JavaOne℠

Sun's 2004 Worldwide Java Developer Conference℠

The New Java™ Technology Memory Model

java.sun.com/javaone/sf

Jeremy Manson and William Pugh
http://www.cs.umd.edu/~pugh

Audience

- Assume you are familiar with basics of Java™ technology-based threads (“Java threads”)
  - Creating, starting and joining threads
  - Synchronization
  - **`wait`** and **`notifyAll`**

Java Thread Specification

- Revised as part of JSR-133
- Part of the new Java Language Spec
  - and the Virtual Machine Spec
- Features talked about here today are in JDK1.5
  - Not all of these ideas are guaranteed to work in previous versions
  - Previous thread spec was broken
    - forbid optimizations performed by many JVMs

Safety Issues in Multithreaded Systems

- Many intuitive assumptions do not hold
- Some widely used idioms are not safe
  - Original Double-checked locking idiom
  - Checking non-volatile flag for thread termination
- Can't use testing to check for errors
  - Some anomalies will occur only on some platforms
    - e.g., multiprocessors
  - Anomalies will occur rarely and non-repeatedly

Revising the Thread Spec

- The Java Thread Specification has undergone significant revision
  - Mostly to correctly formalize existing behavior
  - But a few changes in behavior
- Goals
  - Clear and easy to understand
  - Foster reliable multithreaded code
  - Allow for high performance JVMs
- Has affected JVMs
  - And badly written existing code
    - Including parts of Sun's JDK

This Talk…

- Describe building blocks of synchronization and concurrent programming in Java
  - Both language primitives and util.concurrent abstractions
- Explain what it means for code to be correctly synchronized
- Try to convince you that clever reasoning about unsynchronized code is almost certainly wrong
  - Not needed for efficient and reliable programs

This Talk…

- We will be talking mostly about
  - synchronized methods and blocks
  - volatile fields
- Same principles work with JSR-166 locks and atomic operations
- Will also talk about final fields and immutability.

Taxonomy

- High level concurrency abstractions
  - JSR-166 and java.util.concurrent
- Low level locking
  - `synchronized()` blocks
- Low level primitives
  - volatile variables, java.util.concurrent.atomic classes
  - allows for non-blocking synchronization
- Data races: deliberate undersynchronization
  - Avoid!
  - Not even Doug Lea can get it right

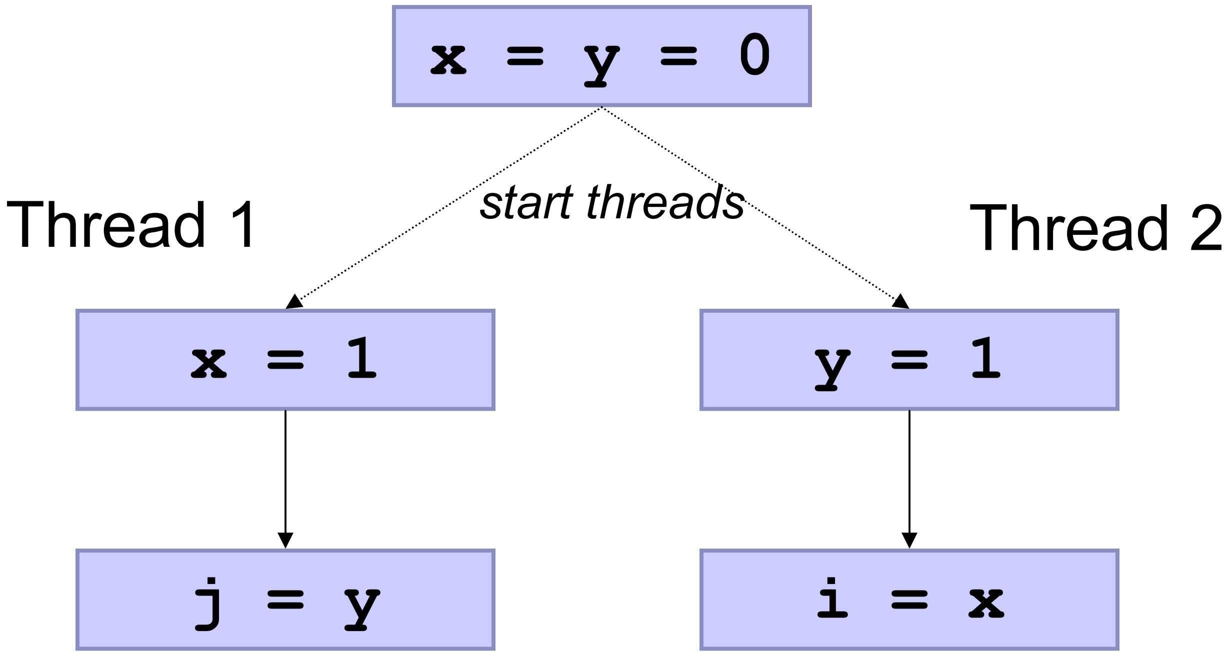
Three Aspects of Synchronization

- Atomicity
  - Locking to obtain mutual exclusion
- Visibility
  - Ensuring that changes to object fields made in one thread are seen in other threads
- Ordering
  - Ensuring that you aren't surprised by the order in which statements are executed

Don’t Try To Be Too Clever

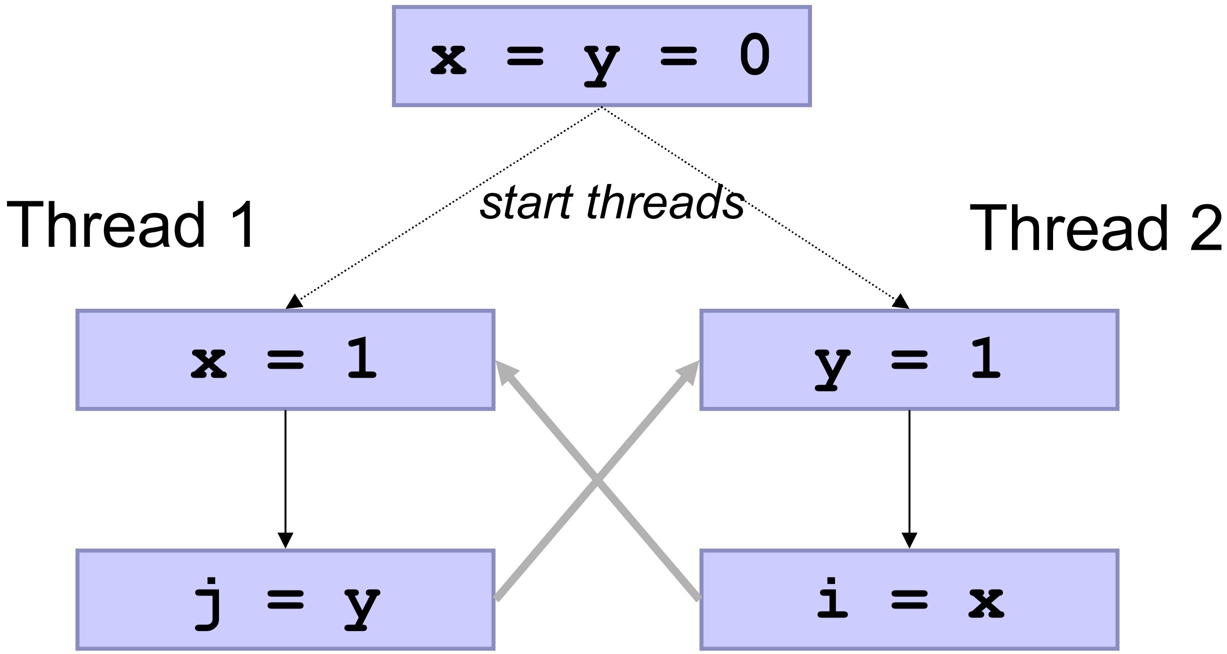
- People worry about the cost of synchronization
  - Try to devise schemes to communicate between threads without using synchronization
    - locks, volatiles, or other concurrency abstractions
- Nearly impossible to do correctly
  - Inter-thread communication without synchronization is not intuitive

Quiz Time



Can this result in i = 0 and j = 0?

Answer: Yes!



How can i = 0 and j = 0?

How Can This Happen?

- Compiler can reorder statements
  - Or keep values in registers
- Processor can reorder them
- On multi-processor, values not synchronized in global memory
- The memory model is designed to allow aggressive optimization
  - including optimizations no one has implemented yet
- Good for performance
  - bad for your intuition about insufficiently synchronized code

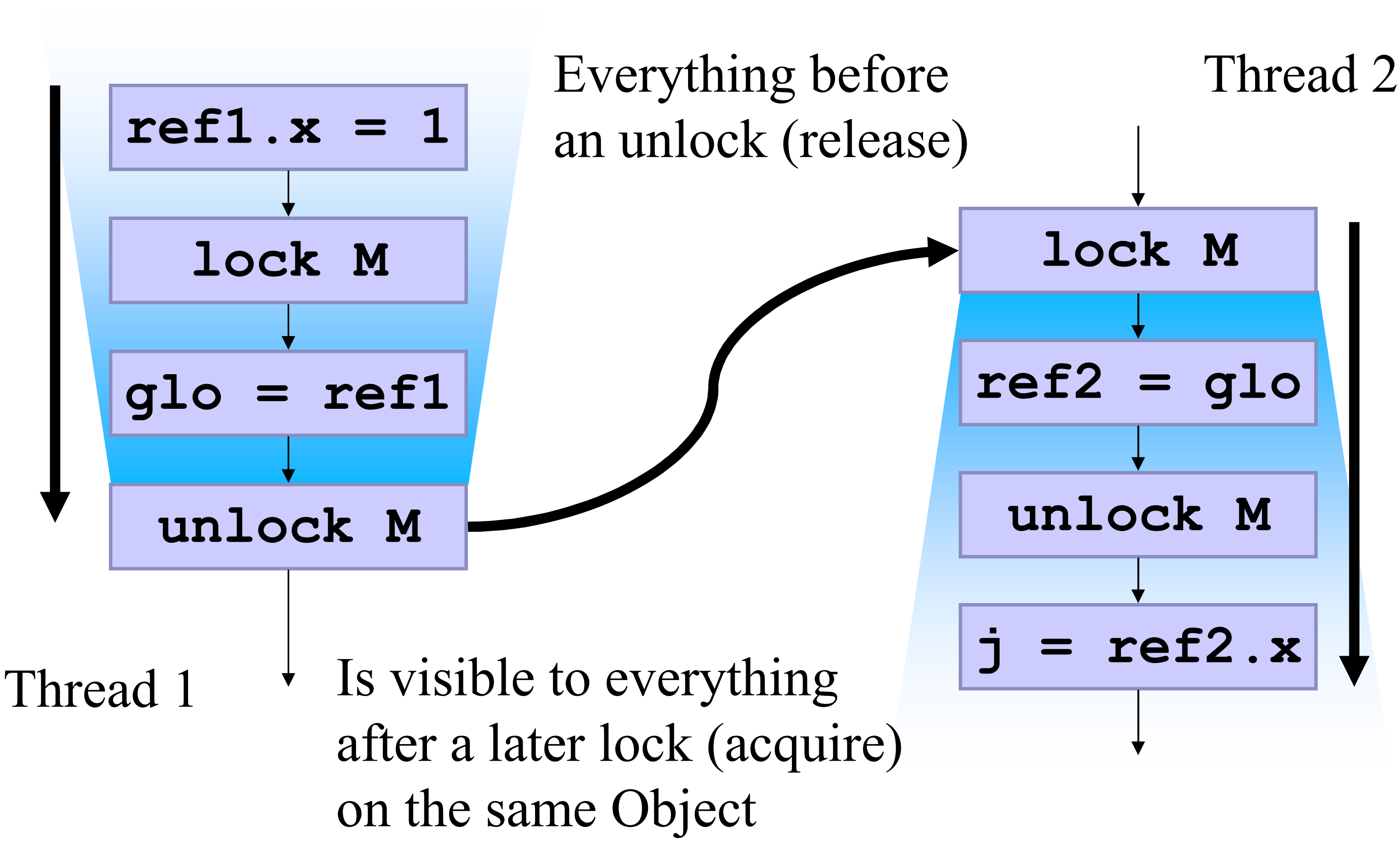
Correctness and Optimizations

- Clever code that depends the order you think the system *must* do things in is almost always wrong in Java
- Dekker's Algorithm (first correct lock implementation) requires this ordering
  - doesn't work in Java, use supplied locks
- Must use synchronization to enforce visibility and ordering
  - As well as mutual exclusion
  - If you use synchronization correctly, you will not be able to see reorderings

Synchronization Actions (*approximately*)

```
// block until obtain lock
synchronized(anObject) {
  // get main memory value of field1 and field2
  int x = anObject.field1;
  int y = anObject.field2;
  anObject.field3 = x+y;
  // commit value of field3 to main memory
}
// release lock
moreCode();
```

When Are Actions Visible to Other Threads?



Release and Acquire

- All accesses before a release
  - are ordered before and visible to
  - any accesses after a matching acquire
- Unlocking a monitor/lock is a release
  - that is acquired by any following lock of *that* monitor/lock

Ordering

- Roach motel ordering
  - Compiler/processor can move accesses into synchronized blocks
  - Can only move them out under special circumstances, generally not observable
- Some special cases:
  - locks on thread local objects are a no-op
  - reentrant locks are a no-op

Volatile fields

- If a field could be simultaneously accessed by multiple threads, and at least one of those accesses is a write
  - make the field volatile
    - documentation
    - gives essential JVM guarantees
  - Can be tricky to get right, but nearly impossible without volatile
- What does volatile do?
  - reads and writes go directly to memory
    - not cached in registers
  - volatile longs and doubles are atomic
    - not true for non-volatile longs and doubles
  - compiler reordering of volatile accesses is restricted
    - roach motel semantics for volatiles and normals
    - no reordering for volatiles and volatiles

Volatile release/acquire

- A volatile write is a release
  - that is acquired by a later read of the same variable
- All accesses before the volatile write
  - are ordered before and visible to all accesses after the volatile read

Volatile guarantees visibility

- `stop` *must* be declared volatile
  - Otherwise, compiler could keep in register

```
class Animator implements Runnable {
  private volatile boolean stop = false;
  public void stop() { stop = true; }
  public void run() {
    while (!stop)
      oneStep();
  }
  private void oneStep() { /*...*/ }
}
```

Volatile guarantees ordering

- If a thread reads **`data`**, there is a release/acquire on **`ready`** that guarantees visibility and ordering

```
class Future {
   private volatile boolean ready;
   private Object data;
   public Object get() {
      if (!ready)
        return null;
      return data;
      }
```

```
public synchronized
  void setOnce(Object o) {
    if (ready) throw … ;
    data = o;
    ready = true;
    }
 }
```

Other Acquires and Releases

- Other actions form release/acquire pairs
- Starting a thread is a release
  - acquired by the run method of the thread
- Termination of a thread is a release
  - acquired by any thread that joins with the terminated thread

Defending against data races

- Attackers can pass instances of your object to other threads via a data race
- Can cause weird things to be observed
  - could be observed in some JVMs
  - in older JVMs, `String` objects might be seen to change
    - change from /tmp to /usr
- If a class is security critical, must take steps
- Choices:
  - use synchronization (even in constructor)
    - object can be made visible to multiple threads before constructor finishes
  - make object immutable by making all fields final

Immutable classes

- Make all critical fields final
- Don’t allow other threads to see object until it is fully constructed
- JVM will be responsible for ensuring that object is perceived as immutable
  - even if malicious code uses data races to attack the class

Optimization of final fields

- New spec allows aggressive optimization of final fields
  - hoisting of reads of final fields across synchronization and unknown method calls
  - still maintains immutability

- Should allow for future JVMs to obtain performance advantages

Finalizers

- Only guaranteed to see writes that occur by the end of the object's constructor.
  - If finalizer needs to see later writes, use synchronization
- Fields may be made final earlier than the program text might imply
  - Synchronization on object also keeps it alive
- Multiple finalizers may be run concurrently
  - Be careful to synchronize properly!

Synchronize When Needed

- Places where threads interact
  - Need synchronization
  - May need careful thought
  - May need documentation
  - Cost of required synchronization not significant
    - For most applications
    - No need to get tricky

Synchronized Classes

- Some classes are synchronized
  - **`Vector`**, **`Hashtable`**, **`Stack`**
  - Most Input/Output Streams
  - Overhead of unneeded synchronization can be measurable
- Contrast with Collection classes
  - By default, not synchronized
  - Can request synchronized version
  - Or can use java.util.concurrent versions (**`Queue`**, **`ConcurrentMap`** implementations)
- Using synchronized classes
  - Often doesn't suffice for concurrent interaction

Synchronized Collections Aren’t Always Enough

- Transactions (DO NOT USE)
  - Violate atomicity…

```
ID getID(String name) {
    ID x = h.get(name);
    if (x == null) {
        x = new ID();
        h.put(name, x);
    }
    return x;
}
```

- Iterators
  - Can’t modify collection while another thread is iterating through it

Concurrent Interactions

- Often need entire transactions to be atomic
  – Reading and updating a Map
  – Writing a record to an OutputStream
- OutputStreams are synchronized
  – Can have multiple threads trying to write to the same OutputStream
  – Output from each thread is nondeterministically interleaved
  – Often essentially useless

util.concurrent

- The stuff in java.util.concurrent is great, use it
- **`ConcurrentHashMap`** has some additional features to get around problems with transactions
  - putIfAbsent
  - concurrent iteration
- **`CopyOnWrite`** classes allow concurrent iteration and non-blocking reads
  - modification is expensive, should be rare

Designing Fast Code

- Make it right before you make it fast
- Reduce synchronization costs
  - Avoid sharing mutable objects across threads
  - avoid old Collection classes (**`Vector`**, **`Hashtable`**)
  - use bulk I/O (or, even better, java.nio classes)
- Use java.util.concurrent classes
  - designed for speed, scalability and correctness
- Avoid lock contention
  - Reduce lock scopes
  - Reduce lock durations

Things That Don't Work

- Thinking about memory barriers
  – There is nothing that gives you the effect of a memory barrier
- Original Double-Check Idiom
  – AKA multithreaded lazy initialization
  – Any unsynchronized non-volatile reads/writes of refs
- Depending on sleep for visibility
- Clever reasoning about cause and effect with respect to data races

Synchronization on Thread Local Objects

- Synchronization on thread local objects
  - (objects that are only accessed by a single thread)
  - has no semantics or meaning
  - compiler can remove it
  - can also remove reentrant synchronization
    - e.g., calling a synchronized method from another synchronized method on same object
- This is an optimization people have talked about for a while
  - not sure if anyone is doing it yet

Thread safe lazy initialization

- Want to perform lazy initialization of something that will be shared by many threads
- Don’t want to pay for synchronization after object is initialized
- Standard double-checked locking doesn’t work
  - making the checked field volatile fixes it
- If two threads might simultaneously access a field, and one of them writes to it
  - the field must be volatile

Wrap-up

- Cost of synchronization operations can be significant
  - But cost of *needed* synchronization rarely is
- Thread interaction needs careful thought
  - But not too clever
  - Don’t want to have to think to hard about reordering
    - No data races in your program, no observable reordering
- Need for inter-thread communication...

Wrap-up - Communication

- Communication between threads
  - Requires *both* threads to interact via synchronization
- JSR-133 & 166 provide new mechanisms for communication
  - High level concurrency framework
  - volatile fields
  - final fields

Q&A