SAKSHAM MITTAL

CS16BTECH11032

Course : Operating Systems–2 (CS3523)

23 January 2018

Programming Assignment 2:

Implementing Critical Section Solution using test_and_set, test_and_set with Bounded Waiting and compare_and_swap in C++.

GOAL OF THIS ASSIGNMENT

The goal of this assignment is to implement a multithreaded solution for critical section problem using the three techniques discussed in the class: test_and_set, test_and_set with Bounded Waiting and compare_and_swap. You have to implement these three algorithms and compare the time taken for each thread to access the critical section. Implement all the algorithms in C++.

DESIGN OF **Assgn2-TASCS16BTECH11032.cpp**

We read the inputs from the file '*inp-params.txt*' which include n (the number of threads), k (number of times a thread tries to access the critical section), csSeed (the seed for time sent in critical section), remSeed (the seed for time spent in remainder section).

So, n threads are created which call *testCS()*. The structure of *testCS()* function is as follows:

It runs a loop for k times to access the Critical Section. And the time at which the thread request its entry is printed on '*TAS-Log.txt*'. To make the print statement atomic, we use ***printf()*** statements instead of ***cout*** statements.

For test_and_set() function we use Standard Library function ***atomic_flag_test_and_set(&lock)*** which is an atomic function and sets the lock to true, thus disabling the access of other threads to Critical Section.

Once the thread comes out of the Critical Section (which is simulated by making the thread sleep for random times), the lock is cleared and some other thread may now enter the Critical Section. The given thread now spends time in the remainder section which is also simulated by making the thread sleep.

The waiting time for each thread is calculated by (*time at request - time at entry*). At last the average is taken for all threads (in seconds) and printed to '*Average-times.txt*' file.

DESIGN OF **Assgn2-TAS_FairCS16BTECH11032.cpp**

The basic outline is same as that of above program, but the following details are added:

Now we include a waiting array to maintain the threads that have been waiting. The pseudo code is changed as follows:

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;

      /* critical section */

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = false;
   else
      waiting[j] = false;

      /* remainder section */
} while (true);
```

This allows the bounded-waiting requirement to be satisfied too. Now, any process waiting to enter critical section will do so within n - 1 turns.

DESIGN OF **Assgn2-CASCS16BTECH11032.cpp**

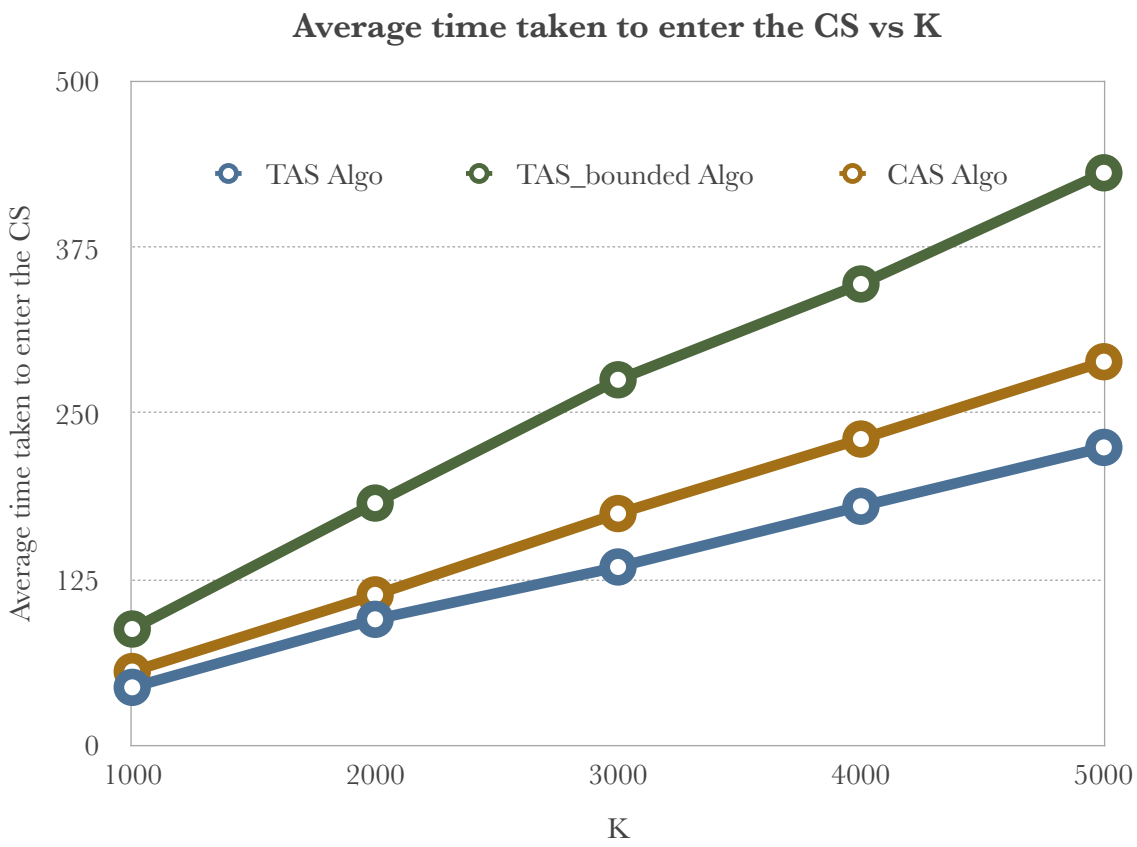
The basic outline is same as that of TAS program, but the following details are added:

Now we use ***atomic_compare_exchange_weak(&lock1, 0, 1***) function for locking. The pseudo code is changed as follows:

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```

This allows the bounded-waiting requirement to be satisfied too. Now, any process waiting to enter critical section will do so within n - 1 turns.

GRAPH FOR COMPARISON



THE EXACT DATA: (N = 10, csSeed = 20, remSeed = 30)

	1000	2000	3000	4000	5000
TAS Algo	43.9	95.1	134.5	180.2	224.5
TAS_bounded Algo	87.8	182.6	275.4	347.5	431.2
CAS Algo	55.8	113.5	174.7	230.8	289

NOTE: All times are in seconds.

Inference:

From the graph we can see that for the average waiting time is maximum for TAS_bounded algorithm, and next is CAS algorithm and at last TAS Algorithm. Also we notice that the average waiting time keeps on increasing with increase in K as more number of threads try to access the critical section, which would require more and more threads to wait for their turn.

A reason for low performance for TAS_bounded algorithm is that it checks with a while loop which process is waiting and for that traverses the waiting[] array. This increase the waiting time when the order of inputs is very large as is in our case.

As for CAS algorithm, we perform an extra instruction than TAS algorithm in which we check whether the expected value is same as the new value. This overhead leads to an increase in average waiting time overall.

Thus, the order of waiting time is justified.