

# 15-410

*“...Arguably less wrong...”*

Synchronization #3  
Jan. 31, 2005

**Dave Eckhardt**

**Bruce Maggs**

# Outline

## Synch 1

- Two building blocks
- Three requirements for mutual exclusion
- Algorithms people *don't* use for mutual exclusion

## Synch 2

- How mutual exclusion is really implemented

## Synch 2

- Condition variables
  - Under the hood
  - The atomic-sleep problem
- Semaphores, monitors – overview

# Road Map

## Two Fundamental operations

√ Atomic instruction sequence

⇒ Voluntary de-scheduling

# Synchronization

## Project 0

- Scores posted in your “grades” directory
- How to read your score
  - Low overall grade (under 85%) should be a warning...
  - Low style/structure grade (under 7/10) should be a warning even if overall grade is “ok”
    - Poorly-structured code tends to have more bugs...
    - ...and they tend to be messier bugs to find.
    - You won't have a lot of extra time in P2 and P3 for extra bugs.

# 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...”
- Go to sleep until resource changes state

# What *not* to do

```
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

# Arguably Less Wrong

```
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();
- 8 - mutex_unlock(&scenario_lk);
```



# Arguably less wrong

## Don't do this either

- How wrong is “a while”?
  - 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 a duration!
  - We want to wait for a *condition*

# Something is missing

## Mutex protects shared state

- Also encapsulates “interfering code sequence” as object
- Good

## How can we sleep for the *right* duration?

- Get an expert to tell us!
- Encapsulate “the right duration”
  - ...into a *condition variable* object

# Once more, with feeling!

```
mutex_lock(&scenario_lk);  
while (cvar = wait_on()) {  
    cond_wait(&scenario_lk, &cvar);  
}  
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);
```

# What wakes us up?

```
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);
```

# Condition Variable Requirements

**Keep track of threads asleep “for a while”**

**Allow notifier thread to wake sleeping 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?

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

**Lock required to access/modify the “world” state**

**Whoever awakens you will need to hold that lock**

- 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**

# Inside a Condition Variable

## **cvar->queue - of sleeping processes**

- FIFO or more exotic

## **cvar->mutex**

- Protects queue against interfering wait()/signal() calls
- This isn't the caller's mutex (locking client's world state)
- This is our secret invisible mutex



# Inside a Condition Variable

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

**What is this “ATOMICALLY” stuff?**

# What We Hope For

<i>cond_wait(m, c);</i>	<i>cond_signal(c);</i>
<code>enq(c-&gt;que, me);</code>	
<code>unlock(m);</code>	
<code>unlock(c-&gt;m);</code>	
<code>kern_thr_pause();</code>	
	<code>lock(c-&gt;m);</code>
	<code>id = deq(c-&gt;que);</code>
	<code>thr_wake(id);</code>
	<code>unlock(c-&gt;m);</code>

# Pathological Execution Sequence

<i>cond_wait(m, c);</i>	<i>cond_signal(c);</i>
<code>enq(c-&gt;que, me);</code>	
<code>unlock(m);</code>	
<code>unlock(c-&gt;m);</code>	
	<code>lock(c-&gt;m);</code>
	<code>id = deq(c-&gt;que);</code>
	<code>thr_wake(id);</code>
	<code>unlock(c-&gt;m);</code>
<code>kern_thr_pause();</code>	

`thr_wake(id) ⇒ ERR_NOT_ASLEEP`

# Achieving wait() Atomicity

**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-sleep interface**

# 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 sleep-until-it's-time

## **Intuition: counted resource**

- Integer represents “number available”
  - Semaphore object initialized to a particular count
- Thread blocks until it is allocated an instance

# Semaphore Concept

**wait(), aka P(), aka proberen (“wait”)**

- wait until value  $> 0$
- decrement value (“taking” one instance)

**signal(), aka V(), aka verhogen (“increment”)**

- 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

<i>Thread 0</i>	<i>Thread 1</i>
	<code>wait (c) ;</code>
<code>result = 42 ;</code>	
<code>signal (c) ;</code>	
	<code>use (result) ;</code>

# “Condition with Memory”

Semaphores *retain memory* of signal() events  
“full/empty bit”

<i>Thread 0</i>	<i>Thread 1</i>
<code>result = 42;</code>	
<code>signal(c);</code>	
	<code>wait(c);</code>
	<code>use(result);</code>

# Semaphore vs. Mutex/Condition

## Good news

- Semaphore is a higher-level construct
- Integrates mutual exclusion, waiting
- Avoids mistakes common in mutex/condition API
  - Lost signal()
  - Reversing signal() and wait()
  - ...

# 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

```
wait(semaphore s)
    ACQUIRE EXCLUSIVE ACCESS
    --s->count;
    if (s->count < 0)
        enqueue(s->queue, my_id());
    ATOMICALLY
        RELEASE EXCLUSIVE ACCESS
        thread_pause()
    else
        RELEASE EXCLUSIVE ACCESS
```



# Semaphore Signal: Inside Story

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

## What's all the shouting?

- An exclusion algorithm much like a mutex, or
- OS-assisted atomic de-scheduling

# 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”

```
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

```
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

```
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

# Summary

## 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