CS5300: PARALLEL AND CONCURRENT PROGRAMMING, Fall 2020

Programming Assignment 2: Implementing Filter and Peterson based Tree Lock Algorithms

SAI HARSHA KOTTAPALLI

CS17BTECH11036

GOAL

The goal of this assignment is to implement Filter and Peterson-based tree lock algorithms.

DESIGN

COMMON PART

We parse input-params.txt for input to obtain the number of threads(n) and the number of CS accesses(k) by each thread. We also take the averages of exponential distributions which will be used for simulation of some work being done inside and outside CS. We then create n threads, each responsible for doing some work inside CS and outside, k times.

For maintaining Mutual Exclusion between threads accessing CS, we use Locks, namely, Filter lock and Peterson Based Tree lock, whose design will be explained below.

Here, we call lock() when a thread wants to enter the CS. When a thread attains the lock successfully, it will be the only thread inside CS at that time until it completes its task in CS, upon which it calls unlock().

The logs stored as the program executes give an idea of how the execution proceeds while holding the mutual exclusion property.

FILTER LOCK

Filter lock is a direct generalization of Peterson lock for n threads where we use n-1 waiting rooms/ levels where threads are filtered until only one thread can access the CS at any time.

So in the algorithm, levels are the waiting rooms where at each level when threads want to access CS, at least one of the threads enters and at least one of the many threads at that particular level(iff many) is blocked. As there are n-1 levels, at any point only one thread can access the CS while the rest wait in the levels. Victim[] is to keep track of where the thread is blocked.

LOCK()

When a thread wants to attain the lock, it traverses all the levels one by one. At any level, it first stores the current level it is in via *levels[]* and like Peterson, gives priority to any other thread which wants to enter CS first via *victim[]*. This is made sure via a spinning lock which spins as long as there is another thread at the current or any higher level while this thread invoking is still the victim.

Finally, at some point, all higher threads would complete their Task in CS and this spin loop is broken, upon which this thread can enter the CS.

For Peterson two-thread lock, instead of identifying which thread acts as thread0 or thread1 at a node, we can instead create an array of size n and use only the thread IDs itself as indexes to act as out boolean[2] flag.

UNLOCK()

When a thread exits CS and calls unlock, we simply reset the level[] it is in. This is so that the next potential candidate breaks out of its spin loop and eventually reaches the CS.

PETERSON BASED TREE LOCK

This is also a generalization of the Peterson two-thread locking mechanism arranging a number of these Peterson locks in a tree. Here, we assume that n is a power of 2. Each thread shares the leaflock with one other thread. On any request for CS, the thread has to lock all the way till the root. Therefore, there will be n/2 leaves in the binary tree. If we notice carefully, there will be a total of n-1 nodes in the tree, hence we can argue that this setup is enough for ME to hold. Since n is a power of two and we construct a binary tree with n/2 leaves, it will be a full binary tree where each node in the tree signifies a Peterson two-thread lock. At every node, only one thread can proceed to upper levels and finally, only one thread will attain the lock for root and proceed to CS.

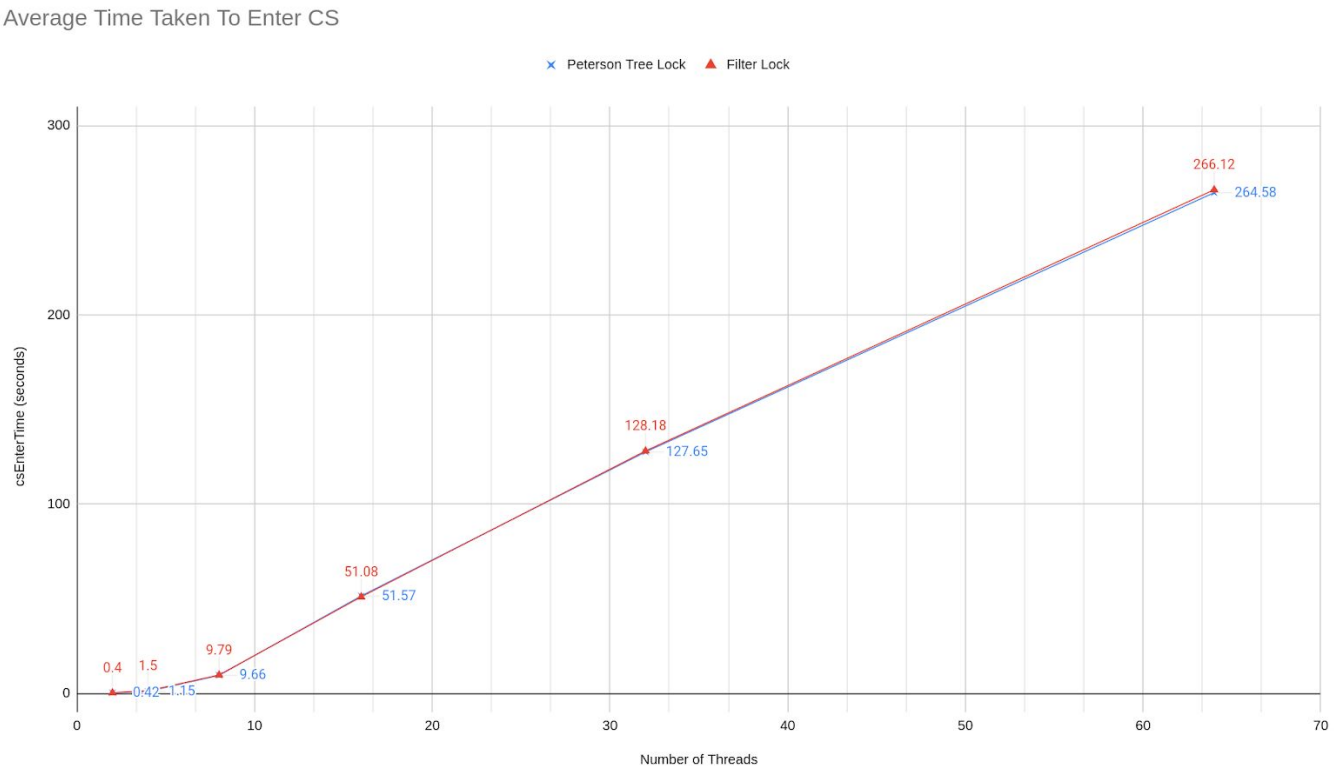
LOCK()

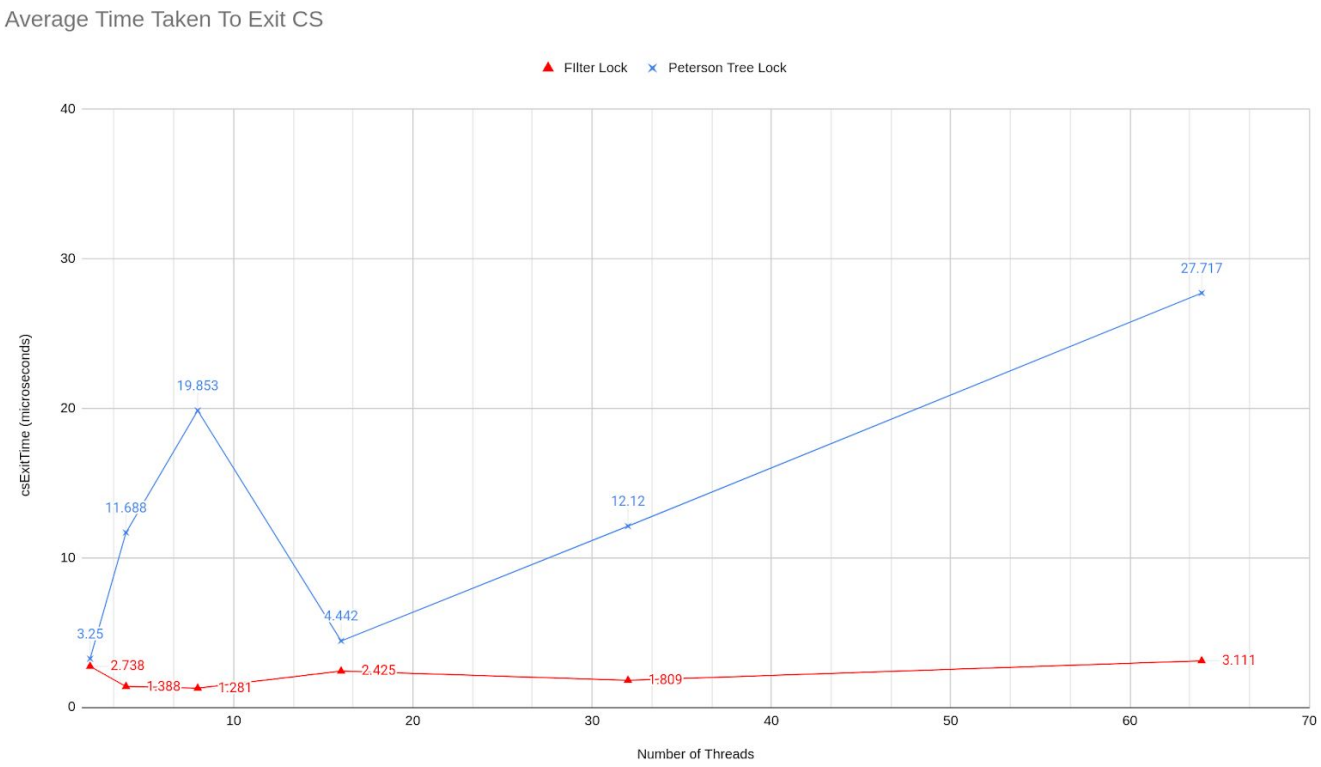
When a thread wants to attain lock, it will try to get all the lock from its respective leaflock till the root. In doing so, as discussed above, only one thread can eventually get access to CS as every other thread is blocked at every node.

UNLOCK()

When a thread finishes its task in CS and wants to exit, it can call unlock(). During which, the thread will unlock all nodes in the path starting from root to its leaflock. This lets any other threads which are stuck in any of the nodes continue to next lock and eventually the CS.

GRAPHS


Average Time Taken To Enter CS
Peterson Tree Lock
Filter Lock
csEnterTime (seconds)
Number of Threads
300
200
100
0
0
10
20
30
40
50
60
70
266.12
264.58
128.18
127.65
51.08
51.57
9.79
9.66
0.4
1.5
0.42
1.15



OBSERVATIONS

- I have run multiple times (5) and taken the average of the required data to plot the graph. The basic inference is that csEnterTime is almost the same for PTL and Filter lock, but it seems that PTL is marginally better.
- Unlock as we can see is in the order of microseconds, though it's clear that Filter lock is much faster than PTL, this would hardly matter as lock() is in the order of seconds and any advantage from this wouldn't make any significant impact. The reason why unlock() is much faster is that we are just zeroing values and essentially not waiting for any other condition.
- In case of PTL, it has to traverse the binary tree, so it takes slightly more time than Filter lock. So, unlock() for Filter is O(1) while for PTL it is O(log(n)).

- For PTL, there is a bump at n=8,
- In the simulation, sleep() does not accurately help in the simulation as thread just sleeps in this case, unlike a realistic scenario where the current thread is doing some work and the scheduler might decide to pause it for a while or similar situations which are unaccounted for. But since we are taking the time in order of seconds, it shouldn't make much difference anyway.