Synchronization (2)

Dave Eckhardt
de0u@andrew.cmu.edu

Synchronization

- Handing in
  - Issues are possible
  - ***Don't mail us your assignment***
  - Try hand-in ***now***
  - Thank you

Outline

- Last time
  - Two building blocks
  - Three requirements for mutual exclusion
  - Algorithms people ***don't*** use for mutual exclusion
- Today
  - Ways to ***really*** do mutual exclusion

Mutual Exclusion: Reminder

- Protects an atomic instruction sequence
  - Do "something" to guard against
    - CPU switching to another thread
    - Thread running on another CPU
- Assumptions
  - Atomic instruction sequence will be “short”
  - No other thread “likely” to compete

Mutual Exclusion: Goals

- Typical case (no competitor) should be fast
- Atypical case can be slow
  - Should not be “too wasteful”

Mutex aka Lock aka Latch

- Object specifies interfering code sequences
  - Data item(s) “protected by the mutex”
- Methods encapsulate entry & exit protocols

```
mutex_lock(&store->lock);
cash = store->cash
cash += 50;
personal_cash -= 50;
store->cash = cash;
mutex_unlock(&store->lock);
```

- What's inside?

Mutual Exclusion: Atomic Exchange

- Intel x86 XCHG instruction
  - intel-isr.pdf page 754
- xchg (%esi), %edi

```
int32 xchg(int32 *lock, int32 val) {
  register int old;
  old = *lock; /* bus is locked */
  *lock = val; /* bus is locked */
  return (old);
}
```

Inside a Mutex

- Initialization

```
int lock_available = 1;
```

- Try-lock

```
i_won = xchg(&lock_available, 0);
```

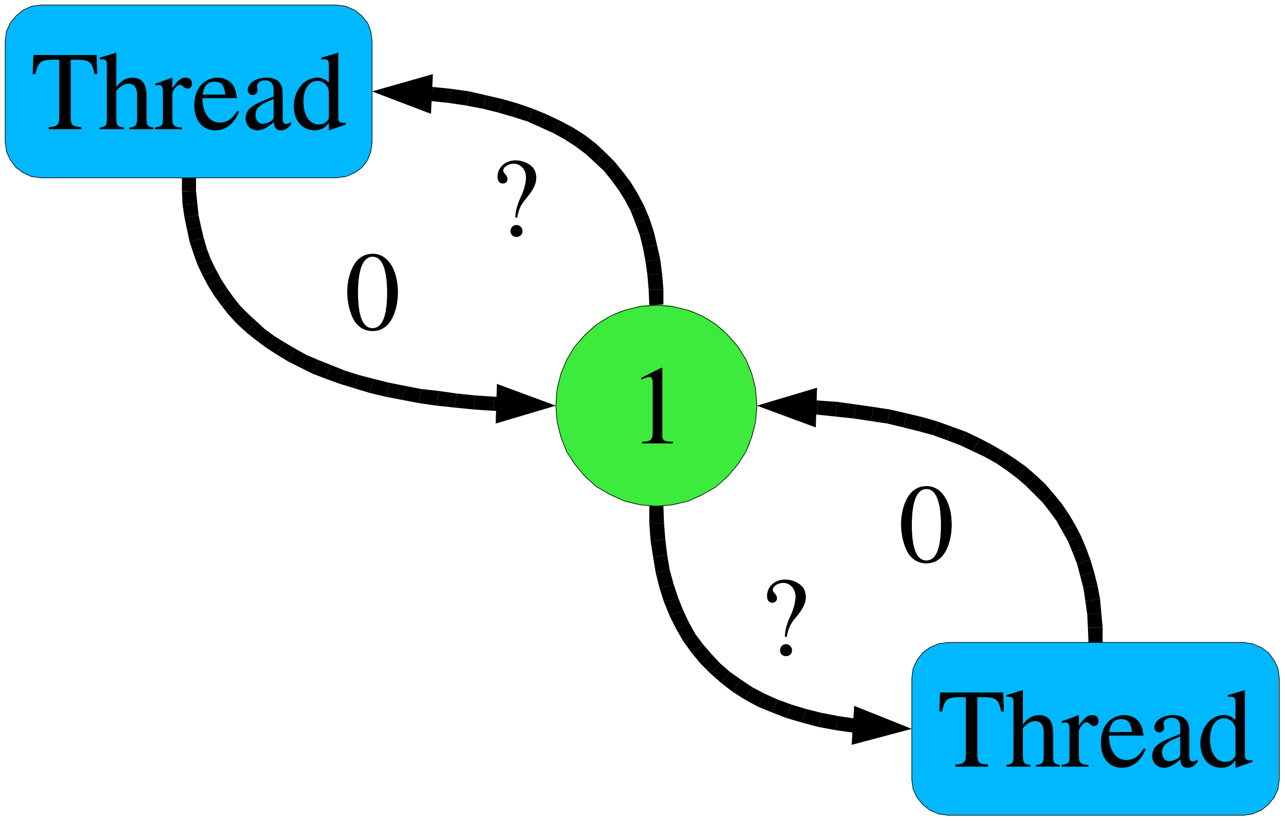
- Spin-wait

```
while (!xchg(&lock_available, 0)
  /* nothing */ ;
```

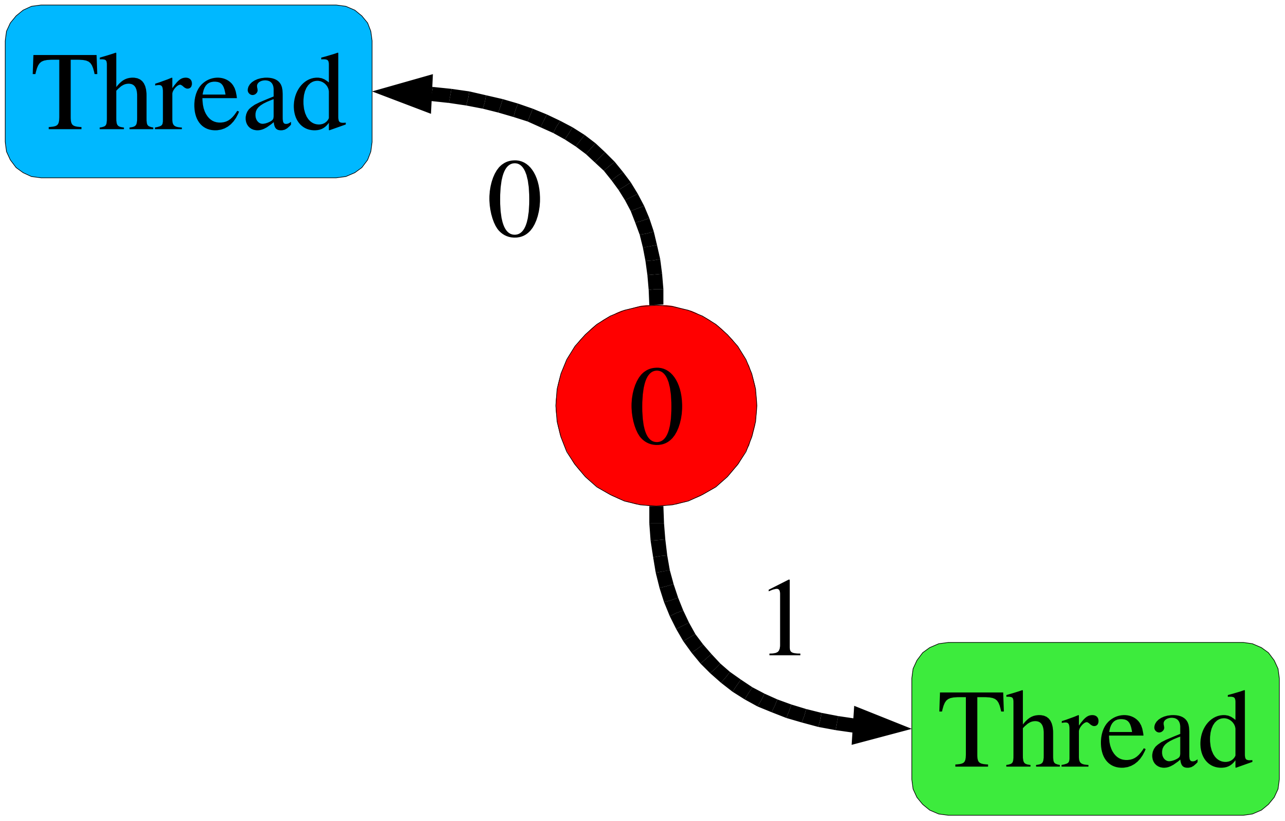
- Unlock

```
xchg(&lock_available, 1); /*expect 0*/
```

Strangers in the Night, Exchanging 0's



And the winner is...



Does it work?

[What are the questions, again?]

Does it work?

- Mutual Exclusion
- Progress
- Bounded Waiting

Does it work?

- Mutual Exclusion
  - Only one thread can see lock_available == 1
- Progress
  - Whenever lock_available == 1 a thread will get it
- Bounded Waiting
  - ***No***
  - A thread can lose ***arbitrarily many times***

Attaining Bounded Waiting

- Lock

```
waiting[i] = true;
got_it = false;
while (waiting[i] && !got_it)
  got_it = xchg(&lock_available,
                false);
waiting[i] = false;
```

Attaining Bounded Waiting

- Unlock

```
j = (i + 1) % n;
while ((j != i) && !waiting[j])
  j = (i + 1) % n;
  if (j == i)
    xchg(&lock_available, true); /*W*/
  else
    waiting[j] = false;
```

Attaining Bounded Waiting

- Versus textbook
  - Swap vs. TestAndSet
  - “Available” vs. “locked”
  - Atomic release vs. normal memory write
    - Locker does XCHG, unlocker does too
    - ***Mandatory*** on many shared-memory processors
    - Text does “blind write” at point “W”

Evaluation

- One awkward requirement
- One unfortunate behavior

Evaluation

- One awkward requirement
  - Everybody knows size of thread population
    - Always & instantly!
    - Or uses an upper bound
- One unfortunate behavior
  - Recall: expect ***zero*** competitors
  - Algorithm: O(n) in ***maximum possible*** competitors
- Am I too demanding?
  - After all, Baker's Algorithm has these misfeatures...

Uniprocessor Environment

- Lock
  - What if xchg() didn't work the first time?
  - Some other process has the lock
    - That process isn't running (because you are)
    - ***xchg() loop is a waste of time***
- Unlock
  - What about bounded waiting?
  - Next xchg() winner “chosen” by thread scheduler
  - How capricious are real thread schedulers?

Multiprocessor Environment

- Lock
  - Spin-waiting probably justified
    - (why?)
- Unlock
  - Next xchg() winner “chosen” by memory hardware
  - How capricious are real memory controllers?

Test&Set

```
boolean testandset(int32 *lock) {
register boolean old;
  old = *lock;  /* bus is locked */
  *lock = true; /* bus is locked */
  return (old);
}
```

- Conceptually simpler than XCHG?
  - Or not

Load-linked, Store-conditional

- For multiprocessors
  - “Bus locking considered harmful”
- Split XCHG into halves
  - ***Load-linked*** fetches old value from memory
  - ***Store-conditional*** stores new value
    - If nobody else did
- Your cache “snoops” the bus
  - Better than locking it!

Intel i860 magic lock bit

- Instruction sets processor in “lock mode”
  - Locks bus
  - Disables interrupts
- Isn't that dangerous?
  - 32-cycle countdown timer triggers unlock
  - Exception triggers unlock
  - Memory write triggers unlock

Mutual Exclusion: Software

- Lamport's “Fast Mutual Exclusion” algorithm
  - 5 writes, 2 reads (if no contention)
  - Not bounded-waiting (in theory, i.e., if contention)
  - http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-7.html
- Why not use it?
  - What ***kind*** of memory writes/reads?

Passing the Buck

- Q: Why not ask the OS to provide mutex_lock()?
- Easy on a uniprocessor...
  - Kernel ***automatically*** excludes other threads
  - Kernel can easily disable interrupts
- Kernel has special power on a multiprocessor
  - Can issue “remote interrupt” to other CPUs
- So why ***not*** rely on OS?

Passing the Buck

- A: Too expensive
  - Because... (you know this song!)

Mutual Exclusion: *Tricky* Software

- Fast Mutual Exclusion for Uniprocessors
  - Bershad, Redell, Ellis: ASPLOS V (1992)
- Want uninterruptable instruction sequences?
  - Pretend!

```
scash = store->cash;
scash += 10;
wallet -= 10;
store->cash = scash;
```

  - *Usually* won't be interrupted...

How can that work?

- Kernel ***detects*** “context switch during sequence”
  - Maybe a small set of instructions
  - Maybe particular memory areas
  - Maybe a flag

```
no_interruption_please = 1;
```

- Kernel ***handles*** unusual case
  - Hand out another time slice? (Is that ok?)
  - Simulate unfinished instructions (yuck)
  - Idempotent sequence: slide PC back to start

Review

- Atomic instruction sequence
  - Nobody else may interleave same/”related” sequence
  - ***Short*** sequence of instructions
    - Ok to force competitors to wait
  - Probability of collision is “low”
    - Avoid expensive exclusion method
- Voluntary de-scheduling
  - Can't proceed with this world state
  - ***Unlock*** world, ***yield*** CPU: other threads enable us