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

Layers of abstraction

<table>
<thead>
<tr>
<th>Abstraction Layer</th>
<th>Form of concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application level</td>
<td>Distributed systems, servers, web services, “multitasking” (popular understanding)</td>
</tr>
<tr>
<td>Language level</td>
<td>Process libraries, tasks/threads (language), synchronisation, message passing, intrinsic, etc.</td>
</tr>
<tr>
<td>Operating system</td>
<td>OS processes/threads, signals, events, multitasking, SMP, virtual parallel machines, etc.</td>
</tr>
<tr>
<td>CPU / instruction level</td>
<td>Logically sequential: pipelines, out-of-order, etc. logically concurrent: multicores, interrupts, etc.</td>
</tr>
<tr>
<td>Device / register level</td>
<td>Parallel adders, SIMD, multiple execution units, caches, prefetch, branch prediction, etc.</td>
</tr>
<tr>
<td>Logic gates</td>
<td>Inherently massively parallel, synchronised by clock; or: asynchronous logic</td>
</tr>
<tr>
<td>Digital circuitry</td>
<td>Multiple clocks, peripheral hardware, memory, etc.</td>
</tr>
<tr>
<td>Analog circuitry</td>
<td>Continuous time and inherently concurrent</td>
</tr>
</tbody>
</table>
Logic - the basic building blocks for digital computers

Constructing logic gates – for instance NAND in CMOS:

\[ \text{NAND} \]

... and subsequently all other logic gates:

- **Half adder:**
- **Full adder:**
- **Ripple carry adder:**
Processor Architectures

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.
We successfully interrupted a sequence of operations ...
Interrupt processing

Interrupt handler

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

Remove local variables
Pop registers

Program
Stack
Code
PC

Registers
Local variables
FP
Base
SP

Global variables
Context
Parameters
We successfully interrupted a sequence of operations ... 

... and now the trick to get to the other side.

The CPU hardware (!) did that, before anything was changed.
Interrupt processing

Interrupt handler

Push registers
Declare local variables

Run handler code
... do some I/O ...
... or run some time
critical code ...

Remove local variables

Program

Stack

Code

PC

Registers

Local variables

FP

Base

SP

Global variables

Context

Parameters

Return address

PC

Flags
Interrupt processing

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

Clear interrupt flag
(Adjust priorities)
(Re-enable interrupt)
Interrupt processing

Interrupt handler

- Clear interrupt flag
- (Adjust priorities)
- (Re-enable interrupt)
- Push other registers
- Declare local variables
- .. do some I/O ...
- .. or run some time critical code ..
- Remove local variables
- Pop other registers
- Return ("bx lr")
**Architectures**

**Interrupt processing**

- **Interrupt handler**

  - **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?

**Interrupt handler**

**Multiple programs**

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

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

**Busy! Do Not Disturb!**
Processor Architectures

Parallel pipelines

Filling parallel pipelines (by alternating incoming commands between pipelines) may employ multiple ALUs.

- (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.

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.

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).

**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.

**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)

**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


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

### Multi-CPU systems

**Scaling up:**

- **Multi-CPU on the same memory**
  - multiple CPUs on the same motherboard and memory bus, e.g., servers, workstations
- **Multi-CPU with high-speed interconnects**
  - various supercomputer architectures, e.g., Cray X6:
    - 12-core AMD Opteron, up to 192 per cabinet (2304 cores)
    - 3D torus interconnect (16GB/sec capacity, 48 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, 26 GB/sec
**Architectures**

**Vector Machines**

**Vectorization**

\[
a \cdot \hat{v} = a \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a \cdot x \\ a \cdot y \\ a \cdot z \end{pmatrix}
\]

- Translates into CPU-level vector operations

```
type Real is digits 15;
type Vectors is array (Positive range <>) of Real;
function Scale (Scalar : Real; Vector : Vectors) return Vectors is
  Scaled_Vector : Vectors (Vector'Range);
begin
  for i in Vector'Range loop
    Scaled_Vector (i) := Scalar \* Vector (i);
  end loop;
  return Scaled_Vector;
end Scale;
```

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

**Reduction**

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

- Translates into CPU-level vector operations

```
type Real is digits 15;
type Vectors is array (Positive range <>) of Real;
function Equal (v1, v2) : bool is
  reduce (v1 = v2);
```

```
const Index = (1 .. 100000000),
Vector_1, Vector_2 : [Index] real = 1.0,
Scale    : real = 5.1,
Scaled   : [Vector] real = Scale \* Vector_1;
```

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

- Function is "promoted"
What is an operating system?

1. A virtual machine!

... offering a more comfortable and safer environment

(e.g. memory protection, hardware abstraction, multitasking, ...)

Translates into CPU-level vector operations as well as multi-core or fully distributed operations
What is an operating system?

1. A virtual machine!

... offering a more comfortable and safer environment

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, ...)
- and 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)
**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.

**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 multiplied in order to
  - guarantee availability (hot stand-by)
  - to increase throughput (heavy duty servers)
Types of current operating systems

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
    ☞ reactive systems

Embedded operating systems

- usually real-time systems, often hard real-time systems
- very small footprint (often a few KBs)
- none or limited user-interaction
☞ 90-95% of all processors are working here!
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.
Typical features of operating systems

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

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

Memory management:
- Allocation / Deallocation
- 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
(or 'the big mess...')

- non-portable
- hard to maintain
- lacks reliability
- all services are in the kernel (on the same privilege level)
- but: may reach high efficiency

Examples:
- MS-DOS (80s), Windows (all non-NT based versions)
- MacOS (until version 9), and many others...

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

Examples:
- current Linux versions

µKernels & virtual machines

- µkernel implements essential process, memory, and message handling
- '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

Examples:
- wide spread concept: as early as the CPM, VM/370 (79)
- as recent as MacOS X (mach kernel + BSD unix), ...
**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

---

**Typical structures of operating systems**

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.

---

**Dynamic process creation**

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

dresulting 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
UNIX

Synchronization in UNIX

Include <signal.h>

Pipes

#include <unistd.h>
#include <sys/types.h>
#include <signal.h>

pid_t id; void catch_stop (int sig_num) {
    /* do something with the signal */
}

Dynamic process creation

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

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 identical to file operations

• not sufficient to design client-server architectures or network communications

include <unistd.h>

write (data_pipe[1], &c, 1)

if (write (data_pipe[1], &c, 1) == -1) {
    perror ("pipe broken");
    close (data_pipe[1]);
    exit (1);
}; }

pid = wait ();

//... the parent's task ...

//... the child's task ...

often implemented as:

exec ("absolute path to executable file", "args");

exit (0);

/* terminate child process */

wait for the termination of one child process

Message passing in UNIX

#include <unistd.h>
#include <sys/types.h>
#include <signal.h>

pid_t id; void catch_stop (int sig_num) {
    /* do something with the signal */
}

// child

close (data_pipe[0]);

while ((c = getchar ()) > 0) {
    if (write (data_pipe[1], &c, 1) == -1) {
        perror ("pipe broken");
        close (data_pipe[1]);
        exit (1);
    }
};

close (data_pipe[1]);

pid = wait ();

// parent

close (data_pipe[0]);

while ((c = getchar ()) > 0) {
    if (write (data_pipe[1], &c, 1) == -1) {
        perror ("pipe broken");
        close (data_pipe[1]);
        exit (1);
    }
};

close (data_pipe[1]);

pid = wait ();

//... the parent's task ...
POSIX - some of the relevant standards...

<table>
<thead>
<tr>
<th>Standard</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1003.1</td>
<td>Single-process, multi-process, job control, signals, user groups, file system, file attributes, file device management, file locking, device I/O, device-specific control, system database, pipes, FIFO, ...</td>
</tr>
<tr>
<td>1003.1b</td>
<td>Real-time extensions: real-time signals, priority scheduling, timers, asynchronous I/O, prioritized I/O, synchronized I/O, file synchronization, memory locking, memory protection, message passing, semaphore, ...</td>
</tr>
<tr>
<td>1003.1c</td>
<td>Threads: multiple threads within a process, includes support for: thread control, thread attributes, priority scheduling, mutexes, mutex priority inheritance, mutex priority ceiling, and condition variables</td>
</tr>
<tr>
<td>1003.1d</td>
<td>Additional real-time extensions: new process creation semantics (spawns), sporadic server scheduling, execution time monitoring of processes and threads, I/O advisory information, timeouts on blocking functions, device control, and interrupt control</td>
</tr>
<tr>
<td>1003.1j</td>
<td>Advanced real-time extensions: typed memory, nanosleep improvements, barrier synchronization, reader/writer locks, spin locks, and persistent notification for message queues</td>
</tr>
<tr>
<td>1003.21</td>
<td>Distributed real-time extensions: buffer management, send control block, asynchronous and synchronous operations, bounded blocking, message priority, message labels, and implementation protocols</td>
</tr>
</tbody>
</table>

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

Architectures

- 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