642

Systems, Networks & Concurrency 2020

9

Architectures

Uwe R. Zimmer - The Australian National University

643

Architectures

References

[Bacon98]
J. Bacon
Concurrent Systems
1998 (2nd Edition) Addison Wesley Longman Ltd, ISBN 0-201-17767-6

[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/

644

Architectures

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

645

Architectures

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 sequential: pipelines, out-of-order, etc. logically concurrent: multicores, interrupts, 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

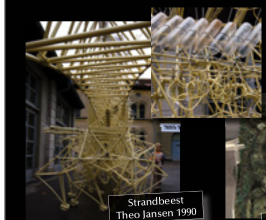
646

Architectures

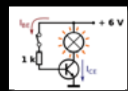
Logic - the basic building blocks

Controllable Switches & Ratios

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

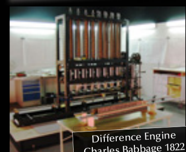


Strandbeest
Theo Jansen 1990



First transistor
John Bardeen and Walter Brattain 1947

Antikythera Mechanism
Greek 150-100 BC-
Credit: Wikipedia



Difference Engine
Charles Babbage 1822

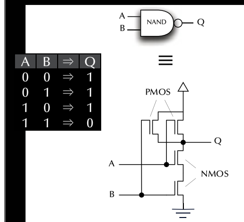
647

Architectures

Logic - the basic building blocks for digital computers

Constructing logic gates – for instance **NAND** in CMOS:

A	B	⇒	Q
0	0	⇒	1
0	1	⇒	1
1	0	⇒	1
1	1	⇒	0



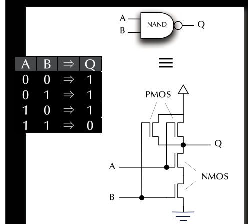
648

Architectures

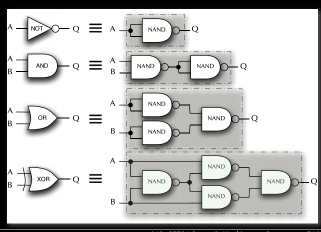
Logic - the basic building blocks for digital computers

Constructing logic gates – for instance **NAND** in CMOS:

A	B	⇒	Q
0	0	⇒	1
0	1	⇒	1
1	0	⇒	1
1	1	⇒	0



... and subsequently all other logic gates:

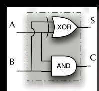


649

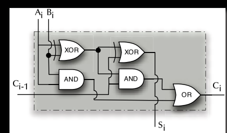
Architectures

Logic - the basic building blocks

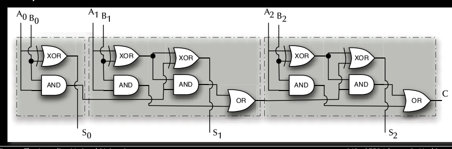
Half adder:



Full adder:



Ripple carry adder:

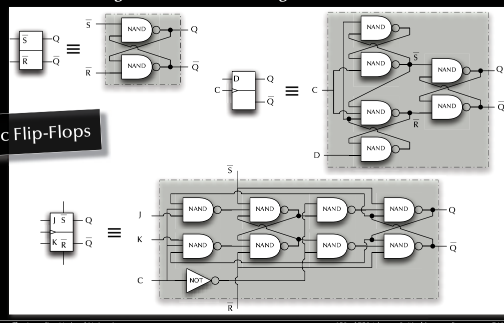


650

Architectures

Logic - the basic building blocks

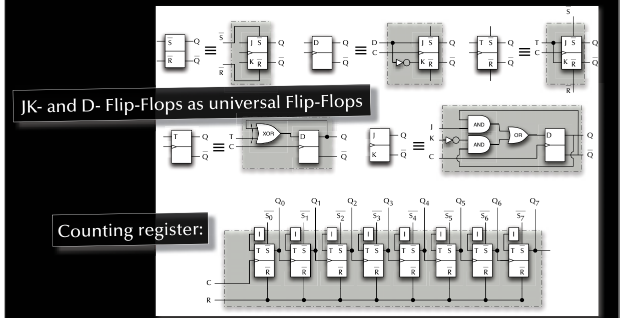
Basic Flip-Flops



651

Architectures

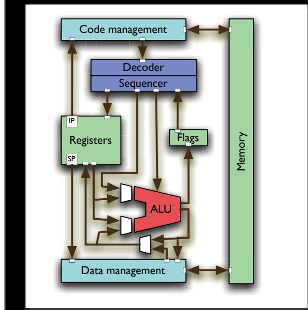
Logic - the basic building blocks



652

Architectures

Processor Architectures



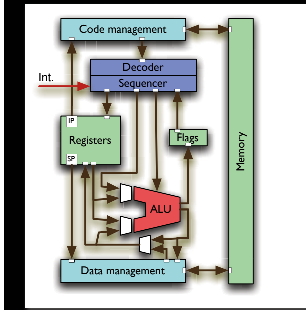
A simple CPU

- **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

653

Architectures

Processor Architectures



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.

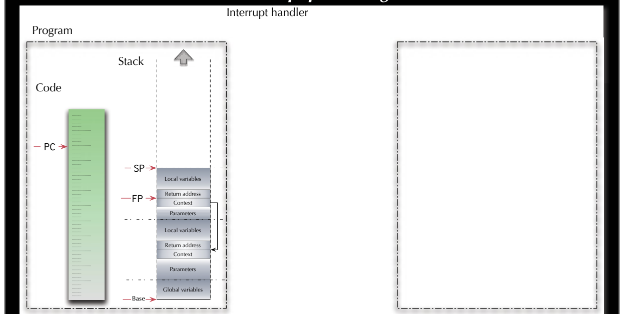
654

Bahia Honda Rail Bridge (Creative Commons Attribution-ShareAlike 3.0, Photography by MrX at English Wikipedia)

655

Architectures

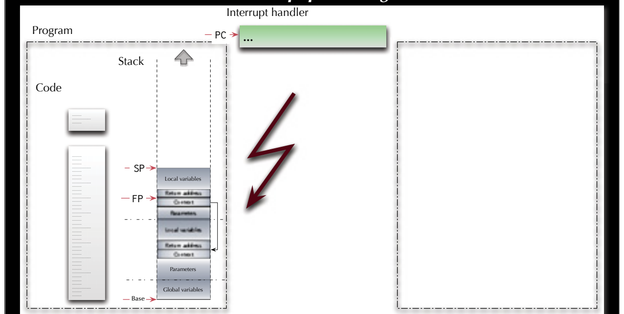
Interrupt processing



656

Architectures

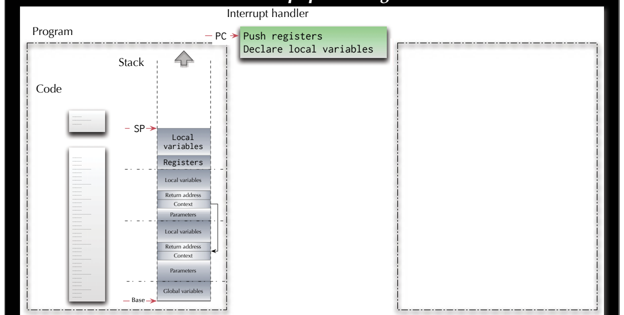
Interrupt processing



657

Architectures

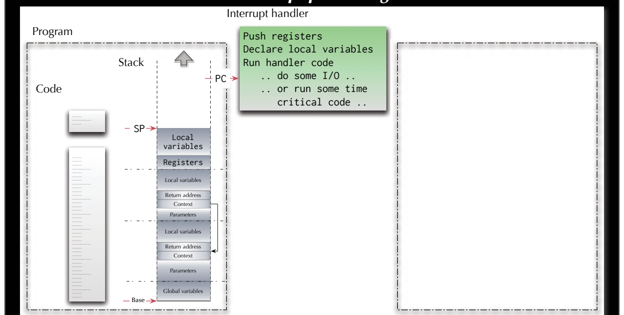
Interrupt processing



658

Architectures

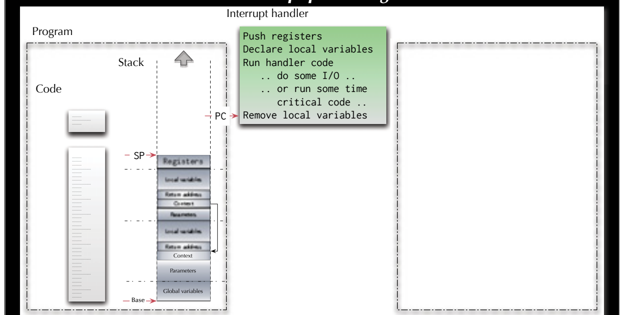
Interrupt processing



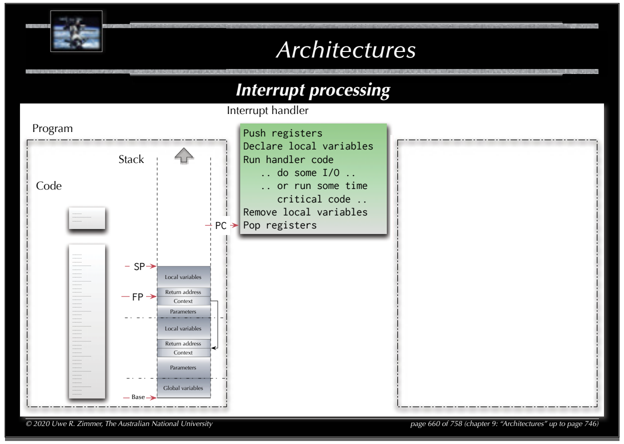
659

Architectures

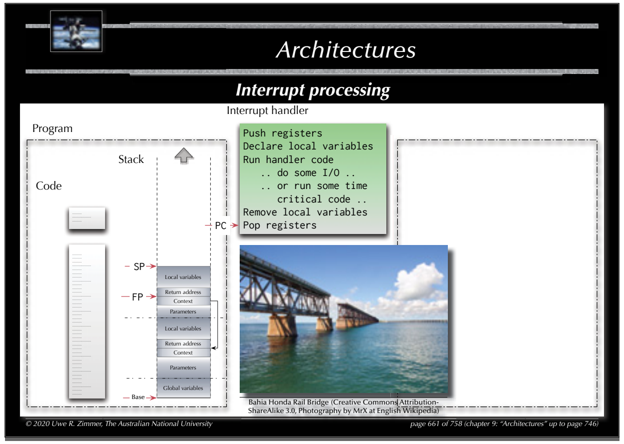
Interrupt processing



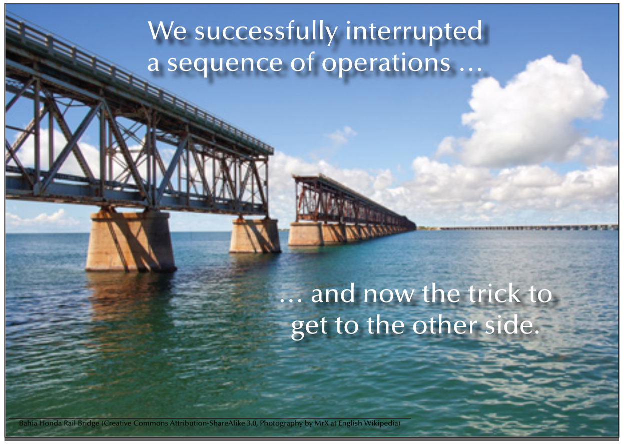
660



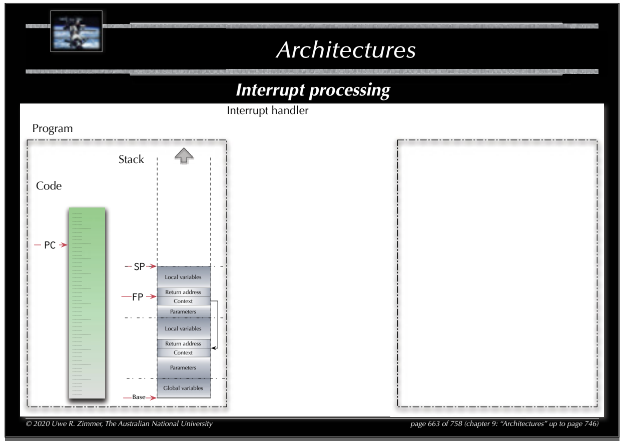
661



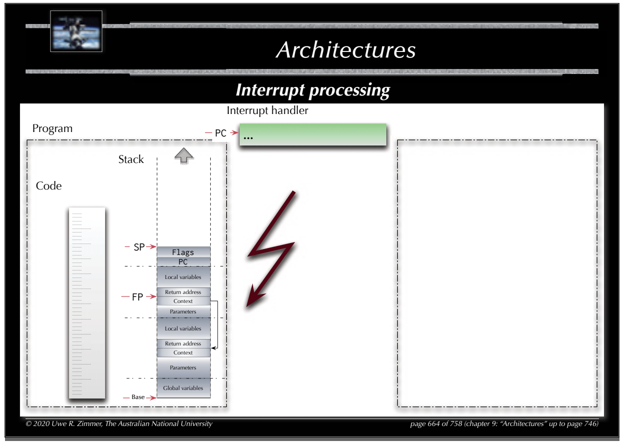
662



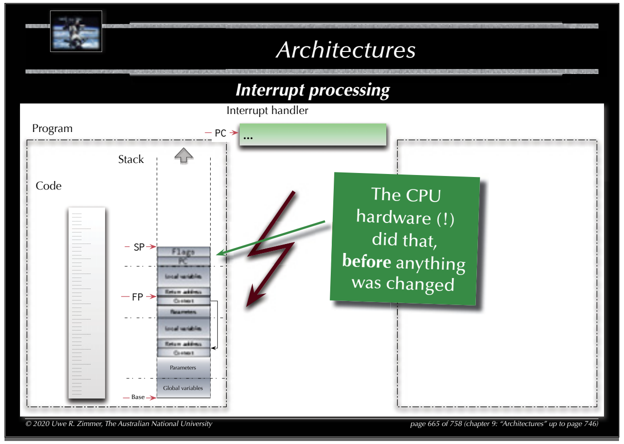
663



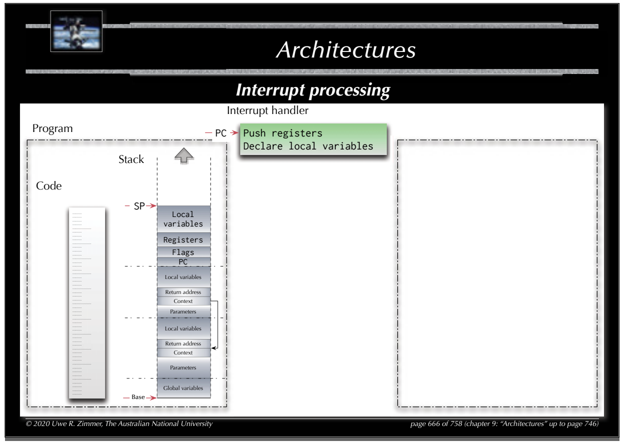
664



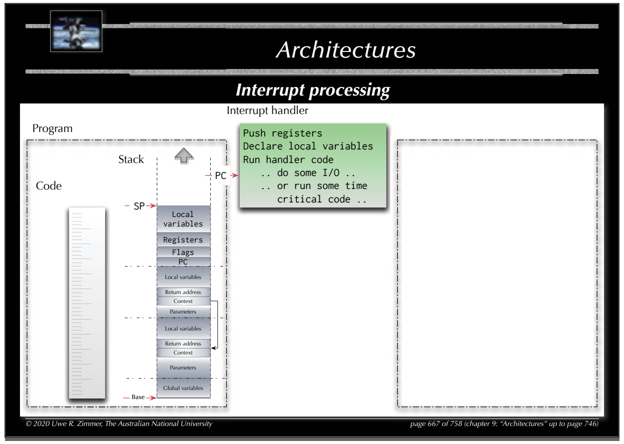
665



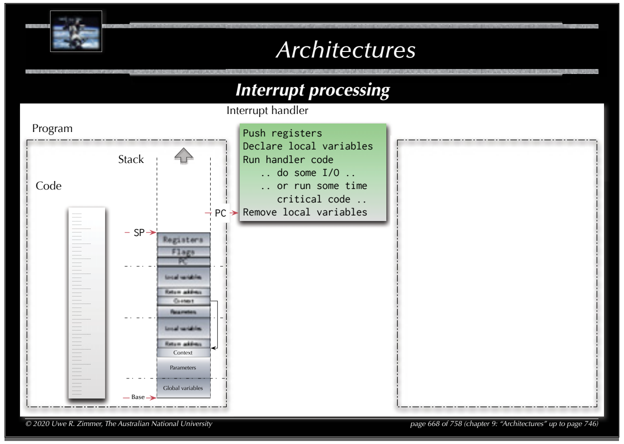
666



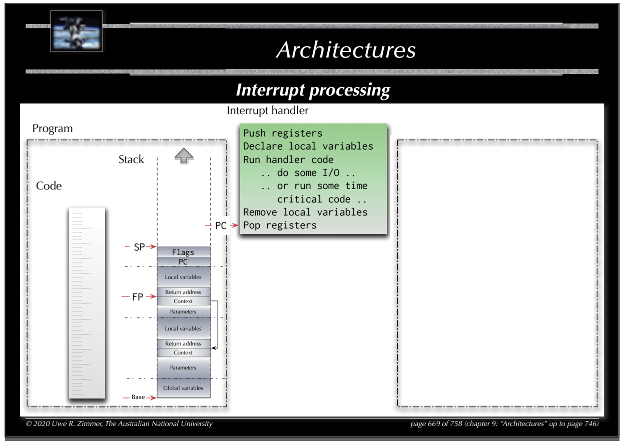
667



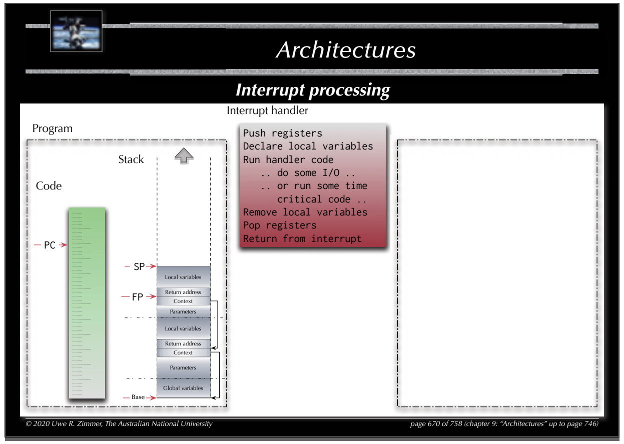
668



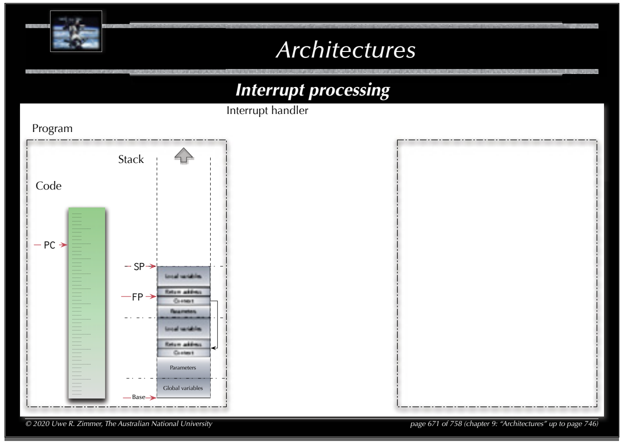
669



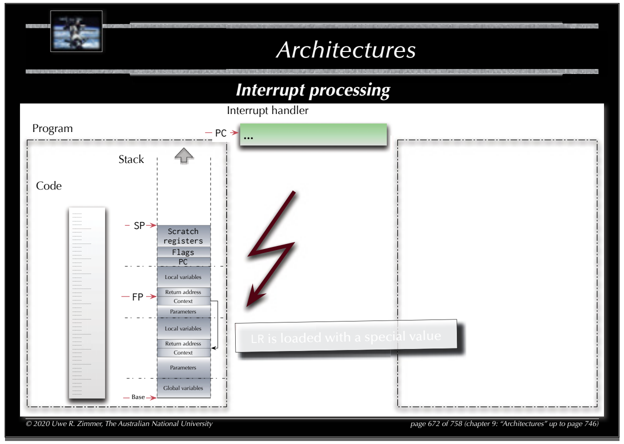
670



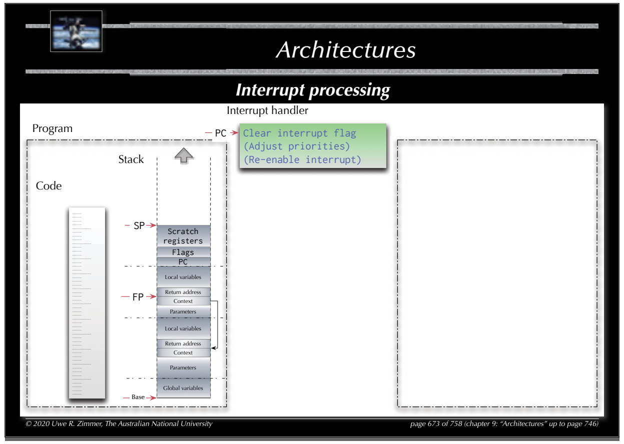
671



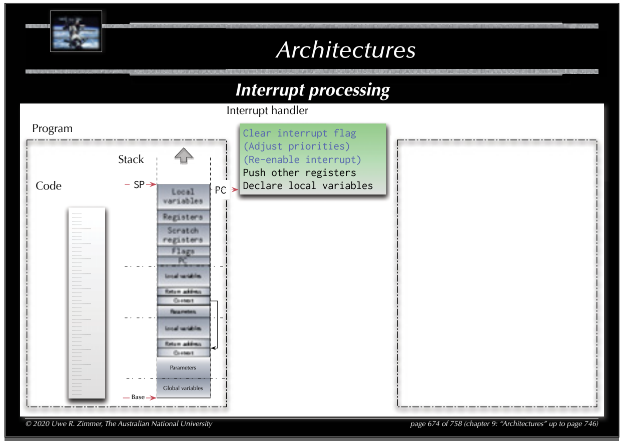
672



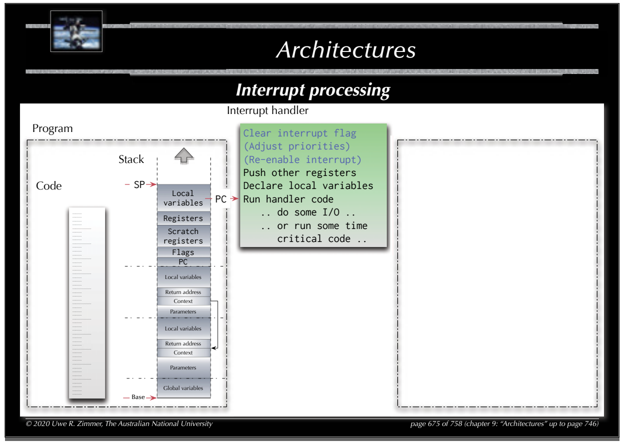
673



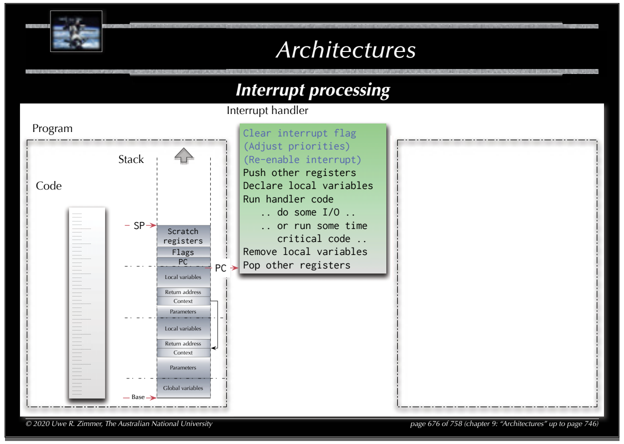
674



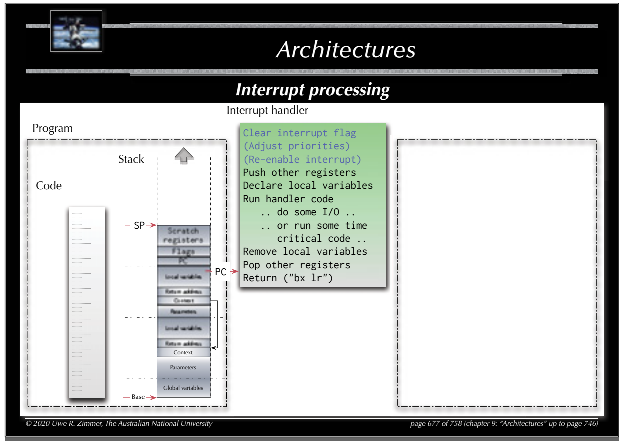
675



676

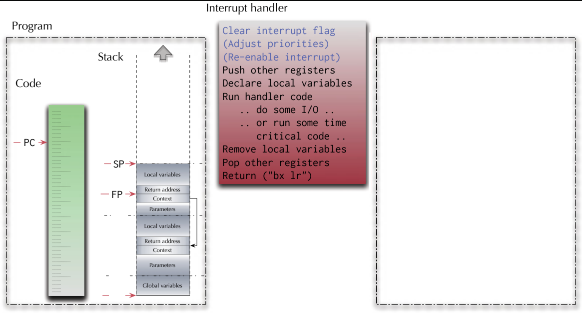


677



Architectures

Interrupt processing



Architectures

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?

Architectures

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?

- ☞ Can we have one of those? Busy! Do Not Disturb!

Architectures

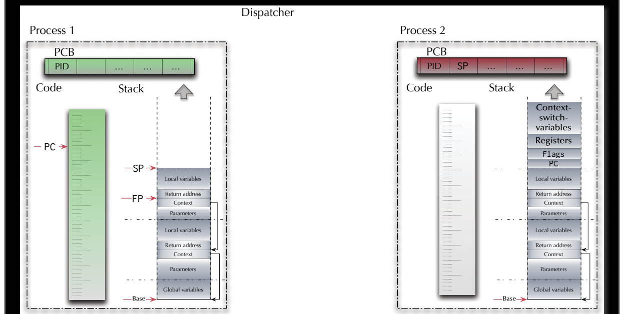
Multiple programs

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

- ☞ Can we then also have multiple "main" programs?

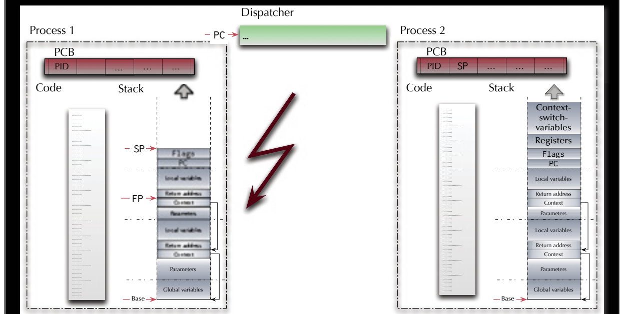
Architectures

Context switch



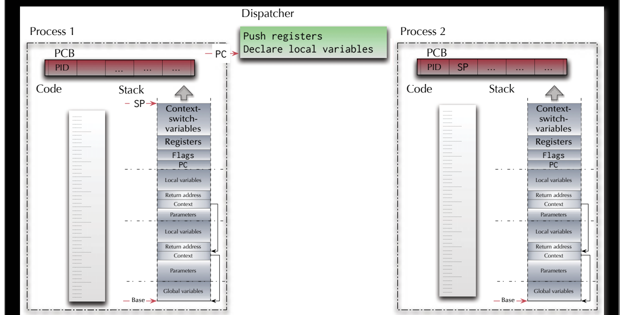
Architectures

Context switch



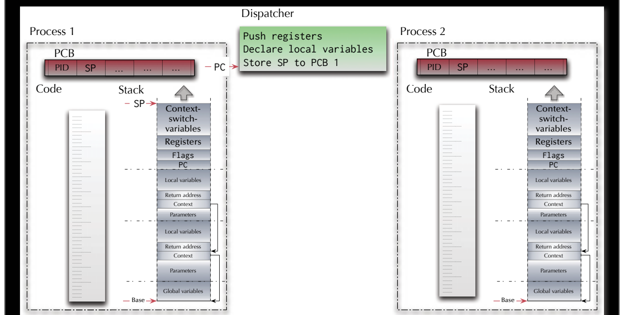
Architectures

Context switch



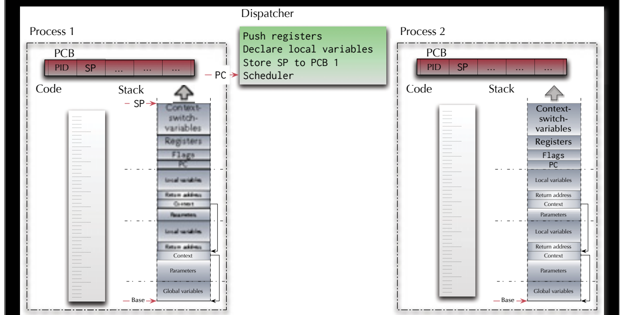
Architectures

Context switch



Architectures

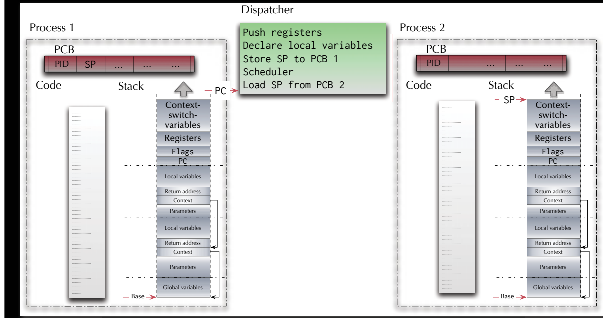
Context switch



687

Architectures

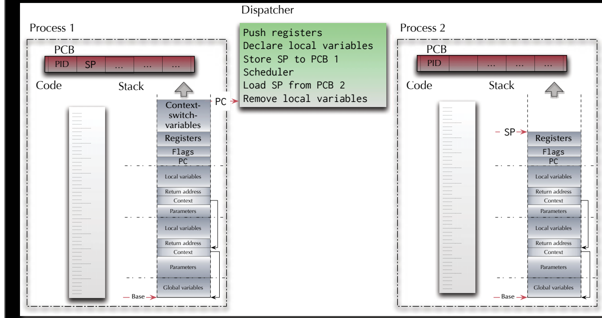
Context switch



688

Architectures

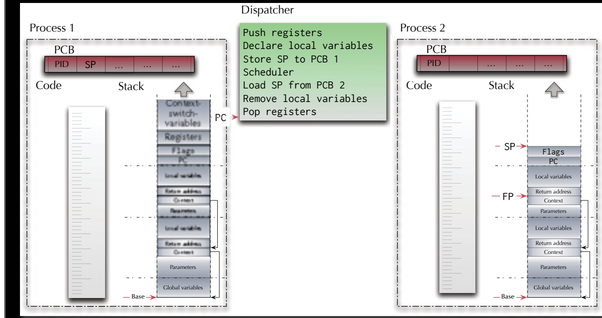
Context switch



689

Architectures

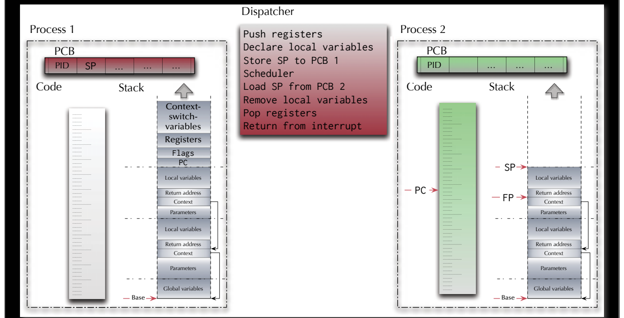
Context switch



690

Architectures

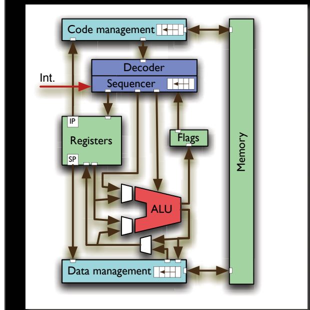
Context switch



691

Architectures

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.

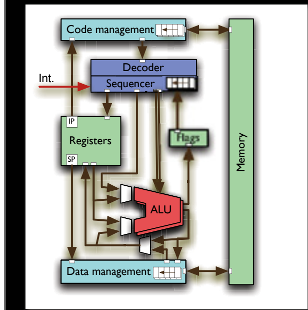
☞ Same latency, yet higher throughput.

☞ (Conditional) branches might break the pipelines
  ☞ Branch predictors become essential.

692

Architectures

Processor Architectures



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.

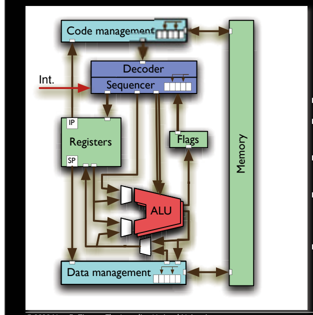
☞ Same latency, yet even higher throughput.

☞ Compilers need to be aware of the options.

693

Architectures

Processor Architectures



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.

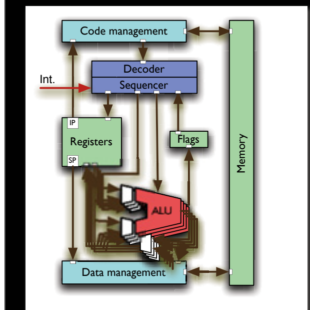
☞ Works better if the presented code sequence has more independent instructions and fewer conditional branches.

☞ This hardware will require (extensive) code optimization to be fully utilized.

694

Architectures

Processor Architectures



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

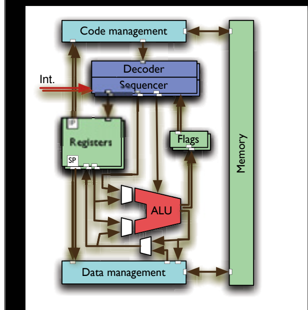
Graphics processor as a vector unit.

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

695

Architectures

Processor Architectures



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.

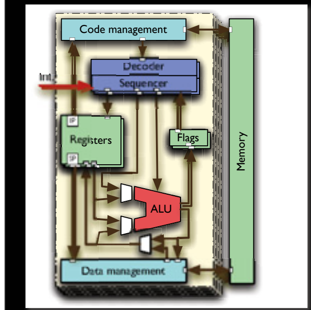
☞ **Requires programming languages with implicit or explicit concurrency.**

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

Architectures

Processor Architectures

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.**

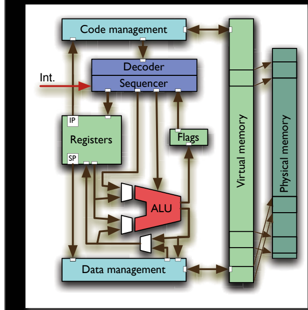
☞ **Requires programming languages with implicit or explicit concurrency.**

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

Architectures

Processor Architectures

Virtual memory



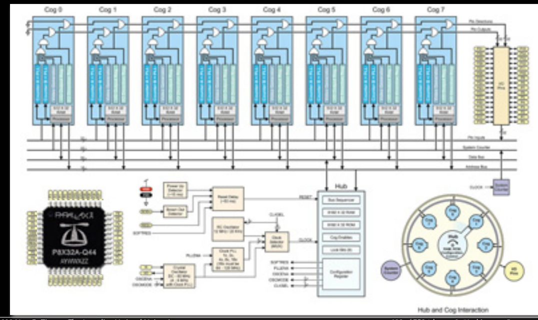
Translates logical memory addresses into physical memory addresses and provides memory protection features.

- Does not introduce concurrency by itself.

☞ Is still essential for concurrent programming as hardware memory protection guarantees memory integrity for individual processes / threads.

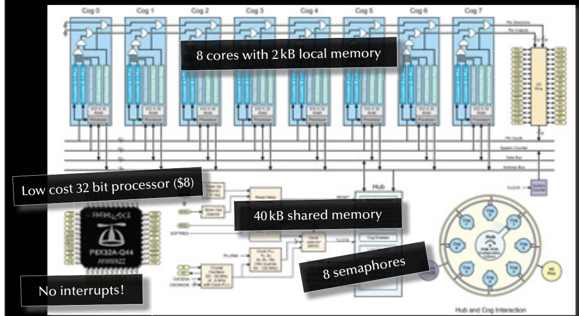
Architectures

Alternative Processor Architectures: Parallax Propeller



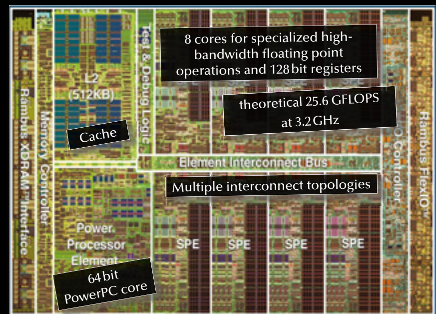
Architectures

Alternative Processor Architectures: Parallax Propeller (2006)



Architectures

Alternative Processor Architectures: IBM Cell processor (2001)



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 GB/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, 2.6 GB/sec

Architectures

Vector Machines

Vectorization

Buzzword collection: AltiVec, SPE, MMX, SSE, NEON, SPU, AVX, …

$$a \cdot \vec{v} = a \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a \cdot x \\ a \cdot y \\ a \cdot z \end{pmatrix}$$

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.

Architectures

Vector Machines

Vectorization

$$a \cdot \vec{v} = a \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a \cdot x \\ a \cdot y \\ a \cdot z \end{pmatrix}$$

Function is **"promoted"**

```
const Index = {1 .. 100000000},
      Vector_1 : [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**

Architectures

Vector Machines

Reduction

$$\vec{v_1} = \vec{v_2} \Rightarrow \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} = \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} \Rightarrow (x_1 = x_2) \wedge (y_1 = y_2) \wedge (z_1 = z_2)$$

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

Translates into **CPU-level vector operations**

$\wedge$-chain is evaluated lazy sequentially.

705

Vector Machines

Reduction

$$\overrightarrow{v_1} = \overrightarrow{v_2} \Rightarrow \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} = \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} \Rightarrow (x_1 = x_2) \wedge (y_1 = y_2) \wedge (z_1 = z_2)$$

```
const Index = {1 .. 100000000},
      Vector_1, Vector_2 : [Index] real = 1.0;
proc Equal (v1, v2) : bool
   {return && reduce (v1 == v2);}
```

$\wedge$-operations are evaluated in a **concurrent divide-and-conquer** (binary tree) structure.

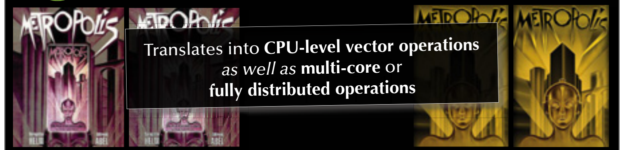
Function is **"promoted"**

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

706

Vector Machines

General Data-parallelism



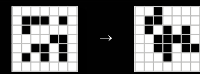
Translates into **CPU-level vector operations** *as well as* **multi-core** or **fully distributed operations**

```
const Mask : [1 .. 3, 1 .. 3] real = ((0, -1, 0), (-1, 5, -1), (0, -1, 0));
proc Unsharp_Mask (P, (i, j) : index (Image)) : real
   {return + reduce (Mask * P [i - 1 .. i + 1, j - 1 .. j + 1]);}
const Sharpened_Picture = forall px in Image do Unsharp_Mask (Picture, px);
```


707

Vector Machines

General Data-parallelism



Cellular automaton transitions from a state into the next state ′:
→ ′ ⇔ ∀ ∈ : → ′ = (,), i.e. all cells of a state transition *concurrently* into new cells by following a rule .

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

John Conway's **Game of Life** rule:

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


708

Operating Systems

What is an operating system?

709

What is an operating system?

1. A virtual machine!

... offering a more comfortable and safer environment

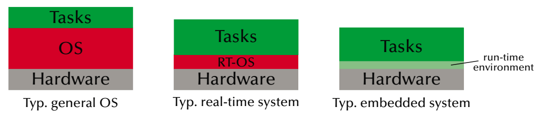
(e.g. memory protection, hardware abstraction, multitasking, ...)

710

What is an operating system?

1. A virtual machine!

... offering a more comfortable and safer environment



711

What is an operating system?

2. A resource manager!

... coordinating access to hardware resources

712

What is an operating system?

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!

713

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, runable 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)

Architectures

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

Architectures

Types of current operating systems

Personal computing systems, workstations, and workgroup servers:

- late 70s: Workstations starting by porting UNIX or VMS to 'smaller' computers.
- 80s: PCs starting with almost none of the classical OS-features and services, but with an user-interface (MacOS) and simple device drivers (MS-DOS)

☞ last 20 years: evolving and expanding into current general purpose OSs, like for instace:
- Solaris (based on SVR4, BSD, and SunOS)
- LINUX (open source UNIX re-implementation for x86 processors and others)
- current Windows (proprietary, partly based on Windows NT, which is 'related' to VMS)
- MacOS X (Mach kernel with BSD Unix and a proprietary user-interface)

- Multiprocessing is supported by all these OSs to some extent.
- None of these OSs are suitable for embedded systems, although trials have been performed.
- None of these OSs are suitable for distributed or real-time systems.

Architectures

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

  or

  - asymmetrical: only one CPU carries the full operating system, the others are operated by small operating system stubs to transfer code or tasks.

Architectures

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 multiplied in order to
  - guarantee availability (hot stand-by)
  - or to increase throughput (heavy duty servers)

Architectures

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?

Architectures

Types of current operating systems

Real-time operating systems

- ~~Fast context switches?~~ should be fast anyway
- ~~Small size?~~ should be small anyway
- ~~Quick response to external interrupts?~~ not 'quick', but predictable
- ~~Multitasking?~~ often, not always
- ~~'low level' programming interfaces?~~ needed in many operating systems
- ~~Interprocess communication tools?~~ needed in almost all operating systems
- ~~High processor utilization?~~ fault tolerance builds on redundancy!

Architectures

Types of current operating systems

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

Architectures

Types of current operating 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!

Architectures

What is an operating system?

Is there a standard set of features for operating systems?

723

Architectures

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.

724

Architectures

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?

725

Architectures

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

726

Architectures

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?

727

Architectures

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

728

Architectures

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

729

Architectures

Typical features of operating systems

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...

730

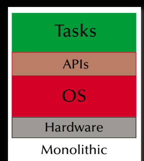
Architectures

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



e.g. most early UNIX systems,
MS-DOS (80s), Windows (all non-NT based versions)
MacOS (until version 9), and many others...

731

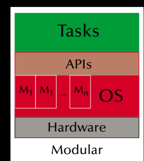
Architectures

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

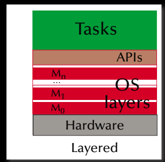
732

Architectures

Typical structures of operating systems

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



Layered

e.g. some current UNIX implementations (e.g. Solaris) to a certain degree, many research OSs (e.g. ‘THE system’, Dijkstra ‘68)

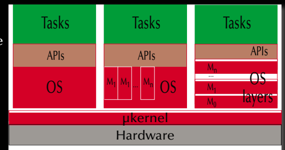
733

Architectures

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



μkernel, virtual machine

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

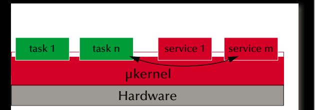
734

Architectures

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



μkernel, client server structure

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

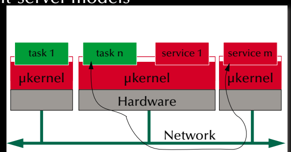
735

Architectures

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: locally and through a network
- highly modular and flexible
- servers can be redundant and easily replaced
- possibly reduced efficiency through increased communications



μkernel, distributed systems

e.g. Java engines, distributed real-time operating systems, current distributed OSs research projects

736

Architectures

UNIX

UNIX features

- Hierarchical file-system (maintained via ‘mount’ and ‘unmount’)
- 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

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

737

Architectures

UNIX

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

738

Architectures

UNIX

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

Frequent usage:

```
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 */
}
```


739

Architectures

UNIX

Synchronization in UNIX ☞ Signals

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

pid_t id;

void catch_stop (int sig_num)
{
  /* do something with the signal */
}
```

```
id = fork ();
if (id == 0) {
  signal (SIGSTOP, catch_stop);
  pause ();
  exit (0);
} else {
  kill (id, SIGSTOP);
  pid = wait ();
}
```


740

Architectures

UNIX

Message passing in UNIX ☞ Pipes

```
int data_pipe [2], c, rc;
if (pipe (data_pipe) == -1) {
  perror (“no pipe“); exit (1);
}
if (fork () == 0) { // child
  close (data_pipe [1]);
  while ((rc = read
    (data_pipe [0], &c, 1)) >0) {
    putchar (c);
  }
  if (rc == -1) {
    perror (“pipe broken“);
    close (data_pipe [0]); exit (1);}
  close (data_pipe [0]); exit (0);
```

```
} else { // 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 ();
}
```

741

Architectures

UNIX

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

742

Architectures

UNIX

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} [connect | listen` ➠ `accept` ➠ `read | write` ➠ `[close | shutdown]`
- **Client side:** `socket` ➠ `bind` ➠ `connect` ➠ `write | read` ➠ `[close | shutdown]`

743

Architectures

POSIX

Portable Operating System Interface for Unix

- 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!

744

Architectures

POSIX - some of the relevant standards...

1003.1 12/01	OS Definition	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, ...
1003.1b 10/93	Real-time Extensions	real-time signals, priority scheduling, timers, asynchronous I/O, prioritized I/O, synchronized I/O, file sync, mapped files, memory locking, memory protection, message passing, semaphore, ...
1003.1c 6/95	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
1003.1d 10/99	Additional Real-time Extensions	new process create semantics (spawn), sporadic server scheduling, execution time monitoring of processes and threads, I/O advisory information, timeouts on blocking functions, device control, and interrupt control
1003.1j 1/00	Advanced Real-time Extensions	typed memory, nanosleep improvements, barrier synchronization, reader/writer locks, spin locks, and persistent notification for message queues
1003.21 -/-	Distributed Real-time	buffer management, send control blocks, asynchronous and synchronous operations, bounded blocking, message priorities, message labels, and implementation protocols

745

Architectures

POSIX - 1003.1b/c

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

746

Architectures

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