CSE 332 Winter 2026
Lecture 19: Concurrency

Nathan Brunelle

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)

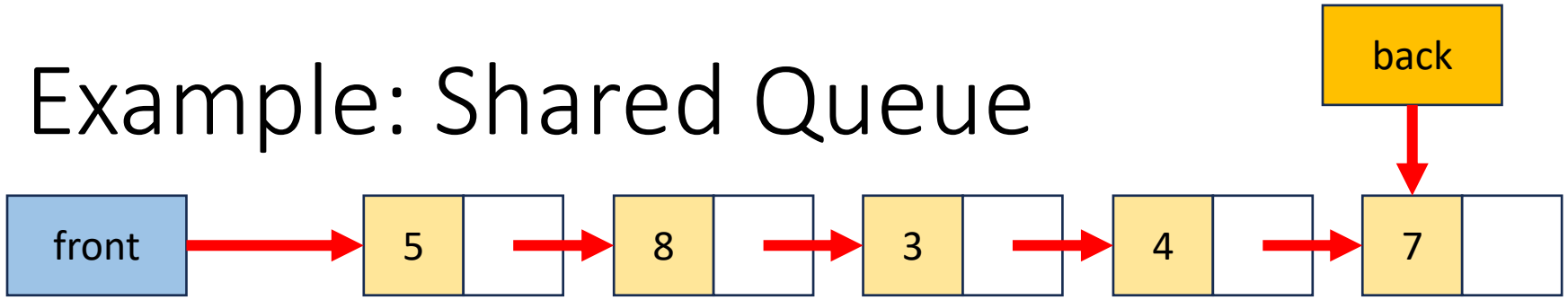
Example: Shared Queue

```
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?

Example: Shared Queue



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;
        }
}
```

Empty Shared Queue

front

back

Suppose the Queue is empty, and the threads execute in this order. Assume Thread 1 enqueues 1, and 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 – first two lines

Thread 1 has decided the queue is empty

front

back

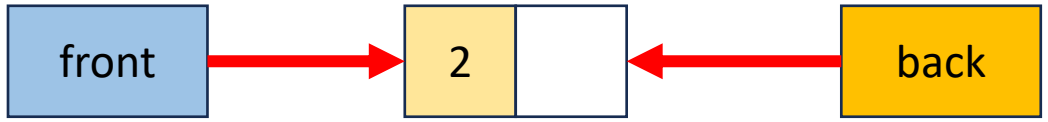
Thread 1

```
~~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 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 trident**
4. Erase the contents of the box
5. Write your name in the box,
   pausing 2 seconds between letters
6. **Put down the trident**
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.
}
```

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);

A “Bad” 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);

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.
}
```

A still “Bad” Interleaving

- Assume the initial balance is 150

Thread 1:

```
withdraw(100);
```

Thread 2:

```
withdraw(75);
```

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

- 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?

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

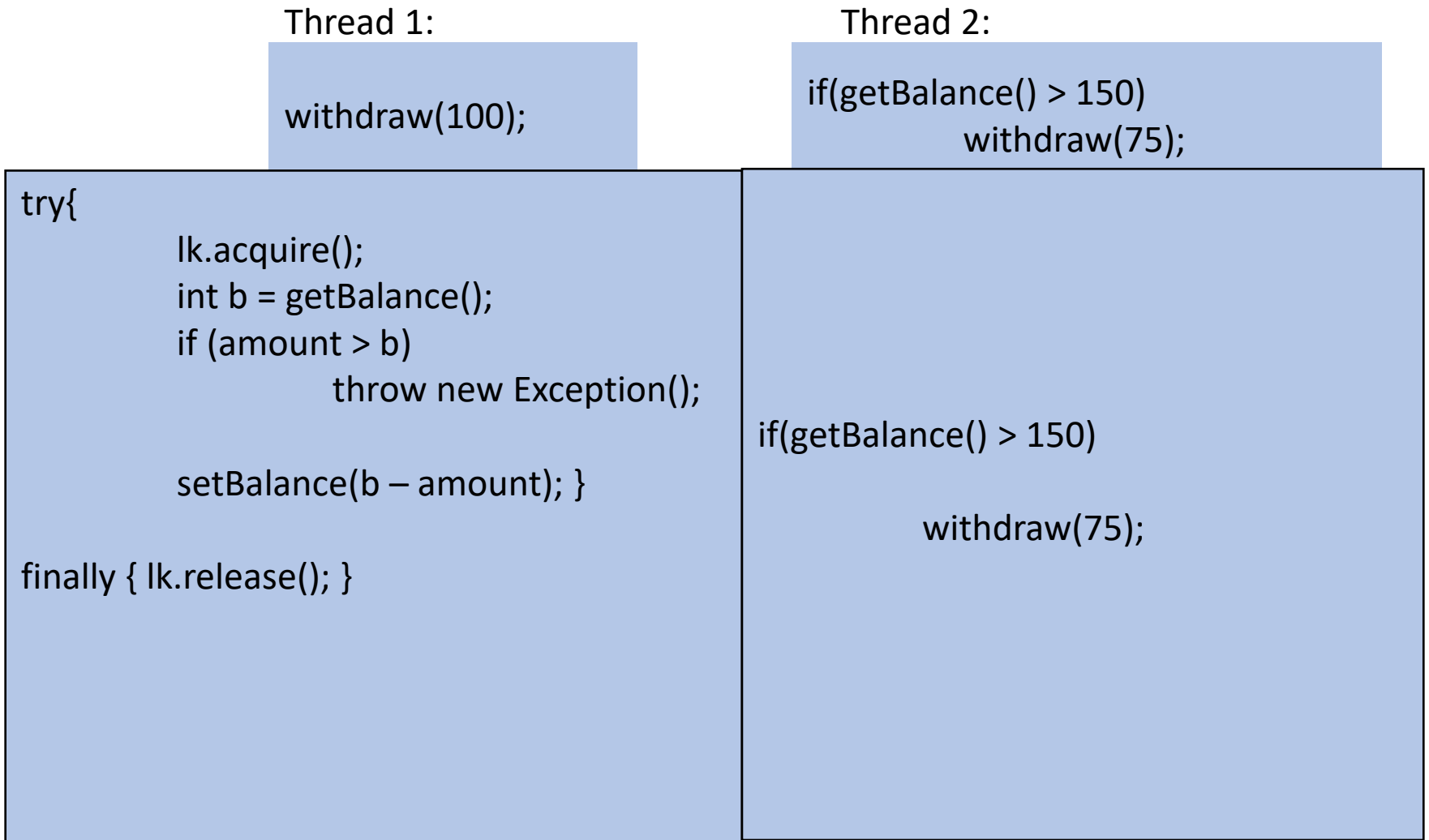
```
class BankAccount {
        private int balance = 0;
        private Lock lck = new Lock();
        int getBalance() { return balance; }
        void setBalance(int x) { balance = x; }
        void withdraw(int amount) {
                try{
                        lk.acquire();
                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

A still “Bad” Interleaving

- Assume the initial balance is 150



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)
}
```

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
  - E.g.: if two threads are withdrawing, different schedules could cause different threads to see the WithdrawTooLargeException
  - The trident did not prevent a race condition, which name is the last one in the box may differ
- 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
  - E.g.: Two threads insert the same into a hash table. The second thread in the schedule will overwrite the insert from the first.
  - 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
  - Usually it looks like exposing a "bad" intermediate state
  - E.g.: Two threads insert into a hash table. We compute the index for each key, then one thread resizes the table, now the other index might be incorrect.
  - 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 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?

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?

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(index);
                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();
 	
push(ans);
return ans;
	push(x);
	
	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();
push(ans);
return ans;
	push(x);
	push(y);
	System.out.println(pop());
	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();
 	
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;
        }
}
```

Fixed!

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?

```
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];
        }
}
```

No! Now it has a data race!

Push/pop will be changing the variable index!