15-410
“...Arguably less wrong...”

Synchronization #3
Sep. 18, 2017

Dave Eckhardt
Dave O'Hallaron
Synchronization

**Project 1 due tonight**
- Again, try not to use a late day
  - But if you do, please carefully read and follow the instructions

**Project 2 out Wednesday**
Outline

Synch 1
- Two building blocks
- Three requirements for critical-section algorithms
- Algorithms people *don't* use for critical sections

Synch 2
- How critical sections are really implemented

Synch 3
- Condition variables
  - Under the hood
  - The atomic-sleep problem
- Semaphores, monitors – overview
Road Map

Two Fundamental operations

✔ Atomic instruction sequence
▷ Voluntary de-scheduling
Voluntary de-scheduling

The Situation

– You hold lock on shared resource
– But it's not in “the right mode”

Action sequence

– Unlock shared resource
– Write down “wake me up when...”
– Block until resource changes state
What **Not** To Do

```c
while (!reckoning) {
    mutex_lock(&scenario_lk);
    if ((date >= 1906-04-18) &&
        (hour >= 5))
        reckoning = true;
    else
        mutex_unlock(&scenario_lk);
}
wreak_general_havoc();
mutex_unlock(&scenario_lk);
```
What **Not** To Do

```c
while (!reckoning) {
    mutex_lock(&scenario lk);
    if ((date >= 1906-04-18) &&
        (hour >= 5))
        reckoning = true;
    else
        mutex_unlock(&scenario lk);
}
wreak_general_havoc();
mutex_unlock(&scenario lk);
```

Is this code correct?
What **Not** To Do

```c
while (!reckoning) {
    mutex_lock(&scenario_lk);
    if ((date >= 1906-04-18) &&
        (hour >= 5))
        reckoning = true;
    else
        mutex_unlock(&scenario_lk);
}
wreak_general_havoc();
mutex_unlock(&scenario_lk);
```
What *Not* To Do

**Why is this wrong?**

- Make sure you understand!
- See previous two lectures
- Do *not* do this in P2 or P3
  - Not even if it is *really tempting* in P3
while (!reckoning) {
    mutex_lock(&scenario_lk);
    if ((date >= 1906-04-18) &&
        (hour >= 5))
        reckoning = true;
    else {
        mutex_unlock(&scenario_lk);
        sleep(1);
    }
}

wreak_general_havoc();
mutex_unlock(&scenario_lk);
“Arguably Less Wrong”

Don't do this either
- How wrong is “sleep(1)”?
“Arguably Less Wrong”

Don't do this either

How wrong is “sleep(1)”?

- N-1 times it's much too short
- Nth time it's much too long
“Arguably Less Wrong”

Don't do this either

- How wrong is “sleep(1)”?
  - N-1 times it's much too short
  - Nth time it's much too long
  - It's wrong *every time*
“Arguably Less Wrong”

Don't do this either

- How wrong is “sleep(1)”?
  - N-1 times it's much too short
  - Nth time it's much too long
  - It's wrong *every time*
- What's the problem?
“Arguably Less Wrong”

Don't do this either

- How wrong is “sleep(1)”?
  - N-1 times it's much too short
  - Nth time it's much too long
  - It's wrong every time

- What's the problem?
  - We don't really want to wait for some duration!
  - We want to wait for a condition change
“Honorable Mention”?

```c
while (!reckoning) {
    mutex_lock(&scenario_lk);
    if ((date >= 1906-04-18) &&
        (hour >= 5))
        reckoning = true;
    else {
        mutex_unlock(&scenario_lk);
        yield(); // Better than sleep()???
    }
}
wreak_general_havoc();
mutex_unlock(&scenario_lk);
```
Something Is Missing...

✔ “Protect shared state” is solved
  - We use a “mutex object”
  - Also encapsulates “Which code interferes with this?”
  - Good

❖ How to solve “block for the right duration”? 
Something Is Missing

✔ “Protect shared state” is solved
  - We use a “mutex object”
  - Also encapsulates “Which code interferes with this?”
  - Good

 böliev How to solve “block for the right duration”? 
  - Get an expert to tell us!
  - Encapsulate “the right duration”...
    • ...into a condition variable object
Once More, With Feeling!

```c
mutex_lock(&scenario_lk);
while (cvarp = wait_on()) {
    cond_wait(cvarp, &scenario_lk);
}
wreak_general_havoc(); /* locked! */
mutex_unlock(&scenario_lk);
```
wait_on()?

if (y < 1906)
    return (&new_year);
else if (m < 4)
    return (&new_month);
else if (d < 18)
    return (&new_day);
else if (h < 5)
    return (&new_hour);
else
    return (0); // done!

// Code is "conceptual example", not 100% correct
What Awakens Us?

for (y = 1900; y < 2000; y++)
  for (m = 1; m <= 12; m++)
    for (d = 1; d <= days(m); d++)
      for (h = 0; h < 24; h++)
        ...
        cond_broadcast(&new_hour);
        cond_broadcast(&new_day);
        cond_broadcast(&new_month);
        cond_broadcast(&new_year);

// Code is “conceptual example”, not 100% correct
Condition Variable Requirements

- Keep track of threads blocked “for a while”
- Allow notifier thread(s) to unblock blocked thread(s)
- Must be “thread-safe”
  - Many threads may call condition_wait() at same time
  - Many threads may call condition_signal() at same time
  - Say, those look like “interfering sequences”...
Why *Two* Parameters?

```c
condition_wait(&cvar, &mutex);
```

Mutex required to examine/modify the “world” state
- If you examine unlocked state, it's changing.

Whoever awakens you will need to hold that mutex
- So you'd better give it up.

When you wake up, you will need to hold it again
- “Convenient” for condition_wait() to un-lock/re-lock

But there's something more subtle
- Try to recall this issue when working on P2...
Inside a Condition Variable

**cvar->queue**
- of blocked threads
- FIFO, or more exotic

**cvar->mutex**
- Protects queue against interfering `wait()`/`signal()` calls
- This isn't the caller's mutex (locking caller's world state)
- This is our secret invisible mutex
Inside a Condition Variable

```c
cond_wait(cvar, world_mutex)
{
    lock(cvar->mutex);
    enq(cvar->queue, my_thread_id());
    unlock(world_mutex);
    ATOMICALLY {
        unlock(cvar->mutex);
        kernel_please_pause_this_thread();
    }
    lock(world_mutex);
}
```

What is this “ATOMICALLY” stuff?
**What We Hope For**

<table>
<thead>
<tr>
<th>$\text{cond_wait}(m, c);$</th>
<th>$\text{cond_signal}(c);$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{enq}(c-&gt;\text{que},\ me);$</td>
<td></td>
</tr>
<tr>
<td>$\text{unlock}(m);$</td>
<td></td>
</tr>
<tr>
<td>$\text{unlock}(c-&gt;m);$</td>
<td></td>
</tr>
<tr>
<td>$\text{kern_thr_pause}();$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\text{lock}(c-&gt;m);$</td>
</tr>
<tr>
<td></td>
<td>$\text{id} = \text{deq}(c-&gt;\text{que});$</td>
</tr>
<tr>
<td></td>
<td>$\text{kern_thr_wake}(\text{id});$</td>
</tr>
<tr>
<td></td>
<td>$\text{unlock}(c-&gt;m);$</td>
</tr>
</tbody>
</table>
Pathological Execution Sequence

<table>
<thead>
<tr>
<th>$\text{cond_wait}(m, c)$;</th>
<th>$\text{cond_signal}(c)$;</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{enq}(c\to\text{que}, \text{me})$;</td>
<td></td>
</tr>
<tr>
<td>$\text{unlock}(m)$;</td>
<td></td>
</tr>
<tr>
<td>$\text{unlock}(c\to\text{m})$;</td>
<td>$\text{lock}(c\to\text{m})$;</td>
</tr>
<tr>
<td></td>
<td>$\text{id} = \text{deq}(c\to\text{que})$;</td>
</tr>
<tr>
<td></td>
<td>$\text{kern_thr_wake}(\text{id})$;</td>
</tr>
<tr>
<td>$\text{kern_thr_pause}()$;</td>
<td>$\text{unlock}(c\to\text{m})$;</td>
</tr>
</tbody>
</table>

$kern\_thr\_wake(\text{id}) \Rightarrow ERR\_NOT\_ASLEEP$
Achieving “wait() Atomicity”

Rules of the game
- There isn't an underlying unlock_and_block() primitive
- We have unlock(), and block(), and maybe “other stuff”
- From outside cond_wait()/cond_signal(), we must achieve *apparent* (as-if) “atomicity of unlock and block”.

Approaches
- Disable interrupts (if you are a kernel)
- Rely on OS to implement condition variables
  - (Why is this not the best idea?)
- Have a better kernel thread-block interface
- Hmmmm....
Achieving “wait() Atomicity”

**P2 challenges**

- Understand the issues!
  - mutex, cvar
- Understand the host kernel we give you
- Put the parts together
  - Don't use “wrong” or “arguably less wrong” approaches!
  - Seek solid, clear solutions
    - There's more than one way to do it
    - Make sure to pick a correct way...
    - Try to pick a *good* way.
Outline

Last time
  - How mutual exclusion is really implemented

Condition variables
  - Under the hood
  - The atomic-sleep problem

⇒ Semaphores

Monitors
Semaphore Concept

Semaphore is a different encapsulation object
  - Can produce mutual exclusion
  - Can produce block-until-it's-time

Intuition: counted resource
  - Integer represents “number available”
    • Number of buffers, number of pairs of scissors, ...
    • Semaphore object initialized to a particular count
  - Thread blocks until it is allocated an instance
Semaphore Concept

**wait(), aka P(), Dutch probeer te verlagen (“try to decrease”)**
- wait until value > 0
- then decrement value (“taking” one instance)

**signal(), aka V(), Dutch verhogen (“increase”)**
- increment value (“releasing” one instance)

**Just one small issue...**
- wait() and signal() **must be atomic**
"Mutex-style" Semaphore

semaphore m = 1;

do {
    wait(m); /* mutex_lock() */
    ..critical section...
    signal(m); /* mutex_unlock() */

    ...remainder section...
} while (1);
“Condition-style” Semaphore

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>wait(c);</td>
</tr>
<tr>
<td>result = 42;</td>
<td></td>
</tr>
<tr>
<td>signal(c);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>use(result);</td>
</tr>
</tbody>
</table>
“Condition with Memory”

Semaphores *retain memory* of signal() events

“full/empty bit” - *unlike* condition variables

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>result = 42;</td>
<td></td>
</tr>
<tr>
<td>wait(c);</td>
<td>use(result);</td>
</tr>
</tbody>
</table>
Semaphore vs. Mutex/Condition

**Good news**
- Semaphore is a higher-level construct
- Integrates mutual exclusion, waiting
- Avoids mistakes common in mutex/condition API
  - `signal()` too early is “lost”
  - ...
Semaphore vs. Mutex/Condition

Bad news

- Semaphore is a higher-level construct
- Integrates mutual exclusion, waiting
  - Some semaphores are “mutex-like”
  - Some semaphores are “condition-like”
  - How’s a poor library to know?
    - Spin-wait or not???
Semaphores - 31 Flavors

**Binary semaphore**
- It counts, but only from 0 to 1!
  - “Available” / “Not available”
- Consider this a hint to the implementor...
  - “Think mutex!”

**Non-blocking semaphore**
- wait(semaphore, timeout);

**Deadlock-avoidance semaphore**
- #include <deadlock.lecture>
My Personal Opinion

One “simple, intuitive” synchronization object
  - In 31 performance-enhancing flavors!!!

“The nice thing about standards is that you have so many to choose from.”
  - Andrew S. Tanenbaum

Conceptually simpler to have two objects
  - One for mutual exclusion
  - One for waiting
  - ...after you've understood what's actually happening
Semaphore Wait: Inside Story

```c
wait(semaphore s)
    ACQUIRE EXCLUSIVE ACCESS
    --s->count;
    if (s->count < 0) {
        enqueue(s->queue, my_id());
        ATOMICALLY {
            RELEASE EXCLUSIVE ACCESS
            thread_block()
        }
    } else
    } else
        RELEASE EXCLUSIVE ACCESS
```
Semaphore Signal: Inside Story

```c
signal(semaphore s)
    // ACQUIRE EXCLUSIVE ACCESS
    ++s->count;
    if (s->count <= 0) {
        tid = dequeue(s->queue);
        thread_unblock(tid);
    }
    // RELEASE EXCLUSIVE ACCESS
```

**What's all the shouting?**
- An exclusion algorithm much like a mutex, or
- OS-assisted atomic de-scheduling / awakening
Monitor

Basic concept

- Semaphores eliminate some mutex/condition mistakes
- Still some common errors
  - Swapping “signal()” & “wait()”
  - Accidentally omitting one

Monitor: higher-level abstraction

- Module of high-level language procedures
  - All access some shared state
- Compiler adds synchronization code
  - Thread running in any procedure blocks all thread entries
Monitor "commerce"

```java
int cash_in_till[N_STORES] = { 0 };
int wallet[N_CUSTOMERS] = { 0 };

boolean buy(int cust, store, price) {
    if (wallet[cust] >= price) {
        cash_in_till[store] += price;
        wallet[cust] -= price;
        return (true);
    } else
        return (false);
}
```
Monitors – What about waiting?

Automatic mutual exclusion is nice...
- ...but it is too strong

Sometimes one thread needs to wait for another
- Automatic mutual exclusion forbids this
- Must leave monitor, re-enter - *when*?

Have we heard this “when” question before?
Monitor Waiting – The Problem

```c
void
stubbornly_cash_check(acct a, check c) {
    while (account[a].bal < check.val) {
        ...Sigh, must wait for a while...
        ...What goes here? I forget...
    }
    account[a].bal -= check.val;
}
```
Monitor Waiting – Wrong Solution

boolean
try_cash_check(acct a, check c)
{
    if (account[a].bal < check.val)
        return (false); /* pass the buck */
    account[a].bal -= check.val;
    return (true);
}

Monitor condition variables

Similar to condition variables we've seen

condition_wait(cvar)

- Only one parameter
- Mutex-to-drop is implicit
  - (the “monitor mutex”)
- Operation
  - “Temporarily exit monitor” -- drop the mutex
  - Wait until signalled
  - “Re-enter monitor” - re-acquire the mutex
Monitor Waiting

```c
void stubbornly_cash_check(acct a, check c) {
    while (account[a].bal < check.val) {
        cond_wait(account[a].activity);
    }
    account[a].bal -= check.val;
}
```

Q: Who would `signal()` this cvar?
Monitor condition variables

signal() policy question - which thread to run?

- Signalling thread? Signalled thread?
  - Can argue either way
- Or: signal() exits monitor as side effect!
- Different signal() policies mean different monitor flavors
Two fundamental operations

- Mutual exclusion for must-be-atomic sequences
- Atomic de-scheduling (and then wakeup)

Mutex/condition-variable ("pthreads") style

- Two objects for two core operations

Semaphores, Monitors

- Semaphore: one object
- Monitor: invisible compiler-generated object
- Same core ideas inside
Summary

What you should know

- Issues/goals
- Underlying techniques
- How environment/application design matters

All done with synchronization?

- Only one minor issue left
  - Deadlock