Operating Systems–1: CS3523 2019 Programming Assignment 2: Implementing TAS, CAS and Bounded Waiting CAS Mutual Exclusion Algorithms Report

Sai Harsha Kottapalli
CS17BTECH11036

February 17, 2019

1 Aim

The goal of this assignment is to implement TAS, CAS and Bounded Waiting with CAS mutual exclusion (ME) algorithms studied in the class. Implement these algorithms in C++.

2 Design of the Program :

We use three types of algo's as mentioned in the aim, to give access to critical section for threads.

2.1 Critical Section :

This part of the code should be accessed by only one thread at a time or as an atomic equivalent. We can avoid race conditions by making sure that no two processes enter their Critical Sections at the same time.

2.2 Entry Section :

This part of the code is accessible for all the threads. The job of this code is to make sure only one thread is given access to the critical section at a particular time. **Note:** starvation might occur in case of TAS and CAS algorithm but in case of CAS-bounded algorithm all threads are given a chance to run fairly often when compared to previous algorithms.

2.3 Exit Section :

This part of the code is where the thread has successfully executed the critical section of the program, and will reset the lock so that other threads can now enter the critical section(if any) while this thread completed the remainder section parallelly.

2.4 Worst Time Taken:

This is defined as the maximum amount of time a thread has to wait after requesting access to the critical section and the access being granted.

2.5 Average Waiting Time :

This is defined as the amount of time each access to critical section takes after requesting for the same.

3 Explanation of program - Common Part

3.1 Header files used

- iostream
  Header that defines the standard input/output stream objects
- thread
  Used for implementing threads.
  It defines the class to represent individual threads of execution.
  A thread of execution is a sequence of instructions that can be executed concurrently with other such sequences in multithreading environments, while sharing a same address space.

- fstream
  Input/output stream class to operate on files.
- random
  This header introduces random number generation facilities.
- atomic
  Atomic types are types that acts like a wrapper around a value whose access is guaranteed to not cause data races and can be used to synchronize memory accesses among different threads.
- cstdio
  Used for **sprintf**, that is, for the format in which the time has to be printed to the log file.
- unistd.h
  Used for -
  localtime - which is used to get a structure of time, with hours, minutes and seconds.
  time - For capturing current time which is processed by above command.
- string
  For processing strings.

3.2 getInput

This is a helper function which helps in keeping the code madular.
It reads the value of the following from the input file, specified in ReadMe -

- number of threads
- number of times each thread requires to access the critical section
- lambda1 and lambda2
  Represents the mean of value of two exponential distribution curves which is to be constructed.

3.3 testCS

The testCS function is responsible for the simulation of the critical, entry, exit and remainder section.We also use this function to calculate the average waiting time and worst time taken which is later used as the parameters for the comparisions for the three different algorithms mentioned in the aim.

3.4 currTime

Takes input time_t structure to generate the time in HH:MM:SS syntax and return this as a string.

3.5 Other important variables used

- exponential_distribution
  Produces random non-negative floating-point values x, distributed according to probability density function defined for exponential distribution about the constant rate given as a parameter.

- default_random_engine
  This is a random number engine class that generates pseudo-random numbers.

- waiting
  Dynamic array of bool type, based on number of threads given in input file.

- thread *th
  th is the pointer for storing the array of n threads.This will be useful for later calling join to properly exit the threads spawned previously.

4 TAS

- lock
  We use atomic⟨bool⟩ and initiate this to **ATOMIC_FLAG_INIT**.

- exchange
  syntax - exchange(T_desired, std::memory_order order = std::memory_order_seq_cst)

Atomically replaces the underlying value with desired. The operation is read-modify-write operation. Memory is affected according to the value of order.
We use this to implement our TAS algorithm.

- return value
  The value of the atomic variable before the call.

5 CAS and CAS-Bounded

- lock
  We use atomic⟨int⟩ and initiate this to **0**.
  Note: use lock(0) instead of lock = 0, as there are some issues with using the latter.

- compare_exchange_strong
  syntax - compare_exchange_strong(T& expected, T_desired)
  Atomically compares the object representation of *this with that of expected, and if those are bitwise-equal, replaces the former with desired (performs read-modify-write operation). Otherwise, loads the actual value stored in *this into expected (performs load operation).

- return value
  true if the underlying atomic value was successfully changed, **false** otherwise.

- **Note:** expected must always be reset to 0 after every iteration and the above command also change expected when required conditions are met.

6 Comparision

6.1 Graphs

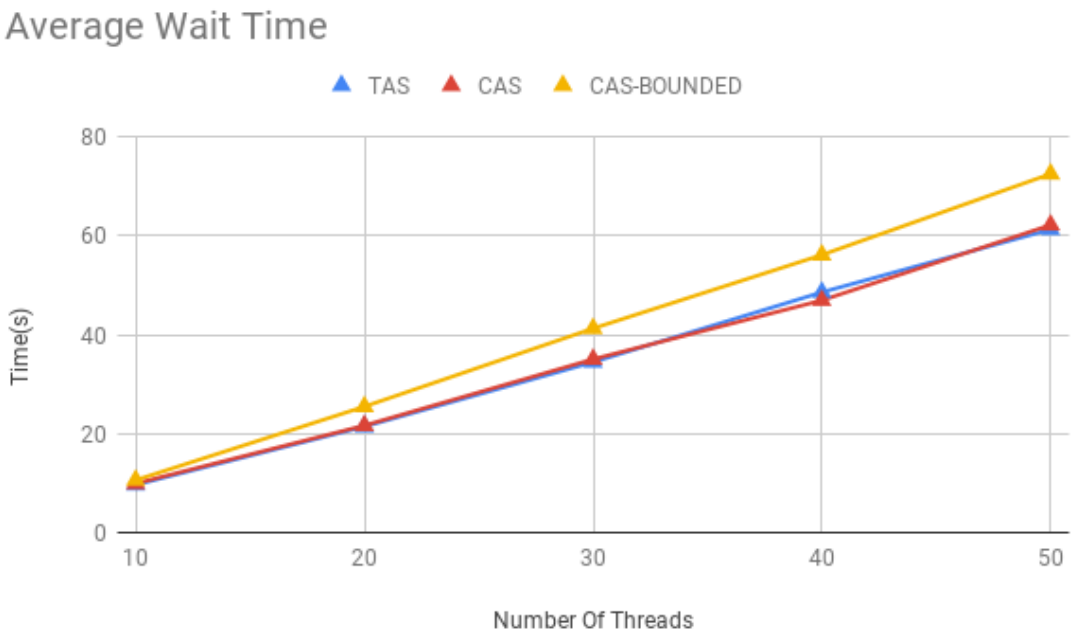

Figure 1: Comparision between Average time taken for accessing the critical section by the algorithms.

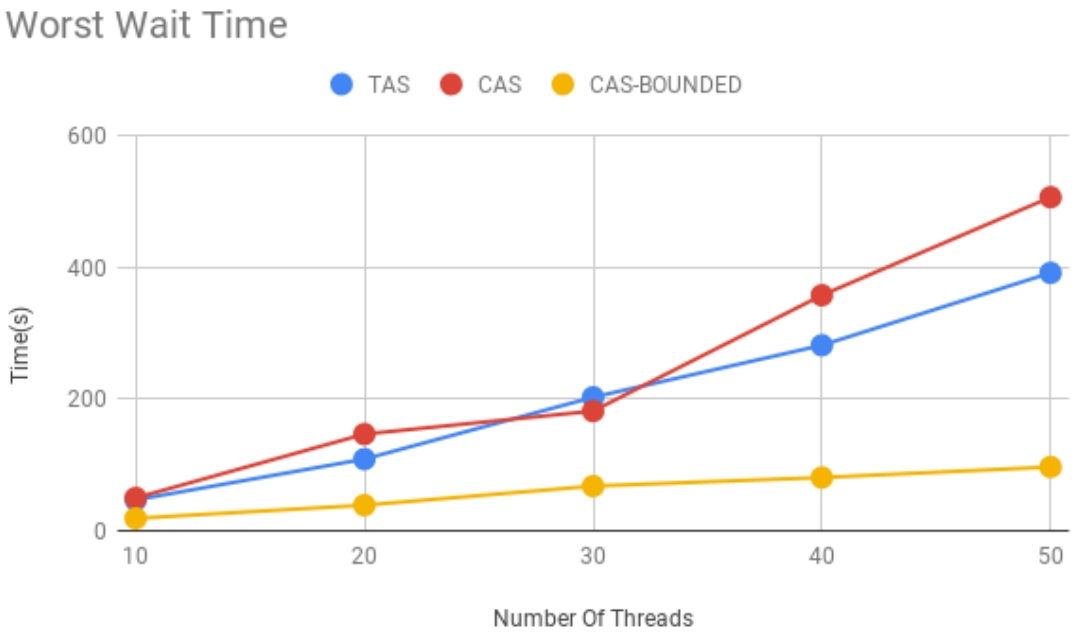

Figure 2: Comparision between the worst time taken by the algorithms.

6.2 Notes

- Parameters used for test cases:
  n = 10, 20, 30, 40 , 50
  k = 10
  $\Lambda_1 = 2$ seconds $\Lambda_2 = 2$ seconds
- Each of the plotted points is the average of 5 testcases.
- With respect to Average waiting time Efficiency of algorithms:
  TAS $\approx$ CAS $>$ CAS-bounded.
- With respect to Worst time taken Efficiency of algorithms:
  CAS-Bounded $>$ TAS $>$ CAS