50.003: Elements of Software Construction

Week 6

Concurrency: Design and Implementation

Thread Safety

- If an object of type A is to be shared by multiple threads, A must be thread safe.
- "A class is thread-safe if no set of operations performs sequentially or concurrently on instances of a thread-safe class can cause an instance to be in an invalid state***." (*Java Concurrency in Practice, Chapter 2)*

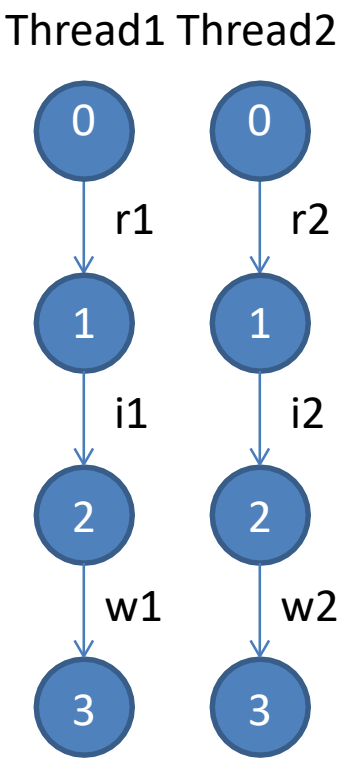
*** whether a state is valid or not is defined by the specification

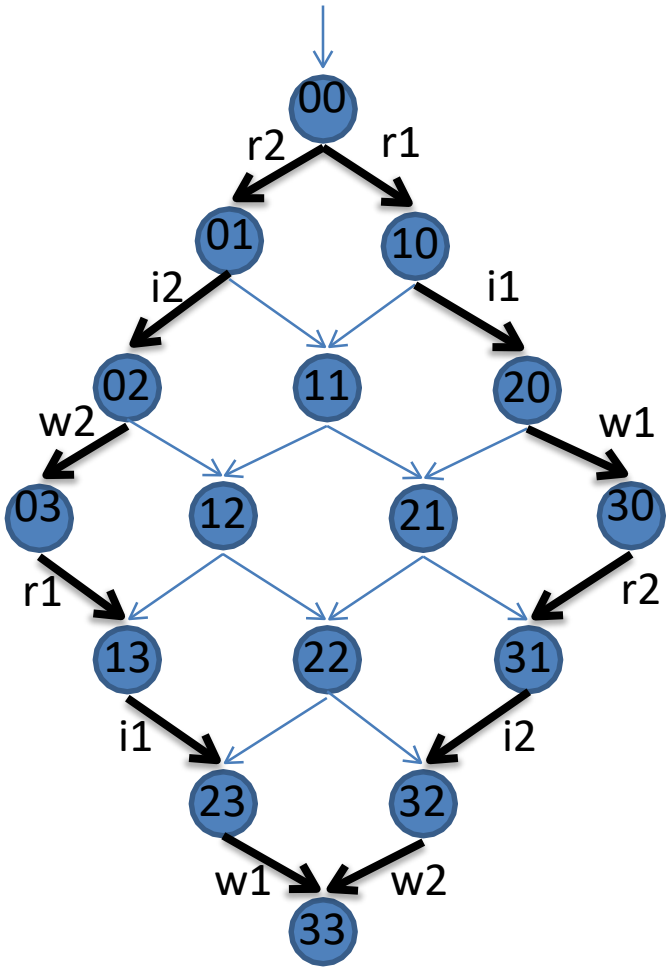
Specification

- What is a “valid state” depends on the specification on the class, which can be captured using:
  - class invariants
  - pre-condition/post-condition
  - assertions
- Example: Stack.java

Why Is Race Condition Often Bad?

Post-condition: count' = count+2





Remedy: Visibility Issue

Volatile Variables

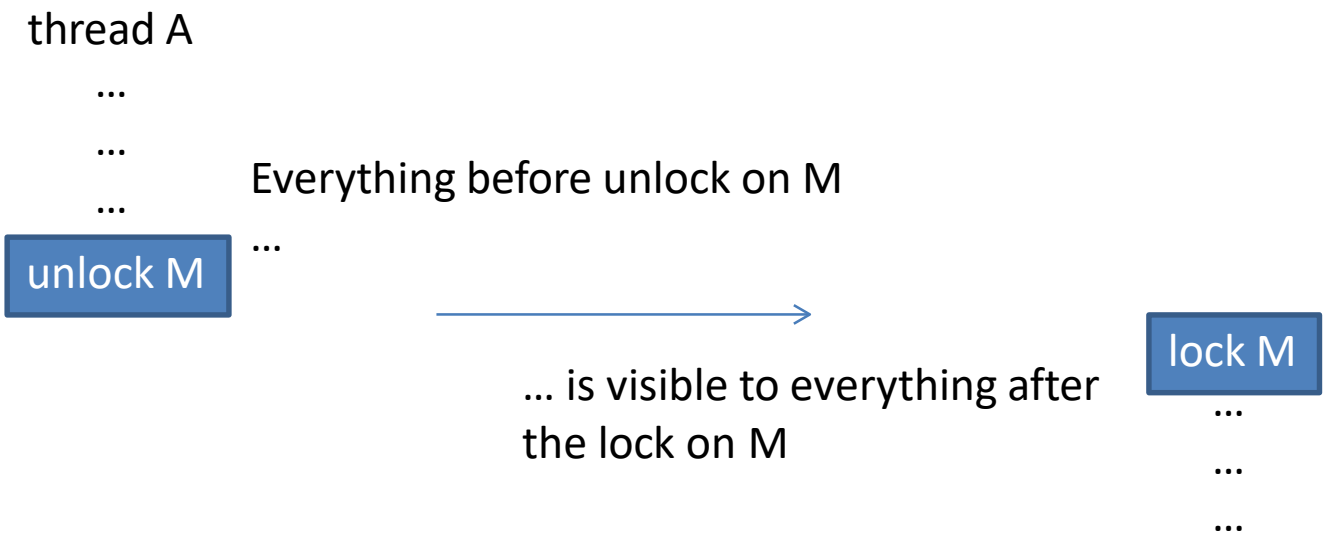
An update to a volatile variable is propagated predictably to other threads.

```
private static volatile boolean ready;
…
while (!ready) {
        Thread.yield();
}
…
```

Example: NoVisibility.java

Remedy: Visibility Issue

- Visibility guarantees for synchronization



- Always use the proper synchronization whenever data is shared across threads.

Example

```
public class MutableInteger {
    private int value;

    public int get() { return value; }
    public void set(int value) { this.value = value;}
}
```

What is the problem here?

```
public class MutableInteger {
    private int value;

    public synchronized int get() { return value; }
    public synchronized void set(int value) { this.value = value;}
}
```

Locking and Visibility

Locking is not just about mutual exclusion; it is also about memory visibility. To ensure that all threads see the most up-to-date values of shared mutable variables, the reading and writing threads must synchronize on a common lock.

Remedy: Execution Ordering

Thread 1	Thread 2
1: r2 = A;	3: r1 = B;
2: B = 1;	4: A = 2;

- If you do want unexpected execution ordering of statements across threads, apply locking.

Requirements

Multi-threaded programs have at least the following additional requirements.

- Thread-safety
- ~~No visibility issue~~
- ~~No execution ordering problem~~
- No deadlocks
- Efficiency!

How do we make sure our multi-threaded program satisfies these requirements?

Remedy for Deadlock

A program that never acquires more than one lock at a time cannot experience lock-ordering deadlocks.

Strive to use open calls (calling a method with no locks held) throughout your program

Sample program: DLExampleFixed.java

```
public synchronized void setLocation(Point location) {
       this.location = location;
       if (location.equals(destination))
           dispatcher.notifyAvailable(this);
   }

   public synchronized Point getDestination() {
       return destination;
   }
```

```
  // FIXED
  public void setLocation(Point location) {
          boolean reachedDestination;
          synchronized (this) {
              this.location = location;
              reachedDestination = location.equals(destination);

          }

          if (reachedDestination) {
              dispatcher.notifyAvailable(this);
          }
      }
```

```
public synchronized Image getImage() {
        Image image = new Image();
        for (Taxi t : taxis)
            image.drawMarker(t.getLocation());
        return image;
    }
```

```
  // FIXED
  public Image getImage() {
          Set<TaxiFixed> copy;

          synchronized (this) {
                  copy = new HashSet<TaxiFixed>(taxis);
          }

           Image image = new Image();
           for (TaxiFixed t : copy)
               image.drawMarker(t.getLocation());
           return image;
       }
```

Remedy for Deadlock

A program will be free of lock-ordering deadlocks if all threads acquire the locks they need in a fixed global order.

- Is this deadlocking? Thread A locks a, b, c, d, e in the sequence and thread B locks c, f, e.
- Is this deadlocking? Thread A locks a, b, c, d, e in the sequence and thread B locks e, f, c.

```
public void transferMoney (Account from, Account to) {
    synchronized (from) {
        synchronized (to) {
            ……
        }
    }
}
```

Click here for a sample program: TransferFixed.java

Avoid Deadlocks

- Use the timed tryLock feature of the explicit Lock class instead of intrinsic locking.

```
final ReentrantLock reentrantLock = new ReentrantLock();
boolean flag = reentrantLock.tryLock(1000, TimeUnit.MILLISECONDS);
if (flag) {
    try {
       System.out.println(“Lock acquired.");
       System.out.println("Performing task...");
    } finally {
       System.out.println(“Lock released.");
       reentrantLock.unlock();
    }
    break;
}
else {
       System.out.println(“Lock not available. Retry again");
}
```

Cohort Exercise 2

- Fix DiningPhil.java by making it deadlock-free, in two different ways.
- Justify why your code is deadlock-free.

Requirements

Multi-threaded programs have at least the following additional requirements.

- Thread-safety
- ~~No visibility issue~~
- ~~No execution ordering problem~~
- ~~No deadlocks~~
- Efficiency!

How do we make sure our multi-threaded program satisfies these requirements?

No Sharing => Thread Safety

Sharing Constants Only
=> Thread Safety

Building Thread-Safe Classes

Otherwise, make sure every shared variable is of a type which is thread-safe.

How to build thread-safe classes?

- From scratch
- By extending the class
- Through client-side locking
- Through composition

Design a Thread-Safe Class

The design process for a thread-safe class should include these four steps:

1. Identify the variables that form the object's state;
2. Identify the requirements (e.g., invariants, post-conditions) that constrain the state variables;
3. Establish a policy for managing concurrent access to the objects state.
4. Implement the policy.

Step 1: Identifying States

- An object's state includes all of its mutable variables
  - If they are all of primitive type, the fields comprise the entire state.
  - If the object has fields that are references to other objects, its state will encompass fields from those as well.

Example: MyStack.java

Tips

- The smaller this state space, the easier it is to reason about.
  - Use “final” as long as you can.

```
public class MyStack {
        private final int maxSize;
        private long[] stackArray;
        private int top;

        …
}
```

Step 2: Identifying Requirements

- You cannot ensure thread safety without understanding an object's specification. Constraints on the valid values or state transitions for state variables can create atomicity and encapsulation requirements.

Example: MyStackThreadSafe.java

Step 3: Designing Policy

- Update related state variables in a single atomic operation
- For each mutable variable that may be accessed by more than one thread, all assesses to that variable must be performed with the same lock held.
- Every shared, mutable variable should be guarded by exactly one lock. Make it clear to maintainers which lock that is.
- For every invariant that involves more than one variable, all the variables involved in that invariant must be guarded by the same lock.

Example: MyStackThreadSafe.java

Step 4: Implementing Policy

- Make sure every access of any variable is guarded by the lock according to the policy.
- Make sure access of the related variables in the same method is in synchronized block.
- Add waiting (and notify) to handle pre-conditions.

Example: MyStackThreadSafe.java

Cohort Exercise 3

Similarly identify the pre-condition/post-condition of other methods in MyStack.java and make the class thread-safe.

- Document what is your locking policy and why.
- Apply wait/notify whenever relevant.

Building Thread-Safe Classes

Otherwise, make sure every shared variable is of a type which is thread-safe.

How to build thread-safe classes?

- ✓ *From scratch*
- By extending the class
- Through client-side locking
- Through composition

Extending the Class

You build a thread-safe class through extending the class if

- there is a thread-safe class supporting almost all the operations your want;
- and you know the locking policy of the class (so that you can follow the same policy);
- and you don’t have the source code.

Example: BetterVector.java

The Problem of Private Locks

- Instead of guarding state variables by locking “this”, private locks can be used too.
- Example:

```
public class PrivateLock {
        private final Object myLock = new Object();
        //@GuardedBy(“myLock”)
        Widget widget;

        void someMethod() {
                synchronized(myLock) {
                        //Access or modify the state of widget
                }
        }
}
```

Private Locks are flexible and risky.

Extending the class

- More fragile than modifying the class directly, because the implementation of the synchronization policy is now distributed over multiple, separately maintained source files.

Example: ExtendedPair.java and ExtendedPairWrong.java

Programs must not use **instance locks** to protect **static shared data** because instance locks are ineffective when two or more instances of the class are created. Consequently, failure to use a **static lock object** leaves the shared state unprotected against concurrent access.

Client-side Locking

You can build a thread-safe class through client-side locking if

- We don’t have the source code or enough access to the internal state so that we can extend the class.
- You know the client’s locking policy (i.e., there are no private locks).

Client-side Locking

Example: Add a method addIfAbsent to Collections.synchronizedList

```
public class ListHelper<E> {
        public java.util.List<E> list =
                Collections.synchronizedList(new ArrayList<E>());
}
```

Question

- Is this thread safe?

```
public class ListHelper<E> {
        public java.util.List<E> list =
                Collections.synchronizedList(new ArrayList<E>());

        public synchronized boolean putIfAbsent(E x) {
                boolean absent = !list.contains(x);
                if (absent) {
                        list.add(x);
                }

                return absent;
        }
}
```

Should be synchronized (list)

Wrong! Correct example: ListHelper.java

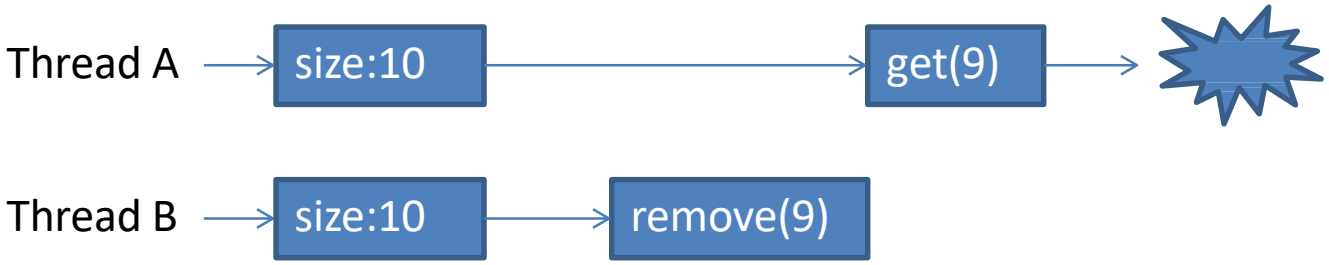
Cohort Exercise 4

Assume that multiple threads may call the static methods defined in FirstExample.java. Refer to the next slide for a particular problem of the implementation. Fix the program.

Hint

```
public static Object getLast(Vector list) {
      int lastIndex = list.size() - 1;
      return list.get(lastIndex);
}

public static void deleteLast(Vector list) {
      int lastIndex = list.size() - 1;
      list.remove(lastIndex);
}
```



Client-Side Locking

- Even more fragile than extending the class because it distributes the locking policy for a class into classes that are totally unrelated.
  - Modifying the class: keeps the locking code in the same class
  - Extending the class: distributes the locking code in the class hierarchy

Composition

- It is less fragile if list is accessed only through improvedList.
- It doesn’t care whether list is thread-safe or not.
- It adds performance penalty due to the extra layer of synchronization.
- Copy all methods by original class and add a synchronized to the method name

Documenting

- Documentation is one of the most powerful tools for managing thread safety.
  - Document a class's thread safety guarantees for its clients; document its locking policy for its maintainers.

Summary

- It is the mutable state, stupid.
- Make fields final unless they need to be mutable.
- Immutable objects are automatically thread-safe.
- Encapsulation makes managing complexity practical.
- Guard each mutable variable with a lock.
- Guard all participants of an invariant with the same lock.
- Hold locks for the duration of compound actions.
- Any program that accesses a mutable from multiple threads without synchronization is a broken program.
- Never rely on clever reasoning on why you do not need. synchronization in those situations.
- Include thread safety into the design process or explicitly document that your class is (intentionally) not thread-safe.
- Document your synchronization policy.

The above methods assume that the only way to access the state in the class is through calls of visible methods.

Instance Confinement

Is the following class thread-safe?

```
public class MyStack {
        private final int maxSize;
        private long[] stackArray;
        public int top;
        //invariant: top < stackArray.length && top >= -1
        …
        //all methods are synchronized.
}
```

Encapsulating Data

- Identify the intended scope of objects
  - To a class instance (e.g., a private data field)
  - To a lexical scope (e.g., a local variable)
  - To a thread (e.g., an object which is not supposed to be shared across threads)
- Instance confinement is one of the easiest ways to build thread-safe classes

Encapsulating Data

- Make sure the objects don't escape their scope
- Bad Example 1:

```
public class MyStack {
        private final int maxSize;
        public long[] stackArray;
        private int top;
        //top < stackArray.length && top >= -1
        …
}
```

Encapsulating Data

- Make sure the objects don't escape their scope
- Bad Example 2:

```
public class MyClass {
        //guard by "this"
        private String[] states = new String[]{…};
        private int size = states.length;
        public synchronized String[] getStates() {
                return states;
        }
}
```

Example: PersonSet.java

Cohort Exercise 6

- Assume a taxi tracking system which tracks taxis in Singapore. The updater threads would modify taxi locations and the view thread would fetch the locations of the taxis and display them. Examine Tracker.java (shared by the updater thread and the view thread) and make it thread-safe. Ensure instance confinement.

```
// Yes, a reference to the locations is created,
// the locations can be modified. Do a deep copy of locations instead
public TrackerSafe(Map<String, MutablePoint> locations) {
//        this.locations = locations;
        this.locations = copy(locations);
    }

// Yes, a reference to the MutablePoint object is returned
// User can thus modify the attributes of the MutablePoint
// return a new instance of MutablePoint & synchronise so
// locations set will be visible
    public synchronized MutablePoint getLocation (String id) {
        MutablePoint loc = locations.get(id);
//      return loc;
        MutablePoint newPoint = new MutablePoint(loc.x,loc.y);
        return newPoint;
    }
```

Publishing State Variables

- When is it safe to publish state variables?
  - If a state variable is thread-safe, does not participate in any invariant that constrain its value, and has no prohibited state transitions for any of its operations, then it can be safely published.
  - It still might not be a good idea, since publishing mutable variables constrains future development and opportunities for sub-classing.

Example: PublishingTracker.java

- Assuming that there is no additional constraints on vehicle locations, other than that they must be this given pair.
- PublishingTracker delegates its thread-safety to ConcurrentHashMap and Point.

Delegating Thread Safety

- If a class is composed to multiple independent thread-safe state variables, then it can delegate thread safety to the underlying state variables.
- If a class has invariants the relate its state variables, then delegation may not work.