



## Operating System & Networks 2003

Uwe R. Zimmer – International University Bremen



### Operating Systems & Networks

what is offered here?

Overviews, Paths, Definitions, Terminology,  
Foundations, Methods, Algorithms  
Realities,  
Current research trends, Projects,  
Perspectives,  
... and some theory

into/for/about *Operating Systems & Networks*

© 2003 Uwe R. Zimmer, International University Bremen

Page 2 of 432 (chapter 0: to 12)



### Operating Systems & Networks

who could be interested in this?

anybody who ...

... would like to see  
how rich, diverse and deep the real world of operating systems goes

... would like to learn how to create predictability  
and fault-tolerant operating systems

... would like to know more about the usage of  
95% of all  $\mu$ processors (and thus operating systems)

© 2003 Uwe R. Zimmer, International University Bremen

Page 3 of 432 (chapter 0: to 12)



### Operating Systems & Networks

who are these people? – introduction

This course will be given by

*Holger Kenn* for the networks sections

and

*Uwe R. Zimmer* for the operating systems sections



© 2003 Uwe R. Zimmer, International University Bremen

Page 4 of 432 (chapter 0: to 12)



### Operating Systems & Networks

how will this all be done?

☞ Lectures (320-202):

- 2 per week ... all the nice stuff and theory  
Tuesday, 8:00-9:15; Friday, 11-12:15 – all in Conrad Naber lecture hall

☞ Labs (Advanced CS lab), independent course, but related (320-222):

- 2 sessions per week ... all the rough stuff and practice  
Monday 15:30-19:30; Tuesday 15:30-19:30

☞ Resources:

- introduced in the lectures and collected on the course page:  
<http://www.faculty.iu-bremen.de/course/FundCS2/>  
... as well as schedules, slides, code, etc. pp. ... keep an eye on these pages!

☞ Assessment:

- Two exams, 50% each, one oral exam, one written exam – assignments for self-checking

© 2003 Uwe R. Zimmer, International University Bremen

Page 5 of 432 (chapter 0: to 12)



### Operating Systems & Networks

#### Topics in operating systems

1. Introduction
2. Hardware basics
3. Processes
4. Memory management

© 2003 Uwe R. Zimmer, International University Bremen

Page 6 of 432 (chapter 0: to 12)



### Operating Systems & Networks

Table of contents

## 2. Hardware Fundamentals

- General computer architecture
- CPU
  - Registers
  - Traps/Interrupts & protected modes
- Memory
  - General memory layout
  - Caching
- I/O systems
  - I/O controllers, I/O buses, device programming
- Some examples of  $\mu$ processors
  - Small scale  $\mu$ controller (68HC05)
  - Full scale integrated processor (MCP565)

© 2003 Uwe R. Zimmer, International University Bremen

Page 7 of 432 (chapter 0: to 12)



### Operating Systems & Networks

Table of contents

## 3. Processes

- Processes and threads
  - Architectures, definitions, process states
- Synchronization
  - Shared memory based synchronization
  - Message based synchronization
- Deadlocks
  - Detection, avoidance, and prevention (& recovery)
- Scheduling
  - Basic performance based scheduling
  - Basic predictable scheduling
  - Aperiodic, sporadic, and synchronized tasks

© 2003 Uwe R. Zimmer, International University Bremen

Page 8 of 432 (chapter 0: to 12)



### Operating Systems & Networks

Table of contents

## 3.1 Synchronization methods

### • Shared memory based synchronization

- Semaphores ☞ 'C, POSIX – Dijkstra
- Conditional critical regions ☞ Edison (experimental)
- Monitors ☞ Modula-1, Mesa – Dijkstra, Hoare, ...
- Mutexes & conditional variables ☞ POSIX
- Synchronized methods ☞ Real-time Java
- Protected objects ☞ Ada95

### • Message based synchronization

- Asynchronous messages ☞ e.g. POSIX, ...
- Synchronous messages ☞ e.g. Ada95, CHILL, Occam2
- Remote invocation, remote procedure call ☞ e.g. Ada95, ...
- Synchronization in distributed systems ☞ e.g. CORBA, ...

© 2003 Uwe R. Zimmer, International University Bremen

Page 9 of 432 (chapter 0: to 12)

## 3.2 Deadlocks

- **Ignorance & recovery**
  - ‘kill some seemingly persistently blocked processes from time to time’ (exasperation)
- **Deadlock detection & recovery**
  - multiple methods for detection, e.g. resource allocation graphs, Banker’s algorithm
  - recovery is mostly ‘ugly’
- **Deadlock avoidance**
  - check system safety before allocating resources, e.g. Banker’s algorithm
- **Deadlock prevention**
  - eliminate one of the pre-conditions for deadlocks

## 3.3 Scheduling

- **Basic performance based scheduling**
  - $C_f$  is not known: first-come-first-served (FCFS), round robin (RR), and feedback-scheduling
  - $C_f$  is known: shortest job first (SJF), highest response ration first (HRRF), shortest remaining time first (SRTF)-scheduling
- **Basic predictable scheduling**
  - Fixed Priority Scheduling (FPS) with Rate Monotonic (RMPO)
  - Earliest Deadline First (EDF)
- **Real-world extensions**
  - Aperiodic, sporadic, soft real-time tasks
  - Synchronized talks (priority inheritance, priority ceiling protocols)

## 4. Memory

- **Requirements & hardware structures**
  - MMU features & requirements
- **Partitioning, segmentation, paging & virtual memory**
  - Simple segmentation
  - Simple paging, multi-level paging, combined segmentation & paging
  - Translation look aside buffers
  - Hashed tables, Inverted page tables
- **Virtual memory management algorithms**
  - Fetching & placement
  - Replacement
  - Resident set management
  - Cleaning
  - Load control



## Introduction

Uwe R. Zimmer – International University Bremen

## References for this chapter

- |  |  |
|--|--|
| <p>[Silberschatz01]<br/>Abraham Silberschatz, Peter Bear Galvin,<br/>Greg Gagne<br/><i>Operating System Concepts</i><br/>John Wiley &amp; Sons, Inc., 2001</p> <p>[Stallings2001]<br/>William Stallings<br/><i>Operating Systems</i><br/>Prentice Hall, 2001</p> | <p>[Tanenbaum97]<br/>Andrew S. Tanenbaum, Albert S. Woodhull<br/><i>Operating Systems: Design and Implementation</i><br/>Prentice Hall, 1997</p> <p>[Tanenbaum95]<br/>Andrew S. Tanenbaum<br/><i>Distributed Operating Systems</i><br/>Prentice Hall, 1995</p> |
|--|--|

all references and some links are available on the course page

## What are operating system based on?

## Hardware environments / configurations:

- stand-alone, universal, single-processor machines
- symmetrical multiprocessor-machines
- local distributed systems
- open, web-based systems
- dedicated/embedded computing

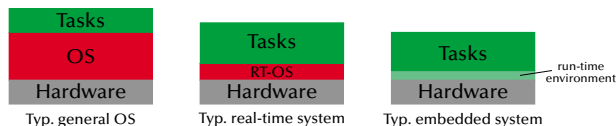
What is the common ground for operating systems?

## What is an operating system?

## What is an operating system?

## 1. A virtual machine!

... offering a more comfortable, robust, reliable, flexible ... machine



## What is an operating system?

## 2. A resource manager!

... dealing with all sorts of devices and coordinating access

Operating systems deal with

- processors,
  - memory
  - mass storage
  - communication channels
  - devices  
(timers, special purpose processors, interfaces, ...)
- and many tasks/processes/programs, which are applying for access to these resources

## What is an operating system?

Is there a standard set of features for an operating system?

- no, the term ‘operating systems’ covers 4KB kernels, as well as 1GB installations of general purpose OSs.

Is there a minimal set of features?

- almost, memory management, process management and inter-process communication/synchronization will be considered essential in most systems.

Is there always an explicit operating system?

- no, some languages and development systems operate with stand-alone run-time-environments.

# Operating Systems & Networks

## The evolution of operating systems

- in the beginning: single user, single program, single task, serial processing  $\Rightarrow$  *no OS*
- 50s: System monitors / batch processing
  - $\Rightarrow$  the monitor ordered the sequence of jobs and triggered their sequential execution
- 50s-60s: Advanced system monitors / batch processing:
  - $\Rightarrow$  the monitor is handling interrupts and timers
  - $\Rightarrow$  first support for memory protection
  - $\Rightarrow$  first implementations of privileged instructions (accessible by the monitor only).
- early 60s: Multiprogramming systems:
  - $\Rightarrow$  employ the long device I/O delays for switches to other, runnable programs
- early 60s: Multiprogramming, time-sharing systems:
  - $\Rightarrow$  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 (MacOS) 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)

© 2003 Uwe R. Zimmer, International University Bremen

Page 19 of 432 (chapter 1: to 89)

# Operating Systems & Networks

## 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)

Currently: standard consumer computers come with

- High speed network connectors (e.g. GB-ethernet)
- Wireless LAN (e.g. IEEE802.11)
- Local device bus-system (e.g. firewire)
- Wireless local device network (e.g. bluetooth)
- Infrared communication (e.g. IrDA)
- Modem

© 2003 Uwe R. Zimmer, International University Bremen

Page 20 of 432 (chapter 1: to 89)

# Operating Systems & Networks

## Types of current operating systems

Personal computing systems and workstations:

- 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)
  - $\Rightarrow$  last 20 years: evolving and expanding into current general purpose OSs:
    - 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 an proprietary user-interface)

- Multiprocessing is supported by all these OSs to some extend.
- None of these OSs is very suitable for embedded systems, also trials have been performed.
- All of these OSs are not suitable at all for distributed or real-time systems.

© 2003 Uwe R. Zimmer, International University Bremen

Page 21 of 432 (chapter 1: to 89)

# Operating Systems & Networks

## 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.

© 2003 Uwe R. Zimmer, International University Bremen

Page 22 of 432 (chapter 1: to 89)

# Operating Systems & Networks

## 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)

© 2003 Uwe R. Zimmer, International University Bremen

Page 23 of 432 (chapter 1: to 89)

# Operating Systems & Networks

## Types of current operating systems

Real-time operating systems

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

© 2003 Uwe R. Zimmer, International University Bremen

Page 24 of 432 (chapter 1: to 89)

# Operating Systems & Networks

## Types of current operating systems

Real-time operating systems requesting ...

- $\Rightarrow$  the logical correctness of the results as well as
- $\Rightarrow$  **the correctness of the time, when the results are delivered**

$\Rightarrow$  **Predictability!**  
(not performance!)

- $\Rightarrow$  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  $\Rightarrow$  reactive systems

© 2003 Uwe R. Zimmer, International University Bremen

Page 25 of 432 (chapter 1: to 89)

# Operating Systems & Networks

## Types of current operating systems

Embedded operating systems

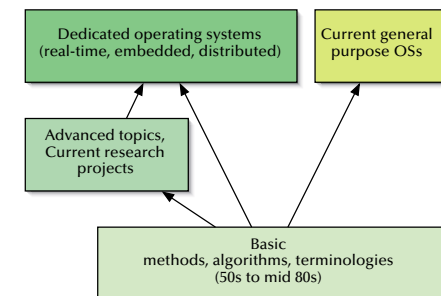
- usually real-time systems, often hard real-time systems
- very small footprint (often a few KBs)
- none or limited user-interaction
- $\Rightarrow$  90-95% of all processors are working here!

© 2003 Uwe R. Zimmer, International University Bremen

Page 26 of 432 (chapter 1: to 89)

# Operating Systems & Networks

## Roots of current commercial operating systems



© 2003 Uwe R. Zimmer, International University Bremen

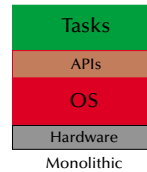
Page 27 of 432 (chapter 1: to 89)

## Operating Systems & Networks

### 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)
- ⇒ may reach very high efficiency



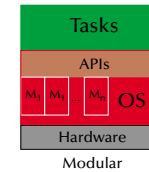
e.g. most early UNIX implementations (70s), MS-DOS (80s), Windows (basically all versions besides NT and NT-based editions), MacOS (until version 9),

## Operating Systems & Networks

### 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 very high efficiency



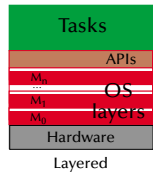
e.g. current LINUX versions

## Operating Systems & Networks

### 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



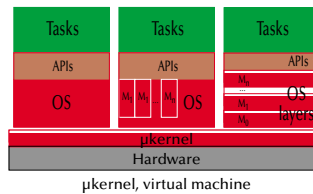
e.g. some current UNIX implementations (e.g. Solaris) to a certain degree, many research OSs (e.g. 'THE system', Dijkstra '68)

## Operating Systems & Networks

### Typical structures of operating systems

#### 'µkernels and 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 need to be ported.
- possibly reduced efficiency through increased communications



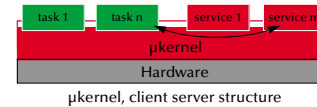
e.g. wide spread concept: as early as the CP/M, VM/370 ('79) or as recent as MacOS X (mach kernel + BSD unix)

## Operating Systems & Networks

### Typical structures of operating systems

#### 'µkernels and client-server models'

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



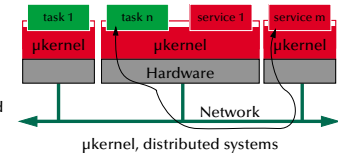
e.g. current µkernel research projects

## Operating Systems & Networks

### Typical structures of operating systems

#### 'µkernels and distributed systems'

- µkernel implements essential process, memory, and message handling
- all 'higher' services are user-level servers
- kernel ensures the reliable message passing between clients and servers: locally and via a communication system
- highly modular and flexible
- servers can be redundant and easily replaced
- possibly reduced efficiency through increased communications



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

## Operating Systems & Networks

### Basic programming styles

- **Imperative** (sequential) ⇒ Ada, JAVA, Eiffel, C, ...
- **Functional** (recursive) ⇒ Lisp, OCaml, ...
- **Declarative** (logic) ⇒ Prolog, ...
- **Data-flow machines** ⇒ Lustre, Signal, ...
- (hierarchical) **Finite state machines** ⇒ synchronous languages: Esterel, syncEifel, synERJY, ...

#### Programming styles alternatives

Imperative ↔ Functional ↔ Declarative ↔ Data-flow ↔ Finite state machines  
 Static ↔ Dynamic  
 Modular ↔ Concurrent ↔ Distributed  
 Synchronous ↔ Continuous time  
 Control oriented ↔ Data oriented

## Operating Systems & Networks

### Programming styles

#### What makes a language suitable for operating systems?

- **Precise expressions on machine level** ⇒ address physical memory + I/O
- **Concurrency** ⇒ support for tasking/threading
- **Distribution** ⇒ support for message passing or rpc
- **Reliability** ⇒ detect errors at compile-time or in the run-time environment
- **Large systems** ⇒ scalable, modular, or object-oriented + separate compilation
- **Predictability** ⇒ no operations which will lead to unforeseeable timing behaviours (e.g. garbage collection)

## Operating Systems & Networks

### Programming styles

#### Languages considered in this course

- C/C++ (for the lab-assignments)
- Ada95 (for your understanding)
- JAVA (for some distribution and object orientated features)
- POSIX (as the IEEE standard for (UNIX-) OS interfaces)
- ... others in places



Ada95 is a **standardized** (ISO/IEC 8652:1995(E)) ‘general purpose’ language with **core** language primitives for

- strong typing, separate compilation (specification and implementation), object-orientation,
- concurrency, monitors, rpcs, timeouts, scheduling, priority ceiling locks
- strong run-time environments

... and **standardized** language-**annexes** for

- additional real-time features, distributed programming, system-level programming, numeric, informations systems, safety and security issues.



## A crash course

... refreshing:

- specification and implementation (body) parts, basic types
- exceptions
- information hiding in specifications (‘private’)
- generic programming
- class-wide programming (‘tagged types’)
- monitors and synchronisation (‘protected’, ‘entries’, ‘selects’, ‘accepts’)
- abstract types and dispatching



## Basics

... introducing:

- specification and implementation (body) parts
- constants
- some basic types (integer specifics)
- some type attributes
- parameter specification



## A simple queue specification

```
package Queue_Pack_Simple is
  QueueSize : constant Positive := 10;
  type Element is new Positive range 1_000..40_000;
  type Marker is mod QueueSize;
  type List is array (Marker'Range) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Elements : List;
  end record;
  procedure Enqueue (Item: in Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
end Queue_Pack_Simple;
```



## A simple queue implementation

```
package body Queue_Pack_Simple is
  procedure Enqueue (Item: in Element; Queue: in out Queue_Type) is
  begin
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free - 1;
  end Enqueue;
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top - 1;
  end Dequeue;
end Queue_Pack_Simple;
```



## A simple queue test program

```
with Queue_Pack_Simple; use Queue_Pack_Simple;
procedure Queue_Test_Simple is
  Queue : Queue_Type;
  Item : Element;
begin
  Enqueue (2000, Queue);
  Dequeue (Item, Queue);
  Dequeue (Item, Queue); -- will produce an unpredictable result!
end Queue_Test_Simple;
```



## Exceptions

... introducing:

- exception handling
- enumeration types
- functional type attributes



## A queue specification with proper exceptions

```
package Queue_Pack_Exceptions is
  QueueSize : constant Integer := 10;
  type Element is (Up, Down, Spin, Turn);
  type Marker is mod QueueSize;
  type List is array (Marker'Range) of Element;
  type Queue_State is (Empty, Filled);
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    State : Queue_State := Empty;
    Elements : List;
  end record;
  procedure Enqueue (Item: in Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
  Queueoverflow, Queueunderflow : exception;
end Queue_Pack_Exceptions;
```



## A queue implementations with proper exceptions

```
package body Queue_Pack_Exceptions is
  procedure Enqueue (Item: in Element; Queue: in out Queue_Type) is
  begin
    if Queue.State = Filled and Queue.Top = Queue.Free then
      raise Queueoverflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Marker'Pred (Queue.Top);
    Queue.State := Filled;
  end Enqueue;
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    if Queue.State = Empty then
      raise Queueunderflow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Marker'Pred (Queue.Top);
    if Queue.Top = Queue.Free then Queue.State := Empty; end if;
  end Dequeue;
end Queue_Pack_Exceptions;
```



## A queue test program with proper exceptions

```

with Queue_Pack_Exceptions; use Queue_Pack_Exceptions;
with Ada.Text_IO; use Ada.Text_IO;

procedure Queue_Test_Exceptions is
  Queue : Queue_Type;
  Item : Element;

begin
  Enqueue (Turn, Queue);
  Dequeue (Item, Queue);
  Dequeue (Item, Queue); -- will produce a 'Queue underflow'

exception
  when Queueunderflow => Put ("Queue underflow");
  when Queueoverflow => Put ("Queue overflow");
end Queue_Test_Exceptions;

```

## Ada95

## Information hiding (private parts)

... introducing:

- private  $\Rightarrow$  assignments and comparisons are allowed
- limited private  $\Rightarrow$  entity cannot be assigned or compared

A queue *specification* with proper information hiding

```

package Queue_Pack_Private is
  QueueSize : constant Integer := 10;
  type Element is new Positive range 1..1000;
  type Queue_Type is limited private;

  procedure Enqueue (Item: in Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
  Queueoverflow, Queueunderflow : exception;

private
  type Marker is mod QueueSize;
  type List is array (Marker'Range) of Element;
  type Queue_State is (Empty, Filled);
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    State : Queue_State := Empty;
    Elements : List;
  end record;
end Queue_Pack_Private;

```

A queue *implementation* with proper information hiding

```

package body Queue_Pack_Private is
  procedure Enqueue (Item: in Element; Queue: in out Queue_Type) is
  begin
    if Queue.State = Filled and Queue.Top = Queue.Free then
      raise Queueoverflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free + 1;
    Queue.State := Filled;
  end Enqueue;

  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    if Queue.State = Empty then
      raise Queueunderflow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
    if Queue.Top = Queue.Free then Queue.State := Empty; end if;
  end Dequeue;
end Queue_Pack_Private;

```

## A queue test program with proper information hiding

```

with Queue_Pack_Private; use Queue_Pack_Private;
with Ada.Text_IO; use Ada.Text_IO;

procedure Queue_Test_Private is
  Queue, Queue_Copy : Queue_Type;
  Item : Element;

begin
  Queue_Copy := Queue;
  -- compiler-error: left hand of assignment must not be limited type
  Enqueue (Item => 1, Queue => Queue);
  Dequeue (Item, Queue);
  Dequeue (Item, Queue); -- will produce a 'Queue underflow'

exception
  when Queueunderflow => Put ("Queue underflow");
  when Queueoverflow => Put ("Queue overflow");
end Queue_Test_Private;

```

## Ada95

## Generic packages

... introducing:

- specification of generic packages
- instantiation of generic packages

A generic queue *specification*

```

generic
  type Element is private;
package Queue_Pack_Generic is
  QueueSize: constant Integer := 10;
  type Queue_Type is limited private;

  procedure Enqueue (Item: in Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
  Queueoverflow, Queueunderflow : exception;

private
  type Marker is mod QueueSize;
  type List is array (Marker'Range) of Element;
  type Queue_State is (Empty, Filled);
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    State : Queue_State := Empty;
    Elements : List;
  end record;
end Queue_Pack_Generic;

```

A generic queue *implementation*

```

package body Queue_Pack_Generic is
  procedure Enqueue (Item: in Element; Queue: in out Queue_Type) is
  begin
    if Queue.State = Filled and Queue.Top = Queue.Free then
      raise Queueoverflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free + 1;
    Queue.State := Filled;
  end Enqueue;

  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    if Queue.State = Empty then
      raise Queueunderflow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
    if Queue.Top = Queue.Free then Queue.State := Empty; end if;
  end Dequeue;
end Queue_Pack_Generic;

```

## A generic queue test program

```

with Queue_Pack_Generic;
with Ada.Text_IO; use Ada.Text_IO;

procedure Queue_Test_Generic is
  package Queue_Pack_Positive is
    new Queue_Pack_Generic (Element => Positive);
  use Queue_Pack_Positive;

  Queue : Queue_Type;
  Item : Positive;

begin
  Enqueue (Item => 1, Queue => Queue);
  Dequeue (Item, Queue);
  Dequeue (Item, Queue); -- will produce a 'Queue underflow'

exception
  when Queueunderflow => Put ("Queue underflow");
  when Queueoverflow => Put ("Queue overflow");
end Queue_Test_Generic;

```

## Object oriented programming I

... introducing:

- tagged types  $\Rightarrow$  the Ada-way to say that this type can be extended
- derivation of tagged types
- method overwriting
- usage of parent entities

An open queue base class *specification*

```
package Queue_Pack_Object_Base is
  QueueSize : constant Integer := 10;
  type Element is new Positive range 1..1000;
  type Marker is mod QueueSize;
  type List is array (Marker'Range) of Element;
  type Queue_State is (Empty, Filled);
  type Queue_Type is tagged record
    Top, Free : Marker := Marker'First;
    State : Queue_State := Empty;
    Elements : List;
  end record;

  procedure Enqueue (Item: in Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);

  Queueoverflow, Queueunderflow : exception;

end Queue_Pack_Object_Base;
```

An open queue base class *implementation*

```
package body Queue_Pack_Object_Base is
  procedure Enqueue (Item: in Element; Queue: in out Queue_Type) is
  begin
    if Queue.State = Filled and Queue.Top = Queue.Free then
      raise Queueoverflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free + 1;
    Queue.State := Filled;
  end Enqueue;

  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    if Queue.State = Empty then
      raise Queueunderflow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
    if Queue.Top = Queue.Free then Queue.State := Empty; end if;
  end Dequeue;

end Queue_Pack_Object_Base;
```

A derived open queue class *specification*

```
with Queue_Pack_Object_Base; use Queue_Pack_Object_Base;
package Queue_Pack_Object is
  type Ext_Queue_Type is new Queue_Type with record
    Reader : Marker := Marker'First;
    Reader_State : Queue_State := Empty;
  end record;

  procedure Enqueue (Item: in Element; Queue: in out Ext_Queue_Type);
  procedure Read_Queue (Item: out Element; Queue: in out Ext_Queue_Type);

end Queue_Pack_Object;
```

A derived open queue class *implementation*

```
package body Queue_Pack_Object is
  procedure Enqueue (Item: in Element; Queue: in out Ext_Queue_Type) is
  begin
    Enqueue (Item, Queue_Type (Queue));
    Queue.Reader_State := Filled;
  end Enqueue;

  procedure Read_Queue (Item: out Element; Queue: in out Ext_Queue_Type) is
  begin
    if Queue.Reader_State = Empty then
      raise Queueunderflow;
    end if;
    Item := Queue.Elements (Queue.Reader);
    Queue.Reader := Queue.Reader + 1;
    if Queue.Reader = Queue.Free then Queue.Reader_State := Empty; end if;
  end Read_Queue;

end Queue_Pack_Object;
```

## An open class test program

```
with Queue_Pack_Object_Base; use Queue_Pack_Object_Base;
with Queue_Pack_Object; use Queue_Pack_Object;
with Ada.Text_IO; use Ada.Text_IO;

procedure Queue_Test_Object is
  Queue : Ext_Queue_Type;
  Item : Element;

begin
  Enqueue (Item => 1, Queue => Queue);
  Read_Queue (Item, Queue);
  Enqueue (Item => 5, Queue => Queue);
  Dequeue (Item, Queue);
  Dequeue (Item, Queue);
  Dequeue (Item, Queue); -- will produce a 'Queue underflow'

exception
  when Queueunderflow => Put ("Queue underflow");
  when Queueoverflow => Put ("Queue overflow");
end Queue_Test_Object;
```

## Object oriented programming II

... introducing:

- private tagged types
- objects which are protected against their children also

An encapsulated queue base class *specification*

```
package Queue_Pack_Object_Base_Private is
  QueueSize : constant Integer := 10;
  type Element is new Positive range 1..1000;
  type Queue_Type is tagged limited private;

  procedure Enqueue (Item: in Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);

  Queueoverflow, Queueunderflow : exception;

private
  type Marker is mod QueueSize;
  type List is array (Marker'Range) of Element;
  type Queue_State is (Empty, Filled);
  type Queue_Type is tagged limited record
    Top, Free : Marker := Marker'First;
    State : Queue_State := Empty;
    Elements : List;
  end record;

end Queue_Pack_Object_Base_Private;
```

An encapsulated queue base class *implementation*

```
package body Queue_Pack_Object_Base_Private is
  procedure Enqueue (Item: in Element; Queue: in out Queue_Type) is
  begin
    if Queue.State = Filled and Queue.Top = Queue.Free then
      raise Queueoverflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free + 1;
    Queue.State := Filled;
  end Enqueue;

  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    if Queue.State = Empty then
      raise Queueunderflow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
    if Queue.Top = Queue.Free then Queue.State := Empty; end if;
  end Dequeue;

end Queue_Pack_Object_Base_Private;
```

## A derived encapsulated queue class *specification*

```
with Queue_Pack_Object_Base_Private; use Queue_Pack_Object_Base_Private;
package Queue_Pack_Object_Private is

  type Ext_Queue_Type is new Queue_Type with private;
  subtype Depth_Type is Positive range 1..QueueSize;

  procedure Look_Ahead (Item: out Element;
    Depth: in Depth_Type; Queue: in out Ext_Queue_Type);

private
  type Ext_Queue_Type is new Queue_Type with null record;
end Queue_Pack_Object_Private;
```

## A derived encapsulated queue class *implementation*

```
package body Queue_Pack_Object_Private is
  procedure Look_Ahead (Item: out Element;
    Depth: in Depth_Type; Queue: in out Ext_Queue_Type) is
    Storage : Queue_Type;
    ShuffleItem : Element;

  begin
    for I in 1..Depth - 1 loop
      Dequeue (ShuffleItem, Queue);
      Enqueue (ShuffleItem, Storage);
    end loop;
    Dequeue (Item, Queue);
    Enqueue (Item, Storage);
  end Look_Ahead;
end Queue_Pack_Object_Private;
```

(...)

```
Read_The_Rest:
begin
  for I in 1..QueueSize - Depth loop
    Dequeue (ShuffleItem, Queue);
    Enqueue (ShuffleItem, Storage);
  end loop;
exception
  when Queueunderflow => null; -- read the rest is done
end Read_The_Rest;
Restore_The_Queue:
begin
  for I in 1..QueueSize - Depth loop
    Dequeue (ShuffleItem, Storage);
    Enqueue (ShuffleItem, Queue);
  end loop;
exception
  when Queueunderflow => null; -- restore is done
end Restore_The_Queue;
end Look_Ahead;
end Queue_Pack_Object_Private;
```

## An encapsulated class test program

```
with Queue_Pack_Object_Base_Private; use Queue_Pack_Object_Base_Private;
with Queue_Pack_Object_Private; use Queue_Pack_Object_Private;
with Ada.Text_IO; use Ada.Text_IO;

procedure Queue_Test_Object_Private is
  Queue : Ext_Queue_Type;
  Item : Element;

begin
  Enqueue (Item => 1, Queue => Queue);
  Enqueue (Item => 1, Queue => Queue);
  Look_Ahead (Item => Item, Depth => 2, Queue => Queue);
  Enqueue (Item => 5, Queue => Queue);
  Dequeue (Item, Queue);
  Dequeue (Item, Queue);
  Dequeue (Item, Queue);
  Dequeue (Item, Queue); -- will produce a 'Queue underflow'

exception
  when Queueunderflow => Put ("Queue underflow");
  when Queueoverflow => Put ("Queue overflow");
end Queue_Test_Object_Private;
```

## Ada95

## Tasks & Monitors

... introducing:

- protected types
- tasks (definition, instantiation and termination)
- task synchronisation
- entry guards
- entry calls
- accept and selected accept statements

## A multitasking protected queue test program

```
with Queue_Pack_Protected; use Queue_Pack_Protected;
with Ada.Text_IO; use Ada.Text_IO;

procedure Queue_Test_Protected is
  Queue : Protected_Queue;

  task Producer is entry shutdown; end Producer;
  task Consumer is
    end Consumer;

  task body Producer is
    Item : Element;
    Got_It : Boolean;
  begin
    loop
      select
        accept shutdown; exit; -- main task loop
      else
        Get_Immediate (Item, Got_It);
        if Got_It then
          Queue.Enqueue (Item); -- task might be blocked here!
        else
          delay 0.1; --sec.
        end if;
      end select;
    end loop;
  end Producer;

  ...
```

## A protected queue *specification*

```
Package Queue_Pack_Protected is
  QueueSize : constant Integer := 10;
  subtype Element is Character;
  type Queue_Type is limited private;

  Protected type Protected_Queue is
    entry Enqueue (Item: in Element);
    entry Dequeue (Item: out Element);

  private
    Queue : Queue_Type;

  end Protected_Queue;

private
  type Marker is mod QueueSize;
  type List is array (Marker'Range) of Element;
  type Queue_State is (Empty, Filled);
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    State : Queue_State := Empty;
    Elements : List;
  end record;
end Queue_Pack_Protected;
```

## A protected queue *implementation*

```
package body Queue_Pack_Protected is
  protected body Protected_Queue is
    entry Enqueue (Item: in Element) when
      Queue.State = Empty or Queue.Top /= Queue.Free is
    begin
      Queue.Elements (Queue.Free) := Item;
      Queue.Free := Queue.Free - 1;
      Queue.State := Filled;
    end Enqueue;

    entry Dequeue (Item: out Element) when
      Queue.State = Filled is
    begin
      Item := Queue.Elements (Queue.Top);
      Queue.Top := Queue.Top + 1;
      if Queue.Top = Queue.Free then Queue.State := Empty; end if;
    end Dequeue;
  end Protected_Queue;
end Queue_Pack_Protected;
```

## A multitasking protected queue test program (cont.)

```
(...)

task body Consumer is
  Item : Element;
begin
  loop
    Queue.Dequeue (Item); -- task might be blocked here!
    Put ("Received: "); Put (Item); Put_Line ("!");
    if Item = 'q' then
      Put_Line ("Shutting down producer"); Producer.Shutdown;
      Put_Line ("Shutting down consumer"); exit; -- main task loop
    end if;
  end loop;
end Consumer;

begin
  null;
end Queue_Test_Protected;
```



*Abstract types & dispatching*

... introducing:

- abstract tagged types
- abstract subroutines
- concrete implementation of abstract types
- dispatching to different packages, tasks, and partitions according to concrete types

*An abstract queue specification*

```
package Queue_Pack_Abstract is
  subtype Element is Character;
  type Queue_Type is abstract tagged limited private;
  procedure Enqueue (Item: in Element; Queue: in out Queue_Type) is
    abstract;
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
    abstract;
private
  type Queue_Type is abstract tagged limited null record;
end Queue_Pack_Abstract;
```

*A concrete queue specification*

```
with Queue_Pack_Abstract; use Queue_Pack_Abstract;
package Queue_Pack_Concrete is
  QueueSize : constant Integer := 10;
  type Real_Queue is new Queue_Type with private;
  procedure Enqueue (Item: in Element; Queue: in out Real_Queue);
  procedure Dequeue (Item: out Element; Queue: in out Real_Queue);
  Queueoverflow, Queueunderflow : exception;
private
  type Marker is mod QueueSize;
  type List is array (Marker'Range) of Element;
  type Queue_State is (Empty, Filled);
  type Real_Queue is new Queue_Type with record
    Top, Free : Marker := Marker'First;
    State : Queue_State := Empty;
    Elements : List;
  end record;
end Queue_Pack_Concrete;
```

*A concrete queue implementation*

```
package body Queue_Pack_Concrete is
  procedure Enqueue (Item: in Element; Queue: in out Real_Queue) is
  begin
    if Queue.State = Filled and Queue.Top = Queue.Free then
      raise Queueoverflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free - 1;
    Queue.State := Filled;
  end Enqueue;
  procedure Dequeue (Item: out Element; Queue: in out Real_Queue) is
  begin
    if Queue.State = Empty then
      raise Queueunderflow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
    if Queue.Top = Queue.Free then Queue.State := Empty; end if;
  end Dequeue;
end Queue_Pack_Concrete;
```

*A multitasking dispatching test program*

```
with Queue_Pack_Abstract; use Queue_Pack_Abstract;
with Queue_Pack_Concrete; use Queue_Pack_Concrete;
procedure Queue_Test_Dispatching is
  type Queue_Class is access all Queue_Type'class;
  task Queue_Holder is -- could be on an individual partition
    entry Queue_Filled;
  end Queue_Holder;
  task Queue_User is -- could be on an individual partition
    entry Send_Queue (Remote_Queue: in Queue_Class);
  end Queue_User;
  (...)
end Queue_Test_Dispatching;
```

```
task body Queue_Holder is
  Local_Queue : Queue_Class;
  Item : Element;
begin
  Local_Queue := new Real_Queue; -- could be a different implementation!
  Queue_User.Send_Queue (Local_Queue);
  accept Queue_Filled do
    Dequeue (Item, Local_Queue.all); -- Item will be 'r'
  end Queue_Filled;
end Queue_Holder;
task body Queue_User is
  Local_Queue : Queue_Class;
  Item : Element;
begin
  Local_Queue := new Real_Queue; -- could be a different implementation!
  accept Send_Queue (Remote_Queue: in Queue_Class) do
    Enqueue ('r', Remote_Queue.all); -- potentially a rpc!
    Enqueue ('l', Local_Queue.all);
  end Send_Queue;
  Queue_Holder.Queue_Filled;
  Dequeue (Item, Local_Queue.all); -- Item will be 'l'
end Queue_User;
begin null; end Queue_Test_Dispatching;
```

*Ada95 language status*

- Established language standard with free and commercial compilers available for all major OSs.
- Stand-alone runtime environments for embedded systems (some are only available commercially).
- Special (yet non-standard) extensions (i.e. language reductions and proof systems) for extreme small footprint embedded systems or high integrity real-time environments available ⇨ Ravenscar profile systems.
- ⇨ has been used and is in use in numberless large scale projects (e.g. in the international space station, and in some spectacular crashes: e.g. Ariane 5)

*Portable Operating System Interface  
for Computing Environments*

- IEEE/ANSI Std 1003.1 and following
- Program Interface (API) [C Language]
- more than 30 different POSIX standards  
(a system is 'POSIX compliant', if it implements parts of just one of them!)

*POSIX – some of the real-time 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

## Frequently employed POSIX features include:

- **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

```
void timer_create(int num_secs, int num_nsecs)
{
    struct sigaction sa;
    struct sigevent sig_spec;
    sigset_t allsigs;
    struct itimerspec tmr_setting;
    timer_t timer_h;

    /* setup signal to respond to timer */
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_SIGINFO;
    sa.sa_sigaction = timer_intr;

    if (sigaction(SIGRTMIN, &sa, NULL) < 0)
        perror('sigaction');

    sig_spec.sigev_notify = SIGEV_SIGNAL;
    sig_spec.sigev_signo = SIGRTMIN;
}
```

## Languages used in this course

	Ada	RT-Java	C/C++	Posix
Predictability	*** (specific run-time env.)	--- (OOP)	implementation dependent	implementation dependent
low-level interfaces	***	-	**	**
Concurrency	***	**	---	**
Distribution	**	***	---	*
Error detection (compiler, tools)	** (strong typing)	**	---	---
Large systems	***	***	OOP C++ style (no support in C)	/

	POSIX 1003.1 (Base POSIX)	POSIX 1003.1b (Real-time extensions)	POSIX 1003.1c (Threads)
Solaris	Full support	Full support	Full support
IRIX	Conformant	Full support	Full support
LynxOS	Conformant	Full support	Conformant (Version 3.1)
QNX Neutrino	Full support	Partial support (no memory locking)	Full support
Linux	Full support	Partial support (no timers, no message queues)	Full support
VxWorks	Partial support (different process model)	Partial support (different process model)	Supported through third party product

```
/* create timer, which uses the REALTIME clock */
if (timer_create(CLOCK_REALTIME, &sig_spec, &timer_h) < 0)
    perror('timer create');

/* set the initial expiration and frequency of timer */
tmr_setting.it_value.tv_sec = 1;
tmr_setting.it_value.tv_nsec = 0;
tmr_setting.it_interval.tv_sec = num_secs;
tmr_setting.it_interval.tv_nsec = num_nsecs;
if (timer_settime(timer_h, 0, &tmr_setting, NULL) < 0)
    perror('settimer');

/* wait for signals */
sigemptyset(&allsigs);
while (1) {
    sigsuspend(&allsigs);
}

/* routine that is called when timer expires */
void timer_intr(int sig, siginfo_t *extra, void *cruft)
{
    /* perform periodic processing and then exit */
}
```

## Introduction to operating systems

- Features (and non-features) of operating system
- Common grounds for operating systems
- Historical perspectives
- Types of current operating systems
- Design principles for system software (monoliths &  $\mu$ kernels)
- Examples of languages considered for system level programming:
  - Java
  - Ada95
  - POSIX interfaces
  - C/C++

## POSIX is a 'C' standard ...

... but **bindings to other languages** are also (suggested) POSIX standards:

- Ada: 1003.5\*, 1003.24 (some PAR approved only, some withdrawn)
- Fortran: 1003.9 (6/92)
- Fortran90: 1003.19 (withdrawn)

... and there are POSIX standards for **task-specific POSIX profiles**, e.g.:

- Super computing: 1003.10 (6/95)
- **Realtime: 1003.13, 1003.13b** (3/98)
  - profiles 51-54: combinations of the above RT-relevant POSIX standards  $\Rightarrow$  RT-Linux
- **Embedded Systems: 1003.13a** (PAR approved only)

```
/* create timer, which uses the REALTIME clock */
if (timer_create(CLOCK_REALTIME, &sig_spec, &timer_h) < 0)
    perror('timer create');

/* set the initial expiration and frequency of timer */
tmr_setting.it_value.tv_sec = 1;
tmr_setting.it_value.tv_nsec = 0;
tmr_setting.it_interval.tv_sec = num_secs;
tmr_setting.it_interval.tv_nsec = num_nsecs;
if (timer_settime(timer_h, 0, &tmr_setting, NULL) < 0)
    perror('settimer');

/* wait for signals */
sigemptyset(&allsigs);
while (1) {
    sigsuspend(&allsigs);
}

/* routine that is called when timer expires */
void timer_intr(int sig, siginfo_t *extra, void *cruft)
{
    /* perform periodic processing and then exit */
}
```

remember the Pearl timers?  
AFTER 30 MIN ALL 5 MIN DURING 1 HAS ACTIVATE Help!



## References for this chapter

[Silberschatz01] – Chapter 2

Abraham Silberschatz, Peter Bear Galvin,  
Greg Gagne  
Operating System Concepts  
John Wiley & Sons, Inc., 2001

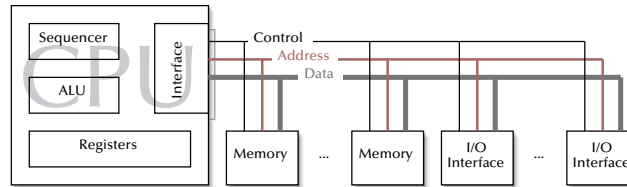
[Stallings2001] – Chapter 1

William Stallings  
Operating Systems  
Prentice Hall, 2001

all references and some links are available on the course page

## Hardware Fundamentals

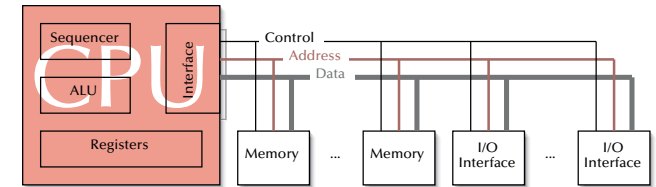
### A common computer architecture:



- Bus-systems carry device, address information and data (8-64 bit wide) as well as control lines in groups such as:
  - arbitration, synchronization, requests, interrupts, priorities

## Hardware Fundamentals

### The CPU



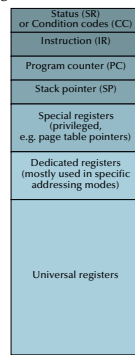
- CPU components relevant for this course:
  - register-set, sequencer ('normal operation'), interrupt controller, protected modes

## Hardware Fundamentals

### Register set

- SR: Status / Condition codes (CC), e.g.: privilege level, interrupt level, result of last operation
- IR: current instruction
- PC: Address of current (next) instruction
- SP: Top of stack address
- Special privileged registers, e.g.: page table entries, memory protection maps
- Dedicated registers, e.g.: registers which can be employed in some contexts only
- Universal registers: registers, which can be employed for any purpose (addressing, storage, index, parameters, ...)

#### Register structure

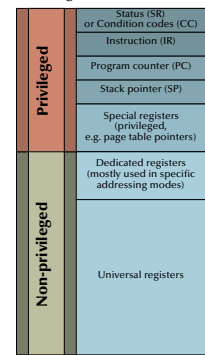


## Hardware Fundamentals

### Register set

- Often divided into a privileged and non-privileged section
- Switch from non-privileged to privileged mode only via traps or interrupts (later in this chapter)
- SR, IR, PC, SP + some general registers (or at least one 'accumulator') are found in all current processor designs
- Special and dedicated registers are not used in all architectures

#### Register structure

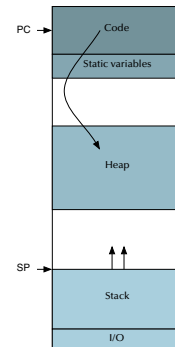


## Hardware Fundamentals

### Memory layout

- Classical usage of the RAM areas in most processors
- Main storage of data in
  - heap
  - stack
  - or local static
- depends on the usage of the programming language

#### Main memory layout

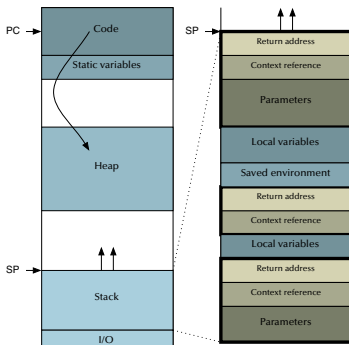


## Hardware Fundamentals

### Stack frames

- Every sub-program call leaves an entry on the stack with all relevant information:
  - parameters
  - context (not in 'C')
  - return address
- Parameters may be removed by:
  - the calling routine ('C')
  - or the called routine
- Special architectures support faster parameter passing (e.g. register-bands)

#### Main memory layout



## Hardware Fundamentals

### Privileged instructions

#### Purpose:

- prevent user level tasks from by-passing the operating system
- restrict access from user-level tasks to resources, which are managed by the operating system:
  - Memory
  - I/O
  - Structures which are used to administer memory or I/O access (e.g. special registers, MMUs, etc.)

#### Implementation:

- declare some instructions privileged
- implement two (or more) protection levels in the CPU
- allow changes to a higher privilege level by means of traps/exceptions/interrupts only.

## Asynchronism

### Interrupts

Required mechanisms for interrupt driven programming:

- Interrupt control:** grouping, encoding, prioritising, and en-/disabling interrupt sources
- Context switching:** mechanisms for cpu-state saving and restoring + task-switching
- Interrupt identification:** Interrupt vectors, interrupt states

hardware-supported

## Interrupts

## Interrupt control:

... at the individual device level

... at the system interrupt controller level

... at the operating system level

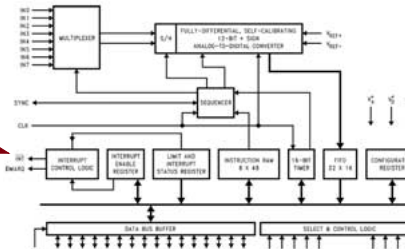
- beyond task-level (interrupt service routines)
- communicating interrupts to task
- transforming interrupts to signals

... at the language level

## LM12L458

(National Semiconductor)

Interrupt signal



☞ only one interrupt signal line available!

☞ in order to identify the interrupt reason, an additional read cycle is required!

## LM12L458

12-Bit + sign, 8 channel, A/D converter, controller and interface

## Controller features:

- Programmable acquisition times and conversion rates
- 32-word conversion FIFO
- Self-calibration and diagnostic mode
- 8- or 16-bit wide data bus microprocessor or DSP

Typ. applications:

- Data Logging
- Process Control

## LM12L458 – accessible registers

A4	A3	A2	A1	Purpose	Type	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	Instruction RAM (RAM Pointer = 00)	R/W	Acquisition Time				Watchdog	8/T2	Timer	Sync	V <sub>IN</sub>				V <sub>IN</sub>			
0	1	1	1																		
0	0	0	0	Instruction RAM (RAM Pointer = 01)	R/W	Don't Care				>R				Sign				Limit #1			
0	1	1	1																		
0	0	0	0	Instruction RAM (RAM Pointer = 10)	R/W	Don't Care				>R				Sign				Limit #2			
0	1	1	1																		
1	0	0	0	Configuration Register	R/W	Don't Care				DIAG	Test = 0	RAM Pointer	I/O Sel	Auto Zero	Chain Mask	Stand-by CAL	Full CAL	Auto Zero	Reset	Start	
1	0	0	1	Interrupt Enable Register	R/W	Number of Conversions in Conversion FIFO to Generate INT2				Sequencer Address to Generate INT1				INT7	Don't Care	INT5	INT4	INT3	INT2	INT1	INT0
1	0	1	0	Interrupt Status Register	R	Actual Number of Conversion Results in Conversion FIFO				Address of Sequencer Instruction being Executed				INST7	0	INST5	INST4	INST3	INST2	INST1	INST0
1	0	1	1	Timer Register	R/W	Timer Preset High Byte				Timer Preset Low Byte											
1	1	0	0	Conversion FIFO	R	Address or Sign				Conversion Data: MSBs				Conversion Data: LSBs							
1	1	0	1	Limit Status Register	R	Limit #2: Status				Limit #1: Status											

every entry in the **instruction RAM** consists of:

- **Loop** (1bit): indicates the last instruction and branches to the first one.
- **Pause** (1bit): halts the sequencer before this instruction.
- **V<sub>IN+</sub>, V<sub>IN-</sub>** (2\*3bit): select the input channels (000 selects ground in V<sub>IN-</sub>).
- **Sync** (1bit): wait for an external sync. signal before this instruction.
- **Timer** (1bit): wait for a preset 16-bit counter delay before this instruction.

## LM12L458 – instruction RAM

A4	A3	A2	A1	Purpose	Type	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	Instruction RAM (RAM Pointer = 00)	R/W	Acquisition Time				Watchdog	8/T2	Timer	Sync	V <sub>IN</sub>				V <sub>IN</sub>			
0	1	1	1																		
0	0	0	0	Instruction RAM (RAM Pointer = 01)	R/W	Don't Care				>R				Sign				Limit #1			
0	1	1	1																		
0	0	0	0	Instruction RAM (RAM Pointer = 10)	R/W	Don't Care				>R				Sign				Limit #2			
0	1	1	1																		

every entry in the **instruction RAM** consists of (cont.):

- **8/12** (1bit): selects the resolution (8 bit + sign or 12 bit + sign).
- **Watchdog** (1bit): activates comparisons with two programmed limits.
- **Acquisition time (D)** (4bit): the converter takes  $9 + 2D$  cycles (12bit mode) or  $2 + 2D$  cycles (8bit mode) to sample to input. Depends on the input resistance:  $D = 0.45 \cdot R_S[k\Omega] \cdot f_{CLK}[MHz]$  for 12 bit conversions.
- **Limits** (including sign and comparator): used for Watchdog operation.

## LM12L458 – instruction RAM

A4	A3	A2	A1	Purpose	Type	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	Instruction RAM (RAM Pointer = 00)	R/W	Acquisition Time				Watchdog	8/T2	Timer	Sync	V <sub>IN</sub>				V <sub>IN</sub>			
0	1	1	1																		

```
type ChannelPlus is (Ch0, Ch1, Ch2, Ch3, Ch4, Ch5, Ch6, Ch7);
type ChannelMinus is (Gnd, Ch1, Ch2, Ch3, Ch4, Ch5, Ch6, Ch7);
type Resolutions is (TwelveBit, EightBit);
type Aquisition_D is new Integer range 0..15; -- 9+2D (12bit), 2+2D (8bit)
```

```
for ChannelPlus use (Ch0 => 0, Ch1 => 1, Ch2 => 2, Ch3 => 3,
                    Ch4 => 4, Ch5 => 5, Ch6 => 6, Ch7 => 7);
for ChannelMinus use (Gnd => 0, Ch1 => 1, Ch2 => 2, Ch3 => 3,
                     Ch4 => 4, Ch5 => 5, Ch6 => 6, Ch7 => 7);
for Resolutions use (TwelveBit => 0, EightBit => 1);
```

```
type Instruction is record
    EndOfLoop, Pause, Sync, Timer, Watchdog : Boolean;
    Uplus : ChannelPlus;
    Uminus : ChannelMinus;
    Resolution : Resolutions;
    AquisitionTime : Aquisition_D;
end record;
```

## LM12L458 – instruction RAM

A4	A3	A2	A1	Purpose	Type	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	Instruction RAM (RAM Pointer = 00)	R/W	Acquisition Time				Watchdog	8/T2	Timer	Sync	V <sub>IN</sub>				V <sub>IN</sub>			
0	1	1	1																		

Units\_Per\_Word : constant Integer := Word\_Size / Storage\_Unit;

```
for Instruction use record
    EndOfLoop at 0*Units_Per_Word range 0.. 0;
    Pause at 0*Units_Per_Word range 1.. 1;
    Uplus at 0*Units_Per_Word range 2.. 4;
    Uminus at 0*Units_Per_Word range 5.. 7;
    Sync at 0*Units_Per_Word range 8.. 8;
    Timer at 0*Units_Per_Word range 9.. 9;
    Resolution at 0*Units_Per_Word range 10..10;
    Watchdog at 0*Units_Per_Word range 11..11;
    AquisitionTime at 0*Units_Per_Word range 12..15;
end record;
```

## LM12L458 – instruction RAM

A4	A3	A2	A1	Purpose	Type	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	Instruction RAM (RAM Pointer = 00)	R/W	Acquisition Time				Watchdog	8/T2	Timer	Sync	V <sub>IN</sub>				V <sub>IN</sub>			
0	1	1	1																		

```
for Instruction'Size use 16; -- Bits
for Instruction'Alignment use 2; -- Storage_Units (Bytes)
for Instruction'Bit_Order use High_Order_First;
```

```
type Instructions is array (0..7) of Instruction;
pragma Pack (Instructions);
```

```
ADC_Instructions : Instructions;
for ADC_Instructions'Address use To_Address (16#0000132D#);
```

## Operating Systems & Networks

### LM12L458 – instruction RAM

A4	A3	A2	A1	Purpose	Type	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0		
0	0	0	0	Instruction RAM	R/W	Acquisition Time				Watch-dog	8/12	Timer	Sync	V <sub>IN-</sub>				V <sub>IN+</sub>				Pause	Loop
0				(RAM Pointer = 00)																			
1	1	1	1																				

```
ADC_Instructions (0) := (EndOfLoop      => False,
                        Pause           => False,
                        Uplus           => Ch0,
                        Uminus          => Gnd,
                        Sync            => True,
                        Timer            => False,
                        Resolution       => EightBit,
                        Watchdog         => False,
                        AquisitionTime  => 10);

ADC_Instructions (1) := (EndOfLoop      => True,  -- last instruction
                        Pause           => False,
                        Uplus           => Ch1,
                        Uminus          => Ch2,
                        Sync            => False,
                        Timer            => False,
                        Resolution       => TwelveBit,
                        Watchdog         => False,
                        AquisitionTime  => 0);
```

## Operating Systems & Networks

### LM12L458 – instruction RAM

A4	A3	A2	A1	Purpose	Type	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0			
0	0	0	0	Instruction RAM	R/W	Acquisition Time				Watch-dog	8/12	Timer	Sync	V <sub>IN</sub> -			V <sub>IN</sub>			Pause	Loop			
0	0	0	0	(RAM Pointer = 00)																				
1	1	1	1																					

### Data structures in 'C':

```
enum ChannelPlus {Ch0=0, Ch1, Ch2, Ch3, Ch4, Ch5, Ch6, Ch7};
enum ChannelMinus {Gnd=0, Ch1, Ch2, Ch3, Ch4, Ch5, Ch6, Ch7};
enum Resolutions {TwelveBit=0, EightBit};

struct {
    unsigned int EndOfLoop      : 1;
    unsigned int Pause          : 1;
    ChannelPlus Uplus           : 3;
    ChannelMinus Uminus         : 3;
    unsigned int Sync           : 1;
    unsigned int Timer          : 1;
    Resolutions Resolution      : 1;
    unsigned int Watchdog       : 1;
    unsigned int AquisitionTime : 4;
} Instruction;
```

© 2003 Uwe R. Zimmer, International University Bremen

Page 110 of 432 (chapter 2: to 157)

## Operating Systems & Networks

### LM12L458 – instruction RAM

A4	A3	A2	A1	Purpose	Type	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	Instruction RAM	R/W	Acquisition		Watch-													
0				(RAM Pointer = 00)		Time		dog		8/T2	Timer	Sync			V <sub>IN-</sub>			V <sub>IN+</sub>		Pause	Loop
1	1	1	1																		

### Data structures in 'C':

```
struct {
    unsigned int EndOfLoop      : 1;
    unsigned int Pause          : 1;
    ChannelPlus Uplus           : 3;
    ChannelMinus Uminus         : 3;
    unsigned int Sync           : 1;
    unsigned int Timer          : 1;
    Resolutions Resolution      : 1;
    unsigned int Watchdog       : 1;
    unsigned int AquisitionTime : 4;
} Instruction;

Instruction InstructionsA[8];
InstructionsA *Instructions;
Instructions = 0x00001320;
```

© 2003 Uwe R. Zimmer, International University Bremen

Page 111 of 432 (chapter 2: to 157)

## Operating Systems & Networks

### LM12L458 – instruction RAM

A4	A3	A2	A1	Purpose	Type	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0		
0	0	0	0	Instruction RAM	R/W	Acquisition Time				Watch-dog	8/12	Timer	Sync	V <sub>IN-</sub>				V <sub>IN+</sub>				Pause	Loop
0	1		1	(RAM Pointer = 00)																			
1	1	1	1																				

### Data structures in 'C':

```
*Instructions (0).EndOfLoop      = 0;
*Instructions (0).Pause          = 0;
*Instructions (0).Uplus           = Ch0;
*Instructions (0).Uminus          = Gnd;
*Instructions (0).Sync            = 1;
*Instructions (0).Timer           = 0;
*Instructions (0).Resolution       = EightBit;
*Instructions (0).Watchdog         = 0;
*Instructions (0).AquisitionTime  = 10;
```

If this works, you were lucky two times:

- The compiler implemented the struct-fields in the intended places and order.
- The bit ordering in your device is the way the compiler assumed it.

© 2003 Uwe R. Zimmer, International University Bremen

Page 112 of 432 (chapter 2: to 157)

## Operating Systems & Networks

### LM12L458 – instruction RAM

A4	A3	A2	A1	Purpose	Type	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	Instruction RAM	R/W	Acquisition Time				Watch-dog	8/12	Timer	Sync	V <sub>IN-</sub>		V <sub>IN+</sub>			Pause	Loop	
0	0	1	1	(RAM Pointer = 00)																	
1	1	1	1																		

### ➤ Macro-Assembler style programming:

In order to produce portable code in 'C', it is necessary to set bits manually:

```
unsigned int setbits (unsigned int *r,          /* set n bits */
                     unsigned int n,          /* at position p */
                     unsigned int p,          /* to bitstring x */
                     unsigned int x)

{
    unsigned int mask;
    mask = ~(~0 << n);
    *r  &= ~(mask << p);
    *r |= (x & mask) << p;
    return (*r);
}
```

© 2003 Uwe R. Zimmer, International University Bremen

Page 113 of 432 (chapter 2: to 157)

## Operating Systems & Networks

### Asynchronism

### Interrupts

#### Interrupt control:

... at the individual device level

... at the system interrupt controller level

... at the operating system level

- beyond task-level (interrupt service routines)
- communicating interrupts to task
- transforming interrupts to signals

... at the language level

© 2003 Uwe R. Zimmer, International University Bremen

Page 114 of 432 (chapter 2: to 157)

## Operating Systems & Networks

### Asynchronism

### Interrupt service routines

(available only in some OSs, e.g. VxWorks)

#### Purpose:

- Allow full access to the interrupt controller (interrupt vectors, priorities).
- Change to an interrupt service routine in a predictable amount of time.

➤ *Cannot* operate on the level of threads or tasks!

➤ Limitations regarding the accessibility of some OS-facilities (task level system calls).

## Operating Systems & Networks

### Asynchronism

### Interrupt service routines

(available only in some OSs, e.g. VxWorks)

Some VxWorks OS entries:

intConnect	Connect a routine to an interrupt vector
intLevelSet	Set the interrupt mask level
intLock	Disable interrupts (besides NMI)
intUnlock	Enable interrupts
intVecBaseSet	Set the interrupt vector base address
intVecBaseGet	Get the interrupt vector base address
intVecSet	Set an interrupt vector
intVecGet	Get an interrupt vector

these calls are employed by the language run-time environment or used directly from 'C'-code

© 2003 Uwe R. Zimmer, International University Bremen

Page 116 of 432 (chapter 2: to 157)

## Operating Systems & Networks

### Asynchronism

### Interrupt service routines

(available only in some OSs, e.g. VxWorks)

Minimal hardware support (supplied by the cpu):

save essential CPU registers (IP, condition flags)  
jump to the vectorized interrupt service routine

Minimal wrapper (supplied by the operating system):

save remaining CPU registers (or switch to another register set)  
save stack-frame  
--> execute user level interrupts service code  
restore stack-frame  
restore CPU registers (or switch back to the former register set)  
restore IP

© 2003 Uwe R. Zimmer, International University Bremen

Page 117 of 432 (chapter 2: to 157)

© 2003 Uwe R. Zimmer, International University Bremen

Page 115 of 432 (chapter 2: to 157)



## Interrupt service routines

(available only in some OSs, e.g. VxWorks)

Interrupt service routine to task communication methods:

- **Shared memory and ring buffers:**  
most low level communication scheme (should be avoided)
- **Semaphore:** trigger a semaphore, where a task has been blocked before.
- **Monitors:**  
free a task, which is blocked at a monitor entry (standard Ada-method: protected object).
- **Message queues:** Send messages to a task (if queue is not full).
- **Pipes:** Write to a pipe (if pipe is not full).
- **Signals:** indicate an asynchronous task switch to the scheduler

## Interrupt service routines

(available only in some OSs, e.g. VxWorks)

Interrupt service routine to task communication methods:

- **Shared memory and ring buffers:**  
most low level communication scheme (should be avoided)
- **Semaphore:** trigger a semaphore, where a task has been blocked before.
- **Monitors:**  
free a task, which is blocked at a monitor entry (standard Ada-method: protected object).
- **Message queues:** Send messages to a task (if queue is not full).
- **Pipes:** Write to a pipe (if pipe is not full).
- **Signals:** indicate an asynchronous task switch to the scheduler

☞ in all of the above: the interrupt service routines *cannot* block!

## Interrupts ⇄ 'Signals'

Interrupt control:

... at the individual device level

... at the system interrupt controller level

... at the operating system level

- beyond task-level (interrupt service routines)
- communicating interrupts to task
- transforming interrupts to signals

... at the language level

## Interrupts ⇄ 'Signals'

Some characteristics of signals:

- Involve a full task-switch operation
- ☞ Hard to predict timing behaviour
- Limited information about the interrupt-source
- Traditionally used to 'kill' processes
- Concept stems from a time before thread models, therefore the signal-to-thread propagation is implementation dependent and sometimes tricky.

## Interrupts ⇄ 'Signals'

Some common UNIX OS entries:

POSIX 1003.1b	BSD-UNIX	
signal (...)	signal (...)	Specify the handler associated with a signal
sigaction (...)	sigvec (...)	Examine or set the signal handler for a signal
kill (...)	kill (...)	Send a signal (overwrite all other pending signals)
sigqueue (...)	N/A	Send a queued signal
sigsuspend (...)	pause (...)	Wait for a signal
sigwaitinfo (...)		Wait for a signal, but do not involve the handler
sigtimedwait (...)		
sigemptyset (...)	sigsetmask (...)	Manipulate and set the mask of blocked signals
sigprocmask (...)	sigblock (...)	Add to a set of blocked signals

## Interrupts ⇄ 'Signals'

- Signals are originally process-level synchronization methods ('kill') and have been expanded to be used for everything from hardware-interrupts and timers to asynchronous task messaging.

☞ Signals are passed through a global task-scheduler.

☞ in many OSs: unpredictable 'work-arounds' for missing direct hardware interrupt propagation.

☞ make sure that you understand the attached strings in your OS, before employing any signals.

## Interrupts

Interrupt control:

... at the individual device level

... at the system interrupt controller level

... at the operating system level

- beyond task-level (interrupt service routines)
- communicating interrupts to task
- transforming interrupts to signals

... at the language level

## Exception/Trap/Interrupt indication

Four cases of modern exception indication:

raised:	from:	run-time environment	task
synchronously		run-time exceptions	exceptions or traps
asynchronously		interrupts / signals	asynchronous transfer of control

## Exception/Trap/Interrupt indication

Ada95:

raised:	from:	run-time environment	task
synchronously		exceptions	
asynchronously		interrupt/signal handler	asynchronous transfer of control

## Ada95: Interrupt handlers

```
package Ada.Interrupts is
  type Interrupt_ID is implementation-defined;
  type Parameterless_Handler is access protected procedure;
  function Is_Reserved (Interrupt : Interrupt_ID) return Boolean;
  function Is_Attached (Interrupt : Interrupt_ID) return Boolean;
  function Current_Handler (Interrupt : Interrupt_ID)
    return Parameterless_Handler;
  procedure Attach_Handler (New_Handler : in Parameterless_Handler;
    Interrupt : in Interrupt_ID);
  procedure Exchange_Handler (Old_Handler : out Parameterless_Handler;
    New_Handler : in Parameterless_Handler;
    Interrupt : in Interrupt_ID);
  procedure Detach_Handler (Interrupt : in Interrupt_ID);
  function Reference (Interrupt : Interrupt_ID) return System.Address;
end Ada.Interrupts;
```

## Ada95: Interrupt handlers

```
package Ada.Interrupts is
  type Interrupt_ID is implementation-defined;
  type Parameterless_Handler is access protected procedure;
  function Is_Reserved (Interrupt : Interrupt_ID) return Boolean;
  function Is_Attached (Interrupt : Interrupt_ID) return Boolean;
  function Current_Handler (Interrupt : Interrupt_ID)
    return Parameterless_Handler;
  procedure Attach_Handler (New_Handler : in Parameterless_Handler;
    Interrupt : in Interrupt_ID);
  procedure Exchange_Handler (Old_Handler : out Parameterless_Handler;
    New_Handler : in Parameterless_Handler;
    Interrupt : in Interrupt_ID);
  procedure Detach_Handler (Interrupt : in Interrupt_ID);
  function Reference (Interrupt : Interrupt_ID) return System.Address;
end Ada.Interrupts;
```

## Ada95: Interrupt handlers

```
package Ada.Interrupts is
  type Interrupt_ID is implementation-defined;
  type Parameterless_Handler is access protected procedure;
  function Is_Reserved (Interrupt : Interrupt_ID) return Boolean;
  function Is_Attached (Interrupt : Interrupt_ID) return Boolean;
  function Current_Handler (Interrupt : Interrupt_ID)
    return Parameterless_Handler;
  procedure Attach_Handler (New_Handler : in Parameterless_Handler;
    Interrupt : in Interrupt_ID);
  procedure Exchange_Handler (Old_Handler : out Parameterless_Handler;
    New_Handler : in Parameterless_Handler;
    Interrupt : in Interrupt_ID);
  procedure Detach_Handler (Interrupt : in Interrupt_ID);
  function Reference (Interrupt : Interrupt_ID) return System.Address;
end Ada.Interrupts;
```

## Ada95: Interrupt handlers

```
package Ada.Interrupts is
  type Interrupt_ID is implementation-defined;
  type Parameterless_Handler is access protected procedure;
  function Is_Reserved (Interrupt : Interrupt_ID) return Boolean;
  function Is_Attached (Interrupt : Interrupt_ID) return Boolean;
  function Current_Handler (Interrupt : Interrupt_ID)
    return Parameterless_Handler;
  procedure Attach_Handler (New_Handler : in Parameterless_Handler;
    Interrupt : in Interrupt_ID);
  procedure Exchange_Handler (Old_Handler : out Parameterless_Handler;
    New_Handler : in Parameterless_Handler;
    Interrupt : in Interrupt_ID);
  procedure Detach_Handler (Interrupt : in Interrupt_ID);
  function Reference (Interrupt : Interrupt_ID) return System.Address;
end Ada.Interrupts;
```

## Ada95: Interrupt handlers

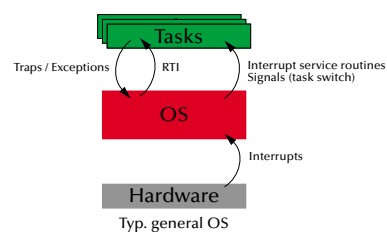
```
package Ada.Interrupts is
  type Interrupt_ID is implementation-defined;
  type Parameterless_Handler is access protected procedure;
  function Is_Reserved (Interrupt : Interrupt_ID) return Boolean;
  function Is_Attached (Interrupt : Interrupt_ID) return Boolean;
  function Current_Handler (Interrupt : Interrupt_ID)
    return Parameterless_Handler;
  procedure Attach_Handler (New_Handler : in Parameterless_Handler;
    Interrupt : in Interrupt_ID);
  procedure Exchange_Handler (Old_Handler : out Parameterless_Handler;
    New_Handler : in Parameterless_Handler;
    Interrupt : in Interrupt_ID);
  procedure Detach_Handler (Interrupt : in Interrupt_ID);
  function Reference (Interrupt : Interrupt_ID) return System.Address;
end Ada.Interrupts;
```

## Ada95: Interrupt handlers

```
package Ada.Interrupts is
  type Interrupt_ID is implementation-defined;
  type Parameterless_Handler is access protected procedure;
  function Is_Reserved (Interrupt : Interrupt_ID) return Boolean;
  function Is_Attached (Interrupt : Interrupt_ID) return Boolean;
  function Current_Handler (Interrupt : Interrupt_ID)
    return Parameterless_Handler;
  procedure Attach_Handler (New_Handler : in Parameterless_Handler;
    Interrupt : in Interrupt_ID);
  procedure Exchange_Handler (Old_Handler : out Parameterless_Handler;
    New_Handler : in Parameterless_Handler;
    Interrupt : in Interrupt_ID);
  procedure Detach_Handler (Interrupt : in Interrupt_ID);
  function Reference (Interrupt : Interrupt_ID) return System.Address;
end Ada.Interrupts;
```

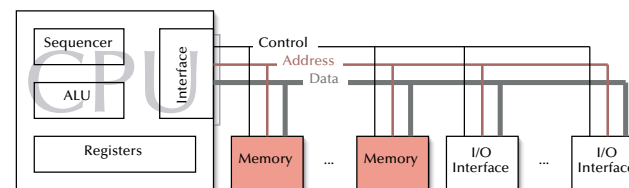
## What is an operating system?

## 3. A virtual machine, which is handling exceptions!



## Hardware Fundamentals

## A common computer architecture:



- Memory:
  - Hierarchy, Caching, Mapping

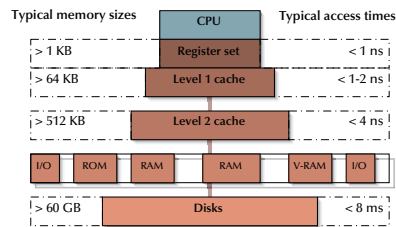
## Hardware Fundamentals

## Memory sizes and access times: (typical workstation)

Basic memory hierarchy		
Typical memory sizes	CPU	Typical access times
> 1 KB	Register set	< 1 ns
> 64 KB	Level 1 cache	< 1-2 ns
> 512 KB	Level 2 cache	< 4 ns
> 256 MB	Main memory	< 8 ns
> 60 GB	Disks	< 8 ms

## Main memory layout:

## Basic memory hierarchy



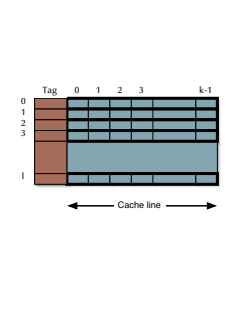
## Caching

- Introduce an intermediate memory (cache), which is:

- faster than the original memory
- organized in 'cache lines'
- addressed via tags and a fast matching hardware (e.g. associative memory)

Caché is actually French, meaning 'hidden', hence the cache memory is supposed to be 'invisible' to the user (the 'shadow memory').

Cache

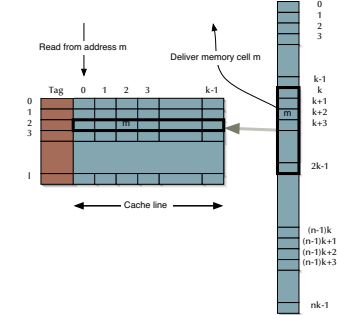


## Cache misses

Memory read requests to cells, which are not currently stored in the cache, result in:

- transfer of the full cache line into an empty of replaceable cache entry.
- transfer of the data directly from the main memory to the requester.

Cache miss

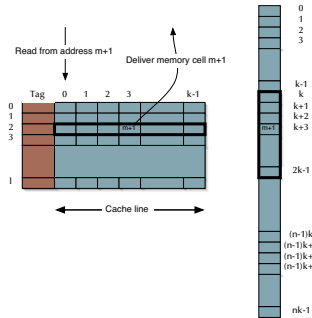


## Cache hits

Memory read requests to cells, which are currently stored in the cache, result in:

- transfer of the requested data from the cache memory to the requester.
- no access to the main memory

Cache hit

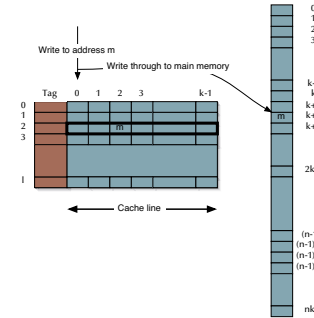


## Cache write through

Write requests to cells, which are currently stored in the cache, result in:

- update of the cache entry
- update of the main memory cell

Cache write through



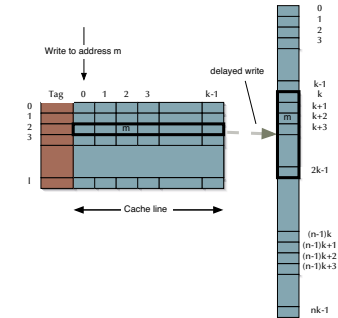
## Cache, delayed writes

Write requests to cells, which are currently stored in the cache, result in:

- update of the cache entry
- transfer of the full cache line (or the 'touched' entries) at a later point in time.

⇒ Critical in multi-processor / shared memory environments!

Cache write (delayed)



## Caching considerations

- Caches (two-level memories) are meant to maximize the **throughput** – not the predictability of a system.
- Cache performance is relying on:

- Spatial locality:** nearby memory cells are likely to be accessed soon
- Temporal locality:** recently addressed memory cells are likely to be accessed again soon

⇒ The length of the cache lines are given by the relation between spatial and temporal locality

- According to some practical evaluations, the locality radius seems to be *independent* of the size of the main memory
- ⇒ thus there is an absolute maximum cache-size, beyond which the performance is no longer improving (memory caches of up to about 128KB are considered adequate in most cases).

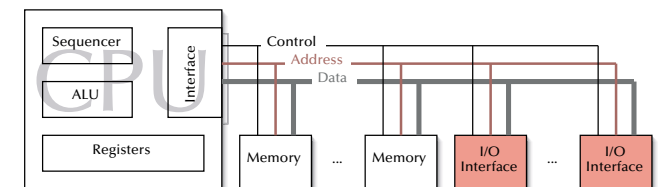
## More on memory locality

- Imperative programming* will generate linear sequences of instructions mostly (⇒ **spatial locality**).
- Functional and declarative programming* turns out to generate more 'jumpy' code, but due to extensive usage of recursions it will show strong **temporal locality**.
- Under all programming paradigms* CPU-time is often spent in relatively small loops/iterations (⇒ **spatial & temporal locality**)
- Languages, which are using *explicit data structures* (like arrays and records) will store this data in a compact format (⇒ **spatial locality**).

⇒ The locality assumptions will thus be justified in the vast majority of all cases

... still it's an heuristic.

## A common computer architecture:



- I/O interfaces:

- devices, controllers, communication with CPU, basic device programming

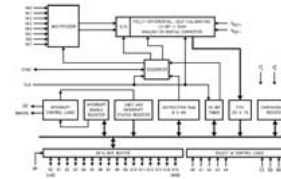
## I/O devices

the essential parts of a computer system, which (may) make the computations meaningful.

- Some typical classes of I/O devices:
  - clocks, timers
  - user-interface devices
  - document I/O devices (scanners, printers, ...)
  - audio & video equipment
  - network interfaces
  - mass storage devices
  - all kinds of sensors and actuators in control applications

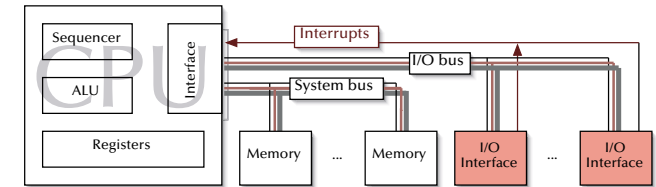
## I/O controllers

- Interfacing between a local bus-system (system bus, peripheral bus) and an concrete hardware device
- Accessible from the CPU via control, status and data registers



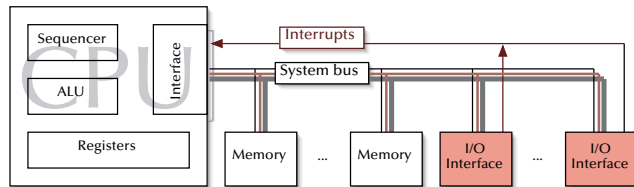
- Major tasks:
  - convert electrical signals
  - buffer data in case of different signal speeds
  - multiplexing different channels
  - communicate with the external device independently of the CPU as far as possible
    - often up to the level of a complete embedded µcontroller

## I/O interfaces via dedicated I/O-buses



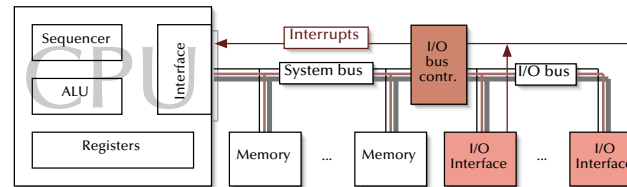
- I/O protection** is given by protected CPU instructions → need to be done in protected mode.
- Potentially less efficient, since *all* I/O operations need to be done in the OS-kernel no obvious DMA - everything needs to be transferred via the CPU, I/O bus is processor specific

## I/O interfaces via system-bus



- I/O protection** requires / is identical with **memory protection**, DMA possibilities, expandible
- System bus can be a bottle-neck, I/O interfaces are processor dependent

## I/O interfaces via system-bus and I/O bus controller

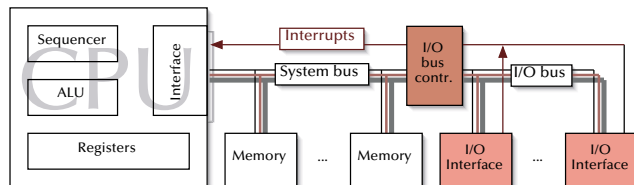


- I/O protection** requires / is identical with **memory protection**, DMA possibilities, expandible
- System bus load can be reduced, I/O bus is platform independent, e.g. PCI, SCSI, ...

## Basic I/O device programming

- Status driven:** the computer polls for information (used in dedicated µcontrollers and pre-scheduled hard real-time environments)
- Interrupt driven:** The data generating device may issue an interrupt when new data had been detected / converted or when internal buffers are full
  - Program controlled:** The interrupts are handled by the CPU directly (by changing tasks, calling a procedure, raising an exception, free tasks on a semaphore, sending a message to a task, ...)
  - Program initiated:** The interrupts are handled by a DMA-controller. No processing is performed. Depending on the DMA setup, cycle stealing can occur and needs to be considered for the worst case computing times.
  - Channel program controlled:** The interrupts are handled by a dedicated channel device. The data is transferred and processed. Optional memory-based communication with the CPU. → the channel controller is usually itself a dedicated engine / µcontroller.

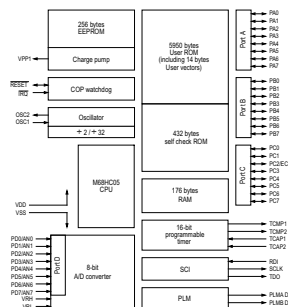
## Concurrency is an intrinsic feature of real architectures!



- Operating systems need to take care of all asynchronous and concurrent resources.
- Concurrency and synchronization are fundamentals of operating systems design!**

## MC68HC05

- Clock:** max. 2.1MHz internal (4.2MHz external)
- Registers:** PC, SP (16 bit); Accu, Index, CC (8 bit)
- RAM:** 176bytes
- ROM:** 5936bytes
- EEPROM:** 256bytes
- Power saving modes** (stop, wait, slow)
- Serial:** 46-76800 baud (at 2.4576MHz)
- Parallel I/O:** 3\*8bit; Parallel in: 1\*8bit
- Timers:** 1\*16bit
- A/D:** 8 channels, 8bit
- PWM:** 2 generators



```

MAIN  BRCLR 6,TSR,MAIN      ;Loop here till Output Compare flag set
      LDA   OCMP+1          ;Low byte of Output Compare register
      ADD   #$D4             ;Add Δt1 = (50ms/4µs)mod28 = $D4
      STA   TEMP4           ;Save till high half calculated
      LDA   OCMP            ;High byte of Output Compare register
      ADC   #$30             ;Add Δt2 = (50ms/4µs)div28 = $30 (+carry)
      STA   OCMP            ;Update high byte of Output Compare register
      LDA   TEMP4           ;Get low half of updated value
      STA   OCMP+1          ;Update low half and reset Output Compare flag
      LDA   TIC              ;Get current TIC value
      INCA                  ;TIC := TIC + 1
      STA   TIC             ;Update TIC
      CMP   #20             ;20th TIC?, 1 second passed?
      BLO  NOSEC            ;If not, skip next clear
      CLR   TIC             ;Clear TIC on 20th
      EQU   *
NOSEC  JSR   TIME            ;Update time-of-day & day-of-week
      JSR   KVPAD           ;Check/service keypad
      JSR   A2D             ;Check Temp Sensors
      JSR   HVAC            ;Update Heat/Air Cond Outputs
      JSR   LCD             ;Update LCD display
      BRA  MAIN             ;End of main loop
  
```



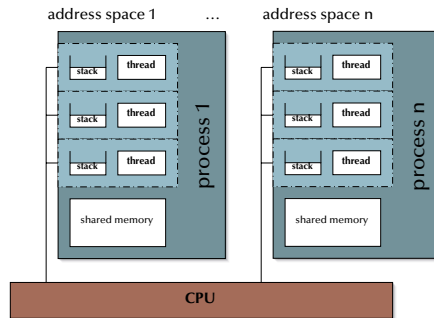




### Threads

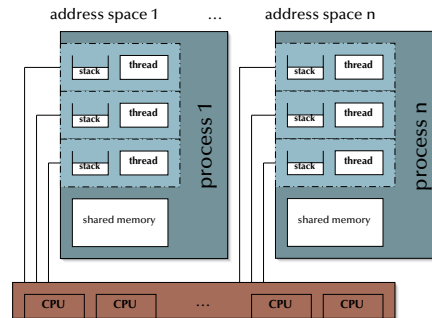
Threads (individual control-flows) can be handled:

- **inside the kernel:**
  - kernel scheduling
  - I/O block-releases according to external signal
- **outside the kernel:**
  - user-level scheduling
  - no signals to threads



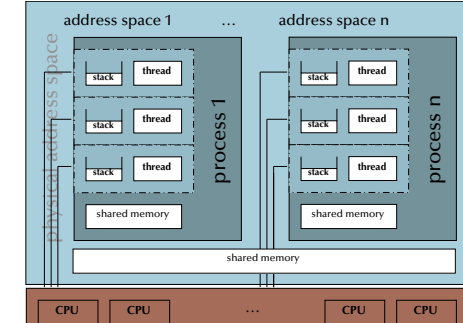
### Multi-processor-systems

- The kernel may execute multiple processes at a time.
- Address space and resource restrictions of individual CPUs and processes/threads need to be considered.
- Caching, synchronization, and memory protection need to be adapted.



### Symmetric Multi-processing (SMP)

- all CPUs share the same physical address space (and access to resources)
- processes/threads can be executed on any available CPU



### Processes ↔ Threads

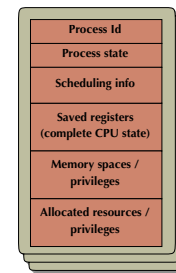
Also processes can share memory and the exact interpretation of threads is different in different operating systems:

- Threads can be regarded as a group of processes, which share some resources (⇒ process-hierarchy)
- Due to the overlap in resources, the attributes attached to threads are less than for 'first-class-citizen-processes'
- Thread switching and inter-thread communications can be more efficient than on full-process-level
- Scheduling of threads depends on the actual thread implementations:
  - e.g. user-level control-flows, which the kernel has no knowledge about at all
  - e.g. kernel-level control-flows, which are handled as processes with some restrictions

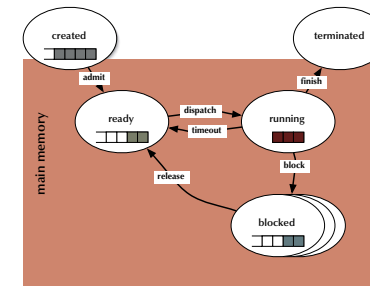
### Process Control Blocks

- **Process Id**
- **Process state:** {created, ready, executing, blocked, suspended, ...}
- **Scheduling info:** priorities, deadlines, consumed CPU-time, ...
- **CPU state:** saved/restored information while context switches (incl. the program counter, stack pointer, ...)
- **Memory spaces / privileges:** memory base, limits, shared areas, ...
- **Allocated resources / privileges:** open and requested devices and files

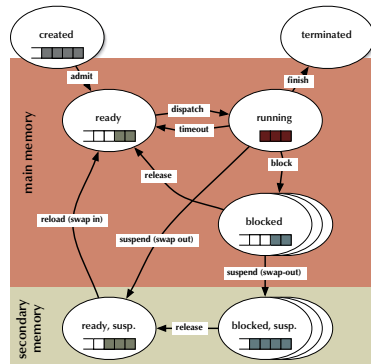
Process Control Blocks (PCBs)



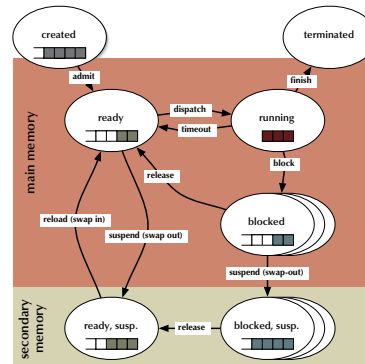
... PCBs are usually enqueued at a certain state or condition



- **created:** the task is ready to run, but not yet considered by any dispatcher – waiting for admission
- **ready:** ready to run – waiting for a free CPU
- **running:** holds a CPU and executes
- **blocked:** not ready to run – waiting for a resource to become available

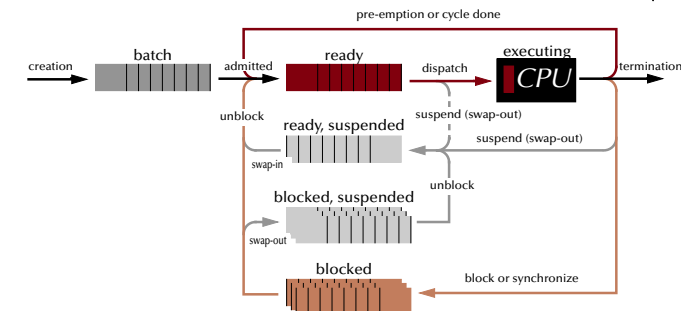


- **created:** the task is ready to run, but not yet considered by any dispatcher – waiting for admission
- **ready:** ready to run – waiting for a free CPU
- **running:** holds a CPU and executes
- **blocked:** not ready to run – waiting for a resource
- **suspended states:** swapped out of main memory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



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

⇒ dispatching and suspending can be independent modules here



## Synchronization methods

## • Shared memory based synchronization

- Semaphores
  - Conditional critical regions
  - Monitors
  - Mutexes & conditional variables
  - Synchronized methods
  - Protected objects
- ⇒ 'C', POSIX — Dijkstra
  - ⇒ Edison (experimental)
  - ⇒ Modula-1, Mesa — Dijkstra, Hoare, ...
  - ⇒ POSIX
  - ⇒ Real-time Java
  - ⇒ Ada95

## • Message based synchronization

- Asynchronous messages
  - Synchronous messages
  - Remote invocation, remote procedure call
  - Synchronization in distributed systems
- ⇒ e.g. POSIX, ...
  - ⇒ e.g. Ada95, CHILL, Occam2
  - ⇒ e.g. Ada95, ...
  - ⇒ e.g. CORBA, ...

## Synchronization in operating systems

⇒ There are many concurrent entities in operating systems:

- Interrupt handlers
- Processes
- Dispatchers
- Timers
- ...

... and ... operating systems need to be expandible or very robust ...

Thus *all* data is declared ...

- ⇒ ... **either** local (and protected by language-, or hardware-mechanisms)
- ⇒ ... **or** it is 'out in the open' and all access need to be synchronized!

## The need for synchronization

Synchronization: the run-time overhead?

⇒ Is the potential overhead justified for simple data-structures:

```

int i;
.....
i++; {in one thread} | i=0; {in another thread}

```

- Are those operations atomic?
- Do we really need to introduce full featured synchronization methods here?

## The need for synchronization

```

int i;
.....
i++; {in one thread} | i=0; {in another thread}

```

- Depending on the hardware and the compiler, it might be atomic, it might be not:
- ⇒ Handling a 64-bit integer on a 8- or 16-bit controller *will not be atomic* ... but perhaps it is an 8-bit integer.
- ⇒ Any manipulations on the main memory *will not be atomic* ... but perhaps it is a register.
- ⇒ Broken down to a load-operate-store cycle, the operations *will not be atomic* ... but perhaps the processor supplies atomic operations for the actual case.
- ⇒ Assuming that all 'perhapses' are applying: how to expand this code?

## The need for synchronization

```

int i;
.....
i++; {in one thread} | i=0; {in another thread}

```

- ⇒ Unfortunately: the chances that such programming errors turn out are usually small and some implicit by chance synchronization in the rest of the system might prevent them at all.
- Many effects stemming from asynchronous memory accesses are interpreted as (hardware) 'glitches', since they are rare and effect usually only some parts of the data.
- On assembler level: synchronization by employing knowledge about the atomicity of CPU-operations and interrupt structures is nevertheless possible and done frequently.

In anything higher than assembler level on small, predictable µcontrollers:

⇒ Measures for synchronization are required!

## Some synchronization terms:

- **Condition synchronization:**  
synchronize a task with an event given by another task.
- **Critical sections:**  
code fragments which contain access to shared resources and need to be executed without interference with other critical sections, sharing the same resources.
- **Mutual exclusion:**  
protection against asynchronous access to critical sections.
- **Atomic operations:**  
the set of operations, which atomicity is guaranteed by the underlying system (e.g. hardware).  
⇒ there must be a set of atomic operations to start with!

## Synchronization by flags

Word-access atomicity:

Assuming that any access to a word in the system is an atomic operation:

e.g. assigning two values (not wider than the size of word) to a memory cell simultaneously:

```

Task 1: x := 0;      |      Task 2: x := 5;

```

will result in **either**  $x = 0$  **xor**  $x = 5$  — and no other value is ever observable.

## Synchronization by flags

Assuming further that there is a shared memory area between two processes:

- A set of processes agree on a (word-size) atomic variable operating as a flag to indicate synchronization conditions.

## Condition synchronization by flags

```

var Flag : boolean := false;

```

```

process P1;                      process P2;
  statement X;                    statement A;

  repeat until Flag;              Flag := true;

  statement Y;                    statement B;
end P1;                           end P2;

```

Sequence of operations:  $[A \mid X] \rightarrow [B \mid Y]$

## Synchronization by flags

Assuming further that there is a shared memory between two processes:

- A set of processes agree on a (word-size) atomic variable operating as a flag to indicate synchronization conditions:

Memory flag method is ok for simple condition synchronization, but ...

- ☞ ... is not sufficient for general mutual exclusion in critical sections!
- ☞ ... busy-waiting is required to poll the synchronization condition!

☞ More powerful synchronization operations are required for critical sections

## Synchronization by semaphores

(Dijkstra 1968)

Assuming further that there is a shared memory between two processes:

- a set of processes agree on a variable **S** operating as a flag to indicate synchronization conditions ... and ...
  - an atomic operation **P** on **S** — **P** stands for 'passen' (Dutch for 'pass'):  
 • **P**: `if S > 0 then S := S - 1` also: 'Wait', 'Suspend\_Until\_True'
  - an atomic operation **V** on **S** — **V** stands for 'vrygeven' (Dutch for 'to release'):  
 • **V**: `S := S + 1` also: 'Signal', 'Set\_True'
- ☞ the variable **S** is then called a **semaphore**.

OS-level: **P** is usually also suspending the current task until **S** > 0.

CPU-level: **P** indicates whether it was successful, but the operation is not blocking.

## Condition synchronization by semaphores

```
var sync : semaphore := 0;

process P1;
statement X;
wait (sync);
statement Y;
end P1;

process P2;
statement A;
signal (sync);
statement B;
end P2;
```

Sequence of operations:  $[A \mid X] \rightarrow [B \mid Y]$

## Mutual exclusion by semaphores

```
var mutex : semaphore := 1;
```

```
process P1;
statement X;
wait (mutex);
statement Y;
signal (mutex);
statement Z;
end P1;

process P2;
statement A;
wait (mutex);
statement B;
signal (mutex);
statement C;
end P2;
```

Sequence of operations:  $[A \mid X] \rightarrow [B \rightarrow Y \text{ xor } Y \rightarrow B] \rightarrow [C \mid Z]$

## Semaphores

Types of semaphores:

- **General semaphores (counting semaphores)**: non-negative number; (range limited by the system) **P** and **V** increment and decrement the semaphore by one.
- **Binary semaphores**: restricted to [0, 1]; Multiple **V** (Signal) calls have the same effect than 1 call.
  - binary semaphores are sufficient to create all other semaphore forms.
  - atomic 'test-and-set' operations at hardware level are usually binary semaphores.
- **Quantity semaphores**: The increment (and decrement) value for the semaphore is specified as a parameter with **P** and **V**.

## Semaphores in Ada95

```
package Ada.Synchronous_Task_Control is
type Suspension_Object is limited private;
procedure Set_True (S : in out Suspension_Object);
procedure Set_False (S : in out Suspension_Object);
function Current_State (S : Suspension_Object) return Boolean;
procedure Suspend_Until_True (S : in out Suspension_Object);
private
  -- not specified by the language
end Ada.Synchronous_Task_Control;
```

## Semaphores in Ada95

```
package Ada.Synchronous_Task_Control is
type Suspension_Object is limited private;
procedure Set_True (S : in out Suspension_Object);
procedure Set_False (S : in out Suspension_Object);
function Current_State (S : Suspension_Object) return Boolean;
procedure Suspend_Until_True (S : in out Suspension_Object);
private
  -- not specified by the language
end Ada.Synchronous_Task_Control;
```

- only one task can be blocked at `Suspend_Until_True`! ('strict version of a binary semaphore') (`Program_Error` will be raised with the second task trying to suspend itself)
- ☞ no queues! ☞ minimal run-time overhead

## Semaphores in Ada95

```
package Ada.Synchronous_Task_Control is
type Suspension_Object is limited private;
procedure Set_True (S : in out Suspension_Object);
procedure Set_False (S : in out Suspension_Object);
function Current_State (S : Suspension_Object) return Boolean;
procedure Suspend_Until_True (S : in out Suspension_Object);
private
  -- not specified by the language
end Ada.Synchronous_Task_Control;
```

- only one task can be blocked at `Suspend_Until_True`! ('strict version of a binary semaphore') (`Program_Error` will be raised with the second task trying to suspend itself)
- ☞ no queues ☞ minimal run-time overhead

## Semaphores in POSIX

```
int sem_init (sem_t *sem_location, int pshared, unsigned int value);
int sem_destroy (sem_t *sem_location);

int sem_wait (sem_t *sem_location);
int sem_trywait (sem_t *sem_location);
int sem_timedwait (sem_t *sem_location, const struct timespec *abstime);

int sem_post (sem_t *sem_location);
int sem_getvalue (sem_t *sem_location, int *value);
```

## Semaphores in POSIX

```

int sem_init (sem_t *sem_location, int pshared, unsigned int value);
int sem_destroy (sem_t *sem_location);
int sem_wait (sem_t *sem_location);
int sem_trywait (sem_t *sem_location);
int sem_timedwait (sem_t *sem_location, const struct timespec *abstime);
int sem_post (sem_t *sem_location);
int sem_getvalue (sem_t *sem_location, int *value);

```

generate semaphore for usage between processes  
(otherwise for threads of the same process only)

## Semaphores in POSIX

```

int sem_init (sem_t *sem_location, int pshared, unsigned int value);
int sem_destroy (sem_t *sem_location);
int sem_wait (sem_t *sem_location);
int sem_trywait (sem_t *sem_location);
int sem_timedwait (sem_t *sem_location, const struct timespec *abstime);
int sem_post (sem_t *sem_location);
int sem_getvalue (sem_t *sem_location, int *value);

```

delivers the number of waiting processes as a negative integer,  
if there are processes waiting on this semaphore

## Semaphores in POSIX

```

void allocate (priority_t P)
{
    sem_wait (&mutex);
    if (busy) {
        sem_post (&mutex);
        sem_wait (&cond[P]);
    }
    busy = 1;
    sem_post (&mutex);
}

void deallocate (priority_t P)
{
    sem_wait (&mutex);
    busy = 0;
    sem_getvalue (&cond[high], &waiting);
    if (waiting < 0) {
        sem_post (&cond[high]);
    }
    else {
        sem_getvalue (&cond[low], &waiting);
        if (waiting < 0) {
            sem_post (&cond[low]);
        }
        else {
            sem_post (&mutex);
        }
    }
}

sem_t mutex, cond[2];
typedef enum {low, high} priority_t;
int waiting;
int busy;

```

## Deadlock by semaphores

```

with Ada.Synchronous_Task_Control; use Ada.Synchronous_Task_Control;
X, Y : Suspension_Object;

```

```

task B;
task body B is
begin
    ...
    Suspend_Until_True (Y);
    Suspend_Until_True (X);
    ...
end B;

```

```

task A;
task body A is
begin
    ...
    Suspend_Until_True (X);
    Suspend_Until_True (Y);
    ...
end A;

```

could raise a `Program_Error` in Ada95.

produces a potential **deadlock** when implemented with general semaphores.

Deadlocks can be generated by all kinds of synchronization methods.

## Criticism of semaphores

- Semaphores are not bound to any resource or method or region
  - Adding or deleting a single semaphore operation some place might stall the whole system
- Semaphores are scattered all over the code
  - hard to read, error-prone

➤ Semaphores are considered not adequate for complex systems.

(all concurrent and real-time languages offer more abstract and safer synchronization methods).

## Conditional critical regions

Basic idea:

- Critical regions are a set of code sections in different processes, which are guaranteed to be **executed in mutual exclusion**:
  - Shared data structures are grouped in named regions and are tagged as being private resources.
  - Processes are prohibited from entering a critical region, when another process is active in any associated critical region.
- **Condition synchronisation** is provided by *guards*:
  - When a process wishes to enter a critical region it evaluates the guard (under mutual exclusion). If the guard evaluates false, the process is suspended / delayed.
- As with semaphores, no access order can be assumed.

## Conditional critical regions

```

buffer : buffer_t;
resource critical_buffer_region : buffer;

```

```

process producer;
loop
    region critical_buffer_region
    when buffer.size < N do
        -- place in buffer etc.
    end region
end loop;
end producer;

process consumer;
loop
    region critical_buffer_region
    when buffer.size > 0 do
        -- take from buffer etc.
    end region
end loop;
end consumer;

```

## Criticism of conditional critical regions

- All guards need to be re-evaluated, when any conditional critical region is left:
  - all involved processes are activated to test their guards
  - there is no order in the re-evaluation phase ➤ potential livelocks
- As with semaphores the conditional critical regions are scattered all over the code.
  - on a larger scale: same problems as with semaphores.

The language Edison uses conditional critical regions for synchronization in a multiprocessor environment (each process is associated with exactly one processor).

## Monitors

(Modula-1, Mesa — Dijkstra, Hoare)

Basic idea:

- Collect all operations and data-structures shared in critical regions in one place, the monitor.
- Formulate all operations as procedures or functions.
- Prohibit access to data-structures, other than by the monitor-procedures.
- Assure mutual exclusion of the monitor-procedures.

## Monitors

```
monitor buffer;
  export append, take;
  var (* declare protected vars *)
  procedure append (I : integer);
  ...
  procedure take (var I : integer);
  ...
begin
  (* initialisation *)
end;
```

How to realize conditional synchronization?

## Monitors with condition synchronization

(Hoare)

Hoare-monitors:

- Condition variables are implemented by semaphores (`Wait` and `Signal`).
  - Queues for tasks suspended on condition variables are realized.
  - A suspended task releases its lock on the monitor, enabling another task to enter.
- ⇒ More efficient evaluation of the guards:  
the task leaving the monitor can evaluate all guards and the right tasks can be activated.
- ⇒ Blocked tasks may be ordered and livelocks prevented.

## Monitors with condition synchronization

```
monitor buffer;
  export append, take;
  var BUF : array [ ... ] of integer;
  top, base : 0..size-1;
  NumberInBuffer : integer;
  spaceavailable, itemavailable : condition;
  procedure append (I : integer);
  begin
    if NumberInBuffer = size then
      wait (spaceavailable);
    end if;
    BUF[top] := I; NumberInBuffer := NumberInBuffer+1;
    top := (top+1) mod size;
    signal (itemavailable);
  end append; ...
```

## Monitors with condition synchronization

```
...
  procedure take (var I : integer);
  begin
    if NumberInBuffer = 0 then
      wait (itemavailable);
    end if;
    I := BUF[base];
    base := (base+1) mod size;
    NumberInBuffer := NumberInBuffer-1;
    signal (spaceavailable);
  end take;
begin (* initialisation *)
  NumberInBuffer := 0;
  top := 0; base := 0;
end;
```

The signalling and the waiting process are both active in the monitor!

## Monitors with condition synchronization

Suggestions to overcome the multiple-tasks-in-monitor-problem:

- A `signal` is allowed **only as the last action** of a process before it leaves the monitor.
- A `signal` operation has the side-effect of **executing a `return` statement**.
- Hoare, Modula-1, POSIX: a `signal` operation which unblocks another process has the side-effect of **blocking the current process**; this process will only execute again once the monitor is unlocked again.
- A `signal` operation which unblocks a process does not block the caller, but the unblocked process must **gain access to the monitor again**.

## Monitors in Modula-1

- `wait (s, r)`:  
delays the caller until condition variable `s` is true (`r` is the rank (or 'priority') of the caller).
- `send (s)`:  
If a process is waiting for the condition variable `s`, then the process at the top of the queue of the highest filled rank is activated (and the caller suspended).
- `awaited (s)`:  
check for waiting processes on `s`.

## Monitors in Modula-1

```
INTERFACE MODULE resource_control;
  DEFINE allocate, deallocate;
  VAR busy : BOOLEAN; free : SIGNAL;
  PROCEDURE allocate;
  BEGIN
    IF busy THEN WAIT (free) END;
    busy := TRUE;
  END;
  PROCEDURE deallocate;
  BEGIN
    busy := FALSE;
    SEND (free); -- or: IF AWAITED (free) THEN SEND (free);
  END;
BEGIN
  busy := false;
END.
```

## Monitors in 'C' / POSIX

(types and creation)

Synchronization between POSIX-threads:

```
typedef ... pthread_mutex_t;
typedef ... pthread_mutexattr_t;
typedef ... pthread_cond_t;
typedef ... pthread_condattr_t;

int pthread_mutex_init (const pthread_mutexattr_t *attr,
                        const pthread_mutex_t *mutex);
int pthread_mutex_destroy (const pthread_mutex_t *mutex);
int pthread_cond_init (const pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
int pthread_cond_destroy (const pthread_cond_t *cond);
...
```

## Monitors in 'C' / POSIX

(types and creation)

Synchronization between POSIX-threads:

```
typedef ... pthread_mutex_t;
typedef ... pthread_mutexattr_t;
typedef ... pthread_cond_t;
typedef ... pthread_condattr_t;

int pthread_mutex_init (const pthread_mutexattr_t *attr,
                        const pthread_mutex_t *mutex);
int pthread_mutex_destroy (const pthread_mutex_t *mutex);
int pthread_cond_init (const pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
int pthread_cond_destroy (const pthread_cond_t *cond);
...
```

Attributes include:

- semantics for trying to lock a mutex which is locked already by the same thread
- sharing of mutexes and condition variables between processes
- priority ceiling
- clock used for timeouts
- ... ..



## Monitors in 'C' / POSIX

(types and creation)

Synchronization between POSIX-threads:

```

typedef ... pthread_mutex_t;
typedef ... pthread_mutexattr_t;
typedef ... pthread_cond_t;
typedef ... pthread_condattr_t;
int pthread_mutex_init (pthread_mutex_t *mutex,
                        pthread_mutexattr_t *attr);
int pthread_mutex_destroy (pthread_mutex_t *mutex);
int pthread_cond_init (pthread_cond_t *cond,
                      pthread_condattr_t *attr);
int pthread_cond_destroy (pthread_cond_t *cond);
...

```

Undefined, if locked

Undefined, if threads are waiting

## Monitors in 'C' / POSIX

(operators)

```

...
int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_trylock (pthread_mutex_t *mutex);
int pthread_mutex_timedlock (pthread_mutex_t *mutex,
                             struct timespec *abstime);
int pthread_mutex_unlock (pthread_mutex_t *mutex);
int pthread_cond_wait (pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_timedwait (pthread_cond_t *cond,
                            pthread_mutex_t *mutex,
                            struct timespec *abstime);
int pthread_cond_signal (pthread_cond_t *cond);
int pthread_cond_broadcast (pthread_cond_t *cond);
...

```

## Monitors in 'C' / POSIX

(operators)

```

...
int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_trylock (pthread_mutex_t *mutex);
int pthread_mutex_timedlock (pthread_mutex_t *mutex,
                              struct timespec *abstime);
int pthread_mutex_unlock (pthread_mutex_t *mutex);
int pthread_cond_wait (pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_timedwait (pthread_cond_t *cond,
                             pthread_mutex_t *mutex,
                             struct timespec *abstime);
int pthread_cond_signal (pthread_cond_t *cond);
int pthread_cond_broadcast (pthread_cond_t *cond);
...

```

unblocking 'at least one' thread

unblocking all threads

## Monitors in 'C' / POSIX

(operators)

```

...
int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_trylock (pthread_mutex_t *mutex);
int pthread_mutex_timedlock (pthread_mutex_t *mutex,
                              struct timespec *abstime);
int pthread_mutex_unlock (pthread_mutex_t *mutex);
int pthread_cond_wait (pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_timedwait (pthread_cond_t *cond,
                             pthread_mutex_t *mutex,
                             struct timespec *abstime);
int pthread_cond_signal (pthread_cond_t *cond);
int pthread_cond_broadcast (pthread_cond_t *cond);
...

```

undefined, if called out of order!

## Monitors in 'C' / POSIX

(operators)

```

...
int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_trylock (pthread_mutex_t *mutex);
int pthread_mutex_timedlock (pthread_mutex_t *mutex,
                              struct timespec *abstime);
int pthread_mutex_unlock (pthread_mutex_t *mutex);
int pthread_cond_wait (pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_timedwait (pthread_cond_t *cond,
                             pthread_mutex_t *mutex,
                             struct timespec *abstime);
int pthread_cond_signal (pthread_cond_t *cond);
int pthread_cond_broadcast (pthread_cond_t *cond);
...

```

can be called any time, anywhere (multiple lock reaction can be specified)

## Monitors in 'C' / POSIX

(example, definitions)

```

#define BUFF_SIZE 10
typedef struct {
    pthread_mutex_t mutex;
    pthread_cond_t buffer_not_full;
    pthread_cond_t buffer_not_empty;
    int count, first, last;
    buf[BUFF_SIZE];
} buffer;

```

## Monitors in 'C' / POSIX

(example, operations)

```

int append (int item, buffer *B) {
    PTHREAD_MUTEX_LOCK (&B->mutex);
    while (B->count == BUFF_SIZE) {
        PTHREAD_COND_WAIT (&B->buffer_not_full,
                           &B->mutex);
    }
    PTHREAD_MUTEX_UNLOCK (&B->mutex);
    PTHREAD_COND_SIGNAL (&B->buffer_not_empty);
    return 0;
}

int take (int *item, buffer *B) {
    PTHREAD_MUTEX_LOCK (&B->mutex);
    while (B->count == 0) {
        PTHREAD_COND_WAIT (&B->buffer_not_empty,
                           &B->mutex);
    }
    PTHREAD_MUTEX_UNLOCK (&B->mutex);
    PTHREAD_COND_SIGNAL (&B->buffer_not_full);
    return 0;
}

```

## Monitors in Java

Java provides two mechanisms to construct monitors:

- Synchronized methods and code blocks  
all methods and code blocks which are using the synchronized tag are mutually exclusive with respect to the addressed class.
- Notification methods: wait, notify, and notifyAll  
can be used only in synchronized regions and are waking any or all threads, which are waiting in the same synchronized object.

## Monitors in Java

Considerations:

## 1. Synchronized methods and code blocks:

- In order to implement a monitor *all* methods in an object need to be synchronized.  
⇒ any other standard method can break the monitor and enter at any time.
- Methods outside the monitor-object can synchronize at this object.  
⇒ it is impossible to analyse a monitor locally, since lock accesses can exist all over the system.
- Static data is shared between all objects of a class.  
⇒ access to static data need to be synchronized over the whole class.

Either in static synchronized blocks: `synchronized (this.getClass()) {...}`  
or in static methods: `public synchronized static <method> {...}`

## Monitors in Java

Considerations:

## 2. Notification methods: wait, notify, and notifyAll

- `wait` suspends the thread and releases the local lock only
  - ⇒ nested `wait`-calls will keep all enclosing locks.
- `notify` and `notifyAll` does not release the lock.
  - ⇒ methods, which are activated via notification need to wait for lock-access.
- `wait`-suspended threads are hold in a queue (Real-time Java only!), thus `notifyAll` is waking the threads in order ⇒ livelocks are prevented at this level .
- There are no explicit conditional variables.
  - ⇒ every notified thread needs to wait for the lock to be released and to re-evaluate its entry condition

## Monitors in Java

(multiple-readers-one-writer-example)

each of the **readers** uses these monitor.calls:

```
startRead ();
// read the shared data only
stopRead ();
```

each of the **writers** uses these monitor.calls:

```
startWrite ();
// manipulate the shared data
stopWrite ();
```

⇒ construct a monitor, which allows multiple readers  
or  
one writer  
at a time inside the critical regions

## Monitors in Java

(multiple-readers-one-writer-example: wait-notifyAll method)

```
public class ReadersWriters
{
    private int    readers      = 0;
    private int    waitingWriters = 0;
    private boolean writing      = false;
    ...
}
```

## Monitors in Java

(multiple-readers-one-writer-example: wait-notifyAll method)

```
... public synchronized void StartWrite () throws InterruptedException
{
    while (readers > 0 || writing)
    {
        waitingWriters++;
        wait();
        waitingWriters--;
    }
    writing = true;
    public synchronized void StopWrite()
    {
        writing = false;
        notifyAll ();
    } ...
}
```

## Monitors in Java

(multiple-readers-one-writer-example: wait-notifyAll method)

```
... public synchronized void StartRead () throws InterruptedException
{
    while (writing || waitingWriters > 0)
    {
        wait();
    }
    readers++;
    public synchronized void StopRead()
    {
        readers--;
        if (readers == 0) notifyAll();
    }
}
```

whenever a synchronized region is left:

- all thread are notified
- all threads are re-evaluating their guards

## Monitors in Java

## Standard monitor solution:

- declare the monitored data-structures private to the monitor object (non-static).
- introduce a class `ConditionVariable`:
 

```
public class ConditionVariable {
    public boolean wantToSleep = false;
}
```
- introduce synchronization-scopes in monitor-methods:
  - ⇒ synchronize on the adequate conditional variables *first* and
  - ⇒ synchronize on the monitor-object *second*.
- make sure that *all* methods in the monitor are implementing the correct synchronizations.
- make sure that *no other method* in the whole system is synchronizing on this monitor-object.

## Monitors in Java

(multiple-readers-one-writer-example: usage of external conditional variables)

```
public class ReadersWriters
{
    private int    readers      = 0;
    private int    waitingReaders = 0;
    private int    waitingWriters = 0;
    private boolean writing      = false;
    ConditionVariable OkToRead = new ConditionVariable ();
    ConditionVariable OkToWrite = new ConditionVariable ();
    ...
}
```

## Monitors in Java

```
... public void StartWrite () throws InterruptedException
{
    synchronized (OkToWrite)
    {
        synchronized (this)
        {
            if (writing | readers > 0) {
                waitingWriters++;
                OkToWrite.wantToSleep = true;
            } else {
                writing = true;
                OkToWrite.wantToSleep = false;
            }
            if (OkToWrite.wantToSleep) OkToWrite.wait ();
        } } ...
}
```

## Monitors in Java

```
... public void StopWrite ()
{
    synchronized (OkToRead)
    {
        synchronized (OkToWrite)
        {
            synchronized (this)
            {
                if (waitingWriters > 0) {
                    waitingWriters--;
                    OkToWrite.notify (); // wakeup one writer
                } else {
                    writing = false;
                    OkToRead.notifyAll (); // wakeup all readers
                    readers = waitingReaders;
                    waitingReaders = 0;
                }
            }
        } } ...
}
```

## Monitors in Java

```
... public void StartRead () throws InterruptedException
{
    synchronized (OkToRead)
    {
        synchronized (this)
        {
            if (writing | waitingWriters > 0) {
                waitingReaders++;
                OkToRead.wantsToSleep = true;
            } else {
                readers++;
                OkToRead.wantsToSleep = false;
            }
        }
        if (OkToRead.wantsToSleep) OkToRead.wait ();
    }
} ...
```

## Monitors in Java

```
... public void StopRead ()
{
    synchronized (OkToWrite)
    {
        synchronized (this)
        {
            readers--;
            if (readers == 0 & waitingWriters > 0) {
                waitingWriters--;
                OkToWrite.notify ();
            }
        }
    }
}
```

## Object-orientation and synchronization

Since mutual exclusion, notification, and condition synchronization schemes need to be designed and analysed considering the implementation of all involved methods and guards:

⇒ new methods cannot be added without re-evaluating the whole class!

In opposition to the general re-usage idea of object-oriented programming, the re-usage of synchronized classes (e.g. monitors) need to be considered carefully.

⇒ The parent class might need to be adapted in order to suit the global synchronization scheme.

⇒ Inheritance anomaly (Matsuoka & Yonezawa '93)

Methods to design and analyse expandible synchronized systems exist, but are fairly complex and are not provided in any current object-oriented language.

## Monitors in POSIX &amp; Java

⇒ flexible and universal,  
but relies on conventions rather than compilers

POSIX offers conditional variables

Java is more supportive than POSIX  
in terms of data-encapsulation

Extreme care must be taken when employing  
object-oriented programming and monitors

## Nested monitor calls

Assuming a thread in a monitor is calling an operation in another monitor and is suspended at a conditional variable there:

- ⇒ the called monitor is aware of the suspension and allows other threads to enter.
- ⇒ the calling monitor is possibly *not* aware of the suspension and **keeps its lock!**
- ⇒ the unjustified locked calling monitor reduces the system performance and leads to potential deadlocks.

Suggestions to solve this situation:

- Maintain the lock anyway: e.g. POSIX, Real-time Java
- Prohibit nested procedure calls: e.g. Modula-1
- Provide constructs which specify the release of a monitor lock for remote calls, e.g. Ada95

## Criticism of monitors

- Mutual exclusion is solved elegantly and safely.
- Conditional synchronization is on the level of semaphores still
  - ⇒ all criticism on semaphores apply
- ⇒ mixture of low-level and high-level synchronization constructs.

## Synchronization by protected objects

Combine

- the **encapsulation** feature of monitors

with

- the **coordinated entries** of conditional critical regions

to

⇒ Protected objects

- all controlled data and operations are encapsulated
- all operations are mutual exclusive
- entry guards are *attached* to operations
- the protected interface allows for operations on data
- no protected data is accessible (other than by defined operations)
- tasks are queued (according to their priorities)

## Synchronization by protected objects in Ada95

(simultaneous read-access)

Some read-only operations *do not need to be mutual exclusive*:

```
protected type Shared_Data (Initial : Data_Item) is
    function Read return Data_Item;
    procedure Write (New_Value : in Data_Item);
private
    The_Data : Data_Item := Initial;
end Shared_Data_Item;
```

- protected *functions* can have 'in' parameters only and are not allowed to alter the private data (enforced by the compiler).
- ⇒ protected functions allow **simultaneous access** (but mutual exclusive with other operations).
- there is no defined priority between functions and other protected operations in Ada95.

## Synchronization by protected objects in Ada95

Condition synchronization is realized in the form of protected procedures combined with boolean conditional variables (**barriers**): ⇒ **entries** in Ada95:

```
Buffer_Size : constant Integer := 10;
type Index is mod Buffer_Size;
subtype Count is Natural range 0 .. Buffer_Size;
type Buffer_T is array (Index) of Data_Item;

protected type Bounded_Buffer is
    entry Get (Item : out Data_Item);
    entry Put (Item : in Data_Item);
private
    First : Index := Index'First;
    Last : Index := Index'Last;
    Num : Count := 0;
    Buffer : Buffer_T;
end Bounded_Buffer;
```

## Synchronization by protected objects in Ada95

(barriers)

```
protected body Bounded_Buffer is
  entry Get (Item : out Data_Item) when Num > 0 is
  begin
    Item := Buffer (First);
    First := First + 1;
    Num := Num - 1;
  end Get;
  entry Put (Item : in Data_Item) when Num < Buffer_Size is
  begin
    Last := Last + 1;
    Buffer (Last) := Item;
    Num := Num + 1;
  end Put;
end Bounded_Buffer;
```

## Synchronization by protected objects in Ada95

Protected entries are used like task entries:

```
Buffer : Bounded_Buffer;

select
  Buffer.Put (Some_Data);
or
  delay 10.0;
  -- do something after 10 s.
end select;

select
  Buffer.Get (Some_Data);
else
  -- do something else
end select;
```

## Synchronization by protected objects in Ada95

(barrier evaluation)

Barrier evaluations and task activations:

- on **calling a protected entry**, the associated barrier is evaluated (only those parts of the barrier which might have changed since the last evaluation).
- on **leaving a protected procedure or entry**, related barriers with tasks queued are evaluated (only those parts of the barriers which might have been altered by this procedure / entry or which might have changed since the last evaluation).

Barriers are not evaluated *while inside* a protected object or *on leaving a protected function*.

## Synchronization by protected objects in Ada95

(operations on entry queues)

The **count** attribute indicate the number of tasks waiting at a specific queue:

```
protected Blocker is
  entry Proceed;
private
  Release : Boolean := False;
end Blocker;

protected body Blocker is
  entry Proceed
    when Proceed'count = 5
    or Release is
  begin
    Release := Proceed'count > 0;
    end Proceed;
end Blocker;
```

## Synchronization by protected objects in Ada95

(operations on entry queues)

The **count** attribute indicate the number of tasks waiting at a specific queue:

```
protected type Broadcast is
  entry Receive (M: out Message);
  procedure Send (M: in Message);
private
  New_Message : Message;
  Arrived : Boolean := False;
end Broadcast;

protected body Broadcast is
  entry Receive (M: out Message)
    when Arrived is
  begin
    M := New_Message;
    Arrived := Receive'count > 0;
  end Proceed;
  procedure Send (M: in Message) is
  begin
    New_Message := M;
    Arrived := Receive'count > 0;
  end Send;
end Blocker;
```

## Synchronization by protected objects in Ada95

(entry families, requeue &amp; private entries)

Further refinements on task control by:

- **Entry families:**  
a protected entry declaration can contain a discrete subtype selector, which can be evaluated by the barrier (other parameters cannot be evaluated by barriers) and implements an array of protected entries.
- **Requeue facility:**  
protected operations can use 'requeue' to redirect tasks to other internal, external, or private entries. The current protected operation is finished and the lock on the object is released.  
'Internal progress first'-rule: internally requeued tasks are placed at the **head** of the waiting queue!
- **Private entries:**  
protected entries which are not accessible from outside the protected object, but can be employed as destinations for requeue operations.

## Synchronization by protected objects in Ada95

(requeue &amp; private entries)

How to implement a queue, at which every task can be released only once per triggering event?

```
package Single_Release is
  entry Wait;
  procedure Trigger;
end Single_Release;
```

## Synchronization by protected objects in Ada95

(requeue &amp; private entries)

How to implement a queue, at which every task can be released only once per triggering event?

☞ e.g. by employing a second (private) entry:

```
package Single_Release is
  entry Wait;
  procedure Trigger;
private
  Front_Door,
  Main_Door : Boolean := False;
  entry Queue;
end Single_Release;
```

## Synchronization by protected objects in Ada95

(requeue &amp; private entries)

```
package body Single_Release is
  entry Wait
    when Front_Door is
  begin
    if Wait'Count = 0 then
      Front_Door := False;
      Main_Door := True;
    end if;
    requeue Queue;
  end Wait;

  entry Queue
    when Main_Door is
  begin
    if Queue'count = 0 then
      Main_Door := False;
    end if;
    end Queue;
    procedure Trigger is
    begin
      Front_Door := True;
    end Trigger;
  end Single_Release;
```

opening the main door before requeueing?

## Synchronization

## Synchronization by protected objects in Ada95

(restrictions applying to protected operations)

Code inside a protected procedure, function or entry is bound to non-blocking operations (which would keep the whole protected object locked).

Thus the following operations are prohibited:

- entry call statements
- delay statements
- task creations or activations
- calls to sub-programs which contains a potentially blocking operation
- select statements
- accept statements

⇒ The **requeue** facility allows for a potentially blocking operation, but releases the current lock!

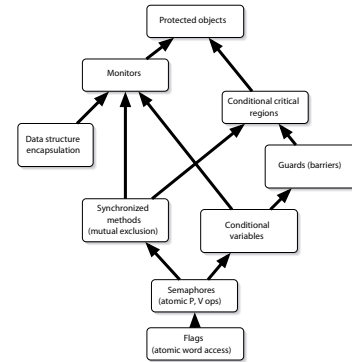
## Summary

## Shared memory based synchronization

## General

Criteria:

- level of abstraction
- centralized vs. distributed concepts
- support for consistency and correctness validations
- error sensitivity
- predictability
- efficiency

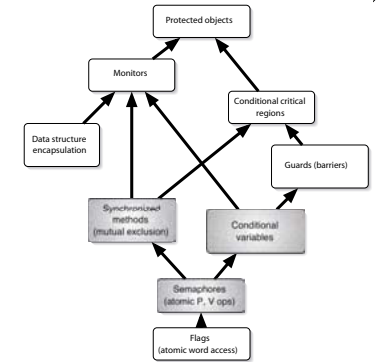


## Summary

## Shared memory based synchronization

## POSIX

- all low level constructs available.
- no connection with the actual data-structures.
- error-prone.
- non-determinism introduced by 'release some' semantics of conditional variables (cond\_signal).

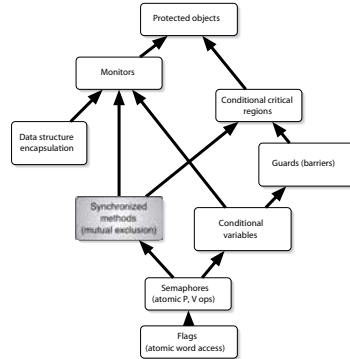


## Summary

## Shared memory based synchronization

## Java

- mutual exclusion (synchronized methods) as the only support.
- general notification feature (no conditional variables)
- non-restricted object oriented extension introduces hard to predict timing behaviours.

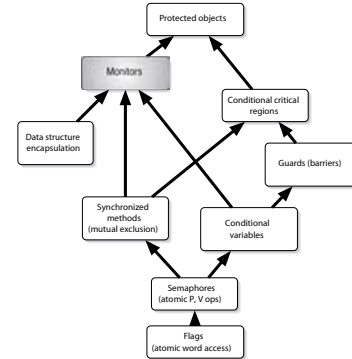


## Summary

## Shared memory based synchronization

## Modula-1, CHILL

- full monitor implementation (Dijkstra-Hoare monitor concept).  
... no more, no less, ...
- ⇒ all features of and criticism about monitors apply.

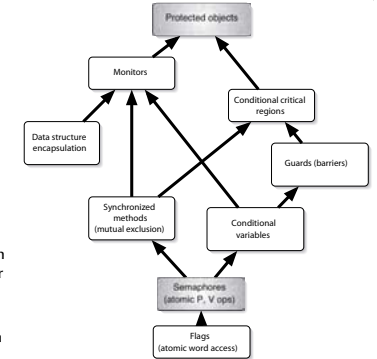


## Summary

## Shared memory based synchronization

## Ada95

- complete synchronization support
  - low-level semaphores for very special cases.
  - predictable timing (⇒ scheduler).
  - ⇒ most memory oriented synchronization conditions are realized by the compiler or the run-time environment directly rather than the programmer.
- (Ada95 is currently without any mainstream competitor in this field)



## Synchronization

## Message-based synchronization

- Synchronization model
  - Asynchronous
  - Synchronous
  - Remote invocation
- Addressing (name space)
  - direct communication
  - mail-box communication
- Message structure
  - arbitrary
  - restricted to 'basic' types
  - restricted to un-typed communications

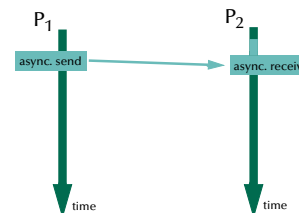
## Synchronization

## Message-based synchronization

## Asynchronous messages

If there is a listener:

⇒ send the message directly



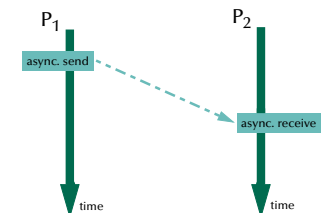
## Synchronization

## Message-based synchronization

## Asynchronous messages

If the receiver becomes available at a later stage:

⇒ the message need to be buffered



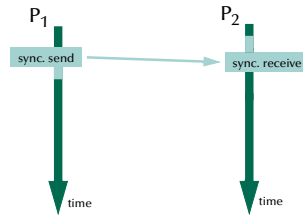


### Message-based synchronization

#### Synchronous messages

Delay the sender:

- until the receiver got the message

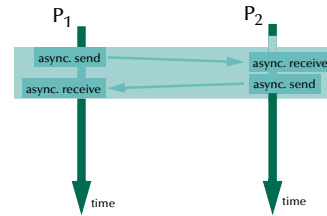


### Message-based synchronization

#### Synchronous messages

Delay the sender:

- until the receiver got the message
- two asynchronous messages required

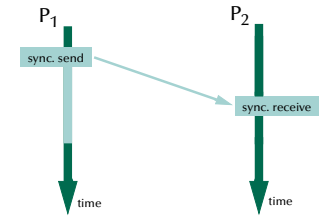


### Message-based synchronization

#### Synchronous messages

Delay the sender until:

- a receiver is available
- a receiver got the message

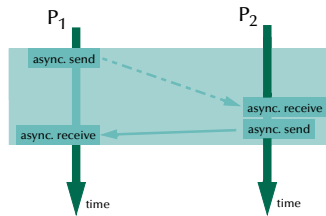


### Message-based synchronization

#### Synchronous messages

If the receiver becomes available at a later stage:

- messages need to be buffered

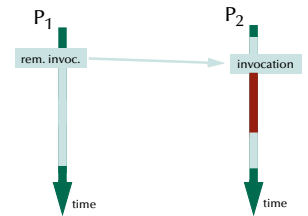


### Message-based synchronization

#### Remote invocation

Delay the sender, until:

- a receiver got the message
- a receiver executed an addressed routine

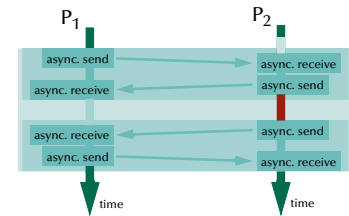


### Message-based synchronization

#### Remote invocation

Delay the sender, until:

- a receiver got the message
- a receiver executed an addressed routine

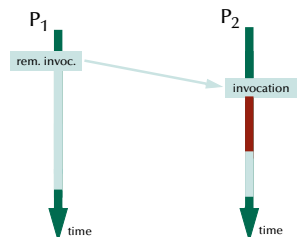


### Message-based synchronization

#### Remote invocation

Delay the sender, until:

- a receiver becomes available
- a receiver got the message
- a receiver executed an addressed routine

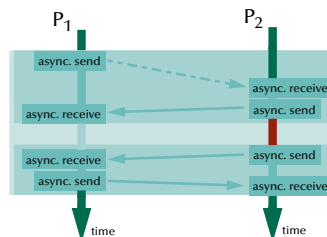


### Message-based synchronization

#### Remote invocation

Delay the sender, until:

- a receiver becomes available
- a receiver got the message
- a receiver executed an addressed routine

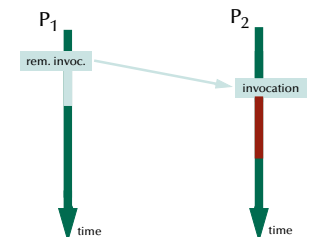


### Message-based synchronization

#### Asynchronous remote invocation

Delay the sender, until:

- a receiver becomes available
- a receiver got the message

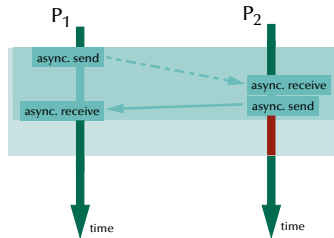


## Message-based synchronization

## Asynchronous remote invocation

Delay the sender, until:

- a receiver becomes available
- a receiver got the message



## Synchronous vs. asynchronous communications

Purpose 'synchronization': ⇨ synchronous messages / remote invocations

Purpose 'in-time delivery': ⇨ asynchronous messages / asynchronous remote invocations

- ⇨ 'Real' synchronous message passing in distributed systems requires hardware support.
- ⇨ Asynchronous message passing requires the usage of (infinite?) buffers.

- Synchronous communications are emulated by a combination of asynchronous messages in some systems.
- Asynchronous communications can be emulated in synchronized message passing systems by introducing 'buffer-tasks' (de-coupling sender and receiver as well as allowing for broadcasts).

## Addressing (name space)

## Direct vs. indirect:

```
send <message> to <process-name>
wait for <message> from <process-name>
send <message> to <mailbox>
wait for <message> from <mailbox>
```

## Asymmetrical addressing:

```
send <message> to ...
wait for <message>
```

- ⇨ Client-server paradigm

## Addressing (name space)

Communication medium:

Connections	Functionality
one-to-one	buffer, queue, synchronization
one-to-many	multicast
one-to-all	broadcast
many-to-one	local server, synchronization
all-to-one	general server, synchronization
many-to-many	general network- or bus-system

## Message structure

- Machine dependent representations need to be taken care of in a distributed environment.
  - Communication system is often outside the typed language environment.
- Most communication systems are handling streams (packets) of a basic element type only.

- ⇨ Conversion routines for data-structures other than the basic element type are supplied ...

... manually (POSIX)  
 ... semi-automatic (Real-time CORBA)  
 ... automatic and are typed-persistent (Ada95)

## Message structure (Ada95)

```
package Ada.Streams is
  pragma Pure (Streams);
  type Root_Stream_Type is abstract tagged limited private;
  type Stream_Element is mod implementation-defined;
  type Stream_Element_Offset is range implementation-defined;
  subtype Stream_Element_Count is
    Stream_Element_Offset range 0..Stream_Element_Offset'Last;
  type Stream_Element_Array is
    array (Stream_Element_Offset range <>) of Stream_Element;
  procedure Read (...) is abstract;
  procedure Write (...) is abstract;
private
  ... -- not specified by the language
end Ada.Streams;
```

## Message structure (Ada95)

Reading and writing values of any type to a stream:

```
procedure S'Write(
  Stream : access Ada.Streams.Root_Stream_Type'Class; Item : in T);
procedure S'Class'Write(
  Stream : access Ada.Streams.Root_Stream_Type'Class; Item : in T'Class);

procedure S'Read(
  Stream : access Ada.Streams.Root_Stream_Type'Class; Item : out T);
procedure S'Class'Read(
  Stream : access Ada.Streams.Root_Stream_Type'Class; Item : out T'Class)
```

Reading and writing values, bounds and discriminants of any type to a stream:

```
procedure S'Output(
  Stream : access Ada.Streams.Root_Stream_Type'Class; Item : in T);

function S'Input(
  Stream : access Ada.Streams.Root_Stream_Type'Class) return T;
```

## Message-based synchronization

Practical message-passing systems:

	"message queues":
POSIX:	⇨ ordered indirect [asymmetrical   symmetrical] asynchronous byte-level many-to-many message passing
CHILL:	"buffers", "signals": ⇨ ordered indirect [asymmetrical   symmetrical] [synchronous   asynchronous] typed [many-to-many   many-to-one] message passing
Occam2:	"channels": ⇨ indirect symmetrical synchronous fully-typed one-to-one message passing
Ada95:	"(extended) rendezvous": ⇨ ordered direct asymmetrical [synchronous   asynchronous] fully-typed many-to-one remote invocation
Java:	no communication via messages available

## Message-based synchronization

Practical message-passing systems:

	* ordered	* symmetrical	* asymmetrical	* synchronous	* asynchronous	direct	indirect	contents	one-to-one	many-to-one	method
POSIX:	*	*	*	*	*		*	bytes		*	message passing
CHILL:	*	*	*	*	*	*	*	typed	*	*	message passing
Occam2:		*	*	*	*	*	*	fully typed	*		message passing
Ada95:	*		*	*	*	*	*	fully typed	*		remote invocation
Java:								no communication via messages available			

- can be used by one writer and one reader process only
- and is synchronous:

```
PAR
  INT reading:
  SEQ i = 0 FOR 1000
    SEQ
      -- generate reading
      SensorChannel ! reading
  INT data:
  SEQ i = 0 FOR 1000
    SEQ
      SensorChannel ? data
      -- employ data
```

tasks are synchronized  
at these points

**CHILL** is the 'CCITT High Level Language', where **CCITT** is the Comité Consultatif International Télégraphique et Téléphonique. The CHILL language development was started in 1973 and standardized in 1979.

```
signal SensorChannel = (int)
...
send SensorChannel (reading)
  to consumer
```

```
receive case
  (SensorChannel in data): ...
esac;
```

**CHILL** is the 'CCITT High Level Language', where **CCITT** is the Comité Consultatif International Télégraphique et Téléphonique. The CHILL language development was started in 1973 and standardized in 1979.

```
signal SensorChannel = (int
...
send SensorChannel (reading
  to consumer
```

```

    receive case
    (SensorChannel in data): ...
    esac;

```

Ada95 supports remote invocations ((extended) rendezvous) in form of:

- **entry points** in tasks
- **full set of parameter profiles** supported

If the local and the remote task are on different architectures, or if an intermediate communication system is employed:

- parameters incl. bounds and discriminants are 'tunnelled' through byte-stream-formats.

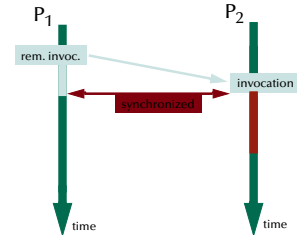
**Synchronization:**

- both tasks are synchronized at the beginning of the remote invocation (⇨ **rendezvous**)
- the calling task is blocked until the remote routine is completed (⇨ **extended rendezvous**)

## Remote invocation (Rendezvous)

Delay the sender, until:

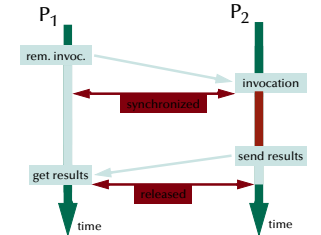
- a receiver becomes available
- a receiver got the message
- a receiver started an addressed routine




Remote invocation  
(Extended rendezvous)

Delay the sender, until:

- a receiver becomes available
- a receiver got the message
- a receiver executed an addressed routine
- a receiver passed the results



(Rendezvous)

```
<entry_name> [(index)] <parameters>  
... -- waiting for synchronization  
...  
...  
  
... accept <entry_name> [(index)]  
...     <parameter_profile>;  
  
...  
...  
...  
...
```

(Rendezvous)

```

...      accept <entry_name> [(index)]
...      <parameter_profile>;
...      -- waiting for synchronization
...      -----
<entry_name> [(index)] <parameters> ←----- synchronized -----→
...
...
...
...

```

(Extended rendezvous)

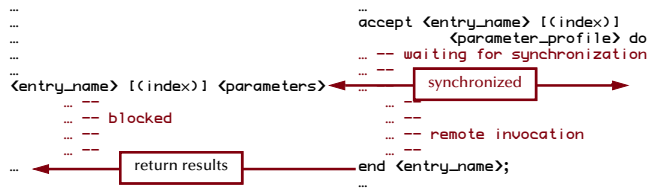
```

... <entry_name> [(index)] <parameters> ...
... -- waiting for synchronization ...
... --
... --
... -- ← [synchronized] → accept <entry_name> [(index)]
...                                     <parameter_profile> do
...                                     ... --
...                                     ... -- blocked
...                                     ... --
...                                     ... -- remote invocation
...                                     ... --
... ← [return results] → end <entry_name>;

```

## Message-based synchronization in Ada95

(Extended rendezvous)



## Message-based synchronization in Ada95

Some things to consider for task-entries:

- In contrast to protected-object-entries, task-entries *can* call other blocking operations.
- Accept statements can be nested (but need to be different).  
⇒ helpful e.g. to synchronize more than two tasks.
- Accept statements can have a dedicated exception handler (like any other code-block).  
Exceptions, which are not handled during the rendezvous phase are propagated to *all* involved tasks.
- Parameters cannot be direct 'access' parameters, but can be access-types.

## Message-based synchronization in Ada95

Some things to consider for task-entries:

- In contrast to protected-object-entries, task-entries *can* call other blocking operations.
- Accept statements can be nested (but need to be different).  
⇒ helpful e.g. to synchronize more than two tasks.
- Accept statements can have a dedicated exception handler (like any other code-block).  
Exceptions, which are not handled during the rendezvous phase are propagated to *all* involved tasks.
- Parameters cannot be direct 'access' parameters, but can be access-types.
- 'count' on task-entries is defined, but is only accessible from inside the tasks owning the entry.
- **Entry families** (arrays of entries) are supported.
- **Private entries** (accessible for internal tasks) are supported.

## Selective waiting

Dijkstra's guarded commands:



- ⇒ the programmer needs to design the alternatives as 'parallel' options:  
all cases need to be covered and overlapping conditions need to lead to the same result

Extremely different philosophy: 'C'-switch:

```

switch (x) {
case 1: r := 3;
case 2: r := 2; break;
case 3: r := 1;
}

```

- ⇒ the sequence of alternatives has a crucial role.

## Message-based selective synchronization in Ada95

Forms of selective waiting:

```

select_statement ::= selective_accept
                  | conditional_entry_call
                  | timed_entry_call
                  | asynchronous_select
                  ... underlying concept: Dijkstra's guarded commands

```

selective\_accept implements ...

- ... wait for more than a single rendezvous at any one time
- ... time-out if no rendezvous is forthcoming within a specified time
- ... withdraw its offer to communicate if no rendezvous is available immediately
- ... terminate if no clients can possibly call its entries

## Message-based selective synchronization in Ada95

selective\_accept in its full syntactical form in Ada95:

```

selective_accept ::= select
  [guard] selective_accept_alternative
  { or [guard] selective_accept_alternative
  [ else sequence_of_statements ]
  end select;

guard ::= when <condition> =>

selective_accept_alternative ::= accept_alternative
                                | delay_alternative
                                | terminate_alternative

accept_alternative ::= accept_statement [ sequence_of_statements ]
delay_alternative  ::= delay_statement [ sequence_of_statements ]
terminate_alternative ::= terminate;

```

## Basic forms of selective synchronization

(select-or)

```

select
  accept ... do ...
end ...
or
  accept ... do ...
end ...
or
  accept ... do ...
end ...
or
  accept ... do ...
end ...
...
end select;

```

- If none of the named entries have been called, the task is suspended until one of the entries is addressed by another task.
- The selection of an accept is non-deterministic, in case that multiple entries are called.  
⇒ The selection can be controlled by means of the real-time systems annex.
- The select statement is completed, when at least one of the entries has been called and those accept-block has been executed.

## Basic forms of selective synchronization

(guarded select-or)

```

select
  when <condition> =>
    accept ... do ...
  end ...
or
  when <condition> =>
    accept ... do ...
  end ...
or
  when <condition> =>
    accept ... do ...
  end ...
...
end select;

```

- Analogue to Dijkstra's guarded commands
- all accepts closed will raise a Program\_Error  
⇒ set of conditions need to be complete

## Basic forms of selective synchronization

(guarded select-or-else)

```

select
  [ when <condition> => ]
    accept ... do ...
  end ...
or
  [ when <condition> => ]
    accept ... do ...
  end ...
or
  [ when <condition> => ]
    accept ... do ...
  end ...
else
  <statements>
...
end select;

```

- If none of the open entries can be accepted immediately, the else alternative is selected.
- There can be only one else alternative and it cannot be guarded.



### Basic forms of selective synchronization

(guarded select-or-delay)

```
select
[ when <condition> => ]
  accept ... do ...
end ...

or

[ when <condition> => ]
  delay ...
  <statements>

or

[ when <condition> => ]
  delay ...
  <statements>

...
end select;
```

- If none of the open entries has been called before the amount of time specified in the earliest open delay alternative, this delay alternative is selected.
- There can be multiple delay alternatives if more than one delay alternative expires simultaneously, either one may be chosen.
- `delay` and `delay until` can be employed.



### Basic forms of selective synchronization

(guarded select-or-terminate)

```
select
[ when <condition> => ]
  accept ... do ...
end ...

or

[ when <condition> => ]
  accept ... do ...
end ...

or

[ when <condition> => ]
  terminate;
```

The terminate alternative is chosen if none of the entries can ever be called again, i.e.:

- all tasks which can possibly call any of the named entries are terminated.
- all remaining active tasks which can possibly call any of the named entries are waiting on selective terminate statements and none of their open entries can be called any longer.

☞ This task and all its dependent waiting-for-termination tasks are terminated together.



### Basic forms of selective synchronization

(guarded select-or-else select-or-delay select-or-terminate)

```
select
  else-delay-terminate
  alternatives
  cannot be mixed!
or
  end ...
[ when <condition> => ]
  delay ...
  <statements>
...
end select;

select
[ when <condition> => ]
  accept ... do ...
end ...

or

[ when <condition> => ]
  terminate;
```



### Non-determinism in selective synchronizations

- ☞ If equal alternatives are given, then the program correctness (incl. the timing specifications) must not be affected by the actual selection.
- If alternatives have different priorities, this can be expressed e.g. by means of the Ada real-time annex.
- Non-determinism in concurrent systems is or can be also introduced by:
  - non-ordered monitor or other queues
  - buffering / routing message passing systems
  - non-deterministic schedulers
  - timer quantization
  - ... any form of asynchronism



### Conditional & timed entry-calls

```
conditional_entry_call ::=
select
  entry_call_statement
  [sequence_of_statements]
else
  sequence_of_statements
end select;

timed_entry_call ::=
select
  entry_call_statement
  [sequence_of_statements]
or
  delay_alternative
end select;

select
  Light_Monitor.Wait_for_Light;
  Lux := True;
else
  Lux := False;
end;
```



### Conditional & timed entry-calls

```
conditional_entry_call ::=
select
  entry_call_statement
  [sequence_of_statements]
else
  sequence_of_statements
end select;

timed_entry_call ::=
select
  entry_call_statement
  [sequence_of_statements]
or
  delay_alternative
end select;

select
  Light_Monitor.Wait_for_
  Lux := True;
else
  Lux := False;
end;
```

There is only **one entry call** and either **one 'else'** or **one 'or delay'**



### Conditional & timed entry-calls

```
conditional_entry_call ::=
select
  entry_call_statement
  [sequence_of_statements]
else
  sequence_of_statements
end select;

timed_entry_call ::=
select
  entry_call_statement
  [sequence_of_statements]
or
  delay_alternative
end select;

select
  Light_Monitor.Wait_for_Light;
  Lux := True;
else
  Lux := False;
end;
```

The idea in both cases is to **withdraw a synchronization request** and **not** to implement polling or busy-waiting.



### Synchronization

- **Shared memory based synchronization**
  - Flags, condition variables, semaphores, ...
  - ... conditional critical regions, monitors, protected objects.
  - Guard evaluation times, nested monitor calls, deadlocks, ...
  - ... simultaneous reading, queue management.
  - Synchronization and object orientation, blocking operations and re-queuing.
- **Message based synchronization**
  - Synchronization models, addressing modes, message structures
  - Selective accepts, selective calls
  - Indeterminism in message based synchronization



### Synchronization may lead to

☞ **DEADLOCKS**

... a closer look on deadlocks and what can be done about them ...



## Reserving resources in reverse order

```
var reserve_1, reserve_2: semaphore := 1;
```

```
process P1;
statement X;
wait (reserve_1);
wait (reserve_2);
statement Y; - employ resources
signal (reserve_2);
signal (reserve_1);
statement Z;
end P1;

process P2;
statement A;
wait (reserve_2);
wait (reserve_1);
statement B; - employ resources
signal (reserve_1);
signal (reserve_2);
statement C;
end P2;
```

Sequence of operations :  $[A \mid X] \rightarrow \{[B \rightarrow Y] \text{ xor } [Y \rightarrow B]\} \rightarrow [C \mid Z]$   
 or :  $[A \mid X] \rightarrow \text{deadlocked!}$

## Circular dependencies

```
var reserve_1, reserve_2, reserve_3: semaphore := 1;
```

```
process P1;
statement X;
wait (reserve_1);
wait (reserve_2);
statement Y;
signal (reserve_2);
signal (reserve_1);
statement Z;
end P1;

process P2;
statement A;
wait (reserve_2);
wait (reserve_3);
statement B;
signal (reserve_3);
signal (reserve_2);
statement C;
end P2;

process P3;
statement K;
wait (reserve_3);
wait (reserve_1);
statement L;
signal (reserve_1);
signal (reserve_3);
statement M;
end P3;
```

Sequence of operations :  $[A \mid X \mid K] \rightarrow \{[B \rightarrow Y \rightarrow L] \text{ xor } \dots\} \rightarrow [C \mid Z \mid M]$   
 or :  $[A \mid X \mid K] \rightarrow \text{deadlocked!}$

## Necessary deadlock conditions:

1. **Mutual exclusion:**  
resources cannot be used simultaneously
2. **Hold and wait:**  
a process applies for a resource, while it is holding another resource (sequential requests)
3. **No pre-emption:**  
resources cannot be pre-empted; only the process itself can release resources
4. **Circular wait:**  
a ring list of processes exists, where every process waits for release of a resource by the next one

⇒ system may be deadlocked, when **all** these conditions apply!

## Deadlock strategies:

1. Ignorance  
⇒ Kill unresponsive processes
2. Deadlock detection & recovery  
⇒ find deadlocked processes and recover the system in a coordinated way
3. Deadlock avoidance  
⇒ the resulting system state is checked before any resources are actually assigned
4. Deadlock prevention  
⇒ the system prevents deadlocks by its structure

## Deadlock prevention

(remove one of the four deadlock conditions)

1. **Mutual exclusion:**  
Applicable to specific cases only; usually this can only be removed by replication of resources.
2. **Hold and wait:**  
Processes are forced to allocate all their required resources at once, often at the time of admittance to the main dispatcher – done in many static realtime-systems.
3. **No pre-emption:**  
If the current state of a resource can be stored and restored easily, then they can be pre-empted. Usually resources are pre-empted from processes, which are currently not ready to run.
4. **Circular wait:**  
A circular wait can be avoided by a global ordering of all resources, e.g. resources can only be requested in a specific order – hard to maintain in a dynamic system configuration.

## Resource Allocation Graphs

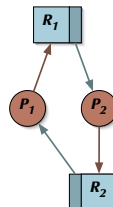
(Silberschatz, Galvin & Gagne)

$RAG = \{V, E\}$ ; vertices and edges  
 $V = P \cup R$ ; vertices are processes or resource types:  
 $P = \{P_1, P_2, \dots, P_n\}$ ; processes  
 $R = \{R_1, R_2, \dots, R_k\}$ ; resource types  
 $E = E_r \cup E_a \cup E_c$ ; claims, requests and assignments  
 $E_c = \{P_i \rightarrow R_j, \dots\}$ ; claims  
 $E_r = \{P_i \rightarrow R_j, \dots\}$ ; requests  
 $E_a = \{R_j \rightarrow P_i, \dots\}$ ; assignments

Note: a resourcefully may have more than one instance

## Resource Allocation Graphs

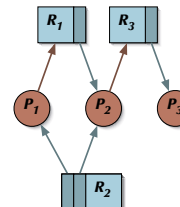
(Silberschatz, Galvin & Gagne)



the two process, reverse allocation deadlock:

## Resource Allocation Graphs

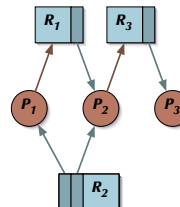
(Silberschatz, Galvin & Gagne)



Is this a deadlock situation? ⇒

## Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)



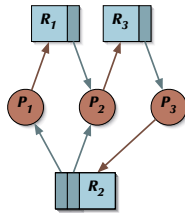
no, there is no circular dependency

## Deadlocks

### Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

Is this a deadlock situation? ⇨



## Deadlocks

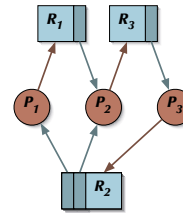
### Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

yes, there are circular dependencies:

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$   
as well as:  $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

⇨ **IF some processes are deadlocked, THEN there are cycles in the resource allocation graph**



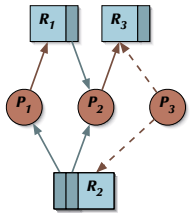
## Deadlocks

### Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

Assuming all claims of  $P_3$  are known in advance,

⇨ Could the deadlock situation be avoided?



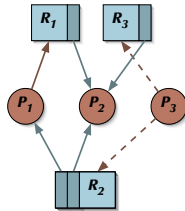
## Deadlocks

### Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

yes, when resources are assigned so that there are no resulting circular dependencies:

⇨ in this case: assign  $R_3$  to  $P_2$  (instead of  $P_3$ )



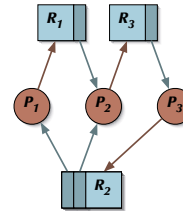
## Deadlocks

### Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$   
as well as:  $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

⇨ **ARE some processes deadlocked, IF there are cycles in the resource allocation graph?**



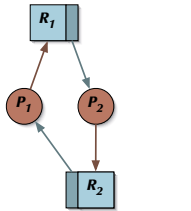
## Deadlocks

### Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

yes,  
if there is only one instance per resource type:

⇨ **IF there are cycles in the resource allocation graph AND there is only one instance per resource type, THEN some processes are deadlocked!**



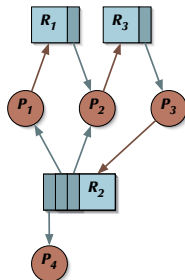
## Deadlocks

### Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

no,  
if there is more than one instance per resource type:

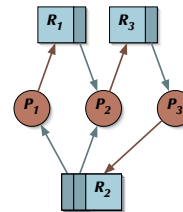
⇨ **IF there are cycles in the resource allocation graph AND there is more than one instance per resource type, THEN some processes may be deadlocked!**



## Deadlocks

### How to detect deadlocks in the general case?

(of multiple instances per resource)



## Deadlocks

### Banker's algorithm

There are  $n$  processes and  $m$  resource types in the system. Let  $i \in 1 \dots n$  and  $j \in 1 \dots m$ :

- **Allocated**[ $i, j$ ]  
⇨ the number of resources of type  $j$  allocated by process  $i$ .
- **Free**[ $j$ ]  
⇨ the number of available resources of type  $j$ .
- **Claimed**[ $i, j$ ]  
⇨ the number of resources of type  $j$  required by process  $i$  to complete eventually.
- **Request**[ $i, j$ ]  
⇨ the number of currently requested resources of type  $j$  by process  $i$ .

Temporary variables:

- **Completed**[ $i$ ]: boolean vector indicating processes, which may complete right now.
- **Simulated\_Free**[ $j$ ]: available resources, if some processes complete and de-allocate.

## Banker's algorithm

Checking for a deadlock situation

1.  $Simulated\_Free \leftarrow Free; \forall i: Completed[i] \leftarrow False$
2. **While**  $\exists i: \neg Completed[i]$   
**and**  $\forall j: Requested[i, j] < Simulated\_Free[j]$  **do**: {request  $i$  can be granted}  
 $\forall j: Simulated\_Free[j] \leftarrow Simulated\_Free[j] + Allocated[i, j]$   
 $Completed[i] \leftarrow True$
3. **If**  $\forall i: Completed[i]$  **then** the system is **deadlock-free!**  
 (otherwise all processes  $i$  with  $Completed[i] = False$  are deadlocked)

## Banker's algorithm

Checking the current system state

1.  $Simulated\_Free \leftarrow Free; \forall i: Completed[i] \leftarrow False$
2. **While**  $\exists i: \neg Completed[i]$   
**and**  $\forall j: Claimed[i, j] < Simulated\_Free[j]$  **do**: {meaning process  $i$  can complete}  
 $\forall j: Simulated\_Free[j] \leftarrow Simulated\_Free[j] + Allocated[i, j]$   
 $Completed[i] \leftarrow True$
3. **If**  $\forall i: Completed[i]$  **then** the system is **safe!**  
 (e.g. no process is currently deadlocked and no process can be deadlocked in any future state)

## Banker's algorithm

Checking the validity of a resource request

```

If (Request < Claimed) and (Request < Free) then
  Free      := Free      - Request;
  Claimed   := Claimed   - Request;
  Allocated := Allocated + Request;
  ⌞ Apply system state check (as above)
  If System_is_safe then
    ⌞ Actually grant request
  else
    -- restore former system state (Free, Claimed, Allocated)
  end if;
end if;

```

## Deadlock recovery

- ⌞ Stop or restart one or multiple of the deadlocked processes and reclaim its resources
- ⌞ Pre-empt one of the involved resources (and restore an earlier state of the victim process)

Deadlock recovery does not deal with the source of the problem!  
 (the system may deadlock again right away)

- ⌞ use deadlock prevention or deadlock avoidance instead

## Deadlocks

- **Ignorance & recovery**
  - ⌞ 'kill some seemingly persistently blocked processes from time to time' (exasperation)
- **Deadlock detection & recovery**
  - ⌞ multiple methods for detection, e.g. resource allocation graphs, Banker's algorithm
  - ⌞ recovery is mostly 'ugly'
- **Deadlock avoidance**
  - ⌞ check system safety before allocating resources, e.g. Banker's algorithm
- **Deadlock prevention**
  - ⌞ eliminate one of the pre-conditions for deadlocks

## Purpose of scheduling

A scheduling scheme provides two features:

- **Ordering the use of resources** (e.g. CPUs, networks)
- **Predicting the worst-case behaviour** of the system  
 when the scheduling algorithm is applied  
 ... in case that some or all information about the expected resource requests are known

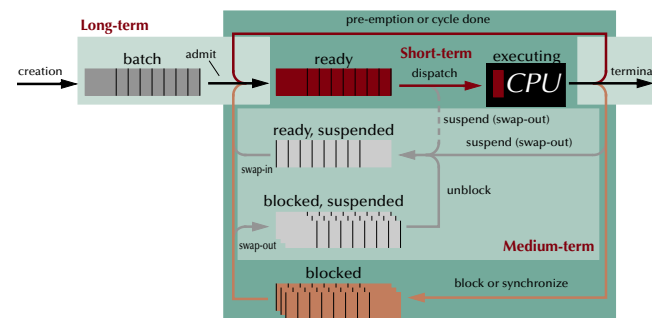
A prediction can then be used

- ⌞ at compile-run: to **confirm the overall resource requirements** of the application, or
- ⌞ at run-time: to **permit acceptance** of additional usage/reservation requests.

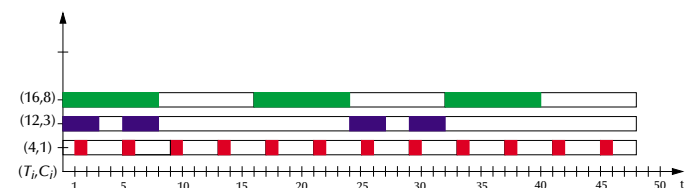
## Criteria for scheduling methods

	<b>Performance criteria:</b> minimize the ...	<b>Predictability criteria:</b> minimize the diversion from given
<b>Process / user perspective:</b>		
Waiting time	maximum / average / variance	minimal and maximal waiting times
Response time	maximum / average / variance	minimal and maximal response times
Turnaround time	maximum / average / variance	deadlines
<b>System perspective:</b>		
Throughput	maximum / average / variance of CPU time per process	—
Utilization	CPU idle time	—

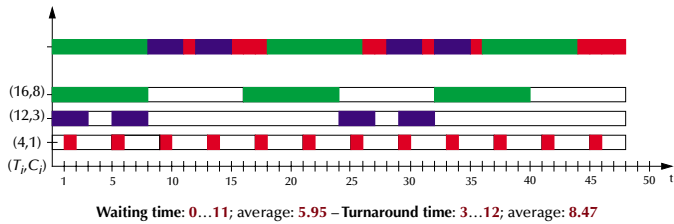
## Time scales of scheduling



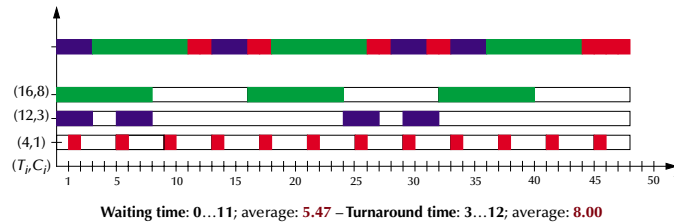
## Example: Requested times



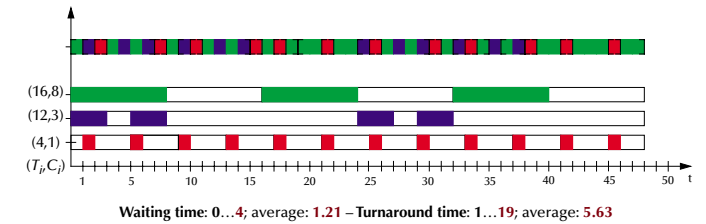
First come, first served (FCFS) – bad case: (arrival order: ■, ■, ■)



First come, first served (FCFS) – nice case: (arrival order: ■, ■, ■)



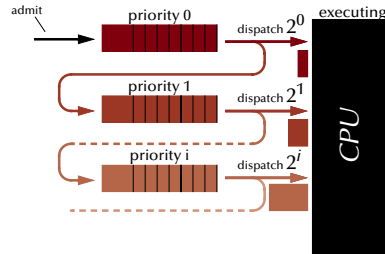
Round robin (RR) – pre-emption



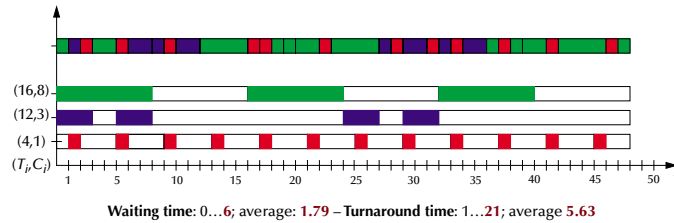
Feedback with  $2^i$  pre-emption intervals – pre-emption

- implement multiple hierarchical ready-queues
- fetch processes from the highest filled ready queue
- dispatch more CPU time for lower priorities ( $2^i$  units)

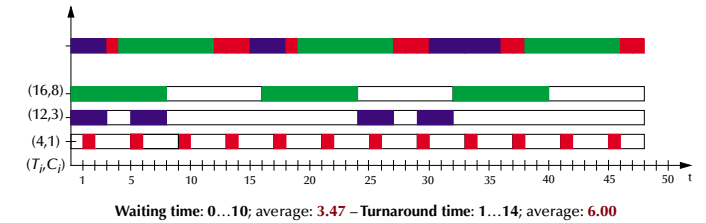
- ☞ processes on lower ranks may suffer starvation
- ☞ new and short tasks will be preferred



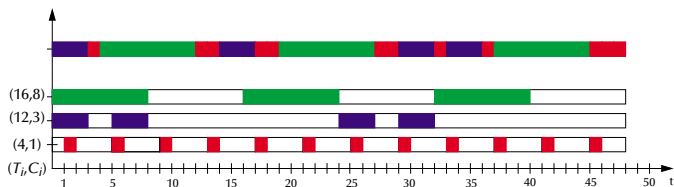
Feedback with  $2^i$  pre-emption intervals – pre-emption



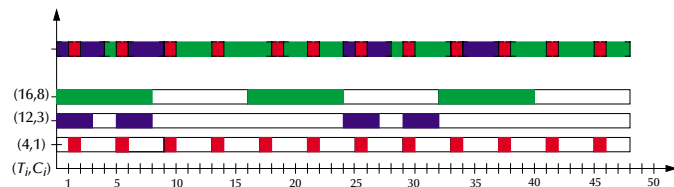
Shortest job first (SJF) –  $C_i$  is known



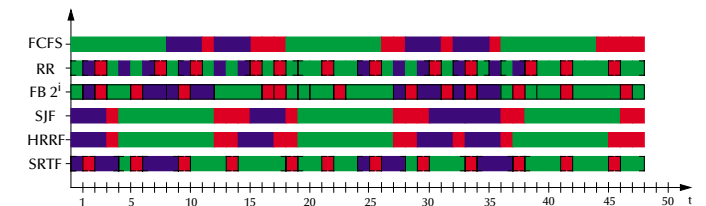
Highest response ratio first (HRRF) –  $C_i$  is known



Shortest remaining time first (SRTF) –  $C_i$  is known + pre-emption



Non-realtime scheduling methods



- ☞ CPU utilization: 100% in all cases.
- ☞ Pre-emptive methods perform better, assuming that the overhead is negligible.
- ☞ Knowledge of  $C_i$  (computation times) has a significant impact on scheduler performance.

	Selection	Pre-emption	Waiting in high load situations	Turnaround	Preferred processes	Starvation possible?
FCFS	$\max(W_i)$	no	possibly long	possibly long	long	no
RR	equal share	yes	bound	possibly long	none	no
Feedback	priority queues	yes	short on average	very short on average, large maximum	short	yes
SJF	$\min(C_i)$	no	short on average	short on average	short	yes
HRRF	$\max\left(\frac{W_i + C_i}{C_i}\right)$	no	short on average, lower variance	short on average, lower variance	balanced, towards short	no
SRTF	$\min(C_i - E_i)$	yes	very short on average	very short on average, large maximum	short	yes

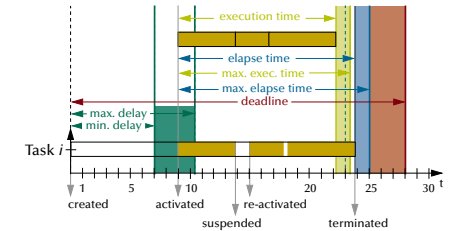
## Towards predictable scheduling ...

- Task behaviours are more specified (restricted).
  - Task requirements from the operating systems are more specific.
  - Task sets are often fully or mostly static.
  - Sporadic and urgent requests (e.g. user interaction, alarms) need to be addressed.
- CPU-utilization and throughput (system oriented performance measures) are not important!

## Temporal scopes

## Common attributes:

- Minimal & maximal delay after creation
- Maximal elapsed time
- Maximal execution time
- Absolute deadline



## Some common scope attributes

## Temporal Scopes can be:

<b>Periodic</b>	– e.g. controllers, samplers, monitors
<b>Aperiodic</b>	– e.g. 'periodic on average' tasks, burst requests
<b>Sporadic / Transient</b>	– e.g. mode changes, occasional services

## Deadlines (absolute, elapse, or execution time) can be:

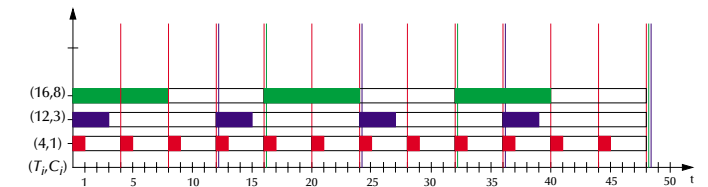
<b>Hard</b>	– single failure leads to severe malfunction
<b>Firm</b>	– results are meaningless after the deadline
<b>Soft</b>	– only multiple or permanent failures threaten the whole system – results may still be useful after the deadline

## A simple process model

- The number of processes in the system is fixed.
- All processes are periodic and all periods are known.
- All deadlines are identical with the process cycle times (periods).
- The worst case execution time is known for all processes.
- All processes are independent.
- All processes are released at once.
- The task-switching overhead is negligible.

→ this model can only be applied to a specific group of hard real-time systems. (extensions to this model will be discussed later in this chapter).

## Introducing deadlines

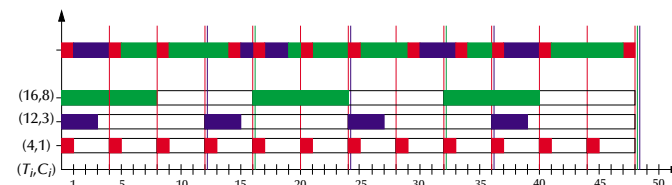


## Earliest deadline first (EDF)

- Determine (one of) the processe(s) with the closest deadline.
- Execute this process
  - until it finishes
  - or until another process' deadline is found closer then the current one.

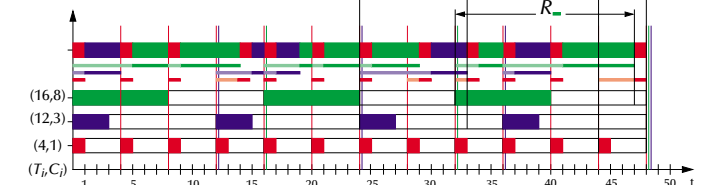
- Pre-emptive scheme
- Dynamic scheme, since the dispatched process is selected at run-time, due to the current deadlines.

## Earliest deadline first



- Schedule the earliest deadline first
- Avoid task switches (in case of equal deadlines)

## Earliest deadline first: Response times

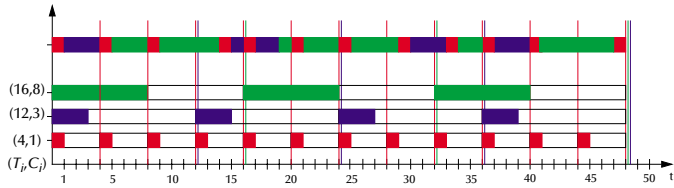


worst case response times  $R_i$  (maximal time in which the request from task  $T_i$  is served):

- can be close or identical to deadlines.
- small or none spare capacity, if any task misses its expected computation time.



### Earliest deadline first: Maximal utilization



maximal possible utilization:  $\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$   $\Leftrightarrow$  sufficient & necessary test!  
with  $C_i, T_i$  the computation and cycle times of task  $i$   
(the deadlines  $D_i$  are assumed to be identical with the cycles times  $T_i$  here)

### Fixed Priority Scheduling (FPS), rate monotonic

- Each process is assigned a fixed priority according to its cycle time  $T_i$ :

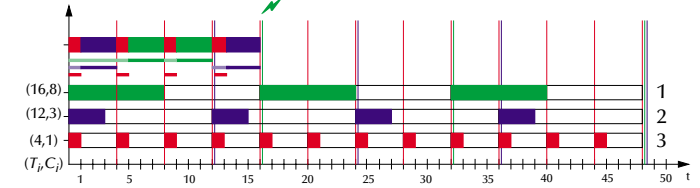
$$T_i < T_j \Rightarrow P_i > P_j$$

- At any point in time: dispatch the process with the highest priority

- Pre-emptive scheme
- Static scheme, since the dispatch order of processes is fixed and calculated off-line.

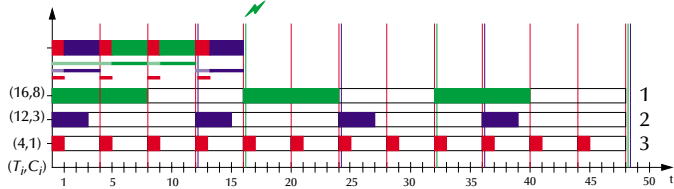
- Rate monotonic ordering is **optimal** (in the framework of fixed priority schedulers), i.e. **if** a process set is schedulable under a FPS-scheme, **then** it is also schedulable by applying rate monotonic priorities.

### Rate monotonic priorities



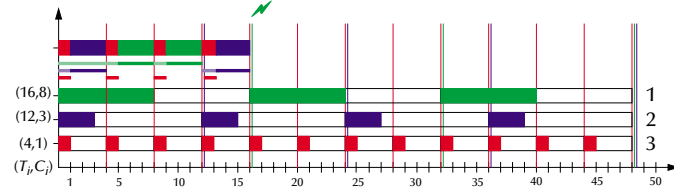
assign task priorities according to the cycle times  $T_i$  (identical to deadline  $D_i$ ).

### Rate monotonic priorities



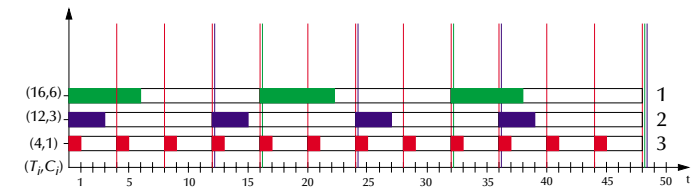
max. utilization test:  $\sum_{i=1}^n \frac{C_i}{T_i} \leq N \left( 2^{\frac{1}{N}} - 1 \right)$   $\Leftrightarrow$  sufficient, but not necessary test!

### Rate monotonic priorities



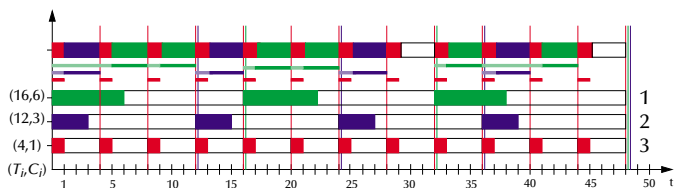
utilization test:  $\sum_{i=1}^n \frac{C_i}{T_i} = 1 > 0.779 \approx N \left( 2^{\frac{1}{N}} - 1 \right)$   $\Leftrightarrow$  not guaranteed!

### Rate monotonic priorities (reduced requests)



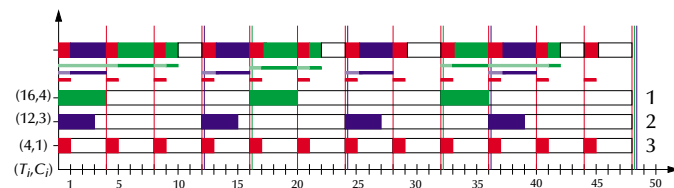
max. utilization test:  $\sum_{i=1}^n \frac{C_i}{T_i} \leq N \left( 2^{\frac{1}{N}} - 1 \right)$

### Rate monotonic priorities (reduced requests)



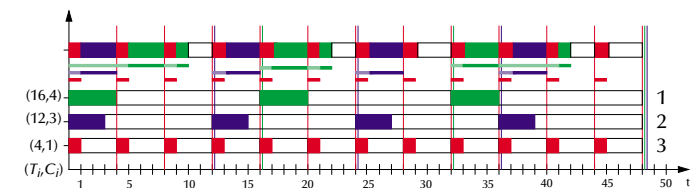
utilization:  $\frac{6}{16} + \frac{3}{12} + \frac{1}{4} = 0.875 > 0.779 \approx 3 \left( 2^{\frac{1}{3}} - 1 \right)$ ;  $\sum_{i=1}^n \frac{C_i}{T_i} \leq N \left( 2^{\frac{1}{N}} - 1 \right)$   $\Leftrightarrow$  not guaranteed!

### Rate monotonic priorities (further reduced requests)



utilization:  $\frac{4}{16} + \frac{3}{12} + \frac{1}{4} = 0.75 \leq 0.779 \approx 3 \left( 2^{\frac{1}{3}} - 1 \right)$ ;  $\sum_{i=1}^n \frac{C_i}{T_i} \leq N \left( 2^{\frac{1}{N}} - 1 \right)$   $\Leftrightarrow$  guaranteed!

### Response time analysis (further reduced requests)



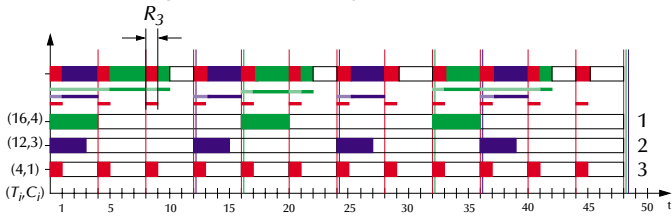
calculate the worst case response times for each task individually.



## Operating Systems & Networks

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis (further reduced requests)



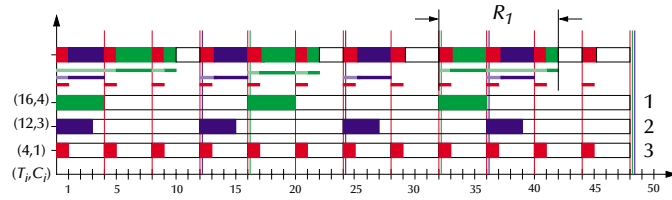
for the highest priority task:  $R_3 = C_3$



## Operating Systems & Networks

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis (further reduced requests)



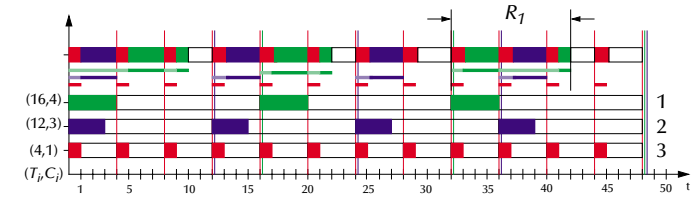
for other tasks:  $R_i = C_i + I_i = \text{computation } C_i + \text{interference } I_i$



## Operating Systems & Networks

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis (further reduced requests)



for other tasks:  $R_i = C_i + \sum_{j>i} \left\lceil \frac{R_j}{T_j} \right\rceil C_j$



## Operating Systems & Networks

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis

$$R_i = C_i + \sum_{j>i} \left\lceil \frac{R_j}{T_j} \right\rceil C_j$$

fixed-point equation!

form recurrent equation:  $R_i^{k+1} = C_i + \sum_{j>i} \left\lceil \frac{R_j^k}{T_j} \right\rceil C_j$  (1)

starting with  $R_i^0 = C_i$

Iterate (1) until  $R_i^{k+1} = R_i^k$  or  $R_i^{k+1} > T_i$



## Operating Systems & Networks

Dynamic scheduling: Earliest Deadline First (EDF)

Response time analysis

The worst case for EDF is *not* necessarily when all tasks are released at once!

all possible combinations in a full hyper-cycle need to be considered!

- The response times are bounded by the cycle times as long as the maximal utilization is  $\leq 1$ .
- Other tasks need to be considered only, if their deadline is closer or equal to the current task.



## Operating Systems & Networks

Dynamic scheduling: Earliest Deadline First (EDF)

Response time analysis

$$R_i(a) = \left\lfloor \frac{a}{T_i} + 1 \right\rfloor C_i + \sum_{j \neq i, \min} \left\lceil \frac{R_j(a)}{T_j} \right\rceil_{\max} \left\{ 0, \left\lfloor \frac{a + T_i - T_j}{T_j} \right\rfloor + 1 \right\} C_j$$

$$R_i^{k+1}(a) = \left\lfloor \frac{a}{T_i} + 1 \right\rfloor C_i + \sum_{j \neq i, \min} \left\lceil \frac{R_j^k(a)}{T_j} \right\rceil_{\max} \left\{ 0, \left\lfloor \frac{a + T_i - T_j}{T_j} \right\rfloor + 1 \right\} C_j$$

starting with  $R_i^0(a) = a + C_i$

Iterate (2) until  $R_i^{k+1}(a) = R_i^k(a)$

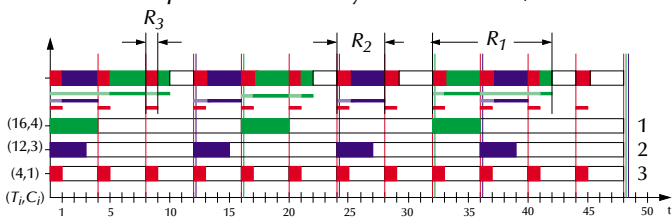
$R_i = \max_{a \in A} \{R_i(a) - a\}$ ; where  $A = \text{scm}\{T_i\}$



## Operating Systems & Networks

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis (further reduced requests)



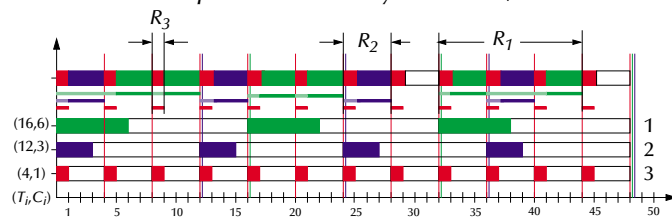
$R_i = C_i + \sum_{j>i} \left\lceil \frac{R_j}{T_j} \right\rceil C_j$ ;  $R_3 = 1\checkmark$ ;  $R_2 = 4\checkmark$ ;  $R_1 = 10\checkmark$  and  $\sum_{i=1}^n \frac{C_i}{T_i} \leq N \left( 2^{\frac{1}{N}} - 1 \right) \checkmark$



## Operating Systems & Networks

Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis (reduced requests)



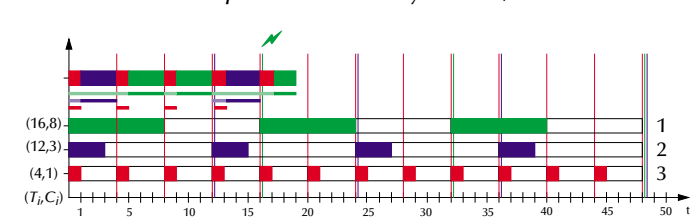
$R_i = C_i + \sum_{j>i} \left\lceil \frac{R_j}{T_j} \right\rceil C_j$ ;  $R_3 = 1\checkmark$ ;  $R_2 = 4\checkmark$ ;  $R_1 = 12\checkmark$  but  $\sum_{i=1}^n \frac{C_i}{T_i} > N \left( 2^{\frac{1}{N}} - 1 \right) \times$



## Operating Systems & Networks

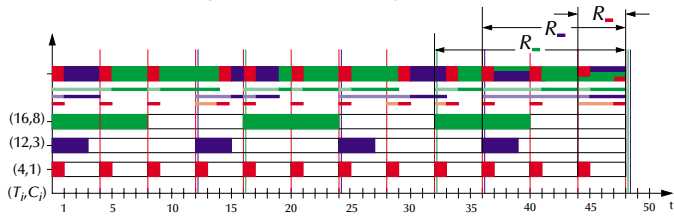
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis (full requests)



$R_i = C_i + \sum_{j>i} \left\lceil \frac{R_j}{T_j} \right\rceil C_j$ ;  $R_3 = 1\checkmark$ ;  $R_2 = 4\checkmark$ ;  $R_1 = 19\checkmark$  and  $\sum_{i=1}^n \frac{C_i}{T_i} > N \left( 2^{\frac{1}{N}} - 1 \right) \times$

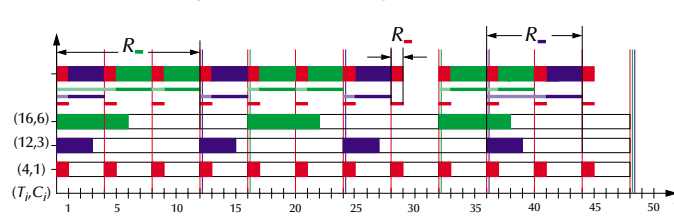
### Response time analysis (full requests)



⇒ testing all combinations in a hyper-period: LCM of  $\{T_i\}$  — here: 48

$$R_- : 16 \leq 16v = T_-; \quad R_- : 12 \leq 12v = T_-; \quad R_- : 4 \leq 4v = T_-$$

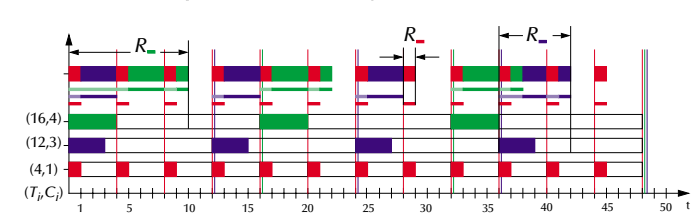
### Response time analysis (reduced requests)



⇒ relaxed task-set changes:

$$R_- : 16 \rightarrow 12 \leq 16v = T_-; \quad R_- : 12 \rightarrow 8 \leq 12v = T_-; \quad R_- : 4 \rightarrow 1 \leq 4v = T_-$$

### Response time analysis (further reduced requests)



⇒ further relaxed task-set changes:

$$R_- : 12 \rightarrow 10 \leq 16v = T_-; \quad R_- : 8 \rightarrow 6 \leq 12v = T_-; \quad R_- : 1 \rightarrow 1 \leq 4v = T_-$$

### Response time analysis (comparison)

	Fixed Priority Scheduling		Earliest Deadline First	
	utilization test	response times $\{R_i\}$	utilization test	response times $\{R_i\}$
$\{(T_p, C_p)\} = \{(16, 8); (12, 3); (4, 1)\}$	✗ (1.000)	$\{x, 4, 1\}$	✓ (1.000)	$\{16, 12, 4\}$
$\{(T_p, C_p)\} = \{(16, 6); (12, 3); (4, 1)\}$	✗ (0.875)	$\{12, 4, 1\}$	✓ (0.875)	$\{12, 8, 1\}$
$\{(T_p, C_p)\} = \{(16, 4); (12, 3); (4, 1)\}$	✓ (0.750)	$\{10, 4, 1\}$	✓ (0.750)	$\{10, 6, 1\}$
	$\sum_{i=1}^n \frac{C_i}{T_i} \leq N \left( \frac{1}{2^N} - 1 \right)$	$C_i + \sum_{j>i} \left\lceil \frac{R_j}{T_j} \right\rceil C_j$	$\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$	check full hyper-cycle

### Fixed Priority Scheduling ↔ Earliest Deadline First

- EDF can handle higher (full) utilization than FPS.
- FPS is easier to implement and implies less run-time overhead
- Graceful degradation features (resource is over-booked):
  - FPS: processes with lower priorities will always miss their deadlines first.
  - EDF: any process can miss its deadline ⇒ and can trigger a cascade of failed deadlines.
- Response time analysis and utilization tests:
  - FPS:  $O(n)$  utilization test — response time analysis: fixed point equation
  - EDS:  $O(n)$  utilization test — response time analysis: fixed point equation in hyper-cycle

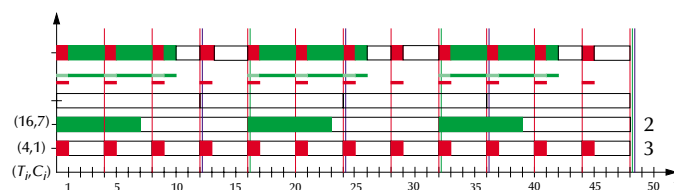
### Extensions which we will introduce:

- tasks are periodic
  - we will introduce **sporadic** and **aperiodic** processes
- tasks are independent
  - we will introduce **schedules for interacting tasks**
- deadlines are identical with task's period time ( $D = T$ )
  - Real-time course
- pre-emptive scheduling
  - Real-time course
- worst case execution times are known
  - Real-time course

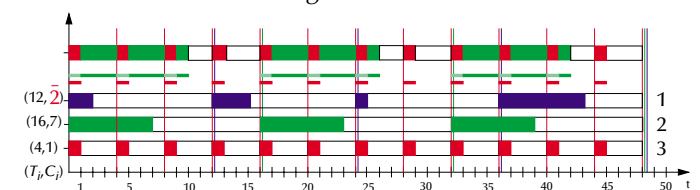
... including

aperiodic, sporadic & 'soft' real-time tasks

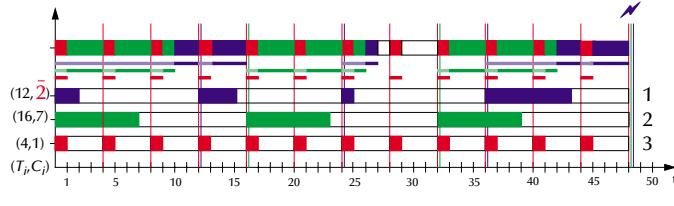
### Hard real-time tasks



### Introducing soft real-time tasks

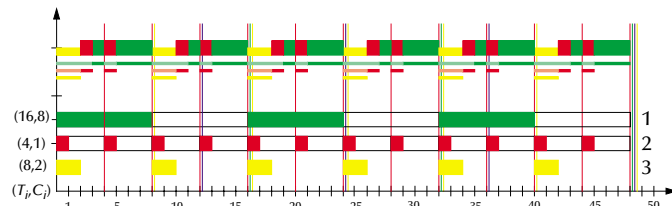


### Introducing soft real-time tasks



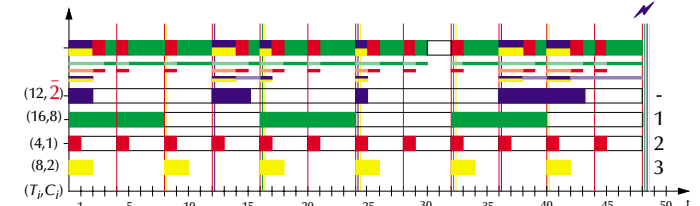
- ☞ set can be scheduled using average computation and period times
- ☞ hard real-time tasks can be scheduled under worst case conditions (including worst case behaviours of soft real-time tasks)

### Introducing a server task



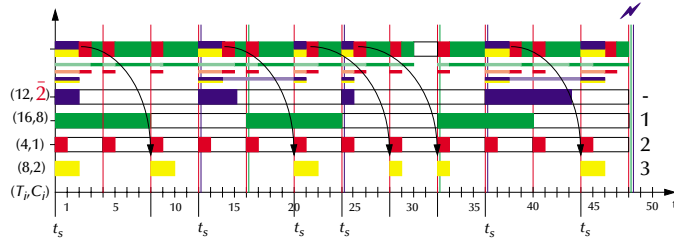
Server is established at a high priority

### Introducing a server task: Deferrable Server



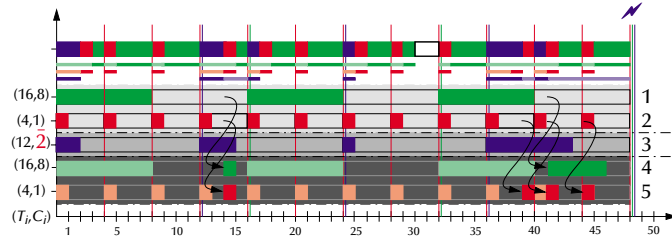
☞ Deferrable Server (DS): Capacity replenished every  $T_s$  (here: 8)

### Introducing a server task: Sporadic Server



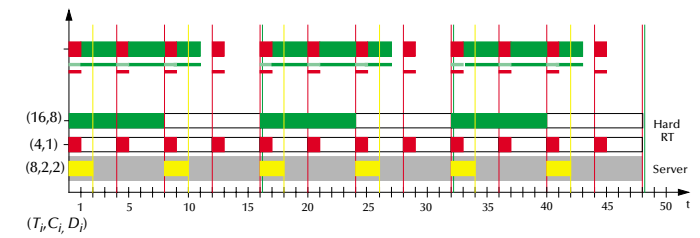
☞ Sporadic Server (SS): Capacity replenished  $T_s$  units after  $t_s$  ⇒ POSIX

### Introducing dual priorities

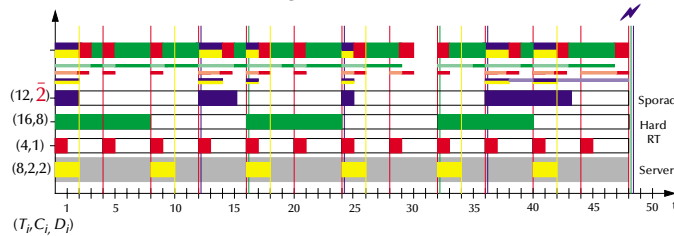


☞ start hard rt-tasks in low priorities; promote them in time to higher ones

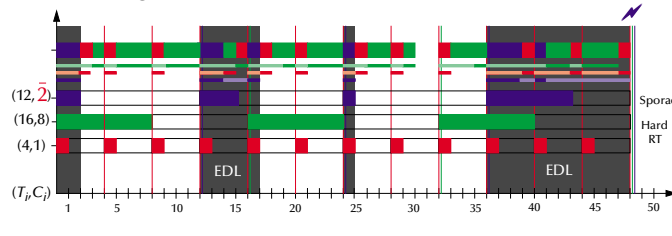
### Introducing a server task to EDF



### Introducing a server task to EDF

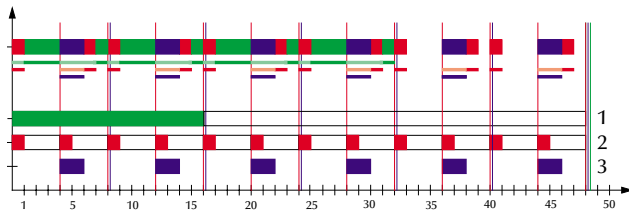


### Switching between EDF & Earliest Deadline Last (EDL)



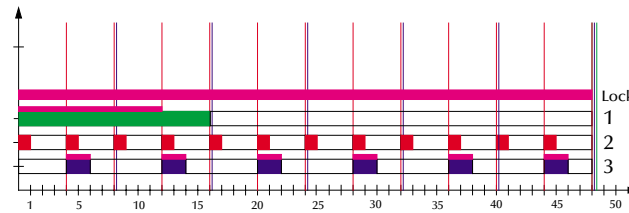
... including  
task interdependencies

### Schedule for independent tasks



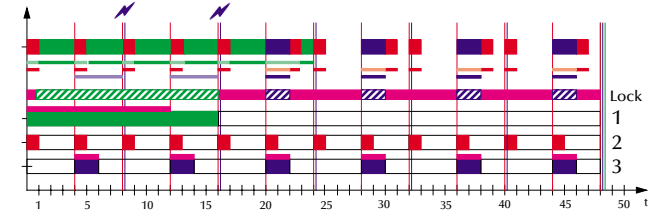
(independent task set)

### Synchronized via lock



(interdependent task set  $\Rightarrow$  lock ■ shared between ■ and ■)

### Synchronized via lock



$\Rightarrow$  Priority inversion

(interdependent task set  $\Rightarrow$  lock ■ shared between ■ and ■)

### Priority inheritance

Task  $t_i$  inherits the priority of  $t_j$ , if:

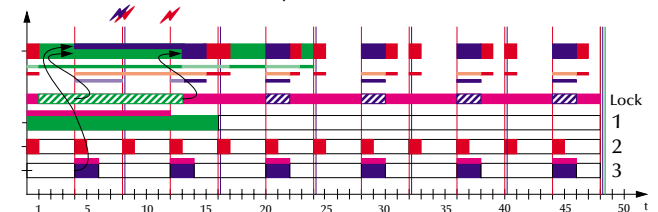
1.  $P_i < P_j$
2. task  $t_j$  has locked a resource  $Q$
3. task  $t_i$  is blocked waiting for resource  $Q$  to be released

### Priority inheritance

$$\text{Maximal blocking time for task } t_i: B_i = \sum_{r=1}^R \text{usage}(r, i) C(r)$$

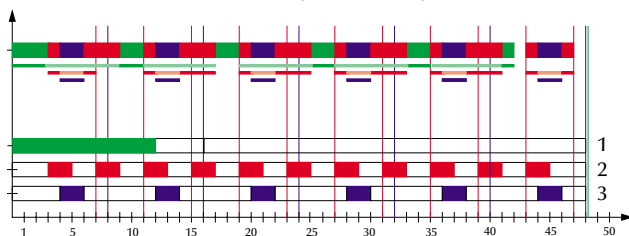
- with  $R$  the number of critical sections
  - $\text{usage}(r, i)$  a boolean (0/1) function indicating that  $r$  is used by at least one  $t_j$  with  $P_j < P_i$  and at least one  $t_k$  with  $P_k \geq P_i$
  - $C(r)$  is the worst case computation time in critical section  $r$
- a task can only be blocked once for each employed resource!

### Priority inheritance



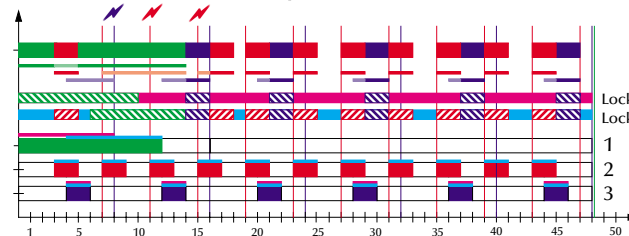
(■ inherits priority of ■, when ■ is in lock and ■ is dispatched)

### A more complex example



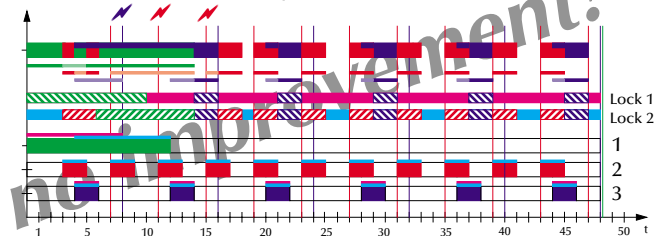
(independent task set)

### Interdependencies



$\Rightarrow$  Priority inversion

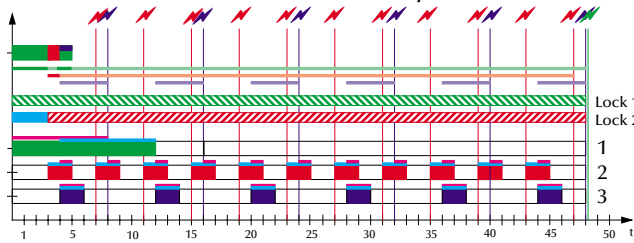
### Priority inheritance



(■ and ■ inherit priority of ■, when in lock and ■ is dispatched)



## One additional lock request



Deadlock

## Immediate ceiling priority protocol (POSIX, Ada, RT-Java)

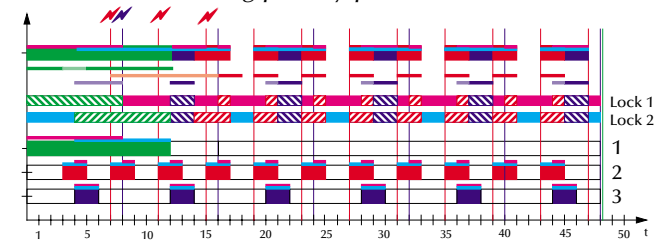
- Each task  $t_i$  has static default priority  $P_i$ .
- Each **resource** (lock, monitor)  $R_k$  has a **static ceiling priority**  $C_k$ , which is the maximum of priorities of the tasks  $t_i$  which employ this resource.

$$C_k = \max_i \{ \text{employ}(i, k) \cdot P_i \}$$

- Each task  $t_i$  has a dynamic priority  $P_i^D$ , which is the maximum of its own static priority and the ceiling priorities of any resource it has locked.

$$P_i^D = \max \{ P_i, \max_k \{ \text{locked}(i, k) \cdot C_k \} \}$$

## Immediate ceiling priority protocol (POSIX, Ada, RT-Java)



( , , and inherit the ceiling priority of or when entering the lock)

## Immediate ceiling priority protocol (POSIX, Ada, RT-Java)

- Tasks are dispatched only if **all** employed resources are available.
- Deadlocks are prevented
- Number of context switches is reduced

## Immediate ceiling priority protocol (POSIX, Ada, RT-Java)

$$\text{Maximal blocking time: } B_i = \max_{r=1}^R \{ \text{usage}(r, i) \cdot C(r) \}$$

- with  $R$  the number of critical sections
- $\text{usage}(r, i)$  a boolean (0/1) function indicating that  $r$  is used by at least one  $t_j$  with  $P_j < P_i$  and at least one  $t_k$  with  $P_k \geq P_i$
- $C(r)$  is the worst case computation time in critical section  $r$

a task can only be blocked once by any lower priority task!

## Scheduling

- Basic performance based scheduling**
  - $C_i$  is not known: first-come-first-served (FCFS), round robin (RR), and feedback-scheduling
  - $C_i$  is known: shortest job first (SJF), highest response ration first (HRRF), shortest remaining time first (SRTF)-scheduling
- Basic predictable scheduling**
  - Fixed Priority Scheduling (FPS) with Rate Monotonic (RMPO)
  - Earliest Deadline First (EDF)
- Real-world extensions**
  - Aperiodic, sporadic, soft real-time tasks
  - Synchronized talks (priority inheritance, priority ceiling protocols)

## Processes

- Processes and threads**
  - Architectures, definitions, process states
- Synchronization**
  - Shared memory based synchronization
  - Message based synchronization
- Deadlocks**
  - Detection, avoidance, and prevention (& recovery)
- Scheduling**
  - Basic performance based scheduling
  - Basic predictable scheduling
  - Aperiodic, sporadic, and synchronized tasks



## Memory

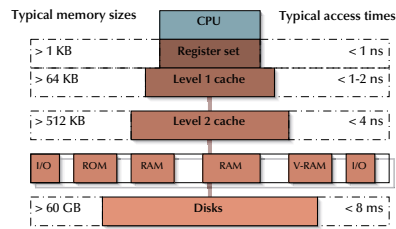
Uwe R. Zimmer – International University Bremen

- |   |  |
|---|--|
| [Silberschatz01] – Chapter 9,10<br>Abraham Silberschatz, Peter Bear Galvin,<br>Greg Gagne<br>Operating System Concepts<br>John Wiley & Sons, Inc., 2001 | [Stallings2001] – Chapter 7,8<br>William Stallings<br>Operating Systems<br>Prentice Hall, 2001 |
|---|--|

all references and some links are available on the course page

## Memory levels and fragments

## Basic memory hierarchy



## What is the challenge?

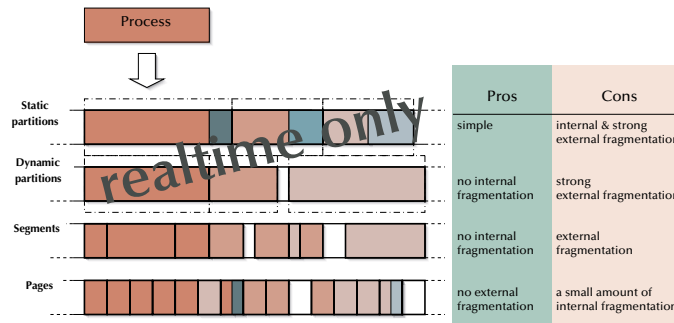
- Main memory is too small (regardless how large it is)
- The operating system needs to place (parts of) processes in and out of main memory during the life-time of the system.
- Swapping memory blocks between primary and secondary memory is an extremely slow operation.
- The operating system needs to supply highly efficient strategies to avoid system stalls or unacceptable delays.

## Goals / optimization criteria

- Supply address spaces, which are independent from the physically available address space.
- Supply multiple memory modes, e.g. allow processes to reside permanently in main memory
- Support for multiple address spaces
- Protection between address spaces
- Supply methods to share address spaces
- Support memory based I/O methods
- Allow for predictable behaviours of memory accesses
- Minimize any overhead for memory accesses and program executions

## Required support

- **Relocation**  
Assembler level addressing modes as well as compilers and linkers need to support relocatable programs and data structures.
- **Protection**  
Memory protection needs hardware support, since the operating system itself has no knowledge which memory cells will be addressed by a specific process next.
- **Sharing**  
The protection scheme needs to be flexible enough to allow for shared memory areas.
- **Control of secondary memory**  
Since swapping speeds between primary and secondary memory is a critical factor, the operating system needs to have close access to the secondary memory interface.
- **Project logical structures to memory modules** (optional)  
It might be useful to supply addressing modes, which allow the use of logical structures in the programs itself as the basis for memory structuring.



## The step from pagination/segmentation to Virtual addressing

## Segmentation / Paging:

- all memory references are logical addresses
- there is support to translate logical to physical addresses at run-time
- processes may be moved in memory and suspended to or loaded from secondary storage
- processes are divided in pages or segments (or both)
- pages or segments can be loaded in any order into primary memory (i.e. they need not to be dense or in sequence)

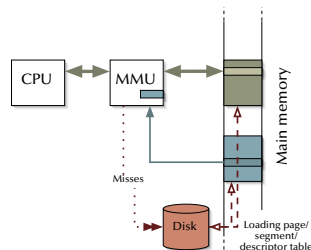
## Virtual addressing:

- **not all pages or segments need to be loaded in order to run a process**

## Translating virtual to physical addresses

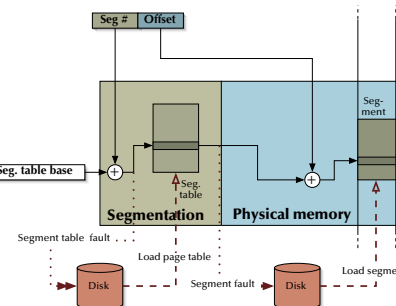
## MMU

1. **Translate virtual to physical addresses**  
without any delay in most cases.
2. **Provide memory protection**  
according to the attributes, which are attached to individual memory areas in form of page or segment descriptors.



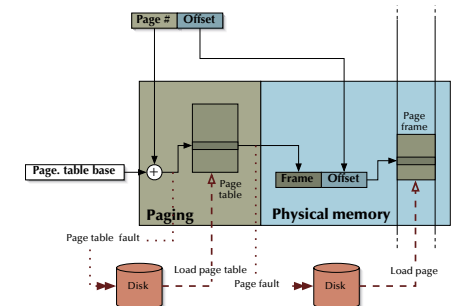
- Segment lengths is stored in segment table & needs to be evaluated by the memory protection unit.
- Segment base address and offset need to be added.
- Parts of segment tables as well as segments themselves can be suspended to secondary memory.

e.g. Intel x86

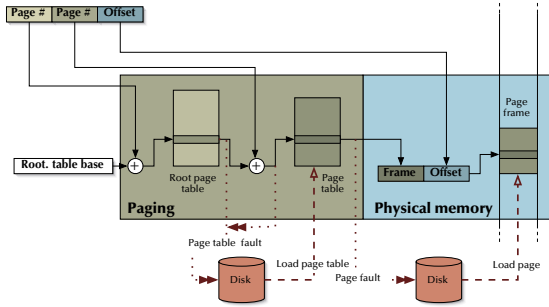


- Page frame address and address offset can be concatenated.
- Parts of page tables as well as pages themselves can be suspended to secondary memory (into 'frames').
- Page tables would be very large for modern processors (32-64bit addressing)

not implemented in this pure form.

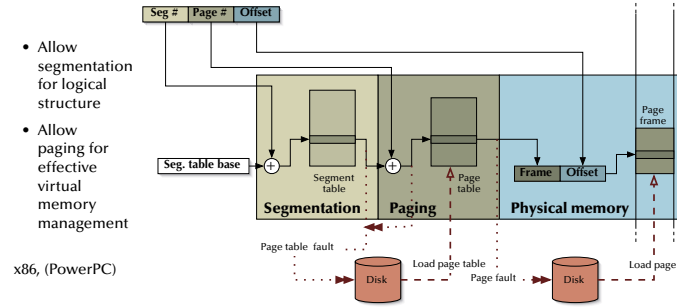


- Reducing page table sizes
- Up to four page levels (Sparc)
- More memory accesses required.



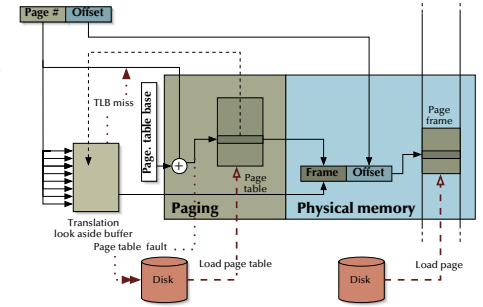
Sparc, PowerPC, Alpha, HP

- Allow segmentation for logical structure
- Allow paging for effective virtual memory management



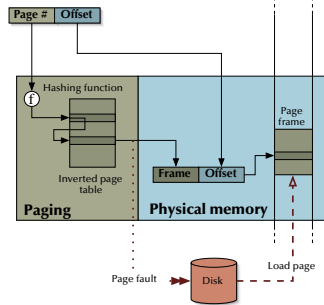
x86, (PowerPC)

- Accessing page tables for each access is ineffective.
- Introducing address translation caches: **Translation look aside buffers (tlb).**
- Access cache (tlb) - memory - disk (in this order) for address translation



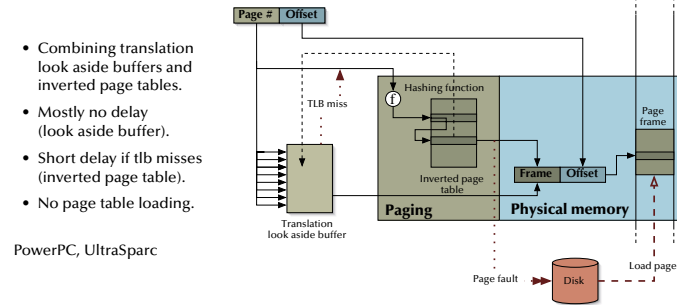
all modern MMUs

- Forward page tables grow with the size of the virtual address space.
- The number of loaded pages is bound by the physical memory.
- Keep only the loaded pages in the page table and resolve the virtual addresses via a hash table: **Inverted page tables (ipt)**
- IPTs are not suspended to secondary memory, but more than one access is required to translate the page number.



not implemented in this pure form.

- Combining translation look aside buffers and inverted page tables.
- Mostly no delay (look aside buffer).
- Short delay if tlb misses (inverted page table).
- No page table loading.



PowerPC, UltraSparc

### Some current MMU implementations

	Physical addresses	Virtual addresses	TLB size	Segments	Pages	Inverted/hashed tables
Pentium 4	36bit	32bit (per segment)	64	different types	4k, 4M (optional)	-
Itanium 2	50bit	64bit	4*32	-	4k ... 4G	-
Power PC 604	32bit	52bit	256	< 256MB, (optional)	4 k	yes
Power PC 970	42bit	64bit	1024	< 256MB, (optional)	4 k	yes
UltraSparc	36bit	64bit	64	-	8k ... 4M	yes
Alpha	41bit	64bit	256	-	8k ... 4M	-

### Design alternatives

- Employ virtual memory in the first place?
- Employ segmentation, pagination, or a combination of those?
- Which algorithms should be applied to answer:
  - when to load a page/segment?
  - where to place a page/segment?
  - which page/segment to suspend?
  - how many pages/segments to load for a specific process?
  - when to suspend a page/segment?
  - which processes to run/suspend?

fetching  
 placement  
 replacement  
 resident set management  
 cleaning  
 load control

### Fetching

#### • Demand paging:

- Fetch pages only if and exactly when requested by a reference to an address inside this page.
- may lead to a burst of page faults in some situations (e.g. starting a process).
- reduces the transfer between primary and secondary storage to a minimum.

#### • Prepaging:

- Predict which pages will also be required in the near future and pre-load them (together with the currently requested page).
- pages may be loaded, which will be never referenced
- multiple page loads can be more efficient if organized as a few transfers of a larger blocks

### Fetching

#### • Demand paging:

- Fetch pages only if and exactly when requested by a reference to an address inside this page.
- may lead to a burst of page faults in some situations (e.g. starting a process).
- reduces the transfer between primary and secondary storage to a minimum.

#### • Prepaging:

- Predict which pages will also be required in the near future and pre-load them (together with the currently requested page).
- pages may be loaded, which will be never referenced
- multiple page loads can be more efficient if organized as a few transfers of a larger blocks

### Placement

- Required for partition or pure segmentation systems
  - apply standard 'best-fit', 'first-fit', etc. strategies to minimize fragmentation
  - there is a trade-off between minimal fragmentation and minimal placement overhead
- Irrelevant for all paging or mixed segmentation/paging systems
  - external fragmentation is not an issue here

### Replacement

In order to load a new page, another page need to be suspended ⇨ which one?

- Optimal:**  
the page which will not be referenced for the longest period of *future* time
- Least Recently Used (LRU):**  
the page which has not be referenced for the longest period of *past* time
- First-In-First-Out (FIFO):**  
the page which resides in primary memory for the longest period of *past* time

### Replacement

The practical implementation aspect of replacement algorithms:

- Optimal:**  
⇨ can only be implemented, if all *future* memory references are known ⇨ ✗
- Least Recently Used (LRU):**  
⇨ can only be implemented, if all past access times/order are known ⇨ check hardware support
- First-In-First-Out (FIFO):**  
⇨ can be implemented without any hardware support ⇨ ✓

### Replacement

#### Full LRU implementations:

- Counter or time-of-access** field in the page table:  
Update this entry with each reference to this page
  - ⇨ need to be supplied by hardware (not implemented in any practical system)
- Page stack:**  
bring a reference to the page on top of a stack with each access to this page (and replace the pages at the bottom of the stack)
  - ⇨ need to be supplied by hardware (not implemented in any practical system)

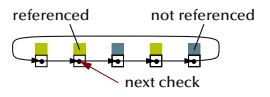
### Replacement

#### LRU-approximations:

- Reference-bit-shift-history algorithm:**  
Shift the reference bit of each page into a bit-field (0 1 1 1 1 1 1 1) in each page table entry at regular intervals (employing a timer-interrupt). Interpret the resulting bit-field as an integer and replace the page with the smallest value  
⇨ requires a reference-bit, which is updates by hardware, as well as a hardware timer (usually provided).

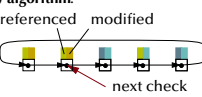
### Replacement

#### LRU-approximations:

- Second-chance (clock) algorithm:**  
  
Implement a circular list of all pages. Search the list for a not referenced page:  
WHILE page was referenced DO  
reset reference bit and proceed to next page  
END WHILE  
⇨ requires a reference-bit, which is updates by hardware (usually provided).

### Replacement

#### LRU-approximations:

- Enhanced second-chance (clock) algorithm:**  
  
Replace pages applying the priorities:
  - not referenced (first scan)
  - referenced-but-not-modified (second scan)
  - referenced-and-modified
 ⇨ requires a reference and a modified-bit, which is updates by hardware (usually provided).

### Replacement

#### Performances:

- Optimal:**  
obviously the best algorithm — impossible to implement
- Least Recently Used (LRU):**  
good approximation of the optimal algorithm — cannot be implemented in any current system
- Approximated Least Recently Used (LRU):**  
approximates the performance of LRU — can be implemented in most systems
- First-In-First-Out (FIFO):**  
performs worst — can be implemented in any system

### Resident set management

How many pages are assigned to a specific process:

- too many:
  - the number of resident processes is reduced
  - due to localities, there is no noticeable speed-up for the specific process
- too few:
  - significant increase in the page-fault rate
- ⇨ Challenge: find the essential working set of pages for each process at any given time

Strategies:

- Number of allocated pages per process can be
  - *fixed*
  - or *variable*
- Replacement can be either
  - *local* (inside each process' page set) – only possibility for fixed allocation scenes
  - *prioritized* (allow higher priority processes to expand their page sets)
  - or *global* (replace pages regardless of the processes which are using them)

- ☞ Challenge:  
*find the essential working page set for each process at any given time*
- Calculating the optimal working set, required full knowledge of the future process behaviour
- Many approximations are suggested (and implemented), mostly employing:
  - **Page Fault Frequencies (PFF)**  
or related statistical information on the past process behaviour
- Problems:
  - "the past does not always predict the future"  
i.e. multiple locality assumptions must hold

- **Demand cleaning:**
  - Clean pages only if and exactly when a free pages is required.
  - ☞ slows down process reaction times, since each page fault will result in a page cleaning.
  - ☞ reduces the total transfer between primary and secondary storage to a minimum.
- **Precleaning:**
  - Clean multiple pages according to replacement criteria introduced above before a page fault occurs.
  - ☞ too many pages might be cleaned, resulting in an increase of page faults
  - ☞ multiple page cleanings can be more efficient if organized as a few transfers of a larger blocks

- **Demand cleaning:**
  - Clean pages only if and exactly when a free pages is required.
  - ☞ slows down process reaction times, since each page fault will result in a page cleaning.
  - ☞ reduces the total transfer between primary and secondary storage to a minimum.
- **Precleaning:**
  - Clean multiple pages according to replacement criteria introduced above before a page fault occurs.
  - ☞ too many pages might be cleaned, resulting in an increase of page faults
  - ☞ multiple page cleanings can be more efficient if organized as a few transfers of a larger blocks

How many processes will be resident in primary memory?

- More processes in primary memory implies less pages per process
- Beyond a critical threshold of pages per process, the page fault rate rises significantly
  - ☞ **Thrashing** occurs
- The overall performance of the system is approaching nil, since most of the time is spent for page loads
- ☞ Reduce the number of resident processes immediately

Which process is to be suspended?

- Lowest priority process
- Process with the highest page fault frequency
- Process with the smallest current resident page set
- Process with the largest current resident page set
- Last activated process
- Process with the largest remaining execution time (see scheduling)

- Employ virtual memory in the first place?
- Employ segmentation, pagination, or a combination of those?
- Which algorithms should be applied to answer:
  - *when to load a page/segment?* ☞ **fetching**
  - *where to place a page/segment?* ☞ **placement**
  - *which page/segment to suspend?* ☞ **replacement**
  - *how many pages/segments to load for a specific process?* ☞ **resident set management**
  - *when to suspend a page/segment?* ☞ **cleaning**
  - *which processes to run/suspend?* ☞ **load control**

- Employ virtual memory in the first place?
- Employ segmentation, pagination, or a combination of those?
- Which algorithms should be applied to answer:
  - *when to load a page/segment?* ☞ **fetching**
  - *where to place a page/segment?* ☞ **placement**
  - *which page/segment to suspend?* ☞ **replacement**
  - *how many pages/segments to load for a specific process?* ☞ **resident set management**
  - *when to suspend a page/segment?* ☞ **cleaning**
  - *which processes to run/suspend?* ☞ **load control**

- **Requirements & hardware structures**
  - MMU features & requirements
- **Partitioning, segmentation, paging & virtual memory**
  - Simple segmentation
  - Simple paging, multi-level paging, combined segmentation & paging
  - Translation look aside buffers
  - Hashed tables, Inverted page tables
- **Virtual memory management algorithms**
  - Fetching & placement
  - Replacement
  - Resident set management
  - Cleaning
  - Load control