Concurrency Programming

It's when there is a software/hardware system that can perform more operations at the same time. It is the opposit of sequential execution. There are then some differences between programming a sequential program and a concurrent/parallel program.

Multi-tasking

Some time ago, not having an OS did mean that you could use only one program at a time that needed to be written directly for the software, because it couldn't be virtualized. Resources were also limited.

As a result, *multi tasking operatying system* were invented.

Multitasking is an illusion of concurrent execution, but if you alternate tasks very fast it works very well.

Process

A process is a unit of execution at the OS level.

- Every process executes in isolation. Each application has its own processes
- OS reserves necessary resources for the process (memory, files, …)

Processes can communicate (inter-process communication) with the use of sockets, signal handlers, semaphores, files, … . However this type of communication its difficult, so *Thread* were invented.

Threads

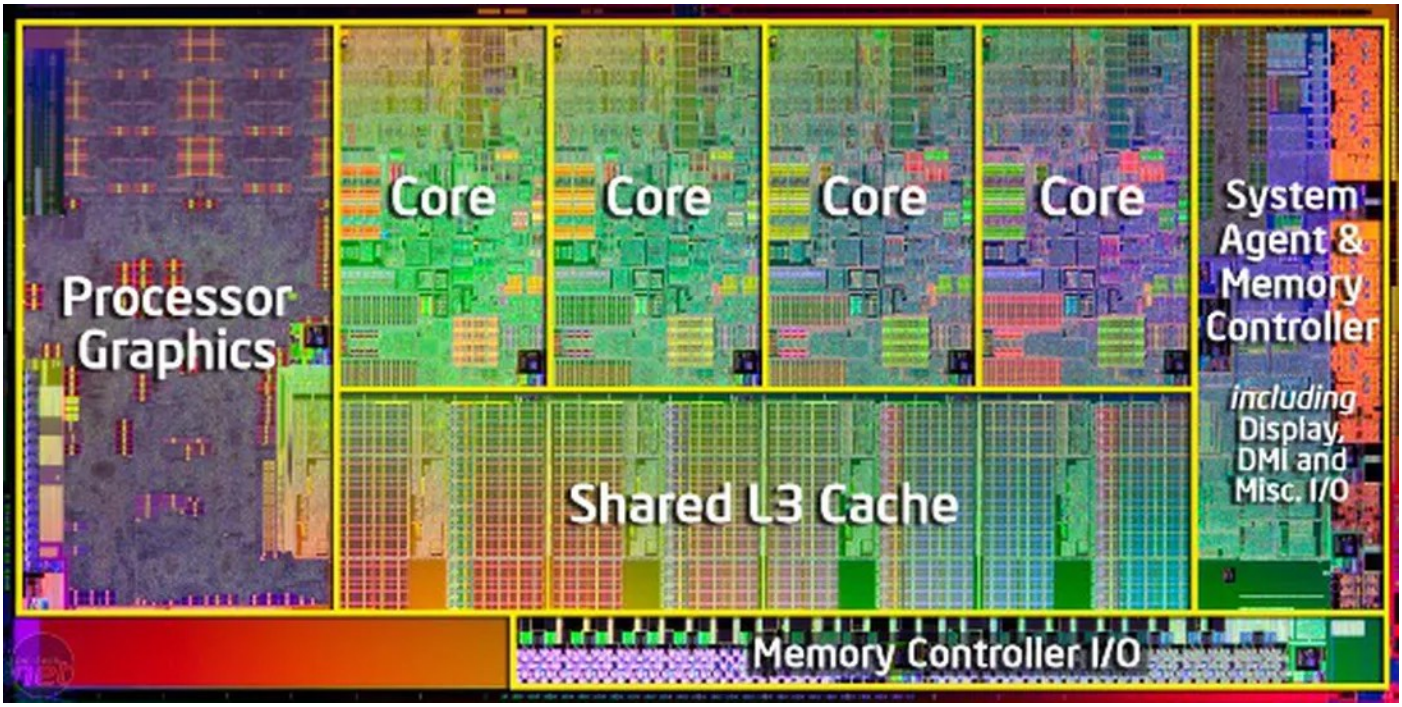
Threads are multiple concurrent streams of program control flow: each application run on a single thread but can create additional ones. Their *execution is simultaneous and asynchronous*.

Properties:

- Has program counter, stack and registers
- Shares process resources: heap and global variables
- Communication between threads is very fast: shared heap memory (static) and global variables

Why multi-core processors

Due to consumption and heat-dissipation problems, it is very difficult to further increase clock-rate speed of processors. Some solutions: *Hyper-threading, Multi-core CPUs, GPUs, SoCs (System on Chip)*.

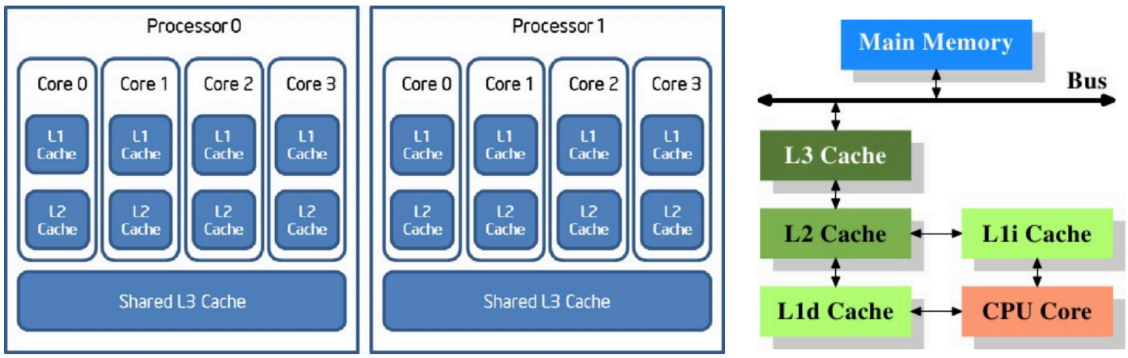


Pipeline and computation units

Instruction pipelining and prefetching, Branch prediction, Speculative exectuion, Out-of-order execution, ..

Caches

We use caches because they are narrower to the processor, so they're faster.



Processors are faster than central memory (where data and programs are stored), so we use caches because they are up to 100 times faster and they can keep up with processor speed.

Unpredictable Execution or Access

We don't know if the program we wrote will be optimized by the processor: how will it access data, if it will be stopped somewhere, how will it be divide into threads, ecc… In fact, there is a risk of random interleaving operations that must be taken into account when programming.

We also don't know how they will access to the memory: it may cause problem when accessing together or when accessing at different times and missing values.

Thread safety problems

Without correct synchronization of threads, problems may occour. Software solutions must be adopted for development of concurrent programs: they allow to decide the concurrent task that compose the application and to manage them to void concurrent access to memory. *Java* can provide those tools.

Concurrent execution with Java

Java provides many features since Java 5. Its instructions are abstract (high level), made to simplify development of concurrent applications so you don't need to know the low-level details (OS and CPU). Java run a Main process but can create *new processes* with the **ProcessBuilder class.** To specify the behavior of threads, we use the classic *sequential programming model* to simply code threads.

Example

```
class SimpleThread extends Thread {
    private int countDown = 5;

    public SimpleThread(int threadCount) {
        super("" + threadCount);
    }

    public String toString() {
        return "#" + getName() + ": " + countDown;
    }

    public void run() {
        while (true) {
            System.out.println(this);
            if (--countDown == 0)
            return;
        }
    }
}

public class TestSimpleThread {
    public static void main(String[] args) {
        for (int i = 0; i < 5; i++)
            new SimpleThread(i).start();
    }
}
```

run() is called with .start()

What happens to the main thread?

1 main thread + 5 threads = 6 total threas ongoing

thread : print

```
#1: 5
#1: 4
#3: 5
#2: 5
#2: 4
#2: 3
#2: 2
#2: 1
#4: 5
#3: 4
#1: 3
#3: 3
#3: 2
#4: 4
#5: 5
#4: 3
#3: 1
#1: 2
#1: 1
#4: 2
#5: 4
#4: 1
#5: 3
#5: 2
#5: 1
```

Calling thread1.start( ) DON'T immediatly start a thread, but it gives the OS the request to run that thread, we don't know when it will actually start!

An application will stop ONLY when ALL non-deamon threads are completed.

JAVA TOOLS

Runnable interface versus Thread interface

A completed thread cannot be restarted, a Runnable can. A Runnable can be run either with a Thread or a Executor. It is possible to implement a thread logic that extends another base class.

How to create

By extending the Thread class (or the Runnable interface):

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Running Task #1");
    }
}
Thread thread1 = new MyThread();
thread1.start();
```

Using an anonymous inner class:

```
Thread thread2 = new Thread(new Runnable() {
    public void run() {
        System.out.println("Running Task #2");
    }
});
thread2.start();
```

Using a lambda expression:

```
Thread thread3 = new Thread(() -> System.out.println("Running Task #3"));
thread3.start();
```

Some useful functions

- Thread.sleep(milliseconds);
- anotherThread.join( ); *puts current thread in wait mode until "anotherThread" finish*

anonymous inner class

```
public class JoinExample {
    public static void main(String[] args) {
        final Thread[] threads = new Thread[10];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(new Runnable() {
                public void run() {
                    System.out.println("Thread started");
                    try {
                        Thread.sleep((long) Math.random() * 1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println("Thread completed");
                }
            });
            threads[i].start();
        }
        for (int i = 0; i < threads.length; i++) {
            try {
                threads[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Thread all completed");
    }
}
```

!

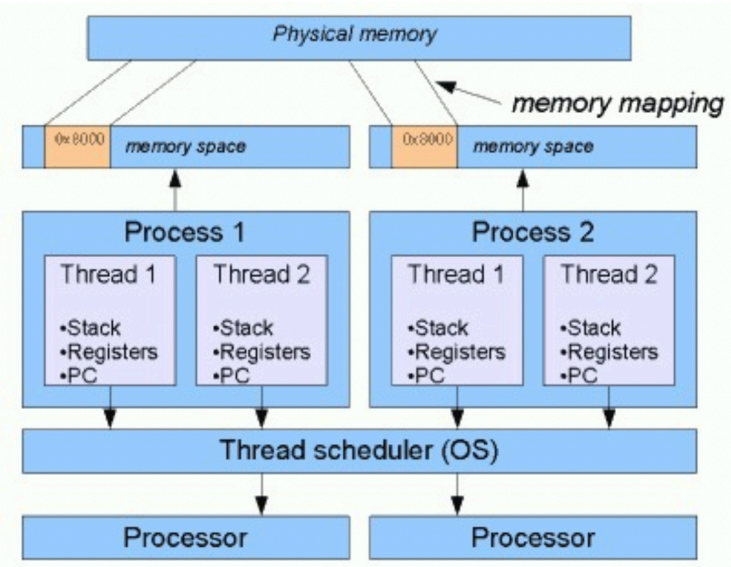
```
Thread started
Thread started
Thread started
Thread started
Thread started
Thread started
Thread completed
Thread completed
Thread completed
Thread started
Thread completed
Thread completed
Thread started
Thread completed
Thread completed
Thread completed
Thread started
Thread completed
Thread started
Thread completed
Thread all completed
```

Thread Safety

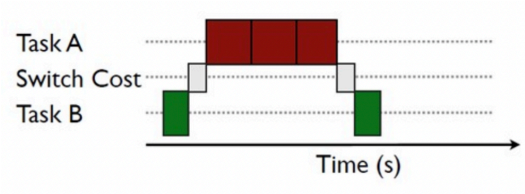
Thread safe means that is functionally correct, no programming errors and executing correctly when running on a multicore processor. A single-thread application is always thread safe.

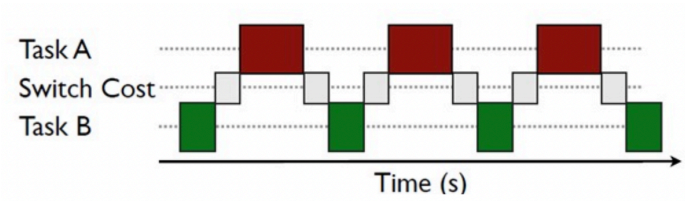
Context Switch

The OS schedule threads and it has also the freedom to stop our threads whenever it wants. So we need to program those in a way that this doesn't create errors in our program. Context-switching is the operation of saving the state of *saving and restoring the state of the thread* in order to correctly resume the execution from the same exact point. Context-switch its an expensive operation because it takes time to save and retrieve those data from the caches.



Context Switching of 2 threads





It takes more time to continuosly switching between threads executions.

MonoThread vs Multithread

In a mono thread program there will always be just one actor accessing to memory, so all data wil be consinstent. When multithread, threads can be stopped at anytime so accessing memory could go wrong. Threads could also modify data in shared regions that can create dirty reads or writings to other threads.

Shared memory problem

var = var + 1 | var = var + 1

Thread 1		Thread 2
read		
task-switch	=>	read
		add
		write
add	<=	*task-switch*
write		

var++ it's not a 1-clock-tick operation. It needs *read, add* and *write*. If one of these operations it's stopped before finishing, it can stop before the writing occours and maybe another thread could modify that variable, leading to an inconsisten error at the end of the first thread operation. It's called "*Race Condition*".

```
class FibonacciCalculator implements Runnable {
    private static int count = 2;
    private FibonacciNumber number = null;

    public FibonacciCalculator(FibonacciNumber number) {
        this.number = number;
    }

    public void run() {
        count++;
        number.setNewValue(number.getPreviousValue() +
                           number.getCurrentValue());
        System.out.println("The " + count + " Fibonacci number is " +
                           number.getCurrentValue());
    }
}
```

```
class FibonacciNumber {
    private int previousValue = 1;
    private int currentValue = 1;

    public int getPreviousValue() {
        return previousValue;
    }

    public int getCurrentValue() {
        return currentValue;
    }

    public void setNewValue(int value) {
        previousValue = currentValue;
        currentValue = value;
    }
}
```

Race condition

Happens when the correctness of the result depends on the order of execution of threads. It may happens when threads operate on shared memory region. Threads could also be running for a long time without never colliding, but it is not a fact.

Synchronization of threads

When threads operate on shared memory region we need to synchronize their access, in particular, the heap space and global variables might contain shared data. We need to take care of variables and object that are shared and mutable (i.e. not for final object or variables).

Each thread has its own stack that is not shared. In contrast, data on heap and static variables are shared. We need then to verify if there are any objects/variables that is shared and mutable (i.e. if there is even just one writing operation!). We will use **MUTEXES**.

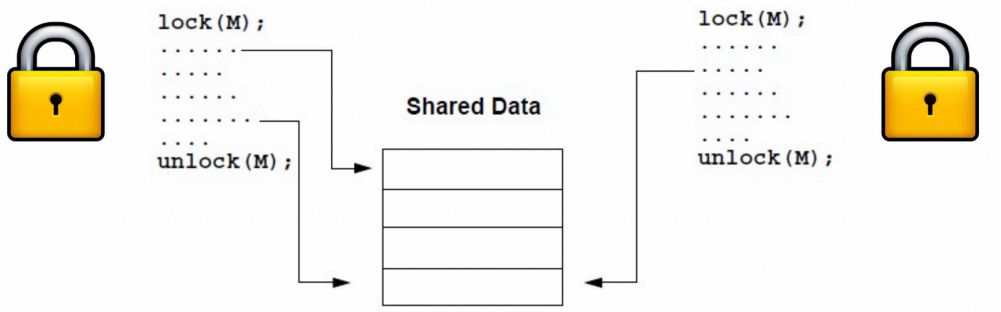
Analying the Fibonacci example…

The “number” variable it’s not shared because its *private*.
The “count” variable it’s shared because its *static* and it’s mutable (its modified with *count*++).

We also have “previousValue” and “currentValue” that are mutable.

Mutexes (mutual exclusion block)

When we use mutexes, only one thread at a time is allowed to own the lock and acces the protected code region. A thread can access a block only if it’s *green*. It then gain the access and puts *red*. Any other threads won’t succeed in accessing. The thread will then put *green* when it ends its operations.



I need to lock *both read and write operation* on each mutable or shared variable! Instead, I don’t need to implement any protection on non-mutable or non-shared variables.

Java tools

There are two ways to implement this approach: synchronized block and synchronized method.
We don’t use the protected variable as a lock!

Synchronized block:

> It has two parts. A reference to an object (our choice, even “new Object( )”) used as a lock and a code block to be protected by the block. The lock is autmatically activated when accessing and automatically deactivated when exiting. Works also with exception throws.

Synchronized method:

> It used the object in which the method its written as a lock. Its like using the synchronized block with “this”. In case of static methods, the class object it’s used as our lock.

Explicit locks:

> We instantiate a lock from *ReentrantLock*. We call *.lock( )* from our lock, then we create a try-finally codeblock where we put our operation, and in the finally we *.unlock( )* the lock. In this way we could also share a lock, or unlock / unlock in other methods instead where we are making the operations.

Lock vs RW-Locks

With the RW-Lock you tell the OS to permit multiple READINGS simultaneously and lock unlock-lock when a write need to be done. The normal lock instead permit only a single operation (reading or writing) for each locking time.

Locks and context-switches

Even if a Thread is into a portion of locked code, it is not prevented that the OS can't stop in a particular moment of that portion of code.

How to find portions of code that we need to protect?

The are some sequences of operations executed by a thread that need to be run *from the first to the last* without any *external modifications of used variables* (e.g. by other threads)

The main reason are dependencies between variables of multiple threads.

The reasoning

We imagine that a thread start a portion of code and simulate that at a certain point it stops and another thread start the same portion of code (or another one) and we see if something bad happens.

Compound actions

Compound actions are sequences of istructions that need to execute atomically to be thread-safe. Compound actions A and B are *atomic* if a thread can only execute A when B has not yet executed or when B has already completely executed (and vice versa).
An action is atomic if it is atomic in relation to all *other actions* that share the same program state (variables/fields), i*ncluding itself.*
If a compound action is not performed atomically there is a *vulnerability window* where one of the two variables was modified and the other one no. Modification can then be lost, being overwritten earlier than required leading to a possible inconsistent execution.

Race conditions

Check-then-act: a (potential wrong) read is used to take a decision on how the program executes (example: lazy initialization).

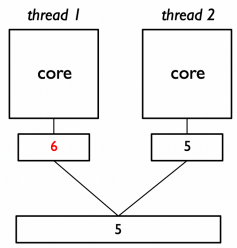
Read-modifiy-write: the state of an object is modified based on its (potentially wrong) previous value (example: ++var).

Volatile variable

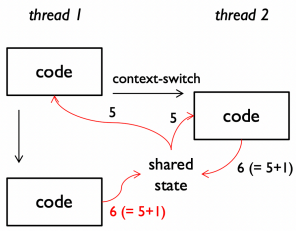
Scrivere cosa signifcano!

You don't want to use both *volatile* variables and *synchronized/locks* together, that's useless.

Thread Safety



Visibility problem

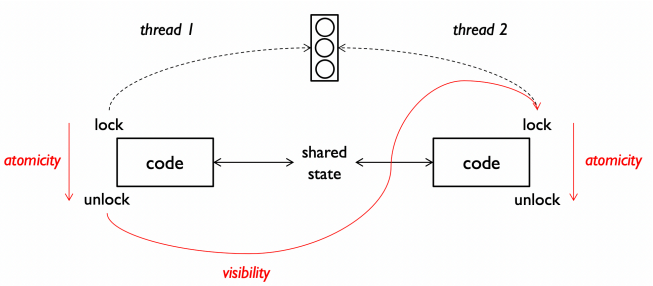


Atomicity problem

Everything that concerns memoy management and synchronization is descriped in the *memory model* in java. The memory model defines if a execution of a program is a legal one. It defines also which values *can be read in any instance*.

The memory model can guarantee the *atomocity* of the *fetch-and-store* operation execpt for *long* and *double* variables. It's guaranteed that without synchronization *at least one old value is seen (out-of-thin-air safety).* But this is not a solution for our problems.

Mutexes can be used to write a correct multi-thread application, synchronizing all threads on a lock.




```
class FibonacciCalculator implements Runnable {
    private static int count = 2;
    private FibonacciNumber number = null;

    public FibonacciCalculator(FibonacciNumber number) {
        this.number = number;
    }

    @Override
    public void run() {
        synchronized (number) {
            count++;
            number.setNewValue(number.getPreviousValue()
                               + number.getCurrentValue());
            System.out.println("The " + count + " Fibonacci number is "
                               + number.getCurrentValue());
        }
    }
}
```

Visibility?!

All visibility problems on all variables are solved by using the same lock by all threads!

Variables and Shared memory

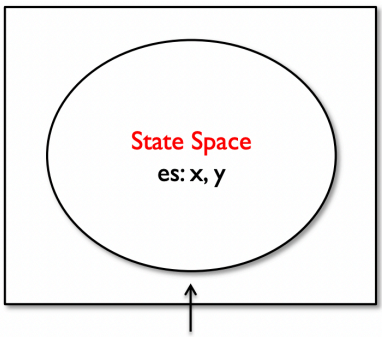
Local variables and parameters are on the stack, no problem. Also *final static variables* are ok, because they are constants.
Instead, static (non final) variables are a problem because acessible and shared, also like primitive types fields if shared between objects.

State Space

Every object has its own state space, that is the space of all the consistent states in which the object can be. The state-space is delimited by the rules that avoid the object to reach an inconsistent state called *invariants.*

Invariants

Invariants have to be established by the constructors and have to be preserved by the methods.



Invariant

example:

0 <= x < 1000 &&

0 <= y < 10 &&

x * y != 5000

Example

```
public class NumberRange {
    // INVARIANT: lower <= upper
    private volatile int lower = 0;
    private volatile int upper = 0;

    public void setLower(int i) {
        if (i > upper)
            throw new IllegalArgumentException("can't set lower to " + i + " > upper");
        lower = i;
    }

    public void setUpper(int i) {
        if (i < lower)
            throw new IllegalArgumentException("can't set upper to " + i + " < lower");
        upper = i;
    }

    public boolean isInRange(int i) {
        return (i >= lower && i <= upper);
    }
}
```

We don't know how many threads will use this class in the future. We don't have (in this case) visibiity problem because the variables are *volatile*.

```java
public final class Planet {

    /** Final primitive data is always immutable. */
    private final double fMass;

    /** An immutable object field. */
    private final String fName;

    /** A mutable object field. */
    private final Date fDateOfDiscovery;

    public Planet(final double aMass, final String aName, final long discoveryTime) {
        fMass = aMass;
        fName = aName;
        fDateOfDiscovery = new Date(discoveryTime);
    }
    ...
```

Immutability

Everything that is read-only is thread-safe (no visibility problem, no race condition, not mutable). The shortcut is using the *final* keyword. A final field can be only assigned once, must be initialized before the end of the constructor. *For reference type only the reference is final, not the content*.

If a class if *final* in cannot be extenden from other classes.

In this example we assign all final values in the constructor. We create a new object for the *Date variable* because if we directly assign an instance of the object from the outside, only the reference its final but that object outside could be modified! In this way, we create our object with our reference that only we have under control.

Easiest threadsafe method

Put synchronized on each method and the variables set private.

```java
public class SynchronizationPatternExample {
    private String name = null;
    private boolean nameModified = false;

    public synchronized String getName() {
        return name;
    }

    public synchronized boolean isNameModified() {
        return nameModified;
    }

    public synchronized void unsetNameModified() {
        this.nameModified = false;
    }

    public synchronized void setName(final String name) {
        this.name = name;
        nameModified = true;
    }
}
```

Onw writer many readers