Forms of concurrency

What is concurrency?

Working definitions:

- **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 (Oxford 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, …
- Technically 'concurrent' is usually defined negatively as:
  - Strict sequential processing is suggested by widely used programming languages.
- Sequential processing delivers some fundamental components 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 be able to design (almost) current processor architectures with concurrent elements and most computer systems are part of a concurrent network.
- In order to model and control such a system, its temporal sequence (i.e. one event has fully terminated before the other one started) needs to be considered.
- Multiple physical, coupled, dynamical systems form the actual environment and/or task at hand
- The system design is usually strictly based on the structure of the given physical system.

A computer scientist's view on concurrency

- Overlapped I/O and computation
- Multi-programming
- Multi-tasking
- Multi-processor systems
- Parallel Machines & distributed operating systems
- General network architectures
- Multiple physical, coupled, dynamical systems form the actual environment and/or task at hand

A computer scientist's view on concurrency

- **SISD** (single instruction, single data)
- **MISD** (multiple instruction, single data)
- **SIMD** (single instruction, multiple data)
- **MIMD** (multiple instruction, multiple data)
- Vector processors
- Pipelined processors
- Multi-processors or computer networks
Introduction to Concurrency

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 (t) (throughput, load, available resources, signals... throughout the execution)
- non-reproducible or debugging!

Meantingful employment of concurrent systems features:

- non-determinism employed where the underlying system is non-deterministic
- synchronization employed where adequate ... but only there

Control & monitor where required (and do it right), but not more...

Introduction to Concurrency

Models and Terminology

The concurrent programming abstraction

1. What appears sequential on a higher abstraction level, is usually concurrent at a lower abstraction level:

   - e.g. Concurrent operating systems or hardware components, which might not be visible at a higher programming level.

2. What appears concurrent on a higher abstraction level, might be sequential at a lower abstraction level:

   - e.g. Multiprocessing system, which are executed on a single, sequential computing node.

The concurrent programming abstraction

- 'concurrent' is technically 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), then these two events are considered concurrent.

- 'concurrent' in the context of programming and logic:
  "Concurrent programming abstraction is the study of interleaved execution sequences of the atomic instructions of sequential processes." (Ben-Ari)

No interaction between concurrent system parts means that we can analyze them individually as pure sequential programs (end of course).

Concurrency on different abstraction levels/perspectives

- Networks
  - Large scale, high bandwidth interconnected nodes ("super computers")
  - Networked computing nodes

- Standalone computing nodes - including local buses & interfaces subsystems

- Operating systems (& distributed operating systems)

- Implicit concurrency
  - Explicit concurrent programming (message passing and synchronization)

- Assembler level concurrent programming
  - Individual concurrent units inside one CPU

- Individual electronics concepts

Multiple sequential programs (processes or threads) which are executed concurrently.

Contention (implicit interaction):

Multiple concurrent execution units compete for one shared resource.

Communication (explicit interaction):

Explicit passing of information and/or explicit synchronization.

Consider time (durations) explicitly:

- Real-time systems - join the appropriate courses

- Non-real-time systems - stay in your seat

Time-line or Sequence?
Introduction to Concurrency

Models and Terminology

The concurrent programming abstraction

Correctness of concurrent non-real-time systems

[logical correctness]:

- does not depend on clock speeds / execution times / delays
- does not depend on actual interleaving of concurrent processes
- holds true for all possible sequences of interaction points (interleavings)

Models and Terminology

The concurrent programming abstraction

Correctness vs. testing in concurrent systems:

- Slight changes in external triggers may (and usually does) result in completely different schedules (interleaving):
  - Concurrent programs which depend in any way on external influences cannot be tested without modeling and embedding these influences into the test process.
  - Designs which are provably correct with respect to the specification are independent of the actual timing behavior are essential.
  - Simple atomic operations are theoretically sufficient, but may lead to complex systems which correctness cannot be proven in practice.

Models and Terminology

The concurrent programming abstraction

Atomic operations:

- 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.

Extended concepts of correctness in concurrent systems:

- Partial correctness: \( P(I) \land \text{terminates}(\text{Program}(I,O)) \Rightarrow Q(I,O) \)
- Total correctness: \( P(I) \Rightarrow \text{terminates}(\text{Program}(I,O)) \land Q(I,O) \)

where \( I,O \) are input and output sets,
\( P \) is a property on the input set, and \( Q \) is a relation between input and output sets.

Extended concepts of correctness in concurrent systems:

- Safety properties: \( P(I) \land \text{Processes}(I,S) \Rightarrow Q(I,S) \)
  - \( Q(I,S) \) does always hold

- Liveness properties: \( P(I) \land \text{Processes}(I,S) \Rightarrow Q(I,S) \)
  - \( Q(I,S) \) eventually holds (and will then stay true) and \( S \) is the current state of the concurrent system

Examples of safety properties:

- Mutual exclusion (no resource collisions)
- Absence of deadlocks
- Specified responsiveness or free capabilities

Examples of liveness properties:

- Interesting liveness properties can be very hard to prove

Introduction to processes and threads

1 CPU per control-flow

Specific configurations:

- Distributed parallelism
- Physical process control systems
- 1 cpu per task, connected via a bus system
- Process management (scheduling) not required
- Shared memory access need to be coordinated

1 CPU for all control-flows

- OS emulate one CPU for each control-flow
- Multi-tasking operating system
- Support for memory protection essential
- Process management (scheduling) required
- Shared memory access need to be coordinated
Introduction to Concurrency

Processes

- Address space
- Control flows
- Kernel has full knowledge about all processes as well as their states, requirements and currently held resources.

Threads

- Individual control-flows can be handled:
  - Inside the OS:
    - Kernel scheduling
  - Outside the OS:
    - User-level scheduling
  - Threads may need to go through their parent process to access OS.

Symmetric Multiprocessing (SMP)

- All CPUs share the same physical address space (and access to resources).
- Any process/thread can be executed on any available CPU.

Processes -- Threads

- Threads can be considered as a group of processes, which share some resources (process hierarchy).
- Due to the overlap in resources, the threads attached to the task are less than the "first-class" or processes.
- Thread switching and inter-thread communication can be more efficient than switching on process level.
- Scheduling of threads depends on the actual thread implementations:
  - e.g. user-level control-flows, which the kernel has no knowledge about at all.
  - e.g. kernel-level control-flows, which are handled as processes with some restrictions.

Process Control Blocks (PCBs)

- Process Id
- Process state:
  - Created: ready to run, but not yet considered by any dispatcher
  - Running: CPU and executes
  - Blocked: not ready to run
  - Suspended: swapped out of main memory
- Scheduling attributes:
  - Priorities, deadlines, consumed CPU time
- CPU state:
  - Saved the saved information to different contexts
  - Memory base, limits, shared areas
- Memory attributes / privileges:
  - Memory base, limits, shared areas
- Allocated resources / privileges:
  - Memory (complete CPU state)
- Process state:
  - Created: ready to run, but not yet considered by any dispatcher
  - Running: CPU and executes
  - Blocked: not ready to run
  - Suspended: swapped out of main memory
- Scheduling info:
  - Process Id
  - Process state
  - Scheduling info
  - Saved registers
  - Memory spaces / privileges
  - Allocated resources / privileges

Process states:

- Created: ready to run, but not yet considered by any dispatcher
- Ready: ready to run
- Running: CPU and executes
- Blocked: not ready to run
- Suspended: swapped out of main memory
- Terminated: task is ready to run, but not yet considered by any dispatcher

PCBs (links thereof) are commonly enqueued at a certain state or condition (waiting for access or change in state).

Process switching and inter-thread communication can be independent modules.
In UNIX systems tasks are created by ‘cloning’

 pid = fork ();
 resulting in a duplication of the current process
 ... returning '0' to the newly created process (the child process)
 ... returning the process id of the child process to the creating process (the parent process)
 ... or returning '1' as C-style indication of a failure in void of actual exception handling

Frequently requested essential elements

pid = fork ();
if (fork () == 0) {
 close (data_pipe [1]);
 if (pipe (data_pipe) == -1) {
 perror ("pipe broken");
 return -1;
 }

... returning the process id of the child process to the creating process (the parent process)
'-1' as C-style indication of a failure in void of actual exception handling

Concurrent programming languages

Language candidates

- Ada, C++, Rust
- C
- C#
- Go
- Chapel, M0
- Occam, CSP
- All .net languages
- Java, Scala, Clojure
- Algol 68, Modula-2, Modula-3
- Modula-3
- Eiffel, Pascal
- Ruby, Python
- [mostly broken due to global interpreter locks]

Explicit concurrency

- Ada, C++, Rust
- C
- C#
- Go
- Chapel, M0
- Occam, CSP
- All .net languages
- Java, Scala, Clojure
- Algol 68, Modula-2, Modula-3

Implicit (potential) concurrency

- Lisp, Haskell, Caml, Modula-3, and any other functional language
- Smalltalk, Squeak
- Prolog
- Eiffel, Jaume, Signal

Concurrent program parts

Parameters can be evaluated independently or could run concurrently

Pure functional programming is side-effect free

Some functional languages allow for lazy evaluation, i.e. sub-expressions are not necessarily evaluated completely:

- In the case of Haskell: qsort (x:xs) = qsort (y | y <- xs, y < x) ++ x ++ qsort (y | y <- xs, y >= x)

Languages with implicit concurrency: e.g. functional programming

Implicit concurrency in some programming schemes

Quicksort in a functional language (here: Haskell):

qsort [] = []
qsort (x:xs) = qsort (y | y <- xs, y < x) ++ x ++ qsort (y | y <- xs, y >= x)

Some functional languages allow for lazy evaluation, i.e. sub-expressions are not necessarily evaluated completely:

- Eiffel, Prolog
- Libraries & interfaces (outward language definitions)
- POSIX
- API/Message Passing Interface

Summary

Concurrency – The Basic Concepts

- No support:
- Python, Pascal
- C
- Fortran, Cobol, Basic...
- Libraries & interfaces (outward language definitions)
- POSIX
- API/Message Passing Interface
- ...