

**15-410**

***“Strangers in the night...”***

**Synchronization #2**  
**Jan. 30, 2009**

**Dave Eckhardt**

# Synchronization

## Project 1 due Monday

- (but you knew that)
- Again, please try your hand-in directory *early*

# Synchronization

## **Register your project partner –sooner is better**

- “Partner registration” page on Projects page
- If you know your partner today, please register today
  - You'll get your shared AFS space sooner
  - Your classmates will appreciate it

# 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

## Upcoming

- Inside voluntary descheduling
- Project 2 –thread library

# 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”

# Interfering Code Sequences

<i>Customer</i>	<i>Delivery</i>
<code>cash = store-&gt;cash;</code>	<code>cash = store-&gt;cash;</code>
<code>cash += 50;</code>	<code>cash -= 2000;</code>
<code>wallet -= 50;</code>	<code>wallet += 2000;</code>
<code>store-&gt;cash = cash;</code>	<code>store-&gt;cash = cash;</code>

Which sequences interfere?

“Easy”: Customer interferes with Customer

Also: Delivery interferes with Customer

# Mutex aka Lock aka Latch

**Specify interfering code sequences via *an object***

- Data item(s) “protected by the mutex”

**Object 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 the object?**

# 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);
```

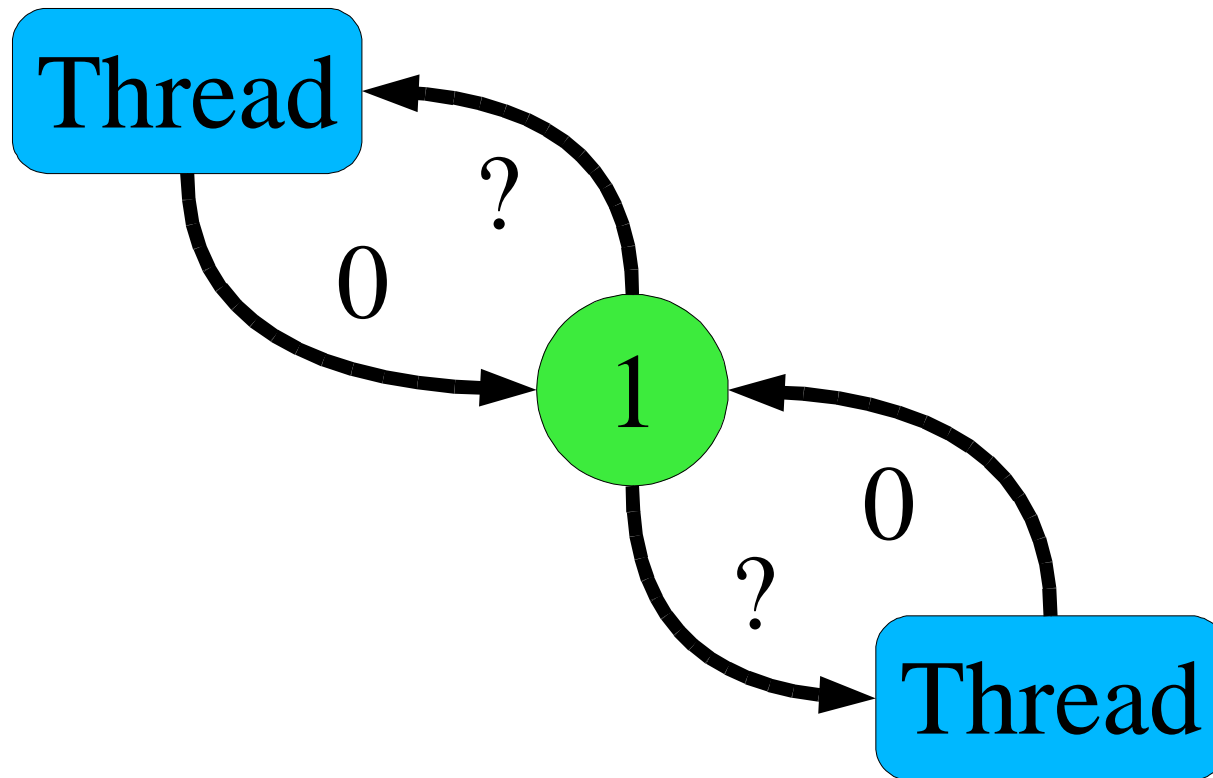
## Spin-wait

```
while (!xchg(&lock_available, 0))  
    continue;
```

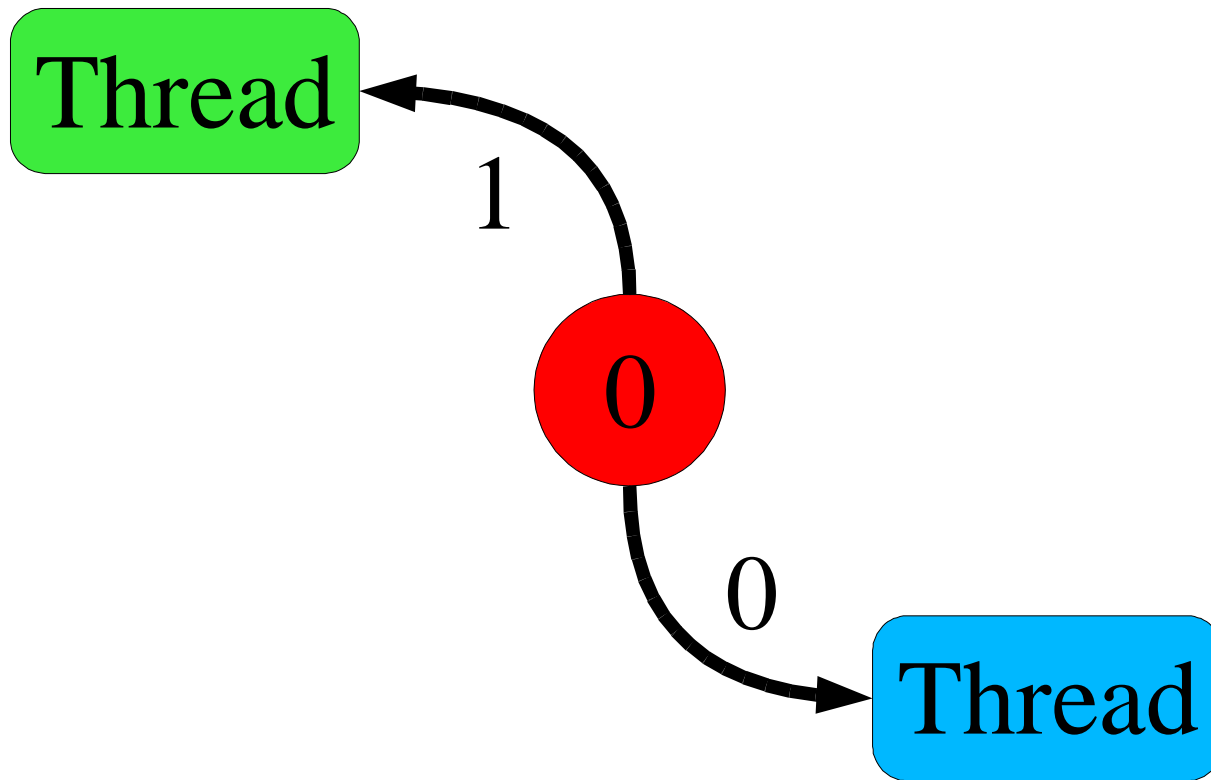
## Unlock

```
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

- There's only one 1; 1's are conserved
- Only one thread can see `lock_available == 1`

# Does it work?

## Mutual Exclusion

- There's only one 1; 1's are conserved
- Only one thread can see `lock_available == 1`

## Progress

- Whenever `lock_available == 1` some thread will get it

# Does it work?

## Mutual Exclusion

- There's only one 1; 1's are conserved
- Only one thread can see `lock_available == 1`

## Progress

- Whenever `lock_available == 1` some thread will get it

## Bounded Waiting

- *No*
- A thread can lose *arbitrarily many times*

# Ensuring Bounded Waiting

## Intuition

- Lots of people might XCHG “at the same time”
- We need a system with some “taking turns” nature

## Possible approaches

- Make sure the next lock-acquisition race condition party has a “fair outcome”
  - This may not be obvious
- Add fairness via the lock *release* procedure
  - Somebody is “in charge”; let's leverage that

# Ensuring Bounded Waiting

## Lock

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

# Ensuring Bounded Waiting

## Unlock

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

# Ensuring Bounded Waiting

## Versus (previous edition of) textbook

- Exchange vs. TestAndSet
- “Available” vs. “locked”
- Atomic release vs. normal memory write
  - Text does “blind write” at point “W”  
`lock_available = true;`
  - This may be illegal on some machines
  - Unlocker may be *required* to use special memory access
    - Exchange, TestAndSet, etc.

# 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

## Is this criticism too harsh?

- After all, Baker's Algorithm has these misfeatures...

# Looking Deeper

## Look beyond abstract semantics

- Mutual exclusion, progress, bounded waiting

## Consider

- *Typical* access pattern
- *Particular* runtime environments

## Environment

- Uniprocessor vs. Multiprocessor
  - Who is doing what when we are trying to lock/unlock?
- Threads aren't mysteriously “running” or “not running”
  - Decision made by scheduling algorithm with properties

# Uniprocessor Environment

## Lock

- What if `xchg()` didn't work the first time?

# Uniprocessor Environment

## Lock

- What if `xchg()` didn't work the first time?
- Some other process has the lock
  - That process isn't running (because we are)
  - *`xchg()` loop is a waste of time*
  - We should let the lock-holder run instead of us

# Uniprocessor Environment

## Lock

- What if `xchg()` didn't work the first time?
- Some other process has the lock
  - That process isn't running (because we are)
  - *`xchg()` loop is a waste of time*
  - We should let the lock-holder run instead of us

## Unlock

- What about bounded waiting?
- When we mark mutex available, who wins next?

# Uniprocessor Environment

## Lock

- What if `xchg()` didn't work the first time?
- Some other process has the lock
  - That process isn't running (because we are)
  - *`xchg()` loop is a waste of time*
  - We should let the lock-holder run instead of us

## Unlock

- What about bounded waiting?
- When we mark mutex available, who wins next?
  - Whoever *runs* next..only one at a time! (“Fake competition”)
  - How unfair are real OS kernel thread schedulers?
  - If scheduler is vastly unfair, the right thread will *never* run!

# Multiprocessor Environment

## Lock

- Spin-waiting probably justified
  - (why?)

## Unlock

- Next `xchg()` winner “chosen” by memory hardware
- How unfair 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

## Other x86 instructions

- XADD, CMPXCHG, CMPXCHG8B, ...

- See “Locked Atomic Operations” in intel-sys.pdf (ISR, too)

# Load-linked, Store-conditional

## For multiprocessors

- “Bus locking considered harmful”

## Split XCHG into halves

- *Load-linked(addr)* fetches old value from memory
- *Store-conditional(addr, val)* stores new value back
  - If nobody else stored to that address in between
  - If so, instruction “fails” (sets an error code)

# Load-linked, Store-conditional

```
loop:  LL    R3, mutex_addr
      BEQ   R3, $0, loop      # avail == 0
      LI    R3, 0             # prep. 0
      SC    R3, mutex_addr    # write 0?
      BEQ   R3, $0, loop      # aborted...
```

## Your cache “snoops” the shared memory bus

- Locking would shut down *all* memory traffic
- Snooping allows all traffic, watches for *conflicting* traffic
- Are aborts “ok”? *When* are they “ok”?

# Intel i860 magic lock bit

## Instruction sets processor in “lock mode”

- Locks bus
- Disables interrupts

## Isn't that dangerous?

- 32-instruction countdown timer triggers exception
- Any exceptions (page fault, zero divide, ...) unlock bus

## Why would you want this?

- Implement test&set, compare&swap, semaphore –you choose

# Mutual Exclusion: Inscrutable 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>

## Cool magic - why not use it?

- What *kind* of memory writes/reads?
- Remember, the computer is “modern”...

# Passing the Buck?

**Q: Why not ask the OS for mutex\_lock() *system call*?**

**Easy on a uniprocessor...**

- Kernel *automatically* excludes other threads
- Kernel can easily disable interrupts
- No need for messy unbounded loop, weird XCHG...

**Kernel has special power on a multiprocessor**

- Can issue “remote interrupt” to other CPUs
- No need for messy unbounded loop...

**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

- Bershada, Redell, Ellis: ASPLOS V (1992)

## Want uninterruptable instruction sequences?

- Pretend!

```
scash = store->cash;
```

```
scash += 10;
```

```
wallet -= 10;
```

```
store->cash = scash;
```

- Uniprocessor: interleaving requires thread switch...
- Short sequence *almost always* won't be interrupted...

# How can that work??

## Kernel *detects* “context switch in atomic 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?)
- Hand-simulate unfinished instructions (yuck?)
- “Idempotent sequence”: slide PC back to start

# Summary

## Atomic instruction sequence

- Nobody else may interleave same/"related" sequence

## Specify interfering sequences via *mutex object*

### Inside a mutex

- Last time: race-condition memory algorithms
- Atomic-exchange, Compare&Swap, Test&Set, ...
- Load-linked/Store-conditional
- Tricky software, weird software

### Mutex strategy

- How should you behave given runtime environment?