JMM

Thursday, August 27, 2009

The Java Memory Model (JMM)

- Java has a complex memory model which gives implementers considerable freedom
- As long as programmers ensures that all shared variables are only accessed by threads when they hold an appropriate monitor lock, they need not be concerned with issues such as multiprocessor implementations, compiler optimizations,
- However, synchronization can be expensive, and there are times when we might want to use shared variables without an locks
- *One example is the so-called double-checked locking idiom. In this idiom, a singleton resource is to be created; this resource may or may not be used during a particular execution of the program. Furthermore, creating the resource is an expensive operation and should be deferred until it is required*

Intuitive Implementation of Resource

```
public class ResourceController {
  private static Resource resource = null;
  public static synchronized Resource getResource() {
    if(resource == null) resource = new Resource();
    return resource;
  }
}
```

- The problem with this solution is that a lock is required on every access to the resource
- In fact, it is only necessary to synchronize on creation of the resource, as the resource will provide its own synchronization when the threads use it

Double-checked Locking Idiom

```
public class ResourceController {
  private static Resource resource = null;
  public static Resource getResource() {
    if(resource == null) {
      synchronized (ResourceController.class){
        if(resource == null)
          resource = new Resource();
      }
    }
    return resource;
  }
}
```

Why is this broken?

The JMM Revisited I

- In the JMM, each thread is considered to has access to its own working memory as well as the main memory which is shared between all threads
- This working memory is used to hold copies of the data which resides in the shared main memory
- It is an abstraction of data held in registers or data held in local caches on a multi-processor system

The JMM Revisited II

- It is a requirement that:
  - inside a synchronized method or statement any read of a shared variable must read the value from main memory
  - before a synchronized block finishes, any variables written to during the method or statement must be written back to main memory
- Data may be written to the main memory at other times as well, however, the programmer just cannot tell when
- Code can be optimized and reorder as long as long as it maintains "as-if-serial" semantics
- For sequential programs, we will not be able to detect these optimizations and reordering. In concurrent systems, they will manifest themselves unless the program is properly synchronized

Double-checked Locking Idiom Revisited I

- Suppose that a compiler implements the `resource = new Resource()` statement logically as follows

```
tmp = create memory for the Resource class
     // tmp points to memory
Resource.construct(tmp)
    // runs the constructor to initialize
resource = tmp // set up resource
```


Double-checked Locking Idiom Revisited II

- Now as a result of optimizations or reordering, suppose the statements are executed in the following order

```
tmp = create memory for the Resource class
     // tmp points to memory
resource = tmp
Resource.construct(tmp)
    // run the constructor to initialize
```

There is a period of time when the `resource` reference has a value but the `Resource` object has not been initialized!

Warning

- The double-checked locking algorithm illustrates that the synchronized method (and statement) in Java serves a dual purpose
- Not only do they enable mutual exclusive access to a shared resource but they also ensure that data written by one thread (the writer) becomes visible to another thread (the reader)
- The visibility of data written by the writer is only guaranteed when it releases a lock that is subsequently acquired by the reader

Memory Models

- Many programming languages do not deal with concurrency, instead a library provides a set of API calls.
- This is brittle in the presence of optimizing compilers and modern architectures
- Java is the first programming language to specify a *memory model* that must be enforced by the compiler
- The memory model gives semantics to concurrent programs, a must if we want to program multicores
  - C++ is getting one -- it must be good

Motivation: Double Checked Locking

- Double checked locking is an idiom that tries to avoid paying the cost of synchronization when not needed
  - Important for lazy initalization
- The prototypical example:

```
// Single threaded version
class Foo {
  private Helper helper = null;
  public Helper getHelper() {
    if (helper == null)
        helper = new Helper();
    return helper;
    }
  // other functions and members...
  }
```

  - See Bill Pugh's web page for the details.

Broken Double Checked Locking

- Avoids cost of synchronization when helper is already initialized:

```
private Helper helper;
Helper getHelper() {
  if (helper == null)
    synchronized(this){if(helper == null) helper = new Helper();}
  return helper;
}
```

- Problems:
  - The writes that initialize the Helper object and the write to the helper field can be out of order. A thread invoking getHelper() could see a non-null reference to a helper object, but see the default values for its fields, rather than the values set in the constructor.
  - If the compiler inlines the call to the constructor, then the writes that initialize the object and the write to the helper field can be reordered
  - Even if the compiler does not reorder, on a multiprocessor the processor or memory system may reorder those writes, as perceived by a thread running on another processor.

More on why it’s broken

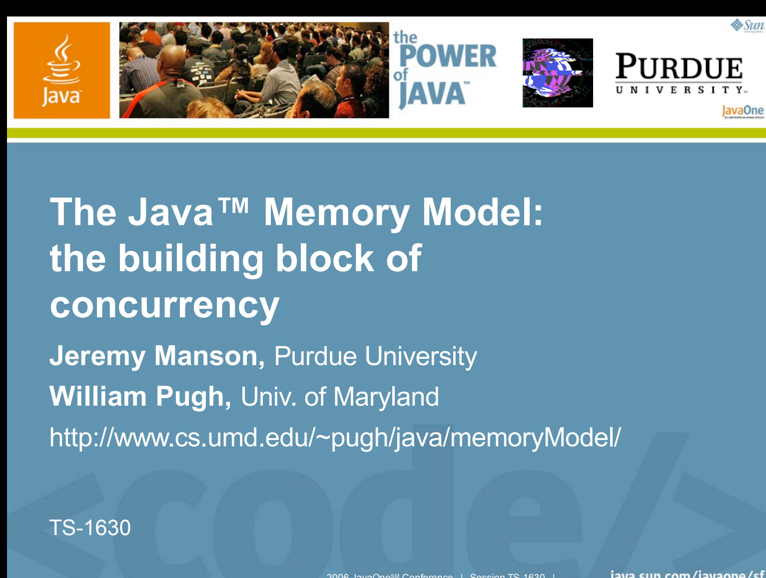
- A test case showing that it doesn't work (by Paul Jakubik). When run on a system using the Symantec JIT:

```
singletons[i].reference = new Singleton();
```

to the following (note that the Symantec JIT using a handle-based object allocation system).

```
0206106A   mov         eax,0F97E78h
0206106F   call        01F6B210                 ; allocate space for
                                                ; Singleton, return result in eax
02061074   mov         dword ptr [ebp],eax      ; EBP is &singletons[i].reference
                                                ; store the unconstructed object here.
02061077   mov         ecx,dword ptr [eax]      ; dereference the handle to
                                                ; get the raw pointer
02061079   mov         dword ptr [ecx],100h     ; Next 4 lines are
0206107F   mov         dword ptr [ecx+4],200h   ; Singleton's inlined constructor
02061086   mov         dword ptr [ecx+8],400h
0206108D   mov         dword ptr [ecx+0Ch],0F84030h
```

As you can see, the assignment to singletons[i].reference is performed before the constructor for Singleton is called. This is completely legal under the existing Java memory model, and also legal in C and C++ (since neither of them have a memory model).



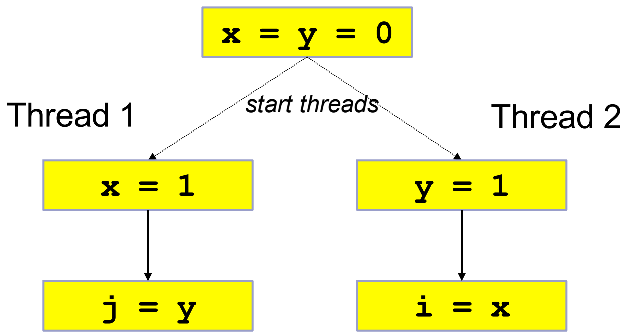
Synchronization is needed for Blocking *and* Visibility

- Synchronization isn't just about mutual exclusion and blocking
- It also regulates when other threads *must* see writes by other threads
  - When writes become visible
- Without synchronization, compiler and processor are allowed to reorder memory accesses in ways that may surprise you
  - And break your code

Don't Try To Be Too Clever

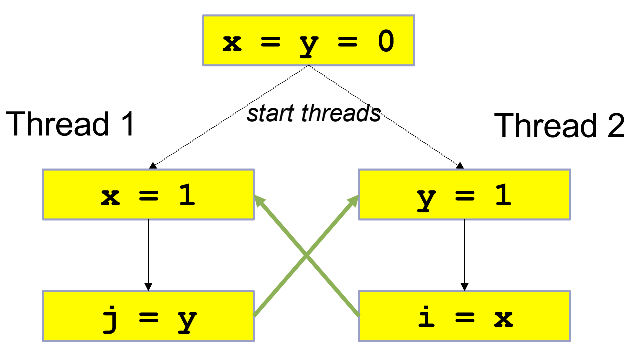
- People worry about the cost of synchronization
  - Try to devise schemes to communicate between threads without using synchronization
    - locks, volatiles, or other concurrency abstractions
- Nearly impossible to do correctly
  - Inter-thread communication without synchronization is not intuitive

Quiz Time



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

Answer: Yes!

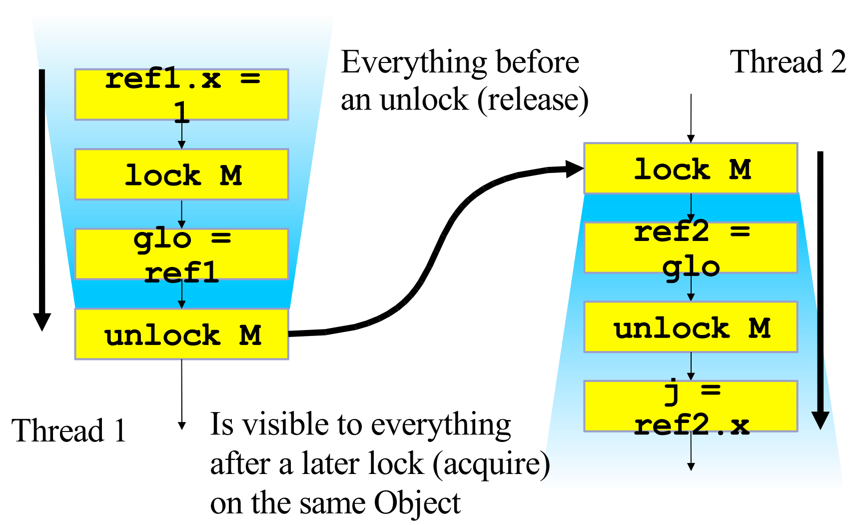


How can i = 0 and j = 0?

How Can This Happen?

- Compiler can reorder statements
  - Or keep values in registers
- Processor can reorder them
- On multi-processor, values not synchronized to 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

When Are Actions Visible to Other Threads?



Release and Acquire

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

Happens-before ordering

- A release and a matching later acquire establish a *happens-before* ordering
- execution order within a thread also establishes a happens-before order
- happens-before order is transitive

Data race

- If there are two accesses to a memory location,
  - at least one of those accesses is a write, and
  - the memory location isn't volatile, then
- the accesses *must* be ordered by happens-before

- Violate this, and you may need a PhD to figure out what your program can do
  - not as bad/unspecified as a buffer overflow in C

Volatile fields

- A field marked **`volatile`** will be treated specially by the compiler
- read/writes go directly to memory and are never cached in registers
- volatile long/double are atomic
- volatile operations can't be reordered by the compiler
- The following example will only be guaranteed to work with volatile because this ensure that stop is not stored in a register:

```
class Animator implements Runnable {
  private volatile boolean stop = false;
  public void stop() { stop = true; }
  public void run() {
    while (!stop)
      oneStep();
      try { Thread.sleep(100); } …;
  }
  private void oneStep() { /*...*/ }
}
```

Volatile fields

- Volatile fields induce happens-before edges, similar to locking
- Incrementing a volatile is not atomic
  - if threads threads try to increment a volatile at the same time, an update might get lost
- volatile reads are very cheap; volatile writes cheaper than synchronization
- atomic operations require compare and swap; provided in JSR-166 (concurrency utils)

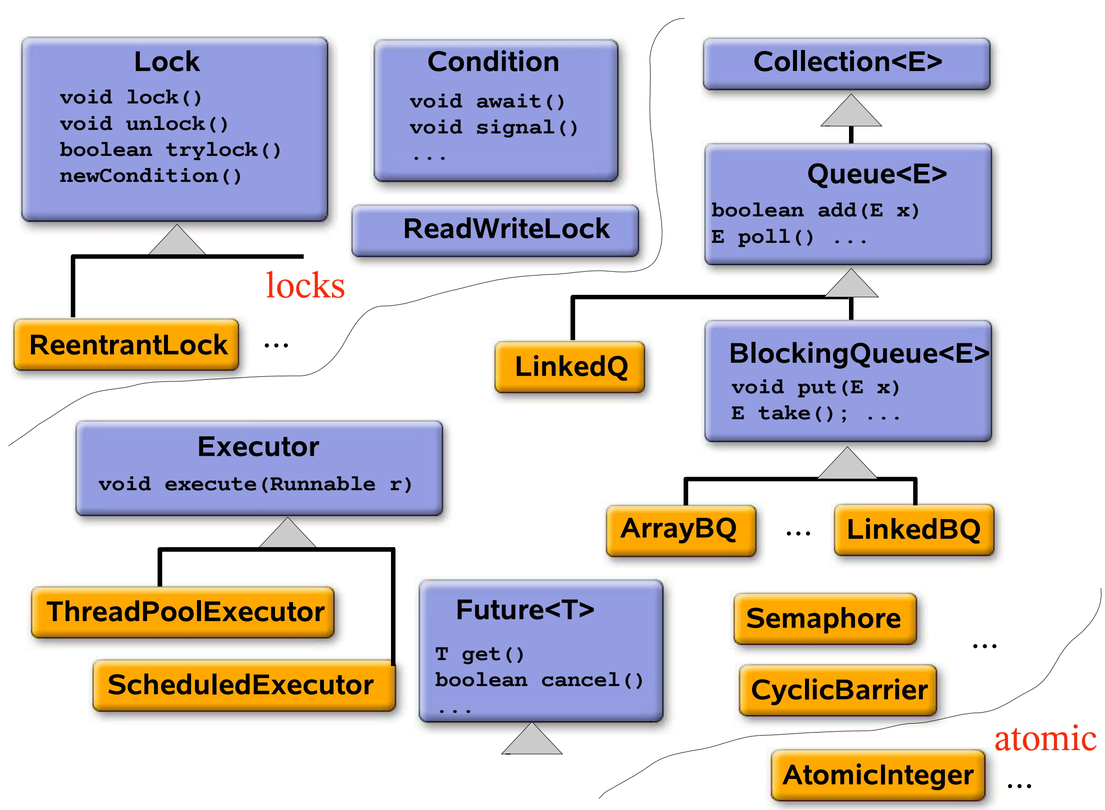
Correct Double Checked Locking

- This avoids the cost of synchronization when helper is already initialized:

```
class Foo {
  private volatile Helper helper;
  Helper getHelper() {
    if (helper == null)
      synchronized(this) {
        if (helper == null) helper = new Helper();
      }
    return helper;
  }
```

Concurrency Utilities and Atomics

- The JSR-166 (designed by Doug Lea at SUNY Oswego) has introduced a rich family of API for concurrent programming



Atomic Variables

- Classes representing scalars supporting

  `boolean compareAndSet(expectedValue, newValue)`

  - Atomically set to `newValue` if currently hold `expectedValue`
  - Also support variant: `weakCompareAndSet`
    - May be faster, but may spuriously fail (as in LL/SC)
- Classes: { *int, long, reference* } X { *value, field, array* } plus boolean value
  - Plus AtomicMarkableReference, AtomicStampedReference
    - (emulated by boxing in J2SE1.5)
- JVMs can use best construct available on a given platform
  - Compare-and-swap, Load-linked/Store-conditional, Locks

http://gee.cs.oswego.edu

Example: `AtomicInteger`

```
class AtomicInteger {
  AtomicInteger(int initialValue);
  int get();
  void set(int newValue);
  int getAndSet(int newValue);
  boolean compareAndSet(int expected, int newVal);
  boolean weakCompareAndSet(int expected, int newVal);
  //      prefetch                   postfetch
  int getAndIncrement();   int incrementAndGet();
  int getAndDecrement();   int decrementAndGet();
  int getAndAadd(int x);   int addAndGet(int x);
}
```

- Integrated with JSR133 memory model semantics for volatile
  - `get` acts as volatile-read
  - `set` acts as volatile-write
  - `compareAndSet` acts as volatile-read and volatile-write
  - `weakCompareAndSet` ordered wrt other accesses to same var

http://gee.cs.oswego.edu

Treiber Stack


```
interface LIFO<E> { void push(E x); E pop(); }

class TreiberStack<E> implements LIFO<E> {
  static class Node<E> {
    volatile Node<E> next;
    final E item;
    Node(E x) { item = x; }
  }

  AtomicReference<Node<E>> head =
       new AtomicReference<Node<E>>();

  public void push(E item) {
    Node<E> newHead = new Node<E>(item);
    Node<E> oldHead;
    do {
      oldHead = head.get();
      newHead.next = oldHead;
    } while (!head.compareAndSet(oldHead, newHead));
  }
```


TreiberStack(2)

http://gee.cs.oswego.edu

```
  public E pop() {
    Node<E> oldHead;
    Node<E> newHead;
    do {
      oldHead = head.get();
      if (oldHead == null) return null;
      newHead = oldHead.next;
    } while (!head.compareAndSet(oldHead,newHead));

    return oldHead.item;
  }

}
```