Operating Systems - 2
Spring 2019

Instructor - Dr. Sathya Peri
Jatin Chauhan - CS17BTECH11019

Design of Program:

The program has been designed to use locks for implementing critical section part of code, using two different strategies -> **test_and_set** and **compare_and_swap.**
To run the threads without starvation, **compare_and_swap** is also run with **bounded_waiting** strategy. The libraries used in the implementation are -

<pthread.h> - for creating threads.
<atomic> - Use for executing operation on the lock variable as atomic.
<random> - To create random number generating engines.
<unistd.h> - To calculate current system time.
<fstream> -Creating file streams.

Various sections of code are explained below:

1) ***Entry Section:*** The segment of code which every thread has access to and where they contend to enter the Critical Section.
2) ***Critical Section:*** The segment of code (with potentially important instructions), which we want only a single thread should access, to avoid data_races.
3) ***Remainder Section:*** The segment of code where the thread releases the lock for other threads to acquire and executes the remaining (potentially not exposed to other threads or processes) instructions.

Waiting Time: The time spent by thread in the entry section to acquire the lock and start the execution of critical section part. The worst waiting time for any thread is therefore termed as the *worst waiting time.*

Implementation of Program:

The implementation of the programs follows same structure in the **main** function. The difference arises in the ***testCS*** function.

Multiple global variables are created to hold the information about the threads, waiting time, worst waiting time of a thread, average time value for critical section, average time value for remainder section, waiting information. These are defined below:

n: Number of threads

times: number of times each thread executes

average_cs_section: value of mean for distribution for simulating critical section.

average_rm_section: value of mean for distribution for simulating remainder section.

wait_time: (Σ wait[i]) / (n * times), where wait[i] is the total waiting time for each thread.

engine_1, engine_2: random number generating engines which gives the means for generating a time value for critical section and remainder section simulations

The common section for the three programs is described here -

The input is read from a file which initializes the values of n, times, average_cs_section and average_rm_section (in the given order). The thread_information array is initialized, where each entry is set equal to **times** variable. Array **thread_workers** is then created to store the **ids** of the created threads, and another array **thread_attr** stores the attributes for each thread. The attributes are initialized using **pthread_attr_init** function.

The threads are created using **pthread_create** function of the **pthread** library. (An additional array **waiting_info** is created for implementing cas_bounded_waiting program, which is boolean array to store which threads are currently available to take hold of the critical section.

The **pthread_create** function takes the ***testCS*** function as its runner method for the threads.

The **log_file** variable is pointer of type **FILE,** which opens two files, one to store the times for all the threads at which they request, enter and exit the critical section. Average and worst waiting time are written to the other file.

The time at which the threads start the loop is noted for every iteration in the **request_time** struct of type **time_t.** This is then created in the hour:mm:sec format using the **localtime** method in the **unistd** header library. This is then written to log file. Similarly the values of **start_time and finish_time** variables are written to the logger file. The critical section and remainder section are simulated by making the threads sleep for some time, which is dynamically calculated at every iteration using the **cs_section and rm_section** distribution objects.

The testCS function is different for all the three programs.

1) **TAS:** The ***lock*** variable is of type ***atomic_flag.*** This is initialized to *ATOMIC_FLAG_INIT.* The test_and_set atomic function is called over this using the ***atomic_flag_test_and_set_explicit*** method of the **atomic** header library, as soon as the thread enter the loop. The simulation of critical section is then run. The count of number of times the thread should run is the reduced. The value of lock variable is then reset using the ***atomic_flag_clear_explicit*** method. The exit time is calculated and written to the log file.

2) **CAS:** The ***lock*** variable is defined as an ***atomic int.*** Here ***compare_exchange_strong*** method is used, which takes the values of expected and current values and returns the values accordingly. If

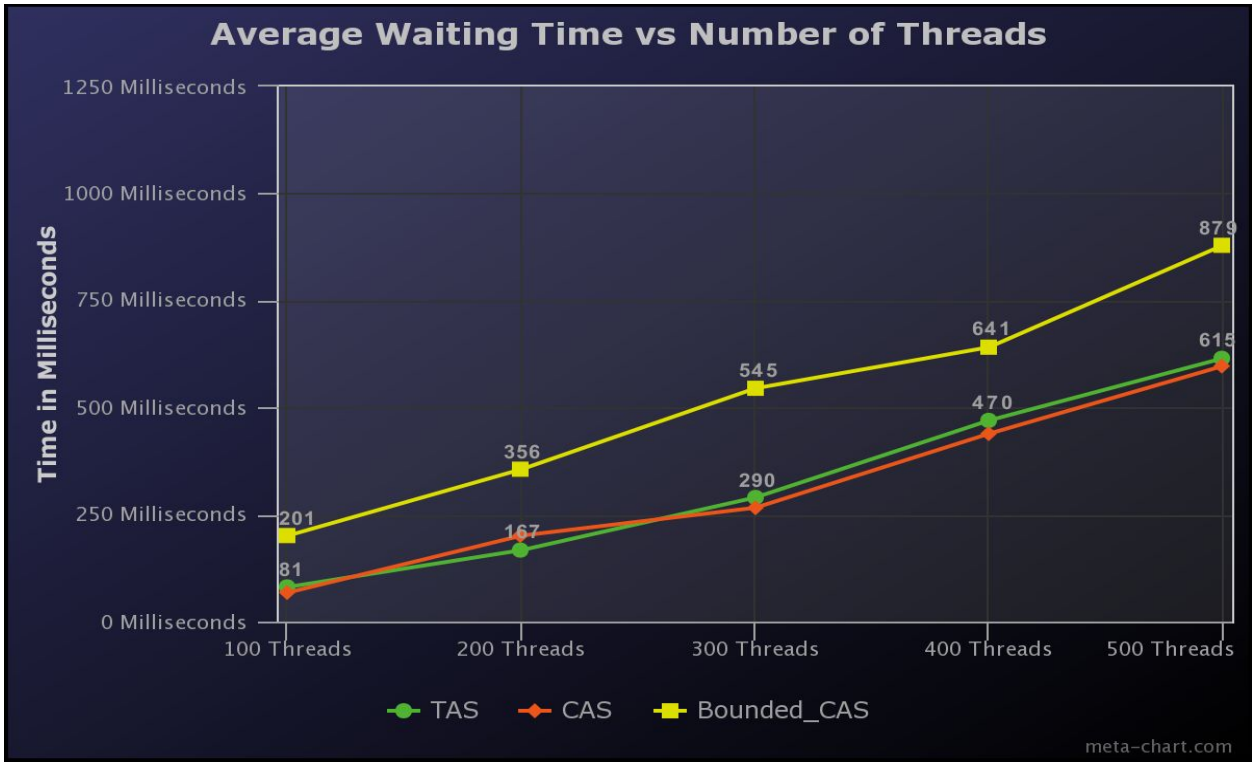
the returned values is True, the while loop breaks and the current thread enters the critical section. To free the lock, it is assigned the value of 0, for its an atomic operation.

3) **Bounded_CAS:** The ***lock*** variable is defined as an ***atomic int.*** Here the value at the index of current_thread_id is set to true in the ***waiting_info*** array to show current thread is ready to enter critical section. The value of thread_key is set to *true.* The thread loops around until waiting_info and thread_key are both true. The thread_key variable is regularly fetched as the output of the ***compare_exchange_strong*** method. As soon as any of the two conditions is violated, the loop breaks, value of waiting_info variable is set to false for the current thread and it enters the critical section. After finish its critical section, the current thread searches for the next thread demanding the critical section. If the value of variable j is not equal to the current thread id, that specific thread's waiting_info is assigned false value, whence it breaks it loop to enter the critical section and enters the critical section. If the value of j is same as the current thread's id, lock variable is set to false. The time at which it leaves the critical section is logged to the log_file.

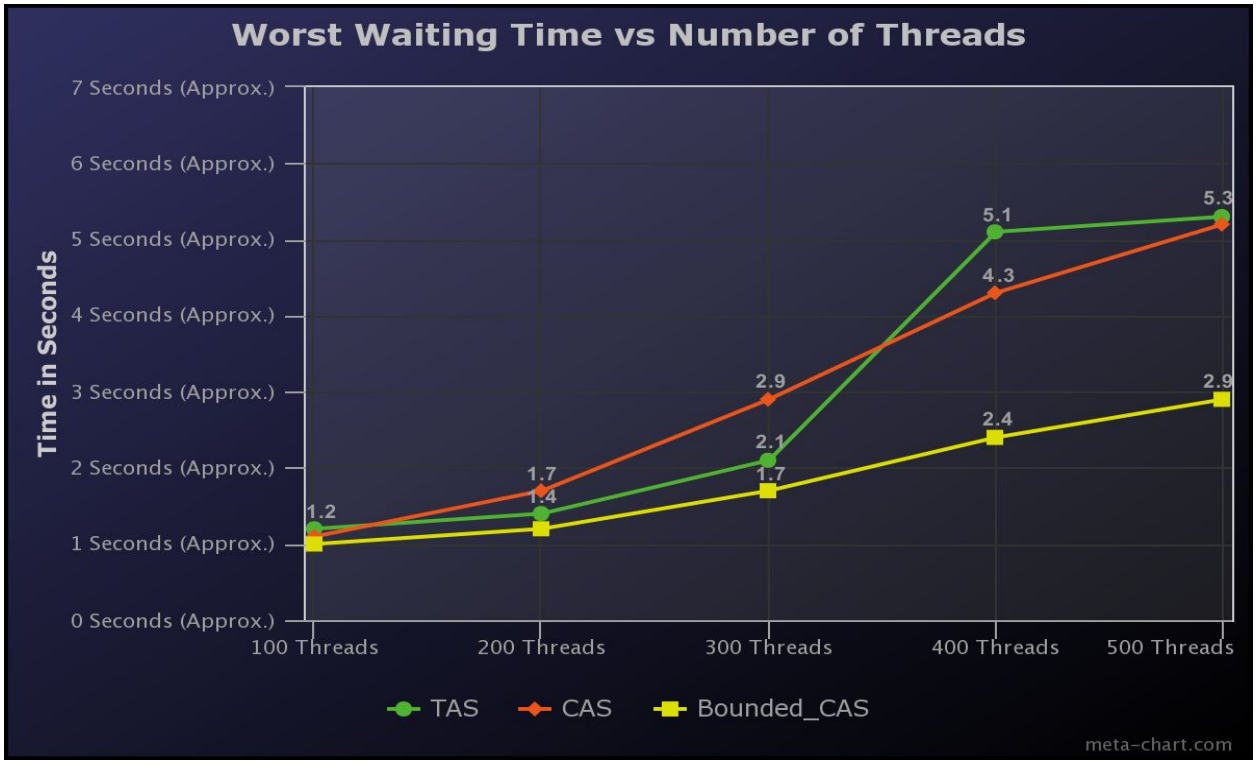
Graphs:

The graphs are plotted for waiting time vs number of threads, with the value of ***times*** variable fixed. The exponential means - ***average_cs_section and average_rm_section*** are fixed at 100. The number of times each thread should run is fixed to 10. Graphs are plotted by averaging over multiple runs.

1) Average Waiting Time:



2) Worst Waiting Time:



We can see that **Bounded_CAS** has higher ***average waiting time*** than **TAS and CAS.** This may subject to multiple reasons - nearly every thread waits for some other, higher complexity of Bounded_CAS which makes it more time consuming, presence of multiple cores, running various other processes alongside these.

However, we see a different scenario for worst case waiting time for a thread. As expected **Bounded_CAS** does much better than **CAS and TAS,** as any thread has to wait for max of ***n-1*** other threads to finish. The difference increases significantly as the number of threads increase.

If we are to ensure more fairness among the executing threads, Bounded_CAS is much preferred. If we are to just minimize average waiting times, CAS or TAS can be used.(However TAS being slightly less complex is preferred for atomic operations can also be done faster.)