Introduction

In this first part, we introduce basic terms to get a common understanding.

Inhalt

1. Concurrent and parallel Programming

The terms parallel and concurrent programming are used in contrast to sequential programming.

Informally, a **concurrent** program is one that does **more than** one thing **at a time**. For example, a **web browser** may be simultaneously perform a HTTP GET request to get an HTML page, playing an audio clip, displaying the number of bytes received of some image, and engaging in an dialog with a user.

However, these **parallel** appearing **activities** are sometimes an **illusion**: on some computers these different actions might indeed be performed by different CPUs; but on other systems they are performed by a **single time-shared CPU** (switching quickly enough that they appear to be simultaneous).

Considering Java let us come to a more precise definition:

> A Java virtual machine (**JVM**) and its underlying operating system provide **mappings** from **apparent simultaneity** to **physical parallelism** (via multiple CPUs), or lack thereof, by allowing **independent activities** to proceed in **parallel when possible** and otherwise by **timesharing**.
>
> **Concurrent programming** consists of **using programming constructs** that are mapped in this way.

Here, we will discuss the possibilities of Java to do concurrent programming. In addition, we consider how to connect more JVMs to realize distributed computing.

2. Concurrent applications

Concurrency and the reasons for employing it can be understand by considering the nature of a few common types of concurrent applications.

Web services

Most HTTP and XML based web services are multithreaded. Usually, the main motivation for supporting **multiple concurrent connections** is to ensure that new incoming requests do not need to wait out completion of others.

Number crunching

Many computation-intensive tasks can be parallelized. Thus, execution is more quickly if multiple CPUs are present.

Here, the goal is to maximize **throughput**.

I/O processing

Even on a one CPU sequential computer, **devices** that perform reads and writes on disks, wires, etc. **operate independently** of the **CPU**.

Concurrent programs can use the time otherwise waiting for slow I/O making more efficient use of a computers resources.

GUI-based applications

Most user interfaces are single-threaded; they often communicate with multithreaded services. Concurrency enables user control to stay responsive ever during time-consuming actions.

Component-based software

Large-granularity software (for example layout editors, ERP application) may internally construct threads in order to assist in blocking, providing multimedia support or improve performance.

Mobile code

Frameworks such as the `java.applet` package execute downloaded code in separate threads to be able to isolate, monitor and control the effects of unknown code.

Embedded systems

Most programs running on small dedicated devices perform real-time control. As defined in *The Java Language Specification*, the Java platform does not support hard real-time control in which correctness depends on actions performed by certain deadlines. But all JVM implementations support soft real-time control, in which timelines and performance are considered quality-of-service issues rather than correctness issues.

1. Concurrent Execution Constructs

They are several **constructs** available for **concurrently executing code**. In the following, we describe common approaches.

The idea is always to **map a new activity** to any of **several abstractions** along a granularity continuum **from computer-networks** to **singe CPU** lightweight execution frameworks.

1.1. Computer systems

In a **networked environment**, you might **map** each **logical unit of execution** to a **different computer**. Each computer can be a uniprocessor, a multiprocessor, or even a cluster of machines administered as a single unit, sharing a common operating system.

This mapping is typically applied for those aspects of systems that require a distributed solution.

We will consider this in the last unit of the lecture.

1.2. Processes

A process is an operating-system abstraction that allows one computer system to support many units of execution. Each process typically represents a separate running program, for example a word-processor or an executing JVM.

Operating systems guarantee some degree of **independence** and **security** among concurrent executing processes. **Processes** are **not allowed** to access **one another's storage** locations and must instead communicate via interprocess communication facilities such as pipes.

We will consider how such facilities are working in the second part of the lecture.

1.3.Threads

A thread is a **call sequence** that **executes independently** of others, while at the same time possibly **sharing** underlying system **resources**. A thread belongs to exactly one process that executes it.

Thread constructs differ in 3 kinds:

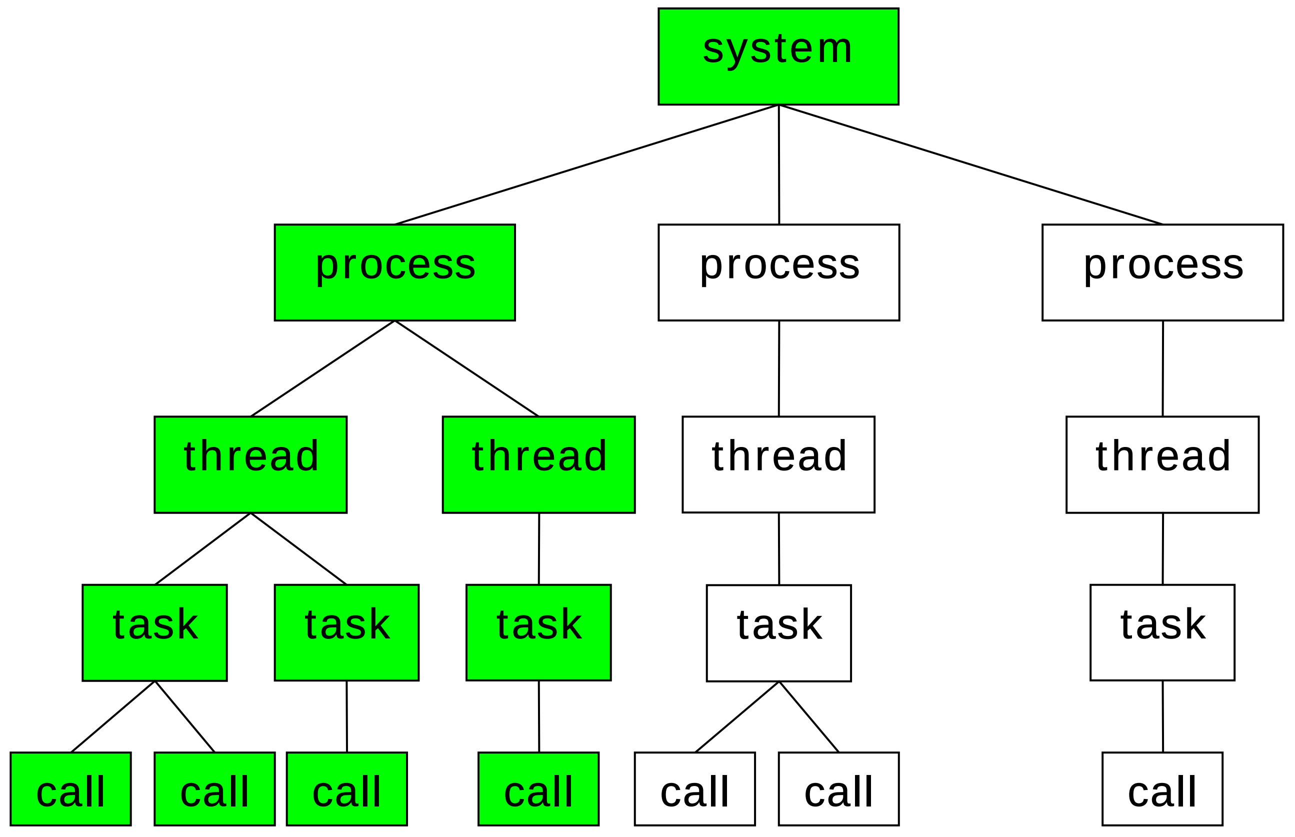
- **Sharing**:
  Threads may share access to the memory, open files and other resources associated with a single process.
  Threads in **Java** share **all** such resources.
  Some operating systems also support threads with some time more restrictions.

- **Scheduling**:
  At **one extreme**, all threads can be treated together as a single-threaded process, in which case they may cooperatively contend with each other so that **only one thread** is **running at a time**, without giving any other thread a chance to run until it blocks.
  At the **other extreme**, the underlying scheduler can allow **all threads** in a system to **contend** directly with each other.
  Threads in the **Java** programming language may be scheduled using **any policy** lying at or anywhere between these extremes.

- **Communication**:
  System interact via communication across wires (or wireless), for example using sockets.

Processes may also communicate in this way, but also use lighter mechanisms such as pipes.
Threads can use all these options, plus other cheaper mechanisms like using the same memory locations.

These considerations let us come to the following picture:



We will consider an environment, where different Java threads will run into a process of the JVM into a computer system which supports Java.

2. Concurrency and Object Oriented Programming

Objects and **concurrency** have been linked since the earliest days of each. The first concurrent OO programming language **Simula** (circa 1966) was also the first OO language. Concurrency was based around coroutines – thread-like constructs where the programmer explicitly had to control the activities.

Several other languages providing OO and concurrency followed (e.g. Ada, C++ with concurrency support library classes).

We will use java, an OO language with build in features for concurrent programming.

EoC