Operating Systems: Implementing synchronization constructs

Shankar

May 3, 2022

Outline

1. Implementing Locks: Overview
2. Locks via Interrupt-Disabling (single-cpu only)
3. Spinlocks via Read-Modify-Write Instructions (multi-cpu)
4. Lock with Spin Waiting + Queue Waiting (multi-cpu)
5. Condition Variables
6. Semaphores
7. Spinlocks via Read and Write Instructions (multi-cpu)
8. SpinLock via RW: Peterson solution
9. Obtaining *N*-user locks from 2-user locks
10. Spinlock via RW: Bakery solution

Implementing Locks: Overview

- Implementations for single-cpu system
  - tcb queues for waiting // tcb: thread control block
  - interrupt-disabling for atomic access to queues
- Implementations for multi-cpu systems
  - interrupt-disabling does not work
  - busy waiting is necessary
- Spinlocks: all waiting is busy (ok for short waits)
  - using atomic read-modify-write instructions
  - using atomic read and write instructions
- "Long-wait" locks: tcb queues + spinlocks to guard queues
- Implementation in GeekOS (see GeekOS overview)

Outline

Lock: single-cpu, intrpt, tcb – 1

- `Lock lck`:

```
    lckFree ← true                              // lck free or not
    lckQueue ← [];                  // threads waiting to acquire lck
```

- `lck.acq()`:

```
                        // here on syscall with interrupts disabled
    if lckFree
      lckFree ← false
      rti                                   // return from interrupt
    else  // lck not free
      update my tcb [ra set to after acq call]
      move my tcb to lckQueue
      scheduler()
```

Note: `scheduler()` called with interrupts off

Lock: single-cpu, intrpt, tcb – 2

- `lck.rel()`: // here on syscall with interrupts disabled

```
if lckQueue ≠ []
  move a tcb from lckQueue to ready queue
  // lckFree stays false
else
  lckFree ← true
rti
```

- For deterministic progress
  - fifo (or any fair) discipline for lock queue
- Alternative `lck.rel()`: move waiting tcb to run queue
  - priority to waiting thread

Outline

Atomic Read-Modify-Write Instructions

- Spinlock data located in memory shared by all cpus
- Examples of atomic RMW instructions
  - `test&set`(x): atomic $\{\text{return } x;\ x \leftarrow \text{true}\}$
  - swap(x): atomic $\{[x, \text{reg}] \leftarrow [\text{reg}, x]\}$
- Expensive instructions: affect caches, memory bus, …

SpinLock via test&set

- Lock `lck`:

```
lckAcqd ← false        // accessible by all processors
```

- `lck.acq()`:

```
while (test&set(lckAcqd)) skip;
return
```

- `lck.rel()`:

```
lckAcqd ← false
return
```

- Probabilistic progress. Why?

Spinlock with deterministic progress – 1

- Approach
  - associate ids with threads, say 0, · · · , N−1
    - notational convenience: assume ids passed in acq/rel calls instead of taken from tcb
  - introduce booleans `w[0]`, · · · , `w[N−1]` where `w[i]` true iff thread `i` is waiting for the lock
  - when a thread `j` does release
    look for next (in modulo-N order) waiting thread,
    if found “pass” the lock to it, else set lock free
- Lock `lck`:

```
acqd ← false
w[0], ··· w[N−1] ← [false, ···, false]
```

Spinlock with deterministic progress – 2

- `lck.acq(i)`:

```
key ← true          // local variable
w[i] ← true
while (w[i] and key)
   key ← test&set(acqd)
w[i] ← false
return
```

- `lck.rel(i)`:

```
j ← (i+1) mod N
while (j ≠ i and not w[j])
   j ← (j+1) mod N
if (j = i)
   acqd ← false
else
   w[j] ← false
return
```

Outline

Lock: spin, tcb, multi-cpu – 1

- Spinlock is not ok if lock can be held for a long time
  - excessive busy waiting
- For locks with potentially long hold times
  - use TCB queues for waiting // like single-processor case
  - use spinlocks to achieve atomic queue access
    // takes the place of interrupt-disabling
- Lock `lck`:

```
lckFree ← true                          // lck free or not
lckQueue ← []                           // processes waiting to acquire lck
lckSplock                               // spinlock for lckFree, lckQueue
```

- Assume
  - `rrSplock`: spinlock to protect ready and run queues
  - `scheduler()`: call with `rrSplock` not free; releases `rrSplock`

Lock: spin, tcb, multi-cpu – 2

- `lck.acq()`:

```
lckSplock.acq()
if lckFree
   lckFree ← false
   lckSplock.rel()
else // lck not free
   rrSplock.acq()
   update my tcb [ra set to after acq() call]
   move my tcb to lckQueue
   lckSplock.rel()
   // note: rrSplock is not free
   scheduler()
```

Lock: spin, tcb, multi-cpu – 3

- `lck.rel()`:

```
lckSplock.acq()
if lckQueue ≠ []
   rrSplock.acq()
   move a tcb from lckQueue to ready queue
   rrSplock.rel()
else
   lckFree ← true
lckSplock.rel()
return
```

- For deterministic progress:
  - fifo (or any fair) discipline for lock queue
  - spinlocks with deterministic progress

Outline

Implementing Conditions – 1

- Approach: condition variable cv associated with lock `lck`
    - cvQueue: for processes waiting on cv
    - cv.wait(): atomic {release `lck`; wait on cvQueue}; acquire `lck`
    - cv.signal(): wakeup on cvQueue
    - spinlock: for atomic access to queues
        - or interrupt-disabling if single-processor

- `cv ← Condition(lck):`

```
cvQueue ← []         // processes waiting on cv
cvSplock             // lock to protect cvQueue
```

- Assume
    - `rrSplock`: spinlock to protect ready and run queues
    - `scheduler()`: call with `rrSplock` not free; releases `rrSplock`

Implementing Conditions – 2

- `cv.wait():`

```
     rrSplock.acq()
     cvSplock.acq()
     update my tcb [ra set to a1]
     move my tcb to cvQueue
     cvSplock.rel()
     lck.rel()
     scheduler()
 a1: lck.acq()
```

- `cv.signal():`

```
     rrSplock.acq()
     cvSplock.acq()
     move a tcb from cvQueue to ready queue
     cvSplock.rel()
     rrSplock.rel()
```

Outline

Semaphores – 1

- Approach: semaphore sem
  - `semVal`: value of sem
  - `semQueue`: for processes waiting on sem
  - P: if `semVal > 0` then decrement it else join `semQueue`
  - V: if `semQueue` not empty then move a tcb to ready queue
    else increment `semVal`
  - spinlocks for atomic access to queues
    - or interrupt-disabling if single processor

- `sem ← Semaphore(N)`:

```
semVal ← N                              // value of sem
semQueue ← []                     // for waiting on sem
semSplock  // spinlock to protect semVal and semQueue
```

- Assume
  - `rrSplock`: spinlock to protect ready and run queues
  - `scheduler()`: call with `rrSplock` not free; releases `rrSplock`

Implementing Semaphores – 2

- `sem.P()`:

```
semSplock.acq()
if (sem.val > 0)
    sem.val ← sem.val − 1
    semSplock.rel()
else // sem.val = 0
    rrSplock.acq()
    update my tcb [ra set to after P() call]
    move my tcb to semQueue
    semSplock.rel()
    scheduler()
```

Implementing Semaphores – 3

- sem.V():

```
semSplock.acq()
if (semQueue = [])
    sem.val ← sem.val + 1
else
    rrSplock.acq()
    move a tcb from semQueue to ready queue
    rrSplock.rel()
semSplock.rel()
return
```

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
- Peterson algorithm solution: $N = 2$
- Bakery algorithm solution: arbitrary N
- Terminology
  - thread is eating if it holds the lock
  - " " hungry if it is acquiring the lock
  - " " thinking otherwise

Outline

1. Implementing Locks: Overview
2. Locks via Interrupt-Disabling (single-cpu only)
3. Spinlocks via Read-Modify-Write Instructions (multi-cpu)
4. Lock with Spin Waiting + Queue Waiting (multi-cpu)
5. Condition Variables
6. Semaphores
7. Spinlocks via Read and Write Instructions (multi-cpu)
8. SpinLock via RW: Peterson solution
9. Obtaining N-user locks from 2-user locks
10. Spinlock via RW: Bakery solution

Peterson Lock

- Threads 0 and 1 // id passed instead of taken from tcb
- 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(i)`:

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

- `rel(i)`:

```
    flag[i] ← 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

1. Implementing Locks: Overview
2. Locks via Interrupt-Disabling (single-cpu only)
3. Spinlocks via Read-Modify-Write Instructions (multi-cpu)
4. Lock with Spin Waiting + Queue Waiting (multi-cpu)
5. Condition Variables
6. Semaphores
7. Spinlocks via Read and Write Instructions (multi-cpu)
8. SpinLock via RW: Peterson solution
9. Obtaining *N*-user locks from 2-user locks
10. Spinlock via RW: Bakery solution

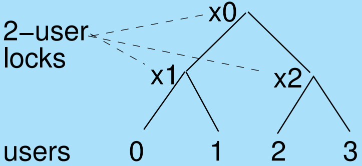
N-user lock from 2-user locks

- 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

1. Implementing Locks: Overview
2. Locks via Interrupt-Disabling (single-cpu only)
3. Spinlocks via Read-Modify-Write Instructions (multi-cpu)
4. Lock with Spin Waiting + Queue Waiting (multi-cpu)
5. Condition Variables
6. Semaphores
7. Spinlocks via Read and Write Instructions (multi-cpu)
8. SpinLock via RW: Peterson solution
9. Obtaining *N*-user locks from 2-user locks
10. Spinlock via RW: Bakery solution

Bakery Solution: Initial version

- Threads 0, · · · , N-1
- Shared non-negative integer variables
  - `num[0]`, · · · , `num[N-1]` ← `0, · · · , 0`
  - `num[i]` is 0 iff `i` thinking; in conflict, smaller num wins
- `acq(i)`:

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

        for p in 0..N-1:
s2:       while (0 < num[p] < num[i]):
            noop
```

- `rel(i)`:

```
      num[i] ← 0
```

- This works if `s1` is atomic.
- It does not work if only reads and writes are atomic.

Analysis assuming 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

Assuming only read-write atomicity

- Flaw 1: threads `i` and `j` leave `s1` with the same `num`
    - `i` and `j` enter `s1` simltaneously
    - each reads the other's `num` before either updates its `num`
    - each updates its `num` and enters `s2`
    - each passes the other, so both acquire the lock

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

Fix assuming only read-write atomicity

- Fixing flaw 1
  - use thread ids to break ties // lexicographic ordering
  - 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]`
  - `i` sets `choosing[i]` true while `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, then `i` executed s1 after `j` left s1.
  - so `num[i]` will be higher than `num[j]`, so `i` cannot pass `j`

Bakery Lock

- Shared variables:

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

- acq(i):

```
t1:  choosing[i] ← true
t2:  num[i] ← max(num[0],···,num[N-1]) + 1
t3:  choosing[i] ← false

     for p in 0..N-1:
t4:    while choosing[p]:
         noop
t5:    while [0, .] < [num[p], p] < [num[i], i]:
         noop
```

- rel(i):

```
    num[i] ← 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