Lecture 4
Issues in Synchronization
Synchronization?

• **Synchronization**: the maintenance of constraints on the order in which in different threads operations occur

• Another term: *concurrency control*

• Locks are used to enforce synchronization among threads accessing a shared object

• We will see other aspects of synchronization as well
Locks and Performance

• Suppose a class is correct (i.e. it satisfies its specification from a sequential perspective)
• One way to make it thread-safe: make all methods synchronized
  – E.g. from BoundedCounter example:
    • public void reset () { value = 0; } becomes
    • public synchronized void reset () { value = 0; }
  – This makes every method atomic and locks every field during a method call
  – If there are no public fields, then no threads can see data violating invariants!
Locks and Performance (2)

• Problem: performance!
  – If every method is synchronized, only one method at a time can execute
  – Some methods can run in parallel however: for example (BoundedCounter)
    • current()
    • isMaxed()
  – If every method is synchronized, then this is not possible
  – On the other hand, both of the above methods should only access consistent data

• BoundedCounter methods are small, so aggressive locking is not so problematic

• For classes with large, time-consuming methods, this creates performance bottlenecks
Designing Locking Protocols

• Locking protocol: how you do locking in order to balance thread-safety, performance
• Making every method synchronized is one example
  – Each data value is “guarded by” this
  – Each method must acquire implicit lock on this to execute
• Another approach
  – Associate same lock with all fields mentioned in an invariant
  – Lock on these, rather than using this
  – Idea
    • Accessing a field should only be done when values are consistent with invariants
    • Using same variable to lock accesses to variables mentioned in same invariants enforces this
    • Fields that are not involved in the invariant can be accessed without disturbing the invariant
public class ColoredMutableLine {

    //@Invariant:  p1 and p2 must be different points.

    Object InvLockP1P2 = new Object ();
    private Point p1;  // Guarded by lock LockP1P2Inv
    private Point p2;  // Guarded by lock LockP1P2Inv

    //@Invariant:  none

    Object InvLockColor = new Object ();
    private int color; // Guarded by lock LockColor

    public Point getP1() { synchronized (InvLockP1P2) { return p1; } }

    public void setP1(Point p1) {
        synchronized (InvLockP1P2) {
            if (!p2.equals(p1)) this.p1 = p1;
            else throw new IllegalArgumentException ("Illegal argument to setP1 : " + p1.toString() + " same as second point");
        }
    }

    public int getColor() { synchronized (InvLockColor) { return color; } }

    public void setColor(int color) {
        synchronized (InvLockColor) { this.color = color; }
    }
}

Example: ColoredMutableLine
Locks and Overlapping Invariants

• What if invariants are:
  – Invariant 1: left ≤ middle
  – Invariant 2: middle ≤ right

• One lock for left and middle, another for middle and right?
  – Not advisable
    • Rule of thumb: each variable should be guarded by one lock
    • This approach would have two locks for middle
  – Better approach: one lock for left, middle and right
    • Methods accessing any of these variables must first acquire this lock
    • This ensures preservation of invariants
A Peril of Locking

What can following code do?

- RunnableAB.java
  public class RunnableAB implements Runnable {

    private Object firstLock; private Object secondLock;

    public RunnableAB (Object a, Object b) { firstLock = a; secondLock = b; }

    public void run () {
      synchronized (firstLock) { synchronized (secondLock) {
        System.out.println ("AB succeeds");
      }}
    }
  }

- RunnableBA.java same, except that first, second locks switched

- DeadlockPossible.java
  public class DeadlockPossible {

    public static void main(String[] args) {
      Object lockA = new Object ();  Object lockB = new Object ();

      Thread t1 = new Thread (new RunnableAB (lockA, lockB));
      Thread t2 = new Thread (new RunnableBA (lockA, lockB));

      t1.start ();
      t2.start ();
    }
  }
Answer: It Can Deadlock

• Consider this sequence
  – AB thread acquires lockA
  – BA thread acquires lockB
  – AB then tries to acquire lockB
  – BA tries to acquire lockA

• Neither thread can acquire the second lock it needs

• The threads both block, and the system “freezes”!
Defining Deadlock

• A set of threads is **deadlocked** if each thread is waiting for a resource (lock) that is held by some other thread in the set

• In the preceding example, the sequence of events leads to Thread AB and Thread B deadlocking
  – Thread AB is waiting for lockB, which is held by BA
  – Thread BA is waiting for lockA, which is held by AB

• Note: a system can sometimes deadlock and sometimes not!
  – Example system has this property
  – Deadlocking behavior is scheduler-dependent
Detecting Deadlock

• Difficult!
  – When threads are deadlocked, nothing is happening
  – When threads are not scheduled, nothing is also happening
  – How can you tell the difference?

• There is an approach based on graphs that can be used
Conditions Necessary for Deadlock

1. Mutual exclusion
   *There is at least one non-sharable resource (e.g. lock)*

2. Hold-and-wait
   *Threads already holding resources may request other resources held by other threads*

3. Non-preemptability
   *No resource held by a thread may be forcibly removed from its control*

4. Circular waiting
   *There is a circular chain of dependencies consisting of one thread waiting for a resource held by another thread*
Circular Waiting and Waits-For Graphs

- Circular waiting can be depicted using graphs (i.e. diagrams)
  - Circles: threads
  - Boxes: locks
- There is an arrow from a lock to a thread if the thread holds the lock
- There is an arrow from a thread to a lock if the thread is waiting for the lock
Waits-For Graph:  Cycle = Deadlock!

• Thread AB has lockA
• Thread AB is waiting for lockB
• Thread BA has lockB
• Thread BA is waiting for lock A
Preventing Deadlock

• Impose an order on the locks
• Every thread that needs multiple locks must acquire them in the order specified
• Example revisited
  – Order could be lockA < lockB, meaning that if you need lockA and lockB, you must acquire lockA first, then lockB
  – Currently, AB follows this order, but BA does not
  – If BA did follow this order, deadlock could not occur, because thread that acquires A first would be guaranteed to acquire B!