CS5300 Theory Assignment 1

Gautam Singh
CS21BTECH11018

1) Amdahl's Law states that the speedup resulting from an N-processor machine executing a program with p fraction of parallel code compared to a single processor machine is $\frac{1}{1-p+\frac{p}{N}}$.

   a) The maximum speedup can be achieved when the remaining $1-M = 0.7$ part of the code is parallelizable. Thus, the speedup on an n-processor machine is

   $$s = \frac{1}{0.3 + \frac{0.7}{n}} = \frac{n}{0.7 + 0.3n} \tag{1}$$

   As $n \to \infty$, $s \to \frac{1}{0.3} \approx 3.33$, thus the limit of speedup is 3.33.

   b) Assuming that the rest of the code is parallelizable, the initial speedup on the n-processor machine is

   $$s = \frac{1}{0.4 + \frac{0.6}{n}} \tag{2}$$

   Using M' instead of M gives us a k-fold speedup, so the new-speedup is

   $$s' = \frac{1}{\frac{0.4}{k} + \frac{1-\frac{0.4}{k}}{n}} = 2s \tag{3}$$

   Thus,

   $$\frac{1}{\frac{0.4}{k} + \frac{1-\frac{0.4}{k}}{n}} = \frac{2}{0.4 + \frac{0.6}{n}} \tag{4}$$
   $$\implies \frac{0.4}{k} + \frac{1-\frac{0.4}{k}}{n} = 0.2 + \frac{0.3}{n} \tag{5}$$
   $$\implies \frac{0.4}{k}\left(1 - \frac{1}{n}\right) = 0.2 - \frac{0.7}{n} \tag{6}$$
   $$\implies k = \frac{4n-4}{2n-7} \tag{7}$$

   c) Suppose that M accounts for x fraction of execution time. Then, the initial speedup is

   $$s = \frac{1}{x + \frac{1-x}{n}} \tag{8}$$

   Now, M' executes in $\frac{x}{4}$ fraction of execution time. Thus, the new speedup turns out to be

   $$s' = \frac{1}{\frac{x}{4} + \frac{1-\frac{x}{4}}{n}} = 2s \tag{9}$$

   Thus,

   $$\frac{x}{4} + \frac{1-\frac{x}{4}}{n} = \frac{x}{2} + \frac{1-x}{2n} \tag{10}$$
   $$\implies \frac{x}{4} = \frac{1+\frac{x}{2}}{2n} \tag{11}$$
   $$\implies x = \frac{2}{n-1} \tag{12}$$

   Hence, M must account for $\frac{2}{n-1}$ fraction of the overall execution time for M' to double the program's speedup.

2) The only possibility of deadlock arises when the two threads, say A and B are waiting indefinitely in the `while` loops of either the `lock` or `unlock` method. Let's analyze these cases one-by-one.

   a) Suppose A and B are both waiting in the `lock` method. We know from the original Peterson's algorithm, that `victim` will be set to either A or B, thus the other thread will be released.
   b) Suppose A and B are both waiting in the `unlock` method. Without loss of generality, assume A executes line 5 of the `unlock` method first. Then, when B sets `flag`$[B] = false$ later on, A will be released. Otherwise, if A is slow, B will be released as `flag`$[A] = false$ already.
   c) Suppose A is waiting in the `lock` method and B is waiting in the unlock method. Then, we must have `flag`$[B] = true$. Thus, B is yet to set `flag`$[B] = false$ in line 2 of the `unlock` method. When it does so, A will be released.

Thus, this variant of Peterson's lock is deadlock-free since deadlock is impossible.
Consider the following execution of two threads A and B.

   a) Initially, the critical section is empty.
   b) Thread A acquires the lock and enters the critical section. Then, it invokes the `unlock` method and sets `flag`$[A] = false$.
   c) After A sets `flag`$[A] = false$, thread B acquires the lock and enters the critical section, after which it invokes the `unlock`

method. By this time, A is waiting at line 5.

d) Now, thread A gets swapped out, so thread B overtakes A and is released from the `unlock` method since $\texttt{flag}[A] = false$.

e) Thread B can re-enter the critical section and overtake thread A an arbitrary number of times as described above, provided thread A sees that $\texttt{flag}[B] = true$ whenever it is executing the check in line 5 of the `unlock` method.

Thus, thread A can wait for an indefinite amount of time, causing it to starve. Hence, this variant of Peterson's lock is *not* starvation-free.

3) Consider threads A, B and C operating on a Herlihy-Wing queue concurrently.
Here is an execution that shows line 15 cannot be the linearization point of `enq`. Suppose that the queue is empty initially.

a) A invokes `enq`.
b) B invokes `enq`.
c) A executes line 15 first, and increments `tail` first.
d) B executes line 15 after A, and then executes line 16 before A, which becomes slow.
e) C invokes `deq` and finds that the first non-null item is the one enqueued by B, since A has not executed line 16 yet. Thus, C returns the item enqueued by B.

This execution shows that even though A executes line 15 before B, the item enqueued by B is dequeued first by C. Thus, line 15 cannot be the linearization point of `enq`.
Here is an execution that shows line 16 cannot be the linearization point of `enq`. Suppose the queue is empty initially.

a) A invokes `enq`.
b) B invokes `enq`.
c) B executes line 15 first, and increments `tail` first.
d) A executes line 15 after A, and then executes line 16 before B, which becomes slow.
e) B executes line 16 after A.
f) After both A and B return from `enq`, C invokes `deq` and finds that the first non-null item is the one enqueued by A. Thus, C returns the item enqueued by A.

This execution shows that even though A executes line 16 before B, the item enqueued by B is dequeued first by C since B got the earlier slot in the queue. Thus, line 16 cannot be the linearization point of `enq`.
This does *not* mean that `enq` is not linearlizable. It means that the linearization point of `enq` can be chosen depending on the execution history to make the history linearlizable.