Lock using Bakery Algorithm

Shankar

April 16, 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 `SimpleLockService(N)` implementation
- We will obtain one using the Bakery algorithm

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

// conventions

The Bakery Approach

- Variables shared by threads `0..N-1`
  - `num[0]`, $\cdots$, `num[N-1]`, initially 0
    // `num[i]` > 0 iff `i` not thinking
- Lock acquire: thread `i` does two scans of `nums`
  - s1: set `num[i]` to a value higher than other `nums`
  - s2: wait at each `j` while 0 < `num[j]` < `num[i]`
- Lock release: thread `i` zeroes `num[i]`
- Refer to the above as simplified bakery
  - works if s1 is atomic but not with read-write atomicity
- Classical bakery handles r/w atomicity but nums unbounded
- Black-white bakery handles r/w atomicity with nums bounded

Outline

Program LockSimplifiedBakery(N)

- Main

```
      num[0..N-1] ← 0
```

- mysid.acq():

```
s1: •num[myid] ← max(num[0], ···, num[N-1]) + 1
      for (p in 0..N-1)
s2:       do •x ← num[p]
             while (0 < x < num[myid])
```

- mysid.rel():

```
      num[myid] ← 0
```

- mysid.end():

```
      num[myid] ← 0
```

- atomicity assumption: the '•'s
- progress assumption: weak fairness

Analysis – 1

- Goal: show simplified bakery implements simple lock service
- Define closed program of
  - `LockSimplifiedBakery(N)` system, `lck`, and
  - `SimpleLockServiceInverse(N)` system, `lsi`
- Assertions to establish

  Y_1 : Inv (thrd at `doAcq(i).ic`) $\Rightarrow$ (no thrd eating)

  Y_2 : thrd `i` in `lck.rel` returns

  Y_3 : thrd `i` in `lck.end` returns

  Y_4 : every hungry thrd becomes eating if eating is bounded
- Y_2 and Y_3 hold trivially // `lck.rel`, `lck.end` non-blocking
- Proofs of Y_1 and Y_4 follow

Analysis – 2

- Instructive to define a hypothetical queue of contenders
- Q: sequence of ids of non-thinking threads in increasing num order
  - i joins / leaves Q when it executes s1 / rel
  - nums in Q are distinct // s1 is atomic
  - arrival joins at tail // " "
- i *ahead-of* j: 0 < num[i] < num[j]
- i *passed* j: i is eating or i is in s2 with i.p > j
- If i ahead-of j then j cannot pass i
  - so only the thread at the head of Q can eat // Y_1
- If i at head of Q then i passes every j
  - so i eats and then leaves Q
  - at which point every j in Q gets closer to the head // Y_4

Outline

Simplified Bakery

- Simplified bakery: fails with only read-write atomicity

Classical Bakery

Black-white Bakery

Flaws with r/w atomicity – 1

- Simplified bakery fails if only reads and writes are atomic
  - problem arises when threads execute s1 simultaneously
- Flaw 1
  - threads `i` and `j` overlapping in s1 can get equal nums
    - e.g., each reads the other's num before either updates its num
  - each thread passes the other: both acquire the lock
    // or each thread waits for the other: deadlock
- Fixing flaw 1
  - use thread ids to break ties in s2
  - let `[num[i],i] < [num[j],j]` denote
    `num[i] < num[j]` or `(num[i] = num[j]` and `i < j)`

Flaws with r/w atomicity – 2

- Flaw 2
  - threads i and j overlap in s1
  - i leaves s1 before j, passes j in s2 because num[j] still 0
  - j leaves s1 later with num[j] < num[i], so j passes i in s2
  - i and j both acquire the lock

- Fixing flaw 2
  - booleans choosing[0], $\cdots$, choosing[N-1], initially false
  - i sets choosing[i] before s1 and resets it after s1
  - in s2, thrd i reads num[j] only after finding choosing[j] false

- Thus i reads an "unstable" num[j] only if
  j started choosing after i finished choosing
  - so num[j] will be higher than num[i] and j will not pass i

Outline

Program LockBakery(N)

- Main:

```
choosing[0..N-1] ← false
num[0..N-1] ← 0
```

- mysid.acq():

```
t1:  choosing[myid] ← true
t2: • num[myid] ← max( • num[0],···, • num[N-1]) + 1
t3: • choosing[myid] ← false
     for (p in 0..N-1)
t4:     while ( • choosing[p]) skip
t5:     do  • x ← num[p]
            while (x ≠ 0  and  [x,p] < [num[myid],myid])
```

- mysid.rel():

```
num[myid] ← 0
```

- mysid.end()

```
endSystem()
```

Analysis: overview

- Goal: show bakery implements simple lock service
- Proceeding as usual
  - closed program of lock and service inverse
  - assertions Y_1–Y_4 to establish
- Y_2–Y_3 hold trivially
- Establish Y_1, Y_4 next

Analysis: auxiliary quantities

- Proof similar to that of simplified bakery
- *Q*: hypothetical queue of ids of non-thinking non-choosing threads in increasing [num,id] order
- i *ahead-of* j: [0,·] < [num[i],i] < [num[j],j]
- passed(i,j): i is eating or i is in t4..t5 with i.p > j
- j is a peer of i if:
  - i and j are non-thinking
  - their choosing intervals overlapped
  - j is still choosing // so not commutative
- peers[i]: set of peers of i // auxiliary var

Analysis: safety proof

C_0(i) : ((i on s2) and i.p = N−1 and
(num[p] = 0 or [num[i.p],i.p] > [num[i],i])) $\Rightarrow$
forall(j in 0..N−1: not acqd[j])

C_1(i,j) : (i $\neq$ j and passed(i,j)) $\Rightarrow$
((not j in peers[i]) and
(not acqd[j] or (j on s1..t2) or
(num[j] > 0 and [num[j],j] > [num[i],i])))

C_2(i,j) : (i$\neq$j and (i on s2) and i.p=j and choosing[j]) $\Rightarrow$
(j not in peers[i])

- *Inv* C_0(i) equivalent to Y_1 given effective atomicity
- C_2(i,j) satisfies invariance rule
- C_1(i,j) satisfies invariance rule assuming *Inv* C_2(i,j)
- C_1(i,j) and C_1(j,i) imply C_0

Analysis: progress proof

$\alpha_{\mathtt{i}}$: # entries ahead-of i $\beta_{\mathtt{i}}$: peers[i].size

D_1 : $[\beta_{\mathtt{i}}, \alpha_{\mathtt{i}}] = [\mathtt{k1}, \mathtt{k2}] > [0,0]$ *unless* $([\beta_{\mathtt{i}}, \alpha_{\mathtt{i}}] < [\mathtt{k1}, \mathtt{k2}])$

D_2 : $\beta_{\mathtt{i}} = \mathtt{k1} > 0$ *leads-to* $\beta_{\mathtt{i}} < \mathtt{k1}$ // choosing bounded

D_3 : $[\beta_{\mathtt{i}}, \alpha_{\mathtt{i}}] = [0,0]$ *leads-to* acqd[i]   // i never blocked

D_4 : $[\beta_{\mathtt{i}}, \alpha_{\mathtt{i}}] = [0,0]$ *leads-to* not acqd[i]   // D_3, eating ends

D_5 : $[\beta_{\mathtt{i}}, \alpha_{\mathtt{i}}] = [\mathtt{k1}, \mathtt{k2}] > [0,0]$ *leads-to* $[\beta_{\mathtt{i}}, \alpha_{\mathtt{i}}] < [\mathtt{k1}, \mathtt{k2}]$

- D_1 holds coz β non-increasing, α increases only if β decreases
- D_5: from D_2, D_1 for k1>0; from D_4.head for k1=0, k2>0
- D_5 and D_3 imply Y_4

The beautiful and the undesirable

- Beautiful: r/w atomicity not needed
  - no overlapping writes to the same location
  - read that overlaps with a write can return any value
    - `i` reads unstable var of `j` only if `j` is choosing
    - so `num[j]` will end up higher than `num[i]`
    - so `i` will never make a wrong decision

- Undesirable: `nums` are not bounded

Outline

Black-white bakery algorithm

- Bounds nums but requires r/w atomicity for a binary flag
- Two hypothetical queues: one black, one white
- Flag, either black or white // indicates the open queue
- Each user has a color (its queue) and the usual $\langle \mathrm{num}, \mathrm{id} \rangle$
  - gets the flag's color, sets its num based on users in its queue
- Priority: $\langle \mathrm{num}, \mathrm{id} \rangle$, except open-queue defers to closed-queue
- When a user eats, it sets the flag to the opposite of its color
- So open$\leftrightarrow$closed happens when an open user starts eating
  - at which point the other queue, which was closed, is empty
  - so the next arrival sets its num starting from 0