Introduction to 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, conjoined, associated (Oxford English Dictionary)
- Strict sequential processing is suggested by widely used programming languages.

Why do we need/have concurrency?

- Physics, engineering, electronics, biology, …
- Future: computing the physical world.

Why would a computer scientist consider concurrency?

- To model and control phenomena of the physical world
- To enhance the reactivity of a system
- To enhance the performance of a system
- …

A computer scientist's view on concurrency

- Overlapped I/O and computation
  - Impacts interrupt programming to handle I/O
- Multi-programming
  - Allows multiple independent programs to be executed on one CPU
- Multi-tasking
  - Allows multiple interacting processes to be executed on one CPU
- Multi-processor systems
  - Add physical real concurrency
- SIMD
  - Single instruction, multiple data
- Vector processors
- Parallel machines
  - Add (non-deterministic) communication channels
- Multi-instruction, single-data (MISD)
- Multiple instruction, single-data (MISD)

A computer scientist's view on concurrency

- Terminology for physically concurrent machines and architectures:
  - SISD (single instruction, single-data)
  - SIMD (single instruction, multiple-data)
  - MIMD (multiple instruction, multiple-data)
  - Multi-processors or computer networks

An engineer's view on concurrency

- Multiple physical, coupled, dynamical systems form the actual environment and/or task at hand
- ...
Introduction to Concurrency

Forms of concurrency

Does concurrency lead to chaos?

Concurrency often leads to the following features / issues / problems:

- non-deterministic phenomena
- non-obvious 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?

Concurrent execution can result in non-deterministic results.

Large scale, high bandwidth interconnected nodes ("supercomputers")

Networked computing nodes

Standalone computing nodes – including local buses & interfaces

Operating systems (& distributed operating systems)

= explicit concurrent programming (message passing and synchronization)

Explicit concurrent programming

Individual concurrent units inside one CPU

Individual electronic circuits

Meaningful employment of concurrent system 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 ...)

Models and Terminology

The concurrent programming abstraction

- 'concurrent' is technically defined negatively as:
  If there is no observer who can identify two events as being in a 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 and logic:
  "Concurrent programming abstraction is the study of interleaved execution sequences of the atomic instructions of sequential processes."

Real-time systems

Non-real-time systems

Interaction occurs in form of:

- Communication (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

Stay in your seat
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

Atomic operations:

- Slight changes in external triggers may (and usually does) result in completely different schedules (interleavings):
  - 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 and with independence of the actual timing behavior are essential.
  - Some testing prerequisites for the scheduling can prevent in non-real-time systems, e.g. 'fairness.'

Introduction to Concurrency

Models and Terminology

The concurrent programming abstraction

Correctness vs. testing in concurrent systems:

- Atomic operations: '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.

Introduction to Concurrency

Models and Terminology

The concurrent programming abstraction

Safety properties:

- (P \land \text{Processes} (I,S)) \Rightarrow Q (I,O)

Examples:

- Mutual exclusion (no resource collisions)
- Absence of deadlocks
- 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 \land \text{Processes} (I,S)) \Rightarrow Q (I,O)

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)
- Interesting liveness properties can be very hard to prove

1 CPU per control-flow

Specific configurations:

- Distributed processors
- Physical process control systems
  - Several systems connected via a bus system
- Process management (scheduling) not required
- Shared memory access need to be coordinated

1 CPU for all control-flows

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

Processes

Introduction to processes and threads

Processes

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

Threads

Introduction to processes and threads

Threads

- Inside the OS:
  - Kernel scheduling
- Outside the OS:
  - User-level scheduling

Process Control Blocks

- Process ID
- Process state
- Scheduling attributes
  - Priorities, deadlines, consumed CPU-time
- CPU state: Stored/unstored information while context switches
- Memory attributes / privileges
  - Memory space
- Allocated resources / privileges
  - Open and requested devices and files

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.

Process states

- Created: the task is ready to run, 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 (none time critical processes) or waiting for main memory space and other resources

Thread switching and inter-thread communication can be more efficient than switching on process level.

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

Process Control Blocks (PCBs)

- Process ID
- Process state
- Scheduling info
- Memory attributes / privileges
- Allocated resources / privileges

© 2020 Uwe R. Zimmer, The Australian National University page 188 of 758 (chapter 1: "Introduction to Concurrency" up to page 202)
In UNIX systems tasks are created by 'cloning'
pid = fork ();
resulting in a duplication of the current process
  ... returning a 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)

Frequent usage:
if (fork () == 0) {
  ... the child's task ...
  ... often implemented as exec: "absolute path to executable file", "args";
  return (0); /~ terminate child process */
} else {
  ... the parent's task ...
  pid = wait (); /~ wait for the termination of one child process */
}

© 2020 Uwe R. Zimmer, The Australian National University page 197 of 758 (chapter 1: "Introduction to Concurrency" up to page 202)