Introduction to Concurrency

Why do we need concurrency?

- Physics, engineering, electronics, biology, …
- Multi-processor systems
- SISD (single instruction, single data)
- MIMD (multiple instruction, multiple data)
- Networked computing nodes
- Standalone computing nodes – including local buses & interfaces
- Operating systems (& distributed operating systems)
- Pipelined processors
- … to employ / design concurrent parts of computer architectures
- Sequential programming delivers some inherent concurrency
- But we need to add a number of further crucial concepts
- Distributed entities
- Operating systems are often preferred over a single high-performance CPU

Does concurrency lead to chaos?

- No interaction between concurrent system parts means that we can
  assume that all parallel parts execute independently (i.e. they are not
  affected by each other)
- Performance may depend on the order of execution
- Events may be non-deterministic
- … to handle different execution scenarios

The concurrent programming abstraction

1. What appears sequential on a higher abstraction level, is
   usually concurrent at a lower abstraction level
   a. e.g. Concurrent operating systems, hardware components
   b. Which might not behave at higher programming level
2. What appears to be concurrent on a higher abstraction level,
   might be sequential at a lower abstraction level
   a. e.g. High-level processing systems
   b. Which are accessed on a single, parallel computing node

Working definitions

- Literally: concurrent means:
  - Temporarily existing in space, as parallel lines; going on side by side; appearing together
- Technically: concurrent is usually defined negatively as:
  - Events or circumstances, existing or arising together or at the same time
  - Events or circumstances that occur at the same time or at the same point in time

Forms of concurrency

- Overlapped I/O and
  - Multiple sequential programs
  - Multi-processing
  - Multi-tasking

A computer scientist’s view on concurrency

- Overlapping I/O (and computation)
- Multi-processing
- Multi-tasking
- Multi-processing systems
- Distributed operating systems
- General network architectures

Concurrency on different abstraction levels/perspectives

- Networks
  - Large scale, high-bandwidth interconnected nodes ("supercomputers")
- Distributed computer architectures
  - Multi-processing systems
- Operating systems (& distributed operating systems)
- Hardware components
- Software components
- … to be able to manage a system's execution
- … to be able to design computer systems that are part of a distributed system
- … to be able to design distributed systems
- … to employ / design concurrent parts of computer architectures

Models and Terminology

The concurrent programming abstraction

- "Concurrency" is technically defined negatively as:
  - If there is no observer who can identify two events as being in
    the same temporal sequence (i.e. one event has fully terminated before
    the other one started), then these two events are considered concurrent.
- "Concurrency" in the context of programming and logic:
  - "Concurrent programming abstraction is the study of
    instruction execution sequences in the atomic instructions of sequential processes."
The concurrent programming abstraction

- No interaction between concurrent system parts means that we can analyze them individually as pure sequential programs, and
- Interaction access or forms of:
  - Contention (implicit interaction)
  - Multiple concurrent execution units compete for shared resource
  - Communication (explicit interaction)
- Explicit passing of information and/or explicit synchronization

Atomic operations:
- Processes: threads in concurrent systems rely on the assumptions of
  - Atomic operations (discussed elsewhere)
- Complete and potential atomic operations are the cornerstone of
  - Complete system correctness proofs
- Atomic operations are essentially indivisible, but may need to
  - Complete system correctness proofs in practice
- Atomic operations can be used to implement non-atomic

Introduction to processes and threads

Processes:
- Processes can be regarded as a group of threads, which
  - Share resources but run concurrently
- Processes can be seen as a group of threads, which
  - Share resources but run concurrently

Symmetric Multiprocessing (SMP)

SMPs share the same physical hardware (CPU and memory)
- Any process that starts at the same time
- All processes share the same physical hardware (CPU and memory)
- SMPs share the same physical hardware (CPU and memory)
- SMPs share the same physical hardware (CPU and memory)

Processes
- Processes are independent units of execution and
- They can be seen as threads that run concurrently
- Process State
- Process Id
- Scheduling info
- Priorities, deadlines, consumed CPU-time, ...

T CPU per control-flow

Specific configuration:
- e.g.:
- Physical process control systems
- T CPU per task, tasks connected to
- Physical management (SMP) not required
- Scheduling info available
- Scheduling info available

T CPU for all control-flows

- SMPs include: CPU, CPUs, functional or kernel-level
- SMPs include: CPU, CPUs, functional or kernel-level
- SMPs include: CPU, CPUs, functional or kernel-level

Liveness properties

- Constraints on the input set,
- Constraints on the input set,
- Constraints on the input set,
Introduction to Concurrency

Process states

created: the task is ready to run, but not yet considered for dispatch or signaling for admission
waiting for admission: holds a CPU and suspension
ready: ready to run
waiting for main memory: waiting for admission
waiting for a resource: waiting for admission
pre-emption: the task is ready to run, but not considered for dispatch or signaling for admission
ready, susp.: ready to run
running: holds a CPU and terminates
unblock: the task is ready to run, but not considered for dispatch or signaling for admission
ready, suspended: ready to run

Languages with implicit concurrency: e.g. functional programming

Quicksort in a functional language (here: Haskell):

```haskell
qsort [] = []
qsort [y | y <- xs, y < x] ++
qsort [y | y <- xs, y ≥ x]
```

Parameters can be evaluated independently, i.e. sub-expressions are evaluated concurrently, possibly before the main expression is started.

Some functional language allow for lazy evaluation, i.e. sub-expressions are evaluated concurrently, possibly before the main expression is started.

Concurrent programming languages

Language candidates

- **Explicit concurrency**: Ada, Chapel, C++, Rust
- **Implicit general concurrency**: Java, Go, Python, C#, C, C++, Lua, Lua, Rust, Swift, Kotlin, Scala, C++, C#, Rust, Swift, Kotlin, Scala
- **Parallelism**: Python, Java, C++, C#, Rust, Swift, Kotlin, Scala
- **Parallelism and concurrency**: Python, Java, C++, C#, Rust, Swift, Kotlin, Scala
- **Support for communication**: messaging, shared memory, ...