Lecture 27: Concurrency

Slides adapted from Dan Grossman

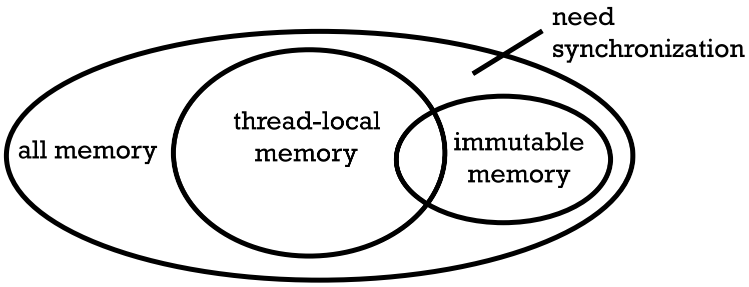
+ Today

- Reading
  - P&C Sections 8 and 9
- Objectives
  - Design for this week's homework
  - Deadlocks
  - (Maps)
- Announcements
  - Quiz on Friday covering P&C

Providing Safe Access

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

1. Thread-local: Only one thread accesses it
2. Immutable: (After initialization) Only read never written
3. Synchronized: Locks used to ensure no race conditions



Guidelines for unavoidable concurrency

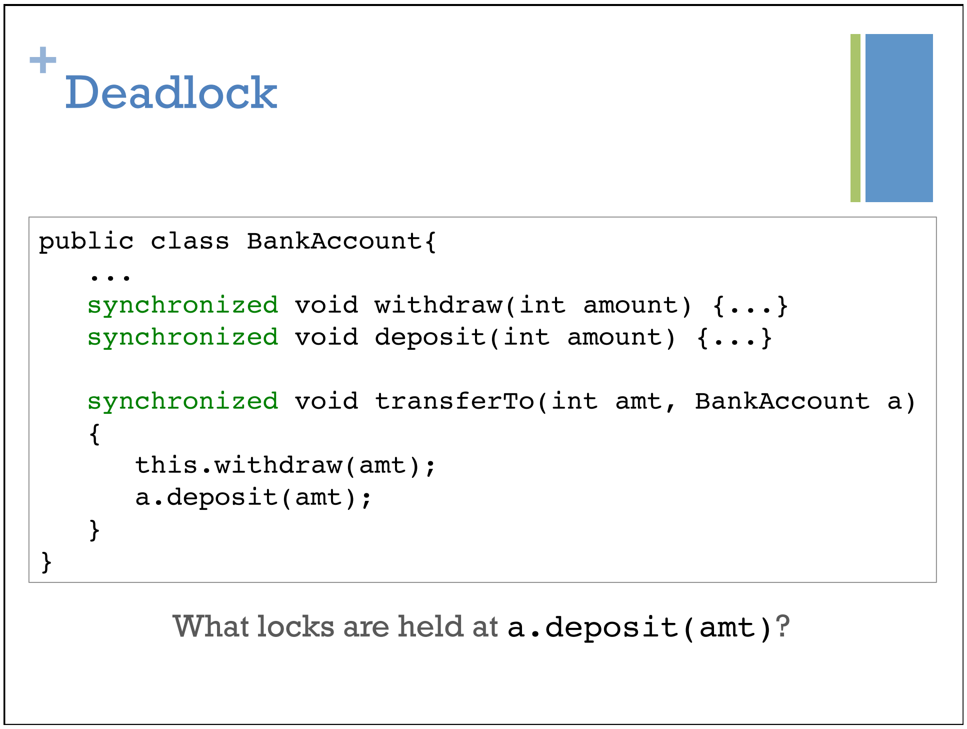
Use threads to ensure no simultaneous read/write or write/write operations to the same field

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

Start with coarse-grained locking and move to fine-grained locking only if *contention* becomes an issue.

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

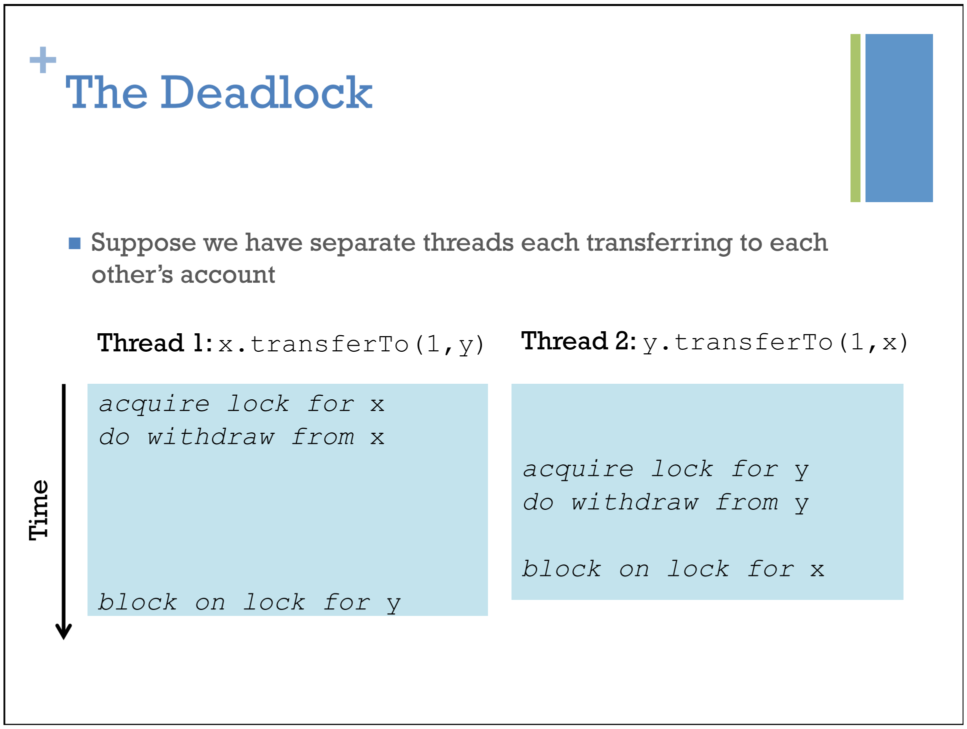
Use built-in libraries whenever they meet your needs



+ Deadlock

```
public class BankAccount{
   ...
   synchronized void withdraw(int amount) {...}
   synchronized void deposit(int amount) {...}

   synchronized void transferTo(int amt, BankAccount a)
   {
      this.withdraw(amt);
      a.deposit(amt);
   }
}
```

What locks are held at `a.deposit(amt)`?



+ The Deadlock

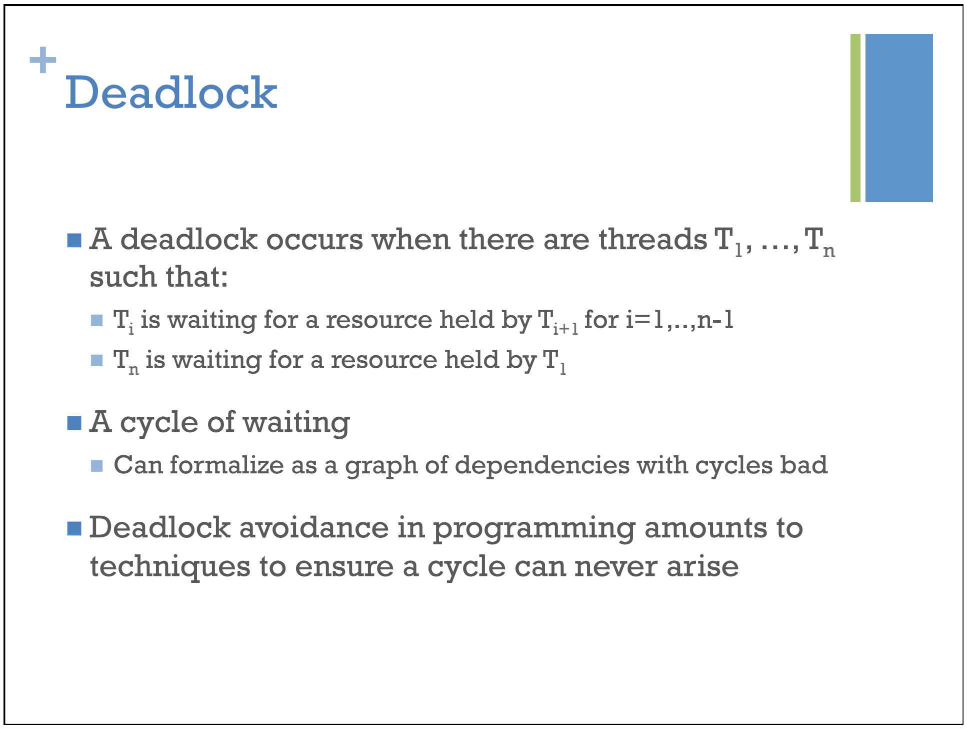
- Suppose we have separate threads each transferring to each other's account

Thread 1: `x.transferTo(1,y)`

```
acquire lock for x
do withdraw from x



block on lock for y
```

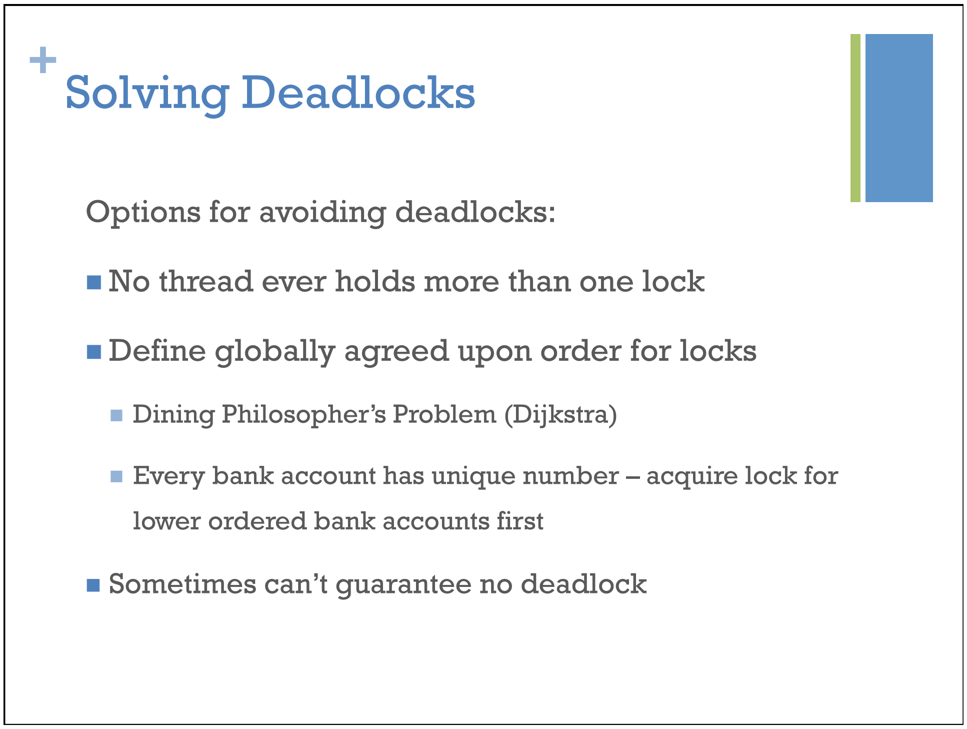
Thread 2: `y.transferTo(1,x)`

```

acquire lock for y
do withdraw from y

block on lock for x
```



Deadlock

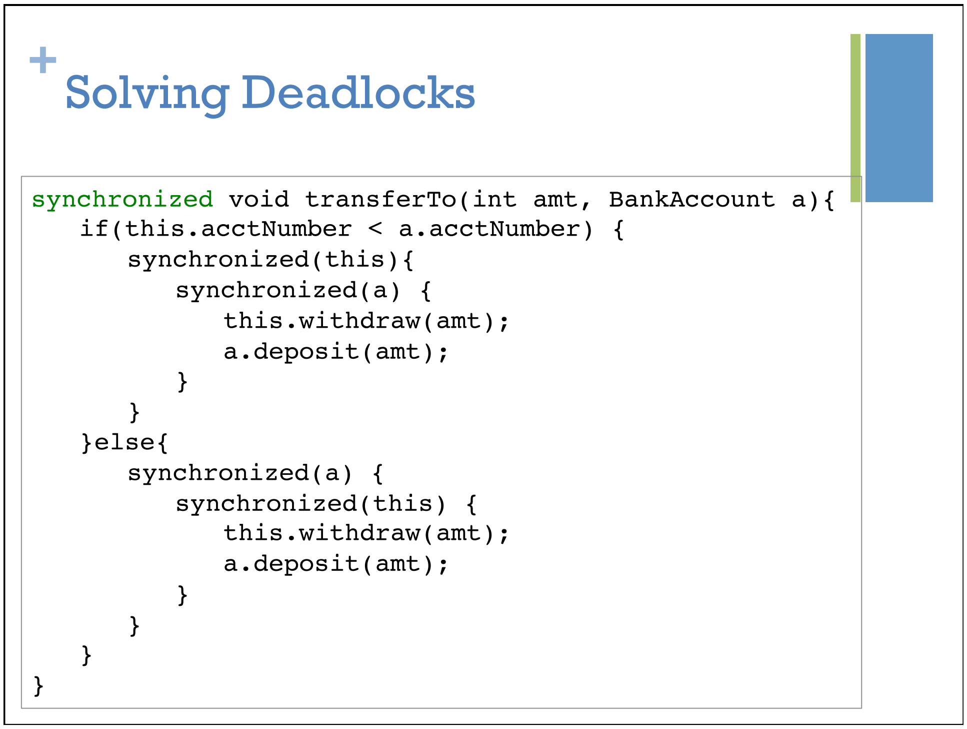
- A deadlock occurs when there are threads $T_1, \ldots, T_n$ such that:
  - T_i is waiting for a resource held by T_{i+1} for i=1,..,n-1
  - T_n is waiting for a resource held by T_1
- A cycle of waiting
  - Can formalize as a graph of dependencies with cycles bad
- Deadlock avoidance in programming amounts to techniques to ensure a cycle can never arise



Solving Deadlocks

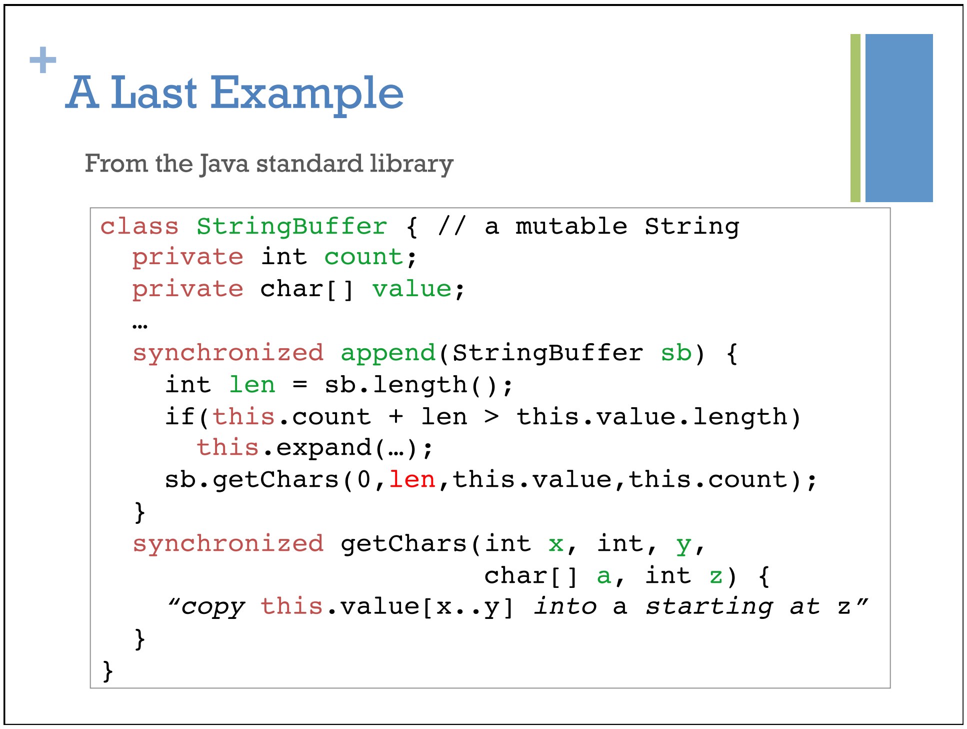
Options for avoiding deadlocks:

- No thread ever holds more than one lock
- Define globally agreed upon order for locks
  - Dining Philosopher's Problem (Dijkstra)
  - Every bank account has unique number – acquire lock for lower ordered bank accounts first
- Sometimes can't guarantee no deadlock



Solving Deadlocks

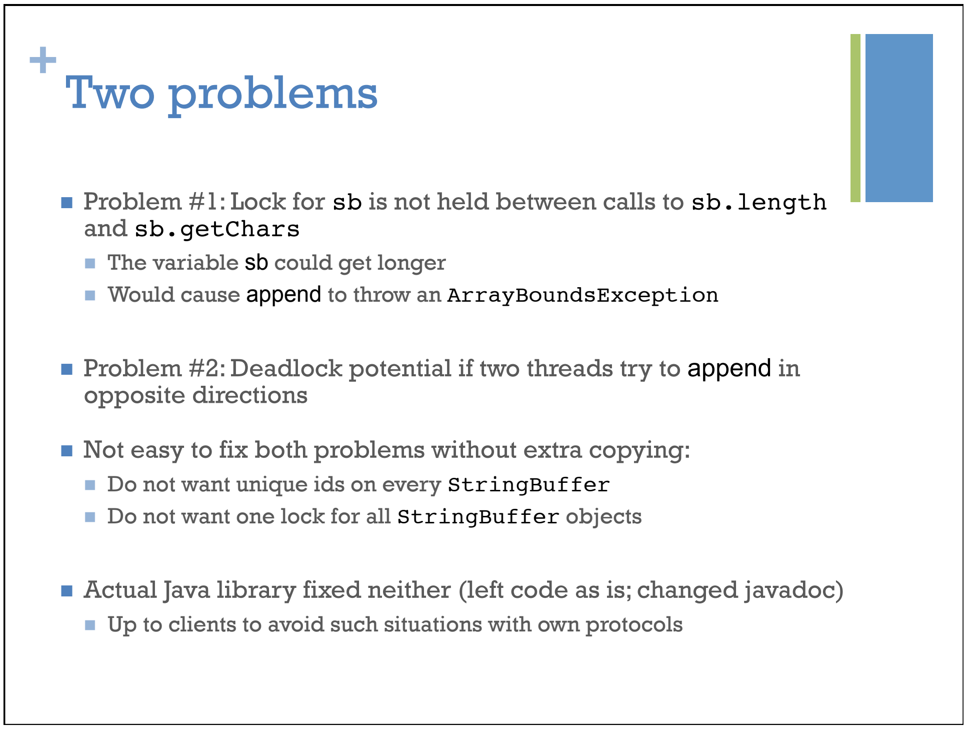
```
synchronized void transferTo(int amt, BankAccount a){
   if(this.acctNumber < a.acctNumber) {
      synchronized(this){
         synchronized(a) {
            this.withdraw(amt);
            a.deposit(amt);
         }
      }
   }else{
      synchronized(a) {
         synchronized(this) {
            this.withdraw(amt);
            a.deposit(amt);
         }
      }
   }
}
```



A Last Example

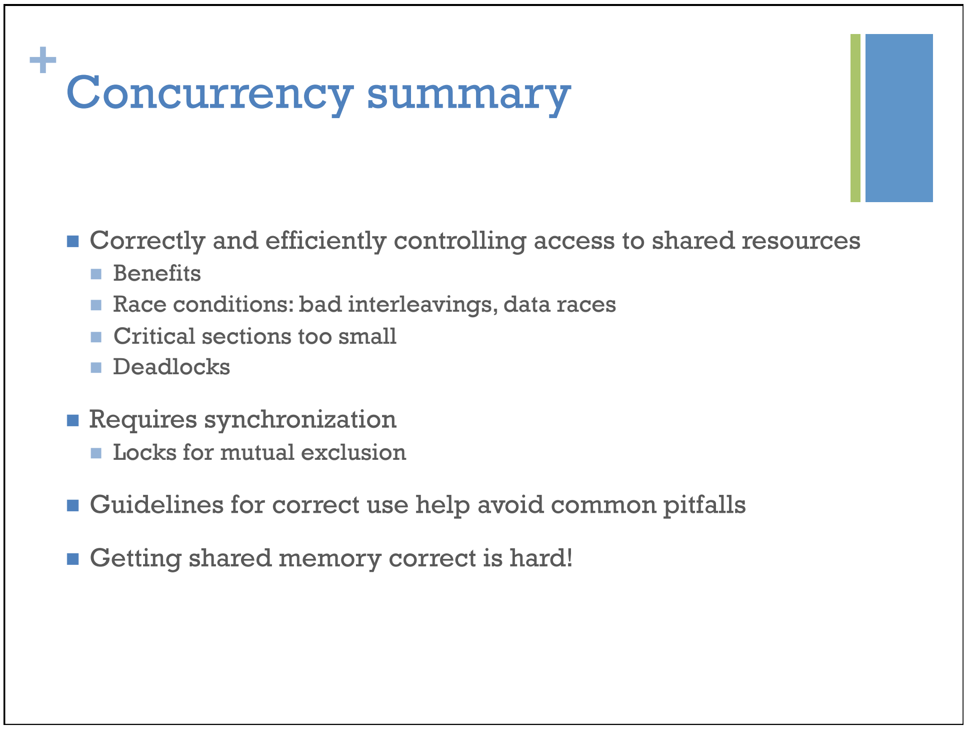
From the Java standard library

```
class StringBuffer { // a mutable String
  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` is not held between calls to `sb.length` and `sb.getChars`
  - The variable sb could get longer
  - Would cause append to throw an `ArrayBoundsException`
- Problem #2: Deadlock potential if two threads try to append in opposite directions
- 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 javadoc)
  - Up to clients to avoid such situations with own protocols



+ Concurrency summary

- Correctly and efficiently controlling access to shared resources
  - Benefits
  - Race conditions: bad interleavings, data races
  - Critical sections too small
  - Deadlocks
- Requires synchronization
  - Locks for mutual exclusion
- Guidelines for correct use help avoid common pitfalls
- Getting shared memory correct is hard!