# CSE 332 Autumn 2023 Lecture 24: Concurrency

Nathan Brunelle

http://www.cs.uw.edu/332

### Reasons to use threads (beyond algorithms)

- Code Responsiveness:
  - While doing an expensive computation, you don't what your interface to freeze
- Processor Utilization:
  - If one thread is waiting on a deep-hierarchy memory access you can still use that processor time
- Failure Isolation:
  - If one portion of your code fails, it will only crash that one portion.

### Memory Sharing With ForkJoin

- Idea of ForkJoin:
  - Reduce span by having many parallel tasks
  - Each task is responsible for its own portion of the input/output
  - If one task needs another's result, use join() to ensure it uses the final answer
- This does not help when:
  - Memory accessed by threads is overlapping or unpredictable
  - Threads are doing independent tasks using same resources (rather than implementing the same algorithm)

#### Example: Shared Queue

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

Imagine two threads are both using the same linked list based queue.

What could go wrong?

### **Concurrent Programming**

- Concurrency:
  - Correctly and efficiently managing access to shared resources across multiple possibly-simultaneous tasks
- Requires synchronization to avoid incorrect simultaneous access
  - Use some way of "blocking" other tasks from using a resource when another modifies it or makes decisions based on its state
  - That blocking task will free up the resource when it's done
- Warning:
  - Because we have no control over when threads are scheduled by the OS, even correct implementations are highly non-deterministic
  - Errors are hard to reproduce, which complicates debugging

### Bank Account Example

- The following code implements a bank account object correctly for a synchronized situation
- Assume the initial balance is 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.
```

What Happens here? withdraw(100); withdraw(75)

#### Bank Account Example - Parallel

• Assume the initial balance is 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:

Thread 2:

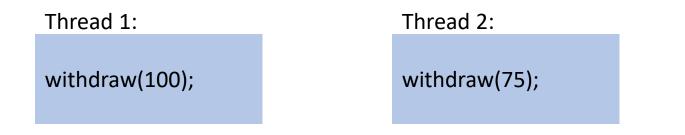
withdraw(75);

withdraw(100);

#### Interleaving

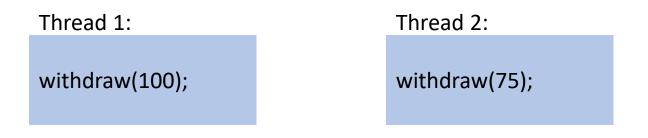
- Due to time slicing, a thread can be interrupted at any time
  - Between any two lines of code
  - Within a single line of code
- The sequence that operations occur across two threads is called an interleaving
- Without doing anything else, we have no control over how different threads might be interleaved

#### A "Good" Interleaving



	<pre>int b = getBalance(); if (amount &gt; b)</pre>
<pre>int b = getBalance();</pre>	
if (amount > b)	
throw new Exception();	
setBalance(b – amount);	

#### A "Bad" Interleaving



<pre>int b = getBalance();</pre>	
	int b = getBalance();
	if (amount > b)
	throw new Exception();
	setBalance(b – amount);
if (amount > b)	
throw new Exception();	
setBalance(b – amount);	

#### Another result?

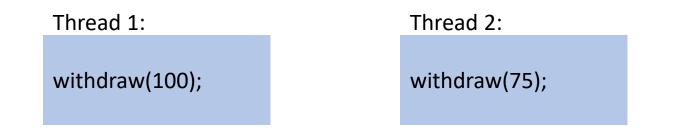


int b = getBalance();	int b = getBalance();
if (amount > b)	if (amount > b)
throw new Exception();	throw new Exception();
setBalance(b – amount);	setBalance(b – amount);

#### A Bad Fix

```
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 still "Bad" Interleaving



	if (amount > getBalance())
if (amount > getBalance())	
throw new Exception();	
<pre>setBalance(getBalance() - amount);</pre>	
<pre>setBalance(getBalance() - amount);</pre>	throw new Exception(); setBalance(getBalance() – amount);

#### What we want – Mutual Exclusion

- While one thread is withdrawing from the account, we want to exclude all other threads from also withdrawing
- 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**.
- The programmer must implement critical sections!
  - It requires programming language primitives to do correctly

#### A Bad attempt at Mutual Exclusion

class BankAccount {

```
private int balance = 0;
```

```
private Boolean busy = false;
```

```
int getBalance() { return balance; }
void setBalance(int x) { balance = x; }
```

```
void withdraw(int amount) {
```

```
while (busy) { /* wait until not busy */ }
```

```
busy = true;
```

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

```
if (amount > b)
```

```
throw new WithdrawTooLargeException();
```

```
setBalance(b – amount);
```

```
busy = false;}
```

```
// other operations like deposit, etc.
```

#### A still "Bad" Interleaving

	Thread 1:			Thread 2:	
	withdraw(100);			withdraw(75);	
while (busy busy = true; int b = getB		*/}	busy = int b = if (am setBa	<pre>(busy) { /* wait until no = true; = getBalance(); ount &gt; b) throw new Exception lance(b – amount); = false;</pre>	
	nrow new Exception(); b – amount);				

#### Solution

- We need a construct from Java to do this
- One Solution A Mutual Exclusion Lock (called a Mutex or Lock)
- We define a **Lock** to be a ADT with operations:
  - New:
    - make a new lock, initially "not held"
  - Acquire:
    - If lock is not held, mark it as "held"
      - These two steps always done together in a way that cannot be interrupted!
    - If lock is held, pause until it is marked as "not held"
  - Release:
    - Mark the lock as "not held"

## Almost Correct Bank Account Example

class BankAccount {

```
private int balance = 0;
private Lock lck = new Lock();
int getBalance() { return balance; }
```

```
void setBalance(int x) { balance = x; }
```

```
void withdraw(int amount) {
```

```
lk.acquire();
int b = getBalance();
if (amount > b)
        throw new WithdrawTooLargeException();
setBalance(b – amount);
lk.release();}
```

// other operations like deposit, etc.

**Questions:** 

- 1. What is the critical section?
- 2. What is the Error?

### Try...Finally

- Try Block:
  - Body of code that will be run
- Finally Block:
  - Always runs once the program exits try block (whether due to a return, exception, anything!)

## Correct (but not Java) Bank Account Example

class BankAccount {

```
private int balance = 0;
private Lock lck = new Lock();
int getBalance() { return balance; }
void setBalance(int x) { balance = x; }
void withdraw(int amount) {
```

#### try{

lk.acquire(); int b = getBalance(); if (amount > b) throw new WithdrawTooLargeException(); setBalance(b - amount); } finally { lk.release(); } } // other operations like deposit, etc.

#### Questions:

- 1. Should deposit have its own lock object, or the same one?
- 2. What about getBalance?
- 3. What about setBalance?

#### A still "Bad" Interleaving

	Thread 1:			Thread 2:	
	withdraw(100);			if(getBalance()<75) setBalance(75);	
int b if (am setBa	quire(); = getBalance(); hount > b) throw new Exception lance(b – amount); } y { lk.release(); }	on();	if(ge	etBalance() < 75) setBalance(75);	

#### What's wrong here...

class BankAccount {

```
private int balance = 0;
```

```
private Lock lck = new Lock();
```

int setBalance(int x) {

try{

```
lk.acquire();
balance = x; }
finally{ lk.release(); } }
void withdraw(int amount) {
```

#### try{

```
lk.acquire();
int b = getBalance();
if (amount > b)
throw new WithdrawTooLargeException();
setBalance(b - amount); }
finally { lk.release(); } }}
```

Withdraw calls setBalance!

Withdraw can never finish because in setBalance the lock will always be held!

### Re-entrant Lock (Recursive Lock)

- Idea:
  - Once a thread has acquired a lock, future calls to acquire on the same lock will not block progress
- If the lock used in the previous slide is re-entrant, then it will work!

#### Re-entrant Lock Details

- A re-entrant lock (a.k.a. recursive lock)
- "Remembers"
  - the thread (if any) that currently holds it
  - a count of "layers" that the thread holds it
- 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

#### Java's Re-entract Lock Class

- java.util.concurrent.locks.ReentrantLock
- Has methods lock() and unlock()
- Important to guarantee that lock is always released!!!
- Recommend something like this:

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

#### How this looks in Java

#### java.util.concurrent.locks.ReentrantLock;

```
class BankAccount {
          private int balance = 0;
          private ReentrantLock lck = new ReentrantLock();
          int setBalance(int x) {
                    try{
                              lk.lock();
                              balance = x; }
                    finally{ lk.unlock(); } }
          void withdraw(int amount) {
                    try{
                              lk.lock();
                              int b = getBalance();
                              if (amount > b)
                                        throw new WithdrawTooLargeException();
                              setBalance(b - amount); }
                    finally { lk.unlock(); } }}
```

#### Java Synchronized Keyword

- Syntactic sugar for re-etrant locks
- You can use the synchronized statement as an alternative to declaring a ReentrantLock
- Syntax: synchronized(/\* expression returning an Object \*/) {statements}
- Any Object can serve as a "lock"
  - Primitive types (e.g. int) cannot serve as a lock
- Acquires a lock and blocks if necessary
  - Once you get past the "{", you have the lock
- Released the lock when you pass "}"
  - Even in the cases of returning, exceptions, anything!
  - Impossible to forget to release the lock

## Back Account Using Synchronize (Attempt 1)

```
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 new Exception();
                  setBalance(b - amount); } } // deposit would also use synchronized(lk)
```

#### Back Account Using Synchronize (Attempt 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 new Exception();
                  setBalance(b - amount); } } // deposit would also use synchronized(lk)
```

Since we have one lock per account regardless of operation, it's more intuitive to use the account object itself as the lock!

#### More Syntactic Sugar!

- Using the object itself as a lock is common enough that Java has convenient syntax for that as well!
- Declaring a method as "synchronized" puts its body into a synchronized block with "this" as the lock

#### Back Account Using Synchronize (Final) 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 new WithdrawTooLargeException();

setBalance(b - amount); }

// other operations like deposit (which would use synchronized)