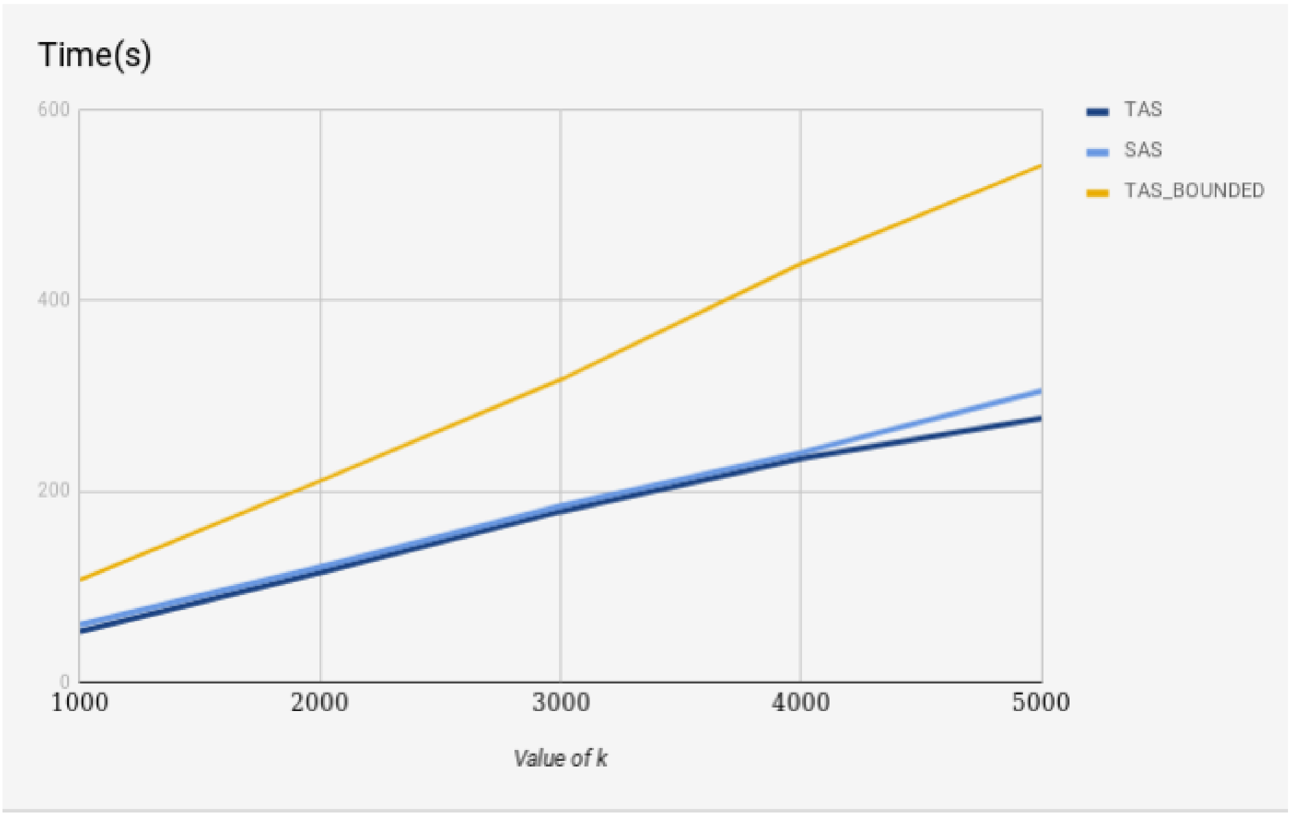
REPORT

Design and Analysis of Test and Swap Program: The proram is written with features of C++11. Thread library is used in the program to implement multithreading. The program is reading inputs from inp-param.txt file. A boolean request array is initialised to keep track of threads who made a request to enter the critical section. Atomic library is used to implement locks which will prevent multiple threads from accessing their critical sections at the same time. The important characteristic of this instruction is that it is executed atomically. Thus, if two test and set() instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order. Mutual exclusion can be implemented by using a variable intialised to false. Atomic flag test and reset is used to check for locks. The Chrono library is used for time calculations.
The thread waits to enter the critical section. After it enters the critical section it spends time there using usleep function. Usleep() gets its parameters from a random number generator. Similarly it spends the time in remaining section.

Design and Analysis of Compare and Swap Program: The program has a similar design as that of previous program. Here we don't need a request array. Locks are implemented using atomic_compare_and_exchange. It prevents multiple threads to execute critical section at the same time. It compares the value of lock(initialied to false) to expected value of false. If both values matches lock becomes true. Time calculations are similar to the previous program. The program differs only in critical section management.

Design and Analysis of Test and Swap with bounded waiting: The design of program is similar to that of Test and Swap. A small portion is added at the end of critical section to implement bounded waiting. A loop checks in a circular way if there is any process which requested access to critical section. If it detects one then the value of it request array index is changed to false. Otherwise is lock is assigned false. The data structures used are-
boolean waiting[n];
boolean lock;
These data structures are initialized to false . To prove that the mutual-exclusion requirement is met, we note that process P i can enter its critical section only if either waiting[i] == false or key == false. The value of key can become false only if the test and set() is executed. The first process to execute the test and set() will find key == false ; all others must wait. The variable waiting[i] can become false only if another process leaves its critical section; only one waiting[i] is set to false , maintaining the mutual-exclusion requirement.
To prove that the progress requirement is met, we note that the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets lock to false or sets waiting[j] to false . Both allow a process that is waiting to enter its critical section to proceed.
To prove that the bounded-waiting requirement is met, we note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i + 1, i + 2, ..., n − 1, 0, ..., i − 1). It designates the first process in this ordering that is in the entry section (waiting[j] == true) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within n − 1 turns.

GRAPH



From the graph it is observed that time taken by Test and Swap is less than the others.

Order of Time Taken: TAS_Bounded >> CAS > TAS

The simple reason for this could be that in TAS_Bounded we have tried to increase the fairness among threads. This has resulted in program becoming more complex and threads waitng unnecessarily. Whereas CAS and TAS are simple and trivial programs. CAS and TAS have took nearly same time.