CS5300: Parallel and Concurrent Programming
Theory Assignment 1

Abburi Venkata Sai Mahesh - CS18BTECH11001

October 1, 2020

This document is generated by LaTeX

Q1 **1. Suppose a computer program has a method M that cannot be parallelized, and that this method accounts for 40% of the program's execution time. What is the limit for the overall speedup that can be achieved by running the program on an n-processor multiprocessor machine?**

A. Given that Sequential Execution (1-p) = 0.4
⇒ Parallel Execution (p) = 0.6
From Amdhal's Law,

$$Speedup(S) = \frac{1}{1 - p + \frac{p}{n}}$$

Limit can be applied when $n \to \infty$

$$\Rightarrow S = \lim_{n\to\infty} \frac{1}{1 - p + \frac{p}{n}} = \frac{1}{1-p} = \frac{1}{1-0.6} = \frac{1}{0.4}$$

$$\boxed{\therefore S = 2.5}$$

2. Suppose the method M accounts for 30% of the program's computation time. What should be the speedup of M so that the overall execution time improves by a factor of 2?

A. Given that Initial Sequential part = 0.3
⇒ Parallel part = 0.7
Let the speedup required for M = k

$$s_n^{'} = 2 \times s_n$$

$$\left(\frac{1}{\frac{0.3}{k} + \frac{0.7}{n}}\right) = 2 \times \left(\frac{1}{0.3 + \frac{0.7}{n}}\right)$$

$$0.3 + \frac{0.7}{n} = \frac{0.6}{k} + \frac{1.4}{n}$$

$$0.3 - \frac{0.7}{n} = \frac{0.6}{k}$$

$$\frac{k}{0.6} = \frac{1}{0.3 - \frac{0.7}{n}}$$

$$\boxed{\therefore k = \frac{6n}{3n-7}}$$

3. **Suppose the method M can be sped up three-fold. What fraction of the overall execution time must M account for in order to double the overall speedup of the program?**

A. Let the sequential part = s

$$\frac{1}{\frac{s}{3}+\frac{1-s}{n}} = 2 \times \left(\frac{1}{s+\frac{1-s}{n}} \right)$$

$$s+\frac{1-s}{n} = \frac{2s}{3} + 2 \times \frac{1-s}{n}$$

$$\frac{s}{3} = \frac{1-s}{n}$$

$$s \times \left(\frac{1}{3}+\frac{1}{n}\right) = \frac{1}{n}$$

$$\boxed{\therefore s = \frac{3}{n+3}}$$

Q2 For Peterson Tree Lock, for each property, either sketch a proof that it holds, or describe a (possibly infinite) execution where it is violated.

1. **Mutual Exclusion**
2. **freedom from deadlock**
3. **freedom from starvation**

A. The Peterson Tree lock is implemented by considering the threads as leaves of the binary tree and every time passing half the number of threads in the present level to enter the next upper level by using the basic 2-thread Peterson Locks for each two threads.
After a series of execution the only thread present on the top level(root node thread) enters into the critical section.
It was already proved in the textbook that the 2-thread Peterson lock satisfies Mutual Exclusion, free from starvation and free from deadlock.

Mutual Exclusion:
Lets us assume that PTL doesn't satisfies Mutual Exclusion. This implies that at least two threads enter into the critical section at the same time. It can be possible when:

1. Root lock has allowed both threads to enter Critical Section.
   This implies that the root lock is not perfectly implemented and is not satisfying mutual Exclusion. This leads to a contradiction to our already proven Lemma that a 2-thread Peterson Lock(inferring to the root lock here) should satisfy mutual exclusion. So this case cannot happen.
2. More than two threads try to acquire the root lock
   This can happen when the at least one lock present in the lower levels has allowed both the threads to acquire the lock. As all the locks are simple 2-thread Peterson lock and should satisfy mutual exclusion, this situation cannot be possible.

As all the expected exceptional cases are not possible to happen with our implementation for PTL, we can declare that it satisfies mutual exclusion.

Freedom from Deadlock:
In PTL implementation, all threads are implemented in bottom up fashion (from leaf node to root node). So a thread in present level will only wait for at most one thread(thread adjacent to it in same level) to acquire the lock of next higher level. This is similar to the check for the deadlock for a single 2-thread Peterson lock. As we already know that 2-thread Peterson lock is free from deadlock, it is guaranteed that PTL is free from deadlock.

Freedom from Starvation:
We have a result that a 2-thread Peterson Lock is starvation free. Lets start from the bottom level of the tree. The lock present in the next upper level is starvation free. It means that the every thread present in the bottom level will surely acquire the lock even after sometime(as the new thread requesting for lock will give turn to the older thread). While the level passes ahead each thread will surely acquire the lock as described above and eventually succeed to enter critical section. So no thread will get starved and the PTL is starvation free.

Q3 In practice, almost all lock acquisitions are uncontended, so the most practical measure of a lock's performance is the number of steps needed for a thread to acquire a lock when no other thread is concurrently trying to acquire the lock. Scientists at Cantaloupe-Melon University have devised the following "wrapper" for an arbitrary lock, shown in Fig. 2.16. They claim that if the base Lock class provides mutual exclusion and is starvation-free, so does the FastPath lock, but it can be acquired in a constant number of steps in the absence of contention. Sketch an argument why they are right, or give a counterexample.

```
1   class FastPath implements Lock {
2       private static ThreadLocal<Integer> myIndex;
3       private Lock lock;
4       private int x, y = -1;
5       public void lock() {
6           int i = myIndex.get();
7           x = i;                          // I'm here
8           while (y != -1) {}               // is the lock free?
9           y = i;                          // me again?
10          if (x != i)                      // Am I still here?
11              lock.lock();                // slow path
12      }
13      public void unlock() {
14          y = -1;
15          lock.unlock();
16      }
17  }
```

Figure 2.16 Fast path mutual exclusion algorithm

A. Given claim is that if the base Lock class provides mutual exclusion and is starvation-free, so does the FastPath lock.

Mutual Exclusion:
For the mutual Exclusion to be satisfied, each thread should wait until the other thread(which acquired lock) should unlock it(calling the unlock function).
Let us assume a simple scenario of two threads(id0, id1) trying to acquire the FastPath lock and no other thread has acquired Base Lock till now.
Now consider the execution order as below.

1	int i = id0;	// starting with thread0
2	x = id0;	// thread0 executing
3	while(y!=-1)	// loop exits as initially y == -1 for thread0
4	int i = id1;	// context switch to thread1 before updating y
5	x = id1;	// x value is re-writed by thread1
6	while(y!=-1)	// loop exits as y is not updated before for thread1
7	y = id0;	// context switch to thread0
8	if(x!=i)	// satisfies and below statement is executed by thread0
9	lock.lock()	// thread0 requesting base lock
10	y = id1;	// context switch to thread1
11	if(x!=i)	// not satisfies for thread1

We can observe that only thread0 is requesting for base lock and thread1 is directly exiting from the function. The acquiring of base lock by thread0 in step 9 completes in no time since there no other request available(so no waiting). For the above execution sequence both the threads can exit the FastPath lock function and enter into the critical section at the same time(that is thread0 can exit the fastlock function before thread1 can call unlock function). This is because only one of the two threads are able to call the base lock function while the other can simply exit it(thread1 is not calling base lock function). Therefore it does not satisfy mutual exclusion principle.

Starvation Free:
For a thread performing lock function there is no chance to provide the turn for other thread. So a thread can execute for infinite times with not giving chance to other thread.So there is a chance for starvation and the algorithm is not starvation free.

So we can say that the claim provided is completely false.