

# *Concurrency – The Basic Concepts*

# *What is concurrency?*

- Literally ‘concurrent’ means:
  - *Adj.:* Running together in space, as parallel lines; going on side by side, as proceedings; occurring together, as events or circumstances; existing or arising together; conjoint, associated [Oxfords English Dictionary]
- Technically ‘concurrent’ is usually defined negatively as:
  - If there is no observer who can identify two events as being in strict temporal sequence (i.e. one event has fully terminated before the other one started) then these two events are considered concurrent.

# *Why do we need/have Concurrency?*

- Physics, engineering, electronics, biology, ...
  - basically *every* real world system is **concurrent!**
- Sequential processing is suggested by most kernel computer architectures
  - *but* almost all current processor architectures have **concurrent elements**
  - **and *most*** computer systems are part of a **concurrent network**
- Strict sequential processing is suggested by the most widely used programming languages
  - which is a reason why you might believe that concurrent computing is rare/exotic/hard
- Sequential programming delivers some *fundamental parts* for concurrent programming
  - ***but we need to add a number of further crucial concepts***

# *Why would a computer scientist consider concurrency?*

- To *be able* to connect computer systems with the real world
- To *be able* to employ / design concurrent parts of computer architectures
- To *construct* complex software packages (operating systems, compilers, databases, ...)
- To *understand* where sequential **and/or** concurrent programming is **required**
  - or where sequential or concurrent programming can be chosen freely
- to *enhance* the **reactivity** of a system

# *A computer scientist's view on concurrency*

- Overlapped I/O and computation
  - employ interrupt programming to handle I/O
- Multi-programming
  - allow multiple independent programs To be executed on one cpu
- Multi-tasking
  - allow multiple interacting processes to be executed on one cpu
- Multi-processor systems
  - add physical/real concurrency
- Parallel Machines & distributed operating systems
  - add (non-deterministic) communication channels
- General network architectures
  - allow for any form of communicating, distributed entities

# *Flynn's Taxonomy of Concurrent Systems*

**A classification of computer architectures, proposed by Michael J. Flynn in 1966:**

- **SISD** [single instruction, single data]
  - standard sequential processors
- **SIMD** [single instruction, multiple data]
  - vector processors
- **MISD** [multiple instruction, single data]
  - pipelines
- **MIMD** [multiple instruction, multiple data]
  - multiprocessors or computer networks

# *An engineer's view on concurrency*

- Multiple **physical, coupled, dynamical systems** form the actual environment and/or task at hand
  - In order to model and control such a system, its **inherent concurrency** needs to be considered
  - **Multiple less powerful processors** are often preferred over a single high-performance cpu
  - The system design of usually strictly **based on the structure of the given physical system.**

# *Does concurrency lead to chaos?*

- Concurrency often leads to the following features / issues / problems:
  - non-deterministic phenomena
  - non-observable system states
  - results may depend on more than just the input parameters and states at start time (timing, throughput, load, available resources, signals ... throughout the execution)
  - non-reproducibility + debugging?
- Meaningful employment of concurrent systems features:
  - non-determinism employed where the underlying system is non-deterministic
  - non-determinism employed where the actual execution sequence is meaningless
  - synchronization employed where adequate ... but only there
- Control & monitor where required (and do it right), but not more ...

# *Concurrency on different abstraction levels / perspectives*

- **Networks**
  - Multi-CPU network nodes and other specialized sub-networks
  - Single-CPU network nodes – still including buses & I/O sub-systems
  - Single-CPU
  - Operating systems (& distributed operating systems)
- **Processes & threads**
- **High-level concurrent programming**
- **Assembler level concurrent programming**
  - Individual concurrent units inside one CPU
  - Individual electronic circuits

# It's Not What it Might Appear!

- What appears *sequential* on a higher abstraction level, is usually *concurrent* at a lower abstraction level:
  - e.g. low-level concurrent I/O drivers, which might not be visible at a high programming level
- What appears *concurrent* on a higher abstraction level, might be *sequential* at a lower abstraction level:
  - e.g. Multi-processing systems, which are executed on a single, sequential CPU

# *Concurrent Programming Abstraction*

- Technically ‘concurrent’ is usually defined negatively as:
  - If there is no observer who can identify two events as being in strict temporal sequence (i.e. one event has fully terminated before the other one starts up), then these two events are considered concurrent.
- ‘Concurrent’ in the context of programming:
  - “Concurrent programming abstraction is the study of interleaved execution sequences of the atomic instructions of sequential processes.”
- See Ben-Ari

# Concurrent Program

Concurrent program ::=

Multiple sequential programs (processes)  
which are executed *simultaneously*

- Generally assumed that simultaneous execution means there is one execution unit (processor) per sequential program
  - even though this is usually not correct, it is an often valid assumption in the context of concurrent programming.

# Interaction Points

- No interaction between concurrent system parts means that we can analyse them individually as pure sequential programs.
- Interaction points:
  - *Contention*: multiple concurrent execution units compete for one shared resource
  - *Communication*: Explicit passing of information **and/or** synchronization
- The latter is a key part of this course

# *Time-line or Sequence?*

- Consider time (durations) explicitly:
  - Real-time systems (COMP4330)
- Consider the sequence of interaction points only:
  - Non-real-time systems (COMP2310)

# *Correctness of concurrent non-real-time systems [logical correctness]:*

- Does *not depend* on speeds / execution times / delays
- Does *not depend* on actual interleaving of concurrent processes [scheduler]
- **Does *depend* on all possible sequences of interaction points**

# *Correctness vs. testing in concurrent systems:*

- Slight changes in external triggers may (and usually will) result in complete different schedules (interleaving):
  - Concurrent programs which depend in any way on external influences **cannot be tested easily**
  - **Designs which are *provably correct*** with respect to the specification and are *independent of the actual timing behaviour* are essential.
- Some timing restrictions for the scheduling still persist in non-real-time systems, e.g. ‘fairness’

# *Atomic operations*

- Correctness proofs / designs in concurrent systems rely on the assumptions of

## **‘atomic operations’ [detailed discussion later]:**

- complex and powerful atomic operations ease the correctness proofs, but may limit flexibility in the design
- simple atomic operations are theoretically sufficient, but may lead to complex systems which correctness cannot be proven in practice.

# Standard concepts of correctness

- Partial correctness:

$$(P(I) \wedge \text{terminates}(\text{Program}(I, O))) \Rightarrow Q(I, O)$$

- Total correctness:

$$P(I) \Rightarrow (\text{terminates}(\text{Program}(I, O)) \wedge Q(I, O))$$

where  $I$  and  $O$  are input and output sets,  
 $P$  is a property on the input set,  
and  $Q$  is a relation between input and output sets

- Do these concepts apply to and are they sufficient for concurrent systems?

# Extended concepts of correctness in concurrent systems:

- Termination is often not intended or even considered a failure
- Safety properties:

$$(P(I) \wedge \text{Processors}(I, S)) \Rightarrow \Box Q(I, S)$$

where  $\Box Q$  means that  $Q$  does *always* hold

- Liveness properties:

$$(P(I) \wedge \text{Processors}(I, S)) \Rightarrow \Diamond Q(I, S)$$

where  $\Diamond Q$  means that  $Q$  does *eventually* hold (and will then stay true)  
and  $S$  is the current state of the concurrent system

# Safety properties

$$(P(I) \wedge \text{Processors}(I, S)) \Rightarrow \Box Q(I, S)$$

where  $\Box Q$  means that  $Q$  does *always* hold

- Examples:
  - Mutual exclusion (no resource collisions)
  - Absence of deadlocks (and other forms of ‘silent death’ and ‘freeze’ conditions)
  - Specified responsiveness or free capabilities (typical in real-time / embedded systems or server applications)

# Liveness properties

$$(P(I) \wedge \text{Processors}(I, S)) \Rightarrow \diamond Q(I, S)$$

where  $\diamond Q$  means that  $Q$  does *eventually* hold (and will then stay true)  
and  $S$  is the current state of the concurrent system

- **Examples:**
  - Requests need eventually to be completed
  - The state of the system needs eventually be displayed to the outside
  - No part of the system is to be delayed forever (fairness)
- Liveness properties can be extremely hard to prove