Introduction to Concurrency

Forms of concurrency

What is concurrency?

Why do we need/concurrent?

A computer scientist's view on concurrency

Forms of concurrency

A computer scientist's view on concurrency

Why would a computer scientist consider concurrency?

An engineer's view on concurrency

References for this chapter

[Ben-Ari06]
M. Ben-Ari
Principles of Concurrent and Distributed Programming

Principles of Concurrent and Distributed Programming
M. Ben-Ari

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 would a computer scientist consider concurrency?

- Physics, engineering, electronics, biology, …
- 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)

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.

Forms of concurrency

A computer scientist's view on concurrency

Terminology for physically concurrent machines architectures:

- Overlapped I/O and computation:
  - Imply 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 concurrency
- Parallel Machines & distributed operating systems:
  - Add (non-deterministic) communication channels
- General network architectures:
  - Allow for any form of communicating distributed entities
- SIMD:
  - Single instruction, single data
- MIMD:
  - Multiple instruction, multiple data
- MISD:
  - Multiple instruction, single data
- MISP:
  - Multiple instruction, parallel processors

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 is usually strictly based on the structure of the given physical system.
Introduction to Concurrency

Forms of 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 (timing, throughput, load, available resources, signals... throughout the execution)
- non-reproducible (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 meaningful
- 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

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)

Atomic operations [detailed discussion later]:
- Complex and powerful atomic operations ease the correctness proofs / designs in concurrent systems.
- Faults and failures due to external factors are handled consistently.
- Atomic operations make it easier to prove logical correctness.

Slight changes in external triggers may (and usually does) result in completely different schedules (interleaving):
- Concurrent programs which depend 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 and are independent of the actual timing behavior are essential.
- Some timing restrictions for the scheduling will prevail in non-real-time systems, e.g. 'fairness'.

Introduction to Concurrency

Models and Terminology

The concurrent programming abstraction

Correctness vs. testing in concurrent systems:

- Termination is often not intended or even considered a failure.
- Partial correctness:
  - Programs terminate and produce the correct output.
- Total correctness:
  - Programs terminate and produce the correct output.

Safety properties:

- Processes (P) and (Q) are safe if (P) ... (Q).
- Processes (P) and (Q) are safe if (P) ... (Q).

Liveness properties:

- Processes (P) and (Q) are live if (P) ... (Q).
- Processes (P) and (Q) are live if (P) ... (Q).

Examples:

- Distributed systems: Processes must communicate correctly.
- Real-time systems: Processes must complete within time constraints.
- Operating systems: Processes must be able to access shared resources.

Introduction to Concurrency

Models and Terminology

The concurrent programming abstraction

Introduction to processes and threads

1 CPU per control-flow

1 CPU for all control-flows

- Support for memory protection is essential.
- Process management (scheduling) required.
- Shared memory access must be coordinated.

Examples:

- Requests need to complete eventually
- The state of the system needs to be displayed eventually
- No part of the system is to be delayed forever (fairness)
- Interleaving liveness properties can be very hard to prove
Introduction to Concurrency

Introduction to processes and threads

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
    - Thread can easily be connected to external events (IO).
  - Outside the OS:
    - User-level scheduling
    - Threads may need to go through their parent process to access IO.

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.

Inside the OS:

- Address space
- Control flow
- Thread can easily be connected to external events (I/O).
- Any process/thread can be executed on any available CPU.

Outside the OS:

- Shared memory
- Threads may need to be connected to external events (I/O).

Process Control Blocks (PCBs)

- Process Id
- Process state:
  - created: ready to run, but not yet considered by any dispatcher or waiting for admission
  - ready: ready to run or waiting for a free CPU
  - running: holds a CPU and executes
  - blocked: not ready to run or waiting for a resource
  - suspended: swapped out of main memory (non-time critical processes) or waiting for main memory space (and other resources)

- Process states
  - created: the task is ready to run, but not yet considered by any dispatcher or waiting for admission
  - ready: ready to run or waiting for a free CPU
  - running: holds a CPU and executes
  - blocked: not ready to run or waiting for a resource
  - suspended: swapped out of main memory (non-time critical processes) or waiting for main memory space (and other resources)
  - terminated: the task is not ready to run or waiting for admission
  - terminated: the task is not ready to run or waiting for admission

Allocated resources / privileges
- Open and requested devices and files, ...

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

CPU scheduling and dispatching can be

Kernel has full knowledge about all processes as well as their states, requirements and currently held resources.

Processes can share memory and the specific definition of threads is different in different operating systems and contexts.

Threads can be regarded as a group of processes, which share certain resources (process control block(PCB)).

Due to the overhead in resources, the threads attached to the tasks are less than the actual 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 Id

- Priorities, deadlines, consumed CPU-time, ...
- CPU state
- Saved registers (complete CPU state)

Current task

- Memory attributes / privileges
- Memory base, limits, shared areas, ...
- Allocated resources / privileges

Shared memory

- Open and requested devices and files, ...

PCBs (links thereof) are commonly enqueued at a certain state or condition (waiting access or change in state)
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 such as an indication of a failure in void of actual exception handling.

Frequent usage:
```
if (fork () == 0) {
    close (data_pipe[1]);
    exit (0);
} else {
    // the parent's task
    pid = wait (); // wait for the termination of one child process
}
```

**Concurrent programming languages**

- **Language candidates**
  - Ada, C++, Rust
  - C, Go
  - Erlang, GHC
  - Haskell, OCaml
  - LLVM, Skylab
  - Python
  - Rust

- **Explicit concurrency**
  - Ada, C++, Rust
  - C, Go
  - Erlang, GHC
  - Haskell, OCaml
  - LLVM, Skylab
  - Python
  - Rust

- **Implicit (potential) concurrency**
  - Ada, C++, Rust
  - C, Go
  - Erlang, GHC
  - Haskell, OCaml
  - LLVM, Skylab
  - Python
  - Rust

- **No support**
  - Ada, C++, Rust
  - C, Go
  - Erlang, GHC
  - Haskell, OCaml
  - LLVM, Skylab
  - Python
  - Rust

**Languages with implicit concurrency: e.g. functional programming**

- **Implicit concurrency in some programming schemes**
    ```
    qsort [] = []
    qsort [y | y <- xs, y < x] ++ [x] ++ qsort (x:xs) =
    ```
    - Pure functional programming is **side-effect free**
    - Some functional languages allow for **lazy evaluation**: sub-expressions are not necessarily evaluated completely:
      ```
      borderline = n /= 0 && (g (n) > h (n))
      ```
    - Parameters can be evaluated independently or could run concurrently

- **Process states**
  - Ada, C++, Rust
  - C, Go
  - Erlang, GHC
  - Haskell, OCaml
  - LLVM, Skylab
  - Python
  - Rust

- **No support**
  - Ada, C++, Rust
  - C, Go
  - Erlang, GHC
  - Haskell, OCaml
  - LLVM, Skylab
  - Python
  - Rust

**Concurrent programming languages**

- **Language candidates**
  - Ada, C++, Rust
  - C, Go
  - Erlang, GHC
  - Haskell, OCaml
  - LLVM, Skylab
  - Python
  - Rust

- **Explicit concurrency**
  - Ada, C++, Rust
  - C, Go
  - Erlang, GHC
  - Haskell, OCaml
  - LLVM, Skylab
  - Python
  - Rust

- **Implicit concurrency**
  - Ada, C++, Rust
  - C, Go
  - Erlang, GHC
  - Haskell, OCaml
  - LLVM, Skylab
  - Python
  - Rust

**Summary**

- **Concurrency – The Basic Concepts**
  - **Forms of concurrency**
  - **Models and terminology**
    - Abstractions and perspectives: computer science, physics & engineering
    - Observations: determinism, concurrency, interactivity, interfacing
    - Concurrency in concurrent systems
  - **Processes and threads**
    - Basic concepts and notions
    - Process states
  - **Concurrent programming languages**
    - Explicit concurrency: e.g. Ada, C, Go
    - Implicit concurrency: functional programming: e.g. Haskell, C, ML
    - Libraries & interfaces (outstanding language definitions)
    - POSIX
    - MPI (Message Passing Interface)