# Java Concurrency / Multithreading Tutorial

# ===================================================

CPU: A central processing unit (CPU) is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control and input/output (I/O) operations specified by the instructions

Some computers employ a multi-core processor, which is a single chip containing two or more CPUs called "cores"

Array processors or vector processors have multiple processors that operate in parallel, with no unit considered central.

It is the brains of the computer.

Old days a computer had a single CPU, and was only capable of executing a single program at a time. Later came multitasking which meant that computers could execute multiple programs at the same time

Example- oil pump passing oil/gap alone at a time: single CPU

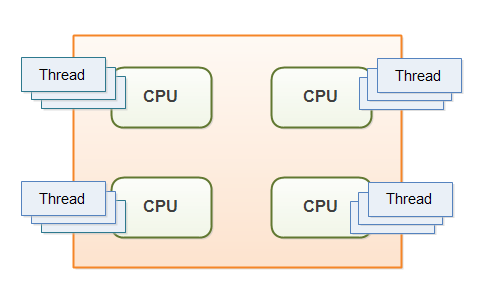
Multitasking: a bundle of pump passing oil, gas, water separately same time independently.

Multithreading: oil, gas, water supply together with gap of air independently but using same pump.

A thread of execution can be thought of as a CPU executing the program. When you have multiple threads executing the same program, it is like having multiple CPUs execute within the same program.

Multithreading can be a great way to increase the performance of some types of programs. However, mulithreading is even more challenging than multitasking. The threads are executing within the same program and are hence reading and writing the same memory simultanously. This can result in errors not seen in a singlethreaded program.

Modern computers, though, come with multi core CPUs, and even with multiple CPUs too. This means that separate threads can be executed by separate cores or CPUs simultanously.



If a thread reads a memory location while another thread writes to it, what value will the first thread end up reading? The old value? The value written by the second thread? Or a value that is a mix between the two? Or, if two threads are writing to the same memory location simultanously, what value will be left when they are done? The value written by the first thread? The value written by the second thread? Or a mix of the two values written?

Without proper precautions any of these outcomes are possible. The behaviour would not even be predictable. The outcome could change from time to time. Therefore it is important as a developer to know how to take the right precautions - meaning learning to control how threads access shared resources like memory, files, databases etc

A lot has happened in the world of concurrent architecture and design since the first Java concurrency books were written, and even since the Java 5 concurrency utilities were released.

# 

# Multithreading Benefits

=====================================================================

**Better resource utilization:-**

Imagine an application that reads and processes files from the local file system. Lets say that reading af file from disk takes 5 seconds and processing it takes 2 seconds. Processing two files then takes

5 seconds reading file A

2 seconds processing file A

5 seconds reading file B

2 seconds processing file B

-----------------------

14 seconds total

When reading the file from disk most of the CPU time is spent waiting for the disk to read the data. The CPU is pretty much idle during that time. It could be doing something else. By changing the order of the operations, the CPU could be better utilized. Look at this ordering:

5 seconds reading file A

5 seconds reading file B + 2 seconds processing file A

2 seconds processing file B

-----------------------

12 seconds total

The CPU waits for the first file to be read. Then it starts the read of the second file. While the second file is being read, the CPU processes the first file.

**More responsive programs**

while(server is active){

listen for request

process request

}

while(server is active){

listen for request

hand request to worker thread

}

# Multithreading Costs

===========================================================

Going from a singlethreaded to a multithreaded application doesn't just provide benefits. It also has some costs

You should have a good idea that the benefits gained by doing so, are larger than the costs. When in doubt, try measuring the performance or responsiveness of the application, instead of just guessing.

## **More complex design**

Code executed by multiple threads accessing shared data need special attention. Thread interaction is far from always simple. Errors arising from incorrect thread synchronization can be very hard to detect, reproduce and fix.

## Context Switching Overhead

When a CPU switches from executing one thread to executing another, the CPU needs to save the local data, program pointer etc. of the current thread, and load the local data, program pointer etc. of the next thread to execute. This switch is called a "context switch". The CPU switches from executing in the context of one thread to executing in the context of another.

Context switching isn't cheap. You don't want to switch between threads more than necessary.

## **Increased Resource Consumption**

A thread needs some resources from the computer in order to run. Besides CPU time a thread needs some memory to keep its local stack. It may also take up some resources inside the operating system needed to manage the thread. Try creating a program that creates 100 threads that does nothing but wait, and see how much memory the application takes when running.

# Concurrency Models

=====================================================================================

A *concurrency model*specifies how threads in the the system collaborate to complete the jobs they are are given. Different concurrency models split the jobs in different ways, and the threads may communicate and collaborate in different ways.

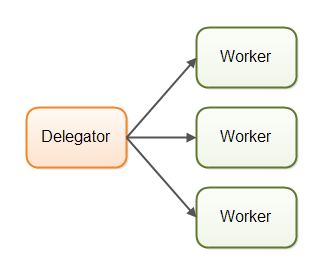
## **Concurrency Models and Distributed System Similarities**

In a concurrent system different threads communicate with each other. In a distributed system different processes communicate with each other (possibly on different computers). Threads and processes are quite similar to each other in nature. That is why the different concurrency models often look similar to different distributed system architectures.

Because concurrency models are similar to distributed system architectures, they can often borrow ideas from each other. For instance, models for distributing work among workers (threads) are often similar to models of [**load balancing in distributed systems**](http://tutorials.jenkov.com/software-architecture/load-balancing.html). The same is true of error handling techniques like logging, fail-over, idempotency of jobs etc.

## **Parallel Workers**

he first concurrency model is what I call the *parallel worker* model. Incoming jobs are assigned to different workers. Here is a diagram illustrating the parallel worker concurrency model:



In the parallel worker concurrency model a delegator distributes the incoming jobs to different workers. Each worker completes the full job. The workers work in parallel, running in different threads, and possibly on different CPUs.

he parallel worker concurrency model is the most commonly used concurrency model in Java applications (although that is changing). Many of the concurrency utilities in the **[java.util.concurrent Java package](http://tutorials.jenkov.com/java-util-concurrent/index.html)** are designed for use with this model.

## **Parallel Workers Advantages**

## Parallel Workers Advantages

Check which employee work faster and better quality