

**CSE 451: Operating Systems
Autumn 2011**

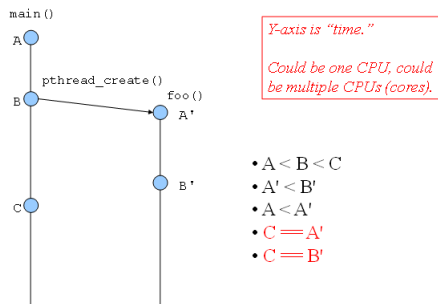
**Module 3
Synchronization**

John Zahorjan
zahorjan@cs.washi
ngton.edu
Allen Center 534

Temporal Relations: Key Concept Review

- Instructions executed by a single thread are totally ordered
 - $A < B < C < \dots$
- Absent *synchronization*, instructions executed by distinct threads are *simultaneous*
 - (not $A < A'$) and (not $A' < A$)
- A sequence of instructions is *atomic* if the effects of all of them appear to occur at once as viewed by any other (correctly operating) thread
- (Nearly all) single *machine* instructions are atomic
 - Write x
 - Read y

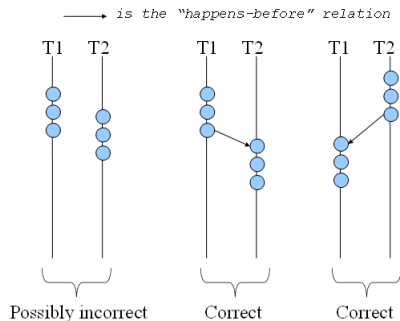
Example: In the beginning...



Critical Sections / Mutual Exclusion

- Sequences of instructions that *may* get incorrect results if executed simultaneously are called *critical sections*
- *Mutual exclusion* means "not simultaneous"
 - $(A < B) \text{ or } (B < A)$
 - We don't care which
- Forcing mutual exclusion between two critical section executions is sufficient to ensure correct execution
 - It's not always necessary (concurrent executions may sometimes get correct results by luck), but it's impractical to try to exploit that
- One way to guarantee mutually exclusive execution is using *locks*

Critical sections



When Do Critical Sections Arise?

- Well... the simple answer is "whenever simultaneous execution could result in incorrect answers," but that isn't very helpful
- One common pattern:
 - read-modify-write of...
 - a shared value (variable) while...
 - in code that can be executed concurrently
 - Note: There may be only one copy of the code (e.g., a procedure), but it can be executed by more than one thread at a time
- Shared variable:
 - Globals and heap allocated
 - NOT local variables
 - Note: never give a reference to a stack allocated (local) variable to another thread (unless you're superhumanly careful...)

The classic example

- Suppose we have to implement a function to withdraw money from a bank account:

```
int withdraw(account, amount) {
    int balance = get_balance(account); // read
    balance -= amount; // modify
    put_balance(account, balance); // write
    return balance;
}
```

- Now suppose that you and your S.O. share a bank account with a balance of \$100.00
 - what happens if you both go to separate ATM machines, and simultaneously withdraw \$10.00 from the account?

10/09/11

© 2011 Gribble, Lazowska, Levy, Zahorjan

7

- Assume the bank's application is multi-threaded
- A random thread is assigned a transaction when it is submitted

```
int withdraw(account, amount) {
    int balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    return balance;
}
```

```
int withdraw(account, amount) {
    int balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    return balance;
}
```

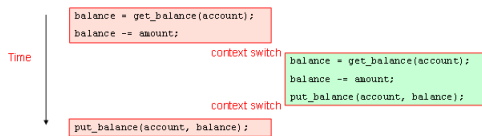
10/09/11

© 2011 Gribble, Lazowska, Levy, Zahorjan

8

Interleaved schedules

- The problem is that the execution of the two threads can be interleaved:



- What's the account balance after this sequence?
- How often is this sequence likely to occur?

10/09/11

© 2011 Gribble, Lazowska, Levy, Zahorjan

9

Aside: Other Execution Orders

- Which interleavings are ok? Which are not?

```
int withdraw(account, amount) {
    int balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    return balance;
}
```

```
int withdraw(account, amount) {
    int balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    return balance;
}
```

10/09/11

© 2011 Gribble, Lazowska, Levy, Zahorjan

10

How About Now?

```
int xfee(from, to, amt) {
    int bal = withdraw(from, amt);
    withdraw(to, -amt);
    return bal;
}
```

```
int xfee(from, to, amt) {
    int bal = withdraw(from, amt);
    withdraw(to, -amt);
    return bal;
}
```

- Morals:
 - Interleavings are hard to reason about
 - We make a lot of mistakes
 - Control-flow analysis is hard for tools to get right
 - Identifying critical sections and ensuring mutually exclusive execution is... "easier"

10/09/11

© 2011 Gribble, Lazowska, Levy, Zahorjan

11

Another Classic Example

```
i++;
```

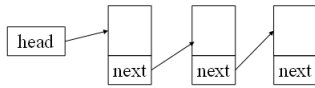
```
i++;
```

10/09/11

© 2011 Gribble, Lazowska, Levy, Zahorjan

12

Final Classic Example



```
for (p=head; p; p = p->next ) {
  <examine *p>
}
```

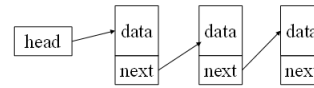
```
while (head) {
  oldHead = head;
  head = head->next;
  free (oldHead);
}
```

10/09/11

© 2011 Gribble, Lazowska, Levy, Zahorjan

13

Other Final Classic Example



```
for (p=head; p; p = p->next ) {
  <examine *p>
}
```

```
for (p=head; p; p = p->next ) {
  p->data = p->data + 1;
}
```

10/09/11

© 2011 Gribble, Lazowska, Levy, Zahorjan

14

"Critical section solution" requirements

- Critical sections have the following requirements
 - **mutual exclusion**
 - at most one thread is in the critical section
 - **progress**
 - if thread T is outside the critical section, then T cannot prevent thread S from entering the critical section
 - **bounded waiting** (no starvation)
 - if thread T is waiting on the critical section, then T will eventually enter the critical section
 - assumes threads eventually leave critical sections
 - vs. fairness?
 - **performance**
 - the overhead of entering and exiting the critical section is small with respect to the work being done within it

10/09/11

© 2011 Gribble, Lazowska, Levy, Zahorjan

15

Mechanisms for building critical sections

- Locks (**today**)
 - very primitive, minimal semantics; used to build others
- Semaphores (**tomorrow**)
 - basic, easy to get the hang of, hard to program with
- Condition Variables (**tomorrow**)
 - general (i.e., primitive) "wait until" mechanism with mutual exclusion
- Monitors (**tomorrow**)
 - high level, requires language support, implicit operations
 - easy to program with; Java "synchronized()" as an example
- Messages (**day after tomorrow**)
 - simple model of communication and synchronization based on (atomic) transfer of data across a channel
 - direct application to distributed systems

10/09/11

© 2011 Gribble, Lazowska, Levy, Zahorjan

16

Locks, But First...

- A possible critical section solution is to arrange for all executions to occur on a single thread
 - E.g., use thread n where $n = \text{account} \% \# \text{threads}$
 - This turns a sharable variable into an un-shared variable
- Pros:
 - Simple
 - Fast
- Cons:
 - Load balancing among threads
 - What to do if the CS involves two accounts (e.g., xfer())?
 - Assigning tasks to threads probably involves a critical section ()
- This idea is useful on multi-cores, and perhaps even more common in distributed systems

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

10/09/11

© 2011 Gribble, Lazowska, Levy, Zahorjan

17

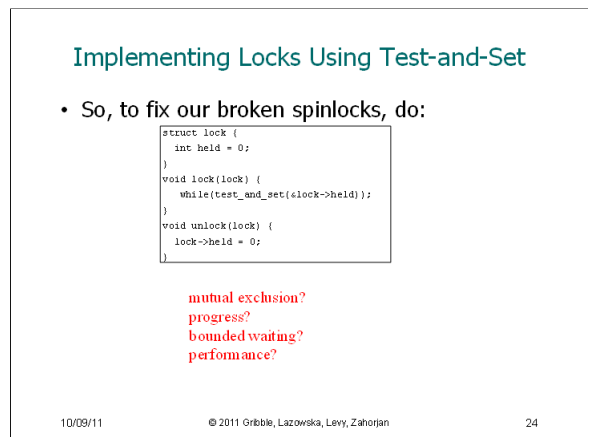
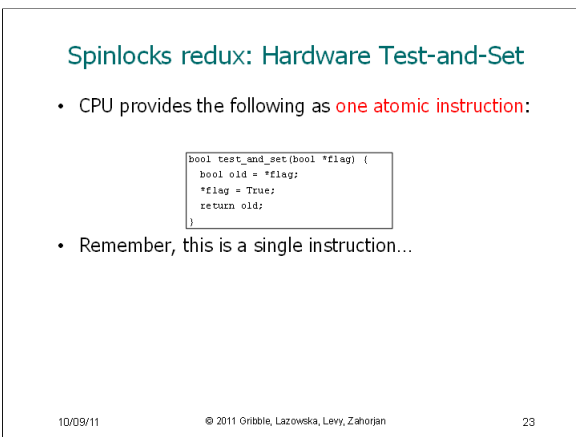
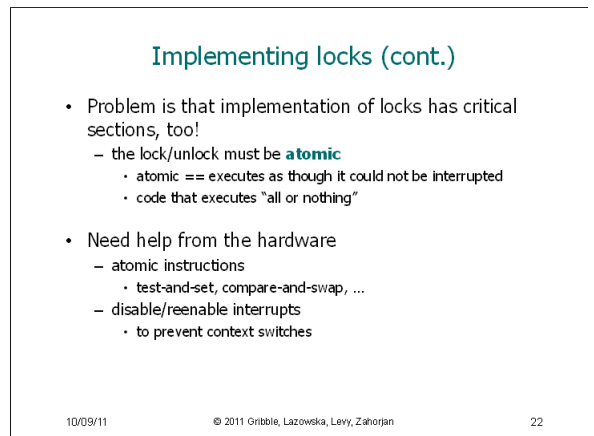
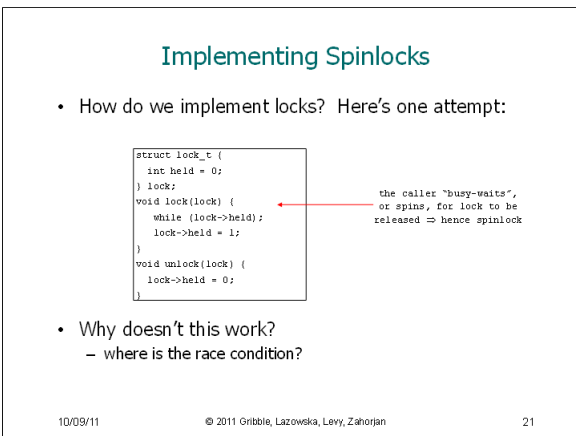
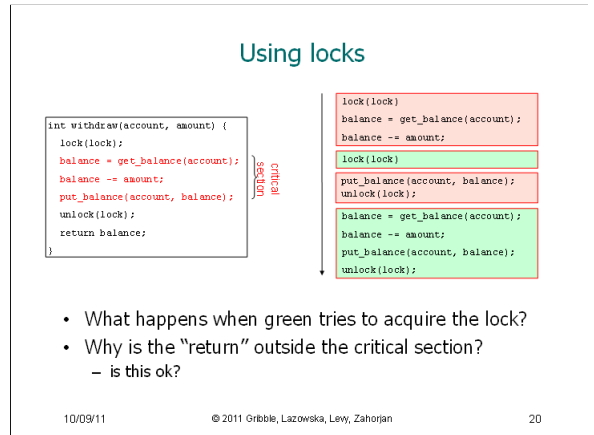
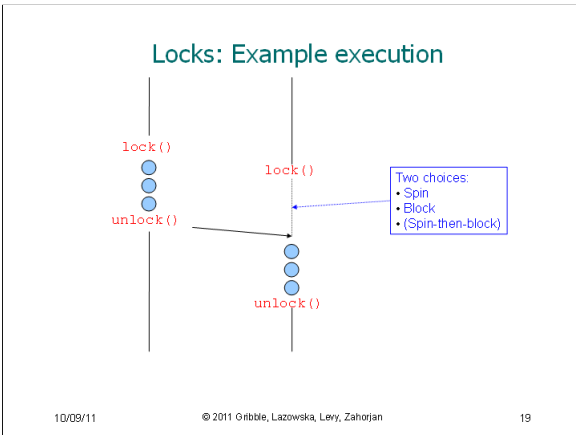
Locks

- Locks are memory objects with two operations
 - `lock()`: obtain the right to enter the critical section
 - `unlock()`: give up the right to be in the critical section
- `lock()` prevents progress of the thread until the lock can be acquired

10/09/11

© 2011 Gribble, Lazowska, Levy, Zahorjan

18



Reminder of use ...

```
int withdraw(account, amount) {
    lock(lock);
    balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    unlock(lock);
    return balance;
}
```

critical section

```
lock(lock);
balance = get_balance(account);
balance -= amount;

lock(lock);
put_balance(account, balance);
unlock(lock);

balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
unlock(lock);
```

- How does a thread spinning on a lock (that is, stuck in a test-and-set loop) yield the CPU?
 - calls `yield()` (*spin-then-block*)
 - there's an involuntary context switch

10/09/11

© 2011 Gribble, Lazowska, Levy, Zahorjan

25

Problems with Locks

- Spinlocks work, but can be horribly wasteful!
 - If the thread holding the lock is not running, you'll spin for a scheduling quantum
 - Certainly the case on a single-core machine
 - (`pthread_spin_t`)
- Blocking locks work, but can be horribly wasteful!
 - If the lock is busy, there's a two context switch overhead cost to be paid to acquire it, minimum
 - The lock might be busy for only a few cycles, so it could have been cheaper to spin
 - (`pthread_mutex_t`)
- Spin-then-block locks
 - Spin for a little while (10's or 100's of cycles), then block
 - Why?
 - If you know the typical lock holding time is small, and it's been 100's of cycles, odds are the lock holder isn't currently running
 - This is an example of residual life that increases (steeply) after some short amount of time has elapsed

10/09/11

© 2011 Gribble, Lazowska, Levy, Zahorjan

26

Race Conditions

- Informally, we say a program has a **race condition** (aka "data race") if the result of an execution depends on timing
 - i.e., is non-deterministic
- Typical symptoms:
 - I run it on the same data, and sometimes it prints 0 and sometimes it prints 4
 - I run it on the same data, and sometimes it prints 0 and sometimes it crashes

10/09/11

© 2011 Gribble, Lazowska, Levy, Zahorjan

27

Race Detectors

- There are tools that try to detect race conditions
 - We'll use one called `helgrind`
- They need a formal definition of what a race is
 - The definition varies, but the key is two accesses to a shared variable that are "simultaneous" (not ordered), at least one of which is a write
- Note: the formal definition can result in many false positives (detections of non-problems)
 - Example: two threads write 0 to shared variable `total`

10/09/11

© 2011 Gribble, Lazowska, Levy, Zahorjan

28

How Race Detectors Work

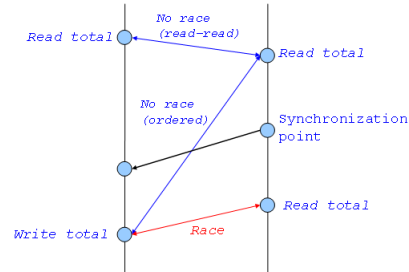
- First of all, they're still a bit exotic / experimental / primitive
- Basically, they monitor thread executions to construct a "happens before" thread graph relating them
 - Happens-before arcs are introduced by things like locks, which they recognize as a call to `pthread_mutex_lock()`
- They then detect unsynchronized accesses by annotating each word/byte of memory with tags indicating where in the thread synchronization graph the operations arose
- They manage that by simulating the hardware instructions...
- They can be "a wee slow"

10/09/11

© 2011 Gribble, Lazowska, Levy, Zahorjan

29

Race Detection Example



10/09/11

© 2011 Gribble, Lazowska, Levy, Zahorjan

30

What's Next?

- Synchronization introduces temporal ordering
 - E.g., adds a “not simultaneous” edge
 - Critical sections
 - Or adds a “happens before” edge to the thread graph
 - Other kinds of synchronization
- Adding synchronization can eliminate races
 - That's handy!
- There are other synchronization primitives
 - For mutual exclusion
 - For “happens before”
- We'll have a look at some...