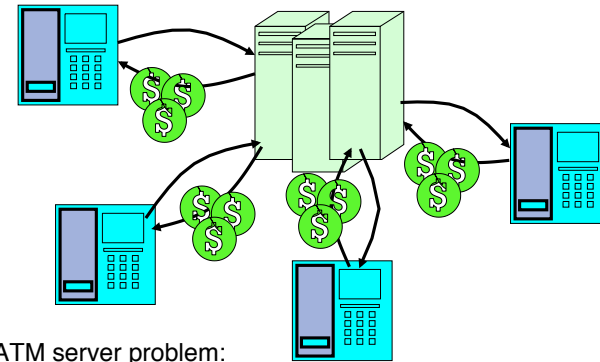


CS162  
Operating Systems and  
Systems Programming  
Lecture 4

Synchronization, Atomic operations,  
Locks

February 4, 2013  
Anthony D. Joseph  
<http://inst.eecs.berkeley.edu/~cs162>

Recap: ATM Bank Server



- ATM server problem:
  - Service a set of requests
  - Do so without corrupting database
  - Don't hand out too much money

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.2

Recap: Challenge of Threads

- Speed up server by using multiple threads (one per request)
  - Can use multi-processor, or overlap comp and I/O

- Requests proceeds to completion, blocking as required:

```
Deposit(acctId, amount) {
    acct = GetAccount(acctId); /* May use disk I/O */
    acct->balance += amount;
    StoreAccount(acct);      /* Involves disk I/O */
}
```

- Unfortunately, shared state can get corrupted:

<u>Thread 1</u>	<u>Thread 2</u>
load r1, acct->balance	load r1, acct->balance
	add r1, amount2
	store r1, acct->balance
add r1, amount1	
store r1, acct->balance	

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.3

Recap: Problem is at the lowest level

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

<u>Thread A</u>	<u>Thread B</u>
x = 1;	y = 2;

- However, What about (Initially, y = 12):

<u>Thread A</u>	<u>Thread B</u>
x = 1;	y = 2;
x = y+1;	y = y*2;

- What are the possible values of x?

<u>Thread A</u>	<u>Thread B</u>
x = 1;	
x = y+1;	
	y = 2;
	y = y*2

x=13

Preemption can occur  
at any time!

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.4

## Recap: Problem is at the lowest level

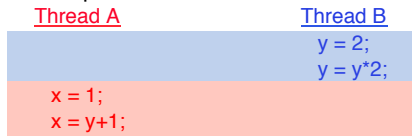
- Most of the time, threads are working on separate data, so scheduling doesn't matter:

<u>Thread A</u>	<u>Thread B</u>
x = 1;	y = 2;

- However, What about (Initially, y = 12):

<u>Thread A</u>	<u>Thread B</u>
x = 1;	y = 2;
x = y+1;	y = y*2;

- What are the possible values of x?



x=5

Preemption can occur at any time!

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.5

## Recap: Problem is at the lowest level

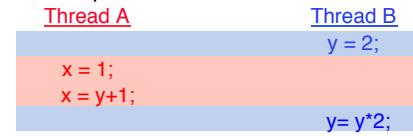
- Most of the time, threads are working on separate data, so scheduling doesn't matter:

<u>Thread A</u>	<u>Thread B</u>
x = 1;	y = 2;

- However, What about (Initially, y = 12):

<u>Thread A</u>	<u>Thread B</u>
x = 1;	y = 2;
x = y+1;	y = y*2;

- What are the possible values of x?



x=3

Preemption can occur at any time!

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.6

## Goals for Today

- Concurrency examples and sharing
- Synchronization
- Hardware Support for Synchronization

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Slides courtesy of Anthony D. Joseph, John Kubiawicz, AJ Shankar, George Necla, Alex Aiken, Eric Brewer, Ras Bodik, Ion Stoica, Doug Tygar, and David Wagner.

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.7

## Correctness Requirements

- Threaded programs must work for all interleavings of thread instruction sequences
  - Cooperating threads inherently non-deterministic and non-reproducible
  - Really hard to debug unless carefully designed!

- Example: Therac-25

- Machine for radiation therapy
  - » Software control of electron accelerator and electron beam/Xray production
  - » Software control of dosage
- Software errors caused overdoses and the death of several patients

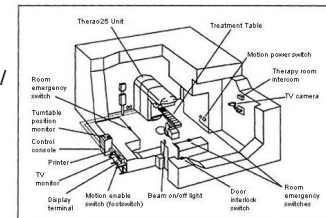


Figure 1. Typical Therac-25 facility

- » A series of race conditions on shared variables and poor software design
- » "They determined that data entry speed during editing was the key factor in producing the error condition: If the prescription data was edited at a fast pace, the overdose occurred."

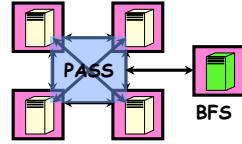
2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.8

## Space Shuttle Example

- Original Space Shuttle launch aborted 20 minutes before scheduled launch
- Shuttle has five computers:
  - Four run the “Primary Avionics Software System” (PASS)
    - » Asynchronous and real-time
    - » Runs all of the control systems
    - » Results synchronized and compared 440 times per second
  - The Fifth computer is the “Backup Flight System” (BFS)
    - » Stays synchronized in case it is needed
    - » Written by completely different team than PASS
- Countdown aborted because BFS disagreed with PASS
  - A 1/67 chance that PASS was out of sync one cycle
  - Bug due to modifications in **initialization** code of PASS
    - » A delayed init request placed into timer queue
    - » As a result, timer queue not empty at expected time to force use of hardware clock
  - Bug not found during extensive simulation



2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.9

## Atomic Operations

- To understand a concurrent program, we need to know what the underlying atomic operations are!
- **Atomic Operation**: an operation that always runs to completion or not at all
  - It is *indivisible*: it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  - Fundamental building block – if no atomic operations, then have no way for threads to work together
- On most machines, memory references and assignments (i.e. loads and stores) of words are atomic
- Many instructions are not atomic
  - Double-precision floating point store often not atomic
  - VAX and IBM 360 had an instruction to copy a whole array

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.10

## Concurrency Challenges

- Multiple computations (threads) executing in parallel to
  - share resources, and/or
  - share data
- Fine grain sharing:
  - ↑ increase concurrency → better performance
  - ↓ more complex
- Coarse grain sharing:
  - ↑ Simpler to implement
  - ↓ Lower performance
- Examples:
  - Sharing CPU for 10ms vs. 1min
  - Sharing a database at the row vs. table granularity

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.11

## Motivation: “Too much milk”

- Great thing about OS’s – analogy between problems in OS and problems in real life
  - Help you understand real life problems better
  - But, computers are much stupider than people
- Example: People need to coordinate:



Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.12

## Definitions

- **Synchronization**: using atomic operations to ensure cooperation between threads
  - For now, only loads and stores are atomic
  - We'll show that is hard to build anything useful with only reads and writes
- **Critical Section**: piece of code that only one thread can execute at once
- **Mutual Exclusion**: ensuring that only one thread executes critical section
  - One thread *excludes* the other while doing its task
  - Critical section and mutual exclusion are two ways of describing the same thing

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.13

## More Definitions

- **Lock**: prevents someone from doing something
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked
- » Important idea: all synchronization involves waiting
- Example: fix the milk problem by putting a lock on refrigerator
  - Lock it and take key if you are going to go buy milk
  - Fixes too much (coarse granularity): roommate angry if only wants orange juice



– Of Course – We don't know how to make a lock yet

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.14

## Too Much Milk: Correctness Properties

- Need to be careful about correctness of concurrent programs, since non-deterministic
  - Always write down **desired** behavior first
  - Impulse is to start coding first, then when it doesn't work, pull hair out
  - Instead, think first, then code
- What are the correctness properties for the "Too much milk" problem?
  - Never more than one person buys
  - Someone buys if needed
- Restrict ourselves to use only atomic load and store operations as building blocks

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.15

## Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of "lock")
  - Remove note after buying (kind of "unlock")
  - Don't buy if note (wait)
- Suppose a computer tries this (remember, only memory read/write are atomic):

```
if (noMilk) {  
    if (noNote) {  
        leave Note;  
        buy milk;  
        remove note;  
    }  
}
```



- Result?

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.16

## Too Much Milk: Solution #1

- Still too much milk **but only occasionally!**

```

Thread A                               Thread B
if (noMilk)                             if (noMilk)
  if (noNote) {                          if (noNote) {
                                          leave Note;
                                          buy milk;
                                          remove note;
                                          }
                                          ...
                                          leave Note;
                                          buy milk;
                                          }

```

- Thread can get context switched after checking milk and note but before leaving note!
- Solution makes problem worse since fails **intermittently**
  - Makes it really hard to debug...
  - Must work despite what the thread dispatcher does!

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.17

## Too Much Milk: Solution #1½

- Clearly the Note is not quite blocking enough
  - Let's try to fix this by placing note first
- Another try at previous solution:

```

leave Note;
if (noMilk) {
  if (noNote) {
    buy milk;
  }
}
remove Note;

```



- What happens here?
  - Well, with human, probably nothing bad
  - With computer: no one ever buys milk

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.18

## Too Much Milk Solution #2

- How about labeled notes?
  - Now we can leave note before checking
- Algorithm looks like this:

```

Thread A                               Thread B
leave note A;                           leave note B;
if (noNote B) {                         if (noNote A) {
  if (noMilk) {                           if (noMilk) {
    buy Milk;                               buy Milk;
  }                                         }
}                                           }
remove note A;                           remove note B;

```

- Does this work?

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.19

## Too Much Milk Solution #2

- Possible for neither thread to buy milk!

```

Thread A                               Thread B
leave note A;                           leave note B;
                                          if (noNote A) {
                                          if (noMilk) {
                                          buy Milk;
                                          }
                                          }
if (noNote B) {                         ...
  if (noMilk) {
    buy Milk;
  }
  ...
  remove note B;
}

```

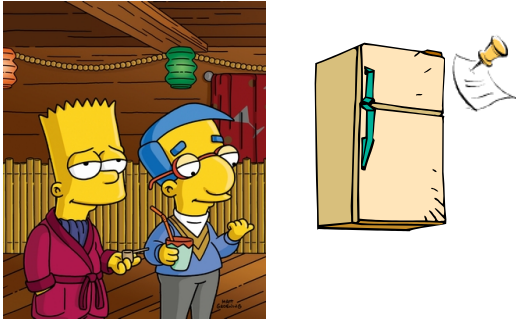
- Really insidious:
  - **Unlikely** that this would happen, but will at worst possible time

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.20

## Too Much Milk Solution #2: problem!



- *I'm not getting milk, You're getting milk*
- This kind of lockup is called "starvation!"

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.21

## Administrivia

- Section assignments posted on Piazza
  - Most groups were assigned 1<sup>st</sup> or 2<sup>nd</sup> preference
  - Attend assigned sections THIS week
- Nachos Project I begins tomorrow (Threads)
  - Start reading walkthrough and code NOW
  - Download Nachos tar file
  - Set up Java environment, Eclipse, version control
  - More details in sections

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.22

## 5min Break

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.23

## Too Much Milk Solution #3

- Here is a possible two-note solution:

```
Thread A                               Thread B
leave note A;                           leave note B;
while (note B) {\\X                       if (noNote A) {\\Y
  do nothing;                             if (noMilk) {
}                                           buy milk;
if (noMilk) {                               }
  buy milk;                                 }
}                                           remove note B;
remove note A;
```

- Does this work? Yes. Both can guarantee that:
  - It is safe to buy, or
  - Other will buy, ok to quit
- At X:
  - if no note B, safe for A to buy,
  - otherwise wait to find out what will happen
- At Y:
  - if no note A, safe for B to buy
  - Otherwise, A is either buying or waiting for B to quit

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.24

### Solution #3 discussion

- Our solution protects a single “Critical-Section” piece of code for each thread:

```
if (noMilk) {  
    buy milk;  
}
```

- Solution #3 works, but it’s really unsatisfactory
  - Really complex – even for this simple an example
    - » Hard to convince yourself that this really works
  - A’s code is different from B’s – what if lots of threads?
    - » Code would have to be slightly different for each thread
  - While A is waiting, it is consuming CPU time
    - » This is called “busy-waiting”
- There’s a better way
  - Have hardware provide better (higher-level) primitives than atomic load and store
  - Build even higher-level programming abstractions on this new hardware support

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.25

### High-Level Picture

- The abstraction of threads is good:
  - Maintains sequential execution model
  - Allows simple parallelism to overlap I/O and computation
- Unfortunately, still too complicated to access state shared between threads
  - Consider “too much milk” example
  - Implementing a concurrent program with only loads and stores would be tricky and error-prone
- We’ll implement higher-level operations on top of atomic operations provided by hardware
  - Develop a “synchronization toolbox”
  - Explore some common programming paradigms



2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.26

### Too Much Milk: Solution #4

- Suppose we have some sort of implementation of a lock (more in a moment)
  - `Lock.Acquire()` – wait until lock is free, then grab
  - `Lock.Release()` – unlock, waking up anyone waiting
  - These must be atomic operations – if two threads are waiting for the lock, only one succeeds to grab the lock

- Then, our milk problem is easy:

```
milklock.Acquire();  
if (nomilk)  
    buy milk;  
milklock.Release();
```

- Once again, section of code between `Acquire()` and `Release()` called a “Critical Section”

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.27

### How to Implement Lock?

- **Lock:** prevents someone from accessing something
  - Lock before entering critical section (e.g., before accessing shared data)
  - Unlock when leaving, after accessing shared data
  - Wait if locked
    - » Important idea: all synchronization involves waiting
    - » Should sleep if waiting for long time
- Hardware lock instructions
  - Is this a good idea?
  - What about putting a task to sleep?
    - » How do handle interface between hardware and scheduler?
  - Complexity?
    - » Each feature makes hardware more complex and slower



2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.28

## Naïve use of Interrupt Enable/Disable

- How can we build multi-instruction atomic operations?
  - Recall: dispatcher gets control in two ways.
    - » Internal: Thread does something to relinquish the CPU
    - » External: Interrupts cause dispatcher to take CPU
  - On a uniprocessor, can avoid context-switching by:
    - » Avoiding internal events (although virtual memory tricky)
    - » Preventing external events by disabling interrupts

- Consequently, naïve Implementation of locks:

```
LockAcquire { disable Ints; }
LockRelease { enable Ints; }
```

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.29

## Naïve use of Interrupt Enable/Disable: Problems

- **Can't let user do this!** Consider following:

```
LockAcquire();
While(TRUE) {;}
```

- Real-Time system—no guarantees on timing!
  - Critical Sections might be arbitrarily long
- What happens with I/O or other important events?
  - “Reactor about to meltdown. Help?”



2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.30

## Better Implementation of Locks by Disabling Interrupts

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable

```
int value = FREE;
```



```
Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep();
        // Enable interrupts?
    } else {
        value = BUSY;
    }
    enable interrupts;
}

Release() {
    disable interrupts;
    if (anyone on wait queue) {
        take thread off wait queue
        Put at front of ready queue
    } else {
        value = FREE;
    }
    enable interrupts;
}
```

2/4/13

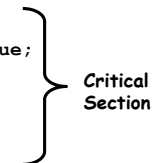
Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.31

## New Lock Implementation: Discussion

- Disable interrupts: avoid interrupting between checking and setting lock value
  - Otherwise two threads could think that they both have lock

```
Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep();
        // Enable interrupts?
    } else {
        value = BUSY;
    }
    enable interrupts;
}
```



- Note: unlike previous solution, critical section very short
  - User of lock can take as long as they like in their own critical section
  - Critical interrupts taken in time

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.32



## Interrupt re-enable in going to sleep

- What about re-enabling ints when going to sleep?

```

Acquire() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    go to sleep();
  } else {
    value = BUSY;
  }
  enable interrupts;
}
    
```

Enable Position  
 Enable Position  
 Enable Position

- Before putting thread on the wait queue?
  - Release can check the queue and not wake up thread
- After putting the thread on the wait queue
  - Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep
  - Misses wakeup and still holds lock (deadlock!)
- Want to put it after sleep(). But, how?

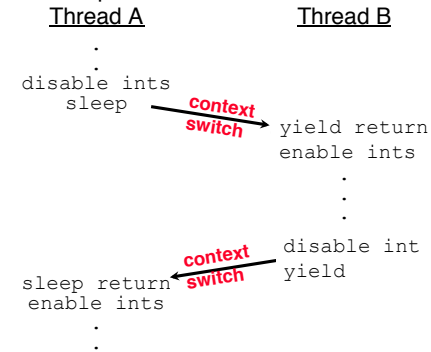
2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.33

## How to Re-enable After Sleep()?

- Since ints are disabled when you call sleep:
  - Responsibility of the next thread to re-enable ints
  - When the sleeping thread wakes up, returns to acquire and re-enables interrupts



2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.34

## Summary

- Introduced important concept: Atomic Operations
  - An operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives
- Showed construction of Locks using interrupts
  - Using careful disabling of interrupts
  - Must be very careful not to waste/tie up machine resources
    - » Shouldn't disable interrupts for long
  - Key ideas: Use a separate lock variable, and use hardware mechanisms to protect modifications of that variable

2/4/13

Anthony D. Joseph CS162 ©UCB Spring 2013

Lec 4.35