CSE 332: Data Structures & Parallelism
Lecture 17: Race Conditions & Deadlock

Arthur Liu

Summer 2022

The Concurrency Part of this class

- Introduction of Parallelism Ideas
  - Java’s Thread
  - ForkJoin Library
- General Parallelism Algorithms
  - Reduce, Map
  - Analysis (span, work)
- Clever Parallelism Ideas
  - Parallel Prefix
  - Parallel Sorts
- Synchronization
  - The need for locks (Concurrency)
- Other Synchronization Issues
  - Race Conditions: Data Races & Bad Interleavings, Deadlock

Race Conditions

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

- If T1 and T2 happened to get scheduled in a certain way, things go wrong
- We, as programmers, cannot control scheduling of threads;
- Thus we need to write programs that work ***independent of scheduling***

Race conditions are bugs that exist only due to concurrency

- No interleaved scheduling problems with only 1 thread!

Typically, problem is that some *intermediate state* can be seen by another thread; screws up other thread

Race Conditions:
Data Races vs. Bad Interleavings

We will make a big distinction between:

R. C.

data races and *bad interleavings*

Data Races

A *data race* is a specific type of *race condition* where there is the ***possibility*** for either:

1. Two different threads to write a variable at the same time W/W
2. One thread writes a variable while another thread reads the variable W/R

Stack Example (pseudocode)

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

Stack Example (pseudocode)

```
class Stack<E> {
  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--];
  }
}
```

Example of a Race Condition, but not a Data Race

R/W

```
class Stack<E> {
  … // state used by isEmpty, push, pop
  synchronized boolean isEmpty() { … }
  synchronized void push(E val) { … }
  synchronized E pop() {
    if(isEmpty())
      throw new StackEmptyException();
    …
  }
  E peek() { // this is wrong
  ↑  E ans = pop();
  ↓  push(ans);
     return ans;
  }
}
```

Problems with `peek`

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

- **`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
- But the way it is implemented creates an inconsistent *intermediate state*
  - Calls to **`push`** and **`pop`** are synchronized
    - So there are no *data races* on the underlying array/index
  - There is still a *race condition* though
- This intermediate state should not be exposed
  - Leads to several *bad interleavings*

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?

Thread 1 (`peek`)

```
E ans = pop();

push(ans);

return ans;
```

Thread 2 (`push + isEmpty`)

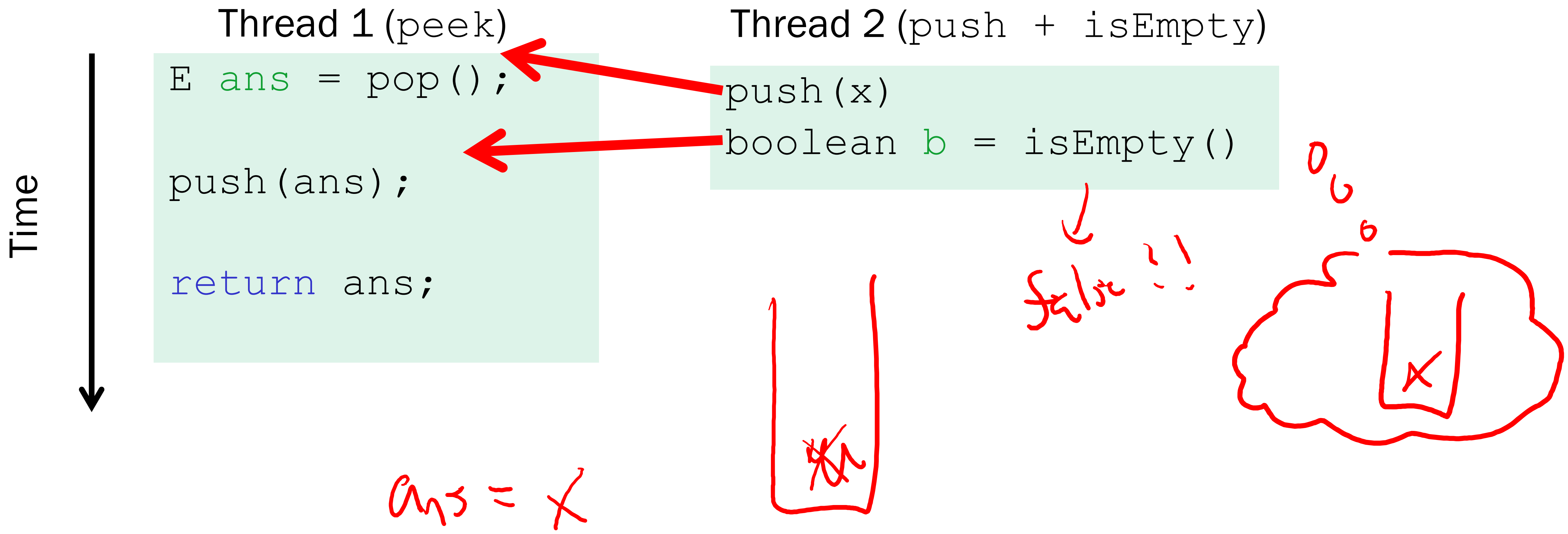
```
push(x)
boolean b = isEmpty()
```

Time

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 2: peek and push

- **Property we want:** Values are returned from **`pop`** in LIFO order
- With **`peek`** as written, property can be violated – how?

Time ↓

Thread 1 (`peek`)

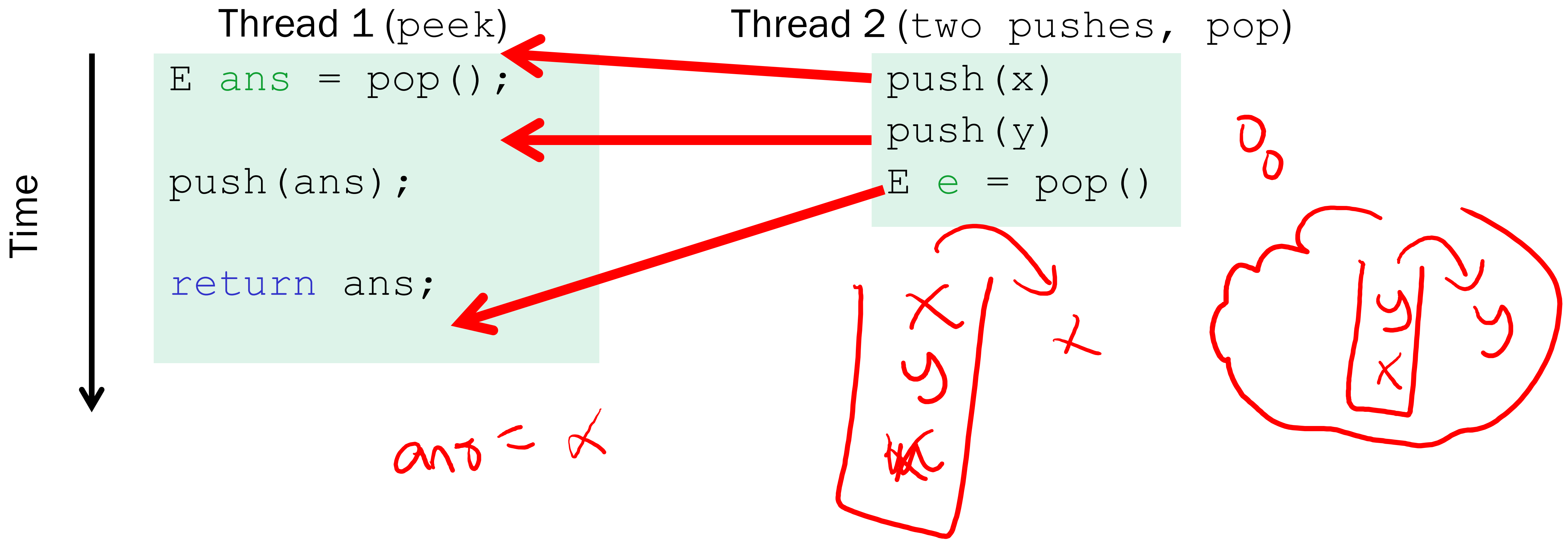
```
E ans = pop();

push(ans);

return ans;
```

Thread 2 (`two pushes, pop`)

```
push(x)
push(y)
E e = pop()
```

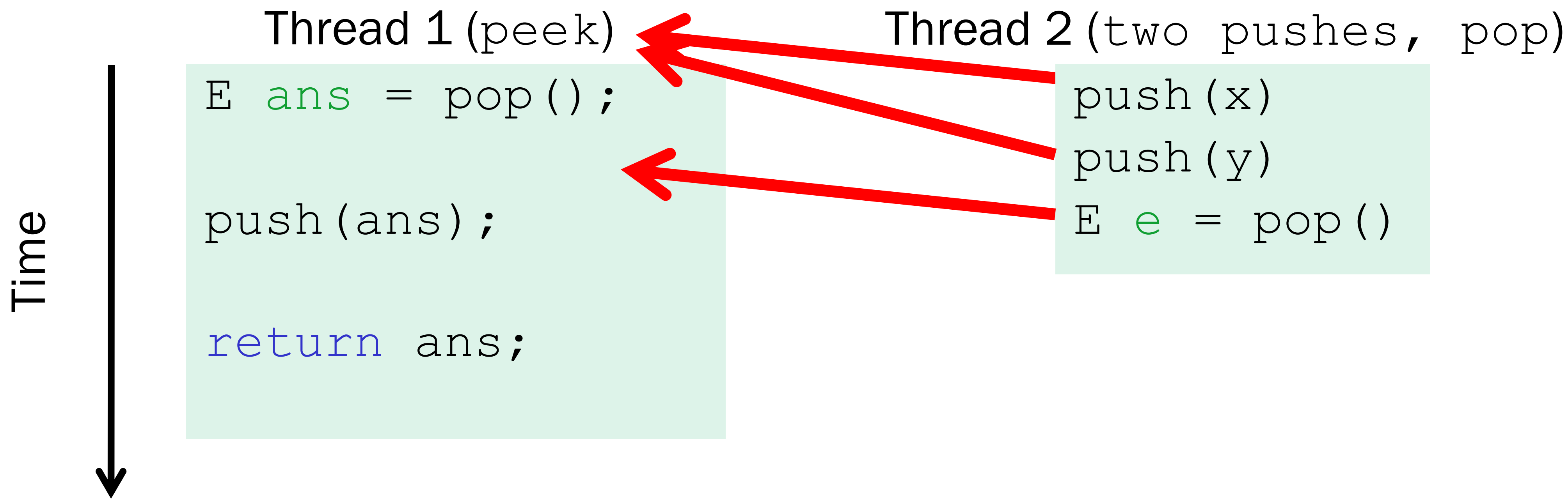
Example 2: peek and push

- **Property we want:** Values are returned from **`pop`** in LIFO order
- With **`peek`** as written, property can be violated – how?



Example 2.5: peek and pop

- **Property we want:** Values are returned from **`pop`** in LIFO order
- With **`peek`** as written, property can be violated – how?



Example 4: peek and peek

- **Property we want: `peek`** doesn't throw an exception unless stack is empty
- With **`peek`** as written, property can be violated – how?

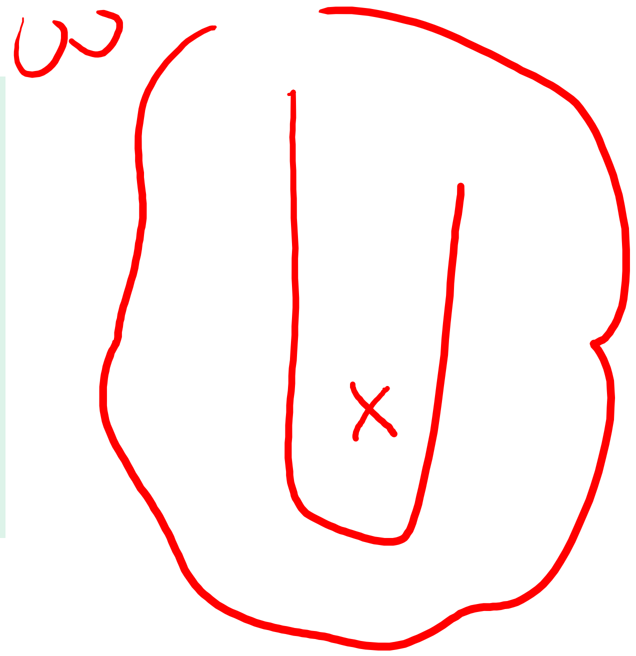
Time ↓

Thread 1 (`peek`)

```
E ans = pop();

push(ans);

return ans;
```

Thread 2 (`peek`)

```
E ans = pop();

push(ans);

return ans;
```



Example 4: peek and peek

- Property we want: **`peek`** doesn't throw an exception unless stack is empty
- With **`peek`** as written, property can be violated – how?

Time

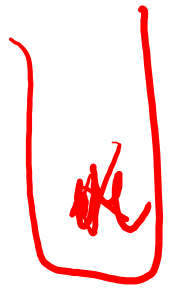
Thread 1 (`peek`)

```
E ans = pop();

push(ans);

return ans;
```

Thread 2 (`peek`)

```
E ans = pop();

push(ans);

return ans;
```

ans = x



ans =

The fix

- In short, **peek** needs synchronization to disallow interleavings
  - The key is to make a *larger critical section*
    - That intermediate state of `peek` needs to be protected
  - Use re-entrant locks; will allow calls to **push** and **pop**
  - Code on right is example of a `peek` external to the `Stack` class

```
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;
    }
  }
}
```

How you might have written peek

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

The wrong "fix"

- **Focus previously:** problems from (a weird) **`peek`** doing writes that lead to an incorrect intermediate state (bad interleavings)

- **Tempting but wrong:** If an implementation of **`peek`** (or **`isEmpty`**) does not write anything, then maybe we can skip the synchronization?

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

Why wrong?

- It *looks like* **`isEmpty`** and **`peek`** can “get away with this” since **`push`** and **`pop`** adjust the state “in 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 the implementation: **`array[++index] = val`** **probably takes at least two steps**
  - Code has a data race, allowing very strange behavior
    - Compiler optimizations may break it in ways you had not anticipated
    - See Grossman notes for more details

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

Recap: the distinction

The term "race condition" can refer to two *different* things resulting from lack of synchronization:

1. Data races: Simultaneous read/write or write/write of the same memory location

2. Bad interleavings: Exposes bad intermediate state to other threads, leads to behavior we find incorrect
   - "Bad" depends on your specification

Getting it right

Avoiding race conditions on shared resources is difficult

- What 'seems fine' in a sequential world can get you into trouble when multiple threads are involved
- Decades of bugs have led to some *conventional wisdom*:

  general techniques that are known to work

Next, we discuss this conventional wisdom!

- Parts paraphrased from "Java Concurrency in Practice"
  - Chapter 2 (rest of book more advanced)
- But none of this is specific to Java or a particular book!
- May be hard to appreciate in beginning, but come back to these guidelines over the years!

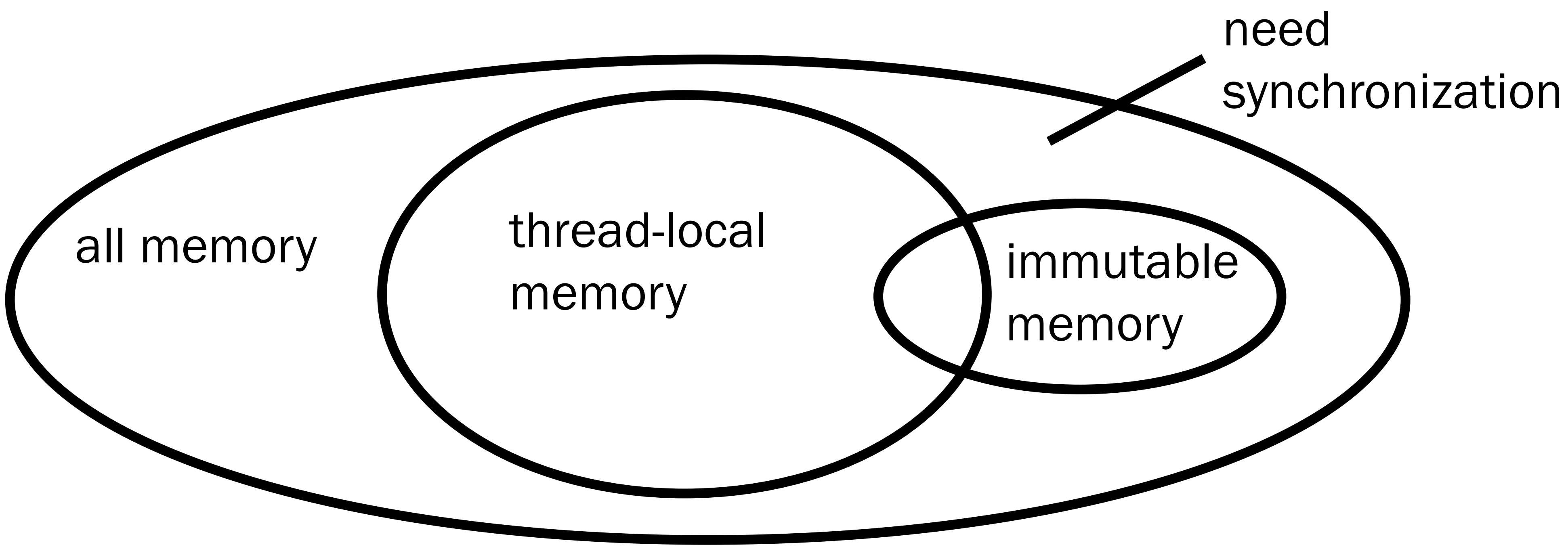
Conventional Wisdom

See Section 8 in Grossman Notes

3 choices

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

1. Thread-local: Do not use the location in > 1 thread
2. Immutable: Do not write to the memory location
3. Shared-and-mutable: Use synchronization to control access to the location



1. Thread-local

Whenever possible, do not share resources

- Easier to have each thread have its own **thread-local** ***copy*** of a resource than to have one with shared updates
- This is correct only if threads do not need to communicate through the resource
  - That is, multiple copies are a 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: shared-memory should be rare – minimize it

2. Immutable

Whenever possible, do not update objects

- Make new objects instead!

- One of the key tenets of *functional programming* (see CSE 341)
  - Generally helpful to avoid *side-effects*
  - Much more helpful in a concurrent setting

- If a location is only read, never written, then no synchronization is necessary!
  - Simultaneous reads are *not* races and *not* a problem

In practice, programmers usually over-use mutation – minimize it

3. The rest: Keep it synchronized

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

Guideline #0: No data races

- *Never allow two threads to read/write or write/write the same location at the same time* (use locks!)
  - Even if it 'seems safe'

Necessary:

a Java or C program with a data race is by definition wrong

But Not sufficient: Our **`peek`** example had no data races, and it's still wrong...

Consistent Locking

Guideline #1: Use consistent locking

- *Every location needing synchronization has a lock that is always held when reading or writing the location*

- We say the lock **guards** the location

- The same lock can (and often should) guard multiple locations (ex. multiple fields in a class)

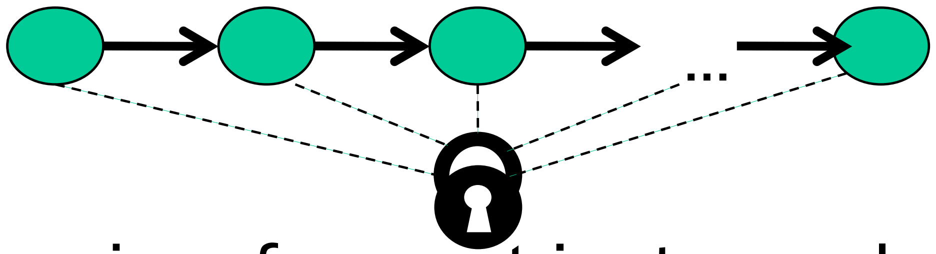
- Clearly document the guard for each location

- In Java, often the guard is the object containing the location
  - `this` inside the object's methods
  - But also often guard a larger structure with one lock to ensure mutual exclusion on the structure

Lock granularity

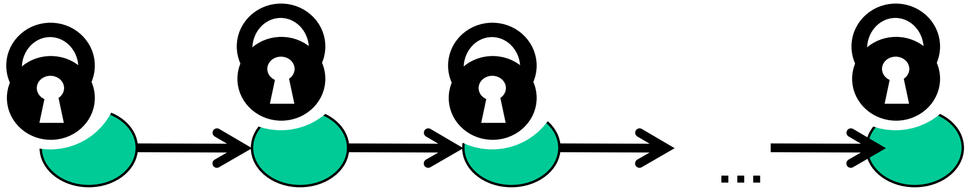
Coarse-grained: Fewer locks, i.e., more objects per lock

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



Fine-grained: More locks, i.e., 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

Poll Everywhere

pollev.com/artliu

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

Which makes implementing **`resize`** easier?

- How would you do it?

coarse

If a hashtable has a **`numElements`** field, maintaining it will destroy the benefits of using separate locks for each bucket, why?

Trade-offs

Coarse-grained advantages:

- Simpler to implement
- 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 advantages:

- More simultaneous access (performance when coarse-grained would lead to unnecessary blocking)
- Can make multi-node operations more difficult: say, rotations in an AVL tree

Guideline #2: *Start with coarse-grained (simpler) and move to fine-grained (performance) only if contention on the coarser locks becomes an issue.*

Critical-section granularity

A second, orthogonal granularity issue is critical-section size

- How much work to do while holding lock(s)?

If critical sections run for too long?

If critical sections are too short?

Critical-section granularity

A second, orthogonal granularity issue is critical-section size

- How much work to do while holding lock(s)?

If critical sections run for **too long**:

- Performance loss because other threads are blocked

If critical sections are **too short**:

- Bugs because you broke up something where other threads should not be able to see intermediate state

Guideline #3: *Don’t do expensive computations or I/O in critical sections, but also don’t introduce race conditions; keep it as small as possible but still be correct*

Example 1: 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
- **`expensive()`** takes in the old value, and computes a new one, but takes a long time

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

Example 2: 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

```
synchronized(lock) {
  v1 = table.lookup(k);
}
v2 = expensive(v1);
synchronized(lock) {
  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"
- Typically want ADT operations atomic, even to other threads running operations on the same ADT

Guideline #4: *Think in terms of what operations need to be atomic*

- Make critical sections just long enough to preserve atomicity
- *Then* design the locking protocol to implement the critical sections correctly

That is: Think about atomicity first and locks second

Don’t roll your own

- In “real life”, it is unusual to have to write your own data structure from scratch
  - Implementations provided in standard libraries
  - Point of CSE332 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`** written by world experts

Guideline #5: *Use built-in libraries whenever they meet your needs*

Deadlock

Motivating Deadlock Issues

Consider a method to transfer 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);
  }
}
```

x. transferTo(5, y)

Potential problems?

The Deadlock

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

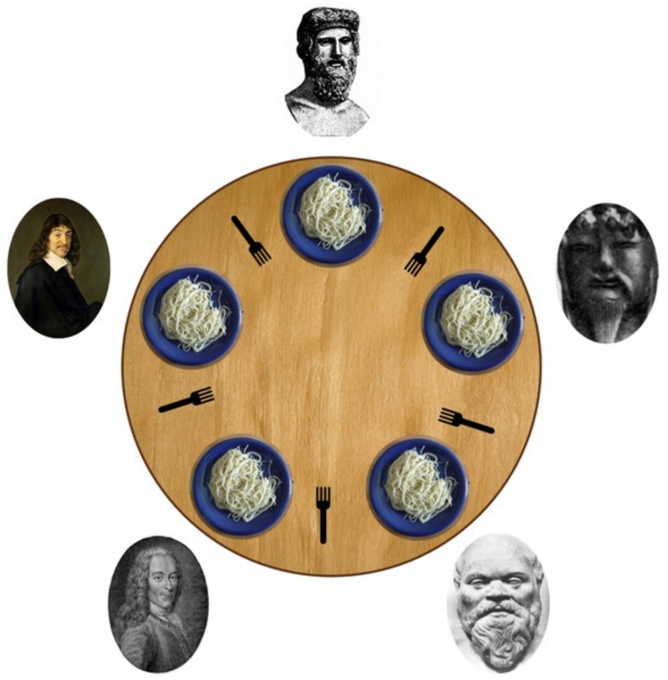
Time	Thread 1: `x.transferTo(1,y)`	Thread 2: `y.transferTo(1,x)`
↓	***acquire lock for x***	
	do withdraw from x	
		acquire lock for y
		do withdraw from y
		block on lock for x
	block on lock for y	

x

y

Another presentation: The Dining Philosophers

- 5 philosophers go out to dinner together at an Italian restaurant
- Sit at a round table; one fork per setting
- When the spaghetti comes, each philosopher proceeds to grab their right fork, then their left fork, then eats
- ‘Locking’ for each fork results in a ***deadlock***



Deadlock, in general

A deadlock occurs when we have a cycle of dependencies

ie: there are threads $T_1, \ldots, T_n$ such that:

- Thread T_i is waiting for a resource held by T_{i+1} and
- T_n is waiting for a resource held by T_1

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 here, but exposes wrong total amount in 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

x → y   y → x

Motivating Deadlock Issues

Consider a method to transfer 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);
  }
}
```

Potential problems?

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);
      }}
  }
}
```

Perspective

- Code like account-transfer are more sneaky examples of deadlock

- Easier case: different types of objects
  - Can 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 in the tree"

Concurrency summary

- Concurrent programming allows multiple threads to access shared resources (e.g. hash table, work queue)
- Introduces new kinds of bugs:
  - Race Conditions { Data races and Bad Interleavings }
  - Critical sections too small
  - Critical sections use wrong locks
  - Deadlocks
- Requires synchronization
  - Locks for mutual exclusion (common, various flavors)
  - Other Synchronization Primitives: (see Grossman notes)
    - Reader/Writer Locks
    - Condition variables for signaling others
- Guidelines for correct use help avoid common pitfalls