15-410
“Strangers in the night...”

Synchronization #2
Sep. 14, 2018

Dave Eckhardt
Dave O'Hallaron
Synchronization

Project 1 due tonight
Synchronization

Project 1 due tonight
- (something like that, anyway)
- Again, please try your hand-in directory *early*
Synchronization

**TOC vs. travel blackouts**

- If you provide a prospective employer with a list of blackout dates before a plane ticket is purchased, trouble can be avoided.
- By now it should be possible for you to estimate blackout dates for most of your classes.
Synchronization

Pass/fail?

- If you are considering switching to pass/fail, this has potentially serious implications for your project partner
- Unless *both* of you are agreed on this, please see me after class today
  - Maybe a brokered partner swap is in order
Outline

Last time
- Two building blocks for threaded programs
- Three requirements for critical-section mechanisms
- Algorithms people *don't* use for critical sections

Today
- Ways to *really* solve the critical-section problem

Upcoming
- Inside voluntary descheduling
- Project 2 – thread library
Critical Section: Reminder

Protects an “atomic instruction sequence”

- We must “do something” to guard against
  - Our CPU switching to another thread
  - A thread running on another CPU

Assumptions

- Atomic instruction sequence will be “short”
- No other thread “likely” to compete
Critical Section: Goals

Typical case (no competitor) should be fast

Atypical case can be slow
  - Should not be “too wasteful”
# Interfering Code Sequences

<table>
<thead>
<tr>
<th>Customer</th>
<th>Delivery</th>
</tr>
</thead>
<tbody>
<tr>
<td>cash = store-&gt;cash;</td>
<td>cash = store-&gt;cash;</td>
</tr>
<tr>
<td>cash += 50;</td>
<td>cash -= 2000;</td>
</tr>
<tr>
<td>wallet -= 50;</td>
<td>wallet += 2000;</td>
</tr>
<tr>
<td>store-&gt;cash = cash;</td>
<td>store-&gt;cash = cash;</td>
</tr>
</tbody>
</table>

**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?
Atomic Exchange

Intel x86 XCHG instruction
- intel-isr.pdf page 754

xchg (%esi), %edi

```c
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

Thread

0

1

0

Thread

?
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 approach**
- Make sure each lock-acquisition XCHG race-condition party has a “fair outcome”
  - Accomplishing this may not be obvious
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 each lock-acquisition XCHG race-condition party has a “fair outcome”
  - Accomplishing 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)
    // “spin on XCHG”, keep the bus warm
    got_it = xchg(&lock_available,
                   false);
waiting[i] = false;
return; // Success: in critical section
Ensuring Bounded Waiting

**Unlock**

\[ j = (i + 1) \mod n; \]

```c
while ( (j != i) && !waiting[j] )
    j = (j + 1) \mod n;
if (j == i)
    xchg(&lock_available, true); /*W*/
else
    waiting[j] = false;
return;
```
Ensuring Bounded Waiting

**Possible variations**

- Exchange vs. TestAndSet
- Field name is “available” vs. “locked”
- Atomic release vs. normal memory write
  - Some people do “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 a 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 can be justified
  - (why?)
Multiprocessor Environment

Lock
- Spin-waiting can be justified
  - (why?)

Unlock
- Next xchg() winner “chosen” by memory hardware
- How unfair are real memory controllers?
Test&Set

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

Conceptually simpler than XCHG??

Other x86 instructions

- XADD, CMPXCHG, CMPXCHG8B, ...
- See “Locked Atomic Operations” in intel-sys.pdf
- We expect you to consult intel-sys and intel_isr about this
Load-linked/Store-conditional

For multiprocessors
- “Bus locking considered harmful”

Split XCHG into two halves
- \textit{Load-linked}(addr) fetches old value from memory
- \textit{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

lock: LA R1, mutex # &mutex in R1
loop: LL R2, 0(R1) # mutex->avail
       BEQ R2, R0, loop # avail == 0?
       MOV R3, R0 # prepare 0
       SC 0(R1), R3 # write 0?
       BEQ R3, R0, 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)

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**
- Bershad, Redell, Ellis: ASPLOS V (1992)

**Want uninterruptable instruction sequences?**
- Pretend!
  
  \[
  \text{scash} = \text{store}->\text{cash}; \\
  \text{scash} += 10; \\
  \text{wallet} -= 10; \\
  \text{store}->\text{cash} = \text{scash};
  \]
- 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?