Operating Systems–2

CS 3510 Spring 2019

Programming Assignment 2:

Implementing TAS, CAS and Bounded Waiting CAS

Mutual Exclusion Algorithms

INSTRUCTOR: DR. SATHYA PERI

Report By:
Vijay Tadikamalla
CS17BTECH11040

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

Task

The goal of this assignment is to implement TAS, CAS and Bounded Waiting with CAS mutual exclusion (ME) algorithms studied in the class. Implement these algorithms in C++.

Approach and Implementation

1. To achieve our above mentioned goal make a *testCS* function which takes thread_index as a parameter. We will pass this function along with the thread index to the each thread and calculate the average waiting time and max waiting time for all three mutual exclusion algorithm.

```
vector<thread> t; // Array of n threads
for(int i=0;i<n;i++) t.push_back(thread(testCS,i));
for(int i=0;i<n;i++) t[i].join();
```

2. Functions and data-types of the chrono library (and other libraries) like
    a. **std::chrono::system_clock, std::chrono::system_clock::now()**
    b. **std::chrono::time_point**
    c. **struct tm - Time structure**
    d. **struct tm *localtime(const time_t *timer)**

   were used to calculate the average waiting time and max waiting time.
3. **int usleep(useconds_t *usec*)** function was used to suspend the execution of the thread for microsecond intervals
4. **template <class RealType = double> class exponential_distribution**: This is a random number distribution that produces floating-point values according to an exponential distribution, which is described by the following probability density function:

   $$p(x|\lambda) = \lambda e^{-\lambda x} \quad , \; x > 0$$

5. We make two exponential distributions and pass the value of 1/λ1, 1/λ2 in the constructor. Later this can be used to obtain random numbers t1 and t2 with values that are exponentially distributed with an average of λ1, λ2 seconds.

```
distribution1 = new exponential_distribution<double>(1/lt1);
distribution2 = new exponential_distribution<double>(1/lt2);
```

6. The implementation of the critical section varies in *testCS* function with the change in mutual exclusion algorithm.

```cpp
void testCS(int id){           // test function
      chrono::time_point<chrono::system_clock> start_time,end_time;
      // Variables for calculating waiting time
      for(int i=0;i<k;i++){
            // Entry Section Starts
            string reqEnterTime=getSysTime();
            start_time = chrono::system_clock::now();

            /*
                Mutual Exclusion algorithm implementation
                          --Only this part changes
            */

            // Critical Section Starts
            end_time = chrono::system_clock::now();
            fout<<i<<"th CS Requested at "<<reqEnterTime;
            fout<<" by thread "<< id <<endl;
            tmp=(end_time - start_time);
            waiting_time +=tmp;
            // Calculating waiting time
            max_waiting_time = max(max_waiting_time,tmp);
            // Updating max time taken
            fout<<i<<"th CS Entered at   "<<getSysTime();
            fout<<" by thread "<< id <<endl;
            usleep((*distribution1)(generator)*scale);
            fout<<i<<"th CS Exited at    "<<getSysTime();
            fout<<" by thread "<< id <<endl;
            flag.clear();
            // Critical Section ends

            // Reminder Section starts
            usleep((*distribution2)(generator)*scale);
            // Reminder Section ends
      }
}
```

7. Implementation of TAS

```cpp
while(atomic_flag_test_and_set(&flag));
```

8. **Implementation of CAS**

```
while(!atomic_compare_exchange_strong(&flag,&flag1,true)) flag1=false;
```

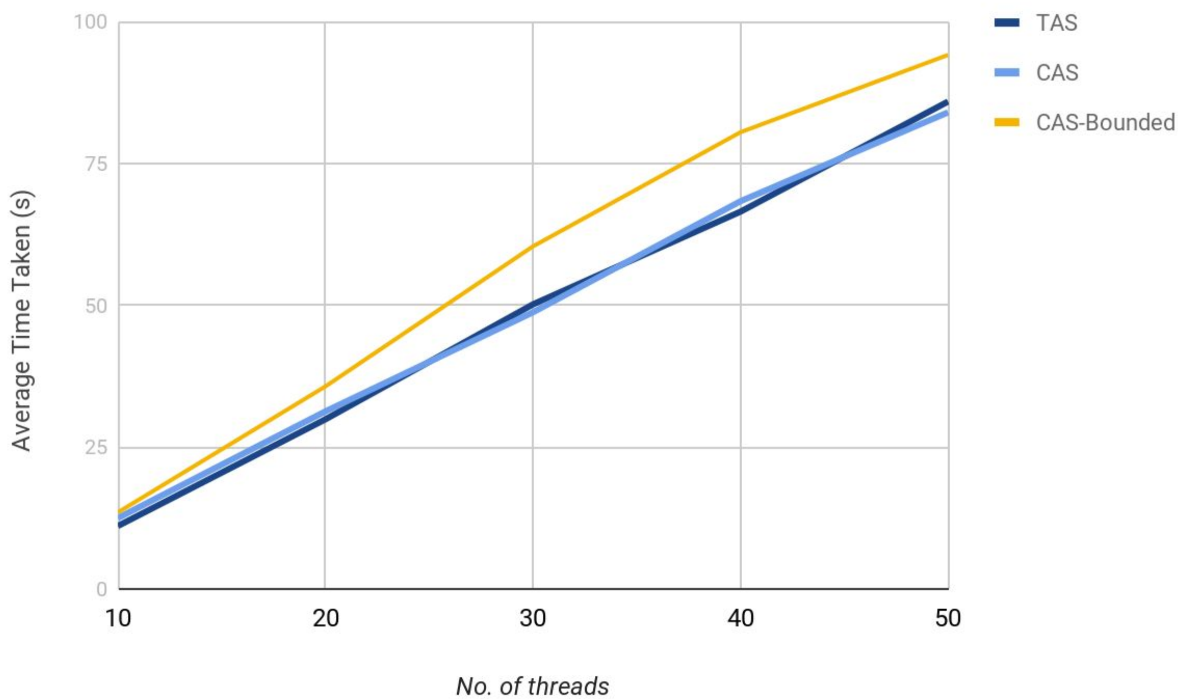
9. **Implementation of CAS-Bounded waiting**
   Here the **waiting[i]==true** represents that thread is waiting to enter the critical section.
   A thread can enter the critical section if **waiting[i]==false or key==false.**
   The later part of the code ensures that no thread starves for a long time.

```
while(waiting[id] && key){//Compare and Swap(Bounded waiting)implementation
      if(atomic_compare_exchange_strong(&flag,&flag1,true)) key=false;
      else flag1=false;
}
// Critical Section Starts
end_time = chrono::system_clock::now();
waiting[id]=false;
usleep((*distribution1)(generator)*scale);
int j=(id+1)%n;
while(j!=i && !waiting[j]) j=(j+1)%n;
if(j==i)flag =false;
else waiting[j]=false;
```

Input Parameters

N: No. of threads vary from 10 to 50 in steps of 10.
K = 10: No. of CS request by each thread
$\lambda 1=2$, $\lambda 2=2$

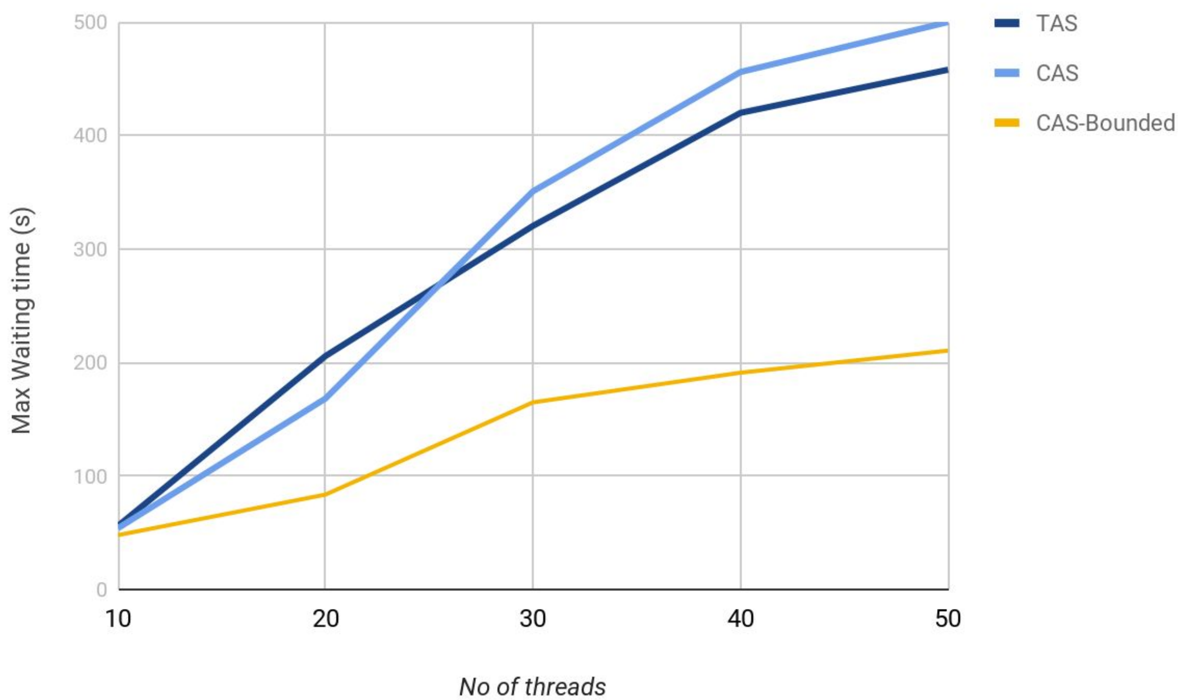
Graph 1 : Avg Waiting Time vs No. of Threads



Output analysis of Graph 1

1. Average waiting time taken by TAS and CAS algorithm is almost the same.
2. Average waiting time taken by Bounded-CAS is greater than both TAS and CAS as it ensures that no thread starve.

Graph 2 : Worst Waiting Time vs No of Threads



Output analysis of Graph 2

1. Worst waiting time taken by TAS and CAS algorithm is almost the same.
2. Worst waiting time taken by Bounded-CAS is much less than both TAS and CAS as it ensures that no thread starves.
3. The difference between the worst waiting times of Bounded-CAS and (TAS,CAS) with increases as the no of threads increase.