

CSE 332: Data Structures and Parallelism

Spring 2022

Richard Anderson

Lecture 19: Concurrency and Locks

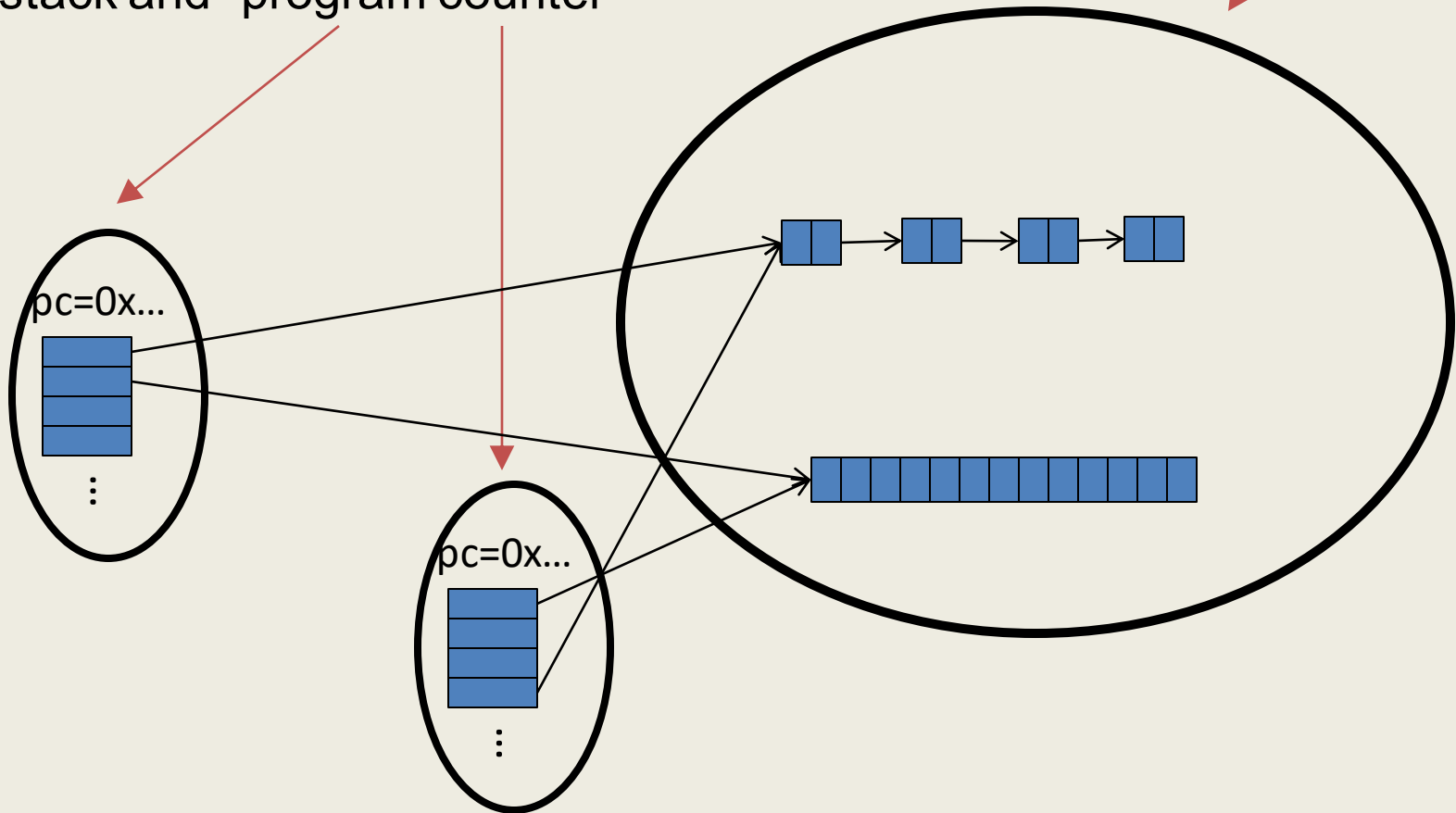
Announcements

- Midterm to be returned Thursday
- Today and Friday - Concurrency

Really sharing memory between Threads

2 Threads, each with own *unshared* call stack and “program counter”

Heap for all objects and static fields, *shared* by all threads



Good sharing

```
class SumTask extends RecursiveTask<Integer> {
    int lo; int hi; int[] arr;
    SumTask(int[] a, int l, int h) { ... }
    protected Integer compute() {
        if(hi - lo < SEQUENTIAL_CUTOFF) {
            int ans = 0;
            for (int i=lo; i < hi; i++)
                ans += arr[i];
            return ans;
        }
        else {
            SumTask left = new SumTask(arr,lo, (hi+lo)/2);
            SumTask right= new SumTask(arr, (hi+lo)/2,hi);
            left.fork();
            int rightAns = right.compute();
            int leftAns = left.join();
            return leftAns + rightAns;
        }
    }
}

static final ForkJoinPool POOL = new ForkJoinPool();
int sum(int[] arr){
    SumTask task = new SumTask(arr,0,arr.length)
    return POOL.invoke(task);
}
```

Banking

- Two threads both trying to `withdraw(100)` from the **same account**:
- Assume initial `balance` 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

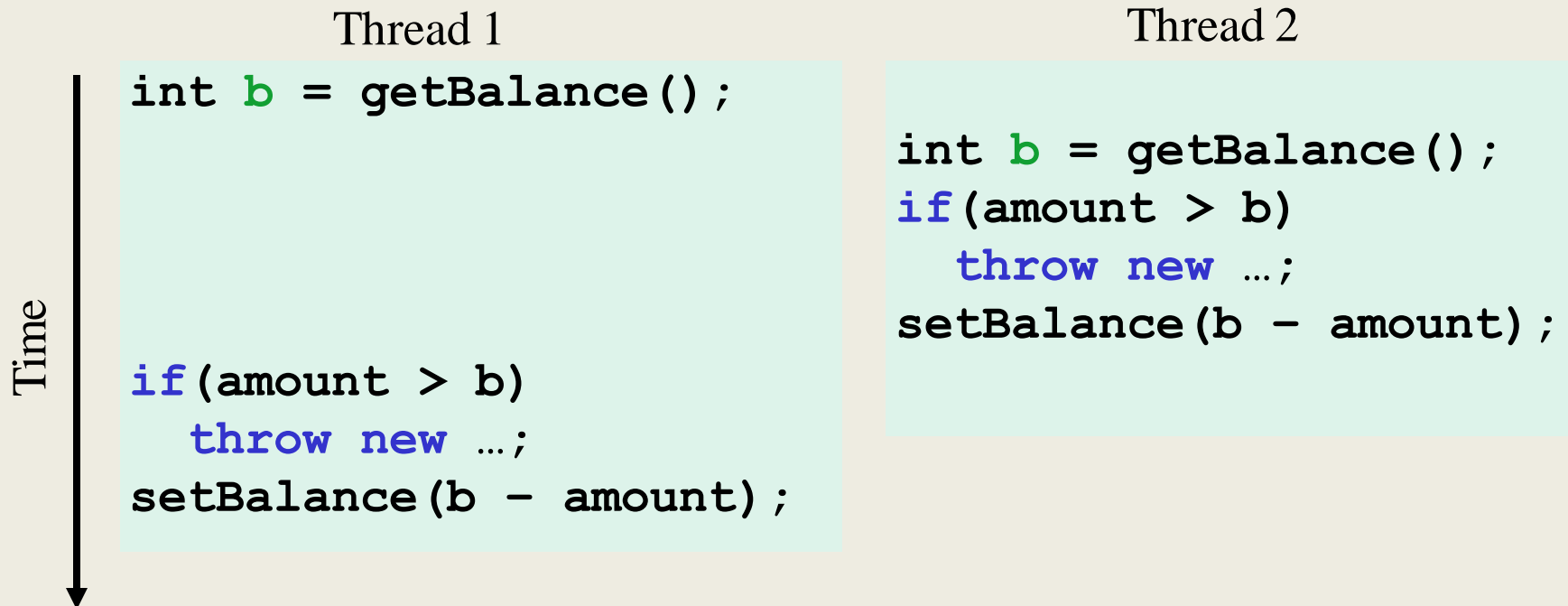
```
x.withdraw(100);
```

Thread 2

```
x.withdraw(100);
```

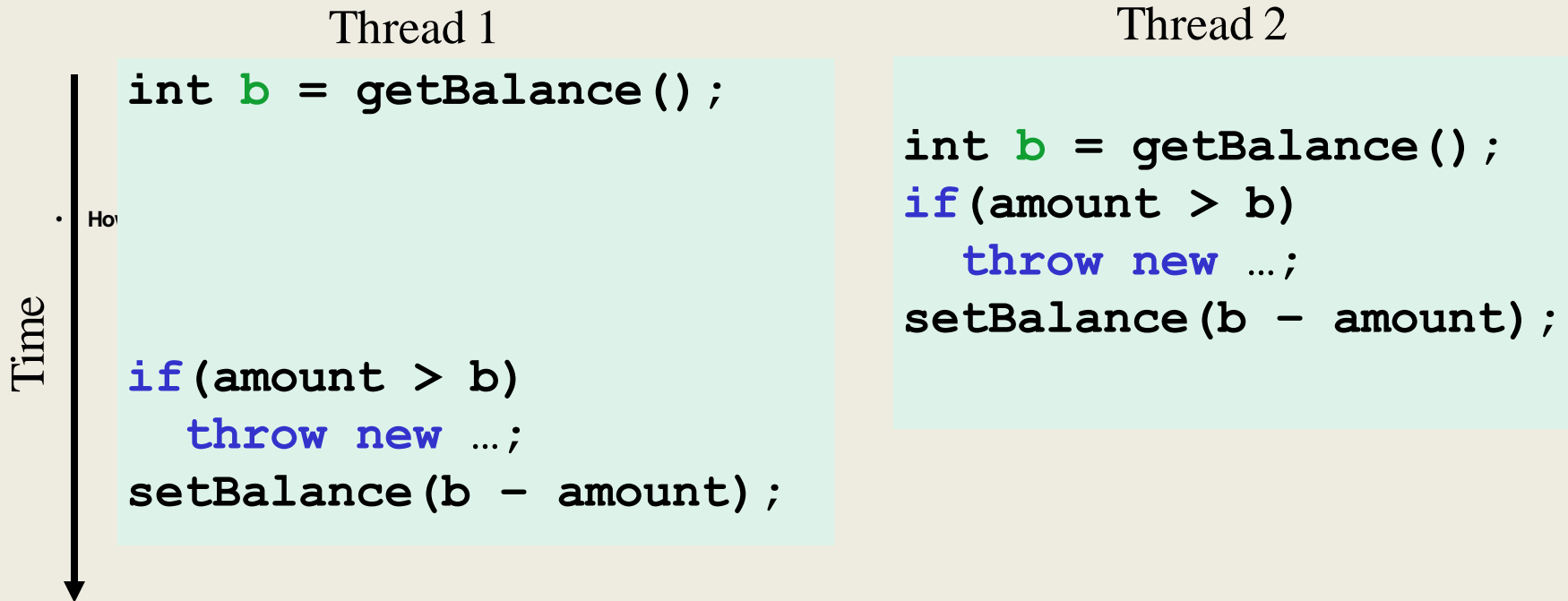
A bad interleaving

- Interleaved **withdraw(100)** calls on the same account
 - Assume initial **balance == 150**



How to fix?

- No way to fix by rewriting the program
 - can always find a bad interleaving -> violation
 - need some kind of synchronization



Race Conditions

A **race condition**: program executes incorrectly due to unexpected order of threads

data race:

- two threads write a variable at the same time
- one thread writes, another reads simultaneously

bad interleaving: wrong result due to unexpected interleaving of statements in two or more threads

Concurrency

Concurrency:

Correctly and efficiently managing access to shared resources from multiple possibly-simultaneous clients

Requires *coordination*

synchronization to avoid incorrect simultaneous access:

make others *block* (wait) until the resource is free

Concurrent applications are often **non-deterministic**

how threads are scheduled affects what operations happen first

non-repeatability complicates testing and debugging

must **work for all possible interleavings!!**

Concurrency Examples

- Bank Accounts
- Airline/hotel reservations
- Wikipedia
- Facebook
- Databases

Locks

- Allow access by at most one thread at a time
 - “mutual exclusion”
 - make others *block* (wait) until the resource is free
 - called a **mutual-exclusion lock** or just **lock**, for short
- Critical sections
 - code that requires mutual exclusion
 - defined by the programmer (compiler can't figure this out)

Lock ADT

We define **Lock** as an ADT 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*” (one thread gets it)
- **release**: makes this lock “*not held*”
 - If ≥ 1 threads are blocked on it, exactly 1 will acquire itAllow access to at most one thread at a time

How can this be implemented?

- acquire (check “not held” -> make “held”) **cannot be interrupted**
- special hardware and operating system-level support

Basic idea *(note Lock is not an actual Java class)*

```
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
}
```

Common Mistakes

- Forgetting to release locks
 - e.g., because of Throws (previous slide)
- Too few locks
 - e.g., all bank accounts share a single lock
- Too many locks
 - separate locks for deposit, withdraw

What Do We Lock?

- Class
 - e.g., all bank accounts?
- Object
 - e.g., a particular account?
- Field
 - e.g., balance
- Code fragment
 - e.g., withdraw

Synchronized: *Locks in Java*

Java has built-in support for locks

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

1. *expression* evaluates to an **object**
 - Any **object** (but not primitive types) can be a lock in Java
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.
 - so *impossible* to forget to release the lock

BankAccount in Java

```
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 ...
            setBalance(b - amount);
        }
    }
    // deposit would also use synchronized(lk)
}
```

Shorthand

Usually simplest to use the class object itself as the lock

```
synchronized (this) {  
    statements  
}
```

This is so common that Java provides a shorthand:

```
synchronized {  
    statements  
}
```

Final Version

```
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 ...
        setBalance(b - amount);
    }
    // deposit would also use synchronized
}
```

Stack Example

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

Why Wrong?

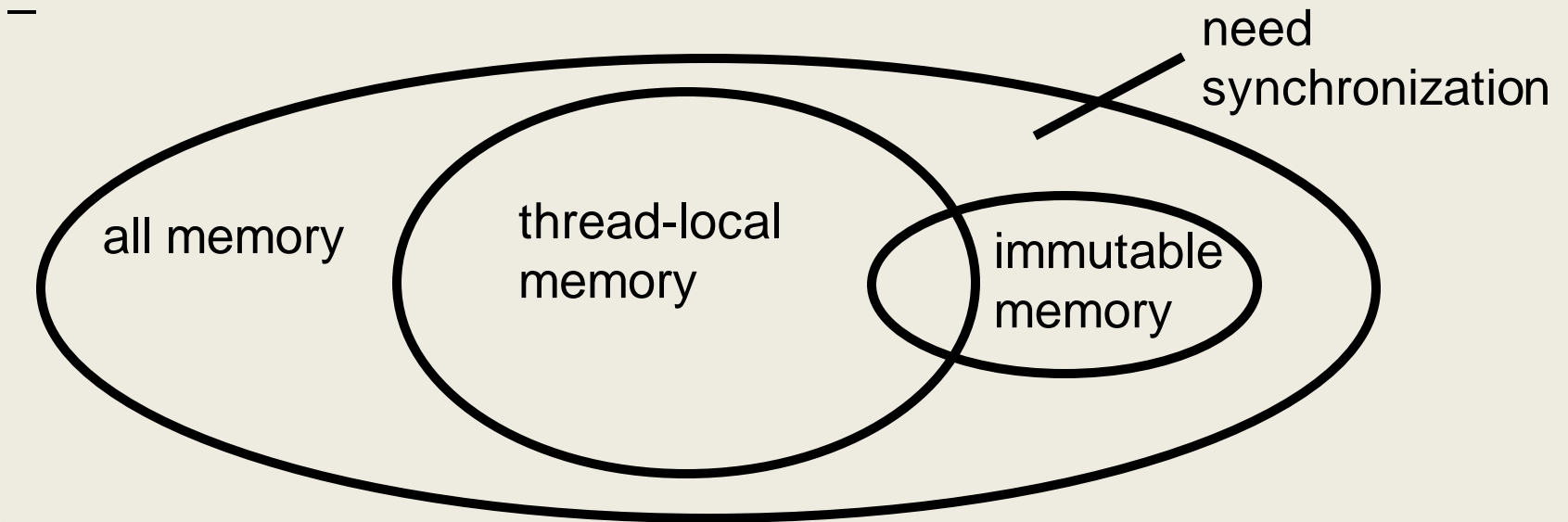
- IsEmpty and push are one-liners. What can go wrong?
 - ans: one line, but multiple operations
 - `array[++index] = val` probably takes at least two ops
 - data race if two pushes happen simultaneously

Stack Example (fixed)

```
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--];  
    }  
}
```

Lock everything? No.

- For every memory location (e.g., object field), obey at least one of the following:
 1. **Thread-local**: only one thread sees it
 2. **Immutable**: read-only
 3. **Shared-and-mutable**: control access via a lock



Thread local

- Whenever possible, do ***not*** share resources
 - easier to give each thread its own local copy
 - only works if threads don't need to communicate via resource
- In typical concurrent programs, the vast majority of objects should be thread local: shared memory should be rare—minimize it

Immutable

- If location is read-only, no synchronization is necessary
- Whenever possible, do **not** update objects
 - make new objects instead!
 - one of the key tenets of *functional programming* (CSE 341)
- In practice, programmers usually over-use mutation – minimize it

The rest: keep it synchronized

Other Forms of Locking in Java

- Java provides many other features and details. See, for example:
 - Chapter 14 of CoreJava, Volume 1 by Horstmann/Cornell
 - Java Concurrency in Practice by Goetz et al

Recall Bank Account Problem

```
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 ...
        setBalance(b - amount);
    }
    // deposit would also use synchronized
}
```

Call to setBalance in withdraw

- tries to lock **this**

Re-Entrant Lock

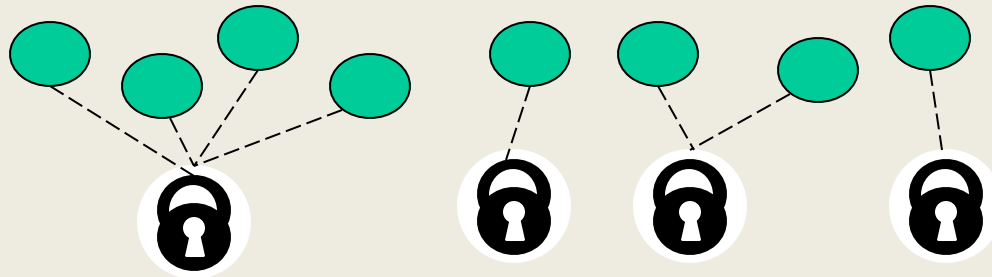
- A **re-entrant lock** (a.k.a. **recursive lock**)
 - If a thread holds a lock, subsequent attempts to acquire the **same lock** in the **same thread** won't block
 - **withdraw** can acquire the lock and **setBalance** can also acquire it
 - implemented by maintaining a count of how many times each lock is acquired in each thread, and decrementing the count on each release.
- Java **synchronize** locks are re-entrant

Locking Guidelines

- Correctness
- Consistency: make it well-defined
- Granularity: coarse to fine
- Critical Sections: make them small, atomic
- Leverage libraries

Consistent Locking

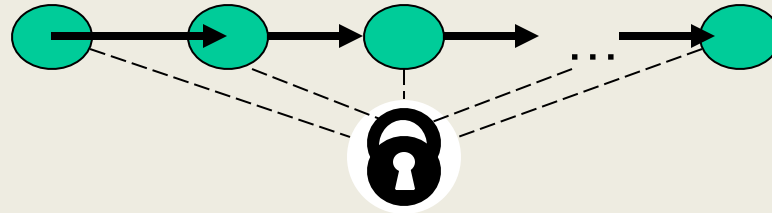
- Clear mapping of locks to resources
 - followed by all methods
 - clearly documented
 - same lock can guard multiple resources



- what's a resource? Conceptual:
 - object
 - field
 - data structure (e.g., linked list, hash table)

Lock Granularity

- **Coarse grained:** fewer locks, more objects per lock
 - e.g., one lock for entire data structure (e.g., linked list)



- advantage:
- disadvantage:

- **Fine grained:** more locks, fewer objects per lock
 - e.g., one lock for each item in the linked list



Lock Granularity

- Example: hashtable with separate chaining
 - coarse grained: one lock for whole table
 - fine grained: one lock for each bucket
- Which supports more concurrency for **insert** and **lookup**?
- Which makes implementing **resize** easier?
- Suppose hashtable maintains a **numElements** field. Which locking approach is better?

Critical Sections

- Critical sections:
 - how much code executes while you hold the lock?
 - want critical sections to be short
 - make them “atomic”: think about smallest sequence of operations that have to occur at once (without data races, interleavings)

Critical Sections

- Suppose we want to change a value in a hash table
 - assume one lock for the entire table
 - computing the new value takes a long time (“expensive”)

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

Critical Sections

- Suppose we want to change a value in the hash table
 - assume one lock for the entire table
 - computing the new value takes a long time (“expensive”)
 - will this work?

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

Critical Sections

- Suppose we want to change a value in the hash table
 - assume one lock for the entire table
 - computing the new value takes a long time (“expensive”)
 - convoluted fix:

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