CMSC 433 Fall 2013
Michael Hicks
(Some slides due to Rance Cleaveland)

Lecture 5
Synchronization and Visibility

Atomicity

- Atomic operations are uninterruptible
  - They have either not started, or have finished: there is no "middle"
  - Procedural abstraction: permits method calls to be viewed as atomic, even though they consist of multiple operations
  - Concurrency breaks procedural abstraction!
- What is guaranteed to be atomic in Java?
  - Reads, writes of non-64-bit primitive types
  - Reads, writes of references
- Thread-safety: use of locking to give illusion of atomicity to method calls *vis à vis* a class specification

Built-in Atomicity in Java

- Reads of non-64-bit variables (ints, chars, booleans, references)
- Writes to non-64-bit variables (ints, chars, booleans, references, etc.)
- Guarantee: if you read a non-64-bit variable, you will see a value that some thread actually wrote to it

64-bit Reads, Writes

- Not guaranteed to be atomic!
  - E.g. `double x = 1.0;`
    - x is a 64-bit variable
    - Java spec says a JVM can implement this as two 32-bit writes
    - If a thread reads this variable during a write operation to it, it can get 32 "stale" bits and 32 "fresh" bits (a value that no thread ever wrote)!
  - Other data type like this: `long`
- For safe reads, writes of these variables, need synchronization

Synchronization and Visibility

- Two aspects to an operation
  - Atomicity: does it have a "middle" that other threads can see?
  - Visibility: when is its effect perceived by other threads?
- Visibility is tricky

What Can Following Code Do? (from textbook)

```
public class NoVisibilityAlt {
  private static boolean ready;
  private static int number;

  private static class ReaderThread
    extends Thread {
    public void run () {
      while (!ready)
        Thread.yield ();
      System.out.println (number);
    }
  }
  public static void main(…) {
    new ReaderThread ().start ();
    number = 42;
    ready = true;
  }
}
```

- Most of the time it prints 42
- It could print 0
- It could even never terminate!
- Why?
  - Assignments to number, ready are atomic
  - However, visibility is not guaranteed
    - Java language specification lets compilers reorder statements, use caches, etc.
    - So while `number = 42` is atomic, the operation's effect may not be visible until after thread executes `println`!
    - In this case, previous stale value of number is what thread sees

Dealing with Visibility: `volatile`

- Previous example highlights visibility anomalies in Java
  - Java language spec allows (unrelated) operations to be reordered, so long as sequential consistency is preserved

    e.g.

    ```
    new ReaderThread ().start ();
    ready = true;
    number = 42;
    ```

    Assignments to number, ready can be reordered because they are unrelated
  - This can wreak havoc with threaded applications
- Some visibility problems can be fixed by declaring variables to be volatile
  - Declaring variables volatile indicates they are shared, and operations should not be reordered
  - E.g.

    ```
    private static volatile boolean ready;
    private static volatile int number;
    ```
  - Ensures that assignment to number occurs before ready is made true, and that there is no delay in thread seeing truth of ready
- Volatility does not make non-reads, writes atomic, however! It just affects visibility of atomic operations

Visibility and Locking (1/3)

- Locking also fixes visibility problems!
- Consider following fragment from synchronized BoundedCounter class:

```
public synchronized int current () {
     return value;
}
…
public synchronized void inc () {
     if (!isMaxed()) ++value;
}
```

- Further suppose a class implementing threads that increment a counter:

```
public class BoundedCounterIncThread implements Runnable {
  private BoundedCounter counter;
  BoundedCounterIncThread (BoundedCounter c){ this.counter = c; }
  public void run () { counter.inc(); }
}
```

Visibility and Locking (2/3)

- What is output of following?

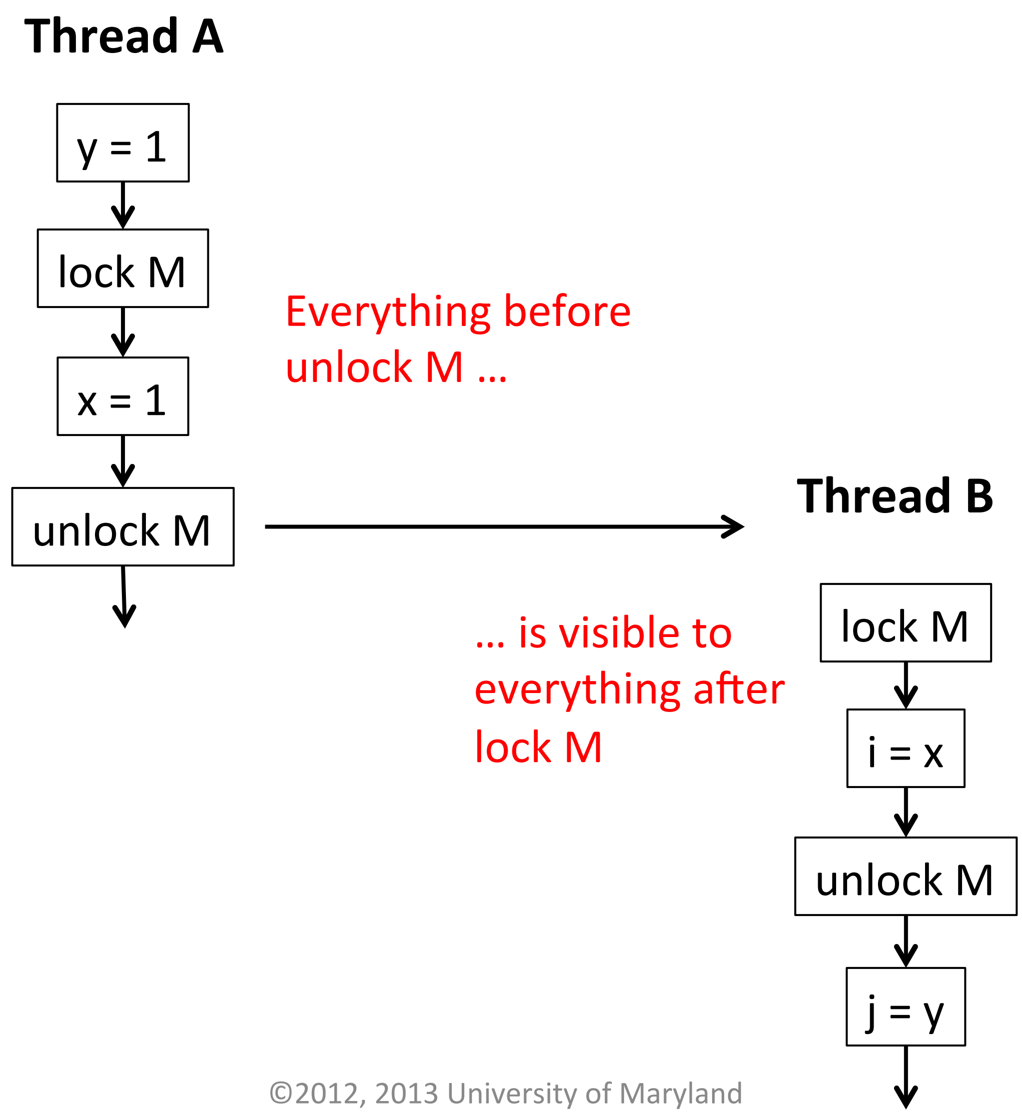
```
public static void main(String[] args) throws
InterruptedException {
  BoundedCounter c = new BoundedCounter (2);
  Thread t1 = new Thread (new
BoundedCounterIncThread (c));
  Thread t2 = new Thread (new
BoundedCounterIncThread (c));
  t1.start();
  t2.start();
  t1.join();
  t2.join();
  System.out.println (c.current());
}
```

Visibility and Locking (3/3)

- Answer: 2
- Why?

  The results of the inc operations performed by one thread are visible to the other
- A general principle of Java
  - When a lock is released, operations guarded by the lock become visible to operations following the reacquisition of the same lock
  - In the previous example, the intrinsic lock of the BoundedCounter object c plays this role!

Locking and Visibility (from textbook)



Visibility in Detail

- The Java Memory Model (part of the Java Language Specification) defines precisely how visibility works
- Key notions
  - Event sequences
  - “happens-before”
- Intuitively: if an event happens before another, the effect of the first event is visible to the second

Event Sequences

- Event sequences record reads, writes to memory during execution of a program
- They also record "relevant synchronization events" (we'll see this later)
- Form of event: ⟨thread, event-spec⟩
  - Thread: name of thread in which event occurs
  - Event spec: four kinds for now
    - begin: event indicating start of thread
    - end: event indicating exit of thread
    - w, location, value: write of value to location
    - r, location, value: read of value from locations

Example

- Consider following sequential program

```
public static void main(String[] args)
{
  number = 42;
  ready = true;
}
```

- Here is an event sequence

⟨main, begin⟩
⟨main, w, number, 42⟩
⟨main, w, ready, true⟩
⟨main, end⟩

"as-if-serial-within-thread"

- What other event sequences are allowed?
- Java Memory Model specifies "as-if-serial-within-thread" restriction
  - Events can be reordered
  - Results however must remain consistent with "program order"

Example Revisited

- Remember sequential program

```
public static void main(String[] args) {
  number = 42;
  ready = true;
}
```

- Here is another event sequence

  ⟨main, begin⟩

  ⟨main, w, ready, true⟩

  ⟨main, w, number, 42⟩

  ⟨main, end⟩

  – Writes to ready, number reordered!
  – This is allowed because cumulative effect at end of thread is the same

Another Allowable Event Sequence

- Sequential program

```
public static void main(String[] args) {
  number = 42;
  ready = true;
}
```

- Here is another event sequence

  ⟨main, begin⟩
  ⟨main, w, temp, 42⟩
  ⟨main, w, number, temp⟩
  ⟨main, w, ready, true⟩
  ⟨main, end⟩

  – Temporary variable introduced for number update!
  – This reflects e.g. putting value in a register, then assigning register to memory location for number
  – This is also allowed because cumulative effect at end of thread is the same

What About Concurrency?

Now consider concurrent program

```
public static void main(String[] args) {
  new ReaderThread ().start ();
  number = 42;
  ready = true;
}
```

where ReaderThread is defined by:

```
private static class ReaderThread extends Thread {
  public void run () {
    while (!ready) Thread.yield ();
    System.out.println (number);
  }
}
```

What are the event sequences?

Adding to Event Sequences

- To accommodate start(), will add an event specification

  thread’, launch: launching of thread named thread’

- Threads performing start() will have launch events now

Lots of Event Sequences! Here are Two

1. 〈main, begin〉 〈main, launch, T_0〉 〈main, w, number, 42〉 〈main, w, ready, true〉 〈main, end〉 〈T_0, begin〉 〈T_0, r, ready, true〉 〈T_0, r, number, 42〉 〈T_0, end〉
2. 〈main, begin〉 〈main, launch, T_0〉 〈main, w, ready, true〉 〈T_0, begin〉 〈T_0, r, ready, true〉 〈T_0, r, number, 0〉 〈main, w, number, 42〉 〈T_0, end〉 〈main, end〉
   - Reordering of events in main thread leads to different outcome
   - This is because write to number in main is not visible (better: not guaranteed to be visible) to read of number in other thread
   - Implication: there is a data race involving number!

What about Locking?

- Add two new event specs
  - lock, M: acquire lock on M
  - unlock, M: release lock on M
- Now consider BoundedCounter progam from before (simplified):

```
public static void main(String[] args) throws InterruptedException
{
  BoundedCounter c = new BoundedCounter (2);
  Thread t1 = new Thread (new BoundedCounterIncThread (c));
  Thread t2 = new Thread (new BoundedCounterIncThread (c));
  t1.start();
  t2.start();
}
```

  - Join statements are left out, as is println
- What are allowed event sequences?
  - Recall Java Memory Model: after an unlock, subsequent lock on same object makes all of operations before unlock visible
  - So what are possible event sequences?

Example Sequences

- Valid

  ⟨main, begin⟩ ⟨main, w, c.value, 0⟩ ⟨main, w, c.upperBound, 2⟩ ⟨main, launch, t1⟩ ⟨main, launch, t2⟩ ⟨t1, begin⟩ ⟨t2, begin⟩ ⟨t1, lock c⟩ ⟨t1, r, c.value. 0⟩ ⟨t1, w, c.value, 1⟩ ⟨t1, unlock c⟩ ⟨t2, lock c⟩ ⟨t2, r, c.value, 1⟩ ⟨t2, w, c.value, 2⟩ ⟨t2, unlock c⟩ ⟨main, end⟩ ⟨t1, end⟩ ⟨t2, end⟩
- Not valid

  ⟨main, begin⟩ ⟨main, w, c.value, 0⟩ ⟨main, w, c.upperBound, 2⟩ ⟨main, launch, t1⟩ ⟨main, launch, t2⟩ ⟨t1, begin⟩ ⟨t2, begin⟩ ⟨t1, lock c⟩ ⟨t1, r, c.value. 0⟩ ⟨t1, w, c.value, 1⟩ ⟨t1, unlock c⟩ ⟨t2, lock c⟩ ⟨t2, r, c.value, 0⟩ ⟨t2, w, c.value, 2⟩ ⟨t2, unlock c⟩ ⟨main, end⟩ ⟨t1, end⟩ ⟨t2, end⟩
- Why is second sequence invalid?
  - t2 reads a value of c.value that is different from the last value assigned to c.value before previous unlock by t1
  - This is not allowed by Java Memory Model!

Join Statement

- t1.join(): wait for thread to exit, then continue
- Another event spec needed!
  - join, thread’: event associated with successful termination of join operation on thread’
  - join events have to follow end events

What Are Valid Event Sequences?

- Formalized using "happened-before" relation
  - Definition given in Java Language Specification (Section 17)
  - Based on seminal work of computer scientist Leslie Lamport in 1978
- Idea: happened-before relation describes constraints on what events must happen before others in a sequence
  - Used in defining visibility in Java!
  - If write "happened-before" read, then no valid reordering of events in event sequence can invalidate value observed in read

Defining Happened-Before

- Recall: events have following form
  - ⟨thread, begin⟩
  - ⟨thread, end⟩
  - ⟨thread, w, location, value⟩
  - ⟨thread, r, location, value⟩
  - ⟨thread, launch, thread'⟩
  - ⟨thread, lock, object⟩
  - ⟨thread, unlock, object⟩
  - ⟨thread, join, value'⟩
- Happened-before, $e_1 \preccurlyeq e_2$ defined as follows, where $e_1 = \langle t_1, spec_1 \rangle$, $e_2 = \langle t_2, spec_2 \rangle$ are events
  - $t_1 = t_2$ (i.e. events are on same thread), $spec_1$ = begin, and $spec_2 \neq$ begin
  - $t_1 = t_2$, $spec_2$ = end, and $spec_1 \neq$ end
  - $t_1 = t_2$ and "as-if-serial-within-thread" requires this ordering
  - $t_1 \neq t_2$ (i.e. events are on different threads), $spec_1$ = launch, t_2, and $spec_2$ = begin
  - $t_1 \neq t_2$, $spec_1$ = end, and $spec_2$ = join, t_1
  - $t_1 \neq t_2$, $spec_1$ = unlock, obj, and $spec_2$ = lock, obj
  - $t_1 \neq t_2$, $spec_1$ = w, location, value, there is an event e' = ⟨t1, unlock obj⟩ following e_1 in program order, and and $spec_2$ = lock, obj
  - There is an e such that $e_1 \preccurlyeq e \preccurlyeq e_2$
- Fact: valid event sequences must obey ≼!

  Implication: locking on same object enforces visibility!

Intuition

- Simple operational model:
  - Each thread has a (thread-local) buffer
    - Each write goes to the buffer
    - Each read checks the buffer first, else main memory
  - The buffer is emptied to main memory
    - Nondeterministically
    - At a synchronization event
      - Acquring/releasing a lock, joining, etc.
- This model is not completely accurate, but gives you a sense of what's going on