Taming Java Threads

Allen I. Holub

Holub Associates

www.holub.com

allen@holub.com

What We'll Do Today

- Programming threads in Java is fraught with peril, but is mandatory in a realistic program.
- This talk discusses traps and pitfalls, along with some solutions
- This talk focuses on material not covered in most books

Who is the Guy?

- Talk based on my JavaWorld™ "Java Toolbox" column, now a book:
  - *Taming Java™ Threads*, (Berkeley: APress, 2000; http://www.apress.com).
- Source code, etc., found at http://www.holub.com.
- My Prejudices and Bias
  - I do not work for Sun
  - I have opinions and plan to express them. The appearance of impartiality is always just appearance
  - Java is the best thing since sliced bread (but bakery bread is better than sliced).

I'm assuming that...

- I'm assuming you know:
  - the language, including inner classes.
  - how to create threads using **`Thread`** and **`Runnable`**
  - **`synchronized`**, **`wait()`**, **`notify()`**
  - the methods of the **`Thread`** class.
- You may still get something out of the talk if you don't have the background, but you'll have to stretch

We'll look at

- Thread creation/destruction problems
- Platform-dependence issues
- Synchronization & Semaphores (**`synchronized`**, **`wait`**, **`notify`**, etc.)
- Memory Barriers and SMP problems
- Lots of other traps and pitfalls
- A catalog of class-based solutions
- An OO-based architectural solution

Books, etc.

- Allen Holub, *Taming Java™ Threads*. Berkeley, APress, 2000.
- Doug Lea. *Concurrent Programming in Java™: Design Principles and Patterns, 2nd Ed.:*. Reading: Addison Wesley, 2000.
- Scott Oaks and Henry Wong. *Java™ Threads*. Sebastopol, Calif.: O'Reilly, 1997.
- Bill Lewis and Daniel J. Berg. *Threads Primer: A Guide to Multithreaded Programming*. Englewood Cliffs: Prentice Hall/SunSoft Press, 1996.
- http://developer.java.sun.com/developer/technicalArticles /Threads/

Words to live by

All nontrivial applications for the Java™ platform are multithreaded, whether you like it or not.

It's not okay to have an unresponsive UI.

It's not okay for a server to reject requests.

Threads vs. processes

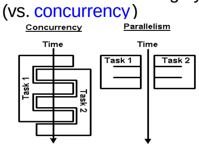
- A *Process* is an address space.
- A *Thread* is a flow of control through that address space.
  - Threads share the process's memory
  - Thread context swaps are much lower overhead than process context swaps

Threads vs. processes in Java

- A process is a JVM instance.
  - The Process contains the heap (everything that comes from **`new`**)
  - The heap holds all static memory
- A thread is a runtime (JVM) state
  - The "Java Stack" (runtime stack)
  - Stored registers
  - Local variables
  - Instruction pointer
- *Thread-safe* code can run in a multithreaded environment
  - Must synchronize access to resources (eg. memory) shared with other threads or be reentrant.
  - Most code in books isn't thread safe

Thread behavior is platform dependent!

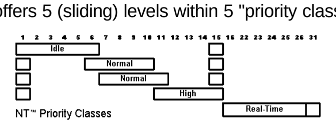
- You need to use the OS threading system to get parallelism (vs. concurrency)



- Different operating systems use different threading models (more in a moment).
- Behavior often based on timing.
- Multithreaded apps can be slower than single-threaded apps (but be better organized)

Priorities

- The Java™ programming language has 10 levels
  - but they're worthless---there are no guarantees that the OS will pay any attention to them.
- The Solaris™ OS has 2^{31} levels
- NT™ offers 5 (sliding) levels within 5 "priority classes."



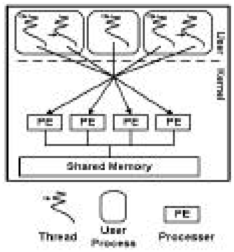
NT™ Priority Classes

- NT priorities change by magic.
  - After certain (unspecified) I/O operations priority is boosted (by an indeterminate amount) for some (unspecified) time.
  - Stick to **`Thread.MAX_PRIORITY`**, **`Thread.NORM_PRIORITY`**, **`Thread.MIN_PRIORITY`**)

Threading models

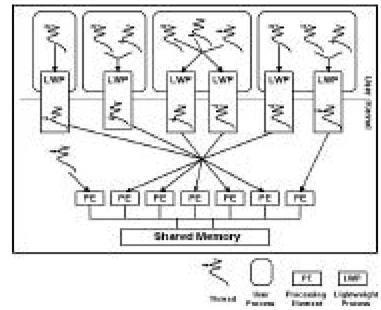
- *Cooperative* (Windows 3.1)
  - A Thread must voluntarily relinquish control of the CPU.
  - Fast context swap, but hard to program and can't leverage multiple processors.
- *Preemptive* (NT)
  - Control is taken away from the thread at effectively random times.
  - Slower context swap, but easier to program and multiple threads can run on multiple processors.
- *Hybrid* (Solaris™ OS, Posix, HPUX, Etc.)
  - Simultaneous cooperative and preemptive models are supported.

NT™ threading model



(Win32 "fibers" are so poorly documented, and so buggy, they are not a real option.)

Solaris™ OS threading model



Do not assume a particular environment

- Assume both of these rules, all the time:
  1. A thread can prevent other threads from running if it doesn't occasionally *yield*
     - by calling **`yield()`**, performing a blocking I/O operation, etc.
  2. A thread can be preempted at any time by another thread
     - even by one that appears to be lower priority than the current one.

Thread creation

- Java's **`Thread`** class isn't (a thread).
  - It's a thread controller

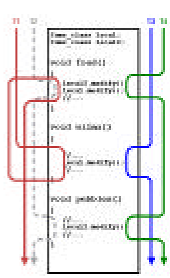
```
class Operation implements Runnable
{   public void run()
    {   // This method (and the methods it calls) are
        // the only ones that run on the thread.
    }
}

Thread thread_controller=new Thread(new Operation());
thread_controller.start();
```


Java™ threads aren't object oriented (1)

- Simply putting a method in a **`Thread`** derivative *does not* cause that method to run on the thread.
  - A method runs on a thread <u>only</u> if it is called from **`run()`** (directly or indirectly).

```
class Fred extends Thread
{   public void run()
    {   // This method (and the methods it calls) are
        // the only ones that run on the thread.
    }
    public foo()
    {   // This method will not run on the thread since
        // it isn't called by run()
    }
}
```


Java™ threads aren't object oriented (2)



- Objects do not run on threads, methods do.
- Several threads can send messages to the same object simultaneously.
  - They execute the same code with the same **`this`** reference, so share the object's state.

Basic concepts: atomic operations (atomicity).

- Atomic operations can't be interrupted (divided)
- Assignment to double or long is not atomic

```
long x;
```

thread 1:

```
x = 0x0123456789abcdef
```

thread 2:

```
x = 0;
```

possible results:

```
0x0123456789abcdef;
0x0123456700000000;
0x0000000089abcdef;
0x0000000000000000;
```

64-bit assignment is effectively implemented as:

```
x.high = 0x01234567
x.low  = 0x89abcdef;
```

You can be preempted between the assignment operations.

Basic concepts: synchronization

- Mechanisms to assure that multiple threads:
  - Start execution at the same time and run concurrently ("*condition variables*" or "events").
  - Do not run simultaneously when accessing the same object ("*monitors*" implemented with A"mutex").
  - Do not run simultaneously when accessing the same code ("*critical sections*").
- The **`synchronized`** keyword is essential in implementing synchronization, but is poorly designed.
  - e.g. No timeout, so deadlock detection is impossible.

Basic concepts: semaphores

- A *semaphore* is any object that two threads can use to synchronize with one another.
  - Don't be confused by Microsoft™ documentation that (incorrectly) applies the word "semaphore" only to a Dijkstra counting semaphore.
- Resist the temptation to use a Java native interface (JNI) call to access the underlying OS synchronization mechanisms.

The mutex (mutual-exclusion semaphore)

- The mutex is the key to a *lock*
  - Though it is sometimes called a "lock."
- Ownership is the critical concept
  - To cross a **`synchronized`** statement, a thread must have the key, otherwise it *blocks* (is suspended).
  - Only one thread can have the key (own the mutex) at a time.
- Every **`Object`** contains an internal mutex:

```
Object mutex = new Object();
synchronized( mutex )
{   // guarded code is here.
}
```

  - Arrays are also objects, as is the **`Class`** object.

Monitors and airplane bathrooms

- A *monitor* is a body of code (not necessarily contiguous), access to which is guarded by a single mutex.
  - Every object has its own monitor (and its own mutex).
- Think "airplane bathroom"
  - Only one person (thread) can be in it at a time (we hope).
  - Locking the door acquires the associated mutex. You can't leave without unlocking the door.
  - Other people must line up outside the door if somebody's in there.
  - Acquisition is not necessarily FIFO order.

Synchronization with individual locks

- Monitors create atomicity by using mutual-exclusion semaphores.
- Enter the monitor by passing over the **`synchronized`** keyword (acquire the mutex).
- Entering the monitor does not restrict access to objects used inside the monitor—it just prevents other threads from entering the monitor.

```
long field;
Object lock = new Object();

synchronized(lock)
{   field = new_value
}
```

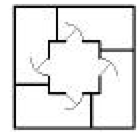
Method-level synchronization

```
class Queue
{   public synchronized void enqueue(Object o)
    { /*…*/ }
    public synchronized Object dequeue()
    { /*…*/ }
}
```

- **The monitor is associated with the object, not the code.**
  - Two threads can happily access the same synchronized code at the same time, provided that different objects receive the request.
  - E.g. Two threads can enqueue to different queues at the same time, but they cannot simultaneously access the same queue:
  - Same as `synchronized(this)`

He came in the Bathroom Window.

- The Bathroom can have several doors
- Acquiring a lock on an object does not prevent other threads from modifying that object.



```
class Bathroom_window
{   private double guard_this;

    public synchronized void ringo(double some_value)
    {   guard_this = some_value;
    }

    public double george()  // WRONG! Needs
    {   return guard_this;  // synchronization
    }
}
```


Constructors can't be synchronized, so always have back doors.

```
class Unpredictable
{   private final int x;
    private final int y;

    public Unpredictable(int init_x, int init_y)
    {  new Thread()
       {   public void run()
           {  System.out.println("x=" + x + " y=" + y);
           }
       }.start();

       x = init_x;
       y = init_y;
    }
}
```

- Putting the thread-creation code at the bottom doesn't help (the optimizer might move it).

Locking the constructor's back door.

```
class Predictable
{  Object lock = new Object();
   public Predictable(int init_x, int init_y)
   {  synchronized( lock )
      {  new Thread()
         {   public void run()
             {   synchronized( lock )
                 { // Use shared var
                 }
             }
         }.start();
         //initialize shared var.
      }
   }
}
```

- `synchronized(this)` **<u>does not work</u> in a constructor. (It's a silent no-op.)**

Be careful to lock the correct object

- An inner-class event handler is also a back door

```
class Outer
{  private double   d;
   private JButton  b = new JButton();
   public Outer()
   {  b.addActionListener
      (  new ActionListener()
         {  public void actionPerformed(ActionEvent e)
            {  d = 0.0; // race condition!
            }
         }
      );
   }
   public void race_condition(double new_value)
   {  d = new_value;
   }
}
```


Synchronizing the inner-class method doesn't work

```
class Outer
{  private double   d;
   private JButton  b = new JButton();
   public Outer()
   {  b.addActionListener
      (  new ActionListener()
         {  synchronized  // grabs the wrong lock!
            public void actionPerformed(ActionEvent e)
            {  d = 0.0;
            }
         }
      );
   }
   public void race_condition(double new_value)
   {  d = new_value;
   }
}
```

Explicitly synchronize on the object that holds the contested fields.

```
class Outer
{   private double   d;
    private JButton  b = new JButton();
    public Outer()
    {   b.addActionListener
        (   new ActionListener()
            {   public void actionPerformed(ActionEvent e)
                {   synchronized( Outer.this )
                    {   d = 0.0;
                    }
                }
            }
        );
    }
    synchronized
    public void race_condition(double new_value)
    {   d = new_value;
    }
}
```


Synchronization isn't cheap

```
class Synch
{   synchronized int locking      ( int a, int b )
                                  { return a + b;}
    int              not_locking  ( int a, int b )
                                  { return a + b;}
   static public void main(String[] arguments)
   {  double start = new Date().get Time();

      for(long i = 1000000; --i >= 0 ;)
         tester.locking(0,0);

      double end = new Date().getTime();

      double locking_time = end - start;
      // repeat for not_locking
   }
}
```


Synchronization isn't cheap

```
% java -verbose:gc Synch
Pass 0: Time lost: 234 ms.     121.39% increase
Pass 1: Time lost: 139 ms.     149.29% increase
Pass 2: Time lost: 156 ms.     155.52% increase
Pass 3: Time lost: 157 ms.     155.87% increase
Pass 4: Time lost: 157 ms.     155.87% increase
Pass 5: Time lost: 155 ms.     154.96% increase
Pass 6: Time lost: 156 ms.     155.52% increase
Pass 7: Time lost: 3,891 ms. 1,484.70% increase
Pass 8: Time lost: 4,407 ms. 1,668.33% increase
```

200MHz Pentium, NT4/SP3, JDK 1.2.1, HotSpot 1.0fcs, E

- Contention in last two passes (Java Hotspot can't use atomic-bit-test-and-set).

Synchronization isn't cheap

BUT

- The cost of stupidity is always higher than the cost of synchronization. *(Bill Pugh)*
  - Pick a fast algorithm.
- Overhead can be insignificant when the synchronized method is doing a time-consuming operation.
  - But in OO systems, small synchronized methods often chain to small synchronized methods.

Reentrant Code

- Reentrant code doesn't need to be synchronized.
  - Code that uses only local variables and arguments (no **`static`** variables, no fields in the class).
- Consider having a synchronized non-reentrant **`public`** method call a reentrant **`private`** method.
  - used values are stale, though.

```
Object some_field = new Some_class();
public synchronized void accessor()
{  workhorse( some_field.clone() );
}
private void workhorse( long some_field )
{  // no fields of class are used in here.
}
```


Volatile

- Atomic operations on **`volatile`** primitive types often do not need to be synchronized.
  - **`volatile`** might not work in all JVMs (HotSpot is okay.)
  - Assignment to all non-64-bit things, including **`boolean`**s and references are usually safe.
  - Assignment to **`volatile double`**s and **`float`**s should be atomic (but most JVMs don't do it).
  - Code may be reordered, so assignment to several atomic variables must be synchronized.

Using Volatile Safely

- One-writer, many-reader strategies are best.
  - But a change of state might not be immediately visible to other threads.
- Assignment to non-Boolean is risky.
  - Works if a single writer is simply incrementing (but the change might not be immediately visible).
  - Will not work if multiple threads perform updates.
- Do not depend on the "current" value of a volatile.
  - The value might change at surprising times.

Synchronization Rules of Thumb

- Don't synchronize on read-only access.
- Synchronize the smallest block possible to minimize the odds of contention.
  - Method-level synchronization should be avoided in very-high-performance systems.
- Don't synchronize the methods of classes that are called only from one thread.
  - Use Collection-style synchronization decorators when you need synchronized behavior.

```
Collection c = new ArrayList();
c = Collections.synchronizedCollection(c);
```


Don't Nest Synchronization

- Don't access **`synchronized`** methods from synchronized methods.
  - Synchronize **`public`** methods. Don't synchronize **`private`** ones.
- E.g.: Avoid **`Vector`** and **`Hashtable`** in favor of **`Collection`** and **`Map`** derivatives.
  - **`Vector`** and **`Hashtable`** access is synchronized, but **`Vector`** and **`Hashtable`** objects are usually used from within synchronized methods.
  - Collections and Maps accessors are not synchronized.

```
Collection c =
  Collections.synchronizedCollection(c);
```


Don't use Buffered Streams

- Avoid heavy use of **`BufferedInputStream`**, **`BufferedOutputStream`**, **`BufferedReader`**, or **`BufferedWriter`**
  - Single-byte access is synchronized!
    - How often do multiple threads simultaneously access the same stream at the byte level?
  - You might use **`write(byte[]), read(byte[]), etc.`**
- Best to roll your own version of **`BufferedOutputStream`** that's not synchronized.
  - You can copy the source and rename the class

Avoid `String` Concatentation and `StringBuffer` Objects.

- The **`StringBuffer`** class's **`append()`** method is synchronized!
- String concatenation uses a StringBuffer:

```
s1 = s2 + s3;
```

is really

```
Stringbuffer t0 = new StringBuffer(s2);
t0.append( s3 );
s1 = t0.toString();
```

- The only solution is not to use string operations or StringBuffers!

Don't use protected

- **No guarantee that derived classes correctly synchronize access to protected fields.**
- **synchronized is <u>not</u> part of the signature**
  - **This is a problem with `public` methods, too.**
  - **No guarantee that derived-class overrides synchronize properly:**

```
public class Foo
{ protected synchronized void f(){/*...*/}
}

class Bar extends Foo
{ protected void f() {/*...*/}  // AAGH!
}
```

Don't Use Finalizers

- **They slow down the garbage collector.**
- **May run while objects referenced by fields are still in use!**
- **Two different objects may be finalized simultaneously.**
  - **Could be disastrous if they share references.**

Do use Immutable objects

- **Synchronization not required (all access read-only).**
- **All fields of the object are final (e.g. String)**
  - *Blank* finals are **`final`** fields without initializers.
  - Blank finals must be initialized in all constructors.

```
class I_am_immutable
{ private final int some_field;
    public I_am_immutable( int initial_value )
    {   some_field = initial_value;
    }
}
```

  - Might not compile with inner classes (there's a long-standing compiler bug)
- Immutable ≠ constant (but it must be constant to be thread safe)
  - A **`final`** reference is constant, but the referenced object can change state.
  - Language has no notion of "constant", so you must guarantee it by hand

Critical sections

- A *critical section* is a body of code that only one thread can enter at a time.
- Do not confuse a critical section with a monitor.
  - The monitor is associated with an object
  - A critical section guards code
- The easiest way to create a critical section is by synchronizing on a **`static`** field:

```
static final Object critical_section = new Object();
synchronized( critical_section )
{   // only one thread at a time
    // can execute this code
}
```


Critical sections can also synchronize on the class object

```
class Flintstone
{ public void fred()
  {   synchronized( Flintstone.class )
      {  // only one thread at a time
         // can execute this code
      }
  }

  public static synchronized void wilma()
  { // synchronizes on the same object
    // as fred().
  }
}
```


Class vs. instance variables

- All **`synchronized static`** methods synchronize on the *same* monitor.
- Think *class variables* vs. *instance variables*:
  - The class (**`static`**) variables and methods are effectively members of the **`Class`** object.
  - The class (**`static`**) variables store the state of the class as a whole.
  - The class (**`static`**) methods handle messages sent to the class as a whole.
  - The instance (non-**`static`**) variables store the state of the individual objects.
  - The instance (non-**`static`**) methods handle messages sent to the individual objects.

But remember the bathroom with multiple doors

```
class Foo
{  static long x = 0;
   synchronized static void set_x( long x )
   { Foo.x = x;
   }
   synchronized /* not static */ double get_x()
   { return x;
   }
}
```

Thread 1:

```
Foo o1 = new Foo();
long x = o1.get_x();
```

Thread 2:

```
Foo.set_x(-1);
```

Results are <u>undefined</u>. (There are two locks here, one on the class object and one on the instance.)

Lock the extra doors

1. Synchronize explicitly on the class object when accessing a **`static`** field from an instance method.

```
class Okay
{  private static long unsafe;
   public void foo(long x)
   { //...
     synchronized( Okay.class )
     {  unsafe = x;
     }
   }
}
```


Lock the extra doors

2. Access all **`static`** fields through **`synchronized static`** methods, even if the accessor is a method of the class that contains the field.

```
class Okay
{  private static long unsafe;
   private static synchronized get()
   {return unsafe;}
   private static synchronized set(long x)
   {unsafe = x;}

   public /*not static*/ void foo(long x)
   {  //...
      set(x);
   }
}
```


Lock the extra doors

3. Encapsulate all **`static`** fields in an inner class and provide exclusive access through **`synchronized`** methods of the inner class.

```
class Okay
{  private static class Class_Variables
   {  private long unsafe;
      public synchronized void do_something(long x)
      {  unsafe = x;  //. . .
      }
   }
   static Class_Variables statics =
                           new Class_Variables();
   public foo(long x)
   {  statics.do_something( x );
   }
}
```


Singletons (one-of-a-kind objects)

- Singletons often use critical sections for initialization.

```
public final class Singleton
{  static
   {   new JDK_11_unloading_bug_fix(Singleton.class);
   }
   private static Singleton instance;
   private Singleton(){} // prevent creation by new

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


Avoiding sychronization in a singleton by using static

- A degraded case, avoids synchronization.

```
public final class Singleton
{  static
   { new JDK_11_unloading_bug_fix(Singleton.class);
   }
   private Singleton(){}

   private static final Singleton instance
                                   = new Singleton();

   public
   /*unsynchronized*/ static Singleton instance()
   {  return instance;
   }
}
```


Or alternatively…

- Thread safe because VM loads only one class at a time and method can't be called until class is fully loaded and initialized.
- No way to control constructor arguments at run time.

```
public final class Singleton
{  private static Singleton instance;
   private Singleton(){}

   static{ instance = new Singleton(); }

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

While we're on the subject…

```
public class JDK_11_unloading_bug_fix
{ public JDK_11_unloading_bug_fix(final Class keep)
  { if (System.getProperty("java.version")
                              .startsWith("1.1") )
    { Thread t =  new Thread()
      {  public void run()
         {  Class singleton_class = keep;
            synchronized(this)
            {  try{ wait();}
               catch(InterruptedException e){}
            }
         }
      };
      t.setDaemon(true);
      t.start();
    }
  }
}
```

In the 1.1 JDK™ All objects not accessible via a local-variable or argument were subject to garbage collection

Condition variables

- All objects have a "condition variable" in addition to a mutex.
  - A thread blocks on a condition variable until the condition becomes true.
  - In the Java™ environment, conditions are "pulsed" — condition reverts to false immediately after waiting threads are released.
- `wait()` and `notify()` use this condition variable.

`wait` and `notify` have problems.

- Implicit condition variables don't stay set!
  - A thread that comes along after the `notify()` has been issued blocks until the next `notify()`.
- `wait(timeout)` does not tell you if it returned because of a timeout or because the wait was satisfied (hard to solve).
- There's no way to test state before waiting.
- `wait()` releases only one monitor, not all monitors that were acquired along the way (nested monitor lockout).

`Notifying_queue()`: `wait()`, `notify()`, and spin locks

```
class Notifying_queue
{   private static final queue_size = 10;
    private Object[]     queue = new Object[queue_size];
    private int          head  = 0;
    private int          tail  = 0;
    public void synchronized enqueue( Object item )
    {   queue[++head %= queue_size] = item;
        this.notify();
    }
    public Object synchronized dequeue( )
    {   try
        {    while( head == tail) //<-- MUST BE A WHILE
               this.wait();       //    (NOT AN IF)
        }
        catch( InterruptedException e )
        {    return null; // wait abandoned
        }
        return queue[++tail %= queue_size ];
    }
}
```


Condition variables. wait is not atomic (1)

```
                                                              T1
synchronized enqueue(. . .) ....this.mutex.acquire();
{
   this.notify(); ............... this.condition.set_true();
}............................... this.mutex.release();
synchronized dequeue(. . .) .... this.mutex.acquire();
{
   while( head == tail ) ........ while( head==tail )
      this.wait(); .................. this.mutex.release();
                                      this.condition.wait_for_true();
                                      this.mutex.acquire();
}............................... this.mutex.release()
```


Condition variables. wait is not atomic (2)

```
                                                              T1
synchronized enqueue(. . .) ....this.mutex.acquire();     T2
{
   this.notify(); ............... this.condition.set_true();
}............................... this.mutex.release();
synchronized dequeue(. . .) .... this.mutex.acquire();
{
   while( head == tail ) ........ while( head==tail )
      this.wait(); .................. this.mutex.release();
                                      this.condition.wait_for_true();
                                      this.mutex.acquire();
}............................... this.mutex.release()
```

Condition variables. wait is not atomic (3)

T1 T2

```
synchronized enqueue(. . .)  ....  this.mutex.acquire();
{
    this.notify();  ...........  this.condition.set_true();
}  .........................  this.mutex.release();

synchronized dequeue(. . .)  ....  this.mutex.acquire();
{
    while( head == tail )  .......  while( head==tail )
        this.wait();  ...........  this.mutex.release();
                                   this.condition.wait_for_true();
                                   this.mutex.acquire();
}  .........................  this.mutex.release()
```


Condition variables. wait is not atomic (4)

T1 T3 T2

```
synchronized enqueue(. . .)  ....  this.mutex.acquire();
{
    this.notify();  ...........  this.condition.set_true();
}  .........................  this.mutex.release();

synchronized dequeue(. . .)  ....  this.mutex.acquire();
{
    while( head == tail )  .......  while( head==tail )
        this.wait();  ...........  this.mutex.release();
                                   this.condition.wait_for_true();
                                   this.mutex.acquire();
}  .........................  this.mutex.release()
```


Condition variables. wait is not atomic (5)

T1 T3 T2

```
synchronized enqueue(. . .)  ....  this.mutex.acquire();
{
    this.notify();  ...........  this.condition.set_true();
}  .........................  this.mutex.release();

synchronized dequeue(. . .)  ....  this.mutex.acquire();
{
    while( head == tail )  .......  while( head==tail )
        this.wait();  ...........  this.mutex.release();
                                   this.condition.wait_for_true();
                                   this.mutex.acquire();
}  .........................  this.mutex.release()
```


Summarizing `wait()` behavior

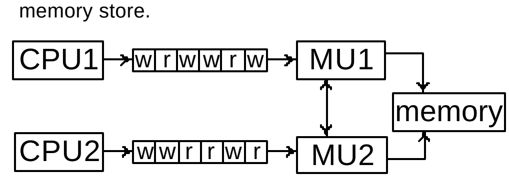
- `wait()` doesn't return until the notifying thread gives up the lock.
- A condition tested before entering a `wait()` may not be true after the `wait` is satisfied.
- There is no way to distinguish a timeout from a `notify()`.

Visibility

- Changes made by a CPU are not transferred from cache to the main memory store immediately.
- It may take time for a change made by one thread to become visible to another thread
  - Threads are running on different processors.
- The order in which changes become visible are not always the order in which the changes are made.

Beware of symmetric multi-processing (SMP) environments

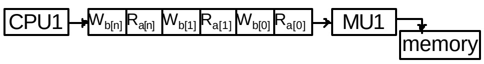
- The CPU does not access memory directly.
- CPU read/write requests are given to a "memory unit," which actually controls the movement (at the hardware level) of data between the CPU and main memory store.



Some common memory operations are inefficient

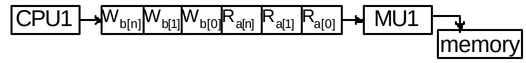
- Processors supporting a “relaxed memory model” can transfer blocks of memory between cache and the main memory store in undefined order!
- Consider:

```
int a[] = new int[10];
int b[] = new int[10];
for( int i = 0; i < a.length; ++i )
   b[i] = a[i];
```

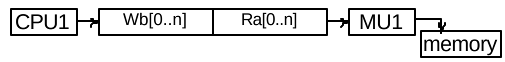


Presto Chango!

- The memory unit notices the inefficiency and rearranges the requests!



- To produce:



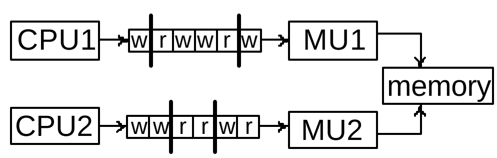
- This change is good—it speeds memory access.

BUT…

- The order in which changes are made in the source code may not be preserved at run time!
- The order in which changes are made may not be the order in which those changes are reflected in main memory.

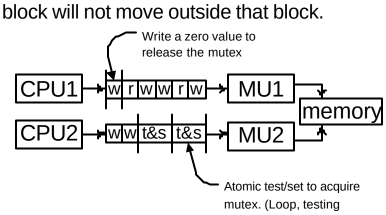
Don’t Panic

- Reordering doesn’t matter in single-threaded systems.
- Reordering not permitted across “memory barriers” (effectively inserted around **`synchronized`** access).



Memory barriers are created indirectly by synchronization

- **`synchronized`** is implemented using a memory barrier
  - so modifications made within a **`synchronized`** block will not move outside that block.



Avoiding synchronization (revisited)

- You cannot use **`volatile`** fields (e.g. **`boolean`**) to guard other code.

```
class I_wont_work
{  private volatile boolean okay  = false;
   private long             field = -1;
   //. . .
   public /*not synchronized*/ void wont_work()
   {   if( okay )
       { do something( field );
       }
   }
   public /*not synchronized*/ void enable()
   {  okay = false;
      field = 0;
      okay = true;
   }
}
```

Might be –1.

Even worse

- Memory modifications made in the constructor may not be visible, even though the object is accessible!

```
class Surprise
{   public long field;
    //. . .
    public Surprise()
    { field = -1;
    }
}
```

Modification of s might become visible before modification of `field` if memory unit rearranges operations.

Holds even if `field` is `final`!

Thread 1:
```
    Surprise s = new Surprise();
```

Thread 2:
```
    System.out.println(s.field);
```


Synchronization can fix things

- This works

```
Object lock = new Object();
```

Thread 1:
```
    synchronized( lock )
    { Surprised s = new Surprised();
    }
```

Thread 2:
```
    synchronized( lock )
    { System.out.println(s.get_field());
    }
```


The Memory Unit doesn't know the word "subroutine."

- All code between read/write requests are subject to reordering, whether or not they are called from a subroutine.

Double-checked locking doesn't work!

- Is unreliable even in single-CPU machine.

```
public final class Singleton
{   static{ new JDK_11_unloading_bug_fix(Std.class); }

    private static Singleton instance;
    private Singleton(){}    // prevent creation by new

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


This doesn't work

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


This doesn't work either

```
class Still_broken_singleton
{
    public static Singleton instance()
    {   if( instance == null )
        {   synchronized( Singleton.class )
            {    if( instance == null )
                 {    instance = factory();
                 }
            }
        }
        return instance;
    }
    // Synchronizing the following subroutine does
    // not affect the incorrect behavior.
    private void Singleton factory()
    {  return new Singleton();
    }
}
```

“Rules to live by” in an SMP environment (gotchas)

- To assure that shared memory is visible to two threads: the writing thread must give up a lock that is subsequently acquired by the reading thread.
- Modifications made while sleeping may not be visible after **`sleep()`** returns.
- Operations are not necessarily executed in source-code order (not relevant if code is synchronized.)
- ??? Modifications to memory made after a thread is created, but before it is started, may not be visible to the new thread.

“Rules to live by” in an SMP environment (things that work)

- Modifications made by a thread before it issues a **`notify()`** will be visible to the thread that’s released from the associated **`wait()`.**
- Modifications made by a thread that terminates are visible to a thread that joins the terminated thread. [must call **`join()`**]
- Memory initialized in a **`static`** initializer is safely accessible by all threads, including the one that caused the class-file load.

A few articles on SMP Problems

- Paul Jakubik (ObjectSpace):
  www.primenet.com/~jakubik/mpsafe/MultiprocessorSafe.pdf
- Bill Pugh (Univ. of Maryland) mailing list:
  www.cs.umd.edu/~pugh/java/memoryModel/
- Allen Holub:
  www.javaworld.com/javaworld/jw-02-2001/jw-0209-toolbox.html
- Brian Goetz:
  www.javaworld.com/javaworld/jw02-2001/jw-0209-double.html

Deadlock: The simplest scenario (1)

- Two or more threads, all waiting for each other.
- Threads trying to acquire multiple locks, but in different order.

Deadlock: The simplest scenario (2)

T1

```
double field1; Object lock1 = new Object();
double field2; Object lock2 = new Object();
public void pebbles()
{   synchronized(lock1){ field1 = 0; }
}
public void bambam()
{   synchronized(lock2){ field2 = 0; }
}
public void fred()
{   synchronized(lock2)
    {   synchronized(lock1)
        {   field2 += field1;
        }
    }
}
public void wilma()
{   synchronized(lock1)
    {   synchronized(lock2)
        {   field2 -= field1;
        }
    }
}
```


Deadlock: The simplest scenario (3)

T1 T2

```
double field1; Object lock1 = new Object();
double field2; Object lock2 = new Object();
public void pebbles()
{   synchronized(lock1){ field1 = 0; }
}
public void bambam()
{   synchronized(lock2){ field2 = 0; }
}
public void fred()
{   synchronized(lock2)
    {   synchronized(lock1)
        {   field2 += field1;
        }
    }
}
public void wilma()
{   synchronized(lock1)
    {   synchronized(lock2)
        {   field2 -= field1;
        }
    }
}
```

Deadlock: A more-realistic scenario

```
class Boss
{   private Sidekick robin;
    synchronized
    void set_side_kick(Sidekick kid)
    {   robin = kid; };
    synchronized void to_the_bat_cave()
    {                                       preempt
        robin.lets_go();
    }
    synchronized void report(String s)
    {/*...*/}
}
class Sidekick
{   private Boss batman;
    Sidekick(Boss boss){batman = boss;}
    synchronized void lets_go(){..}
    synchronized void sock_bam()
    {   batman.report("Ouch!");
    }
}
Boss     batman = new Boss();
Sidekick robin  = new Sidekick(batman);
batman.set_side_kick( robin );
```

1. Thread 1 (Alfred) calls batman.to_the_bat_cave(); Alfred now has the lock on batman.
2. Thread 1 is preempted just before calling lets_go().
3. Thread 2 (Joker) calls robin.sock_bam(). Joker now has the lock on robin.
4. Robin tries to report() to batman (on thread 2), but can't because Alfred has the lock. Joker is blocked.
5. Thread 1 wakes up, tries to call lets_go(), but can't because Joker has the lock.

Nested-monitor lockout

- Can happen any time you call a method that can block from any synchronized method.
- Consider the following (I've removed exception handling):

```
class Black_hole
{ private InputStream input =
               new Socket("www.holub.com",80)
                              .getInputStream();
  public synchronized int read()
  { return input.read();
  }
  public synchronized void close()
  { input.close();
  }
}
```

How do you close the socket?

Nested-monitor lockout: another example

- The notifying queue blocks if you try to dequeue from an empty queue

```
class Black_hole2
{   Notifying_queue queue =
              new Notifying_queue();

    public synchronized void put(Object thing)
    { queue.enqueue(thing);
    }

    public synchronized Object get( )
    { return queue.dequeue();
    }
}
```


Why was `stop()` deprecated?

- NT leaves DLLs (including some system DLLs) in an unstable state when threads are stopped externally.
- `stop()` causes all monitors held by that thread to be released,
  - but thread may be stopped half way through modifying an object, and
  - other threads can access the partially modified (now unlocked) object

Why was stop() deprecated (2)?

- The only way to safely terminate a thread is for **`run()`** to return normally.
- Code written to depend on an external **`stop()`** will have to be rewritten to use **`interrupted()`** or **`isInterrupted()`**.

`interrupt()`, don't `stop()`

```
class Wrong
{ private Thread t =
    new Thread()
    { public void run()
      { while( true )
        { //...
          blocking_call();
        }
      }
    };
  public stop()
  { t.stop();
  }
}
```

```
class Right
{ private Thread t =
  new Thread()
  { public void run()
    { try
      { while( !isInterrupted() )
        {  //...
          blocking_call();
        }
      }catch(InterruptedException e)
      {/*ignore, stop request*/}
    }
  };
  public stop()
  {t.interrupt();}
}
```

- But there's no safe way to stop a thread that doesn't check the "interrupted" flag.

interrupt() gotchas

- `interrupt()` works well only with the methods of the `Thread` and `Object` classes
  - `wait()`, `sleep()`, `join()`, etc.
  - It throws an `InterruptedException`
- Everywhere else `interrupt()` just sets a flag.
  - You have to test the flag manually all over the place.
  - Calling `interrupted()` clears the flag.
  - Calling `isInterrupted()` doesn't clear the flag!
- It is not possible to interrupt out of a blocking I/O operation like `read()`.
  - Would leave the stream in an undefined state.
  - Use the classes in java. nio whenever possible.

Why were suspend() and resume() deprecated?

- The suspend() method does not release the lock

```
class Wrong
{ public synchronized
  void take_a_nap()
  {   suspend();
  }
  public synchronized
  void wake_up()
  {   resume();
  }
}
```

Once a thread has entered **take_a_nap**(), all other threads will block on a call to **wake_up**(). (Someone has gone into the bathroom, locked the door, and fallen into a drug-induced coma)

```
class Right
{   public synchronized
   void take_a_nap()
   {  try
      {    wait();
      }
      catch(InterruptedException e)
      {/*do something reasonable*/}
   }
   public synchronized
   void wake_up()
   {   notify();
   }
}
```

The lock is released by **wait()** before the thread is suspended.

The big-picture coding issues

- Design-to-coding ratio is 10:1 in threaded systems.
- Formal code inspection or pair programming is essential.
- Debugging multithreaded code takes longer.
  - Bugs are usually timing related.
- It's not possible to fully debug multithreaded code in a visual debugger.
  - Instrumented JVMs cannot find all the problems because they change timing.
  - Classic Heisenberg uncertainty: observing the process impacts the process.
- Complexity can be reduced with architectural solutions (e.g. Active Objects).

Given that the best solution isn't finding a new profession…

- Low-level solutions (roll-your-own semaphores)
  - I'll look at a few of the simpler classes covered in depth in *Taming Java Threads*.
  - My intent is to give you a feel for multithreaded programming, not to provide an exhaustive toolkit.
- Architectural solutions (active objects, etc).

Roll your own (A Catalog)

- **Exclusion Semaphore** (mutex)
  - Only one thread can own at one time.
  - Roll-your-own version can contain a timeout.
- **Condition Variable**
  - Wait while condition false.
  - Roll-your-own version can have state.
- **Counting Semaphore**
  - Control pool of resources.
  - Blocks if resource is unavailable.

Roll your own (2)

- **Message Queues** (interthread communication)
  - Thread blocks (with wait/notify) until a message is enqueued.
  - Typically, only thread per queue.
- **Thread Pools**
  - A group of dormant threads wait for something to do.
  - A thread activates to perform an arbitrary task.
- **Timers**
  - Allow operation to be performed at regular intervals
    - Block until a predetermined time interval has elapsed
    - Block until a predetermined time arrives.

Roll your own (3)

- **Reader/Writer Locks**
  - Allow thread-safe access to global resources such as files:
    - Must acquire the lock to access a resource
    - Writing threads are blocked while a read or write operation is in progress
    - Reading threads are blocked only while a write operation is in progress. Simultaneous reads are okay

Threads from an OO perspective

- Think messages, not functions
- Synchronous messages—handler doesn't return until it's done doing whatever sender requests
- Asynchronous messages—handler returns immediately. Meanwhile request is processed in the background.
  `Toolkit.getDefaultToolkit().getImage(some_URL);`

The Java™-language threading model is not OO

- No language-level support for asynchronous messaging.
- Threading system is based entirely on procedural notions of control flow.
- Deriving from **`Thread`** is misleading
  - Novice programmers think that all methods of a class that extends **`Thread`** run on that thread, when in reality, the only methods that run on a thread are methods that are called either directly or indirectly by **`run()`**.

Implementing asynchronous methods —one thread per method

```
class Receiver
{  //. . .
   public asynch_method()
   { new Thread()
     {   public void run()
         {  synchronized( Receiver.this )
            {  // Make local copies of
               // outer-class fields here.
            }
            // Code here doesn't access outer
            // class (or uses only constants).
         }
     }.start();
   }
}
```


A more realistic one-thread-per-method example

```
// This class demonstrates an asynchronous flush of a
// buffer to an arbitrary output stream

class Flush_example
{  public interface Error_handler
   {   void error( IOException e );
   }
   private final OutputStream out;
   private final Reader_writer lock =
                                      new Reader_writer();
   private byte[]               buffer;
   private int                  length;

   public Flush_example( OutputStream out )
   { this.out = out;
   }
```


A more realistic one-thread-per-method example

```
   synchronized void flush( final Error_handler handler )
   {   new Thread()            // Outer object is locked
       {   byte[] copy;        // while initializer runs.
           { copy = new byte[Flush_example.this.length];
             System.arraycopy(Flush_example.this.buffer,
                    0, copy, 0, Flush_example.this.length]);
             Flush_example.this.length = 0;
           }
           public void run()          // Lock is released
           {   try                    // when run executes
               {   lock.request_write();
                   out.write( copy, 0, copy.length );
               }
               catch( IOException e ){ handler.error(e); }
               finally{ lock.write_accomplished(); }
           }
       }.start();
   }
}
```

Problems with one-thread-per-method strategy

- It is a worse-case synchronization scenario.
  - Many threads all access the same outer-class object simultaneously
  - Since synchronization is required, all but one of the threads are typically blocked, waiting to access the object.
- Thread-creation overhead can be stiff:

Create String = .0040 ms.
Create Thread = .0491 ms.
Create & start Thread = .8021 ms. (NT 4.0, 600MHz)

Use Thread Pools

- The real version:
  - Grows from the initial size to a specified maximum if necessary.
  - Shrinks back down to original size when extra threads aren't needed
  - Supports a "lazy" close.

```
public final class Simplified_Thread_pool
{   private Object startup_lock = new Object();
    private final Blocking_queue pool
                                = new Blocking_queue();
```


Implementing a simple thread pool

```
public Simplified_Thread_pool(int pool_size )
{   synchronized( startup_lock )
    {   while( --pool_size >= 0 )
            new Pooled_thread().start();
    }
}

public synchronized void execute(Runnable action)
{   pool.enqueue( action );
}

public synchronized void close()
{   pool.close();
}
```


Implementing a simple thread pool

```
private final class Pooled_thread extends Thread
{   public void run()
    {   synchronized( startup_lock )
        {}
        try
        {   while( !isInterrupted() )
                ((Runnable)pool.dequeue()).run();
        }
        catch(InterruptedException  e){/* ignore */}
        catch(Blocking_queue.Closed e){/* ignore */}
        catch(Throwable e)
        {   // handle unexpected error gracefully...
            e.printStackTrace();
        }
    }
  }
}
```

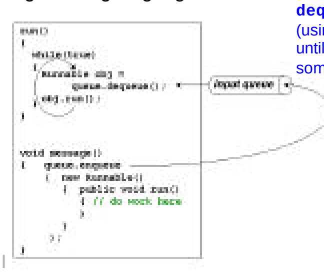

The *Active Object* design pattern

- An architectural solution to threading synchronization.
- Asynchronous requests are executed serially on a thread created for that purpose.
- Think *Tasks*
  - An I/O task, for example, accepts asynchronous read requests to a single file and executes them serially.
  - Message-oriented Middleware (MQS, Tibco …)
  - Ada and Intel RMX (circa 1979)

A generalized active object

- The solution can be generalized in the Java programming language like this:

`dequeue()` blocks (using wait/notify) until there's something to get.



The `javax.swing.*` thread is an active object

- Swing/AWT uses it's own thread to handle the incoming OS-level messages and to dispatch appropriate notifications to listeners.
- Swing is not thread safe.
- The Swing subsystem is effectively a "UI task" to which you enqueue requests:

```
SwingUtilities.invokeLater // enqueue a request
(   new Runnable()
    {   public void run()
        {   some_window.setSize(200,100); }
    }
);
```


Implementing Active Object

```
public class Active_object extends Thread
{   private Notifying_queue requests
                            = new Notifying_queue();
    public Active_object(){ setDaemon( true ); }
    public void run()
    {   try
        {   Runnable to_do;
            while((to_do=(Runnable)(
                        requests.dequeue()))!= null)
            {   to_do.run();
                to_do = null;  yield();
            }
        }catch( InterruptedException e ){}
    }
    public final void dispatch(Runnable operation )
    {   requests.enqueue( operation );
    }
}
```


Using an active object (detangling UNIX™ console output)

```
class Console
{    private static Active_object dispatcher
                     = new Active_object();
   static{ dispatcher.start(); }
   private Console(){}

   public static void println(final String s)
   {  dispatcher.dispatch
      (  new Runnable()
         {  public void run()
            {   System.out.println(s);
            }
         }
      );
   }
}
```


Summing up

- Java™-language threads are not platform independent—they can't be.
- You have to worry about threads, like it or not
  - GUI code is multithreaded
  - No telling where your code will be used in the future
- Programming threads is neither easy nor intuitive.
- **`synchronized`** is your friend. Grit your teeth and use it.
- Supplement language-level primitives to do real work.
- The threading system isn't object oriented.
- Use good architecture, not semaphores.

In-depth coverage and code

For in-depth coverage, see *Taming Java™ Threads* (www.apress.com)

For source code, these slides, etc., go to my web page www.holub.com