Andrew McCarthy

Parallel Programming Theory Problems

2. 1. Safety because it prevents a patron from not being served
2. liveness because it insists that something must come down
3. liveness because is confirms that at least one process will succeed
4. liveness because a message will be printed
5. Also liveness because a message is printed
6. Safety because the cost of living will never decrease
7. liveness: death & taxes will always eventually come
8. Safety because you will never not be able to tell if someone is a harvard man.

3. The feeder and supplier both have cans, if the supplier can is down, then the feeder wants to supply more food. When the supplier supplys the food she stands his can up and pulls his string to knock over the feeder's can.

The feeder protocol works as such: if the feeders can is knocked over, then she can let her pets out. When the food is all gone, the feeder takes the pet inside. Once the pets are inside, the feeder fixes her can & pulls the string to knock over the suppliers can

4. winning strategy 1: The prisoners first pick a designated counter/declarer. The counter will always switch it to the on position. Everyone else will switch it to off but only once. Once the counter has switched it on n times, he can declare.

strategy 2: Same counter as before, but now everyone must switch it off two times (no more no less). Once the counter has switched it on $2P-1$ times, he can declare.

5. The person in the very back will call out red if he sees an odd number of red hats & blue if he sees an even number of red hats. This prisoner has a 50% chance of survival, but everyone in front of him can declare what hat they are wearing with just this information.

* 40% of the method cannot be parallelized. This entails that the time to execute that portion of the code stays the same. If the whole execution time on one processor took 1 second, the execution time on n processors would be 0.4 seconds. When Amdahl's law is applied it results in: $S = \frac{1}{0.4}$ = The limit of the speed up that can be achieved as n, the number of processors, goes to infinity is 2.20.

$$S_n = \frac{1}{1-0.7+\frac{0.7}{n}} = \frac{1}{0.3+\frac{0.7}{n}}$$

$$\frac{S_n}{2} = \frac{1}{0.3\times k+\frac{0.7}{n}}$$

$$S_n = \frac{2}{0.3\times k+\frac{0.7}{n}} = \frac{1}{0.3+\frac{0.7}{n}}$$

$$k = 2\left(0.3+\frac{0.1}{n}\right)$$

$S_n = \frac{k}{\quad}$ → overall performance speedup improved by a factor of 2.

7. $S_2 = \frac{1}{1 + p + \frac{p}{2}} \cdots$ $\quad k(S_2) = \frac{1}{1 - p + p/n}$

$S_n = \frac{1}{1 - p + \frac{p}{n}} \cdots$ $\quad k = \frac{1 - p + \frac{p}{2}}{1 - p + p/n}$

$k = \frac{1 - \frac{p}{2}}{1 - \left(\frac{n-1}{n}\right)p} \cdots$

$S_2 - S_2 p + \frac{S_2 p}{2} = 1$

$1 - S_2 = -S_2 p + \frac{S_2 p}{2}$

$p = \frac{-2 + 2S_2}{S_2}$

8. $proc1 = 5 z/s \quad n1 = 1$
$proc2 = 1 z/s \quad n2 = 10$

$R = proc2 / proc1 = 5$

$T\ proc1 = [(1 - p) + (p/n1)]/R$
$T\ proc2 = (1 - p) + (p/n2)$

$T\ proc2 < T\ proc1$
$1 - p + p/n2 < 1/R * [1 - p + p/n1]$
$1 - p + p/10 < 1/5 * [1 - p + p/1]$
$5 - 5p + p/2 < 1$
$10 - 10p + p < 2$
$10 - 9p < 2$
$-9p < 2 - 10$
$9p > 8$
$p > 8/9$

∴ For the 1 zillion/s processor to perform better, the program must have more than 8/9 parallel parts

↳: satisfies mutual exclusion

11. - For all i that is not equal to j, there cannot be an i in the critical section simultaneously with a j, where i & j are the i^{th} and j^{th} accesses of the critical section by seperate threads. → This is true because when i is in the critical section then turn is equal to i, and j must wait. the opposite is also true
- if thread 1 is entering the second while loop while busy is true & turn = 1, then when thread 2 goes to enter the first while loop & change turn to 2, a deadlock will happen
- ↳ Thread 1 will find that the turn variable is not equal to 1 & wait. while thread two is waiting for busy to be false
→ if it is not deadlock free, it is also not starvation free

14. class filter implements Lock {
  int[] level;
  int[] victim;
  public Lfilter(int n, int l){
    level = new int [max(n-1+1, 0)];
    victim = new int [max(n-1+1, 0)];
    for (int i = 0; i < n-1; i++) {
      level [i] = 0;
    }
  }
  public void lock() {

```
public void lock() {
    int me = ThreadID.get();
    for(int i = 1; i < n-l+1; i++) {
      level[me] = i;
      victim[i] = me
      int above = l+1;
      while (above > 1 && victim[i] == me) {
        above = l
        for(int k=l; k<n; k++) {
          if(level[k] >= i) above++;
        }
      }
    }
}

public void unlock() {
    int me = ThreadID.get();
    level[me] = 0;
  }
}
```

- reduced # of levels by l-1, to solve problem

15. - A & B run until line 8. Both are able to pass line 8 because neither have set victim = i yet. Assume $W_A(x=A) \rightarrow W_B(x=B) \rightarrow r_B(x==B)$. B is therefore allowed to enter the critical section w/o consulting the inner lock. A can now continue. Since x==B, lock.lock() is called. Since the inner lock has not been locked by B, A is able to enter the critical section. ∴ there is no mutual exclusion

16. a) Assume thread A is the first to return STOP. This implies that between the time A executed line 10 & the time it tested last==i, no other thread executed line 10. If any other thread also returned Stop, it must have executed line 10 after t_2. However, by this time goRight=1, so thread B must return RIGHT.

b) if at least one thread returned RIGHT, then no more than n-1 could return DOWN, otherwise, all threads must have passed line 11 before any of them executed line 13. This implies that at some point, all threads must have been concentrated at location 13. Then, the thread i such that the last=i will return STOP and all others will return DOWN

c) The first thread to execute line 11 must output STOP or DOWN, so at most n-1 can output RIGHT.