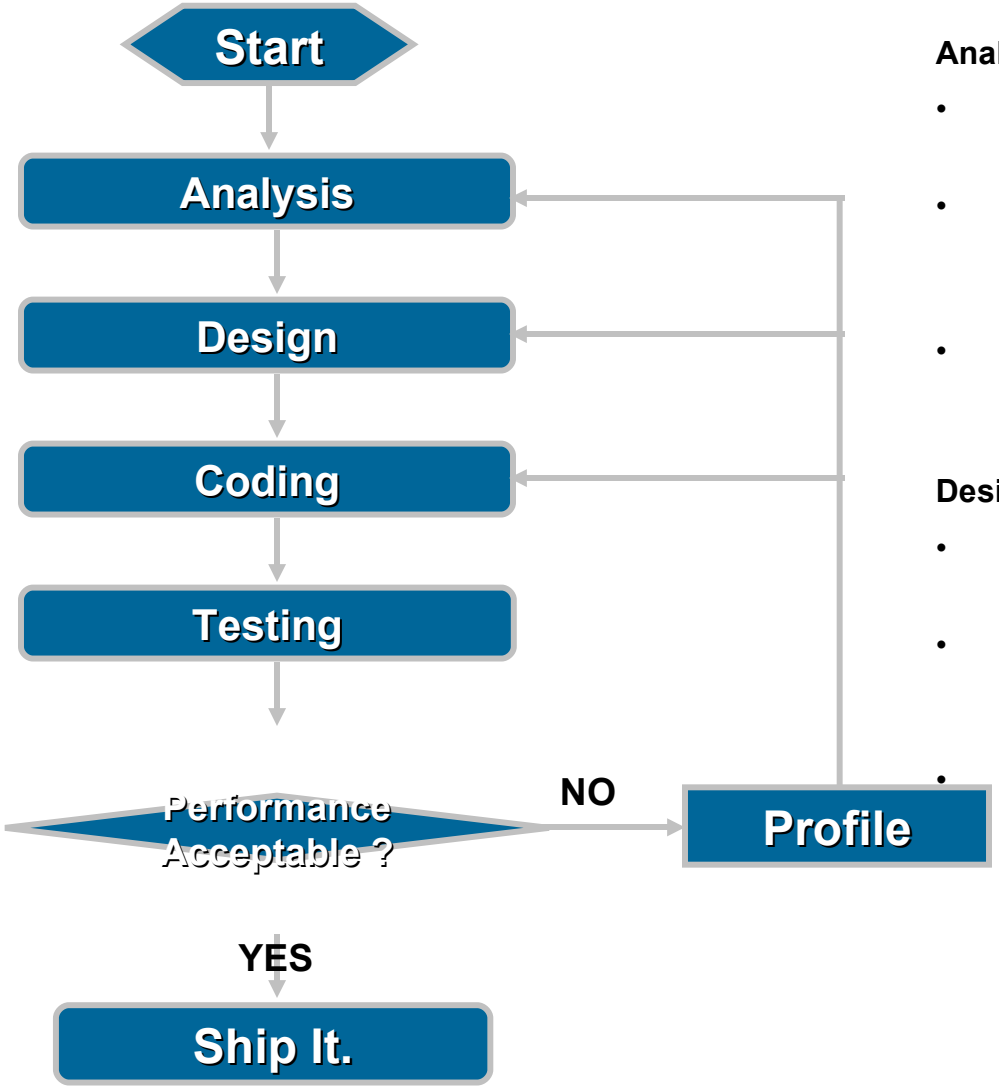
Java Performance

An Overview

What is Performance?

- Speed VS Performance
- Performance Categorization
  - Computation Performance
    - How many instructions are required to execute a statement?
    - How much overhead does a particular virtual method call incur?
    - Should I use a quick-sort or a bubble-sort here?
    - How to handle threading and synchronization issues?
    - Concurrent Data Structures
  - RAM Foot Prints
    - RAM and Disk Based Memory
  - Startup Time
    - Class Loading, Caching
  - Perceived Performance
    - End User Feel

The Performance Process



Analysis Phase Considerations

- During Analysis, do not consider low-level issues like language, syntax, classes, methods, or data structures.
- It's not always possible to quantify all of an application's performance requirements, especially for user interface code e.g. target H/W configuration, response time etc.
- If a goal can't be effectively quantified, qualitative requirements are much better than no requirement at all.

Design Phase Considerations

- Internal Structure of the Objects, Reusability and encapsulations
- Evaluation of different algorithms and data structures to see which is most efficient, saves N/W round trips and pay load
- Easily evolve your design to accommodate new or changed requirements

Performance Measurement

- Measurement is everything.
- Two analysis techniques are crucial for evaluating performance:
  - Benchmarking
    - Qualitatively comparing two or more operations.
    - The processes being compared might be two different algorithms that produce the same results, or two different virtual machines executing exactly the same code.
    - The key aspect of benchmarking is comparison.
    - A single benchmark result isn't interesting-it's only useful when there is something to compare it with.
    - Benchmarks typically measure the amount of time it takes to perform a particular task, but they can also be used to measure other variables, such as the amount of memory required.
      - How long it takes to launch an application
      - How long it takes to open a large document
      - How long it takes to scroll through a very large table of data
      - How long it takes to execute a complex database query
  - Profiling
    - Determining what areas of the system are consuming the most resources.

Performance Measurement

Using System.currentTimeMillis to calculate execution time

```
class TimeTest1 {
public static void main(String[] args) {
      long startTime = System.currentTimeMillis();
      long total = 0;

      for (int i = 0; i < 10000000; i++) {
            total += i; }

      long stopTime = System.currentTimeMillis();
      long elapsedTime = stopTime - startTime;
      System.out.println(elapsedTime);
   }

}
```

Java Threads

Day – 1

Java Threading Model

- A program or process can contain multiple threads that execute instructions according to program code. Like multiple processes that can run on one computer, multiple threads appear to be doing their work in parallel. Implemented on a multi-processor machine, they actually *can* work in parallel. Unlike processes, threads share the same address space; that is, they can read and write the same variables and data structures.
- When you're writing multithreaded programs, you must take great care that no one thread disturbs the work of any other thread. You can liken this approach to an office where the workers function independently and in parallel except when they need to use shared office resources or communicate with one another. One worker can speak to another worker only if the other worker is "listening" and they both speak the same language. Additionally, a worker can't use a copy machine until it is free and in a useable state (no half-completed copy jobs, paper jams, and so on). As we work through this article, you'll see how you can get threads to coordinate and cooperate in a Java program much like workers in a well-behaved organization.
- In a multithreaded program, threads are obtained from the pool of available ready-to-run threads and run on the available system CPUs. The OS can move threads from the processor to either a ready or blocking queue, in which case the thread is said to have "yielded" the processor. Alternatively, the Java virtual machine (JVM) can manage thread movement -- under either a cooperative or preemptive model -- from a ready queue onto the processor, where the thread can begin executing its program code.
- Threads allow the program to perform multiple tasks simultaneously. Process speed can be increased by using threads because the thread can stop or suspend a specific running process and start or resume the suspended processes.
- Multitasking or multiprogramming is delivered through the running of multiple threads concurrently. If your computer does not have multi-processors then the multi-threads really do not run concurrently.

Cooperative and Preemptive Threading

- *Cooperative threading* allows the threads to decide when they should give up the processor to other waiting threads. The application developer determines exactly when threads will yield to other threads, allowing them to work very efficiently with one another. A disadvantage is that a malicious or poorly written thread can starve other threads while it consumes all available CPU time.
- Under the *preemptive threading* model, the OS interrupts threads at any time, usually after allowing them to run for a period of time (known as a time-slice). As a result, no thread can ever unfairly hog the processor. However, interrupting threads at any time poses problems for the program developer.
- consider what would happen if a worker preempts another worker making copies halfway through her copy job: the new worker would start his copy job on a machine that already has originals on the glass or copies in the output tray.
- The preemptive threading model requires that threads use shared resources appropriately, while the cooperative model requires threads to share execution time. Because the JVM specification does not mandate a particular threading model, Java developers must write programs for both models. We'll see how to design programs for either model after looking a bit at threads and communication among threads

Threads & Java Language

- To create a thread using the Java language, you instantiate an object of type Thread (or a subclass) and send it the start() message. (A program can send the start() message to any object that implements the Runnable interface.) The definition of each thread's behavior is contained in its run() method. A run method is equivalent to main() in a traditional program: a thread will continue running until run() returns, at which point the thread dies.
- The example shows how to create a thread and how to implement the thread. In this example we will see that the program prints numbers from 1 to 10 line by line after 5 seconds which has been declared in the sleep function of the thread class.
- Sleep function contains the sleeping time in millisecond and in this program sleep function has contained 5000 millisecond mean 5 second time.
- There is sleep function must caught by the InterruptedException.
- So, this program used the InterruptedException which tells something the user if thread is failed or interrupted.

Coverage

- Thread class
  - run, start methods
  - yield, join
  - sleep
- Synchronization
  - synchronized methods & objects
  - wait/notify/notifyAll

java.lang.Thread

- java.lang.Thread implements the Runnable interface
  - The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method with no arguments called *run*.
- Create a thread by…
  - Define a subclass of java.lang.Thread
    - Define a *run* method
  - In another thread (e.g., the main), create an instance of the Thread subclass
    - Then, call *start* method of that instance

Example 1

- Create 2 threads from the Main, then start them
- Threads will be instances of different thread sub-classes

```
class MyThreadA extends Thread {
   public void run() {
        for (;;) {
                System.out.println("hello world1");
        }
   }
}

class MyThreadB extends Thread {
   public void run() {
        for (;;) {
                System.out.println("hello world2");
        }
   }
}

public class Main1 {
   public static void main(String [] args) {
        MyThreadA t1 = new MyThreadA();
        MyThreadB t2 = new MyThreadB();
        t1.start();
        t2.start();
        // main terminates, but in Java the other threads keep running
        // and hence Java program continues running
   }
}
```

hello world2
hello world2
hello world1
hello world2
hello world1
hello world2
hello world2
hello world1
hello world1
hello world1
hello world1
hello world2
hello world1
hello world1
hello world2
hello world2
hello world1
hello world1
hello world2
hello world2
hello world1
hello world1
hello world2

Example 2

- Create 2 threads from the Main, then start them
- Threads will be instances of the same thread sub-class
- Use argument of constructor of new thread class to pass text name of thread, e.g., “thread1” and “thread2”
  - Data member provides different data per thread; can be used to share data

```
class MyThread extends Thread {
        private String name;
        public MyThread(String name) {
                this.name = name;
        }
        public void run() {
                for (;;) {
                        System.out.println(name + ": hello world");
                }
        }
}

public class Main2 {
        public static void main(String [] args) {
                MyThread t1 = new MyThread("thread1");
                MyThread t2 = new MyThread("thread2");
                t1.start(); t2.start();
        }
}
```

thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread2: hello world

java.lang.Thread

- public static void yield();
  - Method of java.lang.Thread
  - Thread gives up CPU for other threads ready to run

```
class MyThread extends Thread {
        private String name;
        public MyThread(String name) {
                this.name = name;
        }
        public void run() {
                for (;;) {
                        System.out.println(name + ": hello world");
                        yield();
                }
        }
}

public class Main3 {
        public static void main(String [] args) {
                MyThread t1 = new MyThread("thread1");
                MyThread t2 = new MyThread("thread2");
                t1.start(); t2.start();
        }
}
```

Some Output

```
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
thread2: hello world
thread1: hello world
```

Notice the alternation of output

More Thread Members

- public final void join();

```
MyThread t1 = new MyThread("thread1");
t1.start();
t1.join();
```

  - Wait until the thread is “not alive”
  - Threads that have completed are “not alive” as are threads that have not been started

- public static void sleep (long millis) throws InterruptedException;
  - Makes the currently running thread sleep (block) for a period of time
  - The thread does not lose ownership of any monitors.
  - InterruptedException - if another thread has interrupted the current thread.

Join Example

```
class MyThread extends Thread {
	public void run() {
		for (int i=0; i < 1000; i++) {
			System.out.println("hello world1");
		}
	}
}

public class Main4 {
	public static void main(String [] args) {
		MyThread t1 = new MyThread();
		t1.start();
		try {
			t1.join(); // wait for the thread to terminate
		} catch (InterruptedException e) {
			System.out.println("ERROR: Thread was interrupted");
		}

		System.out.println("Thread is done!");
	}
}
```

Some output

```
…
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
hello world1
Thread is done!
```

Thread State

- public Thread.State getState()
  - Returns the state of this thread. This method is designed for use in monitoring of the system state, not for synchronization control

```
public static enum Thread.State
extends Enum<Thread.State>
```

A thread state. A thread can be in one of the following states:

- NEW
  A thread that has not yet started is in this state.
- RUNNABLE
  A thread executing in the Java virtual machine is in this state.
- BLOCKED
  A thread that is blocked waiting for a monitor lock is in this state.
- WAITING
  A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
- TIMED_WAITING
  A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
- TERMINATED
  A thread that has exited is in this state.

A thread can be in only one state at a given point in time. These states are virtual machine states which do not reflect any operating system thread states.

http://java.sun.com/javase/6/docs/api/java/lang/Thread.State.html

Thread Scheduling in Java

- public final void setPriority(int newPriority);
- public final int getPriority();
- public static final int MAX_PRIORITY
  // on my system: 10; Mac OS X 2/21/05
- public static final int MIN_PRIORITY
  // on my system: 1; Mac OS X 2/21/05
- Scheduling
  - Priority inherited from parent, but can be changed
  - Higher priority threads generally run before lower priority threads
  - For equal priority threads, best to call yield() intermittently to handle JVM's with user-level threading (i.e., no time-slicing)

Sharing Data Across Java Threads

- Consider the situation where a parent thread wants to pass data to a child thread
  - e.g., so that child can change data and parent can have access to the changed data
- How can this be done?
- Can pass an object instance to the child thread constructor, and retain that object instance in a data member

```
class SharedData {
	public int a = 0;
	public String s = null;

	public SharedData() {
		a = 10;
		s = "Test";
	}
}

class MyThread extends Thread {
	private SharedData m_data = null;

	public MyThread(SharedData data) {
		m_data = data;
	}

	public void run() {
		for (;;) {
			m_data.a++;
		}
	}
}
```

```
public class Main5 {
     public static void main(String [] args) {
          SharedData data = new SharedData();
          MyThread t1 = new MyThread(data);
          t1.start();

          for (;;) {
               data.a--;
          }
     }
}
```

If we have multiple threads accessing this shared data, how do we synchronize access to ensure it remains in a consistent state?

Basic Tools for Synchronization in Java

- Synchronized methods
- Synchronized objects
- Methods
  - wait
  - notify
  - notifyAll

Synchronized Methods: Monitors

- synchronized keyword used with a method
  - E.g.,

```
public synchronized void SetValue() {
    // Update instance data structure.
    // When the thread executes here,  it exclusively has the monitor lock
}
```

  - Provides *instance-based* mutual exclusion
    - A lock is implicit provided-- allows at most one thread to be executing the method at one time
  - Used on a per method basis; not all methods in a class have to have this
    - But, you'll need to design it right!!

Example

- Construct a queue (FIFO) data structure that can be used by two threads to access the queue data in a synchronized manner

```
// only 1 thread can use Add or Remove at a time for a particular
// SynchQueue instance
class SynchQueue {
      public LinkedList l;

      SynchQueue () {
            l = new LinkedList();
      }

      public synchronized void Add(Object elem) {
            l.addLast(elem);
      }

      public synchronized Object Remove() {
            if (l.size() > 0) {
                  return l.removeFirst();
            } else {
                  return null;
            }
      }
}
```

```
class Producer extends Thread {
	SynchQueue q;
	int curr;

	Producer (SynchQueue q) {
		this.q = q;
		curr = 1;
	}

	public void run() {
		for (;;) {
			Integer i = new Integer(curr);
			q.Add(i);
			curr++;
		}
	}
}

class Consumer extends Thread {
	SynchQueue q;

	Consumer (SynchQueue q) {
		this.q = q;
	}

	public void run() {
		for (;;) {
			Integer i = (Integer) q.Remove();
			if (i != null) {
				System.out.print(i + " ");
			}
		}
	}
}
```

This implementation has a problem! The Consumer prints which slows it down a LOT, and thus the producer is faster, and thus the producer fills up the queue, and causes heap space to run out!!

A good exercise here is to alter this example to limit the maximum number of items that are stored in the queue. See BoundedSynchMain.java

```
import java.util.LinkedList; // not synchronized

public class SynchMain {
      public static void main(String args[]) {
            SynchQueue q = new SynchQueue();
            Producer p = new Producer(q);
            Consumer c = new Consumer(q);
            p.start();
            c.start();
      }
}
```

Any problems?

- How can we make the Remove block when the queue is empty?
- I.e., presently we are doing a “busy wait”

See also:

http://java.sun.com/docs/books/tutorial/collections/implementations/queue.html

java.util.concurrent.LinkedBlockingQueue<E>

http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/BlockingQueue.html

wait (see also java.lang.Object)

- Relative to an Object
  - E.g., Used within a synchronized method
- Releases lock on Object & waits until condition is true
  - Blocks calling process until notify() or notifyAll() is called on same object instance (or exception occurs)
- Typically used within a loop to re-check a condition
- wait(long millis); // bounded wait

notify and notifyAll (see also java.lang.Object)

- Relative to an Object
  - E.g., Used within a synchronized method
- Wakes up a blocked thread (notify) or all blocked threads (notifyAll)
  - One woken thread reacquires lock; The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object.
- For notify, if more than one thread available to be woken, then one is picked

Typical use of *wait* within a synchronized method

```
while (condition not true) {
   try {
        wait(); // this.wait();
   } catch {
        System.out.println("Interrupted!");
   }
}
// After loop, condition now true & thread
// has monitor lock for this object instance
```

Re-checking Monitor Conditions

- wait/notify
  - After receiving a notify, a process waiting on a condition may not be next to gain access to monitor (to the data)
    - E.g., occurs if notifyAll used
  - Process may need to re-check the conditions upon which it was waiting
- An "awakened thread will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened thread enjoys no reliable privilege or disadvantage in being the next thread to lock this object." (http://java.sun.com/j2se/1.5.0/docs/api/)

Example

- Extend the example from before:
  - a queue (FIFO) data structure that can be used by two threads to access the queue data in a synchronized manner
- This time, use wait & notify to block Consumer thread if the queue is empty

Blocking Remove

```
// only 1 thread can use Add or Remove at a time
class SynchQueue {
        public LinkedList<Integer> l;

        SynchQueue () {
                l = new LinkedList<Integer>();
        }

        public synchronized void Add(Integer elem) {
                l.addLast(elem);
                notify();
        }

        public synchronized Integer Remove() {
                while (l.size() == 0) {
                        try {
                                wait();
                        } catch (InterruptedException e) {
                                System.out.println("ERROR: Thread interrupted!");
                        }
                }

                return l.removeFirst();
        }
}
```

Example

- Two producer threads (A & B), and one consumer thread
- Consumer needs one type of item from thread A and one type of item from thread B before it can proceed
- Use a loop and a wait and recheck conditions in the consumer

```
// only 1 thread can use Add or Remove at a time
class AB {
       public boolean aReady = false;
       public boolean bReady = false;

       AB () {
       }

       public synchronized void PutA() {
            aReady = true;
            notify();
       }

       public synchronized void PutB() {
            bReady = true;
            notify();
       }

       public synchronized void GetAB() {
         while ((! aReady) || (! bReady)) {
             try {
                 wait();
             } catch (InterruptedException e) {
                 System.out.println(
                  "ERROR: Interrupted Exception!\n");
             }
         }

            // It must be the case that
            // aReady == true and bReady == true
            // AND we have exclusive access
            // to this object instance,
            // so, go ahead and change
            // the ready state back to false.
            aReady = false;
            bReady = false;
       }
}
```

```
class ProducerA extends Thread {
       AB ab;
       ProducerA (AB ab) {
          this.ab = ab;
       }

       public void run() {
            for (;;) {
              ab.PutA();
            }
       }
}

class ProducerB extends Thread {
       AB ab;
       ProducerB (AB ab) {
          this.ab = ab;
       }

       public void run() {
            for (;;) {
              ab.PutB();
            }
       }
}

class Consumer extends Thread {
       AB ab;

       Consumer (AB ab) {
            this.ab = ab;
       }

       public void run() {
            for (;;) {
             ab.GetAB();
            }
       }
}
```

```
public class Recheck {
        public static void main(String args[]) {
                AB ab = new AB();
           ProducerA a = new ProducerA(ab);
           ProducerB b = new ProducerB(ab);
           Consumer c = new Consumer(ab);
           a.start();
           b.start();
           c.start();
        }
}
```

Another Example

- In SynchQueue, change the Add method so that it blocks until the Queue has less than a maximum number of elements

InterruptedException

- Wait can be woken by the exception, I.e., for reasons other than notify
- Sometimes this can be handled as part of the process of re-checking conditions
- There is another way to handle it too

Exception in Wait

```
// In a synchronized method

// check your condition, e.g., with a semaphore
// operation, test "value" member variable

if /* or while */ (/* condition */) {
    boolean interrupted;
    do {
        interrupted = false;
        try {
            wait();
        } catch (InterruptedException e) {
            interrupted = true;
        }
    } while (interrupted);
}
```

Only allows release from wait caused by notify or notifyAll

Synchronized Blocks

- Synchronized methods
  - Implicitly lock is on *this* object
- Synchronized *blocks*
  - lock on an arbitrary, specified object
  - similar to condition variables in monitors
  - but need to have a synchronized block around an object before wait/notify used
  - use wait/notify on the object itself

Syntax

```
synchronized (object) {
   // object.wait()
   // object.notify()
   // object.notifyAll()
}
```

- For example, this allows you to synchronize just a few lines of code, or to synchronize on the basis of an arbitrary object

Example

- Suppose in a Global File Table, suppose that per open file you keep an

Object Lock;

- you can then use a synchronized block to make sure that some operations only get done in a mutually exclusive manner on the file

```
synchronized (Lock) {
        // if we get to here we're the only one
        // accessing the file
}
```

Lab 5: Agent Simulation

- Could use synchronized blocks to accomplish synchronization on the environment cells

```
synchronized (cell) {
   // check to see if agent can consume food
   // or socialize depending on what the goal
   // of the agent is
}
```

Example

- Implement Semaphore class with Java synchronization
  - Provide constructor, and P (wait) and V (signal) methods
  - Use synchronized methods
    - and Java wait/notify
- Note
  - Java implements Semaphores—
    - http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/Semaphore.html

java.lang.Runnable Interface

- The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called *run*.
- Known Implementing Classes:
  - AsyncBoxView.ChildState, FutureTask, RenderableImageProducer, Thread, TimerTask
- From http://java.sun.com/j2se/1.5.0/docs/api/
- Runnable can be used to create threads

Thread Locking

- Most applications require threads to communicate and synchronize their behavior to one another. The simplest way to accomplish this task in a Java program is with locks.
- To prevent multiple accesses, threads can acquire and release a lock before using resources.
- Locks around shared variables allow Java threads to quickly and easily communicate and synchronize.
- Threads that attempt to acquire a lock in use go to sleep until the thread holding the lock releases it. After the lock is freed, the sleeping thread moves to the ready-to-run queue.
- Each object has a lock; a thread can acquire the lock for an object by using the synchronized keyword. Methods, or synchronized blocks of code, can only be executed by one thread at a time for a given instantiation of a class, because that code requires obtaining the object's lock before execution.

Fine Grained Locks

- Often, using a lock at the object level is too coarse. Why lock up an entire object, disallowing access to any other synchronized method for only brief access to shared resources?
- If an object has multiple resources, it's unnecessary to lock all threads out of the whole object in order to have one thread use only a subset of the thread's resources. Because every object has a lock, we can use dummy objects as simple locks.
- These methods need not be synchronized at the method level by declaring the whole method with the synchronized keyword; they are using the member locks, not the object-wide lock that a synchronized method acquires.

Fine Grained Locks

```
class FineGrainLock {
  MyMemberClass x, y;
  Object xlock = new Object(), ylock = new
    Object();

  public void foo() {
    synchronized(xlock) {
      //access x here
    }

    //do something here - but don't use shared
     resources

    synchronized(ylock) {
      //access y here
    }
  }
```

```
  public void bar() {
    synchronized(xlock) {
      synchronized(ylock) {
        //access both x and y here
      }
    }
    //do something here - but don't use shared
     resources
  }
}
```

Reducing Lock Granularity

Another valuable technique for reducing contention is to spread your synchronizations over more locks. For example, suppose that you have a class that stores user information and service information in two separate hash tables, as shown in example.

```
public class AttributesStore {
  private HashMap usersMap = new HashMap();
  private HashMap servicesMap = new HashMap();

  public synchronized void setUserInfo(String user, UserInfo userInfo) {
    usersMap.put(user, userInfo);
  }

  public synchronized UserInfo getUserInfo(String user) {
    return usersMap.get(user);
  }

  public synchronized void setServiceInfo(String service,
                                          ServiceInfo serviceInfo) {
    servicesMap.put(service, serviceInfo);
  }

  public synchronized ServiceInfo getServiceInfo(String service) {
    return servicesMap.get(service);
  }
}
```

Reducing Lock Granularity

http://www.ibm.com/developerworks/java/library/j-threads2.html

Here, the accessor methods for user and service data are synchronized, which means that they are synchronizing on the AttributesStore object.

While this is perfectly thread-safe, it increases the likelihood of contention for no real benefit. If a thread is executing setUserInfo, it means that not only will other threads be locked out of setUserInfo and getUserInfo, as is desired, but they will also be locked out of getServiceInfo and setServiceInfo. This problem can be avoided by having the accessor simply synchronize on the actual objects being shared (the userMap and servicesMap objects), as shown.

Now threads accessing the services map will not contend with threads trying to access the users map. (In this case, the same effect could also be obtained by creating the maps using the synchronized wrapper mechanism provided by the Collections framework, Collections.synchronizedMap.) Assuming that requests against the two maps are evenly distributed, in this case this technique would cut the number of potential contentions in half.

```
public class AttributesStore {
  private HashMap usersMap = new HashMap();
  private HashMap servicesMap = new HashMap();

  public void setUserInfo(String user, UserInfo userInfo) {
    synchronized(usersMap) {
      usersMap.put(user, userInfo);
    }
  }

  public UserInfo getUserInfo(String user) {
    synchronized(usersMap) {
      return usersMap.get(user);
    }
  }

  public void setServiceInfo(String service,
                             ServiceInfo serviceInfo) {
    synchronized(servicesMap) {
      servicesMap.put(service, serviceInfo);
    }
  }

  public ServiceInfo getServiceInfo(String service) {
    synchronized(servicesMap) {
      return servicesMap.get(service);
    }
```

Reducing Lock Granularity – HashMap Example

- One of the most common contention bottlenecks in server-side Java applications is the HashMap. Applications use HashMap to cache all sorts of critical shared data (user profiles, session information, file contents), and the HashMap.get method may correspond to many bytecode instructions. For example, if you are writing a Web server, and all your cached pages are stored in a HashMap, every request will want to acquire and hold the lock on that map, and it will become a bottleneck.
- We can extend the lock granularity technique to handle this situation, although we must be careful as there are some potential Java Memory Model (JMM) hazards associated with this approach. The LockPoolMap in Listing 5 exposes thread-safe get() and put() methods, but spreads the synchronization over a pool of locks, reducing contention substantially.
- LockPoolMap is thread-safe and functions like a simplified HashMap, but has more attractive contention properties. Instead of synchronizing on the entire map on each get() or put() operation, the synchronization is done at the bucket level. For each bucket, there's a lock, and that lock is acquired when traversing a bucket either for read or write. The locks are created when the map is created (there would be JMM problems if they were not.)
- If you create a LockPoolMap with many buckets, many threads will be able to use the map concurrently with a much lower likelihood of contention. However, the reduced contention does not come for free. By not synchronizing on a global lock, it becomes much more difficult to perform operations that act on the map as a whole, such as the size() method. An implementation of size() would have to sequentially acquire the lock for each bucket, count the number of nodes in that bucket, and release the lock and move on to the next bucket. But once the previous lock is released, other threads are now free to modify the previous bucket. By the time size() finishes calculating the number of elements, it could well be wrong. However, the LockPoolMap technique works quite well in some situations, such as shared caches.

```
import java.util.*;

/**
 * LockPoolMap implements a subset of the Map interface (get, put, clear)
 * and performs synchronization at the bucket level, not at the map
 * level.  This reduces contention, at the cost of losing some Map
 * functionality, and is well suited to simple caches.  The number of
 * buckets is fixed and does not increase.
 */

public class LockPoolMap {
  private Node[] buckets;
  private Object[] locks;

  private static final class Node {
    public final Object key;
    public Object value;
    public Node next;

    public Node(Object key) { this.key = key; }
  }

  public LockPoolMap(int size) {
    buckets = new Node[size];
    locks = new Object[size];
    for (int i = 0; i < size; i++)
      locks[i] = new Object();
  }

  private final int hash(Object key) {
    int hash = key.hashCode() % buckets.length;
    if (hash < 0)
      hash *= -1;
```

Reducing Lock Granularity – HashMap Example

- Table compares the performance of three shared map implementations; a synchronized HashMap, an unsynchronized HashMap (not thread-safe), and a LockPoolMap. The unsynchronized version is present only to show the overhead of contention. A test that does random put() and get() operations on the map was run, with a variable number of threads, on a dual-processor system Linux system using the Sun 1.3 JDK. The table shows the run time for each combination. This test is somewhat of an extreme case; the test programs do nothing but access the map, and so there will be many more contentions than there would be in a realistic program, but it is designed to illustrate the performance penalty of contention.
- While all the implementations exhibit similar scaling characteristics for large numbers of threads, the HashMap implementation exhibits a huge performance penalty when going from one thread to two, because there will be a contention on every single put() and get() operation. With more than one thread, the LockPoolMap technique is approximately 15 times faster than the HashMap technique. This difference reflects the time lost to scheduling overhead and to idle time spent waiting to acquire locks. The advantage of LockPoolMap would be even larger on a system with more processors.

Table - Scalability comparison between HashMap and LockPoolMap

Threads	Unsynchronized HashMap (unsafe)	Synchronized HashMap	LockPoolMap
1	1.1	1.4	1.6
2	1.1	57.6	3.7
4	2.1	123.5	7.7
8	3.7	272.3	16.7
16	6.8	577.0	37.9
32	13.5	1233.3	80.5

Common Locking Problems - Deadlocking

Deadlocking

- Deadlocking is a classic multithreading problem in which all work is incomplete because different threads are waiting for locks that will never be released.
- Imagine two threads, which represent two hungry people who must share one fork and knife and take turns eating. They each need to acquire two locks: one for the shared fork resource and one for the shared knife resource. Imagine if thread "A" acquires the knife and thread "B" acquires the fork. Thread A will now block waiting for the fork, while thread B blocks waiting for the knife, which thread A has.
- Though a contrived example, this sort of situation occurs often, albeit in scenarios much harder to detect. Although difficult to detect and hash out in every case, by following these few rules, a system's design can be free of deadlocking scenarios:
  - Have multiple threads acquire a group of locks in the same order. This approach eliminates problems where the owner of X is waiting for the owner of Y, who is waiting for X.
  - Group multiple locks together under one lock. In our case, create a silverware lock that must be acquired before either the fork or knife is obtained.
  - Label resources with variables that are readable without blocking. After the silverware lock is acquired, a thread could examine variables to see if a complete set of silverware is available. If so, it could obtain the relevant locks; if not, it could release the master silverware lock and try again later.
  - Most importantly, design the entire system thoroughly before writing code. Multithreading is difficult, and a thorough design before you start to code will help avoid difficult-to-detect locking problems.

Common Locking Problems - Volatile Variables

Volatile Variables

- The volatile keyword was introduced to the language as a way around optimizing compilers. Take the following code for example:

```
class VolatileTest {
boolean flag;
public void foo() {
        flag = false;
        if(flag) {
                //this could happen }
        }
}
```

- An optimizing compiler might decide that the body of the if statement would never execute, and not even compile the code. If this class were accessed by multiple threads, flag could be set by another thread after it has been set in the previous code, but before it is tested in the if statement. Declaring variables with the volatile keyword tells the compiler not to optimize out sections of code by predicting the value of the variable at compile time.

Common Locking Problems - Inaccessible Threads

Inaccessible threads

- Occasionally threads have to block on conditions other than object locks. IO is the best example of this problem in Java programming. When threads are blocked on an IO call inside an object, that object must still be accessible to other threads.
- That object is often responsible for canceling the blocking IO operation. Threads that make blocking calls in a synchronized method often make such tasks impossible. If the other methods of the object are also synchronized, that object is essentially frozen while the thread is blocked.
- Other threads will be unable to message the object (for example, to cancel the IO operation) because they cannot acquire the object lock. Be sure to not synchronize code that makes blocking calls, or make sure that a non-synchronized method exists on an object with synchronized blocking code.
- Although this technique requires some care to ensure that the resulting code is still thread safe, it allows objects to be responsive to other threads when a thread holding its locks is blocked.

Designing Using Locks for Different Threading Models

- The determination of whether the threading model is preemptive or cooperative is up to the implementers of the virtual machine and can vary across different implementations. As a result, Java developers must write programs that work under both models.
- Under the preemptive model, threads can be interrupted in the middle of any section of code, except for an atomic block of code. Atomic sections are code segments that once started will be finished by the current thread before it is swapped out.
- In Java programming, assignment to variables smaller than 32 bits is an atomic operation, which excludes variables of types double and long (both are 64 bits). As a result, atomic operations do not need synchronization.
- The use of locks to properly synchronize access to shared resources is sufficient to ensure that a multithreaded program works properly with a preemptive virtual machine.
- For cooperative threads, it is up to the programmer to ensure that threads give up the processor routinely so that they do not deprive other threads of execution time. One way to do this is to call yield(), which moves the current thread off the processor and onto the ready queue. A second approach is to call sleep(), which makes the thread give up the processor and does not allow it to run until the amount of time specified in the argument to sleep has passed.

Designing Using Locks for Different Threading Models

- As you might expect, simply placing wait/notify/notify all calls at arbitrary points in code doesn't always work.
- If a thread is holding a lock (because it's in a synchronized method or block of code), it does not release the lock when it calls yield(). This means that other threads waiting for the same lock will not get to run, even though the running thread has yielded to them. To alleviate this problem, call yield() when not in a synchronized method. Surround the code to be synchronized in a synchronized block within a non-synchronized method and call yield() outside of that block.
- Another solution is to call wait(), which makes the processor give up the lock belonging to the object it is currently in. This approach works fine if the object is synchronized at the method level, because it is only using that one lock.
- If it is using a fine-grained lock, wait() will not give up those locks. In addition, a thread that is blocked on a call to wait() will not awaken until another thread calls notify(), which moves the waiting thread to the ready queue. To wake up all threads that are blocking on a wait() call, a thread calls notifyAll().

Avoiding Contention through Thread Local

http://www.ibm.com/developerworks/java/library/j-threads3.html

- Sometimes, it is very difficult to make a class thread-safe without compromising its functionality, ease of use, or performance. Some classes retain state information from one method invocation to the next, and it is difficult to make such classes thread-safe in any practical way.
- It may be easier to manage the use of a non-thread-safe class than to try and make the class thread-safe. A class that is not thread-safe can often be used safely in a multithreaded program as long as you ensure that instances of that class used by one thread are not used by other threads. For example, the JDBC Connection class is not thread-safe -- two threads cannot safely share a Connection at a fine level of granularity -- but if each thread had its own Connection, then multiple threads can safely perform database operations simultaneously.
- It is certainly possible to maintain a separate JDBC connection (or any other object) for each thread without the use of ThreadLocal; the Thread API gives us all the tools we need to associate objects with threads. However, the ThreadLocal class makes it much easier for us to manage the process of associating a thread with its per-thread data.

Avoiding Contention through Thread Local

What is a Thread Local Variable?

- A *thread-local variable* effectively provides a separate copy of its value for each thread that uses it. Each thread can see only the value associated with that thread, and is unaware that other threads may be using or modifying their own copies.
- Some compilers (such as the Microsoft Visual C++ compiler or the IBM XL FORTRAN compiler) have incorporated support for thread-local variables into the language using a storage-class modifier (like static or volatile). Java compilers offer no special language support for thread-local variables; instead, they are implemented with the ThreadLocal class, which has special support in the core Thread class.
- Because thread-local variables are implemented through a class, rather than as part of the Java language itself, the syntax for using thread-local variables is a bit more clumsy than for language dialects where thread-local variables are built in. To create a thread-local variable, you instantiate an object of class ThreadLocal. The ThreadLocal class behaves much like the various Reference classes in java.lang.ref; it acts as an indirect handle for storing or retrieving a value. Listing below shows

  ThreadLocal interface

```
public class ThreadLocal {
  public Object get();
  public void set(Object newValue);
  public Object initialValue();
}
```

- The get() accessor retrieves the current thread's value of the variable; the set() accessor modifies the current thread's value. The initialValue() method is an optional method that lets you set the initial value of the variable if it has not yet been used in this thread; it allows for a form of lazy initialization.

Art of Locking w/o Locking

- xx

Hardware Based Locking in Java

- xx

Lock Interface

http://java.sun.com/docs/books/tutorial/essential/concurrency/newlocks.html

- Synchronized code relies on a simple kind of reentrant lock. This kind of lock is easy to use, but has many limitations. More sophisticated locking idioms are supported by the java.util.concurrent.locks package.
- We won't examine this package in detail, but instead will focus on its most basic interface, Lock. Lock objects work very much like the implicit locks used by synchronized code. As with implicit locks, only one thread can own a Lock object at a time. Lock objects also support a wait/notify mechanism, through their associated Condition objects.
- The biggest advantage of Lock objects over implicit locks is their ability to back out of an attempt to acquire a lock. The tryLock method backs out if the lock is not available immediately or before a timeout expires (if specified). The lockInterruptibly method backs out if another thread sends an interrupt before the lock is acquired.

Executors

http://java.sun.com/docs/books/tutorial/essential/concurrency/executors.html

- In all of the thread examples, there's a close connection between the task being done by a new thread, as defined by its Runnable object, and the thread itself, as defined by a Thread object. This works well for small applications, but in large-scale applications, it makes sense to separate thread management and creation from the rest of the application. Objects that encapsulate these functions are known as *executors*. The following subsections describe executors in detail.
  - Executor-Interface define the three executor object types.
  - Thread Pool are the most common kind of executor implementation.

Concurrent Collections

http://java.sun.com/docs/books/tutorial/essential/concurrency/collections.html

- The java.util.concurrent package includes a number of additions to the Java Collections Framework. These are most easily categorized by the collection interfaces provided:
  - **BlockingQueue** defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.
  - **ConcurrentMap** is a subinterface of java.util.Map that defines useful atomic operations. These operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization. The standard general-purpose implementation of ConcurrentMap is ConcurrentHashMap, which is a concurrent analog of HashMap.
  - **ConcurrentNavigableMap** is a subinterface of ConcurrentMap that supports approximate matches. The standard general-purpose implementation of ConcurrentNavigableMap is ConcurrentSkipListMap, which is a concurrent analog of TreeMap.
  - All of these collections help avoid Memory Consistency Errors by defining a happens-before relationship between an operation that adds an object to the collection with subsequent operations that access or remove that object.

Object – Hidden Queue

- xx

Concept of Synchronization

http://java.sun.com/docs/books/tutorial/essential/concurrency/sync.html

- Threads communicate primarily by sharing access to fields and the objects reference fields refer to. This form of communication is extremely efficient, but makes two kinds of errors possible: *thread interference* and *memory consistency errors*. The tool needed to prevent these errors is *synchronization*.
  - **Thread Interference** describes how errors are introduced when multiple threads access shared data.
  - **Memory Consistency Errors** describes errors that result from inconsistent views of shared memory.
  - **Synchronized Methods** describes a simple idiom that can effectively prevent thread interference and memory consistency errors.
  - **Implicit Locks and Synchronization** describes a more general synchronization idiom, and describes how synchronization is based on implicit locks.
  - **Atomic Access** talks about the general idea of operations that can't be interfered with by other threads.

Understanding a Thread Safe Class

- xx

Producer Consumer Problem

- xx

Problems with Stopping a Thread

- xx

Interrupting a Thread

http://java.sun.com/docs/books/tutorial/essential/concurrency/interrupt.html

- An *interrupt* is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate. This is the usage emphasized in this lesson.
- A thread sends an interrupt by invoking interrupt on the Thread object for the thread to be interrupted. For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption.
- What if a thread goes a long time without invoking a method that throws InterruptedException? Then it must periodically invoke Thread.interrupted, which returns true if an interrupt has been received. For example:

```
for (int i = 0; i < inputs.length; i++) {
heavyCrunch(inputs[i]);
if (Thread.interrupted()) {
 //We've been interrupted: no more crunching.
 return; } }
```

In this simple example, the code simply tests for the interrupt and exits the thread if one has been received. In more complex applications, it might make more sense to throw an InterruptedException: if (Thread.interrupted()) { throw new InterruptedException(); } This allows interrupt handling code to be centralized in a catch clause.

Handlers

- xx

Instructions Reordering in Processors & Cache

- xx

Cache Coherency of a Processor

- xx

Synchronization - Final & Volatile

- xx

Safe Programming

- xx

Un-Safe Programming

- xx

Handling Contention

- xx

Understanding Hash Map and Concurrent Hash Map

- xx

Understanding Concurrent Linked Queue

- xx

Understanding Concurrent Skip List Map

- xx

Understanding Deque

- xx

Navigable Interface

- xx

Canned Synchronizers

- xx

Concept of Thread Pool

- Multiple threads increase the performance especially in I/O bound jobs
- However, Threads have overheads too
- They burden the VM & garbage collection
- Switching between threads cause overhead too.
- Spawning more & more threads – waste of memory & thread management cycles!
- ? ? Re-use threads…thread pool

Thread pool cont..

- No need to re-start a thread after it is used, but engineered to do another task.
- Put all the tasks you need in a queue or some data structure & have each thread retrieve a job from the queue when its previous job is completed…thread pooling
- The data structure in which tasks are kept is called ‘pool’
- Implementation requires allotting a fixed number of threads when first the pool is created.
- When pool empty each thread waits on the pool
- When a task is added to the pool all waiting threads are informed.

Alternate thread pool

- Put the threads themselves in a pool & have the main program pull threads out of the pool and assign tasks.
- When there is no thread in the pool, the main program can spawn a new thread
- Synchronization is an important problem.
- Use java.util.vector which is fully synchronized.

Multithreading & Thread pool Lab

- Thread pool example from Internet @
- http://www.informit.com/articles/article.asp?
- Another example:
- http://www.java2s.com/Code/Java/

References & Resources

- Resources
  - API docs
    - http://java.sun.com/javase/6/docs/api/
    - java.lang.Thread, java.lang.Runnable
    - java.lang.Object, java.util.concurrent
  - Tutorials
    - http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html
    - http://www.cs.clemson.edu/~cs428/resources/java/tutorial/JTThreads.html
  - Introduction to Java Threads
    - http://www.javaworld.com/javaworld/jw-04-1996/jw-04-threads.html
  - Thread safety
    - http://en.wikipedia.org/wiki/Thread-safety
    - http://www.javaworld.com/jw-08-1998/jw-08-techniques.html