Assignment-4

Vinta Reethu - ES18BTECH11028

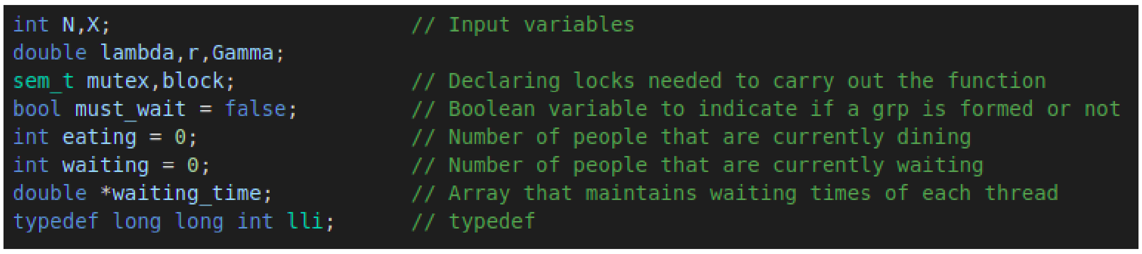
March 29, 2021

Question

Korean Restaurant problem

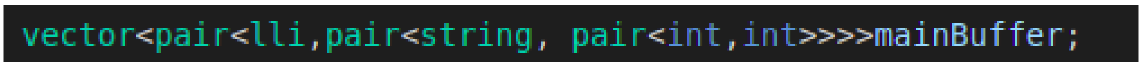
Implementation details

- These are the global declarations. These variables are to store values like number of customers, number of seats at the table, Semaphores, waiting time array etc



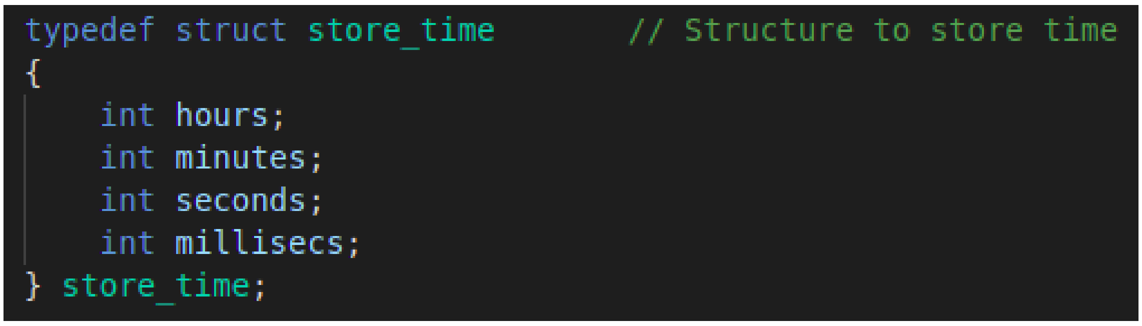
```
int N,X;                          // Input variables
double lambda,r,Gamma;
sem_t mutex,block;                // Declaring locks needed to carry out the function
bool must_wait = false;           // Boolean variable to indicate if a grp is formed or not
int eating = 0;                   // Number of people that are currently dining
int waiting = 0;                  // Number of people that are currently waiting
double *waiting_time;             // Array that maintains waiting times of each thread
typedef long long int lli;        // typedef
```

- A vector named **mainBuffer** is declared in which we dump time elapsed from Jan 1970 in microseconds, timestamp, thread number, an integer to identify it is access request, access given or exiting statement.

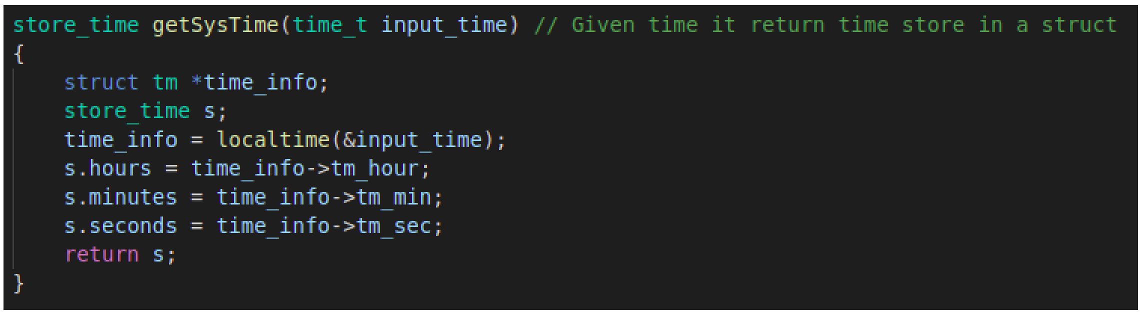


```
vector<pair<lli,pair<string, pair<int,int>>>>mainBuffer;
```

- A struct is defined so that we can save the current time in it. **getSysTime()** is a function that return this struct when we pass a time_t variable into it.



```
typedef struct store_time        // Structure to store time
{
    int hours;
    int minutes;
    int seconds;
    int millisecs;
} store_time;
```



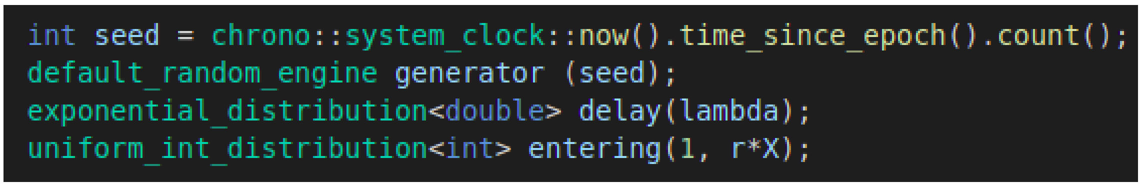
```
store_time getSysTime(time_t input_time) // Given time it return time store in a struct
{
    struct tm *time_info;
    store_time s;
    time_info = localtime(&input_time);
    s.hours = time_info->tm_hour;
    s.minutes = time_info->tm_min;
    s.seconds = time_info->tm_sec;
    return s;
}
```

Algorithm Design

- The algorithm is same as the solution given for the theory assignment.
- We note the time when a particular thread enters the **Restaurant** function and name it as **accessRequest**.
- After this we check if group is formed, if formed we wait on block semaphore else we go inside the restaurant and eat.
- Before the simulation of eating, we record that time as **accessGiven** time.
- While exiting if the entire group left we signal the block so that waiting cutomers can enter inside and eat.
- Inside the mutex lock, present near exiting part, we push accessGiven statements into the buffer. We also update the thread waiting time here.
- We note the time just after decrementing the eating value as **exiting** time and push into the buffer.

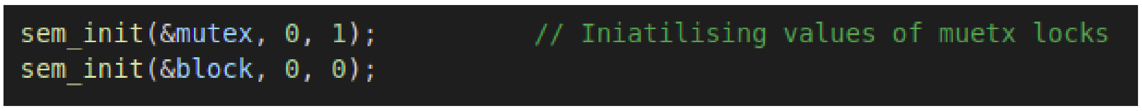
System Model

- We use **fopen** to open the input.txt file and read the inputs.
- We use distributions defined in random library to get exponential, uniform distributions. We use chrono to set the seed.



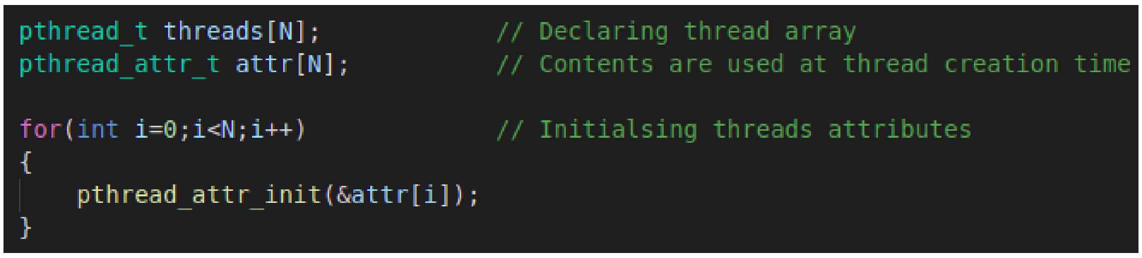
```
int seed = chrono::system_clock::now().time_since_epoch().count();
default_random_engine generator (seed);
exponential_distribution<double> delay(lambda);
uniform_int_distribution<int> entering(1, r*X);
```

- We initialise the semaphores using **sem_init**. We set mutex to 1 and set block to 0.



```
sem_init(&mutex, 0, 1);          // Iniatilising values of muetx locks
sem_init(&block, 0, 0);
```

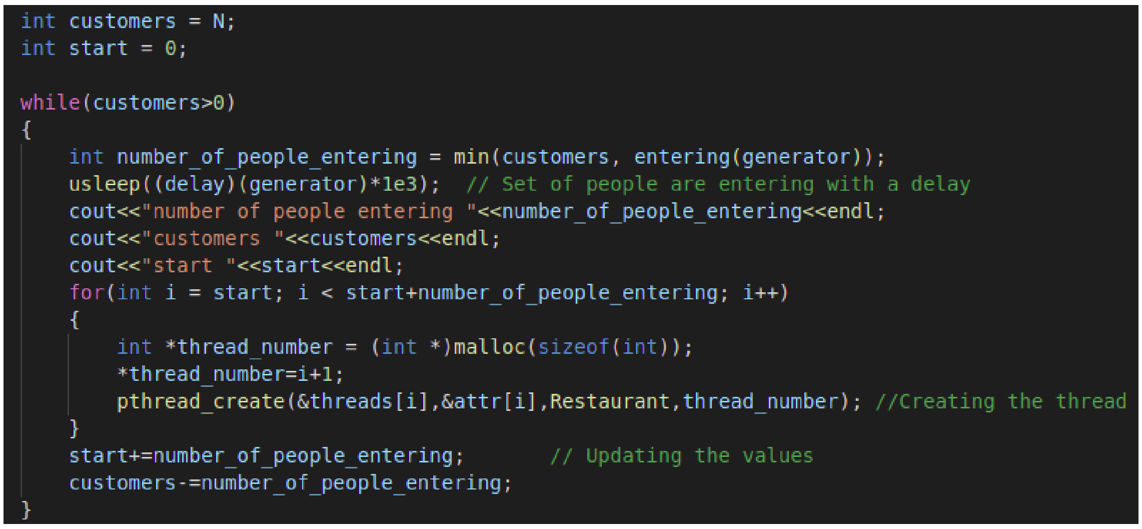
- Here we create N threads and initialise them.



```
pthread_t threads[N];              // Declaring thread array
pthread_attr_t attr[N];            // Contents are used at thread creation time

for(int i=0;i<N;i++)               // Initialsing threads attributes
{
    pthread_attr_init(&attr[i]);
}
```

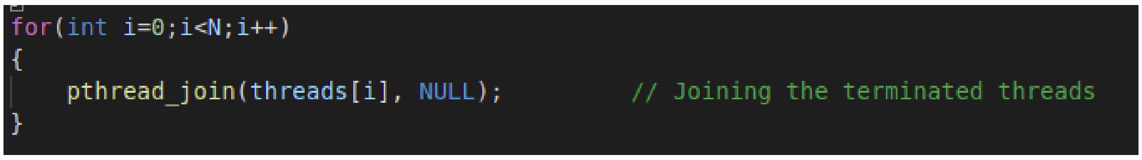
- We run a loop till N customers dine in the restaurant. We obtain the number of people to enter restaurant **s** using uniform distribution and make them sleep for some time. This time is generated by the exponential distribution.

- Here we are using main thread to create worker threads and pass them inside the function. We use a variable named **start** as the first index from where we access **number_of_people_entering** threads and pass them inside the function. Every time we update it by number_of_people_entering. We also update customer variable by number_of_people_entering.



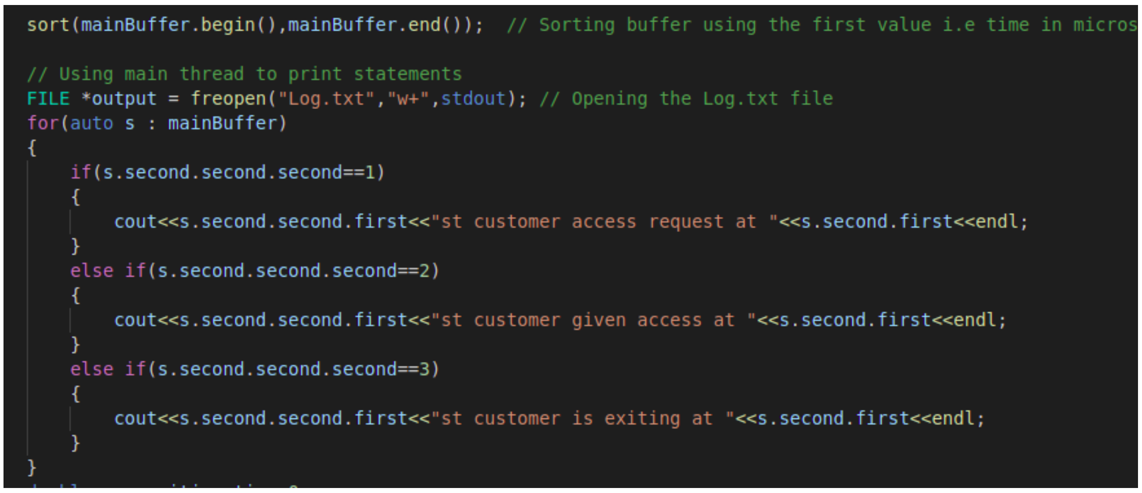
```
int customers = N;
int start = 0;

while(customers>0)
{
    int number_of_people_entering = min(customers, entering(generator));
    usleep((delay)(generator)*1e3);  // Set of people are entering with a delay
    cout<<"number of people entering "<<number_of_people_entering<<endl;
    cout<<"customers "<<customers<<endl;
    cout<<"start "<<start<<endl;
    for(int i = start; i < start+number_of_people_entering; i++)
    {
        int *thread_number = (int *)malloc(sizeof(int));
        *thread_number=i+1;
        pthread_create(&threads[i],&attr[i],Restaurant,thread_number); //Creating the thread
    }
    start+=number_of_people_entering;       // Updating the values
    customers-=number_of_people_entering;
}
```

- Once all the N customers complete eating main threads join these N threads.



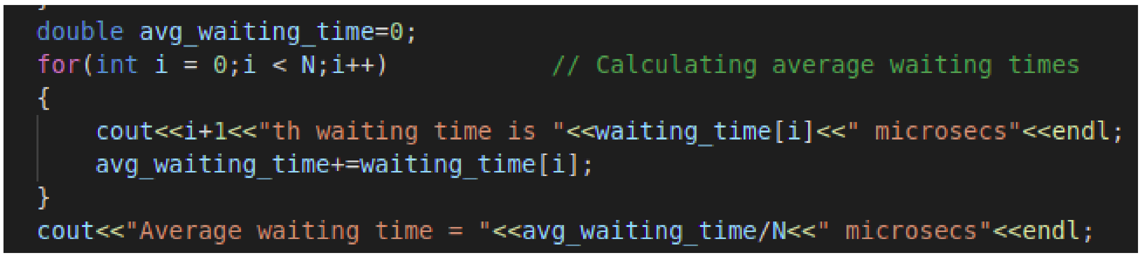
```
for(int i=0;i<N;i++)
{
    pthread_join(threads[i], NULL);         // Joining the terminated threads
}
```

- We stored all the timestamps in mainBuffer. The first value in this is time elapsed from Jan 1970 in microseconds we use this to sort the times. After sorting we print the respective threads.



```
sort(mainBuffer.begin(),mainBuffer.end());  // Sorting buffer using the first value i.e time in micros

// Using main thread to print statements
FILE *output = freopen("Log.txt","w+",stdout); // Opening the Log.txt file
for(auto s : mainBuffer)
{
    if(s.second.second.second==1)
    {
        cout<<s.second.second.first<<"st customer access request at "<<s.second.first<<endl;
    }
    else if(s.second.second.second==2)
    {
        cout<<s.second.second.first<<"st customer given access at "<<s.second.first<<endl;
    }
    else if(s.second.second.second==3)
    {
        cout<<s.second.second.first<<"st customer is exiting at "<<s.second.first<<endl;
    }
}
```

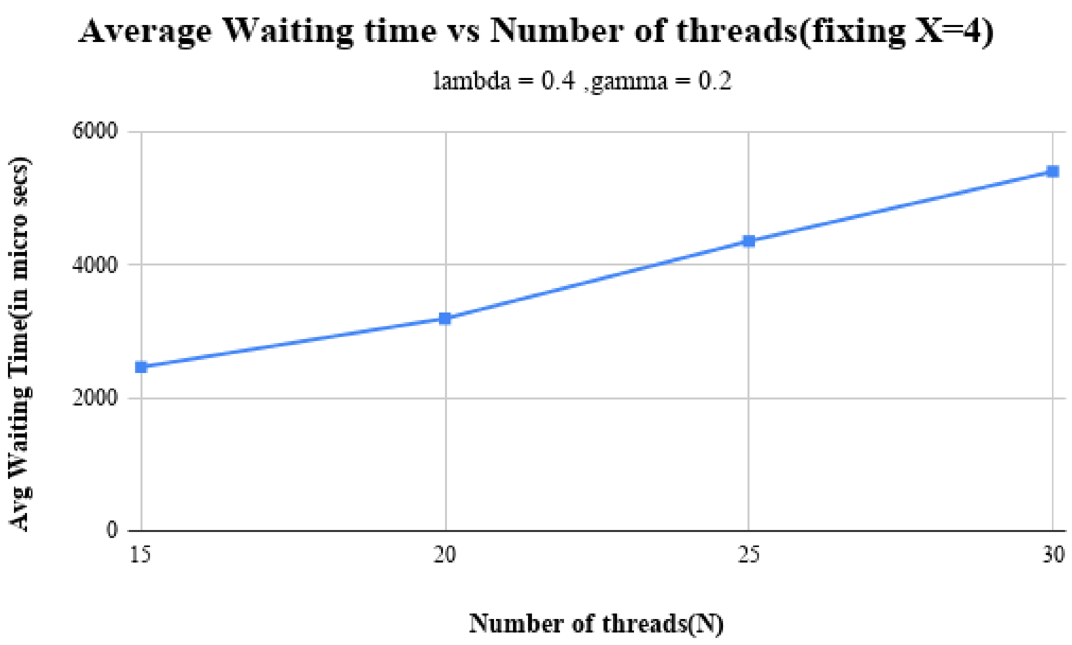
- Calculating the average waiting time and printing it.



```
double avg_waiting_time=0;
for(int i = 0;i < N;i++)           // Calculating average waiting times
{
    cout<<i+1<<"th waiting time is "<<waiting_time[i]<<" microsecs"<<endl;
    avg_waiting_time+=waiting_time[i];
}
cout<<"Average waiting time = "<<avg_waiting_time/N<<" microsecs"<<endl;
```

Graph analysis

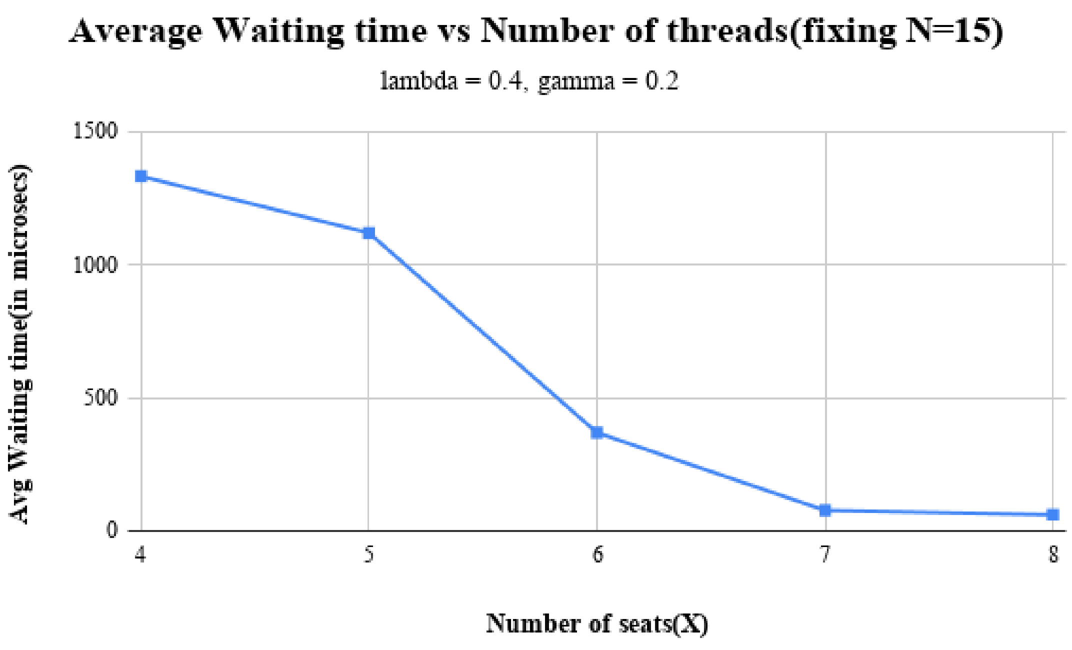
Average waiting time graph drawn by taking X constant value.



Parameters : $\lambda = 0.4$, $\Gamma = 0.2$, $r = 0.7$

- The value I have taken is lambda > gamma, therefore average waiting time for lambda will be less than average waiting time for gamma. Which means that people will take more time to complete eating. But people will be arriving to the restaurant in very small intervals.
- There fore the restaurant will be busy most of the time. Hence the customers have to wait for more time. Therefore as N increase the waiting time should increase.
- From the above graph we can see that as number of threads increase, the waiting time is increasing.
- While plotting these graphs, one could get a low or high waiting value, as there are multiple parameters that vary every time we run. So I have taken average of 5 runs and plotted this graph.

Average waiting time graph drawn by taking N constant value.



Parameters : $\lambda = 0.4$, $\Gamma = 0.2$, $r = 0.7$

- As the value of X increases more number of people can get a seat in the restaurant. Therefore the waiting time should decrease.
- From the above graph we can see that there is a decrease in average waiting times as the X value is increasing.
- While plotting these graphs, one could get a low or high waiting value, as there are multiple parameters that vary every time we run. So I have taken average of 5 runs and plotted this graph.