CMSC 433 Fall 2014
Section 0101
Mike Hicks
(slides due to Rance Cleaveland)

UNIVERSITY OF MARYLAND

Lecture 7
The Java Memory Model

Covers JCIP Chap 16

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 when the effects of operations are guaranteed to be visible
  - Implementations of Java must follow the JMM, as it is part of the language specification
  - Understanding the JMM clarifies issues like visibility, data races
- For Java 7, the JMM is in Chapter 17.4 of the JLS: http://docs.oracle.com/javase/specs/jls/se7/html/index.html

Out-of-Thin-Air Safety

- A guarantee provided by Java for any read of a variable, regardless of synchronization, etc.:
  - *A read of a variable yields some value written by some thread*
  - Technical detail: initializations count as "writes"
- No specific value ("most recent") guaranteed, however!
- Other languages (e.g. C++) cannot even make this guarantee

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 must "complete" before others "start"
    - 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: ⟨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
    - write, location, value: write to location of value
    - read, location, value: read from location of value

Example

- Consider following sequential program

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

- Here is an event sequence

⟨main, begin⟩
⟨main, write, a, 1⟩
⟨main, write, b, 1⟩
⟨main, read, b, 1⟩
⟨main, write, a, 2⟩
⟨main, end⟩

Program Order and Sequential Consistency

- The previous event sequence is said to follow *program order*
  - Events occur in order indicated by the source code
  - It is guaranteed to be *well-formed*
    - begin is first event
    - end is last event (for terminating computations)
    - It is *sequentially consistent*: every read of a variable returns the last value written to the variable
      - Note: The use of this term for single-threaded executions is unusual, but it applies in the same sense to multithreaded executions, as we'll see.
    - It follows the execution semantics given in the JLS
- Every sequential program execution gives rise to a unique program-order event sequence

Other Event Sequences May Arise!

- Java compilers may generate code whose event sequences depart from 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
  - Requirement: optimized code should have "same effect" as original

Another Allowed Event Sequence

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

Sequence S_1

⟨main, begin⟩
⟨main, write, b, 1⟩
⟨main, write, a, 1⟩
⟨main, read, b, 1⟩
⟨main, write, a, 2⟩
⟨main, end⟩

Program-order sequence

⟨main, begin⟩
⟨main, write, a, 1⟩
⟨main, write, b, 1⟩
⟨main, read, b, 1⟩
⟨main, write, a, 2⟩
⟨main, end⟩

Why Is S_1 OK?

- The write of 1 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

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

Sequence S_2

⟨main, begin⟩
⟨main, write, b, 1⟩
⟨main, read, b, 1⟩
⟨main, write, a, 2⟩
⟨main, end⟩

Program-order sequence

⟨main, begin⟩
⟨main, write, a, 1⟩
⟨main, write, b, 1⟩
⟨main, read, b, 1⟩
⟨main, write, a, 2⟩
⟨main, end⟩

Why Is S_2 OK?

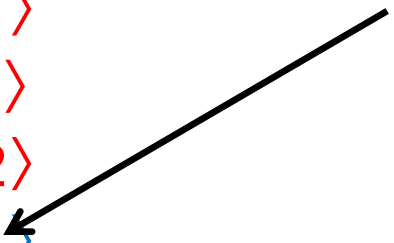
- The write of 1 to a is never read
- The subsequent write of 1 to a "overwrites" this value
- So for efficiency, compiler can remove the event entirely!

A Disallowed Event Sequence

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

Sequence S_3

⟨main, begin⟩
⟨main, write, b, 1⟩
⟨main, read, b, 1⟩
⟨main, write, a, 2⟩
⟨main, write, a, 1⟩
⟨main, end⟩



Program order sequence

⟨main, begin⟩
⟨main, write, a, 1⟩
⟨main, write, b, 1⟩
⟨main, read, b, 1⟩
⟨main, write, a, 2⟩
⟨main, end⟩

Why Is S_3 Not OK?

- The write of 1 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 individual 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 (at termination) write events observed for every variable

As-If-Serial: Example 1

For sample program, S_1 is consistent with program order

- Final write event for a is 2 in both cases
- Final write event for b is 1 in both cases

S_1	Program order
⟨main, begin⟩	⟨main, begin⟩
⟨main, write, b, 1⟩	⟨main, write, a, 1⟩
⟨main, write, a, 1⟩	⟨main, write, b, 1⟩
⟨main, read, b, 1⟩	⟨main, read, b, 1⟩
⟨main, write, a, 2⟩	⟨main, write, a, 2⟩
⟨main, end⟩	⟨main, end⟩

As-If-Serial: Example 2

For sample program, S_2 is consistent with program order

- Final write event for a is 2 in both cases
- Final write event for b is 1 in both cases

S_2

⟨main, begin⟩
⟨main, write, b, 1⟩
⟨main, read, b, 1⟩
⟨main, write, a, 2⟩
⟨main, end⟩

Program order

⟨main, begin⟩
⟨main, write, a, 1⟩
⟨main, write, b, 1⟩
⟨main, read, b, 1⟩
⟨main, write, a, 2⟩
⟨main, end⟩

As-If-Serial: Example 3

For sample program, S_3 is *not consistent* with program order

Final write event for a is 2 in program order, but 1 in S_3

S_3

⟨main, begin⟩
⟨main, write, b, 1⟩
⟨main, read, b, 1⟩
⟨main, write, a, 2⟩
⟨main, write, a, 1⟩
⟨main, end⟩

Program order

⟨main, begin⟩
⟨main, write, a, 1⟩
⟨main, write, b, 1⟩
⟨main, read, b, 1⟩
⟨main, write, a, 2⟩
⟨main, end⟩

Concurrency and the JMM

- A *multithreaded execution* S is an interleaving of single-threaded executions S_1, S_2, S_3, … for threads t_1, t_2, t_3, …, respectively
- Goal of the JMM: *sequentially consistent* multi-threaded executions for **data race-free programs**
  - A multi-threaded execution S is *sequentially consistent* with respect to S_1, S_2, S_3 iff:
    - Order of events in S_1, S_2, S_3 preserved in S
    - S sequentially consistent in the sense given earlier
- Implications of (multi-threaded) sequential consistency
  - All updates appear to be visible right away
  - Execution of program corresponds to single-processor execution

Sequential Consistency Example

- Suppose b = 0 initially, with two threads t_1, t_2
- The following interleaved execution S demonstrates that S_1 and S_2, combined, are sequentially consistent

S_1

$\langle t_1, \text{read}, b, 0 \rangle$

$\langle t_1, \text{write}, b, 1 \rangle$

$\langle t_1, \text{read}, b, 2 \rangle$

S_2

$\langle t_2, \text{read}, b, 0 \rangle$

$\langle t_2, \text{write}, b, 2 \rangle$

S

$\langle t_1, \text{read}, b, 0 \rangle$

$\langle t_2, \text{read}, b, 0 \rangle$

$\langle t_1, \text{write}, b, 1 \rangle$

$\langle t_2, \text{write}, b, 2 \rangle$

$\langle t_1, \text{read}, b, 2 \rangle$

(ignoring the begin/end events to keep the example shorter)

Sequential Consistency (Non-)Example

- Suppose b = 0 initially, with two threads t_1, t_2
- Following execution is *not* sequentially consistent

S_1	S_2
e_1: ⟨t_1, write, b, 1⟩	e_3: ⟨t_2, write, b, 2⟩
e_2: ⟨t_1, read, b, 2⟩	e_4: ⟨t_2, read, b, 1⟩

- e_4 must read e_1's written value, and e_2 must read e_3's written value
- But then can be no linear order S of all four events that is sequentially consistent

(ignoring the begin/end events to keep the example shorter)

Consistent Executions

- Sequentially consistent multi-threaded executions are **not guaranteed** by the JMM
  - The JMM *allows* this execution for the previous example

- But not all executions are OK
  - What defines which are allowed?
- The JMM includes *constraints on the ordering of events* that define *consistent executions*

$\langle t_1, \text{write}, b, 1 \rangle$
$\langle t_2, \text{write}, b, 2 \rangle$
$\langle t_1, \text{read}, b, 2 \rangle$
$\langle t_2, \text{read}, b, 1 \rangle$

Events for Locking and Threads

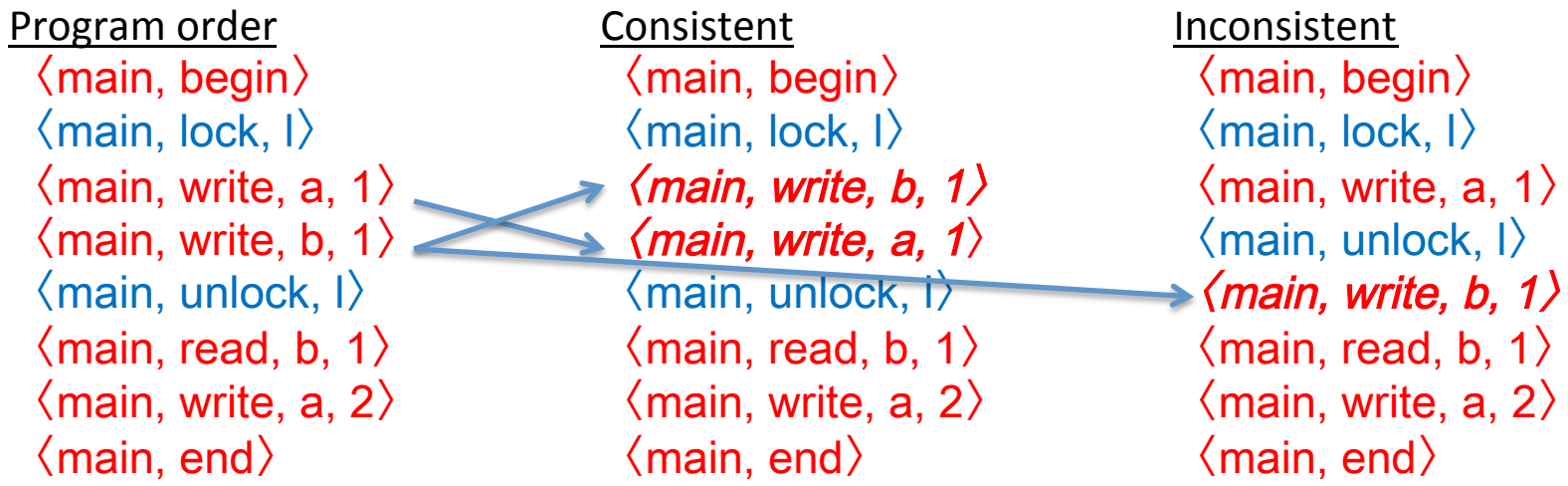
- Locking events
  - lock, M: acquire lock on M
  - unlock, M: release lock on M
- Threading events
  - launch, t ': event associated with successful termination t'.start()
  - join, t': event associated with successful termination of t'.join()
    - These two events are not performed by t'!
    - They are generated by the thread performing t'.start() / t'.join
- ***Constraint***: In any legal interleaving S, event ⟨t, launch, t'⟩ must precede ⟨t', begin⟩ and event ⟨t, join, t'⟩ must come after ⟨t', end⟩
  - According to *happens-before* ordering, which we'll define shortly

Program Order and *As-If-Serial Consistency*

- Program-order definition is adjusted in obvious fashion for synchronization / concurrency events
  - E.g., Can't release a lock before you acquire it
- New consistency conditions are added to "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
- This is sometimes called the "roach motel" constraint
  - Reads / writes *can* be moved inside synchronized blocks
  - They *cannot* be moved outside

Locking Example

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



The JMM and Threads

- Idea: model execution via per-thread event sequences
  - Threads execute their own instructions in as-if-serial fashion
  - There is a **total order** on all *synchronization events* in a given execution
    - begin, end and lock, unlock and launch, join
- "Total order on all synchronization events…"?
  - Intuition: there is a "synchronization server" (i.e. the JVM) that handles all synchronization requests
  - The total order is the order the server processes these requests
    - We'll call it the ***synchronization sequence***
  - The order has to satisfy some (common-sense) constraints, e.g.,
    - Order must be consistent with orders of individual threads
    - Locks can only be acquired if not currently held by a different thread
    - Locks can only be released if currently held

Example

Consider `NoVisibility.java` program (from textbook)

```
  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 do executions look like?

Sample Execution

Sequence for main

⟨main, begin⟩
⟨main, launch, R⟩
⟨main, write, number, 42⟩
⟨main, write, ready, true⟩
⟨main, end⟩

and

Sequence for ReaderThread

⟨R, begin⟩
⟨R, read, ready, true⟩
⟨R, read, number, 42⟩
⟨R, end⟩

and

Synchronization sequence

⟨main, begin⟩
⟨main, launch, R⟩
⟨main, end⟩
⟨R, begin⟩
⟨R, end⟩

yields

Execution

⟨main, begin⟩
⟨main, launch, R⟩
⟨main, write, number, 42⟩
⟨main, write, ready, true⟩
⟨main, end⟩
⟨R, begin⟩
⟨R, read, ready, true⟩
⟨R, read, number, 42⟩
⟨R, end⟩

Another Execution

Sequence for main

⟨main, begin⟩
⟨main, launch, R⟩
⟨main, write, number, 42⟩
⟨main, write, ready, true⟩
⟨main, end⟩

Note:
As if events swapped

and

Sequence for ReaderThread

⟨R, begin⟩
⟨R, read, ready, true⟩
⟨R, read, number, 0⟩
⟨R, end⟩

and

Synchronization sequence

⟨main, begin⟩
⟨main, launch, R⟩
⟨R, begin⟩
⟨main, end⟩
⟨R, end⟩

yields

(Non-SC) Execution

⟨main, begin⟩
⟨main, launch, R⟩
⟨main, write, ready, true⟩
⟨R, begin⟩
⟨R, read, ready, true⟩
⟨R, read, number, 0⟩
⟨main, write, number, 42⟩
⟨main, end⟩
⟨R, end⟩

What Are Valid Event Sequences?

- Formalized using happens-before relation
  - Definition given in Java Language Specification (Section 17.4)
  - Based on seminal work of Leslie Lamport in 1978
- Idea: happens-before defines when one event is guaranteed to complete before another begins in a given execution
  - Has visibility implications!
  - Happens-before also restricts allowed compiler reorderings; cannot invalidate happens-before

Defining Happens-Before

- Recall definition of execution
  - Program-order event sequence S_i for each thread t_i
  - Sequence S, across all synchronization events in the S_i, that is consistent with each S_i
- Notation
  - Execution $\mathcal{E}$ = $\langle \{S_1, \ldots S_n\}, S \rangle$
  - E_i: set of all events in S_i
  - E = $\bigcup_{i=1}^{n} E_i$ (note events in S have to be in E)
- Suppose S is an event sequence, with events e_i = $\langle t_i, spec_i \rangle$, e_2 = $\langle t_j, spec_j \rangle$. Then $e_i \preccurlyeq e_j$ ("e_i happens-before e_j") holds if one of the following is true.
  - $t_i = t_j$ (i.e. events are on same thread) and e_i precedes e_j in S_i
  - $t_i \neq t_j$ (i.e. events are on different threads); $spec_i$ = launch, t_j; and $spec_2$ = begin
  - $t_i \neq t_j$; $spec_i$ = end; and $spec_j$ = join, t_i
  - $t_i \neq t_j$, $spec_i$ = unlock, m; $spec_j$ = lock, m; and e_i precedes e_j in S
  - There is an e in E such that $e_i \preccurlyeq e \preccurlyeq e_j$

Recall Sample Execution

Sequence for main

⟨main, begin⟩
⟨main, launch, R⟩
⟨main, write, number, 42⟩
⟨main, write, ready, true⟩
⟨main, end⟩

Sequence for ReaderThread

⟨R, begin⟩
⟨R, read, ready, true⟩
⟨R, read, number, 0⟩
⟨R, end⟩

Synchronization sequence (one possible)

⟨main, begin⟩
⟨main, launch, R⟩
⟨R, begin⟩
⟨main, end⟩
⟨R, end⟩

Ordering constraints

- ⟨main, begin⟩ $\preccurlyeq$ ⟨main, launch, R⟩
- ⟨main, launch, R⟩ $\preccurlyeq$ ⟨R, begin⟩
- ⟨main, begin⟩ $\preccurlyeq$ ⟨main, write, ready, true⟩
- ⟨main, write, ready, true⟩ $\not\preccurlyeq$ ⟨R, read, ready, true
- ⟨R, read, ready, true⟩ $\not\preccurlyeq$ ⟨main, write, ready, true⟩

No requirement on ordering!

Happens-Before and Data Races

- The happens-before relation can be used to define when data races exist
  - Suppose there is execution $\mathcal{E}$ with events e_i, e_j that conflict (one is a write, other is a read / write on same variable)
  - Suppose further that neither $e_i \preccurlyeq e_j$ nor $e_i \preccurlyeq e_j$ hold
  - Then a data race exists (by definition)!
    - No synchronization forces either event to be visible to the other
    - They conceptually happen "simultaneously"
- In previous example, there was a data race!

Visibility of reads and writes

- If event read, x, V (call it e) is *not* involved in a data race, then there must be some write, x, V' that happens immediately before the read, and V = V'
- If e *is* involved in a data race, what can V be, according to the JMM?
  - If write, x, V' is the most recent write event that happens before e, or it is not ordered with respect to e, then V can be V'
  - Notably, it cannot be due to writes that happen before these writes, or that happen after e

Fixing the Data Race

```
private static Object m = new Object (); …

private static class ReaderThread extends Thread {
  public void run () {
    boolean myReady = false;
    while (!myReady){
       Thread.yield();
       synchronized(m) { myReady = ready; }
    }
    System.out.println (number);
  }
} …
public static void main(String[] args) {
  new ReaderThread().start();
  synchronized(m) {
    number = 42;
    ready = true;
  }
}
```

What do executions look like?

(One) Fixed Execution

Sequence for main

⟨main, begin⟩
⟨main, launch, R⟩
⟨main, lock, M⟩
⟨main, write, number, 42⟩
⟨main, write, ready, true⟩
⟨main, unlock, M⟩
⟨main, end⟩

Sequence for ReaderThread

⟨R, begin⟩
⟨R, lock, M⟩
⟨R, read, ready, true⟩
⟨R, unlock, M⟩
⟨R, read, number, 42⟩
⟨R, end⟩

Synchronization sequence (one possible)

⟨main, begin⟩
⟨main, launch, R⟩
⟨R, begin⟩
⟨main, lock, M⟩
⟨main, unlock, M⟩
⟨R, lock, M⟩
⟨R, unlock, M⟩
⟨main, end⟩
⟨R, end⟩

What are the ordering constraints?

(One more) Fixed Execution

Sequence for main

⟨main, begin⟩
⟨main, launch, R⟩
⟨main, lock, M⟩
⟨main, write, number, 42⟩
⟨main, write, ready, true⟩
⟨main, unlock, M⟩
⟨main, end⟩

Synchronization sequence (one possible)

⟨main, begin⟩
⟨main, launch, R⟩
⟨R, begin⟩
⟨R, lock, M⟩
⟨R, unlock, M⟩
⟨main, lock, M⟩
⟨main, unlock, M⟩
⟨R, lock, M⟩
⟨R, unlock, M⟩
⟨main, end⟩
⟨R, end⟩

Sequence for ReaderThread

⟨R, begin⟩
⟨R, lock, M⟩
⟨R, read, ready, false⟩
⟨R, unlock, M⟩
⟨R, lock, M⟩
⟨R, read, ready, true⟩
⟨R, unlock, M⟩
⟨R, read, number, 42⟩
⟨R, end⟩

What are the ordering constraints?

Proper Synchronization and the JMM

- Execution $\mathcal{E}$ (which we assume adhers to the happens-before relation) is *properly synchronized* if it has no data races
- The JMM stipulates the following:
  - Every thread executes in an "as-if-serial" manner
  - In every execution, a read event for a variable always involves a value produced in some write event for that variable
  - Every properly synchronized execution is sequentially consistent
  - So: "no data races" means "sequential consistency"
  - Thus: writes in such executions can be seen as "immediately visible"

Proper Synchronization of Programs

- A Java program is *properly synchronized* if every possible execution $\mathcal{E}$ of it is properly synchronized
  - This means: no data races are possible when the program is run.
  - In this case the JMM guarantees that every execution is sequentially consistent
  - Implication: no visibility-related bugs
- One way to ensure proper synchronization: locks!
  - If every read, write on the same variable in different threads is ordered by happens-before, then there are no data races
  - Locking can be used to enforce happens-before orderings!
    - Employing the "guarded by" pattern

Volatility and Visibility

- Recall volatile variables
  - E.g. `volatile int number;`
  - Writes are atomic, immediately visible
  - So are writes immediately preceding these operations!
- More formally:
  - **Reads**, **writes** of volatile variables give rise to additional synchronization events (**lock** or **unlock**)
    - Per the encoding mentioned in the last lecture: each is surrounded by a lock or unlock on a "phantom" lock associated with that variable
  - These events also appear in the synchronization sequences associated with executions
  - The write event for a volatile variable "happens-before" *all subsequent reads* of that variable in the synchronization sequence

Visibility Takeaways

- **Synchronization** addresses both **atomicity** and **visibility** issues in concurrent programming
  - Visibility can also an atomicity issue, namely:
  - Propagation of effects of writes may require several operations (cache flushes, register write-backs, etc.)
- Two synchronization mechanisms so far:
  - **Locking**
  - **Volatile** variables
- Locking can be used to fix *both* atomicity and visibility problems
- Volatile variables can *only* fix visibility problems