PROGRAMMING ASSIGNMENT - II

Sri Hari Malla - CS19BTECH11039

February 11, 2022

1 Design:

1.1 Goal:

- Implement filter lock and peterson's lock.
- Simulate Critical section using the locks developed.
- Mimic critical section using random exponential distribution.

1.2 Design:

1.2.1 Recording Time Stamp:

- We need to record time stamps every time when we request,enter and exit.
- We can use C-style `localtime(tm*)` to capture the present time.
- We can subtract two `tm*` pointers to get the actual time gap between two time stamps.

1.2.2 Threading:

To achieve the parallelism, we need to use threads.

- As `C++` thread library is a wrapper implementation of threads available on various architectures and is platform independent,`std::thread` could be used instead of `pthread` as we've scope to produce some high level code.
- For dealing with race conditions we could use `std::mutex` to acquire and release locks in threads.

1.2.3 Locks:

Let's create a pure abstract class which contains the required standard methods which should be present in a lock class.

```
class locker{
public:
    atomic<double> entry_waiting_times;
    atomic<double> exit_waiting_times;
    virtual void lock(int) = 0;
    virtual void unlock(int) = 0;
    virtual string get_method_name() = 0;
};
```

It contains:

- two atomic type variables for waiting times
- lock method
- unlock method
- method which gives the class name.

The following two locks which will be implemented derives from this class.

1.3 Filter Lock:

The filter lock as discussed in the class and in the text book maintains `n-1` waiting rooms which we address as levels for n number of threads. What happens in this lock method is, we maintain `n` levels and victims. First we initialize all the levels to 0. Then when a thread calls a lock method, it progress to a new level updating the levels and victims. When the next thread calls the same lock method, as the victims keep track of the thread already progressed, the new thread stops at a certain level where it is supposed to. The unlock method simply removes the level status which allows other thread to progress. Thus all threads progress to the final level accordingly and maintain proper mutual exclusion and deadlock free. As it was thoroughly discussed, this was explained in brief.

1.4 Peterson's Lock:

The lock is developed based on a binary tree.

- If n threads are set to enter the critical section, let's create `2*n-1` flags and `n` victims.

- The method is adopted from two thread peterson lock. In which we have 1 boolean variable for two threads.
- We'll have binary `log(n)` levels for n threads.
- If a thread calls the lock method, what actually happens is it progresses to the parent level and update the victim.
- It progress to the parent level by actually competing with the complimentary thread which can be achieved by using the busy wait loop.
- At every level, the thread competes with the complimentary parent corresponding to the other thread which is trying to acquire the lock.
- Ultimately one among the n threads reaches the top level and acquire the lock.
- This method actually is similar to the binary tree as we travel from leaf to root for locking.
- Now comes the unlock method which again needs to iterate like the binary tree.
- As we went up locking the levels at particular places, we need to unlock the levels.
- Iterate down to the thread's original number down and clear the levels such that other threads can acquire lock one by one from root to leaf.

1.5 Exponential Mimic:

We need to mimic the critical section using sleep with exponentially random distributed time seed.

- we can use `uniform_real_distribution` for creating a uniform real distribution object.
- we can use `random_device` for creating a random_engine.
- Seeding the random engine in distribution gives us the random time.

2 Observations and Report:

Graph1 : Change in n-threads vs Time (constant repetition set to 10)

As instructed, this is the graph plotted with points observed while running the program with changing the Number of threads and calculating the average time whilst the repetition is set to 10.

- One can clearly see that the Filter method is outperforming Petersons method.

- It is also seen that from 1-10 threads, both the methods are taking almost the same time. But its only visual perception that is because of the scaling of the entire picture.

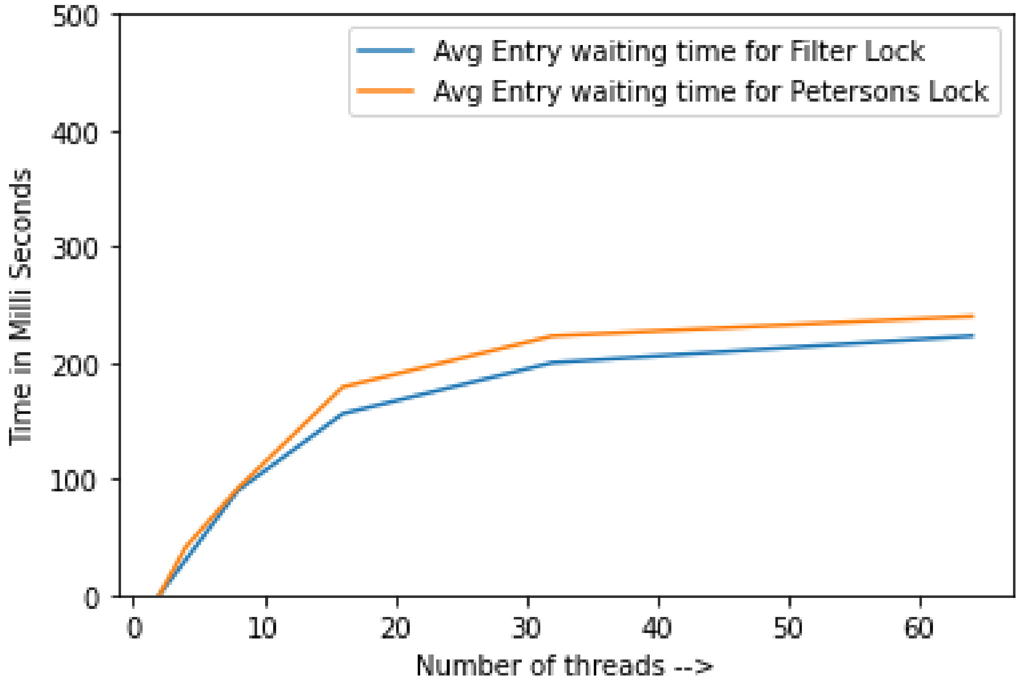

Figure 1: Change in threads vs Average entry waiting Time taken

- It is observed that petersons method is taking more time than filter lock.
- The reason might be the complexity of the petersons lock when compared to the filter lock.
- As the petersons method touches all the parents and a busy loop, when compared to filters method which touches levels,victims and busy loop proves to be some what expensive in terms of computation and time taken.
- With increase in threads, the number of threads which fights for the critical section increases, thus the rotation must give increase in average waiting time which is clearly observed in the graph.

Graph2 : Change in n-threads vs Time (constant repetition set to 8)

As instructed, this is the graph plotted while running the program with changing the number of threads and recorded average wait time taken.

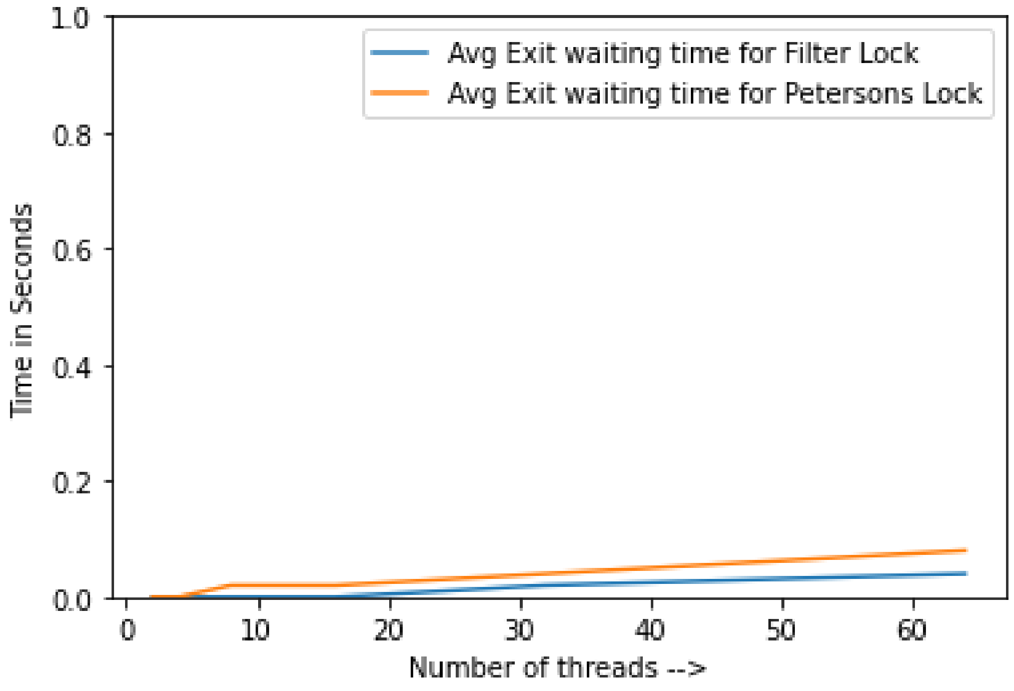

Figure 2: Change in threads vs Average exit waiting Time taken

- One can see clearly that once again the filter lock method is outperforming petersons lock.
- Clearly the petersons's unlock method is more complex and one might encounter it while coding.
- The complexity of the petersons lock increases the actual waiting time for exit.
- This can be clearly observed in the graph that petersons method is taking actually more time than filter method.
- But the average time recorded is so close to 0.

Note: The observations recorded are the average of the observations which were run over 50 times using a shell script to avoid any type of inference and uncertainty from one run.

LaTeX generated document

THE END