The Art of Multiprocessor Programming

Many

June 24, 2024

Contents

1 Mutual exclusion

1.1 Critical sections

A good `Lock` algorithm should satisfy:

- **Mutual exclusion**: At most one thread holds the lock at any time.
- **Freedom from deadlock**: If a thread is attempting to acquire or release the lock, then eventually some thread acquires or relases the lock. If a thread calls `lock()` and never returns, then other threads must complete an infinite number of critical sections (different from normal deadlocks we counter).
- **Freedom from starvation**: Every thread that attempts to acquire or release the lock eventually succeeds.

1.2 The Peterson lock

Lemma 1.1. *The* `Peterson` *lock algorithm satisfies mutual exclusion*

```java
class Peterson implements Lock {
    // thread-local index, 0 or 1
    private boolean[] flag = new boolean[2];
    private int victim;
    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;                  // I'm interested
        victim = i;                      // you go first
        while (flag[j] && victim == i) {} // wait
    }
    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;                 // I'm not interested
    }
}
```

Listing 1: Pseudocode for the `Peterson` lock algorithm

Proof. Suppose not. Consider the last executions of the `lock()` method by threads A and B.

$$\begin{aligned} &write_i(\texttt{flag}[i] = true) \to write_i(\texttt{victim} = i) \\ &\quad \to read_i(\texttt{flag}[j]) \to read_i(\texttt{victim}) \to CS_i \end{aligned}$$

Suppose A was the last thread to write to the `victim` field, then A observed `victim` to be A. Since A nevertheless entered its critical section, it must have observed $\texttt{flag}[B]$ to be *false*, so we have

$$write_A(\texttt{victim} = A) \to read_A(\texttt{flag}[B] == false)$$

and

$$\begin{aligned} &write_B(\texttt{flag}[B] = true) \to write_B(\texttt{victim} = B) \\ &\quad \to write_A(\texttt{victim} = A) \to read_A(\texttt{flag}[B] == false) \end{aligned}$$

A contradiction. □

Lemma 1.2. *The `Peterson` lock algorithm is starvation-free.*

Proof. Suppose not, so some thread runs forever in the `lock()` method. Suppose that it is A.

If B is repeatedly entering and leaving its critical section, then B sets `victim` to B before it reenters the critical section. Therefore A must eventually return from the `lock()`.

So B is also stuck in its `lock()` method. But `victim` cannot be both A and B. □

Corollary 1.3. *The* `Peterson` *lock algorithm is deadlock-free.*

1.3 The filter lock

```
class Filter implements Lock {
    int[] level;
    int[] victim;
    public Filter(int n) {
        level = new int[n];
        victim = new int[n]; // use 1..n-1
        for (int i = 0; i < n; i++) {
            level[i] = 0;
        }
    }
    public void lock() {
        int me = ThreadID.get();
        for (int i = 1; i < n; i++) { // attempt to enter level i
            level[me] = i;
            victim[i] = me;
            // spin while conflicts exist
            while ((k != me) (level[k] >= i && victim[i] == me)) {};
        }
    }
    public void unlock() {
        int me = ThreadID.get();
        level[me] = 0;
    }
}
```

Listing 2: Psudocode for the `Filter` lock algorithm

The `Filter` lock creates $n-1$ **levels**, that a thread must traverse before acquiring the lock. Levels satisfy two properties:

1. At least one thread trying to enter level l succeeds.
2. If more than one thread is trying to enter level l, then at least one is blocked (i.e., continues to wait without entering that level).

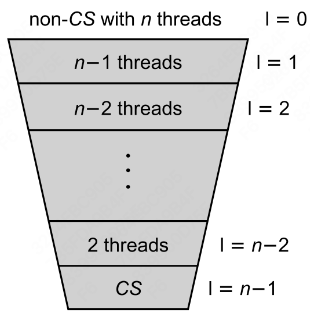

FIGURE 2.9

Threads pass through $n-1$ levels, the last of which is the critical section. Initially, all n threads are at level 0. At most $n-1$ enter level 1, at most $n-2$ enter level 2, and so on, so that only one thread enters the critical section at level $n-1$.

The value of $\texttt{level}[A]$ indicates the highest level that thread A is trying to enter.

Initially, a thread A is at level 0. A **enters** level $l > 0$ when it completes the `while` loop with $\texttt{level}[A] = l$. A enters its critical section when it enters level $n-1$. When A leaves the critical section, it sets $\texttt{level}[A] = 0$.

Lemma 1.4. *For j between 0 and $n-1$, at most $n-j$ threads have entered level j (and not subsequently exited the critical section).*

Proof. Induction. IH implies that at most $n-j+1$ threads have entered level $j-1$. Assume that $n-j+1$ threads have entered level j. Because $j \leq n-1$, there must be at least two such threads ($n-j+1 \geq 2$).

Let A be the last thread to write $\texttt{victim}[j]$. A must have entered level j since $\texttt{victim}[j]$ is written only by threads that have entered level $j-1$, and, by the IH, every thread that has entered level $j-1$ has also entered level j.

Let B be any thread other than A that has entered level j. Inspecting the code, we see that before B enters level j, it first writes j to $\texttt{level}[B]$ and then writes B to $\texttt{victim}[j]$. Since A is the last to write $\texttt{victim}[j]$, we have

$$write_B(\texttt{level}[B] = j) \rightarrow write_B(\texttt{victim}[j]) \rightarrow write_A(\texttt{victim}[j]).$$

We also see that A reads $\texttt{level}[B]$ after it writes to $\texttt{victim}[j]$, so

$$\begin{aligned} &write_B(\texttt{level}[B] = j) \rightarrow write_B(\texttt{victim}[j]) \\ &\quad \rightarrow write_A(\texttt{victim}[j]) \rightarrow read_A(\texttt{level}[B]). \end{aligned}$$

Because B has entered level j, every time A reads $\texttt{level}[B]$, it observes a value greater than or equal to j, and since $\texttt{victim}[j] = A$, A couldn't completed its waiting loop. $\square$

Corollary 1.5. *The `Filter` lock algorithm satisfies mutual exclusion.*

Lemma 1.6. *The `Filter` lock algorithm is starvation-free.*

Proof. We prove by induction on j that every thread that enters level $n - j$ eventually enters and leaves the critical section (assuming that it keeps taking steps and that every thread that enters the critical section eventually leaves it). The base case, with $j = 1$, is trivial because level $n - 1$ is the critical section.

For the induction step, we suppose that every thread that enters level $n - j$ or higher eventually enters and leaves the critical section, and show that every thread that enters level $n - j - 1$ does too.

Suppose, for contradiction, that a thread A has entered level $n - j - 1$ and is stuck. By IH, it never enters level $n - j$, so it must be stuck at loop with $\texttt{level}[A] = n - j$ and $\texttt{victim}[n - j] = A$. After A writes $\texttt{victim}[n - j]$, no thread enters level $n - j - 1$. Furthermore, any other thread B trying to enter level $n - j$ will eventually succeed because $\texttt{victim}[n - j] = A \neq B$, so eventually no threads other than A are trying to enter level $n - j$. Moreover, any thread that enters level $n - j$ will, by IH, enter and leave the critical section, setting its level to 0. In particular, after this point, $\texttt{level}[B] < n - j$ for every thread B other than A, so A can enter level $n - j$, a contradiction. □

Corollary 1.7. *The `Filter` lock algorithm is deadlock-free.*

1.4 wefwef