Design of the code

The programs read input parameters from a file called "inp-params.txt" and write the output to a file called "output.txt". The main function creates a number of threads specified in the input file and runs each thread's testCS function. The testCS function generates random waiting times and repeatedly enters and exits the critical section k times, while measuring the wait time for each entry. The wait times are accumulated and used to compute the average and worst-case wait times for all threads.

Input parameters: The input file contains four parameters: n (the number of threads), k (the number of times each thread enters the critical section), lambda1 (the average execution time between), and lambda2 (the average time between two consecutive critical section requests).

Critical section: The critical section is a simple sleep operation that waits for a random amount of time generated from an exponential distribution with a mean of lambda1. The program measures the time spent waiting for the lock by recording the time before and after spinning on the lock, and calculating the duration between them using the chrono library.

Output: The program writes the output to a file called "output.txt". The output includes the timestamp of each critical section request, entry, and exit, along with the thread ID. At the end of the program, it also prints the average and worst-case wait times for all threads.

The three programs use the following methodologies:

Test-and-set lock: The program uses a TAS lock to enforce mutual exclusion. The lock is an atomic_flag variable initialized to clear state, and each thread spins on the lock until it acquires it by setting the flag to true. When a thread finishes its critical section, it clears the lock by setting the flag to clear state.

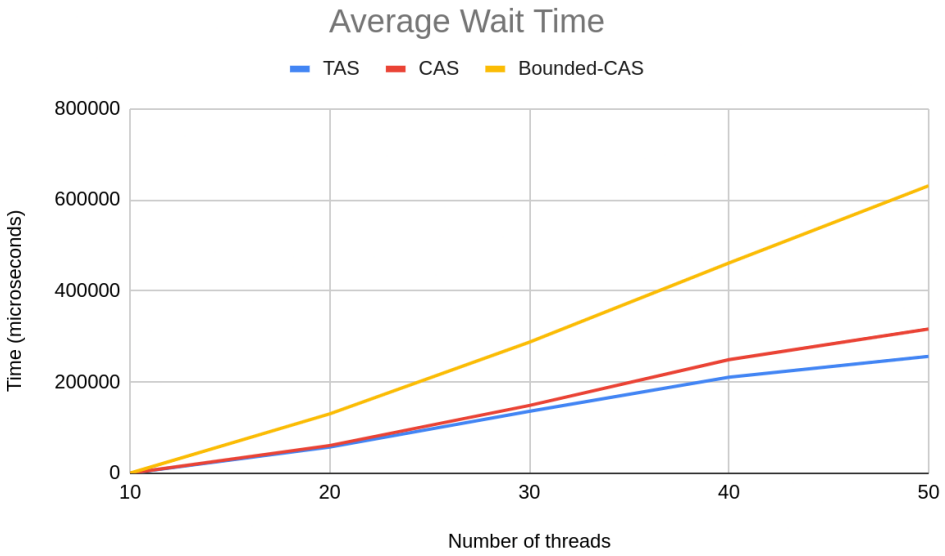
Compare-and-Swap: The program uses an atomic variable lock. Each thread spins on the lock till the lock is 0. If the lock is 0, some waiting thread acquires the lock and sets it to 1. Once the thread finishes its critical section, it frees the lock by setting it to 0. The threads are contending here with each other to acquire the lock.

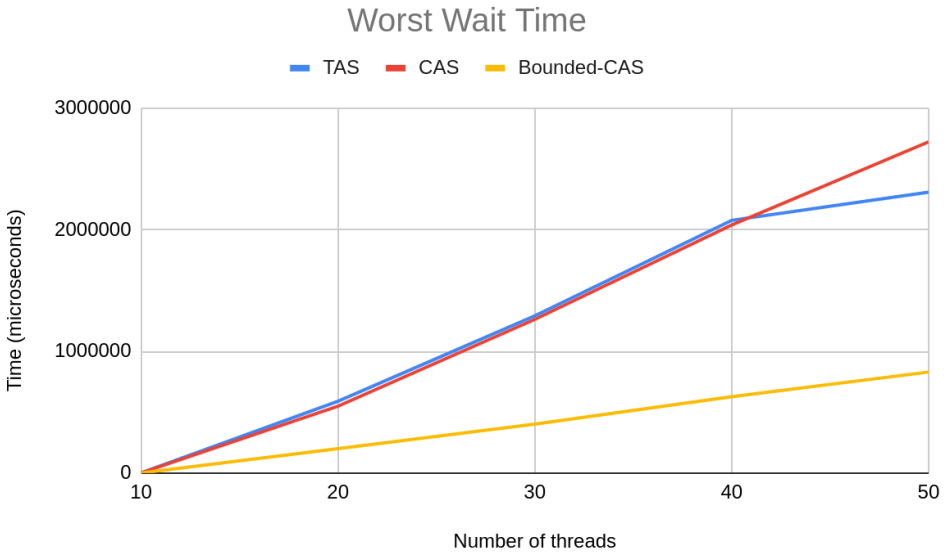
Compare-and-Swap with bounded waiting: The program uses an atomic variable lock. Each thread spins on the lock till the lock is 0. If the lock is 0, the next thread waiting in the waiting queue acquires the lock and sets it to 1. Once the thread finishes its critical

section, it frees the lock by setting it to 0. The threads are numbered from 0 to n-1 and suppose thread i releases the lock, then the thread that acquires the lock is free is the one which comes first in the order i+1, …, n, 0, …, i-1, i.

More code details can be found in the code comments.

Analysis of the results and plots





We can see that as n increases, the average and worst times are increasing. This is as we would expect as the contention for entering into the critical section is increasing and only one thread can be in the critical section at a time.

Test-and-Set has similar performance as Compare-and-Swap. However, Compare-and-Swap with bounded waiting has higher average wait time but shorter worst wait time. This is because Compare-and-Swap with bounded waiting is a fair algorithm. Thus the worst case waiting time is small for each thread and no thread experiences starvations but this leads to a higher average waiting time.