GEBZE TECHNICAL UNIVERSITY
COMPUTER ENGINEERING

SYSTEM PROGRAMMING
CSE 344 - 2020 SPRING

MIDTERM PROJECT REPORT

SILA BENGİSU YÜKSEKLİ
1801042877

➢ *Explanation of the way of work and the aim of the program*

User can compile by typing make on terminal. User should give command line arguments like “-N 3 -M 4 -T 2 -S 4 -L 5 -F filePath” , then it can be run by writing “./program -N 3 -M 4 -T 2 -S 4 -L 5 -F filePath” on terminal.
These arguments refer to the following:

- N: number of cooks
- M: number of students
- T: number of tables
- S: size of counter
- L: total number of rounds to eat again
- F: input file. It contains foods.

The contents of the file must consist of the letters P(soup), C(main course) and D(desert). The user can change the input file as desired, but it should contain exactly 3*l*m characters (l*m S, l*m D, l*m C).
There is also a K variable calculated as 2*l*m +1. It indicates the size of the kitchen.

This program simulates the student mess hall of a university. In this scenario, there is a supplier that delivers the foods to kitchen, there are cooks who take the foods from the kitchen and put them on the counter for students and students who eat the foods. There are three types of foods which are soup, desert and main course. The supplier delivers exactly l*m soups, deserts and main courses in an arbitrary order. Cooks take these foods and put them on the counter and students get foods, sit at a table and eat them.
Each actor was represented as a process. Hence, there are three source files which are program.c, cook.c and student.c .

➢ *Supplier Process*

The supplier acts like the producer in the producer-consumer example. It continues to work until it delivers up to 3*l*m plates. If there is no space to put food in the kitchen, it waits until the space is opened in the kitchen thanks to the emptyKitchen semaphore. When it delivers a total of l*m soups, deserts and main courses, it finishes its job.

➢ *Cook Process*

The cook acts both as a producer and as a consumer. The cook assumes the role of a producer in the relationship with the student and role of a consumer in the relationship with the supplier. Cook processes continue to work until the supplier has finished its work and there is no food left from the kitchen. If there is no more room to put the plates on the counter, they wait until the counter is empty thanks to the emptyCounter semaphore. The cooks bring the food in a certain order to prevent deadlock situation. Thanks to the index variable held in shared memory, cooks know which plate is left on the counter last, so they choose the next plate correctly even if other cooks are running one after another. For this purpose, cooks work one by one by using mutexbetweenCooks semaphore. Deadlock can occur if this lock is not provided. For example, if a cook makes a choice by looking at the instantaneous index variable, but another cook starts working before the previous cook process takes the plate according to that choice, the new cook process may choose the wrong plate. This can hinder three different foods from being on the counter at the same time.

Thanks to the soup_atKitchen, desert_atKitchen, mainCourse_atKitchen semaphores, if there is no soup, desert or main course in the kitchen at the moment, they wait until the foods are provided. When the three types of food are placed on the counter, the flag semaphore is increased, thus information is provided on how many of the triple food groups are present at that moment.

➢ *Student Process*

The student process acts like a consumer. Each of student process continues to work until total of L times. First of all, thanks to the flag semaphore, it checks whether three types of foods are available on the counter. If not, it waits until they are available. It then checks whether there is an empty table after taking a soup, dessert and main course from the counter. If there is no empty table, it waits for a table to be emptied thanks to the table semaphore.

While the student chooses a table to eat, the first empty table is selected and the student sits down on it. After it consumes food, it leaves the table, so it changes the status of the table as empty.

➢ *Communication between supplier process, cook processes and student processes*

Shared memory is used to provide communication between processes. Intercommunication is necessary, because there are many common sources such as tables, foods etc. For example, when a student sits at a table, other students need to know this. Because when that table is full, they should sit at another table, or if there is no other empty table, they should wait for the table which is full. Semaphores are used for this kind of waiting operations. The semaphores that are used and their purpose are listed below:

- **fullKitchen** : It shows how full the kitchen is. Thus, foods are supplied until the kitchen remains empty
- **emptyKitchen** : It shows how empty the kitchen is. If the kitchen has reached its maximum fullness, it allows the supplier to wait until the kitchen is empty.
- **fullCounter** : It shows how full the counter is. Thus, the foods are placed on the counter until the counter remains empty.
- **emptyCounter** : It shows how empty the counter is. If the counter has reached its maximum fullness, it allows the cooks to wait until the counter is empty.
- **soup_atKitchen** : It instantly shows how much soup is in the kitchen.
- **desert_atKitchen** : It instantly shows how much desert is in the kitchen.
- **mainCourse_atKitchen** : It instantly shows how much main course is in the kitchen.
- **soup_atCounter** : It instantly shows how much soup is on the counter.
- **desert_atCounter** : It instantly shows how much desert is on the counter
- **mainCourse_atCounter** : It instantly shows how much main course is on the counter
- **mutexBetweenCooks** : It allows the cooks to work one by one. So when a cook chooses a plate and puts it on the counter, other cooks are waiting for this process to end. The reason for such a locking mechanism is to prevent the cooks from making wrong decisions. If another cook starts to work while choosing the plate to take, the cook may choose the plate incorrectly. Such a choice was made to prevent deadlock.
- **table** : It shows instantly how many empty table there is.
- **flag** : If all three types of food are available at the same time, the student must take the food. If not, they should wait until there are three types of food. This variable is kept to see if three types of food are on the counter at the same time.

Also, some integer pointers are kept in shared memory for communication.The variables that are used and their purpose are listed below:

- **numberOfStudents** : It states how many students are waiting to get food on the counter.
- **soup_value** : It shows how much soup the supplier delivers.
- **desert_value** : It shows how much desert the supplier delivers.
- **mainCourse_value** : It shows how much main course the supplier delivers.
- **s_counter** : It shows how much soup the cook put on the counter.
- **d_counter** : It shows how much desert the cook put on the counter.
- **m_counter** : It shows how much main course the cook put on the counter.
- **_index** : This variable is used in the food selection algorithm of the cook process. Every cook chooses to put soup, desert and main course in order. Zero represents the soup, one represents desert and two represents main course. After a cook process places food on the counter according to the current index number, the index number is increased by one. Since this variable is kept in shared memory, whichever cook process works, it chooses the next food correctly. When the index value becomes three, the value of the index is set to zero again and the selection of food is continued. Thus, three type of foods can be found at the counter at the same time.

➢ *Problems Encountered*

The problem of getting meals all at once

In order for a student to take food, all three types of food must be available on the counter. The student cannot take a soup and wait for the desert to come. It should either take it at the same time or not at all. Therefore, this type of solution cannot be used:

```
sem_wait(soup_atCounter);
sem_wait(desert_atCounter);
sem_wait(mainCourse_atCounter);
...
```

For this purpose, a variable named flag was kept, which indicates whether all of them are present at the same time. Since the cooks place the plates in order according to the index number, when the index number is three, it is understood that there is one group of three foods at the same time. When the index number becomes three, the flag variable is increased one, and this information is given to the student. If three meals are not available at the same time, as can be seen in the following code snippet, the student can wait at once as the flag variable will be zero thanks to sem_wait(flag) line.

```
while(l != L)  {
        …
        *numberOfStudents = *numberOfStudents + 1;
        sem_wait(flag);
        sem_wait(soup_atCounter);
        sem_wait(desert_atCounter);
        sem_wait(mainCourse_atCounter);
        sem_wait(fullCounter);
        sem_wait(fullCounter);
        sem_wait(fullCounter);
        ...
```

Deadlock situation while cooks are working at the same time

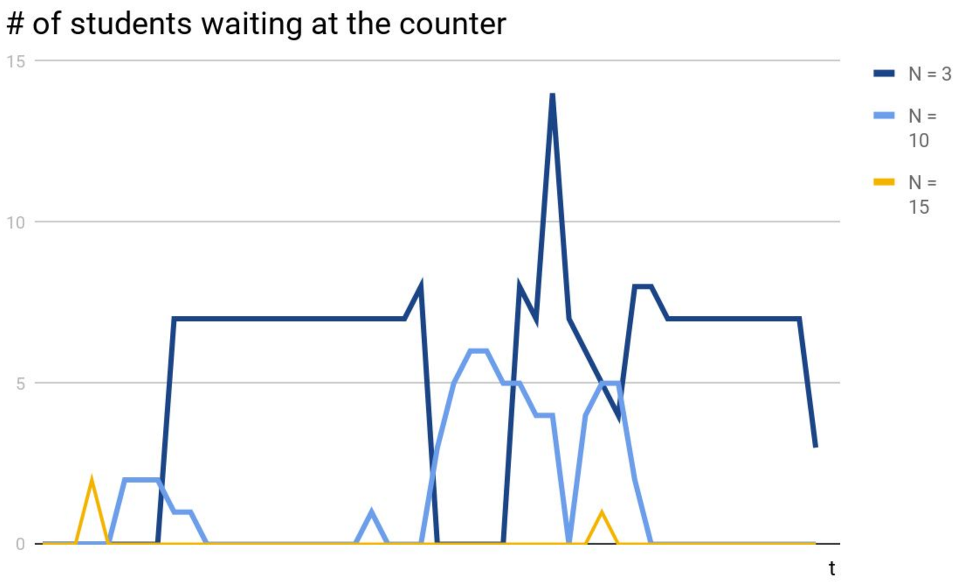
The index variable, which is kept in shared memory, specifies the order in which the plate will be placed, as in the piece of code shown below :

```
if (*_index == 0)  {
        ...
        takePlateFromKitchen(soup_atKitchen);
        supplyPlatetoCounter(soup_atCounter);
        *s_counter = *s_counter + 1;
        *_index = *_index + 1;
        ...
}
else if (*_index == 1)  {
        ...
        takePlateFromKitchen(desert_atKitchen);
        supplyPlatetoCounter(desert_atCounter);
        *d_counter = *d_counter + 1;
        *_index = *_index + 1;
        ...
}
else {
        ...
        takePlateFromKitchen(mainCourse_atKitchen);
        supplyPlatetoCounter(mainCourse_atCounter);
        *m_counter = *m_counter + 1;
        *_index = *_index + 1;
        ...
}
```

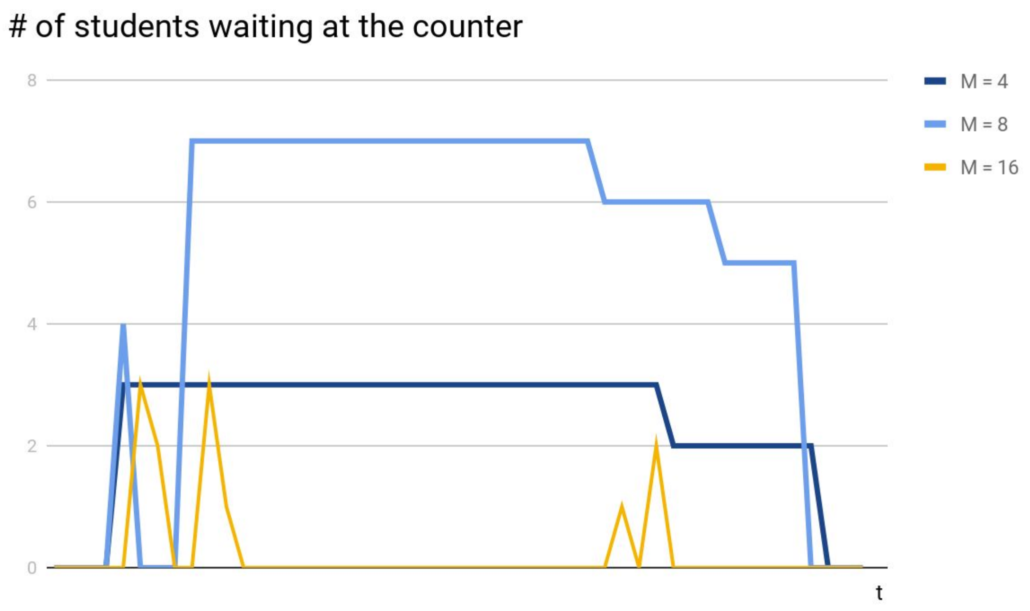
Let's say that while the first cook is working, let the index value be one, and therefore it decide to put the soup by entering the if block. Let's say the cook take the plate from the kitchen, put it on the counter and there is a context switch happened. In this case, since the index variable cannot be increased, the next cook will decide to put soup. If the same situation is repeated in this way, it may cause deadlock for small counter sizes. In order to overcome this, the cook processes are protected amongst themselves with the help of a mutex and they work one by one.

➢ *PLOTS*

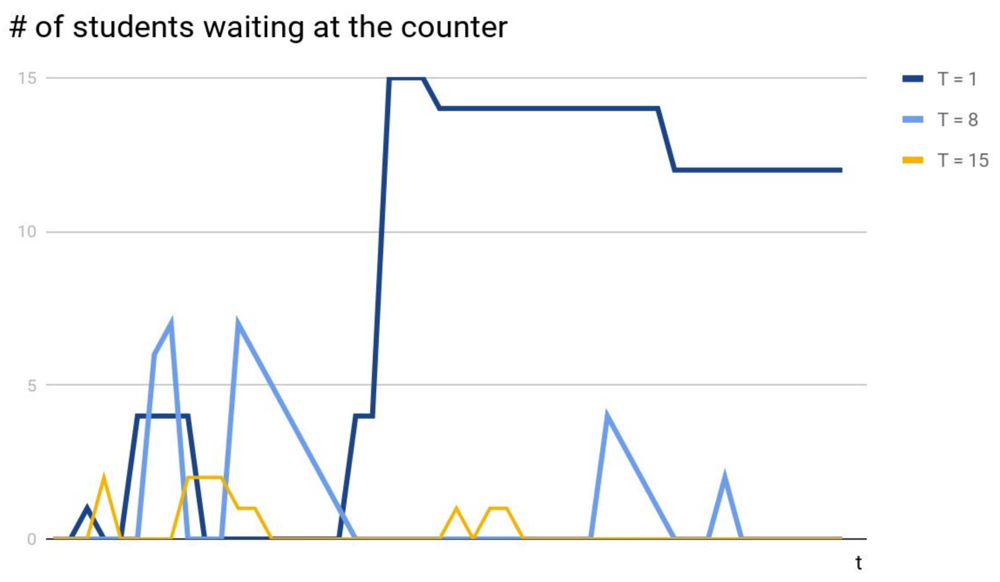
If the value of N changes (M = 16, T = 4, S = 4, L = 3)



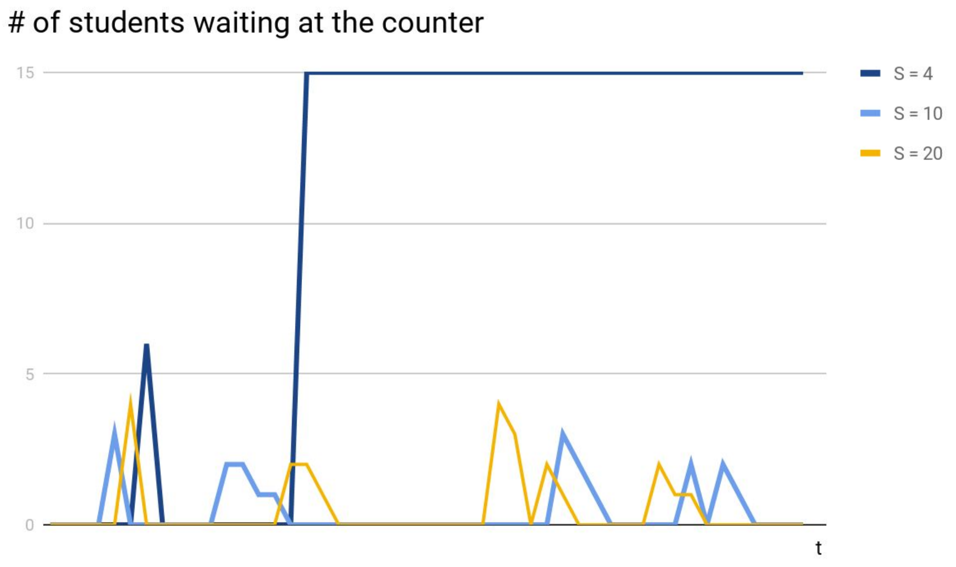
If the value of M changes (N = 3, T = 3, S = 20, L = 3)



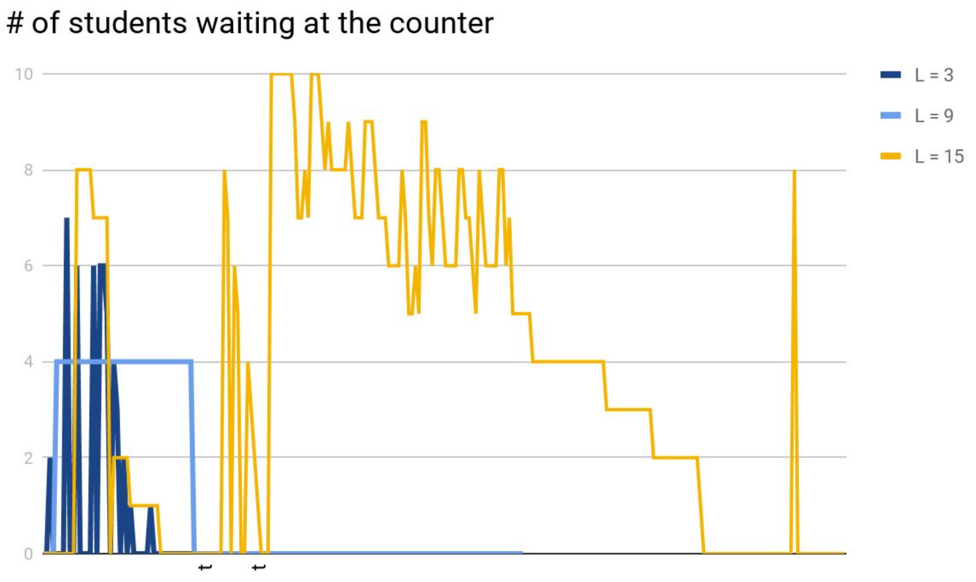
If the value of T changes (N = 3, M = 16, S = 4, L = 3)



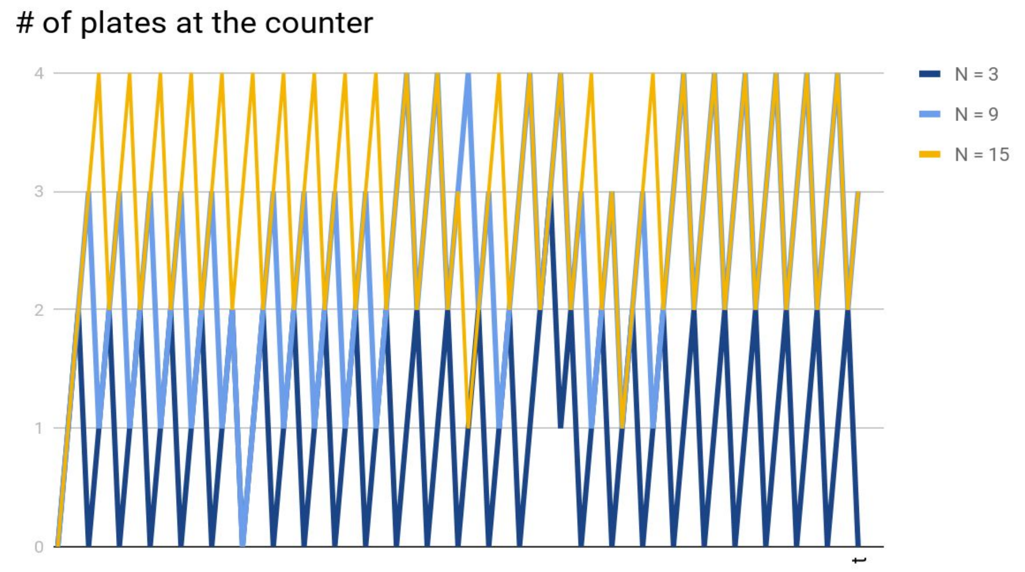
If the value of S changes (N = 4, M = 16, T = 5, L = 3)



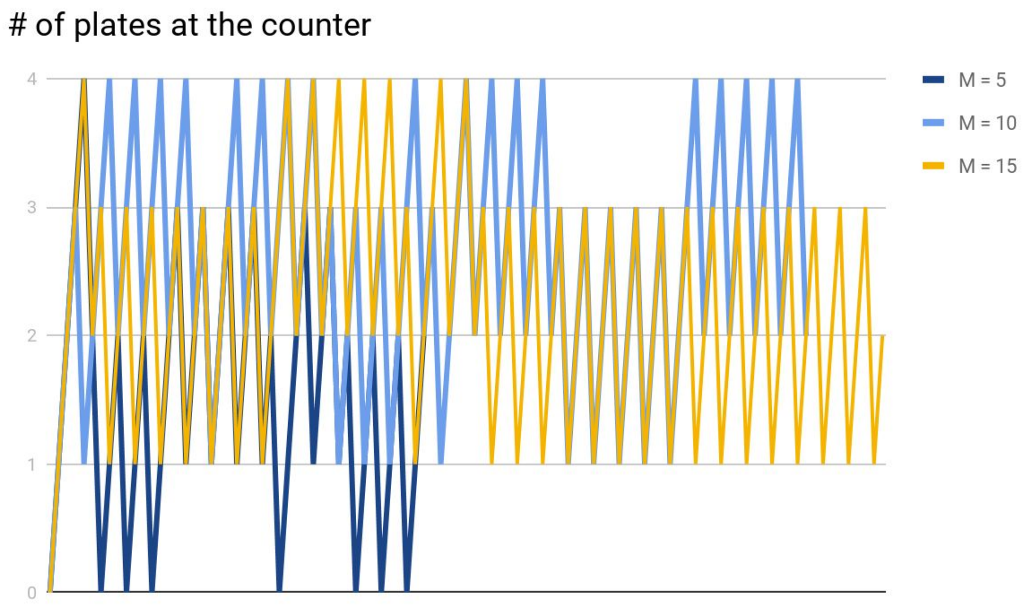
If the value of L changes (N = 3, M = 16, T = 5, S = 20)



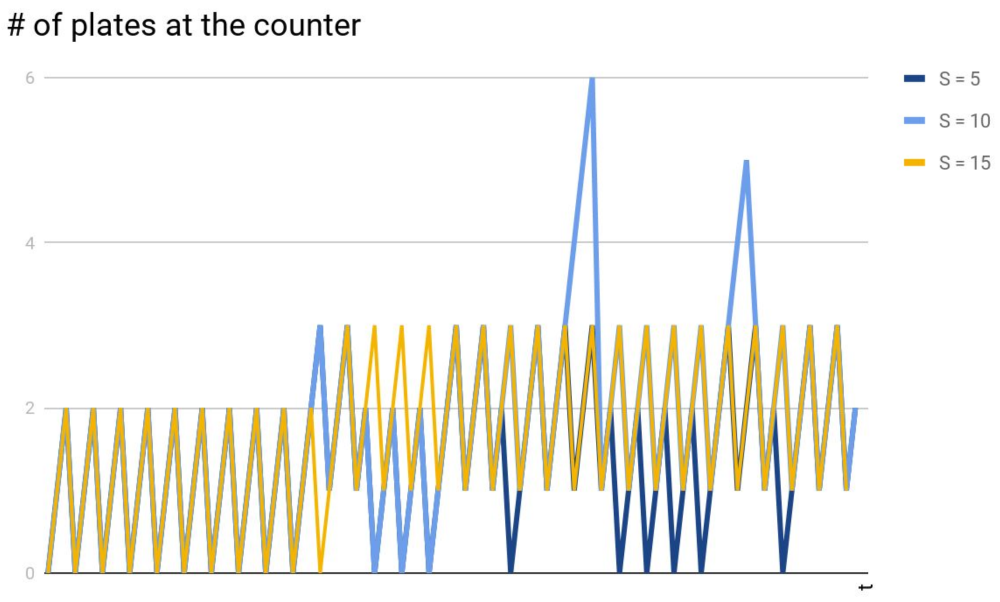
If the value of N changes (M = 16, T = 3, S = 5, L = 3)



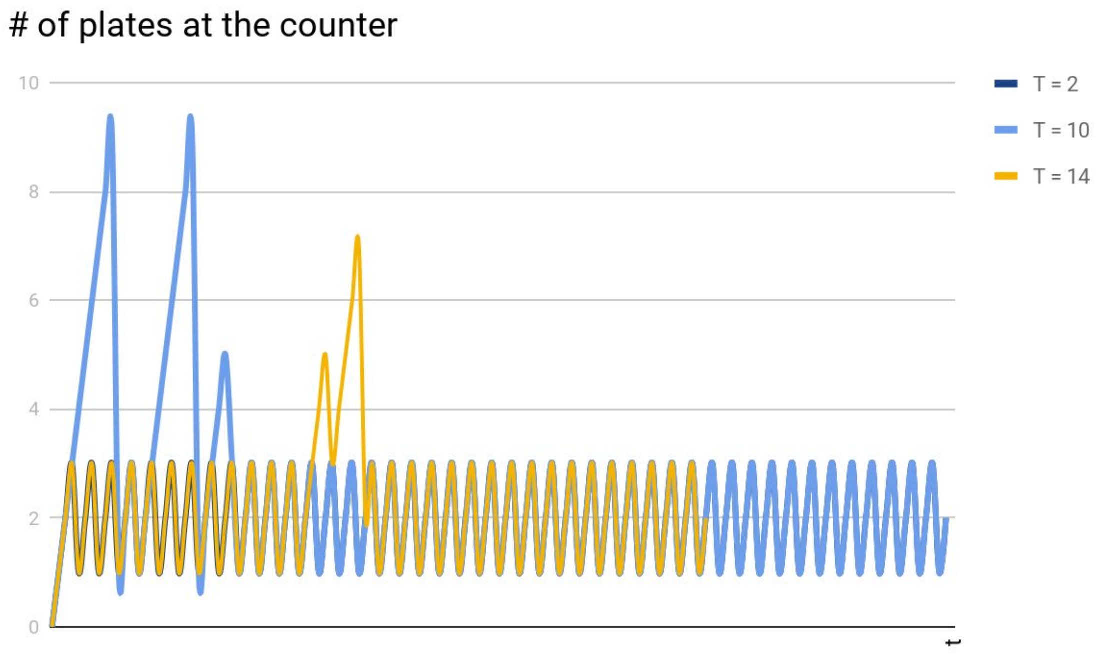
If the value of M changes (N = 4, T = 3, S = 5, L = 3)



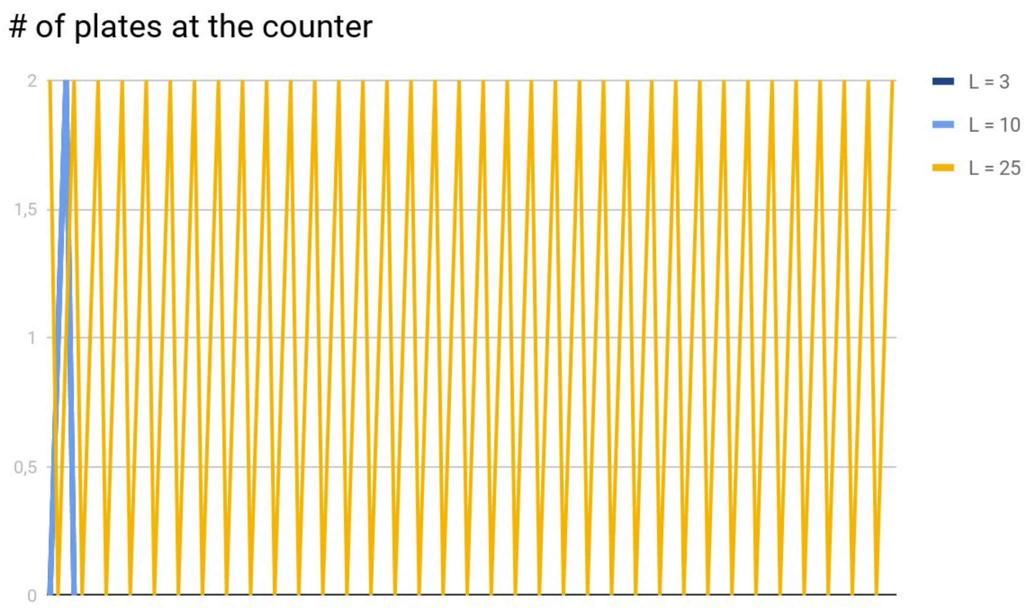
If the value of S changes (N = 4, M = 10, T = 3, L = 3)



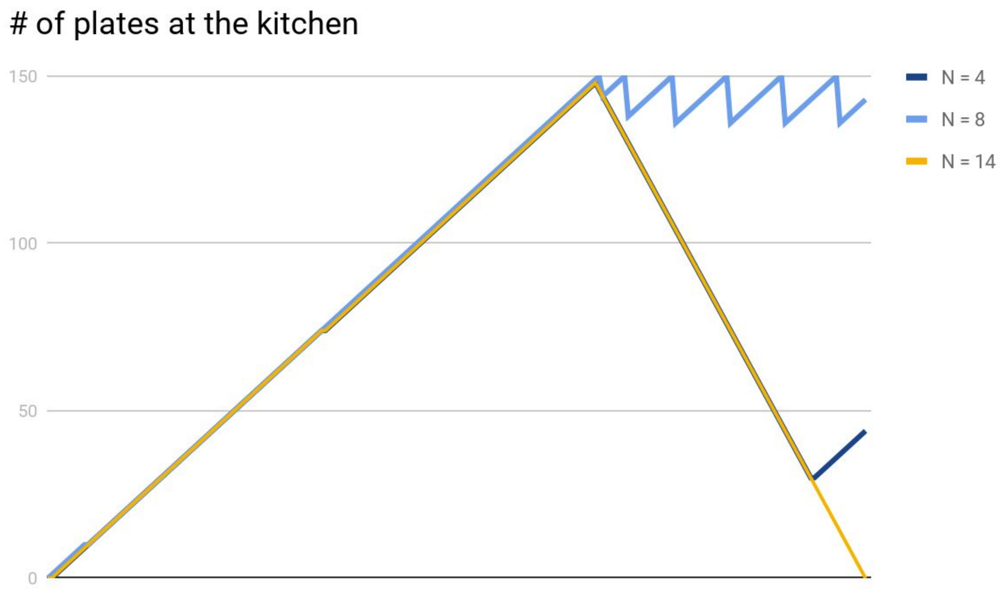
If the value of T changes (N = 4, M = 10, S = 15, L = 3)



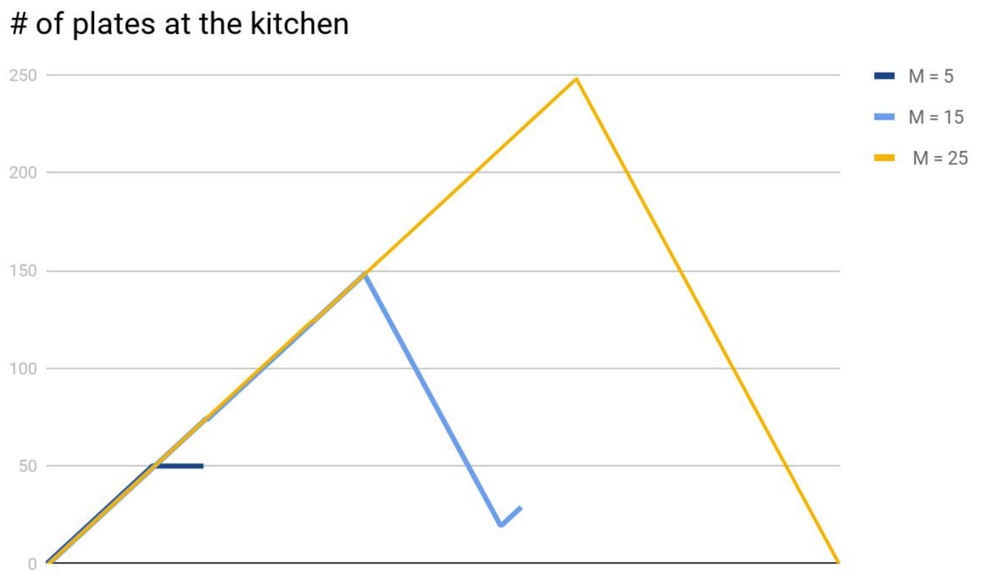
If the value of L changes (N = 4, M = 15, T = 14, S = 15)



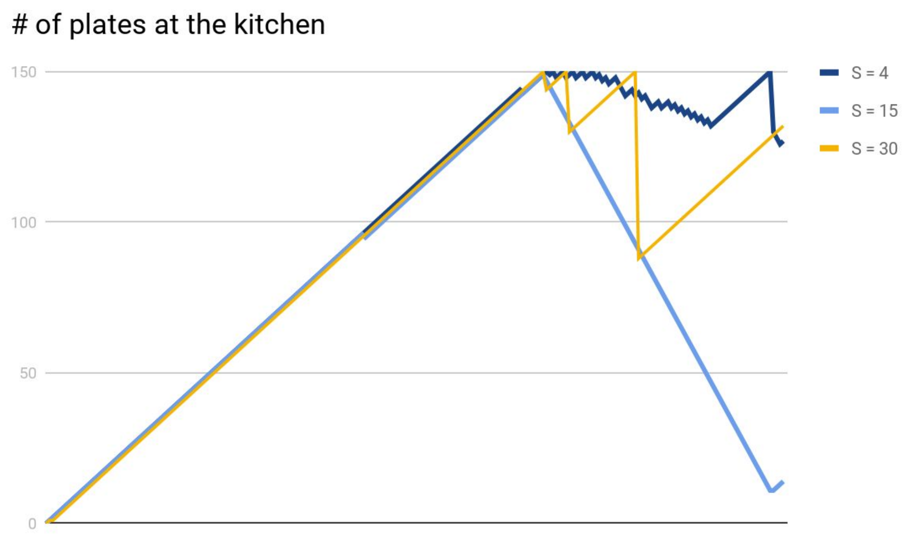
If the value of N changes (M = 15, T = 3, S = 15, L = 5)



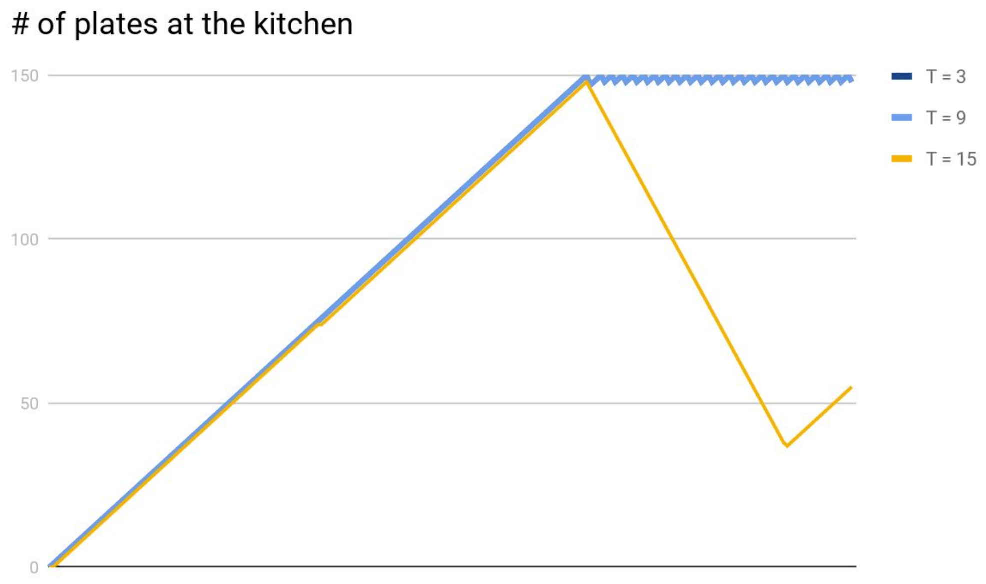
If the value of M changes (N = 4, T = 14, S = 15, L = 5)



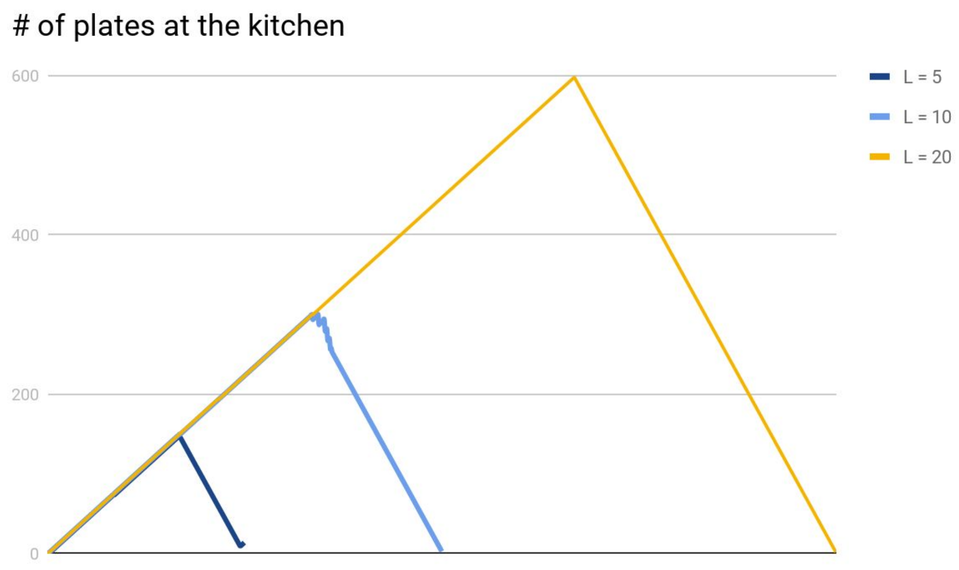
If the value of S changes (N = 4, M = 15, T = 3, L = 15)



If the value of T changes (N = 4, M = 15, S = 4, L = 15)



If the value of L changes (N = 4, M = 15, T = 14, S = 15)



➢ *Handling of Zombie Processes*

To prevent processes from being zombie, there are several methods. In this program wait method is used. As seen in the code snippet below, wait(NULL) will block parent process until any of its children has finished and it reaps the exit status of the child. Thus, such a situation is prevented.

```
while(wait(NULL) > 0);
```

➢ *Case of CTRL-C*

There is a handler for each separate process. All processes must exit gracefully, if program receives this signal. I keep file descriptors, semaphores and pointers as global variables so that if an SIGINT signal comes, handlers can free resources. Also, main program collects the exit status of the children before termination in the handler to prevent processes from being zombie.