CS 624 - HW 3

Max Weiss

17 February 2020

9. Define r-bounded waiting for a given mutual exclusion algorithm to mean that if ${D^j}_A \to {D^k}_B$ then ${CS^j}_A \to {CS^{k+r}}_B$. Is there a way to define a doorway for the Peterson algorithm such that it provides r-bounded waiting for some value of r?

No: between one thread's write to VICTIM on line 9 and its read of VICTIM on line 10, there is no bound on the number of times another thread may execute its critical section.

10. Suppose that in the definition of fairness, we replace the reference to "doorway" with a reference to the initial instruction I of the lock method. We thus get

> A lock is *first-come-first-served-1* if, whenever thread A finishes I before thread B starts I, then A cannot be overtaken by B.

I will argue that *no* lock method can be first-come-first-served-1, because one thread can always "overtake" another.

In a bit more detail, let me write ${I_A}^k$ for the time interval of A's kth execution of I, and similarly write ${I_B}^k$. I will also write $X \hookrightarrow Y$ to mean that $X \to Y$, but also that no instruction were executed in between. Because I_A and I_B are discrete operations, it should be possible that ${I_A}^j \hookrightarrow {I_B}^k$. Suppose that it is possible that ${I_A}^j \hookrightarrow {I_B}^k$ and ${CS_A}^j \to {CS_B}^k$ (and similarly possible with A, B everywhere interchanged). I will argue that it must then also be possible that ${I_A}^j \to {I_B}^k$ while ${CS_B}^k \to {CS_A}^j$.

To this end, I will assume the following: (*) if the state of the machine following ${I_A}^j \hookrightarrow {I_B}^k$ is the same as the state of the machine following ${I_B}^k \hookrightarrow {I_A}^j$, then it is possible for either thread to overtake the other.

If I is a read, then it is clear that the machine state following ${I_A}^j \hookrightarrow {I_B}^k$ is the same as the machine state following ${I_B}^k \hookrightarrow {I_A}^j$. So by (*), B can overtake A.

On the other hand, suppose that I is a write. Let's use $L_A(I)$ to denote the location addressed by thread A in executing I, and likewise for $L_B(I)$. Now the problem splits into two subcases.

If $L_A(I) = L_B(I)$. Then after A's execution of I, thread B executes I as well, and also writes something to A. In this way, B obliterates whatever trace A has left in $L_A(I)$ of having begun the lock method. So again, B cannot tell whether A has entered the lock method.

Finally suppose that $L_A(I) \neq L_B(I)$. Then, the machine state following ${I_A}^j \hookrightarrow {I_B}^k$ is the same as the machine state following ${I_B}^k \hookrightarrow {I_A}^j$, so the conclusion again follows by (*).

11. Consider the algorithm

```
1 class Flaky implements Lock {
2   private int turn;
3   private boolean busy = false;
4   public void lock() {
5     int me = ThreadID.get();
6     do {
7       do {
8         turn = me;
9       } while (busy);
10      busy = true;
11    } while (turn != me);
12  }
13 public void unlock() {
14   busy = false;
15   }
16 }
```

a. Flaky does satisfy mutual exclusion. For, suppose that A, B are in the critical section simultaneously. Without loss of generality, suppose that A was the latest to read from TURN. Since A then read from this a different value than B last did, A must have last written to TURN after B read from it. But then, B must have last written BUSY to be true *before* A read busy to be false, but between A's last read of BUSY and B's last write to it, neither thread otherwise writes to it. Hence, A read the value true to be false, a contradiction.

b. It is not starvation-free. Immediately after A writes TURN to be A's, it is possible for B to run through the lock method, the critical section, and the unlock method unboundedly many times; and A's progress will be blocked provided it is unlucky enough each time to try to read from BUSY immedately after B has written it to be true.

c. So, suppose that after A writes BUSY to be true but before A reads from TURN, B writes TURN to be B's. Then both of A and B will enter their inner loops while BUSY has been set to true, and neither will make further progress.

12. The Filter lock allows some threads to overtake others an arbitrary number of times. Suppose, anyway, that there are three threads A, B, C. Let A first

pass through its doorway at level 1. At this point and before A completes a spin cycle, the following can happen arbitrarily many times: thread B then passes through the level 1 doorway, thread C passes through the doorway; thread B advances through to the critical section and unlocks; thread C advances through to the critical section and unlocks.

13. I will say that a thread A *passes through* a lock L when A has completed the L's lock method but A has not released L (so A has still flagged L). I will argue that if two threads pass through L, then two threads also pass through some child of L. To see this, note that by the mutual exclusion property of the Peterson lock, some third thread must have called the lock method of L. Now, L cannot be a leaf lock, because only two threads are assigned to a leaf. Therefore, L must have two children, and two of the three threads must have passed through one of them.

It follows that if two threads pass through a single lock, then the tree lock contains an infinite descending chain of locks, which is absurd.

Therefore: no two threads pass through a single lock. In particular, no two threads pass through the root lock. However, a thread in the critical section passes through the root lock. So, two threads cannot both be in the critical section.

Is a tree lock free from deadlock? Suppose the tree lock is really deadlocked, so that at least one thread has called the main lock method, but no thread is or will be in the critical section. Then, there must be some lock L such that some thread A has flagged L (n times for some $n > 0$), but no thread will complete L's lock method (for the nth time). At least one other thread B must flag L, since otherwise A will complete L's lock method. However, threads A and B cannot both be victim, and one who is not victim will complete L's lock method, a contradiction.

As for starvation freedom, note that since two threads cannot pass through a single lock, it follows that more than two threads cannot flag a single lock (else, two will have passed through a child lock, or a more than two will be registered to a leaf). I will argue that each thread A which calls the lock method of a given lock L eventually exits it. If no other thread flags L, then A will exit the lock method of L on its next loop iteration. If some other thread B flags L and B is the victim, then again A will exit. So, suppose A is the victim. Then B will exit on its next iteration. So on A's next subsequent loop iteration, either A remains victim but is the sole thread flagging L, or some other thread flags L and has become the victim. In either case, A will exit the lock method on its next loop iteration.

14. A natural approach to the i-exclusion problem is to modify the Filter lock in the following way: for n threads, use $n - i$ levels instead of $n - 1$. The counterpart of Lemma 2.4.1 is this: at any level j between 0 and $n - i$, at most $n - j$ threads are at level j. It follows that at most $n - (n - i) = i$ threads are at level $n - i$, so that at most i threads are ever in the critical section. The proof

of starvation-freedom also carries over, under the extra assumption that at most $i - 1$ threads fail in the critical section.