



CSE 332 Data Abstractions:
**Data Races and Memory,
Reordering, Deadlock,
Readers/Writer Locks, and
Condition Variables (oh my!)**

Kate Deibel
Summer 2012

ominous music

THE FINAL EXAM

The Final

It is next Wednesday, August 15

It will take up the entire class period

Is it comprehensive? Yes and No

- Will primarily call upon only what we covered since the midterm (starting at sorting up through next Monday's lecture on minimum spanning trees)
- Still, you will need to understand algorithmic analysis, big-Oh, and best/worst-case for any data structures we have discussed
- You will NOT be doing tree or heap manipulations but you may (i.e., will) do some graph algorithms

Specific Topics

Although the final is by no means finalized, knowing the following would be good:

- How to do Big-Oh (yes, again!)
- Best and worst case for all data structures and algorithms we covered
- Sorting algorithm properties (in-place, stable)
- Graph representations
- Topological sorting
- Dijkstra's shortest-path algorithm
- Parallel Maps and Reductions
- Parallel Prefix, Pack, and Sorting
- ForkJoin Library code
- Key ideas / high-level notions of concurrency

Book, Calculator, and Notes

The exam is closed book

You can bring a calculator if you want

You can bring a limited set of notes:

- One 3x5 index card (both sides)
- Must be handwritten (no typing!)
- You must turn in the card with your exam

Some horses like wet tracks or dry tracks or muddy tracks...

MORE ON RACE CONDITIONS

Races

A **race condition** occurs when the computation result depends on scheduling (how threads are interleaved on ≥ 1 processors)

- Only occurs if T1 and T2 are scheduled in a particular way
- As programmers, we cannot control the scheduling of threads
- Program correctness must be independent of scheduling

Race conditions are bugs that exist only due to concurrency

- No interleaved scheduling with 1 thread

Typically, the problem is some *intermediate state* that "messes up" a concurrent thread that "sees" that state

We will distinguish between **data races** and **bad interleavings**, both of which are types of race condition bugs

Data Races

A **data race** is a type of *race condition* that can happen in two ways:

- Two threads **potentially** write a variable at the same time
- One thread **potentially** write a variable while another reads

Not a race: simultaneous reads provide no errors

Potentially is important

- We claim that code itself has a data race independent of any particular actual execution

Data races are bad, but they are not the only form of race conditions

- We can have a race, and bad behavior, without any data race

Stack Example

```
class Stack<E> {
    private E[] array = (E[])new Object[SIZE];
    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--];
    }
}
```

A Race Condition: But Not a Data Race

```
class Stack<E> {  
    ...  
    synchronized boolean isEmpty() {...}  
    synchronized void push(E val) {...}  
    synchronized E pop(E val) {...}  
  
    E peek() {  
        E ans = pop();  
        push(ans);  
        return ans;  
    }  
}
```

In a sequential world, this code is of iffy, ugly, and questionable *style*, but *correct*

The "algorithm" is the only way to write a **peek** helper method if this interface is all you have to work with

Note that peek() throws the StackEmpty exception via its call to pop()

peek in a Concurrent Context

`peek` has no *overall* effect on the shared data

- It is a "reader" not a "writer"
- State should be the same after it executes as before

This implementation creates an inconsistent *intermediate state*

- Calls to `push` and `pop` are synchronized, so there are no ***data races*** on the underlying array
- But there is still a ***race condition***
- This intermediate state should not be exposed
 - Leads to several *bad interleavings*

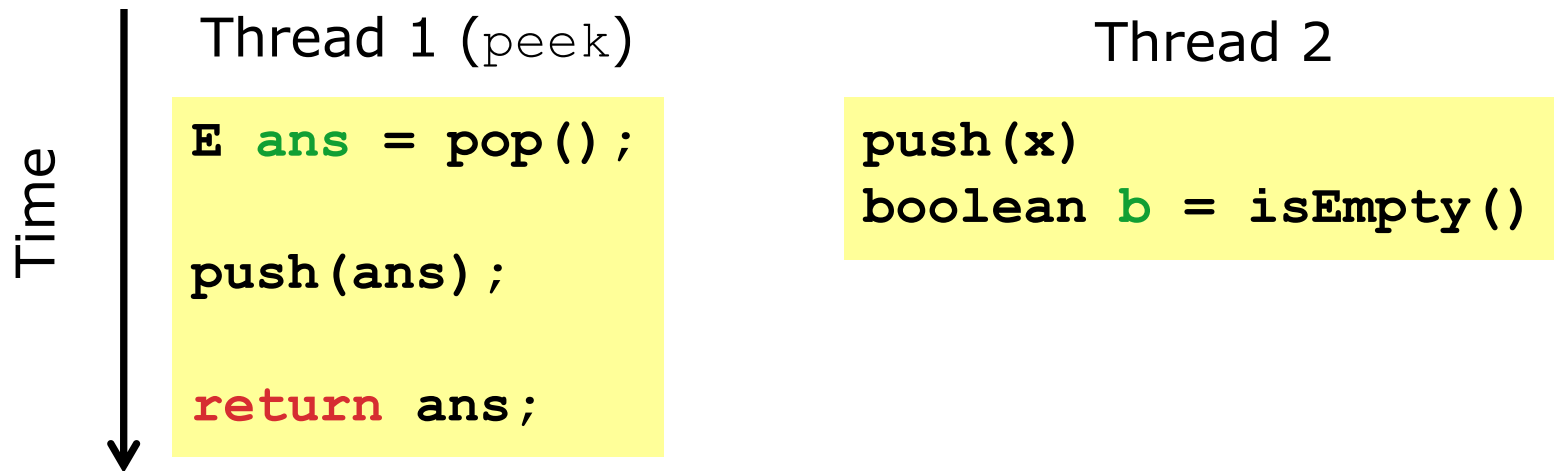
```
E peek () {  
    E ans = pop () ;  
    push (ans) ;  
    return ans ;  
}
```

Example 1: peek and isEmpty

Property we want:

If there has been a **push** (and no **pop**), then **isEmpty** should return **false**

With **peek** as written, property can be violated – how?



Example 1: peek and isEmpty

Property we want:

If there has been a **push** (and no **pop**), then **isEmpty** should return **false**

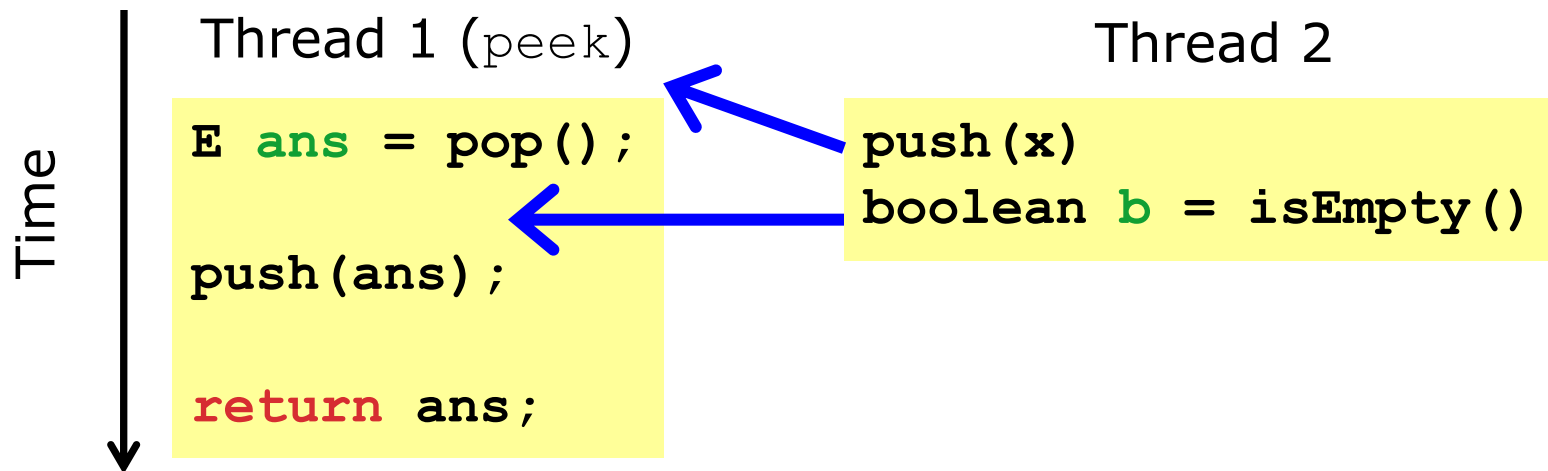
With **peek** as written, property is violated – how?

Race causes error with:

T2: push(x)

T1: pop()

T2: isEmpty()

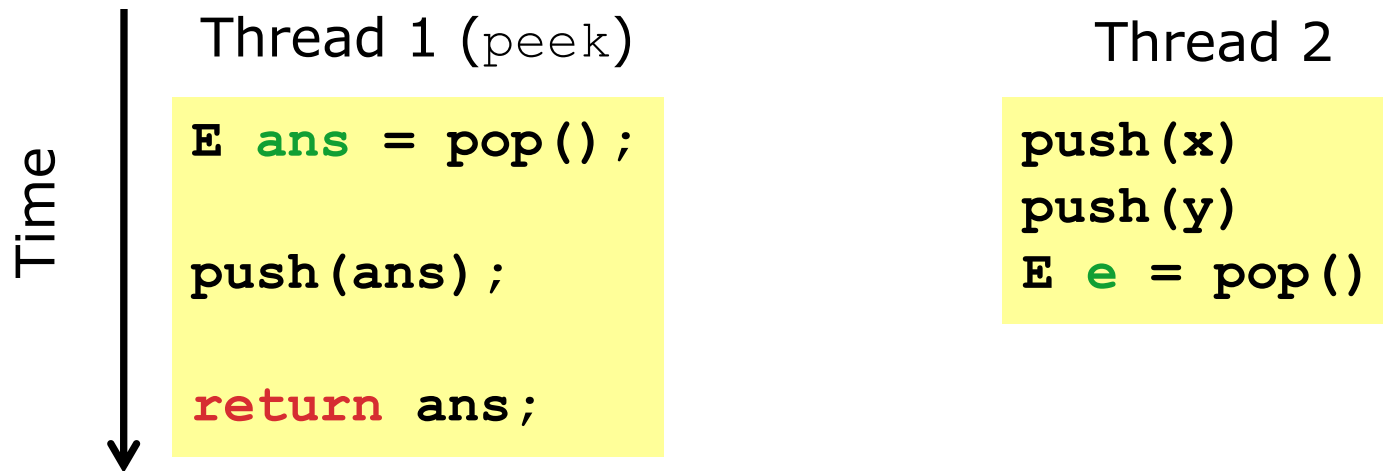


Example 2: peek and push

Property we want:

Values are returned from pop in LIFO order

With **peek** as written, property can be violated – how?



Example 2: peek and push

Property we want:

Values are returned from pop in LIFO order

With `peek` as written, property is violated – how?

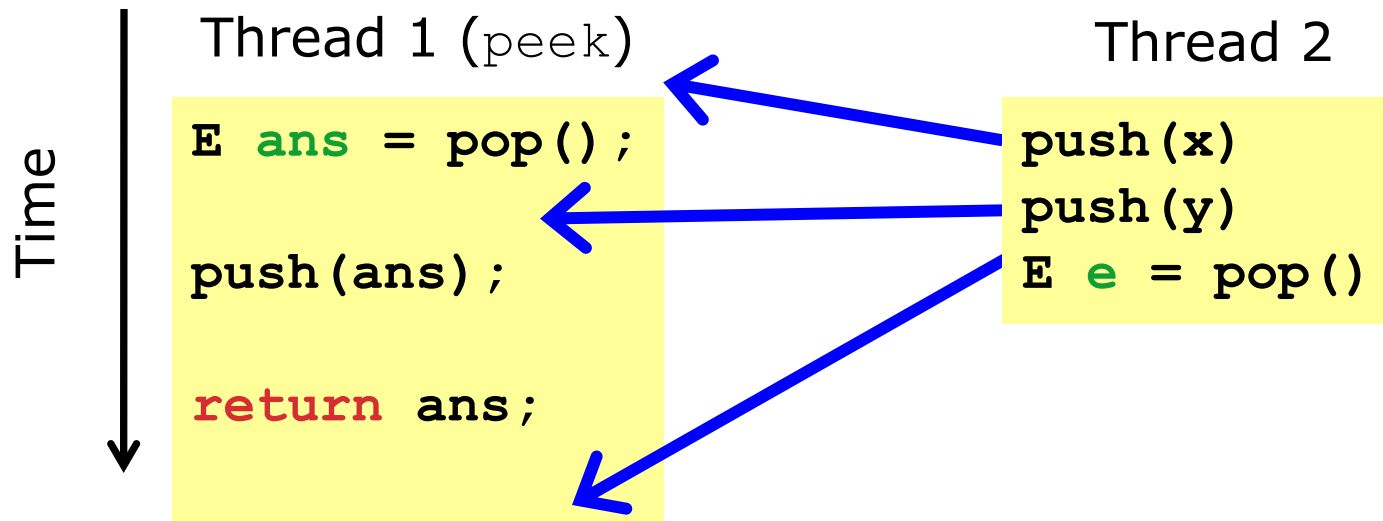
Race causes error with:

T2: push(x)

T1: pop()

T2: push(x)

T1: push(x)

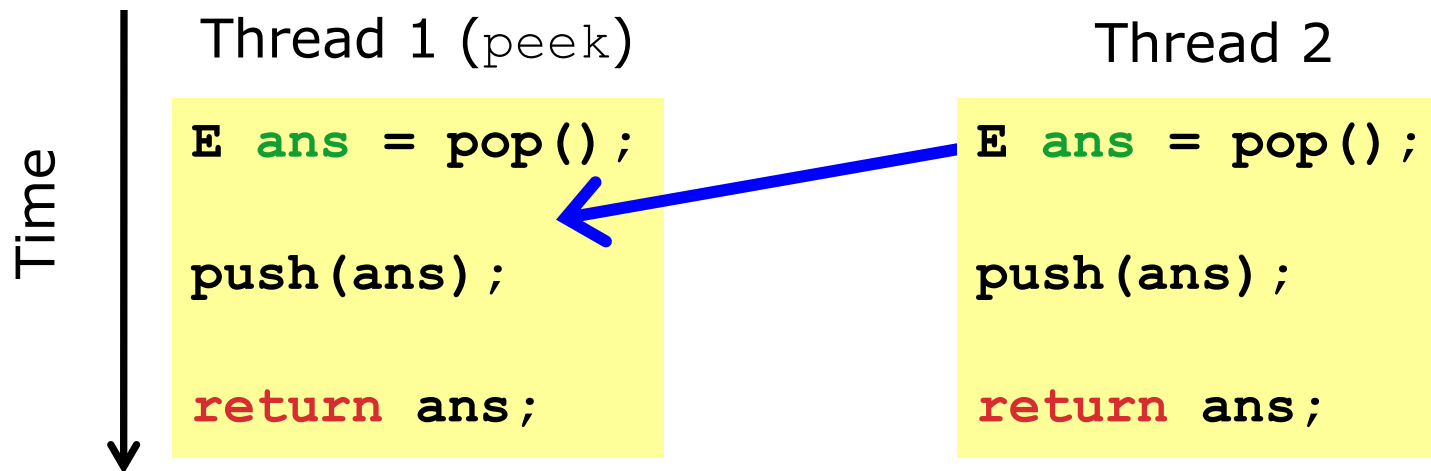


Example 3: peek and peek

Property we want:

peek does not throw an exception unless the stack is empty

With **peek** as written, property can be violated – how?



The Fix

`peek` needs synchronization to disallow interleavings

- The key is to make a *larger critical section*
- This protects the intermediate state of `peek`
- Use re-entrant locks; will allow calls to `push` and `pop`
- Can be done in stack (left) or an external class (right)

```
class Stack<E> {  
    ...  
    synchronized E peek () {  
        E ans = pop ();  
        push (ans) ;  
        return ans ;  
    }  
}
```

```
class C {  
    <E> E myPeek (Stack<E> s) {  
        synchronized (s) {  
            E ans = s.pop ();  
            s.push (ans) ;  
            return ans ;  
        }  
    }  
}
```

An Incorrect "Fix"

So far we have focused on problems created when `peek` performs `writes` that lead to an incorrect intermediate state

A tempting but incorrect perspective

- If an implementation of `peek` does not write anything, then maybe we can skip the synchronization?

Does `not` work due to *data races* with `push` and `pop`

- Same issue applies with other readers, such as `isEmpty`

Another Incorrect Example

```
class Stack<E> {
    private E[] array = (E[])new Object[SIZE];
    int index = -1;
    boolean isEmpty() { // unsynchronized: wrong?!
        return index==-1;
    }
    synchronized void push(E val) {
        array[++index] = val;
    }
    synchronized E pop() {
        return array[index--];
    }
    E peek() { // unsynchronized: wrong!
        return array[index];
    }
}
```

Why Wrong?

It *looks like* `isEmpty` and `peek` can "get away with this" because `push` and `pop` adjust the stack's state using "just one tiny step"

But this code is still *wrong* and depends on language-implementation details you cannot assume

- Even "tiny steps" may require multiple steps in implementation: `array[++index] = val` probably takes at least two steps
- Code has a **data race**, allowing very strange behavior

Do not introduce a data race, even if every interleaving you can think of is correct

Getting It Right

Avoiding race conditions on shared resources is difficult

- Decades of bugs have led to some conventional wisdom and general techniques known to work

We will discuss some key ideas and trade-offs

- More available in the suggested additional readings
- None of this is *specific* to Java or a particular book
 - May be hard to appreciate in beginning
 - Come back to these guidelines over the years
 - Do not try to be fancy

Yale University is the best place to study locks...

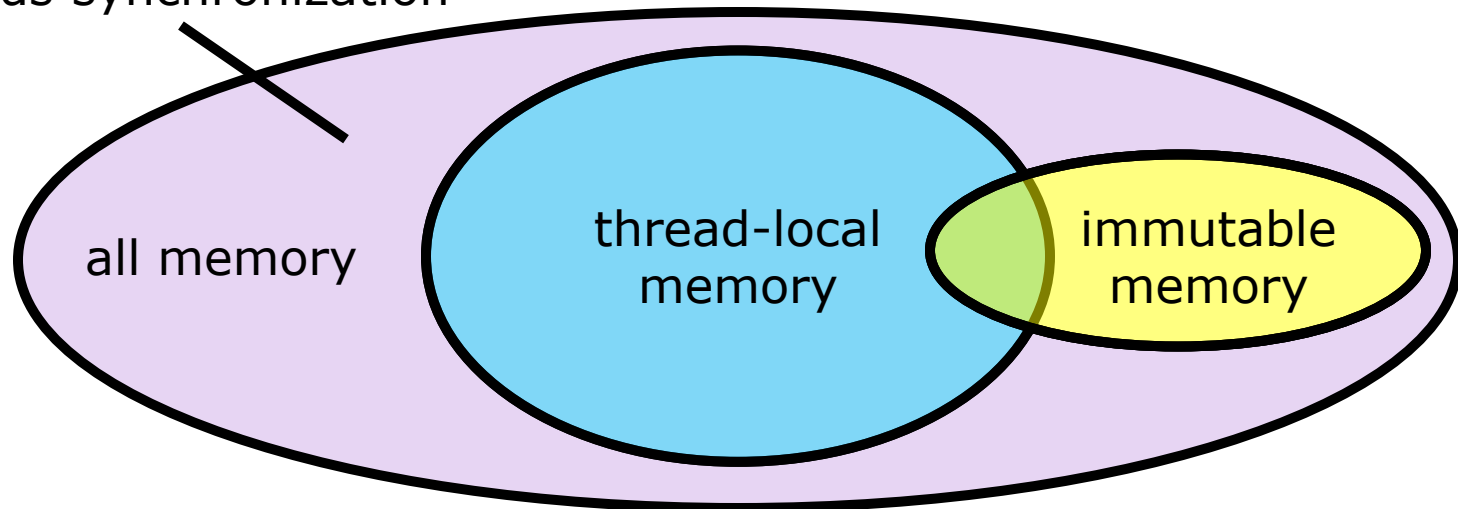
***GOING FURTHER WITH
EXCLUSION AND LOCKING***

Three Choices for Memory

For every **memory location** in your program (e.g., object field), you must obey at least one of the following:

1. **Thread-local**: Do not use the location in >1 thread
2. **Immutable**: Never write to the memory location
3. **Synchronized**: Control access via synchronization

needs synchronization



Thread-Local

Whenever possible, do not share resources!

- Easier for each thread to have its own **thread-local copy** of a resource instead of one with shared updates
- Correct only if threads do not communicate through resource
 - In other words, multiple copies are correct approach
 - Example: **Random** objects
- Note: Because each call-stack is thread-local, never need to synchronize on local variables

In typical concurrent programs, the vast majority of objects should be thread-local and shared-memory usage should be minimized

Immutable

Whenever possible, do not update objects

- Make new objects instead

One of the key tenets of *functional programming* (see CSE 341 Programming Languages)

- Generally helpful to avoid *side-effects*
- Much more helpful in a concurrent setting

If a location is only ever read, never written, no synchronization needed

- Simultaneous reads are *not* races (not a problem!)

In practice, programmers usually over-use mutation so you should do your best to minimize it

Everything Else: Keep it Synchronized

After minimizing the amount of memory that is both (1) thread-shared and (2) mutable, we need to follow guidelines for using locks to keep that data consistent

Guideline #0: No data races

- Never allow two threads to read/write or write/write the same location at the same time

Necessary:

In Java or C, a program with a data race is almost always wrong

But Not Sufficient:

Our `peek` example had no data races

Consistent Locking

Guideline #1: Consistent Locking

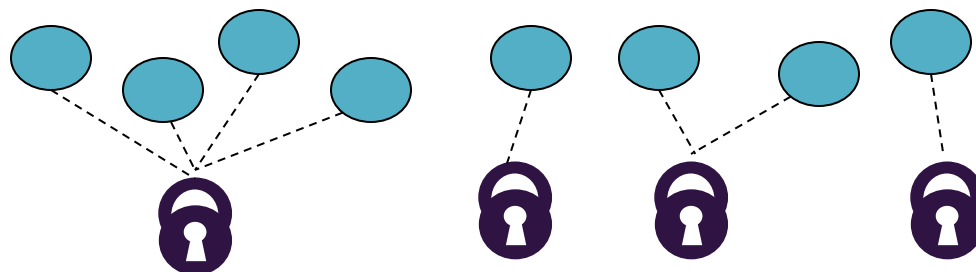
For each location that requires synchronization, we should have a lock that is always held when reading or writing the location

- We say the lock **guards** the location
- The same lock can guard multiple locations (and often should)
- Clearly document the guard for each location
- In Java, the guard is often the object containing the location
 - **this** inside object methods
 - Also common to guard a larger structure with one lock to ensure mutual exclusion on the structure

Consistent Locking

The mapping from locations to guarding locks is *conceptual*, and must be enforced by you as the programmer

- It partitions the shared-&-mutable locations into "which lock"



Consistent locking is:

Not Sufficient:

It prevents all data races, but still allows bad interleavings

- Our peek example used consistent locking, but had exposed intermediate states and bad interleavings

Not Necessary:

- Can dynamically change the locking protocol

Beyond Consistent Locking

Consistent locking is an excellent guideline

- A "default assumption" about program design
- You will save yourself many a headache using this guideline

But it is not required for correctness:

Different *program phases* can use different locking techniques

- Provided all threads coordinate moving to the next phase

Example from Project 3 Version 5:

- A shared grid being updated, so use a lock for each entry
- But after the grid is filled out, all threads except 1 terminate thus making synchronization no longer necessary (i.e., now only thread local)
- And later the grid is only read in response to queries thereby making synchronization doubly unnecessary (i.e., immutable)

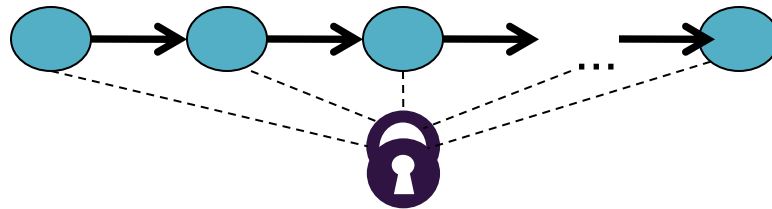
Whole-grain locks are better than overly processed locks...

LOCK GRANULARITY

Lock Granularity

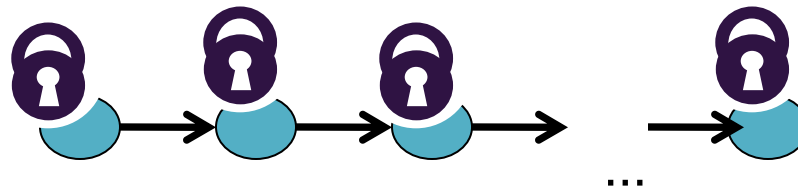
Coarse-Grained: Fewer locks (more objects per lock)

- Example: One lock for entire data structure (e.g., array)
- Example: One lock for all bank accounts



Fine-Grained: More locks (fewer objects per lock)

- Example: One lock per data element (e.g., array index)
- Example: One lock per bank account



"Coarse-grained vs. fine-grained" is really a continuum

Trade-Offs

Coarse-grained advantages

- Simpler to implement
- Faster/easier to implement operations that access multiple locations (because all guarded by the same lock)
- Easier to implement modifications of data-structure shape

Fine-grained advantages

- More simultaneous access (improves performance when coarse-grained would lead to unnecessary blocking)

Guideline #2: Lock Granularity

Start with coarse-grained (simpler), move to fine-grained (performance) only if *contention* on coarse locks is an issue. Alas, often leads to bugs.

Example: Separate Chaining Hashtable

Coarse-grained: One lock for entire hashtable

Fine-grained: One lock for each bucket

Which supports more concurrency for **insert** and **lookup**?

Fine-grained; allows simultaneous access to different buckets

Which makes implementing **resize** easier?

Coarse-grained; just grab one lock and proceed

Maintaining a **numElements** field will destroy the potential benefits of using separate locks for each bucket, why?

Updating each insert without a coarse lock would be a data race

Critical-Section Granularity

A second, orthogonal granularity issue is the size of critical-sections

- How much work should we do while holding lock(s)

If critical sections run for too long:

- Performance loss as other threads are blocked

If critical sections are too short:

- Bugs likely as you broke up something where other threads shouldn't be able to see intermediate state

Guideline #3: Granularity

Do not do expensive computations or I/O in critical sections, but also do not introduce race conditions

Example: Critical-Section Granularity

Suppose we want to change the value for a key in a hashtable without removing it from the table

- Assume `lock` guards the whole table

Papa Bear's critical section was too long

Table is locked during the expensive call

```
synchronized(lock) {  
    v1 = table.lookup(k);  
    v2 = expensive(v1);  
    table.remove(k);  
    table.insert(k, v2);  
}
```

Example: Critical-Section Granularity

Suppose we want to change the value for a key in a hashtable without removing it from the table

- Assume `lock` guards the whole table

Mama Bear's critical section was too short

If another thread updated the entry, we will lose the intervening update

```
synchronized(lock) {  
    v1 = table.lookup(k);  
}  
v2 = expensive(v1);  
synchronized(lock) {  
    table.remove(k);  
    table.insert(k, v2);  
}
```

Example: Critical-Section Granularity

Suppose we want to change the value for a key in a hashtable without removing it from the table

- Assume `lock` guards the whole table

Baby Bear's critical section was just right

if another update occurred, we will try our update again

```
done = false;
while (!done) {
    synchronized(lock) {
        v1 = table.lookup(k);
    }
    v2 = expensive(v1);
    synchronized(lock) {
        if (table.lookup(k) == v1) {
            done = true;
            table.remove(k);
            table.insert(k, v2);
        }
    }
}
```

Atomicity

An operation is *atomic* if no other thread can see it partly executed

- Atomic as in "appears indivisible"
- We typically want ADT operations atomic, even to other threads running operations on the same ADT

Guideline #4: Atomicity

- Think in terms of what operations need to be *atomic*
- Make critical sections just long enough to preserve atomicity
- *Then* design locking protocol to implement critical sections

In other words:

Think about atomicity first and locks second

Do Not Roll Your Own

In real life, you rarely write your own data structures

- Excellent implementations provided in standard libraries
- Point of CSE 332 is to understand the key trade-offs, abstractions, and analysis of such implementations

Especially true for concurrent data structures

- Far too difficult to provide fine-grained synchronization without race conditions
- Standard **thread-safe** libraries like **ConcurrentHashMap** are written by world experts and been extensively vetted

Guideline #5: Libraries

Use built-in libraries whenever they meet your needs

Motivating Memory-Model Issues

Tricky and *surprisingly wrong* unsynchronized concurrent code

```
class C {
  private int x = 0;
  private int y = 0;

  void f() {
    x = 1;
    y = 1;
  }

  void g() {
    int a = y;
    int b = x;
    assert(b >= a);
  }
}
```

First understand why it looks like the assertion cannot fail:

Easy case:

A call to g ends before any call to f starts

Easy case:

At least one call to f completes before call to g starts

If calls to f and g interleave...

Interleavings Are Not Enough

There is no interleaving of \mathbf{f} and \mathbf{g} such that the assertion fails

Proof #1:

Exhaustively consider all possible orderings of access to shared memory (there are 6)

Interleavings Are Not Enough

Proof #2:

Exhaustively consider all possible orderings of access to shared memory (there are 6)

If $!(b \geq a)$, then $a == 1$ and $b == 0$.

But if $a == 1$, then $y = 1$ happened before $a = y$.

Because programs execute in order:

$a = y$ happened before $b = x$

and $x = 1$ happened before $y = 1$

So by transitivity, $b == 1$.

Contradiction.

Thread 1: f

```
x = 1;  
  ↓  
y = 1;
```

Thread 2: g

```
int a = y;  
  ↓  
int b = x;  
  
assert(b >= a);
```

Wrong

However, the code has a *data race*

- Unsynchronized read/write or write/write of the memory same location

If code has data races, you cannot reason about it with interleavings

- This is simply the rules of Java (and C, C++, C#, other languages)
- Otherwise we would slow down all programs just to "help" those with data races, and that would not be a good engineering trade-off
- So the assertion can fail

Why

For performance reasons, the compiler and the hardware will often reorder memory operations

- Take a compiler or computer architecture course to learn more as to why this is good thing

Thread 1: f

```
x = 1;
```


```
y = 1;
```

Thread 2: g

```
int a = y;
```

```
int b = x;
```

```
assert(b >= a);
```



Of course, compilers cannot just reorder anything they want without careful consideration

- Each thread computes things by executing code in order
- Consider: **x=17; y=x;**

The Grand Compromise

The compiler/hardware will NEVER:

- Perform a memory reordering that affects the result of a single-threaded program
- Perform a memory reordering that affects the result of a **data-race-free** multi-threaded program

So: If no interleaving of your program has a data race, then you can *forget about all this reordering nonsense*: the result will be equivalent to some interleaving

The Big Picture:

- Your job is to **avoid data races**
- The compiler/hardware's job is to give illusion of interleaving *if you do your job right*

Fixing Our Example

Naturally, we can use synchronization to avoid data races and then, indeed, the assertion cannot fail

```
class C {
  private int x = 0;
  private int y = 0;
  void f() {
    synchronized(this) { x = 1; }
    synchronized(this) { y = 1; }
  }
  void g() {
    int a, b;
    synchronized(this) { a = y; }
    synchronized(this) { b = x; }
    assert(b >= a);
  }
}
```

A Second Fix: Stay Away from This

Java has **volatile** fields: accesses do not count as data races

- But you cannot read-update-write

```
class C {
    private volatile int x = 0;
    private volatile int y = 0;
    void f() {
        x = 1; y = 1;
    }
    void g() {
        int a = y; int b = x;
        assert(b >= a);
    }
}
```

Implementation Details

- Slower than regular fields but faster than locks
- Really for experts: avoid them; use standard libraries instead
- And why do you need code like this anyway?

Code That is Wrong

Here is a more realistic example of code that is wrong

- No *guarantee* Thread 2 will ever stop (due to data race)
- But honestly it will "likely work in practice"

```
class C {
    boolean stop = false;

    void f() {
        while(!stop) {
            // draw a monster
        }
    }

    void g() {
        stop = didUserQuit();
    }
}
```

Thread 1: f()

Thread 2: g()

Not nearly as silly as Deathlok from Marvel comics...

DEADLOCK

Motivating Deadlock Issues

Consider the following method for transferring money between bank accounts

```
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);
    }
}
```

During call to `a.deposit`, the thread holds two locks

- Let's investigate when this may be a problem

The Deadlock

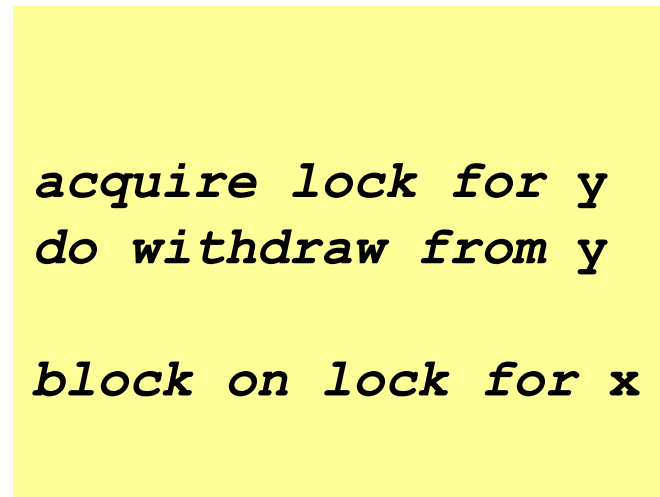
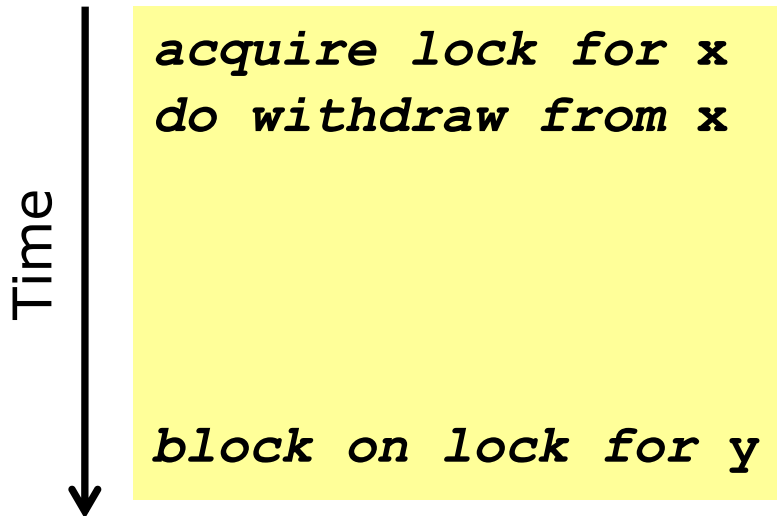
Suppose **x** and **y** are fields holding accounts

Thread 1:

`x.transferTo(1, y)`

Thread 2:

`y.transferTo(1, x)`



The Dining Philosophers

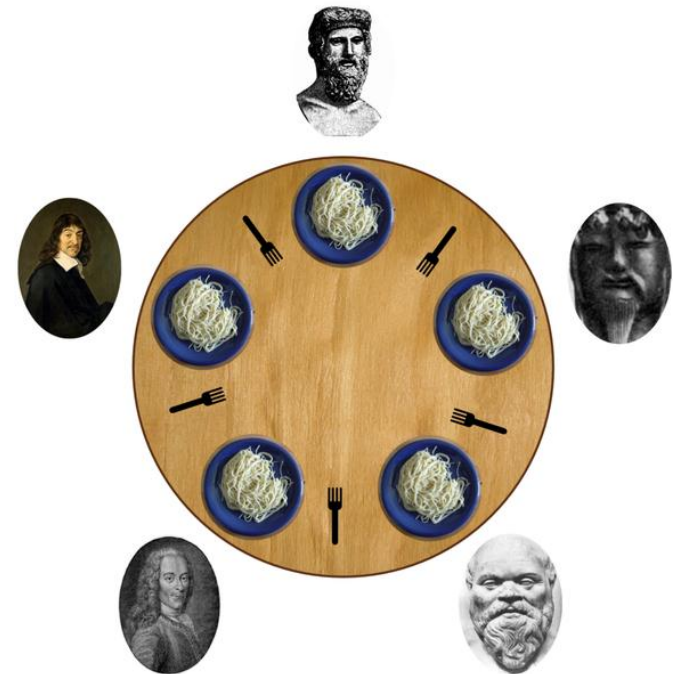
Five philosophers go out to dinner together at an Italian restaurant

They sit at a round table; one fork per plate setting

For etiquette reasons, the philosophers need two forks to eat spaghetti properly

When the spaghetti comes, each philosopher proceeds to grab their right fork, then their left fork

'Locking' for each fork results in a **deadlock**



Deadlock

A deadlock occurs when there are threads $\mathbf{T}_1, \dots, \mathbf{T}_n$ such that:

- For $i=1$ to $n-1$, \mathbf{T}_i is waiting for at least one resource held by \mathbf{T}_{i+1}
- \mathbf{T}_n is waiting for a resource held by \mathbf{T}_1

In other words, there is a *cycle* of waiting

- More formally, a graph of dependencies is cyclic

Deadlock avoidance in programming amounts to techniques to ensure a cycle can never arise

Back to Our Example

Options for deadlock-proof transfer:

1. Make a smaller critical section:
`transferTo` not synchronized
 - Exposes intermediate state after `withdraw` before `deposit`
 - May be okay, but exposes wrong total amount to bank
2. Coarsen lock granularity:
One lock for all accounts allowing transfers between them
 - Works, but sacrifices concurrent deposits/withdrawals
3. Give every bank-account a unique number and always acquire locks in the same order
 - *Entire program* should obey this order to avoid cycles
 - Code acquiring only one lock can ignore the order

Ordering Locks

```
class BankAccount {
    ...
    private int acctNumber; // must be unique
    void transferTo(int amt, BankAccount a) {
        if(this.acctNumber < a.acctNumber)
            synchronized(this) {
                synchronized(a) {
                    this.withdraw(amt);
                    a.deposit(amt);
                }
            }
        else
            synchronized(a) {
                synchronized(this) {
                    this.withdraw(amt);
                    a.deposit(amt);
                }
            }
    }
}
```

StringBuffer Example

From the Java standard library

```
class StringBuffer {
    private int count;
    private char[] value;
    ...
    synchronized append(StringBuffer sb) {
        int len = sb.length();
        if(this.count + len > this.value.length)
            this.expand(...);
        sb.getChars(0, len, this.value, this.count);
    }
    ...
    synchronized getChars(int x, int, y,
                           char[] a, int z) {
        "copy this.value[x..y] into a starting at z"
    }
}
```


Two Problems

Problem #1:

Lock for `sb` not held between calls to `sb.length` and `sb.getChars`

- So `sb` could get longer
- Would cause `append` to throw an `ArrayBoundsException`

Problem #2:

Deadlock potential if two threads try to `append` in opposite directions, identical to the bank-account first example

Not easy to fix both problems without extra copying:

- Do not want unique ids on every `StringBuffer`
- Do not want one lock for all `StringBuffer` objects

Actual Java library:

Fixed neither (left code as is; changed documentation)

- Up to clients to avoid such situations with their own protocols

Perspective

Code like account-transfer and string-buffer append are difficult to deal with for deadlock

Easier case: different types of objects

- Can establish and document a fixed order among types
- Example: "When moving an item from the hashtable to the work queue, never try to acquire the queue lock while holding the hashtable lock"

Easier case: objects are in an acyclic structure

- Can use the data structure to determine a fixed order
- Example: "If holding a tree node's lock, do not acquire other tree nodes' locks unless they are children"

We encourage multiple readers...

***IMPROVING LITERACY:
READER/WRITER LOCKS***

Reading vs. Writing

Recall:

- Multiple concurrent reads of same memory: *Not a problem*
- Multiple concurrent writes of same memory: **Problem**
- Multiple concurrent read & write of same memory: **Problem**

So far:

- If concurrent write/write or read/write might occur, use synchronization to ensure one-thread-at-a-time

But this is unnecessarily conservative:

- Could still allow multiple simultaneous readers!

Example

Consider a hashtable with one coarse-grained lock

- Only one thread can perform operations at a time

But suppose:

- There are many simultaneous `lookup` operations
- And `insert` operations are very rare

Note:

Critically important that `lookup` does not actually mutate shared memory, like a move-to-front list or splay tree operation would

Readers/Writer locks

A new synchronization ADT: the **readers/writer lock**

A lock's states fall into three categories:

- "not held"
- "held for writing" by one thread
- "held for reading" by *one or more* threads

ADT Invariants:

$$0 \leq \text{writers} \leq 1$$

$$0 \leq \text{readers}$$

$$\text{writers} \times \text{readers} = 0$$

Operations:

- **new:** make a new lock, initially "not held"
- **acquire_write:** block if currently "held for reading" or if "held for writing", else make "held for writing"
- **release_write:** make "not held"
- **acquire_read:** block if currently "held for writing", else make/keep "held for reading" and increment *readers count*
- **release_read:** decrement readers count, if 0, make "not held"

Pseudocode Example (not Java)

```
class Hashtable<K,V> {  
    ...  
    // coarse-grained, one lock for table  
    RWLock lk = new RWLock();  
    V lookup(K key) {  
        int bucket = hasher(key);  
        lk.acquire_read();  
        ... read array[bucket] ...  
        lk.release_read();  
    }  
    void insert(K key, V val) {  
        int bucket = hasher(key);  
        lk.acquire_write();  
        ... write array[bucket] ...  
        lk.release_write();  
    }  
}
```

Readers/Writer Lock Details

A readers/writer lock implementation (which is “not our problem”) usually gives *priority* to writers:

- After a writer blocks, no readers *arriving later* will get the lock before the writer
- Otherwise an `insert` could *starve*

Re-entrant (same thread acquires lock multiple times)?

- Mostly an orthogonal issue
- But some libraries support *upgrading* from reader to writer

Why not use readers/writer locks with more fine-grained locking? Like on each bucket?

- Not wrong, but likely not worth it due to low contention

In Java

[Note: Not needed in your project/homework]

Java's **synchronized** statement does not support readers/writer

Instead, the Java library has

`java.util.concurrent.locks.ReentrantReadWriteLock`

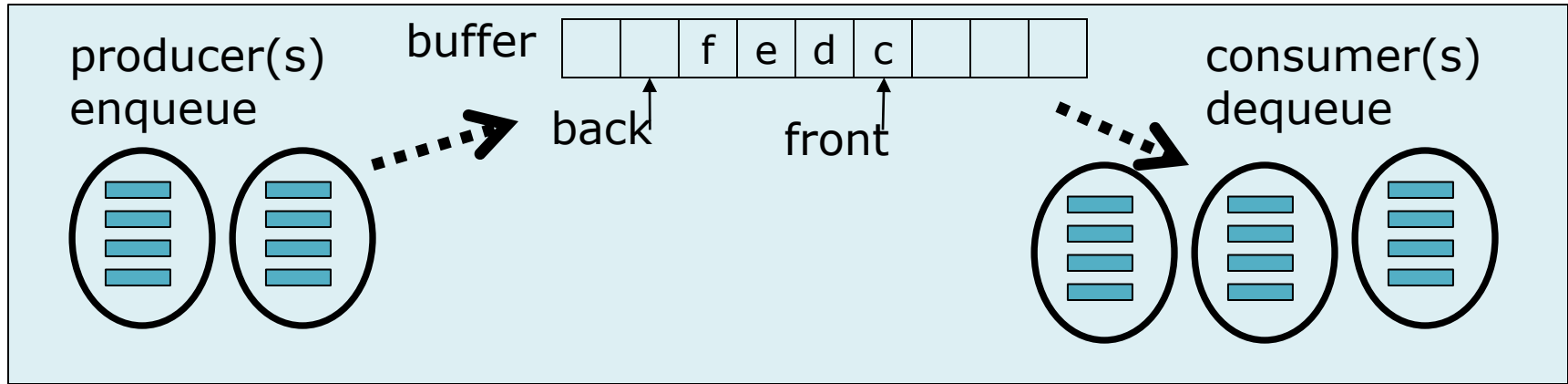
Details:

- Implementation is different
- methods `readLock` and `writeLock` return objects that themselves have `lock` and `unlock` methods
- Does *not* have writer priority or reader-to-writer upgrading
- If you want to use them, be sure to read the documentation

The natural successor to shampoo variables

CONDITION VARIABLES

Motivating Condition Variables



To motivate condition variables, consider the canonical example of a **bounded buffer** for sharing work among threads

Bounded buffer: A queue with a fixed size

- Only slightly simpler if unbounded, core need still arises

For sharing work – think an assembly line:

- Producer thread(s) do some work and enqueue result objects
- Consumer thread(s) dequeue objects and do next stage
- Must synchronize access to the queue

First Attempt

```
class Buffer<E> {
    E[] array = (E[])new Object[SIZE];
    ... // front, back fields, isEmpty, isFull methods
    synchronized void enqueue(E elt) {
        if(isFull())
            ???
        else
            ... add to array and adjust back ...
    }
    synchronized E dequeue()
        if(isEmpty())
            ???
        else
            ... take from array and adjust front ...
    }
}
```

Waiting

enqueue to a full buffer should *not* raise an exception but should **wait** until there is room

dequeue from an empty buffer should *not* raise an exception but should **wait** until there is data

One bad approach is to *spin-wait* (wasted work and keep grabbing lock)

```
void enqueue(E elt) {
    while(true) {
        synchronized(this) {
            if(isFull()) continue;
            ... add to array and adjust back ...
            return;
        }
    }
}
// dequeue similar
```

What we Want

Better would be for a thread to simply *wait* until it can proceed

- It should not spin/process continuously
- Instead, it should be *notified* when it should try again
- In the meantime, let other threads run

Like locks, not something you can implement on your own

- Language or library gives it to you, typically implemented with operating-system support

An ADT that supports this: **condition variable**

- Informs waiter(s) when the *condition* that causes it/them to wait has *varied*

Terminology not completely standard; will mostly stick with Java

Java Approach: *Not Quite Right*

```
class Buffer<E> {  
    ...  
    synchronized void enqueue(E elt) {  
        if(isFull())  
            this.wait(); // releases lock and waits  
        add to array and adjust back  
        if(buffer was empty)  
            this.notify(); // wake somebody up  
    }  
    synchronized E dequeue() {  
        if(isEmpty())  
            this.wait(); // releases lock and waits  
        take from array and adjust front  
        if(buffer was full)  
            this.notify(); // wake somebody up  
    }  
}
```

Key Ideas You Should Know

Java is a bit weird:

- Every object “is” a condition variable (also a lock)
- Other languages/libraries often make them separate

wait:

- “Register” running thread as interested in being woken up
- Then atomically: release the lock and block
- When execution resumes, *thread again holds the lock*

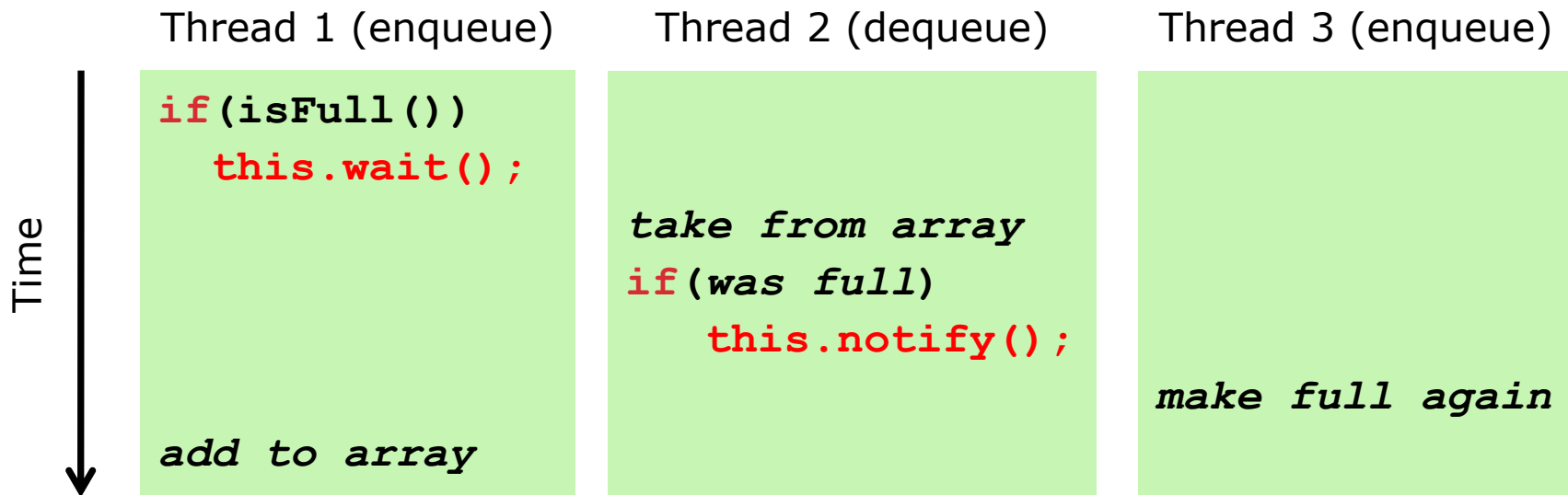
notify:

- Pick one waiting thread and wake it up
- No guarantee woken up thread runs next, just that it is no longer blocked on the *condition*, now waiting for the *lock*
- If no thread is waiting, then do nothing

The Bug in the Earlier Code

```
synchronized void enqueue(E elt) {  
    if(isFull())  
        this.wait();  
    add to array and adjust back  
    ...  
}
```

Between the time a thread is notified and it re-acquires the lock, the condition can become false again!



Bug Fix

```
synchronized void enqueue(E elt) {  
    while(isFull())  
        this.wait();  
    ...  
}  
synchronized E dequeue() {  
    while(isEmpty())  
        this.wait();  
    ...  
}
```

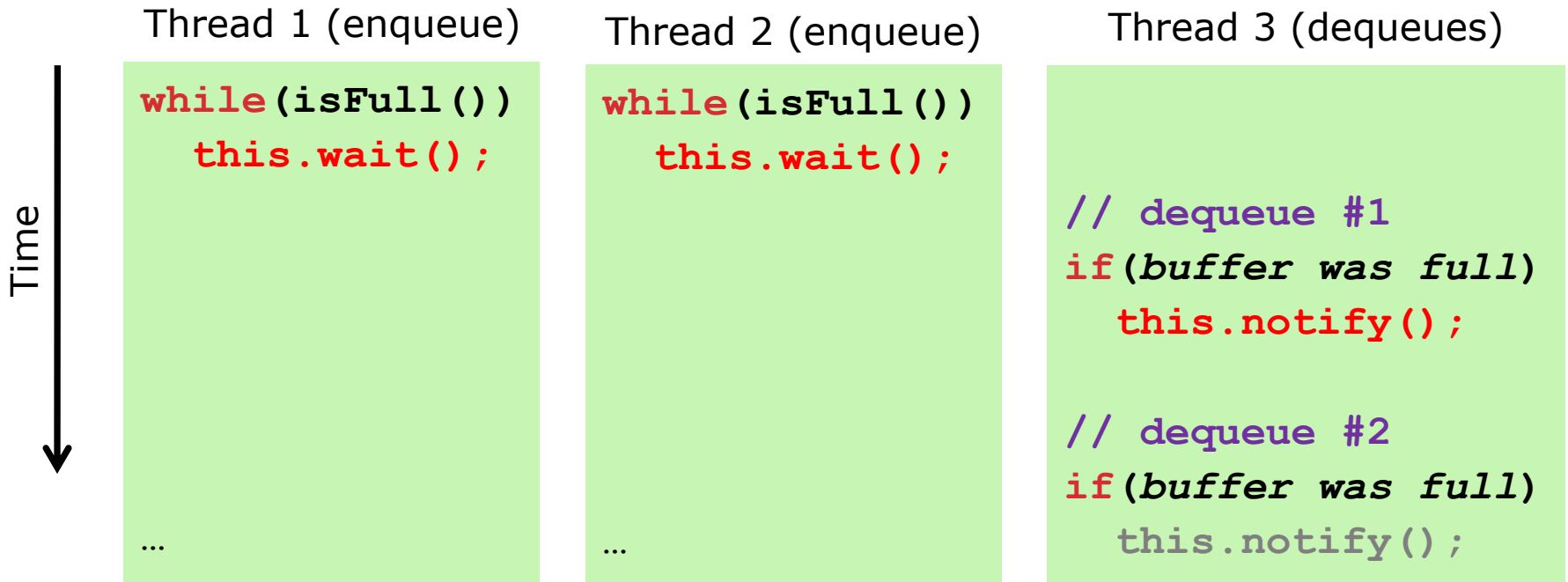
Guideline: *Always* re-check the condition after re-gaining the lock

For obscure (!!) reasons, Java is technically allowed to notify a thread *spuriously* (i.e., for no reason and without actually making a call to `notify`)

Another Bug

If multiple threads are waiting, we wake up only one

- Sure only one can do work *now*, but we cannot forget the others!



Bug Fix

```
synchronized void enqueue(E elt) {  
    ...  
    if(buffer was empty)  
        this.notifyAll(); // wake everybody up  
}  
synchronized E dequeue() {  
    ...  
    if(buffer was full)  
        this.notifyAll(); // wake everybody up  
}
```

`notifyAll` wakes up all current waiters on the condition variable

Guideline: If in any doubt, use `notifyAll`

- Wasteful waking is much better than never waking up (because you already need to re-check condition)

So why does `notify` exist? Well, it is faster when correct...

Alternate Approach

An alternative is to call `notify` (not `notifyAll`) on every `enqueue` / `dequeue`, not just when the buffer was empty / full

- Easy: just remove the `if` statement

Alas, makes our code subtly **wrong** since it is technically possible that both an `enqueue` and a `dequeue` are both waiting.

Works fine if buffer is unbounded (linked list) because then only dequeuers will ever wait

Alternate Approach Fixed

An alternate approach works if the enqueueers and dequeuers wait on *different* condition variables

- But for mutual exclusion both condition variables must be associated with the same lock

Java's "everything is a lock / condition variable" does not support this: each condition variable is associated with itself

Instead, Java has classes in `java.util.concurrent.locks` for when you want multiple conditions with one lock

- `class ReentrantLock` has a method `newCondition` that returns a new `Condition` object associate with the lock
- See the documentation if curious

Final Comments on Condition Variable

`notify/notifyAll` often called `signal/broadcast` or `pulse/pulseAll`

Condition variables are subtle and harder to use than locks

But when you need them, you need them

- Spinning and other workarounds do not work well

Fortunately, like most things you see in a data-structures course, the common use-cases are provided in libraries written by experts and have been thoroughly vetted

- Example: `java.util.concurrent.ArrayBlockingQueue<E>`
All condition variables hidden; just call `put` and `take`

Concurrency Summary

Access to shared resources introduces new kinds of bugs

- Data races
- Critical sections too small
- Critical sections use wrong locks
- Deadlocks

Requires synchronization

- Locks for mutual exclusion (common, various flavors)
- Condition variables for signaling others (less common)

Guidelines for correct use help avoid common pitfalls

Not always clear shared-memory is worth the pain

- But other models not a panacea (e.g., message passing)