Lock using Peterson's Algorithm

Shankar

April 15, 2014

Overview

- Classical mutual exclusion problem
  - given program with "critical sections" and threads `0..N-1`
  - obtain "entry" and "exit" code for each critical section st
    - at most one thread in a critical section
    - thread in entry code eventually enters critical section if no thread stays in critical section forever
    - assume only atomic reads and writes
- Any solution provides a `SimpleLock(N)` implementation
- Here we use Peterson's solution for $N = 2$

- hungry: ongoing request for the lock
- eating: holds the lock; in critical section
- thinking: neither hungry nor eating

// conventions

Peterson's algorithm

- Threads 0 and 1
- Share three binary variables
  - th_0 // true iff thrd 0 thinking
  - th_i // " " 1 "
  - turn // 0..1: winner if both hungry
- Thread i becomes hungry:

  $th_i \leftarrow$ false; turn $\leftarrow$ j;
  busy wait while not th_j and turn is i

  i, j in 0..1, i $\neq$ j
- Thread i becomes thinking:

  $th_i \leftarrow$ true;

Program LockPeterson() – 1

- Main

```
boolean[2] th ← true;
int turn ← 0;  // 1 is ok also
return mysid;
```

- input mysid.acq()

```
s1:    th[mytid] ← false;
s2:    int j ← 1 - mytid;
s3: • turn ← j;
       while
s4:      ( • not th[j]
s5:       and • turn = j);
       return;
```

Program LockPeterson() – 2

- input mysid.rel()

  s6: th[mytid] $\leftarrow$ true;
  return;

- input void mysid.end()

  endSystem();

- atomicity assumption: reads and writes of turn, th_0, th_1
- progress assumption: weak fairness for every thread

LockPeterson() implements SimpleLockService(2)

- SimpleLockService(N) and its inverse: defined in chapter 2
- program Z()

```
lck ← startSystem(LockPeterson());
lsi ← startSystem(SimpleLockServiceInverse(2,lck));
```

- Assertions to establish

Y_1 : Inv (thrd at doAcq.ic) $\Rightarrow$ (not $acqd_0$ and not $acqd_1$)

Y_2 : (thrd i in lck.rel) *leads-to* (not i in lck.rel)

Y_3 : (thrd i in lck.end) *leads-to* (not i in lck.end)

Y_4 : forall(i: $acqd_i$ *leads-to* not $acqd_i$) $\Rightarrow$
forall(i: (thread i on lck.acq) *leads-to* $acqd_i$)

- Y_2, Y_3 hold trivially // lck.rel, lck.end non-blocking
- Proofs of Y_1 and Y_4 follow

Safety proof: Y_1

Y_1 : Inv (thrd at doAcq.ic) $\Rightarrow$ (not acqd_0 and not acqd_1)

- Intermediate assertions

  A_1(i) : (thread i on s4..s5) and (th[j] or turn $\neq$ j)
  $\Rightarrow$ (not acqd[0] and not acqd[1])

  A_2(i) : th[i] $\Rightarrow$ not acqd_i

  A_3(i) : (i on s3..s5) $\Rightarrow$ not acqd_i

  A_4 : turn in 0..1

  A_5(i) : ((i on s4..s5) and (turn $\neq$ j)) $\Rightarrow$ not acqd[j]

- Y_1 equivalent to $Inv\ A_1$ // effective atomicity
- A_2, A_3, A_4 each satisfy invariance rule
- A_5 satisfies invariance rule assuming $Inv\ A_2$–A_4
- A_2–$A_5 \Rightarrow A_1$

Progress proof: Y_4 – 1

Y_4 : $P_0(0)$ and $P_0(1)$ $\Rightarrow$ $P_1(0)$ and $P_1(1)$ // P_0, P_1 below

P_0(i) : acqd$_\text{i}$ *leads-to* not acqd$_\text{i}$

P_1(i) : (i on lck.acq) *leads-to* acqd$_\text{i}$

P_2(i) : (i on s4..s5) *leads-to* acqd[i] // equiv to P_1(i)

P_3(i) : (i on s4..s5) and turn=i *leads-to* acqd[i] // via i

α(i) : (i on s4..s5) and turn=j

P_4(i) : α(i) and th[j] *leads-to* (acqd[i] or turn=i) // via i

P_5(i) : α(i) and th[j] *leads-to* acqd[i] // via i, P_3(i), P_4(i)

Progress proof: Y_4 – 2

P_6(i) : α(i) and not th[j]

$leads\text{-}to$ α(i) and ((j on s3..s5) or acqd[j])

$leads\text{-}to$ (α(i) and acqd[j]) // P_3(j)

$leads\text{-}to$ (α(i) and th[j]) // P_0(j)

$leads\text{-}to$ acqd[i] // P_5(i)

- P_1(i) follows from P_6(i), P_5(i), and P_3(i)