Lecture 25
The Java Memory Model
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 exactly how visibility works
  - The JMM explains 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: \(\langle \text{thread}, \text{event-spec} \rangle\)
  – “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
  ```java
  public int a;
  public int b;
  ...
  public static void main(String[] args) {
    a = 1;
    b = 1;
    a = b+1;
  }
  ```

• Here is an event sequence
  ```plaintext
  ⟨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
    • 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

```java
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 $S_3$

\begin{align*}
\langle & \text{main, begin} \rangle \\
\langle & \text{main, write, b, 1} \rangle \\
\langle & \text{main, read, b, 1} \rangle \\
\langle & \text{main, write, a, 2} \rangle \\
\langle & \text{main, write, a, 1} \rangle \\
\langle & \text{main, end} \rangle \\
\end{align*}

Program order sequence

\begin{align*}
\langle & \text{main, begin} \rangle \\
\langle & \text{main, write, a, 1} \rangle \\
\langle & \text{main, write, b, 1} \rangle \\
\langle & \text{main, read, b, 1} \rangle \\
\langle & \text{main, write, a, 2} \rangle \\
\langle & \text{main, write, a, 1} \rangle \\
\langle & \text{main, end} \rangle \\
\end{align*}
Why Is S₃ Not OK?

• The write of 1 to a has been moved to the end, after the write of 2 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 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

$S_1$

\[
\begin{align*}
&\langle\text{main, begin}\rangle \\
&\langle\text{main, write, b, 1}\rangle \\
&\langle\text{main, write, a, 1}\rangle \\
&\langle\text{main, read, b, 1}\rangle \\
&\langle\text{main, write, a, 2}\rangle \\
&\langle\text{main, end}\rangle
\end{align*}
\]

Program order

\[
\begin{align*}
&\langle\text{main, begin}\rangle \\
&\langle\text{main, write, a, 1}\rangle \\
&\langle\text{main, write, b, 1}\rangle \\
&\langle\text{main, read, b, 1}\rangle \\
&\langle\text{main, write, a, 2}\rangle \\
&\langle\text{main, end}\rangle
\end{align*}
\]
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

$S_2$

\[
\langle \text{main, begin} \rangle
\]

\[
\langle \text{main, write, b, 1} \rangle
\]

\[
\langle \text{main, read, b, 1} \rangle
\]

\[
\langle \text{main, write, a, 2} \rangle
\]

\[
\langle \text{main, end} \rangle
\]

Program order

\[
\langle \text{main, begin} \rangle
\]

\[
\langle \text{main, write, a, 1} \rangle
\]

\[
\langle \text{main, write, b, 1} \rangle
\]

\[
\langle \text{main, read, b, 1} \rangle
\]

\[
\langle \text{main, write, a, 2} \rangle
\]

\[
\langle \text{main, end} \rangle
\]
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 $S_3$

$S_3$

\[
\langle \text{main, begin} \rangle
\]

\[
\langle \text{main, write, b, 1} \rangle
\]

\[
\langle \text{main, read, b, 1} \rangle
\]

\[
\langle \text{main, write, a, 2} \rangle
\]

\[
\langle \text{main, write, a, 1} \rangle
\]

\[
\langle \text{main, end} \rangle
\]

Program order

\[
\langle \text{main, begin} \rangle
\]

\[
\langle \text{main, write, a, 1} \rangle
\]

\[
\langle \text{main, write, b, 1} \rangle
\]

\[
\langle \text{main, read, b, 1} \rangle
\]

\[
\langle \text{main, write, a, 2} \rangle
\]

\[
\langle \text{main, end} \rangle
\]
Concurrency and the JMM

• Goal of the JMM: *sequential consistency for multi-threaded programs* that have no data races
  – Consider execution of threads \( t_1, t_2, t_3 \); each has its own event sequence (program order): \( S_1, S_2, S_3 \)
  – The (multi-threaded) execution is sequentially consistent if there is an interleaving \( S \) of \( S_1, S_2, S_3 \) with:
    • Order of events in \( S_1, S_2, S_3 \) preserved in \( S \)
    • \( S \) sequentially consistent in sense given on previous slide

• 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, and there are two threads \( t_1, t_2 \)
- Following execution is sequentially consistent, as shown by \( S \)

\[
\begin{align*}
S_1 & \quad \langle t_1, \text{read}, b, 0 \rangle \quad \langle t_1, \text{write}, b, 1 \rangle \quad \langle t_1, \text{read}, b, 2 \rangle \\
S_2 & \quad \langle t_2, \text{read}, b, 0 \rangle \quad \langle t_2, \text{write}, b, 2 \rangle \\
S & \quad \langle t_1, \text{read}, b, 0 \rangle \quad \langle t_2, \text{read}, b, 0 \rangle \quad \langle t_1, \text{write}, b, 1 \rangle \quad \langle t_2, \text{write}, b, 2 \rangle \quad \langle t_1, \text{read}, b, 2 \rangle
\end{align*}
\]
Sequential Consistency (Non-)Example

• Suppose b = 0 initially, and there are two threads $t_1$, $t_2$
• Following execution is not sequentially consistent (why not?)

$S_1$

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

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

$S_2$

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

$\langle t_2, \text{read}, b, 1 \rangle$
Synchronization and Concurrency

- The JMM defines which executions are allowed for multi-threaded Java programs
- Needed: event specifications for locking, threading
  - **Locks**
    - **lock, M:** acquire lock on M
    - **unlock, M:** release lock on M
  - **Threads**
    - **launch, t’:** event associated with successful termination of t’.start()
    - **join, t’:** event associated with successful termination of t’.join()

- launch, t’ and join, t’ events are not performed by t’!
  - They are generated by the thread performing t’.start() / t’.join
  - Event ⟨t, launch, t’⟩ must “precede” ⟨t’, begin⟩
  - Event ⟨t, join, t’⟩ must “follow” ⟨t’, end⟩
Program Order and As-If-Serial Consistency

- Program-order definition is adjusted in obvious fashion for synchronization / concurrency events
- 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
public static void main(String[] args) {
    synchronized (l) {
        a = 1;
        b = 1;
    }
    a = b + 1;
}

Program order
〈main, begin〉
〈main, lock, l〉
〈main, write, a, 1〉
〈main, write, b, 1〉
〈main, unlock, l〉
〈main, read, b, 1〉
〈main, write, a, 2〉
〈main, end〉

Consistent
〈main, begin〉
〈main, lock, l〉
〈main, write, b, 1〉
〈main, write, a, 1〉
〈main, unlock, l〉
〈main, read, b, 1〉
〈main, write, a, 2〉
〈main, end〉

Inconsistent
〈main, begin〉
〈main, lock, l〉
〈main, write, a, 1〉
〈main, unlock, l〉
〈main, write, b, 1〉
〈main, read, b, 1〉
〈main, write, a, 2〉
〈main, end〉
The JMM and Threads

- Idea: model execution via synchronized per-thread event sequences
  - Threads execute their own instructions in as-if-serial fashion
  - There is a total order on all synchronization events across in a given execution
    - begin, end
    - lock, unlock
    - 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
  - The order has to satisfy some (common-sense) constraints
    - 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
    - Etc.
Consider concurrent program (from textbook)

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

```java
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〉

Sequence for ReaderThread ("R")

〈R, begin〉
〈R, read, ready, true〉
〈R, read, number, 0〉
〈R, end〉

Synchronization sequence

〈main, begin〉
〈main, launch, r〉
〈r, begin〉
〈main, end〉
〈r, end〉
Another Execution

Sequence for main

〈main, begin〉
〈main, launch, R〉
〈main, write, number, 42〉
〈main, write, ready, true〉
〈main, end〉

Sequence for ReaderThread ("r")

〈R, begin〉
〈R, read, ready, true〉
〈R, read, number, 42〉
〈R, end〉

Synchronization sequence

〈main, begin〉
〈main, launch, R〉
〈main, end〉
〈R, begin〉
〈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 $e_i = \langle t_i, \text{spec}_i \rangle$, $e_j = \langle t_j, \text{spec}_j \rangle$. Then $e_i \preceq e_j$ (“$e_i$ 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, $t_i$; and spec$_j$ = 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 $e_i \preceq e \preceq e_j$
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”)

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

Synchronization sequence

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

• ⟨main, begin⟩ ≤ ⟨main, launch, R⟩
• ⟨main, launch, R⟩ ≤ ⟨R, begin⟩
• ⟨main, begin⟩ ≤ ⟨main, write, ready, true⟩
• ⟨main, write, ready, true⟩ ≤ ⟨R, read, ready, true⟩; ⟨R, read, ready, true⟩ ̴ ⟨main, write, ready, true⟩
Happens-Before and Data Races

• The happens-before relation can be used to define when data races exist
  – Suppose there is execution $E$ with events $e_i, e_j$ that conflict (one is a write, other is a read / write on same variable)
  – Suppose further that neither $e_i \preceq e_j$ nor $e_i \preceq e_j$ hold
  – Then a data race exists!
    • No synchronization forces either event to be visible to the other
    • They conceptually can happen “simultaneously”

• In previous example, there was a 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?
Proper Synchronization and the JMM

• Execution $E$ is well-formed if happens-before relation is a partial order (no cycles)
• Well-formed execution $E$ is \emph{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 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!
Volatility and Visibility

• Recall volatile variables
  – E.g. `volatile int number;`
  – Writes are atomic and visible
  – So are writes immediately preceding these operations!

• More formally:
  – Reads, writes of volatile variables give rise to synchronization events (like lock / unlock, etc.)
  – 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 Take-aways

• Synchronization addresses both atomicity and visibility issues in concurrent programming
  – Visibility can also be seen as 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