bob can take orders, make the food, handle transactions and serve customers independently from each other. This also means that all work tasks which are relevant to the performance of the system are available for all servers. For example, Bob doesen't have to wait for Alice to cut onions in order to make a burger, he can just cut the onions himself and that will count towards him contributing to his active order.
- **Assumption 2**: The restaurant operates in a strict First-In-First-Out (FIFO) order. This means that a server can not take more than one order simultaneously and must complete a full order before going on the next.

From these assumptions, we can start to model the system's current state and how it operates. The system follows a M/M/c queue because it is fitting to use in scenarios where the calling population is theoretically infinite and could arrive at any given time.

It is also fitting because the M/M/c queue also assumes that customers arrive in a FIFO manner. This means that the process of customers arriving and the servers serving them follow a poison process where the arrivals and servings are exponentially distributed.

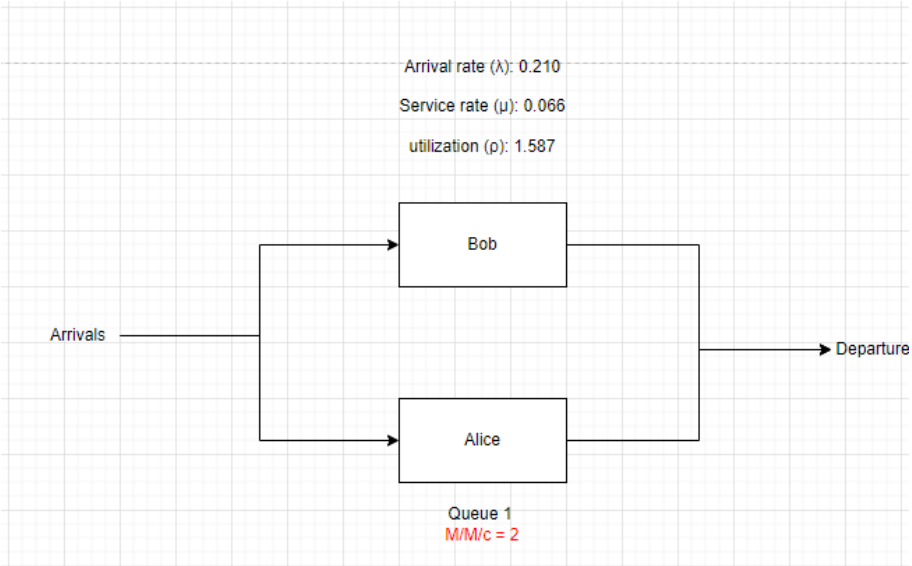

Figure 3.3: Queueing network model for Bob's and Alices restaurant.

We modeled this restaurant as a simple M/M/2 queue with the two servers Bob and Alice taking orders and working in parallel. In order to get the arrival rate we created a simple python script that would strip down the **A1 TakeAway-Dataset.csv** file to only include the *Order date* column and then added the minutes between each order date per day and then takes the mean in order to get the average interarrival time. This gave us an arrival rate $\lambda = 0.210$ customers per minute. The service rate was calculated in a similar fashion, we extracted the *Delivery Time Taken (mins)* column and calculated the average delivery time and then took the reciprocal. But since we have two parallel servers, we also have to multiply this number by 2 in order to get the total service rate which is $\mu = 0.066$ customers per minute. This comes from the assumptions that Bob and Alice work in parallel and are independent servers.

From this we can see that our utilization factor $\rho = 1.587$ which means that the system is severely over-utilized and overloaded. This also makes all performance metrics $L, 1, wQ, LQ$ and $P0$ negative which is a strong indication that the system is critically overloaded. This becomes even more apparent when plotting the order dates together with the delivery times:

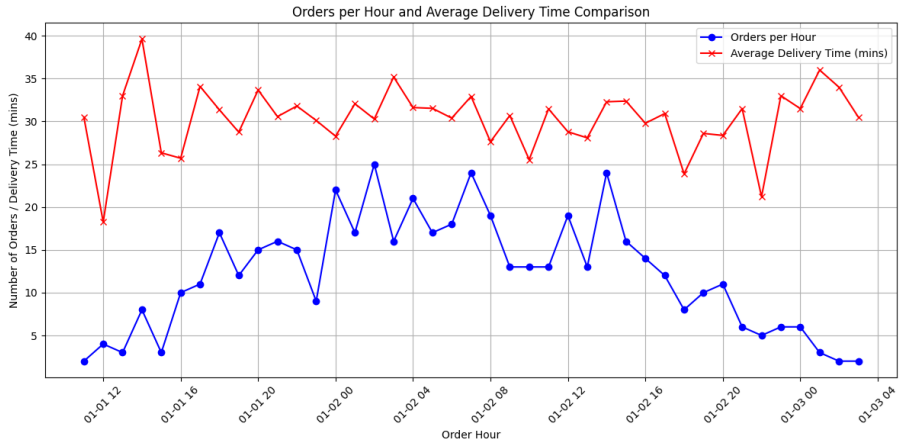

Figure 3.4: Plot showing the relation between orders and deliveries.

Here we can see that the amount of orders increase quite drastically the whole day of 2022-01-02 but the service times are for the most part at the same level on average which tells us that there is a bottleneck in Bob's and Alice's ability to serve the customers properly.

Our biggest recommendation for improving performance and reducing the bottleneck in the system would be to hire a new worker which would act as a third parallel server in the system. We illustrate this by adding a third worker called Kemal. The queueing network would in that case look like this:

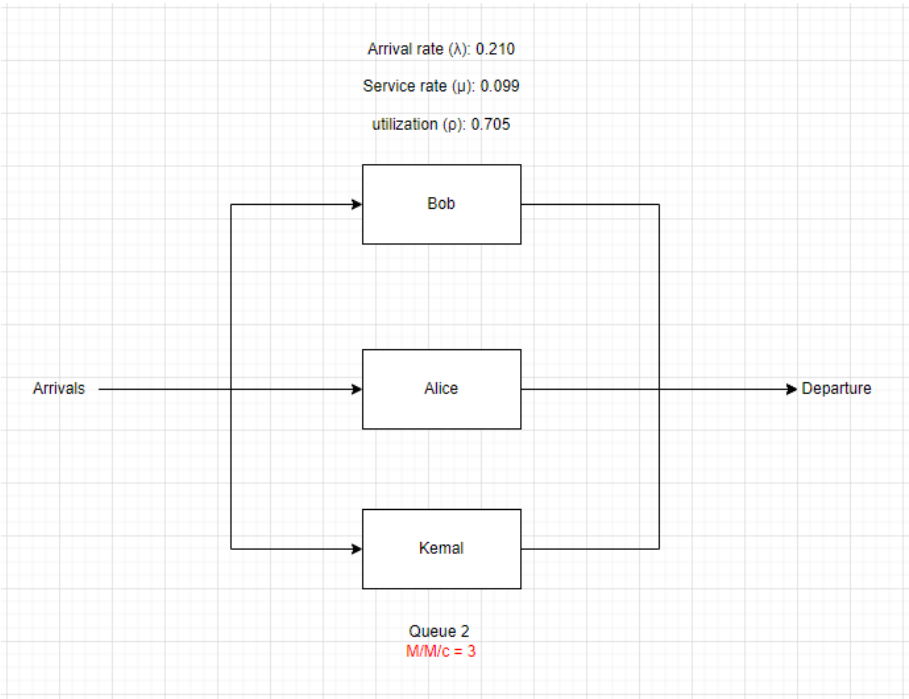

Figure 3.5: Improved version of Alice's and Bob's resturant.

From this improvement we can see that the arrival rate is the same, this makes sense because the rate at which the customers arrive is independent from the rate at which the restaurant can operate. The service rate however, has drastically improved because of the added worker because as we can now multiply the average service rate by a factor of 3

in order to get a total service rate of μ = 0.099 which drastically improves our utilization factor to ρ = 0.705 which now means that the system is not overloaded anymore and can now operate in a more stable manner. Because we now have utilization which falls in the range [0, 1] we can now draw conclusions about other performance metrics in the system:

$$L = c\rho + \frac{(c\rho)^{c+1}P_0}{c!(c(c!)(1-\rho)^2)} = c\rho + \frac{\rho P(L(\infty) \geq c)}{1-\rho} = 3.314 customers \tag{21}$$

$$w = \frac{L}{\lambda} = 15.818 minutes \tag{22}$$

$$w_Q = w - \frac{1}{\mu} = 5.717 minutes \tag{23}$$

$$L_Q = \lambda w_Q = \frac{(c\rho)^{c+1}P_0}{(c(c!)(1-\rho)^2)} = \frac{\rho P(L(\infty) \geq c)}{1-\rho} = 1.198 customers \tag{24}$$

$$P_0 = \left[\sum_{n=0}^{c-1} \frac{\left(\frac{\lambda}{\mu}\right)^n}{n!} + \left(\frac{\left(\frac{\lambda}{\mu}\right)^c}{c!}\left(\frac{c\mu}{c\mu-\lambda}\right)\right)\right]^{-1} \tag{25}$$

$$= \left[\sum_{n=0}^{c-1} \frac{(c\rho)^n}{n!} + \left(\frac{(c\rho)^c}{c!}\frac{1}{1-\rho}\right)\right]^{-1} = 0.093 \tag{26}$$

In conclusion, we found that adding another worker to the workforce is the most reasonable and effective way to make the restaurant operate under an acceptable level of stress given the information and data which was available to us.

Raw data available in A1 TakeAway-Dataset.csv, sorted_order_dates.csv and stripped_order_dates.csv