# CSE 332 Winter 2026
# Lecture 19: Concurrency 2

Nathan Brunelle

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

# 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

# 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

# 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.
}
```

# 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 an 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”

# Correct Withdraw

```
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?
2. What about getBalance?
3. What about setBalance?
4. Should they share one?

# Solution: getBalance needs the lock

```
class BankAccount {
        private int balance = 0;
        private Lock lck = new Lock();
        int getBalance(int x) {
                try{
                        lk.acquire();
                        return balance; }
                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(); } }}
```

One more issue!

Withdraw calls getBalance!

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

# 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-entrant 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-entrant 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 (version 1)

```
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 new Exception();
                        setBalance(b – amount); } }
}
```

# Back Account Using Synchronize (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 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)
}
```

# Concurrency Races

- **Race Condition:**
  - Occurs when the computation result depends on scheduling (how threads are interleaved)
  - **Java Example:** Two threads call withdraw. Different schedules cause different threads to see the Exception
  - **Trident Example:** The trident did not prevent a race condition, *which* name is the last one in the box may differ, depending on who grabbed the trident first
- **Data Race:**
  - Either:
    - There is the potential for two threads to be writing to a variable in parallel
    - There is the potential for one thread to be reading a variable while another writes to it
  - **Java Example:** Two different threads attempt to change a static variable
  - **Trident Example:** The trident corrected a data race, no two people can use the box at the same time
- **Bad Interleaving:**
  - An error due to a race condition other than a data race
    - It's a race condition the results in behavior our specification says in incorrect
  - Usually it looks like exposing a "bad" intermediate state
  - **Java Example:** Two threads insert into a hash table. One thread computes the index, the other resizes the table. Now the index might be incorrect.
  - **Trident Example:** The race condition present after the trident is not a bad interleaving because either result is still correct

# Example: Shared Stack (no problems so far)

```
class Stack {
        private E[] array = (E[])new Object[SIZE];
        private int end = -1;
        synchronized boolean isEmpty() {
                return end ==-1;
        }
        synchronized void push(E val) {
                array[++ end] = val;
        }
        synchronized E pop() {
                if(isEmpty())
                        throw new StackEmptyException();
                return array[end--];
        } }
```

Critical sections of this code?

# Race Condition, but no Data Race

```
class Stack {
        private E[] array = (E[])new Object[SIZE];
        private int end = -1;
        synchronized boolean isEmpty() { … }
        synchronized void push(E val) { … }
        synchronized E pop() { … }
        E peek(){
                E ans = pop();
                push(ans);
                return ans;
        }
}
```

Critical sections of this code?

# Two Peeks Break LIFO Order

Thread 1:

```
E ans = pop();
push(ans);
return ans;
```

Thread 2:

```
E ans = pop();
push(ans);
return ans;
```

| | |
|---|---|
| E ans = pop();<br><br>push(ans);<br>return ans; | E ans = pop();<br><br><br>push(ans);<br>return ans; |

# Race Condition, including a Data Race

```
class Stack {
        private E[] array = (E[])new Object[SIZE];
        private int index = -1;
        synchronized boolean isEmpty() { … }
        synchronized void push(E val) { … }
        synchronized E pop() { … }
        E peek(){
                System.out.println(end);
                E ans = pop();
                push(ans);
                return ans;
        }
}
```

# Peek and isEmpty

**Expected Behavior:**
Thread 2 should not see an empty stack if there is a push but no pop.

Thread 1:

```
peek();
```

Thread 2:

```
push(x);
boolean b = isEmpty();
```

| | |
|---|---|
| E ans = pop();<br><br>push(ans);<br>return ans; | push(x);<br><br>boolean b = isEmpty(); |

# Peek and Push

**Expected Behavior:**
Items from a stack are popped in LIFO order

Thread 1:

```
peek();
```

Thread 2:

```
push(x);
push(y);
System.out.println(pop());
System.out.println(pop());
```

| | |
|---|---|
| E ans = pop();<br>push(ans);<br>return ans; | push(x);<br>push(y);<br>System.out.println(pop());<br>System.out.println(pop()); |

# Peek and Push

**Expected Behavior:**
Thread 2 items from a stack are popped in LIFO order

Thread 1:

```
peek();
```

Thread 2:

```
push(x);
push(y);
System.out.println(pop());
System.out.println(pop());
```

| | |
|---|---|
| E ans = pop();<br><br>push(ans);<br>return ans; | push(x);<br><br>push(y);<br><br><br>System.out.println(pop());<br>System.out.println(pop()); |

# How to fix this?

Make a bigger critical section

```
class Stack {
        private E[] array = (E[])new Object[SIZE];
        private int end = -1;
        synchronized boolean isEmpty() { … }
        synchronized void push(E val) { … }
        synchronized E pop() { … }
        E peek(){
                E ans = pop();
                push(ans);
                return ans;
        }
}
```

# Fixed!

Make a bigger critical section

```
class Stack {
        private E[] array = (E[])new Object[SIZE];
        private int end = -1;
        synchronized boolean isEmpty() { … }
        synchronized void push(E val) { … }
        synchronized E pop() { … }
        synchronized E peek(){
                E ans = pop();
                push(ans);
                return ans;
        }
}
```

# Did this fix it?

```
class Stack {
        private E[] array = (E[])new Object[SIZE];
        private int end = -1;
        synchronized boolean isEmpty() { … }
        synchronized void push(E val) { … }
        synchronized E pop() { … }
        E peek(){
                return array[end];
        }
}
```

No! Now it has a data race!

Push/pop will be changing end!

# Deadlock

- Occurs when two or more threads are mutually blocking each other
- T1 is blocked by T2, which is blocked by T3, …, Tn is blocked by T1
  - A cycle of blocking

- Three requirements for deadlock:
  - Multiple threads each need to acquire **multiple locks**
  - The locks need to be **held at the same time** by the threads
  - The locks may be **acquired in multiple orders**

# Bank Account

```
class BankAccount {
        …
        synchronized void withdraw(int amt) {…}
        synchronized void deposit(int amt) {…}
        synchronized void transferTo(int amt, BankAccount a) {
                this.withdraw(amt);
               a.deposit(amt);
        }
}
```

# Deadlock Example

Thread 1:

```
x.transferTo(1,y);
```

Thread 2:

```
y.transferTo(1,x);
```

| | |
|---|---|
| **acquire lock for account x** b/c transferTo is synchronized<br>**acquire lock for account y** b/c deposit is synchronized<br>**release lock for account y** after depost<br>**release lock for account x** at end of transferTo | **acquire lock for account y** b/c transferTo is synchronized<br>**acquire lock for account x** b/c deposit is synchronized<br>**release lock for account x** after deposit<br>**release lock for account y** at end of transferTo |

# Deadlock Example

Thread 1:

```
x.transferTo(1,y);
```

Thread 2:

```
y.transferTo(1,x);
```

| | |
|---|---|
| **acquire lock for account x** b/c transferTo is synchronized<br><br>**acquire lock for account y** b/c deposit is synchronized<br><br>**release lock for account y** after depost<br><br>**release lock for account x** at end of transferTo | **acquire lock for account y** b/c transferTo is synchronized<br><br>**acquire lock for account x** b/c deposit is synchronized<br><br>**release lock for account x** after deposit<br><br>**release lock for account y** at end of transferTo |

# Resolving Deadlocks

- Option 1: Address the “multiple locks” requirement
  - Have a coarser lock granularity
  - E.g. one lock for ALL bank accounts
- Option 2: Address the “held at the same time” requirement
  - Have a finer critical section so that only one lock is needed at a time
  - E.g. instead of a synchronized transferTo, have the withdraw and deposit steps locked separately
- Option 3: Address the “acquired in multiple orders” requirement
  - Force the threads to always acquire the locks in the same order
  - E.g. make transferTo acquire both locks before doing either the withdraw or deposit, make sure both threads agree on the order to aquire

# Option 1: Coarser Locking

```
static final Object BANK = new Object();
class BankAccount {
        …
        synchronized void withdraw(int amt) {…}
        synchronized void deposit(int amt) {…}
        void transferTo(int amt, BankAccount a) {
                synchronized(BANK){
                        this.withdraw(amt);
                        a.deposit(amt);
                }
        }
}
```

# Option 2: Finer Critical Section

```
class BankAccount {
        …
        synchronized void withdraw(int amt) {…}
        synchronized void deposit(int amt) {…}
        void transferTo(int amt, BankAccount a) {
                synchronized(this){
                        this.withdraw(amt);
                }
                synchronized(a){
                        a.deposit(amt);
                }
        }
}
```

# Option 3: First Get All Locks In A Fixed Order

```
class BankAccount {
        …
        synchronized void withdraw(int amt) {…}
        synchronized void deposit(int amt) {…}
        void transferTo(int amt, BankAccount a) {
                if (this.acctNum < a.acctNum){
                        synchronized(this){
                                synchronized(a){
                                        this.withdraw(amt);
                                        a.deposit(amt);
                } } }
                else {
                        synchronized(a){
                                synchronized(this){
                                        this.withdraw(amt);
                                        a.deposit(amt);
                } } }
        }
}
```