CSE 332 Summer 2026
Lecture 20: Concurrency 2

Michael Whitmeyer

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

Last Time: Locks

- **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”

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(); } }}
```

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 (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)
  - **Example:** Two threads call withdraw. Different schedules cause different threads to see the Exception (withdraw(100) vs. withdraw(75)).
- **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
  - **Example:** Two different threads attempt to change the same object in the shared heap
- **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
  - **Example:** Two threads insert into a hash table. One thread computes the index, the other resizes the table. Now the index might be incorrect.

Example: Weird Stack

Critical sections of this code?

```
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;
        }
}
```

.peek() is.. weird

Race condition? Data Race?

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(); push(ans); return ans;	E ans = pop(); push(ans); 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); // this line creates a data race
                E ans = pop();
                push(ans);
                return ans;
        }
}
```

Bad Interleaving - Peek and isEmpty

Expected Behavior:
We push but never pop, so the stack should not be empty.

Thread 1:

```
peek();
```

Thread 2:

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

E ans = pop();
 	
push(ans);
return ans;
	push(x);
	
	boolean b = isEmpty();

Challenge - 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();
push(ans);
return ans;
	push(x);
	push(y);
	System.out.println(pop());
	System.out.println(pop());

Bad Interleaving - 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();
 	
push(ans);
return ans;
	push(x);
	
	push(y);
	
	System.out.println(pop());
	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];
        }
}
```

Did this fix it?

No! Now it has a data race!

Push/pop will be changing end!

```
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];
        }
}
```

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**

Analogy – Need the Phone to Dance, Need the Book to Speak

Michael

Acquire the phone

Do a little dance move

Acquire the book

Spell my last name

Release the book

Release the phone

Volunteer

Acquire the book

Spell your last name

Acquire the phone

Do a little dance move

Release the phone

Release the book

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 – Locking Order

Thread 1:

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

Thread 2:

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

acquire lock for account x b/c transferTo is synchronized **acquire lock for account y** b/c deposit is synchronized **release lock for account y** after deposit **release lock for account x** at end of transferTo	**acquire lock for account y** b/c transferTo is synchronized **acquire lock for account x** b/c deposit is synchronized **release lock for account x** after deposit **release lock for account y** at end of transferTo

Deadlock Example – Deadlock Interleaving

Thread 1:

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

Thread 2:

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

acquire lock for account x b/c transferTo is synchronized	
acquire lock for account y b/c deposit is synchronized	
release lock for account y after depost	
release lock for account x at end of transferTo	
	acquire lock for account y b/c transferTo is synchronized
	acquire lock for account x b/c deposit is synchronized
	release lock for account x after deposit
	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 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);
                }
        }
}
```

(Redundant code, just to highlight)

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);
                }}}
        }
}
```

Warning! Deadlock is Tricky!

```
public class ParallelArrayList{
        // Suppose other methods and fields are here.
        public void swap(int x, int y){
                Object xObj = get(x);
                Object yObj = get(y);
                if(x < y){
                        synchronized(xObj){
                                synchronized(yObj){
                                        set(a, yObj); set(b, xObj); } } }
                else{
                        synchronized(yObj){
                                synchronized(xObj){
                                        set(a, yObj); set(b, xObj); } } } }
```

Issue?

Tricky Deadlock

Suppose a < b < c

Thread 1:

```
swap(a, b);
```

Thread 2:

```
swap(b, c);
```

Thread 3:

```
swap(a, c);
```

```
cObj = get(a); \\ got swapped!
bObj = get(b);
Acquire lock for cObj

Wait for bObj lock
```

```
bObj = get(b);
cObj = get(c);



Acquire lock for bObj

Wait for cObj lock
```

```
swap(a, c);
```

Parallelism/Concurrency Discussion

Memory Categories

All memory must fit one of three categories:

1. Thread Local: Each thread has its own copy
2. Shared and Immutable: There is just one copy, but nothing will ever write to it
3. Shared and Mutable: There is just one copy, it may change
   - Requires Synchronization!

Lock Granularity

- Coarse Grained: Fewer locks guarding more things each
  - One lock for an entire data structure
  - One lock shared by multiple objects (e.g. one lock for all bank accounts)
- Fine Grained: More locks guarding fewer things each
  - One lock per data structure location (e.g. array index)
  - One lock per object or per field in one object (e.g. one lock for each account)
- Note: there’s really a continuum between them…

Example: Separate Chaining Hashtable

- Coarse-grained: One lock for the entire hashtable
- Fine-grained: One lock for each bucket
- Which supports more parallelism in insert and find?
- Which makes rehashing easier?
- What happens if you want to have a size field?

Tradeoffs

- Coarse-Grained Locking:
  - Simpler to implement and avoid race conditions
  - Faster/easier to implement operations that access multiple locations (because all guarded by the same lock)
  - Much easier for operations that modify data-structure shape
- Fine-Grained Locking:
  - More simultaneous access (performance when coarse grained would lead to unnecessary blocking)
  - Can make multi-location operations more difficult: say, rotations in an AVL tree
- Guideline:
  - Start with coarse-grained, make finer only as necessary to improve performance

Atomicity

- Atomic: indivisible
- Atomic operation: one that should be thought of as a single step
- Some sequences of operations should behave as if they are one unit
  - Between two operations you may need to avoid exposing an intermediate state
  - Usually ADT operations should be atomic
    - You don’t want another thread trying to do an insert while another thread is rotating the AVL tree
- Think first in terms of what operations need to be atomic
  - Design critical sections and locking granularity based on these decisions

Use Pre-Tested Code

- Whenever possible, use built-in libraries!
- Other people have already invested tons of effort into making things both efficient and correct, use their work when you can!
  - Especially true for concurrent data structures
  - Use thread-safe data structures when available
    - E.g. Java has **ConcurrentHashMap!**