Operating Assignment_3

Es21btech11003

Implementing TAS, CAS and Bounded Waiting CAS Mutual Exclusion Algorithms

This report contains a comparison of the performance of TAS, CAS and Bounded CAS algorithms.

At first let us analyse the output of each Mutual Exclusion Algorithms individually

Test_and_set Algorithm :

The implementation of of test_and_set algorithm here , has been done using `atomic_flag_test_and_set` which make the test and set operations to be atomic and help us in create a spin lock for competing threads and allows only a single thread to enter the critical section

```
while(std::atomic_flag_test_and_set(&lock_)); //lock
```

Output Analysis :

Test case : input : “5 5 20 5” i.e. no. of thread = 5 , k = 5 , lamda_1 = 20 , lamda_2 = 5

First 20 line of output are as:

Output:

1th CS Requested at 0.296000 by thread 1
1th CS Requested at 0.364000 by thread 4
1th CS Requested at 0.333000 by thread 3
1th CS Requested at 0.354000 by thread 5
1th CS Entered at 0.368000 by thread 1 // first entry of thread 1
1th CS Exited at 3.026000 by thread 1 // thread 1 exits
2th CS Requested at 3.053000 by thread 1
1th CS Requested at 0.289000 by thread 2
1th CS Entered at 3.026000 by thread 3 // thread 3 enters after thread 1 exits
1th CS Exited at 14.895000 by thread 3
1th CS Entered at 14.895000 by thread 2 // thread 2 enters after thread 3 exits
2th CS Requested at 14.903000 by thread 3
1th CS Exited at 14.908000 by thread 2
2th CS Entered at 14.909000 by thread 1 // thread 1 again enter but thread 4 and 5
2th CS Exited at 14.961000 by thread 1 never entered (fairness is not followed)
3th CS Requested at 14.973000 by thread 1
2th CS Requested at 14.913000 by thread 2
2th CS Entered at 14.961000 by thread 3 // thread 3 again enters after thread 1 exits

2th CS Entered at 16.065000 by thread 2
2th CS Exited at 16.255000 by thread 2

Here we can see each thread first comes and requests for entering critical section , each thread enters the CS one by one and wait for others to exits - hence follows mutual exclusion
But the 4 and 5 , never got a chance to enter the CS before thread 1 entered again - fairness and bounded waiting not followed .

Compare_and_swap Algorithm :

For compare and swap algorithm , we have used **`atomic_compare_exchange_strong`** function which help us make compare and swap operations atomic and under the hood using compare and swap algorithm creates a spin lock for competing threads also allowing only one thread to enter critical section.

```
//Spin lock
        while(true){
            int expected_value = 0;
            if(std::atomic_compare_exchange_strong(&lock_, &expected_value, 1)!= 0){
                break;
            }
            //spin until the lock is acquired
        }
```

Output Analysis :

Test case : input : “5 5 20 5” i.e. no. of thread = 5 , k = 5 , lamda_1 = 20 , lamda_2 = 5
First 20 line of output are as:
Output:
1th CS Requested at 0.228 ms by thread 1
1th CS Requested at 0.295 ms by thread 5
1th CS Requested at 0.263 ms by thread 3
1th CS Requested at 0.278 ms by thread 4
1th CS Entered at 0.296 ms by thread 1 // thread 1 enters
1th CS Exited at 0.859 ms by thread 1 // thread 1 exits
1th CS Requested at 0.252 ms by thread 2
2th CS Requested at 0.911 ms by thread 1
1th CS Entered at 0.859 ms by thread 3 // thread 3 enters after thread 1 exits
1th CS Exited at 5.101 ms by thread 3 // thread 3 exits

1th CS Entered at 5.101 ms by thread 4 // thread 4 enters after thread 3 exits
2th CS Requested at 5.106 ms by thread 3
2th CS Entered at 5.114 ms by thread 3 // thread 3 enters after thread 4 exits
1th CS Exited at 5.114 ms by thread 4 | // thread 4 exits
2th CS Exited at 5.121 ms by thread 3 | { Note : here output is in jumbled order as
1th CS Entered at 5.122 ms by thread 5 | fprintf store output in buffer and
3th CS Requested at 5.126 ms by thread 3 | takes time to print the output ,thread 4
1th CS Exited at 5.139 ms by thread 5 | enters only after thread 3 has exited}
2th CS Entered at 5.139 ms by thread 1 |
2th CS Requested at 5.144 ms by thread 5

Here also we can see each thread first comes and requests for entering critical section , each thread enters the CS one by one and wait for others to exits - hence follows mutual exclusion But in this case 5 , never got a chance to enter the CS once while thread 3 entered twice - fairness and bounded waiting not followed .

Bounded_Compare_and_swap Algorithm :

The implementation of of bounded_compare_and_swap algorithm also uses **`atomic_compare_exchange_strong`** function , the difference here is that we update the compare_swap algorithm to check for waiting thread is an ordered manner to implement bounded waiting

```
//Spin lock
while(waiting[Thread_id - 1] && key == 1){
    int expected_value = 0;
    key = std::atomic_compare_exchange_strong(&lock_, &expected_value, 1);
}
// spin until the lock is acquired
waiting[Thread_id - 1] = false;
```

```
int j = (Thread_id)%n;
while ((j != (Thread_id - 1)) && (!waiting[j])){
    j = (j + 1) % n;
}

if(j == (Thread_id - 1)){ lock_ = 0;}
else{waiting[j] = false;}
```

Using while loop we check for next waiting thread ; if no thread is waiting we release the lock else if a thread is found waiting then we hand over CS to that thread

Output Analysis :

Test case : input : “5 5 20 5” i.e. no. of thread = 5 , k = 5 , lamda_1 = 20 , lamda_2 = 5

First 20 line of output are as:

Output:

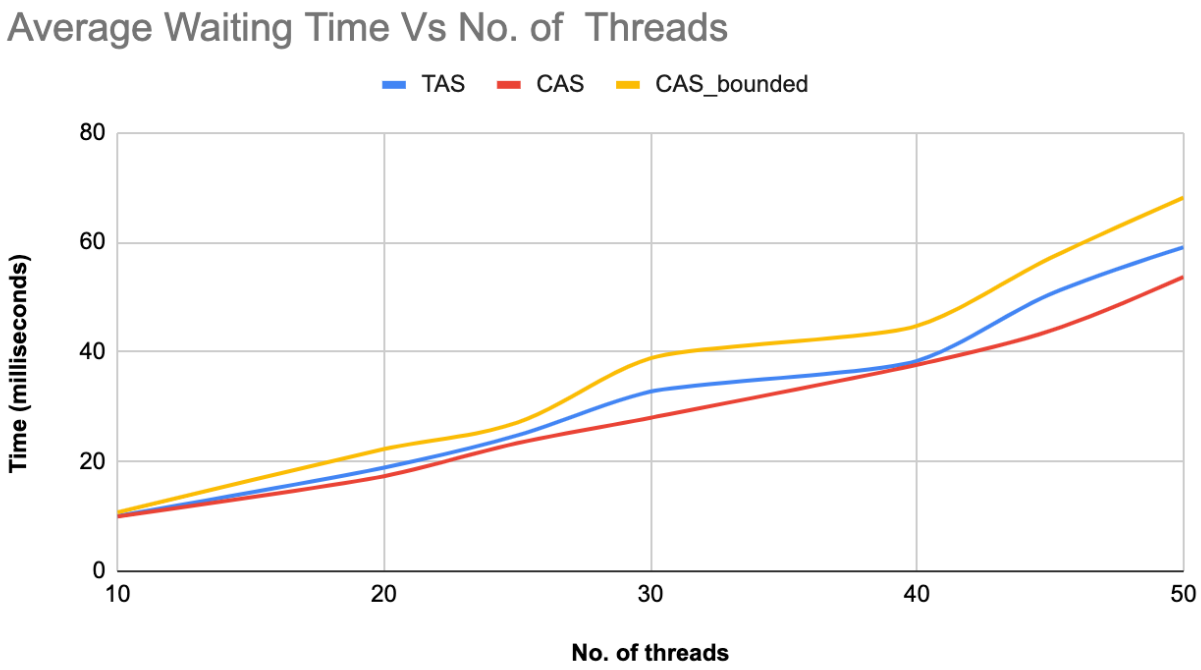
```
1th CS Requested at 1.339 ms by thread 1
1th CS Entered at 1.382 ms by thread 1          // thread 1 enters
1th CS Requested at 1.345 ms by thread 2
1th CS Exited at 1.405 ms by thread 1           // thread 1 exits
1th CS Requested at 1.364 ms by thread 3
1th CS Entered at 1.414 ms by thread 2          // thread 2 enters
1th CS Requested at 1.382 ms by thread 4
1th CS Exited at 1.426 ms by thread 2           // thread 2 exits
1th CS Entered at 1.429 ms by thread 3          // thread 3 enter
1th CS Exited at 1.439 ms by thread 3           // thread 3 exits
1th CS Entered at 1.470 ms by thread 4          // thread 4 enters
1th CS Exited at 1.483ms by thread 4            // thread 4 exits
1th CS Requested at 1.399 ms by thread 5
1th CS Entered at 1.484 ms by thread 5          // thread 5 enters
2th CS Requested at 1.436 ms by thread 2
2th CS Entered at 1.495 ms by thread 2
1th CS Exited at 1.494 ms by thread 5           // thread 5 exits
2th CS Requested at 1.503 ms by thread 5
2th CS Entered at 1.506 ms by thread 5
2th CS Exited at 1.505 ms by thread 2
```

Similar to TAS and CAS the mutual exclusion is satisfied , thing to note here is that if a thread has already made a requested then it is ensured to be ran in a bounded period of time as we can see from the example .

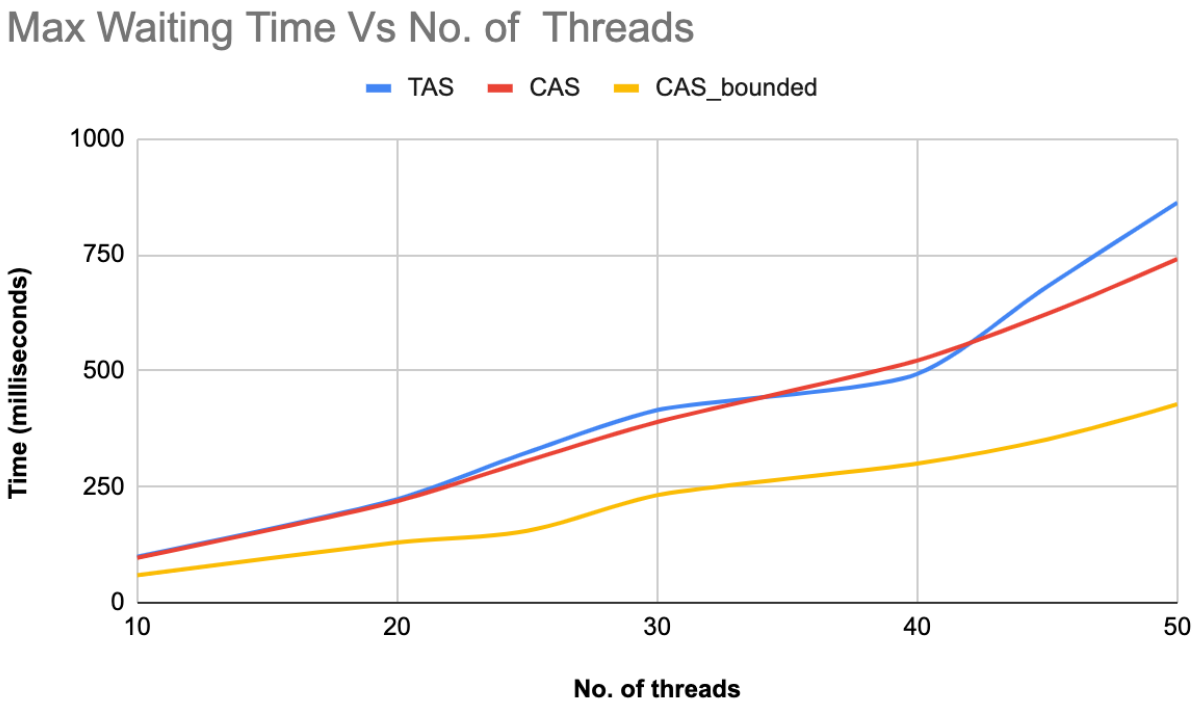
Bounded_Compare_and_swap Algorithm :

No. of Threads	TIME(milliseconds)					
	TAS		CAS		CAS_bounded	
	Avg Wait	Max wait	Avg Wait	Max wait	Avg wait	Max wait
10	10.071388	98.7876	9.98104	96.5306	10.7399	58.9104
20	18.8878	223.005	17.3349	218.759	22.292	129.797
25	24.7731	323.849	23.37	305.839	27.127	154.872
30	32.7751	415.252	27.9904	389.611	38.854	231.897
40	38.3691	493.197	37.6597	522.234	44.755	299.78
45	50.5484	681.594	43.8591	622.79	57.13	352.21
50	59.1111	862.628	53.6927	740.952	68.15	427.83

These times have been calculated in reference the “start” time variable whose value is initialized to the start time of execution , each output is averaged over five times
As we can see - In average wait time comparison each perform equally well , with bounded_CAS algo being the slightly on the higher end.



Where as in comparison of worst wait time , the wait time of bounded_CAS is least as compared to TAS , CAS . This is as expected as Bounded CAS follows bounded waiting



Comments :

1) The input file name should be “inp-params.txt” as metioned in the assignment problem also
2) Time library used - chrono
3) The Time is measured in reference to “start” time variable whose value is initialized to the start time of execution.
4) Each program outputs a file “Outfile<ProgName>.txt” in which it prints the desired output