In this chapter

Hardware architectures:
☞ From simple logic to multi-core CPUs
☞ Concurrency on different levels

Software architectures:
☞ Languages of Concurrency
☞ Operating systems and libraries

Layer of abstraction

Abstraction Layer | Form of concurrency
--- | ---
Application level (user interface, specific functionality...) | Distributed systems, servers, web services, "multitasking" (popular understanding)
Language level (data types, tasks, classes, API, ...) | Process libraries, tasks/threads (language), synchronisation, message passing, intrinsic, ...
Operating system (HAL, processes, virtual memory) | OS processes/threads, signals, events, multitasking, SMP, virtual parallel machines,...
CPU / instruction level (assembly instructions) | Logically concurrent: pipelines, out-of-order, etc.
Device / register level (arithmetic units, registers,...) | Parallel adders, SIMD, multiple execution units, caches, prefetch, branch prediction, etc.
Logic gates (and, ‘or’, ‘not’, flip-flop, etc.) | Inherently massively parallel, synchronised by clock; or: asynchronous logic
Digital circuitry (gates, buses, clocks, etc.) | Multiple clocks, peripheral hardware, memory, ...
Analog circuitry (transistors, capacitors, ...) | Continuous time and inherently concurrent

References

[Bacon98]
J. Bacon
Concurrent Systems

[Stallings2001]
Stallings, William
Operating Systems
Prentice Hall, 2001

[Intel2010]
Intel® 64 and IA-32 Architectures Optimization Reference Manual
http://www.intel.com/products/processor/manuals/
Architectures

Logic - the basic building blocks for digital computers

Controllable Switches & Ratios

as transistors, relays, vacuum tubes, valves, etc.

Logic - the basic building blocks for digital computers

Constructing logic gates – for instance NAND in CMOS:

Half adder:

**Full adder:**

Ripple carry adder:
A simple CPU

- **Decoder/Sequencer**
  Can be a machine in itself which breaks CPU instructions into concurrent micro code.

- **Execution Unit / Arithmetic-Logic-Unit (ALU)**
  A collection of transformational logic.

- **Memory**

- **Registers**
  Instruction pointer, stack pointer, general purpose and specialized registers

- **Flags**
  Indicating the states of the latest calculations.

- **Code/Data management**
  Fetching, Caching, Storing

Interrupts

- One or multiple lines wired directly into the sequencer
  - Required for:
    - Pre-emptive scheduling, Timer driven actions, Transient hardware interactions, …
  - Usually preceded by an external logic ("interrupt controller") which accumulates and encodes all external requests.

On interrupt (if unmasked):

- CPU stops normal sequencer flow.
- Lookup of interrupt handler's address
- Current IP and state pushed onto stack.
- IP set to interrupt handler.
Interrupt processing

- Push registers
- Declare local variables
- Run handler code
  - do some I/O
  - or run some critical code
- Return address
- Context
- Parameters
- Global variables
- Local variables
- Flags

Program

Interrupt handler
Interrupt processing

Interrupt handler

1. Push registers
2. Declare local variables
3. Run handler code
   - do some I/O ...
   - or run some time-critical code ...
4. Remove local variables

Program

Stack

Code

Return address

Context

Parameters

Global variables

Local variables

FP

Base

SP

Push registers
Declare local variables
Run handler code
   - do some I/O ...
   - or run some time-critical code ...
Remove local variables

R e t u r n f r o m i n t e r r u p t
Clear interrupt flag
(Adjust priorities)
(Re-enable interrupt)
Push other registers
Declare local variables
Run handler code
... do some I/O ...
... or run some time
   critical code ...
### Interrupt processing

**Interrupt handler**

- Clear interrupt flag (Adjust priorities)
- Push other registers
- Declare local variables
- Run handler code
- ... or run some time critical code...
- Remove local variables
- Pop other registers

```
... do some I/O ... or run some time critical code ...
```

```
Return address  
Context
Parameters
Local variables
Global variables
Base
```

```
PC
SP
FP
```

### Things to consider

- **☞** Interrupt handler code can be interrupted as well.
- **☞** Are you allowing to interrupt an interrupt handler with an interrupt on the same priority level (e.g. the same interrupt)?
- **☞** Can you overrun a stack with interrupt handlers?
Multiple programs

If we can execute interrupt handler code “concurrently” to our “main” program:

- Can we then also have multiple “main” programs?
Architectures

Context switch

Dispatcher

Push registers, declare local variables, Store SP to PCB 1

Declares local variables, Store SP to PCB 1

Scheduler

Load SP from PCB 2

Remove local variables
Processor Architectures

Pipeline

Some CPU actions are naturally sequential (e.g., instructions need to be first loaded, then decoded before they can be executed).

More fine grained sequences can be introduced by breaking CPU instructions into micro code.

- Overlapping those sequences in time will lead to the concept of pipelines.
- Same latency, yet higher throughput.
- (Conditional) branches might break the pipelines.
- Branch predictors become essential.

Parallel pipelines

Filling parallel pipelines (by alternating incoming commands between pipelines) may employ multiple ALU's.

- (Conditional) branches might again break the pipelines.
- Interdependencies might limit the degree of concurrency.
- Same latency, yet even higher throughput.
- Compilers need to be aware of the options.
**Out of order execution**

Breaking the sequence inside each pipeline leads to "out of order" CPU designs.

- Replace pipelines with hardware scheduler.
- Results need to be "re-sequentialized" or possibly discarded.
- "Conditional branch prediction" executes the most likely branch or multiple branches.
- Works better if the presented code sequence has more independent instructions and fewer conditional branches.
- This hardware will require (extensive) code optimization to be fully utilized.

**SIMD ALU units**

Provides the facility to apply the same instruction to multiple data concurrently. Also referred to as "vector units".

Examples: Altivec, MMX, SSE[2|3|4], ...

- Requires specialized compilers or programming languages with implicit concurrency.

**GPU processing**

Graphics processor as a vector unit.

- Unifying architecture languages are used (OpenCL, CUDA, GPGPU).

**Hyper-threading**

Emulates multiple virtual CPU cores by means of replication of:
- Register sets
- Sequencer
- Flags
- Interrupt logic

while keeping the "expensive" resources like the ALU central yet accessible by multiple hyper-threads concurrently.

- Requires programming languages with implicit or explicit concurrency.

Examples: Intel Pentium 4, Core i5/i7, Xeon, Atom, Sun UltraSPARC T2 (8 threads per core)

**Multi-core CPUs**

Full replication of multiple CPU cores on the same chip package.

- Often combined with hyper-threading and/or multiple other means (as introduced above) on each core.
- Cleanest and most explicit implementation of concurrency on the CPU level.

- Requires synchronized atomic operations.
- Requires programming languages with implicit or explicit concurrency.

Historically the introduction of multi-core CPUs ended the "GHz race" in the early 2000's.

- Low cost 32 bit processor ($8)
- 8 cores with 2 kB local memory
- 40 kB shared memory
- No interrupts!
- 8 semaphores

Virtual memory

Translates logical memory addresses into physical memory addresses and provides memory protection features.

- Does not introduce concurrency by itself.
- Is still essential for concurrent programming as hardware memory protection guarantees memory integrity for individual processes / threads.

Alternative Processor Architectures: IBM Cell processor (2001)

- Theoretical 25.6 GFLOPS at 3.2 GHz
- 8 cores for specialized high-bandwidth floating point operations and 128 bit registers
- Multiple interconnect topologies

Virtual memory translates logical memory addresses into physical memory addresses and provides memory protection features.
**Architectures**

**Multi-CPU systems**

**Scaling up:**
- Multi-CPU on the same memory
  - multiple CPUs on same motherboard and memory bus, e.g. servers, workstations
- Multi-CPU with high-speed interconnects
  - various supercomputer architectures, e.g. Cray XE6:
    - 12-core AMD Opteron, up to 192 per cabinet (2304 cores)
    - 3D torus interconnect (160 GByte/sec capacity, 46 ports per node)
- Cluster computer (Multi-CPU over network)
  - multiple computers connected by network interface, e.g. Sun Constellation Cluster at ANU:
    - 1492 nodes, each: 2x Quad core Intel Nehalem, 24 GB RAM
    - QDR Infiniband network, 2.6 GB/sec

---

**Vector Machines**

**Vectorization**

\[
\begin{align*}
a \cdot \vec{v} &= a \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} \\
&= \begin{pmatrix} a \cdot x \\ a \cdot y \\ a \cdot z \end{pmatrix}
\end{align*}
\]

**Function is “promoted”**

- Translates into CPU-level vector operations as well as multi-core or fully distributed operations

**Example Code:**

```plaintext
const Index = [1 .. 10000000],
Vector_1 : [Index] real = 1.0,
Scale    : real = 5.1,
Scaled   : [Vector] real = Scale * Vector_1;

Translates into
CPU-level vector operations
```

---

**Reduction**

\[
\begin{align*}
\vec{v}_1 = \vec{v}_2 &\Rightarrow \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} = \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} \\
&\Rightarrow (x_1 = x_2) \wedge (y_1 = y_2) \wedge (z_1 = z_2)
\end{align*}
\]

**Example Code:**

```plaintext
type Real is digits 15;
type Vectors is array (Positive range <>) of Real;
function "=\" (Vector_1, Vector_2 : Vectors) return Boolean is
begin
  for i in Vector_1'Range loop
    V_1 (i) := Vector_1 (i);
  end loop;
return V_1 = Vector_2;
end Vector_1 = Vector_2;
```

**Combined with in-lining, loop unrolling and caching this is as fast as a single CPU will get.**
Vector Machines

**Reduction**

\[
V_1 = V_2 \Rightarrow \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} = \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} \Rightarrow (x_1 = x_2) \land (y_1 = y_2) \land (z_1 = z_2)
\]

- Operations are evaluated in a concurrent divide-and-conquer (binary tree) structure.
- Translates into CPU-level vector operations as well as multi-core or fully distributed operations.

Cellular automaton transitions from a state \( S \) into the next state \( S' \):

\( S \to S' \parallel \forall c \in S; c \to c' = r(S, c) \), i.e., all cells of a state transition concurrently into new cells by following a rule \( r \).

Next State = forall World_Indices in World do Rule (State, World_Indices);

John Conway's Game of Life rule:

\[
\text{proc Rule} (\text{S}, (i, j) : \text{index} (\text{World})) : \text{Cell} \\
\quad \text{const Population} : \text{index} ([0 .. 9]) = \\
\quad \quad \text{reduce Count (Cell.Alive, S [i - 1 .. i + 1, j - 1 .. j + 1]);} \\
\quad \text{return} (\text{if Population == 3} \\
\quad \quad \text{|| (Population == 4} \&\& \text{S [i, j] == Cell.Alive} \text{then Cell.Alive} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{else Cell.Dead);})
\]

What is an operating system?
What is an operating system?

1. A virtual machine!
   ... offering a more comfortable and safer environment
   (e.g. memory protection, hardware abstraction, multitasking, ...)

2. A resource manager!
   ... coordinating access to hardware resources

Operating systems deal with:
- processors
- memory
- mass storage
- communication channels
- devices (timers, special purpose processors, peripheral hardware, ...)
- tasks/processes/programs which are applying for access to these resources!
The evolution of operating systems

- in the beginning: single user, single program, single task, serial processing - no OS
- 50s: System monitors / batch processing
  - the monitor ordered the sequence of jobs and triggered their sequential execution
- 50s-60s: Advanced system monitors / batch processing:
  - the monitor is handling interrupts and timers
  - first support for memory protection
  - first implementations of privileged instructions (accessible by the monitor only).
- early 60s: Multiprogramming systems:
  - employ the long device I/O delays for switches to other, runnable programs
- early 60s: Multiprogramming, time-sharing systems:
  - assign time-slices to each program and switch regularly
- early 70s: Multitasking systems – multiple developments resulting in UNIX (besides others)
- early 80s: Single user single tasking systems, with emphasis on user interface or APIs
  - MS-DOS, CP/M, MacOS and others first employed ‘small scale’ CPUs (personal computers).
- mid-80s: Distributed/multiprocessor operating systems - modern UNIX systems (SYSV, BSD)

The evolution of communication systems

- 1901: first wireless data transmission (Morse-code from ships to shore)
- 56: first transmission of data through phone-lines
- 62: first transmission of data via satellites (Telstar)
- 69: ARPA-net (predecessor of the current internet)
- 80s: introduction of fast local networks (LANs): ethernet, token-ring
- 90s: mass introduction of wireless networks (LAN and WAN)

Current standard consumer computers might come with:
- High speed network connectors (e.g. GB-Ethernet)
- Wireless LAN (e.g. IEEE802.11g, …)
- Local device bus-system (e.g. Firewire 800, Fibre Channel or USB 3.0)
- Wireless local device network (e.g. Bluetooth)
- Infrared communication (e.g. IrDA)
- Modem/ADSL

Types of current operating systems

Parallel operating systems

- support for a large number of processors, either:
  - symmetrical: each CPU has a full copy of the operating system
  - asymmetrical: only one CPU carries the full operating system, the others are operated by small operating system stubs to transfer code or tasks.
Types of current operating systems

Distributed operating systems
- all CPUs carry a small kernel operating system for communication services.
- all other OS-services are distributed over available CPUs
- services may migrate
- services can be multiplexed in order to
  - guarantee availability (hot stand-by)
  - or to increase throughput (heavy duty servers)

Real-time operating systems
- fast context switches?
- small size?
- quick response to external interrupts?
- multitasking?
- "low level" programming interfaces?
- interprocess communication tools?
- high processor utilization?

Real-time operating systems
- need to provide...
  - the logical correctness of the results as well as
  - the correctness of the time, when the results are delivered
  - predictability! (not performance!)
  - all results are to be delivered just-in-time – not too early, not too late.
  - timing constraints are specified in many different ways...
    - often as a response to external events
    - in reactive systems
What is an operating system?

Is there a standard set of features for operating systems?

**No:**
the term ‘operating system’ covers 4 kB microkernels, as well as > 1 GB installations of desktop general purpose operating systems.
What is an operating system?

Is there a standard set of features for operating systems?

☞ no:
the term ‘operating system’ covers 4 kB microkernels,
as well as > 1 GB installations of desktop general purpose operating systems.

Is there a minimal set of features?

☞ almost:
memory management, process management and inter-process communication/synchronisation
will be considered essential in most systems

Is there always an explicit operating system?

☞ no:
some languages and development systems operate with standalone runtime environments

Typical features of operating systems

Process management:
- Context switch
- Scheduling
- Book keeping (creation, states, cleanup)

☞ context switch:
☞ needs to...
  - ‘remove’ one process from the CPU while preserving its state
  - choose another process (scheduling)
  - ‘insert’ the new process into the CPU, restoring the CPU state

Some CPUs have hardware support for context switching, otherwise:
☞ use interrupt mechanism
Typical features of operating systems

Memory management:
- Allocation / De-allocation
- Virtual memory: logical vs. physical addresses, segments, paging, swapping, etc.
- Memory protection (privilege levels, separate virtual memory segments, ...)
- Shared memory

Synchronisation / Inter-process communication
- Semaphores, mutexes, cond. variables, channels, mailboxes, MPI, etc. (chapter 4)
  - Tightly coupled to scheduling / task switching!

Hardware abstraction
- Device drivers
- API
- Protocols, file systems, networking, everything else...

Typical structures of operating systems

Monolithic & Modular
- Modules can be platform independent
- Easier to maintain and to develop
- Reliability is increased
- All services are still in the kernel (on the same privilege level)
  - May reach high efficiency
  - E.g. current Linux versions

Monolithic & layered
- Easily portable
- Significantly easier to maintain
- Crashing layers do not necessarily stop the whole OS
- Possibly reduced efficiency through many interfaces
- Rigorous implementation of the stacked virtual machine perspective on OSs
  - E.g. some current UNIX implementations (e.g. Solaris) to a certain degree, many research OSs (e.g. ‘THE system’, Dijkstra ’68)
**Typical structures of operating systems**

### µKernels & virtual machines

- µkernel implements essential process, memory, and message handling
- all 'higher' services are dealt with outside the kernel - no threat for the kernel stability
- significantly easier to maintain
- multiple OSs can be executed at the same time
- µkernel is highly hardware dependent - only the µkernel needs to be ported.
- possibly reduced efficiency through increased communications

- e.g. wide spread concept: as early as the CP/M, VM/370 ('79) or as recent as MacOS X (mach kernel + BSD unix), ...

### µKernels & client-server models

- µkernel implements essential process, memory, and message handling
- all 'higher' services are user level servers
- significantly easier to maintain
- kernel ensures reliable message passing between clients and servers
- highly modular and flexible
- servers can be redundant and easily replaced
- possibly reduced efficiency through increased communications

- e.g. current research projects, L4, etc.

---

**UNIX**

**UNIX features**

- Hierarchical file-system (maintained via 'mount' and 'umount')
- Universal file-interface applied to files, devices (I/O), as well as IPC
- Dynamic process creation via duplication
- Choice of shells
- Internal structure as well as all APIs are based on ‘C’
- Relatively high degree of portability

- UNIX,/BSD, XENIX, System V, QNX, SunOS, Ultrix, Sunix, Mach, Plan 9, NeXTSTEP, AIX, HP-UX, Solaris, NetBSD, FreeBSD, Linux, OPENSTEP, OpenBSD, Darwin, QNX/Neutrino, OS X, QNX RTOS, ...

---

© 2017 Uwe R. Zimmer, The Australian National University
**UNIX**

**Dynamic process creation**

```c
pid = fork ();
```

resulting a *duplication of the current process*
- returning 0 to the newly created process
- returning the *process id* of the child process to the creating process (the 'parent' process)
  or -1 for a failure

**Frequent usage:**

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

**Message passing in UNIX ★ Signals**

```c
int data_pipe [2], c, rc;
if (pipe (data_pipe) == -1) {
    perror ("no pipe");
    exit (1);
}
```
**Processes & IPC in UNIX**

- **Processes:**
  - Process creation results in a duplication of address space (‘copy-on-write’ becomes necessary)
  - Inefficient, but can generate new tasks out of any user process – no shared memory!
- **Signals:**
  - Limited information content, no buffering, no timing assurances (signals are not interrupts!)
  - Very basic, yet not very powerful form of synchronisation
- **Pipes:**
  - Unstructured byte-stream communication, access is identical to file operations
  - Not sufficient to design client-server architectures or network communications

**POSIX**

- IEEE/ANSI Std 1003.1 and following.
- Library Interface (API)
  - [C Language calling conventions – types exit mostly in terms of (open) lists of pointers and integers with overloaded meanings].
- More than 30 different POSIX standards (and growing / changing).
  - A system is ‘POSIX compliant’, if it implements parts of one of them!
  - A system is ‘100% POSIX compliant’, if it implements one of them!

**Sockets in BSD UNIX**

- Sockets try to keep the paradigm of a universal file interface for everything and introduce:

  **Connectionless interfaces (e.g. UDP/IP):**
  - Server side: `socket -> bind -> recvfrom -> close`
  - Client side: `socket -> sendto -> close`

  **Connection oriented interfaces (e.g. TCP/IP):**
  - Server side: `socket -> bind -> (select | listen | accept | read | write | [close | shutdown]`)
  - Client side: `socket -> bind -> connect -> write | read | [close | shutdown]`
Frequently employed POSIX features include:

- **Threads**: a common interface to threading - differences to 'classical UNIX processes'
- **Timers**: delivery is accomplished using POSIX signals
- **Priority scheduling**: fixed priority, 32 priority levels
- **Real-time signals**: signals with multiple levels of priority
- **Semaphore**: named semaphore
- **Memory queues**: message passing using named queues
- **Shared memory**: memory regions shared between multiple processes
- **Memory locking**: no virtual memory swapping of physical memory pages

**Summary**

- **Hardware architectures** - from simple logic to supercomputers
  - logic, CPU architecture, pipelines, out-of-order execution, multithreading, ...
- **Data-Parallelism**
  - Vectorization, Reduction, General data-parallelism
- **Concurrency in languages**
  - Some examples: Haskell, Occam, Chapel
- **Operating systems**
  - Structures: monolithic, modular, layered, µkernels
  - UNIX, POSIX