CSE 332: Data Structures & Parallelism

Lecture 17: Shared-Memory Concurrency & Mutual Exclusion

Ruth Anderson

Autumn 2016

Toward sharing resources (memory)

So far, we have been studying parallel algorithms using the fork-join model

- Reduce span via parallel tasks

Fork-Join algorithms all had a very simple *structure* to avoid race conditions

- Each thread had memory "only it accessed"
  - Example: each array sub-range accessed by only one thread
- Result of forked process not accessed until after join() called
- So the structure (mostly) ensured that bad simultaneous access wouldn't occur

Strategy won't work well when:

- **Memory** accessed by threads **is overlapping** or unpredictable
- Threads are doing **independent tasks** needing **access to same resources** (rather than implementing the same algorithm)

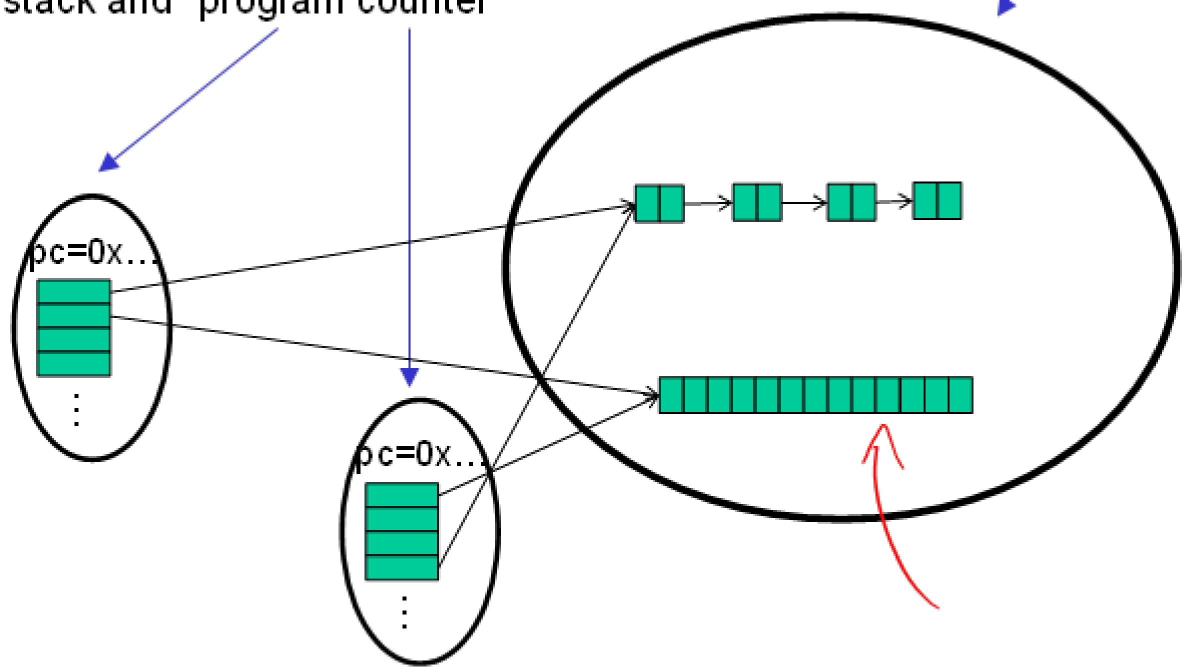
Each thread accesses a different sub-range of the array: Array is shared, but no overlap

```
class SumTask extends RecursiveTask<Integer> {
  int lo; int hi; int[] arr; // fields to know what to do
  SumTask(int[] a, int l, int h) { … }
  protected Integer compute(){// return answer
    if(hi - lo < SEQUENTIAL_CUTOFF) {
      int ans = 0; // local var, not a field
      for(int i=lo; i < hi; i++)
        ans += arr[i];
      return ans;
    } else {
      SumTask left = new SumTask(arr,lo,(hi+lo)/2);
      SumTask right= new SumTask(arr,(hi+lo)/2,hi);
      left.fork(); // fork a thread and calls compute
      int rightAns = right.compute();//call compute
directly
      int leftAns  = left.join(); // get result from left
      return leftAns + rightAns;
    }
  }
}
static final ForkJoinPool POOL = new ForkJoinPool();
int sum(int[] arr){
   SumTask task = new SumTask(arr,0,arr.length)
   return POOL.invoke(task);
```

Really sharing memory between Threads

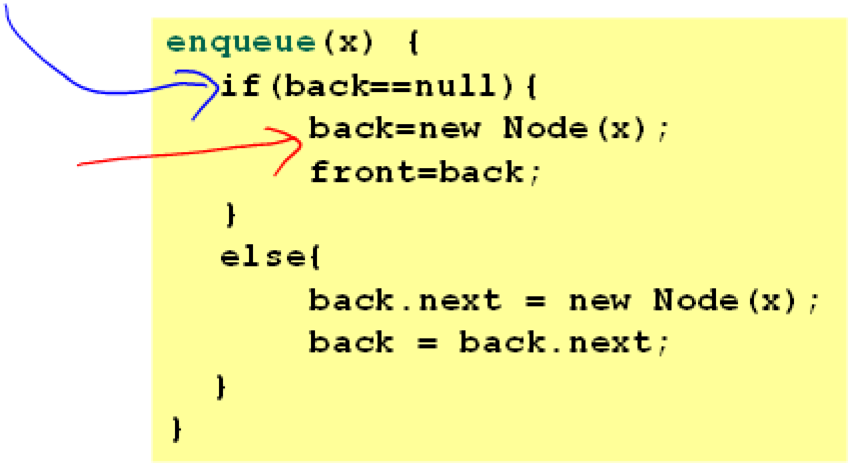
Heap for all objects
and static fields, *shared*
by all threads

2 Threads, each with own *unshared*
call stack and “program counter”



Sharing a Queue….

- Imagine 2 threads, running at the same time,
- both with access to a shared linked-list based queue (initially empty)



```
enqueue(x)  {
   if(back==null){
        back=new Node(x);
        front=back;
   }
   else{
        back.next = new Node(x);
        back = back.next;
   }
}
```

Sharing a Queue….

- Imagine 2 threads, running at the same time, both with access to a shared linked-list based queue (initially empty)

```
enqueue(x)  {
   if(back==null){
        back=new Node(x);
        front=back;
   }
   else{
        back.next = new Node(x);
        back = back.next;
   }
}
```

- Each thread has own program counter (and local stack)
- Queue is shared, so both threads indirectly use the same ‘front’ and ‘back’ (which is the whole point of sharing the queue)
- We have no guarantee what happens first between different threads; they can (and will) arbitrarily ‘interrupt’ each other
- Many things can go wrong: say, one tries to enqueue “a”, the other “b”, and both verify that back is ‘null’ before other sets back
    - Result: One assignment of back will be ‘forgotten’
- In general, any ‘interleaving’ of results is possible if enqueue were called at the same time for both

Concurrent Programming

Concurrency: Correctly and efficiently managing access to shared resources from multiple possibly-simultaneous clients

Requires *coordination*, particularly synchronization to avoid incorrect simultaneous access: make somebody *block* (wait) until the resource is free

- `join` is not what we want
- Want to block until another thread is "done using what we need" not "completely done executing"

Even correct concurrent applications are usually highly non-deterministic

- how threads are scheduled affects what operations happen first
- non-repeatability complicates testing and debugging

Concurrency Examples

What if we have multiple threads:

1. Processing different bank-account operations
   - What if 2 threads change the same account at the same time?

2. Using a shared cache (e.g., hashtable) of recent files
   - What if 2 threads insert the same file at the same time?

3. Creating a pipeline (think assembly line) with a queue for handing work from one thread to next thread in sequence?
   - What if enqueuer and dequeuer adjust a circular array queue at the same time?

Why threads?

Unlike parallelism, not about implementing algorithms faster

But threads still useful for:

- *Code structure for responsiveness*
  - Example: Respond to GUI events in one thread while another thread is performing an expensive computation
- *Processor utilization (mask I/O latency)*
  - If 1 thread "goes to disk," have something else to do
- *Failure isolation*
  - Convenient structure if want to *interleave* multiple tasks and do not want an exception in one to stop the other

Sharing, again

It is common in concurrent programs that:

- Different threads might access the same resources in an unpredictable order or even at about the same time

- Program correctness requires that simultaneous access be prevented using synchronization

- Simultaneous access is rare
  - Makes testing difficult
  - Must be much more disciplined when designing / implementing a concurrent program
  - Will discuss common idioms known to work

Canonical example

Correct code in a single-threaded world

```
class BankAccount {
  private int balance = 0;
  int  getBalance()       { return balance; }
  void setBalance(int x) { balance = x; }
  void withdraw(int amount) {
    int b = getBalance();
    if(amount > b)
      throw new WithdrawTooLargeException();
    setBalance(b – amount);
  }
  … // other operations like deposit, etc.
}
```

Interleaving

Suppose:

- Thread **T1** calls `x.withdraw(100)`
- Thread **T2** calls `y.withdraw(100)`

If second call starts before first finishes, we say the calls interleave

- Could happen even with one processor since a thread can be pre-empted at any point for time-slicing
  - e.g. T1 runs for 50 ms, pauses somewhere, T2 picks up for 50ms

If `x` and `y` refer to different accounts, no problem

- "You cook in your kitchen while I cook in mine"
- But if `x` and `y` alias, possible trouble…

Activity: What is the balance at the end?

Two threads both trying to `withdraw()` from the same account:

- Assume initial `balance` 150

```
class BankAccount {
  private int balance = 0;
  int  getBalance()       { return balance; }
  void setBalance(int x) { balance = x; }
  void withdraw(int amount) {
    int b = getBalance();
    if(amount > b)
      throw new WithdrawTooLargeException();
    setBalance(b - amount);
  }
  … // other operations like deposit, etc.
}
```

Thread 1

```
x.withdraw(100);
```

Thread 2

```
x.withdraw(75);
```

x's balance = ~~75~~ 50

Activity: A bad interleaving

Interleaved `withdraw()` calls on the same account

- Assume initial `balance == 150`
- This ***should*** cause a `WithdrawTooLarge` exception

b=150

Time

Thread 1: `withdraw(100)`

```
int b = getBalance();
b=150


if(amount > b)
  throw new …;
setBalance(b - amount);
```

getBalance()

getBalance()

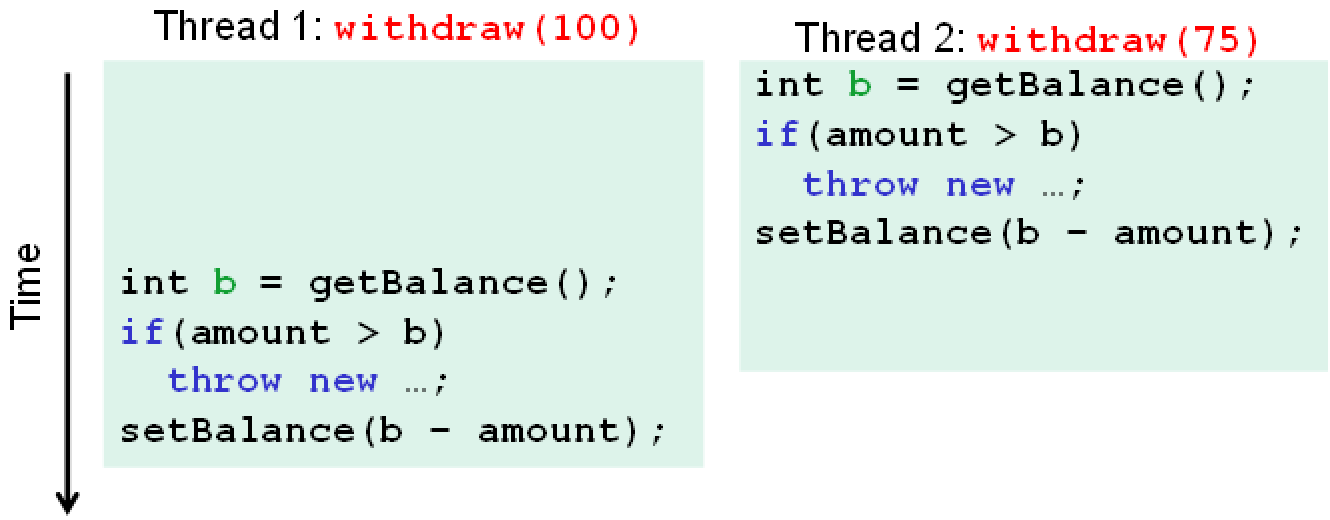
Thread 2: `withdraw(75)`

```
int b = getBalance();
if(amount > b)
  throw new …;
setBalance(b - amount);
```

Activity: A "good" execution is also possible

Interleaved `withdraw()` calls on the same account

- Assume initial `balance == 150`
- This ***should*** cause a `WithdrawTooLarge` exception



Thread 1: `withdraw(100)`

```
int b = getBalance();
if(amount > b)
  throw new …;
setBalance(b – amount);
```

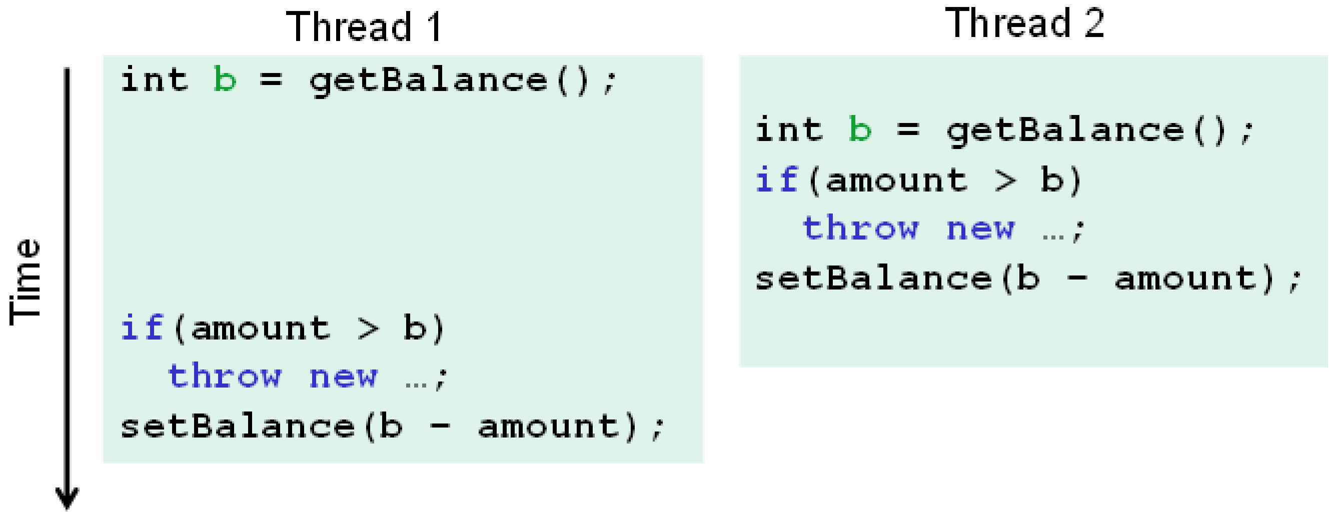
Thread 2: `withdraw(75)`

```
int b = getBalance();
if(amount > b)
  throw new …;
setBalance(b – amount);
```

Example: A bad interleaving

Interleaved `withdraw(100)` calls on the same account

- Assume initial `balance == 150`
- This ***should*** cause a `WithdrawTooLarge` exception



Thread 1

```
int b = getBalance();




if(amount > b)
  throw new …;
setBalance(b – amount);
```

Thread 2

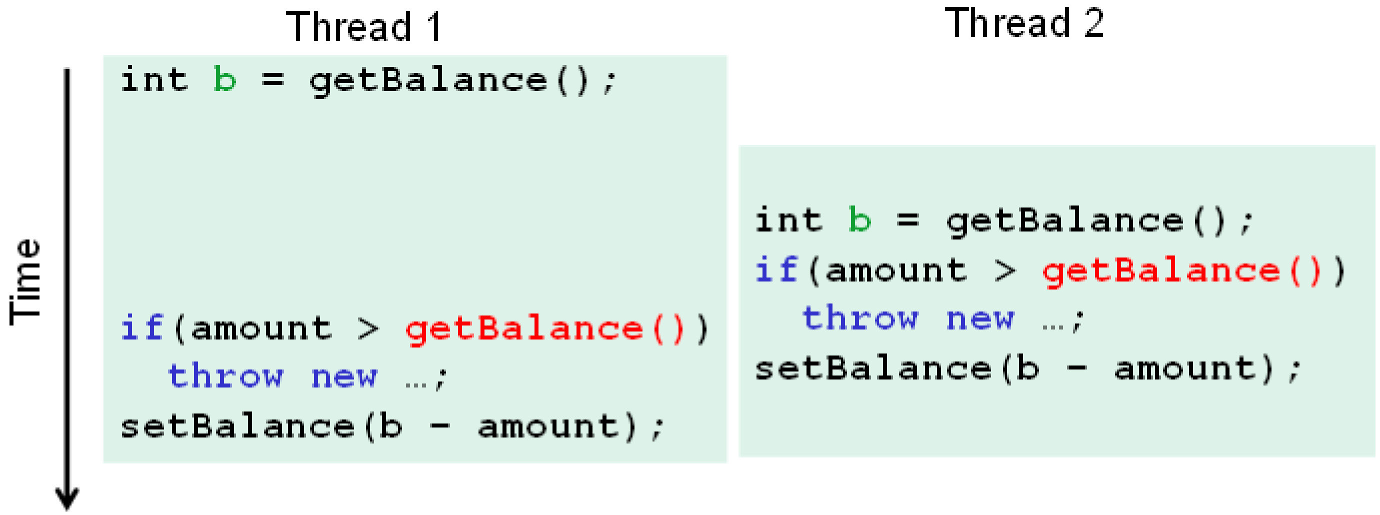
```

int b = getBalance();
if(amount > b)
  throw new …;
setBalance(b – amount);
```

A bad fix, Another bad interleaving

Two threads both trying to `withdraw(100)` from the same account:

- Assume initial `balance` 150
- This ***should*** cause a `WithdrawTooLarge` exception



Thread 1

```
int b = getBalance();



if(amount > getBalance())
  throw new …;
setBalance(b – amount);
```

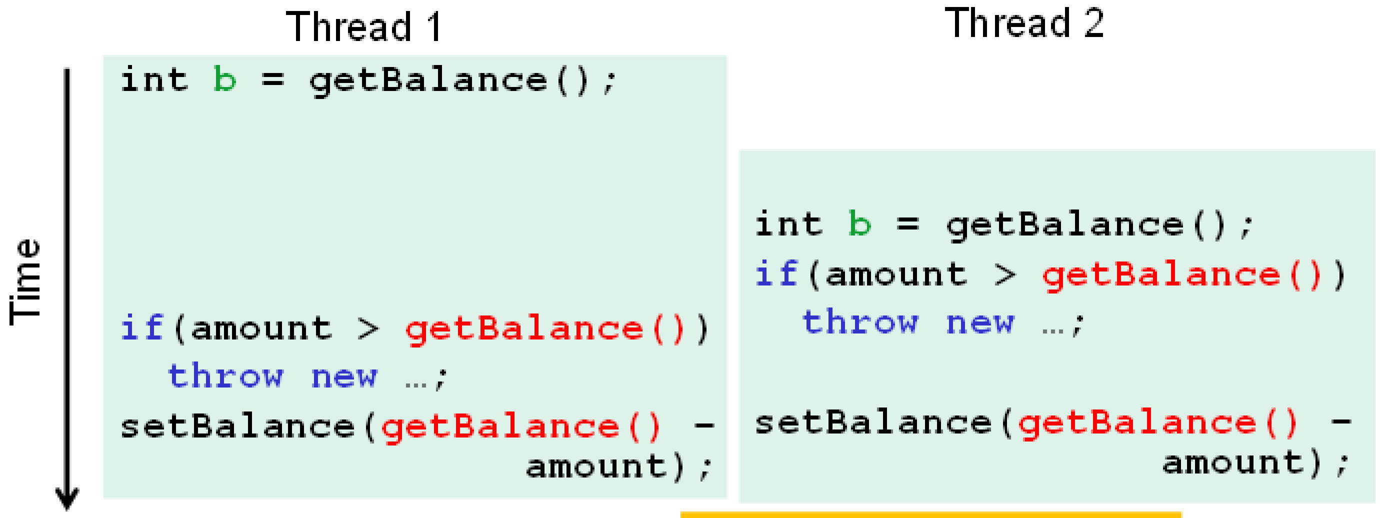
Thread 2

```
int b = getBalance();
if(amount > getBalance())
  throw new …;
setBalance(b – amount);
```

Still a bad fix, Another bad interleaving

Two threads both trying to `withdraw(100)` from the same account:

- Assume initial `balance` 150
- This ***should*** cause a `WithdrawTooLarge` exception



Thread 1

```
int b = getBalance();




if(amount > getBalance())
  throw new …;
setBalance(getBalance() –
                    amount);
```

Thread 2

```
int b = getBalance();
if(amount > getBalance())
  throw new …;

setBalance(getBalance() –
                    amount);
```

In all 3 of these examples, instead of an exception, we have a “Lost withdraw”

Incorrect "fix"

It is tempting and almost always wrong to fix a bad interleaving by rearranging or repeating operations, such as:

```
void withdraw(int amount) {
  if(amount > getBalance())
    throw new WithdrawTooLargeException();
  // maybe balance changed
  setBalance(getBalance() - amount);
}
```

This fixes nothing!

- Narrows the problem by one statement
- (Not even that since the compiler could turn it back into the old version because you didn't indicate need to synchronize)
- And now a negative balance is possible – why?

What we want: ***Mutual exclusion***

The fix: Allow at most one thread to withdraw from account **A** at a time

- *Exclude* other simultaneous operations on **A** too (e.g., deposit)

Called **mutual exclusion**:

- One thread using a resource (here: a bank account) means another thread must wait
- We call the area of code that we want to have mutual exclusion (only one thread can be there at a time) a **critical section**.

Programmer (you!) must implement **critical sections**:

- "The compiler" has no idea what interleavings should or should not be allowed in your program
- But you need language primitives to do it!

Why is this Wrong?

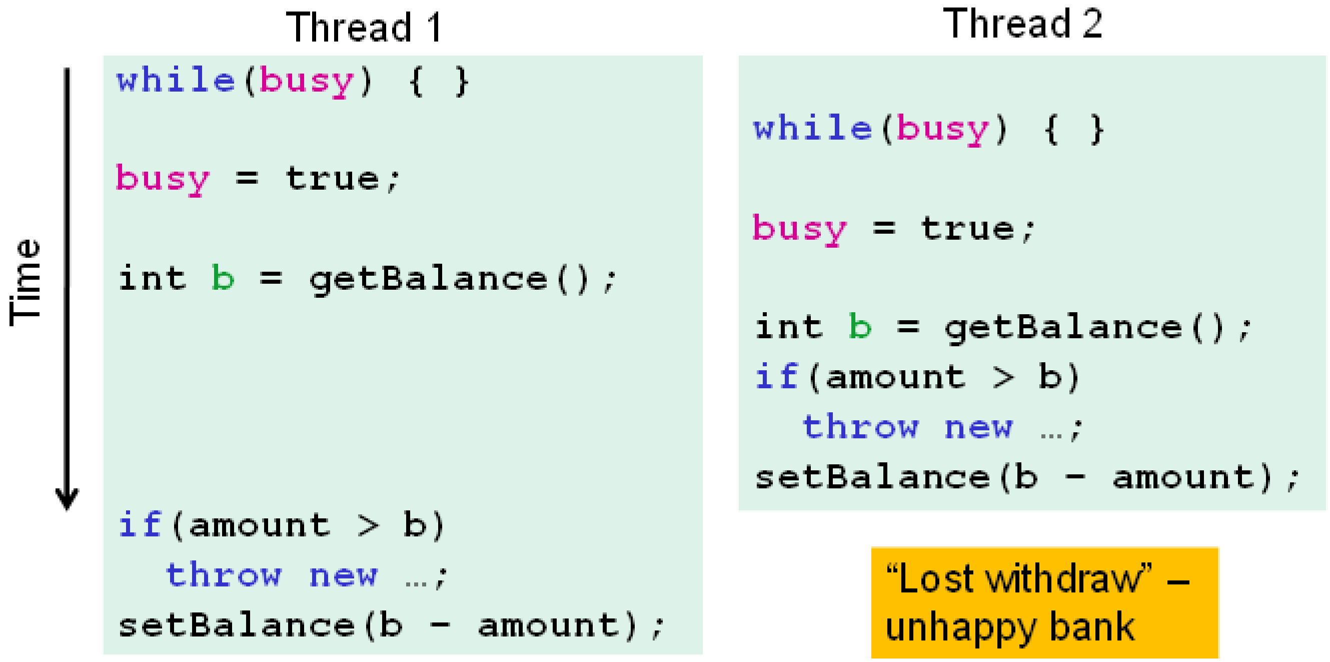
Why can't we implement our own mutual-exclusion protocol?

- Say we tried to coordinate it ourselves using a boolean variable – "**busy**"
- It's technically possible under certain assumptions, but won't work in real languages anyway

```
class BankAccount {
  private int balance = 0;
  private boolean busy = false;
  void withdraw(int amount) {
    while(busy) { /* "spin-wait" */ }
    busy = true;
    int b = getBalance();
    if(amount > b)
      throw new WithdrawTooLargeException();
    setBalance(b – amount);
    busy = false;
  }
  // deposit would spin on same boolean
}
```

`Busy` is initially = false

Still just moved the problem!



Thread 1

```
while(busy) { }

busy = true;

int b = getBalance();




if(amount > b)
  throw new …;
setBalance(b - amount);
```

Thread 2

```

while(busy) { }

busy = true;

int b = getBalance();
if(amount > b)
  throw new …;
setBalance(b - amount);
```

"Lost withdraw" – unhappy bank

Time elapses between checking `busy` and setting `busy = true`
A thread can be interrupted there, allowing another thread to "sneak in".

What we need

There are many ways out of this conundrum,
but we need help from the programming language…

One solution: **Mutual-Exclusion Locks** (aka Mutex, or just Lock)

- Still on a conceptual level at the moment, 'Lock' is not a Java class (though Java's approach is similar)

We will define Lock as an ADT with operations:

- `new`: make a new lock, initially *"not held"*
- `acquire`: blocks if this lock is already currently *"held"*
  - Once *"not held"*, makes lock *"held"* [all at once!]
  - Checking & setting happen together, and cannot be interrupted
  - Fixes problem we saw before!!
- `release`: makes this lock *"not held"*
  - If >= 1 threads are blocked on it, exactly 1 will acquire it

Why that works

- A Lock ADT with operations `new`, `acquire`, `release`

- The lock implementation ensures that given simultaneous acquires and/or releases, a correct thing will happen
  - Example:
    - If we have two acquires: one will “win” and one will block

- How can this be implemented?
  - Need to “check if held and if not make held” “all-at-once”
  - Uses special hardware and O/S support
    - See computer-architecture or operating-systems course
  - In CSE 332, we take this as a primitive and use it

Almost-correct pseudocode

Note: 'Lock' is not an actual Java class

```
class BankAccount {
  private int balance = 0;
  private Lock lk = new Lock();
  …
  void withdraw(int amount) {
    lk.acquire(); // may block
    int b = getBalance();
    if(amount > b)
      throw new WithdrawTooLargeException();
    setBalance(b - amount);
    lk.release();
  }
  // deposit would also acquire/release lk
}
```

Some mistakes

- A lock is a very primitive mechanism
  - Still up to you to use correctly to implement critical sections
- **Incorrect**: Use different locks for `withdraw` and `deposit`
  - Mutual exclusion works only when using same lock
  - `balance` field is the shared resource being protected
- **Poor performance**: Use same lock for every bank account
  - No simultaneous operations on *different* accounts
- **Incorrect**: Forget to release a lock (blocks other threads forever!)
  - Previous slide is wrong because of the exception possibility!

```
if(amount > b) {
  lk.release(); // hard to remember!
  throw new WithdrawTooLargeException();
}
```

Other operations

- If `withdraw` and `deposit` use the same lock, then simultaneous calls to these methods are properly synchronized

- But what about `getBalance` and `setBalance`?
  - Assume they are `public`, which may be reasonable

- If they *do not* acquire the same lock, then a race between `setBalance` and `withdraw` could produce a wrong result

- If they *do* acquire the same lock, then `withdraw` would block forever because it tries to acquire a lock it already has!

One (not very good) possibility

```
int setBalance1(int x) {
   balance = x;
}
int setBalance2(int x) {
   lk.acquire();
   balance = x;
   lk.release();
}
void withdraw(int amount) {
   lk.acquire();
   …
   setBalance1(b - amount);
   lk.release();
}
```

Have two versions of setBalance!

- `withdraw` calls `setBalance1` (since it already has the lock)
- Outside world calls `setBalance2`
- Could work (if adhered to), but not good style; also not very convenient

- Alternately, we can modify the meaning of the Lock ADT to support ***re-entrant locks***
  - Java does this
  - Then just always use `setBalance2`

Re-entrant lock idea

A re-entrant lock (a.k.a. recursive lock)

- **The idea**: Once acquired, the lock is held by the Thread, and subsequent calls to `acquire` *in that Thread* won't block

- **Result**: `withdraw` can acquire the lock, and then call `setBalance`, which can also acquire the lock
  - Because they're in the same thread & it's a re-entrant lock, the inner `acquire` won't block!!

Re-entrant lock

A re-entrant lock (a.k.a. recursive lock)

- “Remembers”
  - the thread (if any) that currently holds it
  - a *count*
- When the lock goes from *not-held* to *held*, the count is set to 0
- If (code running in) the current holder calls `acquire` :
  - it does not block
  - it increments the count
- On `release` :
  - if the count is > 0, the count is decremented
  - if the count is 0, the lock becomes *not-held*

Re-entrant locks work

```
int setBalance(int x) {
  lk.acquire();
  balance = x;
  lk.release();
}

void withdraw(int amount) {
  lk.acquire();
  …
  setBalance(b - amount);
  lk.release();
}
```

This simple code works fine provided `lk` is a reentrant lock

- Okay to call `setBalance` directly
- Okay to call `withdraw` (won't block forever)

Java's Re-entrant Lock

- `java.util.concurrent.locks.ReentrantLock`
- Has methods `lock()` and `unlock()`
- As described above, it is conceptually owned by the Thread, and shared within that thread
- Important to guarantee that lock is ***always*** released!!!
- Recommend something like this:

```
myLock.lock();
try { // method body }
finally { myLock.unlock(); }
```

- Despite what happens in 'try', the code in finally will execute afterwards

`Synchronized`: *A Java convenience*

Java has built-in support for re-entrant locks

- You can use the `synchronized` statement as an alternative to declaring a `ReentrantLock`

```
synchronized (expression) {
  statements
}
```

1. Evaluates *expression* to an **object**
   - Every **object** (but not primitive types) "is a lock" in Java
2. Acquires the lock, blocking if necessary
   - "If you get past the `{`, you have the lock"
3. Releases the lock "at the matching `}`"
   - Even if control leaves due to `throw`, `return`, etc.
   - So *impossible* to forget to release the lock!

Java version #1 (correct but can be improved)

```
class BankAccount {
  private int balance = 0;
  private Object lk = new Object();
  int getBalance()
    { synchronized (lk) { return balance; } }
  void setBalance(int x)
    { synchronized (lk) { balance = x; } }
  void withdraw(int amount) {
    synchronized (lk) {
      int b = getBalance();
      if(amount > b)
        throw …
      setBalance(b - amount);
    }
  }
  // deposit would also use synchronized(lk)
}
```

Improving the Java

- As written, the lock is **private**
  - Might seem like a good idea
  - But also prevents code in other classes from writing operations that synchronize with the account operations

- More idiomatic is to synchronize on `this`…
  - Also more convenient: no need to have an extra object!

Java version #2

```
class BankAccount {
  private int balance = 0;
  int getBalance()
    { synchronized (this){ return balance; } }
  void setBalance(int x)
    { synchronized (this){ balance = x; } }
  void withdraw(int amount) {
    synchronized (this) {
      int b = getBalance();
      if(amount > b)
        throw …
      setBalance(b – amount);
    }
  }
  // deposit would also use synchronized(this)
}
```

Syntactic sugar

Version #2 is slightly poor style because there is a shorter way to say the same thing:

Putting `synchronized` before a method declaration means the entire method body is surrounded by

```
synchronized(this){…}
```

Therefore, **version #3 (next slide) means exactly the same thing as version #2** but is more concise

Java version #3 (final version)

```
class BankAccount {
  private int balance = 0;
  synchronized int getBalance()
    { return balance; }
  synchronized void setBalance(int x)
    { balance = x; }
  synchronized void withdraw(int amount) {
     int b = getBalance();
     if(amount > b)
       throw …
     setBalance(b – amount);
  }
  // deposit would also use synchronized
}
```

More Java notes

- Class `java.util.concurrent.locks.ReentrantLock` works much more like our pseudocode
  - Often use `try { … } finally { … }` to avoid forgetting to release the lock if there's an exception

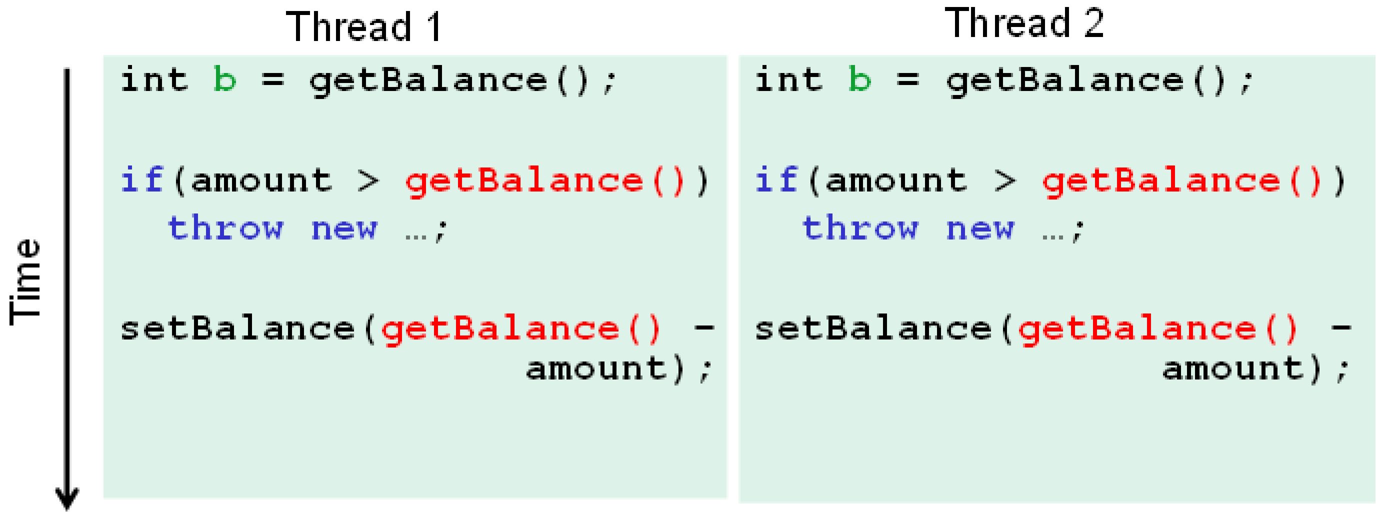
- Also library and/or language support for *readers/writer locks* and *condition variables* (see Grossman notes)

- Java provides many other features and details. See, for example:
  - Chapter 14 of CoreJava, Volume 1 by Horstmann/Cornell
  - Java Concurrency in Practice by Goetz et al

What could happen in this code?

Two threads both trying to `withdraw(100)` from the same account:

- Assume initial `balance` 150



Thread 1

```
int b = getBalance();

if(amount > getBalance())
  throw new …;

setBalance(getBalance() -
                 amount);
```

Thread 2

```
int b = getBalance();

if(amount > getBalance())
  throw new …;

setBalance(getBalance() -
                 amount);
```

Which of the following outcomes are possible given that we have two threads both trying to `withdraw(100)` from the same account (Assume initial `balance` 150)

```
class BankAccount {
  private int balance = 0;
  int  getBalance()      { return balance; }
  void setBalance(int x) { balance = x; }
  void withdraw(int amount) {
    if(amount > getBalance())
      throw new WithdrawTooLargeException();
    setBalance(getBalance() – amount);
  }
  … // other operations like deposit, etc.
}
```

a) balance = -50

b) balance = 50

c) balance = 50, and one thread throws an exception

d) balance = 150

e) balance = -50, and one thread throws an exception