

# 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*

- 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 from other threads they see when

- non-repeatability complicates testing and debugging

1

# Examples



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 of recent files (e.g., hashtable)
  - 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 to next thread in sequence?
  - What if enqueueer and dequeuer adjust a circular array queue at the same time?

2

## Java Concurrency



- The `java.lang.Runnable` interface
  - `void run();`
- The `java.lang.Thread` class
  - `Thread(Runnable r);`
  - `void start();`
  - `static void sleep(long millis);`
  - `void join();`
  - `boolean isAlive();`
  - `static Thread currentThread();`
- See `IncrementTest.java`

## Atomicity



- An action is atomic if it is indivisible
  - Effectively, it happens all at once
  - No effects of the action are visible until it is complete
  - No other actions have an effect during the action
- In Java, integer increment (`i++`) is not atomic
  1. Load data from variable
  2. Increment data by 1
  3. Store data to variable `i`

## Race Condition



- A race condition is when multiple threads access shared data and unexpected results occur depending on the order of their actions

## Alice, Bob Bill and the Bank



### A. Alice to pay Bob \$30

#### Bank actions

- A1. Does Alice have \$30 ?
- A2. Give \$30 to Bob
- A3. Take \$30 from Alice

### B. Alice to pay Bill \$30

#### Bank actions

- B1. Does Alice have \$30 ?
- B2. Give \$30 to Bill
- B3. Take \$30 from Alice

*If Alice starts with \$40, can Bob and Bill both get \$30?*

## Alice, Bob Bill and the Bank



**A. Alice to pay Bob \$30**

**Bank actions**

**A1. Does Alice have \$30 ?**

**A2. Give \$30 to Bob**

**A3. Take \$30 from Alice**

**A.1**

**A.2**

**B.1**

**B.2**

**A.3**

**B.3**

**B. Alice to pay Bill \$30**

**Bank actions**

**B1. Does Alice have \$30 ?**

**B2. Give \$30 to Bill**

**B3. Take \$30 from Alice**

*If Alice starts with \$40, can  
Bob and Bill both get \$30?*

## 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 resources



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

9

## 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.
}
```

10

## 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

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...

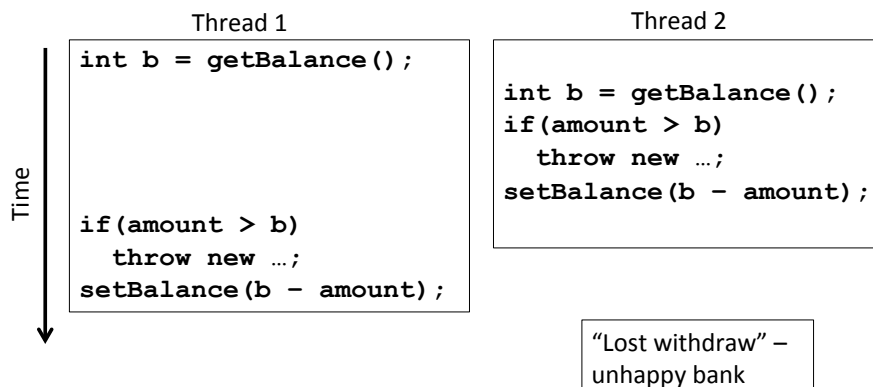
11

## A bad interleaving



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

- Assume initial **balance == 150**



12

## 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?

13

## Mutual exclusion



Sane 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: an account) means another thread must wait

- a.k.a. critical sections, which technically have other requirements

Programmer must implement critical sections

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

14

## Wrong!

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

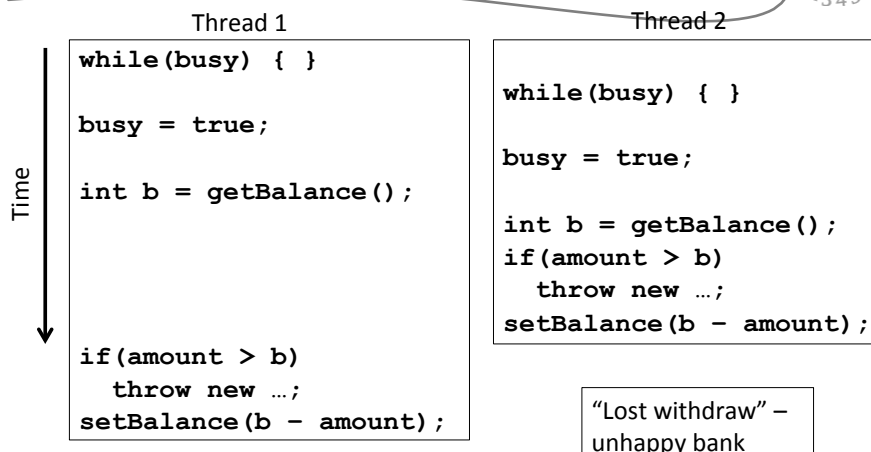
- It's technically possible under certain assumptions, but won't work in real languages anyway

```
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
}
```

15



## Just moved the problem!



16





## What we need



- ☞ We need help from the language
- ☞ One basic solution: Locks
  - Not Java yet, though Java's approach is similar and slightly more convenient
- ☞ 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!]
  - **release**: makes this lock "not held"
    - ✓ If  $\geq 1$  threads are blocked on it, exactly 1 will acquire it

17

## Why that works



- ☞ An ADT with operations **new**, **acquire**, **release**
- ☞ The lock implementation ensures that given simultaneous acquires and/or releases, a correct thing will happen
  - Example: 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
  - Here, we take this as a primitive and use it

18

## 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
}
```

19

## 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();
}
```

20

## 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

21

## Re-acquiring locks?



```
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();
}
```

- ✎ Can't let outside world call **setBalance1**
- ✎ Can't have **withdraw** call **setBalance2**
- ✎ Alternately, we can modify the meaning of the Lock ADT to support *re-entrant locks*

22

## 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*

23

## 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)

24

## Now some Java



Java has built-in support for re-entrant locks

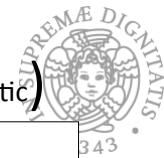
- Several differences from our pseudocode
- Focus on the **synchronized** statement

```
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

25

## Java version #1 (correct but non-idiomatic)



```
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)
}
```

26

## 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

27

## 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)
}

```

28

## 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

29

## 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
}
```

30

## 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* (future lecture)
- Java provides many other features and details. See, for example:
  - Java Concurrency in Practice by Goetz et al

31



32