Operating Systems2(CS3523) Programming Assignment 3 Implementing TAS, CAS and Bounded Waiting CAS Mutual Exclusion Algorithms

Govinda Rohith Y
CS21BTECH11062

Low level design:

For all three methods all the functions except the function `void* each_thread_tas(void *arg)` is different. So below exaplanation is given for each function which are used in common:

1. `main` function:

   The input values n,k,l1(λ_1),l2(λ_2) is read from file "inp-params.txt" and 3 2D sting arrays each of size n×k are allocated and intiliased to zero, one 2D array to store each request time for critical section `string ** req_t`,next array to store entry time `string ** entry_t` and another array to store exit time `string ** exit_t`. All the variables described above are declared globally. The two exponential generating objects distribution1 and distribution2 are intiliased(refer `var_rand(bool is_t1)` description given below). Now the function `foo();` is called which creates required number of threads and execute it accordingly(refer `void foo()` description given below). After completion of all threads to print the average and worst time to enter CS the function `void metric();` if called. At last all the allocated memory is freed to avoid memory leaks.

2. `exponential_distribution<double> var_rand(int meaner)`

   At high level this function returns object which generates exponential distributed numbers when called whose mean is given as argument. To generate this first `random_device rd` is intiliased globally which is used to produce non deterministic random numbers. This is used to generate seed value for `mt19937 gen(rd())` which is very efficient pseudo-random number generator , by Mersenne twister algorithm , this is used later to generate random numbers. In the function `var_rand(int meaner) exponential_distribution<double>` object is intiliased according to given mean and this is returned. Two objects are intiliased `exponential_distribution<double> distribution1,distribution2` to generate two exponentially distributed random number when called each time (`distribution1(gen)`) whose means are $\lambda_1$andλ_2 respectively. Note the gen is seed for mt19937 object which is intiliased before.

3. `void foo()`

   This function creates 'n' p_threads and waits for them to complete their work. A `pthread_t *pid` array is intiliased to store thread ids of each created thread and for loop is run to generate n threads and call each thread function `void * each_thread_tas(void *arg)` with thread number-1 as an argument. Then for loop is run to wait for completion of work by each thread. Then function `void file_printer()` is called to print output file accordingly.

4. `void file_printer()`

   This function is to print output file as per given requirements. All the request,entry,exit times stored in arrays are read and printed into "output.txt" as per given instructions.

5. `std::string get_time()`

   This function returns current system time in HH:MM:SS.mmm (mmm is milliseconds) format as string. `using clock = system_clock` creates an alias for system_clock. clock::now gives the system time and assign to variable current_time_point and in next two steps it is converted into duration from epoch. Then the next expression (`duration_cast<milliseconds> (current_time_since_epoch).count() % 1000`) calculates the number of milliseconds that have elapsed since the current second began. Then the time is converted into string format HH.MM.SS.mmm (where m is milliseconds) and returned.

6. `void sleeper(double req_time)` This function is used to sleep the current thread for time in seconds which is given as argument `double req_time`. This is mainly implemented by using nanosleep() function. The parameter `req_time` is divided by 1000 to get time in milliseconds. `struct timespec tim` is intiliased with tim.tv_sec=(int)req_time/1 which is time in seconds and tim.tv_nsec=(req_time-(int)req_time/1)*1000000000 which is time in nanoseconds example if 2004.678 milliseconds is the parameter value then tim.tv_sec= 2 seconds and tim.tv_nsec=4678000 nanoseconds. Then nanoleep function is called which make sleep the current thread according to specified time.

7. `void metric()` This function is to print statistical data like average time required to enter CS section and worst time required to enter CS section which is used in graph plotting. After completion of all threads based on time stamps stored in 3 2D arrays average and worst case time is calculated. Two for loops are run to calculate each value, where each value is calculated by extracting the substrings of string `(string.substr(start index,int length))` which contain the timestamps these substring are converted into integers by stoi method. Then the time difference between the entry time and request time (`entry_t-req_t`) in milliseconds gives the CS entry times and average and worst case (max time) is calculated accordingly.

The functions which differ for 3 algorithms is `void * each_thread_tas(void *arg)` this is the function which runs on each thread. So for all algorithms a for loop is run for k times as specified. For TAS algorithm for each iteration the request time for CS section is calculated by function `void file_printer()` and is assigned into 2D array example for thread i and iteration j the request time is assigned into req_t[i][j]. Then spinning is done on atomic variable lock_stream by using `while(lock_stream.test_and_set())` which is inbuilt C++ test and set algorithm. When any one of the thread completes spinnning and exits while loop it enters into critical section and entry time is recorded to variable entry_t[i][j](for thread i and iteration j) in the same way as before. Then the simulation of critical section is done by generating a number `(distribution1(gen))` and making the thread sleeep for that time using function `void sleeper(double req_time)`. After completion of CS the lock stream is set to false by using inbuilt c++ function `lock_stream.clear()` indicating that CS section is done and other thread can enter. Now the exit time is recorded into variable exit_t[i][j](for thread i and iteration j). Then simulation of remainder section is done by generating a number using (`distribution2(gen)`) and making the sleep for that time.

For CAS algorithm everything is similar to TAS except that spinning is done on atomic variable flag by using `while (!flag.compare_exchange_weak(expect,true)) expect=false;` which is inbuilt C++ compare and exchange method.Then the simulation of critical section is done by generating a number `(distribution1(gen))` and making the thread sleeep for that time using function `void sleeper(double req_time)`. After completion of CS the atomic variable flag is set to false by using inbuilt C++ function `flag.store(false, std::memory_order_release);` indicating that CS section is done and other thread can enter. Now the exit time is recorded into variable exit_t[i][j](for thread i and iteration j). Then simulation of remainder section is done by generating a number using (`distribution2(gen)`) and making the sleep for that time.

For CAS Bounded waiting algorithm boolean array of size n is intiliased to false `bool *waiting` and a boolean local variable key is intiliased to false and now spinning is done on variables key and waiting[i](i represents thread number) for each thread and the while loop which does spinning uses inbuilt C++ compare and exchange method `key=flag.compare_exchange_weak(expect,true, std::memory_order_acq_rel);` essentially used to make wait other threads while one thread is in CS. Before spinning the request time is recorded into variable req_t[i][j](for thread i and iteration j). When the spinning is over one of thread enters into critical section now the entry time is recorded into variable entry_t[i][j](for thread i and iteration j) and waiting[i] is set to false indicating that thread i is not waiting for CS. Then the simulation of critical section is done by generating a number `(distribution1(gen))` and making the thread sleeep for that time using function `void sleeper(double req_time)` and exit time is recorded into exit_t[i][j]. After completion of CS next thread number is assigned into variable 'j' (`j = (i + 1) % n`) now a while loop is run to find to find which thread is waiting based on waiting[j] (if waiting[j] is true then thread j is waiting for CS). If there exists a thread which is waiting then it is allowed into critical section by setting waiting[j]=true if no thread exits then the atomic variable flag is set to false by using inbuilt C++ function `flag.store(false, std::memory_order_release);`. Then simulation of remainder section is done by generating a number using (`distribution2(gen)`) and making the sleep for that time.

Graph analysis and anomalies:

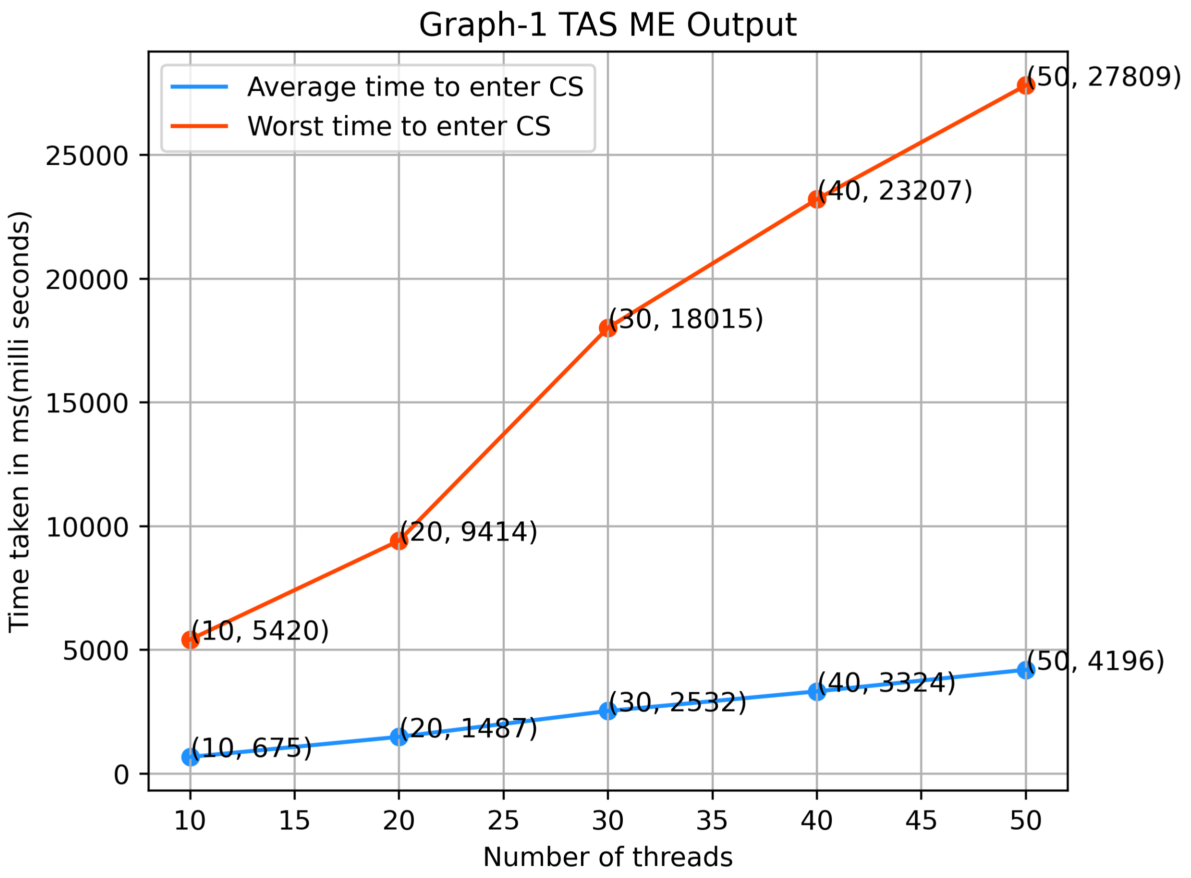

Figure 1: Plot showing the average and worst time of entering the CS Section by TAS instruction

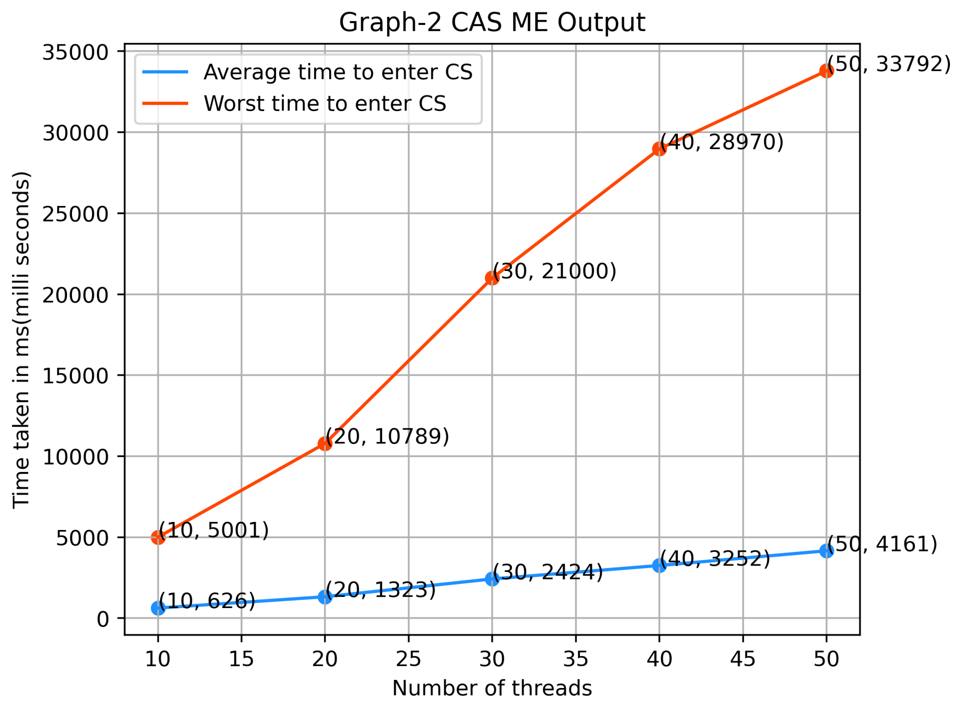

Figure 2: Plot showing the average and worst time of entering the CS Section by CAS ME instruction

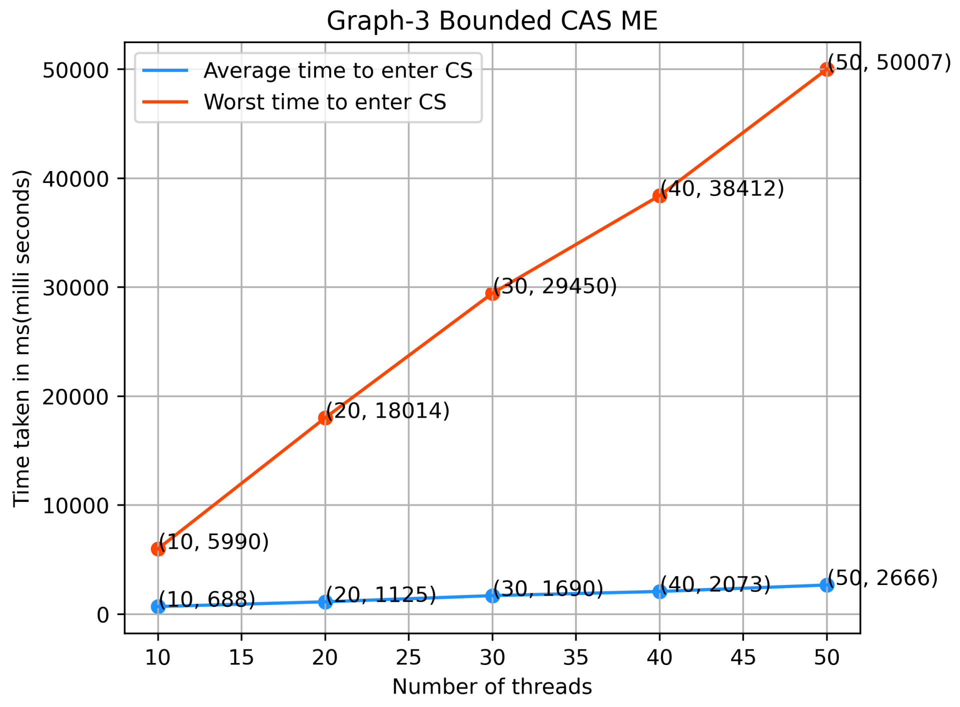

Figure 3: Plot showing the average and worst time of entering the CS Section by Bounded CAS ME method

For all above graphs Figure 1,Figure 2,Figure 3 we can see increasing trend for both plots . For average time plot, increasing trend is observed because as the number of threads are increasing the competetion to enter CS section also increases therefore the average time is increasing with increase in number of threads. The worst case time is also increasing because of more threads the waiting time is more to enter critical section(Starving problem). One anomaly is that the rate of increase of worst case plot is more than average case plot which shows the starving problem of thread. As number of threads are increased the starving of each thread also increases so the worst case is increasing at higher rate.

Performance comaparision of three algorithms:

For all above graphs Figure 1,Figure 2,Figure 3 we can see that average plots for Bounded CAS algorithm is better than other two because in Bounded CAS algorithm the threads are entered into critical section in adjacent cyclic order because of which average time is overall same. For other two algorithms the average plot is increasing at significant rate which shows any thread can enter into critical section.

For Worst case plots we can see that Bounded CAS algorithm is having more worst case time this is obvious because of starving problem. Like if thread 1 is executing CS section among 100 waiting threads the 100th thread has to starve untill completion of other 99 threads so the worst case time is very high. But for CAS ME and TAS any ONE thread can enter into critical section so the worst case times are significantly lower. But when TAS and CAS ME are compared the worst case time of CAS ME > TAS this shows starving problem in CAS ME.

Note that in output file no two entry times of two threads are same for all three algorithms, which shows three algorithms satisfy mutual exclusion principle that no two threads show execute critical section at same time.

Analysis of the algorithms:

For given n=number of threads and k=number of iterations TAS,CAS ME algorithms take space complexity of O(3×n×k) to store the time stamps . Bounded CAS ME algorithm takes O(3×n×k+n) that extra n is because of array bool* waiting. For all three algorithms the time complexity is $<$ O(($\lambda_1 + \lambda_2$)×n×k). Time complexity can be calculated as assuming each thread sleeps for approximately λ_1 amount of time to simulate critical section and λ_2 amount of time for remainder section so for total of n threads and each thread with k iterations takes $<$ O(($\lambda_1 + \lambda_2$)×n×k). That "less than" is occuring because of the fact that, the remainder section is executed in parellel so the time may decrease.