


CSE 332 Winter 2024

Lecture 23: Race Conditions, Deadlock

A red hand-drawn underline is present under the words "Race Conditions," and "Deadlock". The line starts under the comma after "Conditions," and extends to the right, then loops back under "Deadlock" and ends under the "k".

Nathan Brunelle

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

Race Condition

- Occurs when the computation result depends on scheduling (how threads are interleaved)
 - We, as programmers can't influence scheduling of threads
 - We need to write programs that work independent of scheduling
- Data Race:
 - When there is the potential for two threads to be writing a variable in parallel
 - When there is the potential for one thread to be reading a variable while another writes to it
- Bad Interleaving:
 - A race condition other than a data race
 - Usually it looks like exposing a "bad" intermediate state

Example: Shared Stack (no problems so far)

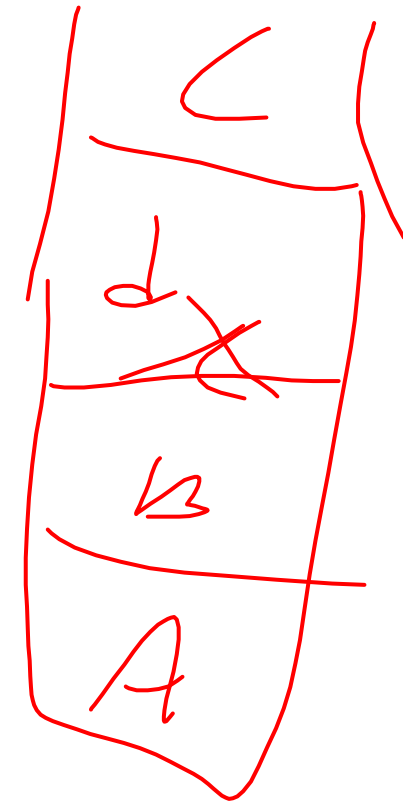
```
class Stack {  
    private E[] array = (E[])new Object[SIZE];  
    private int index = -1;  
    synchronized boolean isEmpty() {  
        return index == -1;  
    }  
    synchronized void push(E val) {  
        array[++index] = val;  
    }  
    synchronized E pop() {  
        if (isEmpty())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

Critical sections of this code?

↳
EXCL!

Race Condition, but no Data Race

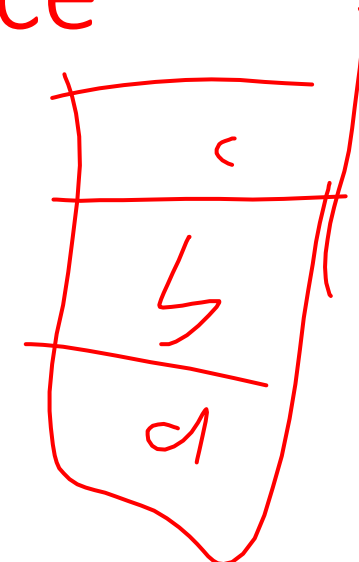
```
class Stack {  
    private E[] array = (E[])new Object[SIZE];  
    private int index = -1;  
    synchronized boolean isEmpty() { ... }  
    synchronized void push(E val) { ... }  
    synchronized E pop() { ... }  
    E peek(){  
        E ans = pop();  
        push(ans);  
        return ans;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



Critical sections of this code?

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



3
2

Peek and isEmpty

Expected Behavior:

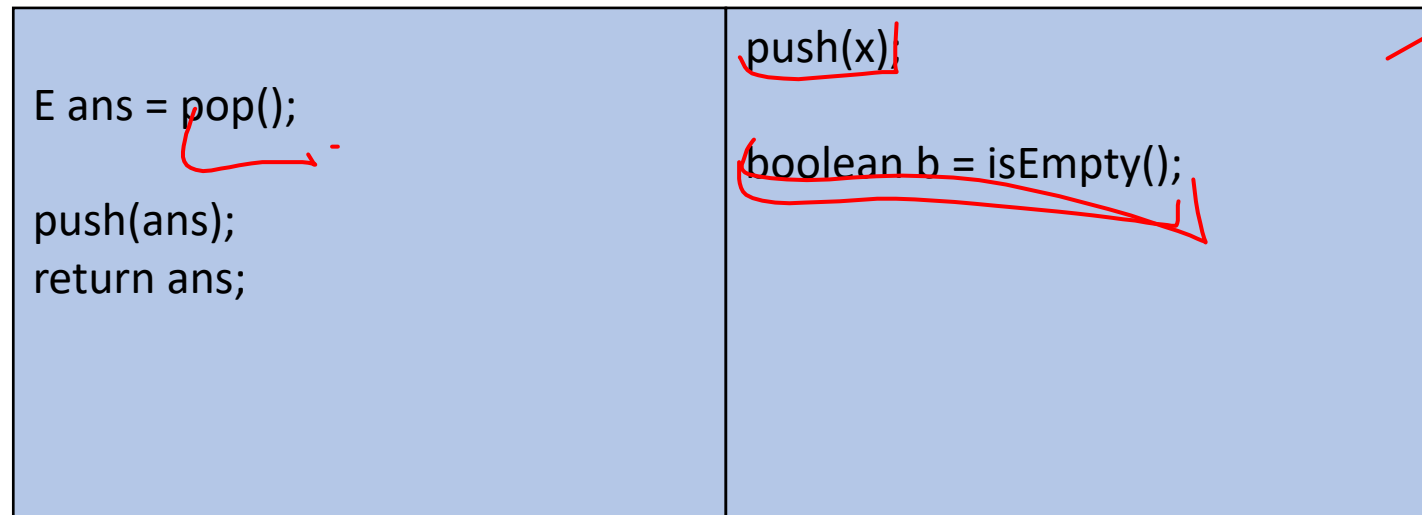
Thread 2 should not see an empty stack if there is a push but no pop.

Thread 1:

```
peek();
```

Thread 2:

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



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());
```

```
int ans = pop();
```

```
push(ans);  
return ans;
```

```
push(x);  
push(y);
```

```
System.out.println(pop());
```

```
System.out.println(pop());
```

Peek and Pop

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());
```

```
int 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 index = -1;  
    synchronized boolean isEmpty() { ... }  
    synchronized void push(E val) { ... }  
    synchronized E pop() { ... }  
    E peek(){  
        E ans = pop();  
        push(ans);  
        return ans;  
    }  
}
```

How to fix this?

Make a bigger critical section

```
class Stack {  
    private E[] array = (E[])new Object[SIZE];  
    private int index = -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?

No! Now it has 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(){  
        return array[index];  
    }  
}
```

Parallel Code Conventional Wisdom

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!

Thread Local Memory

- Whenever possible, avoid sharing resources
- Dodges all race conditions, since no other threads can touch it!
 - No synchronization necessary! (Remember Ahmdal's law)
- Use whenever threads do not need to communicate using the resource
 - E.g., each thread should have its own Random object
- In most cases, most objects should be in this category

Immutable Objects

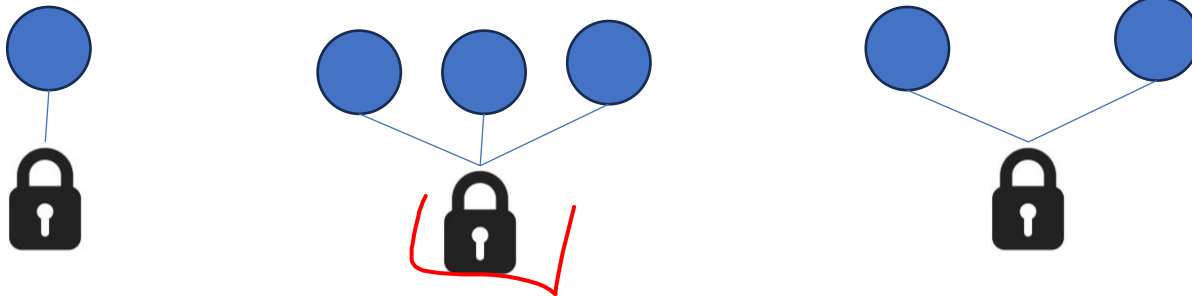
- Whenever possible, avoid changing objects
 - ~~Make~~ new objects instead
- Parallel reads are not data races
 - If an object is never written to, no synchronization necessary!
- Many programmers over-use mutation, minimize it

Shared and Mutable Objects

- For everything else, use locks
- Avoid all data races
 - Every read and write should be protected with a lock, even if it “seems safe”
 - Almost every Java/C program with a data race is wrong
- Even without data races, it still may be incorrect
 - Watch for bad interleavings as well!

Consistent Locking

- For each location needing synchronization, have a lock that is always held when reading or writing the location
- The same lock can (and often should) “guard” multiple fields/objects
 - Clearly document what each lock guards!
 - In Java, the ~~lock should usually be the object itself~~ (i.e. “this”)
- Have a mapping between memory locations and lock objects and stick to it!



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

Similar But Separate Issue: Critical Section Granularity

- Coarse-grained
 - For every method that needs a lock, put the entire method body in a lock
- Fine-grained
 - Keep the lock only for the sections of code where it's necessary
- Guideline:
 - Try to structure code so that expensive operations (like I/O) can be done outside of your critical section
 - E.g., if you're trying to print all the values in a tree, maybe copy items into an array inside your critical section, then print the array's contents outside.

Atomicity

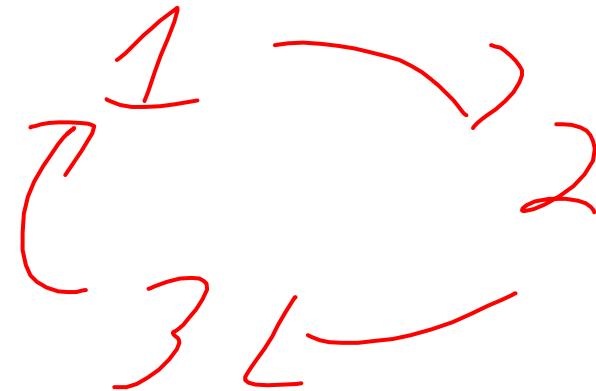
- Atomic: indivisible
- Atomic operation: one that should be thought of as a single step
- Some sequences of operations should behave as if they are one unit
 - Between two operations you may need to avoid exposing an intermediate state
 - Usually ADT operations should be atomic
 - You don't want another 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 as ConcurrentHashMap

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



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);
```

```
    }
```

```
}
```

The Deadlock

Expected Behavior:

Thread 2 items from a stack are popped in LIFO 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

The Deadlock

Expected Behavior:

Thread 2 items from a stack are popped in LIFO 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 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

- Deadlocks occur when there are multiple locks necessary to complete a task and different threads may obtain them in a different order
- Option 1:
 - Have a coarser lock granularity
 - E.g. one lock for ALL bank accounts
- Option 2:
 - 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:
 - 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 acquire

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);  
                }  
            }  
        }  
    }  
}
```