Vinta Reethu
Operating systems - 3
Indian Institute of Technology Hyderabad
Assignment-2

Assignment-3

Vinta Reethu - ES18BTECH11028

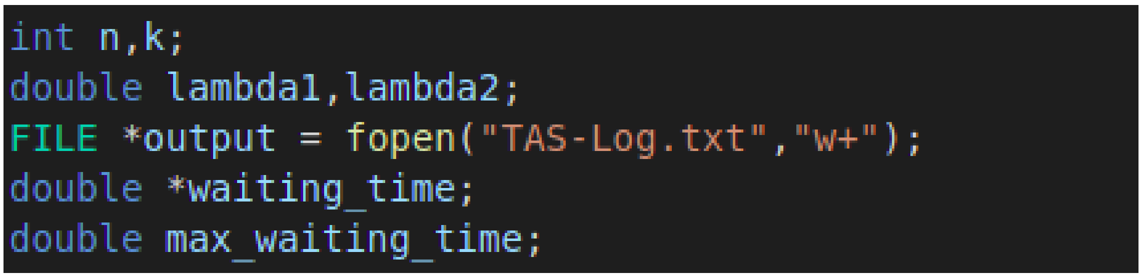
March 17, 2021

Question

Implementing TAS, CAS and Bounded Waiting with CAS mutual exclusion (ME) algorithms

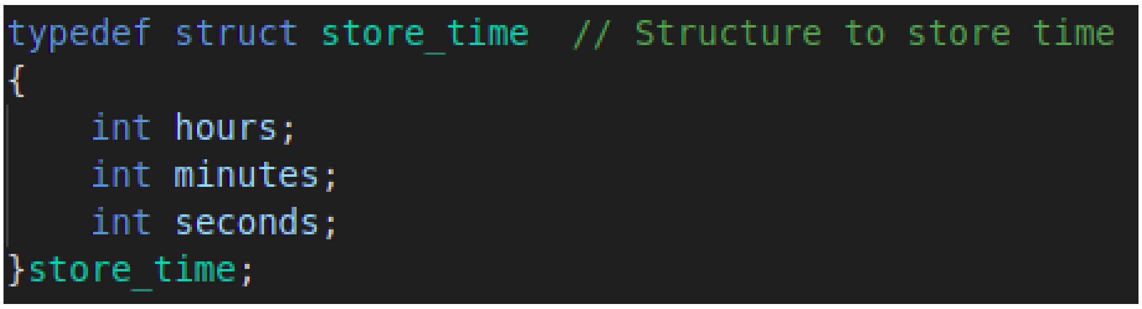
Implementation

- These are the global declarations present in three files. These variables are to store parameters like number of threads, number of instances a thread has to enter CS, lambda1, lambda2, waiting time array, maximum waiting time. We also open output file globally to write into it.



```
int n,k;
double lambda1,lambda2;
FILE *output = fopen("TAS-Log.txt","w+");
double *waiting_time;
double max_waiting_time;
```

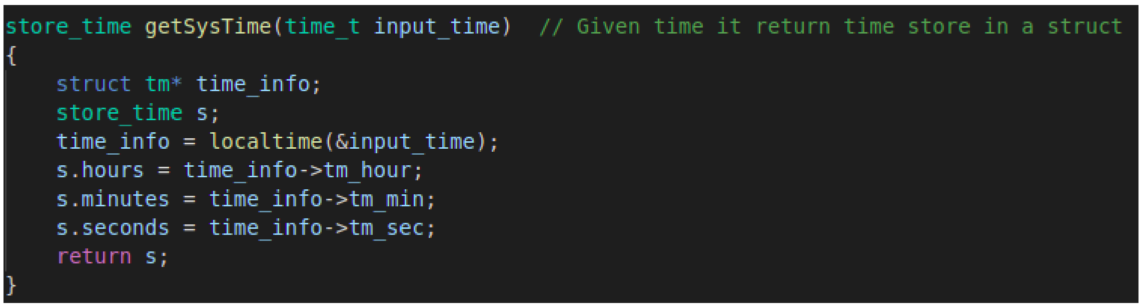
- The store_time struct contains three fields to store hours, minutes, seconds.



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

Vinta Reethu
Operating systems - 3
Indian Institute of Technology Hyderabad
Assignment-2

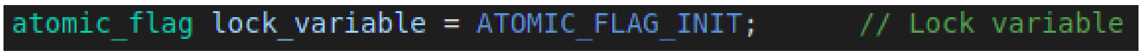
- This function take the time of **time_t** type and stores hours, minutes, seconds in a structure and return it.



```
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;
}
```

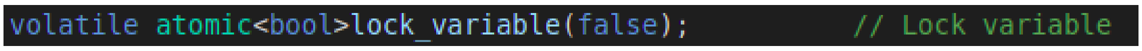
Lock variable for the three implementations is as follows.

- Lock variable for **test_and_set** is given as this.



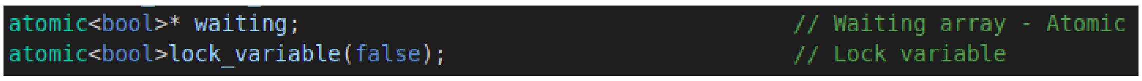
```
atomic_flag lock_variable = ATOMIC_FLAG_INIT;       // Lock variable
```

- Lock variable for **compare_and_swap** is given as this.



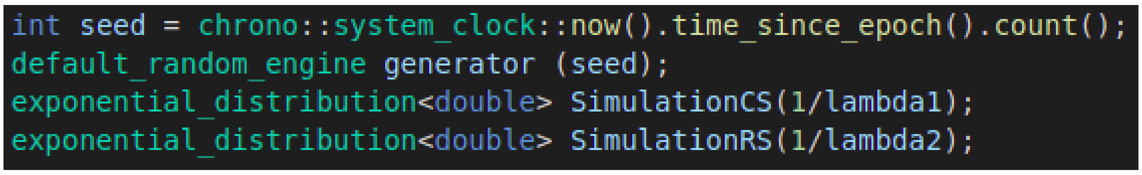
```
volatile atomic<bool>lock_variable(false);          // Lock variable
```

- Lock variable for **compare_and_swap bounded** is given as this. Here waiting is a boolean array whose value will tell if a particular thread can enter critical section or not.



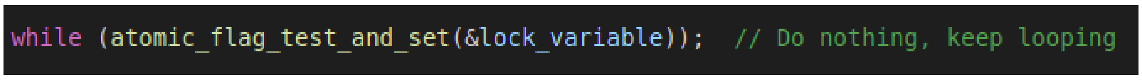
```
atomic<bool>* waiting;                              // Waiting array - Atomic
atomic<bool>lock_variable(false);                   // Lock variable
```

- Here we create two exponential distributions and pass $1/\lambda_1$,$1/\lambda_2$ values. We then use this distribution to obtain t1, t2 values to simulate critical and remainder section. The values we get from the distribution are in milliseconds, so we multiply the values by $1e3x$ before sending into **usleep()** function. **usleep()** function takes input in micro seconds.



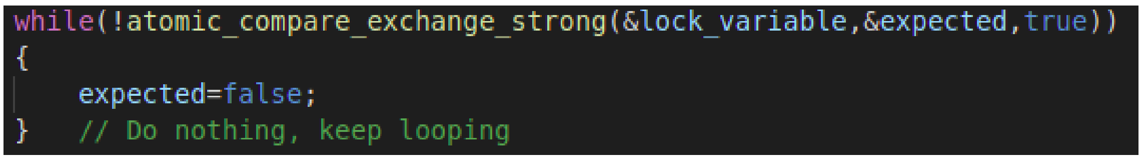
```
int seed = chrono::system_clock::now().time_since_epoch().count();
default_random_engine generator (seed);
exponential_distribution<double> SimulationCS(1/lambda1);
exponential_distribution<double> SimulationRS(1/lambda2);
```

- Implementation of **TAS** Algorithm. We use **atomic_flag_test_and_set** function.



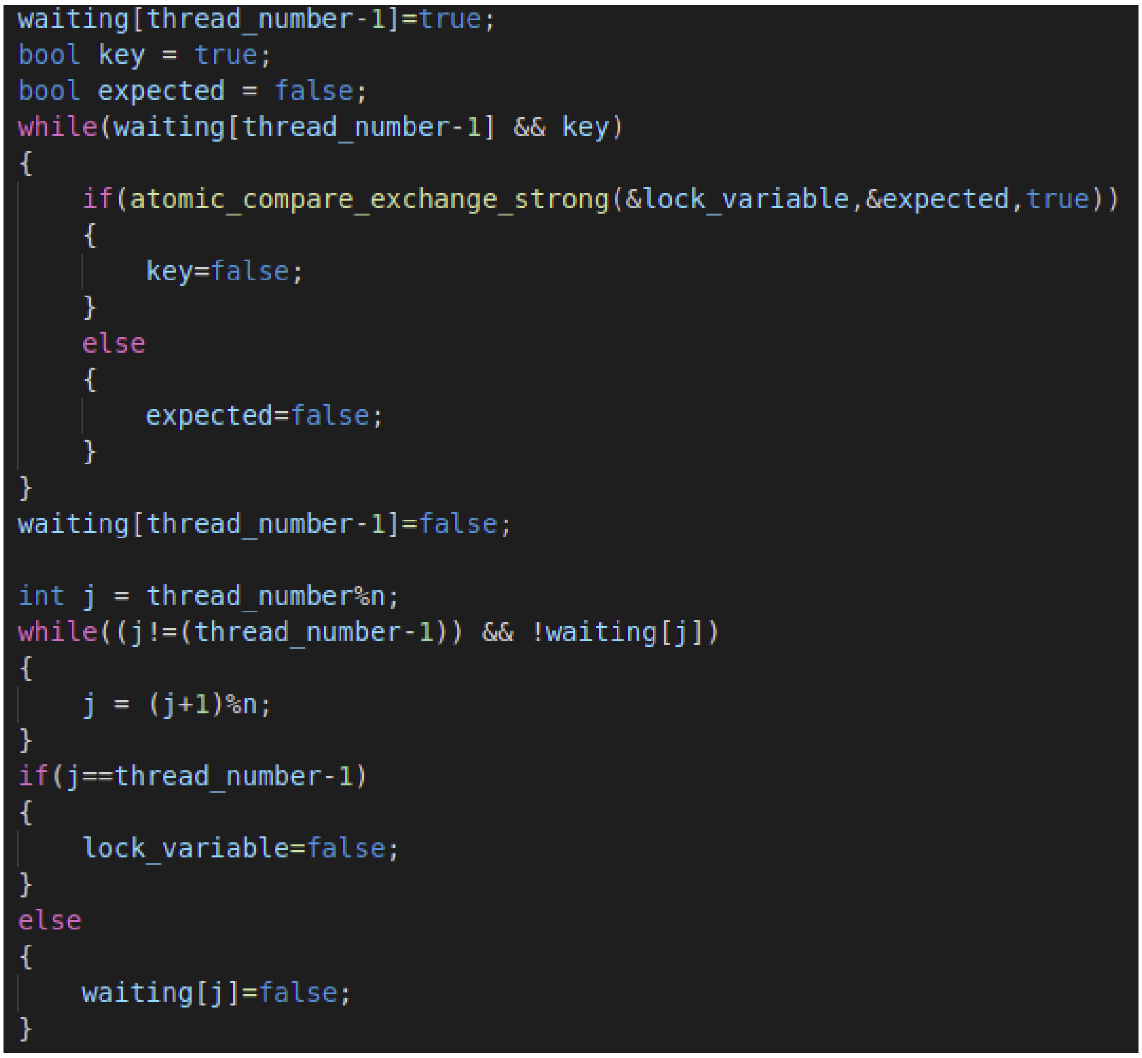
```
while (atomic_flag_test_and_set(&lock_variable));  // Do nothing, keep looping
```

- Implementation of **CAS** Algorithm. The **atomic_compare_exchange_strong** compares value contained in obj with the value of expected. If it is true, it replaces the contained value with val. Else it replaces the value pointed by expected with the contained value. Every time the value in expected should be false. Hence we update it's value in the loop.



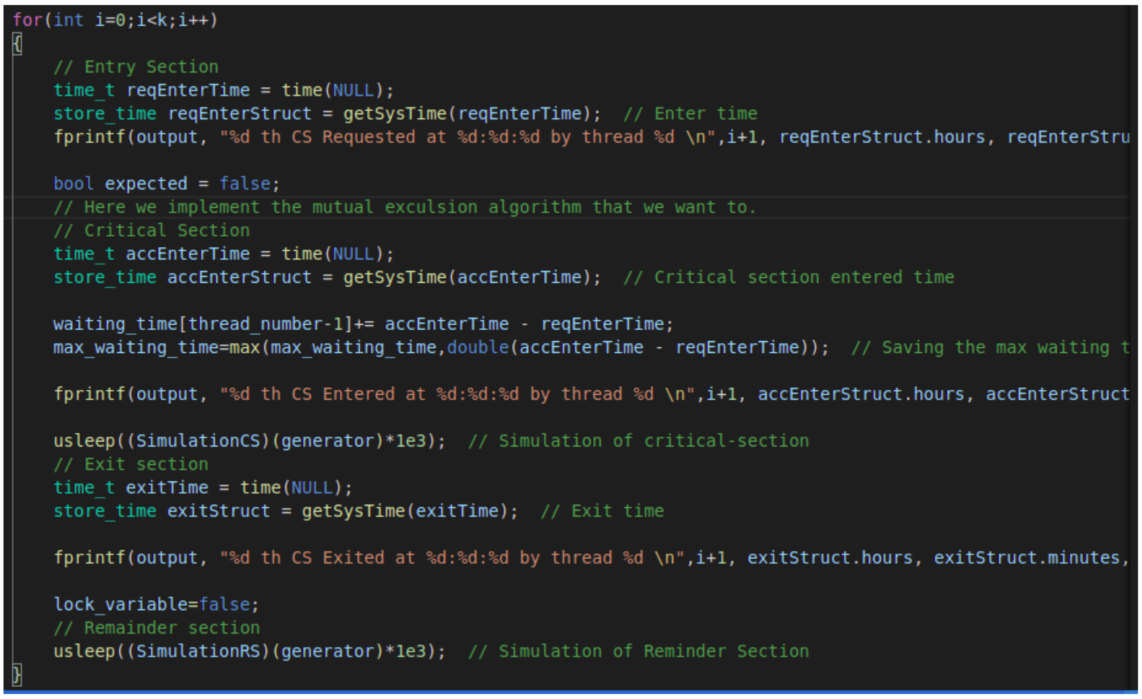
```
while(!atomic_compare_exchange_strong(&lock_variable,&expected,true))
{
    expected=false;
}   // Do nothing, keep looping
```

- Implementation of **CAS-Bounded** Algorithm. A thread can enter its critical section only if either waiting[i] == false or key == 0. If a thread leaves its critical section, it scans the array waiting in the cyclic ordering. It gives chance to the first process in this ordering that is in the entry section (waiting[j] == true) as the next one to enter the critical section.



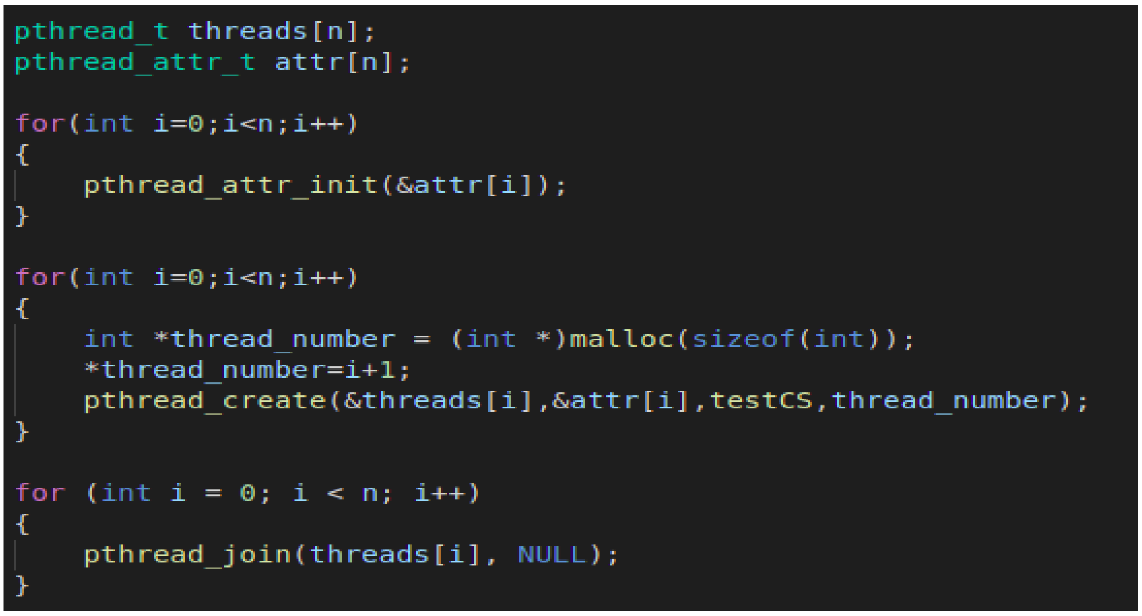
```
waiting[thread_number-1]=true;
bool key = true;
bool expected = false;
while(waiting[thread_number-1] && key)
{
    if(atomic_compare_exchange_strong(&lock_variable,&expected,true))
    {
        key=false;
    }
    else
    {
        expected=false;
    }
}
waiting[thread_number-1]=false;

int j = thread_number%n;
while((j!=(thread_number-1)) && !waiting[j])
{
    j = (j+1)%n;
}
if(j==thread_number-1)
{
    lock_variable=false;
}
else
{
    waiting[j]=false;
}
```

- The main implementation of **testCS** is as given. Most of it same, the mutual exclusion algorithm is as shown above. Firstly a thread requests to enter CS, we store this time. Then based on the ME algorithm it enters into the CS, we store this time as well. Here we simulate CS using usleep() function. After the simulation the thread exit. We try to simulate remainder function using sleep().



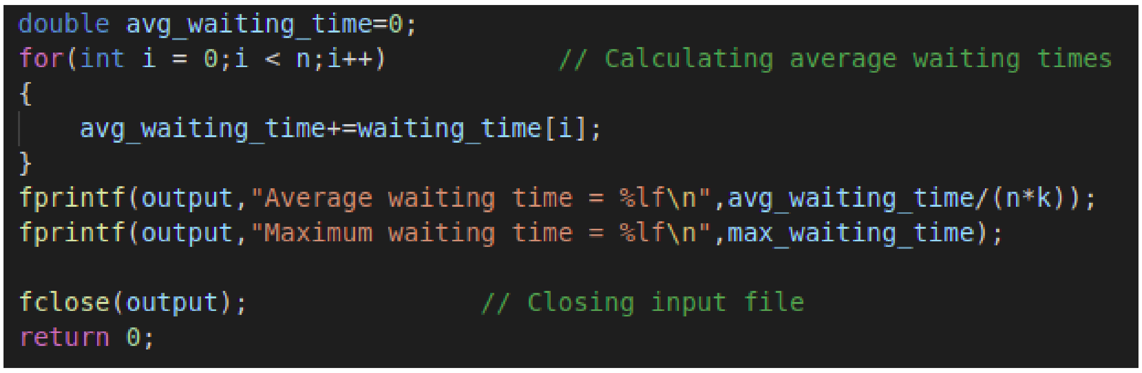
```
for(int i=0;i<k;i++)
{
    // Entry Section
    time_t reqEnterTime = time(NULL);
    store_time reqEnterStruct = getSysTime(reqEnterTime);  // Enter time
    fprintf(output, "%d th CS Requested at %d:%d:%d by thread %d \n",i+1, reqEnterStruct.hours, reqEnterStru

    bool expected = false;
    // Here we implement the mutual exculsion algorithm that we want to.
    // Critical Section
    time_t accEnterTime = time(NULL);
    store_time accEnterStruct = getSysTime(accEnterTime);  // Critical section entered time

    waiting_time[thread_number-1]+= accEnterTime - reqEnterTime;
    max_waiting_time=max(max_waiting_time,double(accEnterTime - reqEnterTime));  // Saving the max waiting t

    fprintf(output, "%d th CS Entered at %d:%d:%d by thread %d \n",i+1, accEnterStruct.hours, accEnterStruct

    usleep((SimulationCS)(generator)*1e3);  // Simulation of critical-section
    // Exit section
    time_t exitTime = time(NULL);
    store_time exitStruct = getSysTime(exitTime);  // Exit time

    fprintf(output, "%d th CS Exited at %d:%d:%d by thread %d \n",i+1, exitStruct.hours, exitStruct.minutes,

    lock_variable=false;
    // Remainder section
    usleep((SimulationRS)(generator)*1e3);  // Simulation of Reminder Section
}
```

- In the main function **n** number of threads are initialised, created by passing testCS function. After thread is terminated we join them using pthread_join.



```
pthread_t threads[n];
pthread_attr_t attr[n];

for(int i=0;i<n;i++)
{
    pthread_attr_init(&attr[i]);
}

for(int i=0;i<n;i++)
{
    int *thread_number = (int *)malloc(sizeof(int));
    *thread_number=i+1;
    pthread_create(&threads[i],&attr[i],testCS,thread_number);
}

for (int i = 0; i < n; i++)
{
    pthread_join(threads[i], NULL);
}
```

- We have stored waiting time of each thread in the waiting_time array. Using this we calculate average waiting time and and print it. We also print maximum waiting time.



```
double avg_waiting_time=0;
for(int i = 0;i < n;i++)          // Calculating average waiting times
{
    avg_waiting_time+=waiting_time[i];
}
fprintf(output,"Average waiting time = %lf\n",avg_waiting_time/(n*k));
fprintf(output,"Maximum waiting time = %lf\n",max_waiting_time);

fclose(output);                // Closing input file
return 0;
```

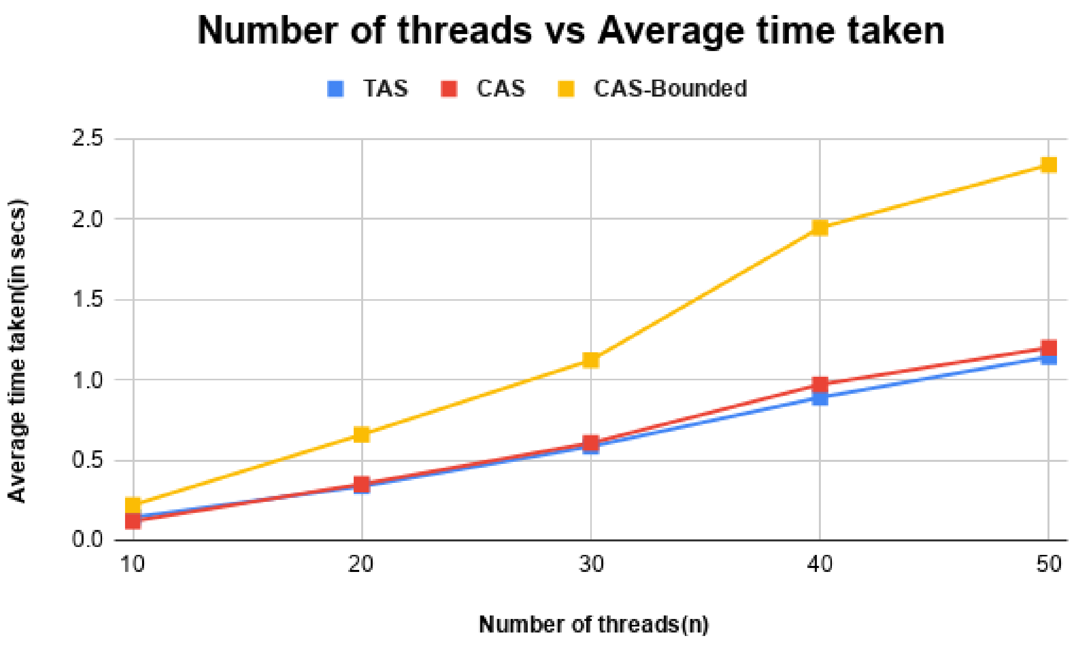
Comparison of the performance

Below 2 graphs are drawn using,

$$\mathbf{k} = 10,\ \mathbf{lambda1} = 20,\ \mathbf{lambda2} = 20$$

Each point in the 2 graphs is obtained by averaging over five runs.

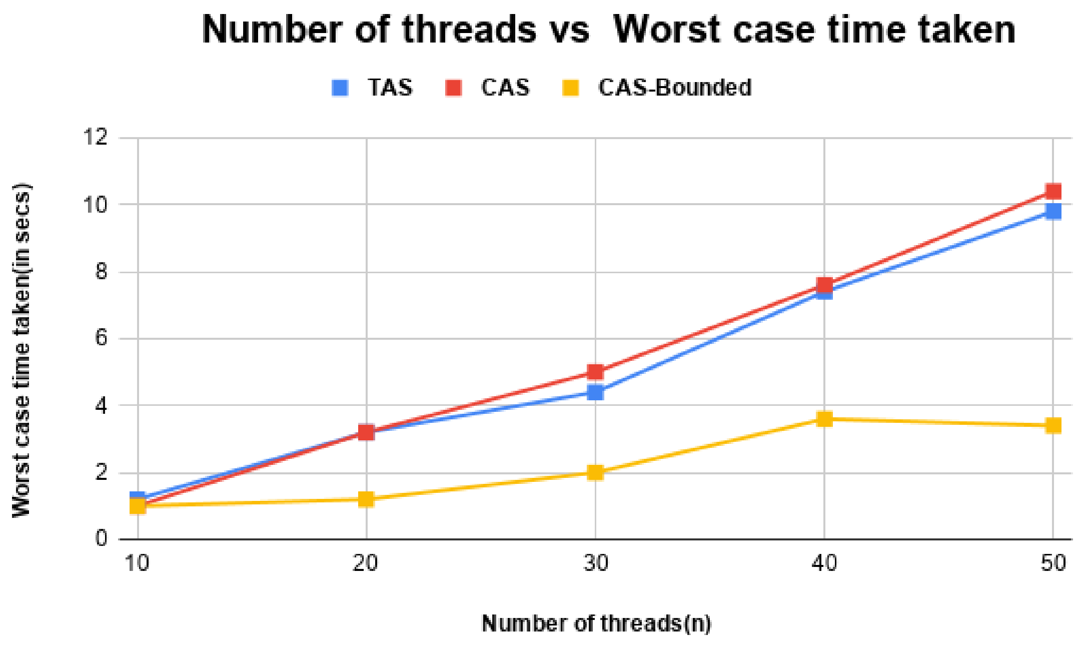
Average time taken graph :



- From the above graph we can see that as number of threads increase the average time taken in the three mutual exclusion algorithms is increasing.
- Average time taken by CAS and TAS almost coincides.

- Out of these 3 ME Algos, CAS Bounded is taking more time.
- CAS Bounded algorithm is taking more time as it ensures there is no starvation in threads. To obtain this average waiting time of each process is slightly compromised.

Worst case time taken graph :



- From the above graph we can see that as number of threads increase the worst case time taken in the three mutual exclusion algorithms is increasing.
- Worst case time taken by TAS and CAS almost coincides.
- It can be seen that CAS Bounded takes the least worst case time.
- CAS Bounded algorithm is taking the least worst case waiting time as it make sure that no thread starve.