

# Lecture 21:

# Shared-Memory Concurrency & Mutual Exclusion

CSE 332: Data Structures & Parallelism

Yafqa Khan

Summer 2025

# Today

- Fancier Parallel Patterns (Algorithms)
  - Prefix
  - Pack
- Concurrency: Synchronization
  - Concurrent Programming
  - Mutual Exclusion (Mutex)
  - Locks
  - Re-entrant Locks
- Concurrency: Synchronization Issues
  - Race Conditions: Data Races & Bad Interleavings
  - Deadlocks

# Today

- Fancier Parallel Patterns (Algorithms)
  - Prefix
  - Pack
- Concurrency: Synchronization
  - Concurrent Programming
  - Mutual Exclusion (Mutex)
  - Locks
  - Re-entrant Locks
- Concurrency: Synchronization Issues
  - Race Conditions: Data Races & Bad Interleavings
  - Deadlocks

# Sharing Resources

So far we've been writing parallel algorithms that don't share resources.

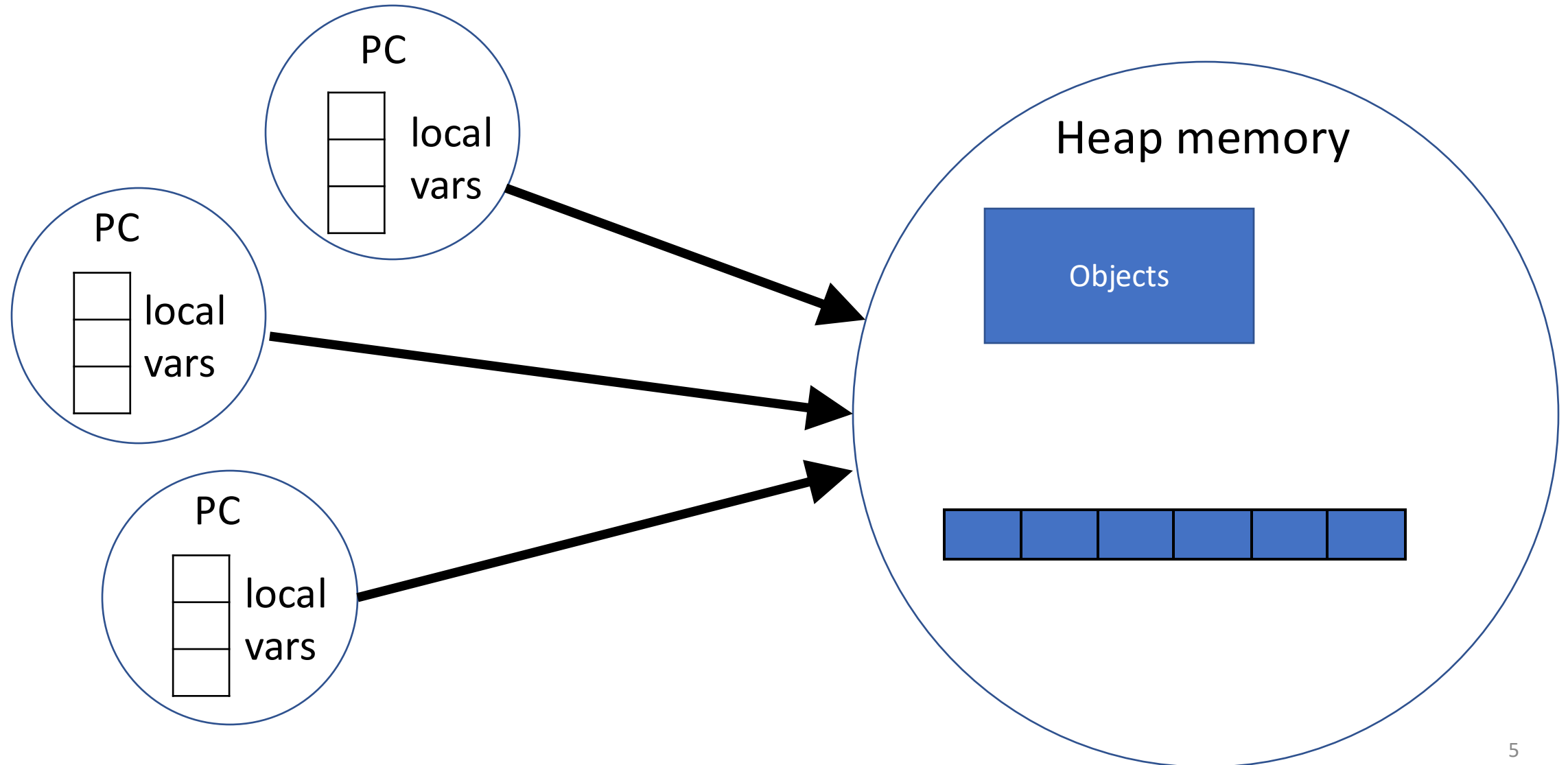
Fork-join algorithms all had a simple structure

- Each thread had memory only it accessed
- Results of one thread not accessed until joined.
- The **structure** of the code ensured sharing didn't go wrong.

Can't always use the same strategy when memory overlaps

- Thread doing independent tasks on same resources.

# Parallel Code



# Why Concurrency?

If we're not using them to solve the same big problem faster, why threads?

Threads useful for:

- *Code 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

# Concurrency

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

Instead of planning (ex: splitting up a task into multiple pieces), we need to *coordinate* how we use the same resources! (We might not be even doing the same thing!)

Even correct concurrent applications are usually highly **non-deterministic**

- how threads are scheduled affects what operations happen first
- non-repeatability complicates testing and debugging
- (Unproven) Magic property where code works when testing but fails during demo...

# 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;  
    }  
}
```



# Bad Interleaving

Any interleaving is possible!

Time  
↓

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

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

# 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.
}
```

# Activity: What is the balance at the end?

Two threads run: one withdrawing 100, another withdrawing 75, (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);`

# Activity: What is the balance at the end?

```
void withdraw(int amount) {  
    int b = getBalance();  
    if (amount > b)  
        throw new WithdrawTooLargeException();  
    setBalance(b - amount);  
}
```

Thread 1  
`x.withdraw(100);`

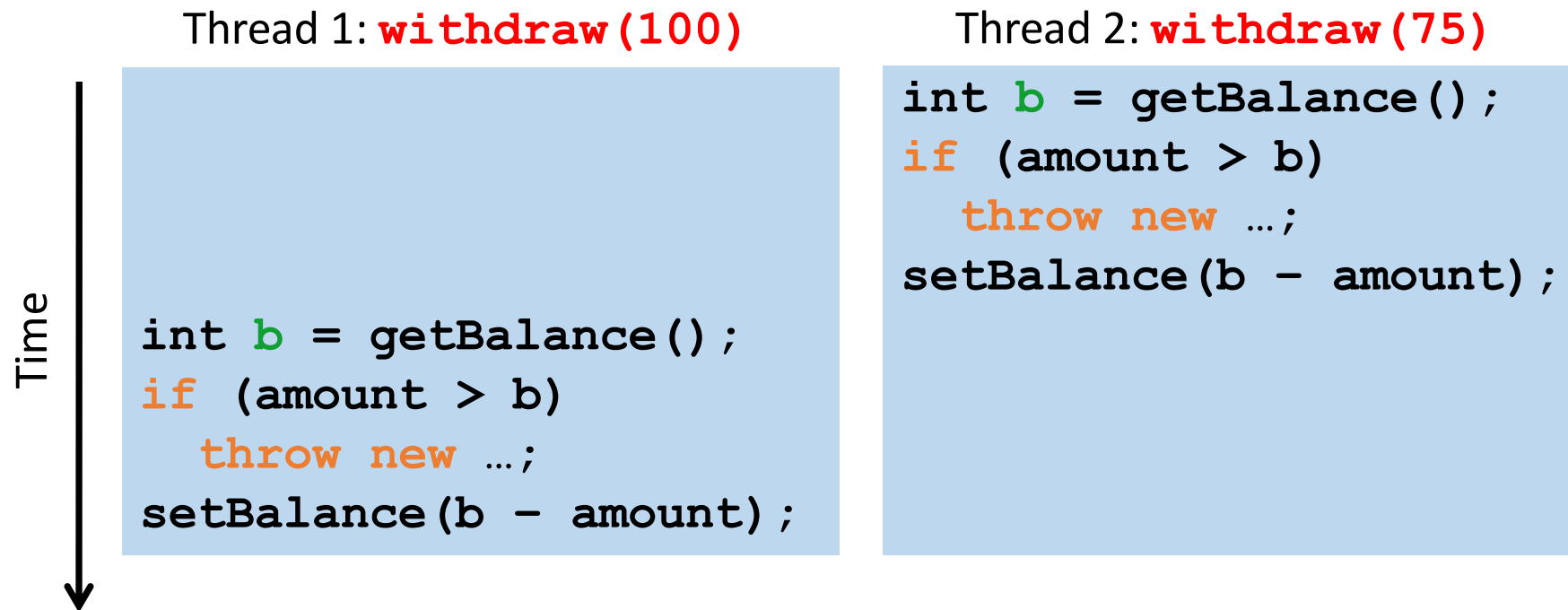
```
void withdraw(int amount) {  
    int b = getBalance();  
    if (amount > b)  
        throw new WithdrawTooLargeException();  
    setBalance(b - amount);  
}
```

Thread 2  
`x.withdraw(75);`

# Activity: A “good” execution is also possible

Interleaved **withdraw()** calls on the same account

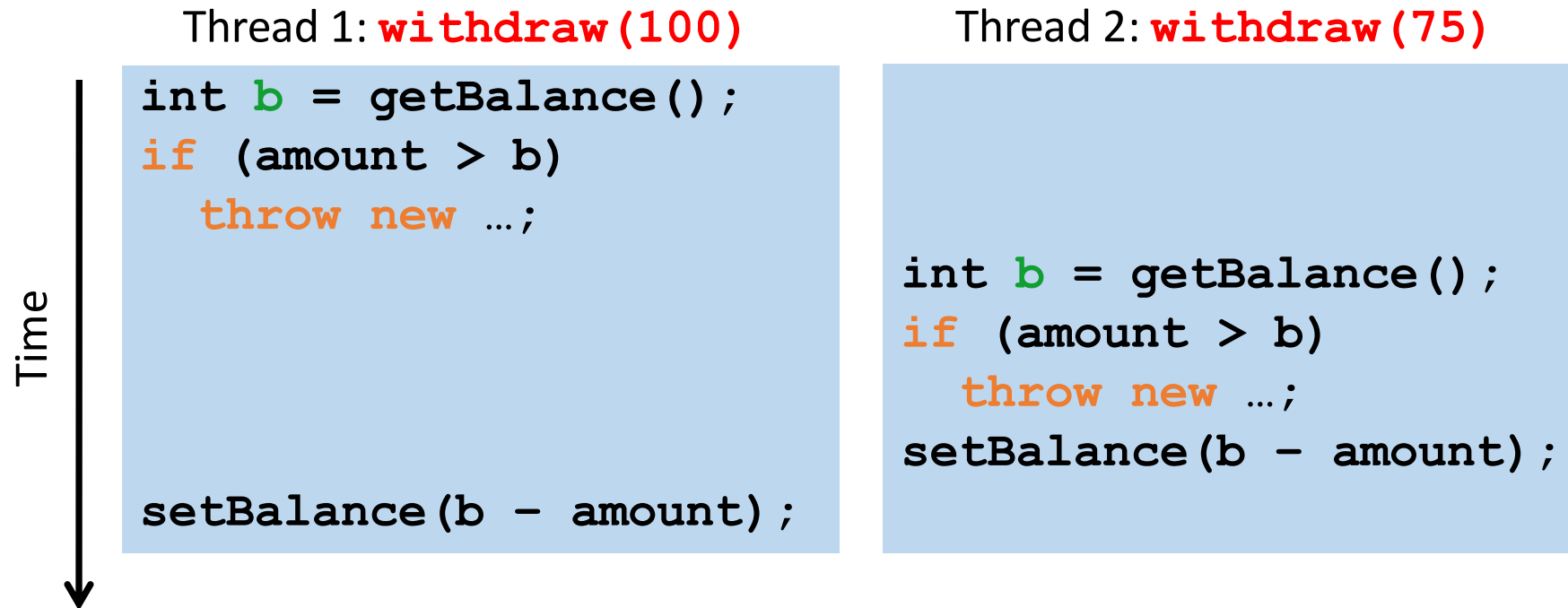
- Assume initial **balance == 150**
- This *should* cause a **WithdrawTooLarge** exception



# Activity: A bad interleaving

Interleaved **withdraw()** calls on the same account

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



# Bad Interleavings

- What's the problem?
- We stored the result of `balance` locally, but another thread overwrote it after we stored it.
- The value became stale.

# A Principle

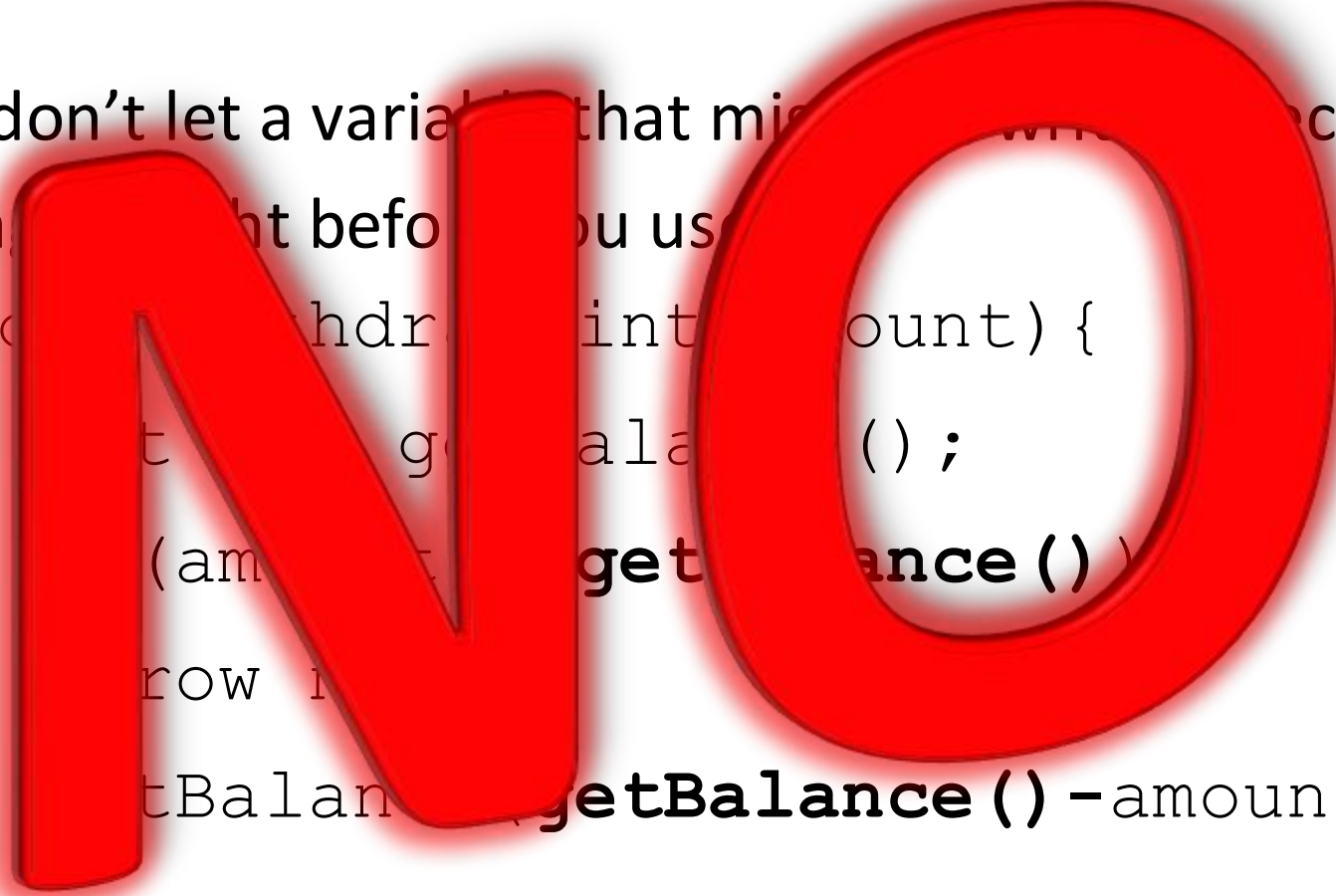
- Principle: don't let a variable that might be written become stale.
- Ask for it again right before you use it

```
void withdraw(int amount) {  
    int b = getBalance();  
    if (amount > getBalance())  
        throw new ...;  
    setBalance(getBalance() - amount);  
}
```



# A Principle

- Principle: don't let a variable that might become stale.
- Ask for it again before you use it.



```
void withdraw(int amount) {  
    if (getBalance() < amount) {  
        return;  
    }  
    setBalance(getBalance() - amount);  
}
```

**That's not a real concurrency principle. It doesn't solve anything.**

# 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?***

# There's still a bad interleaving, find one

Thread 1

`x.withdraw(100);`

Thread 2

`x.withdraw(75);`

```
void withdraw(int amount) {  
    int b = getBalance();  
    if (amount > getBalance())  
        throw new WithdrawTooLargeException();  
    setBalance(getBalance() - amount);  
}
```

```
void withdraw(int amount) {  
    int b = getBalance();  
    if (amount > getBalance())  
        throw new WithdrawTooLargeException();  
    setBalance(getBalance() - amount);  
}
```

# There's still a bad interleaving, find one

Thread 1

`x.withdraw(100);`

Thread 2

`x.withdraw(75);`

```
void withdraw(int amount) {  
    int b = getBalance();  
    if (amount > getBalance())  
        throw new WithdrawTooLargeException();  
  
    setBalance(getBalance() - amount);  
}
```

```
void withdraw(int amount) {  
  
    int b = getBalance();  
    if (amount > getBalance())  
        throw new WithdrawTooLargeException();  
    setBalance(getBalance() - amount);  
}
```

In this version, we can have negative balances without throwing the exception!

# There's still a bad interleaving, find one

Thread 1

`x.withdraw(100);`

Thread 2

`x.withdraw(75);`

```
void withdraw(int amount) {  
    int b = getBalance();  
    if (amount > getBalance())  
        throw new WithdrawTooLargeException();  
  
    getBalance() - amount  
  
    setBalance(<saved computation>);  
}
```

```
void withdraw(int amount) {  
  
    int b = getBalance();  
    if (amount > getBalance())  
        throw new WithdrawTooLargeException();  
  
    getBalance() - amount  
  
    setBalance(<saved computation>);  
}
```

# A Real Principle: Mutual Exclusion

**Mutual Exclusion** (aka Mutex, aka Locks)

Rewrite our code so at most one thread can use a resource at a time  
All other threads must wait.

We need to identify the **critical section**

Portion of the code only a single thread should be allowed to be in at once.

This **MUST** be done by the programmer.

But you need language primitives to do it!

# Implementing our own Mutex?

Idea: Maybe try using a Boolean flag?

```
void withdraw(int amount) {  
  
    int b = getBalance();  
    if (amount > b)  
        throw new WithdrawTooLargeException();  
    setBalance(b - amount);  
  
}  
  
// deposit would spin on same boolean
```

# Why is this Wrong?

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

```
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
```



# Still just moved the problem!

**Busy** is initially = false

Thread 1

Time ↓

```
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

# Locks

- We can still have a bad interleaving.
- If two threads see `busy = false` and get past the loop simultaneously.
- We need a single operation that
  - Checks if `busy` is false
  - AND sets it to true if it is
  - AND where no other thread can interrupt us.
- An operation is **atomic** if no other threads can interrupt it/interleave with it.

# What we need: Locks

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  $\geq 1$  threads are blocked on it, exactly 1 will acquire it

Note: 'Lock' is not an actual Java class

# Almost-correct pseudocode

```
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  
}
```

# Using Locks

## Questions:

1. What is the critical section (i.e. the part of the code protected by the lock)?
2. How many locks should we have
  - a) One per BankAccount object?
  - b) Two per BankAccount object (one in withdraw and a different lock in deposit)?
  - c) One (static) one for the entire class (shared by all BankAccount objects)?
3. There is a subtle bug in withdraw(), what is it?
4. Do we need locks for
  - a) getBalance()?
  - b) setBalance()?

For the purposes of this question, assume those methods are public.

# Some mistakes

## 2.b) **Incorrect**: Use different locks for **withdraw** and **deposit**

- Mutual exclusion works only when using same lock
- **balance** field is the shared resource being protected, not the methods themselves

## 2.c) **Poor performance**: Use same lock for every bank account

- Not technically incorrect, but...
- No simultaneous operations on *different* accounts

# Using Locks

## 3. The bug in withdraw:

When you throw an exception, you still hold onto the lock!

- You could release the lock before throwing the exception.

Or use try{} finally{} blocks

```
try { critical section }  
finally { lk.release() }
```

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

# Re-entrant Locks

4. Do we need to lock `setBalance()`  
If it's public, yes.

But now we have a problem:  
withdraw will acquire the lock,  
Then call `setBalance()`...  
Which needs the same lock



# 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 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)

Lock needs to know which **release** call is the “real” release, and which one is just the end of an inner method call.

Intuition: have a counter. Increment it when you “re-acquire” the lock, decrement when you release. Until releasing on 0 then really release.

Take an operating systems course to learn more.

# Real Java Locks

## `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) {  
    critical section  
}
```

1. *expression* must be 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

Any Questions?