REPORT

DESIGN:

Implementation of every algorithm will be the same. After reading the input file, we create the threads and give them each the function 'testCS ()' to implement. Each algorithm's implementation code differs in this 'testCS ().'

FOR TAS:

We use the 'atomic_flag' type to create the 'lock' variable. We used the function 'atomic_flag_test_and_set_explicit(...)' in the while statement to do busy waiting by applying TAS on 'lock.' In the exit section, we used the 'atomic_flag_clear_explicit(...)' function to release the lock. We can observe the mutual exclusion and waiting times in the output file- 'TAS-ME.txt,' where no statement overlaps with other threads' information, which shows only one process is in a critical section at a time.

FOR CAS:

here we created the two functions 'entry_section ()' and 'exit_section ().' These functions will be called at the required positions before and after CS. We use the 'atomic<int>' type to create the 'lock' variable. The function 'entry_section' contains the busy waiting where we implement CAS algorithm using the atomic function '...compare_exchange_weak (....).' And in the function 'exit_section (),' we release the atomic 'lock.' We can observe the mutual exclusion in the output file- 'CAS-ME.txt,' where we see no statement overlaps with the other statement of other thread, which shows only one process is in the critical section at a time.

FOR BOUNDED-CAS:

Here like CAS, we use two functions, 'entry_section (int thread_num)' and 'exit_section (int thread_num).' These functions will be called at the required positions before and after CS. We pass the 'thread_num' to implement the bounded waiting in the functions. Similarly, we use the 'atomic<int>' type to create the 'lock' variable. We even create the array of 'waiting for []' list using the 'atomic<bool>' type. We implement the busy waiting in the function 'entry_section(...)' using the atomic function '...compare_exchange_strong(...)' to implement the CAS algorithm along with bounded-waiting to use the list 'waiting [].' In the function 'exit_section(...),' we implement the bounded-waiting code to select the next thread to go into the critical section. We can observe the mutual exclusion in the output file- 'Bounded-CAS-ME.txt,' where we see no statement overlaps with another statement of another thread, which shows only one process is in a critical section at a time. We can even see the bounded waiting by observing the time.

We can observe that TAS and CAS output looks randomised where different threads are permitted at random. But in Bounded-CAS, the production sequence is unique systematically; we call this bounded-waiting. Here, the permission to go into a critical section is bounded by a function.

Finally, we used the 'exponential_distribution' feature to calculate the waiting time in the critical section and remainder section to create the illusion of necessary heavy work.

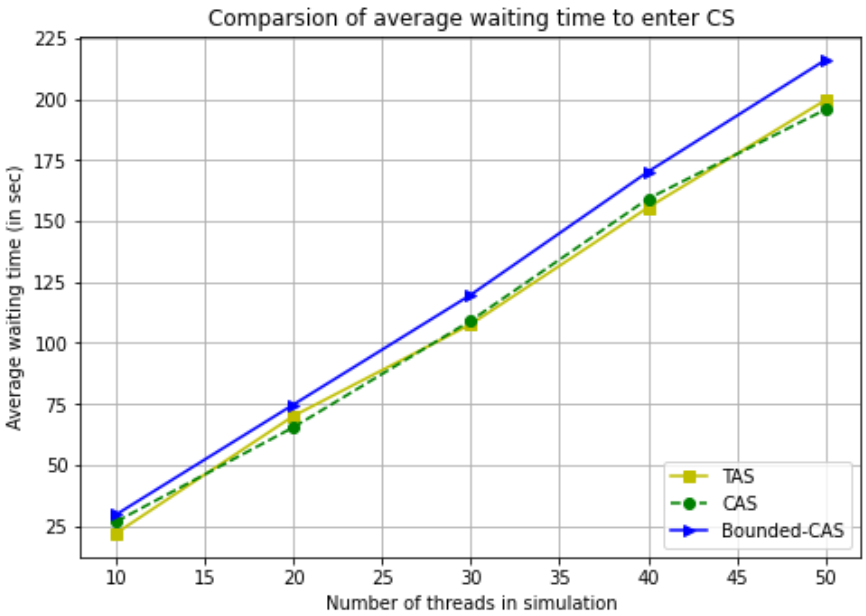
(You can print the waiting time by removing the comments for the statement to publish it in the 'testCS ()' function for easy cross-checking)

PERFORMANCE COMPARISON:

a. The comparison of average waiting time, which is the average of five runs of the testing:

   X-axis: The number of threads varying from 10 to 50 (10, 20, 30, 40, 50)

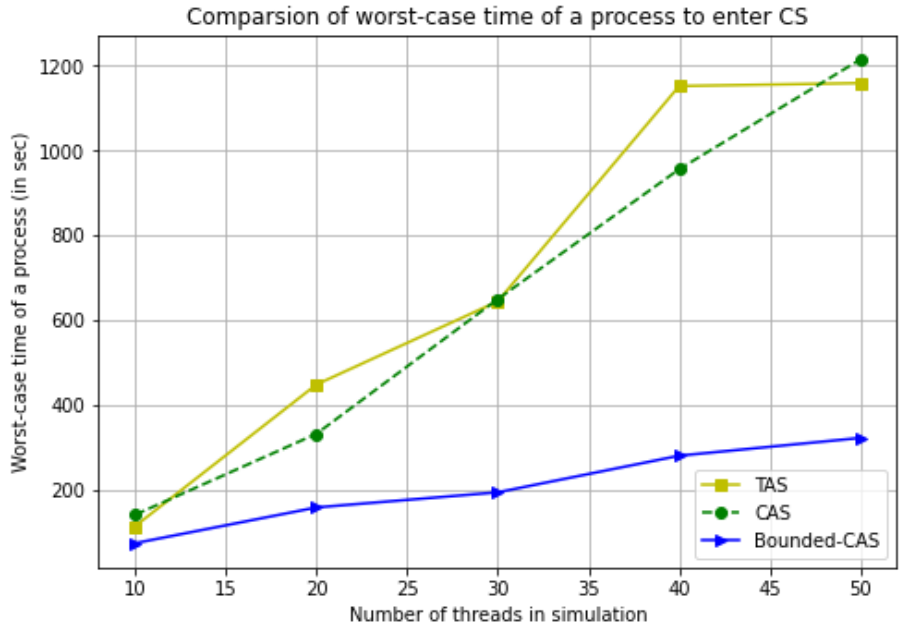
Y-axis: This shows the average waiting time of a thread in one of its iterations to enter its critical section after requesting a critical section corresponding to the number of threads.



b. The comparison of worst-case waiting time of a thread, which is the average of five runs of the testing:

X-axis: the number of threads varying from 10 to 50 (10, 20, 30, 40, 50)

Y-axis: shows the worst-case waiting time of a thread in one of its iterations to enter its critical section after requesting for critical section corresponding to the number of threads.



We used lambda1 = 5; lambda2 = 20 to generate waiting time in the above results.

Analysis of Performance:

Average-Waiting Time:

We can observe that there is not much difference between the average waiting time of the three-algorithm implementation. The waiting time is evenly or unevenly distributed among the processes, giving us an almost identical average.

Conclusion:

We can conclude that average-waiting time is not suitable for comparing TAS, CAS and Bounded-CAS.

Worst-Case Waiting Time:

Here, we can see a substantial difference from Bounded-CAS, and of course, we can even see that TAS CAS has the same worst-case waiting time again. The difference from Bounded-CAS is due to implementing the bounded-waiting feature in Bounded-CAS, which makes the total waiting time evenly distributed and looks fair (Of course not suitable in other aspects).

Conclusion:

Worst-case waiting is an excellent parameter to differentiate Bounded-CAS from TAS and CAS. The almost same waiting time of CAS and TAS is because the implementation environment and parameter used here to compare are not optimum or worst-case conditions.