CMSC 433

Design of concurrency abstractions
Release Tests Updated

• The release tests have been updated (Monday night)
• A variant of the last release test case was added
• The name of what had been the old last release test was changed to make it more descriptive
Designing and Building Concurrency Abstractions

• Let’s compare our project 3 with a similar project from a previous semester
  – project 4 from Spring 2005
  – *In this project, you will use threads, thread synchronization, and thread communication to write a simulation of cars, ships, and a draw bridge (a bridge that lifts so that boats can pass under it).*
Bridge simulation

• The bridge has a number of lanes
• There can be at most one car crossing per lane at any time.
• There can be at most one ship crossing under the bridge at any time
Ship/Car interaction

• Ships can only cross once the bridge is lifted, cars can only cross once the bridge is down. The bridge cannot be lifted while cars are crossing the bridge and it cannot be brought down while ships are crossing under the bridge.

• Once a ship arrives at the bridge, no cars can start crossing the bridge until the ship has finished crossing. However, the cars that are already crossing the bridge when the ship arrives should finish crossing before the bridge is lifted. In short, ships have priority over cars.
Questions

• Sound complicated?
• Sound familiar?

• What is the basic problem being addressed?
Find the basic problem

- Abstract the problem you need solving
- Remove irrelevant distractions
- Easier to test
- Easier to reuse
Connections?

• Connections between old project 4 and new project 3?
Implementing concurrency abstractions

• Concurrency is hard
• Make your implementation as simple as possible
  – but no simpler
What do you lock on?

• You can lock on whatever you want
• a synchronized method locks this
  – synchronized static methods lock on .class variable
• Be careful with inner classes
  – a synchronized method in inner class locks a different this than a synchronized method in outer class
Simplest possible design

- Just one object that all threads synchronize on
- Avoid confusion, create a special object used for nothing except locking
- Object lock = new Object()
When is this OK?

• More OK than you might think
• Only one lock means deadlock impossible
  – other issues simplified
• But can increase contention
  – time when threads have to block, waiting to acquire lock
Simple design, 2nd try

- Each operation only obtains locks on one object
- If operation X might need to block while operation Y runs, they should lock the same object
- If operation X and Y touch the same data, they should lock the same object
- Otherwise, they shouldn’t obtain lock on the same object
The most important (1.4) design pattern for concurrency

synchronized (lock) {
    while (!GoodToGo())
        lock.wait();
    // know good to go
    changeState();
    lock.notifyAll(); // if my changes might
    // make another thread
    // good to go
Notes

• can skip while loop and call to wait if you are always good to go

• can skip notifyAll if the changes you make to the state never make another thread good to go
OK, what is your state

- Once you understand the basic synchronized/wait/notify pattern
- The next thing you have to understand is your state.
- SimpleLock:
  - a boolean
- BlockingNonNegativeCounter:
  - an integer
State diagrams

- **Locked state transitions:**
  - From unlocked to locked
  - From locked to unlocked

- **Increment state transitions:**
  - From 0 to 1
  - From 1 to 2
  - From 2 to \( \ldots \)

- **Decrement state transitions:**
  - From 0 to 0
  - From 1 to 0
  - From 2 to 1

- **Labels:**
  - Inc: Increment
  - Dec: Decrement
public class SimpleLock {
    boolean isLocked = false;
    Object lock = new Object();

    public void lock() throws InterruptedException {
        synchronized (lock) {
            while (isLocked) lock.wait();
            isLocked = true;
        }
    }

    public void unlock() {
        synchronized (lock) {
            isLocked = false;
            lock.notifyAll();
        }
    }
}
public class BlockingNonNegativeCounter {
    volatile long value;
    Object lock = new Object();

    public long getValue() {
        return value;
    }

    public void increment() {
        synchronized (lock) {
            value++;
            lock.notifyAll();
        }
    }

    public void decrement() throws InterruptedException {
        synchronized (lock) {
            while (value == 0) lock.wait();
            value--;
        }
    }
}
Mini-project 3

- Two component locks: A and B
- One global/compound lock: G
- No priority for compound lock requests

- Going to devise the simplest possible solution to mini-3

- What is our state?
Reentrant lock

• No two threads may hold the lock at the same time
  – but a particular thread can lock it multiple times

• What is your state?
Read/Write Lock

• Can request a write/exclusive lock
  – no other thread can hold any kind of lock while a thread holds the write lock
• Can request a read lock
  – no other thread can hold a write lock
  – but other threads can hold read locks
• Not required to be reentrant
• What is your state?