Lecture 6  
The Java Memory Model  

Note: these slides are intended to replace slides 11 – 24 in Lecture 5.
Visibility Reconsidered

• Lock release / acquisition affects visibility of updates to a variable
  – If a variable is “written” before a lock is released ...
  – Then the assignment is visible to threads acquiring the same lock later

• Questions
  – Do other operations besides lock have this property?
  – What, precisely, is this property anyway?

• The Java Memory Model (part of the Java Language Specification) defines precisely how visibility works
  – The JMM defines what values threads can read from memory, based on the execution behavior of the system
  – Implementers of Java must follow the JMM, as it is part of the language specification
  – Understanding the JMM will help you understand notions like visibility and data races better
The Java Memory Model

• Key notions
  – Event sequences
    • Programs are understood in terms of events they generate
    • Events can be reads, writes, lock acquisition, etc.
  – “happens-before”
    • Some events are guaranteed to happen before others
    • Others can be reordered

• 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: \( \langle \text{thread}, \text{event-spec} \rangle \)
  – Thread: name of thread in which event occurs
  – Event spec: four kinds for now
    • \text{begin}: event indicating start of thread
    • \text{end}: event indicating exit of thread
    • \text{write, location, value}: write of value to location
    • \text{read, location, value}: read of value from locations
Example

• Consider following sequential program

```java
public static void main(String[] args) {
    int a = 0;
    int b = 0;
    a = b + 1;
}
```

• Here is an event sequence

```plaintext
〈main, begin〉
〈main, write, a, 0〉
〈main, write, b, 0〉
〈main, read, b, 0〉
〈main, write, a, 1〉
〈main, end〉
```
Program Order

• The previous event sequence is said to follow “program order”
  – Events occur in order indicated by the source code
  – It is well-formed
    • begin is first event
    • end is last event
    • Every read of a variable returns the last value written to the variable

• Every sequential program execution gives rise to a unique “program-order” event sequence
Other Event Sequences Are Allowed!

- The Java Language specification allows other event sequences besides “program order”!
- Why? *Optimizations*
  - Performing operations in different orders may be more efficient
  - Use of registers / caches / etc. (“copies of memory”) may make the program faster
Another Allowed Event Sequence

public static void main(String[] args) {
    int a = 0;
    int b = 0;
    a = b+1;
}

• **Sequence $S_1$**
  - 〈main, begin〉
  - 〈main, write, b, 0〉
  - 〈main, write, a, 0〉
  - 〈main, read, b, 0〉
  - 〈main, write, a, 1〉
  - 〈main, end〉

• **Program-order sequence**
  - 〈main, begin〉
  - 〈main, write, a, 0〉
  - 〈main, write, b, 0〉
  - 〈main, read, b, 0〉
  - 〈main, write, a, 1〉
  - 〈main, end〉
Why Is $S_1$ OK?

- The write of 0 to variable $a$ is delayed until after the write to $b$
- This is allowed because the initialization of $a$ has no impact on the value to which $b$ is initialized
Yet Another Allowed Event Sequence

```java
public static void main(String[] args) {
    int a = 0;
    int b = 0;
    a = b+1;
}
```

- **Sequence $S_2$**
  - ⟨main, begin⟩
  - ⟨main, write, b, 0⟩
  - ⟨main, read, b, 0⟩
  - ⟨main, write, a, 1⟩
  - ⟨main, end⟩

- **Program-order sequence**
  - ⟨main, begin⟩
  - ⟨main, write, a, 0⟩
  - ⟨main, write, b, 0⟩
  - ⟨main, read, b, 0⟩
  - ⟨main, write, a, 1⟩
  - ⟨main, end⟩
Why Is $S_2$ OK?

- The write of 0 to $a$ is never read
- The subsequent write of 1 to $a$ “overwrites” this value
- So for efficiency, compiler can remove the event entirely!
An Unallowed Event Sequence

public static void main(String[] args) {
    int a = 0;
    int b = 0;
    a = b+1;
}

• Sequence $S_3$
  \[
  \langle \text{main, begin} \rangle \\
  \langle \text{main, write, } b, 0 \rangle \\
  \langle \text{main, read, } b, 0 \rangle \\
  \langle \text{main, write, } a, 1 \rangle \\
  \langle \text{main, write, } a, 0 \rangle \\
  \langle \text{main, end} \rangle 
  \]

• Program order sequence
  \[
  \langle \text{main, begin} \rangle \\
  \langle \text{main, write, } a, 0 \rangle \\
  \langle \text{main, write, } b, 0 \rangle \\
  \langle \text{main, read, } b, 0 \rangle \\
  \langle \text{main, write, } a, 1 \rangle \\
  \langle \text{main, write, } a, 0 \rangle \\
  \langle \text{main, end} \rangle 
  \]
Why Is $S_3$ Not OK?

• The write of 0 to a has been moved to the end, after the write of 1 to a
• This alters the value of a that is “in effect” when the program terminates
Event Sequences in the Sequential Case

• So: what other event sequences are allowed, and what are not?

• Java Memory Model specifies an as-if-serial semantics for single threads
  – Events can be reordered
  – Results must remain consistent with program order
    • Precise definition of “consistent” is tricky!
    • For now:
      – Assume terminating threads
      – Then consistent means: same final write events observed for every variable
As-If-Serial Example 1

For sample program, $S_1$ is consistent with program-order sequence:
- Final write event for $a$ is 1 in both cases
- Final write event for $b$ is 0 in both cases

$S_1$

- $\langle$main, begin$\rangle$
- $\langle$main, write, b, 0$\rangle$
- $\langle$main, write, a, 0$\rangle$
- $\langle$main, read, b, 0$\rangle$
- $\langle$main, write, a, 1$\rangle$
- $\langle$main, end$\rangle$

Program-order sequence

- $\langle$main, begin$\rangle$
- $\langle$main, write, a, 0$\rangle$
- $\langle$main, write, b, 0$\rangle$
- $\langle$main, read, b, 0$\rangle$
- $\langle$main, write, a, 1$\rangle$
- $\langle$main, end$\rangle$
As-If-Serial: Example 2

For sample program, $S_2$ is consistent with program-order sequence

- Final write event for $a$ is 1 in both cases
- Final write event for $b$ is 0 in both cases

- $S_2$
  
  \[
  \langle \text{main}, \text{begin} \rangle \\
  \langle \text{main}, \text{write}, b, 0 \rangle \\
  \langle \text{main}, \text{read}, b, 0 \rangle \\
  \langle \text{main}, \text{write}, a, 1 \rangle \\
  \langle \text{main}, \text{end} \rangle 
  \]

- Program-order sequence
  
  \[
  \langle \text{main}, \text{begin} \rangle \\
  \langle \text{main}, \text{write}, a, 0 \rangle \\
  \langle \text{main}, \text{write}, b, 0 \rangle \\
  \langle \text{main}, \text{read}, b, 0 \rangle \\
  \langle \text{main}, \text{write}, a, 1 \rangle \\
  \langle \text{main}, \text{end} \rangle 
  \]
As-If-Serial: Example 3

For sample program, $S_3$ is not consistent with program-order sequence

- Final write event for a is 1 in program-order sequence, but 0 in new sequence 3

• $S_3$
  ⟨main, begin⟩
  ⟨main, write, b, 0⟩
  ⟨main, read, b, 0⟩
  ⟨main, write, a, 1⟩
  ⟨main, write, a, 0⟩
  ⟨main, end⟩

• Program order sequence
  ⟨main, begin⟩
  ⟨main, write, a, 0⟩
  ⟨main, write, b, 0⟩
  ⟨main, read, b, 0⟩
  ⟨main, write, a, 1⟩
  ⟨main, write, a, 1⟩
  ⟨main, end⟩
Concurrency and the JMM

- Goal of the JMM: sequential consistency for threaded programs that have no data races
  - Suppose you have threads $t_1$, $t_2$, $t_3$
  - In general, they may execute in parallel, or interleaved, or some combination, depending on number of cores
  - An execution of $t_1$, $t_2$ and $t_3$ is sequentially consistent if it corresponds to executing $t_1$, $t_2$, $t_3$ in a purely interleaved fashion on a single processor
  - Definition of JMM guarantees this property for every execution lacking a data race
The JMM and Locks

• We are discussing locks before threads? Yes ....

• To model locking in event sequences
  – Add two new event specs
    • lock, M: acquire lock on M
    • unlock, M: release lock on M
  – Add new consistency conditions on “as-if-serial” semantics
    • If a memory access (read or write) precedes an unlock event in the program-order sequence then it must precede the unlock event in any consistent sequence
    • If a memory access follows a lock event in the program-order sequence then it must follow that lock event in any consistent sequence
Locking Example

```java
public static void main(String[] args) {
    synchronized (l) {
        int a = 0;
        int b = 0;
    }
    a = b+1;
}
```

Program-order
- main, begin
- main, lock, l
- main, write, a, 0
- main, write, b, 0
- main, unlock, l
- main, read, b, 0
- main, write, a, 1
- main, end

Consistent
- main, begin
- main, lock, l
- main, write, b, 0
- main, unlock, l
- main, read, b, 0
- main, write, a, 1
- main, end

Inconsistent
- main, begin
- main, lock, l
- main, write, a, 0
- main, unlock, l
- main, read, b, 0
- main, write, a, 1
- main, end
The JMM and Threads

• Idea: model concurrency via interleavings of thread behavior
  – Threads execute their own instructions in as-if-serial fashion (terminology: “within-thread-as-if-serial”)
  – Locks can only be acquired if not currently held by a different thread

• New kind of event: launching threads
  – Event specification: launch, t’
  – Idea: a thread generates this event if it performs t’.start();
Concurrency Example

Consider concurrent program (from textbook)

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

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

What are the event sequences?
 Lots! Here Are Two

- **Sequence E₁**
  - (main, begin)
  - (main, launch, T_0)
  - (main, write, number, 42)
  - (main, write, ready, true)
  - (main, end)
  - (T_0, begin)
  - (T_0, read, ready, true)
  - (T_0, read, number, 42)
  - (T_0, end)

- **Sequence E₂**
  - (main, begin)
  - (main, launch, T_0)
  - (main, write, ready, true)
  - (T_0, begin)
  - (T_0, read, ready, true)
  - (T_0, read, number, 0)
  - (main, write, number, 42)
  - (T_0, end)
  - (main, end)

- **T_0** is name of ReaderThread
- Reordering of events in main thread leads to different outcome
  - Reordering obeys “within-thread-as-if-serial” rules (initializations of number, ready swapped)
  - Based on the ordering, T_0 may or may not see the update to number before it prints value
  - Implication: there is a data race involving number!
Join Statement

• `t1.join()`: wait for thread to exit, then continue

• Another event spec needed!
  – `join, t’`: event associated with successful termination of join operation on t’
  – Join events are generated by the thread performing the join, not t’
  – `join, t’` events have to follow end events for t’
Example: BoundedCounters

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

public static void main(String[] args) throws ... {
    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();
}
```

- What are allowed event sequences?
Lots and Lots! Here Are Two Samples

- Sequence $T_1$ (valid)

  - (main, begin)
  - (main, write, c.value, 0)
  - (main, write, c.upperBound, 2)
  - (main, launch, t1)
  - (main, launch, t2)
  - (t1, begin)
  - (t2, begin)
  - (t1, lock c)
  - (t1, lock c)
  - (t1, read, c.value, 0)
  - (t1, unlock c)
  - (t1, write, c.value, 1)
  - (t1, unlock c)
  - (t2, lock c)
  - (t2, lock c)
  - (t2, read, c.value, 1)
  - (t2, unlock c)
  - (t2, write, c.value, 2)
  - (t2, unlock c)
  - (t1, end)
  - (t2, end)
  - (main, join, t1)
  - (main, join, t2)
  - (main, end)

- Sequence $T_2$ (invalid)

  - (main, begin)
  - (main, write, c.value, 0)
  - (main, write, c.upperBound, 2)
  - (main, launch, t1)
  - (main, launch, t2)
  - (t1, begin)
  - (t2, begin)
  - (t1, lock c)
  - (t1, lock c)
  - (t1, read, c.value, 0)
  - (t1, unlock c)
  - (t1, write, c.value, 1)
  - (t1, unlock c)
  - (t2, lock c)
  - (t2, lock c)
  - (t2, read, c.value, 0)
  - (t2, unlock c)
  - (t2, w, c.value, 1)
  - (t2, unlock c)
  - (t1, end)
  - (t2, end)
  - (main, join, t1)
  - (main, join, t2)
  - (main, end)
Why Is Sequence $T_2$ Invalid?

- It does not read the most recent value of c.value
- In fact, it can never read c.value == 0 in this any “as-if-serial” equivalent of this execution!
  - $t_1$’s write to c comes before the lock on c
  - Because of “as-if-serial” semantics, this write cannot be moved to after the unlock
  - $t_2$ subsequently locks c before reading c.value
  - Because of “as-if-serial” semantics, this read cannot be moved to before the lock
  - Therefore $t_1$’s write of c must happen before $t_2$’s read!
What Are Valid Event Sequences?

• Formalized using “happens-before” relation
  – Definition given in Java Language Specification (Section 17)
  – Based on seminal work of computer scientist Leslie Lamport in 1978

• Idea: happens-before relation describes what orderings in a sequence must be preserved in any “consistent” reordering of the sequence
  – Idea: fix the order in which the locks are acquired, released
  – Any orderings that follow this lock ordering is a consistent reordering
    • Some of the reorderings may be due to intra-thread optimizations ("as-if-serial")
    • Others may be due to different scheduling decisions
Defining Happens-Before

- Recall: events have following form
  - \langle thread, begin \rangle
  - \langle thread, end \rangle
  - \langle thread, write, location, value \rangle
  - \langle thread, read, location, value \rangle
  - \langle thread, launch, t' \rangle
  - \langle thread, lock, object \rangle
  - \langle thread, unlock, object \rangle
  - \langle thread, join, t' \rangle
- Suppose S is an event sequence, with events \( e_1 = \langle t_1, \text{spec}_1 \rangle \), \( e_2 = \langle t_2, \text{spec}_2 \rangle \). Then \( e_1 \preceq e_2 \) (“\( e_1 \) happens-before \( e_2 \)”) holds if \( e_1 \) precedes \( e_2 \) in the sequence and one of the following is true.
  - \( t_1 = t_2 \) (i.e. events are on same thread), \( \text{spec}_1 = \text{begin} \) and \( \text{spec}_2 \neq \text{begin} \)
  - \( t_1 = t_2 \), \( \text{spec}_2 = \text{end} \) and \( \text{spec}_1 \neq \text{end} \)
  - \( t_1 = t_2 \) and “within-thread-as-if-serial” requires this ordering
  - \( t_1 \neq t_2 \) (i.e. events are on different threads), \( \text{spec}_1 = \text{launch}, t_2 \) and \( \text{spec}_2 = \text{begin} \)
  - \( t_1 \neq t_2 \), \( \text{spec}_1 = \text{end} \) and \( \text{spec}_2 = \text{join}, t_1 \)
  - \( t_1 \neq t_2 \), \( \text{spec}_1 = \text{unlock}, \text{obj} \) and \( \text{spec}_2 = \text{lock}, \text{obj} \)
  - There is an \( e \) such that \( e_1 \preceq e \preceq e_2 \)
- Valid event sequences must obey \( \preceq \)!
  Implication: locking on same object enforces visibility!
Happens-Before and Data Races

The happens-before relation can be used to define when data races exist

- Suppose there is a sequence $S$ with events $e_1, e_2$ that conflict (one is a write, the other is a read or write)
- Suppose further that neither $e_1 \leq e_2$ nor $e_2 \leq e_1$ hold
- Then there is another sequence in which the order of the events can be switched