Lecture 4
Thread Safety
Thread Anomalies

• Scheduler determines when threads execute
  – Thread computation can be interleaved on a single processor, or
  – Threads computations can be on different processors, or
  – Some combination of both

• Programmer can have some influence via yield(), setPriority(), etc.

• But most decisions are outside user control, leading to possibilities for
  – Nondeterminism
  – Interference: threads overwrite each other’s work
Anomaly from Lecture 1

- IncThread.java

  ```java
  public class IncThread implements Runnable {

  private static int shared = 0;  // Shared variable
  private String name = "";      // Name of thread

  IncThread (String name) { this.name = name; }

  public void run() {
    int myShared = shared;
    System.out.println (name + " read shared = " + myShared);
    myShared++;
    shared = myShared;
    System.out.println(name + " assigned to shared: " + myShared);
  }

  }
  ```

- Main thread created two instances, t1 and t2, and started both
Two Threads

- Different schedules can leave shared = 2, shared = 1
- This is an example of a *data race*

```
myShared = shared;
print myShared;
myShared++;
shared = myShared;
print myShared;
```

```
myShared = shared;
print myShared;
myShared++;
shared = myShared;
print myShared;
```
Data Races and Race Conditions

• A *data race* occurs when the same memory location can be accessed simultaneously by two threads, with at least one of accesses a write.

• They “seem bad” ... but why?
  – In previous example, if it does not matter if shared is 1 or 2, then is there an error?
  – On the other hand, if shared should only be 2, then there is an error

• A *race condition* occurs when a program’s correctness depends on scheduling decisions
  – If the correct outcome of the previous example is shared = 2, then the data race induces a race condition
  – If the correct outcome is shared = 1 or shared = 2, then there is no race condition!
Correctness?

- Definition of race condition mentions program correctness
- We will adopt a class-based view:
  A class is correct if it satisfies its specification
- So what is a “class specification”?
Class Specifications

• Classes are used to define objects
• Classes contain static members
• Objects contain instance members
• Some members are fields, while others are methods
• Classes generally enforce consistency constraints on static, instance members
  – Field values should be “consistent”
  – Methods should preserve consistency, compute the right thing
Example: Line Class

- **Point.java**
  ```java
  public class Point {
    private final double x; private final double y;

    Point(int x, int y) { this.x = x; this.y = y; }

    double getX() { return x; }
    double getY() { return y; }
  }
  ```

- **Line.java**
  ```java
  public class Line {
    private Point p1; private Point p2;

    Line(Point p1, Point p2) { this.p1 = p1; this.p2 = p2; }

    public double slope() {
      return ((p1.getY() - p2.getY()) / (p1.getX() + p2.getX()));
    }
  }
  ```
Notions of Consistency for Line?

- Would like to know that points are different!
- *Invariants* capture notion of consistency
  - Invariants describe properties that must always hold among instance variables
  - They reflect relationships you can “rely on”
- Here is an invariant for Line:
  p1 and p2 must be different points
- Is Line class correct? No!
  - Constructor does not check that points are different
  - So constructor can construct objects violating invariant
Corrected Line Class

- CorrectLine.java – change constructor to:

```java
CorrectLine(Point p1, Point p2) throws IllegalArgumentException {
    if ((p1.getX() != p2.getX()) ||
        (p1.getY() != p2.getY())) {
        this.p1 = p1;
        this.p2 = p2;
    } else {
        throw new IllegalArgumentException (
            "Points to Line constructor must differ:  "
            + p1.toString() + "given twice.");
    }
}
```

- Note that when invariant violation is detected, no updating is performed, and exception is thrown!
Is the CorrectLine Class Correct?

• Some would say yes …
• … and yet there is one more issue: division by zero!
  – If \( p_1, p_2 \) have the same \( x \)-value, then the slope calculation involves dividing by 0
  – This can throw a run-time exception!
• This is not a consistency issue among fields, but instead a property of methods.
Class Specifications: Preconditions / Postconditions / Exception Conditions

• To specify the behavior of methods, need
  – Preconditions: what should hold of inputs, fields in order to ensure correct termination
  – Postconditions: what will hold when method exits normally
  – Exceptions: what happens when precondition violated

• In case of slope method ...
  – Specification should indicate that if points form a vertical line, then method will throw an exception; otherwise, slope is returned
  – Header for method should be changed to reflect this
Corrected slope() Method

• CorrectedLine.java

    // Precondition: p1, p2 do not form vertical line
    // Postcondition: return slope of line thru p1, p2
    // Exception: if p1, p2 form vertical line, throw
    //    ArithmeticException

    public double slope() throws ArithmeticException {
        return ((p1.getY() - p2.getY()) / (p1.getX() + p2.getX()));
    }
Class Specifications

• Invariants on fields
• Preconditions / postconditions / exceptions for all methods!
  – Put this in documentation
  – Ongoing research ("formal methods") on better support for this
  – This specification methodology is sometimes called design-by-contract
Class Correctness

• When is a class correct with respect to a specification?
  – The fields always satisfy the invariant (except when a method is in the middle of executing)
  – Each method produces results consistent with the postcondition when started with inputs / field values satisfying the precondition
  – Each method produces results consistent with the exception condition when started with inputs / field values violating the precondition

• The Line class is not correct for the given specification, while CorrectLine is!
Establishing Correctness in the Sequential Case

• Check that each constructor returns an object satisfying the invariant
• Check that each method leaves the invariant true if it starts with the invariant true
• Check preconditions / postcondition / exceptions
• Works because of validity of procedural abstraction!
  – Method call can be viewed as one *atomic* operation that is equivalent to executing body of method
  – So analyzing correctness can be done on a method-by-method basis
Problems with Threads

• Even if a class is correct with respect to a specification, threads can break invariants!
• This happens because:
  – A class can be correct even though methods might break the invariants in the middle of their execution
    Methods only have to make sure the invariants hold when they terminate.
  – Concurrency breaks procedural abstraction!
    • One thread can see the intermediate results of another thread’s execution
    • If the second thread is in the middle of a method call, the class’s invariants might not be true
    • The first thread then gets an inconsistent view of the corresponding object
Example: IncThread Revisited

- **IncThread.java**

  ```java
  ... 
  private static int shared = 0; // Shared variable 
  ...
  public void run() {
    int myShared = shared;
    myShared++;
    shared = myShared;
  }
  ```

- **Specification**
  - Invariant: shared records the number of times run() has been invoked
  - Precondition / postcondition / exception for run(): no requirements

- **IncThread is correct (sequentially)!**
  - Initially, invariant is true, since shared == 0
  - run() increments shared, so invariant is true when run() finishes if it is true when run() starts

- **There are erroneous runs when there are multiple threads!**
  - Until run() increments shared invariant is not true
  - Another thread can then read an inconsistent value of shared!
Thread Safety

A correct class is *thread-safe* if every execution of any threaded application using the class preserves the specification’s invariants and method specifications

- Thread safety only makes sense if you have a class specification!
- This fact is crucial but often overlooked
Example Re-revisited

• Suppose IncThread invariant is changed to:
  The value of \textit{shared} is \leq \text{the number of times } \textit{run()} \text{ is executed}

• Then IncThread is thread-safe!
  – Every value any thread might read of \textit{shared} is \leq \text{the number of times } \textit{run()} \text{ has been invoked}
  – Every thread increments \textit{shared}
  – Even though there is a data race, the class can be used as is in a threaded application, \textit{for this specification}

• Again: thread-safety is a property of a class \textit{and its specification}, not just of a class
Recap

• A class can be correct with respect to its specification and still not be thread-safe

• Why?
  – The methods in a correct class will preserve the specifications invariants before and after each executes
  – During execution of a method, the invariants might not be true
  – In a multi-threaded application, another thread might see this inconsistent state of an object, since procedural abstraction is violated!

• Implication: if a class is not thread-safe, it cannot be counted on to be correct in a multi-threaded execution
Fixing Thread Safety Problems

• Thread-safety is guaranteed for immutable objects
  – In immutable objects, the fields never change after construction
  – So if the fields of an object satisfy an invariant after it is built, it will never violate the invariant

• Rule of thumb: when feasible, use immutable objects
Implementing Points

• Immutable: Point.class
  
  ```java
  public class Point {
      private final double x;
      private final double y;
      
      Point(double x, double y) { this.x = x; this.y = y; }
  }
  
  For any specification of Point, if Point is correct then it is thread-safe!
  
  • Mutable: MutablePoint.class
  
  ```

  ```java
  public class MutablePoint {
      private double x;
      private double y;
      
      MutablePoint(double x, double y) { this.x = x; this.y = y; }
  }
  
  Depending on other operations, specification, this class may not be thread safe (e.g. if there are setters as well as getters)
  ```
Fixing Thread-Safety Problems: Locks

• Thread-safety problems are often related to methods inducing invariant errors while “in flight”
  – The invariant errors are fixed before the method terminates
  – If another thread sees this intermediate erroneous data, it can use it without realizing it.

• The issue: procedural abstraction
  – We would like to think of method calls as atomic, i.e. as either not having started or having finished, like single machine instructions
  – This perspective is valid in a sequential program
  – It is not in a multi-threaded program

• A solution: use locks to give illusion of atomicity!
Lock Fundamentals

- Examples of a **concurrency-control** primitive
  - As the name suggests, concurrency-control primitives are intended to control concurrency!
  - The idea: eliminate the possibility of concurrency while critical operations are taking place

- A lock is a data structure
  - Two states: *locked, unlocked*
  - Two operations: **acquire, release**
    - acquire: block execution until the state of the lock is unlocked, then set state to locked.
    - release: set status of lock to unlocked
  - Both operations are *atomic*
  - Variations:
    - Releasing a lock whose status is unlocked may or may not throw an exception
    - Some locks have more states (e.g. *read-locked*)
Using Locks to Fix Thread-Safety Issues

• Idea
  – Associate lock(s) with classes
  – Methods must acquire appropriate locks before performing internal operations that may violate invariants
  – Methods release locks when invariant is restored

• This ensures that multiple threads cannot see intermediate changes that methods make to fields during execution!
Locks in Java

• Several types
  – Intrinsic / monitor locks
  – Various classes whose objects are locks

• We will first study intrinsic / monitor locks (both terms are used)
Intrinsic / Monitor Locks

• Every object in Java has a lock associated with it, called the *monitor (lock)* or *intrinsic lock*

• No explicit acquire / release operations; rather the state of an intrinsic lock is modified using *synchronized* blocks
  
  – **Basic form:**
    
    ```java
    synchronized (obj) {
      statements
    }
    ```
  
  – **Semantics**
    
    • Acquire intrinsic lock of obj
    • Execute statements
    • Release intrinsic lock of obj when block exits (terminates, throws an exception, breaks, etc.)
Fixing IncRace.java

- **SyncIncThread.java**
  
  ```java
  public class SyncIncThread implements Runnable {

      private static int shared = 0;
      static Object lock = new Object ();    // Lock must be static!

      ...
      public void run() {
          synchronized (lock) {
              int myShared = shared;
              myShared++;
              shared = myShared;
          }
      }
  }
  ```

- The specification invariant that `shared` is the number of invocations of `run()`.
- The class-wide object `lock` is used to “guard” the part of `run()` where the invariant is violated (i.e. where `shared` is not yet updated).
- When one thread is executing its synchronized block, all other threads are waiting outside theirs.
- After run updates `shared` the invariant has been restored, and the lock can be released.
Synchronized Instance Methods

- In many cases we want entire methods to execute atomically
- Java provides a short-hand for this, by allowing methods to be declared `synchronized`
  - E.g.
    ```java
    public synchronized void setP1 (Point p1) {
        this.p1 = p1;
    }
    ```
  - This is an abbreviation for the following, since the method is an instance method
    ```java
    public boolean setP2(Point p2) {
        synchronized (this) {
            this.p2 = p2;
        }
    }
    ```
Synchronized Static Methods

• Static (class) methods may also be synchronized
  – For example, could add following method to `SyncIncThread`
    ```java
    public synchronized static void incShared () {
        ++shared;
    }
    ```
  – What object’s intrinsic lock is used in this case?
  – Answer: the class object associated with the relevant class!
  – In this case, here is equivalent code:
    ```java
    public static void altIncShared() {
        synchronized (SyncIncThread.class) {
            ++shared;
        }
    }
    ```
Reentrant Locking

- Intrinsic locks are reentrant!
  - If a thread acquires an intrinsic lock, it can acquire it again without blocking
  - A thread with multiple acquisitions on an intrinsic lock frees it only when the number of releases equals the number of acquisitions
- Huh?
  - Consider following code used to do atomic updating of a bounded counter
    ```java
    public synchronized boolean isMaxed() {
        return (value == upperBound);
    }
    
    public synchronized void inc () {
        if (!isMaxed()) ++inc;
    }
    
    Without reentrant locking, every call to inc() would block forever!
Example: Bounded Counter Class

• BoundedCounter.java: a correct, but not thread-safe class.

• How do we make it thread safe?
public class BoundedCounter {
    private int value = 0;
    private int upperBound = 0;

    //INVENANT: in all instances 0 <= value <= upperBound

    //Precondition: argument must be >= 0
    //Postcondition: object created
    //Exception: If argument < 0, IllegalArgumentException thrown
    BoundedCounter(int upperBound) throws IllegalArgumentException {
        if (this.upperBound >= 0) this.upperBound = upperBound;
        else throw new IllegalArgumentException("Bad argument to BoundedCounter: " + upperBound + "; must be >= 0");
    }

    //Precondition: none
    //Postcondition: current value returned
    //Exception: none
    public int current() { return value; }

    public void reset() { value = 0; }

    public boolean isMaxed() { return (value == upperBound); }

    //Precondition: none
    //Postcondition: increment value if not maxed; otherwise, do nothing.
    //Exception: none
    public void inc() { if (!isMaxed()) ++value; }
}
Design Considerations

• Whose job is it to enforce correctness?
  – Class? Or User
  – In BoundedCounter.java, could have incremented inc as:
    ```java
    public void inc() { ++value; }
    ```
    • This would put burden on maintaining correctness on user
    • But it is more efficient
  – A better perspective
    • Class should enforce correctness
    • Class designer, though, can choose what notion of correctness is
    • In the inc example, invariant could be relaxed to say that only correctness criterion is 0 <= value

• A similar question: whose job is it to enforce thread safety
  – So far: we have said class
  – A common alternative: it is user’s job to implement correct synchronization (reason: performance!)
  – The “better perspective” comment applies here also!
    • Commit to a notion of correctness
    • Make class thread-safe with respect to that notion