The Korean Restaurant Problem

Ananya Mantravadi
CS19B1004

CODE DESIGN & IMPLEMENTATION

These are the globally declared variables, so that the threads can access simultaneously. must_wait indicates if there is a group formed currently. exec is a signal for a thread to start eating once it is given access to the table.

```
int eating = 0;
int waiting = 0;
sem_t mutex, block;
bool must_wait = false, exec = false;
int n, x;
double lambda, r, lambda2, waiting_time = 0;
exponential_distribution<double> ex_di_2;
default_random_engine gen, gen1, gen2;
```

In the main function, we initialize mutex and block semaphores. We generate a set of s customer threads (chosen uniformly between 1 to r*x) entering the thread function with an exponential delay, parameter ƛ. The number of threads will be equal to the number of customers in the restaurant, n, and x is the number of seats at the table. Once the threads finish their execution, they are joined to terminate/

```
sem_init(&mutex, 0, 1);
sem_init(&block, 0, 0);

thread threads[n];
uniform_int_distribution<int> uni(1, r*x);

for(int i=0; i<n;){
        usleep(ex_di_1(gen)*1000);
        int k = uni(gen1);
        int s = min(k, max(n-i, 1));
        for (int j=0; j<s; j++)
        {
            threads[i] = thread(test, i);
            i++;
            if(i==n) break;
        }
    }


for(int i=0;i<n;i++){
    threads[i].join();
}
```

/* thread function */

```
sem_wait(&mutex);
auto reqEnterTime = chrono::system_clock::now();
  time_t t_enter = chrono::system_clock::to_time_t (reqEnterTime);
  struct tm* time = localtime(&t_enter);
  sprintf(timestring, "%.2d:%.2d",time->tm_min, time->tm_sec);
  temp = to_string(id+1) + suffix + " customer access request at " +
  (string)timestring + "\n";
  output << temp;
```

Once the thread enters the function, we update the log with its entry. Depending on the condition of the table, if the group is currently formed on the arrival of the thread, it is blocked.

```
if(must_wait==true||eating+1>x){

  waiting++;
  must_wait = true;
  sem_post(&mutex);
  sem_wait(&block);
```

//give access after releasing block, update log, and add difference in time to waiting time

```
exec = true;
must_wait = (eating==x && waiting>0);


else{
  eating++;   //give access, update log, and add difference in time to waiting time

 exec = true;
 must_wait = (eating==x && waiting>0);
 sem_post(&mutex);
```

Once the thread is given access, the time each thread eats parallelly is exponentially distributed with an average of Γ, after which it leaves the dining table. Once this is done, the updation of number of eating threads and log is done atomically to ensure there are no race conditions.

Finally, we check if the the table is empty, i.e, if eating is zero. If yes, we let the next set of customers take their seats at the table simlutaneously, atomically. This set will the minimum of the number of currently waiting customers and the maximum capacity of

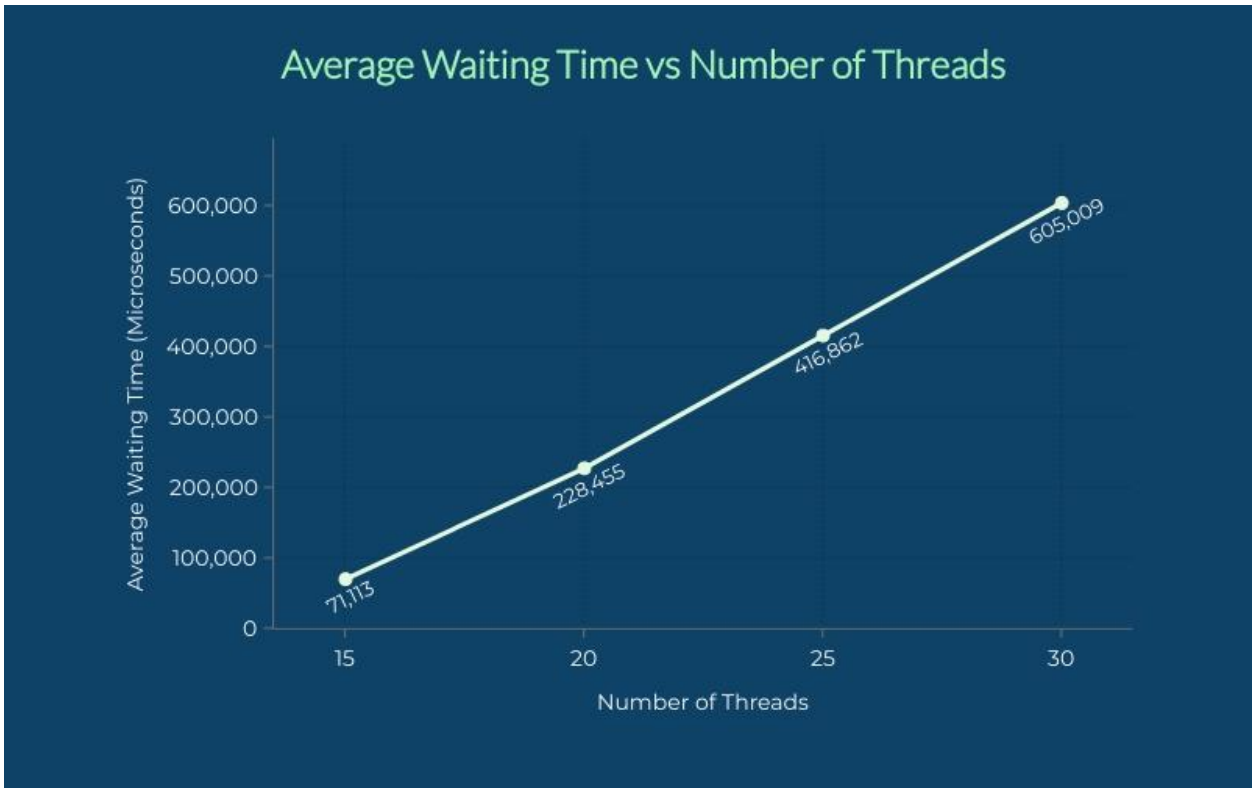
the table. To give this set of threads access, if release the block semaphore those many times.

```
if(eating==0){
  sem_wait(&mutex);
  int k = min(x, waiting);
  waiting -= k;
  eating += k;
  must_wait = (eating==x && waiting>0);
  for(int i=0; i<k; i++)
    sem_post(&block);
  sem_post(&mutex);
}
```

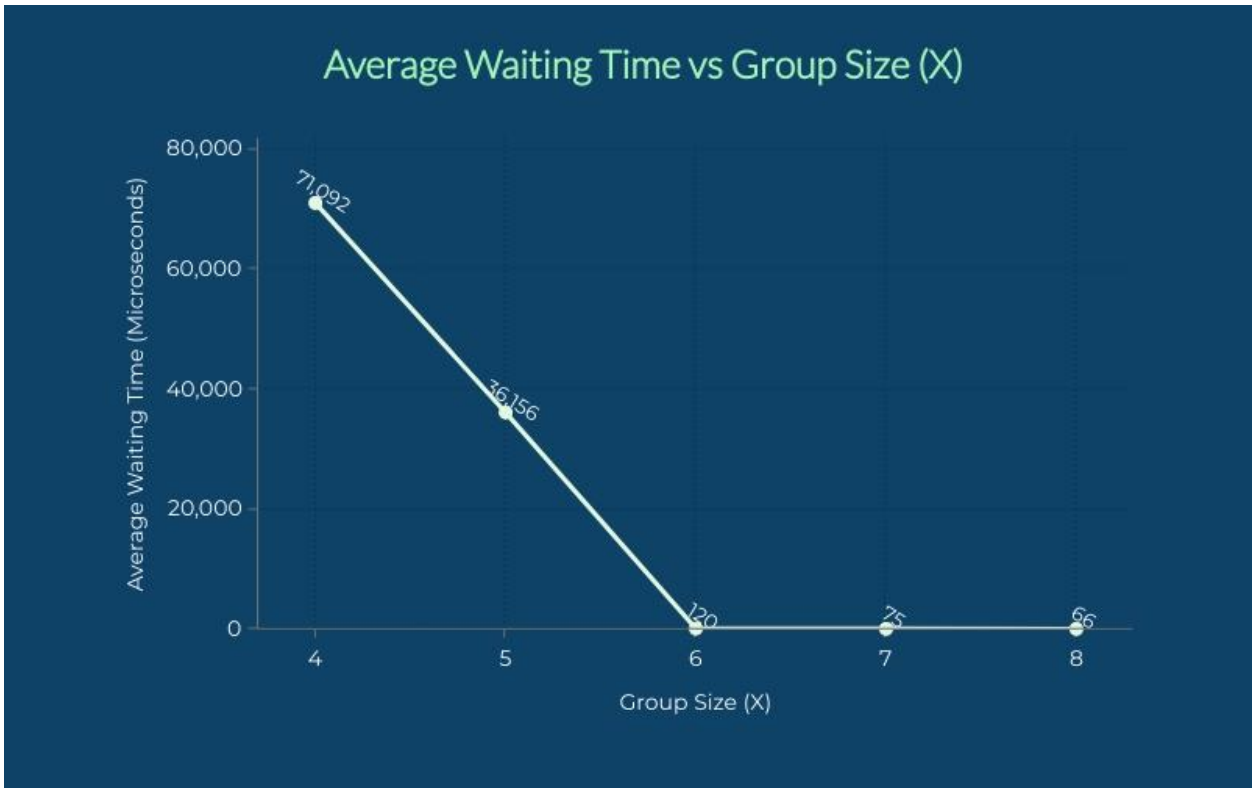
The output log and the average waiting time is generated in “log.txt” and “Stats.txt” respectively.

For the following two graphs, the values of λ and Γ chosen are 100, 200; r is 0.75.

GRAPH 1: Average Waiting Time vs Number of Threads



GRAPH 2: Average Waiting Time vs Group Size (X)



CONCLUSION

We observe that the average waiting time increases with the number of threads - more and more threads request access and end up waiting for longer times before given access due to the fixed limited seats at the table. But if we keep the number of threads constant and vary the number of seats, the average waiting time goes down with increase in the table size, as more threads have the seats available to dine now.