Spin Locks from Read-Write Atomicity

Shankar

October 29, 2013

Outline

Critical-Section Problem

- Given program with
  - threads 0, $\cdots$, `N-1` that execute concurrently
  - parts of the program designated as critical sections (CSs)

- To obtain `entry` and `exit` code around each CS so that
  - at any time there is at most one thread in all of the CSs
  - any thread in entry code eventually enters its CS provided no thread stays in a CS forever
  - code requires only read-write atomicity
    - no read-modify-write atomicity (eg, no test&set)

Locks from Critical-Section Solutions

- Any solution yields a lock requiring only read-write atomicity
  - lock definition: variables of CS solution
  - lock acquire body: entry code
  - lock release body: exit

- Two of the simplest solutions
  - Peterson algorithm: $N = 2$
  - Bakery algorithm: arbitrary N

  We will obtain locks from these two solutions

- Terminology
  - thread is eating if it holds the lock
  - " " hungry if it is acquiring the lock
  - " " thinking otherwise

Outline

Peterson Lock

- Threads 0 and 1
- Shared variables
  - `flag[0]` ← `false` // true iff thread 0 is non-thinking
  - `flag[1]` ← `false` // true iff thread 1 is non-thinking
  - turn ← 0 or 1 // identifies winner in case of conflict
- `acq()`:

```
     j ← 1 - myid                 // j is other thread's id
s1:  flag[myid] ← true
s2:  turn ← j
s3:  while (flag[j] and turn = j) skip
```

- `rel()`:

```
     flag[myid] ← false
```

Peterson Lock: Safety

Suppose thread `i` leaves s3 at time t_0.
Need to show that thread `j` is not eating at t_0.

- Only two ways that `i` leaves s3.
- Case 1: `i` leaves s3 because `flag[j]` is false.

  Then at t_0, `j` is thinking and so does not hold the lock.
- Case 2: `i` leaves s3 because `flag[j]` is true and turn is `i`.

  Thread `i` executed s2 at some t_1 $(< t_0)$, setting turn to `j`.
  Because turn is `i` at t_0, `j` executed s2 at some t_2 in $[t_1, t_0]$.
  Hence `flag[i]` is true and turn is `i` during $[t_2, t_0]$.
  Hence `j` is stuck in s3.

Peterson Lock: Progress

Suppose `i` calls `acq(i)` and is in `s3` at time t_0.
Need to show that `i` eventually leaves `s3`.

C_1: Suppose `turn` is `i` at t_0.
It remains so. Hence `i` eventually leaves `s3`.

C_2: Suppose `flag[j]` is false at t_0.
Eventually `i` leaves `s3` or `j` does `s1;s2` ($\rightarrow C_1$).

C_3: Suppose `flag[j]` is true and `turn` is `j` at t_0.
So `j` is eating or hungry.

C_{3a}: If `j` is eating, it eventually stops eating ($\rightarrow C_2 \rightarrow C_1$)

C_{3b}: If `j` is at `s2`, it eventually does `s2` ($\rightarrow C_1$).

C_{3c}: If `j` is in `s3`, then `turn` remains `j`, so `j` eventually eats ($\rightarrow C_{3a} \rightarrow C_2 \rightarrow C_1$)

So eventually C_1 holds, which leads to `i` eating.

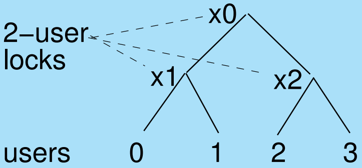
Outline

Obtaining a *N*-user lock from 2-user locks

- Define a binary tree of (at least) *N* leaf nodes.
- Associate a distinct 2-user lock with every non-leaf node.
- Associate the *N* users with distinct leaf nodes.
- A thread acquires the *N*-user lock by acquiring in order the 2-user locks on the path from my leaf to root
- A thread releases the *N*-user lock by releasing the acquired 2-user locks (in any order)

4-user lock example

- thread 0 acquires x1, x0
- thread 2 acquires x2, x0



- But there are better ways to implement *N*-user locks

Outline

Bakery Solution: Approach

- Threads 0, $\cdots$, N−1
- Share variables `num[0]`, $\cdots$, `num[N-1]`, initially 0
  - `num[i]` is 0 if `i` thinking, else > 0;   in conflict, smaller num wins

- Lock acquire: thread `i` does two scans of the nums
  - s1: set `num[i]` to a value higher than other nums
  - s2: wait at each `num[j]` until
    `num[j]` is 0 or greater than `num[i]`

- Lock release: thread `i` zeroes `num[i]`

- This works if s1 is atomic, but not with read-write atomicity.

- Next
  - define a "XBakery" lock based on the above
  - show how it fails with read-write atomicity
  - show how to fix it, resulting in the Bakery lock

XBakery Lock

- Lock:

```
num[0..N-1] ← [0,···,0]
```

- acq():

```
s1:  num[myid] ← max(num[0],···,num[N-1]) + 1

     for (p in 0..N-1)
s2:     do
           x ← num[p]
           while (x = 0 or x < num[myid]) skip
```

- rel():

```
num[myid] ← 0
```

XBakery Lock with s1 atomic

- Define
  - *Q*: hypothetical queue of ids of non-thinking threads
    in increasing num order
    - `i` joins *Q* when thread `i` executes s1
    - `i` leaves *Q* when thread `i` executes `rel()`
  - `i` *is ahead of* `j`: 0 < `num[i]` < `num[j]` holds
  - `i` *has passed* `j`: `i` is eating or `i` is in s2 with `i.p` > `j`.
- Properties
  - arrival to *Q* joins at tail // coz s1 is atomic, right?
  - threads in *Q* have distinct nums " " " "
  - if `i` is ahead of `j` then `j` cannot pass `i`
  - so only the thread at the head of *Q* can eat
  - if `i` is ahead of `j` then `i` eventually passes `j`
  - so the thread at the head of *Q* will eventually eat

XBakery Lock with read-write atomicity

- XBakery lock does not work if only reads and writes are atomic.

- Flaw 1
  - threads `i` and `j` enter s1 simltaneously
  - each reads the other's num before either updates its num
  - hence `num[i]` equals `num[j]` and both threads are in s2
  - each thread passes the other, both acquire the lock. v

- Flaw 2
  - threads `i`, `j`, `k` enter s1 simultaneously
  - `i` completes s1 except for updating `num[i]`, to say x
  - `j` completes s1, setting `num[j]` to x
  - `k` completes s1, setting `num[k]` to $x + 1$
  - `k` enters s2, passes `i` (because `num[i]` is 0)
  - `i` completes s1, setting `num[j]` to x
  - `i` enters s2 and passes `k` (because `num[k]` > `num[i]`)
  - `i` and `j` can now both acquire the lock

Fixing XBakery for read-write atomicity

- Fixing flaw 1
    - use thread ids to break ties
    - let `[num[i],i] < [num[j],j]` denote
      `num[i] < num[j]` or (`num[i]` = `num[j]` and `i` < `j`)
- Fixing flaw 2
    - introduce booleans `choosing[0]`, $\cdots$, `choosing[N` $-$ `1]`
      such that `choosing[i]` true if `i` in s1
    - in s2, thread `j` reads `num[i]` only after finding `choosing[i]` false
    - so if `num[i]` changes after `j` reads it, it is because of `i`
      executing s1 after `j` left s1.
    - so `num[i]` will be higher than `num[j]`, so `i` cannot pass `j`

Bakery Lock

- Lock:

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

- 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])
```

- rel():

```
    num[myid] ← 0
```

Bakery Lock Analysis: Definitions

- Define
  - `i` is *choosing*: `choosing[i]` is true (ie, `i` on t2,t3)

  - `j` is a *peer* of `i`:
    - `i` and `j` are non-thinking
    - their choosing intervals overlapped
    - `j` is still choosing

  - *Q*: hypothetical queue of ids of non-thinking non-choosing threads in increasing `[num,id]` order

    // "non-choosing" simply makes the argument cleaner: once a
    // thread enters *Q*, it is nobody's peer (but it can have peers)

  - `i` *is ahead of* `j`: `[0,·]` < `[num[i],i]` < `[num[j],j]` holds
  - `i` *has passed* `j`: `i` is eating or `i` is in t4..t5 with `i.p` > `j`

Bakery Lock Analysis: Properties

- While thread `i` is in *Q*
  - set of its peers keeps decreasing // choosing is non-blocking
  - only a peer can join *Q* ahead of `i`
  - so at most `N-1` threads can join *Q* ahead of `i`

- When thread `i` reads `num[j]` in `t5`
  - `j` is not currently a peer of `i`
    // `j` not choosing, or started choosing after `i` finished choosing
  - so `i` may pass `j` based on an unstable `num[j]`
    but `j` will not pass `i` // coz `num[j]` will exceed `num[i]`

- only the head eats // coz `i` passes `j` only if `i` is ahead of `j`

- every hungry `i` eventually eats
  - eventually `i` has no peers // coz choosing is non-blocking
  - after this, no thread joins ahead of `i`, the head eventually eats,
    so `i` eventually becomes the head and eats