

CSE 332 Summer 2026

Lecture 19: Concurrency

Michael Whitmeyer

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

Memory Sharing With ForkJoin

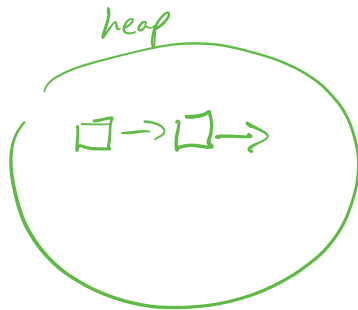
- Structure of ForkJoin:
 - All threads are executing the same algorithm
 - Each task is responsible for **its own portion** of the input/output
 - If one task needs another's result, use `join()` to ensure it uses the final answer
- This does not help when:
 - Memory accessed by threads is overlapping 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;  
    }  
    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?



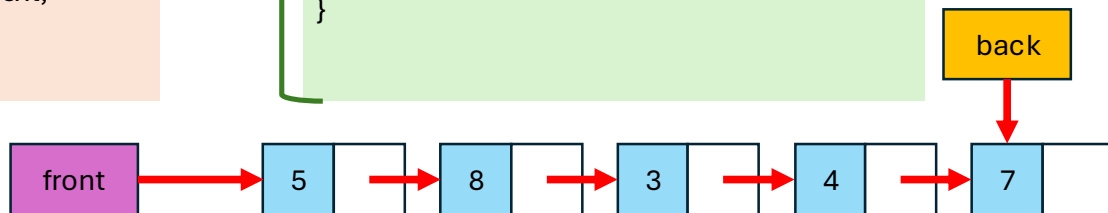
Shared Queue Interleaving

Thread

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

Thread

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



Empty Shared Queue

Suppose the Queue is empty, and the threads execute in this order. Assume Thread 1 enqueues 1, and Thread 2 enqueues 2.

1

Thread

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

front

2

Thread

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

back

front → [2] ← back
front → [1] ← back

Thread 1 – first two lines

Thread 1 has decided the queue is empty

Thread 1

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

Thread 2

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

front

back

Thread 2 – all lines

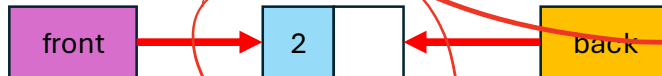
Thread 1 has decided the queue is empty
Meanwhile, thread 2 enqueues 2

Thread 1

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

Thread 2

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



Thread 1 – remaining lines

Thread 1 has decided the queue is empty
Meanwhile, thread 2 enqueues 2
Thread 1 continues as if the queue is still
empty, overwriting Thread 2's work

```
Thread 1  
enqueue(x){  
    _____ if ( back == null ){  
        _____  
        _____ back = new Node(x);  
        _____ front = back;  
    _____ }  
    _____ else {  
        _____ back.next = new Node(x);  
        _____ back = back.next;  
    _____ }  
}
```

```
Thread 2  
enqueue(x){  
    _____ if ( back == null ){  
        _____ back = new Node(x);  
        _____ front = back;  
    _____ }  
    _____ else {  
        _____ back.next = new Node(x);  
        _____ back = back.next;  
    _____ }  
}
```



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

Analogue Example – Data Race

1. Count to 10 aloud
2. Clap three times
3. Erase the contents of the box
4. Write your name in the box,
pausing 2 seconds between letters
5. Wave your arms in the air

Analogue Example – With “Mutual Exclusion”

1. Count to 10 aloud
2. Clap three times
- 3. Pick up the book**
4. Erase the contents of the box
5. Write your name in the box
- 6. Put down the book**
7. Wave your arms in the air

There is only one  trident, so no two threads (both executing this same code) can execute lines 4 and 5 at the same time.

All threads after the first must wait at line 3 for the trident to become available.

The trident causes any one thread to “lock out” (exclude) all other threads.

Any thread being between lines 3 and 6 excludes all other threads, so this is called “mutual exclusion”.

The trident itself we call a “**lock**”

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.  
}
```

balance 50

Thread 1:

withdraw(100);

Thread 2:

withdraw(75);

An “OK” Interleaving

- Assume the initial balance is 150

Thread 1:

```
withdraw(100);
```

Thread 2:

```
withdraw(75);
```

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

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

account 75

A “Bad” Interleaving

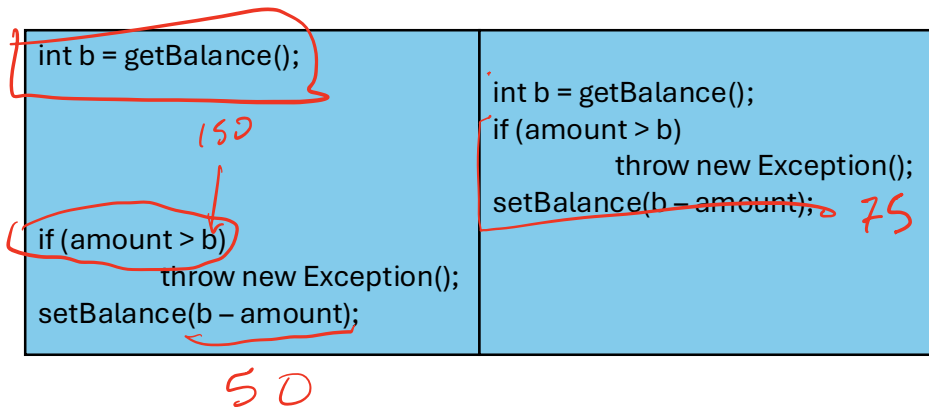
- Assume the initial balance is 150

Thread 1:

```
withdraw(100);
```

Thread 2:

```
withdraw(75);
```



A Bad Fix

- 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) {  
        if (amount > getBalance())  
            throw new WithdrawTooLargeException();  
        setBalance(getBalance() - amount); }  
    // other operations like deposit, etc.  
}
```

Avoiding Variables doesn't work

- Assume the initial balance is 150

Thread 1:

```
withdraw(100);
```

Thread 2:

```
withdraw(75);
```

<pre>if (amount > getBalance()) throw new Exception(); setBalance(getBalance() - amount); setBalance(getBalance() - amount);</pre> Handwritten annotations: <i>150</i> above the first <code>if</code>, <i>150</i> above the first <code>setBalance</code>, and <i>50</i> below the entire block.	<pre>if (amount > getBalance()) throw new Exception(); setBalance(getBalance() - amount);</pre> Handwritten annotations: <i>150</i> above the first <code>if</code>, a red circle around the <code>setBalance</code> line, and <i>75</i> below the entire block.
---	--

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.  
    }  
}
```

will draw

The “Busy Signal” Doesn’t Work

- Assume the initial balance is 150

Thread 1:

withdraw(100);

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

busy = true;

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

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

```
    throw new Exception();
```

```
setBalance(b - amount);
```

```
busy = false;
```

Thread 2:

withdraw(75);

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

busy = true;

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

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

```
    throw new Exception();
```

```
setBalance(b - amount);
```

```
busy = false;
```

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”

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?

*critical
section*

*does not release the
lock.*

Exceptions End the Method!

```
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.  
    }  
}
```

If we throw an exception, we never finish the method!

The lock would never get released!

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

Both Threads Need the Lock for Exclusion

- Assume the initial balance is 150

Thread 1:

withdraw(100);

Thread 2:

closeAccount();

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

```
int b = getBalance();  
setBalance(0);  
return b;
```

50

150

Solution: All methods using balance need the lock

```
class BankAccount {  
    private int balance = 0;  
    private Lock lk = 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(); } }  
}
```

Solution: All methods using balance need 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(); ] can't 0  
            int b = getBalance(); ] can't 1  
            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

import java.util.concurrent.locks.ReentrantLock;

```
class BankAccount {  
    private int balance = 0;  
    private ReentrantLock lk = 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*/) {...}
- 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

critical

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)  
}
```