ProKarma

Java Concurrency

Nadeem Mohammad

Date: 02/07/2013

Make your problems open new doors for innovation and proudly say Processsing by Innovation

*ProKarma Recognized by **Inc. Magazine** as the "Fastest-Growing IT Services Company in America"*

ABOUT ME

- Software Gardener
- CRP, AMA
- 5+ years with Prokarma
- Two Sons
  - Ibraheem
  - Imran

AGENDA

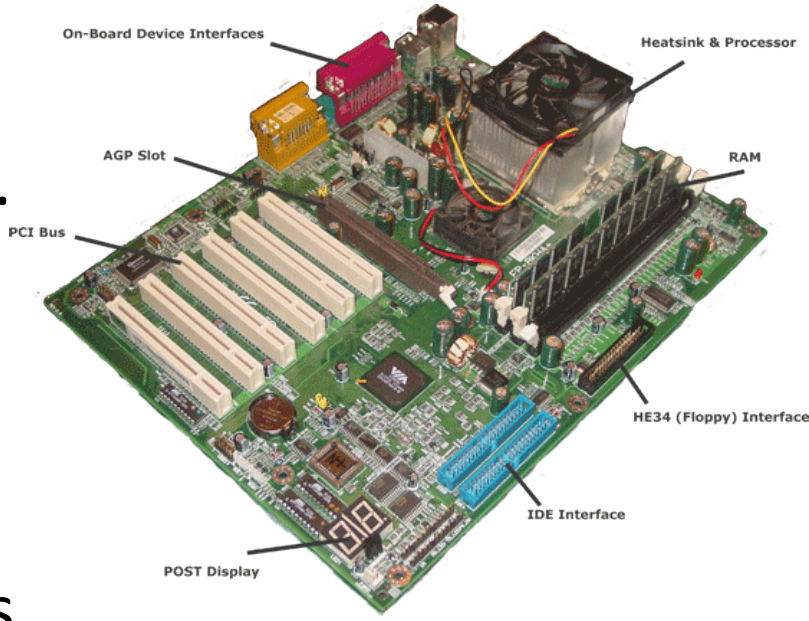
- What is Concurrency?
- Why Concurrency?
- Perils of Concurrency?
- Multi Processing and Multi Cores
- Process & Thread
- Multitasking & Multithreading
- Threads in Java

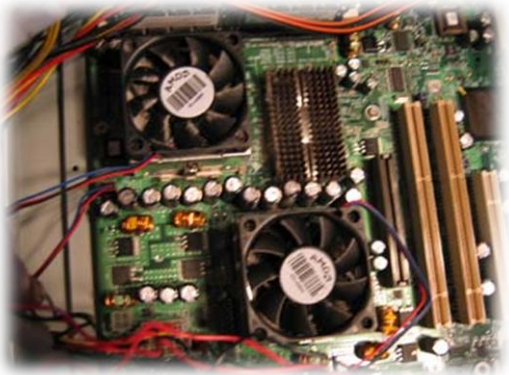
What is Concurrency?

- The ability to run several parts of a program or several programs in parallel.

How Concurrency Achieved?

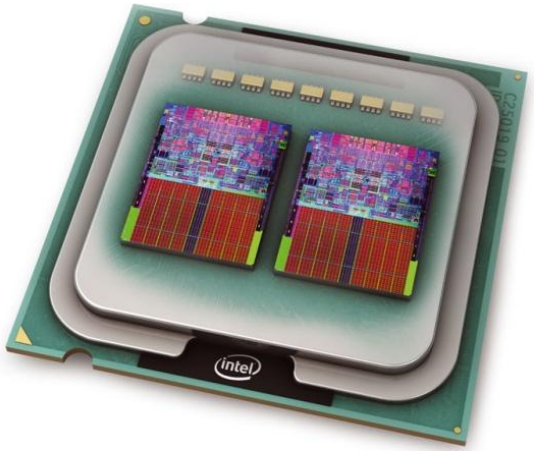
- Single Processor:
  - Has only one actual processor.
  - Concurrent tasks are often multiplexed or multi tasked.
- Multi Processor.
  - Has more than one processors.
  - Concurrent tasks are executed on different processors.

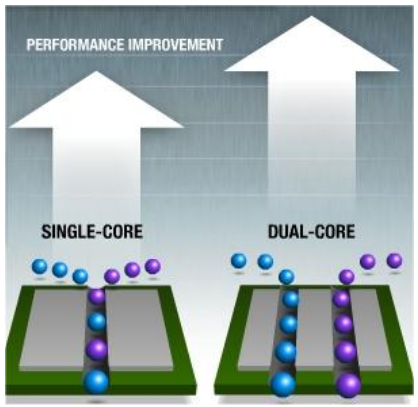




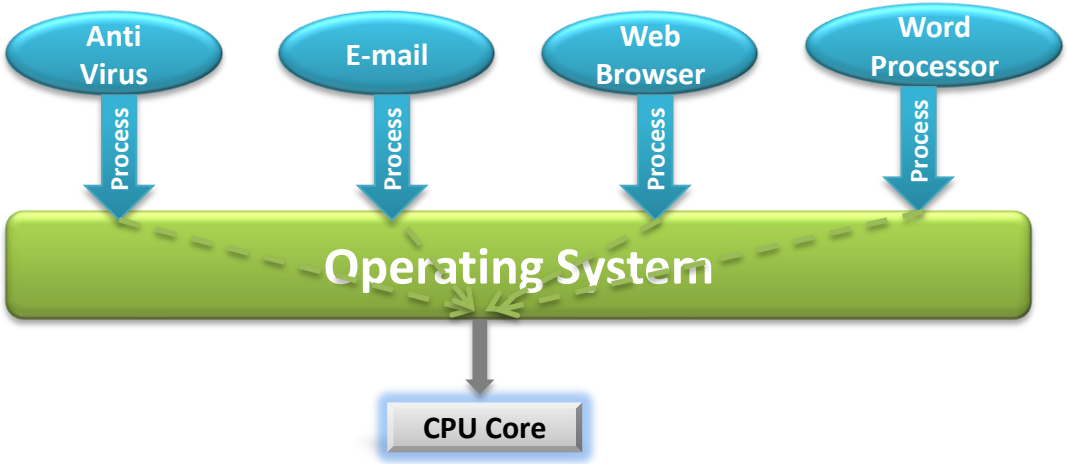
Multi Core?

- Both types of systems can have more than one core per processor (Multicore).
- Multicore processors behave the same way as if you had multiple processors

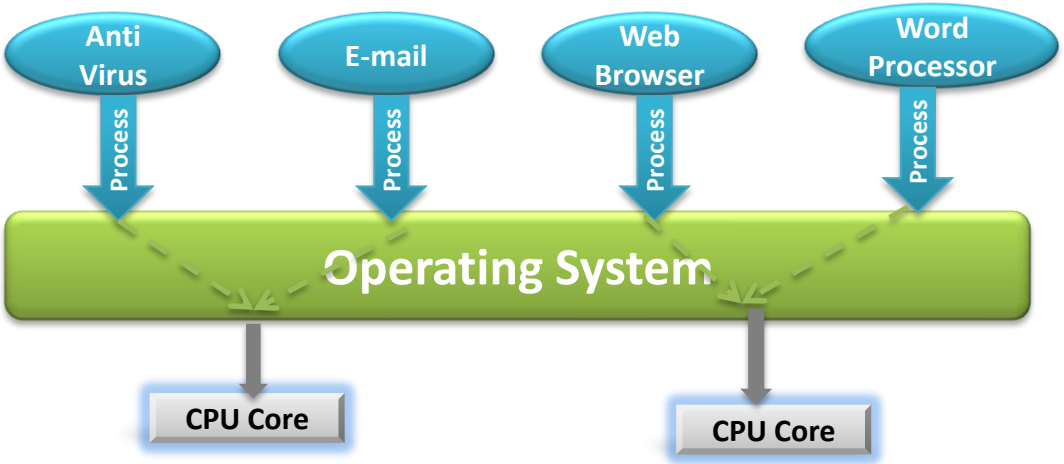




Cores



Single-core systems schedule tasks on 1 CPU to multitask



Dual-core systems enable multitasking operating systems to execute two tasks simultaneously

Why Concurrency?

- Make apps more responsive.
- Make apps faster.
- More efficient CPU usage.
- Better System reliability.

"...and you spent 5.73 years of your life deleting spam from your e-mail."

Perils of Concurrency?

- Starvation.
- Deadlock
- Race conditions

Remember!!!

- Writing correct programs is hard; writing correct concurrent programs is harder. There are simply more things that can go wrong in a concurrent program than in a sequential one.
- Programming concurrency is hard, yet the benefits it provides make all the troubles worthwhile.
- A modern computer has several CPU's or several cores within one CPU.
- The ability to leverage these multi-cores can be the key for a success of an applicatio

Remember!!!

- We may run programs on a single core with multiple threads and later deploy them on multiple cores with multiple threads. When our code runs within a single JVM, both these deployment options have some common concerns :
  - how do we *create* and *manage* **threads**,
  - how do we *ensure* **integrity of data**,
  - how do we *deal* with **locks** and **synchronization**,
    and are our threads **crossing the memory barrier** at the *appropriate times*...?

Starvation

- When a thread waits for
  - input from a user,
  - some external event to occur,
  - another thread to release a lock.
- Prevent starvation by placing a timeout

Deadlock

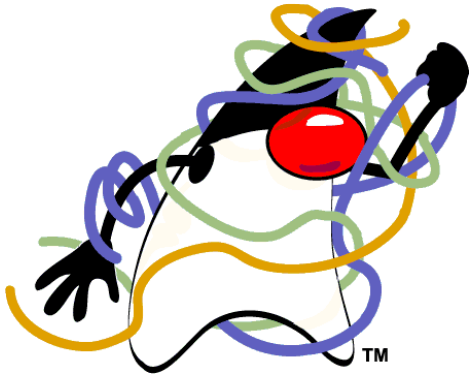
- If two or more threads are waiting on each other for some action or resource.
- Placing a time out will not help avoid deadlocks.
- Prevent deadlock
  - by acquiring resources in a specific order.
  - Avoiding explicit locks and the mutable state that goes with them.
- Tools such as JConsole can help detect deadlocks

Race conditions

- If two threads compete to use the same resource or data, we have a race condition.
- A race condition doesn't just happen when two threads modify data. It can happen even when one is changing data while the other is trying to read it
- Race conditions can render a program's behavior unpredictable, produce incorrect execution, and yield incorrect results.

What is Thread?

- A **Thread** is the smallest unit of processing that can be performed in an OS. In most modern operating systems, a thread exists within a process - that is, a single process may contain multiple threads.
- An abstraction of a unit of execution.



ProKarma

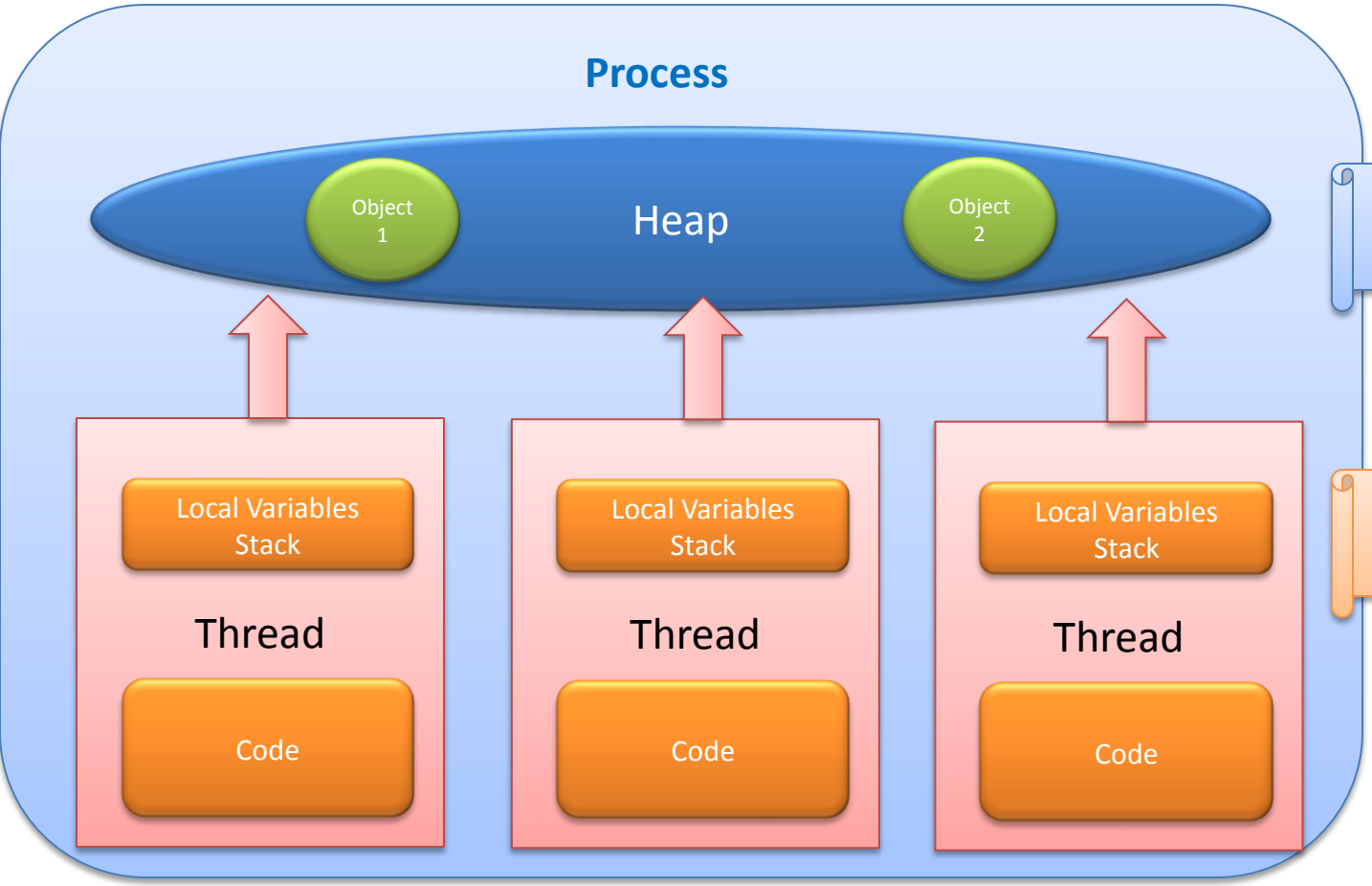
Process and Thread?

- A **Process** is an executing instance of an application. when you double-click the Word Processor icon, you start a process that runs Word Processor.
- A **Thread** is a path of execution within a process. Also, a process can contain multiple threads. When you start Word Processor, the operating system creates a process and begins executing the primary thread of that process. And it has some background threads to print the document while you edit it.

Process and Thread?

- Each Thread has its own local memory
- Threads within the same process share the same address space, whereas different processes do not. This allows threads to read from and write to the same data structures and variables, and also facilitates communication between threads. Communication between processes – also known as IPC– is quite difficult and resource-intensive.
- Sections of code that modify the data structures shared by multiple threads are called **Critical Sections** When a critical section is running in one thread it's extremely important that no other thread be allowed into that critical section. This is called **synchronization**.
- A Thread can do anything a process can do. But since a process can consist of multiple threads, a thread could be considered a '**lightweight**' process.
- If a thread reads shared data it stores this data in its own memory cache.
- Context switching between threads is generally less expensive than in processes.
- The overhead (the cost of communication) between threads is very low relative to processes.

Process and Thread?

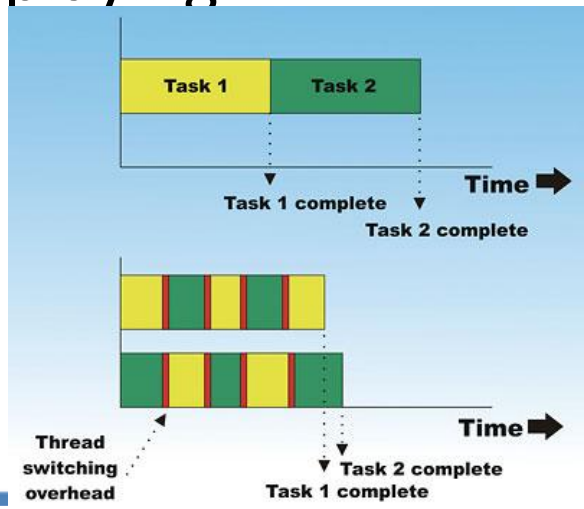


Single Heap per process, shared by all threads

Each Thread has its own memory

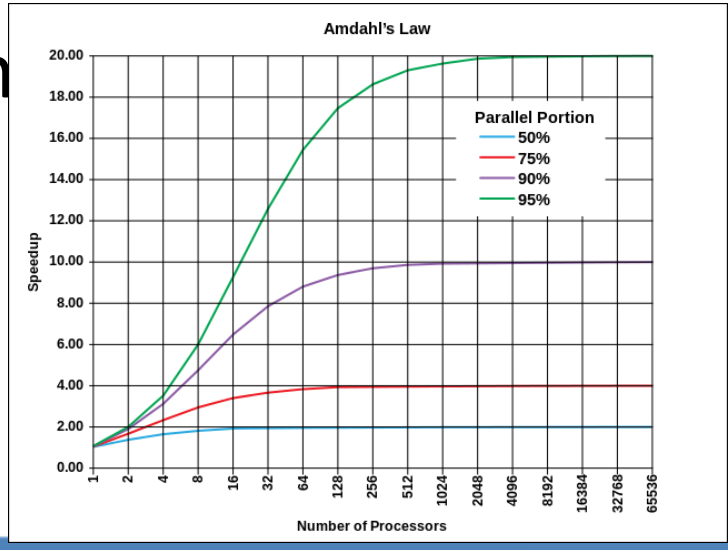
Multi-Tasking Multi-Threading

- You can imagine multitasking as something that allows processes to run concurrently,
- Multithreading allows sub-processes (threads) to run concurrently. Some Examples :
  - You are downloading a video while playing it at the same time.
  - Multithreading is also used extensively in computer-generated animations.

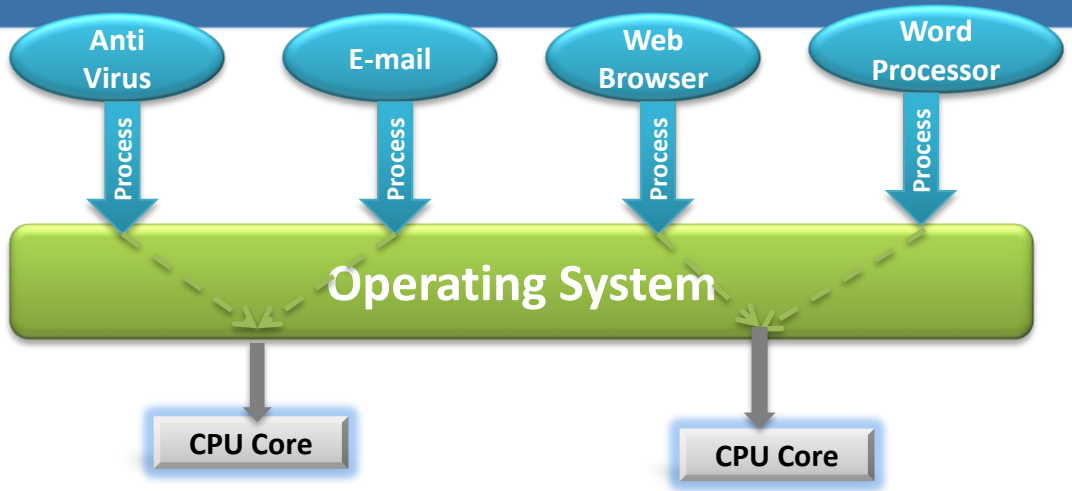


Amdahl's Law

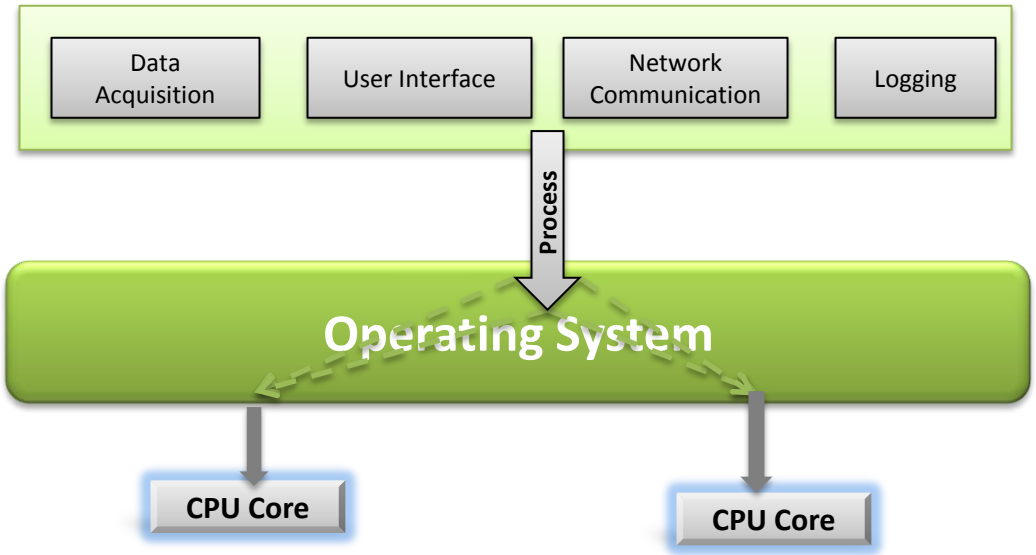
- *Amdahl's Law* is a law governing the speedup of using parallel processors on a problem, versus using only one serial processor.
- If **F** is the percentage of the program which can not run in parallel and **N** is the number of processes then the maximum performance gain is **1/ (F+ ((1-F)/n))**.



Multi-Tasking Multi-Threading



Multi tasking in Dual Core



Multi threading in dual Core

AGENDA - Threads in Java

- Basic Thread Operations
- Thread Creation
- Thread Lifecycle
- Synchronization Mechanisms
- Inter thread Communication
- Executors
- Atomic Variables
- Concurrent Collection
- Old Versus New API
- Java Memory Model
- Tools

Threads in Java

- Java makes concurrency available through the language and APIs.
- Java concurrency is build upon Thread Class which implements Runnable interface (both in java.lang)
- When a Java program starts up, one thread begins running immediately. This is usually called the "**main**" thread of your *program* and is spawned by the Java Virtual Machine.
- Many Java programs have more than one thread without you realizing it. For example, a Swing application has a thread for processing events in addition to the *main* thread.
- A Java program ends when all its threads finish (more specifically, when all its non-daemon threads finish)
- If the initial thread (the one that executes the main() method) ends, the rest of the threads will continue with their execution until they finish. If one of the threads use the System.exit() instruction to end the execution of the program, all the threads end their execution.
- Are there any other threads? Lets see…….

More Background Threads

Nothing little app

```
public class ThreadDumpSample {

    public static void main(String[] args)
        Thread.sleep(100000);
    }
}
```

```
C:\Users\nadeem\Desktop>jps -l
3796 sun.tools.jps.Jps
4756
4276 com.prokarma.app.gtp.thread.ThreadDumpSample

C:\Users\nadeem\Desktop>jstack 4276 > ThreadDump.txt
```

This is how, thread dump is created

Thread Dump of ThreadDumpSample

```
Full thread dump Java HotSpot(TM) 64-Bit Server VM (20.45-b01 mixed mode):
"Low Memory Detector" daemon prio=6 tid=0x00000000064b5800 nid=0x1238 runnable [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
"C2 CompilerThread1" daemon prio=10 tid=0x00000000064b2000 nid=0x11b0 waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
"C2 CompilerThread0" daemon prio=10 tid=0x0000000000529000 nid=0xe30 waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
"Attach Listener" daemon prio=10 tid=0x0000000000527800 nid=0xdbc waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
"Signal Dispatcher" daemon prio=10 tid=0x00000000064a0800 nid=0x81c runnable [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
"Finalizer" daemon prio=8 tid=0x0000000000512800 nid=0x804 in Object.wait() [0x000000000645f000]
   java.lang.Thread.State: WAITING (on object monitor)
    at java.lang.Object.wait(Native Method)
    - waiting on <0x00000007d5801300> (a java.lang.ref.ReferenceQueue$Lock)
    at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:118)
    - locked <0x00000007d5801300> (a java.lang.ref.ReferenceQueue$Lock)
    at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:134)
    at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:171)
"Reference Handler" daemon prio=10 tid=0x0000000000509800 nid=0xf30 in Object.wait() [0x000000000635f000]
   java.lang.Thread.State: WAITING (on object monitor)
    at java.lang.Object.wait(Native Method)
    - waiting on <0x00000007d58011d8> (a java.lang.ref.Reference$Lock)
    at java.lang.Object.wait(Object.java:485)
    at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:116)
    - locked <0x00000007d58011d8> (a java.lang.ref.Reference$Lock)
"main" prio=6 tid=0x000000000060b000 nid=0x1368 waiting on condition [0x000000000254f000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
    at java.lang.Thread.sleep(Native Method)
    at com.prokarma.app.gtp.thread.ThreadDumpSample.main(ThreadDumpSample.java:6)
"VM Thread" prio=10 tid=0x0000000000501000 nid=0x13e4 runnable
"GC task thread#0 (ParallelGC)" prio=6 tid=0x000000000045f000 nid=0x1104 runnable
"GC task thread#1 (ParallelGC)" prio=6 tid=0x0000000000460800 nid=0x4a4 runnable
"VM Periodic Task Thread" prio=10 tid=0x00000000064cc000 nid=0x1398 waiting on condition
JNI global references: 883
```

ProKarma

Basic Operations on Thread

```
public class BasicThreadOperations {

    public static void main(String[] args) {

        // Get Control of the current thread
        Thread currentThread = Thread.currentThread();
        //Print Thread information
        System.out.println("Thread Name : " + currentThread.getName());
        System.out.println("Thread Group : " + currentThread.getThreadGroup());
        System.out.println("Thread Priority : " + currentThread.getPriority());
        System.out.println("Thread is Daemon : " + currentThread.isDaemon());
        System.out.println("Thread State : " + currentThread.getState());
        //Update current thread details
        currentThread.setName("The Main Thread");
        currentThread.setPriority(6);

        System.out.println("\nNew Thread Name : " + currentThread.getName());
        System.out.println("New Thread Priority : " + currentThread.getPriority());
        System.out.println();

        for (int i = 0; i < 6; i++) {
            System.out.println("Current value of i = " + i);
            System.out.println("Going to Sleep for 1 second");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("Somebody interrupted " + currentThread + " for un
            }
            System.out.println("Woke up");
        }
        System.out.println("Going to Die...don't stop me.");
    }
}
```

Out put

```
Thread Name : main
Thread Group : java.lang.ThreadGroup[name=main,maxpri=10]
Thread Priority : 5
Thread is Daemon : false
Thread State : RUNNABLE

New Thread Name : The Main Thread
New Thread Priority : 6

Current value of i = 0
Going to Sleep for 1 second
Woke up
Current value of i = 1
Going to Sleep for 1 second
Woke up
Current value of i = 2
Going to Sleep for 1 second
Woke up
Current value of i = 3
Going to Sleep for 1 second
Woke up
Current value of i = 4
Going to Sleep for 1 second
Woke up
Current value of i = 5
Going to Sleep for 1 second
Woke up
Going to Die...don't stop me.
```

Exercise

- Thread.interrupted() & isInterrupted()
- Play with Daemon and non Daemon threads.
- Process Un controlled exception in Thread using UncaughtExceptionHandler
- Play with ThreadGroups.

Thread Creation

- Thread class encapsulate an object that is runnable, There are two ways to create Threads
  - Extend java.lang.Thread Class and overriding the run() method
  - Building a class that implements the Runnable interface and then creating an object of the Thread class passing the Runnable object as a parameter.

Extending Thread Class

- Create a Class by extending *Thread* Class
- Override the *run* method from Thread Class, to write the functionality the thread should perform.
  - run() is the entry point for new thread that is created.
  - run() method is where the entire action of the thread would take place, it is almost like main() method.
- Create an instance of the newly created Class in main class.
- Start the thread by calling start() of thread.
  - Calling run() of thread is like calling any plain method of an object.
  - Java would only crate real thread if start() method is invoked

Implementing Runnable

- Create a Class by Implementing *Runnable* Interface
- Implement the *run* method, to write the functionality the thread should perform.
- Create an Object of Thread class passing the instance of Runnable, you just created.
- Start the thread by calling start() of thread.

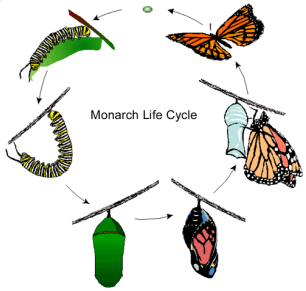
Best Way To Create Threads

- ?

Lab Time

- Run DemoExtendedThread and observe the output.
- Run DemoMultiplicationTableCalculator and observe the output

Thread Lifecycle



Thread t = new Thread();

New

Thread.start();

Ready to Run

Scheduler Swap

Chosen by Scheduler

Running

Thread.sleep(time) over

Thread.sleep(time);

Thread.run() execution is over (done)

Acquire Lock Interrupt I/O Completes

Issue I/O Request Or Enter Synchronized Statement

Thread.wait();
Thread.wait(timeout);

Interval Expires
Object.notify/Object.notifyAll

Sleeping

Terminated

Another Thread closes Socket

Blocked On IO or Sync

Waiting

July 2, 2013

Important methods of Thread

- **start()**
  - Makes the Thread Ready to run
- **isAlive()**
  - A Thread is alive if it has been started and not died.
- **sleep(milliseconds)**
  - Sleep for number of milliseconds.
- **isInterrupted()**
  - Tests whether this thread has been interrupted.
- **Interrupt()**
  - Indicate to a Thread that we want to finish. If the thread is blocked in a method that responds to interrupts, an InterruptedException will be thrown in the other thread, otherwise the interrupt status is set.
- **join()**
  - Wait for this thread to die.
- **yield()**
  - Causes the currently executing thread object to temporarily pause and allow other threads to execute.

sleep() yield() wait()

- sleep(n) says ***"I'm done with my time slice, and please don't give me another one for at least n milliseconds."*** The OS doesn't even try to schedule the sleeping thread until requested time has passed.
- yield() says ***"I'm done with my time slice, but I still have work to do."*** The OS is free to immediately give the thread another time slice, or to give some other thread or process the CPU the yielding thread just gave up.
- .wait() says ***"I'm done with my time slice. Don't give me another time slice until someone calls notify()."*** As with sleep(), the OS won't even try to schedule your task unless someone calls notify() (or one of a few other wakeup scenarios occurs).

sleep() and wait()

Object.wait()	Thread.sleep()
Called from Synchronized block	No Such requirement
Monitor is released	Monitor is not released
Awake when notify() and notifyAll() method is called on the monitor which is being waited on.	Not awake when notify() or notifyAll() method is called, it can be interrupted.
not a static method	static method
wait() is generally used on condition	sleep() method is simply used to put your thread on sleep.
Can get *spurious wakeups* from wait (i.e. the thread which is waiting resumes for no apparent reason). We Should always wait whilst spinning on some condition , ex : *synchronized {* *while (!condition) monitor.wait();* *}*	This is deterministic.
Releases the lock on the object that wait() is called on	Thread does *not* release the locks it holds

Synchronization

- Coordinating activities and data access among multiple threads is **Synchronization**.
- In Concurrent applications it is normal that multiple threads read/write same variable, have access to the same file. These shared resources can provoke error situations or data inconsistency if effective mechanism is not employed.
- **Critical Section** comes to our rescue in these
- A *critical section* is a block of code that access a shared resource and can't be executed by more than one thread at the same time.

Synchronization cont..

- To help programmers implement critical section Java offers synchronization mechanism.
- When a thread wants access to a critical section, it uses one of those synchronization mechanisms to find out if there is any other thread executing the critical section If not, the thread enters the critical section. Otherwise, the thread is suspended by the synchronization mechanism until the thread that is executing the critical section ends it.
- When more than one thread is waiting for a thread to finish the execution of a critical section, the JVM chooses one of them, and the rest wait for their turn.

Monitor

- Threads are synchronized in Java through the use of a *monitor.*
- Think of a monitor as an object that enables a thread to access a resource. Only one thread can use a monitor at any one time period.
- A thread can own a monitor only if no other thread owns the monitor.
- If the monitor is available, a thread can own the monitor and have exclusive access to the resource associated with the monitor.
- If the monitor is not available, the thread is suspended until the monitor becomes available. Programmers say that the thread is waiting for the monitor.

Monitor

- Every object instance has a monitor that can be locked by one thread at a time i.e., every object contains a “monitor” that can be used to provide mutual exclusion access to critical sections of code. The critical section is specified by marking a *method* or *code block* as **synchronized.**
- The task of acquiring a monitor for a resource, happens behind the scenes in Java.
- Java's monitor supports two kinds of thread synchronization
  - *mutual exclusion (of Critical Section):* supported in the Java virtual machine vi ***synchronized*** keyword, enables multiple threads to independently work on shared data without interfering with each other
  - *Cooperation (using Condition):* supported in the Java virtual machine via the wait and notify methods of class Object, enables threads to work together towards a common goal.

Volatile

- What is the expected output?
- What is the problem here?

```
public class RaceCondition {

    private static boolean done;

    public static void main(String[] args) throws InterruptedException {

        new Thread(new Runnable() {
            public void run() {
                int i = 0;
                while(!done) { i++; }
                System.out.println("Done! [" + Thread.currentThread().getName() + "]");

            }
        }).start();

        TimeUnit.SECONDS.sleep(1);
        done = true;
        System.out.println("flag done set to true [" + Thread.currentThread().getName() + "]");
    }
}
```

- The program just prints (in windows 7 x64 bit) "flag done set to true [main]" and stuck for ever.

Volatile cont..

- The problem with the previous example is that the change by the main thread to the field ***done*** may not be visible to the thread we created.
- First, the JIT compiler may optimize the while loop; after all, it does not see the variable ***done*** changing within the context of the thread.
- Furthermore, the second thread may end up reading the value of the flag from its registers or cache instead of going to memory. As a result, it may never see the change made by the first thread to this flag.
- We can quickly fix the problem by marking the flag ***done*** as volatile.

volatile to the rescue..

- The volatile keyword tells the JIT compiler not to perform any optimization that may affect the ordering of access to that variable
- It warns that the variable may change behind the back of a thread and that each access, read or write, to this variable should bypass cache and go all the way to the memory.

Volatile Issues.

- Making all variables volatile may avoid the problem but will result in very poor performance because every access has to cross the *memory barrier*.
- *volatile* does not help with atomicity when multiple fields are accessed, because the access to each of the *volatile* fields is separately handled and not coordinated into one access—this would leave a wide opportunity for threads to see partial changes to some fields and not the others.

Volatile Issues.

- Volatile atomicity issue can be addressed by preventing direct access to the *flag* and channeling all access through the ***synchronized*** getter and setter.
- The ***synchronized*** marker makes the calling threads cross the memory barrier both when they enter and when they exit the synchronized block.
- A thread is guaranteed to see the change made by another thread if both threads synchronize on the same instance and the change-making thread happens before the other thread

Memory Barrier

- It is the copying from local or working memory to main memory.
- A change made by one thread is guaranteed to be visible to another thread only if the writing thread crosses the memory barrier and then the reading thread crosses the memory barrier
- ***synchronized*** and ***volatile*** keywords force that the changes are globally visible on a timely basis; these help cross the *memory barrier*.
- Quite a few operations in the concurrency API implicitly cross the memory barrier:
  - volatile, synchronized, methods on Thread such as start() and interrupt(),
  - methods on ExecutorService
  - and some synchronization facilitators like CountDownLatch

Local Thread Variables

- Synchronization is often an expensive operation that can limit the performance of a multi-threaded program. Using thread-local data structures instead of data structures shared by the threads can reduce synchronization in certain cases, allowing a program to run faster.
- Thread local variables are specific to a particular thread such that every thread owns a separate copy for that variable
- After a thread terminates its all thread-local variables are garbage collected unless they are not referenced by other objects.
- Thread local storage support is available since Java 1.2 and its performance has been significantly enhanced with the latest Java releases.

Local Thread Variable Advantages

- Reduce synchronization overheads since there will be no locking when each thread accesses its own thread-local variable
- Simplify coding (no need to worry about critical sections and locking …)
- Encapsulate non-thread-safe classes to make them thread safe

Basic Inter Thread Communication (cooperation)

- It is all about making synchronized threads communicate with each other.
- A thread is paused running in its critical section and another thread is allowed to enter(or lock) into same critical section.
- Inter thread communication is achieved using methods of java.lang.Object.
  - Wait() : tells calling thread to give up the monitor and go to sleep, until some other thread enter the same monitor and call notify() or notifyAll().
  - Notify() : wakes up the single thread that is waiting on this object's monitor, if there are more threads waiting on this object, one of them is chosen to be awakened. The Choice is arbitrary and occurs at the decision of JVM.
  - NotifyAll() : wakes up all the threads that are waiting on this object's monitor.

Basic Inter Thread Communication (cooperation)

- All three methods can be called only from within a **synchronized** context or an IllegalMonitorStateException will be thrown.
- Always wait inside a loop that checks the condition being waited on – this addresses the timing issue if another thread satisfies the condition before the wait begins. Also, it protects your code from spurious wake-ups that can (and do) occur.
- Always ensure that you satisfy the waiting condition before calling notify or notifyAll. Failing to do so will cause a notification but no thread will ever be able to escape its wait loop.

Java Synchronization Mechanisms

- synchronized keyword.
- Explicit Locks
- CountDownLatch
- CyclicBarrier
- Semaphores

Synchronized Keyword

- The synchronized keyword can be applied either to a block or to a method.
- It indicates that before entering the block or method, a thread must acquire the appropriate lock
  - For a method, that means acquiring the lock belonging to the object instance (or the lock belonging to the Class object for static synchronized methods).
  - For a block, the programmer should indicate which object's lock is to be acquired.

Synchronized Keyword

- Only objects—not primitives—can be locked.
- Locking an array of objects doesn't lock the individual objects.
- A synchronized method can be thought of as equivalent to a **synchronized (this) { ... }** block that covers the entire method (but note that they're represented differently in byte code).
- A static synchronized method locks the Class object, because there's no instance object to lock.
- If you need to lock a class object, consider carefully whether you need to do so explicitly, or by using getClass(), because the behavior of the two approaches will be different in a subclass.
- Synchronization in an inner class is independent of the outer class (to see why this is so, remember how inner classes are implemented).
- synchronized doesn't form part of the method signature, so it can't appear on a method declaration in an interface.
- Unsynchronized methods don't look at or care about the state of any locks, and they can progress while synchronized methods are running.
- Java's locks are reentrant. That means a thread holding a lock that encounters a synchronization point for the same lock (such as a synchronized method calling another synchronized method in the same class) will be allowed to continue.

Synchronized Keyword

- Synchronizing instance method

```
class SpeechSynthesizer {
    synchronized void say( String words ) {
        // speak
    }
}
```

- Synchronizing multiple methods.

```
class SpreadSheet {
    int cellA1, cellA2, cellA3;

    synchronized int sumRow() {
        return cellA1 + cellA2 + cellA3;
    }

    synchronized void setRow( int a1, int a2, int a3 ) {
        cellA1 = a1;
        cellA2 = a2;
        cellA3 = a3;
    }
    ...
}
```

- Synchronizing a block of code.

```
synchronized ( myObject ) {
    // Functionality that needs exclusive access to resources
}
```

```
synchronized void myMethod () {
    ...
}
```

is equivalent to:

```
void myMethod () {
    synchronized ( this ) {
        ...
    }
}
```

Synchronized Keyword

- Marking a method as synchronized

```
public class Incrementor {

    private int value = 0;

    public synchronized int increment() {
        return ++value;

    }
}
```

- Using explicit lock object

```
public class Incrementor {

    private final Object lock = new Object();

    private int value = 0;

    public int increment() {
        synchronized (lock) {
            return ++value;
        }
    }
}
```

- Using the this Monitor

```
public class Incrementor {

    private int value = 0;

    public int increment() {
        synchronized (this) {
            return ++value;
        }
    }
}
```

Explicit Locks

- Java provides another mechanism for the synchronization of blocks of code. It's a more powerful and flexible mechanism than the synchronized keyword.
- It's based on the Lock interface and classes that implement it (as ReentrantLock)

```
Lock lock = new ReentrantLock();

// method or block
lock.lock();
try {
     // body of method or block ...
} finally {
    lock.unlock()
}
```

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Incrementor {
    private final Lock lock = new ReentrantLock();
    private int value = 0;
    public int increment() {
        lock.lock();
        try {
            return ++value;
        } finally {
            lock.unlock();
        }
    }
}
```

Explicit Locks

- Advantages of new Approach
  - It allows the structuring of synchronized blocks in a more flexible way. The Lock interfaces allow you to get more complex structures to implement your critical section
  - different types of locks (such as reader and writer locks).
  - Not restrict locks to blocks (allow a lock in one method and unlock in another).
  - If a thread cannot acquire a lock (for example, if another thread has the lock), allow the thread to back out or carry on or do something else—a tryLock() method.
  - Allow a thread to attempt to acquire a lock and give up after a certain amount of time.
  - better performance than the synchronized keyword

Explicit Locks

- The key to realizing all of these possibilities is the Lock interface in java.util.concurrent.locks
- This ships with a couple of implementations:
  - **ReentrantLock :** This is essentially the equivalent of the familiar lock used in Java synchronized blocks, but it's slightly more flexible.
  - **ReentrantReadWriteLock :** This can provide better performance in cases where there are many readers but few writers.

Lock Condition

- Condition implements the wait/notify semantics in an API but with several additional features:
  - create multiple Conditions per Lock
  - Interruptible waiting
- Conditions are obtained from a Lock instance as follows:

```
Lock lock = ...
Condition condition = lock.newCondition();
lock.lock();
condition.await(); // block, waiting for signal()
lock.unlock();

// meanwhile, in another thread...
lock.lock();
condition.signal();
lock.unlock();
```

Lock Condition Example

```
public class BoundedBlockingBuffer<T> {

    private final T[] buffer;
    private final int capacity;
    private int front;
    private int rear;
    private int count;

    private final Lock lock = new ReentrantLock();
    private final Condition notFull = lock.newCondition();
    private final Condition notEmpty = lock.newCondition();

    @SuppressWarnings("unchecked")
    public BoundedBlockingBuffer(int capacity) {
        this.capacity   = capacity;
        buffer          =(T[]) new Object[capacity];
    }

    public void deposit(T data) throws InterruptedException {
        lock.lock();
        try {
            while (count == capacity) {
                notFull.await();
            }
            buffer[rear] = data;
            rear = (rear + 1) % capacity;
            count++;
            notEmpty.signal();

        } finally {
            lock.unlock();
        }
    }
    public T fetch() throws InterruptedException {
        lock.lock();
        try {

            while (count == 0) {
                notEmpty.await();
            }
            T result = buffer[front];
            front = (front + 1) % capacity;
            count--;
            notFull.signal();
            return result;

        } finally {
            lock.unlock();
        }
    }
}
```

Waiting for multiple concurrent events using CountDownLatch

- `CountDownLatch` is a class that allows one or more threads to wait until a set of operations are made. i.e A *countdown latch* lets one or more threads wait at a "gate" until another thread opens this gate, at which point these other threads can continue.
- Suppose a stone can be lifted by 10 people so you will wait for all 10 to come. Then only you can lift the stone.
- The `CountDownLatch` class has three basic elements:
  - The initialization value that determines how many events the `CountDownLatch` class waits for
  - The `await()` method, called by the threads that wait for the finalization of all the events
  - The `countDown()` method, called by the events when they finish their execution

CountDownLatch

- Example

```
CountDownLatch latch = new CountDownLatch( 2 ); // count from 2

// thread 1
latch.await(); // blocks thread 1

// thread 2
latch.countDown(); // count is 1
latch.countDown(); // count is 0, thread 1 proceeds
```

- Limitations
  - A Latch cannot be reused after reaching the final count
  - You need to know in advance how many threads would be created.

Two way of using CountDownLatch

- Many threads blocking on "await()" that would all start simultaneously when the countdown reached zero.

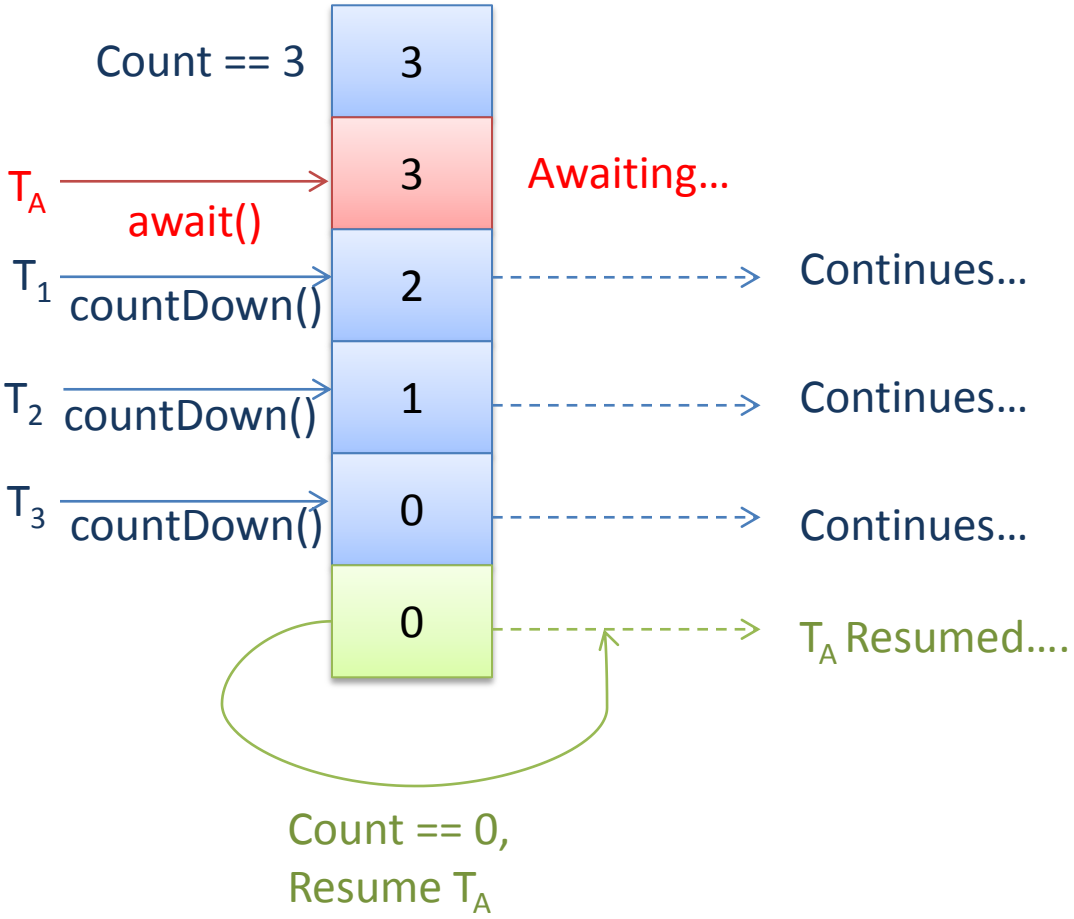
```
final CountDownLatch countdown = new CountDownLatch(1);
for (int i = 0; i < 10; ++ i){
   Thread racecar = new Thread() {
      public void run()    {
         countdown.await(); //all threads waiting
         System.out.println("Vroom!");
      }
   };
   racecar.start();
}
System.out.println("Go");
countdown.countDown();   //all threads start now!
```

- Main thread blocking on "await()" until all threads finishes execution and calls countDown on the same latch

```
final CountDownLatch latch = new CountDownLatch(num_threads);
for (int i = 0; i < num_threads; ++ i)
{
   Thread t = new Thread() {
      public void run() {
         // no lock to acquire!
         System.out.println("Going to count down...");
         latch.countDown();
      }
   };
   t.start();
}

System.out.println("Going to await...");
latch.await();
System.out.println("Done waiting!");
```

CountDownLatch

Works like a counter – allows one or more threads to wait on await() method for another N threads to callcountDown() method N times (total number of calls should be N).



As the illustration shows only T_A is awaiting. $T_{1,2,3}$ call countDown() and proceed their execution. When T_3 calls countDown(), T_A gets unlocked and can resume its work.

Synchronizing tasks in a common point using CyclicBarrier

- A *cyclic barrier* lets a group of threads wait for each other to reach a common *barrier point*.
- If you are going to a picnic, and you need to first meet at some common point from where you all will start your journey.
- The `java.util.concurrent.CyclicBarrier` class implements this synchronizer

Synchronizing tasks in a common point using CyclicBarrier

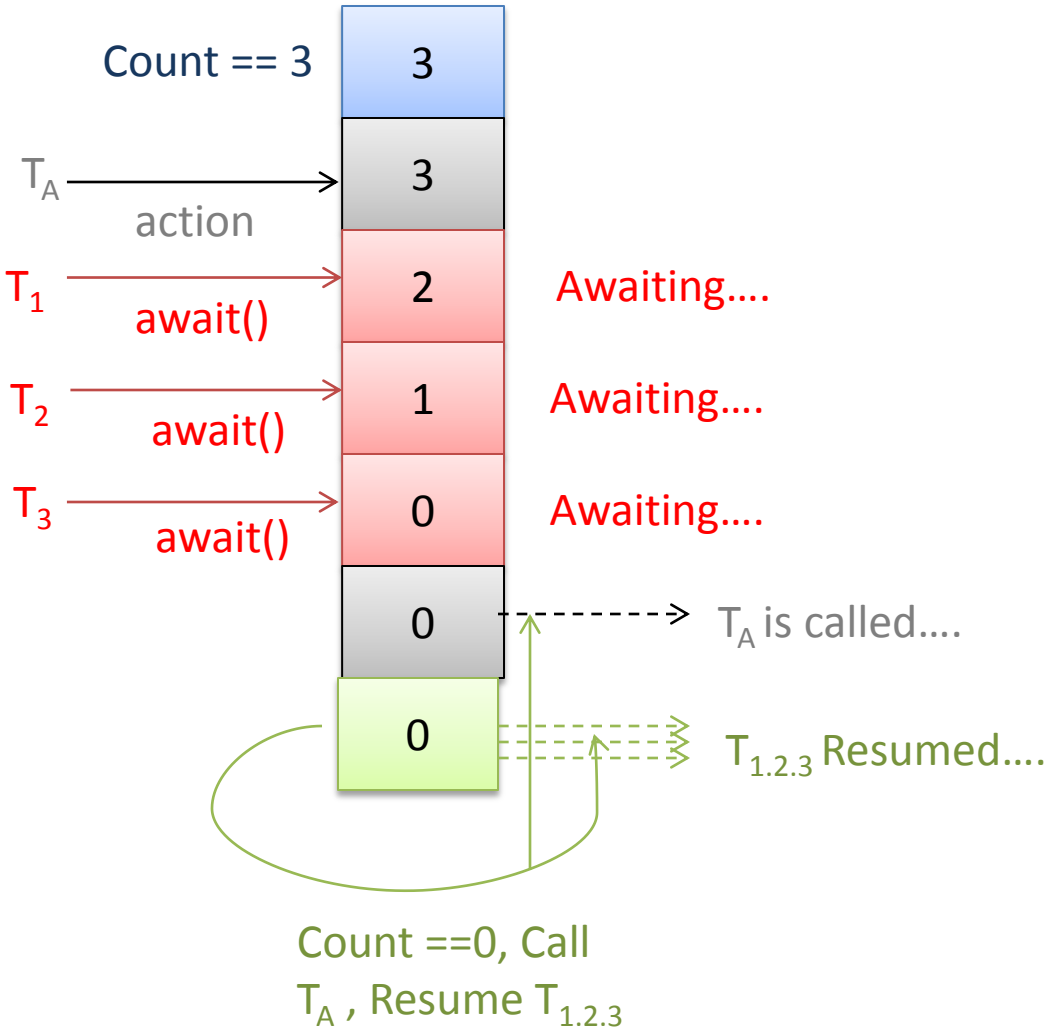
- The barrier is constructed with the following:
  - the **number of threads** that will be participating in the parallel operation;
  - optionally, an **amalgamation routine** to run at the end of each stage/iteration.
- Then, at each stage (or on each iteration) of the operation:
  - each thread carries out its portion of the work;
  - after doing its portion of the work, each thread **calls the barrier's await()** method;
  - the await() method **returns *only* when**:
    - *all* threads have called await();
    - the **amalgamation** method has run (the barrier calls this on the last thread to call await() before releasing the awaiting threads).
  - if *any* of the threads is **interrupted or times out** while waiting for the barrier, then **the barrier is "broken"** and *all* other waiting threads receive a `BrokenBarrierException`.

CyclicBarrier

Allows N-1 treads to wait on await() method until Nth thread calls await() method and then they resume their execution

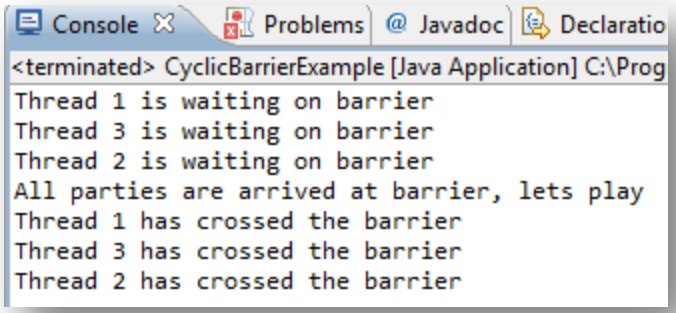
As this illustration shows there is a barrier for 3 parties. When T1 and T2 come they wait. When T3 comes an optional TA handler is called and (after it completes?) T1,2,3 resume their work.

You can reuse CyclicBarrier many times by calling reset() method.



CyclicBarrier Example

```
8  public class CyclicBarrierExample {
9      //Runnable task for each thread
10     private static class Task implements Runnable {
11
12         private CyclicBarrier barrier;
13
14         public Task(CyclicBarrier barrier) {
15             this.barrier = barrier;
16         }
17
18         public void run() {
19             try {
20                 System.out.println(Thread.currentThread().getName() + " is waiting on barrier");
21                 barrier.await();
22                 System.out.println(Thread.currentThread().getName() + " has crossed the barrier");
23             } catch (InterruptedException ex) {
24                 Logger.getLogger(CyclicBarrierExample.class.getName()).log(Level.SEVERE, null, ex);
25             } catch (BrokenBarrierException ex) {
26                 Logger.getLogger(CyclicBarrierExample.class.getName()).log(Level.SEVERE, null, ex);
27             }
28         }
29     }
30
31     public static void main(String args[]) {
32
33         //creating CyclicBarrier with 3 parties i.e. 3 Threads needs to call await()
34         final CyclicBarrier cb = new CyclicBarrier(3, new Runnable() {
35             public void run(){
36                 //This task will be executed once all thread reaches barrier
37                 System.out.println("All parties are arrived at barrier, lets play");
38             }
39         });
40
41         //starting each of thread
42         Thread t1 = new Thread(new Task(cb), "Thread 1");
43         Thread t2 = new Thread(new Task(cb), "Thread 2");
44         Thread t3 = new Thread(new Task(cb), "Thread 3");
45
46         t1.start();
47         t2.start();
48         t3.start();
49
50     }
51 }
```



Changing data between concurrent tasks using Exchanger

- An *exchanger* lets a pair of threads exchange objects at a synchronization point
- The java.util.concurrent.Exchanger class implements this synchronizer.
- In more detail, the Exchanger class allows the definition of a synchronization point between two threads. When the two threads arrive to this point, they interchange a data structure so the data structure of the first thread goes to the second one and the data structure of the second thread goes to the first one.

Exchanger

- Example

```
Exchanger<ByteBuffer> xchange = new Exchanger<ByteBuffer>();

// thread 1
Buffer nextBuf = xchange.exchange( buffer1 ); // blocks

// thread 2
Buffer nextBuf = xchange.exchange( buffer2 );

// buffers exchanged, both threads continue...
```

Controlling concurrent access to a resource using Semaphores

- A *semaphore* maintains a set of permits for restricting the number of threads that can access a limited resource.
- The java.util.concurrent.Semaphore class implements this synchronizer

Semaphores

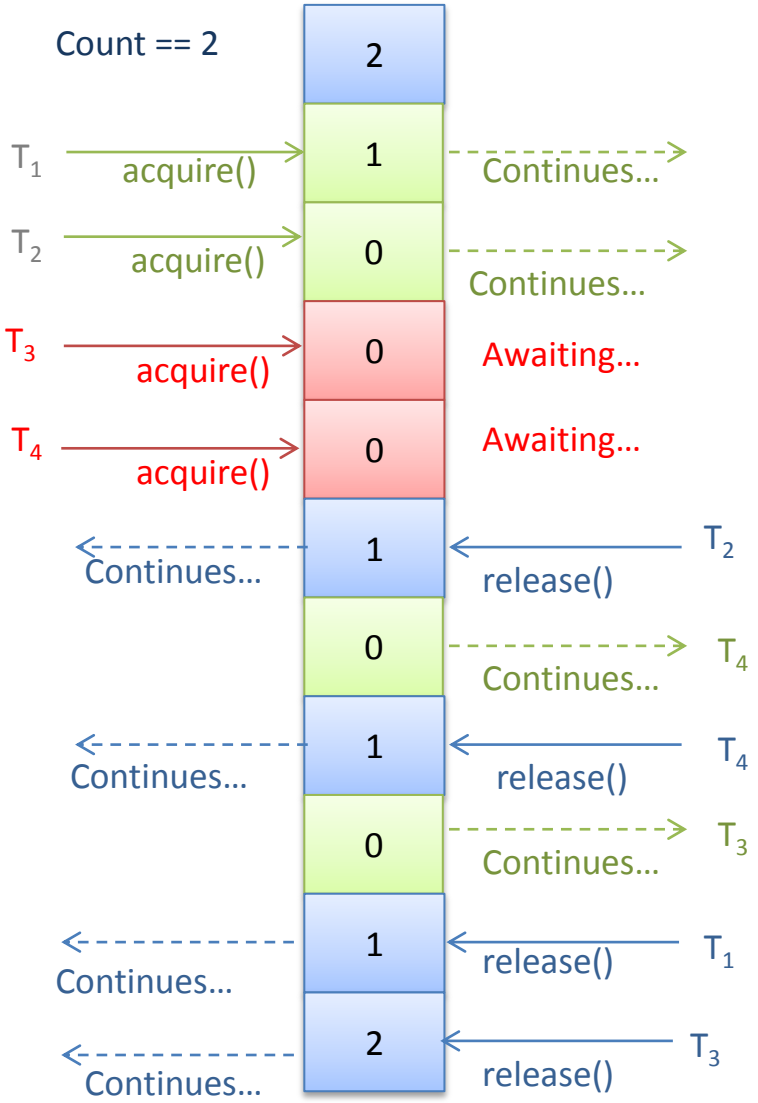
- Example

```
int concurrentReaders = 5;
boolean fair = true;
Semaphore sem = new Semaphore( concurrentReaders, fair );

Data readData() throws InterruptedException {
    sem.acquire();
    // read data ...
    sem.release();

    return data;
}
```

Semaphores

Maintains N permits which is shared between M threads. Each thread can acquire or release permits. If there are not enough permits thread blocks until other threads release necessary amount of permits.

There are 2 permits and 4 threads. Each thread acquires/releases only 1 permit. T1 and T2 do this and proceed their execution without being blocked. T3 and T4 block and wait for permits because there are no any permits available. T2 releases one permit and T4 proceeds its calculations (T3 is also might be chosen?). Next T4 releases its permit and T3 resumes. An finally T1 and T3 release their permits.



Executor Framework

- Old threading API handles threads and provides thread safety but doesn't quite consider performance or scalability.
- Even though threads are relatively lightweight, it takes time and resources to create them.
- Thread safety is provided at the expense of scalability—overly conservative synchronization limits performance.

Executor Framework

- If you have to develop a program that runs lot of concurrent tasks the old approach (create `Runnable` objects and then create corresponding `Thread` objects to execute them) has following disadvantages
  - Coding required to Management (creation, ending, obtaining results) of Thread objects.
  - to execute a big number of tasks, creating a thread object per task can affect the through put of the application and may saturate entire system.

Executor Framework

- Since Java 5, the Java concurrency API provides a mechanism that aims at resolving problems. This mechanism is called the **Executor framework** and is around the `Executor` interface, its sub interface `ExecutorService`, and the `ThreadPoolExecutor` class that implements both interfaces.
- *Executor framework* takes care of thread life cycle management (invoking, scheduling, and executing) according to a set of policies so that you can focus on the tasks to be done.
- Also provides an implementation of thread pools through the ThreadPoolExecutor class.
- This provides a way to decouple task submission from task execution policy.
- Makes it easy to change task execution policy
- Supports several different execution policies by default, and developers can create Executors supporting arbitrary execution policies
- It is based on Producer-Consumer pattern. Activities that submit tasks are producers and activities that perform tasks are consumers.

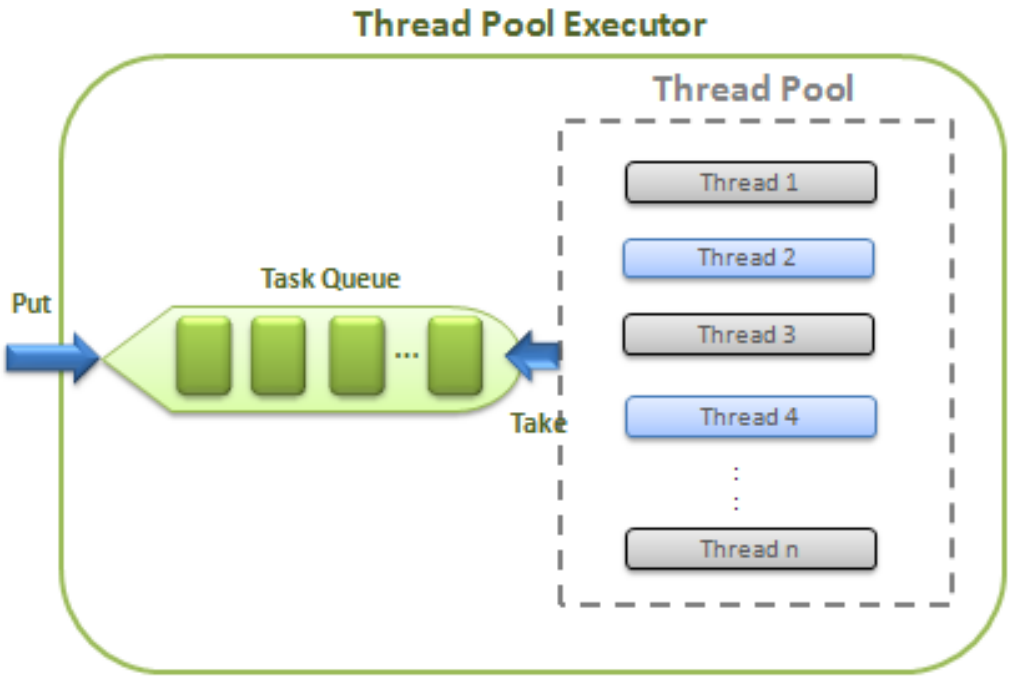
Thread Pools

- Consider writing a web server that handles an arbitrary number of clients
- One way of implementing this is to spawn a new thread for each client that makes a request
- However under high load this could crash the server, due to the large amount of resources used to create and maintain the threads
- Worse we are creating far more threads than can be handled by the server, probably most threads will sit around waiting for CPU time.

Thread Pools

- A smarter way to handle this is to use the idea of a thread pool
- We create a fixed number of threads called a thread pool
- We use a queue data structure into which tasks are submitted
- Each free thread picks a task of the queue and processes it
- If all threads are busy, tasks wait in queue for threads to free up
- This way we never overload the server, and get the most efficient implementation

ThreadPoolExecutor



Executor Framework

- Executor interface describes an object which executes Runnable.

```
public interface Executor {

    /**
     * Executes the given command at some time in the future.  The command
     * may execute in a new thread, in a pooled thread, or in the calling
     * thread, at the discretion of the <tt>Executor</tt> implementation.
     *
     * @param command the runnable task
     * @throws RejectedExecutionException if this task cannot be
     * accepted for execution.
     * @throws NullPointerException if command is null
     */
    void execute(Runnable command);
}
```

- A class that wishes to use the Executor framework, must implement the Executor interface and provide an implementation for execute.

```
//This executor creates a new thread for each task submitted
public class SimpleExecutor implements Executor {

    public void execute(Runnable command) {
        new Thread(command).start();
    }
}
```

Limitation of Executor Interface

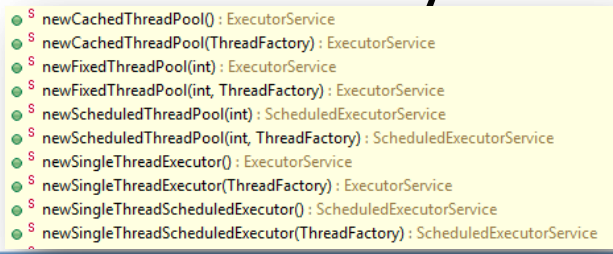
- Although **Executor** is easy to use, this interface is limited in various ways:
  - **Executor** focuses exclusively on Runnable. Because Runnable's run() method does not return a value, there is no convenient way for a runnable task to return a value to its caller.
  - **Executor** does not provide a way to track the progress of executing runnable tasks, cancel an executing runnable task, or determine when the runnable task finishes execution.
  - **Executor** cannot execute a collection of runnable tasks.
  - **Executor** does not provide a way for an application to shut down an **executor** (much less to properly shut down an **executor**).

ExecutorService, ScheduledExecutorService

- These limitations are addressed by java.util.concurrent.ExecutorService interface, which extends **Executor**, and whose implementation is typically a thread pool (a group of reusable threads).
- ScheduledExecutorService interface extends ExecutorService and describes an **executor** that lets you schedule tasks to run once or to execute periodically after a given delay.
- Although you could create your own Executor, ExecutorService, and ScheduledExecutorService implementations, the concurrency utilities offer a simpler alternative: java.util.concurrent.Executors.

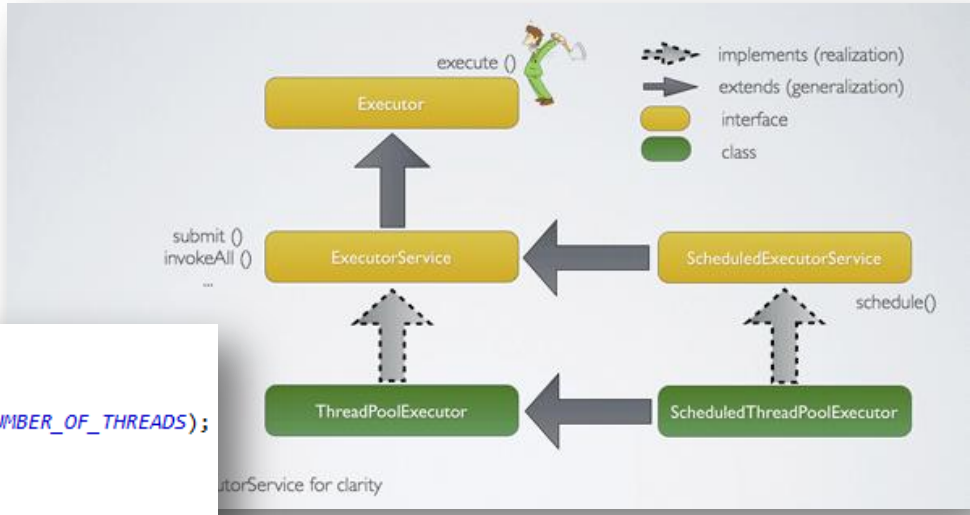
Executors – The Factory Guy

- The Executor framework comes with several implementations of Executor that implement different execution policies.
  - **Executors.newCachedThreadPool**: Creates a thread pool of unlimited size, but if threads get freed up, they are reused
  - **Executors.newFixedThreadPool**: Create a thread pool of fixed size, if pool is exhausted, tasks must wait till a thread becomes free
  - **Executors.newSingleThreadExecutor**: Creates only a single thread, tasks are executed sequentially form the queue.



Executor Example

- A WebServer




```
public class WebServer {
    private static final int DEFAULT_PORT       = 80;
    private static final int NUMBER_OF_THREADS  = 100;
    private static final Executor executor  = Executors.newFixedThreadPool(NUMBER_OF_THREADS);

    public static void main(String[] args) throws IOException {
        ServerSocket socket = newSoceketServer();
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = newRequestHandler(connection);
            executor.execute(task);
        }
    }

    private static Runnable newRequestHandler(final Socket connection) {
        Runnable task = new Runnable() {
            public void run() {
                handleRequest(connection);
            }
            private void handleRequest(Socket connection) {
            }
        };
        return task;
    }

    private static ServerSocket newSoceketServer() throws IOException {
        return new ServerSocket(DEFAULT_PORT);
    }
}
```

Algorithm for Using Executor

1. **Create an Executor**
   - You first create an instance of an *Executor* or *ExecutorService* in some global context (Utilizing Executors)
2. **Create one or more tasks**
   - You are required to have one or more tasks to be performed as instances of either *Runnable* or *Callable*.
3. **Submit the task to the Executor**
   - submit a task to *ExecutorService* using either the *submit()* or *execute()* methods.
4. **Execute the task**
   - The *Executor* is then responsible for managing the task's execution as well as the thread pool and queue.
5. **Shutdown the Executor**
   - At application shutdown, we terminate the executor by invoking its *shutdown()* method.

Determining Number of Threads

- Number of threads = Number of Available Cores / (1 - Blocking Coefficient)
  - where the blocking coefficient is between 0 and 1.
- A computation-intensive task has a blocking coefficient of 0, whereas an IO-intensive task has a value close to 1
- To determine the number of threads, we need to know two things:
  - The number of available cores (`Runtime.getRuntime().availableProcessors();`)
  - The blocking coefficient of tasks

Callable , Future and FutureTask

- Another important advantage of the *Executor framework* is the Callable interface.
- It's similar to the Runnable interface, but offers two improvements, which are as follows:
  - The main method of this interface, named call(), may return a result.
  - When you send a Callable object to an executor, you get an object that implements the Future interface. You can use this object to control the status and the result of the Callable object.
- If you need the functionality of a Future where only Runnables are supported (as in Executor), you can use FutureTask as a bridge. FutureTask implements both Future and Runnable so that you can submit the task as a Runnable and use the task itself as a Future in the caller.

CompletionService

- Beyond the common pattern of a pool of workers and an input queue, it is common for each task to produce a result that must be accumulated for further processing.
- The CompletionService interface allows a user to submit Callable and Runnable tasks but also to take or poll for results from the results queue:
  - Future<V> take() – take if available
  - Future<V> poll() – block until available
  - Future<V> poll(long timeout, TimeUnit unit) – block until timeout ends
- The ExecutorCompletionService is the standard implementation of CompletionService

Executor Example

- Simple Example One

```
Executor executor = Executors.newFixedThreadPool( 3 ) ; // 3 threads

List<Runnable> runnables = ... ;
for( Runnable task : runnables )
    executor.execute( task );
```

- Simple Example Two

```
class MyCallable implements Callable<Integer> {
    public Integer call() { return 2+2; }
}
```

```
// or anonymously
Callable<Integer> callable = new Callable<Integer>()
    public Integer call() { return 2+2; }
};
```

```
Future<Integer> result = executorService.submit( callable );
int val = result.get();  // blocks until ready
```

Executor Example

- Collective Tasks

```
List<Callable<Integer>> taskList = ...;
ExecutorService execService = Executors.newFixedThreadPool(3);
List<Future<Integer>> resultList = execService.invokeAll( taskList );
```

- invokeAny() method returns just the first successfully completed task's result (cancelling all the remaining unexecuted tasks):

```
int result = execService.invokeAny( taskList );
```

Executor Example

- ScheduledTasks

```
ScheduledExecutorService exec = Executors.newScheduledThreadPool(3);

exec.schedule( runnable, 60, TimeUnit.SECONDS ); // run one minute in the
                                                 // future

// run at specified date and time
Calendar futureDate = ...; // convertfrom calendar
Date date = futureDate.getTime(); // to Date
long delay  = date.getTime() - System.currentTimeMillis(); // to relative
                                                           // millis
exec.schedule( runnable, delay, TimeUnit.MILLISECONDS ); // run at specified
                                                         // date
```

Executor Example

- Completion Service

```
Executor executor = Executors.newFixedThreadPool(3);
CompletionService<Integer> completionService =
    new ExecutorCompletionService<Integer>( executor );

completionService.submit( callable );
completionService.submit( runnable, resultValue );

// poll for result
Future<Integer> result = completionService.poll();
if ( result != null )
    // use value...

// block, waiting for result
Future<Integer> result = completionService.take();
```

Atomic Variables

- One shortcoming of volatile is that while it provides visibility guarantees, you cannot both check and update a volatile field in a single atomic call.
- The java.util.concurrent.atomic package contains a set of classes that support atomic compound actions on a single value in a lock-free manner similar to volatile.
- Atomic classes are provided for Booleans, integers, longs, and object references as well as arrays of integers, longs, and object references.

```
import java.util.concurrent.atomic.AtomicInteger;

public class Counter {
    private AtomicInteger value = new AtomicInteger();
    public int next() {
        return value.incrementAndGet();
    }
}
```

Concurrent Collection

- The synchronized collection classes of Java lock the whole data set for any operations, increasing the scope and the duration of exclusive locks. Furthermore, those classes do not have any locking to guard compound actions, requiring client-side locking.
- Since JDK 1.5, Java provides a rich set of collection classes that are designed to help increase the scalability of multi-threaded applications by reducing the scope of exclusive locks while working on data sets. They also provide locking to guard compound actions such as iteration, navigation, and conditional operations

Concurrent Collection

- Several new Collection classes are added in Java 5 and Java 6 specially concurrent alternatives of standard synchronized ArrayList, Hashtable and synchronized HashMap collection classes.
  - ConcurrentHashMap
  - CopyOnWriteArrayList and CopyOnWriteArraySet
  - BlockingQueue
  - Deque and BlockingDeque
  - ConcurrentSkipListMap and ConcurrentSkipListSet

Old versus new Threading API

- The old threading API has several deficiencies
  - We'd use and throw away the instances of the thread class since they don't allow restart.
  - We have to write a quite a bit of code to manage threads.
  - Methods like wait() and notify() require synchronization and are quite hard to get right when used to communicate between threads
  - The join() method leads us to be concerned about the death of a thread rather than a task being accomplished.
  - the synchronized keyword lacks granularity. It doesn't give us a way to time out if we do not gain the lock. It also doesn't allow concurrent multiple readers
  - it is very difficult to unit test for thread safety if we use synchronized

Old versus new Threading API

- The newer generation of concurrency APIs in the java.util.concurrent package, spearheaded by Doug Lea, among others, has nicely replaced the old threading API.
  - Wherever we use the Thread class and its methods, we can now rely upon the ExecutorService class and related classes.
  - If we need better control over acquiring locks, we can rely upon the Lock interface and its methods.
  - Wherever we use wait/notify, we can now use synchronizers such as CyclicBarrier and CountdownLatch.

Java Concurrency Framework Components

- **Task Scheduling Framework**: The Executor is a framework for handling the invocation, scheduling and execution of tasks
- **Great Synchronization**: The new synchronization primitives offer finer granularity—thus more control—than the original primitives.
- **Concurrent Collection**: Concurrent implementations of commonly used classes like map, list and queue (no more reinventing the wheel).
- **Atomic Variables**: classes for atomic manipulation of single variables, provides higher performance than simply using synchronization primitives
- **Locks**: High performance implementation of locks with the same semantics as the *synchronized* keyword, with additional functionality like timeouts.
- **Timers**: Provides access to a nanosecond timer for making fine grained timing measurements, useful for methods that accept timeouts.

Advantages of using the framework

- **Reusability and effort reduction**: Many commonly used concurrent classes already implemented
- **Superior performance**: Inbuilt implementation highly optimized and peer reviewed by experts
- **Higher reliability, less bugs**: Working from already developed building blocks lead to more reliable programs
- **Improved maintainability and scalability**: Reusable components leads to programs that are easier to maintain and scale much better
- **Increased productivity**: Developers no longer have to reinvent the wheel each time, programs easier to debug, more likely to understand standard library implementations

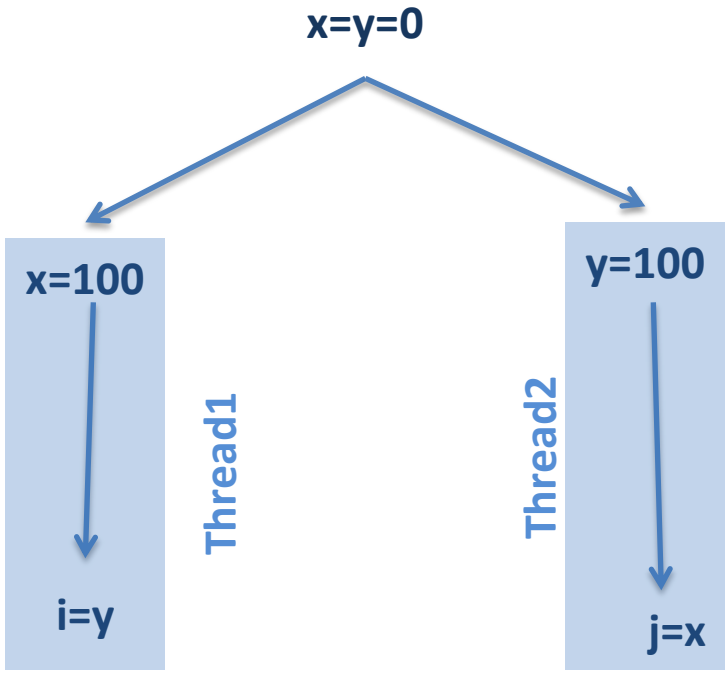
Shutdown Hooks

- Every Java Program can attach a shutdown hook to JVM, i.e. piece of instructions that JVM should execute before it terminates.
- An application may go down because of several reasons
  - Because all of its threads have completed execution
  - Because of call to System.exit()
  - Because user hit CNTRL-C
  - System level shutdown or User Log-Off

Shutdown Hooks

```
public class ShutdownHookDemo {

    public static void main(String... args) throws Exception {

        Runtime runtime = Runtime.getRuntime();

        //adding the first shutdown hook
        runtime.addShutdownHook(new ShutDownHook());

        //A quicker way to add a shutdown hook using an inner class
        runtime.addShutdownHook(new Thread() {
            public void run() {
                System.out.println("Second shutdown hook called.");
        }});
        System.out.println("Application Terminating ...");
    }

    private static class ShutDownHook extends Thread {
        public void run() {
            System.out.println("First shutdown hook called.");
        }
    }
}
```

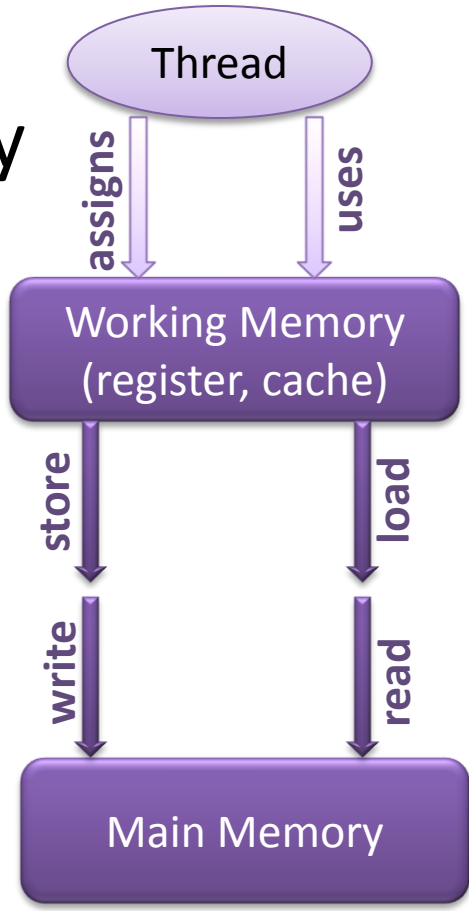
Quick Question



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

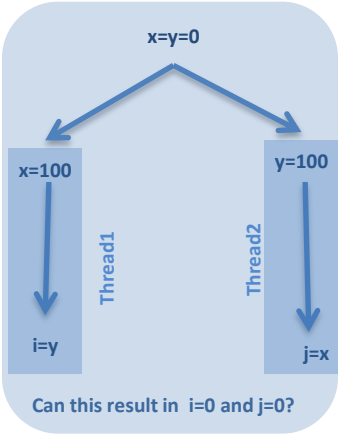
Working Memory Versus Main Memory

- Every thread has a *working memory* in which it keeps its local working copy of variables that it must *use* or *assign*. As the thread executes a program, it operates on these working copies.
- The *main memory* contains the master copy of every variable.



How can that happen?

- Just-in-Time (JIT) Compiler can reorder statement or keep values in registers.
- Processor can reorder them.
- On multiprocessor, values not synchronized in global memory.



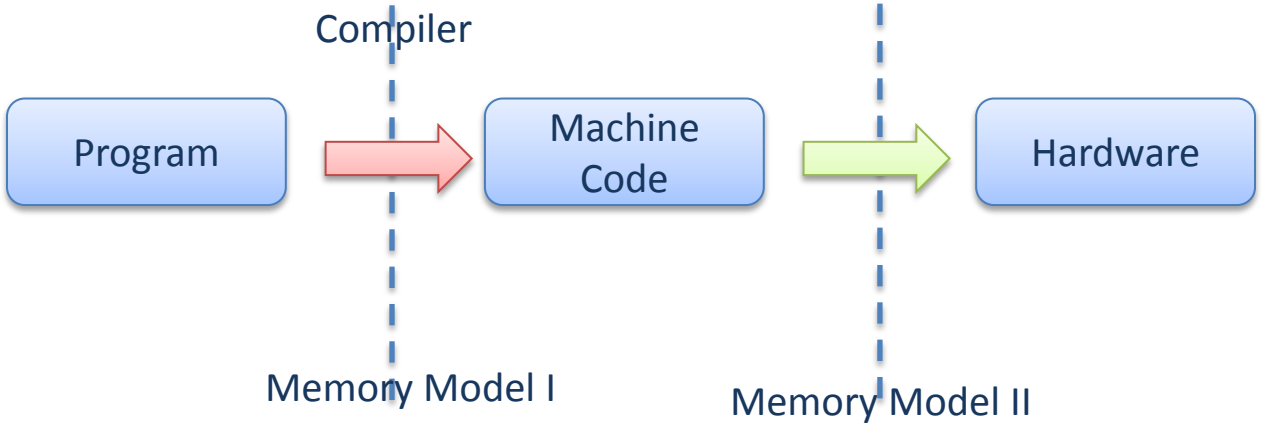
Use **synchronization** to enforce *visibility* and *ordering* as well as *mutual exclusion.*

Memory Model

- "A formal specification of how the memory system will appear to the programmer, eliminating the gap between the behavior expected by the programmer and the actual behavior supported by a system."
- Memory model specifies:
  - How threads interact through memory
  - when one thread's actions are guaranteed to be *visible* to another
    - What value a read can return
    - When does a value update become visible to other threads
  - What assumptions are allowed to make about memory when writing a program or applying some program optimization.
  - What hardware/compiler can do and cannot do.

Why do we care?

- Memory model affects:
  - Programmability
  - Performance
  - Portability



Java Memory Model (JMM)

- A contract between Java programmers, compiler writers and JVM implementers
  - What transformations can compiler do?
  - What transformations can JVM do?
  - What optimization can hardware perform?
  - What can Java programmers can expect?
- Defines the behaviors of multi threaded Java programs

Why the new JMM

- Prior to Java 5, the Java Memory Model (JMM) was ill defined. It was possible to get all kinds of strange results when shared memory was accessed by multiple threads, such as:
  - A thread not seeing values written by other threads: a visibility problem
  - A thread observing 'impossible' behavior of other threads, caused by instructions not being executed in the order expected: an instruction reordering problem.
  - Treat final fields same as any other fields in synchronization.
  - Allow volatile writes to be reordered with ordinary reads and writes

The Goal of new JMM

- Provide an intuitive programming model for Java programmers
  - Sequential consistency guarantee for correctly synchronized or data-race-free programs
- Preserve Java's safety and security properties
- Allow common compiler and hardware optimizations

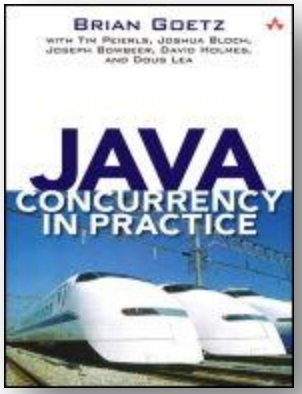
Best practices which must be followed

- Always run your java code against static analysis tools like PMD , FindBugs and Fortify to look for deeper issues. They are very helpful in determining ugly situations which may arise in future.
- Always cross check and better plan a code review with peer/senior guys to detect and possible deadlock or livelock in code during execution. Adding a health monitor in your application to check the status of running tasks is an excellent choice in most of the scenarios.
- In multi-threaded programs, make a habit of catching errors too, not just exceptions. Sometimes unexpected things happen and Java throws an error at you, apart from an exception.
- Please note that the whole point of *executors* is to abstract away the specifics of execution, so ordering is not guaranteed unless explicitly stated.

Tools

- Jconsole
- Jprofiler
- JProbe
- YourKit
- Jvisualvm
- AppDynamics

References



ProKarma

Reach Me @

mnadeem@prokarmasoftech.com

Mobile : +91 996-993-9709

VOIP : +1 402-536-4164

*ProKarma Recognized by **Inc. Magazine** as the "Fastest-Growing IT Services Company in America"*