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")

```java
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?

```java
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:

```java
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);
        }
    }
}
```
Leverage Libraries

• Use built-in libraries whenever possible
• 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
  – Very difficult to provide fine-grained synchronization without race conditions
  – Standard thread-safe libraries like ConcurrentHashMap written by world experts
Another Bank Operation

Consider transferring money:

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);
    }
}

What can go wrong?
Deadlock

\( x \) and \( y \) are two different accounts

Thread 1: \( x . \text{transferTo}(1, y) \)

Thread 2: \( y . \text{transferTo}(1, x) \)
Dining Philosopher’s Problem

- 5 Philosopher’s eating rice around a table
- one chopstick to the left and right of each
- first grab the one on your left, then on your right…
Deadlock = Cycles

- Multiple threads depending on each other in a cycle
  - T2 has lock that T1 needs
  - T3 has lock that T2 needs
  - T1 has lock that T3 needs

- Solution?
How to Fix Deadlock?

In Banking example

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);
    }
}
How to Fix Deadlock?

Separate withdraw from deposit

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);
    }
}

Problems?
Possible Solutions

1. `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 each pair of accounts allowing transfers between them
   - works, but sacrifices concurrent deposits/withdrawals

3. Give every bank-account a unique ID and always acquire locks in the same ID order
   - *Entire program* should obey this order to avoid cycles
Ordering Accounts

Transfer from bank account 5 to account 9

1. lock A5
2. lock A9
3. withdraw from A5
4. deposit to A9
Ordering Accounts

Transfer from bank account 5 to account 9

1. lock A5
2. lock A9
3. withdraw from A5
4. deposit to A9

Transfer from bank account 9 to account 5

1. lock
2. lock
3. withdraw from
4. deposit to
Ordering Accounts

Transfer from bank account 5 to account 9

1. lock A5
2. lock A9
3. withdraw from A5
4. deposit to A9

Transfer from bank account 9 to account 5

1. lock
2. lock
3. withdraw from A5
4. deposit to A9

No interleavings will produce deadlock!

- T1 cannot block on A9 until it has A5
- T2 cannot acquire A9 until it has A5
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);
                }
            }
        }
    }
}
Lock Ordering

• Useful in many situations
  – e.g., when moving an item from work queue A to B, need to acquire locks in a particular order

• Doesn’t always work
  – not all objects can be naturally ordered
  – Java StringBuffer append is subject to deadlocks
    ‣ thread 1: append string A onto string B
    ‣ thread 2: append string B onto string A
Locking a Hashtable

• Consider a hashtable with
  – many simultaneous lookup operations
  – rare insert operations

• What’s the right locking strategy?
Read vs. Write Locks

- Recall race conditions
  - two simultaneous write to same location
  - one write, one simultaneous read

- But two simultaneous reads OK

- Synchronize is too strict
  - blocks simultaneous reads
Readers/Writer Locks

A new synchronization ADT: The readers/writer lock

- A lock’s states fall into three categories:
  - “not held”
  - “held for writing” by one thread
  - “held for reading” by one or more threads

- **new**: make a new lock, initially “not held”
- **acquire_write**: block if currently “held for reading” or “held for writing”, else make “held for writing”
- **release_write**: make “not held”
- **acquire_read**: block if currently “held for writing”, else make/keep “held for reading” and increment *readers count*
- **release_read**: decrement *readers count* count, if 0, make “not held”
In Java

Java’s `synchronized` statement does not support readers/writer

Instead, library
`java.util.concurrent.locks.ReentrantReadWriteLock`

- Different interface: methods `readLock` and `writeLock` return objects that themselves have `lock` and `unlock` methods
Concurrency Summary

- Parallelism is powerful, but introduces new concurrency issues:
  - Data races
  - Interleaving
  - Deadlocks

- Requires synchronization
  - Locks for mutual exclusion

- Guidelines for correct use help avoid common pitfalls