Paradigmas de Computação Paralela

Concurrent/Parallel Programming in OO /Java

João Luís Ferreira Sobral

jls@...

Specification of concurrency/parallelism

- **Benefits from concurrent programming**
  - Programs that require multiple activities
  - Active objects in real world
  - Better service availability
  - Supports asynchronous message/invocation
  - **Take advantage of parallelism on multi-core / multi-CPU systems**
  - Required concurrency (certain Java classes execute concurrently, ex. Swing, applet, beans)

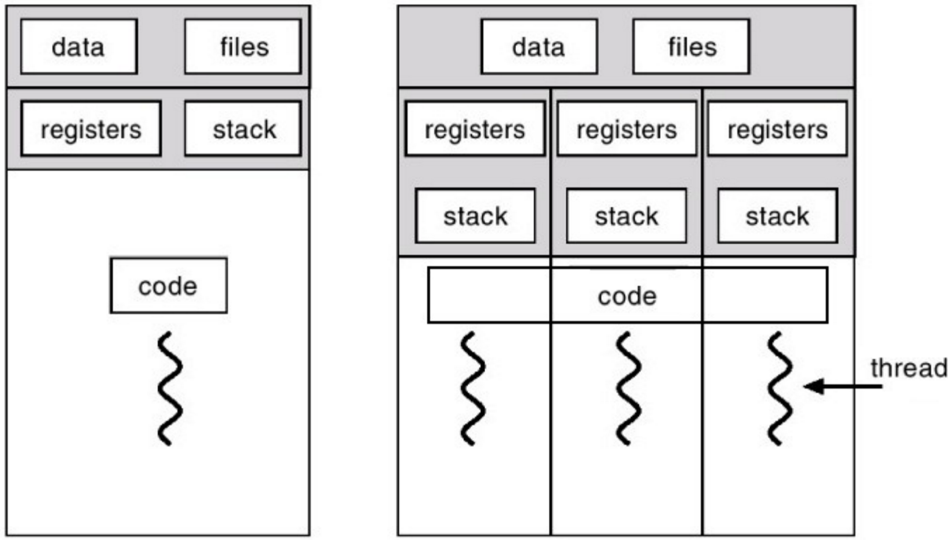
- **Concurrent/parallel activities: concepts**
  - Tasks versus Thread versus Process
  - Parallelism: logic versus physical
  - Pre-emption
  - Scheduling and priorities

Specification of concurrency/parallelism

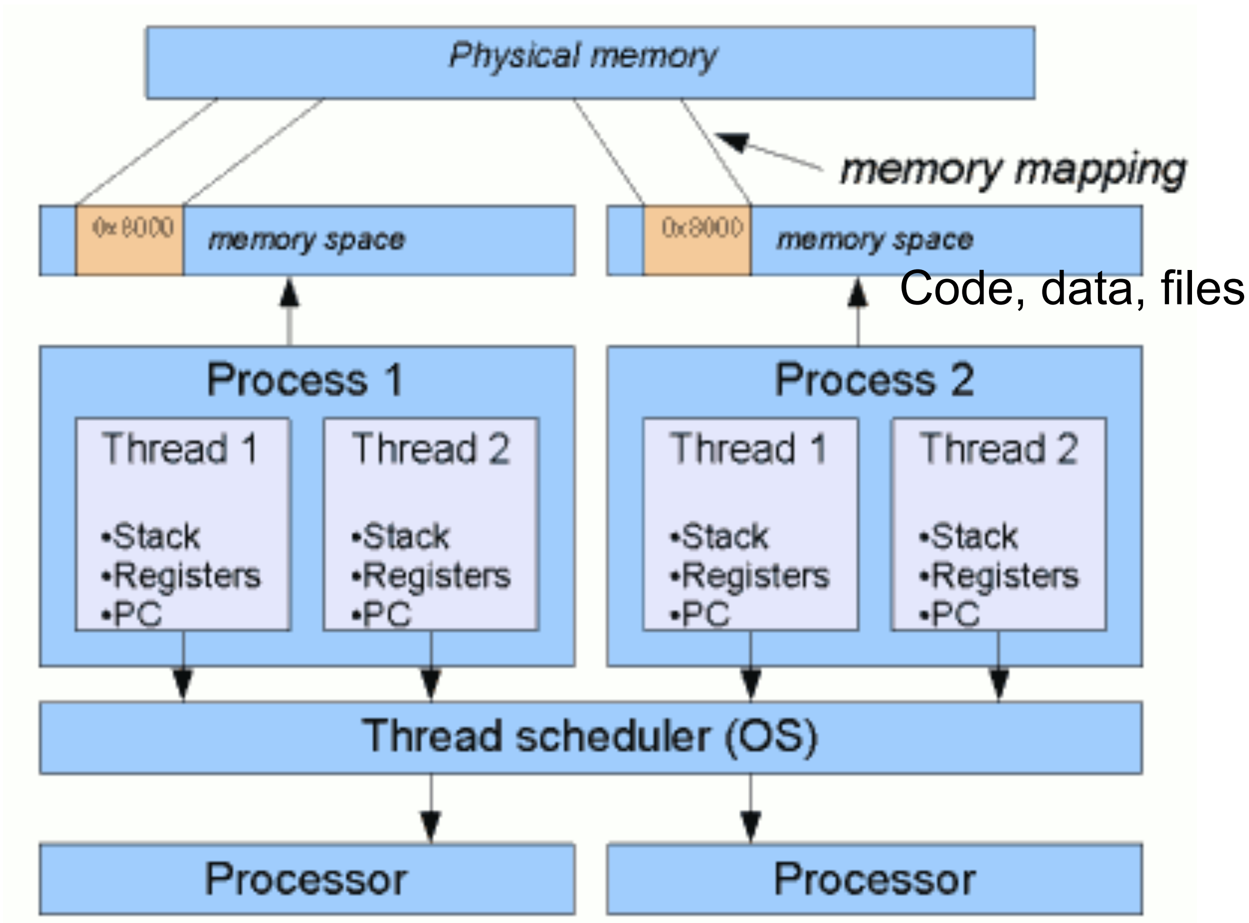
- **Processes**
  - Used for unrelated tasks
    - (e.g., a program)
  - Own address space
    - Address space is proteded from other process
  - Swithching at the kernel level

- **Threads**
  - Are part from the same job
  - Share address space, code, data and files
  - Swithching at the user or kernel level

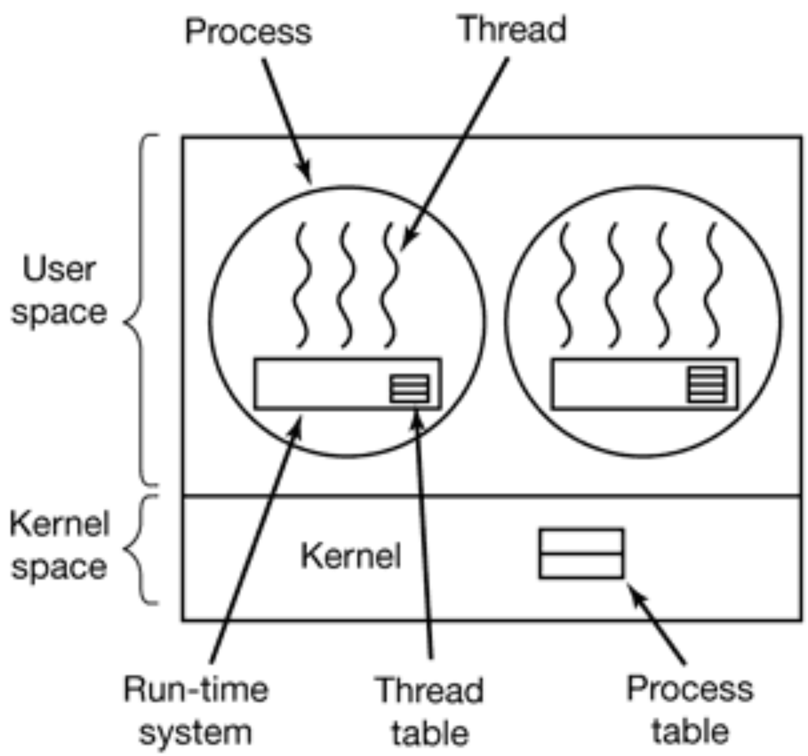
Every process has at lest one thread



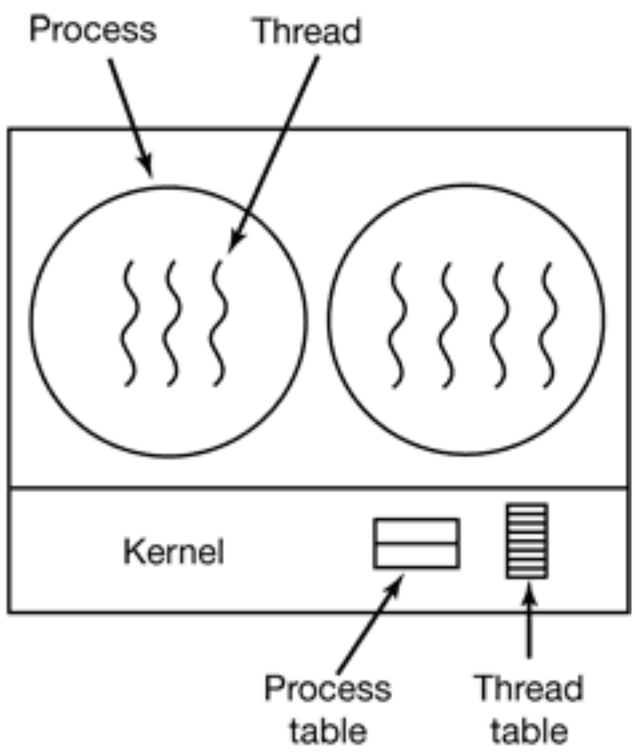
Thread vs Process



Process/Thread scheduling



user-level thread scheduling

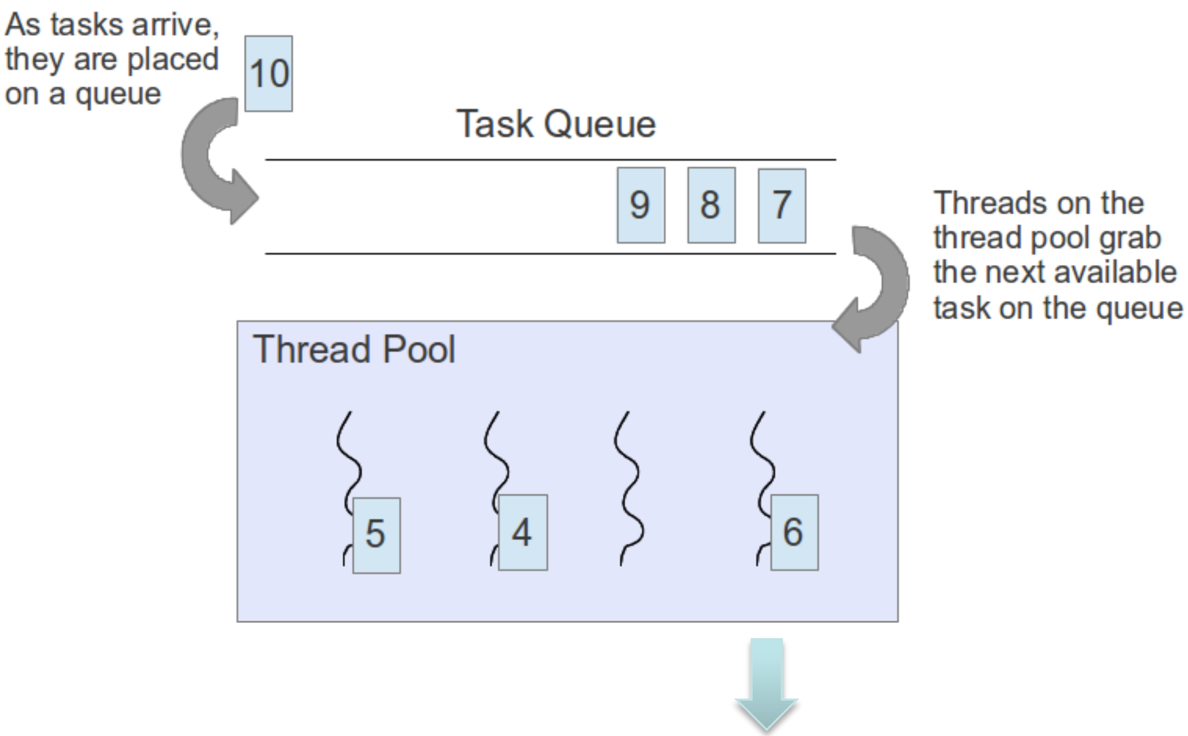


kernel-level thread scheduling

Process/Thread vs Tasks

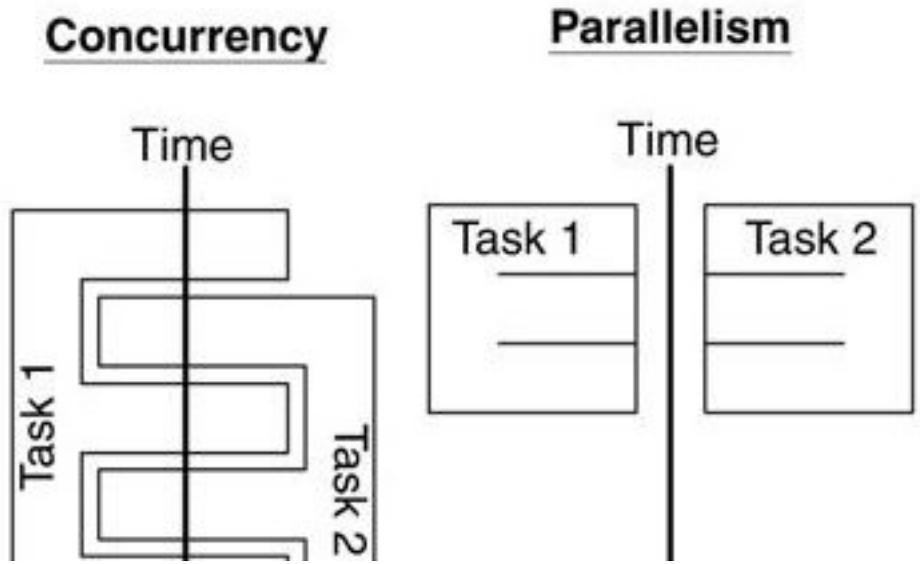
- **Task**: sequence of instructions
- **Thread/process**: execution context for a task
- **Processor/core**: hardware that runs a thread/process

In Java
- Runnable object
- Thread
- Processor core

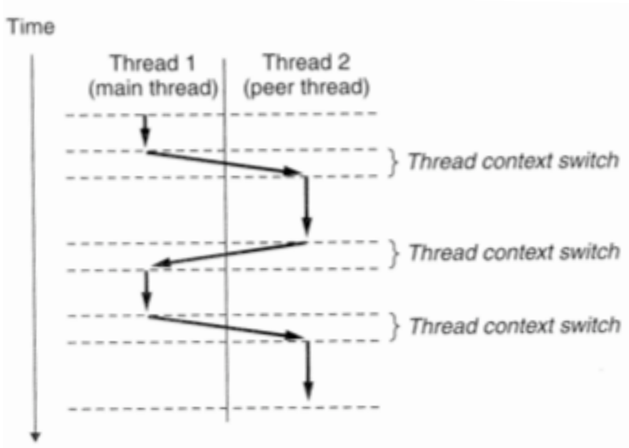


Threads are scheduled on available cores

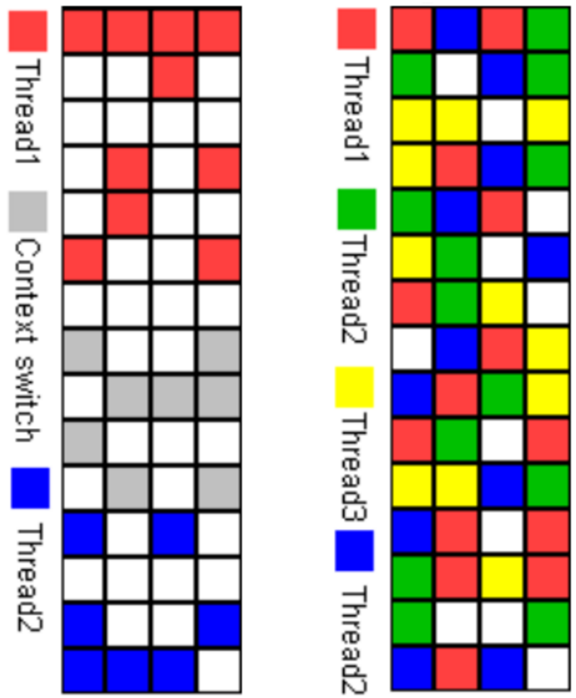
Logic vs physical parallelism



Pre-emption



Intel SMT



Advantages/disadvantages of threads

- **Benefits from threads**
  - Shared variables!
  - Easy communications between tasks/contexts
    - Multiple threads coordinate their execution and share data through reading and writing shared variables
- **Problems**
  - Hidden dependencies are hard to debug
    - Shared variables may be updated by other threads
  - Performance prediction
- **OOP to the rescue**
  - Object encapsulation to support threading
  - Classes control access to shared data via synchronization

Specification of concurrency/parallelism

- **Problems introduced by concurrent programming**

  - **safety** - inconsistencies in the execution of programs
  - **liveness** - deadlocks in the execution of programs
  - introduces non-determinism in program execution
  - in OO systems there are fewer objects than asynchronous activities
  - not useful for local execution of methods in a model of call / response
  - introduces overhead due to the creation, scheduling and synchronization of threads

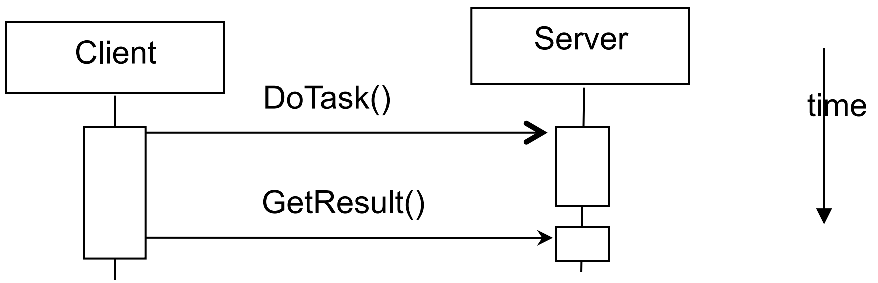
- **Concurrency in traditional approaches**

  - Models based on *fork/join*, *cobegin/coend*, and *parfor*
    - Synchronization is done using semaphores, barriers or monitors
  - Active process (CSP)
    - Makes processing through an active body, interacting through message passing:
      - blocking synchronous, synchronous non-blocking or asynchronous

Specification of concurrency/parallelism

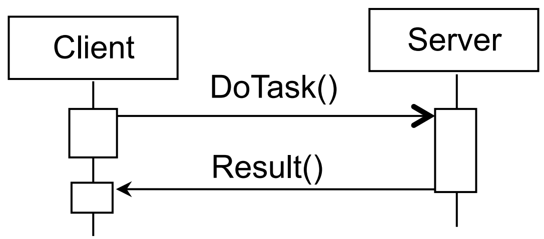
- **Concurrency in object oriented applications**
  - **Synchronous invocations (traditional models)**
    - The client is blocked while the method is executed by the server, even if there is no return value
  - **Asynchronous invocations with no return value (one way)**
    - When the invoked method does not return a value the client can continue running simultaneously with the execution of the method on the server.
  - **Asynchronous invocations with return value**
    - When there is a return value, the invocation can also be asynchronous
    - There are three alternatives to get the return result:

Synchronous deferred - The client makes a second invocation of the server to obtain the result

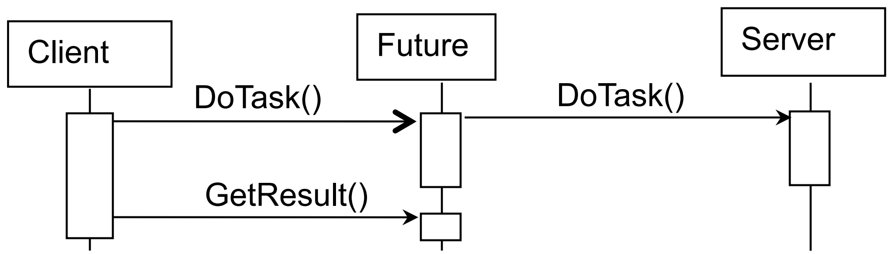


Specification of concurrency/parallelism

- **Concurrency in object oriented applications (cont.)**
  - **Asynchronous invocations with return value** (cont)
    - **With callback** - The server performs an invocation of a predefined method of the client when the task completes



    - **With future -** The invocation is delegated to another object that stores the result



Concurrent Programming in Java

- Java was one of the first languages with support for concurrent programming
- Interface ***Runnable***
  - Must be implemented by classes to be executed by a thread
  - Method run() contains the code to be executed

```
interface Runnable {
    public void run();
}
```

- Class *java.lang.Thread*: (also implements the Runnable interface)
  - Thread() or Thread(Runnable r);// class constructor
  - start(); // creates a thread and invokes r.run()
  - join(); // waits for thread completion
  - sleep(int ms); // suspends the thread
  - setPriority(int Priority); // changes thread priority

Concurrent Programming in Java

- **Example** (simpler option)
  - Two threads increment their own counter

```
public class Cont extends Thread { // implicit: implements Runnable
        public Cont() { }
        public void run() {
            for (int i=0; i<100; i++)
                        System.out.println(Thread.currentThread() + " i= " + i);
        }
}
```

  - **Sequential execution:**

```
...
new Cont().run();
new Cont().run();
...
```

  - **Parallel execution (fork&join model):**

```
...
Thread t1 = new Cont();
Thread t2 = new Cont();
t1.start();  // fork
t2.start();  // fork
... // or t2.run();
 t1.join(); // wait for the end of t1 execution
```

Concurrent Programming in Java

- **Example** (more flexible alternative)
  - Two threads increment their own counter

```
public class Cont implements Runnable {
    public Cont() { }
    public void run() {
        for (int i=0; i<100; i++)  System.out.println(" i= " + i);
    }
}
```

- **Sequential execution:**

```
...
new Cont().run();
new Cont().run();
...  // join
```

- **Parallel execution:**

```
...
Cont c1 = new Cont();
Cont c2 = new Cont();
Thread t1 = new Thread(c1);
Thread t2 = new Thread(c2);
t1.start();
t2.start();
... // t1.join, to wait for the end of execution
```

Concurrent Programming (in Java)

- Security - nothing bad should happen in a program
- Liveness - something good must happen in a program

- Example of lack of security:
  - Execution of method inc() by two threads simultaneously can lead to a inconsistent value of variable ct

```
public class Cont {
    protected long ct;
    public Cont() { ct=0; }
    public void inc() { ct = ct + 1; }
}
```

Thread 1		Thread 2
ct = ct +1		ct = ct +1
Load $R1,ct		Load $R2,ct
Inc $R1	ct?	Inc $R2
Store $R1,ct		Store $R2,ct

Concurrent Programming (in Java)

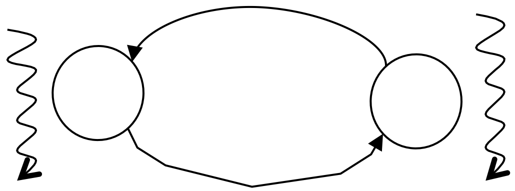
- Specification of synchronization (increases security)
  - Blocks of code and synchronized methods *(mutex)*
    - synchronized method() { ... } / / method has exclusive access to the object
    - synchronized(oneObj) { ... } / / Gets exclusive access to oneObj
  - Java memory model
    - A thread of execution can keep local copies of values. Synchronized blocks ensure that all threads "see" consistent values
  - With monitors (implemented by the Object class)
    - wait () - wait for access to the monitor
    - wait (int timeout) - wait, with timing
    - notify () - wakes up a thread waiting for access
    - notifyAll () - wakes up all threads waiting

Concurrent Programming (in Java)

- Example of a lack of liveness (deadlock):
  - Execution of method *inc ()* with two threads simultaneously on objects with cross-references

Obj1/Thread1:

```
                    ...
synchronized void inc() {
                    obj2.inc()
}
```



Obj2/Thread2:

```
                    ...
synchronized void inc() {
                    obj1.inc()
}
```

Concurrent Programming (in Java)

- Patterns to improve safety
  - Stateless or immutable objects (e.g. String class)

```
public int[] sort(int[] arr) {
        int[] copy = arr.clone();  // local copy
        …  //  sort
        return(copy);
}
```

  - Objects enclosed in other objects

- Patterns to improve liveness
  - Methods that only read the object state usually do not need be synchronized (except double and long)
  - No need to synchronize the variables that are written only once:

```
void setEnd() { end = True; }
```

Concurrent Programming (in Java)

- Patterns to improve liveness (cont.)
  - Separated synchronization to access to parts of the state (or divide the state into two objects)

```
Class twoPoints {
    Point p1, p2;
    public void movexp1(int x) {
            synchronized (p1) {    p1.movex(x);  }
    }
    public void movexp2(int x) {
            synchronized (p2) {    p2.movex(x);  }
    }
}
```

  - Resources should be accessed by the same order

```
public void update() {
    synchronized(obj1) {
        synchronized(obj2) {
                ...  // do update
        }
    }
}
```

Asynchronous method invocation in Java

- With no return value
  - Implemented through the pattern command, where the command is executed in parallel with the client. The command parameters are passed in the constructor
  - Example: Writing data to file in background - activated by the client:

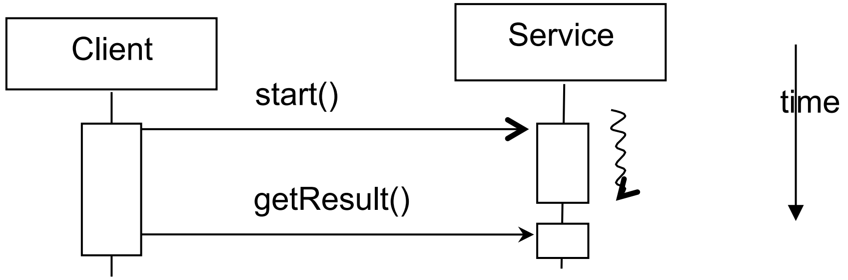
```
public class FileWriter extends Thread {
    private String nm;
    private byte[] d;
    public FileWriter(String n, byte data[]) {
        nm = n;
        d = data;
    }
    public void run() {
        writeBytes(nm,d);
    }
}
```

```
// client code
(new FileWriter("Pic",rawPicture)).start();
```

Asynchronous method invocation in Java

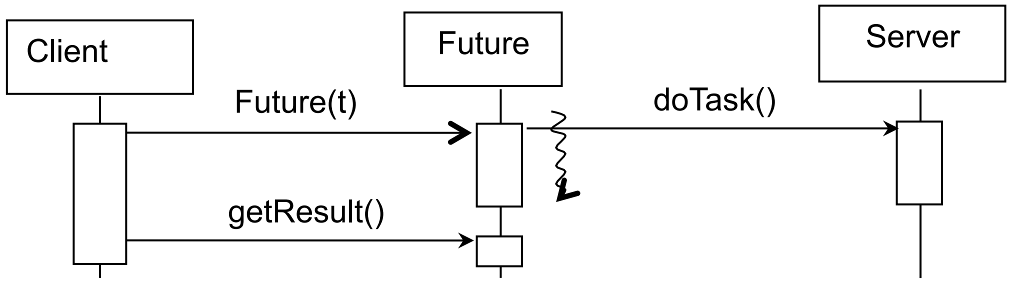
- Synchronous deferred
  - Using the method Thread.join()

```
r = new Service().start();
..  // doWork();
r.join();
r.getResult();
```



- Future
  - The future will contain the result of the operation and blocks the client if the vaue is requested it is available

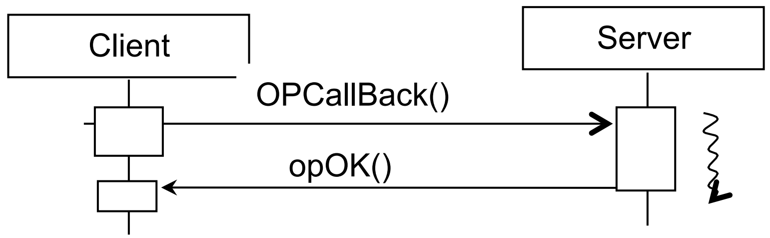
```
class Future extends Thread {
    private Task tk=null;
    public Future(Task tsk) {
        tk = tsk;
        start();
    }
    public Task getResult() {
        join();
        return(tk);

    public void run() { tk = doTask();  } // do task
}
```




```
// client code
Future f = new Future(task);
… // do other work
 f.getResult();
```

Asynchronous method invocation in Java

- Callback




```
public interface Client {
    public void opOK(Task);
}

class OPCallBack extends Thread {
    private Client cl=null;
    private Task tk=null;
    public OPCallBack(Task tsk, Client clk) {
            tk = tsk;
            cl = clk
            start();
    }
    public run() {
            tk = doTask(tk);
            cl.opOK(tk);   // callback
    }
}
```

Extensions in Java 5

- Executors (Thread Pool)

```
void Executor.execute(Runnable task)    // Thread Pool

// (new Thread(r)).start(); becomes e.execute(r)

Future<T> Executor.submit(Callable<T> task)
```

```
interface Future<V> {
    V get();
}

interface Callable<V> {
    V call();
}
```

- High performance locks: (ReentrantLock, Condition, ReadWriteLock)

```
Lock l = ...;
l.lock();
try {
        // access the resource protected by this lock
 } finally {
        l.unlock();
}
```

```
interface Lock {
   lock();
   tryLock();
   unlock();
}
```

- Generic classes for synchronization: semaphores, mutexes, barriers, latches, and exchangers

- Concurrent collections: ConcurrentHashMap, BlockingQueue

- Atomic variables: java.util.concurrent.atomic

Extensions in Java 7/8

- Lambda expressions can replace *Runnable* and *Callable* interfaces
  - Avoids the overhead of creating a class and of passing parameters and returning a value
  - Syntax:

    p1 [, p2, p3 …] -> { body statement }

  - Example:

    Person -> { Person.getAge() > 18; }

- Steams use lambda functions to express parallel operations on collections

```
int sum = widgets.parallelStream()
                    .filter(b -> b.getColor() == RED)
                    .mapToInt(b -> b.getWeight())
                    .sum();
```

- New executor: forkJoinPool (Java 7)

	Call from non-fork/join clients	**Call from within fork/join computations**
Arrange async execution	**execute(ForkJoinTask)**	**ForkJoinTask.fork()**
Await and obtain result	**invoke(ForkJoinTask)**	**ForkJoinTask.invoke()**
Arrange exec and obtain Future	**submit(ForkJoinTask)**	**ForkJoinTask.fork() (ForkJoinTasks are Futures)**