

Parallel Programming

Shared memory concurrency, locks and data races

Toward sharing resources (memory)

Have been studying **parallel algorithms** using fork-join

- Lower span via parallel tasks

Algorithms all had a simple *structure* to avoid race conditions

- Each thread had memory “only it accessed”, e.g: array sub-range
- On **fork**, “loan” some memory to “forkee” and do not access that memory again until after **join** on the “forkee”

Strategy won't work well when:

- Memory accessed by threads is overlapping or unpredictable
- Threads are doing independent tasks needing access to same resources (rather than implementing the same algorithm)

Managing state

Main challenge for parallel programs

Approaches:

- immutability
 - data do not change
 - best option, should be used when possible
- isolated mutability
 - data can change, but only one thread/task can access them
- mutable/shared data
 - data can change / all tasks/threads can potentially access them

Mutable/Shared data

- present in shared memory architectures
- **however:** concurrent accesses may lead to inconsistencies
- **solution:** protect state by allowing only **one** task/thread to access it at a time

Dealing with mutable/shared state

State needs to be protected (in general)

- exclusive access
- intermediate inconsistent states should not be observed

Methods:

- locks: mechanism to ensure exclusive access/atomicity
 - ensuring good performance / correctness with locks can be hard (especially for “programming in the large”)
- Transactional memory: programmer describes a set of actions that need to be atomic
 - easier for the programmer, but getting good performance might be challenging

Canonical example

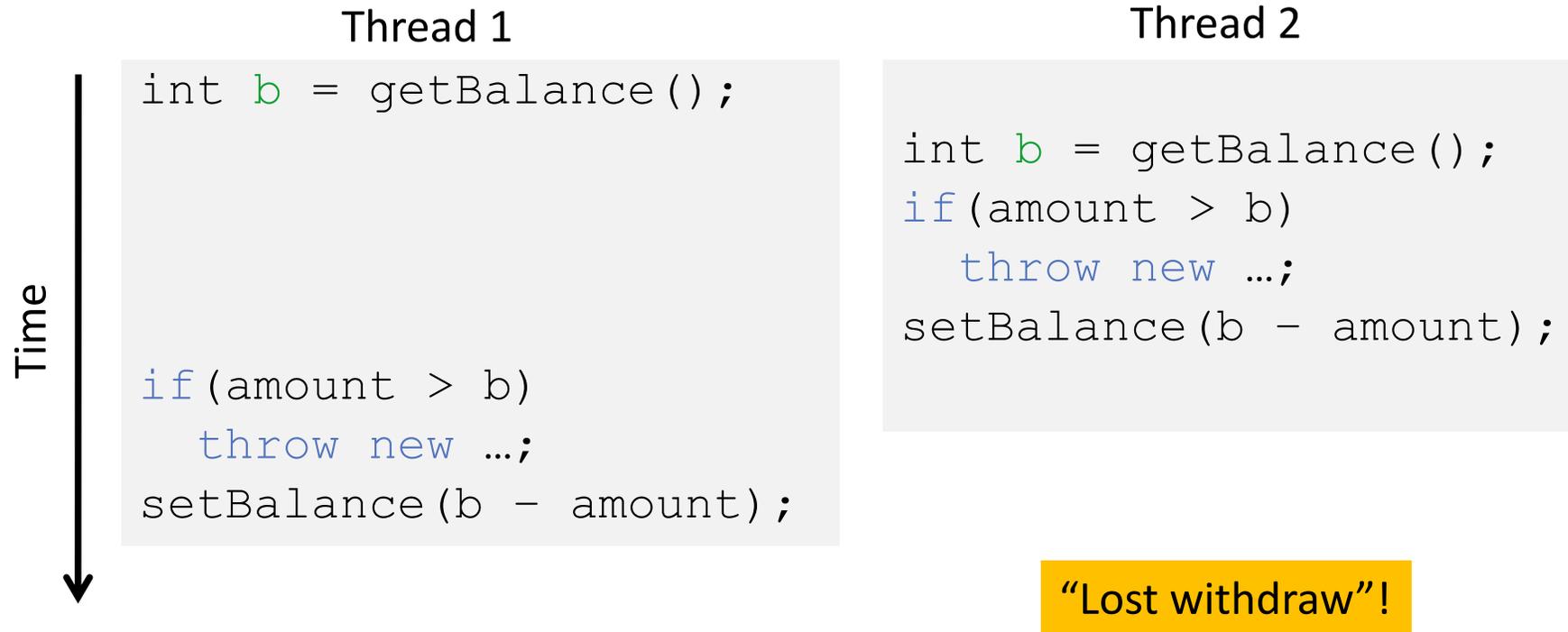
Correct code in a single-threaded world

```
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.
}
```

A bad interleaving

Interleaved `withdraw(100)` calls on the same account

- Assume initial `balance == 150`



Interleaving (recap)

If second call starts before first ends, we say the calls **interleave**

- Could happen even with one processor since a thread can be **pre-empted** at any point for time-slicing

If **x** and **y** refer to different accounts, no problem

- “You cook in your kitchen while I cook in mine”
- But if **x** and **y** alias, possible trouble...

Incorrect “fix”

It is tempting and almost always **wrong** to fix a bad interleaving by rearranging or repeating operations, such as:

```
void withdraw(int amount) {  
    if(amount > getBalance())  
        throw new WithdrawTooLargeException();  
    // maybe balance changed  
    setBalance(getBalance() - amount);  
}
```

This fixes nothing!

- Narrows the problem by one statement
- (Not even that since the compiler could turn it back into the old version because you didn't indicate need to synchronize)
- And now a negative balance is possible – why?

Mutual exclusion

Sane fix: Allow at most one thread to withdraw from account **A** at a time

- Exclude other simultaneous operations on **A** too (e.g., deposit)

Called **mutual exclusion**: One thread using a resource (here: an account) means another thread must wait

- a.k.a. **critical sections**, which technically have other requirements

Programmer must implement critical sections

- “The compiler” has no idea what interleavings should or should not be allowed in your program
- But you need language primitives to do it!

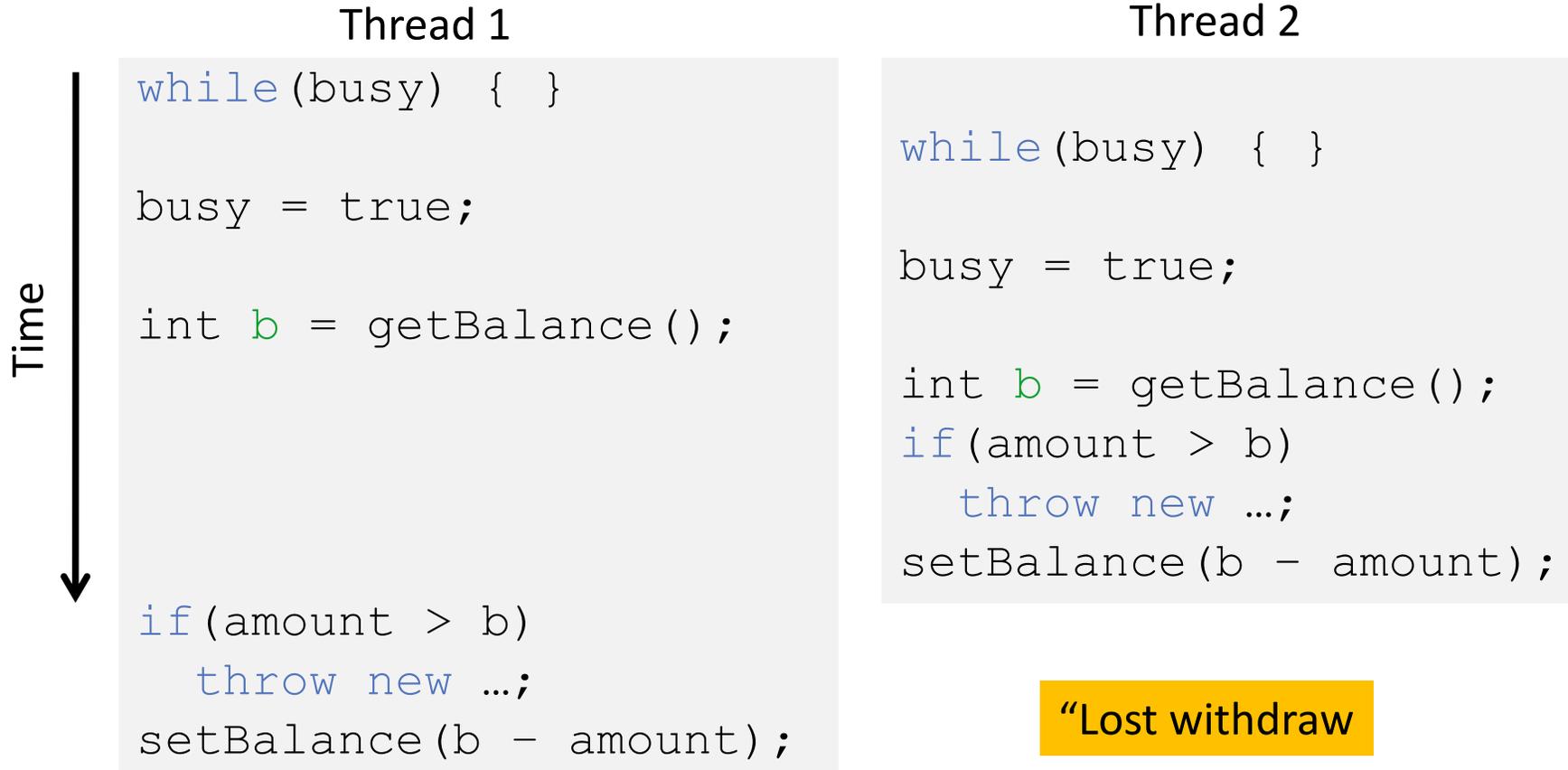
Wrong!

Why can't we implement our own mutual-exclusion protocol?

- It's technically possible under certain assumptions, but won't work in real languages anyway

```
class BankAccount {
    private int balance = 0;
    private boolean busy = false;
    void withdraw(int amount) {
        while(busy) { /* "spin-wait" */ }
        busy = true;
        int b = getBalance();
        if(amount > b)
            throw new WithdrawTooLargeException();
        setBalance(b - amount);
        busy = false;
    }
    // deposit would spin on same boolean
}
```

Just moved the problem!



What we need

- Many ways out of this conundrum, but we need help from the language
- A basic solution: **Locks**
 - Not Java yet, though Java's approach is similar and slightly more convenient
- Basic synchronization primitive with operations:
 - **new**: make a new lock, initially *"not held"*
 - **acquire**: blocks if this lock is already currently *"held"*
 - Once *"not held"*, makes lock *"held"* [all at once!]
 - **release**: makes this lock *"not held"*
 - If ≥ 1 threads are blocked on it, exactly 1 will acquire it

Why that works

- A primitive with atomic operations **new**, **acquire**, **release**
- The lock implementation ensures that given simultaneous acquires and/or releases, a correct thing will happen
 - Example: Two acquires: one will “win” and one will block
- How can this be implemented?
 - Need to “check if held and if not make held” “all-at-once”
 - Uses special hardware and O/S support
 - See computer-architecture or operating-systems course
 - Here, we take this as a primitive and use it

Topics Today

- Critical Sections, Mutual Exclusion, Lock Objects
- Java's **synchronized** (recap)
- Races: Data Races and Bad Interleavings
- Guidelines for Concurrent Programming

Critical Sections and Mutual Exclusion

Critical Section

Piece of code that may be executed by at most one process (thread) at a time

```
int b = getBalance();  
    if (amount > b) throw new WithdrawTooLargeException();  
setBalance(b - amount);
```

Mutual exclusion

Algorithm to implement a critical section

```
acquire_mutex(); // entry algorithm  
...             // critical section  
release_mutex(); // exit algorithm
```

Required Properties of Mutual Exclusion

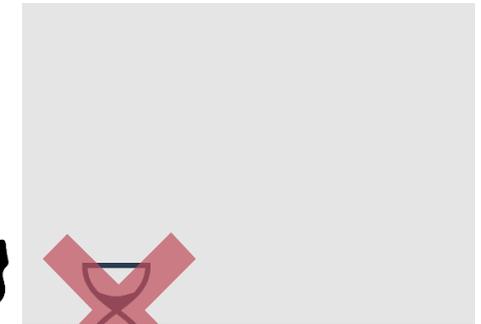
Safety Property

- At most one process executes the critical section code



Liveness

- *Minimally*: `acquire_mutex` must terminate in finite time when no process executes in the critical section



Lock Object

Shared object that satisfies the following interface

```
public interface Lock{  
    public void lock();    // entering CS  
    public void unlock(); // leaving CS  
}
```

providing the following semantics

- new Lock** make a new lock, initially *“not held”*
- acquire** blocks (only) if this lock is already currently *“held”*
Once *“not held”*, makes lock *“held”* [all at once!]
- release** makes this lock *“not held”*
If ≥ 1 threads are blocked on it, exactly 1 will acquire it



Why that works

The lock implementation ensures that given simultaneous acquires and/or releases, a correct thing will happen

- Example: Two acquires: one will “win” and one will block
- A lock thus implements a mutual exclusion algorithm.

How can this be implemented?

- Need to “check if held and if not make held” “all-at-once”
- Uses special hardware and O/S support
- For the time being, we take this as a primitive and use it

Almost-correct pseudocode

```
class BankAccount {  
    private int balance = 0;  
    private Lock lk = new Lock();  
    ...  
    void withdraw(int amount) {  
        lk.acquire(); // may block  
        int b = getBalance();  
        if(amount > b)  
            throw new WithdrawTooLargeException();  
        setBalance(b - amount);  
        lk.release();  
    }  
    // deposit would also acquire/release lk  
}
```

One lock for
each account

Possible mistakes

Incorrect: Use different locks for **withdraw** and **deposit**

- Mutual exclusion works only when using same lock
- **balance** field is the shared resource being protected

Poor performance: Use same lock for every bank account

- No simultaneous operations on different accounts

Incorrect: Forget to release a lock (blocks other threads forever!)

- Previous slide is **wrong** because of the exception possibility!

```
if(amount > b) {  
    lk.release(); // hard to remember!  
    throw new WithdrawTooLargeException();  
}
```

Other operations

If **withdraw** and **deposit** use the same lock, then simultaneous calls to these methods are properly synchronized

But what about **getBalance** and **setBalance**?

- Assume they are **public**, which may be reasonable
- If they *do not* acquire the same lock, then a race between **setBalance** and **withdraw** could produce a wrong result
- If they *do* acquire the same lock, then **withdraw** would block forever because it tries to acquire a lock it already has

```
public void setBalance(int x) { .. }  
public int getBalance() { .. }  
public void withdraw(int amount) {  
    ..  
    b = getBalance()  
    ..  
    setBalance(b - amount);  
    ..  
}  
public void deposit(int amount){  
    ..  
    b = getBalance()  
    ..  
    setBalance(b + amount);  
    ..  
}
```

Re-acquiring locks?

One approach:

Can't let outside world call **setBalance1**

Can't have **withdraw** call **setBalance2**

Another approach:

Can modify the meaning of the Lock to support *re-entrant locks*

- Java does this
- Then just use **setBalance2**

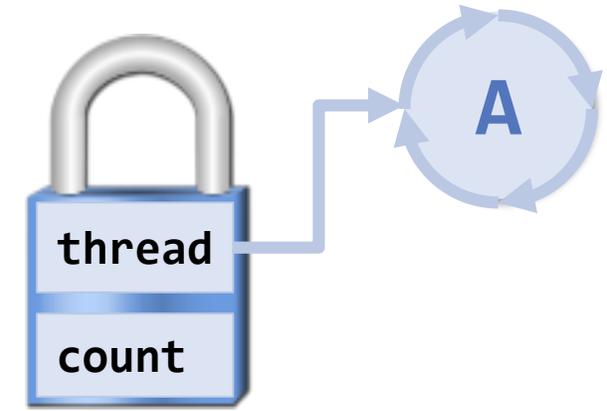
```
int setBalance1(int x) {
    balance = x;
}
int setBalance2(int x) {
    lk.acquire();
    setBalance1(x);
    lk.release();
}
void withdraw(int amount) {
    lk.acquire();
    ...
    setBalance1(b - amount);
    lk.release();
}
```

Re-entrant lock

A **re-entrant lock** (a.k.a. **recursive lock**)

“remembers”

- the thread (if any) that currently holds it
- a *count*



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*

Re-entrant locks work

- This simple code works fine provided **lk** is a reentrant lock
- Okay to call **setBalance** directly
- Okay to call **withdraw** (won't block forever)

```
int setBalance(int x) {  
    lk.acquire();  
    balance = x;  
    lk.release();  
}  
  
void withdraw(int amount) {  
    lk.acquire();  
    ...  
    setBalance(b - amount);  
    lk.release();  
}
```

Now some Java (a bit of recap)

Java has built-in support for re-entrant locks

- Several differences from our pseudocode
- Focus on the **synchronized** statement

```
synchronized (expression)  
{  
  statements  
}
```

1. Evaluates *expression* to an object
Every object “is a lock” in Java (but not primitive types)

2. Acquires the lock, blocking if necessary
“If you get past the {, you have the lock”

3. Releases the lock “at the matching }
Even if control leaves due to throw, return, etc.

More Java notes

Class `java.util.concurrent.locks.ReentrantLock` works much more like our pseudocode

- Often use `try { ... } finally { ... }` to avoid forgetting to release the lock if there's an exception

Also library and/or language support for *readers/writer locks* and *conditional variables* (future lectures)

Java provides many other features and details. See, for example:

- Java “Concurrency in Practice” by Goetz et al
- Chapter 30 of “Introduction to Java Programming” by Daniel Liang
- Chapter 14 of “CoreJava”, Volume 1 by Horstmann/Cornell

Races

Roughly: a **race condition** occurs when the computation result depends on the scheduling (how threads are interleaved)

Bugs that exist only due to concurrency

No interleaved scheduling with only one *thread* [no concurrency]

[But interleaved scheduling with only one processor *is* possible!]

Typically, problem is some *intermediate state* that “messes up” a concurrent thread that “sees” that state

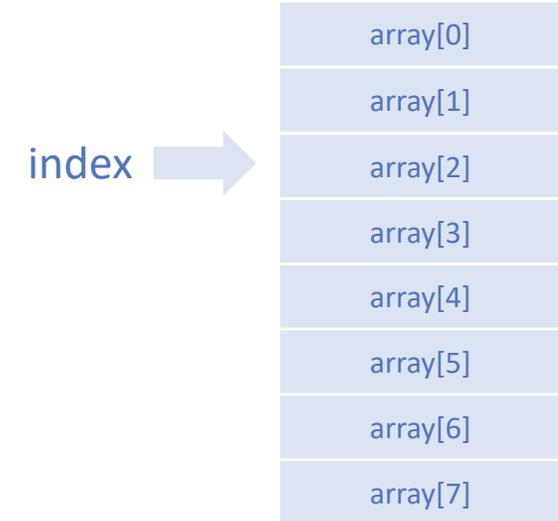
Note: This lecture makes a big distinction between *data races* and *bad interleavings*, both kinds of race-condition bugs

- Confusion often results from not distinguishing these or using the ambiguous “race condition” to mean only one

Example: Bounded Stack

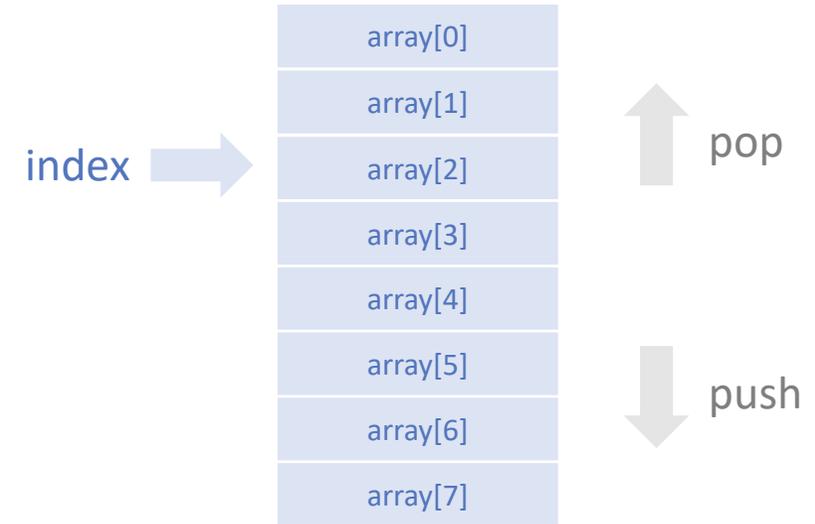
```
class StackFullException extends Exception {}  
class StackEmptyException extends Exception {}
```

```
public class Stack <E> {  
    E[] array;  
    int index;  
  
    public Stack(int entries){  
        // hack to generate a generic array, initialized with NIL values  
        array = (E[]) new Object[entries];  
        index = 0;  
    }  
    ...  
}
```



Example: Bounded Stack

```
public class Stack <E> {  
...  
    synchronized boolean isEmpty() {  
        return index==0;  
    }  
  
    synchronized void push(E val) throws StackFullException {  
        if (index==array.length)  
            throw new StackFullException();  
        array[index++] = val;  
    }  
  
    synchronized E pop() throws StackEmptyException {  
        if (index==0) throw new StackEmptyException();  
        return array[--index];  
    }  
}
```



Peek ?

```
public class Stack <E> {
```

```
...
```

```
    E peek() {
```

```
        E ans = pop();
```

```
        push(ans);
```

```
        return ans;
```

```
    }
```

```
}
```

wrong !

peek, sequentially speaking

In a sequential world, this code is of questionable *style*, but unquestionably *correct*

The “algorithm” is the only way to write a **peek** helper method if all you had was this interface:

```
interface Stack<E> {
    boolean isEmpty();
    void push(E val);
    E pop();
}

class C implements Stack {
    static <E> E myPeek(Stack<E> s) { ??? }
}
```

peek, concurrently speaking

peek has no *overall* effect on the shared data

It is a “reader” not a “writer”

But the way it is implemented creates an *inconsistent intermediate state*

Even though calls to **push** and **pop** are synchronized so there are *no data races* on the underlying array/list/whatever

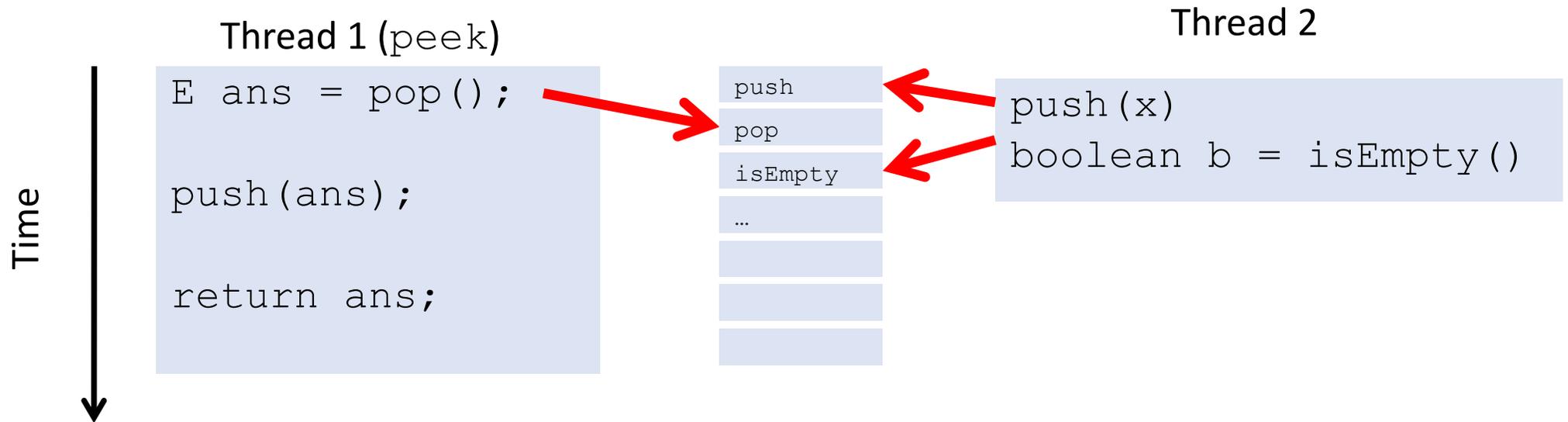
This intermediate state should not be exposed

Leads to several *bad interleavings*

peek and isEmpty

Property we want: If there has been a **push** and no **pop**, then **isEmpty** returns **false**

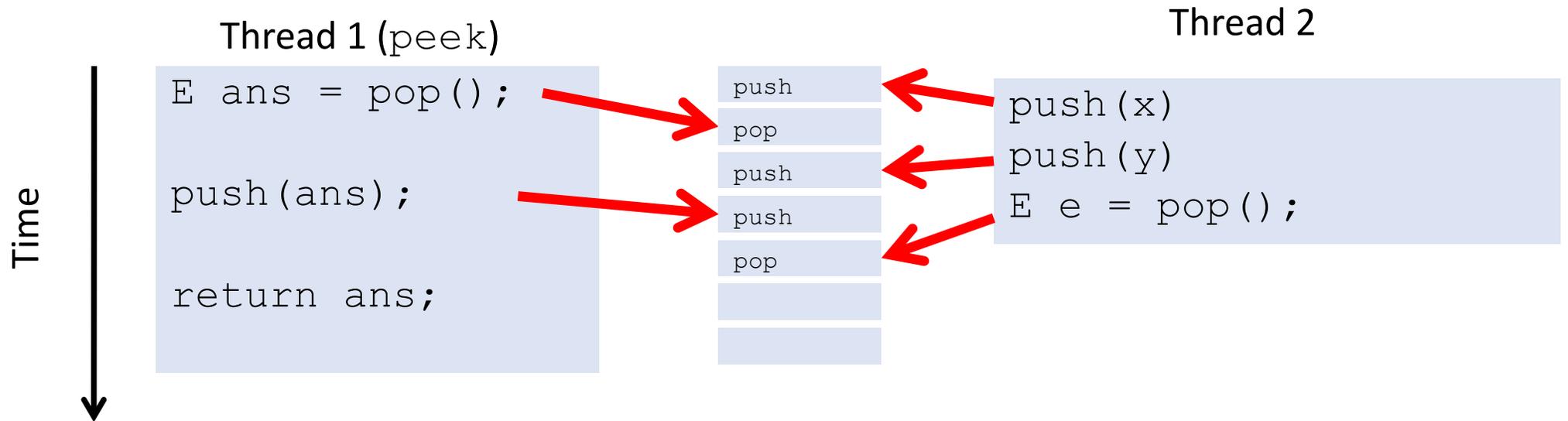
With **peek** as written, property can be violated



peek and LIFO

Property we want: Values are returned from **pop** in LIFO order

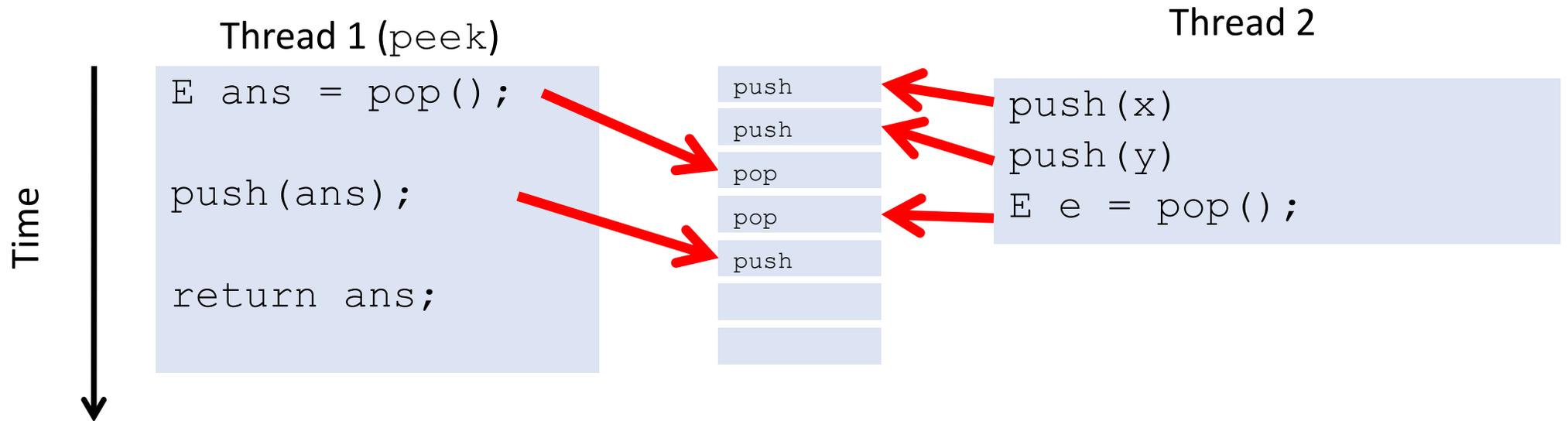
With **peek** as written, property can be violated



peek and LIFO

Property we want: Values are returned from **pop** in LIFO order

With **peek** as written, property can be violated



The fix

In short, **peek** needs synchronization to disallow interleavings

- The key is to make a *larger critical section*
- Re-entrant locks allow calls to **push** and **pop**

```
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;  
        }  
    }  
}
```

The wrong “fix”

```
boolean isEmpty() {  
    return index==0;  
}
```

Focus so far: problems from **peek** doing writes that lead to an incorrect intermediate state

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**...

The distinction

Data Race [aka *Low Level Race Condition, low semantic level*]

Erroneous program behavior caused by insufficiently synchronized accesses of a shared resource by multiple threads, e.g. Simultaneous read/write or write/write of the same memory location

(for mortals) **always an error**, due to compiler & HW

Original **peek** example has no data races

Bad Interleaving [aka *High Level Race Condition, high semantic level*]

Erroneous program behavior caused by an unfavorable execution order of a multithreaded algorithm that makes use of otherwise well synchronized resources.

“Bad” depends on your specification

Original **peek** had several

Getting it right

Avoiding race conditions on shared resources is difficult

- Decades of bugs have led to some *conventional wisdom*:
general techniques that are known to work

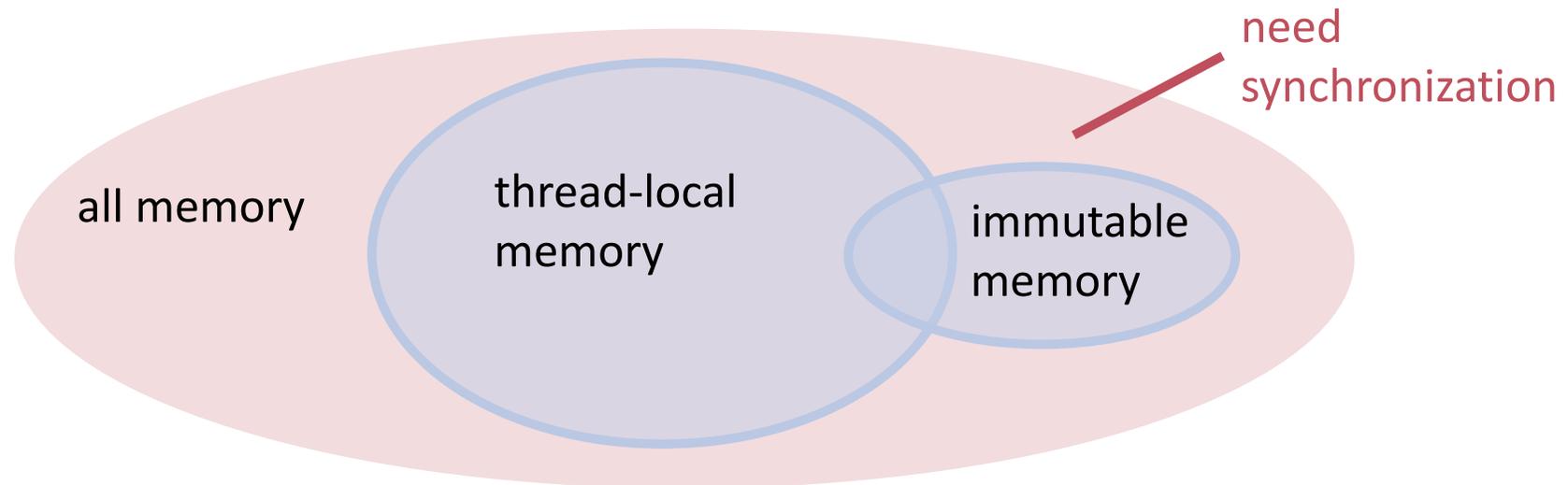
Rest of lecture distills key ideas and trade-offs

- Parts paraphrased from “Java Concurrency in Practice”
- But none of this is specific to Java or a particular book!

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. **Synchronized:** Use synchronization to control access to the location



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

Immutable

Whenever possible, do not update objects

- Make new objects instead
- One of the key tenets of *functional programming*
 - 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

The rest

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. Do not make any assumptions on the orders of reads or writes.

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

Not sufficient: Our **peek** example had no data races

Consistent Locking

Guideline #1: For each location needing synchronization, have 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
- Clearly document the guard for each location

In Java, often the guard is the object containing the location

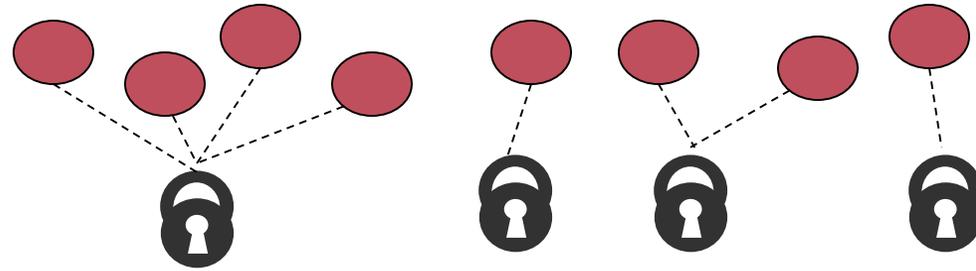
- **this** inside the object's methods
- But also often guard a larger structure with one lock to ensure mutual exclusion on the structure

Consistent Locking continued

The mapping from locations to guarding locks is *conceptual*

- Up to you as the programmer to follow it

It partitions the shared-and-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

Beyond consistent locking

Consistent locking is an *excellent guideline*

- A “default assumption” about program design

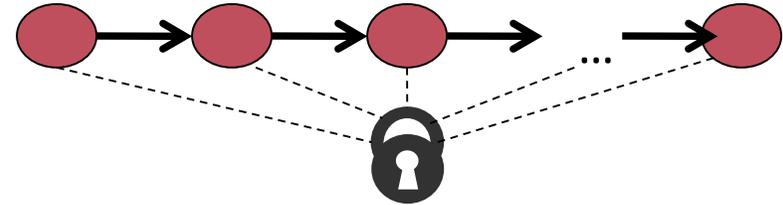
Consistent locking is *not required* for correctness: Can have different program phases use different invariants

- Provided all threads coordinate moving to the next phase

Lock granularity

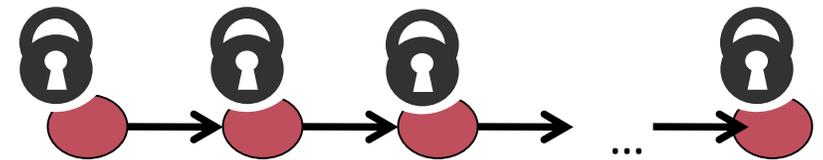
Coarse-grained: Fewer locks, i.e., more objects per lock

- 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

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: operations that modify data-structure shape

Fine-grained advantages

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

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

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
- Performance loss because of frequent thread switching and cache trashing.

Guideline #3: Do not do expensive computations or I/O in critical sections, but also don't introduce race conditions

Example

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

*critical section was too long
(table locked during expensive call)*

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

Example

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

*critical section was too short
(if another thread updated
the entry, we will lose an update)*

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

Example

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

critical section was just right

*(if another update occurred,
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”
- 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

It is rare that you should write your own data structure

- Provided in standard libraries

Particularly 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

Practical Guideline: Use built-in libraries whenever they meet your needs

Guideline for this course: do everything to understand it yourself!