Advanced Java Multithreading Techniques

Peter Haggar

Senior Software Engineer

IBM Corporation

Research Triangle Park, NC

haggar@us.ibm.com

About Me

- Practicing Software Engineer for 16+ years
- Currently working on Web Services for mobile devices and high performance Web Services
- Specification lead for Real-time Java™ (JSR-000001)
- Author of *Practical Java™ Programming Language Guide*, Addison-Wesley, ISBN: 0-201-61646-7
- Please email me with any questions about this presentation

Agenda

- Visibility and Ordering
- Atomicity
  - 32-bit variables
  - 64-bit variables
- Volatile keyword
  - What it’s supposed to do
  - What it does
  - Why/When you should use it
- Stopping Threads

Agenda

- Changing a Reference of a locked object
- Double Checked Locking
  - The problem it's trying to solve
  - Why it doesn't work
  - More reasons why it doesn't work
    - And even more reasons…
  - Alternatives
- References

Visibility and Ordering

Visibility

- The value of a variable written by one thread at time t_0 is not guaranteed to be read by another thread at time t_1
  - The Java memory model doesn't guarantee that threads will see updates to variables made by other threads, unless both threads synchronize on the same monitor
  - More on this later

Ordering

- The Java Memory Model does not assume that memory operations performed by one thread will be perceived as happening in the same order by another thread

Ordering

- Consider this code and two threads:

```
private boolean stop = false;
private int num = 0;

Thread 1 executes:
{
   num = 100;  //This can appear to happen second
   stop = true;  //This can appear to happen first
}

Thread 2 executes:
if (stop)
  num += num;  //num can == 0!
```

Ordering

- To fix this problem you can try to declare the variables volatile

```
private volatile boolean stop = false;
private volatile int num = 0;

Thread 1 executes:
{
   num = 100;  //This can appear to happen second
   stop = true;  //This can appear to happen first
}

Thread 2 executes:
if (stop)
  num += num;  //num can == 0!
```

Ordering

- volatile variables are supposed to guarantee sequential consistency
  - Compiler or runtime should not reorder
  - However, not all JVMs implement the sequential consistency guarantees of volatile variables
    - IBM and Sun 1.3 and 1.4 Windows JVMs are OK
    - Others may not be
    - Test program found here:
      - ✓ http://www.cs.umd.edu/~pugh/java/memoryModel/

Ordering

- If sequential consistency of volatile does not work on your JVM, use synchronization

```
private boolean stop = false;
private int num = 0;

Thread 1 executes:
synchronized (lock) {
  num = 100;
  stop = true;
}

Thread 2 executes:
synchronized (lock) {
  if (stop)
    num += num;  //num can NEVER == 0
}
```

Atomicity

Atomicity

- JVM guarantee:
  - Reads and writes of 32 bits or less of data is atomic
    - Will not be interrupted before completion
    - Will not read/write a partial value
- How do you know how big your data is?
  - You don't
    - Unless you look at VM source code
    - Try some programming tricks

Data Size

- JLS does not make guarantees about data ***storage*** size
  - It does make guarantees about the ***range of values*** for each type
- Therefore, a JVM is free to use as much storage for each type as it wants to
  - So long as it can handle the JLS mandated range of values for each type

Data Size

- For a 32-bit machine, you can assume
  - 32 bits of storage for
    - reference, int, float, boolean, char, short, byte
    - Smaller types, like char, short, and byte, can be represented more compactly
      - Often promoted to ints
  - 64 bits of storage for
    - double and long

Atomicity

- Reads and writes of 32-bit variables are atomic
  - But not necessarily thread safe
    - Remember visibility
- Multiple threads that access shared variables still must be protected
- See \atomic\RealTimeClock.java

RealTimeClock.java

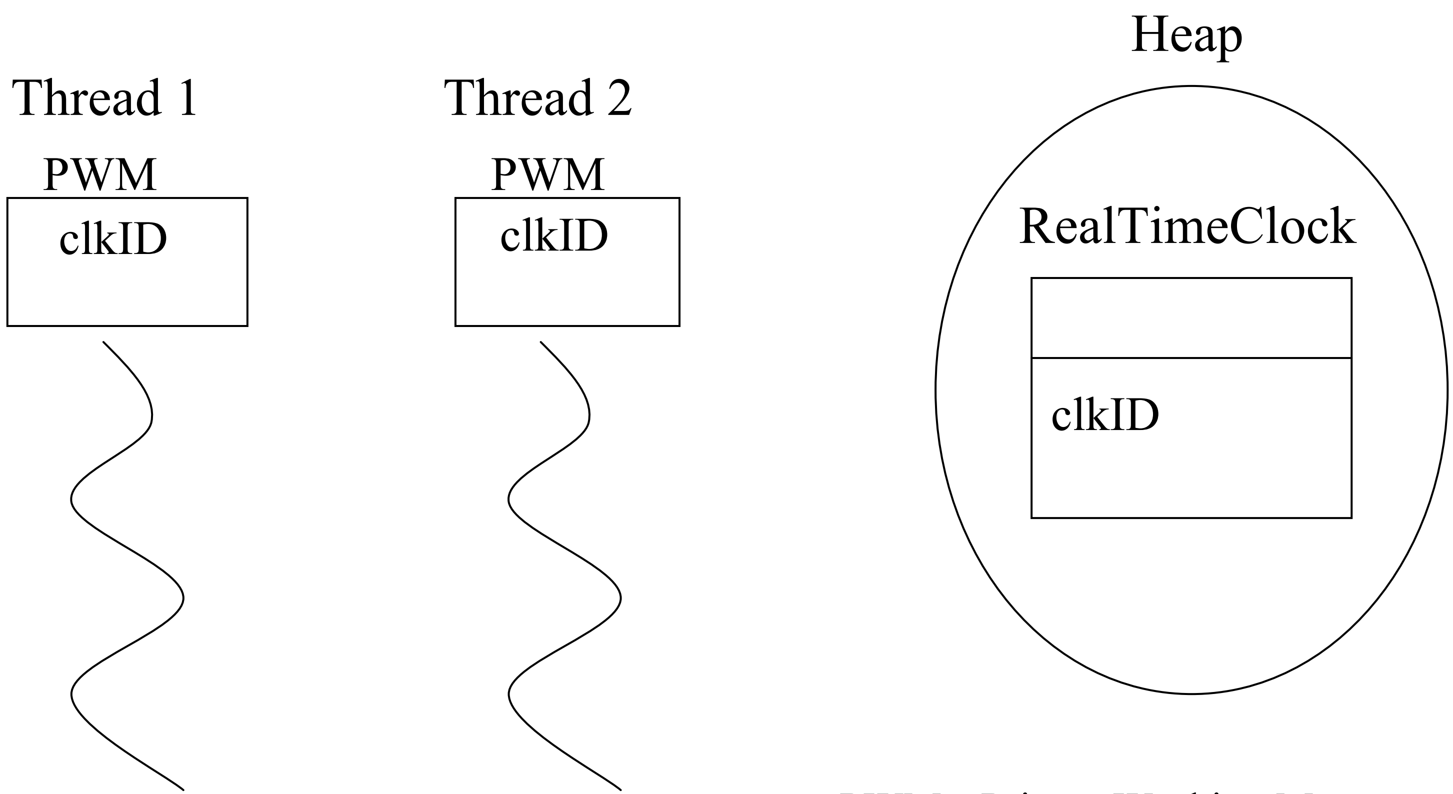
```
class RealTimeClock
{
  private int clkID;

  public int clockID()
  {
    return clkID;
  }
  public void setClockID(int id)
  {
    clkID = id;
  }
  //...
}
```

Atomicity

- Variables are stored in main memory
  - Java™ programming language allows threads to store variables in private working memory of the thread
    - Enables more efficient execution of threads
    - Variables don't have to be reconciled with main memory every time they are manipulated
    - Reconciliation happens at specific synchronization points

Atomicity

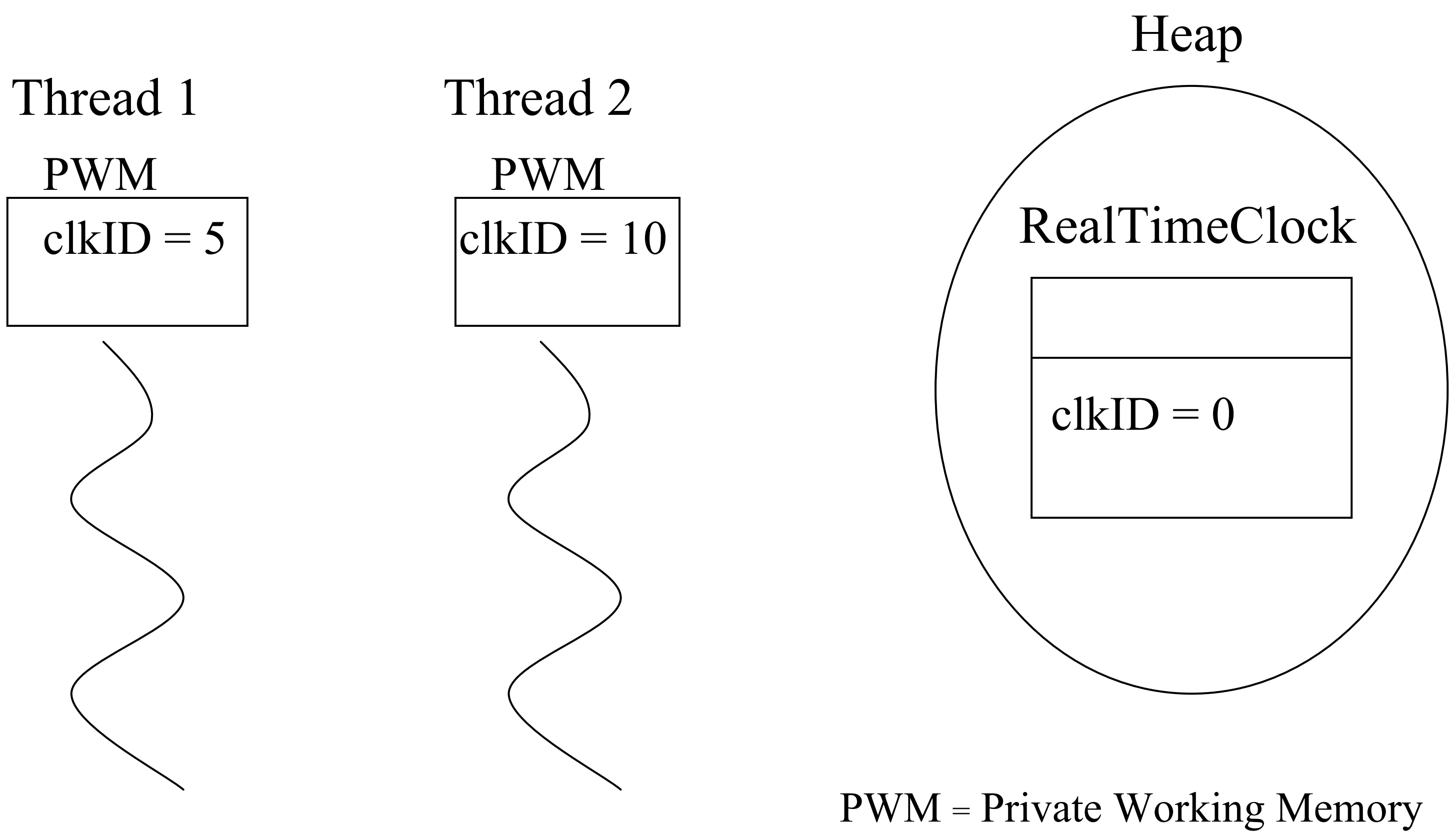


PWM = Private Working Memory

Atomicity

- Consider one instance of RealTimeClock
  - Two threads calling its methods
    - T1 calls setClockID passing 5
      - ✓ 5 placed in T1's private working memory
    - T2 calls setClockID passing 10
      - ✓ 10 placed in T2's private working memory
    - T1 calls clockID which returns 5
      - ✓ 5 returned from T1's private working memory

Atomicity


Thread 1
PWM
clkID = 5
Thread 2
PWM
clkID = 10
Heap
RealTimeClock
clkID = 0
PWM = Private Working Memory

Atomicity

- Note that clkID is an int
  - Operations on it are atomic…but not thread safe
- There is no guarantee that this will happen
  - And no guarantee that it won't
- 2 solutions
  - Access clkID from a synchronized block
  - Declare clkID volatile

Atomicity

- Problem occurs because variables are accessed without protection
- Never read or write a variable in a multithreaded environment without protection
  - synchronized
  - volatile

Atomicity

- synchronized block
  - Private working memory is reconciled with main memory when lock is obtained and when lock is released
- volatile
  - Private working memory is reconciled with main memory on each access of the variable

Atomicity

- Consider volatile when
  - Not updating many variables
    - Want increased concurrency for better performance
- Consider synchronized when
  - Updating many variables
    - Want to eliminate the frequent reconciliation required by volatile for better performance

Atomicity and 64-bit Variables

- The preceding slides indicate how a JVM works with 32-bit variables
- JLS says that 64-bit variables are treated as two 32-bit reads/writes (on a 32-bit machine)
  - They therefore require protection with multiple threads

Atomicity and 64-bit Variables

- According to the JLS, you have two options
  - Use volatile
  - Use synchronized
- JLS says that reads and writes of 64-bit variables declared with volatile are atomic
  - Problem: Not all JVMs implement the semantics of volatile correctly

Atomicity and 64-bit Variables

- See \volatile\AtomicLong.java
- This code and more discussion can be found at:
  - http://www.cs.umd.edu/~pugh/java/memoryModel/

AtomicLong.java

```
public class AtomicLong extends Thread
{
  static volatile long val;
  static int count = 10000000;
  long key;

  AtomicLong(int k)
  {
    long temp = k;
    key = (temp<< 32) | temp;  //key = 00000001 00000001 for
                               //thread 1
  }                    //key = 00000002 00000002 for thread 2 etc

  public void run()
  {
    for(int i = 0; i < count; i++)
    {
      //This 64 bit assignment is supposed to be atomic since val is
      //declared volatile.
      long temp = val;//temp should always = 00000001 00000001
                      //for thread 1
      long temp1 = temp>>>32; //temp1 = 00000000 00000001
      long temp2 = temp<<32;  //temp2 = 00000001 00000000
      temp2 = temp2>>>32;     //temp2 = 00000000 00000001
      if (temp1 != temp2)
      {
        System.out.println("Saw: " + Long.toHexString(temp));
        System.out.println(" temp1 is: " + Long.toHexString(temp1));
        System.out.println(" temp2 is: " + Long.toHexString(temp2));
        System.exit(1);
      }

      //This 64 bit assignment is supposed to be atomic since val is
      //declared volatile.  temp1 should always equal temp2.
      val = key;  //val should always = 00000001 00000001 for
                  //thread 1
    }
  }

  public static void main(String args[])
  {
    for(int t = 1; t < 10; t++)
      new AtomicLong(t).start();
  }
}
```

Atomicity and 64-bit Variables

- Broken on:
  - IBM JDK 1.3 and Sun JDK 1.3 (Windows)
- Fixed on:
  - IBM JDK 1.4 and Sun JDK 1.4 (Windows)

Synchronized

- If your JVM is broken it is fixed with use of synchronized
  - See \synchronized\AtomicLong.java
- You must use synchronized to guarantee atomic behavior of 64 bit variables
  - On pre-1.4 IBM and Sun JVMs (Windows)
    - Test other VMs first
      - ✓ Many will not work

Volatile and Synchronized

- Don't depend on the behavior of volatile unless you test your VM first
- There are other issues with the Java Memory Model
  - Most are things you might not run into
  - Tough to figure out if you do
    - Especially if you don't know about them ahead of time

64-bit Machines

- A 64-bit machine has atomic operations on 64-bits of data
  - The atomicity problems with 64-bit variables and volatile go away with a 64-bit machine
    - Sequential consistency can still be an issue

Stopping a Thread

Stopping a Thread

- Thread.stop() is deprecated
  - It was deemed unsafe to use
  - All locks are released when calling stop()
  - However, code could be in an inconsistent state when stop() is called
- Therefore, to stop a thread you must use a polling loop
  - In all other cases, polling loops should never be used
    - They are terribly inefficient

Stopping a Thread

- Two safe ways to stop a thread
  - Use Thread.interrupt()
  - Roll your own
- Both methods work and have the same limitations
  - Use the dreaded polling loop
  - Thread may not immediately stop
    - Must wait until the thread reaches the top of its loop to exit
      - ✓ Top of loop is the control point

Thread.interrupt

- This is the more standard way to stop a thread
  - See \thread\interrupt\StopTest.java
- Note that interrupt() won't interrupt
  - Code waiting on a lock
    - Synchronized method or block
  - Blocking I/O operation

StopTest.java

```
class WorkerThread extends Thread {
  public void run()   {
    while (!isInterrupted())   {
      //do work
    }
    System.out.println("exiting thread");
  }
}

class StopTest  {
  public static void main(String args[])   {
    WorkerThread wt = new WorkerThread();
    wt.start();
    try {
      Thread.sleep(2000);
    }
    catch(InterruptedException ie) {}
    System.out.println("calling interrupt");
    wt.interrupt();
  }
}
```

Thread.interrupt

- If you are blocked on a socket
  - You can close the socket to unblock and continue
- If you are blocked on a file
  - There is no graceful way to unblock
- interrupt() will interrupt wait() and sleep() with the InterruptedException

"Roll Your Own"

- This way works just as well but beware to use volatile
  - volatile must be used to ensure the correct stop variable value is seen by both threads
  - See \thread\stop\StopTest.java

StopTest.java

```
class WorkerThread extends Thread {
  private volatile boolean stop;

  public void stopThread() {
    stop = true;
  }
  public void run() {
    while (!stop)  {
      //do work
    }
    System.out.println("exiting thread");
  }
}
class StopTest  {
  public static void main(String args[])  {
    WorkerThread wt = new WorkerThread();
    wt.start();
    try {
      Thread.sleep(2000);
    }
    catch(InterruptedException ie) {}
    System.out.println("calling stopThread()");
    wt.stopThread();
  }
}
```

Changing the Reference of a Locked Object

Changing the Reference of a Locked Object

- Synchronizing on an object, locks the object
- What happens if you change the object reference of the locked object?
- See \lockedobj\Stack.java

Stack.java

```
class EmptyStackException extends RuntimeException
{}
class Stack implements Runnable
{
  private int stackSize = 5;
  private int[] intArr = new int[stackSize];
  private int index;  //next available slot in stack

  public static void main(String args[])
  {
    Stack stk = new Stack();
    for (int i=0;i<5;i++)
      stk.push(i);
    Thread thd = new Thread(stk);
    thd.start();
    stk.push(6);
  }

  public void run()
  {
    pop();
  }
```

Stack.java

```
public void push(int val)
 {
  synchronized(intArr) {
   //reallocate integer array(our stack) if it is full.
   if (index == intArr.length)
   {
    System.out.println("inside sync block in push");
    stackSize *= 2;
    int[] newintArr = new int[stackSize];
    System.arraycopy(intArr, 0 , newintArr, 0, intArr.length);
    intArr = newintArr;
    System.out.println("reassigned intarr...sleeping");
    try {Thread.currentThread().sleep(1000);}catch(InterruptedException e){}
    System.out.println("exiting sync block in push");
   }
   intArr[index] = val;
   index++;
  }
 }
```

Stack.java

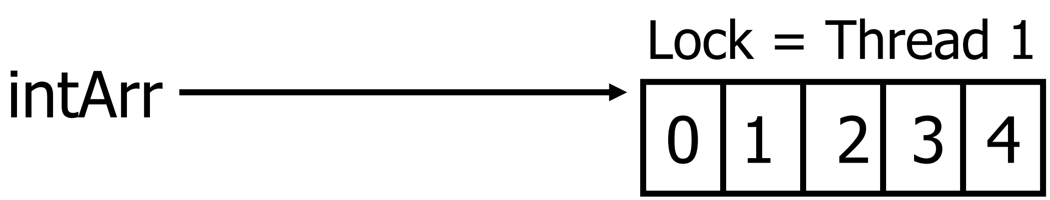
```
public int pop() throws EmptyStackException
 {
  int retval;
  synchronized(intArr) {
   System.out.println("inside synchronized block in pop");
   if (index > 0)
   {
    retval = intArr[index-1];
    index--;
    System.out.println("exiting synchronized block in pop");
    return retval;
   }
   System.out.println("exiting synchronized block in pop");
  }
  throw new EmptyStackException();
 }
}
```

Changing the Reference of a Locked Object

- This presents a dangerous situation
- The code synchronizes on the integer array object – intArr
  - This reference is reassigned inside of the push() method

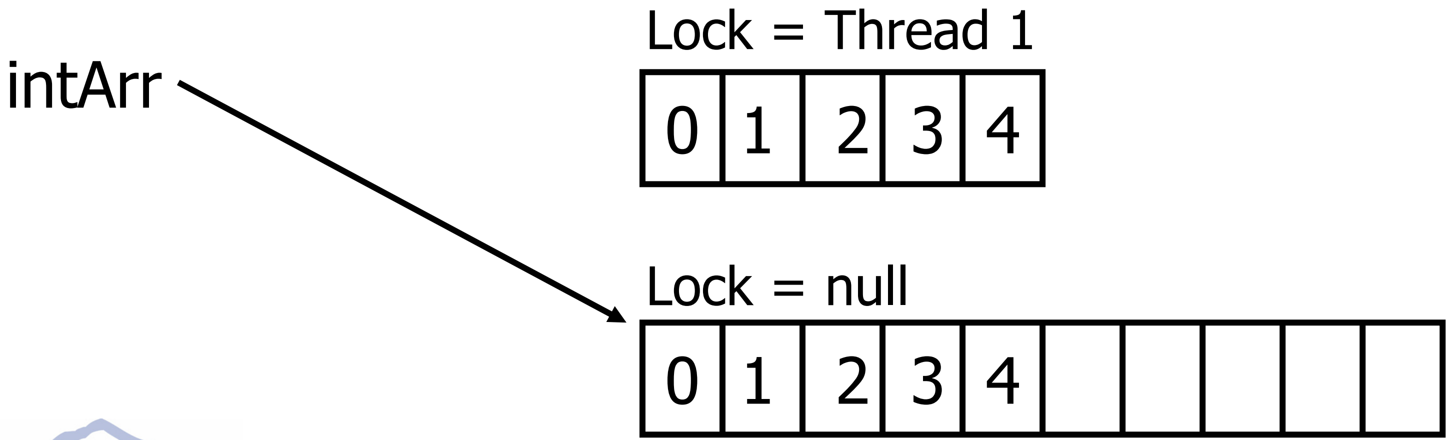
Changing the Reference of a Locked Object

- Thread 1 calls the push method and acquires the intArr object lock



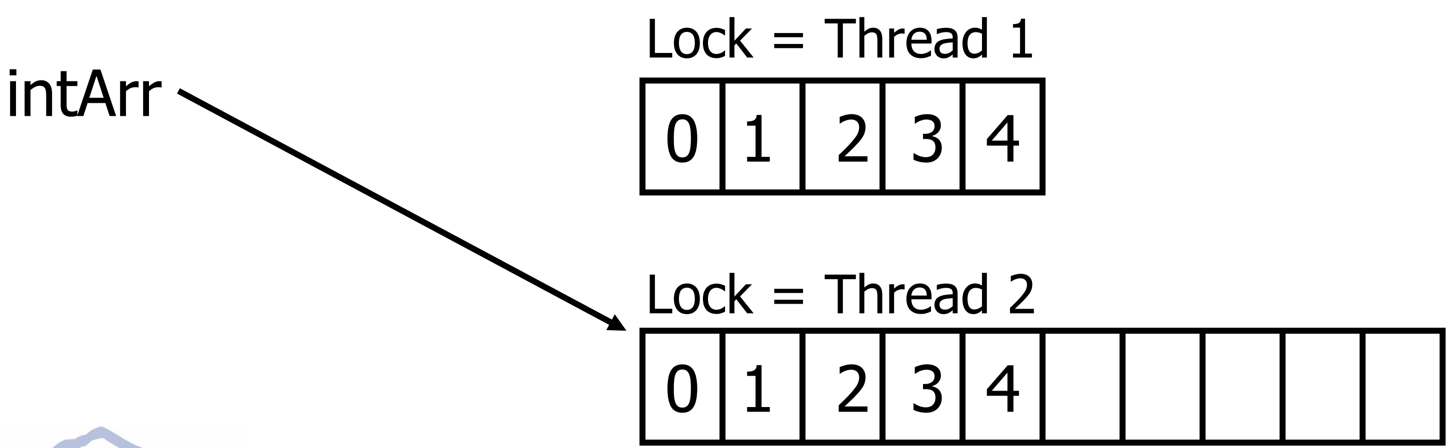
Changing the Reference of a Locked Object

- Thread 1 reallocates the array. The intArr variable now references a different object.
  - This new object is NOT locked
- Thread 1 is preempted by thread 2



Changing the Reference of a Locked Object

- Thread 2 calls the pop method. This method acquires the lock on intArr
  - The push and pop methods are executing concurrently



Changing the Reference of a Locked Object

- The previous code is fixed by declaring both the push and pop methods as synchronized
  - public synchronized void push(int val)
  - public synchronized int pop()
- …and removing the synchronized block from push()

Keyword – Synchronized

- The synchronized keyword is often described as a:
  - Mutex
  - Critical section
- Some programmers tend to believe that code in a synchronized method or block can only be executed by one thread at a time
  - This is not true
  - Synchronization locks objects, not methods or code
- Code in synchronized methods or blocks can be executed by multiple threads concurrently

Keyword – Synchronized

- For code protected by a synchronized method or block not to be executed by multiple threads concurrently
  - The code must be synchronized on the same object or class object (for static methods)

Keyword – Synchronized

- What if you invoke a synchronized static method and a synchronized instance method concurrently?
  - Will one block the other?
- The methods are not mutually exclusive
- They acquire two different locks

Keyword – Synchronized Deficiencies

- Can't lock multiple objects
- When blocking with synchronized
  - You must wait forever
  - There are no timeout facilities
  - Calling interrupt() won't help

Keyword – Synchronized Deficiencies

- To fix this you must write your own mutex class with a timeout facility
- If you need this functionality, see the book, "Taming Java™ Threads" in References

Double-Checked Locking

Double-Checked Locking

- Common singleton creation idiom
  - See \dcl\SingleThread\Singleton.java
- The simple multithreaded singleton version
  - See \dcl\MultiThread\Singleton.java
  - For all invocations except the first, no synchronization is needed

Double-Checked Locking

- In an effort to make the multithreaded singleton implementation more efficient
  - Double-checked locking was born
  - See \dcl\dblchk\Singleton.java
- Different idioms exist
  - Described in various books/articles
  - One problem… None of them are guaranteed to work

Singleton.java

```
import java.util.*;
class Singleton
{
  private static Singleton instance;
  private Vector v;
  private boolean inUse;

  private Singleton()
  {
    v = new Vector(100);
    //populate Vector with a value
    v.addElement(new Object());
    inUse = true;
  }

  public static Singleton getInstance()
  {
    if (instance == null)
    {
      synchronized(Singleton.class) {
        if (instance == null)
          instance = new Singleton();
      }
    }
    return instance;
  }
}
```

Double-Checked Locking

- Can break on single processor machines
- Can break on multi-processor machines
- The problem with the previous DCL code is:
  - Breaks the rule
    - Accesses a shared variable without protection
    - Can cause and Out of Order Write condition

Double-Checked Locking

- Out of order writes

```
//Broken double-checked locking code
public static Singleton getInstance()
{
 if (instance == null)
 {
  synchronized(Singleton.class) {
   if (instance == null)
    instance = new Singleton();  //1
  }
 }
 return instance;
}
```

Double-Checked Locking

- What can happen?
  - Thread 1 and Thread 2 call getInstance() concurrently
  - Thread 1 makes *instance* non-null but **before** the ctor executes (at //1)
    - Out of order writes
  - Thread 1 is interrupted at //1
  - Thread 2 executes and because *instance* is non-null, returns an object with default values instead of values set in ctor

Double-Checked Locking

- Pseudo code for
  - instance = new Singleton();

```
mem = allocateSomeMemory for Singleton object
instance = mem;  //Note that instance is now non-null,
                 //but has not been initialized
ctorSingleton(instance);  //Invoke ctor for Singleton
                          //passing instance
```

Older Symantec JIT compilers generate code like the above. This is perfectly legal code.

Double-Checked Locking

- To prove this, run relevant code in an infinite loop
- Break its execution with the MS VC++ debugger
  - Examine the assembler – code generated by the JIT
- See \dcl\jit\test.java

JDK 1.2 JIT assembler code

- See \dcl\jit\getInstance.asm
- This clearly shows how, in a multithreaded environment, the getInstance() method could return a Singleton object whose constructor has not been run
  - Its instance variables would be their default values
  - Not the values set in the constructor

getInstance.asm

```
;asm code generated for getInstance
054D20B0   mov    eax,[049388C8]          ;load instance ref
054D20B5   test   eax,eax                 ;test for null
054D20B7   jne    054D20D7
054D20B9   mov    eax,14C0988h
054D20BE   call   503EF8F0                ;allocate memory
054D20C3   mov    [049388C8],eax          ;store pointer in
                                          ;instance ref. instance
                                          ;non-null and ctor
                                          ;has not run
054D20C8   mov    ecx,dword ptr [eax]
054D20CA   mov    dword ptr [ecx],1       ;inline ctor - inUse=true;
054D20D0   mov    dword ptr [ecx+4],5     ;inline ctor - val=5;
054D20D7   mov    ebx,dword ptr ds:[49388C8h]
054D20DD   jmp    054D20B0
```

Double-Checked Locking

- Possible fix:

```
public static Singleton getInstance()
{
  if (instance == null)
  {
    synchronized(Singleton.class) {
      Singleton inst = instance;
      if (inst == null)
      {
        synchronized(Singleton.class) {
          inst = new Singleton();
        }
        instance = inst;  //1
      }
    }
  }
  return instance;
}
```

Double-Checked Locking

- This code tries to force a memory barrier with the inner synchronized block
  - Memory barriers at block entry and exit
- Memory barrier should prevent the reordering of the initialization of the Singleton object and the assignment of the *instance* field
  - Achieve memory barrier with synchronized block

Double-Checked Locking

- This won't work
  - Legal for compiler to move code at //1 into synchronized block
    - Now you have the same problem as before
  - JLS doesn't allow code to be moved out of synchronized blocks, but does allow code to be moved INTO synchronized blocks

Double-Checked Locking

```
public static Singleton getInstance()
{
  if (instance == null)
  {
    synchronized(Singleton.class) {
      Singleton inst = instance;
      if (inst == null)
      {
        synchronized(Singleton.class) {
          //inst = new Singleton();
          instance = new Singleton();
        }
        //MOVED instance = inst;
      }
    }
  }
  return instance;
}
```

Another "Solution" That Won't Work

```
private volatile boolean initialized = false;
private static Singleton instance;

 public static Singleton getInstance()
 {
  if (instance == null || !initialized)        //1
  {
   synchronized (Singleton.class) {
    if (instance == null)
     instance = new Singleton();
   }
   initialized = (instance != null);
  }
  return instance;
 }
```

Another "Solution" That Won't Work

- Tries to tie the value of initialized to instance
  - initialized = (instance != null);
    - Again, this statement could be moved into the synchronized block
    - initialized could be flushed to main memory BEFORE instance
      - ✓ Note that initialized is not accessed within a synchronized block at //1
      - ✓ If it were, there would not be a problem, but it wouldn't be DCL either and hence, not a solution

Double-Checked Locking

- Two solutions for static singleton
  - Accept synchronization
    - See \dcl\MultiThread\Singleton.java
  - Forgo synchronization with a static field
    - See \dcl\StaticSingle\Singleton.java

Singleton.java

```
class Singleton
{
  private Vector v;
  private boolean inUse;
  private static Singleton instance = new Singleton();

  private Singleton()
  {
    v = new Vector(100);
    //populate Vector with a value
    v.addElement(new Object());
    inUse = true;
  }

  public static Singleton getInstance()
  {
    return instance;
  }
}
```

Double-Checked Locking

- Synchronized keyword used not only for locking objects
  - Used to force visibility and ordering

Double-Checked Locking

- Some JITs/Hotspot emit code that appears it will work
  - IBM JDK 1.3, 1.4 (Windows)
  - Sun JDK 1.3, 1.4 (Windows)
  - See \dcl\jit\correct\getInstance.asm
- There are even more reasons why this code might not work

Double-Checked Locking

- Many programmers try to fool the compiler into generating "correct" DCL code
  - None of it will work
  - There are more subtle reasons why DCL fails
    - Temporary variables can be optimized away
    - Statements outside of synchronized blocks can be moved into synchronized blocks
    - Processor and cache can affect the order that threads see memory updates

Double-Checked Locking – Summary

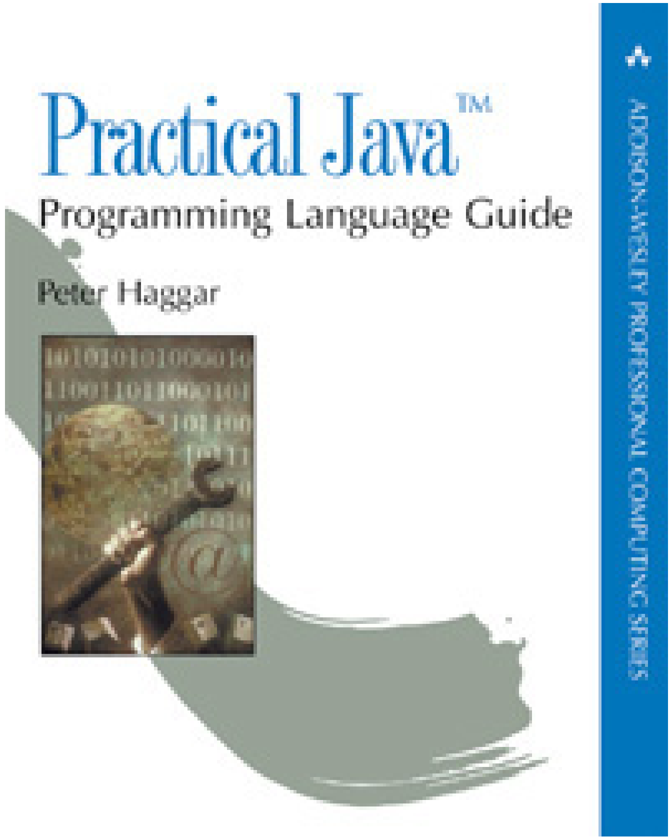
- Double-checked locking fails for many reasons
- Main point:
  - Java programs do not necessarily execute sequentially in a predictable order
    - Operations can occur in parallel or in a non-obvious order
    - These things are done in the name of performance

Bottom Line

- To avoid memory problems in your code:
  - Synchronize access to all variables that might have been written by another thread
  - Synchronize access to all variables that can be read by another thread
- OR
  - Forgo synchronization and declare such variables volatile
    - 32-bit variables only!
    - Consider using volatile for 64-bit variables only if your JVM implements volatile correctly
      - ✓ Otherwise, use synchronization

References

- *Practical Java™ Programming Language Language Guide*, Peter Haggar, Addison-Wesley, 2000, ISBN: 0-201-61646-7



References

- "Does Java Guarantee Thread Safety" – May 2002 *Dr. Dobb's Journal* by Peter Haggar
- "Double-checked locking and the Singleton Pattern" – IBM DeveloperWorks Web site by Peter Haggar
  http://www-106.ibm.com/developerworks/java/library/j-dcl.html
- *Taming Java™ Threads*, Allen Holub, Apress, 2000, ISBN 1-893115-10-0
- *Effective Java™ Programming Language Guide*, Joshua Bloch, Addison-Wesley, 2001, ISBN 0-201-31005-8

References

- http://www.javaworld.com/javaworld/jw-05-2001/jw-0525-double.html
  - Very good article by Brian Goetz
- http://www.cs.umd.edu/~pugh/java/memoryModel
  - Java Memory model discussion
  - Includes link to double-checked locking paper
    - http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html
- http://jcp.org/jsr/detail/133.jsp
  - JSR 133 is for the revision of the Java Memory Model

References

- Article on Specific Notification by Peter Haggar –
  http://www-106.ibm.com/developerworks/java/library/j-spnotif/
- "Multithreaded Exception Handling in Java" – August 1998 - *Java Report Magazine* by Peter Haggar and Joe DeRusso