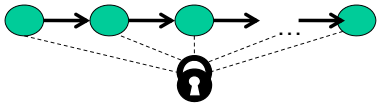
Lock granularity

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

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`?

Which makes implementing `resize` easier?
- How would you do it?

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

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*

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*

Motivating Deadlock Issues

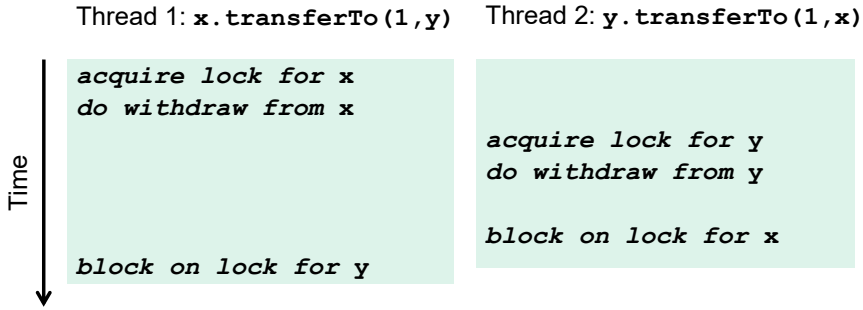
Consider a method to transfer money between bank accounts

```
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?

The Deadlock

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



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);
       }}
  }
}
```


Concurrency summary

- Concurrent programming allows multiple threads to access shared resources (e.g. hash table, work queue)
- Introduces new kinds of bugs:
  - 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
- Shared Memory model is not only approach, but other approaches (e.g., message passing) are not painless