

Advanced Java



Rajeev Gupta

Java Trainer & Consultant (MTech CS)

rgupta.mtech@gmail.com



Rajeev Gupta

FreeLancer Corporate Java JEE/ Spring Trainer

freelance • Institution of Electronics and Telecommunication Engineers IETE

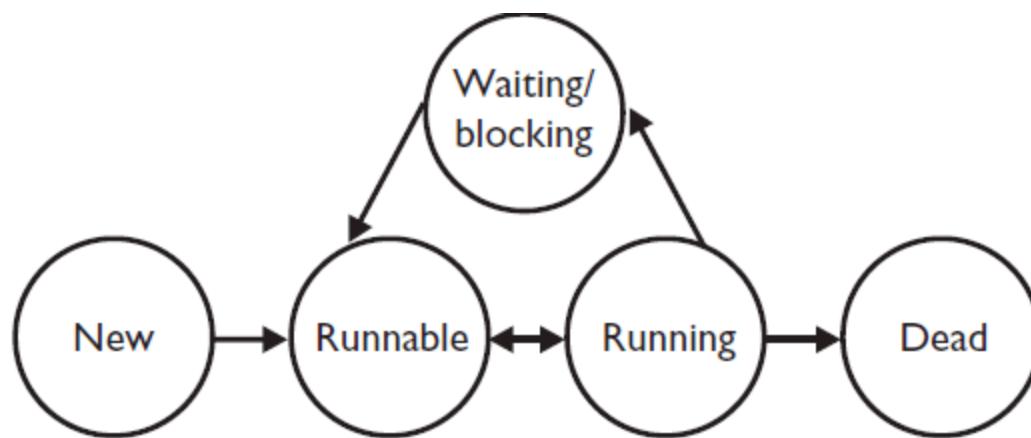
New Delhi Area, India • 500+ 

-
1. Expert trainer for Java 8, GOF Design patterns, OOAD, JEE 7, Spring 5, Hibernate 5, Spring boot, microservice, netflix oss, Spring cloud, angularjs, Spring MVC, Spring Data, Spring Security, EJB 3, JPA 2, JMS 2, Struts 1/2, Web service
 2. Helping technology organizations by training their fresh and senior engineers in key technologies and processes.
 3. Taught graduate and post graduate academic courses to students of professional degrees.

I am open for advance java training /consultancy/ content development/ guest lectures/online training for corporate / institutions on freelance basis

- **Java Concurrency**

- Create worker threads using Runnable, Callable and use an ExecutorService to concurrently execute tasks
- Identify potential threading problems among deadlock, starvation, livelock, and race conditions
- Use synchronized keyword and java.util.concurrent.atomic package to control the order of thread execution
- Use java.util.concurrent collections and classes including CyclicBarrier and CopyOnWriteArrayListUse parallel Fork/Join Framework
- Use parallel Streams including reduction, decomposition, merging processes, pipelines and performance.



How you designed it

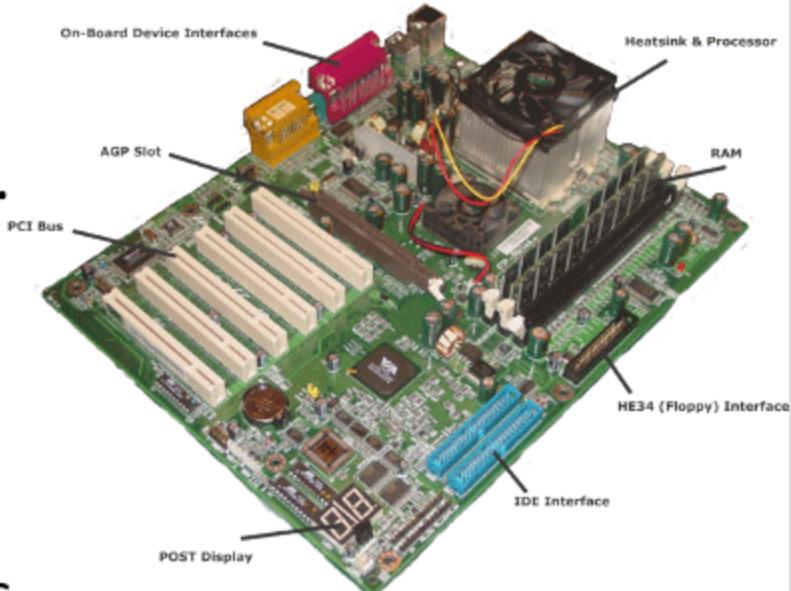


What happens in reality

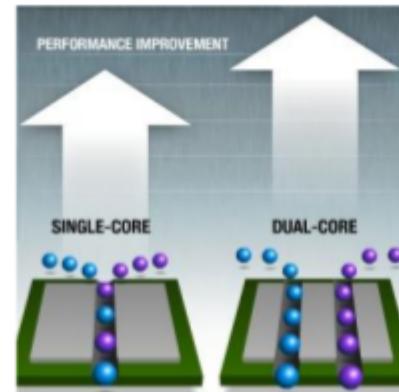
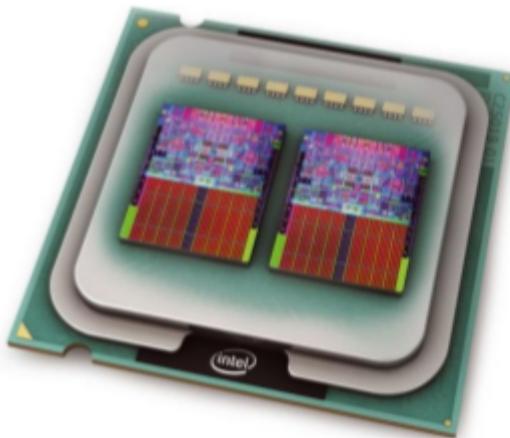


How Concurrency Achieved?

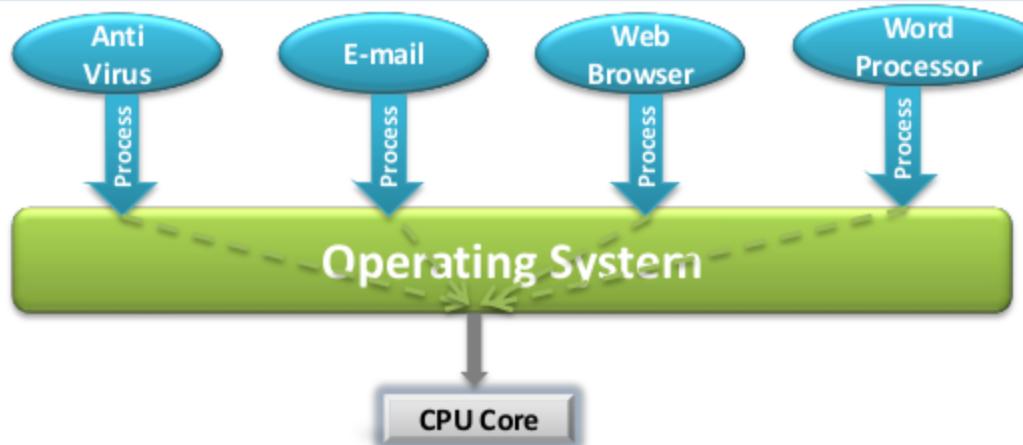
- Single Processor:
 - Has only one actual processor.
 - Concurrent tasks are often multiplexed or multi tasked.
- Multi Processor.
 - Has more than one processors.
 - Concurrent tasks are executed on different processors.



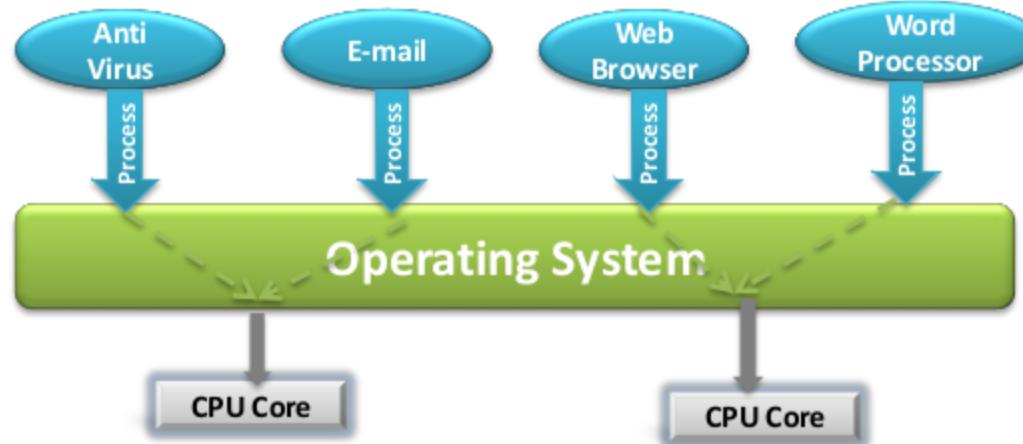
- Both types of systems can have more than one core per processor (Multicore).
- Multicore processors behave the same way as if you had multiple processors



Cores

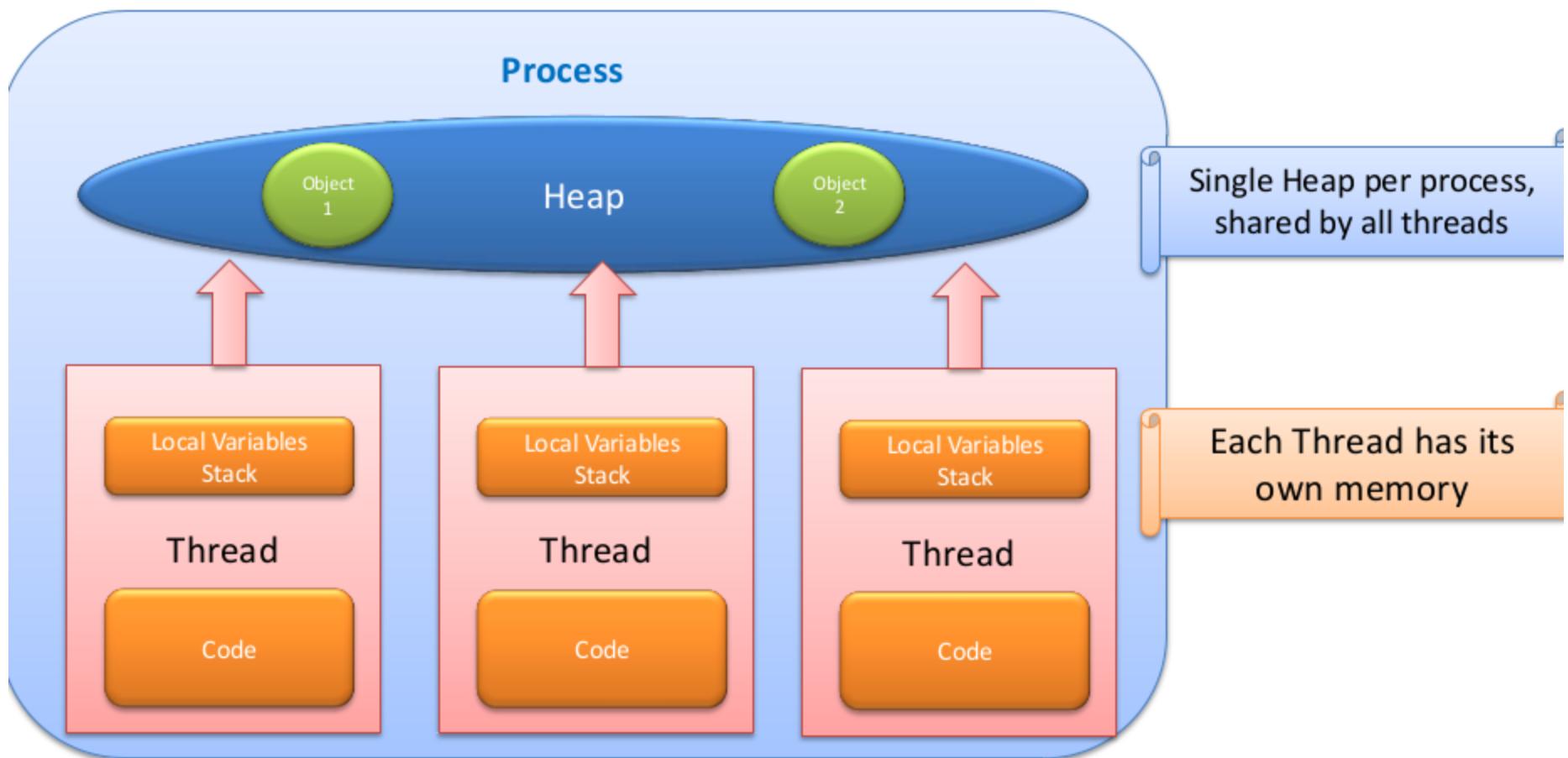


Single-core systems schedule tasks on 1 CPU to multitask



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

Process and Thread?



More Background Threads

Nothing little app

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

```
C:\Users\nadeen\Desktop>jps -l  
3296 sun.tools.jps.Jps  
4756 com.prokarma.app.gtp.thread.ThreadDumpSample  
C:\Users\nadeen\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=0x000000000064b5800 nid=0x1238 runnable [0x0000000000000000]  
    java.lang.Thread.State: RUNNABLE  
"C2 CompilerThread1" daemon prio=10 tid=0x000000000064b2000 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=0xdbe waiting on condition [0x0000000000000000]  
    java.lang.Thread.State: RUNNABLE  
"Signal Dispatcher" daemon prio=10 tid=0x000000000064a0800 nid=0x81c runnable [0x0000000000000000]  
    java.lang.Thread.State: RUNNABLE  
"Finalizer" daemon prio=8 tid=0x0000000000512800 nid=0x804 in Object.wait() [0x0000000000645f000]  
    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$FinalizerThread.run(Finalizer.java:171)  
"Reference Handler" daemon prio=10 tid=0x0000000000509800 nid=0xf30 in Object.wait() [0x000000000636f000]  
    java.lang.Thread.State: WAITING (on object monitor)  
    at java.lang.Object.wait(Native Method)  
    - waiting on <0x00000007d58011d8> (a java.lang.ref.Reference$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
```

Basic Operations on Thread

Out put

```
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.");  
    }  
}
```

```
Thread Name : main  
Thread Group : java.lang.ThreadGroup[name=main,maxpri=1]  
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.
```



Let's clarify some terms

Concurrency ≠ Parallelism

Multithreading ≠ Parallelism



Few definitions

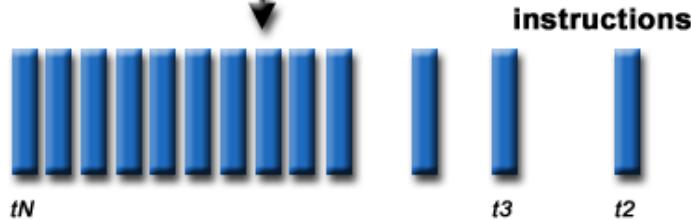
Concurrency: A condition that exists when at least two threads are *making progress*

Parallelism: A condition that arises when at least two threads are **executing simultaneously**

Multithreading: Allows access to two or more threads. Execution occurs in more than one thread of control, using parallel or concurrent processing

*Source: Sun's Multithreaded Programming Guide
<http://dlc.sun.com/pdf/816-5137/816-5137.pdf>*

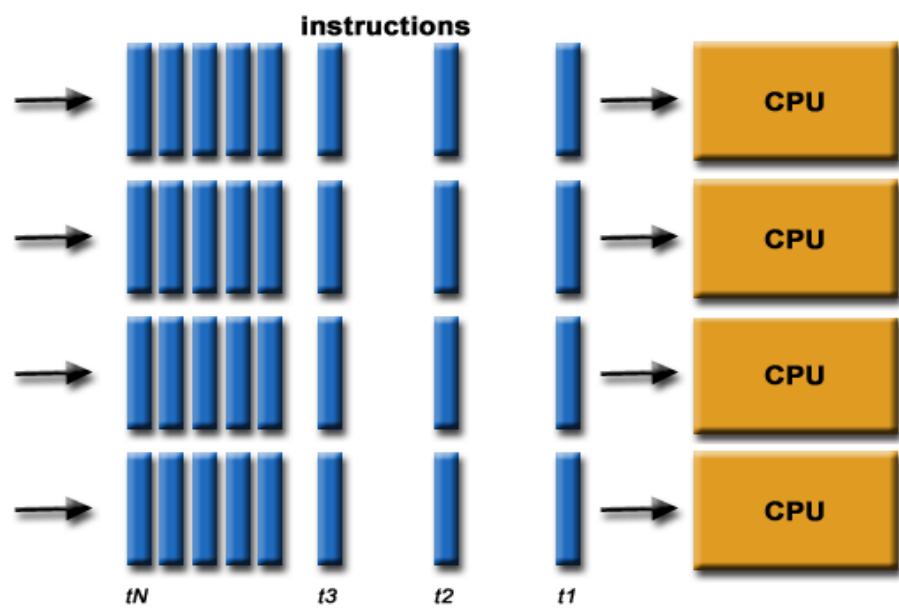
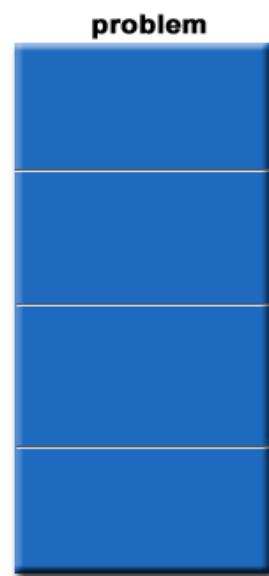




Concurrency

vs

Parallelism



tN

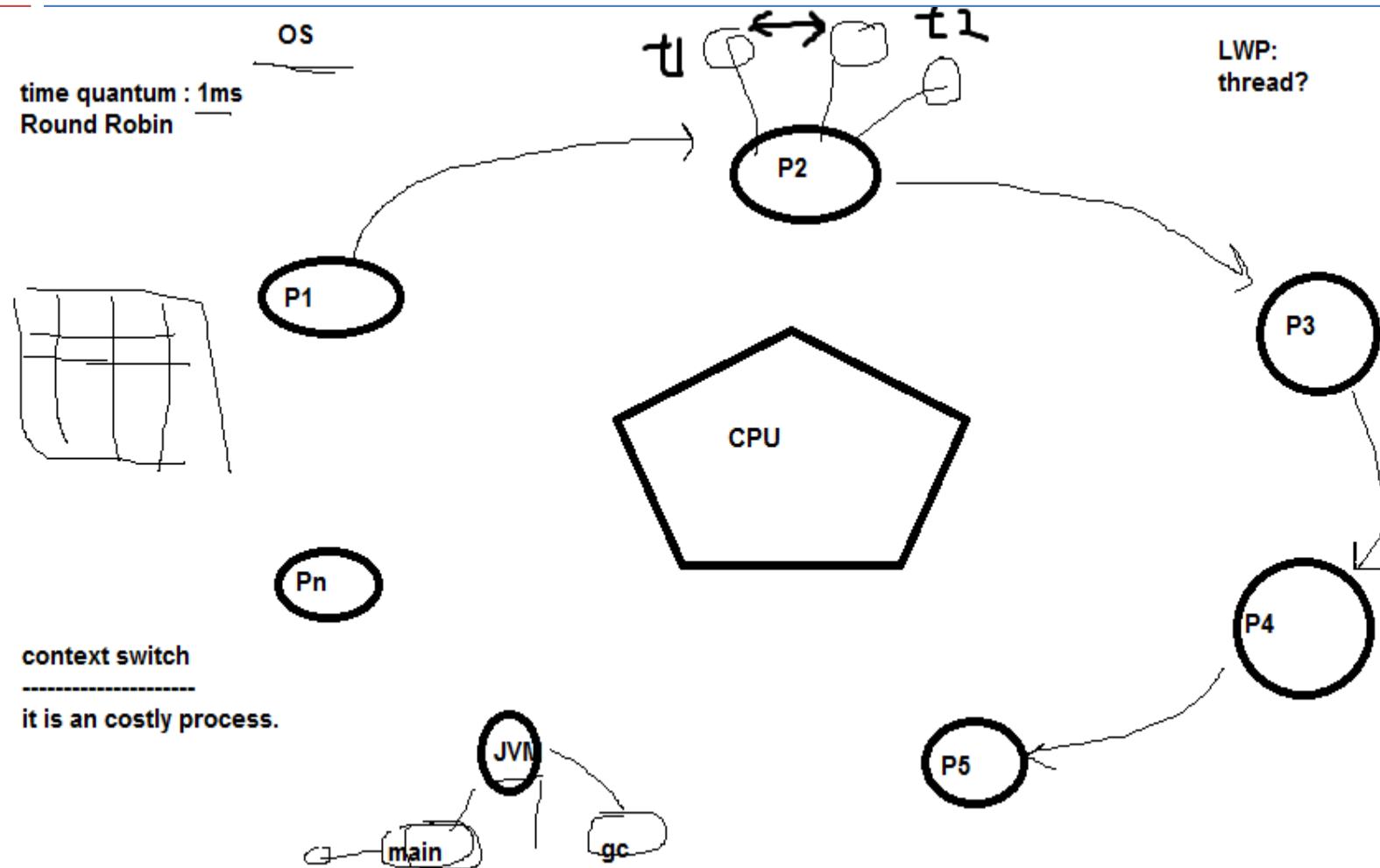
$t3$

$t2$

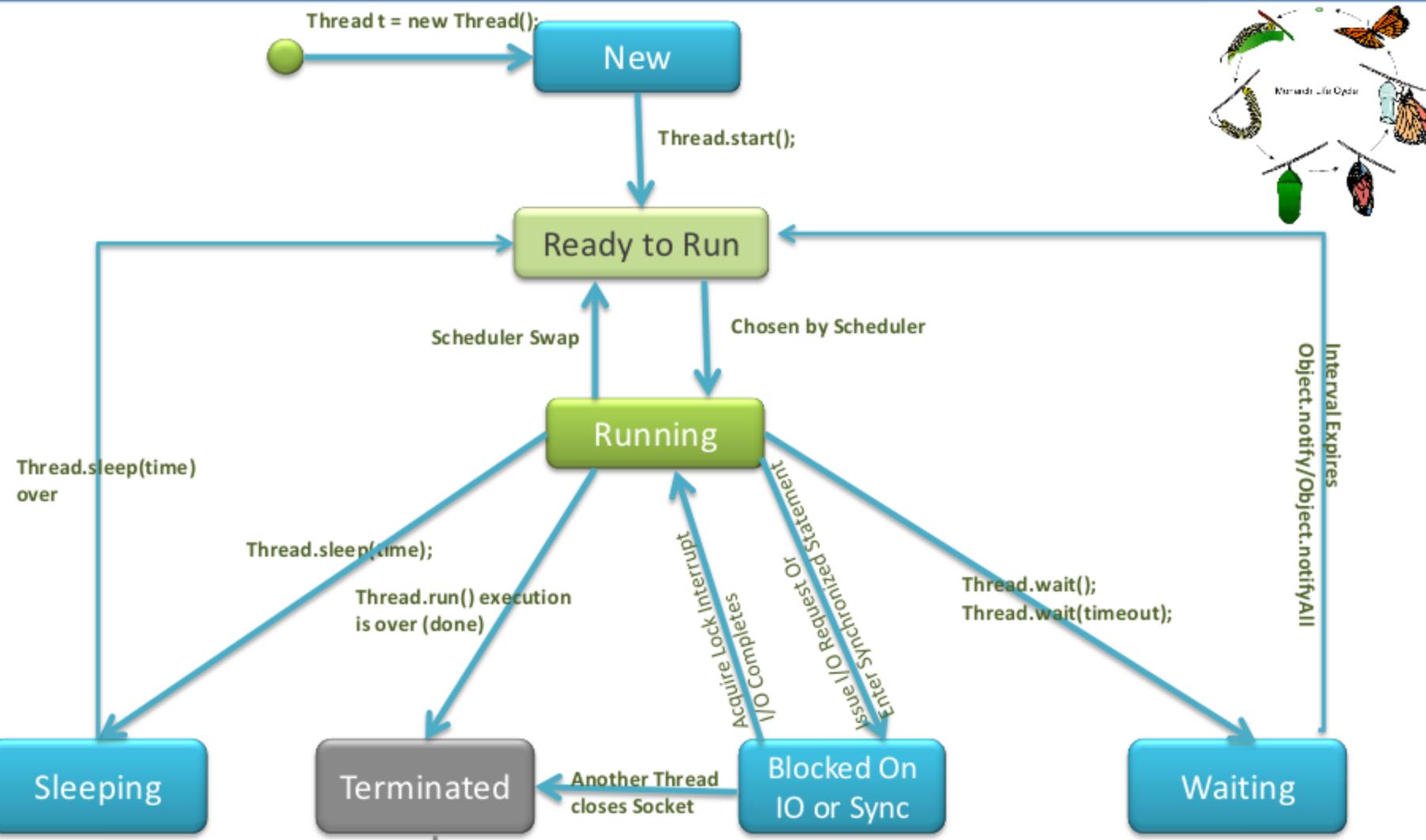
$t1$



What is threads? LWP

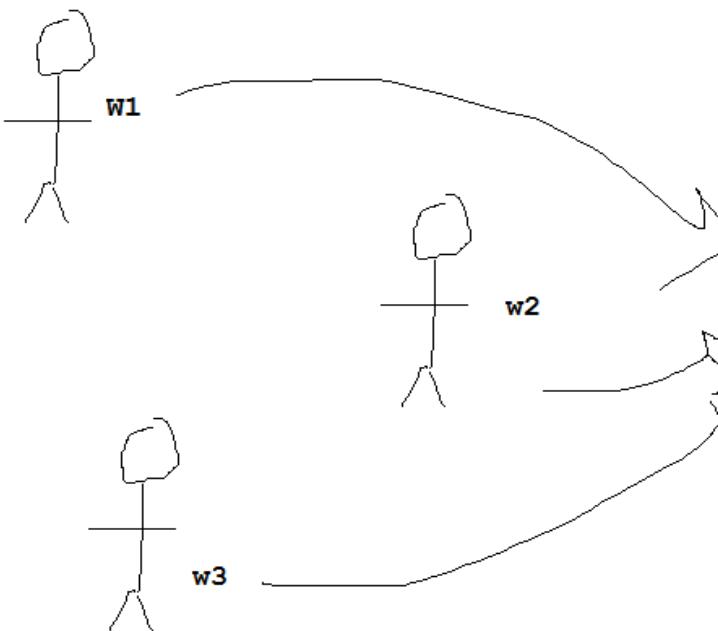


Thread Lifecycle



Creating threads Java?

- Implements Runnable interface
- Extending Thread class.....
- Job and Worker analogy...



Thread

consider object of threads as
worker

Implementation of Runnable as
Job



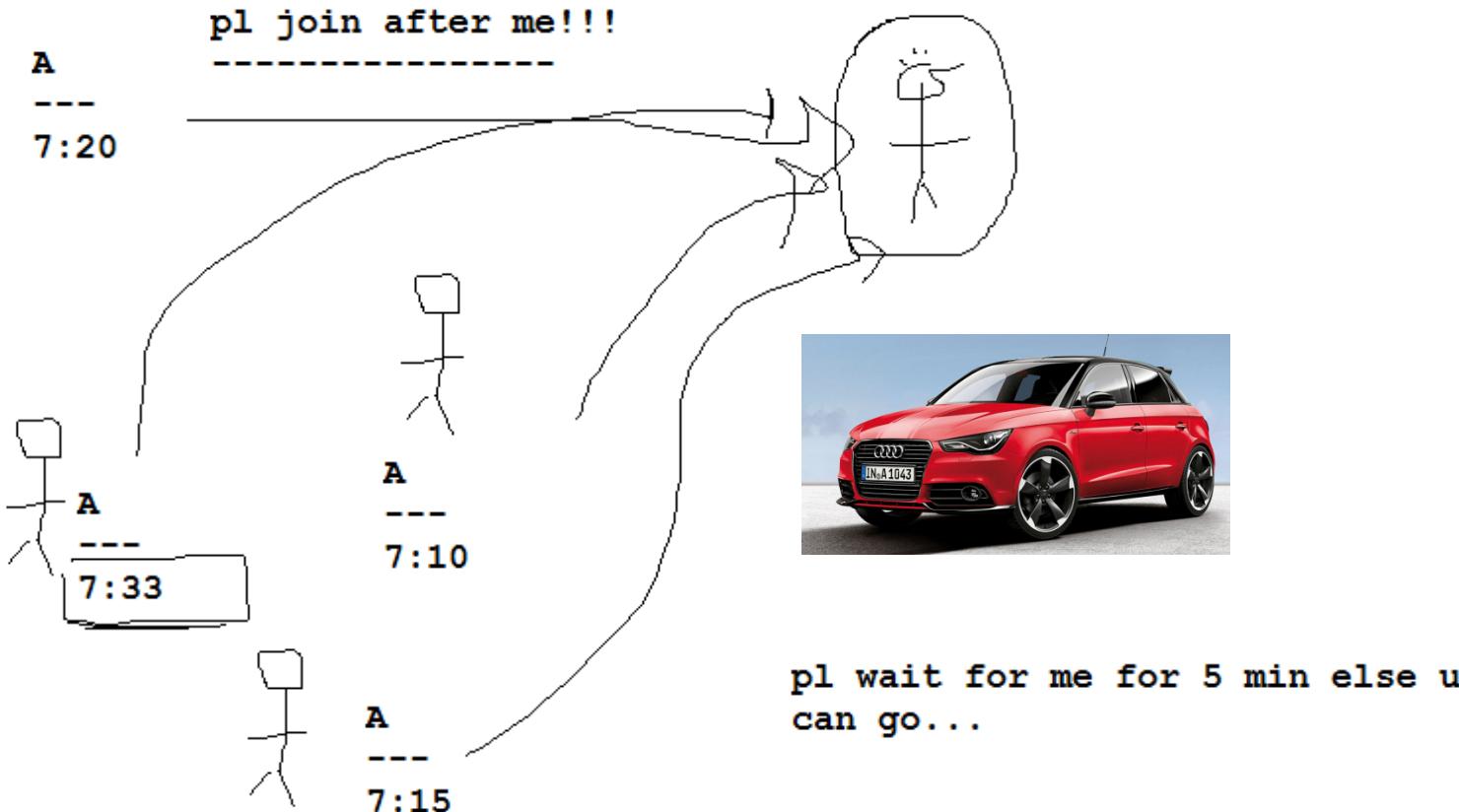
simulation

sleep()

```
class Job implements Runnable{  
  
    public void run() {  
        // TODO Auto-generated method stub  
    }  
  
}
```

```
class MyThread extends Thread{  
    public void run() {  
        // TODO Auto-generated method stub  
    }  
}
```

Understanding join() method



Checking thread priorities

```
class Clicker implements Runnable
{
    int click=0;
    Thread t;
    private volatile boolean running=true;
    public Clicker(int p)
    {
        t=new Thread(this);
        t.setPriority(p);
    }
    public void run()
    {
        while(running)
            click++;
    }
    public void stop()
    {
        running=false;
    }
    public void start()
    {
        t.start();
    }
}
```

```
.....
Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
Clicker hi=new Clicker(Thread.NORM_PRIORITY+2);
Clicker lo=new Clicker(Thread.NORM_PRIORITY-2);
lo.start();
hi.start();
try
{
    Thread.sleep(10000);
}
catch(InterruptedException ex){}
lo.stop();
hi.stop();
//wait for child to terminate
try
{
    hi.t.join();
    lo.t.join();
}
catch(InterruptedException ex)
{
}

System.out.println("Low priority thread:"+lo.click);
System.out.println("High priority thread:"+hi.click);
.....
....
```

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 <i>spurious wakeups</i> from wait (i.e. the thread which is waiting resumes for no apparent reason). We Should always wait whilst spinning on some condition , ex : <pre>synchronized { while (!condition) monitor.wait(); }</pre>	This is deterministic.
Releases the lock on the object that wait() is called on	Thread does <i>not</i> release the locks it holds

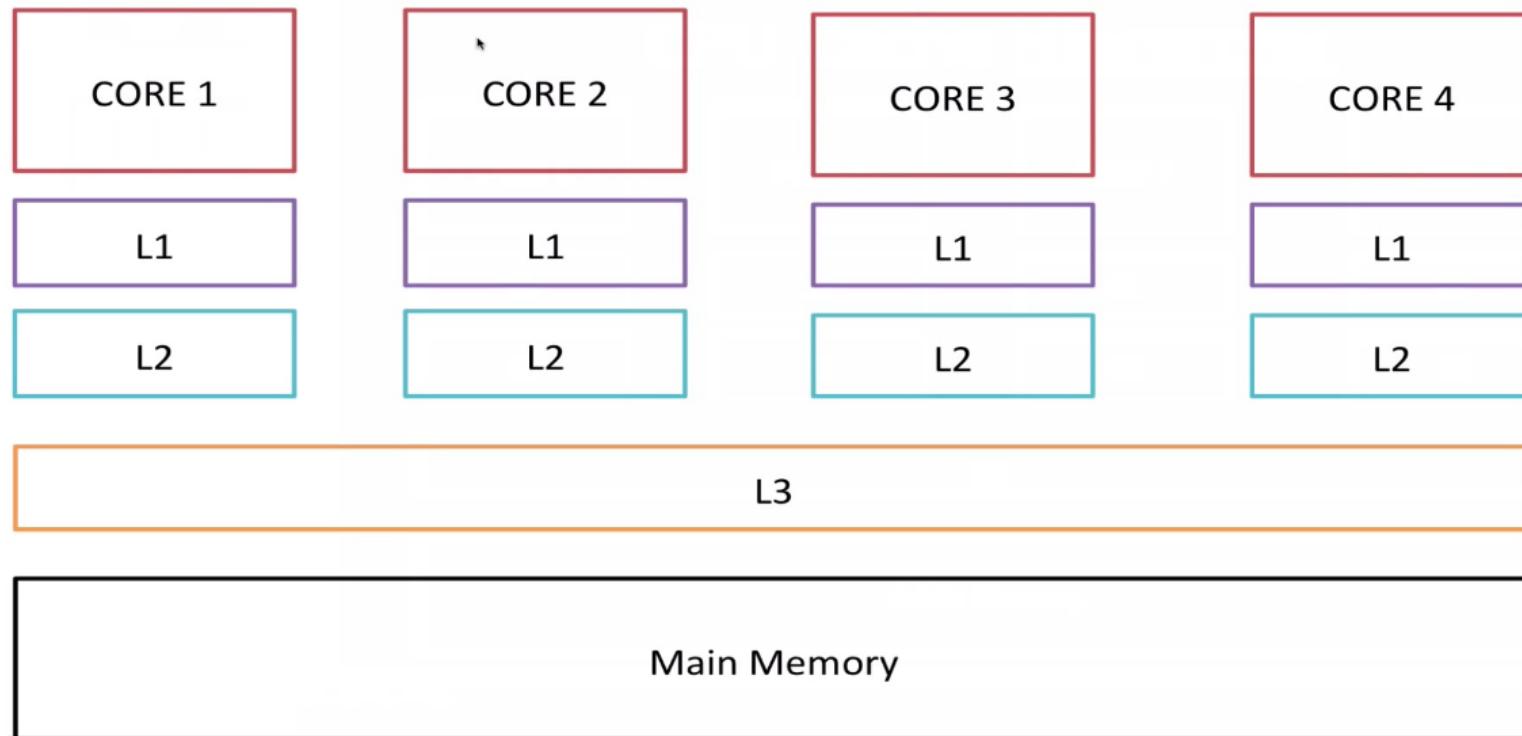
Volatile

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

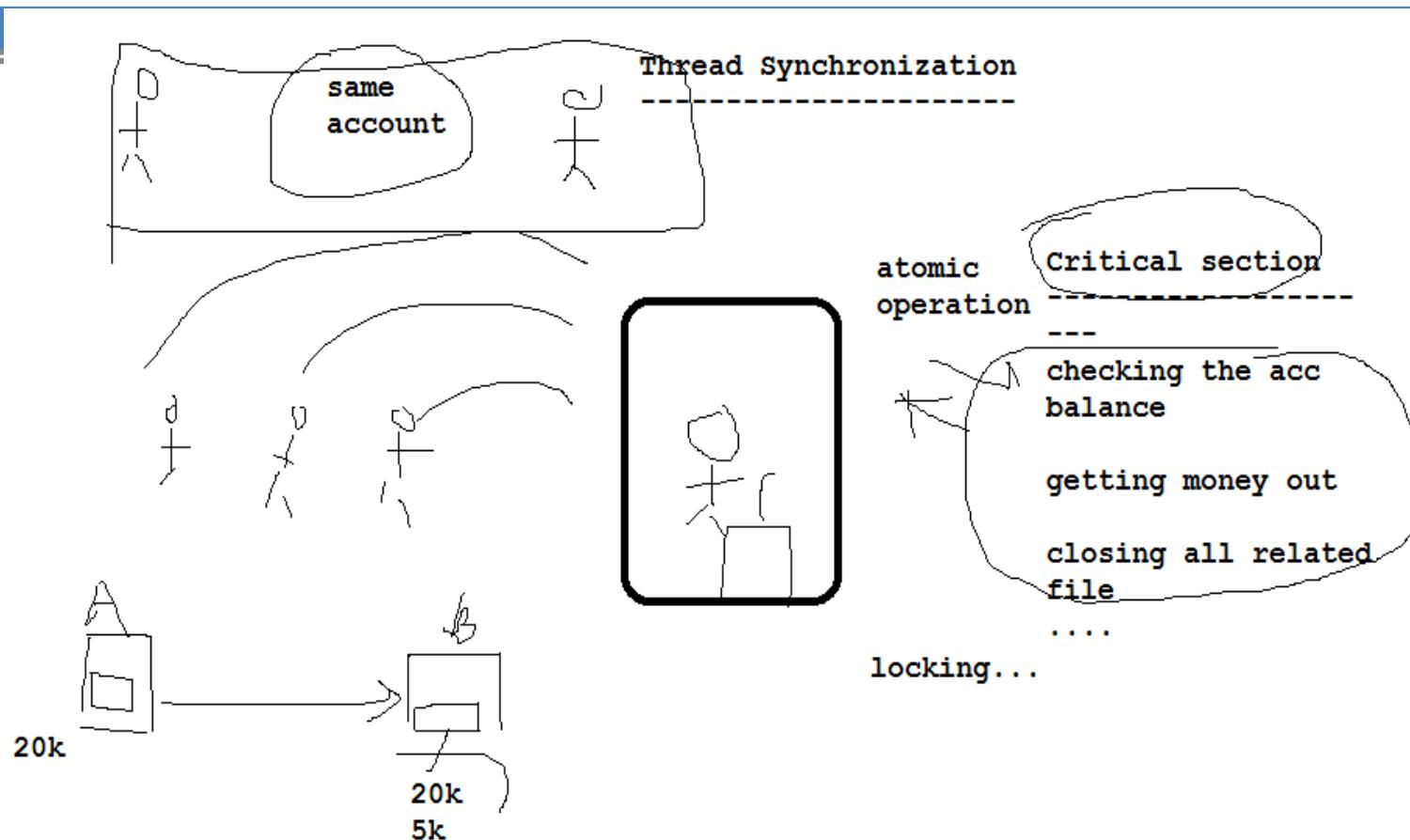
```
class Test extends Thread {  
  
    volatile boolean keepRunning = true;  
  
    public void run() {  
        while (keepRunning) {  
            }  
  
        System.out.println("Thread terminated.");  
    }  
  
    public static void main(String[] args) throws InterruptedException {  
        Test t = new Test();  
        t.start();  
        Thread.sleep(1000);  
        t.keepRunning = false;  
        System.out.println("keepRunning set to false.");  
    }  
}
```

volatile

CPU Cache Hierarchy



Understanding thread synchronization



Synchronization

- Synchronization
 - Mechanism to controls the order in which threads execute
 - Competition vs. cooperative synchronization
- Mutual exclusion of threads
 - Each synchronized method or statement is guarded by an object.
 - When entering a synchronized method or statement, the object will be locked until the method is finished.
 - When the object is locked by another thread, the current thread must wait.

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;  
        }  
    }  
}
```

Using thread synchronization

```
class CallMe
{
    synchronized void call(String msg)
    {
        System.out.print("[ "+msg);
        try
        {
            Thread.sleep(500);
        }
        catch(InterruptedException ex) {}
        System.out.println(" ]");
    }
}
```

```
class Caller implements Runnable
{
    String msg;
    CallMe target;
    Thread t;
    public Caller(CallMe targ,String s)
    {
        target=targ;
        msg=s;
        t=new Thread(this);
        t.start();
    }
    public void run()
    {
        target.call(msg);
    }
}
```

```
Caller ob1=new Caller(target, "Hello");
Caller ob2=new Caller(target,"Synchronized");
Caller ob3=new Caller(target,"Java");
```

Inter thread communication

- Java have elegant Interprocess communication using `wait()` `notify()` and `notifyAll()` methods
- All these method defined final in the Object class
- Can be only called from a synchronized context

wait() and notify(), notifyAll()

▫ wait()

- Tells the calling thread to give up the monitor and go to the sleep until some other thread enter the same monitor and call notify()

▫ notify()

- Wakes up the first thread that called wait() on same object

▫ notifyAll()

- Wakes up all the thread that called wait() on same object, highest priority thread is going to run first

Incorrect implementation of

```
class Q
{
    int n;
    synchronized int get()
    {
        System.out.println("got:"+n);
        return n;
    }
    synchronized void put(int n)
    {
        this.n=n;
        System.out.println("Put:"+n);
    }
}
```

```
public class PandC {
public static void main(String[] args) {
    Q q=new Q();
    new Producer(q);
    new Consumer(q);
    System.out.println("ctrl C for exit");
}
}
```

S

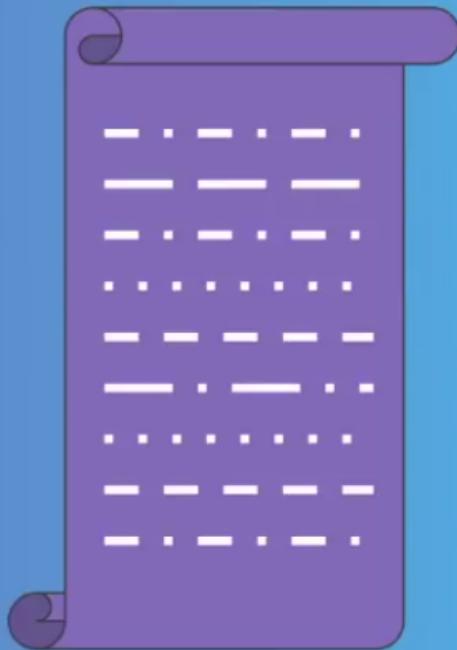
```
class Producer implements Runnable
{
    Q q;
    public Producer(Q q) {
        this.q=q;
        new Thread(this,"Producer").start();
    }
    public void run()
    {
        int i=0;
        while(true)
            q.put(i++);
    }
}
```

```
class Consumer implements Runnable
{
    Q q;
    Consumer(Q q)
    {
        this.q=q;
        new Thread(this,"consumer").start();
    }
    public void run()
    {
        while(true)
            q.get();
    }
}
```

Correct implementation of producer consumer ...

```
class Q
{   int n;
    boolean valueSet=false;
    synchronized int get()
    {
        if(!valueSet)
            try
            {
                wait();
            }
        catch(InterruptedException ex){}
        System.out.println("got:"+n);
        valueSet=false;
        notify();
        return n;
    }
    synchronized void put(int n)
    {
        if(valueSet)
            try
            {
                wait();
            }
        catch(InterruptedException ex){}
        this.n=n;
        valueSet=true;
        System.out.println("Put:"+n);
        notify();
    }
}
```

Java Memory Model



What is Java Memory Model?

JMM is a specification which guarantees visibility of fields (aka happens before) amidst reordering of instructions.

Out of order execution

Performance driven changes
done by
Compiler, JVM or CPU

Instructions

```
a = 3;  
b = 2;  
a = a + 1;
```

→ Load a
→ Set to 3
→ Store a

→ Load b
→ Set to 2
→ Store b

→ Load a
→ Set to 4
→ Store a

Consider this code

Instructions

a = 3;

a = a + 1;

b = 2;

- Load a
- Set to 3
- Set to 4
- Store a

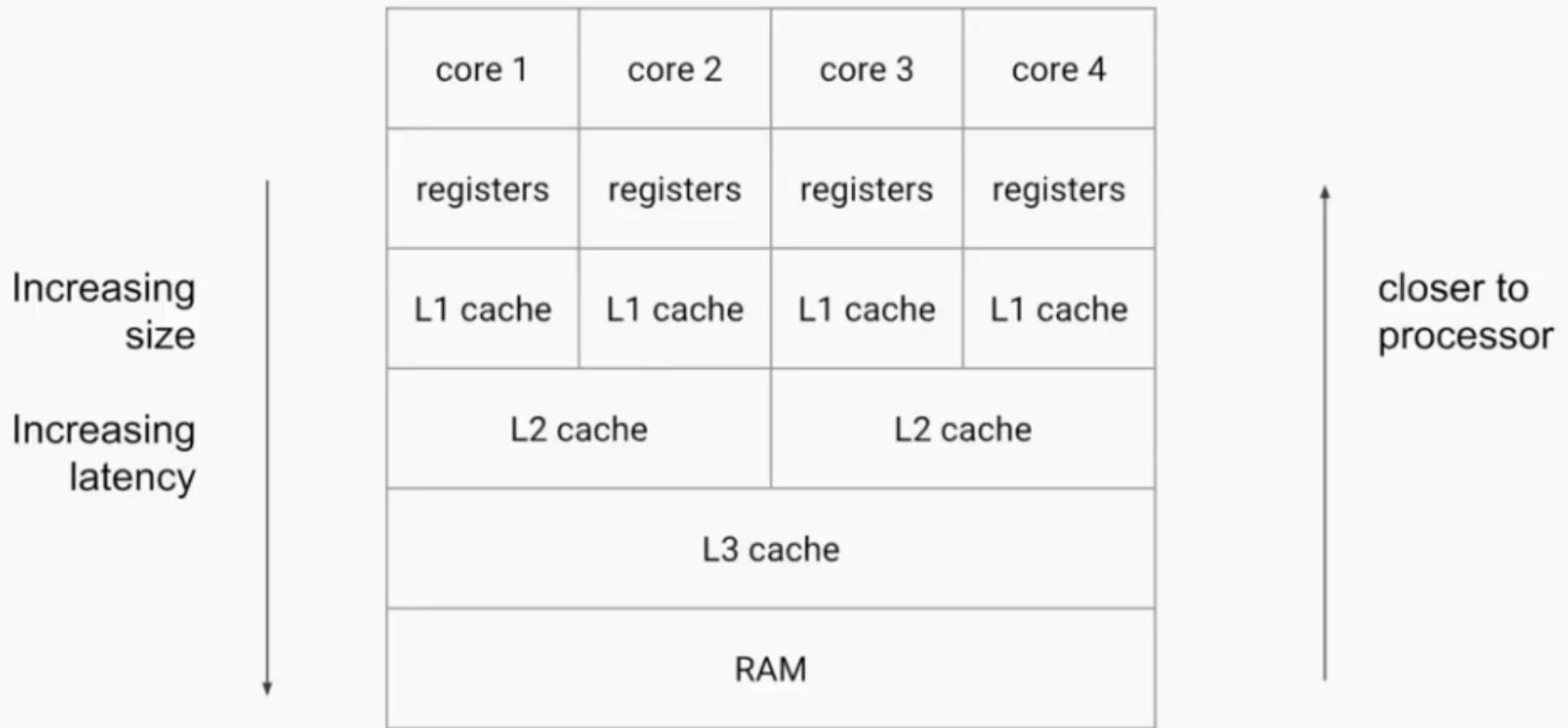
- Load b
- Set to 2
- Store b

Field Visibility

In presence of multiple threads

a.k.a

Concurrency



CPU caches (multi-core processor)

```
public class FieldVisibility {  
  
    int x = 0;  
  
    public void writerThread() {  
        x = 1;  
    }  
  
    public void readerThread() {  
        int r2 = x;  
    }  
}
```

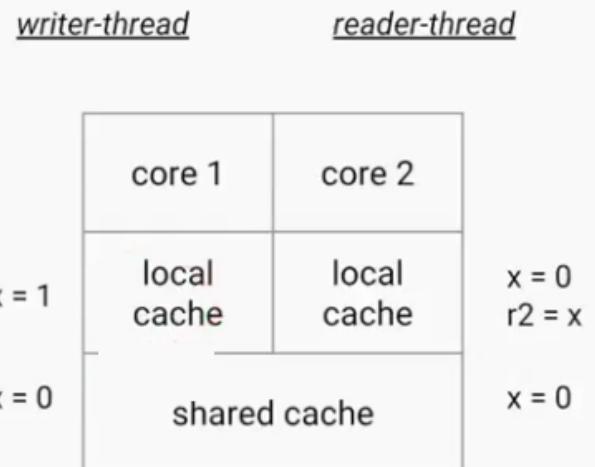
writer-thread *reader-thread*

core 1	core 2
local cache	local cache
shared cache	

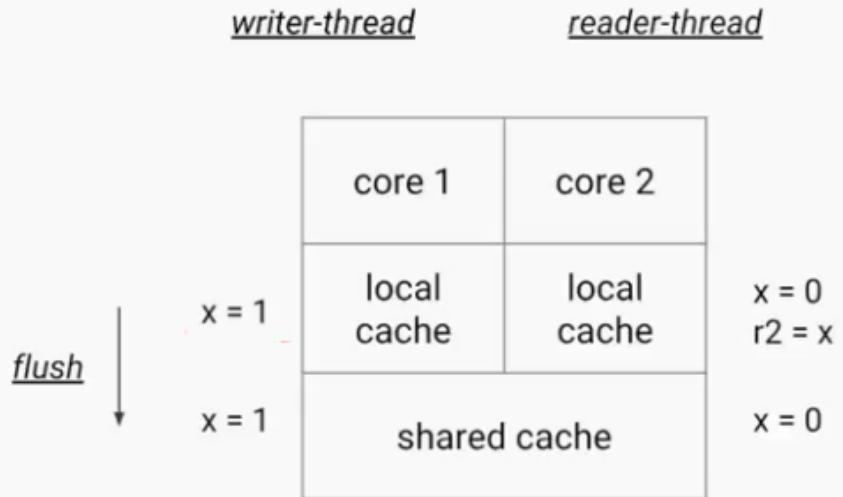
x = 0

x = 0

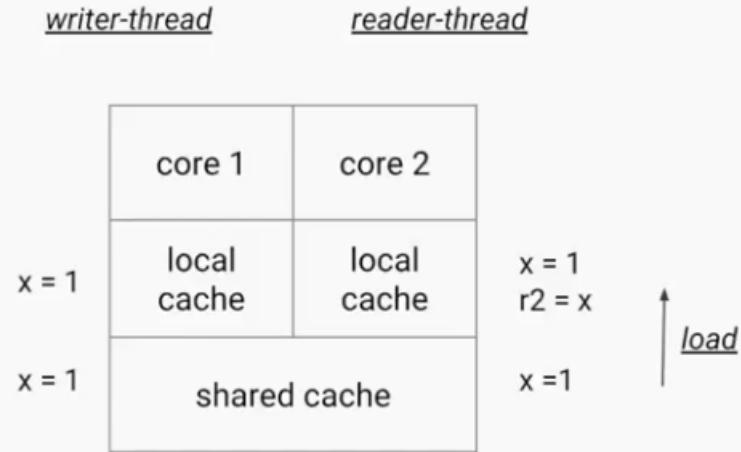
```
public class FieldVisibility {  
  
    int x = 0;  
  
    public void writerThread() {  
        x = 1;  
    }  
  
    public void readerThread() {  
        int r2 = x;  
    }  
}
```



```
public class VolatileVisibility {  
  
    volatile int x = 0;  
  
    public void writerThread() {  
        x = 1;  
    }  
  
    public void readerThread() {  
        int r2 = x;  
    }  
}
```



```
public class VolatileVisibility {  
    volatile int x = 0;  
    •  
    public void writerThread() {  
        x = 1;  
    }  
    public void readerThread() {  
        int r2 = x;  
    }  
}
```



```
public class VolatileFieldsVisibility {

    int a = 0, b = 0, c = 0;
    volatile int x = 0;

    public void writerThread() {

        a = 1;
        b = 1;
        c = 1;

        x = 1; // write of x
    }

    public void readerThread() {

        int r2 = x; // read of x

        int d1 = a;
        int d2 = b;
        int d3 = c;
    }
}
```

JMM rule:
Whatever happens before `x=1`
in writer thread
must be visible in reader thread
after `int r2=x;`

“Happens-before” relationship for volatile

Not only volatile

Also

1. Synchronized
2. Locks
3. Concurrent collections
4. Thread operations (join,start)

final fields (special behavior)

```
public class SynchronizedFieldsVisibility {

    int a = 0, b = 0, c = 0;
    volatile int x = 0;

    public void writerThread() {
        a = 1;
        b = 1;
        c = 1;

        synchronized (this) {
            x = 1;
        }
    }

    public void readerThread() {
        synchronized (this) {
            int r2 = x;
        }

        int d1 = a;
        int d2 = b;
        int d3 = c;
    }
}
```

Same behavior with synchronized blocks. V.Imp: Has to be synchronized on same object!!

```
public class SynchronizedFieldsVisibility {

    int a = 0, b = 0, c = 0;
    volatile int x = 0;

    public void writerThread() {

        synchronized (this) {
            a = 1;
            b = 1;
            c = 1;
            x = 1;
        }
    }

    public void readerThread() {

        synchronized (this) {
            int r2 = x;
            int d1 = a;
            int d2 = b;
            int d3 = c;
        }
    }
}
```

Better to be more clear though!

```
public class LockVisibility {

    int a = 0, b = 0, c = 0, x = 0;
    Lock lock = new ReentrantLock();

    public void writerThread() {

        lock.lock();
        a = 1;
        b = 1;
        c = 1;
        x = 1;
        lock.unlock();
    }

    public void readerThread() {

        lock.lock();
        int r2 = x;
        int d1 = a;
        int d2 = b;
        int d3 = c;
        lock.unlock();
    }
}
```

Same behavior with Locks.

```
public class VolatileVisibility {  
  
    boolean flag = true;  
  
    public void writerThread() {  
        flag = false;  
    }  
  
    public void readerThread() {  
        while (flag) {  
            // do some operations  
        }  
    }  
}
```

wrong

```
public class VolatileVisibility {  
  
    volatile boolean flag = true;  
  
    public void writerThread() {  
        flag = false;  
    }  
  
    public void readerThread() {  
        while (flag) {  
            // do some operations  
        }  
    }  
}
```

right

Code on the left can have reader thread keep running. Fixed with volatile flag.



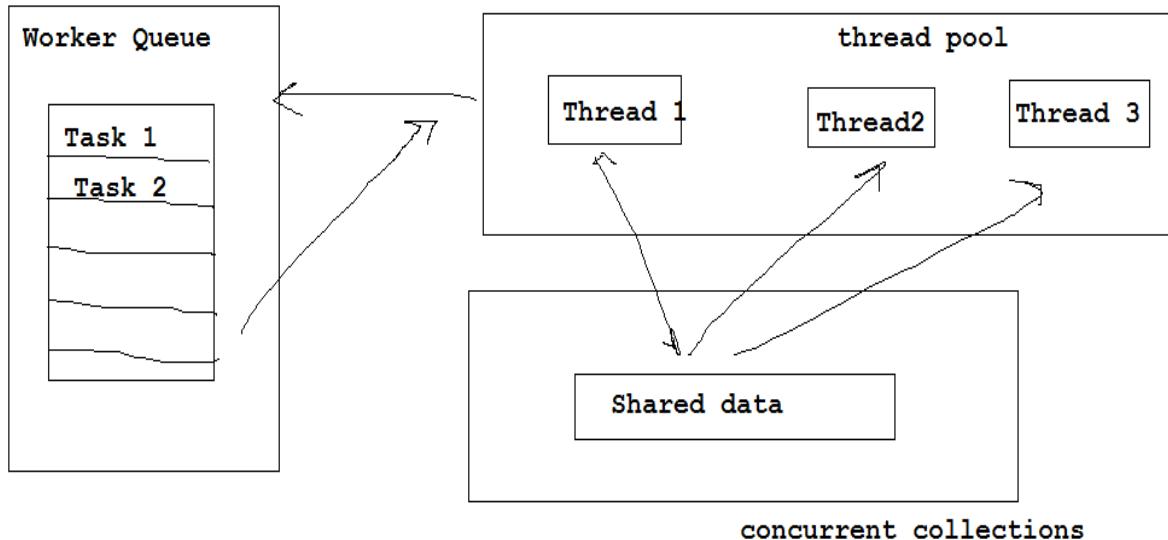
java.util.concurrency j.u.c

□ Introduction

- Concurrency : why very active now?
- Building block of concurrent application
- Java concurrency -over the year
- Fork join -introduction
- Coding examples....

Building block of concurrent application

Building block of concurrent applications



- Thread pool
- Shared data
- Worker queue

Java concurrency -Pre Java 5

- Task Executor
 - No such framework
 - Runnable/Thread
 - No of task= no of threads
 - Use wait() and notify() for thread intercommunication
- Collection
 - Vector, Hashtable
- Worker queue
 - No framework for task dispatching
 - Create an thread when a task to run
- Too primitive low level
 - High cost of thread management
 - Synchronization=poor performance
 - <<Runnable>> can not return values

Java concurrency -Java 5

- Task Executor
 - Executor Service Framework
 - Support scheduling (Timed execution)
 - Thread pools (lifecycle management)
 - Specify order of execution
 - Task
 - Object of <<Callable>> or <<Runnable>>
 - <<Callable>> support for asynchronous communication call() and return values
 - Future: hold result of processing
- Collections
 - ConcurrentHashMap, CopyOnWriteList
- Worker queue
 - Blocking Queue, Deque.....
- Issues
 - No of task can't altered at run time
 - Very difficult and performance intensive when applied to recursive style of problems.....

Java concurrency -Java 7

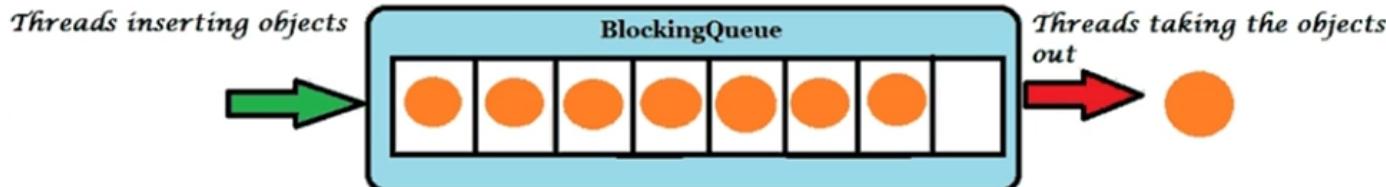
- Fork join framework
 - For supporting a style of programming in which problem are solved recursively splitting them into subtask that are solved in parallel, waiting for them to complete and then composing results !
- JSR 166 Java 7
 - Extend Executor framework for recursive style of problems
- Implements Work stealing algorithm
 - Idle worker “steal” the works from worker who are busy..

Blocking Queue: introduction

- BlockingQueue is an interface under juc package. It help to contain objects when few threads is inserting object and other threads are taking the object out of it
-
- The threads will keep on inserting the objects until the max capacity of BlockingQueue is reached, after which the threads will be blocked and they will not be able to insert further objects, threads will be in blocked state until few objects are taken out from BlockingQueue by other threads and there is space available in the blockingqueue
-
- The threads will keep on taking the objects out from the blockingqueue until there is no object left in blockingqueue, after which the threads will be blocked and they will remain in blocked state until few objects are inserted in the blockingqueue by other threads
-
- Use put() and take() to implement P and C

The **poll()** doesn't wait if the `messageQueue` is empty. It doesn't care. It just gets the value even if the `messageQueue` is empty. Finally the application ended.

The **take()** method waits at that particular line if the `messageQueue` is empty. It only goes to next line if the `messageQueue` got something inside to process. So **take()** method block the thread and keeps it, which makes the application runs endlessly.



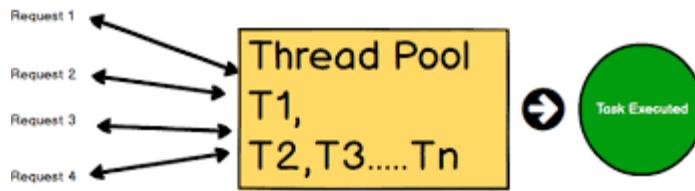
Blocking Queue: Thread safe DS

- `BlockingQueue<E>` is an interface with the following methods

	<i>Throws exception</i>	<i>Special value</i>	<i>Blocks</i>	<i>Times out</i>
Insert	<code>add(e)</code>	<code>offer(e)</code>	<code>put(e)</code>	<code>offer(e, time, unit)</code>
Remove	<code>remove()</code>	<code>poll()</code>	<code>take()</code>	<code>poll(time, unit)</code>
Examine	<code>element()</code>	<code>peek()</code>	<i>N/A</i>	<i>N/A</i>

- until the method returns. If a method *times out* then the method *sblocks* until the time specified is reached, then the method returns.

- Important implementations of `BlockingQueue` are `ArrayBlockingQueue`, `LinkedBlockingQueue`, `PriorityBlockingQueue` and `SynchronousQueue`.



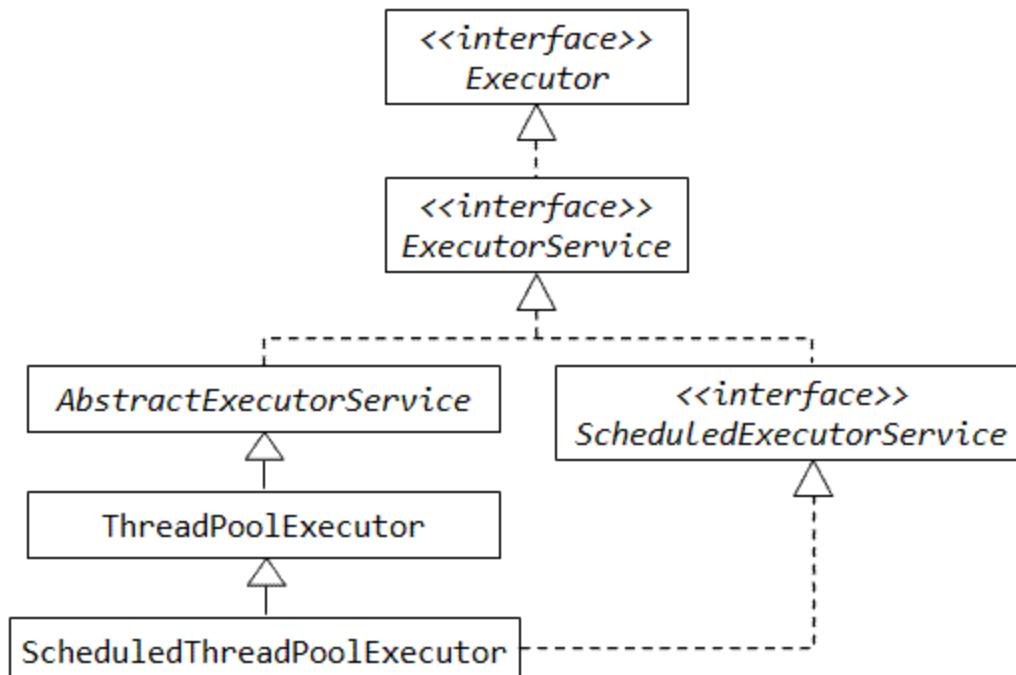
Thread Pooling

Thread Pools are useful when you **need** to limit the number of **threads** running in your application at the same time. ... Instead of starting a new **thread** for every task to execute concurrently, the task can be passed to a **thread pool**. As soon as the **pool** has any idle **threads** the task is assigned to one of them and executed.

Oct 15, 2015

Executor framework

- It separate task of creation and execution of thread
- We only have to implements Runnable (Make a job) and send it to executor
- Then executor is responsible for execution management
- Executor maintain an thread pool
 - avoid continuously spawning of threads



Executors, A framework for creating and managing threads. Executors framework helps you with -

- ▶ **Thread Creation:** It provides various methods for creating threads, more specifically a pool of threads, that your application can use to run tasks concurrently.
- ▶ **Thread Management:** It manages the life cycle of the threads in the thread pool. You don't need to worry about whether the threads in the thread pool are active or busy or dead before submitting a task for execution.
- ▶ **Task submission and execution:** Executors framework provides methods for submitting tasks for execution in the thread pool, and also gives you the power to decide when the tasks will be executed. For example, You can submit a task to be executed now or schedule them to be executed later or make them execute periodically.

Java Concurrency API defines the following three executor interfaces that covers everything that is needed for creating and managing threads -

- ▶ **Executor** - A simple interface that contains a method called `execute(Runnable command)` to launch a task specified by a `Runnable` object.
- ▶ **ExecutorService** - A sub-interface of `Executor` that adds functionality to manage the lifecycle of the tasks. It also provides a `submit()` method whose overloaded versions can accept a `Runnable` as well as a `Callable` object. `Callable` objects are similar to `Runnable` except that the task specified by a `Callable` object can also return a value.
- ▶ **ScheduledExecutorService** - A sub-interface of `ExecutorService`. It adds functionality to schedule the execution of the tasks.

Executor framework Ex With <<Runnable>>

- Consider Task class that contain job to be submitted to executor framework.....

```
class Task implements Runnable
{
    private String name;

    Task(String name){this.name=name;}

    @Override
    public void run() {
        System.out.println("Thread started the job...:"+Thread.currentThread().getName());
        try
        {
            Thread.sleep(100);
        }
        catch(InterruptedException ex){}
        System.out.println("Thread finished the job...:"+Thread.currentThread().getName());
    }
}
```

```
class Server
{
    private ThreadPoolExecutor exe=(ThreadPoolExecutor) Executors.newCachedThreadPool();

    public void executeTask(Task task){
        System.out.println("server: a new job arrived.....");

        exe.execute(task);
        System.out.println("server: pool size:"+exe.getPoolSize());//no of thread in pool

        System.out.println("server: Active count:"+exe.getActiveCount());//no of active threads

        System.out.println("server: Completed task:"+exe.getCompletedTaskCount());//no of task completed

    }

    public void endServer()
    {
        exe.shutdown();
    }
}
```

```
public class ExecutorDemoWithRunnable {

    public static void main(String[] args) {
        Server server=new Server();

        for(int i=0;i<10;i++)
        {
            Task t=new Task("task "+i);
            server.executeTask(t);//providing job to executor framework
        }

        server.endServer();
    }
}
```

Executor with <<Callable>>

- One advantage of Executor framework that We can use <<Callable>> override call() method that can return results
- <<Future>> helps us to retrieve results

```
class FactorialCalculator implements Callable<Integer>
{
    private Integer number;
    FactorialCalculator(Integer n){number=n;}

    @Override
    public Integer call() throws Exception
    {
        int result=1;
        if(number==0 || number==1)
            result=1;
        else
        {
            for(int i=2;i<=number;i++)
            {
                result=result*i;

                Thread.sleep(100); //assume heavy processing
            }
        }

        System.out.println("result:"+result);
        return result;
    }
}
```

```
public static void main(String[] args) throws InterruptedException, ExecutionException {
    ThreadPoolExecutor exe=(ThreadPoolExecutor) Executors.newFixedThreadPool(2);
    for(int i=3;i<30;i++)
    {
        FactorialCalculator cal=new FactorialCalculator(i);
        Future<Integer>result=exe.submit(cal);
        System.out.println(result.get());
    }
    System.out.println("finished all jobs.....");
    exe.shutdown();
}
```

```
public static void main(String[] args) {  
  
    Thread thread1 = new Thread(new Task());  
    thread1.start();  
    System.out.println("Thread Name: " + Thread.currentThread().getName());  
}  
  
static class Task implements Runnable {  
    public void run() {  
        System.out.println("Thread Name: " + Thread.currentThread().getName());  
    }  
}
```

Running a task asynchronously

main thread

t1.start()

sout (in main)

thread-0

sout (in other thread)



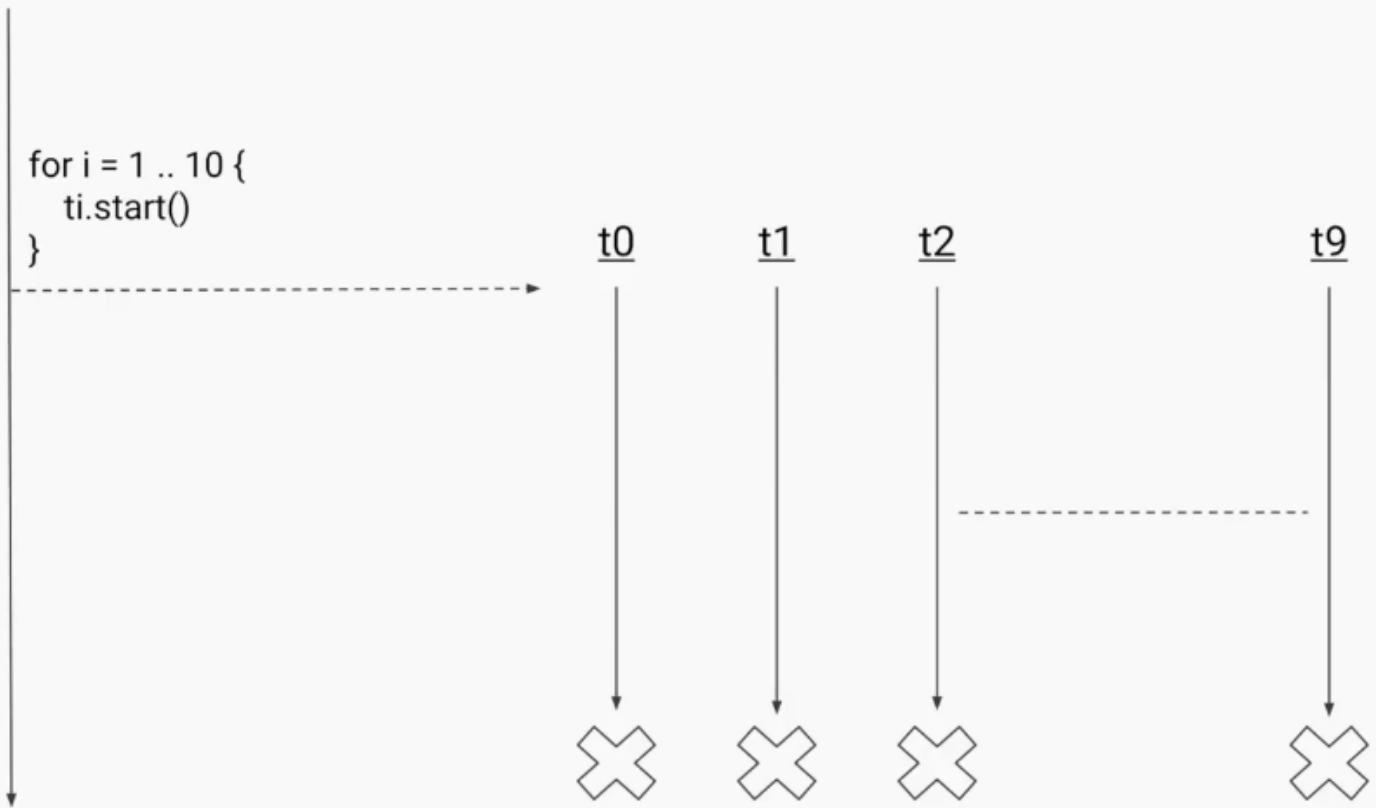
Running multiple tasks asynchronously

```
public static void main(String[] args) {  
  
    for (int i = 0; i < 10; i++) {  
        Thread thread = new Thread(new Task());  
        thread.start();  
    }  
    System.out.println("Thread Name: " + Thread.currentThread().getName());  
}  
  
static class Task implements Runnable {  
    public void run() {  
        System.out.println("Thread Name: " + Thread.currentThread().getName())  
    }  
}
```

Running a task asynchronously

main thread

```
for i = 1 .. 10 {  
    ti.start()  
}
```

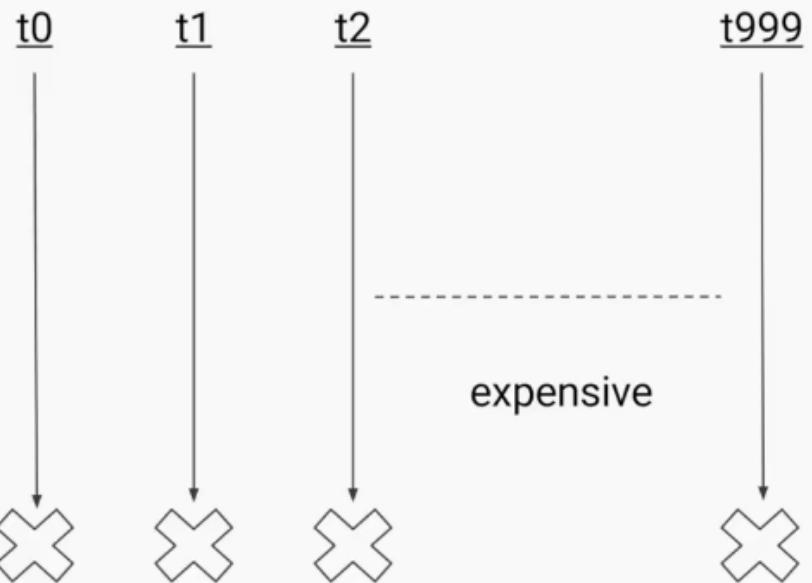


Running multiple tasks asynchronously

main thread

```
for i = 1 .. 1000 {  
    ti.start()  
}
```

1 Java thread = 1 OS thread



Running 1000s of tasks asynchronously

main thread

create pool

```
for i = 1 .. 1000 {  
    submit-tasks  
}
```

Fixed number of threads

t0

t1

t2

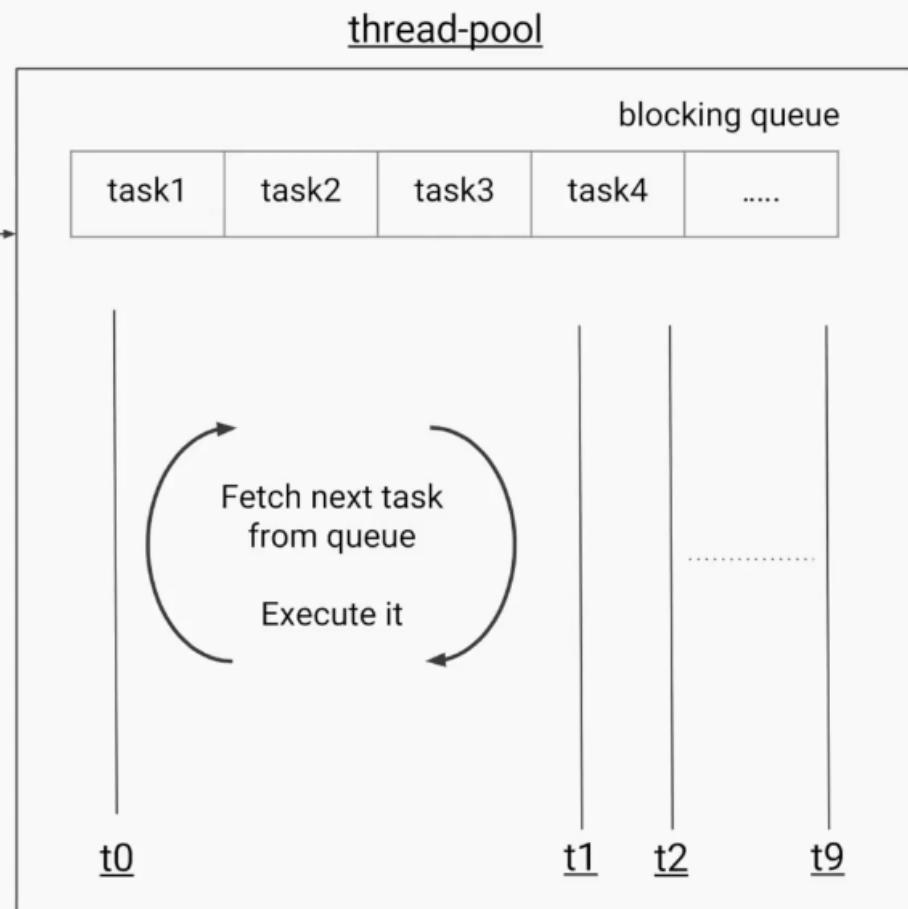
t9

```
public static void main(String[] args) {  
    // create the pool  
    ExecutorService service = Executors.newFixedThreadPool( nThreads: 10 );  
  
    // submit the tasks for execution  
    for (int i = 0; i < 100; i++) {  
        service.execute(new Task());  
    }  
    System.out.println("Thread Name: " + Thread.currentThread().getName());  
}  
  
static class Task implements Runnable {  
    public void run() {  
        System.out.println("Thread Name: " + Thread.currentThread().getName());  
    }  
}
```

Submit tasks using ThreadPool

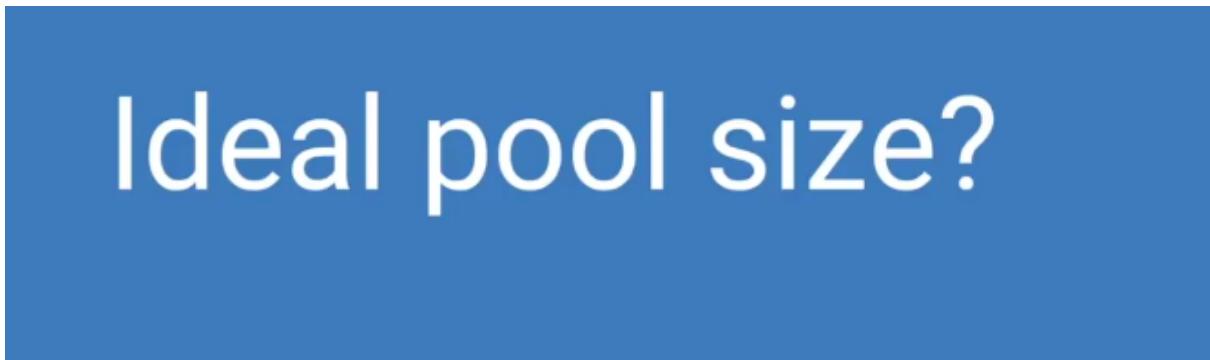
main thread

```
for i = 1 .. 100 {  
    service.execute(new Task());  
}
```



How thread pool internally works

- 
1. FixedThreadPool
 2. CachedThreadPool
 3. ScheduledThreadPool
 4. SingleThreadedExecutor
- 

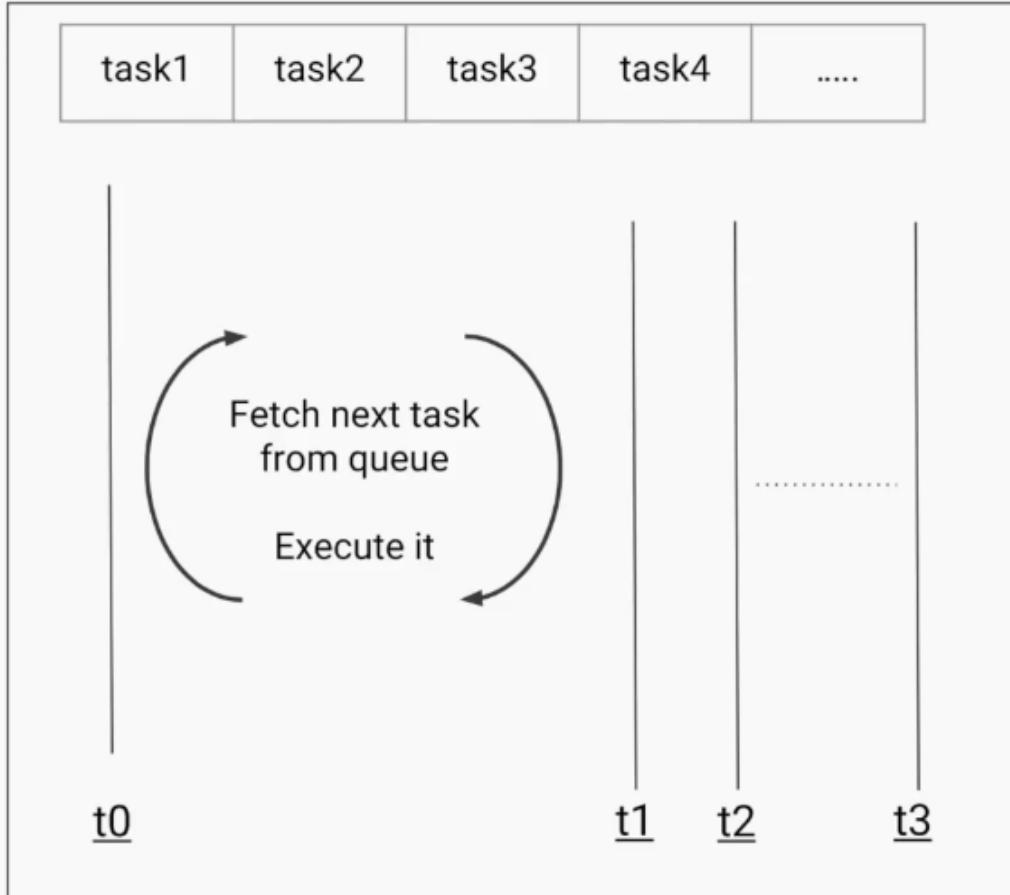


Ideal pool size?

Task Type	Ideal pool size	Considerations
CPU intensive	CPU Core count	How many other applications (or other executors/threads) are running on the same CPU.
IO intensive	High	Exact number will depend on rate of task submissions and average task wait time. Too many threads will increase memory consumption too.

Pool size depends on the task type

thread-pool



CPU

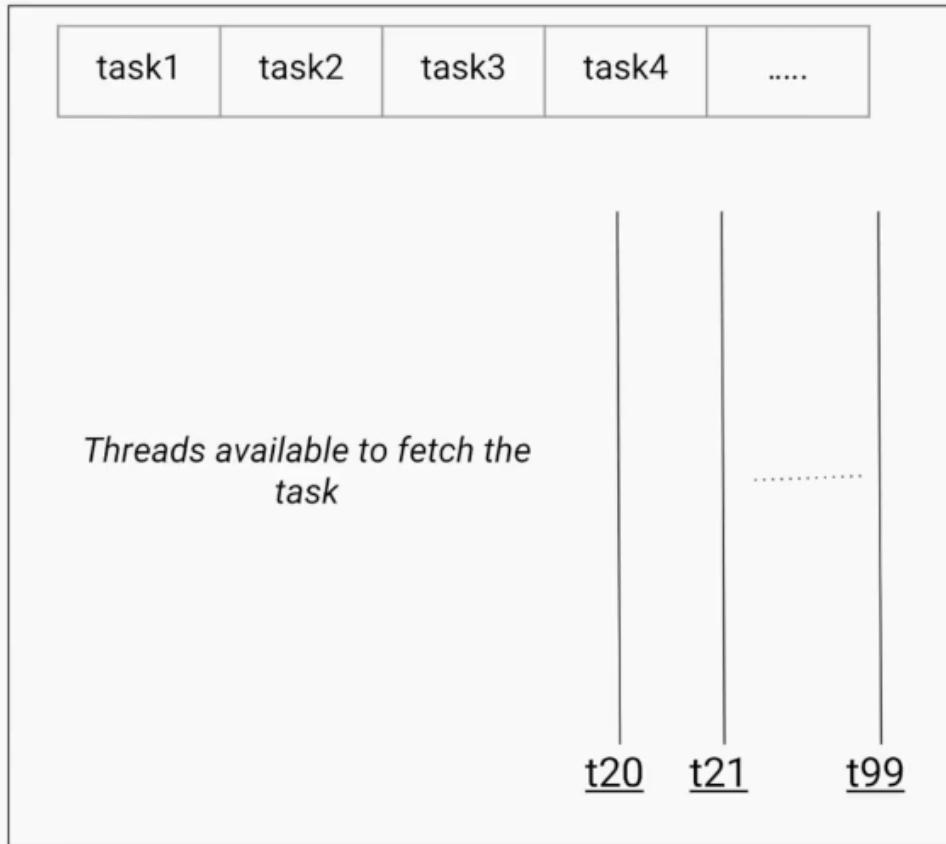
Core 1	Core 2
Core 3	Core 4

Max 4 threads can run at a time

```
public static void main(String[] args) {  
  
    // get count of available cores  
    int coreCount = Runtime.getRuntime().availableProcessors();  
    ExecutorService service = Executors.newFixedThreadPool(coreCount);  
  
    // submit the tasks for execution  
    for (int i = 0; i < 100; i++) {  
        service.execute(new CpuIntensiveTask());  
    }  
}  
  
static class CpuIntensiveTask implements Runnable {  
    public void run() {  
        // some CPU intensive operations  
    }  
}
```

Pool size for CPU intensive Tasks

thread-pool



CPU

Core 1	Core 2
Core 3	Core 4

Max 4 threads can run at a time

Waiting threads

t3	t5	t6	t7
----	----	----	----	------

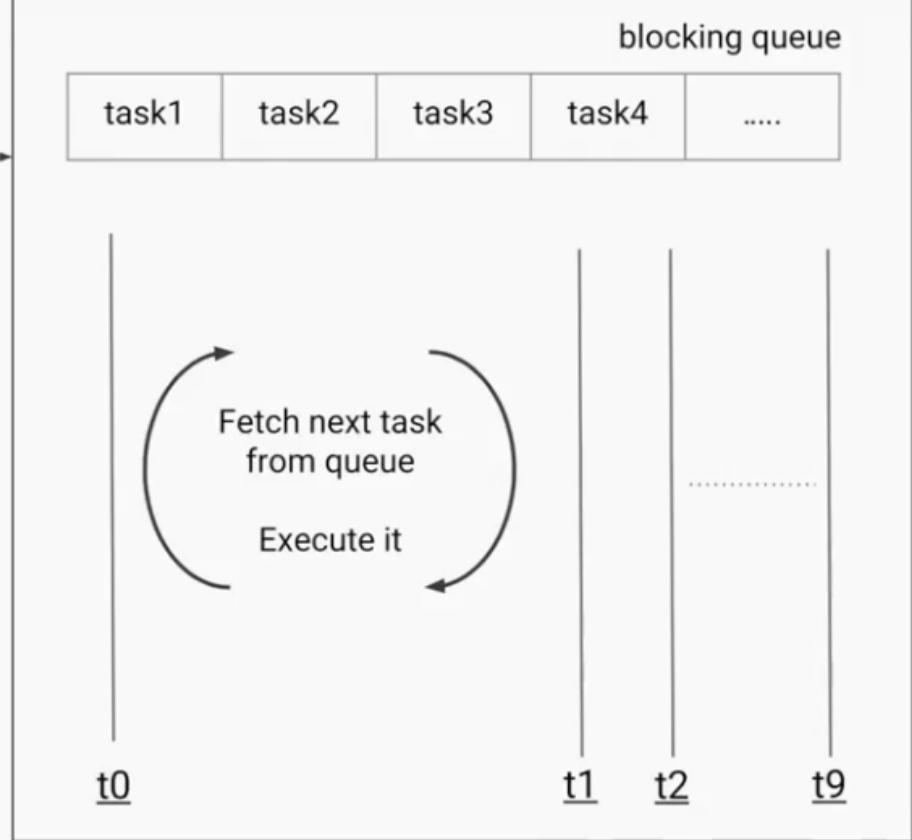
Threads waiting for IO operation response from the OS

```
public static void main(String[] args) {  
    // much higher count for IO tasks  
    ExecutorService service = Executors.newFixedThreadPool( nThreads: 100 );  
  
    // submit the tasks for execution  
    for (int i = 0; i < 100; i++) {  
        service.execute(new IOTask());  
    }  
}  
  
static class IOTask implements Runnable {  
    public void run() {  
        // some IO operations which will cause thread to block/wait  
    }  
}
```

main thread

```
for i = 1 .. 100 {  
    service.execute(new Task());  
}
```

thread-pool



How thread pool internally works

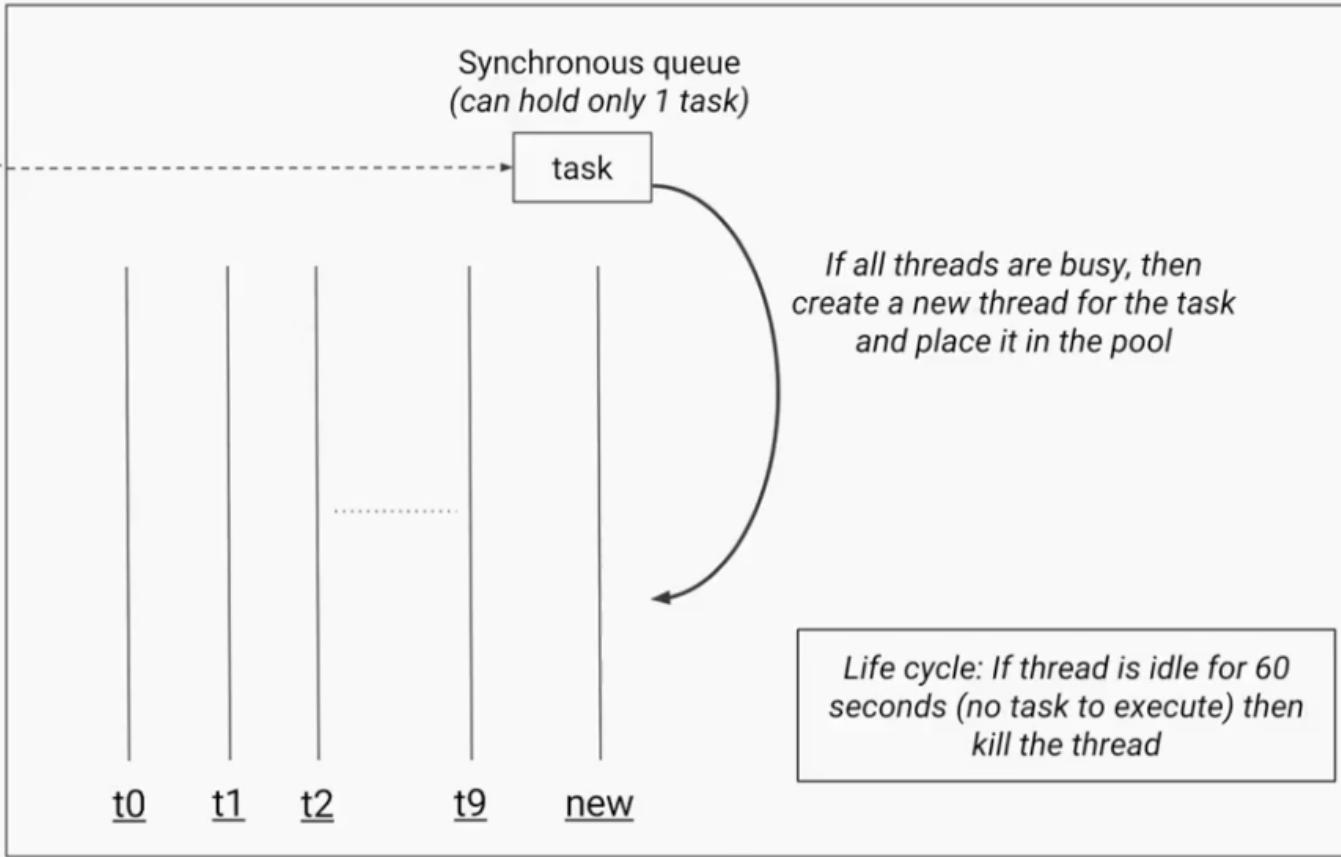
```
public static void main(String[] args) {  
    // create the pool  
    ExecutorService service = Executors.newFixedThreadPool( nThreads: 10 );  
  
    // submit the tasks for execution  
    for (int i = 0; i < 100; i++) {  
        service.execute(new Task());  
    }  
    System.out.println("Thread Name: " + Thread.currentThread().getName());  
}  
  
static class Task implements Runnable {  
    public void run() {  
        System.out.println("Thread Name: " + Thread.currentThread().getName())  
    }  
}
```

Submit tasks using ThreadPool

main-thread

```
for i = 1 .. 100 {  
    service.execute(new Task());  
}
```

thread-pool



```
public static void main(String[] args) {  
  
    // for lot of short lived tasks  
    ExecutorService service = Executors.newCachedThreadPool();  
  
    // submit the tasks for execution  
    for (int i = 0; i < 100; i++) {  
        service.execute(new Task());  
    }  
}  
  
static class Task implements Runnable {  
    public void run() {  
        // short lived task  
    }  
}
```

main-thread

Service.schedule

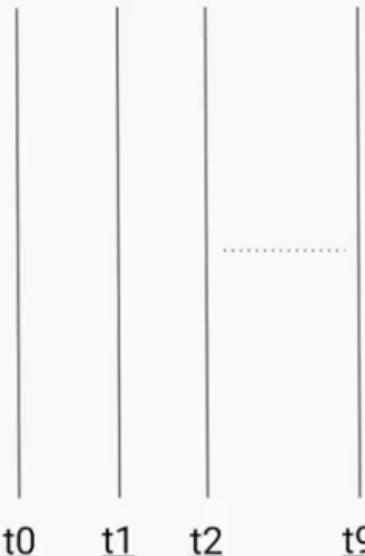
service.scheduleAtFixedRate

service.scheduleAtFixedDelay

thread-pool

Delay queue

task1	task2	task3	task4
-------	-------	-------	-------	-------



Schedule the tasks to run based on time delay (and retrigger for fixedRate / fixedDelay)

Life Cycle: More threads are created if required.

Scheduled Thread Pool =



- Pool sizes changes
- Queue types
- Task rejections
- Life cycle methods

ThreadPoolExecutor Constructor customization

```
ExecutorService service = Executors.newFixedThreadPool( nThreads: 10 );
```



```
public static ExecutorService newFixedThreadPool(int nThreads) {  
    return new ThreadPoolExecutor(nThreads, nThreads,  
        keepAliveTime: 0L, TimeUnit.MILLISECONDS,  
        new LinkedBlockingQueue<Runnable>());  
}
```

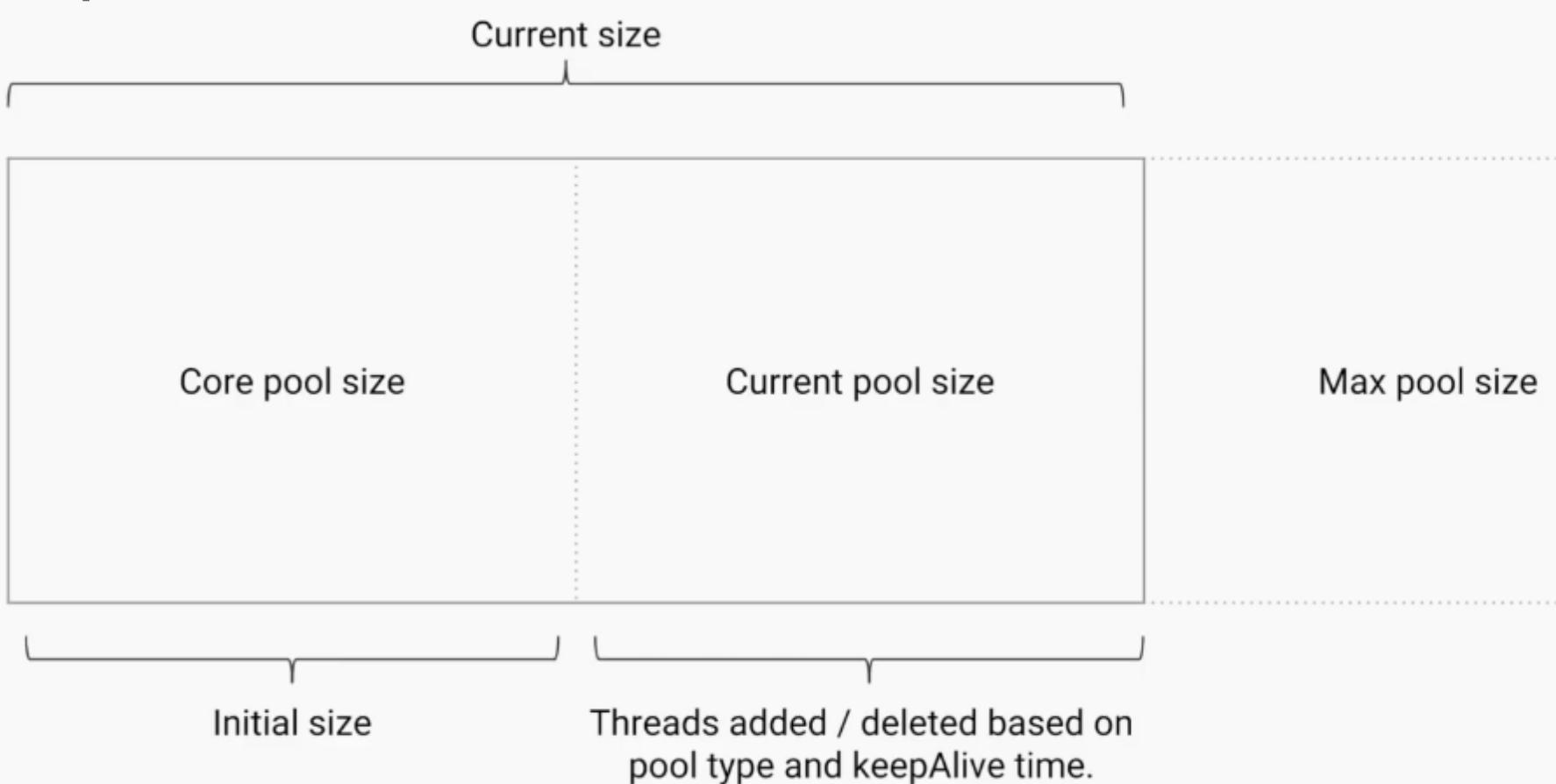


```
public ThreadPoolExecutor(int corePoolSize,  
    int maximumPoolSize,  
    long keepAliveTime,  
    TimeUnit unit,  
    BlockingQueue<Runnable> workQueue,  
    ThreadFactory threadFactory,  
    RejectedExecutionHandler handler) {
```

ThreadPoolExecutor Constructor parameters

Parameter	Type	Meaning
corePoolSize	int	Minimum/Base size of the pool
maxPoolSize	int	Maximum size of the pool
keepAliveTime + unit	long	Time to keep an idle thread alive (after which it is killed)
workQueue	BlockingQueue	Queue to store the tasks from which threads fetch them
threadFactory	ThreadFactory	The factory to use to create new threads
handler	RejectedExecutionHandler	Callback to use when tasks submitted are rejected

ThreadPoolExecutor :understanding pool size



ThreadPoolExecutor :understanding pool size

Parameter	FixedThreadPool	CachedThreadPool	ScheduledThreadPool	SingleThreaded
corePoolSize	constructor-arg	0	constructor-arg	1
maxPoolSize	same as corePoolSize	Integer.MAX_VALUE	Integer.MAX_VALUE	1
keepAliveTime	0 seconds	60 seconds	60 seconds	0 seconds

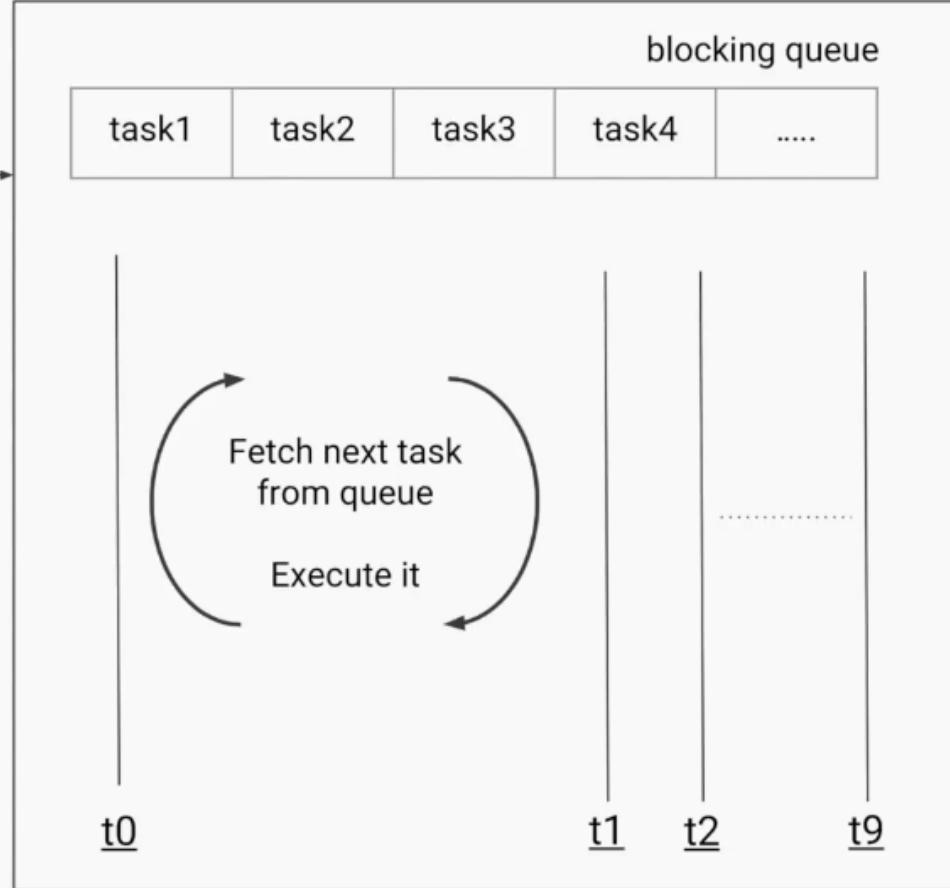
*Note: Core pool threads are never killed unless
allowCoreThreadTimeOut(boolean value) is set to true.*

Type of Queues used in thread pool

main thread

```
for i = 1 .. 100 {  
    service.execute(new Task());  
}
```

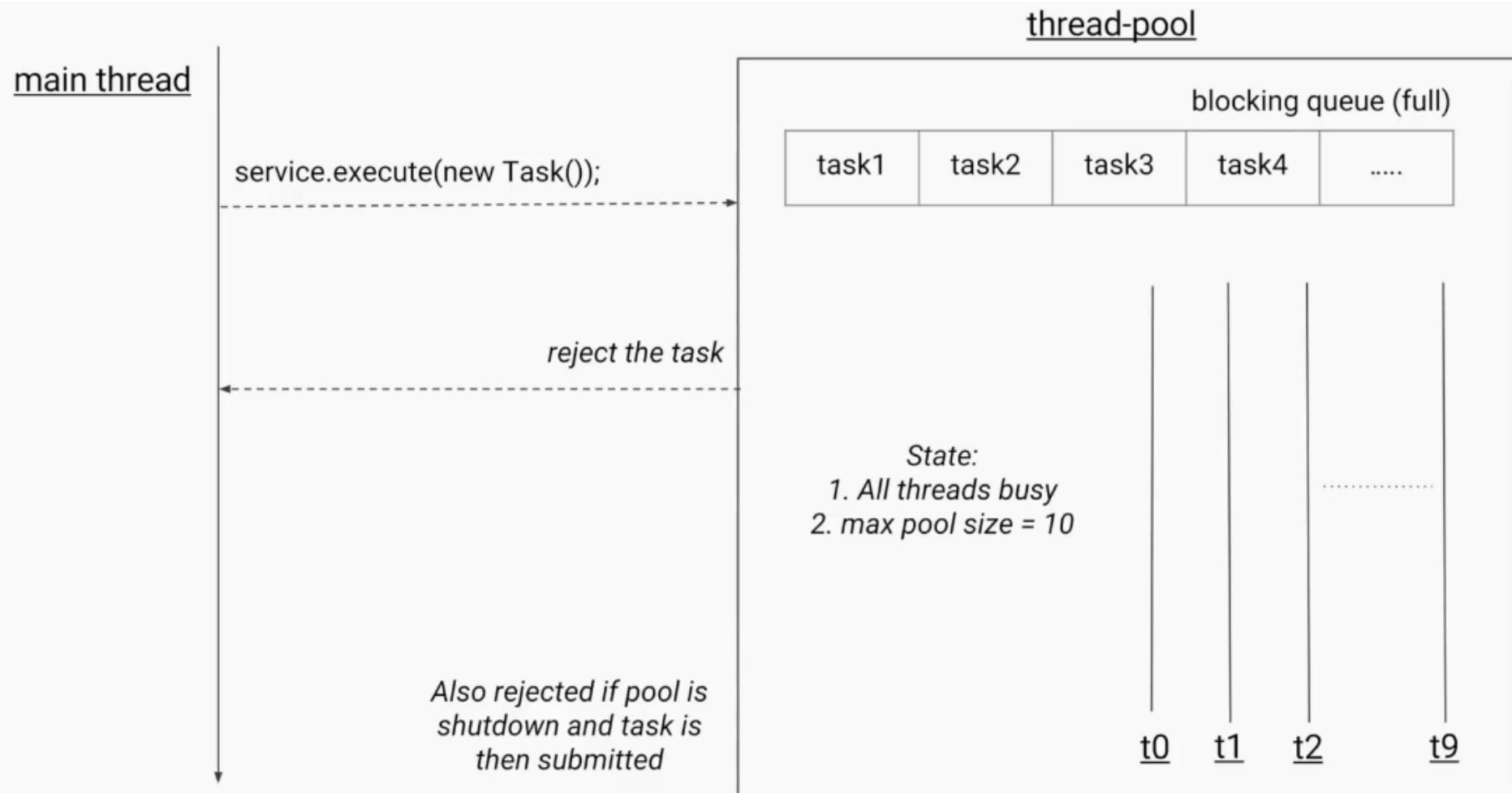
thread-pool



ThreadPoolExecutor :understanding Queue types

Pool	Queue Type	Why?
FixedThreadPool	LinkedBlockingQueue	Threads are limited, thus unbounded queue to store all tasks.
SingleThreadExecutor	LinkedBlockingQueue	<i>Note: Since queue can never become full, new threads are never created.</i>
CachedThreadPool	SynchronousQueue	Threads are unbounded, thus no need to store the tasks. Synchronous queue is a queue with single slot
ScheduledThreadPool	DelayedWorkQueue	Special queue that deals with schedules/time-delays
Custom	ArrayBlockingQueue	Bounded queue to store the tasks. If queue gets full, new thread is created (as long as count is less than maxPoolSize).

thread pool: handling rejections



thread pool: rejection policies

Policy	What it means?
AbortPolicy	Submitting new tasks throws RejectedExecutionException (Runtime exception)
DiscardPolicy	Submitting new tasks silently discards it.
DiscardOldestPolicy	Submitting new tasks drops existing oldest task, and new task is added to the queue.
CallerRunsPolicy	Submitting new tasks will execute the task on the caller thread itself. This can create feedback loop where caller thread is busy executing the task and cannot submit new tasks at fast pace.

thread pool: RejectedExecutionException

```
ExecutorService service
    = new ThreadPoolExecutor(
        corePoolSize: 10,
        maximumPoolSize: 100,
        keepAliveTime: 120, TimeUnit.SECONDS,
        new ArrayBlockingQueue<>( capacity: 300));

try {
    service.execute(new Task());
} catch (RejectedExecutionException e) {
    System.err.println("task rejected " + e.getMessage());
}
```

thread pool: RejectedExecutionException

```
ExecutorService service
    = new ThreadPoolExecutor(
        corePoolSize: 10,
        maximumPoolSize: 100,
        keepAliveTime: 120, TimeUnit.SECONDS,
        new ArrayBlockingQueue<>( capacity: 300),
        new CustomRejectionHandler());
    •

private static class CustomRejectionHandler implements RejectedExecutionHandler{
    @Override
    public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
        // logging / operations to perform on rejection
    }
}
```

thread pool: Life Cycle method

```
ExecutorService service = Executors.newFixedThreadPool( nThreads: 10 );
for (int i = 0; i < 100; i++) {
    service.execute(new Task());
}

// initiate shutdown
service.shutdown();

// will throw RejectionExecutionException
// service.execute(new Task());

// will return true, since shutdown has begun
service.isShutdown();

// will return true if all tasks are completed
// including queued ones
service.isTerminated();

// block until all tasks are completed or if timeout occurs
service.awaitTermination( timeout: 10, TimeUnit.SECONDS );

// will initiate shutdown and return all queued tasks
List<Runnable> runnables = service.shutdownNow();
```

thread pool: Example

```
// create the pool
ExecutorService service = Executors.newFixedThreadPool( nThreads: 10 );

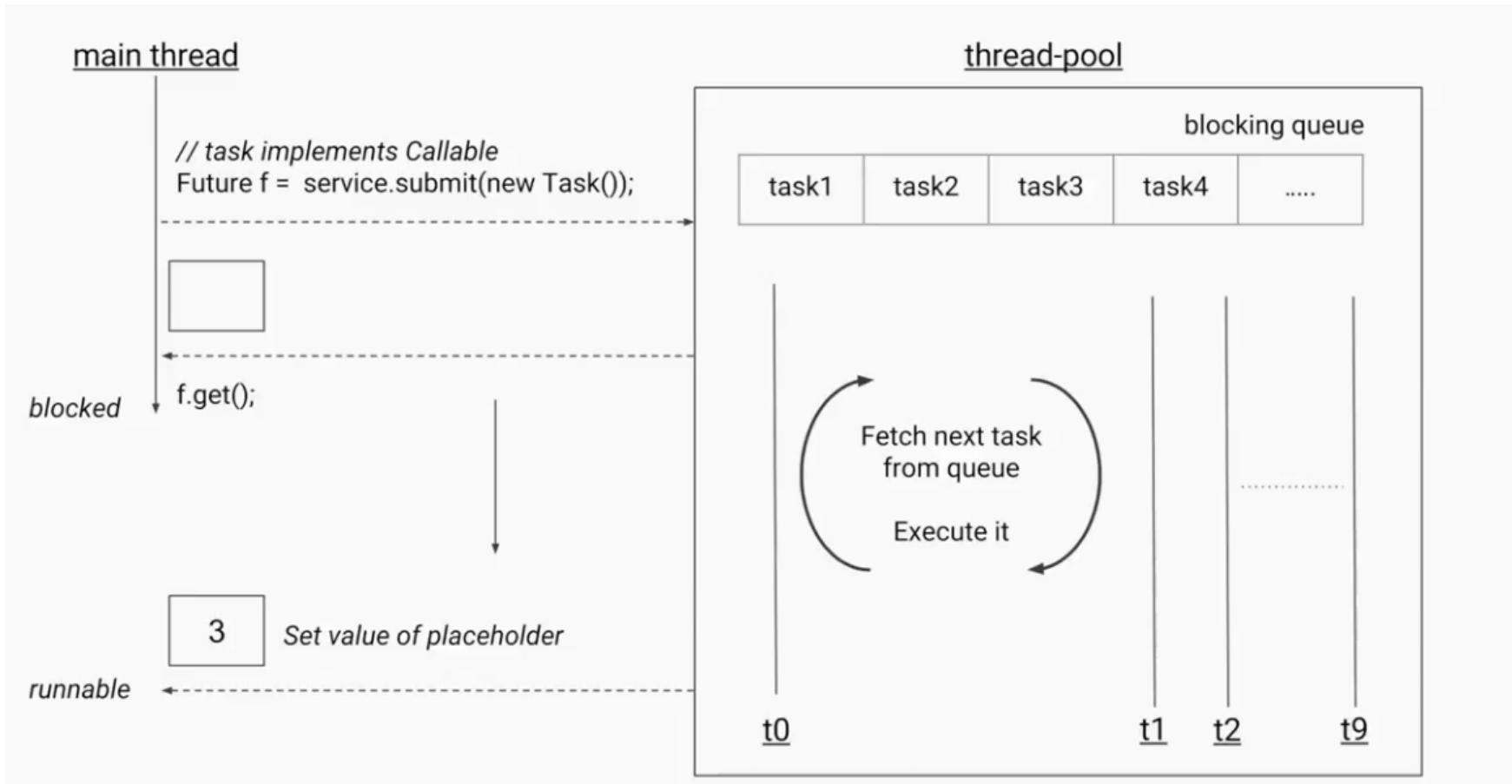
// submit the tasks for execution
List<Future> allFutures = new ArrayList<>();
for (int i = 0; i < 100; i++) {
    Future<Integer> future = service.submit(new Task());
    allFutures.add(future);
}

// 100 futures, with 100 placeholders.

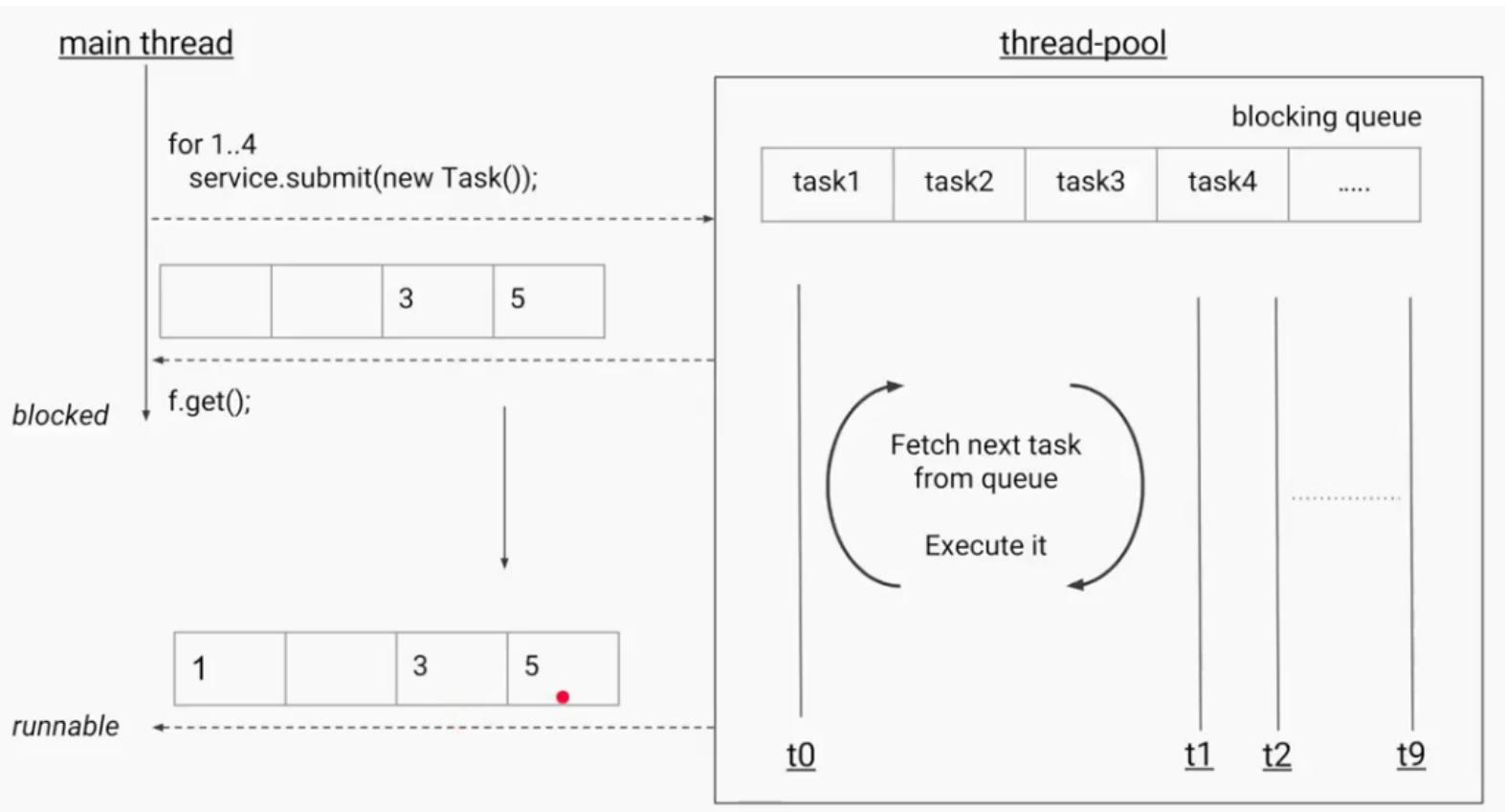
// perform some unrelated operations

// 1 sec
try {
    Integer result = future.get(); // blocking
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    e.printStackTrace();
}
```

thread pool: Future how it work?



thread pool: Blocked till all work is not done! Caution



```
public static void main(String[] args) {  
  
    // for scheduling of tasks  
    ScheduledExecutorService service = Executors.newScheduledThreadPool( corePoolSize: 10 );  
  
    // task to run after 10 second delay  
    service.schedule(new Task(), delay: 10, SECONDS);  
  
    // task to run repeatedly every 10 seconds  
    service.scheduleAtFixedRate(new Task(), initialDelay: 15, period: 10, SECONDS);  
  
    // task to run repeatedly 10 seconds after previous task completes  
    service.scheduleWithFixedDelay(new Task(), initialDelay: 15, delay: 10, TimeUnit.SECONDS);  
  
}  
  
static class Task implements Runnable {  
    public void run() {  
        // task that needs to run  
        // based on schedule  
    }  
}
```

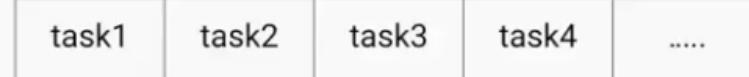
Scheduled Executor

main thread

```
for i = 1 .. 100 {  
    service.execute(new Task());  
}
```

thread-pool

blocking queue



Fetch next task
from queue

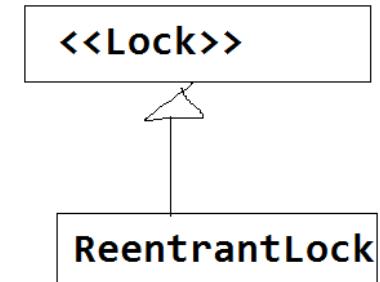
Execute it

t0

*Life Cycle: Recreates thread if killed
because of the task.*

Synchronization with Locks: ReentrantLock

- Synchronization with Locks
 - More flexible
 - Better performance as compared to `wait()` `notify()`



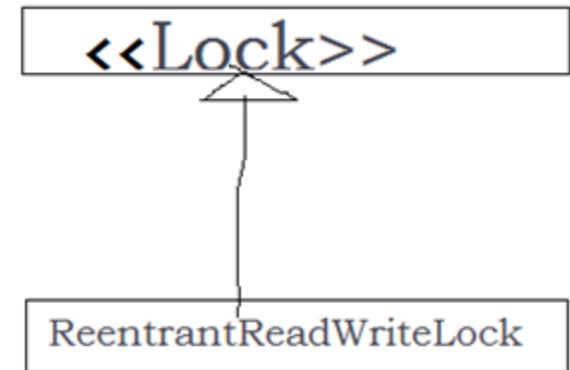
```
class PrintQueue
{
    private final Lock queLock=new ReentrantLock();           ← creating an lock
    public void printJob()
    {
        queLock.lock();                                     ← getting control of
                                                               lock
        try{                                                 ← must be unlock in
            System.out.println("Thread "+Thread.currentThread().getName()+" Done the job");
            Thread.sleep(100);
        }
        catch(InterruptedException ex){}

        finally{
            queLock.unlock();                                ← must be unlock in
        }
    }
}
```

The code snippet illustrates the use of a `ReentrantLock`. It shows a `PrintQueue` class with a private field `queLock` of type `Lock`, which is annotated with `<<Lock>>`. Inside the `printJob` method, the `lock` method is called on `queLock`, annotated with "getting control of lock". Inside the `try` block, there is a `System.out.println` statement and a `Thread.sleep` call. Finally, in the `finally` block, the `unlock` method is called on `queLock`, annotated with "must be unlock in finally block".

Synchronization with Locks: ReentrantReadWriteLock

- ReentrantReadWriteLock has two locks
 - One for read
 - More than one thread can read
 - One for write
 - Only one thread can write
- Ex: Lets consider case when one thread writing Price Information that is read by more than one reader

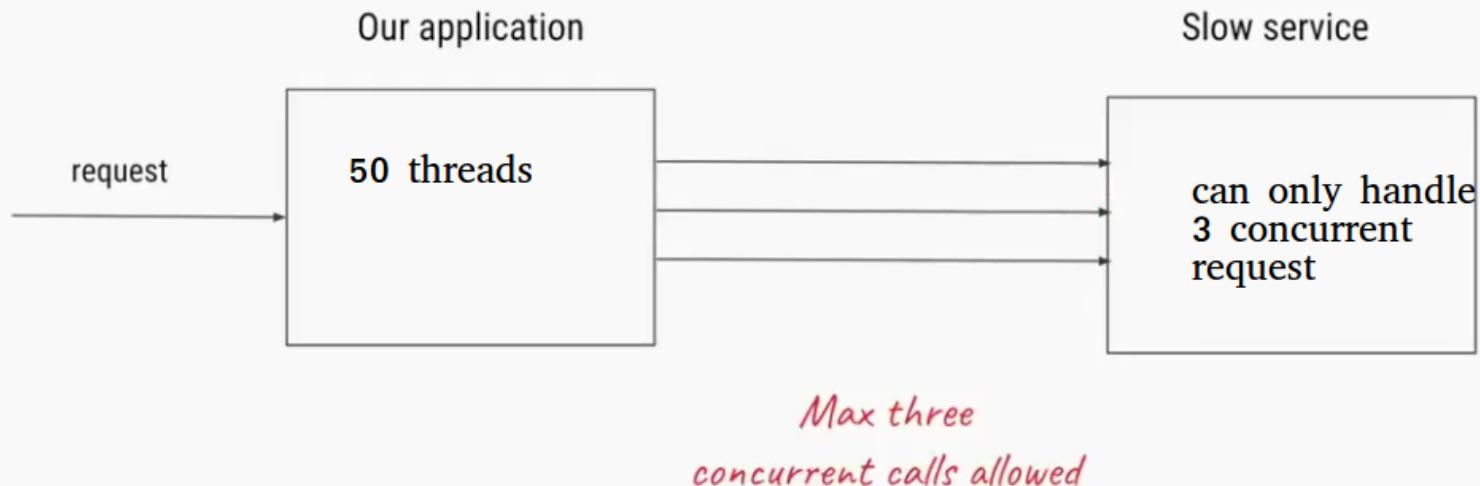


ReentrantReadWriteLock

```
class PriceInfo
{
    private double price1,price2;
    private ReadWriteLock lock=new ReentrantReadWriteLock();
    PriceInfo(){
        price1=1.0;
        price2=2.0;
    }
    public double getPrice1(){
        lock.readLock().lock();
        double value=price1;
        lock.readLock().unlock();
        return value;
    }
    public double getPrice2(){
        lock.readLock().lock();
        double value=price2;
        lock.readLock().unlock();
        return value;
    }
    public void setPrice(double price1, double price2){
        lock.writeLock().lock();
        this.price1=price1;
        this.price2=price2;
        lock.writeLock().unlock();
    }
}
```

Semaphores: use cases

Use Case: We have a slow API to class from our application, it can only handle 3 concurrent class while we have 50 thread to call that service? How to handle?

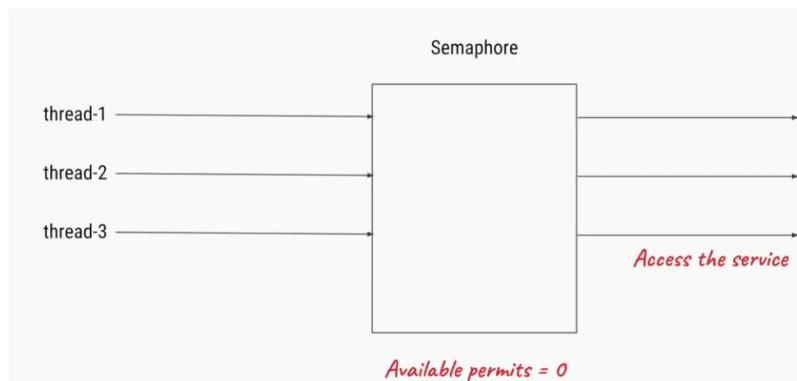
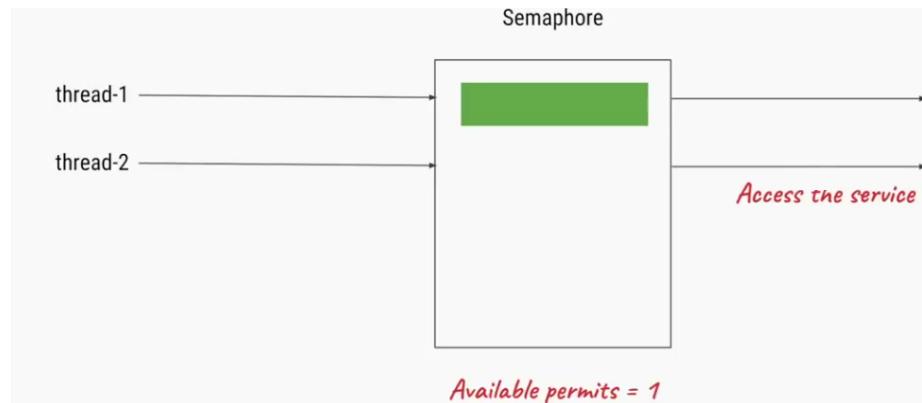


Semaphores: use cases

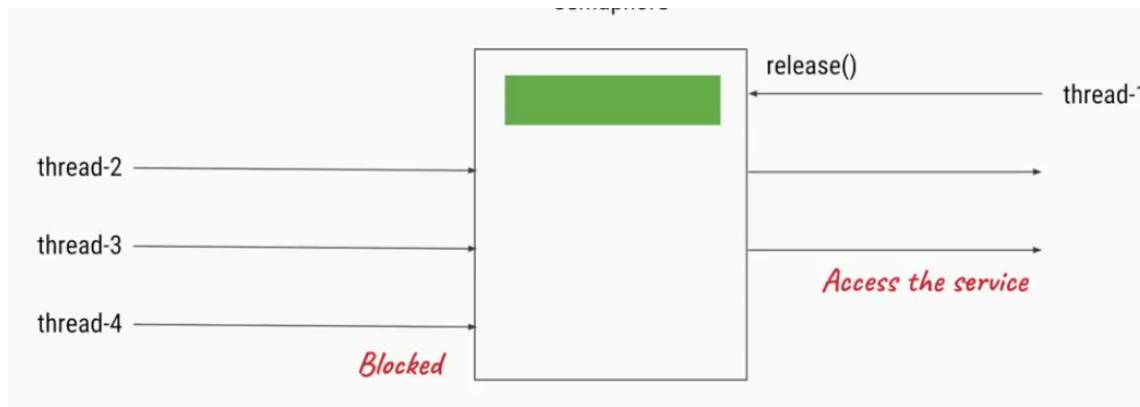
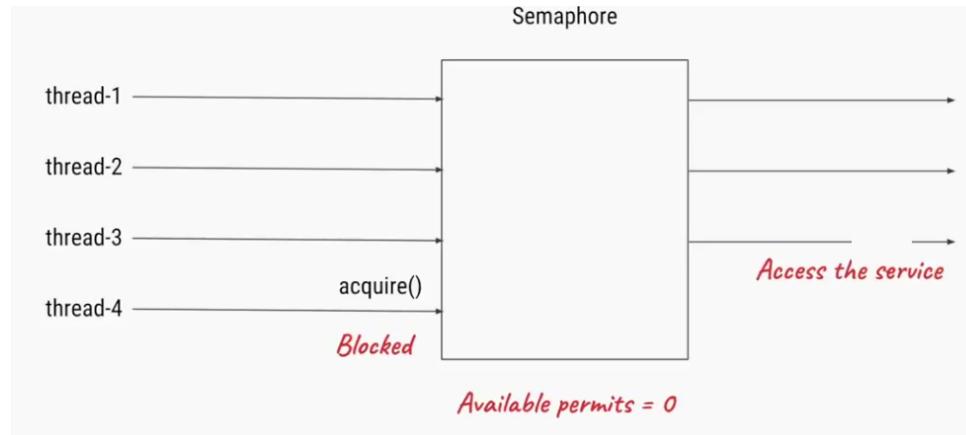
```
public static void main(String[] args) throws InterruptedException {  
    ExecutorService service = Executors.newFixedThreadPool( nThreads: 50 );  
    IntStream.of(1000).forEach(i -> service.execute(new Task()));  
  
    service.shutdown();  
    service.awaitTermination( timeout: 1, TimeUnit.MINUTES );  
}  
  
static class Task implements Runnable {  
  
    @Override  
    public void run() {  
        // some processing  
        // IO call to the slow service  
        // rest of processing  
    }  
}
```

← This might be called 50 times concurrently!!

Semaphores: how permit works?

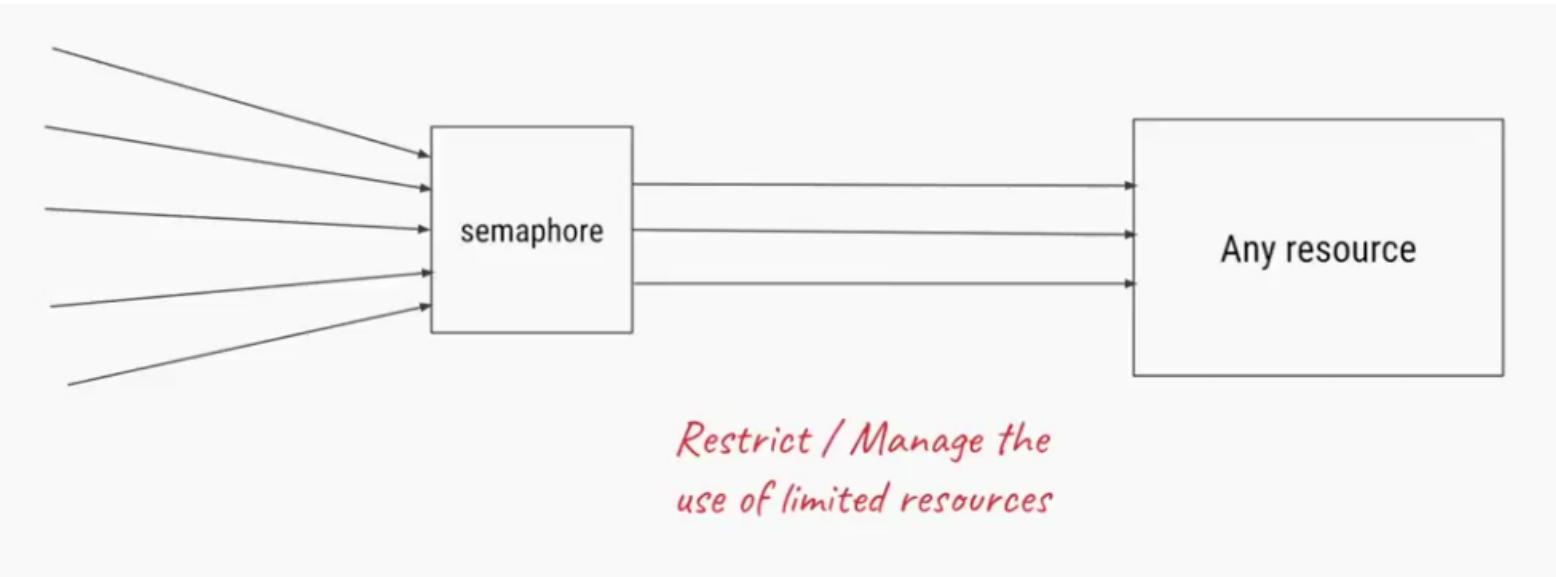


Semaphores: how permit works?



```
public static void main(String[] args) throws InterruptedException {  
    Semaphore semaphore = new Semaphore( permits: 3);  
  
    ExecutorService service = Executors.newFixedThreadPool( nThreads: 50);  
    IntStream.of(1000).forEach(i -> service.execute(new Task(semaphore)));  
  
    service.shutdown();  
    service.awaitTermination( timeout: 1, TimeUnit.MINUTES);  
}  
  
static class Task implements Runnable {  
  
    @Override  
    public void run() {  
        // some processing  
  
        semaphore.acquire(); ← Only 3 threads can acquire  
        // IO call to the slow service  
        semaphore.release();  
        // rest of processing  
    }  
}
```

```
public static void main(String[] args) throws InterruptedException {  
    Semaphore semaphore = new Semaphore( permits: 3);  
  
    ExecutorService service = Executors.newFixedThreadPool( nThreads: 50);  
    IntStream.of(1000).forEach(i -> service.execute(new Task(semaphore)));  
  
    service.shutdown();  
    service.awaitTermination( timeout: 1, TimeUnit.MINUTES);  
}  
  
static class Task implements Runnable {  
  
    @Override  
    public void run() {  
        // some processing  
  
        semaphore.acquireUninterruptibly(); Only 3 threads can acquire  
        // IO call to the slow service at a time  
        semaphore.release();  
  
        // rest of processing  
    }  
}
```



Method	Meaning
tryAcquire	Try to acquire, if no permit available, do not block. Continue doing something else.
tryAcquire (timeout)	Same as above but with timeout
availablePermits	Returns count of permits available
new Semaphore (count, fairness)	FIFO. Fairness guarantee for threads waiting the longest.

Thread synchronization utilities:CountDownLatch

- Waiting for multiple concurrent events
- Allow one/more thread to wait until a set of operations are completed..
- CountDownLatch class initialized with an integer number i.e. no of operations that thread must wait to complete before they finished
- When a thread want to wait for executions of those operations, it uses await() method, it put thread to sleep until operation are complete
- When one of these operation complete, it use CountDownLatches, countDown() method to decrease the count
- When it (counter) become zero, the class wake up all the threads that sleeping in await() method

```
public static void main(String[] args) throws InterruptedException {  
    ExecutorService executor = Executors.newFixedThreadPool( nThreads: 4 );  
  
    CountDownLatch latch = new CountDownLatch(3);  
    executor.submit(new DependentService(latch));  
    executor.submit(new DependentService(latch));  
    executor.submit(new DependentService(latch));  
  
    latch.await();  
  
    System.out.println("All dependant services initialized");  
    // program initialized, perform other operations  
}
```

```
public static class DependentService implements Runnable {  
  
    private CountDownLatch latch;  
    public DependentService(CountDownLatch latch) { this.latch = latch; }  
  
    @Override  
    public void run() {  
        // startup task  
        latch.countDown();  
        // continue w/ other operations  
    }  
}
```

Main thread

Dependant service threads

Latch count = 3

Latch count = 2

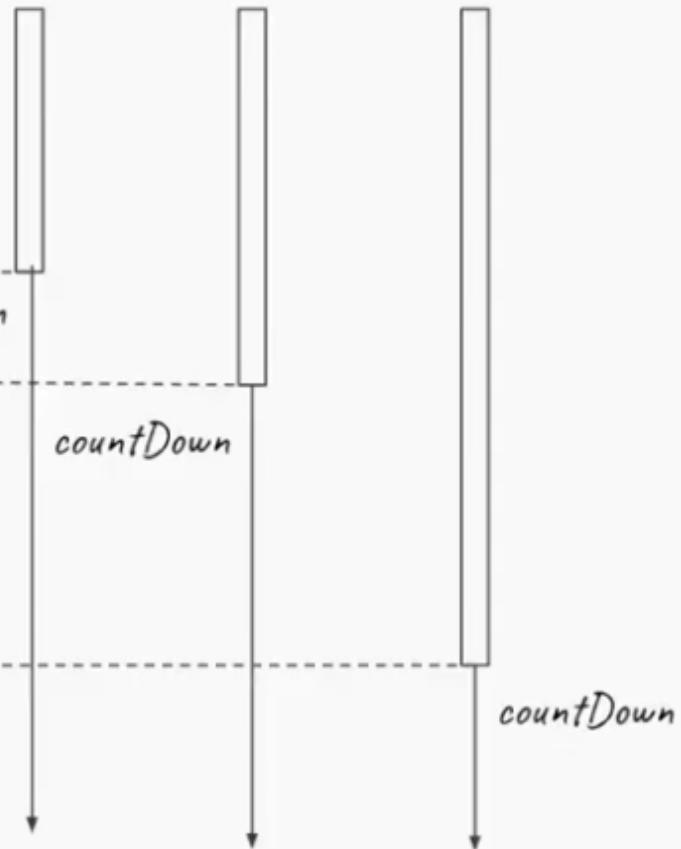
Latch count = 1

Latch count = 0

countDown

countDown

countDown



CyclicBarrier: Use cases

- Let we have a game of 3 player, we want to repeatedly send message to all 3 player without discrimination, the task run in infine loop (repedatly), all thread threds say barrier.await() (all can do in different time) , all three thread need to wait then when all arived then they continue processing

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

    ExecutorService executor = Executors.newFixedThreadPool( nThreads: 4);

    CyclicBarrier barrier = new CyclicBarrier( parties: 3);
    executor.submit(new Task(barrier));
    executor.submit(new Task(barrier));
    executor.submit(new Task(barrier));

    Thread.sleep( millis: 2000);
}

public static class Task implements Runnable {

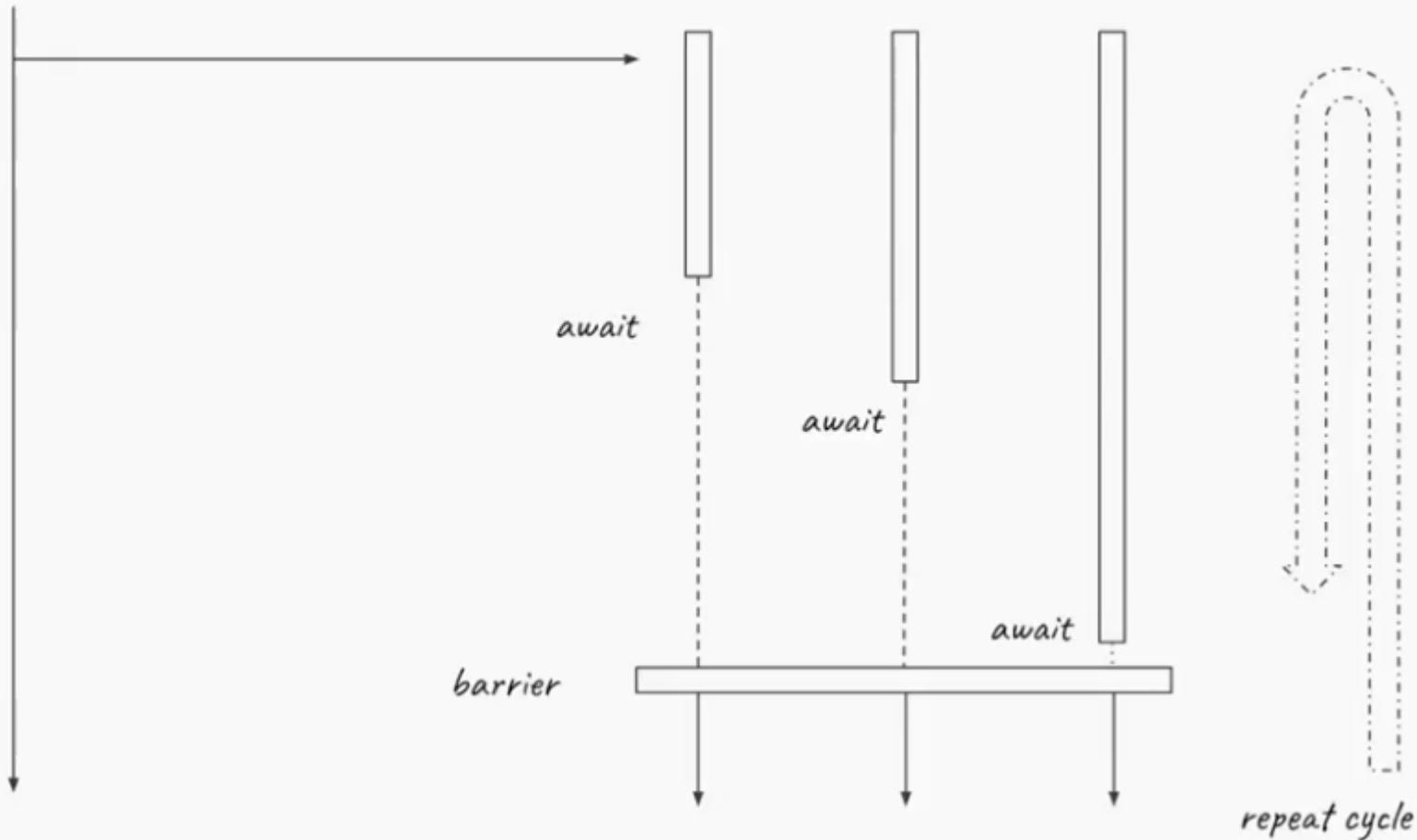
    private CyclicBarrier barrier;
    public Task(CyclicBarrier barrier) { this.barrier = barrier; }

    @Override
    public void run() {

        while (true) {
            try {
                barrier.await();
            } catch (InterruptedException | BrokenBarrierException e) {
                e.printStackTrace();
            }
            // send message to corresponding system
        }
    }
}
```

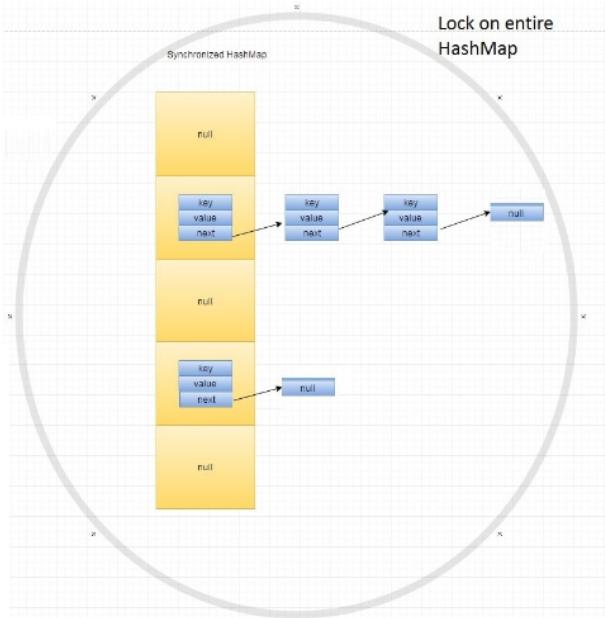
Main thread

Tasks to perform repeatedly

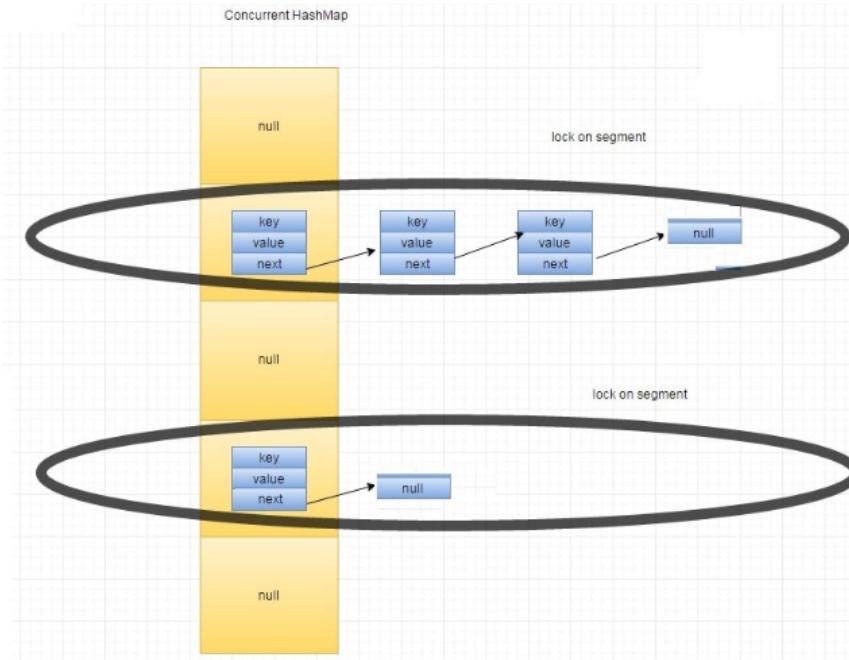


Thread safe DS java

SynchronizedMap-



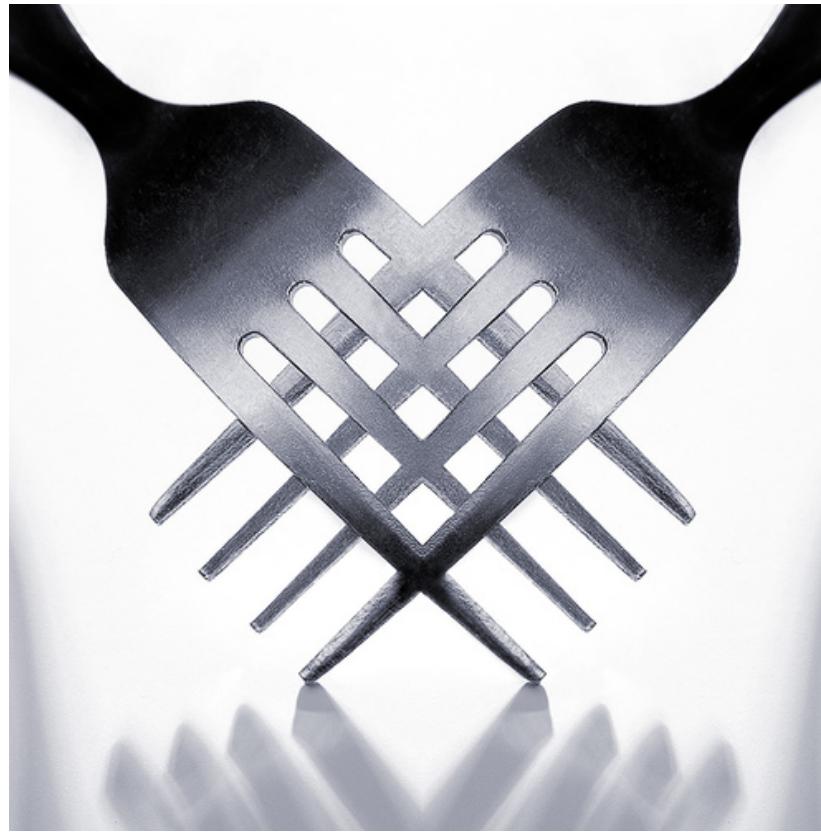
ConcurrentHashMap-



ExecutorService

Java's thread pool framework

Fork-Join framework



Fork Join Framework

Parallel version of Divide and Conquer

1. Recursively break down the problem into sub problems
2. Solve the sub problems in parallel
3. Combine the solutions to sub-problems to arrive at final result

General Form

```
// PSEUDOCODE
Result solve(Problem problem) {
    if (problem.size < SEQUENTIAL_THRESHOLD)
        return solveSequentially(problem);
    else {
        Result left, right;
        INVOKE-IN-PARALLEL {
            left = solve(extractLeftHalf(problem));
            right = solve(extractRightHalf(problem));
        }
        return combine(left, right);
    }
}
```



Java library for Fork-Join framework

Started with JSR 166 efforts

JDK 7 includes it in `java.util.concurrent`

Can be used with JDK 6.

- Download jsr166y package maintained by Doug Lea
- <http://gee.cs.oswego.edu/dl/concurrency-interest/>



Java library for Fork-Join framework

Started with JSR 166 efforts

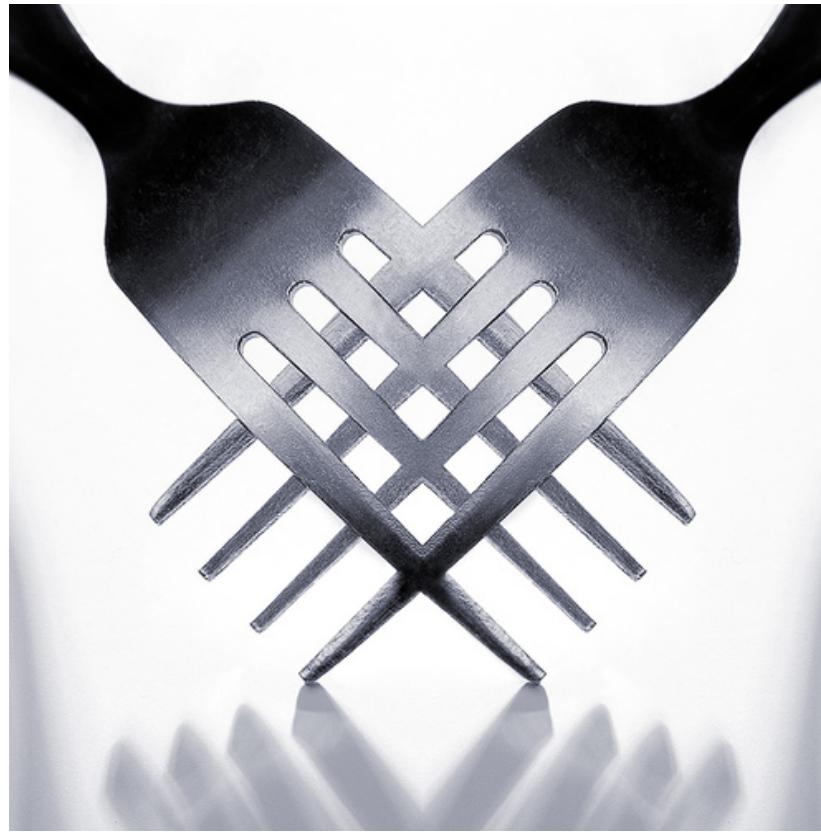
JDK 7 includes it in `java.util.concurrent`

Can be used with JDK 6.

- Download jsr166y package maintained by Doug Lea
- <http://gee.cs.oswego.edu/dl/concurrency-interest/>



Fork-Join framework



Selecting a max number

```
public class SelectMaxProblem {  
    private final int[] numbers;  
    private final int start;  
    private final int end;  
    public final int size;  
  
    public SelectMaxProblem(int[] numbers2, int i, int j) {  
        this.numbers = numbers2;  
        this.start = i;  
        this.end = j;  
        this.size = j - i;  
        System.out.println("start:" + start + ",end:" + end + ",size:" + size);  
    }  
  
    public int solveSequentially() {  
        int max = Integer.MIN_VALUE;  
        for (int i=start; i<end; i++) {  
            int n = numbers[i];  
            if (n > max)  
                max = n;  
        }  
        System.out.println("returning max:" + max);  
        return max;  
    }  
  
    public SelectMaxProblem subproblem(int subStart, int subEnd) {  
        return new SelectMaxProblem(numbers, start + subStart,  
                                    start + subEnd);  
    }  
}
```

Sequential algorithm



Selecting max with Fork-Join framework

```
13 public class MaxWithForkJoin extends RecursiveAction {  
14     private final int threshold;  
15     private final SelectMaxProblem problem;  
16     public long result;  
17  
18     public MaxWithForkJoin(SelectMaxProblem problem, int threshold) {  
19         this.problem = problem;  
20         this.threshold = threshold;  
21     }  
22  
23     @Override  
24     protected void compute() {  
25  
26         if (problem.size < threshold) {  
27             result = problem.solveSequentially();  
28         }  
29         else {  
30             int midpoint = problem.size / 2;  
31  
32             MaxWithForkJoin left = new MaxWithForkJoin(problem.subproblem(0, midpoint), threshold);  
33             MaxWithForkJoin right = new MaxWithForkJoin(problem.subproblem(midpoint + 1, problem.size), threshold);  
34  
35             invokeAll(left, right);  
36  
37             result = Math.max(left.result, right.result);  
38         }  
39     }  
40  
41     }  
42  
43 }  
^
```

A Fork-Join Task

: Solve sequentially if problem is small

Otherwise Create sub tasks & Fork

Join the results

Fork-Join framework implementation

Basic threads (Thread.start() , Thread.join())

- May require more threads than VM can support

Conventional thread pools

- Could run into thread starvation deadlock as fork join tasks spend much of the time waiting for other tasks

ForkJoinPool is an optimized thread pool executor (for fork-join tasks)



Fork-Join framework implementation

Basic threads (Thread.start() , Thread.join())

- May require more threads than VM can support

Conventional thread pools

- Could run into thread starvation deadlock as fork join tasks spend much of the time waiting for other tasks

ForkJoinPool is an optimized thread pool executor (for fork-join tasks)



ForkJoinTask as is could improve performance...

However some boilerplate work is still needed. Is there a better way?

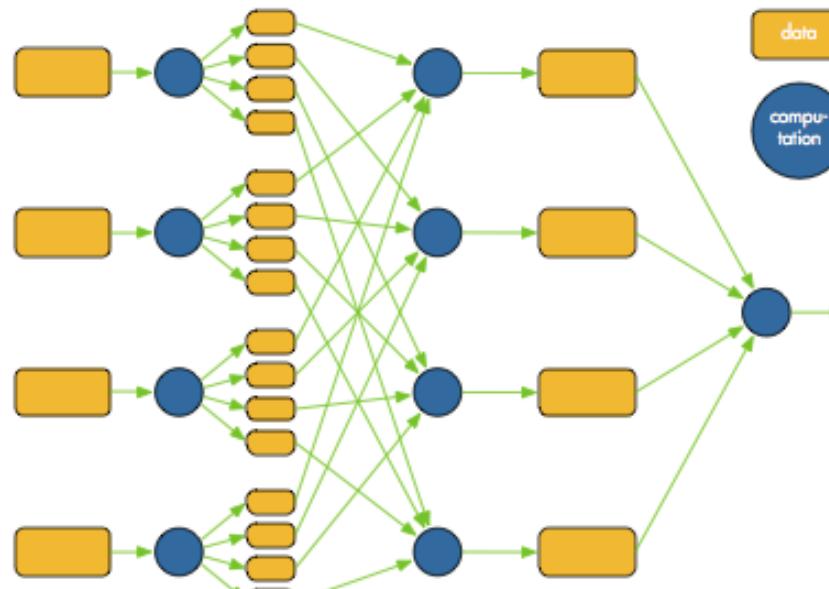
i.e. declarative specification of parallelization



Options for Transparent Parallelism

ParallelArray

- ForkJoinPool with an array
- Versions for primitives and objects
- Provide parallel aggregate operations
- Can be used on JDK 6+ with extra166y package from <http://gee.cs.oswego.edu/dl/concurrency-interest/>



Example: ParallelArray Operations

```
30 ForkJoinPool forkJoinPool = new ForkJoinPool(Runtime.getRuntime().availableProcessors());
31 SLAData[] testData = getTestData(arrayLength);
32 ParallelArray<SLAData> slaData = ParallelArray.createUsingHandoff(testData, forkJoinPool);
33
34 Predicate<SLAData> june2013 = new Predicate<SLAData>() {
35     public boolean op(SLAData s) {
36         return s.year == 2013 && s.month == 6;
37     }
38 };
39
40 Predicate<SLAData> peakHour = new Predicate<SLAData>() {
41     public boolean op(SLAData s) {
42         return s.hour == 14;
43     }
44 };
45 Predicate<SLAData> deleteCommand = new Predicate<SLAData>() {
46     public boolean op(SLAData s) {
47         return s.command.equalsIgnoreCase("DEL-COMMAND");
48     }
49 };
50 Ops.ObjectToDouble<SLAData> selectResponseTime = new Ops.ObjectToDouble<SLAData>() {
51     public double op(SLAData s) {
52         return s.responseTime;
53     }
54 };
55
56 ParallelDoubleArray slaDataProcessed = slaData.withFilter(june2013)
57     .withFilter(peakHour )
58     .withFilter(deleteCommand)
59     .withMapping(selectResponseTime )
60     .all();
61 double average = slaDataProcessed.sum() / slaDataProcessed.size();
62
63 System.out.println("June 2013 DEL-COMMAND Average response time during peak hours:" + average );
64
65
66 double june2012PeakHourDeleteMaxResponseTime = slaData.withFilter(june2013)
67     .withFilter(deleteCommand)
68     .withMapping(selectResponseTime )
69     .max();
70
71 System.out.println("June 2013 DEL-COMMAND Max response time:" + june2012PeakHourDeleteMaxResponseTime);
```

ParallelArray
created with
ForkJoinPool

A filter
operation

A map
operation

SQL like..
Batches
operations in a
single parallel
step

Select max is
a single step

Lambda's make life easier..

```
Predicate<SLAData> june2013 = new Predicate<SLAData>() {
    public boolean op(SLAData s) {
        return s.year == 2013 && s.month == 6;
    }
};

Predicate<SLAData> peakHour = new Predicate<SLAData>() {
    public boolean op(SLAData s) {
        return s.hour == 14;
    }
};
Predicate<SLAData> deleteCommand = new Predicate<SLAData>() {
    public boolean op(SLAData s) {
        return s.command.equalsIgnoreCase("DEL-COMMAND");
    }
};
Ops.ObjectToDouble<SLAData> selectResponseTime = new Ops.ObjectToDouble<SLAData>() {
    public double op(SLAData s) {
        return s.responseTime;
    }
};

ParallelDoubleArray slaDataProcessed = slaData.withFilter(june2013)
    .withFilter(peakHour )
    .withFilter(deleteCommand)
    .withMapping(selectResponseTime )
    .all();

double average = slaDataProcessed.sum() / slaDataProcessed.size();
```

With Lambda's

```
slaData.parallelStream().filter(s -> s.getYear() == 2013)
    .filter(s -> s.getMonth() == JUNE)
    .filter(s -> s.hour() == 14)
    .filter(s -> s.getCommand() == DELETE)
    .mapToInt(s -> s.getResponseTime())
    .sum();
```



Fork-Join and Map-Reduce

	Fork-Join	Map-Reduce
Processing on	Cores on a single compute node	Independent compute nodes
Division of tasks	Dynamic	Decided at start-up
Inter-task communication	Allowed	No
Task redistribution	Work-stealing	Speculative execution
When to use	Data size that can be handled in a node	Big Data
Speed-up	Good speed-up according to #cores, works with decent size data	Scales incredibly well for huge data sets

Source: <http://www.macs.hw.ac.uk/cs/techreps/docs/files/HW-MACS-TR-0096.pdf>

