


1




# Systems, Networks & Concurrency 2020

Uwe R. Zimmer - The Australian National University

2

# Systems, Networks & Concurrency 2020




## Organization & Contents

Uwe R. Zimmer - The Australian National University

3

# Organization & Contents

*what is offered here?*



## Fundamentals & Overview

as well as perspectives, paths, methods, implementations, and open questions

of/into/for/about

## Concurrent & Distributed Systems

© 2020 Uwe R. Zimmer, The Australian National University page 3 of 758 ("Organization & Contents" up to page 179)

4

# Organization & Contents

*who could be interested in this?*

anybody who ...

- ... wants to work with **real-world** scale computer systems
- ... would like to learn how to **analyse and design operational and robust systems**
- ... would like to understand more about the existing trade-off between *theory, the real-world, traditions, and pragmatism* in computer science
- ... would like to understand why *concurrent systems* are an **essential basis** for most contemporary devices and systems

© 2020 Uwe R. Zimmer, The Australian National University page 4 of 758 ("Organization & Contents" up to page 179)

5

# Organization & Contents

*who are these people? - introductions*

Uwe R. Zimmer & Charles Martin

Abigail (Abi) Thomas, Aditya Chilukuri, Brent Schuetze, Calum Snowdon, Chinmay Garg, Felix Friedlander, Johannes (Johnny) Schmalz, Nicholas Philip Miehlsbradt, Tommy Liu, William (Will) Cashman & Yaya Lu



© 2020 Uwe R. Zimmer, The Australian National University page 5 of 758 ("Organization & Contents" up to page 179)

6

# Organization & Contents

*how will this all be done?*

- Lectures:**
  - 2x 1.5 hours lectures per week ... all the nice stuff Tuesday 12:00 & Friday 11:00 (all live on-line)
- Laboratories:**
  - 3 hours per week ... all the rough and action stuff
  - time slots: on our web-site
  - enrolment: <https://cs.anu.edu.au/streams/> (open since last Monday, more slots today)
- Resources:**
  - Introduced in the lectures and collected on the course page: <https://cs.anu.edu.au/courses/comp2310/> ... as well as schedules, slides, sources, links to forums, etc. pp. ... Keep an eye on this page!
- Assessment (for discussion):**
  - Exam at the end of the course (50%)
  - plus one hurdle lab in week 4 (5%)
  - plus two assignments (15% + 15%)
  - plus one mid-semester exam (15%)

© 2020 Uwe R. Zimmer, The Australian National University page 6 of 758 ("Organization & Contents" up to page 179)

7

# Organization & Contents

## Text book for the course

[Ben-Ari06]  
M. Ben-Ari  
*Principles of Concurrent and Distributed Programming*  
2006, second edition, Prentice-Hall, ISBN 0-13-711821-X

- Many algorithms and concepts for the course are in there – *but not all!*
- References for specific aspects of the course are provided during the course and are found on our web-site.

© 2020 Uwe R. Zimmer, The Australian National University page 7 of 758 ("Organization & Contents" up to page 179)

8

# Organization & Contents

## Topics

- Language refresher [3]
1. Concurrency [3]
2. Mutual exclusion [2]
3. Communication & Synchronization [4]
4. Non-determinism [2]
5. Data Parallelism [1]
6. Scheduling [2]
7. Safety and liveness [2]
8. Distributed systems [4]
9. Architectures [1]

© 2020 Uwe R. Zimmer, The Australian National University page 8 of 758 (chapter 1: "Organization & Contents" up to page 179)

9

# Organization & Contents

## Topics

1. Concurrency [3]
  - 1.1. Forms of concurrency [1]
    - Coupled dynamical systems
  - 1.2. Models and terminology [1]
    - Abstractions
    - Interleaving
    - Atomicity
    - Proofs in concurrent and distributed systems
  - 1.3. Processes & threads [1]
    - Basic definitions
    - Process states
    - Implementations
2. Mutual exclusion [2]
3. Communication & Synchronization [4]
4. Non-determinism [2]
5. Data Parallelism [1]
6. Scheduling [2]
7. Safety and liveness [2]
8. Distributed systems [4]
9. Architectures [1]

© 2020 Uwe R. Zimmer, The Australian National University page 9 of 758 ("Organization & Contents" up to page 179)

## Organization & Contents

### Topics

1. Concurrency [3]
2. Mutual exclusion [2]
3. Communication & Synchronization [4]
4. Non-determinism [2]
5. Data Parallelism [1]
6. Scheduling [2]
7. Safety and liveness [2]
8. Distributed systems [4]
9. Architectures [1]

2.1. by shared variables [1]

- Failure possibilities
- Dekker's algorithm

2.2. by test-and-set hardware support [0.5]

- Minimal hardware support

2.3. by semaphores [0.5]

- Dijkstra definition
- OS semaphores

© 2020 Dave R. Zineer, The Australian National University page 10 of 758 ("Organization & Contents" up to page 19)

## Organization & Contents

### Topics

1. Concurrency [3]
2. Mutual exclusion [2]
3. Communication & Synchronization [4]
4. Non-determinism [2]
5. Data Parallelism [1]
6. Scheduling [2]
7. Safety and liveness [2]
8. Distributed systems [4]
9. Architectures [1]

3.1. Shared memory synchronization [2]

- Semaphores
- Cond. variables
- Conditional critical regions
- Monitors
- Protected objects

3.2. Message passing [2]

- Asynchronous / synchronous
- Remote invocation / rendezvous
- Message structure
- Addressing

© 2020 Dave R. Zineer, The Australian National University page 11 of 758 ("Organization & Contents" up to page 19)

## Organization & Contents

### Topics

1. Concurrency [3]
2. Mutual exclusion [2]
3. Condition synchronization [4]
4. Non-determinism [2]
- 4.1. Correctness under non-determinism [1]
- 4.2. Select statements [1]
5. Data Parallelism [1]
6. Scheduling [2]
7. Safety and liveness [2]
8. Distributed systems [4]
9. Architectures [1]

4.1. Correctness under non-determinism [1]

- Forms of non-determinism
- Non-determinism in concurrent/distributed systems
- Is consistency/correctness plus non-determinism a contradiction?

4.2. Select statements [1]

- Forms of non-deterministic message reception

© 2020 Dave R. Zineer, The Australian National University page 12 of 758 ("Organization & Contents" up to page 19)

## Organization & Contents

### Topics

1. Concurrency [3]
2. Mutual exclusion [2]
3. Condition synchronization [4]
4. Non-determinism [2]
5. Data Parallelism [1]
- 5.1. Data-Parallelism
- 5.2. Examples
6. Scheduling [2]
7. Safety and liveness [2]
8. Distributed systems [4]
9. Architectures [1]

5.1. Data-Parallelism

- Vectorization
- Reduction
- General data-parallelism

5.2. Examples

- Image processing
- Cellular automata

© 2020 Dave R. Zineer, The Australian National University page 13 of 758 ("Organization & Contents" up to page 19)

## Organization & Contents

### Topics

1. Concurrency [3]
2. Mutual exclusion [2]
3. Condition synchronization [4]
4. Non-determinism [2]
5. Data Parallelism [1]
6. Scheduling [2]
- 6.1. Problem definition and design space [1]
- 6.2. Basic scheduling methods [1]
7. Safety and liveness [2]
8. Distributed systems [4]
9. Architectures [1]

6.1. Problem definition and design space [1]

- Which problems are addressed / solved by scheduling?

6.2. Basic scheduling methods [1]

- Assumptions for basic scheduling
- Basic methods

© 2020 Dave R. Zineer, The Australian National University page 14 of 758 ("Organization & Contents" up to page 19)

## Organization & Contents

### Topics

1. Concurrency [3]
2. Mutual exclusion [2]
3. Condition synchronization [4]
4. Non-determinism [2]
5. Data Parallelism [1]
6. Scheduling [2]
7. Safety and liveness [2]
- 7.1. Safety properties
- 7.2. Liveness, fairness
- 7.3. Deadlocks
- 7.4. Failure modes
- 7.5. Idempotent & atomic operations
8. Distributed systems [4]
9. Architectures [1]

7.1. Safety properties

- Essential time-independent safety properties

7.2. Liveness, fairness

- Forms of liveness
- Classification of fairness

7.3. Deadlocks

- Detection
- Avoidance
- Prevention (& recovery)

7.4. Failure modes

7.5. Idempotent & atomic operations

- Definitions

© 2020 Dave R. Zineer, The Australian National University page 15 of 758 ("Organization & Contents" up to page 19)

## Organization & Contents

### Topics

1. Concurrency [3]
2. Mutual exclusion [2]
3. Condition synchronization [4]
4. Non-determinism [2]
5. Data Parallelism [1]
6. Scheduling [2]
7. Safety and liveness [3]
8. Distributed systems [4]
- 8.1. Networks [1]
- 8.2. Global times [1]
- 8.3. Distributed states [1]
- 8.4. Distributed communication [1]
- 8.5. Distributed safety and liveness [1]
- 8.6. Forms of distribution/redundancy [1]
- 8.7. Transactions [2]
9. Architectures [1]

8.1. Networks [1]

- OSI model
- Network implementations

8.2. Global times [1]

- Synchronized clocks
- Logical clocks

8.3. Distributed states [1]

- Consistency
- Snapshots
- Termination

8.4. Distributed communication [1]

- Name spaces
- Multi-casts
- Elections
- Network identification

8.5. Distributed safety and liveness [1]

- Consistency
- Termination

8.6. Forms of distribution/redundancy [1]

- computation
- memory
- operations

8.7. Transactions [2]

- Dynamical groups

© 2020 Dave R. Zineer, The Australian National University page 16 of 758 ("Organization & Contents" up to page 19)

## Organization & Contents

### Topics

1. Concurrency [3]
2. Mutual exclusion [2]
3. Condition synchronization [4]
4. Non-determinism [2]
5. Data Parallelism [1]
6. Scheduling [2]
7. Safety and liveness [2]
8. Distributed systems [4]
9. Architectures [1]
- 9.1. Hardware architecture
- 9.2. Language architecture

9.1. Hardware architecture

- From switches to registers and address
- CPU architecture
- Hardware concurrency

9.2. Language architecture

- Chapel
- Occam
- Rust
- Ada
- C++


© 2020 Dave R. Zineer, The Australian National University page 17 of 758 ("Organization & Contents" up to page 19)

## Organization & Contents

### 24 Lectures

1. Concurrency [3]
- 1.1. Forms of concurrency [1]
- 1.2. Mutual exclusion [2]
2. Mutual exclusion [2]
- 2.1. by shared variables [1]
- 2.2. by test-and-set hardware support [0.5]
- 2.3. by semaphores [0.5]
3. Communication & Synchronization [4]
- 3.1. Shared memory synchronization [2]
- 3.2. Examples
4. Non-determinism [2]
- 4.1. Correctness under non-determinism [1]
- 4.2. Select statements [1]
5. Data Parallelism [1]
- 5.1. Data-Parallelism
- 5.2. Examples
6. Scheduling [2]
- 6.1. Problem definition and design space [1]
- 6.2. Basic scheduling methods [1]
7. Safety and liveness [2]
- 7.1. Safety properties
- 7.2. Liveness, fairness
- 7.3. Deadlocks
- 7.4. Failure modes
- 7.5. Idempotent & atomic operations
8. Distributed systems [4]
- 8.1. Networks [1]
- 8.2. Global times [1]
- 8.3. Distributed states [1]
- 8.4. Distributed communication [1]
- 8.5. Distributed safety and liveness [1]
- 8.6. Forms of distribution/redundancy [1]
- 8.7. Transactions [2]
9. Architectures [1]
- 9.1. Hardware architecture
- 9.2. Language architecture

© 2020 Dave R. Zineer, The Australian National University page 18 of 758 ("Organization & Contents" up to page 19)

 <h1>Organization &amp; Contents</h1> <h2>Laboratories &amp; Assignments</h2>			
Laboratories [11]	Communications Tasks [1]	Assignments [2]	Examinations [3]
<b>1. Structured Programming [2]</b> <ul style="list-style-type: none"> <li>• Program structures</li> <li>• Data structures</li> </ul>	<b>5. Communication Tasks [1]</b> <ul style="list-style-type: none"> <li>• Rendezvous</li> </ul>	<b>1. Concurrent programming [15%]</b> <ul style="list-style-type: none"> <li>• Programming task involving:               <ul style="list-style-type: none"> <li>• Mutual exclusion</li> <li>• Synchronization</li> <li>• Message passing</li> </ul> </li> </ul>	<b>1. Hurdle check [15%]</b> <ul style="list-style-type: none"> <li>• Week 4 lab exam</li> </ul>
<b>2. Tasks [1]</b> <ul style="list-style-type: none"> <li>• Generics</li> <li>• Abstract types</li> </ul>	<b>6. Distributing Server [1]</b> <ul style="list-style-type: none"> <li>• Entry families</li> <li>• Response facility</li> </ul>	<b>2. Concurrent programming in multi-core systems [15%]</b> <ul style="list-style-type: none"> <li>• Multi-core programming task involving:               <ul style="list-style-type: none"> <li>• Process communication</li> </ul> </li> </ul>	<b>2. Mid-semester check [15%]</b> <ul style="list-style-type: none"> <li>• Exam on 5/1/16</li> </ul>
<b>3. Protection [1]</b> <ul style="list-style-type: none"> <li>• Memory based synchronization</li> </ul>	<b>7. Implicit Concurrency [1]</b>	<b>3. Final exam [50%]</b> <ul style="list-style-type: none"> <li>• Examining the complete course</li> </ul>	<b>3. Final exam [50%]</b> <ul style="list-style-type: none"> <li>• Examining the complete course</li> </ul>
<b>4. Task Lifetimes [1]</b> <ul style="list-style-type: none"> <li>• Creation</li> <li>• Termination</li> </ul>	<b>8. Synchronized Data [1]</b>	<b>9. Distribution [1]</b> <ul style="list-style-type: none"> <li>• Multi-core process creation, termination</li> <li>• Multi-core process communication</li> </ul>	<b>Marking</b> <p>The final mark is based on the assignments (30%) plus the examinations (65%) plus the lab mark (5%).</p>
	<b>10. Pipelines [1]</b>		

## Systems, Networks & Concurrency 2020



### Language refresher / introduction course

Uwe R. Zimmer - The Australian National University

## Language refresher / introduction course

### References for this chapter

[Ada 2012 Language Reference Manual]  
see course pages or <http://www.ada-auth.org/standards/ada12.html>

[Chapel 1.13 Language Specification Version 0.981]  
see course pages or  
[http://chapel.cray.com/docs/latest/\\_downloads/chapelLanguageSpec.pdf](http://chapel.cray.com/docs/latest/_downloads/chapelLanguageSpec.pdf)  
released on 7. April 2016

## Language refresher / introduction course

### Languages explicitly supporting concurrency: e.g. Ada

Ada is an **ISO standardized** (ISO/IEC 8652:201x(E)) 'general purpose' language with focus on "program reliability and maintenance, programming as a human activity, and efficiency".

It provides **core language primitives** for:

- Strong typing, contracts, separate compilation (specification and implementation), abstract data types, generics, object-orientation.
- Concurrency, message passing, synchronization, monitors, rpcs, timeouts, scheduling, priority ceiling locks, hardware mappings, fully typed network communication.
- Strong run-time environments (incl. stand-alone execution).

... as well as **standardized language annexes** for:

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

## Language refresher / introduction course

### Ada

### A crash course

... refreshing for some, X<sup>th</sup>-language introduction for others:

- **Specification and implementation** (body) parts, basic types
- **Exceptions**
- Information hiding in specifications ('private')
- **Contracts**
- **Generic programming** (polymorphism)
- **Tasking**
- Monitors and synchronisation ('protected', 'entries', 'selects', 'accepts')
- **Abstract types and dispatching**

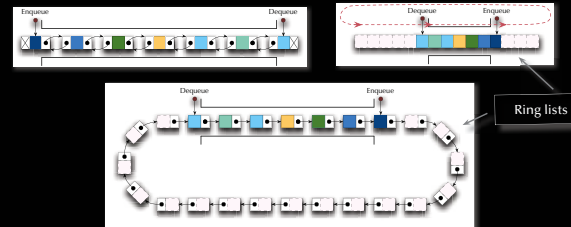
Not mentioned here: general object orientation, dynamic memory management, foreign language interfaces, marshalling, basics of imperative programming, ...

## Language refresher / introduction course

### Data structure example

### Queues

Forms of implementation:

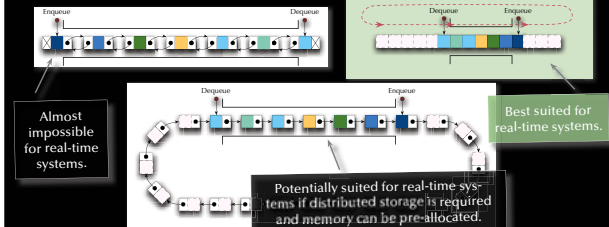


## Language refresher / introduction course

### Data structure example

### Queues

Forms of implementation:



## Language refresher / introduction course

### Ada

### Basics

... introducing:

- **Specification and implementation** (body) parts
- **Constants**
- Some **basic types** (integer specifics)
- Some **type attributes**
- **Parameter specification**

## Language refresher / introduction course

### 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) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
  procedure Enqueue (Item: Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;
```

## Language refresher / introduction course

### 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) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
  procedure Enqueue (Item: Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;
```

Specifications define an interface to provided types and operations. Syntactically enclosed in a package block.

## Language refresher / introduction course

### 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) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
  procedure Enqueue (Item: Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;

```

Variables should be initialized.  
Constants must be initialized.

Assignments are denoted by the "=" symbol,  
... leaving the "=" symbol for comparisons.

© 2020 Dave R. Zimmerman, The Australian National University page 29 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### 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) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
  procedure Enqueue (Item: Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;

```

Default initializations can be selected to be:  
as is (random memory content),  
initialized to invalids, e.g. 999  
or valid, predictable values, e.g. 1\_000

© 2020 Dave R. Zimmerman, The Australian National University page 30 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### 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) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
  procedure Enqueue (Item: Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;

```

Numerical types can be specified by:  
range, modulo,  
number of digits (≠ floating point)  
or delta increment (≠ fixed point).

Always be as specific as the language allows.  
... and don't repeat yourself!

© 2020 Dave R. Zimmerman, The Australian National University page 31 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### 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) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
  procedure Enqueue (Item: Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;

```

All types come with a long list of built-in attributes.  
Let the compiler fill in what you already (implicitly) specified!

© 2020 Dave R. Zimmerman, The Australian National University page 32 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### 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) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
  procedure Enqueue (Item: Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;

```

Parameters can be passed as 'in' (default), 'out' or 'in out'.

© 2020 Dave R. Zimmerman, The Australian National University page 33 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### 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) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
  procedure Enqueue (Item: Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;

```

All specifications are used in  
Code optimizations (optional),  
Compile time checks (mandatory)  
Run-time checks (suppressible).

© 2020 Dave R. Zimmerman, The Australian National University page 34 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### 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) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
  procedure Enqueue (Item: Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
end Queue_Pack_Simple;

```

... anything on this slide still not perfectly clear?

© 2020 Dave R. Zimmerman, The Australian National University page 35 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### A simple queue implementation

```

package body Queue_Pack_Simple is
  procedure Enqueue (Item: Element; Queue: in out Queue_Type) is
  begin
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free + 1;
    Queue.Is_Empty := False;
  end Enqueue;
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
    Queue.Is_Empty := Queue.Top = Queue.Free;
  end Dequeue;
  function Is_Empty (Queue : Queue_Type) return Boolean is
  Queue.Is_Empty;
  function Is_Full (Queue : Queue_Type) return Boolean is
  not Queue.Is_Empty and then Queue.Top = Queue.Free;
end Queue_Pack_Simple;

```

© 2020 Dave R. Zimmerman, The Australian National University page 36 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### A simple queue implementation

```

package body Queue_Pack_Simple is
  procedure Enqueue (Item: Element; Queue: in out Queue_Type) is
  begin
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free + 1;
    Queue.Is_Empty := False;
  end Enqueue;
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
    Queue.Is_Empty := Queue.Top = Queue.Free;
  end Dequeue;
  function Is_Empty (Queue : Queue_Type) return Boolean is
  Queue.Is_Empty;
  function Is_Full (Queue : Queue_Type) return Boolean is
  not Queue.Is_Empty and then Queue.Top = Queue.Free;
end Queue_Pack_Simple;

```

Implementations are defined in a separate file.  
Syntactically enclosed in a package body block.

© 2020 Dave R. Zimmerman, The Australian National University page 37 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### A simple queue implementation

```

package body Queue_Pack_Simple is
  procedure Enqueue (Item: Element; Queue: in out Queue_Type) is
  begin
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free + 1;
    Queue.Is_Empty := False;
  end Enqueue;
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
    Queue.Is_Empty := Queue.Top = Queue.Free;
  end Dequeue;
  function Is_Empty (Queue : Queue_Type) return Boolean is
    (Queue.Is_Empty);
  function Is_Full (Queue : Queue_Type) return Boolean is
    (not Queue.Is_Empty and then Queue.Top = Queue.Free);
end Queue_Pack_Simple;

```

Modulo type, hence no index checks required.

© 2010 Dave R. Zimmerman, The Australian National University page 38 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### A simple queue implementation

```

package body Queue_Pack_Simple is
  procedure Enqueue (Item: Element; Queue: in out Queue_Type) is
  begin
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free + 1;
    Queue.Is_Empty := False;
  end Enqueue;
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
    Queue.Is_Empty := Queue.Top = Queue.Free;
  end Dequeue;
  function Is_Empty (Queue : Queue_Type) return Boolean is
    (Queue.Is_Empty);
  function Is_Full (Queue : Queue_Type) return Boolean is
    (not Queue.Is_Empty and then Queue.Top = Queue.Free);
end Queue_Pack_Simple;

```

... anything on this slide still not perfectly clear?

© 2010 Dave R. Zimmerman, The Australian National University page 41 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### 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);
end Queue_Test_Simple;

```

A top level procedure is read as the code which needs to be executed.

© 2010 Dave R. Zimmerman, The Australian National University page 44 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### A simple queue implementation

```

package body Queue_Pack_Simple is
  procedure Enqueue (Item: Element; Queue: in out Queue_Type) is
  begin
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free + 1;
    Queue.Is_Empty := False;
  end Enqueue;
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
    Queue.Is_Empty := Queue.Top = Queue.Free;
  end Dequeue;
  function Is_Empty (Queue : Queue_Type) return Boolean is
    (Queue.Is_Empty);
  function Is_Full (Queue : Queue_Type) return Boolean is
    (not Queue.Is_Empty and then Queue.Top = Queue.Free);
end Queue_Pack_Simple;

```

Boolean expressions

© 2010 Dave R. Zimmerman, The Australian National University page 39 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### 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);
end Queue_Test_Simple;

```

© 2010 Dave R. Zimmerman, The Australian National University page 42 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### 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);
end Queue_Test_Simple;

```

Variables are declared Algol style: "Item is of type Element".

© 2010 Dave R. Zimmerman, The Australian National University page 45 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### A simple queue implementation

```

package body Queue_Pack_Simple is
  procedure Enqueue (Item: Element; Queue: in out Queue_Type) is
  begin
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free + 1;
    Queue.Is_Empty := False;
  end Enqueue;
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
    Queue.Is_Empty := Queue.Top = Queue.Free;
  end Dequeue;
  function Is_Empty (Queue : Queue_Type) return Boolean is
    (Queue.Is_Empty);
  function Is_Full (Queue : Queue_Type) return Boolean is
    (not Queue.Is_Empty and then Queue.Top = Queue.Free);
end Queue_Pack_Simple;

```

Side-effect free, single expression functions can be expressed without begin-end blocks.

© 2010 Dave R. Zimmerman, The Australian National University page 40 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### 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);
end Queue_Test_Simple;

```

Importing items from other packages is done with with-clauses. use-clauses allow to use names with qualifying them with the package name.

© 2010 Dave R. Zimmerman, The Australian National University page 43 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### 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);
end Queue_Test_Simple;

```

Will produce a result according to the chosen initialization: Raises an "invalid data" exception if initialized to invalids.

... hmm, ok ... so this was rubbish ...

© 2010 Dave R. Zimmerman, The Australian National University page 46 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### 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);
end Queue_Test_Simple;
```

... anything on this slide still not perfectly clear?

© 2020 Dave R. Zimmerman, The Australian National University page 47 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### A queue specification with proper exceptions

```
package Queue_Pack_Exceptions is
  QueueSize : constant Positive := 10;
  type Element is (Up, Down, Spin, Turn);
  type Marker is mod QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
  procedure Enqueue (Item : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean is (Queue.Is_Empty);
  function Is_Full (Queue : Queue_Type) return Boolean is
    (not Queue.Is_Empty and then Queue.Top = Queue.Free);
  Queue_overflow, Queue_underflow : exception;
end Queue_Pack_Exceptions;
```

Enumeration types are first-class types and can be used e.g. as array indices. The representation values can be controlled and do not need to be continuous (e.g. for purposes like interfacing with hardware).

Nothing else changes in the specifications.

Exceptions need to be declared.

© 2020 Dave R. Zimmerman, The Australian National University page 50 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### A queue implementation with proper exceptions

```
package body Queue_Pack_Exceptions is
  procedure Enqueue (Item : Element; Queue : in out Queue_Type) is
  begin
    if Is_Full (Queue) then
      raise Queue_overflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Marker'Succ (Queue.Free);
    Queue.Is_Empty := False;
  end Enqueue;

  procedure Dequeue (Item : out Element; Queue : in out Queue_Type) is
  begin
    if Is_Empty (Queue) then
      raise Queue_underflow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Marker'Succ (Queue.Top);
    Queue.Is_Empty := Queue.Top = Queue.Free;
  end Dequeue;
end Queue_Pack_Exceptions;
```

Raised exceptions break the control flow and "propagate" to the closest "exception handler" in the call-chain.

All Types come with a long list of built-in operators. Syntactically expressed as attributes.

Type attributes often make code more generic: 'Succ works for instance on enumeration types as well ... "+ 1" does not.

© 2020 Dave R. Zimmerman, The Australian National University page 53 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### Ada Exceptions

... introducing:

- Exception handling
- Enumeration types
- Type attributed operators

© 2020 Dave R. Zimmerman, The Australian National University page 48 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### A queue specification with proper exceptions

```
package Queue_Pack_Exceptions is
  QueueSize : constant Positive := 10;
  type Element is (Up, Down, Spin, Turn);
  type Marker is mod QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
  procedure Enqueue (Item : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean is (Queue.Is_Empty);
  function Is_Full (Queue : Queue_Type) return Boolean is
    (not Queue.Is_Empty and then Queue.Top = Queue.Free);
  Queue_overflow, Queue_underflow : exception;
end Queue_Pack_Exceptions;
```

Nothing else changes in the specifications.

Exceptions need to be declared.

© 2020 Dave R. Zimmerman, The Australian National University page 51 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### A queue implementation with proper exceptions

```
package body Queue_Pack_Exceptions is
  procedure Enqueue (Item : Element; Queue : in out Queue_Type) is
  begin
    if Is_Full (Queue) then
      raise Queue_overflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Marker'Succ (Queue.Free);
    Queue.Is_Empty := False;
  end Enqueue;

  procedure Dequeue (Item : out Element; Queue : in out Queue_Type) is
  begin
    if Is_Empty (Queue) then
      raise Queue_underflow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Marker'Succ (Queue.Top);
    Queue.Is_Empty := Queue.Top = Queue.Free;
  end Dequeue;
end Queue_Pack_Exceptions;
```

Raised exceptions break the control flow and "propagate" to the closest "exception handler" in the call-chain.

All Types come with a long list of built-in operators. Syntactically expressed as attributes.

Type attributes often make code more generic: 'Succ works for instance on enumeration types as well ... "+ 1" does not.

© 2020 Dave R. Zimmerman, The Australian National University page 54 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### A queue specification with proper exceptions

```
package Queue_Pack_Exceptions is
  QueueSize : constant Positive := 10;
  type Element is (Up, Down, Spin, Turn);
  type Marker is mod QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
  procedure Enqueue (Item : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean is (Queue.Is_Empty);
  function Is_Full (Queue : Queue_Type) return Boolean is
    (not Queue.Is_Empty and then Queue.Top = Queue.Free);
  Queue_overflow, Queue_underflow : exception;
end Queue_Pack_Exceptions;
```

© 2020 Dave R. Zimmerman, The Australian National University page 49 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### A queue specification with proper exceptions

```
package Queue_Pack_Exceptions is
  QueueSize : constant Positive := 10;
  type Element is (Up, Down, Spin, Turn);
  type Marker is mod QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
  procedure Enqueue (Item : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean is (Queue.Is_Empty);
  function Is_Full (Queue : Queue_Type) return Boolean is
    (not Queue.Is_Empty and then Queue.Top = Queue.Free);
  Queue_overflow, Queue_underflow : exception;
end Queue_Pack_Exceptions;
```

Nothing else changes in the specifications.

Exceptions need to be declared.

© 2020 Dave R. Zimmerman, The Australian National University page 52 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### A queue implementation with proper exceptions

```
package body Queue_Pack_Exceptions is
  procedure Enqueue (Item : Element; Queue : in out Queue_Type) is
  begin
    if Is_Full (Queue) then
      raise Queue_overflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Marker'Succ (Queue.Free);
    Queue.Is_Empty := False;
  end Enqueue;

  procedure Dequeue (Item : out Element; Queue : in out Queue_Type) is
  begin
    if Is_Empty (Queue) then
      raise Queue_underflow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Marker'Succ (Queue.Top);
    Queue.Is_Empty := Queue.Top = Queue.Free;
  end Dequeue;
end Queue_Pack_Exceptions;
```

Raised exceptions break the control flow and "propagate" to the closest "exception handler" in the call-chain.

All Types come with a long list of built-in operators. Syntactically expressed as attributes.

Type attributes often make code more generic: 'Succ works for instance on enumeration types as well ... "+ 1" does not.

© 2020 Dave R. Zimmerman, The Australian National University page 55 of 758 ("Language refresher / introduction course" up to page 160)

### A queue **implementation** with proper exceptions

```
package body Queue_Pack_Exceptions is
  procedure Enqueue (Item : Element; Queue : in out Queue_Type) is
  begin
    if Is_Full (Queue) then
      raise Queue_overflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Marker'Succ (Queue.Free);
    Queue.Is_Empty := False;
  end Enqueue;

  procedure Dequeue (Item : out Element; Queue : in out Queue_Type) is
  begin
    if Is_Empty (Queue) then
      raise Queue_underflow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Marker'Succ (Queue.Top);
    Queue.Is_Empty := Queue.Top = Queue.Free;
  end Dequeue;
end Queue_Pack_Exceptions;
```

... anything on this slide still not perfectly clear?

### 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

exception
  when Queue_underflow => Put ("Queue underflow");
  when Queue_overflow => Put ("Queue overflow");
end Queue_Test_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

exception
  when Queue_underflow => Put ("Queue underflow");
  when Queue_overflow => Put ("Queue overflow");
end Queue_Test_Exceptions;
```

An **exception handler** has a choice to **handle**, **pass**, or **re-raise** the same or a different exception.

**Raised exceptions** break the control flow and "propagate" to the closest "exception handler" in the call-chain.

Control flow is continued after the **exception handler** in case of a handled exception.

### 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

exception
  when Queue_underflow => Put ("Queue underflow");
  when Queue_overflow => Put ("Queue overflow");
end Queue_Test_Exceptions;
```

... anything on this slide still not perfectly clear?

### A queue **specification** with proper exceptions

```
package Queue_Pack_Exceptions is
  QueueSize : constant Positive := 10;

  type Element is (Up, Down, Spin, Turn);
  type Marker is mod QueueSize;
  type List is array (Marker) of Element;

  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;

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

  function Is_Empty (Queue : Queue_Type) return Boolean is (Queue.Is_Empty);
  function Is_Full (Queue : Queue_Type) return Boolean is
    (not Queue.Is_Empty and then Queue.Top = Queue.Free);
  Queue_overflow, Queue_underflow : exception;
end Queue_Pack_Exceptions;
```

This package provides access to "internal" structures which can lead to inconsistent access.



## Language refresher / introduction course

### Ada Information hiding

... introducing:

- **Private declarations**  $\Rightarrow$  needed to compile specifications, yet not accessible for a user of the package.
- **Private types**  $\Rightarrow$  assignments and comparisons are allowed
- **Limited private types**  $\Rightarrow$  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 : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
  Queueoverflow, Queueunderflow : exception;

private
  type Marker is mod QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Private;
```

### 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 : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
  Queueoverflow, Queueunderflow : exception;

private
  type Marker is mod QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Private;
```

private splits the specification into a **public** and a **private** section.

The private section is only here so that the specifications can be separately compiled.

### 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 : Element; Queue : in out Queue_Type);
  procedure Dequeue (Item : out Element; Queue : in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
  Queueoverflow, Queueunderflow : exception;

private
  type Marker is mod QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Private;
```

Queue\_Type can now be used outside this package **without any way to access its internal structure.**

limited disables assignments and comparisons for this type. A user of this package would now e.g. not be able to make a copy of a Queue\_Type value.



### A queue *specification* with proper information hiding

```
package Queue_Pack_Private is
  Queue_Size : constant Integer := 10;
  type Element is new Positive range 1..1000;
  type Queue_Type is limited private;
  procedure Enqueue (Item: Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
  Queueoverflow, Queueunderflow : exception;
private
  type Marker is mod Queue_Size;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Private;
```

Queue\_Type can now be used outside this package **without any way to access its internal structure.**

Alternatively '=' and '<=' operations can be replaced with type-specific versions (overloaded) or default operations can be allowed.

... anything on this slide still not perfectly clear?

### A queue *specification* with proper information hiding

```
package Queue_Pack_Private is
  Queue_Size : constant Integer := 10;
  type Element is new Positive range 1..1000;
  type Queue_Type is limited private;
  procedure Enqueue (Item: Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
  Queueoverflow, Queueunderflow : exception;
private
  type Marker is mod Queue_Size;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Private;
```

... anything on this slide still not perfectly clear?

### A queue *implementation* with proper information hiding

```
package body Queue_Pack_Private is
  procedure Enqueue (Item: Element; Queue: in out Queue_Type) is
  begin
    if Is_Full (Queue) then
      raise Queueoverflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Marker'Pred (Queue.Free);
    Queue.Is_Empty := False;
  end Enqueue;
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    if Is_Empty (Queue) then
      raise Queueunderflow; end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Marker'Pred (Queue.Top);
    Queue.Is_Empty := Queue.Top = Queue.Free;
  end Dequeue;
  function Is_Empty (Queue : Queue_Type) return Boolean is (Queue.Is_Empty);
  function Is_Full (Queue : Queue_Type) return Boolean is
    (not Queue.Is_Empty and then Queue.Top = Queue.Free);
end Queue_Pack_Private;
```

© 2010 Drew R. Zimmerman, The Australian National University page 67 of 758 ("Language reference / introduction course" up to page 116)

### A queue *implementation* with proper information hiding

```
package body Queue_Pack_Private is
  procedure Enqueue (Item: Element; Queue: in out Queue_Type) is
  begin
    if Is_Full (Queue) then
      raise Queueoverflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Marker'Pred (Queue.Free);
    Queue.Is_Empty := False;
  end Enqueue;
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    if Is_Empty (Queue) then
      raise Queueunderflow; end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Marker'Pred (Queue.Top);
    Queue.Is_Empty := Queue.Top = Queue.Free;
  end Dequeue;
  function Is_Empty (Queue : Queue_Type) return Boolean is (Queue.Is_Empty);
  function Is_Full (Queue : Queue_Type) return Boolean is
    (not Queue.Is_Empty and then Queue.Top = Queue.Free);
end Queue_Pack_Private;
```

... besides the implementation of the two functions which has been moved to the implementation section.

... anything on this slide still not perfectly clear?

### A queue *implementation* with proper information hiding

```
package body Queue_Pack_Private is
  procedure Enqueue (Item: Element; Queue: in out Queue_Type) is
  begin
    if Is_Full (Queue) then
      raise Queueoverflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Marker'Pred (Queue.Free);
    Queue.Is_Empty := False;
  end Enqueue;
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    if Is_Empty (Queue) then
      raise Queueunderflow; end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Marker'Pred (Queue.Top);
    Queue.Is_Empty := Queue.Top = Queue.Free;
  end Dequeue;
  function Is_Empty (Queue : Queue_Type) return Boolean is (Queue.Is_Empty);
  function Is_Full (Queue : Queue_Type) return Boolean is
    (not Queue.Is_Empty and then Queue.Top = Queue.Free);
end Queue_Pack_Private;
```

... anything on this slide still not perfectly clear?

### 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); -- would produce a "Queue underflow"
exception
  when Queueunderflow => Put ("Queue underflow");
  when Queueoverflow => Put ("Queue overflow");
end Queue_Test_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); -- would produce a "Queue underflow"
exception
  when Queueunderflow => Put ("Queue underflow");
  when Queueoverflow => Put ("Queue overflow");
end Queue_Test_Private;
```

Illegal operation on a limited type.

### 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); -- would produce a "Queue underflow"
exception
  when Queueunderflow => Put ("Queue underflow");
  when Queueoverflow => Put ("Queue overflow");
end Queue_Test_Private;
```

Parameters can be named or passed by order of definition. (Named parameters do not need to follow the definition order.)

### 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); -- would produce a "Queue underflow"
exception
  when Queueunderflow => Put ("Queue underflow");
  when Queueoverflow => Put ("Queue overflow");
end Queue_Test_Private;
```

... anything on this slide still not perfectly clear?



## Language refresher / introduction course

Ada

### Contracts

... introducing:

- Pre- and Post-Conditions on methods
- Invariants on types
- For all, For any predicates

### A contracting queue specification

```
package Queue_Pack_Contract is
  Queue_Size : constant Positive := 10;
  type Element is new Positive range 1 .. 1000;
  type Queue_Type is private;

  procedure Enqueue (Item : Element; Q : in out Queue_Type) with
    Pre => not Is_Full (Q),
    Post => not Is_Empty (Q) and then Length (Q) = Length (Q'Old) + 1
      and then Lookahead (Q, Length (Q)) = Item
      and then (for all ix in 1 .. Length (Q'Old)
        => Lookahead (Q, ix) = Lookahead (Q'Old, ix));

  procedure Dequeue (Item : out Element; Q : in out Queue_Type) with
    Pre => not Is_Empty (Q),
    Post => not Is_Full (Q) and then Length (Q) = Length (Q'Old) - 1
      and then (for all ix in 1 .. Length (Q)
        => Lookahead (Q, ix) = Lookahead (Q'Old, ix + 1));

  function Is_Empty (Q : Queue_Type) return Boolean;
  function Is_Full (Q : Queue_Type) return Boolean;
  function Length (Q : Queue_Type) return Natural;
  function Lookahead (Q : Queue_Type; Depth : Positive) return Element;
end Queue_Pack_Contract;
```

### A contracting queue specification

```
package Queue_Pack_Contract is
  Queue_Size : constant Positive := 10;
  type Element is new Positive range 1 .. 1000;
  type Queue_Type is private;

  procedure Enqueue (Item : Element; Q : in out Queue_Type) with
    Pre => not Is_Full (Q),
    Post => not Is_Empty (Q) and then Length (Q) = Length (Q'Old) + 1
      and then Lookahead (Q, Length (Q)) = Item
      and then (for all ix in 1 .. Length (Q'Old)
        => Lookahead (Q, ix) = Lookahead (Q'Old, ix));

  procedure Dequeue (Item : out Element; Q : in out Queue_Type) with
    Pre => not Is_Empty (Q),
    Post => not Is_Full (Q) and then Length (Q) = Length (Q'Old) - 1
      and then (for all ix in 1 .. Length (Q)
        => Lookahead (Q, ix) = Lookahead (Q'Old, ix + 1));

  function Is_Empty (Q : Queue_Type) return Boolean;
  function Is_Full (Q : Queue_Type) return Boolean;
  function Length (Q : Queue_Type) return Natural;
  function Lookahead (Q : Queue_Type; Depth : Positive) return Element;
end Queue_Pack_Contract;
```

Pre- and Post-predicates are checked before and after each execution resp.

Original (Pre) values can still be referred to.

forall and exists quantifiers are expressed as "for all" and "for some" expressions resp.

### A contracting queue specification

```
package Queue_Pack_Contract is
  Queue_Size : constant Positive := 10;
  type Element is new Positive range 1 .. 1000;
  type Queue_Type is private;

  procedure Enqueue (Item : Element; Q : in out Queue_Type) with
    Pre => not Is_Full (Q),
    Post => not Is_Empty (Q) and then Length (Q) = Length (Q'Old) + 1
      and then Lookahead (Q, Length (Q)) = Item
      and then (for all ix in 1 .. Length (Q'Old)
        => Lookahead (Q, ix) = Lookahead (Q'Old, ix));

  procedure Dequeue (Item : out Element; Q : in out Queue_Type) with
    Pre => not Is_Empty (Q),
    Post => not Is_Full (Q) and then Length (Q) = Length (Q'Old) - 1
      and then (for all ix in 1 .. Length (Q)
        => Lookahead (Q, ix) = Lookahead (Q'Old, ix + 1));

  function Is_Empty (Q : Queue_Type) return Boolean;
  function Is_Full (Q : Queue_Type) return Boolean;
  function Length (Q : Queue_Type) return Natural;
  function Lookahead (Q : Queue_Type; Depth : Positive) return Element;
end Queue_Pack_Contract;
```

... anything on this slide still not perfectly clear?

### A contracting queue specification (cont.)

```
private
  type Marker is mod Queue_Size;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List; -- will be initialized to invalids
  end record with Type_Invariant
    => (not Queue_Type.Is_Empty or else Queue_Type.Top = Queue_Type.Free)
      and then (for all ix in 1 .. Length (Queue_Type)
        => Lookahead (Queue_Type, ix)'Valid);

  function Is_Empty (Q : Queue_Type) return Boolean is (Q.Is_Empty);
  function Is_Full (Q : Queue_Type) return Boolean is
    (not Q.Is_Empty and then Q.Top = Q.Free);
  function Length (Q : Queue_Type) return Natural is
    (if Is_Full (Q) then Queue_Size else Natural (Q.Free - Q.Top));
  function Lookahead (Q : Queue_Type; Depth : Positive) return Element is
    (Q.Elements (Q.Top + Marker (Depth - 1)));
end Queue_Pack_Contract;
```

### A contracting queue specification (cont.)

```
private
  type Marker is mod Queue_Size;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List; -- will be initialized to invalids
  end record with Type_Invariant
    => (not Queue_Type.Is_Empty or else Queue_Type.Top = Queue_Type.Free)
      and then (for all ix in 1 .. Length (Queue_Type)
        => Lookahead (Queue_Type, ix)'Valid);

  function Is_Empty (Q : Queue_Type) return Boolean is (Q.Is_Empty);
  function Is_Full (Q : Queue_Type) return Boolean is
    (not Q.Is_Empty and then Q.Top = Q.Free);
  function Length (Q : Queue_Type) return Natural is
    (if Is_Full (Q) then Queue_Size else Natural (Q.Free - Q.Top));
  function Lookahead (Q : Queue_Type; Depth : Positive) return Element is
    (Q.Elements (Q.Top + Marker (Depth - 1)));
end Queue_Pack_Contract;
```

Type-Invariants are checked on return from any operation defined in the public part.

### A contracting queue specification (cont.)

```
private
  type Marker is mod Queue_Size;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List; -- will be initialized to invalids
  end record with Type_Invariant
    => (not Queue_Type.Is_Empty or else Queue_Type.Top = Queue_Type.Free)
      and then (for all ix in 1 .. Length (Queue_Type)
        => Lookahead (Queue_Type, ix)'Valid);

  function Is_Empty (Q : Queue_Type) return Boolean is (Q.Is_Empty);
  function Is_Full (Q : Queue_Type) return Boolean is
    (not Q.Is_Empty and then Q.Top = Q.Free);
  function Length (Q : Queue_Type) return Natural is
    (if Is_Full (Q) then Queue_Size else Natural (Q.Free - Q.Top));
  function Lookahead (Q : Queue_Type; Depth : Positive) return Element is
    (Q.Elements (Q.Top + Marker (Depth - 1)));
end Queue_Pack_Contract;
```

... anything on this slide still not perfectly clear?

### A contracting queue implementation

```
package body Queue_Pack_Contract is
  procedure Enqueue (Item : Element; Q : in out Queue_Type) is
  begin
    Q.Elements (Q.Free) := Item;
    Q.Free := Q.Free + 1;
    Q.Is_Empty := False;
  end Enqueue;

  procedure Dequeue (Item : out Element; Q : in out Queue_Type) is
  begin
    Item := Q.Elements (Q.Top);
    Q.Top := Q.Top + 1;
    Q.Is_Empty := (Q.Top = Q.Free);
  end Dequeue;
end Queue_Pack_Contract;
```

No checks in the implementation part, as all required conditions have been guaranteed via the specifications.

### A contracting queue test program

```
with Ada.Text_IO; use Ada.Text_IO;
with Exceptions; use Exceptions;
with Queue_Pack_Contract; use Queue_Pack_Contract;
with System.Assertions; use System.Assertions;

procedure Queue_Test_Contract is
  Queue : Queue_Type;
  Item : Element;
begin
  Enqueue (Item => 1, Q => Queue);
  Enqueue (Item => 2, Q => Queue);
  Dequeue (Item, Queue); Put (Element'Image (Item));
  Dequeue (Item, Queue); Put (Element'Image (Item));
  Dequeue (Item, Queue); -- will produce an Assert_Failure
  Put (Element'Image (Item));
  Put ("Queue is empty on exit: "); Put (Boolean'Image (Is_Empty (Queue)));
exception
  when Exception_Id : Assert_Failure => Show_Exception (Exception_Id);
end Queue_Test_Contract;
```

**A contracting queue test program**

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

with Exceptions;
with Queue_Pack_Contract; use Queue_Pack_Contract;
with System.Assertions; use System.Assertions;
procedure Queue_Test_Contract is
  Queue : Queue_Type;
  Item : Element;
begin
  Enqueue (Item => 1, 0 => Queue);
  Enqueue (Item => 2, 0 => Queue);
  Dequeue (Item, Queue); Put (Element'Image (Item));
  Dequeue (Item, Queue); Put (Element'Image (Item));
  Dequeue (Item, Queue); --- will produce an ASSERT_FAILURE
  Put (Element'Image (Item));
  Put ("Queue is empty on exit."); Put (Boolean'Image (Is_Empty (Queue)));
exception
  when Exception_Id : Assert_Failure => Show_Exception (Exception_Id);
end Queue_Test_Contract;
```

Violated Pre-condition will raise an assert failure exception.

**Ada**

**Language refresher / introduction course**

**Generic (polymorphic) packages**

- Introducing:
- Specification of generic packages
- Instantiation of generic packages

**A contracting queue test program**

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

with Exceptions;
with Queue_Pack_Contract; use Queue_Pack_Contract;
with System.Assertions; use System.Assertions;
procedure Queue_Test_Contract is
  Queue : Queue_Type;
  Item : Element;
begin
  Enqueue (Item => 1, 0 => Queue);
  Enqueue (Item => 2, 0 => Queue);
  Dequeue (Item, Queue); Put (Element'Image (Item));
  Dequeue (Item, Queue); Put (Element'Image (Item));
  Dequeue (Item, Queue); --- will produce an ASSERT_FAILURE
  Put (Element'Image (Item));
  Put ("Queue is empty on exit."); Put (Boolean'Image (Is_Empty (Queue)));
exception
  when Exception_Id : Assert_Failure => Show_Exception (Exception_Id);
end Queue_Test_Contract;
```

... anything on this slide still not perfectly clear?

**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: Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
  function Is_Empty (Queue: Queue_Type) return Boolean;
  function Is_Full (Queue: Queue_Type) return Boolean;
  QueueOverflow, QueueUnderFlow: exception;
private
  type Marker is mod QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type record
    Top, Free: Marker := Marker'First;
    Is_Empty: Boolean := True;
  end record;
  Elements: List;
end Queue_Pack_Generic;
```

**A contracted q1**

```
package Queue_Pack_Contract is
  (...)
  procedure Enqueue (Item : Element; Q : Queue_Type);
  Pre => not Is_Full (Q), -- could also be ">= True" according to specifications
  Post => not Is_Empty (Q) and then length (Q) = length (Q@10) + 1
  and then lookhead (Q, length (Q)) = Item
  and then (for all ix in 1 .. length (Q@10)
    => lookhead (Q, ix) = lookhead (Q@10, ix));
  procedure Dequeue (Item : out Element; Q : in out Queue_Type) with
  Pre => not Is_Empty (Q), -- could also be ">= True" according to specifications
  Post => not Is_Full (Q) and then length (Q) = length (Q@10) - 1
  and then (for all ix in 1 .. length (Q)
    => lookhead (Q, ix) = lookhead (Q@10, ix + 1));
  (...)
end Queue_Pack_Contract;
```

Exceptions are commonly preferred to handle rare, yet valid situations. Contracts are commonly used to test program correctness with respect to its specifications.

Those contracts can be used to fully specify operations and types. Specifications should be complete, consistent and canonical, while using as little implementation details as possible.

**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: Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
  function Is_Empty (Queue: Queue_Type) return Boolean;
  function Is_Full (Queue: Queue_Type) return Boolean;
  QueueOverflow, QueueUnderFlow: exception;
private
  type Marker is mod QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type record
    Top, Free: Marker := Marker'First;
    Is_Empty: Boolean := True;
  end record;
  Elements: List;
end Queue_Pack_Generic;
```

The type of Element now becomes a parameter of a generic package.

No restrictions (private) have been set for the type of Element.

Haskell syntax:  
enqueue :: a -> Queue a -> Queue a

**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: Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
  function Is_Empty (Queue: Queue_Type) return Boolean;
  function Is_Full (Queue: Queue_Type) return Boolean;
  QueueOverflow, QueueUnderFlow: exception;
private
  type Marker is mod QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type record
    Top, Free: Marker := Marker'First;
    Is_Empty: Boolean := True;
  end record;
  Elements: List;
end Queue_Pack_Generic;
```

Generic aspects can include:

- Type categories
- Incomplete types
- Constants
- Procedures and functions
- Other packages
- Objects (interfaces)

Default values can be provided (making those parameters optional)

**A generic queue implementation**

```
package body Queue_Pack_Generic is
  procedure Enqueue (Item: Element; Queue: in out Queue_Type) is
  begin
    if Is_Full (Queue) then
      raise QueueOverflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free := Queue.Free + 1;
  end Enqueue;
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
  begin
    if Is_Empty (Queue) then
      raise QueueUnderFlow;
    end if;
    Item := Queue.Elements (Queue.Top);
    Queue.Top := Queue.Top + 1;
  end Dequeue;
  function Is_Empty (Queue: Queue_Type) return Boolean is (Queue.Is_Empty);
  function Is_Full (Queue: Queue_Type) return Boolean is
  (not Queue.Is_Empty and then Queue.Top = Queue.Free);
end Queue_Pack_Generic;
```

**Identical**

### A generic queue test program

```
with Queue_Pack_Generic; -- cannot apply 'use' clause here
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; -- 'use' clause can be applied to instantiated package
  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;
```

### 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: Element; Queue: in out Queue_Type);
  procedure Dequeue (Item: out Element; Queue: in out Queue_Type);
  function Is_Empty (Queue : Queue_Type) return Boolean;
  function Is_Full (Queue : Queue_Type) return Boolean;
  Queueoverflow, Queueunderflow : exception;
private
  type Marker is mod QueueSize;
  type List is array (Marker) of Element;
  type Queue_Type is record
    Top, Free : Marker := Marker'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Generic;
```

None of the packages so far can be used in a concurrent environment.

### A generic protected queue *specification*

```
generic
  type Element is private;
  type Index is mod <>; -- Modulo defines size of the queue.
package Queue_Pack_Protected_Generic is
  type Queue_Type is limited private;
  protected type Protected_Queue is
    entry Enqueue (Item : Element);
    entry Dequeue (Item : out Element);
    procedure Empty_Queue;
    function Is_Empty return Boolean;
    function Is_Full return Boolean;
  private
    Queue : Queue_Type;
  end Protected_Queue;
private
  type List is array (Index) of Element;
  type Queue_Type is record
    Top, Free : Index := Index'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Protected_Generic;
```

Generic components of the package:  
Element can be anything while the Index need to be a modulo type.

### A generic queue test program

```
with Queue_Pack_Generic; -- cannot apply 'use' clause here
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; -- 'use' clause can be applied to instantiated package
  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;
```

Instantiate generic package



## Language refresher / introduction course

Ada

### Access routines for concurrent systems

... introducing:

- Protected objects
- Entry guards
- Side-effecting (mutually exclusive) entry and procedure calls
- Side-effect-free (concurrent) function calls

### A generic protected queue *specification*

```
generic
  type Element is private;
  type Index is mod <>; -- Modulo defines size of the queue.
package Queue_Pack_Protected_Generic is
  type Queue_Type is limited private;
  protected type Protected_Queue is
    entry Enqueue (Item : Element);
    entry Dequeue (Item : out Element);
    procedure Empty_Queue;
    function Is_Empty return Boolean;
    function Is_Full return Boolean;
  private
    Queue : Queue_Type;
  end Protected_Queue;
private
  type List is array (Index) of Element;
  type Queue_Type is record
    Top, Free : Index := Index'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Protected_Generic;
```

Queue is protected for safe concurrent access.  
Three categories of access routines are distinguished by the keywords:  
entry, procedure, function

### A generic queue test program

```
with Queue_Pack_Generic; -- cannot apply 'use' clause here
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; -- 'use' clause can be applied to instantiated package
  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;
```

... anything on this slide still not perfectly clear?

### A generic protected queue *specification*

```
generic
  type Element is private;
  type Index is mod <>; -- Modulo defines size of the queue.
package Queue_Pack_Protected_Generic is
  type Queue_Type is limited private;
  protected type Protected_Queue is
    entry Enqueue (Item : Element);
    entry Dequeue (Item : out Element);
    procedure Empty_Queue;
    function Is_Empty return Boolean;
    function Is_Full return Boolean;
  private
    Queue : Queue_Type;
  end Protected_Queue;
private
  type List is array (Index) of Element;
  type Queue_Type is record
    Top, Free : Index := Index'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Protected_Generic;
```

### A generic protected queue *specification*

```
generic
  type Element is private;
  type Index is mod <>; -- Modulo defines size of the queue.
package Queue_Pack_Protected_Generic is
  type Queue_Type is limited private;
  protected type Protected_Queue is
    entry Enqueue (Item : Element);
    entry Dequeue (Item : out Element);
    procedure Empty_Queue;
    function Is_Empty return Boolean;
    function Is_Full return Boolean;
  private
    Queue : Queue_Type;
  end Protected_Queue;
private
  type List is array (Index) of Element;
  type Queue_Type is record
    Top, Free : Index := Index'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Protected_Generic;
```

Procedures are **mutually exclusive** to all other access routines.

Rationale:  
Procedures can modify the protected data.  
Hence they need a guarantee for exclusive access.

### A generic protected queue *specification*

```

generic
  type Element is private;
  type Index is mod <>; -- Modulo defines size of the queue.
package Queue_Pack_Protected_Generic is
  type Queue_Type is limited private;
  protected type Protected_Queue is
    entry Enqueue (Item : Element);
    entry Dequeue (Item : out Element);
    procedure Empty_Queue;
    function Is_Empty return Boolean;
    function Is_Full return Boolean;
  private
    Queue : Queue_Type;
  end Protected_Queue;
private
  type List is array (Index) of Element;
  type Queue_Type is record
    Top, Free : Index := Index'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Protected_Generic;

```

Functions are mutually exclusive to procedures and entries, yet concurrent to other functions.

**Rationale:**  
The compiler enforces those functions to be side-effect-free with respect to the protected data. Hence concurrent access can be granted among functions without risk.

### A generic protected queue *implementation*

```

package body Queue_Pack_Protected_Generic is
  protected body Protected_Queue is
    entry Enqueue (Item : Element) when not Is_Full is
    begin
      Queue.Elements (Queue.Free) := Item; Queue.Free := Index'Succ (Queue.Free);
      Queue.Is_Empty := False;
    end Enqueue;
    entry Dequeue (Item : out Element) when not Is_Empty is
    begin
      Item := Queue.Elements (Queue.Top); Queue.Top := Index'Succ (Queue.Top);
      Queue.Is_Empty := Queue.Top = Queue.Free;
    end Dequeue;
    procedure Empty_Queue is
    begin
      Queue.Top := Index'First; Queue.Free := Index'First; Queue.Is_Empty := True;
    end Empty_Queue;
    function Is_Empty return Boolean is (Queue.Is_Empty);
    function Is_Full return Boolean is
      (not Queue.Is_Empty and then Queue.Top = Queue.Free);
  end Protected_Queue;
end Queue_Pack_Protected_Generic;

```

### A generic protected queue *test program*

```

with Ada.Task_Identification; use Ada.Task_Identification;
with Ada.Text_IO; use Ada.Text_IO;
with Queue_Pack_Protected_Generic;
procedure Queue_Test_Protected_Generic is
  type Queue_Size is mod 3;
  package Queue_Pack_Protected_Character is
    new Queue_Pack_Protected_Generic (Element => Character, Index => Queue_Size);
  use Queue_Pack_Protected_Character;
  Queue : Protected_Queue;
  type Task_Index is range 1 .. 3;
  task type Producer;
  task type Consumer;
  Producers : array (Task_Index) of Producer;
  Consumers : array (Task_Index) of Consumer;
  (..)
begin
  null;
end Queue_Test_Protected_Generic;

```

### A generic protected queue *specification*

```

generic
  type Element is private;
  type Index is mod <>; -- Modulo defines size of the queue.
package Queue_Pack_Protected_Generic is
  type Queue_Type is limited private;
  protected type Protected_Queue is
    entry Enqueue (Item : Element);
    entry Dequeue (Item : out Element);
    procedure Empty_Queue;
    function Is_Empty return Boolean;
    function Is_Full return Boolean;
  private
    Queue : Queue_Type;
  end Protected_Queue;
private
  type List is array (Index) of Element;
  type Queue_Type is record
    Top, Free : Index := Index'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Protected_Generic;

```

Entries are mutually exclusive to all other access routines and also provide one guard per entry which need to evaluate to True before entry is granted. The guard expressions are defined in the implementation part.

**Rationale:**  
Entries can be blocking even if the protected object itself is unlocked. Hence a separate task waiting queue is provided per entry.

### A generic protected queue *implementation*

```

package body Queue_Pack_Protected_Generic is
  protected body Protected_Queue is
    entry Enqueue (Item : Element) when not Is_Full is
    begin
      Queue.Elements (Queue.Free) := Item; Queue.Free := Index'Succ (Queue.Free);
      Queue.Is_Empty := False;
    end Enqueue;
    entry Dequeue (Item : out Element) when not Is_Empty is
    begin
      Item := Queue.Elements (Queue.Top); Queue.Top := Index'Succ (Queue.Top);
      Queue.Is_Empty := Queue.Top = Queue.Free;
    end Dequeue;
    procedure Empty_Queue is
    begin
      Queue.Top := Index'First; Queue.Free := Index'First; Queue.Is_Empty := True;
    end Empty_Queue;
    function Is_Empty return Boolean is (Queue.Is_Empty);
    function Is_Full return Boolean is
      (not Queue.Is_Empty and then Queue.Top = Queue.Free);
  end Protected_Queue;
end Queue_Pack_Protected_Generic;

```

Guard expressions follow after when in the implementation of entries.

Tasks are automatically blocked or released depending on the state of the guard. Guard expressions are re-evaluated on exiting an entry or procedure (no point to re-check them at any other time). Exactly one waiting task on one entry is released.

### A generic protected queue *test program*

```

with Ada.Task_Identification; use Ada.Task_Identification;
with Ada.Text_IO; use Ada.Text_IO;
with Queue_Pack_Protected_Generic;
procedure Queue_Test_Protected_Generic is
  type Queue_Size is mod 3;
  package Queue_Pack_Protected_Character is
    new Queue_Pack_Protected_Generic (Element => Character, Index => Queue_Size);
  use Queue_Pack_Protected_Character;
  Queue : Protected_Queue;
  type Task_Index is range 1 .. 3;
  task type Producer;
  task type Consumer;
  Producers : array (Task_Index) of Producer;
  Consumers : array (Task_Index) of Consumer;
  (..)
begin
  null;
end Queue_Test_Protected_Generic;

```

If more than one instance of a specific task is to be run then a task type (as opposed to a concrete task) is declared.

### A generic protected queue *specification*

```

generic
  type Element is private;
  type Index is mod <>; -- Modulo defines size of the queue.
package Queue_Pack_Protected_Generic is
  type Queue_Type is limited private;
  protected type Protected_Queue is
    entry Enqueue (Item : Element);
    entry Dequeue (Item : out Element);
    procedure Empty_Queue;
    function Is_Empty return Boolean;
    function Is_Full return Boolean;
  private
    Queue : Queue_Type;
  end Protected_Queue;
private
  type List is array (Index) of Element;
  type Queue_Type is record
    Top, Free : Index := Index'First;
    Is_Empty : Boolean := True;
    Elements : List;
  end record;
end Queue_Pack_Protected_Generic;

```

... anything on this slide still not perfectly clear?

### A generic protected queue *implementation*

```

package body Queue_Pack_Protected_Generic is
  protected body Protected_Queue is
    entry Enqueue (Item : Element) when not Is_Full is
    begin
      Queue.Elements (Queue.Free) := Item; Queue.Free := Index'Succ (Queue.Free);
      Queue.Is_Empty := False;
    end Enqueue;
    entry Dequeue (Item : out Element) when not Is_Empty is
    begin
      Item := Queue.Elements (Queue.Top); Queue.Top := Index'Succ (Queue.Top);
      Queue.Is_Empty := Queue.Top = Queue.Free;
    end Dequeue;
    procedure Empty_Queue is
    begin
      Queue.Top := Index'First; Queue.Free := Index'First; Queue.Is_Empty := True;
    end Empty_Queue;
    function Is_Empty return Boolean is (Queue.Is_Empty);
    function Is_Full return Boolean is
      (not Queue.Is_Empty and then Queue.Top = Queue.Free);
  end Protected_Queue;
end Queue_Pack_Protected_Generic;

```

... anything on this slide still not perfectly clear?

### A generic protected queue *test program*

```

with Ada.Task_Identification; use Ada.Task_Identification;
with Ada.Text_IO; use Ada.Text_IO;
with Queue_Pack_Protected_Generic;
procedure Queue_Test_Protected_Generic is
  type Queue_Size is mod 3;
  package Queue_Pack_Protected_Character is
    new Queue_Pack_Protected_Generic (Element => Character, Index => Queue_Size);
  use Queue_Pack_Protected_Character;
  Queue : Protected_Queue;
  type Task_Index is range 1 .. 3;
  task type Producer;
  task type Consumer;
  Producers : array (Task_Index) of Producer;
  Consumers : array (Task_Index) of Consumer;
  (..)
begin
  null;
end Queue_Test_Protected_Generic;

```

Multiple instances of a task can be instantiated e.g. by declaring an array of this task type.

Tasks are started right when such an array is created.

## A generic protected queue test program

```

with Ada.Task_Identification; use Ada.Task_Identification;
with Ada.Text_IO; use Ada.Text_IO;
with Queue_Pack_Protected_Generic;
procedure Queue_Test_Protected_Generic is
  type Queue_Size is mod 3;
  package Queue_Pack_Protected_Character is
    new Queue_Pack_Protected_Generic (Element => Character, Index => Queue_Size);
  use Queue_Pack_Protected_Character;
  Queue : Protected_Queue;
  type Task_Index is range 1 .. 3;
  task type Producer;
  task type Consumer;
  Producers : array (Task_Index) of Producer;
  Consumers : array (Task_Index) of Consumer;
  (...)
begin
  null;
end Queue_Test_Protected_Generic;

```

These declarations spawned off all the production code.

Often there are no statements for the "main task" (here explicitly stated by a null statement).

This task is prevented from terminating though until all tasks inside its scope terminated.

## A generic protected queue test program

```

subtype Some_Characters is Character range 'a' .. 'f';
task body Producer is
begin
  for Ch in Some_Characters loop
    Put_Line ("Task " & Image (Current_Task) & " finds the queue to be " &
      (if Queue.Is_Empty then "EMPTY" else "not empty") &
      " and " &
      (if Queue.Is_Full then "FULL" else "not full") &
      " and prepares to add: " & Character'Image (Ch) &
      " to the queue.");
    Queue.Enqueue (Ch); -- task might be blocked here!
  end loop;
  Put_Line ("<---- Task " & Image (Current_Task) & " terminates.");
end Producer;

```

The executable code for a task is provided in its body.

## A generic protected queue test program

```

subtype Some_Characters is Character range 'a' .. 'f';
task body Producer is
begin
  for Ch in Some_Characters loop
    Put_Line ("Task " & Image (Current_Task) & " finds the queue to be " &
      (if Queue.Is_Empty then "EMPTY" else "not empty") &
      " and " &
      (if Queue.Is_Full then "FULL" else "not full") &
      " and prepares to add: " & Character'Image (Ch) &
      " to the queue.");
    Queue.Enqueue (Ch); -- task might be blocked here!
  end loop;
  Put_Line ("<---- Task " & Image (Current_Task) & " terminates.");
end Producer;

```

... anything on this slide still not perfectly clear?

## A generic protected queue test program

```

with Ada.Task_Identification; use Ada.Task_Identification;
with Ada.Text_IO; use Ada.Text_IO;
with Queue_Pack_Protected_Generic;
procedure Queue_Test_Protected_Generic is
  type Queue_Size is mod 3;
  package Queue_Pack_Protected_Character is
    new Queue_Pack_Protected_Generic (Element => Character, Index => Queue_Size);
  use Queue_Pack_Protected_Character;
  Queue : Protected_Queue;
  type Task_Index is range 1 .. 3;
  task type Producer;
  task type Consumer;
  Producers : array (Task_Index) of Producer;
  Consumers : array (Task_Index) of Consumer;
  (...)
begin
  null;
end Queue_Test_Protected_Generic;

```

... anything on this slide still not perfectly clear?

## A generic protected queue test program

```

subtype Some_Characters is Character range 'a' .. 'f';
task body Producer is
begin
  for Ch in Some_Characters loop
    Put_Line ("Task " & Image (Current_Task) & " finds the queue to be " &
      (if Queue.Is_Empty then "EMPTY" else "not empty") &
      " and " &
      (if Queue.Is_Full then "FULL" else "not full") &
      " and prepares to add: " & Character'Image (Ch) &
      " to the queue.");
    Queue.Enqueue (Ch); -- task might be blocked here!
  end loop;
  Put_Line ("<---- Task " & Image (Current_Task) & " terminates.");
end Producer;

```

There are three of those tasks and they are all 'hammering' the queue at full CPU speed.

## A generic protected queue test program

```

subtype Some_Characters is Character range 'a' .. 'f';
task body Producer is
begin
  for Ch in Some_Characters loop
    Put_Line ("Task " & Image (Current_Task) & " finds the queue to be " &
      (if Queue.Is_Empty then "EMPTY" else "not empty") &
      " and " &
      (if Queue.Is_Full then "FULL" else "not full") &
      " and prepares to add: " & Character'Image (Ch) &
      " to the queue.");
    Queue.Enqueue (Ch); -- task might be blocked here!
  end loop;
  Put_Line ("<---- Task " & Image (Current_Task) & " terminates.");
end Producer;

```

## A generic protected queue test program

```

subtype Some_Characters is Character range 'a' .. 'f';
task body Producer is
begin
  for Ch in Some_Characters loop
    Put_Line ("Task " & Image (Current_Task) & " finds the queue to be " &
      (if Queue.Is_Empty then "EMPTY" else "not empty") &
      " and " &
      (if Queue.Is_Full then "FULL" else "not full") &
      " and prepares to add: " & Character'Image (Ch) &
      " to the queue.");
    Queue.Enqueue (Ch); -- task might be blocked here!
  end loop;
  Put_Line ("<---- Task " & Image (Current_Task) & " terminates.");
end Producer;

```

Tasks automatically terminate once they reach their end declaration (and once all inner tasks are terminated).

## A generic protected queue test program

```

task body Consumer is
  Item : Character;
  Counter : Natural := 0;
begin
  loop
    Queue.Dequeue (Item); -- task might be blocked here!
    Counter := Natural'Succ (Counter);
    Put_Line ("Task " & Image (Current_Task) &
      " received: " & Character'Image (Item) &
      " and the queue appears to be " &
      (if Queue.Is_Empty then "EMPTY" else "not empty") &
      " and " &
      (if Queue.Is_Full then "FULL" else "not full") &
      " afterwards.");
    exit when Item = Some_Characters'Last;
  end loop;
  Put_Line ("<---- Task " & Image (Current_Task) &
    " terminates and received " & Natural'Image (Counter) & " items.");
end Consumer;

```

## A generic protected queue test program

```

task body Consumer is
  Item : Character;
  Counter : Natural := 0;
begin
  loop
    Queue.Dequeue (Item); -- task might be blocked here!
    Counter := Natural'Succ (Counter);
    Put_Line ("Task " & Image (Current_Task) &
      " received: " & Character'Image (Item) &
      " and the queue appears to be " &
      (if Queue.Is_Empty then "EMPTY" else "not empty") &
      " and " &
      (if Queue.Is_Full then "FULL" else "not full") &
      " afterwards.");
    exit when Item = Some_Characters'Last;
  end loop;
  Put_Line ("<---- Task " & Image (Current_Task) &
    " terminates and received " & Natural'Image (Counter) & " items.");
end Consumer;

```

Another three tasks are all 'hammering' the queue at this end and at full CPU speed.

### A generic protected queue test program

```

task body Consumer is
  Item : Character;
  Counter : Natural := 0;
begin
  loop
    Queue.Dequeue (Item); -- task might be blocked here!
    Counter := Natural'Succ (Counter);
    Put_Line ("Task " & Image (Current_Task) &
      " received: " & Character'Image (Item) &
      " and the queue appears to be " &
      (if Queue.Is_Empty then "EMPTY" else "not empty") &
      " and " &
      (if Queue.Is_Full then "FULL" else "not full") &
      " afterwards.");
    exit when Item = Some_Characters'Last;
  end loop;
  Put_Line ("<---- Task " & Image (Current_Task) &
    " terminates and received " & Natural'Image (Counter) & " items.");
end Consumer;

```

... anything on this slide still not perfectly clear?

### Language refresher / introduction course

Ada

#### Abstract types & dispatching

... introducing:

- Abstract tagged types & subroutines (Interfaces)
- Concrete implementation of abstract types
- Dynamic dispatching to different packages, tasks, protected types or partitions.
- Synchronous message passing.

### An abstract queue specification

```

generic
  type Element is private;
package Queue_Pack_Abstract is
  type Queue_Interface is synchronized interface;
  procedure Enqueue (Q : in out Queue_Interface; Item : Element) is abstract;
  procedure Dequeue (Q : in out Queue_Interface; Item : out Element) is abstract;
end Queue_Pack_Abstract;

```

**Motivation:**  
Different, derived implementations (potentially on different computers) can be passed around and referred to with the same common interface as defined here.

### A generic protected queue test program

```

Task producers(1) finds the queue to be EMPTY and not full and prepares to add: 'a' to the queue.
Task producers(1) finds the queue to be not empty and not full and prepares to add: 'b' to the queue.
Task producers(1) finds the queue to be not empty and not full and prepares to add: 'c' to the queue.
Task producers(1) finds the queue to be not empty and FULL and prepares to add: 'd' to the queue.
Task producers(2) finds the queue to be not empty and FULL and prepares to add: 'a' to the queue.
Task producers(3) finds the queue to be not empty and FULL and prepares to add: 'a' to the queue.
Task consumers(1) received: 'a' and the queue appears to be not empty and FULL afterwards.
Task consumers(1) received: 'b' and the queue appears to be not empty and FULL afterwards.
Task consumers(1) received: 'c' and the queue appears to be not empty and FULL afterwards.
Task consumers(1) received: 'd' and the queue appears to be not empty and not full afterwards.
Task consumers(1) received: 'a' and the queue appears to be not empty and not full afterwards.
<---- Task producers(1) terminates.
...
Task consumers(3) received: 'b' and the queue appears to be EMPTY and not full afterwards.
<---- Task consumers(2) terminates and received 1 items.
...
Task producers(2) terminates.
...
Task producers(3) terminates.
...
Task consumers(1) terminates and received 12 items.
<---- Task consumers(3) terminates and received 5 items.

```

What is going on here?

### Language refresher / introduction course

Ada

#### Abstract types & dispatching

... introducing:

- Abstract tagged types & subroutines (Interfaces)
- Concrete implementation of abstract types
- Dynamic dispatching to different packages, tasks, protected types or partitions.
- Synchronous message passing.

– Advanced topic –  
**Proceed with caution!**

### An abstract queue specification

```

generic
  type Element is private;
package Queue_Pack_Abstract is
  type Queue_Interface is synchronized interface;
  procedure Enqueue (Q : in out Queue_Interface; Item : Element) is abstract;
  procedure Dequeue (Q : in out Queue_Interface; Item : out Element) is abstract;
end Queue_Pack_Abstract;

```

synchronized means that this interface can only be implemented by synchronized entities like protected objects (as seen above) or synchronous message passing.

Abstract, empty type definition which serves to define interface templates.

### A generic protected queue test program

```

Task producers(1) finds the queue to be EMPTY and not full and prepares to add: 'a' to the queue.
Task producers(2) finds the queue to be EMPTY and not full and prepares to add: 'a' to the queue.
Task producers(1) finds the queue to be not empty and not full and prepares to add: 'b' to the queue.
Task consumers(1) received: 'a' and the queue appears to be EMPTY and not full afterwards.
Task producers(3) finds the queue to be EMPTY and not full and prepares to add: 'a' to the queue.
Task producers(1) finds the queue to be EMPTY and not full and prepares to add: 'c' to the queue.
Task producers(2) finds the queue to be EMPTY and not full and prepares to add: 'b' to the queue.
Task consumers(2) received: 'a' and the queue appears to be EMPTY and not full afterwards.
Task consumers(3) received: 'b' and the queue appears to be EMPTY and not full afterwards.
...
<---- Task producers(1) terminates.
Task producers(2) finds the queue to be not empty and FULL and prepares to add: 'f' to the queue.
Task consumers(2) received: 'f' and the queue appears to be not empty and not full afterwards.
Task consumers(3) received: 'e' and the queue appears to be not empty and not full afterwards.
Task producers(3) finds the queue to be not empty and not full and prepares to add: 'f' to the queue.
Task consumers(1) received: 'd' and the queue appears to be not empty and not full afterwards.
<---- Task producers(2) terminates.
<---- Task consumers(2) terminates and received 5 items.
Task consumers(3) received: 'e' and the queue appears to be not empty and not full afterwards.
<---- Task producers(3) terminates.
Task consumers(1) received: 'f' and the queue appears to be not empty and not full afterwards.
Task consumers(3) received: 'f' and the queue appears to be EMPTY and not full afterwards.
<---- Task consumers(1) terminates and received 6 items.
<---- Task consumers(3) terminates and received 7 items.

```

Does this make any sense?

### An abstract queue specification

```

generic
  type Element is private;
package Queue_Pack_Abstract is
  type Queue_Interface is synchronized interface;
  procedure Enqueue (Q : in out Queue_Interface; Item : Element) is abstract;
  procedure Dequeue (Q : in out Queue_Interface; Item : out Element) is abstract;
end Queue_Pack_Abstract;

```

### An abstract queue specification

```

generic
  type Element is private;
package Queue_Pack_Abstract is
  type Queue_Interface is synchronized interface;
  procedure Enqueue (Q : in out Queue_Interface; Item : Element) is abstract;
  procedure Dequeue (Q : in out Queue_Interface; Item : out Element) is abstract;
end Queue_Pack_Abstract;

```

Abstract methods need to be overridden with concrete methods when a new type is derived from it.

### An abstract queue specification

```
generic
  type Element is private;
package Queue_Pack_Abstract is
  type Queue_Interface is synchronized interface;
  procedure Enqueue (Q : in out Queue_Interface; Item : Element) is abstract;
  procedure Dequeue (Q : in out Queue_Interface; Item : out Element) is abstract;
end Queue_Pack_Abstract;
```

... this does not require an implementation package (as all procedures are abstract)

... anything on this slide still not perfectly clear?

### A concrete queue specification

```
with Queue_Pack_Abstract;
generic
  with package Queue_Instance is new Queue_Pack_Abstract (<>);
  type Index is mod <>; -- Modulo defines size of the queue.
package Queue_Pack_Concrete is
  use Queue_Instance;
  type Queue_Type is limited private;
  protected type Protected_Queue is new Queue_Interface with
    overriding entry Enqueue (Item : Element);
    overriding entry Dequeue (Item : out Element);
  procedure Empty_Queue;
  function Is_Empty return Boolean;
  function Is_Full return Boolean;
private
  Queue : Queue_Type;
end Protected_Queue;
private
  (...) -- as all previous private queue declarations
end Queue_Pack_Concrete;
```

A synchronous implementation of the abstract type Queue\_Interface

All abstract methods are overridden with concrete implementations.

### A concrete queue implementation

```
package body Queue_Pack_Concrete is
  protected body Protected_Queue is
    entry Enqueue (Item : Element) when not Is_Full is
    begin
      Queue.Elements (Queue.Free) := Item; Queue.Free := Index'Succ (Queue.Free);
      Queue.Is_Empty := False;
    end Enqueue;
    entry Dequeue (Item : out Element) when Is_Empty is
    begin
      Item := Queue.Elements (Queue.Top); Queue.Top := Index'Succ (Queue.Top);
      Queue.Is_Empty := Queue.Top = Queue.Free;
    end Dequeue;
  procedure Empty_Queue is
  begin
    Queue.Top := Index'First; Queue.Free := Index'First; Queue.Is_Empty := True;
  end Empty_Queue;
  function Is_Empty return Boolean is (Queue.Is_Empty);
  function Is_Full return Boolean is
    (not Queue.Is_Empty and then Queue.Top = Queue.Free);
  end Protected_Queue;
end Queue_Pack_Concrete;
```

### A concrete queue specification

```
with Queue_Pack_Abstract;
generic
  with package Queue_Instance is new Queue_Pack_Abstract (<>);
  type Index is mod <>; -- Modulo defines size of the queue.
package Queue_Pack_Concrete is
  use Queue_Instance;
  type Queue_Type is limited private;
  protected type Protected_Queue is new Queue_Interface with
    overriding entry Enqueue (Item : Element);
    overriding entry Dequeue (Item : out Element);
  not overriding procedure Empty_Queue;
  not overriding function Is_Empty return Boolean;
  not overriding function Is_Full return Boolean;
private
  Queue : Queue_Type;
end Protected_Queue;
private
  (...) -- as all previous private queue declarations
end Queue_Pack_Concrete;
```

### A concrete queue specification

```
with Queue_Pack_Abstract;
generic
  with package Queue_Instance is new Queue_Pack_Abstract (<>);
  type Index is mod <>; -- Modulo defines size of the queue.
package Queue_Pack_Concrete is
  use Queue_Instance;
  type Queue_Type is limited private;
  protected type Protected_Queue is new Queue_Interface with
    overriding entry Enqueue (Item : Element);
    overriding entry Dequeue (Item : out Element);
  not overriding procedure Empty_Queue;
  not overriding function Is_Empty return Boolean;
  not overriding function Is_Full return Boolean;
private
  Queue : Queue_Type;
end Protected_Queue;
private
  (...) -- as all previous private queue declarations
end Queue_Pack_Concrete;
```

Other (not-overriding) methods can be added.

### A dispatching test program

```
with Ada.Text_IO; use Ada.Text_IO;
with Queue_Pack_Abstract;
with Queue_Pack_Concrete;
procedure Queue_Test_Dispatching is
  package Queue_Pack_Abstract_Character is
    new Queue_Pack_Abstract (Character);
  use Queue_Pack_Abstract_Character;
  type Queue_Size is mod 3;
  package Queue_Pack_Character is
    new Queue_Pack_Concrete (Queue_Pack_Abstract_Character, Queue_Size);
  use Queue_Pack_Character;
  type Queue_Class is access all Queue_Interface'Class;
  task Queue_Holder; -- could be on an individual partition / separate computer
  task Queue_User is -- could be on an individual partition / separate computer
  entry Send_Queue (Remote_Queue : Queue_Class);
  end Queue_User;
  (...)
begin
  null;
end Queue_Test_Dispatching;
```

### A concrete queue specification

```
with Queue_Pack_Abstract;
generic
  with package Queue_Instance is new Queue_Pack_Abstract (<>);
  type Index is mod <>; -- Modulo defines size of the queue.
package Queue_Pack_Concrete is
  use Queue_Instance;
  type Queue_Type is limited private;
  protected type Protected_Queue is new Queue_Interface with
    overriding entry Enqueue (Item : Element);
    overriding entry Dequeue (Item : out Element);
  procedure Empty_Queue;
  function Is_Empty return Boolean;
  function Is_Full return Boolean;
private
  Queue : Queue_Type;
end Protected_Queue;
private
  (...) -- as all previous private queue declarations
end Queue_Pack_Concrete;
```

A generic package which takes another generic package as a parameter.

### A concrete queue specification

```
with Queue_Pack_Abstract;
generic
  with package Queue_Instance is new Queue_Pack_Abstract (<>);
  type Index is mod <>; -- Modulo defines size of the queue.
package Queue_Pack_Concrete is
  use Queue_Instance;
  type Queue_Type is limited private;
  protected type Protected_Queue is new Queue_Interface with
    overriding entry Enqueue (Item : Element);
    overriding entry Dequeue (Item : out Element);
  procedure Empty_Queue;
  function Is_Empty return Boolean;
  function Is_Full return Boolean;
private
  Queue : Queue_Type;
end Protected_Queue;
private
  (...) -- as all previous private queue declarations
end Queue_Pack_Concrete;
```

... anything on this slide still not perfectly clear?

### A dispatching test program

```
with Ada.Text_IO; use Ada.Text_IO;
with Queue_Pack_Abstract;
with Queue_Pack_Concrete;
procedure Queue_Test_Dispatching is
  package Queue_Pack_Abstract_Character is
    new Queue_Pack_Abstract (Character);
  use Queue_Pack_Abstract_Character;
  type Queue_Size is mod 3;
  package Queue_Pack_Character is
    new Queue_Pack_Concrete (Queue_Pack_Abstract_Character, Queue_Size);
  use Queue_Pack_Character;
  type Queue_Class is access all Queue_Interface'Class;
  task Queue_Holder; -- could be on an individual partition / separate computer
  task Queue_User is -- could be on an individual partition / separate computer
  entry Send_Queue (Remote_Queue : Queue_Class);
  end Queue_User;
  (...)
begin
  null;
end Queue_Test_Dispatching;
```

Sequence of instantiations



## A dispatching test program

```

with Ada.Text_IO; use Ada.Text_IO;
with Queue_Pack_Abstract;
with Queue_Pack_Concrete;
procedure Queue_Test_Dispatching is
  package Queue_Pack_Abstract_Character is
    new Queue_Pack_Abstract (Character);
  use Queue_Pack_Abstract_Character;
  type Queue_Size is mod 3;
  package Queue_Pack_Character is
    new Queue_Pack_Concrete (Queue_Pack_Abstract_Character, Queue_Size);
  use Queue_Pack_Character;
  type Queue_Class is access all Queue_Interface'Class;
  task Queue_Holder; -- could be on an individual partition / separate computer
  task Queue_User is -- could be on an individual partition / separate computer
  entry Send_Queue (Remote_Queue : Queue_Class);
  end Queue_User;
  (...)
begin
  null;
end Queue_Test_Dispatching;

```

Type which can refer to any instance of Queue\_Interface

## A dispatching test program (cont.)

```

task body Queue_Holder is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item : Character;
begin
  Queue_User.Send_Queue (Local_Queue);
  Local_Queue.all.Dequeue (Item);
  Put_Line ("Local dequeue (Holder): " & Character'Image (Item));
end Queue_Holder;
task body Queue_User is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item : Character;
begin
  accept Send_Queue (Remote_Queue : Queue_Class) do
    Remote_Queue.all.Enqueue ('r'); -- potentially a remote procedure call!
    Local_Queue.all.Enqueue ('1');
  end Send_Queue;
  Local_Queue.all.Dequeue (Item);
  Put_Line ("Local dequeue (User) : " & Character'Image (Item));
end Queue_User;

```

## A dispatching test program (cont.)

```

task body Queue_Holder is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item : Character;
begin
  Queue_User.Send_Queue (Local_Queue);
  Local_Queue.all.Dequeue (Item);
  Put_Line ("Local dequeue (Holder): " & Character'Image (Item));
end Queue_Holder;
task body Queue_User is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item : Character;
begin
  accept Send_Queue (Remote_Queue : Queue_Class) do
    Remote_Queue.all.Enqueue ('r'); -- potentially a remote procedure call!
    Local_Queue.all.Enqueue ('1');
  end Send_Queue;
  Local_Queue.all.Dequeue (Item);
  Put_Line ("Local dequeue (User) : " & Character'Image (Item));
end Queue_User;

```

Adding to both queues

## A dispatching test program

```

with Ada.Text_IO; use Ada.Text_IO;
with Queue_Pack_Abstract;
with Queue_Pack_Concrete;
procedure Queue_Test_Dispatching is
  package Queue_Pack_Abstract_Character is
    new Queue_Pack_Abstract (Character);
  use Queue_Pack_Abstract_Character;
  type Queue_Size is mod 3;
  package Queue_Pack_Character is
    new Queue_Pack_Concrete (Queue_Pack_Abstract_Character, Queue_Size);
  use Queue_Pack_Character;
  type Queue_Class is access all Queue_Interface'Class;
  task Queue_Holder; -- could be on an individual partition / separate computer
  task Queue_User is -- could be on an individual partition / separate computer
  entry Send_Queue (Remote_Queue : Queue_Class);
  end Queue_User;
  (...)
begin
  null;
end Queue_Test_Dispatching;

```

Declaring two concrete tasks.  
(Queue\_User has a synchronous message passing entry)

## A dispatching test program (cont.)

```

task body Queue_Holder is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item : Character;
begin
  Queue_User.Send_Queue (Local_Queue);
  Local_Queue.all.Dequeue (Item);
  Put_Line ("Local dequeue (Holder): " & Character'Image (Item));
end Queue_Holder;
task body Queue_User is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item : Character;
begin
  accept Send_Queue (Remote_Queue : Queue_Class) do
    Remote_Queue.all.Enqueue ('r'); -- potentially a remote procedure call!
    Local_Queue.all.Enqueue ('1');
  end Send_Queue;
  Local_Queue.all.Dequeue (Item);
  Put_Line ("Local dequeue (User) : " & Character'Image (Item));
end Queue_User;

```

Declaring local queues in each task.

## A dispatching test program (cont.)

```

task body Queue_Holder is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item : Character;
begin
  Queue_User.Send_Queue (Local_Queue);
  Local_Queue.all.Dequeue (Item);
  Put_Line ("Local dequeue (Holder): " & Character'Image (Item));
end Queue_Holder;
task body Queue_User is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item : Character;
begin
  accept Send_Queue (Remote_Queue : Queue_Class) do
    Remote_Queue.all.Enqueue ('r'); -- potentially a remote procedure call!
    Local_Queue.all.Enqueue ('1');
  end Send_Queue;
  Local_Queue.all.Dequeue (Item);
  Put_Line ("Local dequeue (User) : " & Character'Image (Item));
end Queue_User;

```

Tasks could run on separate computers

These two calls can be very different in nature:  
The first call is potentially tunneled through a network to another computer and thus uses a remote data structure.  
The second call is always a local call and using a local data-structure.

## A dispatching test program

```

with Ada.Text_IO; use Ada.Text_IO;
with Queue_Pack_Abstract;
with Queue_Pack_Concrete;
procedure Queue_Test_Dispatching is
  package Queue_Pack_Abstract_Character is
    new Queue_Pack_Abstract (Character);
  use Queue_Pack_Abstract_Character;
  type Queue_Size is mod 3;
  package Queue_Pack_Character is
    new Queue_Pack_Concrete (Queue_Pack_Abstract_Character, Queue_Size);
  use Queue_Pack_Character;
  type Queue_Class is access all Queue_Interface'Class;
  task Queue_Holder; -- could be on an individual partition / separate computer
  task Queue_User is -- could be on an individual partition / separate computer
  entry Send_Queue (Remote_Queue : Queue_Class);
  end Queue_User;
  (...)
begin
  null;
end Queue_Test_Dispatching;

```

... anything on this slide still not perfectly clear?

## A dispatching test program (cont.)

```

task body Queue_Holder is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item : Character;
begin
  Queue_User.Send_Queue (Local_Queue);
  Local_Queue.all.Dequeue (Item);
  Put_Line ("Local dequeue (Holder): " & Character'Image (Item));
end Queue_Holder;
task body Queue_User is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item : Character;
begin
  accept Send_Queue (Remote_Queue : Queue_Class) do
    Remote_Queue.all.Enqueue ('r'); -- potentially a remote procedure call!
    Local_Queue.all.Enqueue ('1');
  end Send_Queue;
  Local_Queue.all.Dequeue (Item);
  Put_Line ("Local dequeue (User) : " & Character'Image (Item));
end Queue_User;

```

Handing over the Holder's queue via synchronous message passing.

## A dispatching test program (cont.)

```

task body Queue_Holder is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item : Character;
begin
  Queue_User.Send_Queue (Local_Queue);
  Local_Queue.all.Dequeue (Item);
  Put_Line ("Local dequeue (Holder): " & Character'Image (Item));
end Queue_Holder;
task body Queue_User is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item : Character;
begin
  accept Send_Queue (Remote_Queue : Queue_Class) do
    Remote_Queue.all.Enqueue ('r'); -- potentially a remote procedure call!
    Local_Queue.all.Enqueue ('1');
  end Send_Queue;
  Local_Queue.all.Dequeue (Item);
  Put_Line ("Local dequeue (User) : " & Character'Image (Item));
end Queue_User;

```

Reading out 'r'

Reading out '1'

### A dispatching test program (cont.)

```

task body Queue_Holder is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item       : Character;
begin
  Queue_User.Send_Queue (Local_Queue);
  Local_Queue.all.Dequeue (Item);
  Put_Line ("Local dequeue (Holder) : " & Character'Image (Item));
end Queue_Holder;

task body Queue_User is
  Local_Queue : constant Queue_Class := new Protected_Queue;
  Item       : Character;
begin
  accept Send_Queue (Remote_Queue : Queue_Class) do
    Remote_Queue.all.Enqueue ('r'); -- potentially a remote procedure call!
    Local_Queue.all.Enqueue ('1');
  end Send_Queue;
  Local_Queue.all.Dequeue (Item);
  Put_Line ("Local dequeue (User) : " & Character'Image (Item));
end Queue_User;
    
```

... anything on this slide still not perfectly clear?

© 2010 Dave R. Zineer, The Australian National University page 146 of 758 ("Language refresher / introduction course" up to page 160)

### A data-parallel stencil program

```

config const n = 100,
  max_iterations = 50,
  epsilon = 1.0E-5,
  initial_border = 1.0;

const Matrix_w_Borders = {0 .. n+1, 0 .. n+1, 0 .. n+1},
  Matrix = Matrix_w_Borders [1 .. n, 1 .. n, 1 .. n],
  Single_Border = Matrix.exterior (1, 0, 0);

var Field : [Matrix_w_Borders] real,
  Next_Field : [Matrix] real;

proc Stencil (M : [/* Matrix_w_Borders */] real, (i, j, k) : index (Matrix)) : real {
  return (M [i - 1, j, k]
    + M [i + 1, j, k]
    + M [i, j - 1, k]
    + M [i, j + 1, k]
    + M [i, j, k + 1]
    + M [i, j, k - 1]) / 6;
}
    
```

© 2010 Dave R. Zineer, The Australian National University page 149 of 758 ("Language refresher / introduction course" up to page 160)

### A data-parallel stencil program

```

config const n = 100,
  max_iterations = 50,
  epsilon = 1.0E-5,
  initial_border = 1.0;

const Matrix_w_Borders = {0 .. n+1, 0 .. n+1, 0 .. n+1},
  Matrix = Matrix_w_Borders [1 .. n, 1 .. n, 1 .. n],
  Single_Border = Matrix.exterior (1, 0, 0);

var Field : [Matrix_w_Borders] real,
  Next_Field : [Matrix] real;

proc Stencil (M : [/* Matrix_w_Borders */] real, (i, j, k) : index (Matrix)) : real {
  return (M [i - 1, j, k]
    + M [i + 1, j, k]
    + M [i, j - 1, k]
    + M [i, j + 1, k]
    + M [i, j, k + 1]
    + M [i, j, k - 1]) / 6;
}
    
```



Declaring matrices of different, yet related dimensions.

© 2010 Dave R. Zineer, The Australian National University page 152 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### Ada

### Ada language status

- Established language standard with free and professionally supported compilers available for all major OSs and platforms.
- Emphasis on maintainability, high-integrity and efficiency.
- Stand-alone runtime environments for embedded systems.
- High integrity, real-time profiles part of the standard e.g. Ravenscar profile.

Used in many large scale and/or high integrity projects

- Commonly used in aviation industry, high speed trains, metro-systems, space programs and military programs.
- ... also increasingly on small platforms / micro-controllers.

© 2010 Dave R. Zineer, The Australian National University page 147 of 758 ("Language refresher / introduction course" up to page 160)

### A data-parallel stencil program

```

config const n = 100,
  max_iterations = 50,
  epsilon = 1.0E-5,
  initial_border = 1.0;

const Matrix_w_Borders = {0 .. n+1, 0 .. n+1, 0 .. n+1},
  Matrix = Matrix_w_Borders [1 .. n, 1 .. n, 1 .. n],
  Single_Border = Matrix.exterior (1, 0, 0);

var Field : [Matrix_w_Borders] real,
  Next_Field : [Matrix] real;

proc Stencil (M : [/* Matrix_w_Borders */] real, (i, j, k) : index (Matrix)) : real {
  return (M [i - 1, j, k]
    + M [i + 1, j, k]
    + M [i, j - 1, k]
    + M [i, j + 1, k]
    + M [i, j, k + 1]
    + M [i, j, k - 1]) / 6;
}
    
```

Configuration constants can be set via command line options:  
./Stencil --n=500

© 2010 Dave R. Zineer, The Australian National University page 150 of 758 ("Language refresher / introduction course" up to page 160)

### A data-parallel stencil program

```

config const n = 100,
  max_iterations = 50,
  epsilon = 1.0E-5,
  initial_border = 1.0;

const Matrix_w_Borders = {0 .. n+1, 0 .. n+1, 0 .. n+1},
  Matrix = Matrix_w_Borders [1 .. n, 1 .. n, 1 .. n],
  Single_Border = Matrix.exterior (1, 0, 0);

var Field : [Matrix_w_Borders] real,
  Next_Field : [Matrix] real;

proc Stencil (M : [/* Matrix_w_Borders */] real, (i, j, k) : index (Matrix)) : real {
  return (M [i - 1, j, k]
    + M [i + 1, j, k]
    + M [i, j - 1, k]
    + M [i, j + 1, k]
    + M [i, j, k + 1]
    + M [i, j, k - 1]) / 6;
}
    
```


Note the index type

Function which calculates a "stencil" value at a spot inside a given matrix

© 2010 Dave R. Zineer, The Australian National University page 153 of 758 ("Language refresher / introduction course" up to page 160)

## Language refresher / introduction course

### Chapel




Currently under development at Cray.  
(originally for the DARPA High Productivity Computing Systems initiative.)

Targeted at massively parallel computers

Language primitives for ...

- Data parallelism:
  - Distributed data storage with fine grained control ("domains").
  - Concurrent map operations (forall).
  - Concurrent fold operations (scan, reduce).
- Task parallelism:
  - concurrent loops and blocks (cobegin, coforall).
- Synchronization:
  - Task synchronization, synchronized variables, atomic sections.



© 2010 Dave R. Zineer, The Australian National University page 148 of 758 (chapter 2: "Language refresher / introduction course" up to page 160)

### A data-parallel stencil program

```

config const n = 100,
  max_iterations = 50,
  epsilon = 1.0E-5,
  initial_border = 1.0;

const Matrix_w_Borders = {0 .. n+1, 0 .. n+1, 0 .. n+1},
  Matrix = Matrix_w_Borders [1 .. n, 1 .. n, 1 .. n],
  Single_Border = Matrix.exterior (1, 0, 0);

var Field : [Matrix_w_Borders] real,
  Next_Field : [Matrix] real;

proc Stencil (M : [/* Matrix_w_Borders */] real, (i, j, k) : index (Matrix)) : real {
  return (M [i - 1, j, k]
    + M [i + 1, j, k]
    + M [i, j - 1, k]
    + M [i, j + 1, k]
    + M [i, j, k + 1]
    + M [i, j, k - 1]) / 6;
}
    
```

Defining domains to be used for multi-dimensional array declarations and assignments.

© 2010 Dave R. Zineer, The Australian National University page 151 of 758 ("Language refresher / introduction course" up to page 160)

### A data-parallel stencil program

```

config const n = 100,
  max_iterations = 50,
  epsilon = 1.0E-5,
  initial_border = 1.0;

const Matrix_w_Borders = {0 .. n+1, 0 .. n+1, 0 .. n+1},
  Matrix = Matrix_w_Borders [1 .. n, 1 .. n, 1 .. n],
  Single_Border = Matrix.exterior (1, 0, 0);

var Field : [Matrix_w_Borders] real,
  Next_Field : [Matrix] real;

proc Stencil (M : [/* Matrix_w_Borders */] real, (i, j, k) : index (Matrix)) : real {
  return (M [i - 1, j, k]
    + M [i + 1, j, k]
    + M [i, j - 1, k]
    + M [i, j + 1, k]
    + M [i, j, k + 1]
    + M [i, j, k - 1]) / 6;
}
    
```

... anything on this slide still not perfectly clear?

© 2010 Dave R. Zineer, The Australian National University page 154 of 758 ("Language refresher / introduction course" up to page 160)

### A data-parallel stencil program (cont.)

```
Field [Single_Border] = initial_border;
for l in 1 .. max_iterations {
  forall Matrix_Indices in Matrix do
    Next_Field (Matrix_Indices) = Stencil (Field, Matrix_Indices);
  const delta = max reduce abs (Field [Matrix] - Next_Field);
  Field [Matrix] = Next_Field;
  if delta < epsilon then break;
}
```

### A data-parallel stencil program (cont.)

```
Field [Single_Border] = initial_border;
for l in 1 .. max_iterations {
  forall Matrix_Indices in Matrix do
    Next_Field (Matrix_Indices) = Stencil (Field, Matrix_Indices);
  const delta = max reduce abs (Field [Matrix] - Next_Field);
  Field [Matrix] = Next_Field;
  if delta < epsilon then break;
}
```

Data parallel (divide-and-conquer) application of the max function to the component-wise differences.

"3-d data-parallel version" of (Haskell):  
foldr max minBound \$ zipwith (-) field next\_field

### A data-parallel stencil program (cont.)

```
Field [Single_Border] = initial_border;
for l in 1 .. max_iterations {
  forall Matrix_Indices in Matrix do
    Next_Field (Matrix_Indices) = Stencil (Field, Matrix_Indices);
  const delta = max reduce abs (Field [Matrix] - Next_Field);
  Field [Matrix] = Next_Field;
  if delta < epsilon then break;
}
```

Scalar to 2-d array-slice assignment  
(Technically a 3-d domain with two degenerate dimensions)

3-d array to 3-d array-slice assignment

### A data-parallel stencil program (cont.)

```
Field [Single_Border] = initial_border;
for l in 1 .. max_iterations {
  forall Matrix_Indices in Matrix do
    Next_Field (Matrix_Indices) = Stencil (Field, Matrix_Indices);
  const delta = max reduce abs (Field [Matrix] - Next_Field);
  Field [Matrix] = Next_Field;
  if delta < epsilon then break;
}
```

... anything on this slide still not perfectly clear?

### A data-parallel stencil program (cont.)

```
Field [Single_Border] = initial_border;
for l in 1 .. max_iterations {
  forall Matrix_Indices in Matrix do
    Next_Field (Matrix_Indices) = Stencil (Field, Matrix_Indices);
  const delta = max reduce abs (Field [Matrix] - Next_Field);
  Field [Matrix] = Next_Field;
  if delta < epsilon then break;
}
```

Data parallel application of the Stencil function to the whole 3-d matrix




## Language refresher / introduction course

### Summary

### Language refresher / introduction course

- Specification and implementation (body) parts, basic types
- Exceptions & Contracts
- Information hiding in specifications ('private')
- Generic programming
- Tasking
- Monitors and synchronisation ('protected', 'entries', 'selects', 'accepts')
- Abstract types and dispatching
- Data parallel operations

# Systems, Networks & Concurrency 2020



# 1

## Introduction to Concurrency

Uwe R. Zimmer - The Australian National University

## Introduction to Concurrency

### Forms of concurrency

### What is concurrency?

Working definitions:

- Literally 'concurrent' means:  
Adj.: Running together in space, as parallel lines; going on side by side, as proceedings; occurring together, as events or circumstances; existing or arising together; conjoint, associated [Oxfords English Dictionary]
- Technically 'concurrent' is usually defined negatively as:  
If there is no observer who can identify two events as being in strict temporal sequence (i.e. one event has fully terminated before the other one started) then these two events are considered concurrent.

© 2020 Uwe R. Zimmer, The Australian National University page 164 of 758 (chapter 1: "Introduction to Concurrency") up to page 202

## Introduction to Concurrency

### Forms of concurrency

### A computer scientist's view on concurrency

- Overlapped I/O and computation
  - Employ interrupt programming to handle I/O
- Multi-programming
  - Allow multiple independent programs to be executed on one CPU
- Multi-tasking
  - Allow multiple interacting processes to be executed on one CPU
- Multi-processor systems
  - Add physical/real concurrency
- Parallel Machines & distributed operating systems
  - Add (non-deterministic) communication channels
- General network architectures
  - Allow for any form of communicating, distributed entities

© 2020 Uwe R. Zimmer, The Australian National University page 167 of 758 (chapter 1: "Introduction to Concurrency") up to page 202

## Introduction to Concurrency

### References for this chapter

[Ben-Ari06]  
M. Ben-Ari  
*Principles of Concurrent and Distributed Programming*  
2006, second edition, Prentice-Hall, ISBN 0-13-711821-X

© 2020 Uwe R. Zimmer, The Australian National University page 162 of 758 (chapter 1: "Introduction to Concurrency") up to page 202

## Introduction to Concurrency

### Forms of concurrency

### Why do we need/have concurrency?

- Physics, engineering, electronics, biology, ...
  - basically every real world system is concurrent!
- Sequential processing is suggested by most core computer architectures
  - yet (almost) all current processor architectures have concurrent elements
  - and most computer systems are part of a concurrent network.
- Strict sequential processing is suggested by widely used programming languages.
  - Sequential programming delivers some fundamental components for concurrent programming
  - but we need to add a number of further crucial concepts

© 2020 Uwe R. Zimmer, The Australian National University page 165 of 758 (chapter 1: "Introduction to Concurrency") up to page 202

## Introduction to Concurrency

### Forms of concurrency

### A computer scientist's view on concurrency

Terminology for physically concurrent machines architectures:

- SISD** [single instruction, single data]
  - Sequential processors
- SIMD** [single instruction, multiple data]
  - Vector processors
- MISD** [multiple instruction, single data]
  - Pipelined processors
- MIMD** [multiple instruction, multiple data]
  - Multi-processors or computer networks

© 2020 Uwe R. Zimmer, The Australian National University page 168 of 758 (chapter 1: "Introduction to Concurrency") up to page 202

## Introduction to Concurrency

### Forms of concurrency

### What is concurrency?

Working definitions:

- Literally 'concurrent' means:  
Adj.: Running together in space, as parallel lines; going on side by side, as proceedings; occurring together, as events or circumstances; existing or arising together; conjoint, associated [Oxfords English Dictionary]

© 2020 Uwe R. Zimmer, The Australian National University page 163 of 758 (chapter 1: "Introduction to Concurrency") up to page 202

## Introduction to Concurrency

### Forms of concurrency

### Why would a computer scientist consider concurrency?

- ... to be able to connect computer systems with the real world
- ... to be able to employ / design concurrent parts of computer architectures
- ... to construct complex software packages (operating systems, compilers, databases, ...)
- ... to understand when sequential and/or concurrent programming is required
  - or: to understand when sequential or concurrent programming can be chosen freely
- ... to enhance the reactivity of a system
- ... to enhance the performance of a system
- ... to be able to design embedded systems

© 2020 Uwe R. Zimmer, The Australian National University page 166 of 758 (chapter 1: "Introduction to Concurrency") up to page 202

## Introduction to Concurrency

### Forms of concurrency

### An engineer's view on concurrency

- Multiple physical, coupled, dynamical systems form the actual environment and/or task at hand
- In order to model and control such a system, its inherent concurrency needs to be considered
- Multiple less powerful processors are often preferred over a single high-performance cpu
- The system design of usually strictly based on the structure of the given physical system.

© 2020 Uwe R. Zimmer, The Australian National University page 169 of 758 (chapter 1: "Introduction to Concurrency") up to page 202

## Introduction to Concurrency

### Forms of concurrency

### Does concurrency lead to chaos?

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

- **non-deterministic** phenomena
- **non-observable** system states
- results may depend on more than just the input parameters and states at start time (timing, throughput, load, available resources, signals ... *throughout* the execution)
- **non-reproducible**  $\Rightarrow$  debugging?

© 2020 Dave R. Zimmerman, The Australian National University page 170 of 758 (chapter 1: "Introduction to Concurrency" - up to page 202)

## Introduction to Concurrency

### Models and Terminology

### The concurrent programming abstraction

1. What appears sequential on a higher abstraction level, is usually concurrent at a lower abstraction level:

- $\Rightarrow$  e.g. Concurrent operating system or hardware components, which might not be visible at a higher programming level

2. What appears concurrent on a higher abstraction level, might be sequential at a lower abstraction level:

- $\Rightarrow$  e.g. Multi-processing system, which are executed on a single, sequential computing node

© 2020 Dave R. Zimmerman, The Australian National University page 173 of 758 (chapter 1: "Introduction to Concurrency" - up to page 202)

## Introduction to Concurrency

### Models and Terminology

### The concurrent programming abstraction

$\Rightarrow$  No interaction between concurrent system parts means that we can analyze them individually as pure sequential programs [end of course].

© 2020 Dave R. Zimmerman, The Australian National University page 176 of 758 (chapter 1: "Introduction to Concurrency" - up to page 202)

## Introduction to Concurrency

### Forms of concurrency

### Does concurrency lead to chaos?

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

- **non-deterministic** phenomena
- **non-observable** system states
- results may depend on more than just the input parameters and states at start time (timing, throughput, load, available resources, signals ... *throughout* the execution)
- **non-reproducible**  $\Rightarrow$  debugging?

Meaningful employment of concurrent systems features:

- non-determinism employed where the **underlying system is non-deterministic**
- non-determinism employed where the **actual execution sequence is meaningless**
- **synchronization** employed where adequate ... but only there

$\Rightarrow$  Control & monitor where required (and do it right), but not more ...

© 2020 Dave R. Zimmerman, The Australian National University page 171 of 758 (chapter 1: "Introduction to Concurrency" - up to page 202)

## Introduction to Concurrency

### Models and Terminology

### The concurrent programming abstraction

- "concurrent" is technically defined negatively as:  
If there is no observer who can identify two events as being in strict temporal sequence (i.e. one event has fully terminated before the other one starts up), then these two events are considered *concurrent*.
- "concurrent" in the context of programming and logic:  
"Concurrent programming abstraction is the study of interleaved execution sequences of the atomic instructions of sequential processes."  
(Ben-Ari)

© 2020 Dave R. Zimmerman, The Australian National University page 174 of 758 (chapter 1: "Introduction to Concurrency" - up to page 202)

## Introduction to Concurrency

### Models and Terminology

### The concurrent programming abstraction

$\Rightarrow$  No interaction between concurrent system parts means that we can analyze them individually as pure sequential programs [end of course].

$\Rightarrow$  Interaction occurs in form of:

- **Contention** (implicit interaction):  
Multiple concurrent execution units compete for one shared resource.
- **Communication** (explicit interaction):  
Explicit passing of information and/or explicit synchronization.

© 2020 Dave R. Zimmerman, The Australian National University page 177 of 758 (chapter 1: "Introduction to Concurrency" - up to page 202)

## Introduction to Concurrency

### Models and Terminology

### Concurrency on different abstraction levels/perspectives

$\Rightarrow$  Networks

- Large scale, high bandwidth interconnected nodes ("supercomputers")
- Networked computing nodes
- Standalone computing nodes – including local buses & interfaces sub-systems
- Operating systems (& distributed operating systems)

$\Rightarrow$  Implicit concurrency

$\Rightarrow$  Explicit concurrent programming (message passing and synchronization)

$\Rightarrow$  Assembler level concurrent programming

- Individual concurrent units inside one CPU
- Individual electronic circuits
- ...

© 2020 Dave R. Zimmerman, The Australian National University page 172 of 758 (chapter 1: "Introduction to Concurrency" - up to page 202)

## Introduction to Concurrency

### Models and Terminology

### The concurrent programming abstraction

**Concurrent program ::=**  
Multiple sequential programs (processes or threads) which are executed *concurrently*.

P.S. it is generally assumed that concurrent execution means that there is one execution unit (processor) per sequential program

- even though this is usually not technically correct, it is still an often valid, conservative assumption in the context of concurrent programming.

© 2020 Dave R. Zimmerman, The Australian National University page 175 of 758 (chapter 1: "Introduction to Concurrency" - up to page 202)

## Introduction to Concurrency

### Models and Terminology

### The concurrent programming abstraction

### Time-line or Sequence?

Consider time (durations) explicitly:

$\Rightarrow$  Real-time systems  $\Rightarrow$  join the appropriate courses

Consider the sequence of interaction points only:

$\Rightarrow$  Non-real-time systems  $\Rightarrow$  stay in your seat

© 2020 Dave R. Zimmerman, The Australian National University page 178 of 758 (chapter 1: "Introduction to Concurrency" - up to page 202)

Introduction to Concurrency  
 Models and Terminology  
 The concurrent programming abstraction  
 Correctness of concurrent non-real-time systems  
**[logical correctness]:**

- does *not* depend on clock speeds / execution times / delays
- does *not* depend on actual interleaving of concurrent processes

☞ holds true for **all possible sequences of interaction points (interleavings)**

© 2020 Dave R. Zimmerman, The Australian National University page 179 of 758 (chapter 1: "Introduction to Concurrency" - up to page 202)

Introduction to Concurrency  
 Models and Terminology  
 The concurrent programming abstraction  
 Correctness vs. testing in concurrent systems:

Slight changes in external triggers may (and usually does) result in completely different schedules (interleaving):

☞ Concurrent programs which depend in any way on external influences cannot be tested without modelling and embedding those influences into the test process.

☞ Designs which are provably correct with respect to the specification and are **independent** of the *actual timing behavior* are essential.

P.S. some timing restrictions for the scheduling still persist in non-real-time systems, e.g. 'fairness'

© 2020 Dave R. Zimmerman, The Australian National University page 180 of 758 (chapter 1: "Introduction to Concurrency" - up to page 202)

Introduction to Concurrency  
 Models and Terminology  
 The concurrent programming abstraction  
 Atomic operations:

Correctness proofs / designs in concurrent systems rely on the assumptions of 'Atomic operations' [detailed discussion later]:

- Complex and powerful atomic operations ease the correctness proofs, but may limit flexibility in the design
- Simple atomic operations are theoretically sufficient, but may lead to complex systems which correctness cannot be proven in practice.

© 2020 Dave R. Zimmerman, The Australian National University page 181 of 758 (chapter 1: "Introduction to Concurrency" - up to page 202)

Introduction to Concurrency  
 Models and Terminology  
 The concurrent programming abstraction

Standard concepts of correctness:

- **Partial correctness:**  
 $(P(I) \wedge \text{terminates}(\text{Program}(I, O))) \Rightarrow Q(I, O)$
- **Total correctness:**  
 $P(I) \Rightarrow (\text{terminates}(\text{Program}(I, O)) \wedge Q(I, O))$

where  $I, O$  are input and output sets,  
 $P$  is a property on the input set,  
 and  $Q$  is a relation between input and output sets

☞ do these concepts apply to and are sufficient for concurrent systems?

© 2020 Dave R. Zimmerman, The Australian National University page 182 of 758 (chapter 1: "Introduction to Concurrency" - up to page 202)

Introduction to Concurrency  
 Models and Terminology  
 The concurrent programming abstraction

Extended concepts of correctness in concurrent systems:  
 - Termination is often not intended or even considered a failure

**Safety properties:**  
 $(P(I) \wedge \text{Processes}(I, S)) \Rightarrow \Box Q(I, S)$   
 where  $\Box Q$  means that  $Q$  does always hold

**Liveness properties:**  
 $(P(I) \wedge \text{Processes}(I, S)) \Rightarrow \Diamond Q(I, S)$   
 where  $\Diamond Q$  means that  $Q$  does eventually hold (and will then stay true) and  $S$  is the current state of the concurrent system

© 2020 Dave R. Zimmerman, The Australian National University page 183 of 758 (chapter 1: "Introduction to Concurrency" - up to page 202)

Introduction to Concurrency  
 Models and Terminology  
 The concurrent programming abstraction

**Safety properties:**  
 $(P(I) \wedge \text{Processes}(I, S)) \Rightarrow \Box Q(I, S)$   
 where  $\Box Q$  means that  $Q$  does always hold

Examples:

- Mutual exclusion (no resource collisions)
- Absence of deadlocks (and other forms of 'silent death' and 'freeze' conditions)
- Specified responsiveness or free capabilities (typical in real-time / embedded systems or server applications)

© 2020 Dave R. Zimmerman, The Australian National University page 184 of 758 (chapter 1: "Introduction to Concurrency" - up to page 202)

Introduction to Concurrency  
 Models and Terminology  
 The concurrent programming abstraction

**Liveness properties:**  
 $(P(I) \wedge \text{Processes}(I, S)) \Rightarrow \Diamond Q(I, S)$   
 where  $\Diamond Q$  means that  $Q$  does eventually hold (and will then stay true) and  $S$  is the current state of the concurrent system

Examples:

- Requests need to complete eventually
- The state of the system needs to be displayed eventually
- No part of the system is to be delayed forever (fairness)

☞ Interesting *liveness* properties can be very hard to prove

© 2020 Dave R. Zimmerman, The Australian National University page 185 of 758 (chapter 1: "Introduction to Concurrency" - up to page 202)

Introduction to Concurrency  
 Introduction to processes and threads

**1 CPU per control-flow**

Specific configurations only, e.g.:

- Distributed pcontrollers.
- Physical process control systems:  
 1 cpu per task, connected via a bus-system.

☞ **Process management** (scheduling) not required.

☞ **Shared memory access** need to be coordinated.

© 2020 Dave R. Zimmerman, The Australian National University page 186 of 758 (chapter 1: "Introduction to Concurrency" - up to page 202)

Introduction to Concurrency  
 Introduction to processes and threads

**1 CPU for all control-flows**

- OS: emulate one CPU for every control-flow:  
 Multi-tasking operating system

☞ Support for memory protection essential.

☞ **Process management** (scheduling) required.

☞ **Shared memory access** need to be coordinated.

© 2020 Dave R. Zimmerman, The Australian National University page 187 of 758 (chapter 1: "Introduction to Concurrency" - up to page 202)

## Introduction to Concurrency

### Introduction to processes and threads

### Processes

Process ::= Address space + Control flow(s)

- Kernel has full knowledge about all processes as well as their states, requirements and currently held resources.

© 2020 Dave R. Zimmerman, The Australian National University page 188 of 758 chapter 1: "Introduction to Concurrency" up to page 202

## Introduction to Concurrency

### Introduction to processes and threads

### Threads

Threads (individual control-flows) can be handled:

- Inside the OS:**
  - Kernel scheduling.
  - Thread can easily be connected to external events (I/O).
- Outside the OS:**
  - User-level scheduling.
  - Threads may need to go through their parent process to access I/O.

© 2020 Dave R. Zimmerman, The Australian National University page 189 of 758 chapter 1: "Introduction to Concurrency" up to page 202

## Introduction to Concurrency

### Introduction to processes and threads

### Symmetric Multiprocessing (SMP)

All CPUs share the same physical address space (and access to resources).

- Any process / thread can be executed on any available CPU.

© 2020 Dave R. Zimmerman, The Australian National University page 190 of 758 chapter 1: "Introduction to Concurrency" up to page 202

## Introduction to Concurrency

### Introduction to processes and threads

### Processes ↔ Threads

Also processes can share memory and the specific definition of threads is different in different operating systems and contexts:

- Threads can be regarded as a group of processes, which share some resources (e.g. 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 communication can be more efficient than switching on 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.

© 2020 Dave R. Zimmerman, The Australian National University page 191 of 758 chapter 1: "Introduction to Concurrency" up to page 202

## Introduction to Concurrency

### Introduction to processes and threads

### Process Control Blocks

Process Control Blocks (PCBs)

- Process Id**
- Process state:** (created, ready, executing, blocked, suspended, bored ...)
- Scheduling attributes:** Priorities, deadlines, consumed CPU-time, ...
- CPU state:** Saved/restored information while context switches (incl. the program counter, stack pointer, ...)
- Memory attributes / privileges:** Memory base, limits, shared areas, ...
- Allocated resources / privileges:** Open and requested devices and files, ...

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

© 2020 Dave R. Zimmerman, The Australian National University page 192 of 758 chapter 1: "Introduction to Concurrency" up to page 202

## Introduction to Concurrency

### Process states

- 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

© 2020 Dave R. Zimmerman, The Australian National University page 193 of 758 chapter 1: "Introduction to Concurrency" up to page 202

## Introduction to Concurrency

### Process states

- 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 (none time critical processes)
  - waiting for main memory space (and other resources)

© 2020 Dave R. Zimmerman, The Australian National University page 194 of 758 chapter 1: "Introduction to Concurrency" up to page 202

## Introduction to Concurrency

### Process states

- 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 (none time critical processes)
  - waiting for main memory space (and other resources)

dispatching and suspending can now be independent modules

© 2020 Dave R. Zimmerman, The Australian National University page 195 of 758 chapter 1: "Introduction to Concurrency" up to page 202

## Introduction to Concurrency

### Process states

© 2020 Dave R. Zimmerman, The Australian National University page 196 of 758 chapter 1: "Introduction to Concurrency" up to page 202

## Introduction to Concurrency

### UNIX processes

*In UNIX systems tasks are created by 'cloning'*

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

resulting in a *duplication* of the *current* process

- ... returning '0' to the newly created process (the 'child' process)
- ... returning the **process id** of the child process to the creating process (the 'parent' process)
- ... or returning '-1' as C-style indication of a failure (in void of actual exception handling)

Frequent usage:

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

© 2010 Dave R. Ziminec, The Australian National University page 197 of 758 chapter 1: "Introduction to Concurrency" up to page 202.

## Introduction to Concurrency

### Concurrent programming languages

### Language candidates

☞ **Explicit concurrency**

- Ada, C++, Rust
- Chill
- Erlang
- Go
- Chapel, X10
- Occam, CSP
- All .net languages
- Java, Scala, Clojure
- Algol 68, Modula-2, Modula-3
- ...

☞ **Implicit (potential) concurrency**

- Lisp, Haskell, Caml, Miranda, and any other functional language
- Smalltalk, Squeak
- Prolog
- Esterel, Lustre, Signal

☞ **Wannabe concurrency**

- Ruby, Python [mostly broken due to global interpreter locks]

☞ **No support:**

- Eiffel, Pascal
- C
- Fortran, Cobol, Basic...

☞ **Libraries & interfaces (outside language definitions)**

- POSIX
- MPI (Message Passing Interface)
- ...

© 2010 Dave R. Ziminec, The Australian National University page 200 of 758 chapter 1: "Introduction to Concurrency" up to page 202.

## Introduction to Concurrency

### UNIX processes

### Communication between UNIX tasks ('pipes')

```
int data_pipe [2], c, rc;
if (pipe (data_pipe) == -1) {
  perror ("no pipe"); exit (1);
}
if (fork () == 0) {
  close (data_pipe [1]);
  while ((rc = read (data_pipe [0], &c, 1)) > 0) {
    putchar (c);
  }
  if (rc == -1) {
    perror ("pipe broken");
    close (data_pipe [0]);
    exit (1);
  }
  close (data_pipe [0]); exit (0);
} else {
  close (data_pipe [0]);
  while ((c = getchar ()) > 0) {
    if (write (data_pipe [1], &c, 1) == -1) {
      perror ("pipe broken");
      close (data_pipe [1]);
      exit (1);
    }
    close (data_pipe [1]);
    pid = wait ();
  }
  close (data_pipe [1]); exit (0);
}
```

© 2010 Dave R. Ziminec, The Australian National University page 198 of 758 chapter 1: "Introduction to Concurrency" up to page 202.

## Introduction to Concurrency

### Languages with implicit concurrency: e.g. functional programming

### Implicit concurrency in some programming schemes

Quicksort in a functional language (here: Haskell):

```
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y < x] ++ [x] ++ qsort [y | y <- xs, y > x]
```

Pure functional programming is **side-effect free**

☞ Parameters can be evaluated independently ☞ *could* run concurrently

Some functional languages allow for **lazy evaluation**, i.e. sub-expressions are not necessarily evaluated completely:

```
borderline = (n /= 0) && (g (n) > h (n))
```

☞ If n equals zero then the evaluation of g(n) and h(n) can be stopped (or not even be started).

☞ Concurrent program parts **should be interruptible** in this case.

**Short-circuit evaluations** in imperative languages assume explicit sequential execution:

```
if Pointer /= nil and then Pointer.next = nil then ...
```

© 2010 Dave R. Ziminec, The Australian National University page 201 of 758 chapter 1: "Introduction to Concurrency" up to page 202.

## Introduction to Concurrency

### Concurrent programming languages

### Requirement

- Concept of tasks, threads or other potentially concurrent entities

### Frequently requested essential elements

- Support for **management** of concurrent entities (create, terminate, ...)
- Support for **contention management** (mutual exclusion, ...)
- Support for **synchronization** (semaphores, monitors, ...)
- Support for **communication** (message passing, shared memory, rpc ...)
- Support for **protection** (tasks, memory, devices, ...)

© 2010 Dave R. Ziminec, The Australian National University page 199 of 758 chapter 1: "Introduction to Concurrency" up to page 202.

## Introduction to Concurrency

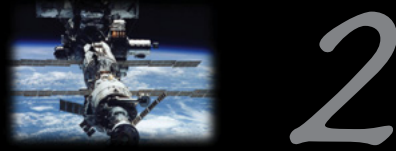
### Summary

### Concurrency – The Basic Concepts

- **Forms of concurrency**
- **Models and terminology**
  - Abstractions and perspectives: computer science, physics & engineering
  - Observations: non-determinism, atomicity, interaction, interleaving
  - Correctness in concurrent systems
- **Processes and threads**
  - Basic concepts and notions
  - Process states
- **Concurrent programming languages:**
  - Explicit concurrency: e.g. Ada, Chapel
  - Implicit concurrency: functional programming – e.g. Haskell, Caml

© 2010 Dave R. Ziminec, The Australian National University page 202 of 758 chapter 1: "Introduction to Concurrency" up to page 202.





## Mutual Exclusion

Uwe R. Zimmer - The Australian National University

## Mutual Exclusion

## References for this chapter

[Ben-Ari06]  
M. Ben-Ari  
*Principles of Concurrent and Distributed Programming*  
2006, second edition, Prentice-Hall, ISBN 0-13-711821-X

## Mutual Exclusion

## Problem specification

## The general mutual exclusion scenario

- $N$  processes execute (infinite) instruction sequences concurrently. Each instruction belongs to either a *critical* or *non-critical* section.
- ☞ Safety property 'Mutual exclusion':  
Instructions from *critical sections* of two or more processes must never be interleaved!
- More required properties:
  - **No deadlocks:** If one or multiple processes try to enter their critical sections then *exactly one* of them *must succeed*.
  - **No starvation:** Every process which tries to enter one of his critical sections *must succeed eventually*.
  - **Efficiency:** The decision which process may enter the critical section must be made *efficiently* in all cases, i.e. also when there is no contention in the first place.

## Mutual Exclusion

## Problem specification

## The general mutual exclusion scenario

- $N$  processes execute (infinite) instruction sequences concurrently. Each instruction belongs to either a *critical* or *non-critical* section.
- ☞ Safety property 'Mutual exclusion':  
Instructions from *critical sections* of two or more processes must never be interleaved!
- Further assumptions:
  - Pre- and post-protocols *can be executed* before and after each critical section.
  - Processes may *delay infinitely* in *non-critical* sections.
  - Processes do *not delay infinitely* in *critical* sections.

## Mutual Exclusion

## Mutual exclusion: Atomic load &amp; store operations

## Atomic load &amp; store operations

- ☞ Assumption 1: every individual base memory cell (word) load and store access is *atomic*
- ☞ Assumption 2: there is *no* atomic combined load-store access

G : Natural := 0; -- assumed to be mapped on a 1-word cell in memory

```

task body P1 is          task body P2 is          task body P3 is
begin
  G := 1;                G := 2;                G := 3;
  G := G + G;            G := G + G;            G := G + G;
end P1;                  end P2;                  end P3;

```

- ☞ What is the value of G?

## Mutual Exclusion

## Mutual exclusion: Atomic load &amp; store operations

## Atomic load &amp; store operations

- ☞ Assumption 1: every individual base memory cell (word) load and store access is *atomic*
- ☞ Assumption 2: there is *no* atomic combined load-store access

G : Natural := 0; -- assumed to be mapped on a 1-word cell in memory

```

task body P1 is          task body P2 is          task body P3 is
begin
  G := 1;                G := 2;                G := 3;
  G := G + G;            G := G + G;            G := G + G;
end P1;                  end P2;                  end P3;

```

- ☞ After the first global initialisation, G can have **almost any** value between 0 and 24
- ☞ After the first global initialisation, G will have **exactly one** value between 0 and 24
- ☞ After all tasks terminated, G will have **exactly one** value between 2 and 24

## Mutual Exclusion

## Mutual exclusion: First attempt

```

type Task_Token is mod 2;
Turn: Task_Token := 0;
task body P0 is          task body P1 is
begin                    begin
  loop                    loop
    ----- non_critical_section_0;
    loop exit when Turn = 0; end loop;
    ----- critical_section_0;
    Turn := Turn + 1;
  end loop;
end P0;                  end P1;

```

- ☞ Mutual exclusion?
- ☞ Deadlock?
- ☞ Starvation?
- ☞ Work without contention?

## Mutual Exclusion

## Mutual exclusion: First attempt

```

type Task_Token is mod 2;
Turn: Task_Token := 0;
task body P0 is          task body P1 is
begin                    begin
  loop                    loop
    ----- non_critical_section_0;
    loop exit when Turn = 0; end loop;
    ----- critical_section_0;
    Turn := Turn + 1;
  end loop;
end P0;                  end P1;

```

- ☞ Mutual exclusion!
- ☞ No deadlock!
- ☞ No starvation!
- ☞ Locks up, if there is no contention!

## Mutual Exclusion

## Mutual exclusion: First attempt

```

type Task_Token is mod 2;
Turn: Task_Token := 0;
task body P0 is          task body P1 is
begin                    begin
  loop                    loop
    ----- non_critical_section_0;
    loop exit when Turn = 0; end loop;
    ----- critical_section_0;
    Turn := Turn + 1;
  end loop;
end P0;                  end P1;

```

- ☞ Mutual exclusion!
  - ☞ No deadlock!
  - ☞ No starvation!
  - ☞ Inefficient!
- scatter:  
if Turn = myTurn then  
  Turn := Turn + 1;  
end if  
into the non-critical sections

### Mutual Exclusion

#### Mutual exclusion: Second attempt

```

type Critical_Section_State is (In_CS, Out_CS);
C1, C2: Critical_Section_State := Out_CS;

task body P1 is
begin
loop
----- non_critical_section_1;
loop
exit when C2 = Out_CS;
end loop;
C1 := In_CS;
----- critical_section_1;
C1 := Out_CS;
end loop;
end P1;

task body P2 is
begin
loop
----- non_critical_section_2;
loop
exit when C1 = Out_CS;
end loop;
C2 := In_CS;
----- critical_section_2;
C2 := Out_CS;
end loop;
end P2;

```

Any better?

© 2020 Dave R. Zimmerman, The Australian National University page 212 of 758 chapter 2: "Mutual Exclusion" up to page 233

### Mutual Exclusion

#### Mutual exclusion: Second attempt

```

type Critical_Section_State is (In_CS, Out_CS);
C1, C2: Critical_Section_State := Out_CS;

task body P1 is
begin
loop
----- non_critical_section_1;
loop
exit when C2 = Out_CS;
end loop;
C1 := In_CS;
----- critical_section_1;
C1 := Out_CS;
end loop;
end P1;

task body P2 is
begin
loop
----- non_critical_section_2;
loop
exit when C1 = Out_CS;
end loop;
C2 := In_CS;
----- critical_section_2;
C2 := Out_CS;
end loop;
end P2;

```

No mutual exclusion!

© 2020 Dave R. Zimmerman, The Australian National University page 213 of 758 chapter 2: "Mutual Exclusion" up to page 233

### Mutual Exclusion

#### Mutual exclusion: Third attempt

```

type Critical_Section_State is (In_CS, Out_CS);
C1, C2: Critical_Section_State := Out_CS;

task body P1 is
begin
loop
----- non_critical_section_1;
C1 := In_CS;
loop
exit when C2 = Out_CS;
end loop;
----- critical_section_1;
C1 := Out_CS;
end loop;
end P1;

task body P2 is
begin
loop
----- non_critical_section_2;
C2 := In_CS;
loop
exit when C1 = Out_CS;
end loop;
----- critical_section_2;
C2 := Out_CS;
end loop;
end P2;

```

Any better?

© 2020 Dave R. Zimmerman, The Australian National University page 214 of 758 chapter 2: "Mutual Exclusion" up to page 233

### Mutual Exclusion

#### Mutual exclusion: Third attempt

```

type Critical_Section_State is (In_CS, Out_CS);
C1, C2: Critical_Section_State := Out_CS;

task body P1 is
begin
loop
----- non_critical_section_1;
C1 := In_CS;
loop
exit when C2 = Out_CS;
end loop;
----- critical_section_1;
C1 := Out_CS;
end loop;
end P1;

task body P2 is
begin
loop
----- non_critical_section_2;
C2 := In_CS;
loop
exit when C1 = Out_CS;
end loop;
----- critical_section_2;
C2 := Out_CS;
end loop;
end P2;

```

Mutual exclusion!  
Potential deadlock!

© 2020 Dave R. Zimmerman, The Australian National University page 215 of 758 chapter 2: "Mutual Exclusion" up to page 233

### Mutual Exclusion

#### Mutual exclusion: Forth attempt

```

type Critical_Section_State is (In_CS, Out_CS);
C1, C2: Critical_Section_State := Out_CS;

task body P1 is
begin
loop
----- non_critical_section_1;
C1 := In_CS;
loop
exit when C2 = Out_CS;
C1 := Out_CS; C1 := In_CS;
end loop;
----- critical_section_1;
C1 := Out_CS;
end loop;
end P1;

task body P2 is
begin
loop
----- non_critical_section_2;
C2 := In_CS;
loop
exit when C1 = Out_CS;
C2 := Out_CS; C2 := In_CS;
end loop;
----- critical_section_2;
C2 := Out_CS;
end loop;
end P2;

```

Making any progress?

© 2020 Dave R. Zimmerman, The Australian National University page 216 of 758 chapter 2: "Mutual Exclusion" up to page 233

### Mutual Exclusion

#### Mutual exclusion: Forth attempt

```

type Critical_Section_State is (In_CS, Out_CS);
C1, C2: Critical_Section_State := Out_CS;

task body P1 is
begin
loop
----- non_critical_section_1;
C1 := In_CS;
loop
exit when C2 = Out_CS;
C1 := Out_CS; C1 := In_CS;
end loop;
----- critical_section_1;
C1 := Out_CS;
end loop;
end P1;

task body P2 is
begin
loop
----- non_critical_section_2;
C2 := In_CS;
loop
exit when C1 = Out_CS;
C2 := Out_CS; C2 := In_CS;
end loop;
----- critical_section_2;
C2 := Out_CS;
end loop;
end P2;

```

Mutual exclusion! No Deadlock!  
Potential starvation! Potential global livelock!

© 2020 Dave R. Zimmerman, The Australian National University page 217 of 758 chapter 2: "Mutual Exclusion" up to page 233

### Mutual Exclusion

#### Mutual exclusion: Decker's Algorithm

```

type Task_Range is mod 2;
type Critical_Section_State is (In_CS, Out_CS);
CSS : array (Task_Range) of Critical_Section_State := (others => Out_CS);
Turn : Task_Range := Task_Range'First;

task type One_Of_Two_Tasks
(this_Task : Task_Range);

task body One_Of_Two_Tasks is
other_Task : Task_Range := this_Task + 1;
begin
----- non_critical_section
CSS (this_Task) := In_CS;
loop
exit when
CSS (other_Task) = Out_CS;
if Turn = other_Task then
CSS (this_Task) := Out_CS;
loop
exit when Turn = this_Task;
end loop;
CSS (this_Task) := In_CS;
end if;
----- critical section
CSS (this_Task) := Out_CS;
Turn := other_Task;
end One_Of_Two_Tasks;

```

© 2020 Dave R. Zimmerman, The Australian National University page 218 of 758 chapter 2: "Mutual Exclusion" up to page 233

### Mutual Exclusion

#### Mutual exclusion: Decker's Algorithm

```

type Task_Range is mod 2;
type Critical_Section_State is (In_CS, Out_CS);
CSS : array (Task_Range) of Critical_Section_State := (others => Out_CS);
Turn : Task_Range := Task_Range'First;

task type One_Of_Two_Tasks
(this_Task : Task_Range);

task body One_Of_Two_Tasks is
other_Task : Task_Range := this_Task + 1;
begin
----- non_critical_section
CSS (this_Task) := In_CS;
loop
exit when
CSS (other_Task) = Out_CS;
if Turn = other_Task then
CSS (this_Task) := Out_CS;
loop
exit when Turn = this_Task;
end loop;
CSS (this_Task) := In_CS;
end if;
----- critical section
CSS (this_Task) := Out_CS;
Turn := other_Task;
end One_Of_Two_Tasks;

```

Two tasks only!  
Mutual exclusion! No starvation!  
No deadlock! No livelock!

© 2020 Dave R. Zimmerman, The Australian National University page 219 of 758 chapter 2: "Mutual Exclusion" up to page 233

### Mutual Exclusion

#### Mutual exclusion: Peterson's Algorithm

```

type Task_Range is mod 2;
type Critical_Section_State is (In_CS, Out_CS);
CSS : array (Task_Range) of Critical_Section_State := (others => Out_CS);
Last : Task_Range := Task_Range'First;

task type One_Of_Two_Tasks
(this_Task : Task_Range);

task body One_Of_Two_Tasks is
other_Task : Task_Range := this_Task + 1;
begin
----- non_critical_section
Last := this_Task;
loop
exit when
CSS (other_Task) = Out_CS
or else Last /= this_Task;
end loop;
----- critical section
CSS (this_Task) := Out_CS;
end One_Of_Two_Tasks;

```

© 2020 Dave R. Zimmerman, The Australian National University page 220 of 758 chapter 2: "Mutual Exclusion" up to page 233

## Mutual Exclusion

### Mutual exclusion: Peterson's Algorithm

Two tasks only!

```

type Task_Range is mod 2;
type Critical_Section_State is (In_CS, Out_CS);
CSS : array (Task_Range) of Critical_Section_State := (others => Out_CS);
Last : Task_Range := Task_Range'First;

task type One_Of_Two_Tasks
  (this_Task : Task_Range);

task body One_Of_Two_Tasks is
  other_Task : Task_Range := this_Task + 1;
begin
  ----- non_critical_section
  ----- Mutual exclusion!  ----- No starvation!
  ----- No deadlock!     ----- No livelock!
  CSS (this_Task) := In_CS;
  Last := this_Task;
  loop
    exit when
      CSS (other_Task) = Out_CS
      or else Last /= this_Task;
  end loop;
  ----- critical_section
  CSS (this_Task) := Out_CS;
end One_Of_Two_Tasks;

```

© 2020 Dave R. Zimine, The Australian National University page 221 of 758 chapter 2: "Mutual Exclusion" up to page 233

## Mutual Exclusion

### Mutual exclusion: Bakery Algorithm

```

No_Of_Tasks : constant Positive := ...;
type Task_Range is mod No_Of_Tasks;
Choosing : array (Task_Range) of Boolean := (others => False);
Ticket : array (Task_Range) of Natural := (others => 0);

task type P (this_id: Task_Range);
task body P is
begin
  loop
    ----- non_critical_section_1;
    Choosing (this_id) := True;
    Ticket (this_id) := Max (Ticket) + 1;
    Choosing (this_id) := False;
    for id in Task_Range loop
      if id /= this_id then
        loop
          exit when not Choosing (id);
        end loop;
      end loop;
    end if;
    ----- critical_section_1;
    Ticket (this_id) := 0;
  end loop;
end P;

```

© 2020 Dave R. Zimine, The Australian National University page 224 of 758 chapter 2: "Mutual Exclusion" up to page 233

## Mutual Exclusion

### Mutual exclusion: atomic test-and-set operation

```

type Flag is Natural range 0..1; C : Flag := 0;

task body Pi is
  L : Flag;
begin
  loop
    [L := C; C := 1];
    exit when L = 0;
    ----- change process
  end loop;
  ----- critical_section_i;
  C := 0;
end loop;
end Pi;

task body Pj is
  L : Flag;
begin
  loop
    [L := C; C := 1];
    exit when L = 0;
    ----- change process
  end loop;
  ----- critical_section_j;
  C := 0;
end loop;
end Pj;

```

Does that work?

© 2020 Dave R. Zimine, The Australian National University page 227 of 758 chapter 2: "Mutual Exclusion" up to page 233

## Mutual Exclusion

### Problem specification

### The general mutual exclusion scenario

- N processes execute (infinite) instruction sequences concurrently. Each instruction belongs to either a *critical* or *non-critical* section.
- Safety property 'Mutual exclusion': Instructions from *critical sections* of two or more processes must never be interleaved!
- More required properties:
  - No deadlocks: If one or multiple processes try to enter their critical sections then exactly one of them must succeed.
  - No starvation: Every process which tries to enter one of his critical sections must succeed eventually.
  - Efficiency: The decision which process may enter the critical section must be made *efficiently* in all cases, i.e. also when there is no contention.

© 2020 Dave R. Zimine, The Australian National University page 222 of 758 chapter 2: "Mutual Exclusion" up to page 233

## Mutual Exclusion

### Mutual exclusion: Bakery Algorithm

```

No_Of_Tasks : constant Positive := ...;
type Task_Range is mod No_Of_Tasks;
Choosing : array (Task_Range) of Boolean := (others => False);
Ticket : array (Task_Range) of Natural := (others => 0);

task type P (this_id: Task_Range);
task body P is
begin
  loop
    ----- non_critical_section_1;
    Choosing (this_id) := True;
    Ticket (this_id) := Max (Ticket) + 1;
    Choosing (this_id) := False;
    for id in Task_Range loop
      if id /= this_id then
        loop
          exit when not Choosing (id);
        end loop;
      end loop;
    end if;
    ----- critical_section_1;
    Ticket (this_id) := 0;
  end loop;
end P;

```

Extensive and communication intensive protocol (even if there is no contention)

Works for N processes!

© 2020 Dave R. Zimine, The Australian National University page 225 of 758 chapter 2: "Mutual Exclusion" up to page 233

## Mutual Exclusion

### Mutual exclusion: atomic test-and-set operation

```

type Flag is Natural range 0..1; C : Flag := 0;

task body Pi is
  L : Flag;
begin
  loop
    [L := C; C := 1];
    exit when L = 0;
    ----- change process
  end loop;
  ----- critical_section_i;
  C := 0;
end loop;
end Pi;

task body Pj is
  L : Flag;
begin
  loop
    [L := C; C := 1];
    exit when L = 0;
    ----- change process
  end loop;
  ----- critical_section_j;
  C := 0;
end loop;
end Pj;

```

Mutual exclusion!, No deadlock!, No global live-lock!

Works for any dynamic number of processes.

Individual starvation possible! Busy waiting loops!

© 2020 Dave R. Zimine, The Australian National University page 228 of 758 chapter 2: "Mutual Exclusion" up to page 233

## Mutual Exclusion

### Mutual exclusion: Bakery Algorithm

### The idea of the Bakery Algorithm

A set of N Processes  $P_1, \dots, P_N$  competing for mutually exclusive execution of their critical regions. Every process  $P_i$  out of  $P_1, \dots, P_N$  supplies: a globally readable number  $t_i$  ("ticket") (initialized to '0').

- Before a process  $P_i$  enters a critical section:
  - $P_i$  draws a new number  $t_i > t_j, \forall j \neq i$
  - $P_i$  is allowed to enter the critical section iff:  $\forall j \neq i: t_j < t_i$  or  $t_j = 0$
- After a process left a critical section:
  - $P_i$  resets its  $t_j = 0$

Issues:

- Can you ensure that processes won't read each others ticket numbers while still calculating?
- Can you ensure that no two processes draw the same number?

© 2020 Dave R. Zimine, The Australian National University page 226 of 758 chapter 2: "Mutual Exclusion" up to page 233

## Mutual Exclusion

### Beyond atomic memory access

### Realistic hardware support

Atomic test-and-set operations:

- [L := C; C := 1]

Atomic exchange operations:

- [Temp := L; L := C; C := Temp]

Memory cell reservations:

- $L := \overset{\ominus}{C}$ ; - read by using a *special instruction*, which puts a 'reservation' on C
- ... calculate a <new value> for C ...
- $C := \overset{\ominus}{\text{<new value>}}$ ; - succeeds iff C was not manipulated by other processors or devices since the reservation

© 2020 Dave R. Zimine, The Australian National University page 226 of 758 chapter 2: "Mutual Exclusion" up to page 233

## Mutual Exclusion

### Mutual exclusion: atomic exchange operation

```

type Flag is Natural range 0..1; C : Flag := 0;

task body Pi is
  L : Flag := 1;
begin
  loop
    [Temp := L; L := C; C := Temp];
    exit when L = 0;
    ----- change process
  end loop;
  ----- critical_section_i;
  L := 1; C := 0;
end loop;
end Pi;

task body Pj is
  L : Flag := 1;
begin
  loop
    [Temp := L; L := C; C := Temp];
    exit when L = 0;
    ----- change process
  end loop;
  ----- critical_section_j;
  L := 1; C := 0;
end loop;
end Pj;

```

Does that work?

© 2020 Dave R. Zimine, The Australian National University page 229 of 758 chapter 2: "Mutual Exclusion" up to page 233

### Mutual Exclusion

#### Mutual exclusion: atomic exchange operation

```

type Flag is Natural range 0..1; C : Flag := 0;

task body Pi is
  L : Flag := 1;
begin
  loop
    [Temp := L; L := C; C := Temp];
    exit when L = 0;
    ----- change process
  end loop;
  ----- critical_section_i;
  L := 1; C := 0;
end loop;
end Pi;

task body Pj is
  L : Flag := 1;
begin
  loop
    [Temp := L; L := C; C := Temp];
    exit when L = 0;
    ----- change process
  end loop;
  ----- critical_section_j;
  L := 1; C := 0;
end loop;
end Pj;

```

- ☞ Mutual exclusion!, No deadlock!, No global live-lock!
- ☞ Works for any dynamic number of processes.
- ☞ Individual starvation possible! Busy waiting loops!

© 2010 Dave R. Ziminec, The Australian National University page 210 of 758 chapter 2: "Mutual Exclusion" up to page 233

### Mutual Exclusion

#### Mutual exclusion ... or the lack thereof

```

Count : Integer := 0;

task body Enter is
begin
  for i := 1 .. 100 loop
    Count := Count + 1;
  end loop;
end Enter;

task body Leave is
begin
  for i := 1 .. 100 loop
    Count := Count - 1;
  end loop;
end Leave;

```

☞ What is the value of Count after both programs complete?

© 2010 Dave R. Ziminec, The Australian National University page 213 of 758 chapter 2: "Mutual Exclusion" up to page 233

```

Count: .word 0x00000000
Lock: .word 0x00000000 ; #0 means unlocked

ldr r3, =Lock
ldr r4, =Count
mov r1, #1

for_enter:
  cmp r1, #100
  bgt end_for_enter

fail_enter:
  ldr r0, [r3]
  cbnz r0, fail_enter ; if locked
  mov r0, #1 ; lock value
  str r0, [r3] ; lock

  ldr r2, [r4]
  add r2, #1
  str r2, [r4]

  add r1, #1
  b for_enter
end_for_enter:

```

```

for_leave:
  cmp r1, #100
  bgt end_for_leave

fail_leave:
  ldr r0, [r3]
  cbnz r0, fail_leave ; if locked
  mov r0, #1 ; lock value
  str r0, [r3] ; lock

  ldr r2, [r4]
  sub r2, #1
  str r2, [r4]

  add r1, #1
  b for_leave
end_for_leave:

```

© 2010 Dave R. Ziminec, The Australian National University page 216 of 758 chapter 2: "Mutual Exclusion" up to page 233

### Mutual Exclusion

#### Mutual exclusion: memory cell reservation

```

type Flag is Natural range 0..1; C : Flag := 0;

task body Pi is
  L : Flag;
begin
  loop
    L := C; C := 1;
    exit when Untouched and L = 0;
    ----- change process
  end loop;
  ----- critical_section_i;
  C := 0;
end loop;
end Pi;

task body Pj is
  L : Flag;
begin
  loop
    L := C; C := 1;
    exit when Untouched and L = 0;
    ----- change process
  end loop;
  ----- critical_section_j;
  C := 0;
end loop;
end Pj;

```

☞ Does that work?

© 2010 Dave R. Ziminec, The Australian National University page 211 of 758 chapter 2: "Mutual Exclusion" up to page 233

```

Count: .word 0x00000000

ldr r4, =Count
mov r1, #1

for_enter:
  cmp r1, #100
  bgt end_for_enter

  ldr r2, [r4]
  add r2, #1
  str r2, [r4]

  add r1, #1
  b for_enter
end_for_enter:

```

Negotiate who goes first

```

ldr r4, =Count
mov r1, #1

for_leave:
  cmp r1, #100
  bgt end_for_leave

  ldr r2, [r4]
  sub r2, #1
  str r2, [r4]

  add r1, #1
  b for_leave
end_for_leave:

```

Indicate critical section completed

© 2010 Dave R. Ziminec, The Australian National University page 214 of 758 chapter 2: "Mutual Exclusion" up to page 233

```

Count: .word 0x00000000
Lock: .word 0x00000000 ; #0 means unlocked

ldr r3, =Lock
ldr r4, =Count
mov r1, #1

for_enter:
  cmp r1, #100
  bgt end_for_enter

fail_enter:
  ldrex r0, [r3]
  cbnz r0, fail_enter ; if locked
  mov r0, #1 ; lock value
  strex r0, [r3] ; try lock
  cbnz r0, fail_enter ; if touched
  dmb ; sync memory

  ldr r2, [r4]
  add r2, #1
  str r2, [r4]

  add r1, #1
  b for_enter
end_for_enter:

```

Any context switch needs to clear reservations

```

for_leave:
  cmp r1, #100
  bgt end_for_leave

fail_leave:
  ldrex r0, [r3]
  cbnz r0, fail_leave ; if locked
  mov r0, #1 ; lock value
  strex r0, [r3] ; try lock
  cbnz r0, fail_leave ; if touched
  dmb ; sync memory

  ldr r2, [r4]
  sub r2, #1
  str r2, [r4]

  add r1, #1
  b for_leave
end_for_leave:

```

© 2010 Dave R. Ziminec, The Australian National University page 217 of 758 chapter 2: "Mutual Exclusion" up to page 233

### Mutual Exclusion

#### Mutual exclusion: memory cell reservation

```

type Flag is Natural range 0..1; C : Flag := 0;

task body Pi is
  L : Flag;
begin
  loop
    L := C; C := 1;
    exit when Untouched and L = 0;
    ----- change process
  end loop;
  ----- critical_section_i;
  C := 0;
end loop;
end Pi;

task body Pj is
  L : Flag;
begin
  loop
    L := C; C := 1;
    exit when Untouched and L = 0;
    ----- change process
  end loop;
  ----- critical_section_j;
  C := 0;
end loop;
end Pj;

```

Any context switch needs to clear reservations

☞ Mutual exclusion!, No deadlock!, No global live-lock!

☞ Works for any dynamic number of processes.

☞ Individual starvation possible! Busy waiting loops!

© 2010 Dave R. Ziminec, The Australian National University page 212 of 758 chapter 2: "Mutual Exclusion" up to page 233

```

Count: .word 0x00000000
Lock: .word 0x00000000 ; #0 means unlocked

ldr r3, =Lock
ldr r4, =Count
mov r1, #1

for_enter:
  cmp r1, #100
  bgt end_for_enter

fail_enter:
  ldr r0, [r3]
  cbnz r0, fail_enter ; if locked

  ldr r2, [r4]
  add r2, #1
  str r2, [r4]

  add r1, #1
  b for_enter
end_for_enter:

```

Any context switch needs to clear reservations

```

for_leave:
  cmp r1, #100
  bgt end_for_leave

fail_leave:
  ldr r0, [r3]
  cbnz r0, fail_leave ; if locked

  ldr r2, [r4]
  sub r2, #1
  str r2, [r4]

  add r1, #1
  b for_leave
end_for_leave:

```

© 2010 Dave R. Ziminec, The Australian National University page 215 of 758 chapter 2: "Mutual Exclusion" up to page 233

```

Count: .word 0x00000000
Lock: .word 0x00000000 ; #0 means unlocked

ldr r3, =Lock
ldr r4, =Count
mov r1, #1

for_enter:
  cmp r1, #100
  bgt end_for_enter

fail_enter:
  ldrex r0, [r3]
  cbnz r0, fail_enter ; if locked
  mov r0, #1 ; lock value
  strex r0, [r3] ; try lock
  cbnz r0, fail_enter ; if touched
  dmb ; sync memory

  ldr r2, [r4]
  add r2, #1
  str r2, [r4]

  dmb ; sync memory
  mov r0, #0 ; unlock value
  str r0, [r3] ; unlock

  add r1, #1
  b for_enter
end_for_enter:

```

Any context switch needs to clear reservations

```

for_leave:
  cmp r1, #100
  bgt end_for_leave

fail_leave:
  ldrex r0, [r3]
  cbnz r0, fail_leave ; if locked
  mov r0, #1 ; lock value
  strex r0, [r3] ; try lock
  cbnz r0, fail_leave ; if touched
  dmb ; sync memory

  ldr r2, [r4]
  sub r2, #1
  str r2, [r4]

  dmb ; sync memory
  mov r0, #0 ; unlock value
  str r0, [r3] ; unlock

  add r1, #1
  b for_leave
end_for_leave:

```

© 2010 Dave R. Ziminec, The Australian National University page 218 of 758 chapter 2: "Mutual Exclusion" up to page 233

```

Count: .word 0x00000000
Lock: .word 0x00000000 ; #0 means unlocked

ldr r3, =Lock
ldr r4, =Count
mov r1, #1

for_enter:
cmp r1, #100
bgt end_for_enter

fail_enter:
ldrex r0, [r3]
cbnz r0, fail_enter ; if locked
mov r0, #1 ; lock value
strex r0, [r3] ; try lock
cbnz r0, fail_enter ; if touched
dmb ; sync memory

ldr r2, [r4]
add r2, #1
str r2, [r4]

dmb ; sync memory
mov r0, #0 ; unlock value
str r0, [r3] ; unlock
add r1, #1
b for_enter
end_for_enter:

for_leave:
ldr r3, =Lock
ldr r4, =Count
mov r1, #1

fail_leave:
ldrex r0, [r3]
cbnz r0, fail_leave ; if locked
mov r0, #1 ; lock value
strex r0, [r3] ; try lock
cbnz r0, fail_leave ; if touched
dmb ; sync memory

ldr r2, [r4]
ldr r2, [r4]
sub r2, #1
str r2, [r4]

dmb ; sync memory
mov r0, #0 ; unlock value
str r0, [r3] ; unlock
add r1, #1
b for_leave
end_for_leave:

```

Any context switch needs to clear reservations

Asks for permission

Critical section

Critical section

© 2020 Dave R. Zimenc, The Australian National University page 239 of 758 chapter 2: "Mutual Exclusion" (up to page 233)

### Mutual Exclusion

#### Beyond atomic hardware operations

## Semaphores

... as supplied by operating systems and runtime environments

- a set of processes  $P_1, \dots, P_N$  agree on a variable  $S$  operating as a flag to indicate synchronization conditions
- an atomic operation Wait on  $S$ : (aka 'Suspend\_Until\_True', 'sem\_wait', ...)
 

Process  $P_i$ : Wait (S):

```
[if S > 0 then S := S - 1
else suspend  $P_i$  on S]
```
- an atomic operation Signal on  $S$ : (aka 'Set\_True', 'sem\_post', ...)
 

Process  $P_i$ : Signal (S):

```
[if  $\exists P_j$  suspended on S then release  $P_j$ 
else S := S + 1]
```

then the variable  $S$  is called a Semaphore in a scheduling environment.

© 2020 Dave R. Zimenc, The Australian National University page 242 of 758 chapter 2: "Mutual Exclusion" (up to page 233)

```

Count: .word 0x00000000
Sema: .word 0x00000001

ldr r3, =Sema
ldr r4, =Count
mov r1, #1

for_enter:
cmp r1, #100
bgt end_for_enter

wait_1:
ldr r0, [r3]
cbz r0, wait_1 ; if Semaphore = 0
sub r0, #1 ; dec Semaphore
str r0, [r3] ; update

...

Critical section

add r1, #1
b for_enter
end_for_enter:

for_leave:
ldr r3, =Sema
ldr r4, =Count
mov r1, #1

wait_2:
ldr r0, [r3]
cbz r0, wait_2 ; if Semaphore = 0
sub r0, #1 ; dec Semaphore
str r0, [r3] ; update

...

Critical section

add r1, #1
b for_leave
end_for_leave:

```

Critical section

Critical section

© 2020 Dave R. Zimenc, The Australian National University page 245 of 758 chapter 2: "Mutual Exclusion" (up to page 233)

### Mutual Exclusion

#### Beyond atomic hardware operations

## Semaphores

Basic definition (Dijkstra 1968)

Assuming the following three conditions on a shared memory cell between processes:

- a set of processes agree on a variable  $S$  operating as a flag to indicate synchronization conditions
- an atomic operation P on  $S$  – for 'passen' (Dutch for 'pass'):

$P(S): [as\ soon\ as\ S > 0\ then\ S := S - 1]$  then this is a potentially delaying operation

- an atomic operation V on  $S$  – for 'vr'ygeven' (Dutch for 'to release'):

$V(S): [S := S + 1]$

then the variable  $S$  is called a Semaphore.

Light weight solution – sometimes referred to as "lock-free" or "lockless".

```

Count: .word 0x00000000

ldr r4, =Count
mov r1, #1

for_enter:
cmp r1, #100
bgt end_for_enter

enter_strex_fail:
ldrex r2, [r4] ; tag [r4] as exclusive
add r2, #1
strex r2, [r4] ; only if untouched
cbnz r2, enter_strex_fail

add r1, #1
b for_enter
end_for_enter:

ldr r4, =Count
mov r1, #1

for_leave:
cmp r1, #100
bgt end_for_leave

leave_strex_fail:
ldrex r2, [r4] ; tag [r4] as exclusive
sub r2, #1
strex r2, [r4] ; only if untouched
cbnz r2, leave_strex_fail

add r1, #1
b for_leave
end_for_leave:

```

Any context switch needs to clear reservations

Asks for forgiveness

© 2020 Dave R. Zimenc, The Australian National University page 240 of 758 chapter 2: "Mutual Exclusion" (up to page 233)

### Mutual Exclusion

#### Beyond atomic hardware operations

## Semaphores

### Types of semaphores:

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

All types of semaphores must be initialized: often the number of processes which are allowed inside a critical section, i.e. '1'.

© 2020 Dave R. Zimenc, The Australian National University page 243 of 758 chapter 2: "Mutual Exclusion" (up to page 233)

```

Count: .word 0x00000000
Sema: .word 0x00000001

ldr r3, =Sema
ldr r4, =Count
mov r1, #1

for_enter:
cmp r1, #100
bgt end_for_enter

wait_1:
ldrex r0, [r3]
cbz r0, wait_1 ; if Semaphore = 0
sub r0, #1 ; dec Semaphore
strex r0, [r3] ; try update
cbnz r0, wait_1 ; if touched
dmb ; sync memory

...

Critical section

add r1, #1
b for_enter
end_for_enter:

for_leave:
ldr r3, =Sema
ldr r4, =Count
mov r1, #1

wait_2:
ldrex r0, [r3]
cbz r0, wait_2 ; if Semaphore = 0
sub r0, #1 ; dec Semaphore
strex r0, [r3] ; try update
cbnz r0, wait_2 ; if touched
dmb ; sync memory

...

Critical section

add r1, #1
b for_leave
end_for_leave:

```

Any context switch needs to clear reservations

© 2020 Dave R. Zimenc, The Australian National University page 246 of 758 chapter 2: "Mutual Exclusion" (up to page 233)

### Mutual Exclusion

#### Beyond atomic hardware operations

## Semaphores

Basic definition (Dijkstra 1968)

Assuming the following three conditions on a shared memory cell between processes:

- a set of processes agree on a variable  $S$  operating as a flag to indicate synchronization conditions
- an atomic operation P on  $S$  – for 'passen' (Dutch for 'pass'):

$P(S): [as\ soon\ as\ S > 0\ then\ S := S - 1]$  then this is a potentially delaying operation

- an atomic operation V on  $S$  – for 'vr'ygeven' (Dutch for 'to release'):

$V(S): [S := S + 1]$

then the variable  $S$  is called a Semaphore.

© 2020 Dave R. Zimenc, The Australian National University page 241 of 758 chapter 2: "Mutual Exclusion" (up to page 233)

```

Count: .word 0x00000000
Sema: .word 0x00000001

ldr r3, =Sema
ldr r4, =Count
mov r1, #1

for_enter:
cmp r1, #100
bgt end_for_enter

wait_1:
ldr r0, [r3]
cbz r0, wait_1 ; if Semaphore = 0

...

Critical section

add r1, #1
b for_enter
end_for_enter:

for_leave:
ldr r3, =Sema
ldr r4, =Count
mov r1, #1

wait_2:
ldr r0, [r3]
cbz r0, wait_2 ; if Semaphore = 0

...

Critical section

add r1, #1
b for_leave
end_for_leave:

```

Critical section

Critical section

© 2020 Dave R. Zimenc, The Australian National University page 244 of 758 chapter 2: "Mutual Exclusion" (up to page 233)

```

Count: .word 0x00000000
Sema: .word 0x00000001

ldr r3, =Sema
ldr r4, =Count
mov r1, #1

for_enter:
cmp r1, #100
bgt end_for_enter

wait_1:
ldrex r0, [r3]
cbz r0, wait_1 ; if Semaphore = 0
sub r0, #1 ; dec Semaphore
strex r0, [r3] ; try update
cbnz r0, wait_1 ; if touched
dmb ; sync memory

...

Critical section

ldr r0, [r3]
add r0, #1 ; inc Semaphore
str r0, [r3] ; update

add r1, #1
b for_enter
end_for_enter:

for_leave:
ldr r3, =Sema
ldr r4, =Count
mov r1, #1

wait_2:
ldrex r0, [r3]
cbz r0, wait_2 ; if Semaphore = 0
sub r0, #1 ; dec Semaphore
strex r0, [r3] ; try update
cbnz r0, wait_2 ; if touched
dmb ; sync memory

...

Critical section

ldr r0, [r3]
add r0, #1 ; inc Semaphore
str r0, [r3] ; update

add r1, #1
b for_leave
end_for_leave:

```

Any context switch needs to clear reservations

© 2020 Dave R. Zimenc, The Australian National University page 247 of 758 chapter 2: "Mutual Exclusion" (up to page 233)

```

Count: .word 0x00000000
Sema: .word 0x00000001

ldr r3, =Sema
ldr r4, =Count
mov r1, #1

for_enter:
cmp r1, #100
bgt end_for_enter

wait_1:
ldrex r0, [r3]
cbz r0, wait_1 ; if Semaphore = 0
sub r0, #1 ; dec Semaphore
strex r0, [r3] ; try update
cbnz r0, wait_1 ; if touched
dmb ; sync memory

signal_1:
ldrex r0, [r3]
add r0, #1 ; inc Semaphore
strex r0, [r3] ; try update
cbnz r0, signal_1 ; if touched
dmb ; sync memory

add r1, #1
b for_enter
end_for_enter:

ldr r3, =Sema
ldr r4, =Count
mov r1, #1

for_leave:
cmp r1, #100
bgt end_for_leave

wait_2:
ldrex r0, [r3]
cbz r0, wait_2 ; if Semaphore = 0
sub r0, #1 ; dec Semaphore
strex r0, [r3] ; try update
cbnz r0, wait_2 ; if touched
dmb ; sync memory

signal_2:
ldrex r0, [r3]
add r0, #1 ; inc Semaphore
strex r0, [r3] ; try update
cbnz r0, signal_2 ; if touched
dmb ; sync memory

add r1, #1
b for_leave
end_for_leave:

```

Any context switch needs to clear reservations

Critical section

Critical section

© 2010 Dave R. Zimmerman, The Australian National University page 248 of 758 chapter 2: "Mutual Exclusion" up to page 253

### Mutual Exclusion

#### Semaphores

```

S1, S2 : Semaphore := 1;

task body Pi is
begin
loop
----- non_critical_section_i;
wait (S1);
wait (S2);
----- critical_section_i;
signal (S2);
signal (S1);
end loop;
end Pi;

task body Pj is
begin
loop
----- non_critical_section_j;
wait (S2);
wait (S1);
----- critical_section_j;
signal (S1);
signal (S2);
end loop;
end Pj;

```

Works too?

© 2010 Dave R. Zimmerman, The Australian National University page 251 of 758 chapter 2: "Mutual Exclusion" up to page 253

### Mutual Exclusion

#### Semaphores

```

S : Semaphore := 1;

task body Pi is
begin
loop
----- non_critical_section_i;
wait (S);
----- critical_section_i;
signal (S);
end loop;
end Pi;

task body Pj is
begin
loop
----- non_critical_section_j;
wait (S);
----- critical_section_j;
signal (S);
end loop;
end Pj;

```

Works?

© 2010 Dave R. Zimmerman, The Australian National University page 249 of 758 chapter 2: "Mutual Exclusion" up to page 253

### Mutual Exclusion

#### Semaphores

```

S1, S2 : Semaphore := 1;

task body Pi is
begin
loop
----- non_critical_section_i;
wait (S1);
wait (S2);
----- critical_section_i;
signal (S2);
signal (S1);
end loop;
end Pi;

task body Pj is
begin
loop
----- non_critical_section_j;
wait (S2);
wait (S1);
----- critical_section_j;
signal (S1);
signal (S2);
end loop;
end Pj;

```

Works?

- Works for any dynamic number of processes.
- Individual starvation possible!
- Deadlock possible!

© 2010 Dave R. Zimmerman, The Australian National University page 252 of 758 chapter 2: "Mutual Exclusion" up to page 253

### Mutual Exclusion

#### Semaphores

```

S : Semaphore := 1;

task body Pi is
begin
loop
----- non_critical_section_i;
wait (S);
----- critical_section_i;
signal (S);
end loop;
end Pi;

task body Pj is
begin
loop
----- non_critical_section_j;
wait (S);
----- critical_section_j;
signal (S);
end loop;
end Pj;

```

- Mutual exclusion!, No deadlock!, No global live-lock!
- Works for any dynamic number of processes
- Individual starvation possible!

© 2010 Dave R. Zimmerman, The Australian National University page 250 of 758 chapter 2: "Mutual Exclusion" up to page 253

### Mutual Exclusion

#### Summary

#### Mutual Exclusion

- Definition of mutual exclusion
- Atomic load and atomic store operations
  - ... some classical errors
  - Decker's algorithm, Peterson's algorithm
  - Bakery algorithm
- Realistic hardware support
  - Atomic test-and-set, Atomic exchanges, Memory cell reservations
- Semaphores
  - Basic semaphore definition
  - Operating systems style semaphores

© 2010 Dave R. Zimmerman, The Australian National University page 253 of 758 chapter 2: "Mutual Exclusion" up to page 253

## Systems, Networks & Concurrency 2020



# 3

## Communication & Synchronization

Uwe R. Zimmer - The Australian National University

## Communication & Synchronization

### Motivation

### Side effects

Operations have side effects which are visible ...

either

- ☞ ... **locally only**  
(and protected by runtime-, os-, or hardware-mechanisms)

or

- ☞ ... **outside the current process**

☞ If side effects transcend the local process then all forms of access need to be synchronized.

© 2020 Uwe R. Zimmer, The Australian National University page 257 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Sanity check

### Do we need to? – really?

```
int i; {declare globally to multiple threads}
i++;          if i > n {i=0;}
              {in one thread}   {in another thread}
```

- ☞ Handling a 64-bit integer on a 8- or 16-bit controller will not be atomic  
... yet perhaps it is an 8-bit integer.
- ☞ Unaligned manipulations on the main memory will usually not be atomic  
... yet perhaps it is a aligned.
- ☞ Broken down to a load-operate-store cycle, the operations will usually not be atomic  
... yet perhaps the processor supplies atomic operations for the actual case.
- ☞ Many schedulers interrupt threads irrespective of shared data operations  
... yet perhaps this scheduler is aware of the shared data.
- ☞ Local caches might not be coherent  
... yet perhaps they are.

© 2020 Uwe R. Zimmer, The Australian National University page 260 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### References for this chapter

[Ben-Ari06]  
M. Ben-Ari  
*Principles of Concurrent and Distributed Programming*  
2006, second edition, Prentice-Hall, ISBN 0-13-711821-X

[Barnes2006]  
Barnes, John  
*Programming in Ada 2005*  
Addison-Wesley Pearson education, ISBN-13 978-0-321-34078-8, Harlow, England, 2006

[Gosling2005]  
Gosling, James, Joy, B, Steele, Guy & Bracha, Gilad  
*The Java™ Language Specification - third edition*  
2005

[AdaRM2012]  
*Ada Reference Manual - Language and Standard Libraries; ISO/IEC 8652:201x (E)*

[Chapel 1.11.0 Language Specification Version 0.97]  
see course pages or <http://chapel.cray.com/spec/spec-0.97.pdf> released on 2. April 2015

[Saraswat2010]  
Saraswat, Vijay  
*Report on the Programming Language X10 Version 2.01*  
Draft — January 13, 2010

© 2020 Uwe R. Zimmer, The Australian National University page 255 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Sanity check

### Do we need to? – really?

```
int i; {declare globally to multiple threads}
i++;          if i > n {i=0;}
              {in one thread}   {in another thread}
```

What's the worst that can happen?

© 2020 Uwe R. Zimmer, The Australian National University page 258 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Sanity check

### Do we need to? – really?

```
int i; {declare globally to multiple threads}
i++;          if i > n {i=0;}
              {in one thread}   {in another thread}
```

- ☞ 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 usually rare, yet often disastrous.)
- ☞ On assembler level on very simple CPU architectures: synchronization by employing knowledge about the atomicity of CPU-operations and interrupt structures is nevertheless possible and utilized in practice.

In anything higher than assembler level on single core, predictable  $\mu$ -controllers:

☞ **Measures for synchronization are required!**

© 2020 Uwe R. Zimmer, The Australian National University page 261 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Overview

### 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
- ☞ Java, C#, ...
- Protected objects
- ☞ Ada
- Atomic blocks
- ☞ Chapel, X10

#### Message based synchronization

- Asynchronous messages
- ☞ e.g. POSIX, ...
- Synchronous messages
- ☞ e.g. Ada, CHILL, Occam2, ...
- Remote invocation, remote procedure call
- ☞ e.g. Ada, ...

© 2020 Uwe R. Zimmer, The Australian National University page 256 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Sanity check

### Do we need to? – really?

```
int i; {declare globally to multiple threads}
i++;          if i > n {i=0;}
              {in one thread}   {in another thread}
```

- ☞ Handling a 64-bit integer on a 8- or 16-bit controller will not be atomic  
... yet perhaps it is an 8-bit integer.
- ☞ Unaligned manipulations on the main memory will usually not be atomic  
... yet perhaps it is a aligned.
- ☞ Broken down to a load-operate-store cycle, the operations will usually not be atomic  
... yet perhaps the processor supplies atomic operations for the actual case.
- ☞ Many schedulers interrupt threads irrespective of shared data operations  
... yet perhaps this scheduler is aware of the shared data.
- ☞ Local caches might not be coherent  
... yet perhaps they are.

© 2020 Uwe R. Zimmer, The Australian National University page 259 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Towards synchronization

### Condition synchronization by flags

Assumption: word-access atomicity:

i.e. assigning two values (not wider than the size of a 'word') to an aligned memory cell concurrently:

$$x := 0 \quad | \quad x := 500$$

will result in *either*  $x = 0$  or  $x = 500$  – and no other value is ever observable

© 2020 Uwe R. Zimmer, The Australian National University page 262 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Towards synchronization

#### Condition 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:

```

process P1;
statement X;
repeat until Flag;
statement Y;
end P1;

```

```

process P2;
statement A;
Flag := true;
statement B;
end P2;

```

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

© 2020 Dave R. Zimmerman, The Australian National University page 263 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Basic synchronization by Semaphores

Basic definition (Dijkstra 1968)

Assuming the following three conditions on a shared memory cell between processes:

- a set of processes agree on a variable  $S$  operating as a flag to indicate synchronization conditions
- an atomic operation  $P$  on  $S$  — for 'passen' (Dutch for 'pass'):  
 $P(S): [as\ soon\ as\ S > 0\ then\ S := S - 1]$   $\Leftrightarrow$  this is a potentially delaying operation aka: 'Wait', 'Suspend\_Until\_True', 'sem\_wait', ...
- an atomic operation  $V$  on  $S$  — for 'vrygeven' (Dutch for 'to release'):  
 $V(S): [S := S + 1]$   
 aka 'Signal', 'Set-True', 'sem\_post', ...

$\Leftrightarrow$  then the variable  $S$  is called a **Semaphore**.

© 2020 Dave R. Zimmerman, The Australian National University page 266 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Towards synchronization

#### Semaphores in Ada

```

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;

```

$\Leftrightarrow$  This is "queueless" and can translate into a single machine instruction.

only one task can be blocked at Suspend\_Until\_True!  
(Program\_Error will be raised with a second task trying to suspend itself)

$\Leftrightarrow$  no queues!  $\Leftrightarrow$  minimal run-time overhead

© 2020 Dave R. Zimmerman, The Australian National University page 269 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Towards synchronization

#### Condition synchronization by flags

```

var Flag : boolean := false;

```

```

process P1;
statement X;
repeat until Flag;
statement Y;
end P1;

```

```

process P2;
statement A;
Flag := true;
statement B;
end P2;

```

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

© 2020 Dave R. Zimmerman, The Australian National University page 264 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Towards synchronization

#### 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:  $A \rightarrow B; [X \mid A] \rightarrow Y; [X, Y \mid B]$

© 2020 Dave R. Zimmerman, The Australian National University page 267 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Towards synchronization

#### Semaphores in Ada

```

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;

```

$\Leftrightarrow$  for special cases only ... otherwise:

only one task can be blocked at Suspend\_Until\_True!  
(Program\_Error will be raised with a second task trying to suspend itself)

$\Leftrightarrow$  no queues!  $\Leftrightarrow$  minimal run-time overhead

© 2020 Dave R. Zimmerman, The Australian National University page 270 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Towards synchronization

#### Condition 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:

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

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

$\Leftrightarrow$  More powerful synchronization operations are required for critical sections

© 2020 Dave R. Zimmerman, The Australian National University page 265 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Towards synchronization

#### 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:  
 $A \rightarrow B \rightarrow C; X \rightarrow Y \rightarrow Z; [X, Z \mid A, B, C] [A, C \mid X, Y, Z] \neg [B \mid Y]$

© 2020 Dave R. Zimmerman, The Australian National University page 268 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Towards synchronization

#### Malicious use of "queueless semaphores"

with Ada.Synchronous\_Task\_Control; use Ada.Synchronous\_Task\_Control;

```

X : Suspension_Object;

```

```

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

```

```

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

```

$\Leftrightarrow$  Could raise a Program\_Error as multiple tasks potentially suspend on the same semaphore (occurs only with high efficiency semaphores which do not provide process queues)

© 2020 Dave R. Zimmerman, The Australian National University page 271 of 758 (chapter 3: "Communication & Synchronization") up to page 369



## Communication & Synchronization

### Towards synchronization

#### Malicious use of "queueless 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);
  Set_True (X);
end B;

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

☞ Will result in a deadlock (assuming no other Set\_True calls)

© 2010 Dave R. Zimmerman, The Australian National University page 272 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Towards synchronization

#### Semaphores in POSIX

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

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);
    }
  }
}
```

☞ Deadlock?  
☞ Livelock?  
☞ Mutual exclusion?

© 2010 Dave R. Zimmerman, The Australian National University page 275 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Distributed synchronization

#### Conditional Critical Regions

Basic idea:

- Critical regions are a *set of associated 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 to false, the process is suspended / delayed.
- Generally, no access order can be assumed ☞ potential livelocks

© 2010 Dave R. Zimmerman, The Australian National University page 278 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Towards synchronization

#### Malicious use of "queueless 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;
```

☞ Will potentially result in a deadlock (with general semaphores) or a Program\_Error in Ada.

© 2010 Dave R. Zimmerman, The Australian National University page 273 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Towards synchronization

#### Semaphores in Java (since 2004)

```
Semaphore (int permits, boolean fair)
void acquire () (int permits)
void acquireUninterruptibly (int permits)
boolean tryAcquire ()
boolean tryAcquire (int permits, long timeout, TimeUnit unit)
int availablePermits ()
protected void reducePermits (int reduction)
int drainPermits ()
void release ()
void release (int permits)
protected Collection <Thread> getQueuedThreads ()
int getQueueLength ()
boolean hasQueuedThreads ()
boolean isFair ()
String toString ()
```

☞ wait  
☞ check and manipulate  
☞ signal  
☞ administration

© 2010 Dave R. Zimmerman, The Australian National University page 276 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Distributed synchronization

#### 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;
```

© 2010 Dave R. Zimmerman, The Australian National University page 279 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Towards synchronization

#### Semaphores in POSIX

☞ pshared is actually a Boolean indicating whether the semaphore is to be shared between processes

```
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);
```

☞ \*value indicates the number of waiting processes as a negative integer in case the semaphore value is zero

© 2010 Dave R. Zimmerman, The Australian National University page 274 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Towards synchronization

#### Review of semaphores

- Semaphores are **not bound to any resource or method or region**
  - ☞ Compiler has no idea what is supposed to be protected by a semaphore.
- Semaphores are scattered all over the code
  - ☞ Hard to read and highly error-prone.
  - ☞ Adding or deleting a single semaphore operation usually stalls a whole system.
- ☞ Semaphores are generally considered inadequate for non-trivial systems.
  - (all concurrent languages and environments offer efficient and higher-abstraction synchronization methods)
  - ☞ Special (usually close-to-hardware) applications exist.

© 2010 Dave R. Zimmerman, The Australian National University page 277 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Distributed synchronization

#### Review of Conditional Critical Regions

- Well formed synchronization blocks and synchronization conditions.
- Code, data and synchronization primitives are associated (known to compiler and runtime).
- 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
- Condition synchronisation inside the critical code sections requires to leave and re-enter a critical region.
- As with semaphores the conditional critical regions are distributed all over the code.
  - ☞ on a larger scale: same problems as with semaphores.

(The language Edison (Per Brinch Hansen, 1981) uses conditional critical regions for synchronization in a multiprocessor environment (each process is associated with exactly one processor).)

© 2010 Dave R. Zimmerman, The Australian National University page 280 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Centralized synchronization

#### 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 and functions.
- Assure mutual exclusion of all monitor-procedures and functions.

© 2020 Dave R. Zimmerman, The Australian National University page 281 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### Monitors

```

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

```

How to implement conditional synchronization?

© 2020 Dave R. Zimmerman, The Australian National University page 282 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### Monitors with condition synchronization

(Hoare '74)

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.

© 2020 Dave R. Zimmerman, The Australian National University page 283 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### 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; ...

```

© 2020 Dave R. Zimmerman, The Australian National University page 284 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### 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!

© 2020 Dave R. Zimmerman, The Australian National University page 285 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### 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 re-gain access to the monitor.

© 2020 Dave R. Zimmerman, The Australian National University page 286 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### Monitors in Modula-1

- procedure wait (s, r): delays the caller until condition variable s is true (r is the rank (or 'priority') of the caller).
- procedure 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).
- function awaited (s) return integer: check for waiting processes on s.

© 2020 Dave R. Zimmerman, The Australian National University page 287 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### 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.

```

© 2020 Dave R. Zimmerman, The Australian National University page 288 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### Monitors in POSIX ('C')

(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,
                        const pthread_mutexattr_t *attr);
int pthread_mutex_destroy ( pthread_mutex_t *mutex);

int pthread_cond_init ( pthread_cond_t *cond,
                       const pthread_condattr_t *attr);
int pthread_cond_destroy ( pthread_cond_t *cond);
...

```

© 2020 Dave R. Zimmerman, The Australian National University page 289 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

### Communication & Synchronization

#### Centralized synchronization

#### Monitors in POSIX ('C')

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

© 2020 Dave R. Zineff, The Australian National University page 290 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

### Communication & Synchronization

#### Centralized synchronization

#### Monitors in POSIX ('C')

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

Undefined while locked

Undefined while threads are waiting

© 2020 Dave R. Zineff, The Australian National University page 291 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

### Communication & Synchronization

#### Centralized synchronization

#### Monitors in POSIX ('C')

(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,
                             const 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,
                            const struct timespec *abstime);

int pthread_cond_signal ( pthread_cond_t *cond);
int pthread_cond_broadcast ( pthread_cond_t *cond);
```

unlocks 'at least one' thread

unlocks all threads

© 2020 Dave R. Zineff, The Australian National University page 292 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

### Communication & Synchronization

#### Centralized synchronization

#### Monitors in POSIX ('C')

(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,
                              const 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,
                             const 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' i.e. mutex is not locked

© 2020 Dave R. Zineff, The Australian National University page 293 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

### Communication & Synchronization

#### Centralized synchronization

#### Monitors in POSIX ('C')

(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,
                              const 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,
                             const 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 times

© 2020 Dave R. Zineff, The Australian National University page 294 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

### Communication & Synchronization

#### Centralized synchronization

```
#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;
                int buf [BUFF_SIZE];
                } buffer;

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;
}

}
}

© 2020 Dave R. Zineff, The Australian National University page 295 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

```

### Communication & Synchronization

#### Centralized synchronization

```
#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;
                int buf [BUFF_SIZE];
                } buffer;

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;
}

}
}

need to be called with a locked mutex

better to be called after unlocking all mutexes (as it is itself potentially blocking)


© 2020 Dave R. Zineff, The Australian National University page 296 of 758 (chapter 3: "Communication & Synchronization" up to page 369)


```

### Communication & Synchronization

#### Centralized synchronization

#### Monitors in C#

```
using System;
using System.Threading;

static long data_to_protect = 0;

static void Reader()
{
    try {
        Monitor.Enter (data_to_protect);
        Monitor.Wait (data_to_protect);
        ... read out protected data
    }
    finally {
        Monitor.Exit (data_to_protect);
    }
}

static void Writer()
{
    try {
        Monitor.Enter (data_to_protect);
        ... write protected data
        Monitor.Pulse (data_to_protect);
    }
    finally {
        Monitor.Exit (data_to_protect);
    }
}

}
}

© 2020 Dave R. Zineff, The Australian National University page 297 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

```

### Communication & Synchronization

#### Centralized synchronization

#### Monitors in Visual C++

```
using namespace System;
using namespace System::Threading;
private: integer data_to_protect;

void Reader()
{
    try {
        Monitor::Enter (data_to_protect);
        Monitor::Wait (data_to_protect);
        ... read out protected data
    }
    finally {
        Monitor::Exit (data_to_protect);
    }
};

void Writer()
{
    try {
        Monitor::Enter (data_to_protect);
        ... write protected data
        Monitor::Pulse (data_to_protect);
    }
    finally {
        Monitor::Exit (data_to_protect);
    }
};

}
}

© 2020 Dave R. Zineff, The Australian National University page 298 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

```

## Communication & Synchronization

### Centralized synchronization

#### Monitors in Visual Basic

```
Imports System
Imports System.Threading
Private Dim data_to_protect As Integer = 0

Public Sub Reader
    Try
        Monitor.Enter (data_to_protect)
        Monitor.Wait (data_to_protect)
        ... read out protected data
    Finally
        Monitor.Exit (data_to_protect)
    End Try
End Sub

Public Sub Writer
    Try
        Monitor.Enter (data_to_protect)
        ... write protected data
        Monitor.Pulse (data_to_protect)
    Finally
        Monitor.Exit (data_to_protect)
    End Try
End Sub
```

© 2020 Dave R. Zimmerman, The Australian National University page 299 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### Monitors in Java

(by means of language primitives)

Considerations:

- 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 a Java monitor and enter at any time.
  - Methods outside the monitor-object can synchronize at this object.
    - it is impossible to analyse a Java 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 with all objects of a class.

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

© 2020 Dave R. Zimmerman, The Australian National University page 302 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### 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 ();
    ...
}
```

© 2020 Dave R. Zimmerman, The Australian National University page 305 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### Monitors in Java

```
Monitor mon = new Monitor();
Monitor.Condition Condvar = mon.new Condition();

public void reader
throws InterruptedException {
    mon.enter();
    Condvar.await();
    ... read out protected data
    mon.leave();
}

public void writer
throws InterruptedException {
    mon.enter();
    ... write protected data
    Condvar.signal();
    mon.leave();
}
```

... the Java library monitor connects data or condition variables to the monitor by convention only!

© 2020 Dave R. Zimmerman, The Australian National University page 300 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### Monitors in Java

(by means of language primitives)

Considerations:

- 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 do not release the lock!
    - methods, which are activated via notification need to wait for lock-access.
  - Java does *not* require any specific release order (like a queue) for wait-suspended threads
    - livelocks are not prevented at this level (in opposition to RT-Java).
  - There are no explicit conditional variables associated with the monitor or data.
    - notified threads need to wait for the lock to be released and to re-evaluate its entry condition.

© 2020 Dave R. Zimmerman, The Australian National University page 303 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### Monitors in Java

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

```
... 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 ();
        }
    }
}
```

© 2020 Dave R. Zimmerman, The Australian National University page 306 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### Monitors in Java

(by means of language primitives)

Java provides two mechanisms to construct a monitors-like structure:

- 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.

© 2020 Dave R. Zimmerman, The Australian National University page 301 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### Monitors in Java

(by means of language primitives)

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 *adequate* 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 or interfering with this monitor-object in any way *by convention*.

© 2020 Dave R. Zimmerman, The Australian National University page 304 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### Monitors in Java

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

```
... 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;
                }
            }
        }
    }
}
```

© 2020 Dave R. Zimmerman, The Australian National University page 307 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

### Monitors in Java

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

```

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

```

© 2020 Dave R. Zimmerman, The Australian National University page 308 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

### Object-orientation and synchronization

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

- ☞ New methods cannot be added without re-evaluating the class!

Re-usage concepts of object-oriented programming do not translate to synchronized classes (e.g. monitors) and thus 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, yet they are complex and not offered in any concurrent programming language. Alternatively, inheritance can be banned in the context of synchronization (e.g. Ada).

© 2020 Dave R. Zimmerman, The Australian National University page 311 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

### Criticism of monitors

- Mutual exclusion is solved elegantly and safely.
- Conditional synchronization is on the level of semaphores still
  - ☞ all criticism about semaphores applies inside the monitors

☞ Mixture of low-level and high-level synchronization constructs.

© 2020 Dave R. Zimmerman, The Australian National University page 314 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

### Monitors in Java

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

```

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

```

© 2020 Dave R. Zimmerman, The Australian National University page 309 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

### Monitors in POSIX, Visual C++, C#, Visual Basic & Java

- ☞ All provide lower-level primitives for the construction of monitors.
- ☞ All rely on **convention** rather than compiler checks.
- ☞ Visual C++, C+ & Visual Basic offer data-encapsulation and connection to the monitor.
- ☞ Java offers data-encapsulation (yet not with respect to a monitor).
- ☞ POSIX (being a collection of library calls) does not provide any data-encapsulation by itself.
- ☞ Extreme care must be taken when employing object-oriented programming and synchronization (incl. monitors)

© 2020 Dave R. Zimmerman, The Australian National University page 312 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

### 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**.
- Operations are **mutual exclusive** (with exceptions for read-only operations).
- **Guards** (predicates) are **syntactically attached** to entries.
- **No** protected data is accessible (other than by the defined operations).
- **Fairness** inside operations is guaranteed by **queuing** (according to their priorities).
- **Fairness** across all operations is guaranteed by the "internal progress first" rule.
- Re-blocking provided by **re-queuing** to entries (no internal condition variables).

© 2020 Dave R. Zimmerman, The Australian National University page 315 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

### Monitors in Java

Per Brinch Hansen (1938-2007) in 1999:

*Java's most serious mistake was the decision to use the sequential part of the language to implement the run-time support for its parallel features. It strikes me as absurd to write a compiler for the sequential language concepts only and then attempt to skip the much more difficult task of implementing a secure parallel notation. This wishful thinking is part of Java's unfortunate inheritance of the insecure C language and its primitive, error-prone library of threads methods.*

*"Per Brinch Hansen is one of a handful of computer pioneers who was responsible for advancing both operating systems development and concurrent programming from ad hoc techniques to systematic engineering disciplines." (from his IEEE 2002 Computer Pioneer Award)*

© 2020 Dave R. Zimmerman, The Australian National University page 310 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

### 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, Java
- Prohibit nested monitor calls: e.g. Modula-1
- Provide constructs which specify the release of a monitor lock for remote calls, e.g. Ada

© 2020 Dave R. Zimmerman, The Australian National University page 313 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

### Synchronization by protected objects

(Simultaneous read-access)

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

```

protected type Shared_Data (Initial : Data_Item) is
    function Read return Data_Item;
    procedure Write (New_Value : 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 Ada.

© 2020 Dave R. Zimmerman, The Australian National University page 316 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### Synchronization by protected objects

(Condition synchronization: entries & barriers)

Condition synchronization is realized in the form of **protected procedures** combined with boolean predicates (**barriers**): called entries in Ada:

```

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 : Data_Item);
private
  First : Index := Index'First;
  Last : Index := Index'Last;
  Num : Count := 0;
  Buffer : Buffer_T;
end Bounded_Buffer;

```

© 2020 Dave R. Zimmerman, The Australian National University page 317 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### Synchronization by protected objects

(Withdrawing entry calls)

```

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;

```

```

select
  Buffer.Get (Some_Data);
then abort
  -- meanwhile try something else
end select;

select
  delay 10.0;
then abort
  Buffer.Put (Some_Data);
  -- try to enter for 10 s.
end select;

```

© 2020 Dave R. Zimmerman, The Australian National University page 320 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### Synchronization by protected objects

(Operations on entry queues)

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

```

protected type Broadcast is
  entry Receive (M: out Message);
  procedure Send (M: 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 Receive;
  procedure Send (M: Message) is
  begin
    New_Message := M;
    Arrived := Receive'count > 0;
  end Send;
end Broadcast;

```

© 2020 Dave R. Zimmerman, The Australian National University page 323 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### Synchronization by protected objects

(Condition synchronization: entries & 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 : Data_Item) when Num < Buffer_Size is
  begin
    Last := Last + 1;
    Buffer (Last) := Item;
    Num := Num + 1;
  end Put;
end Bounded_Buffer;

```

© 2020 Dave R. Zimmerman, The Australian National University page 318 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### Synchronization by protected objects

(Barrier evaluation)

Barrier in protected objects need to be evaluated only on two occasions:

- on *creating a protected object*, all barrier are evaluated according to the initial values of the internal, protected data.
- on *leaving a protected procedure or entry*, all potentially altered barriers are re-evaluated.

Alternatively an implementation may choose to evaluate barriers on those two occasions:

- on *calling a protected entry*, the one associated barrier is evaluated.
- on *leaving a protected procedure or entry*, all potentially altered barriers with tasks queued up on them are re-evaluated.

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

© 2020 Dave R. Zimmerman, The Australian National University page 321 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### Synchronization by protected objects

(Entry families, requeue & private entries)

Additional, essential primitives for concurrent control flows:

- 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*: external tasks are only considered for queuing on barriers once no internally requeued task can be progressed any further!
- Private entries:** Protected entries which are not accessible from outside the protected object, but can be employed as destinations for requeue operations.

© 2020 Dave R. Zimmerman, The Australian National University page 324 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### Synchronization by protected objects

(Withdrawing entry calls)

```

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;

```

© 2020 Dave R. Zimmerman, The Australian National University page 319 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### Synchronization by protected objects

(Operations on entry queues)

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

```

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

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

```

© 2020 Dave R. Zimmerman, The Australian National University page 322 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

#### Synchronization by protected objects

(Entry families)

```

package Modes is
  type Mode_T is
    (Takeoff, Ascend, Cruising,
     Descent, Landing);
  protected Mode_Gate is
    procedure Set_Mode (Mode: Mode_T);
    entry Wait_For_Mode (Mode:T);
  private
    Current_Mode : Mode_Type := Takeoff;
    end Mode_Gate;
end Modes;

package body Modes is
  protected body Mode_Gate is
    procedure Set_Mode (Mode: Mode_T) is
    begin
      Current_Mode := Mode;
    end Set_Mode;
    entry Wait_For_Mode (Mode:T)
      (For Mode in Mode_T)
      when Current_Mode = Mode is
    begin null;
    end Wait_For_Mode;
  end Mode_Gate;
end Modes;

```

© 2020 Dave R. Zimmerman, The Australian National University page 325 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Centralized synchronization

### Synchronization by protected objects

(Entry families, requeue & private entries)

How to moderate the flow of incoming calls to a busy server farm?

```

type Urgency is (urgent, not_so_urgent);
type Server_Farm is (primary, secondary);
protected Pre_Filter is
  entry Reception (U : Urgency);
private
  entry Server (Server_Farm) (U : Urgency);
end Pre_Filter;
  
```

© 2020 Dave R. Zineac, The Australian National University page 326 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Centralized synchronization

### Synchronization by protected objects

(Entry families, requeue & private entries)

```

protected body Pre_Filter is
  entry Reception (U : Urgency)
    when Server (primary)'count = 0 or else Server (secondary)'count = 0 is
  begin
    if U = urgent and then Server (primary)'count = 0 then
      requeue Server (primary);
    else
      requeue Server (secondary);
    end if;
  end Reception;
  entry Server (for S in Server_Farm) (U : Urgency) when True is
  begin null; -- might try something even more useful
  end Server;
end Pre_Filter;
  
```

© 2020 Dave R. Zineac, The Australian National University page 327 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Centralized synchronization

### Synchronization by protected objects

(Restrictions for protected operations)

All code inside a protected procedure, function or entry is bound to non-blocking operations. Thus the following operations are prohibited:

- entry call statements
- delay statements
- task creations or activations
- select statements
- accept statements
- ... as well as calls to sub-programs which contain any of the above

☞ The requeue facility allows for a potentially blocking operation, and releases the current lock!

© 2020 Dave R. Zineac, The Australian National University page 328 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Shared memory based synchronization

### General

Criteria:

- Levels of abstraction
- Centralized versus distributed
- Support for automated (compiler based) consistency and correctness validation
- Error sensitivity
- Predictability
- Efficiency

© 2020 Dave R. Zineac, The Australian National University page 329 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Shared memory based synchronization

### POSIX

- All low level constructs available
- Connection with the actual data-structures by means of convention only
- Extremely error-prone
- Degree of non-determinism introduced by the 'release some' semantic
- 'C' based
- Portable

© 2020 Dave R. Zineac, The Australian National University page 330 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Shared memory based synchronization

### Java

- Mutual exclusion available.
- General notification feature (not connected to other locks, hence not a conditional variable)
- Universal object orientation makes local analysis hard or even impossible
- Mixture of high-level object oriented features and low level concurrency primitives

© 2020 Dave R. Zineac, The Australian National University page 331 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Shared memory based synchronization

### C#, Visual C++, Visual Basic

- Mutual exclusion via library calls (convention)
- Data is associated with the locks to protect it
- Condition variables related to the data protection locks
- Mixture of high-level object oriented features and low level concurrency primitives

© 2020 Dave R. Zineac, The Australian National University page 332 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Shared memory based synchronization

### C++14

- Mutual exclusion in scopes
- Data is not strictly associated with the locks to protect it
- Condition variables related to the mutual exclusion locks
- Set of essential primitives without combining them in a syntactically strict form (yet?)

© 2020 Dave R. Zineac, The Australian National University page 333 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Shared memory based synchronization

### Rust

- Mutual exclusion in scopes
- Data is strictly associated with locks to protect it
- Condition variables related to the mutual exclusion locks
- Combined with the message passing semantics already a power set of tools.
- Concurrency features migrated to a standard library.

© 2020 Dave R. Zineac, The Australian National University page 334 of 758 (chapter 3: "Communication & Synchronization") up to page 369

## Communication & Synchronization

### Shared memory based synchronization

Modula-1, Chill, Parallel Pascal, ...

- Full implementation of the Dijkstra / Hoare monitor concept

The term **monitor** appears in many other concurrent languages, yet it is usually not associated with an actual language primitive.

© 2010 Dave R. Zimmerman, The Australian National University page 335 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Shared memory based synchronization

#### Ada

- High-level synchronization support which scales to large size projects.
- Full compiler support incl. potential deadlock analysis
- Low-Level semaphores for very special cases

Ada has still no mainstream competitor in the field of explicit concurrency. (2018)

© 2010 Dave R. Zimmerman, The Australian National University page 336 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### High Performance Computing

### Synchronization in large scale concurrency

High Performance Computing (HPC) emphasizes on keeping as many CPU nodes busy as possible:

- Avoid contention on sparse resources.
- Data is assigned to individual processes rather than processes synchronizing on data.
- Data integrity is achieved by keeping the CPU nodes in approximate "lock-step", yet there is still a need to re-sync concurrent entities.

Traditionally this has been implemented using the Message Passing Interface (MPI) while implementing separate address spaces.

- Current approaches employ partitioned address spaces, i.e. memory spaces can overlap and be re-assigned. (Chapel, Fortress, X10)
- Not all algorithms break down into independent computation slices and so there is a need for memory integrity mechanisms in shared/partitioned address spaces.

© 2010 Dave R. Zimmerman, The Australian National University page 337 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Current developments

### Atomic operations in X10

X10 offers only atomic blocks in unconditional and conditional form.

- Unconditional atomic blocks are guaranteed to be non-blocking, which means that they cannot be nested and need to be implemented using roll-backs.
- Conditional atomic blocks can also be used as a pure notification system (similar to the Java notify method).
- Parallel statements (incl. parallel, i.e. unrolled 'loops').
- Shared variables (and their access mechanisms) are not defined.
- The programmer does not specify the scope of the locks (atomic blocks) but they are managed by the compiler/runtime environment.

Code analysis algorithms are required in order to provide efficiently, otherwise the runtime environment needs to associate every atomic block with a global lock.

© 2010 Dave R. Zimmerman, The Australian National University page 338 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Current developments

### Synchronization in Chapel

Chapel offers a variety of concurrent primitives:

- Parallel operations on data (e.g. concurrent array operations)
- Parallel statements (incl. parallel, i.e. unrolled 'loops')
- Parallelism can also be explicitly limited by serializing statements
- Atomic blocks for the purpose to construct atomic transactions
- Memory integrity needs to be programmed by means of synchronization statements (waiting for one or multiple control flows to complete) and/or atomic blocks

Further Chapel semantics are still forthcoming ... so there is still hope for a stronger shared memory synchronization / memory integrity construct.

© 2010 Dave R. Zimmerman, The Australian National University page 339 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Synchronization

### Message-based synchronization

<h4>Synchronization model</h4> <ul style="list-style-type: none"> <li>Asynchronous</li> <li>Synchronous</li> <li>Remote invocation</li> </ul>	<h4>Message structure</h4> <ul style="list-style-type: none"> <li>arbitrary</li> <li>restricted to 'basic' types</li> <li>restricted to un-typed communications</li> </ul>
<h4>Addressing (name space)</h4> <ul style="list-style-type: none"> <li>direct communication</li> <li>mail-box communication</li> </ul>	

© 2010 Dave R. Zimmerman, The Australian National University page 340 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization

### Message protocols

#### Synchronous message (sender waiting)

Delay the sender process until

- Receiver becomes available
- Receiver acknowledges reception

© 2010 Dave R. Zimmerman, The Australian National University page 341 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization

### Message protocols

#### Synchronous message (receiver waiting)

Delay the receiver process until

- Sender becomes available
- Sender concludes transmission

© 2010 Dave R. Zimmerman, The Australian National University page 342 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization

### Message protocols

#### Asynchronous message

Neither the sender nor the receiver is blocked:

- Message is not transferred directly
- A buffer is required to store the messages
- Policy required for buffer sizes and buffer overflow situations

© 2010 Dave R. Zimmerman, The Australian National University page 343 of 758 (chapter 3: "Communication & Synchronization" up to page 369)



## Communication & Synchronization

### Message-based synchronization Message protocols

Asynchronous message (simulated by synchronous messages)

Introducing an intermediate process:

- Intermediate needs to be accepting messages at all times.
- Intermediate also needs to send out messages on request.

⚠ While processes are blocked in the sense of synchronous message passing, they are not actually delayed as the intermediate is always ready.

© 2020 Dave R. Zinin, The Australian National University page 344 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization Message protocols

Remote invocation (simulated by asynchronous messages)

- Simulate two synchronous messages
- Processes are never actually synchronized

© 2020 Dave R. Zinin, The Australian National University page 347 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization

#### Synchronous vs. asynchronous communications

Purpose 'synchronization': ⚠ synchronous messages / remote invocations  
 Purpose 'last message(s) only': ⚠ asynchronous messages

⚠ Synchronous message passing in distributed systems requires hardware support.  
 ⚠ Asynchronous message passing requires the usage of buffers and overflow policies.

Can both communication modes emulate each other?

© 2020 Dave R. Zinin, The Australian National University page 350 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization Message protocols

Synchronous message (simulated by asynchronous messages)

Introducing two asynchronous messages:

- Both processes voluntarily suspend themselves until the transaction is complete.
- As no immediate communication takes place, the processes are never actually synchronized.
- The sender (but not the receiver) process knows that the transaction is complete.

© 2020 Dave R. Zinin, The Australian National University page 345 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization Message protocols

Remote invocation (no results)

Shorter form of remote invocation which does not wait for results to be passed back.

- Still both processes are actually synchronized at the time of the invocation.

© 2020 Dave R. Zinin, The Australian National University page 348 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization

#### Synchronous vs. asynchronous communications

Purpose 'synchronization': ⚠ synchronous messages / remote invocations  
 Purpose 'last message(s) only': ⚠ asynchronous messages

⚠ Synchronous message passing in distributed systems requires hardware support.  
 ⚠ Asynchronous message passing requires the usage of buffers and overflow policies.

Can both communication modes emulate each other?

- Synchronous communications are emulated by a combination of asynchronous messages in some systems (not identical with hardware supported synchronous communication).
- Asynchronous communications can be emulated in synchronized message passing systems by introducing a 'buffer-task' (de-coupling sender and receiver as well as allowing for broadcasts).

© 2020 Dave R. Zinin, The Australian National University page 351 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization Message protocols

Remote invocation

- Delay sender or receiver until the first rendezvous point
- Pass parameters
- Keep sender blocked while receiver executes the local procedure
- Pass results
- Release both processes out of the rendezvous

© 2020 Dave R. Zinin, The Australian National University page 346 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization Message protocols

Remote invocation (no results) (simulated by asynchronous messages)

- Simulate one synchronous message
- Processes are never actually synchronized

© 2020 Dave R. Zinin, The Australian National University page 349 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization

#### Addressing (name space)

Direct versus 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

© 2020 Dave R. Zinin, The Australian National University page 352 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization

#### 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

© 2010 Dave R. Ziminec, The Australian National University page 353 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization

#### Message structure (Ada)

Reading and writing values of any subtype S of a specific type T 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 subtype S of a specific type T 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;
    
```

© 2010 Dave R. Ziminec, The Australian National University page 355 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization

#### Message-based synchronization in Occam2

Communication is ensured by means of a 'channel', which:

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

```

CHAN OF INT SensorChannel:
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
    
```

concurrent entities are synchronized at these points

© 2010 Dave R. Ziminec, The Australian National University page 357 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization

#### 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, C)
- ... semi-automatic (CORBA)
- ... automatic (compiler-generated) and typed-persistent (Ada, CHILL, Occam2)

© 2010 Dave R. Ziminec, The Australian National University page 354 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization

#### Message-passing systems examples:

	ordered	symmetrical	asymmetrical	synchronous	asynchronous	direct	indirect	contents	one-to-one	many-to-one	many-to-many	method
POSIX: "message queues":	☑	☑	☑	☑	☑	☑	☑	byte-stream	☑	☑	☑	message queues
MPI: "message passing":	☑	☑	☑	☑	☑	☑	☑	memory-blocks	☑	☑	☑	message passing
CHILL: "buffers", "signals":	☑	☑	☑	☑	☑	☑	☑	basic types	☑	☑	☑	message passing
Occam2: "channels":	☑	☑	☑	☑	☑	☑	☑	fully typed	☑	☑	☑	message passing
Ada: "(extended) rendezvous":	☑	☑	☑	☑	☑	☑	☑	fully typed	☑	☑	☑	remote invocation
Go: "channels":	☑	☑	☑	☑	☑	☑	☑	fully typed	☑	☑	☑	channels
Erlang: "processes":	☑	☑	☑	☑	☑	☑	☑	fully typed	☑	☑	☑	message passing
Java: "no message passing system defined"	☑	☑	☑	☑	☑	☑	☑	no message passing system defined	☑	☑	☑	message passing

© 2010 Dave R. Ziminec, The Australian National University page 359 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization

#### Message-based synchronization in Occam2

Communication is ensured by means of a 'channel', which:

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

```

CHAN OF INT SensorChannel:
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
    
```

Essential Occam2 keywords

```

ALT PAR SEQ PRI
DATA TYPE RECORD OFFSETOF PACKED
BOOL BYTE INT REAL
CASE IF ELSE FOR FROM WHILE
FUNCTION RESULT PROC IS
PROCESSOR PROTOCOL TIMER
SKIP STOP VALOF
        
```

☞ Concurrent, distributed, real-time programming language!

© 2010 Dave R. Ziminec, The Australian National University page 360 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization

#### Message structure (Ada)

```

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;
    
```

© 2010 Dave R. Ziminec, The Australian National University page 355 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization

#### Message-passing systems examples:

	ordered	symmetrical	asymmetrical	synchronous	asynchronous	direct	indirect	contents	one-to-one	many-to-one	many-to-many	method
POSIX: "message queues":	☑	☑	☑	☑	☑	☑	☑	byte-stream	☑	☑	☑	message queues
MPI: "message passing":	☑	☑	☑	☑	☑	☑	☑	memory-blocks	☑	☑	☑	message passing
CHILL: "buffers", "signals":	☑	☑	☑	☑	☑	☑	☑	basic types	☑	☑	☑	message passing
Occam2: "channels":	☑	☑	☑	☑	☑	☑	☑	fully typed	☑	☑	☑	message passing
Ada: "(extended) rendezvous":	☑	☑	☑	☑	☑	☑	☑	fully typed	☑	☑	☑	remote invocation
Go: "channels":	☑	☑	☑	☑	☑	☑	☑	fully typed	☑	☑	☑	channels
Erlang: "processes":	☑	☑	☑	☑	☑	☑	☑	fully typed	☑	☑	☑	message passing
Java: "no message passing system defined"	☑	☑	☑	☑	☑	☑	☑	no message passing system defined	☑	☑	☑	message passing

© 2010 Dave R. Ziminec, The Australian National University page 361 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization

#### Message-based synchronization in CHILL

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.

☞ strong support for concurrency, synchronization, and communication (monitors, buffered message passing, synchronous channels)

```

dcl SensorBuffer buffer (32) int;

--
send SensorBuffer (reading);
--
signal SensorChannel = (int) to consumertype;
--
send SensorChannel (reading)
to consumer
--
receive case
(SensorBuffer in data) : ...
esac;

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

© 2010 Dave R. Ziminec, The Australian National University page 361 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization

### Message-based synchronization in CHILL

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.

- strong support for concurrency, synchronization, and communication (monitors, buffered message passing, synchronous channels)

```

dcl SensorBuffer buffer (32) int;

--
send SensorBuffer (reading asynchronous) receive case
--> (SensorBuffer in data) : ...
esac;

signal SensorChannel = (int) to consumertype;

--
send SensorChannel (reading) receive case
to consumer synchronous --> (SensorChannel in data): ...
esac;

```

© 2020 Dave R. Zimmerman, The Australian National University page 362 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization

### Message-based synchronization in Ada

(Extended rendezvous)

```

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

```

time time

© 2020 Dave R. Zimmerman, The Australian National University page 365 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization

### Message-based synchronization in Ada

Some things to consider for task-entries:

- In contrast to protected-object-entries, task-entry bodies *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 which owns the entry.
- Entry families** (arrays of entries) are supported.
- Private entries** (accessible for internal tasks) are supported.

© 2020 Dave R. Zimmerman, The Australian National University page 368 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization

### Message-based synchronization in Ada

Ada 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 then:

- 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 if blocked until the remote routine is completed (≙ 'extended rendezvous')

© 2020 Dave R. Zimmerman, The Australian National University page 363 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization

### Message-based synchronization in Ada

(Rendezvous)

```

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

```

time time

© 2020 Dave R. Zimmerman, The Australian National University page 366 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Summary

### Communication & 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
  - Examples

© 2020 Dave R. Zimmerman, The Australian National University page 369 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization

### Message-based synchronization in Ada

(Rendezvous)

```

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

```

time time

© 2020 Dave R. Zimmerman, The Australian National University page 364 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

## Communication & Synchronization

### Message-based synchronization

### Message-based synchronization in Ada

(Extended rendezvous)

```


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

```

time time

© 2020 Dave R. Zimmerman, The Australian National University page 367 of 758 (chapter 3: "Communication & Synchronization" up to page 369)

# Systems, Networks & Concurrency 2020



# 4

## Non-determinism

Uwe R. Zimmer - The Australian National University

# Non-determinism

## Non-determinism by design

Dijkstra's **guarded commands** (non-deterministic case statements):

```

if x <= y -> m := x
□ x >= y -> m := y
fi

```

Selection is non-deterministic for  $x=y$

☞ 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  
All true case statements in any language are potentially concurrent and non-deterministic.

© 2020 Uwe R. Zimmer, The Australian National University page 373 of 758 (chapter 4: "Non-determinism" up to page 395)

# Non-determinism

## Non-determinism by interaction

### Selective waiting in Occam2

```

ALT
  Guard1
    Process1
  Guard2
    Process2
...

```

- Guards are referring to boolean expressions and/or channel input operations.
- The boolean expressions are local expressions, i.e. if none of them evaluates to true at the time of the evaluation of the ALT-statement, then the process is stopped.
- If all triggered channel input operations evaluate to false, the process is suspended until further activity on one of the named channels.
- Any Occam2 process can be employed in the ALT-statement
- The ALT-statement is non-deterministic (there is also a deterministic version: PR1 ALT).

© 2020 Uwe R. Zimmer, The Australian National University page 376 of 758 (chapter 4: "Non-determinism" up to page 395)

# Non-determinism

## References for this chapter

[Ben-Ari06]  
M. Ben-Ari  
*Principles of Concurrent and Distributed Programming*  
2006, second edition, Prentice-Hall, ISBN 0-13-711821-X

[AdaRM2012]  
Ada Reference Manual - Language and Standard Libraries;  
ISO/IEC 8652:201x (E)

[Barnes2006]  
Barnes, John  
*Programming in Ada 2005*  
Addison-Wesley Pearson education, ISBN-13 978-0-321-34078-8, Harlow, England, 2006

© 2020 Uwe R. Zimmer, The Australian National University page 371 of 758 (chapter 4: "Non-determinism" up to page 395)

# Non-determinism

## Non-determinism by design

Dijkstra's **guarded commands** (non-deterministic case statements):

```

if x <= y -> m := x
□ x >= y -> m := y
fi

```

Selection is non-deterministic for  $x=y$

☞ 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  
All true case statements in any language are potentially concurrent and non-deterministic.

Numerical non-determinism in **concurrent statements** (Chapel):

```

writeln (* reduce [i in 1..10] exp (i));
writeln (* reduce [i in 1..1000000] i ** 2.0);

```

Results may be non-deterministic depending on numeric type

☞ The programmer needs to understand the numerical implications of out-of-order expressions.

© 2020 Uwe R. Zimmer, The Australian National University page 374 of 758 (chapter 4: "Non-determinism" up to page 395)

# Non-determinism

## Non-determinism by interaction

### Selective waiting in Occam2

```

ALT
  NumberInBuffer < Size & Append ? Buffer [Top]
  SEQ
    NumberInBuffer := NumberInBuffer + 1
    Top := (Top + 1) REM Size
  NumberInBuffer > 0 & Request ? ANY
  SEQ
    Take ! Buffer [Base]
    NumberInBuffer := NumberInBuffer - 1
    Base := (Base + 1) REM Size

```

- Synchronization on input-channels only (channels are directed in Occam2):
  - to initiate the sending of data (Take ! Buffer [Base]), a request need to be made first which triggers the condition: (Request ? ANY)

CSP (Communicating Sequential Processes, Hoare 1978) also supports non-deterministic selective waiting

© 2020 Uwe R. Zimmer, The Australian National University page 377 of 758 (chapter 4: "Non-determinism" up to page 395)

# Non-determinism

## Definitions

**Non-determinism by design:**

*A property of a computation which may have more than one result.*

**Non-determinism by interaction:**

*A property of the operation environment which may lead to different sequences of (concurrent) stimuli.*

© 2020 Uwe R. Zimmer, The Australian National University page 372 of 758 (chapter 4: "Non-determinism" up to page 395)

# Non-determinism

## Non-determinism by design

### Motivation for non-deterministic design

By explicitly leaving the sequence of evaluation or execution undetermined:

- ☞ The compiler / runtime environment can directly (i.e. without any analysis) translate the source code into a concurrent implementation.
- ☞ The implementation gains potentially significantly in performance
- ☞ The programmer does not need to handle any of the details of a concurrent implementation (access locks, messages, synchronizations, ...)

A programming language which allows for those formulations is required!

☞ current language support: Ada, X10, Chapel, Fortress, Haskell, OCaml, ...

© 2020 Uwe R. Zimmer, The Australian National University page 375 of 758 (chapter 4: "Non-determinism" up to page 395)

# Non-determinism

## Non-determinism by interaction

### Select function in POSIX

```

int pselect(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
            const struct timespec *timeout, sigset_t *sigmask);

```

with:

- n being one more than the maximum of any file descriptor in any of the sets.
- after return the sets will have been reduced to the channels which have been triggered.
- the return value is used as success / failure indicator.

The POSIX select function implements parts of general selective waiting:

- pselect returns if one or multiple I/O channels have been triggered or an error occurred.
- Branching into individual code sections is not provided.
- Guards are not provided.

After return it is required that the following code implements a *sequential* testing of all channels in the sets.

© 2020 Uwe R. Zimmer, The Australian National University page 378 of 758 (chapter 4: "Non-determinism" up to page 395)

## Non-determinism

### Selective Synchronization

#### Message-based selective synchronization in Ada

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

© 2010 Dave R. Zimmerman, The Australian National University page 379 of 758 (chapter 4: "Non-determinism" up to page 395)

## Non-determinism

### Selective Synchronization

#### Basic forms of selective synchronization

(select-guarded-accept)

```
select
when <condition> => accept ...
or
when <condition> => accept ...
or
when <condition> => accept ...
--
end select;
```

- If all conditions are 'true'
  - ☞ **identical to the previous form.**
- If some condition evaluate to 'true'
  - ☞ **the accept statement after those conditions are treated like in the previous form.**
- If all conditions evaluate to 'false'
  - ☞ **Program\_Error is raised.**

Hence it is important that the set of conditions covers all possible states.

This form is identical to Dijkstra's guarded commands.

© 2010 Dave R. Zimmerman, The Australian National University page 382 of 758 (chapter 4: "Non-determinism" up to page 395)

## Non-determinism

### Selective Synchronization

#### Basic forms of selective synchronization

(select-guarded-accept-terminate)

```
select
when <condition> => accept ...
or
when <condition> => accept ...
or
when <condition> => accept ...
--
when <condition> => terminate;
--
end select;
```

- If none of the open entries have waiting calls and none of them can ever be called again
  - ☞ **The terminate alternative is chosen, i.e. the task is terminated.**

This situation occurs if:

- ☞ ... all tasks which can possibly call on any of the open entries are terminated.
- ☞ or ... all remaining tasks which can possibly call on any of the open entries are waiting on select-terminate statements themselves and none of their open entries can be called either. In this case all those waiting-termination tasks are terminated as well.

terminate cannot be mixed with else or delay

© 2010 Dave R. Zimmerman, The Australian National University page 385 of 758 (chapter 4: "Non-determinism" up to page 395)

## Non-determinism

### Selective Synchronization

#### Message-based selective synchronization in Ada

```
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;
```

```
accept_statement ::= accept entry_direct_name [(entry_index)] parameter_profile [do
handled_sequence_of_statements
end [entry_identifier]];
delay_statement ::= delay until delay_expression; | delay delay_expression;
```

... underlying concept: Dijkstra's guarded commands

conditional\_entry\_call and timed\_entry\_call implements ...

- ... the possibility to withdraw an outgoing call.
- ... this might be restricted if calls have already been partly processed.

© 2010 Dave R. Zimmerman, The Australian National University page 380 of 758 (chapter 4: "Non-determinism" up to page 395)

## Non-determinism

### Selective Synchronization

#### Basic forms of selective synchronization

(select-guarded-accept-else)

```
select
when <condition> => accept ...
or
when <condition> => accept ...
or
when <condition> => accept ...
--
else
<statements>
end select;
```

- If all currently open entries have no waiting calls or all entries are closed
  - ☞ **The else alternative is chosen, the associated statements executed and the select statement completes.**
- Otherwise ☞ **one of the open entries with waiting calls is chosen as above.**

This form never suspends the task.

This enables a task to *withdraw* its offer to accept a set of calls if no tasks are currently waiting.

© 2010 Dave R. Zimmerman, The Australian National University page 383 of 758 (chapter 4: "Non-determinism" up to page 395)

## Non-determinism

### Selective Synchronization

#### Message-based selective synchronization in Ada

Forms of selective waiting:

```
select_statement ::= selective_accept |
conditional_entry_call |
timed_entry_call |
asynchronous_select
```

... underlying concept: Dijkstra's guarded commands

conditional\_entry\_call and timed\_entry\_call implements ...

- ... the possibility to withdraw an outgoing call.
- ... this might be restricted if calls have already been partly processed.

© 2010 Dave R. Zimmerman, The Australian National University page 386 of 758 (chapter 4: "Non-determinism" up to page 395)

## Non-determinism

### Selective Synchronization

#### Basic forms of selective synchronization

(select-accept)

```
select
accept ...
or
accept ...
or
accept ...
--
end select;
```

- If none of the entries have waiting calls
  - ☞ **the process is suspended until a call arrives.**
- If exactly one of the entries has waiting calls
  - ☞ **this entry is selected.**
- If multiple entries have waiting calls
  - ☞ **one of those is selected (non-deterministically).** The selection can be prioritized by means of the real-time-systems annex.

The code following the selected entry (if any) is executed and the select statement completes.

© 2010 Dave R. Zimmerman, The Australian National University page 381 of 758 (chapter 4: "Non-determinism" up to page 395)

## Non-determinism

### Selective Synchronization

#### Basic forms of selective synchronization

(select-guarded-accept-delay)

```
select
when <condition> => accept ...
or
when <condition> => accept ...
or
when <condition> => accept ...
--
when <condition> => delay [until] ...
<statements>
or
when <condition> => delay [until] ...
<statements>
--
end select;
```

- If none of the open entries have waiting calls before the deadline specified by the earliest open delay alternative
  - ☞ **This earliest delay alternative is chosen and the statements associated with it executed.**
- Otherwise ☞ **one of the open entries with waiting calls is chosen as above.**

This enables a task to *withdraw* its offer to accept a set of calls if no other task is calling after some time.

© 2010 Dave R. Zimmerman, The Australian National University page 384 of 758 (chapter 4: "Non-determinism" up to page 395)

## Non-determinism

### Selective Synchronization

#### Conditional entry-calls

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

- If the call is not accepted immediately
  - ☞ **The else alternative is chosen.**

This is e.g. useful to probe the state of a server before committing to a potentially blocking call.

Even though it is tempting to use this statement in a "busy-waiting" semantic, there is usually no need to do so, as better alternatives are available.

There is only *one* entry-call and *one* else alternative.

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

© 2010 Dave R. Zimmerman, The Australian National University page 387 of 758 (chapter 4: "Non-determinism" up to page 395)

## Non-determinism

### Selective Synchronization

#### Timed entry-calls

```

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

```

- If the call is not accepted before the deadline specified by the delay alternative the delay alternative is chosen.

This is e.g. useful to withdraw an entry call after some specified time-out.

There is only one entry-call and one delay alternative.

**Example:**

```

select
  Controller_Request (Some_Item);
  ----- process_data
or
  delay 45.0; ----- seconds
  ----- try something else
end select;

```

© 2020 Dave R. Zimmerman, The Australian National University page 388 of 758 (chapter 4: "Non-determinism" up to page 395)

## Non-determinism

### Non-determinism

#### Correctness of non-deterministic programs

**Partial correctness:**  
 $(P(I) \wedge \text{terminates}(\text{Program}(I, O))) \Rightarrow Q(I, O)$

**Total correctness:**  
 $P(I) \Rightarrow (\text{terminates}(\text{Program}(I, O)) \wedge Q(I, O))$

**Safety properties:**  
 $(P(I) \wedge \text{Processes}(I, S)) \Rightarrow \Box Q(I, S)$   
 where  $\Box Q$  means that Q does always hold

**Liveness properties:**  
 $(P(I) \wedge \text{Processes}(I, S)) \Rightarrow \Diamond Q(I, S)$   
 where  $\Diamond Q$  means that Q does eventually hold (and will then stay true) and S is the current state of the concurrent system

© 2020 Dave R. Zimmerman, The Australian National University page 391 of 758 (chapter 4: "Non-determinism" up to page 395)

## Non-determinism

### Non-determinism

#### Correctness of non-deterministic programs

```

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

```

Concrete:

- Every time you formulate a non-deterministic statement like the one on the left you need to formulate an invariant which holds true whichever alternative will actually be chosen.

This is very similar to finding loop invariants in sequential programs

© 2020 Dave R. Zimmerman, The Australian National University page 394 of 758 (chapter 4: "Non-determinism" up to page 395)

## Non-determinism

### Selective Synchronization

#### Message-based selective synchronization in Ada

Forms of selective waiting:

```

select_statement ::= selective_accept |
                  conditional_entry_call |
                  timed_entry_call |
                  asynchronous_select

```

... underlying concept: Dijkstra's guarded commands

asynchronous\_select implements ...

... the possibility to escape a running code block due to an event from outside this task. (outside the scope of this course check: Real-Time Systems)

© 2020 Dave R. Zimmerman, The Australian National University page 389 of 758 (chapter 4: "Non-determinism" up to page 395)

## Non-determinism

### Non-determinism

#### Correctness of non-deterministic programs

- Correctness predicates need to hold true irrespective of the actual sequence of interaction points.

or

- Correctness predicates need to hold true for all possible sequences of interaction points.

Therefore correctness predicates need to be based on invariants, i.e. invariant predicates which are independent of the potential execution sequences, yet support the overall correctness predicates.

© 2020 Dave R. Zimmerman, The Australian National University page 392 of 758 (chapter 4: "Non-determinism" up to page 395)

## Non-determinism

### Summary

#### Non-Determinism

- Non-determinism by design:**
  - Benefits & considerations
- Non-determinism by interaction:**
  - Selective synchronization
  - Selective accepts
  - Selective calls
- Correctness of non-deterministic programs:**
  - Sources of non-determinism
  - Predicates & invariants

© 2020 Dave R. Zimmerman, The Australian National University page 395 of 758 (chapter 4: "Non-determinism" up to page 395)

## Non-determinism

### Non-determinism

#### Sources of Non-determinism

As concurrent entities are not in "lockstep" synchronization, they "overtake" each other and arrive at synchronization points in non-deterministic order, due to (just a few):

- Operating systems / runtime environments:
  - Schedulers are often non-deterministic.
  - System load will have an influence on concurrent execution.
  - Message passing systems react load depended.
- Networks & communication systems:
  - Traffic will arrive in an unpredictable way (non-deterministic).
  - Communication systems congestions are generally unpredictable.
- Computing hardware:
  - Timers drift and clocks have granularities.
  - Processors have out-of-order units.
- ... basically: **Physical systems (and computer systems connected to the physical world) are intrinsically non-deterministic.**

© 2020 Dave R. Zimmerman, The Australian National University page 390 of 758 (chapter 4: "Non-determinism" up to page 395)

## Non-determinism

### Non-determinism

#### Correctness of non-deterministic programs

For example (in verbal form):  
 "Mutual exclusion accessing a specific resource holds true, for all possible numbers, sequences or interleavings of requests to it"

An invariant would for instance be that the number of writing tasks inside a protected object is less or equal to one.

- Those invariants are the only practical way to guarantee (in a logical sense) correctness in concurrent / non-deterministic systems. (as enumerating all possible cases and proving them individually is in general not feasible)

© 2020 Dave R. Zimmerman, The Australian National University page 393 of 758 (chapter 4: "Non-determinism" up to page 395)

Systems, Networks & Concurrency 2020




# 5

## Data Parallelism

Uwe R. Zimmer - The Australian National University

Data Parallelism



Vector Machines  
Vectorization


```

type Real_Precision = Float
type Scalar = Real_Precision
type Vector = [Real_Precision]
scale :: Scalar -> Vector -> Vector
scale scalar vector = map (scalar *) vector
  
```

Potentially concurrent, yet  
Executed sequentially.

© 2020 Uwe R. Zimmer, The Australian National University page: 399 of 758 (chapter 5: "Data Parallelism" up to page 427)

Data Parallelism



Vector Machines  
Vectorization

```

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

Buzzword collection:  
AltVec, SPE, MMX, SSE,  
NEON, SPU, AVX, ...

Translates into  
CPU-level vector operations

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

© 2020 Uwe R. Zimmer, The Australian National University page: 402 of 758 (chapter 5: "Data Parallelism" up to page 427)

Data Parallelism

References


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

[Ada 2012 Language Reference Manual]  
see course pages or <http://www.ada-auth.org/standards/ada12.html>

[Chapel 1.13 Language Specification Version 0.981]  
see course pages or  
[http://chapel.cray.com/docs/latest/\\_downloads/chapelLanguageSpec.pdf](http://chapel.cray.com/docs/latest/_downloads/chapelLanguageSpec.pdf)  
released on 7. April 2016

© 2020 Uwe R. Zimmer, The Australian National University page: 397 of 758 (chapter 5: "Data Parallelism" up to page 427)

Data Parallelism



Vector Machines  
Vectorization

```


import Control.Parallel.Strategies
type Real_Precision = Float
type Scalar = Real_Precision
type Vector = [Real_Precision]
scale :: Scalar -> Vector -> Vector
scale scalar vector = parMap rpar (scalar *) vector
  
```

Executed in parallel.

This may be faster or slower  
than a sequential execution.

© 2020 Uwe R. Zimmer, The Australian National University page: 400 of 758 (chapter 5: "Data Parallelism" up to page 427)

Data Parallelism



Vector Machines  
Vectorization


```

const Index = {1 .. 100000000},
  Vector : [Index] real = 1.0,
  Scale : real = 5.1,
  Scaled : [Vector] real = Scale * Vector;
  
```

Function is  
"promoted"

© 2020 Uwe R. Zimmer, The Australian National University page: 403 of 758 (chapter 5: "Data Parallelism" up to page 427)

Data Parallelism




Vector Machines  
Vectorization

```

type Real_Precision = Float
type Scalar = Real_Precision
type Vector = [Real_Precision]
scale :: Scalar -> Vector -> Vector
scale scalar vector = map (scalar *) vector
  
```

© 2020 Uwe R. Zimmer, The Australian National University page: 398 of 758 (chapter 5: "Data Parallelism" up to page 427)

Data Parallelism




Vector Machines  
Vectorization

```

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

© 2020 Uwe R. Zimmer, The Australian National University page: 401 of 758 (chapter 5: "Data Parallelism" up to page 427)

Data Parallelism



Vector Machines  
Vectorization

```

const Index = {1 .. 100000000},
  Vector : [Index] real = 1.0,
  Scale : real = 5.1,
  Scaled : [Vector] real = Scale * Vector;
  
```

Function is  
"promoted"

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

© 2020 Uwe R. Zimmer, The Australian National University page: 404 of 758 (chapter 5: "Data Parallelism" up to page 427)

**Data Parallelism**

**Vector Machines**

**Reduction**

```

type Real_Precision = Float
type Vector = [Real_Precision]
equal :: Vector -> Vector -> Bool
equal v_1 v_2 = foldr (&&) True $ zipWith (==) v_1 v_2

```

© 2010 Dave R. Zimmerman, The Australian National University page 405 of 758 (chapter 5: "Data Parallelism" up to page 427)

**Data Parallelism**

**Vector Machines**

**Reduction**

```

type Real_Precision = Float
type Vector = [Real_Precision]
equal :: Vector -> Vector -> Bool
equal v_1 v_2 = foldr (&&) True $ zipWith (==) v_1 v_2

```

Potentially concurrent, yet:  
Executed lazy sequentially.

© 2010 Dave R. Zimmerman, The Australian National University page 406 of 758 (chapter 5: "Data Parallelism" up to page 427)

**Data Parallelism**

**Vector Machines**

**Reduction**

```

type Real_Precision = Float
type Vector = [Real_Precision]
equal :: Vector -> Vector -> Bool
equal = (==)

```

Potentially concurrent, yet:  
Executed lazy sequentially.

© 2010 Dave R. Zimmerman, The Australian National University page 407 of 758 (chapter 5: "Data Parallelism" up to page 427)

**Data Parallelism**

**Vector Machines**

**Reduction**

```

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

```

© 2010 Dave R. Zimmerman, The Australian National University page 408 of 758 (chapter 5: "Data Parallelism" up to page 427)

**Data Parallelism**

**Vector Machines**

**Reduction**

```

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

```

Translates into  
CPU-level vector operations  
^chain is evaluated lazy sequentially.

© 2010 Dave R. Zimmerman, The Australian National University page 409 of 758 (chapter 5: "Data Parallelism" up to page 427)

**Data Parallelism**

**Vector Machines**

**Reduction**

```

type Real is digits 15;
type Vectors is array (Positive range <>) of Real;
function "=" (Vector_1, Vector_2 : Vectors) return Boolean is (Vector_1 = Vector_2);

```

Infinite recursion

Translates into  
CPU-level vector operations  
^chain is evaluated lazy sequentially.

© 2010 Dave R. Zimmerman, The Australian National University page 410 of 758 (chapter 5: "Data Parallelism" up to page 427)

**Data Parallelism**

**Vector Machines**

**Reduction**

```

type Real is digits 15;
type Vectors is array (Positive range <>) of Real;
function Equal (Vector_1, Vector_2 : Vectors) return Boolean is (Vector_1 = Vector_2);

```

Translates into  
CPU-level vector operations  
^chain is evaluated lazy sequentially.

© 2010 Dave R. Zimmerman, The Australian National University page 411 of 758 (chapter 5: "Data Parallelism" up to page 427)

**Data Parallelism**

**Vector Machines**

**Reduction**

```

type Real is digits 15;
type Vectors is array (Positive range <>) of Real;
function Equal (Vector_1, Vector_2 : Vectors) return Boolean renames "=";

```

Translates into  
CPU-level vector operations  
^chain is evaluated lazy sequentially.

© 2010 Dave R. Zimmerman, The Australian National University page 412 of 758 (chapter 5: "Data Parallelism" up to page 427)

**Data Parallelism**

**Vector Machines**

**Reduction**

```

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

```

Translates into  
CPU-level vector operations  
^chain is evaluated lazy sequentially.

© 2010 Dave R. Zimmerman, The Australian National University page 413 of 758 (chapter 5: "Data Parallelism" up to page 427)



**Data Parallelism**

Vector Machines

Reduction

```
const Index = {1 .. 10000000},
  Vector_1, Vector_2 : [Index] real = 1.0;
proc Equal (v1, v2) : bool
  {return && reduce (v1 == v2);}
```


Function is "promoted"

© 2010 Uwe R. Zimmer, The Australian National University page 414 of 758 (chapter 5: "Data Parallelism" up to page 427)

**Data Parallelism**

Vector Machines

General Data-parallelism




© 2010 Uwe R. Zimmer, The Australian National University page 417 of 758 (chapter 5: "Data Parallelism" up to page 427)

**Data Parallelism**

Vector Machines

General Data-parallelism



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

© 2010 Uwe R. Zimmer, The Australian National University page 420 of 758 (chapter 5: "Data Parallelism" up to page 427)

**Data Parallelism**

Vector Machines

Reduction

```
const Index = {1 .. 10000000},
  Vector_1, Vector_2 : [Index] real = 1.0;
proc Equal (v1, v2) : bool
  {return && reduce (v1 == v2);}
```

Function is "promoted"

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


^-operations are evaluated in a concurrent divide-and-conquer (binary tree) structure.

© 2010 Uwe R. Zimmer, The Australian National University page 415 of 758 (chapter 5: "Data Parallelism" up to page 427)

**Data Parallelism**

Vector Machines

General Data-parallelism




© 2010 Uwe R. Zimmer, The Australian National University page 418 of 758 (chapter 5: "Data Parallelism" up to page 427)

**Data Parallelism**

Vector Machines

General Data-parallelism



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

© 2010 Uwe R. Zimmer, The Australian National University page 421 of 758 (chapter 5: "Data Parallelism" up to page 427)

**Data Parallelism**

Vector Machines

Reduction

```
const Index = {1 .. 10000000},
  Vector_1, Vector_2 : [Index] real = 1.0;
proc Equal (v1, v2) : bool
  {return v1 == v2;}
writeln (Equal (Vector_1, Vector_2));
```


Type mismatch

© 2010 Uwe R. Zimmer, The Australian National University page 416 of 758 (chapter 5: "Data Parallelism" up to page 427)

**Data Parallelism**

Vector Machines

General Data-parallelism



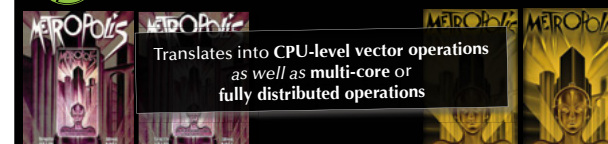
```
const Mask : [1 .. 3, 1 .. 3] real = ((0, -1, 0), (-1, 5, -1), (0, -1, 0));
```

© 2010 Uwe R. Zimmer, The Australian National University page 419 of 758 (chapter 5: "Data Parallelism" up to page 427)

**Data Parallelism**

Vector Machines

General Data-parallelism



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

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


© 2010 Uwe R. Zimmer, The Australian National University page 422 of 758 (chapter 5: "Data Parallelism" up to page 427)

**Data Parallelism**

---

Vector Machines

General Data-parallelism



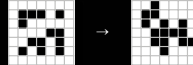
© 2010 Dave R. Zinner, The Australian National University page 423 of 758 (chapter 5: "Data Parallelism" up to page 427)

**Data Parallelism**

---

Vector Machines

General Data-parallelism



Cellular automaton transitions from a state  $S$  into the next state  $S'$ :  
 $S \rightarrow S' \Leftrightarrow \forall c \in S: c \rightarrow c' = r(S, c)$ , i.e. all cells of a state transition *concurrently* into new cells by following a rule  $r$ .

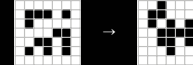
© 2010 Dave R. Zinner, The Australian National University page 424 of 758 (chapter 5: "Data Parallelism" up to page 427)

**Data Parallelism**

---

Vector Machines

General Data-parallelism



Cellular automaton transitions from a state  $S$  into the next state  $S'$ :  
 $S \rightarrow S' \Leftrightarrow \forall c \in S: c \rightarrow c' = r(S, c)$ , i.e. all cells of a state transition *concurrently* into new cells by following a rule  $r$ .

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

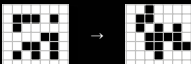
© 2010 Dave R. Zinner, The Australian National University page 425 of 758 (chapter 5: "Data Parallelism" up to page 427)

**Data Parallelism**

---

Vector Machines

General Data-parallelism



Cellular automaton transitions from a state  $S$  into the next state  $S'$ :  
 $S \rightarrow S' \Leftrightarrow \forall c \in S: c \rightarrow c' = r(S, c)$ , i.e. all cells of a state transition *concurrently* into new cells by following a rule  $r$ .

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

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

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

© 2010 Dave R. Zinner, The Australian National University page 426 of 758 (chapter 5: "Data Parallelism" up to page 427)

**Data Parallelism**

---


Summary

**Data Parallelism**

- **Data-Parallelism**
  - Vectorization
  - Reduction
  - General data-parallelism
- **Examples**
  - Image processing
  - Cellular automata

© 2010 Dave R. Zinner, The Australian National University page 427 of 758 (chapter 5: "Data Parallelism" up to page 427)

# Systems, Networks & Concurrency 2020



# 6

## Scheduling

Uwe R. Zimmer - The Australian National University

## Scheduling

### Motivation and definition of terms

#### Purpose of scheduling

Two scenarios for scheduling algorithms:

- Ordering resource assignments (CPU time, network access, ...).  
 ↳ live, on-line application of scheduling algorithms.
- Predicting system behaviours under anticipated loads.  
 ↳ simulated, off-line application of scheduling algorithms.

Predictions are used:

- at *compile time*: to confirm the feasibility of the system, or to predict resource needs, ...
- at *run time*: to permit admittance of new requests or for load-balancing, ...

## Scheduling

### Definition of terms

#### Time scales of scheduling

## Scheduling

### References for this chapter

[Ben2006]  
Ben-Ari, M.  
*Principles of Concurrent and Distributed Programming*  
second edition, Prentice-Hall 2006

[AdaRM2012]  
*Ada Reference Manual - Language and Standard Libraries*;  
ISO/IEC 8652:201x (E)

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

## Scheduling

### Motivation and definition of terms

#### Criteria

	Performance criteria:	Predictability criteria:
Process / user perspective:		
Waiting time	minimize the ...	minimize <i>deviation</i> from given ...
Response time	minima / maxima / average / variance	value / minima / maxima
Turnaround time	minima / maxima / average / variance	value / minima / maxima / deadlines
System perspective:		
Throughput	maximize the ...	
Utilization	minima / maxima / average CPU busy time	

## Scheduling

### Definition of terms

#### Time scales of scheduling

## Scheduling

### Motivation and definition of terms

#### Purpose of scheduling

## Scheduling

### Definition of terms

#### Time scales of scheduling

## Scheduling

### Performance scheduling

#### Requested resource times

Tasks have an average time between instantiations of and a constant computation time of

### Scheduling

#### Performance scheduling

##### First come, first served (FCFS)

Waiting time: 0..11, average: 5.9 – Turnaround time: 3..12, average: 8.4

As tasks apply *concurrently* for resources, the actual sequence of arrival is non-deterministic.  
 ↳ hence even a deterministic scheduling schema like FCFS can lead to different outcomes.

© 2020 Dave R. Zinner, The Australian National University page 437 of 758 (chapter 6: "Scheduling" up to page 439)

### Scheduling

#### Performance scheduling

##### First come, first served (FCFS)

Waiting time: 0..11, average: 5.4 – Turnaround time: 3..12, average: 8.0

↳ In this example:  
 the average waiting times vary between 5.4 and 5.9  
 the average turnaround times vary between 8.0 and 8.4

↳ **Shortest possible maximal turnaround time!**

© 2020 Dave R. Zinner, The Australian National University page 438 of 758 (chapter 6: "Scheduling" up to page 439)

### Scheduling

#### Performance scheduling

##### Round Robin (RR)

Waiting time: 0..5, average: 1.2 – Turnaround time: 1..20, average: 5.8

↳ Optimized for swift initial responses.  
 ↳ "Stretches out" long tasks.

↳ **Bound maximal waiting time!** (depended only on the number of tasks)

© 2020 Dave R. Zinner, The Australian National University page 439 of 758 (chapter 6: "Scheduling" up to page 439)

### Scheduling

#### Performance scheduling

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

- 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.

© 2020 Dave R. Zinner, The Australian National University page 440 of 758 (chapter 6: "Scheduling" up to page 439)

### Scheduling

#### Performance scheduling

##### Feedback with $2^i$ pre-emption intervals - sequential

Waiting time: 0..5, average: 1.5 – Turnaround time: 1..21, average: 5.7

↳ Optimized for swift initial responses.  
 ↳ Prefers short tasks and long tasks can suffer starvation.

↳ **Very short initial response times!** and good average turnaround times.

© 2020 Dave R. Zinner, The Australian National University page 441 of 758 (chapter 6: "Scheduling" up to page 439)

### Scheduling

#### Performance scheduling

##### Feedback with $2^i$ pre-emption intervals - overlapping

Waiting time: 0..3, average: 0.9 – Turnaround time: 1..45, average: 7.7

↳ Optimized for swift initial responses.  
 ↳ Prefers short tasks and long tasks can suffer **starvation**.  
 ↳ **Long tasks are delayed until all queues run empty!**

© 2020 Dave R. Zinner, The Australian National University page 442 of 758 (chapter 6: "Scheduling" up to page 439)

### Scheduling

#### Performance scheduling

##### Shortest job first

Waiting time: 0..11, average: 3.7 – Turnaround time: 1..14, average: 6.3

↳ Optimized for good average performance with minimal task-switches.  
 ↳ Prefers short tasks but all tasks will be handled.

↳ **Good choice if computation times are known and task switches are expensive!**

© 2020 Dave R. Zinner, The Australian National University page 443 of 758 (chapter 6: "Scheduling" up to page 439)

### Scheduling

#### Performance scheduling

##### Shortest job first

Waiting time: 0..10, average: 3.4 – Turnaround time: 1..14, average: 6.0

↳ Can be sensitive to non-deterministic arrival sequences.

© 2020 Dave R. Zinner, The Australian National University page 444 of 758 (chapter 6: "Scheduling" up to page 439)

### Scheduling

#### Performance scheduling

##### Highest Response Ratio $\frac{W_i + C_i}{C_i}$ First (HRRF)

Waiting time: 0..9, average: 4.1 – Turnaround time: 2..13, average: 6.6

↳ Blend between Shortest-Job-First and First-Come-First-Served.  
 ↳ Prefers short tasks but long tasks gain preference over time.

↳ **More task switches and worse averages than SJF but better upper bounds!**

© 2020 Dave R. Zinner, The Australian National University page 445 of 758 (chapter 6: "Scheduling" up to page 439)

### Scheduling

#### Performance scheduling

#### Shortest Remaining Time First (SRTF)

Waiting time: 0.6, average: 0.7 – Turnaround time: 1.21, average: 4.4

- Optimized for good averages.
- Prefers short tasks and long tasks can suffer starvation.
- Better averages than Feedback scheduling but with longer absolute waiting times!

© 2010 Dave R. Zinner, The Australian National University page 446 of 758 (chapter 6: "Scheduling" up to page 439)

### Scheduling

#### Performance scheduling

#### Comparison by shortest average waiting

- Providing short average waiting times
- Very swift response in most cases

© 2010 Dave R. Zinner, The Australian National University page 449 of 758 (chapter 6: "Scheduling" up to page 439)

### Scheduling

#### Performance scheduling

#### Comparison overview

	Selection	Pre-emption	Waiting	Turnaround	Preferred jobs	Starvation possible?
Methods without any knowledge about the processes						
FCFS	$\max(W_i)$	no	long	long average & short maximum	equal	no
RR	equal share	yes	bound	good average & large maximum	short	no
FB	priority queues	yes	very short	short average & long maximum	short	no
Methods employing computation time $C_i$ and elapsed time $E_i$						
SJF	$\min(C_i)$	no	medium	medium	short	yes
HRRF	$\max(\frac{W_i + C_i}{C_i})$	no	controllable compromise	controllable compromise	controllable	no
SRTF	$\min(C_i - E_i)$	yes	very short	wide variance	short	yes

© 2010 Dave R. Zinner, The Australian National University page 452 of 758 (chapter 6: "Scheduling" up to page 439)

### Scheduling

#### Performance scheduling

#### Comparison (in order of appearance)

- Providing upper bounds to turnaround times
- No tasks are left behind

© 2010 Dave R. Zinner, The Australian National University page 447 of 758 (chapter 6: "Scheduling" up to page 439)

### Scheduling

#### Performance scheduling

#### Comparison by shortest maximal turnaround

- Providing upper bounds to turnaround times
- No tasks are left behind

© 2010 Dave R. Zinner, The Australian National University page 450 of 758 (chapter 6: "Scheduling" up to page 439)

### Scheduling

#### Predictable scheduling

#### Towards predictable scheduling ...

Task requirements (Quality of service):

- Guarantee data flow levels
- Guarantee reaction times
- Guarantee deadlines
- Guarantee delivery times
- Provide bounds for the variations in results

Examples:

- Streaming media broadcasts, playing HD videos, live mixing audio/video, ...
- Reacting to users, Reacting to alarm situations, ...
- Delivering a signal to the physical world at the required time, ...

© 2010 Dave R. Zinner, The Australian National University page 453 of 758 (chapter 6: "Scheduling" up to page 439)

### Scheduling

#### Performance scheduling

#### Comparison by shortest maximal waiting

- Providing upper bounds to waiting times
- Swift response systems

© 2010 Dave R. Zinner, The Australian National University page 448 of 758 (chapter 6: "Scheduling" up to page 439)

### Scheduling

#### Performance scheduling

#### Comparison by shortest average turnaround

- Providing good average performance
- High throughput systems

© 2010 Dave R. Zinner, The Australian National University page 451 of 758 (chapter 6: "Scheduling" up to page 439)

### Scheduling

#### Predictable scheduling

#### Temporal scopes

Common attributes:

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

© 2010 Dave R. Zinner, The Australian National University page 454 of 758 (chapter 6: "Scheduling" up to page 439)

## Scheduling

### Predictable scheduling

#### Temporal scopes

Common attributes:

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

Task *i*

Timeline: 1 (created), 5, 10 (activated), 20, 25, 30

© 2010 Dave R. Zimmerman, The Australian National University page 455 of 758 (chapter 6: "Scheduling") up to page 459

## Scheduling

### Predictable scheduling

#### Common temporal scope attributes

Temporal scopes can be:

Periodic	↳ controllers, routers, schedulers, streaming processes, ...
Aperiodic	↳ periodic 'on average' tasks, i.e. regular but not rigidly timed, ...
Sporadic / Transient	↳ user requests, alarms, I/O interaction, ...

Deadlines can be:

Semantics defined by application	"Hard"	↳ single failure leads to severe malfunction and/or disaster
	"Firm"	↳ results are meaningless after the deadline
	"Soft"	↳ only multiple or permanent failures lead to malfunction

© 2010 Dave R. Zimmerman, The Australian National University page 458 of 758 (chapter 6: "Scheduling") up to page 459

## Scheduling

### Predictable scheduling

#### Temporal scopes

Common attributes:

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

Task *i*

Timeline: 1 (created), 5, 10 (activated), 20 (suspended), 25 (re-activated), 30 (terminated)

© 2010 Dave R. Zimmerman, The Australian National University page 456 of 758 (chapter 6: "Scheduling") up to page 459

## Scheduling

### Summary

#### Scheduling

- Basic performance scheduling
  - Motivation & Terms
  - Levels of knowledge / assumptions about the task set
  - Evaluation of performance and selection of appropriate methods
- Towards predictable scheduling
  - Motivation & Terms
  - Categories & Examples

© 2010 Dave R. Zimmerman, The Australian National University page 459 of 758 (chapter 6: "Scheduling") up to page 459

## Scheduling

### Predictable scheduling

#### Temporal scopes

Common attributes:


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

Task *i*

Timeline: 1 (created), 5, 10 (activated), 20 (suspended), 25 (re-activated), 30 (terminated)

© 2010 Dave R. Zimmerman, The Australian National University page 457 of 758 (chapter 6: "Scheduling") up to page 459

# Systems, Networks & Concurrency 2020



## Safety & Liveness

Uwe R. Zimmer - The Australian National University

# Safety & Liveness

## References for this chapter

[Ben2006]  
Ben-Ari, M  
*Principles of Concurrent and Distributed Programming*  
second edition, Prentice-Hall 2006

[Chandy1983]  
Chandy, K, Misra, Jayadev & Haas, Laura  
*Distributed deadlock detection*  
Transactions on Computer Systems (TOCS) 1983 vol. 1 (2)

[Silberschatz2001]  
Silberschatz, Abraham, Galvin, Peter & Gagne, Greg  
*Operating System Concepts*  
John Wiley & Sons, Inc., 2001

© 2020 Uwe R. Zimmer, The Australian National University page 461 of 758 (chapter 7: "Safety & Liveness" up to page 461)

# Safety & Liveness

## Repetition

### Correctness concepts in concurrent systems

Extended concepts of correctness in concurrent systems:  
→ Termination is often not intended or even considered a failure

**Safety properties:**  
 $(P(I) \wedge \text{Processes}(I, S)) \Rightarrow \Box Q(I, S)$   
where  $\Box Q$  means that  $Q$  does always hold

**Liveness properties:**  
 $(P(I) \wedge \text{Processes}(I, S)) \Rightarrow \Diamond Q(I, S)$   
where  $\Diamond Q$  means that  $Q$  does eventually hold (and will then stay true) and  $S$  is the current state of the concurrent system

© 2020 Uwe R. Zimmer, The Australian National University page 462 of 758 (chapter 7: "Safety & Liveness" up to page 462)

# Safety & Liveness

## Repetition

### Correctness concepts in concurrent systems

**Liveness properties:**  
 $(P(I) \wedge \text{Processes}(I, S)) \Rightarrow \Diamond Q(I, S)$   
where  $\Diamond Q$  means that  $Q$  does eventually hold (and will then stay true)

Examples:

- Requests need to complete eventually.
- The state of the system needs to be displayed eventually.
- No part of the system is to be delayed forever (fairness).

☞ Interesting liveness properties can become very hard to proof

© 2020 Uwe R. Zimmer, The Australian National University page 463 of 758 (chapter 7: "Safety & Liveness" up to page 463)

# Safety & Liveness

## Liveness

### Fairness

**Liveness properties:**  
 $(P(I) \wedge \text{Processes}(I, S)) \Rightarrow \Diamond Q(I, S)$   
where  $\Diamond Q$  means that  $Q$  does eventually hold (and will then stay true)

**Fairness** (as a means to avoid starvation): Resources will be granted ...

- **Weak fairness:**  $\Diamond \Box R \Rightarrow \Diamond G$  ... eventually, if a process requests continually.
- **Strong fairness:**  $\Box \Diamond R \Rightarrow \Diamond G$  ... eventually, if a process requests infinitely often.
- **Linear waiting:**  $\Diamond R \Rightarrow \Diamond G$  ... before any other process had the same resource granted more than once (common fairness in distributed systems).
- **First-in, first-out:**  $\Diamond R \Rightarrow \Diamond G$  ... before any other process which applied for the same resource at a later point in time (common fairness in single-node systems).

© 2020 Uwe R. Zimmer, The Australian National University page 464 of 758 (chapter 7: "Safety & Liveness" up to page 464)

# Safety & Liveness

## Revisiting

### Correctness concepts in concurrent systems

**Safety properties:**  
 $(P(I) \wedge \text{Processes}(I, S)) \Rightarrow \Box Q(I, S)$   
where  $\Box Q$  means that  $Q$  does always hold

Examples:

- **Mutual exclusion** (no resource collisions) ☞ has been addressed
- **Absence of deadlocks** ☞ to be addressed now (and other forms of 'silent death' and 'freeze' conditions)
- **Specified responsiveness** or free capabilities ☞ Real-time systems (typical in real-time / embedded systems or server applications)

© 2020 Uwe R. Zimmer, The Australian National University page 465 of 758 (chapter 7: "Safety & Liveness" up to page 465)

# Safety & Liveness

## Deadlocks

### Most forms of synchronization may lead to

## Deadlocks

(Avoidance / prevention of deadlocks is one central safety property)

- ☞ How to predict them?
- ☞ How to find them?
- ☞ How to resolve them?
- ☞ ... or are there structurally dead-lock free forms of synchronization?

© 2020 Uwe R. Zimmer, The Australian National University page 466 of 758 (chapter 7: "Safety & Liveness" up to page 466)

# Safety & Liveness

## Towards synchronization

### 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 all resources
signal (reserve_2);
signal (reserve_1);
statement Z;
end P1;
```

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

Sequence of operations:  $A \rightarrow B \rightarrow C; X \rightarrow Y \rightarrow Z; [X, Z \mid A, B, C]; [A, C \mid X, Y, Z]; \neg[B \mid Y]$   
or:  $[A \mid X]$  followed by a deadlock situation.

© 2020 Uwe R. Zimmer, The Australian National University page 467 of 758 (chapter 7: "Safety & Liveness" up to page 467)

# Safety & Liveness

## Towards synchronization

### 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:  $A \rightarrow B \rightarrow C; X \rightarrow Y \rightarrow Z; K \rightarrow L \rightarrow M; [X, Z \mid A, B, C \mid K, M]; [A, C \mid X, Y, Z \mid K, M]; [A, C \mid X, Z]; \neg[B \mid Y \mid L]$   
or:  $[A \mid X \mid K]$  followed by a deadlock situation.

© 2020 Uwe R. Zimmer, The Australian National University page 468 of 758 (chapter 7: "Safety & Liveness" up to page 468)

**Safety & Liveness**

**Deadlocks**

*Necessary deadlock conditions:*

- Mutual exclusion:**  
resources cannot be used simultaneously.

© 2020 Uwe R. Zimmer, The Australian National University page: 469 of 758 (chapter 7: "Safety & Liveness" up to page 513)

**Safety & Liveness**

**Deadlocks**

*Necessary deadlock conditions:*

- Mutual exclusion:**  
resources cannot be used simultaneously.
- Hold and wait:**  
a process applies for a resource, while it is holding another resource (sequential requests).

© 2020 Uwe R. Zimmer, The Australian National University page: 470 of 758 (chapter 7: "Safety & Liveness" up to page 513)

**Safety & Liveness**

**Deadlocks**

*Necessary deadlock conditions:*

- Mutual exclusion:**  
resources cannot be used simultaneously.
- Hold and wait:**  
a process applies for a resource, while it is holding another resource (sequential requests).
- No pre-emption:**  
resources cannot be pre-empted; only the process itself can release resources.

© 2020 Uwe R. Zimmer, The Australian National University page: 471 of 758 (chapter 7: "Safety & Liveness" up to page 513)

**Safety & Liveness**

**Deadlocks**

*Necessary deadlock conditions:*

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

© 2020 Uwe R. Zimmer, The Australian National University page: 472 of 758 (chapter 7: "Safety & Liveness" up to page 513)

**Safety & Liveness**

**Deadlocks**

*Necessary deadlock conditions:*

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

☞ A system *may* become deadlocked, if *all* these conditions apply!

© 2020 Uwe R. Zimmer, The Australian National University page: 473 of 758 (chapter 7: "Safety & Liveness" up to page 513)

**Safety & Liveness**

**Deadlocks**

*Deadlock strategies:*

- Ignorance & restart
  - ☞ Kill or restart unresponsive processes, power-cycle the computer, ...
- Deadlock detection & recovery
  - ☞ find deadlocked processes and recover the system in a coordinated way
- Deadlock avoidance
  - ☞ the resulting system state is checked before any resources are actually assigned
- Deadlock prevention
  - ☞ the system prevents deadlocks by its structure

© 2020 Uwe R. Zimmer, The Australian National University page: 474 of 758 (chapter 7: "Safety & Liveness" up to page 513)

**Safety & Liveness**

**Deadlocks**

**Deadlock prevention**  
(Remove one of the four necessary deadlock conditions)

- Break Mutual exclusion:**

Mutual exclusion  
 Hold and wait  
 No pre-emption  
 Circular wait

© 2020 Uwe R. Zimmer, The Australian National University page: 475 of 758 (chapter 7: "Safety & Liveness" up to page 513)

**Safety & Liveness**

**Deadlocks**

**Deadlock prevention**  
(Remove one of the four necessary deadlock conditions)

- Break Mutual exclusion:**  
By replicating critical resources, mutual exclusion becomes unnecessary (only applicable in very specific cases).
- Break Hold and wait:**

Mutual exclusion  
 Hold and wait  
 No pre-emption  
 Circular wait

© 2020 Uwe R. Zimmer, The Australian National University page: 476 of 758 (chapter 7: "Safety & Liveness" up to page 513)

**Safety & Liveness**

**Deadlocks**

**Deadlock prevention**  
(Remove one of the four necessary deadlock conditions)

- Break Mutual exclusion:**  
By replicating critical resources, mutual exclusion becomes unnecessary (only applicable in very specific cases).
- Break Hold and wait:**  
Allocation of all required resources in one request.  
Processes can either hold *none* or *all* of their required resources.
- Introduce Pre-emption:**

Mutual exclusion  
 Hold and wait  
 No pre-emption  
 Circular wait

© 2020 Uwe R. Zimmer, The Australian National University page: 477 of 758 (chapter 7: "Safety & Liveness" up to page 513)



## Safety & Liveness

### Deadlocks

#### Deadlock prevention

(Remove one of the four necessary deadlock conditions)

- Break Mutual exclusion:**  
By replicating critical resources, mutual exclusion becomes unnecessary (only applicable in very specific cases).
- Break Hold and wait:**  
Allocation of all required resources in one request. Processes can either hold none or all of their required resources.
- Introduce Pre-emption:**  
Provide the additional infrastructure to allow for pre-emption of resources. Mind that resources cannot be pre-empted, if their states cannot be fully stored and recovered.
- Break Circular waits:**

Mutual exclusion  
Hold and wait  
No pre-emption  
Circular wait

© 2020 Uwe R. Zimmer, The Australian National University page: 478 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Deadlocks

#### Deadlock prevention

(Remove one of the four necessary deadlock conditions)

- Break Mutual exclusion:**  
By replicating critical resources, mutual exclusion becomes unnecessary (only applicable in very specific cases).
- Break Hold and wait:**  
Allocation of all required resources in one request. Processes can either hold none or all of their required resources.
- Introduce Pre-emption:**  
Provide the additional infrastructure to allow for pre-emption of resources. Mind that resources cannot be pre-empted, if their states cannot be fully stored and recovered.
- Break Circular waits:**  
E.g. order all resources globally and restrict processes to request resources in that order only.

Mutual exclusion  
Hold and wait  
No pre-emption  
Circular wait

© 2020 Uwe R. Zimmer, The Australian National University page: 479 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Deadlocks

#### Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

$RAG = \{V, E\}$ ; Resource allocation graphs consist of vertices  $V$  and edges  $E$ .  
 $V = P \cup R$ ; Vertices  $V$  can be processes  $P$  or Resource types  $R$ .  
 with processes  $P = \{P_1, \dots, P_n\}$   
 and resources types  $R = \{R_1, \dots, R_k\}$

$E = E_c \cup E_r \cup E_a$ ; Edges  $E$  can be "claims"  $E_c$ , "requests"  $E_r$  or "assignments"  $E_a$   
 with claims  $E_c = \{P_i \rightarrow R_j, \dots\}$   
 requests  $E_r = \{P_i \rightarrow R_j, \dots\}$   
 and assignments  $E_a = \{R_j \rightarrow P_i, \dots\}$

Note: any resource type  $R_j$  can have more than one instance of a resource.

© 2020 Uwe R. Zimmer, The Australian National University page: 480 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Deadlocks

#### Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

© 2020 Uwe R. Zimmer, The Australian National University page: 481 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Deadlocks

#### Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

☞ Two process, reverse allocation deadlock:

© 2020 Uwe R. Zimmer, The Australian National University page: 482 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Deadlocks

#### Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

© 2020 Uwe R. Zimmer, The Australian National University page: 483 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Deadlocks

#### Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

☞ No circular dependency ☞ no deadlock:

© 2020 Uwe R. Zimmer, The Australian National University page: 484 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Deadlocks

#### Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

© 2020 Uwe R. Zimmer, The Australian National University page: 485 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Deadlocks

#### Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

☞ Two circular dependencies ☞ deadlock:  
 $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$   
 as well as:  $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

Derived rule:  
 If some processes are deadlocked then there are cycles in the resource allocation graph.

© 2020 Uwe R. Zimmer, The Australian National University page: 486 of 758 (chapter 7: "Safety & Liveness" up to page 513)

### Safety & Liveness

#### Deadlocks

#### Edge Chasing

(for the distributed version see Chandy, Misra & Haas)

blocking processes:

- Send a probe to all requested yet unassigned resources containing ids of: [the blocked, the sending, the targeted node].

nodes on probe reception:

- Propagate the probe to all processes holding the critical resources or to all requested yet unassigned resources – while updating the second and third entry in the probe.

a process receiving its own probe:  
(blocked-id = targeted-id)

- Circular dependency detected.

© 2020 Uwe R. Zimmer, The Australian National University page 487 of 758 (chapter 7: "Safety & Liveness" up to page 513)

### Safety & Liveness

#### Deadlocks

#### Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

Knowledge of claims:

Claims are potential future requests which have no blocking effect on the claiming process – while actual requests are blocking.

© 2020 Uwe R. Zimmer, The Australian National University page 488 of 758 (chapter 7: "Safety & Liveness" up to page 513)

### Safety & Liveness

#### Deadlocks

#### Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

Assignment of resources such that circular dependencies are avoided:

© 2020 Uwe R. Zimmer, The Australian National University page 489 of 758 (chapter 7: "Safety & Liveness" up to page 513)

### Safety & Liveness

#### Deadlocks

#### Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

Earlier derived rule:

If some processes are deadlocked then there are cycles in the resource allocation graph.

- Reverse rule for multiple instances: If there are cycles in the resource allocation graph and there are multiple instances per resource then the involved processes are potentially deadlocked.
- Reverse rule for single instances: If there are cycles in the resource allocation graph and there is exactly one instance per resource then the involved processes are deadlocked.

© 2020 Uwe R. Zimmer, The Australian National University page 490 of 758 (chapter 7: "Safety & Liveness" up to page 513)

### Safety & Liveness

#### Deadlocks

#### Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

Reverse rule for single instances:

If there are cycles in the resource allocation graph and there is exactly one instance per resource then the involved processes are deadlocked.

- Actual deadlock identified

© 2020 Uwe R. Zimmer, The Australian National University page 491 of 758 (chapter 7: "Safety & Liveness" up to page 513)

### Safety & Liveness

#### Deadlocks

#### Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

Reverse rule for multiple instances:

If there are cycles in the resource allocation graph and there are multiple instances per resource then the involved processes are potentially deadlocked.

- Potential deadlock identified

© 2020 Uwe R. Zimmer, The Australian National University page 492 of 758 (chapter 7: "Safety & Liveness" up to page 513)

### Safety & Liveness

#### Deadlocks

#### Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

Reverse rule for multiple instances:

If there are cycles in the resource allocation graph and there are multiple instances per resource then the involved processes are potentially deadlocked.

- Potential deadlock identified – yet clearly not an actual deadlock here

© 2020 Uwe R. Zimmer, The Australian National University page 493 of 758 (chapter 7: "Safety & Liveness" up to page 513)

### Safety & Liveness

#### Deadlocks

#### Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

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

(multiple instances per resource)

© 2020 Uwe R. Zimmer, The Australian National University page 494 of 758 (chapter 7: "Safety & Liveness" up to page 513)

### Safety & Liveness

#### Deadlocks

#### Banker's Algorithm

There are processes  $P_i \in \{P_1, \dots, P_n\}$  and resource types  $R_j \in \{R_1, \dots, R_m\}$  and data structures:

- Allocated [i, j]
  - the number of resources of type j currently allocated to process i.
- Free [j]
  - the number of currently available resources of type j.
- Claimed [i, j]
  - the number of resources of type j required by process i eventually.
- Requested [i, j]
  - the number of currently requested resources of type j by process i.
- Completed [i]
  - boolean vector indicating processes which may complete.
- Simulated\_Free [j]
  - Number of available resources assuming that complete processes deallocate their resources.

© 2020 Uwe R. Zimmer, The Australian National University page 495 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Deadlocks

### Banker's Algorithm

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:  
     $\forall j$ : Simulated\_Free [j]  $\leftarrow$  Simulated\_Free [j] + Allocated [i, j];  
    Completed [i]  $\leftarrow$  True;
3. If  $\forall i$ : Completed [i] then the system is currently **deadlock-free!**  
else all processes i with  $\neg$ Completed [i] are involved in a **deadlock!**.

© 2020 Uwe R. Zimmer, The Australian National University page 496 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Deadlocks

### Banker's Algorithm

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:  
     $\forall j$ : 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!**

A **safe** system is a system in which future deadlocks can be avoided assuming the current set of available resources.

© 2020 Uwe R. Zimmer, The Australian National University page 497 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Deadlocks

### Banker's Algorithm

Check potential future system safety by simulating a granted request:  
(**Deadlock avoidance**)

```

if (Request < Claimed) and (Request < Free) then
  Free := Free - Request;
  Claimed := Claimed - Request;
  Allocated := Allocated + Request;
  if System_is_safe (checked by e.g. Banker's algorithm) then
    Grant request
  else
    Restore former system state: (Free, Claimed, Allocated)
end if;
end if;

```

© 2020 Uwe R. Zimmer, The Australian National University page 498 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Deadlocks

### Distributed deadlock detection

Observation: Deadlock detection methods like Banker's Algorithm are too communication intensive to be commonly applied in full and at high frequency in a distributed system.

Therefore a distributed version needs to:

- Split the system into nodes of reasonable locality (keeping most processes close to the resources they require).
- Organize the nodes in an adequate topology (e.g. a tree).
- Check for deadlock inside nodes with blocked resource requests and detect/avoid local deadlock *immediately*.
- Exchange resource status information between nodes occasionally and detect **global deadlocks eventually**.

© 2020 Uwe R. Zimmer, The Australian National University page 499 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Deadlocks

### Deadlock recovery

A deadlock has been detected now what?

Breaking the circular dependencies can be done by:

- Either *pre-empt* an assigned resource which is part of the deadlock.
- or *stop* a process which is part of the deadlock.

Usually neither choice can be implemented 'gracefully' and deals only with the symptoms.

Deadlock recovery does not address the reason for the problem!  
(i.e. the deadlock situation can re-occur again immediately)

© 2020 Uwe R. Zimmer, The Australian National University page 500 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Deadlocks

### Deadlock strategies:

- **Deadlock prevention**  
System prevents deadlocks by its structure or by full verification. *The best approach if applicable.*
- **Deadlock avoidance**  
System state is checked with every resource assignment. *More generally applicable, yet computationally very expensive.*
- **Deadlock detection & recovery**  
Detect deadlocks and break them in a 'coordinated' way. *Less computationally expensive (as lower frequent), yet usually 'messy'.*
- **Ignorance & random kill**  
Kill or restart unresponsive processes, power-cycle the computer, ... *More of a panic reaction than a method.*

© 2020 Uwe R. Zimmer, The Australian National University page 501 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Atomic & idempotent operations

### Atomic operations

Definitions of atomicity:

An operation is atomic if the processes performing it ...

- (by 'awareness') ... are not aware of the existence of any other active process, and no other active process is aware of the activity of the processes during the time the processes are performing the atomic operation.
- (by communication) ... do not communicate with other processes while the atomic operation is performed.
- (by means of states) ... cannot detect any outside state change and do not reveal their own state changes until the atomic operation is complete.

Short:

An atomic operation can be considered to be **indivisible and instantaneous.**

© 2020 Uwe R. Zimmer, The Australian National University page 502 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Atomic & idempotent operations

### Atomic operations

© 2020 Uwe R. Zimmer, The Australian National University page 503 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Atomic & idempotent operations

### Atomic operations

Important implications:

1. An atomic operation is either performed *in full* or *not at all*.
2. A failed atomic operation cannot have any impact on its surroundings (must keep or re-instantiate the full initial state).
3. If any part of an atomic operation fails, then the whole atomic operation is declared failed.
4. All parts of an atomic operations (including already completed parts) must be prepared to declare failure until the final global commitment.

© 2020 Uwe R. Zimmer, The Australian National University page 504 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Atomic & idempotent operations

### Idempotent operations

Definition of idempotent operations:  
An operation is idempotent if the observable effect of the operation are identical for the cases of executing the operation:

- once,
- multiple times,
- infinitely often.

Observations:

- Idempotent operations are often atomic, but do not need to be.
- Atomic operations do not need to be idempotent.
- Idempotent operations can ease the requirements for synchronization.

© 2020 Uwe R. Zimmer, The Australian National University page 505 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Reliability, failure & tolerance

### 'Terminology of failure' or 'Failing terminology'?

**Reliability** ::= measure of success  
with which a system conforms to its *specification*.  
::= low failure rate.

**Failure** ::= a deviation of a system from its *specification*.

**Error** ::= the system state which leads to a failure.

**Fault** ::= the reason for an error.

© 2020 Uwe R. Zimmer, The Australian National University page 506 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Reliability, failure & tolerance

### Faults during different phases of design

- Inconsistent or inadequate specifications  
⊗ frequent source for disastrous faults
- Software design errors  
⊗ frequent source for disastrous faults
- Component & communication system failures  
⊗ rare and mostly predictable

© 2020 Uwe R. Zimmer, The Australian National University page 507 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Reliability, failure & tolerance

### Faults in the logic domain

- Non-termination / -completion  
Systems 'frozen' in a deadlock state, blocked for missing input, or in an infinite loop  
⊗ Watchdog timers required to handle the failure
- Range violations and other inconsistent states  
⊗ Run-time environment level exception handling required to handle the failure
- Value violations and other wrong results  
⊗ User-level exception handling required to handle the failure

© 2020 Uwe R. Zimmer, The Australian National University page 508 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Reliability, failure & tolerance

### Faults in the time domain

- Transient faults  
⊗ Single 'glitches', interference, ... very hard to handle
- Intermittent faults  
⊗ Faults of a certain regularity ... require careful analysis
- Permanent faults  
⊗ Faults which stay ... the easiest to find

© 2020 Uwe R. Zimmer, The Australian National University page 509 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Reliability, failure & tolerance

### Observable failure modes

```

graph TD
    FM[Failure modes] --> FN[fail never]
    FM --> TD[Time domain]
    FM --> VD[Value domain]
    FM --> FU[fail uncontrolled]
    TD --> TE[too early]
    TD --> TL[too late]
    TD --> NO[never (omission)]
    VD --> CE[Constraint error]
    VD --> VE[Value error]
    NO --> FS[fail silent]
    NO --> FST[fail stop]
    NO --> FCC[fail controlled]
  
```

© 2020 Uwe R. Zimmer, The Australian National University page 510 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Reliability, failure & tolerance

### Fault prevention, avoidance, removal, ...

and / or

⊗ **Fault tolerance**

© 2020 Uwe R. Zimmer, The Australian National University page 511 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

### Reliability, failure & tolerance

### Fault tolerance

- Full fault tolerance  
the system continues to operate in the presence of 'foreseeable' error conditions, without any significant loss of functionality or performance — even though this might reduce the achievable total operation time.
- Graceful degradation (fail soft)  
the system continues to operate in the presence of 'foreseeable' error conditions, while accepting a partial loss of functionality or performance.
- Fail safe  
the system halts and maintains its integrity.

⊗ Full fault tolerance is not maintainable for an infinite operation time!  
⊗ Graceful degradation might have multiple levels of reduced functionality.

© 2020 Uwe R. Zimmer, The Australian National University page 512 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Safety & Liveness

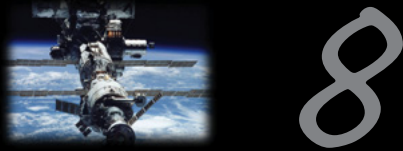
### Summary

### Safety & Liveness

- Liveness
  - Fairness
- Safety
  - Deadlock detection
  - Deadlock avoidance
  - Deadlock prevention
- Atomic & Idempotent operations
  - Definitions & implications
- Failure modes
  - Definitions, fault sources and basic fault tolerance

© 2020 Uwe R. Zimmer, The Australian National University page 513 of 758 (chapter 7: "Safety & Liveness" up to page 513)

## Systems, Networks & Concurrency 2020



### Distributed Systems

Uwe R. Zimmer - The Australian National University

## Distributed Systems

### References for this chapter

[Bacon1998]

Bacon, J  
Concurrent Systems  
Addison Wesley Longman  
Ltd (2nd edition) 1998

[Ben2006]

Ben-Ari, M  
Principles of Concurrent and Distributed Programming  
second edition, Prentice-Hall 2006

[Schneider1990]

Schneider, Fred  
Implementing fault-tolerant services using the state machine approach: a tutorial  
ACM Computing Surveys 1990  
vol. 22 (4) pp. 299-319

[Tanenbaum2001]

Tanenbaum, Andrew  
Distributed Systems: Principles and Paradigms  
Prentice Hall 2001

[Tanenbaum2003]

Tanenbaum, Andrew  
Computer Networks  
Prentice Hall, 2003

© 2020 Uwe R. Zimmer, The Australian National University

page 515 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Network protocols & standards

#### OSI network reference model

Standardized as the

**Open Systems Interconnection (OSI)** reference model by the International Standardization Organization (ISO) in 1977

- 7 layer architecture
- Connection oriented

Hardly implemented anywhere in full ...

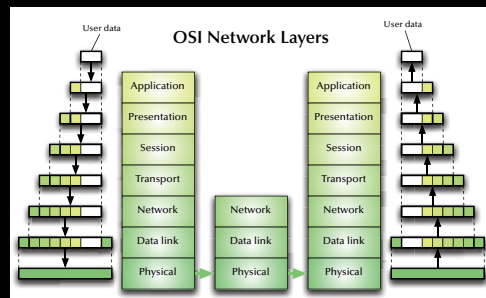
...but its **concepts and terminology** are widely used, when describing existing and designing new protocols ...

© 2020 Uwe R. Zimmer, The Australian National University

page 516 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Network protocols & standards



© 2020 Uwe R. Zimmer, The Australian National University

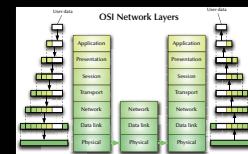
page 517 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Network protocols & standards

#### 1: Physical Layer

- **Service:** Transmission of a raw bit stream over a communication channel
- **Functions:** Conversion of bits into electrical or optical signals
- **Examples:** X.21, Ethernet (cable, detectors & amplifiers)



© 2020 Uwe R. Zimmer, The Australian National University

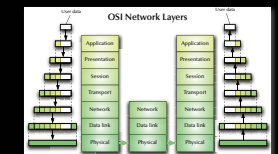
page 518 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Network protocols & standards

#### 2: Data Link Layer

- **Service:** Reliable transfer of frames over a link
- **Functions:** Synchronization, error correction, flow control
- **Examples:** HDLC (high level data link control protocol), LAP-B (link access procedure, balanced), LAP-D (link access procedure, D-channel), LLC (link level control), ...



© 2020 Uwe R. Zimmer, The Australian National University

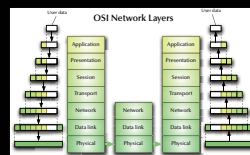
page 519 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Network protocols & standards

#### 3: Network Layer

- **Service:** Transfer of packets inside the network
- **Functions:** Routing, addressing, switching, congestion control
- **Examples:** IP, X.25



© 2020 Uwe R. Zimmer, The Australian National University

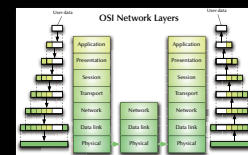
page 520 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Network protocols & standards

#### 4: Transport Layer

- **Service:** Transfer of data between hosts
- **Functions:** Connection establishment, management, termination, flow-control, multiplexing, error detection
- **Examples:** TCP, UDP, ISO TP0-TP4



© 2020 Uwe R. Zimmer, The Australian National University

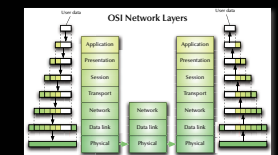
page 521 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Network protocols & standards

#### 5: Session Layer

- **Service:** Coordination of the dialogue between application programs
- **Functions:** Session establishment, management, termination
- **Examples:** RPC

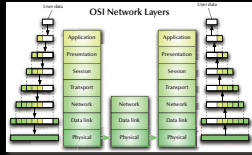


© 2020 Uwe R. Zimmer, The Australian National University

page 522 of 758 (chapter 8: "Distributed Systems" up to page 641)

### Distributed Systems

#### Network protocols & standards



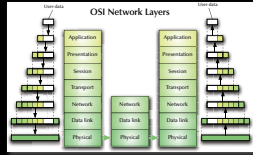
## 6: Presentation Layer

- **Service:** Provision of platform independent coding and encryption
- **Functions:** Code conversion, encryption, virtual devices
- **Examples:** ISO code conversion, PGP encryption

© 2020 Univ. R. Ziemec, The Australian National University page 523 of 758 (chapter 8: "Distributed Systems" - up to page 641)

### Distributed Systems

#### Network protocols & standards



## 7: Application Layer

- **Service:** Network access for application programs
- **Functions:** Application/OS specific
- **Examples:** APIs for mail, ftp, ssh, scp, discovery protocols ...



© 2020 Univ. R. Ziemec, The Australian National University page 524 of 758 (chapter 8: "Distributed Systems" - up to page 641)

### Distributed Systems

#### Network protocols & standards

## Serial Peripheral Interface (SPI)

- Used by gazillions of devices ... and it's not even a formal standard!
- Speed only limited by what both sides can survive.
- Usually push-pull drivers, i.e. fast and reliable, yet not friendly to wrong wiring/programming.

1.8" COLOR TFT LCD display from Adruino SanDisk marketing photo

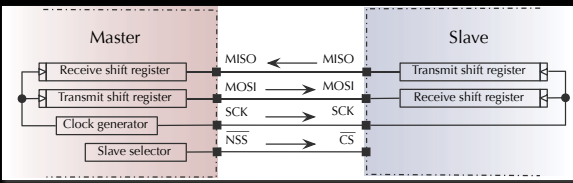
© 2020 Univ. R. Ziemec, The Australian National University page 525 of 758 (chapter 8: "Distributed Systems" - up to page 641)

### Distributed Systems

#### Network protocols & standards

## Serial Peripheral Interface (SPI)

Full Duplex, 4-wire, flexible clock rate

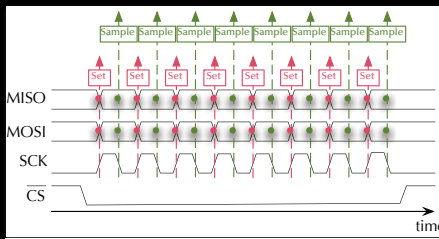


© 2020 Univ. R. Ziemec, The Australian National University page 526 of 758 (chapter 8: "Distributed Systems" - up to page 641)

### Distributed Systems

#### Network protocols & standards

## Serial Peripheral Interface (SPI)



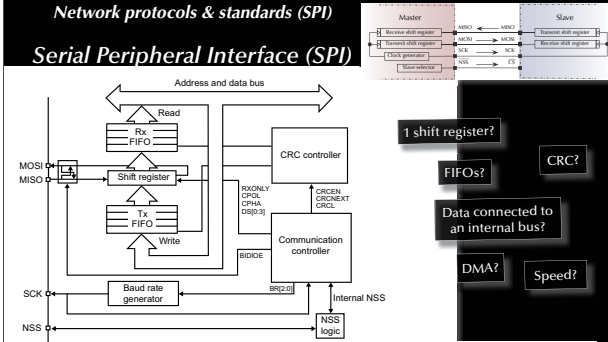
Clock phase and polarity need to be agreed upon

© 2020 Univ. R. Ziemec, The Australian National University page 527 of 758 (chapter 8: "Distributed Systems" - up to page 641)

### Distributed Systems

#### Network protocols & standards (SPI)

## Serial Peripheral Interface (SPI)

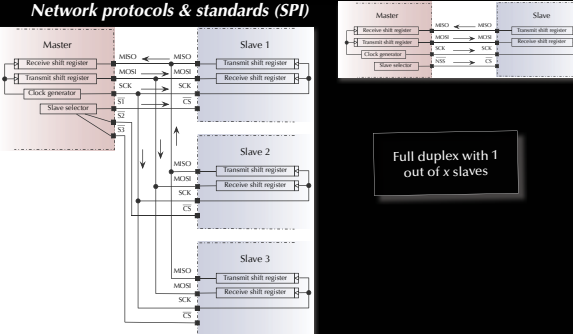


1 shift register?  
FIFOs?  
CRC?  
Data connected to an internal bus?  
DMA?  
Speed?

© 2020 Univ. R. Ziemec, The Australian National University page 528 of 758 (chapter 8: "Distributed Systems" - up to page 641)

### Distributed Systems

#### Network protocols & standards (SPI)

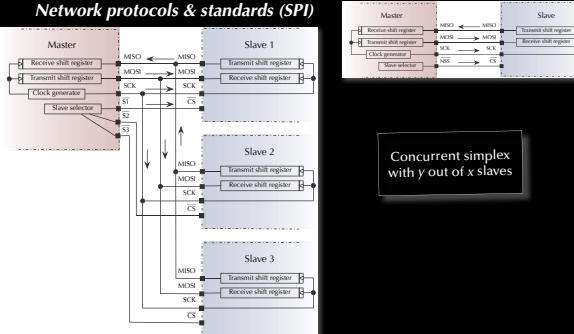


Full duplex with 1 out of x slaves

© 2020 Univ. R. Ziemec, The Australian National University page 529 of 758 (chapter 8: "Distributed Systems" - up to page 641)

### Distributed Systems

#### Network protocols & standards (SPI)

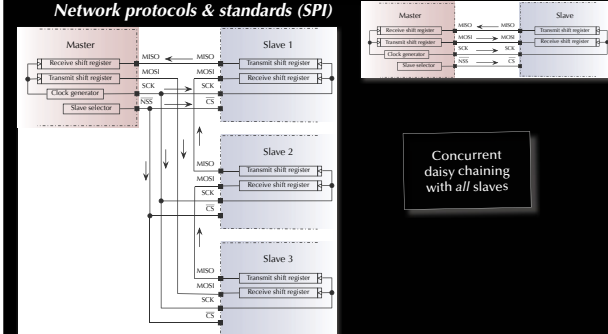


Concurrent simplex with y out of x slaves

© 2020 Univ. R. Ziemec, The Australian National University page 530 of 758 (chapter 8: "Distributed Systems" - up to page 641)

### Distributed Systems

#### Network protocols & standards (SPI)



Concurrent daisy chaining with all slaves

© 2020 Univ. R. Ziemec, The Australian National University page 531 of 758 (chapter 8: "Distributed Systems" - up to page 641)

## Distributed Systems

### Network protocols & standards

The diagram shows three vertical stacks of layers. The first and third stacks are labeled 'OSI' and the middle one 'TCP/IP'. Each stack has seven layers: Application, Presentation, Session, Transport, Network, Data link, and Physical. Arrows indicate data flow between these stacks.

© 2010 Dave R. Zinner, The Australian National University page 532 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Network protocols & standards

### Ethernet / IEEE 802.3

Local area network (LAN) developed by Xerox in the 70's

- 10Mbps specification 1.0 by DEC, Intel, & Xerox in 1980.
- First standard as IEEE 802.3 in 1983 (10Mbps over thick co-ax cables).
- currently 1 Gbps (802.3ab) copper cable ports used in most desktops and laptops.
- currently standards up to 100Gbps (IEEE 802.3ba 2010).
- more than 85% of current LAN lines worldwide (according to the International Data Corporation (IDC)).

☞ **Carrier Sense Multiple Access with Collision Detection (CSMA/CD)**

© 2010 Dave R. Zinner, The Australian National University page 535 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Network protocols & standards

### Ethernet / IEEE 802.11

Wireless local area network (WLAN) developed in the 90's

- First standard as IEEE 802.11 in 1997 (1-2Mbps over 2.4GHz).
- Typical usage at 54Mbps over 2.4GHz carrier at 20MHz bandwidth.
- Current standards up to 780 Mbps (802.11ac) over 5GHz carrier at 160 MHz bandwidth.
- Future standards are designed for up to 100Gbps over 60 GHz carrier.
- Direct relation to IEEE 802.3 and similar OSI layer association.

☞ **Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA)**

☞ **Direct-Sequence Spread Spectrum (DSSS)**

© 2010 Dave R. Zinner, The Australian National University page 538 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Network protocols & standards

The diagram maps AppleTalk protocols to the seven layers of the OSI model. AppleTalk protocols include: AppleTalk Filing Protocol (AFP), AT Data Stream Protocol, AT Session Protocol, Zone Info Protocol, Printer Access Protocol, Routing Table Maintenance Prot., AT Update Based Routing Protocol, Name Binding Prot., AT Transaction Protocol, AT Echo Protocol, Datagram Delivery Protocol (DDP), and AppleTalk Address Resolution Protocol (AARP). Physical layer protocols include IEEE 802.3, LocalTalk, Token Ring IEEE 802.5, and FDDI.

© 2010 Dave R. Zinner, The Australian National University page 533 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Network protocols & standards

### Ethernet / IEEE 802.3

OSI relation: PHY, MAC, MAC-client

The diagram shows the mapping of IEEE 802.3 protocols to the OSI model. It highlights the PHY (Physical) layer, MAC (Media Access Control) layer, and MAC-client layer. The OSI reference model and IEEE 802.3 reference model are compared.

© 2010 Dave R. Zinner, The Australian National University page 536 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Network protocols & standards

### Bluetooth

Wireless local area network (WLAN) developed in the 90's with different features than 802.11:

- Lower power consumption.
- Shorter ranges.
- Lower data rates (typically < 1 Mbps).
- Ad-hoc networking (no infrastructure required).

☞ **Combinations of 802.11 and Bluetooth OSI layers are possible to achieve the required features set.**

© 2010 Dave R. Zinner, The Australian National University page 539 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Network protocols & standards

The diagram shows how AppleTalk over IP maps to the OSI model. It includes protocols like AppleTalk Filing Protocol (AFP), AT Data Stream Protocol, AT Session Protocol, Zone Info Protocol, Printer Access Protocol, Routing Table Maintenance Prot., AT Update Based Routing Protocol, Name Binding Protocol, AT Transaction Protocol, AT Echo Protocol, Datagram Delivery Protocol (DDP), and AppleTalk Address Resolution Protocol (AARP).

© 2010 Dave R. Zinner, The Australian National University page 534 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Network protocols & standards

### Ethernet / IEEE 802.3

OSI relation: PHY, MAC, MAC-client

This diagram details the IEEE 802.3 network layers and physical components. It shows the mapping of MAC Client, 802.3 MAC, Physical medium-independent layer, MII, PHY, and MDI to the OSI model layers. It also illustrates the link, medium, and transmission rate.

© 2010 Dave R. Zinner, The Australian National University page 537 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Network protocols & standards

### Token Ring / IEEE 802.5 / Fibre Distributed Data Interface (FDDI)

- "Token Ring" developed by IBM in the 70's
- IEEE 802.5 standard is modelled after the IBM Token Ring architecture (specifications are slightly different, but basically compatible)
- IBM Token Ring requests are star topology as well as twisted pair cables, while IEEE 802.5 is unspecified in topology and medium
- Fibre Distributed Data Interface combines a token ring architecture with a dual-ring, fibre-optical, physical network.

☞ **Unlike CSMA/CD, Token ring is deterministic (with respect to its timing behaviour)**

☞ **FDDI is deterministic and failure resistant**

☞ **None of the above is currently used in performance oriented applications.**

© 2010 Dave R. Zinner, The Australian National University page 540 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Network protocols & standards Fibre Channel

- Developed in the late 80's.
- ANSI standard since 1994.
- Current standards allow for 16Gbps per link.
- Allows for three different topologies:
  - Point-to-point: 2 addresses
  - Arbitrated loop (similar to token ring): 127 addresses
  - Switched fabric: 2<sup>24</sup> addresses, many topologies and concurrent data links possible
- Defines OSI equivalent layers up to the session level.
- Mostly used in storage arrays, but applicable to super-computers and high integrity systems as well.

© 2010 Univ R. Zimenc, The Australian National University page 541 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Network protocols & standards Fibre Channel

Mapping of Fibre Channel to OSI layers:

© 2010 Univ R. Zimenc, The Australian National University page 542 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Network protocols & standards InfiniBand

- Developed in the late 90's
- Defined by the InfiniBand Trade Association (IBTA) since 1999.
- Current standards allow for 25Gbps per link.
- Switched fabric topologies.
- Concurrent data links possible (commonly up to 12 or 300Gbps).
- Defines only the *data-link* layer and parts of the *network* layer.
- Existing devices use copper cables (instead of optical fibres).
- Mostly used in super-computers and clusters but applicable to storage arrays as well.
- Cheaper than Ethernet or FibreChannel at high data-rates.
- Small packets (only up to 4 kB) and no session control.

© 2010 Univ R. Zimenc, The Australian National University page 543 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems Distribution! Motivation

Possibly ...

- ... fits an **existing physical distribution** (e-mail system, devices in a large craft, ...).
- ... **high performance** due to potentially high degree of parallel processing.
- ... **high reliability/integrity** due to redundancy of hardware and software.
- ... **scalable**.
- ... **integration** of heterogeneous devices.

Different specifications will lead to substantially different distributed designs.

© 2010 Univ R. Zimenc, The Australian National University page 544 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems What can be distributed?

- State**      Common operations on *distributed* data
- Function**      *Distributed* operations on *central* data
- State & Function**      Client/server clusters
- none of those**      Pure replication, redundancy

© 2010 Univ R. Zimenc, The Australian National University page 545 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems Common design criteria

- Achieve **De-coupling** / high degree of local autonomy
- Cooperation** rather than central control
- Consider **Reliability**
- Consider **Scalability**
- Consider **Performance**

© 2010 Univ R. Zimenc, The Australian National University page 546 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems Some common phenomena in distributed systems

- Unpredictable delays** (communication)
  - Are we done yet?
- Missing or imprecise time-base**
  - Causal relation or temporal relation?
- Partial failures**
  - Likelihood of individual failures increases
  - Likelihood of complete failure decreases (in case of a good design)

© 2010 Univ R. Zimenc, The Australian National University page 547 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems Time in distributed systems

Two alternative strategies:

**Based on a shared time**      Synchronize clocks!

**Based on sequence of events**      Create a virtual time!

© 2010 Univ R. Zimenc, The Australian National University page 548 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems 'Real-time' clocks

are:

- discrete** – i.e. time is *not* dense and there is a minimal granularity
- drift affected**:

Maximal clock drift  $\delta$  defined as:

$$(1 + \delta)^{-1} \leq \frac{C(t_2) - C(t_1)}{t_2 - t_1} \leq (1 + \delta)$$

often specified as PPM (Parts-Per-Million)  
(typical  $\approx$  20 PPM in computer applications)

© 2010 Univ R. Zimenc, The Australian National University page 549 of 758 (chapter 8: "Distributed Systems" up to page 641)



## Distributed Systems

### Distributed Systems

#### Synchronize a 'real-time' clock (bi-directional)

Resetting the clock drift by regular reference time re-synchronization:

Maximal clock drift  $\delta$  defined as:

$$(1 + \delta)^{-1} \leq \frac{C(t_2) - C(t_1)}{t_2 - t_1} \leq (1 + \delta)$$

'real-time' clock is adjusted forwards & backwards

☞ Calendar time

© 2010 Dave R. Zinnec, The Australian National University page 550 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Distributed critical regions with synchronized clocks

#### Analysis

- No deadlock, no individual starvation, no livelock.
- Minimal request delay:  $2L$ .
- Minimal release delay:  $L$ .
- Communications requirements per request:  $2(N - 1)$  messages (can be significantly improved by employing broadcast mechanisms).
- Clock drifts affect fairness, but not integrity of the critical region.

Assumptions:

- $L$  is known and constant ☞ violation leads to loss of mutual exclusion.
- No messages are lost ☞ violation leads to loss of mutual exclusion.

© 2010 Dave R. Zinnec, The Australian National University page 553 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Virtual (logical) time

$$a \rightarrow b \Rightarrow C(a) < C(b)$$

Implications:

$$C(a) < C(b) \Rightarrow \neg(b \rightarrow a)$$

$$C(a) = C(b) \Rightarrow a \parallel b$$

$$C(a) = C(b) < C(c) \Rightarrow ?$$

$$C(a) < C(b) < C(c) \Rightarrow ?$$

© 2010 Dave R. Zinnec, The Australian National University page 556 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Synchronize a 'real-time' clock (forward only)

Resetting the clock drift by regular reference time re-synchronization:

Maximal clock drift  $\delta$  defined as:

$$(1 + \delta)^{-1} \leq \frac{C(t_2) - C(t_1)}{t_2 - t_1} \leq 1$$

'real-time' clock is adjusted forwards only

☞ Monotonic time

© 2010 Dave R. Zinnec, The Australian National University page 551 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Virtual (logical) time [Lamport 1978]

$$a \rightarrow b \Rightarrow C(a) < C(b)$$

with  $a \rightarrow b$  being a causal relation between  $a$  and  $b$ , and  $C(a)$ ,  $C(b)$  are the (virtual) times associated with  $a$  and  $b$

$a \rightarrow b$  iff:

- $a$  happens earlier than  $b$  in the same sequential control-flow or
- $a$  denotes the sending event of message  $m$ , while  $b$  denotes the receiving event of the same message  $m$  or
- there is a transitive causal relation between  $a$  and  $b$ :  $a \rightarrow e_1 \rightarrow \dots \rightarrow e_n \rightarrow b$

Notion of concurrency:

$$a \parallel b \Rightarrow \neg(a \rightarrow b) \wedge \neg(b \rightarrow a)$$

© 2010 Dave R. Zinnec, The Australian National University page 554 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Virtual (logical) time

$$a \rightarrow b \Rightarrow C(a) < C(b)$$

Implications:

$$C(a) < C(b) \Rightarrow \neg(b \rightarrow a) = (a \rightarrow b) \vee (a \parallel b)$$

$$C(a) = C(b) \Rightarrow a \parallel b = \neg(a \rightarrow b) \wedge \neg(b \rightarrow a)$$

$$C(a) = C(b) < C(c) \Rightarrow ?$$

$$C(a) < C(b) < C(c) \Rightarrow ?$$

© 2010 Dave R. Zinnec, The Australian National University page 557 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Distributed critical regions with synchronized clocks

- $\nabla$  times:
  - $\nabla$  received Requests: Add to local RequestQueue (ordered by time)
  - $\nabla$  received Release messages: Delete corresponding Requests in local RequestQueue

1. Create OwnRequest and attach current time-stamp. Add OwnRequest to local RequestQueue (ordered by time). Send OwnRequest to all processes.
2. Delay by  $2L$  ( $L$  being the time it takes for a message to reach all network nodes)
3. While Top (RequestQueue)  $\neq$  OwnRequest: delay until new message
4. Enter and leave critical region
5. Send Release-message to all processes.

© 2010 Dave R. Zinnec, The Australian National University page 552 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Virtual (logical) time

$$a \rightarrow b \Rightarrow C(a) < C(b)$$

Implications:

$$C(a) < C(b) \Rightarrow ?$$

$$C(a) = C(b) \Rightarrow ?$$

$$C(a) = C(b) < C(c) \Rightarrow ?$$

$$C(a) < C(b) < C(c) \Rightarrow ?$$

© 2010 Dave R. Zinnec, The Australian National University page 555 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Virtual (logical) time

$$a \rightarrow b \Rightarrow C(a) < C(b)$$

Implications:

$$C(a) < C(b) \Rightarrow \neg(b \rightarrow a) = (a \rightarrow b) \vee (a \parallel b)$$

$$C(a) = C(b) \Rightarrow a \parallel b = \neg(a \rightarrow b) \wedge \neg(b \rightarrow a)$$

$$C(a) = C(b) < C(c) \Rightarrow \neg(c \rightarrow a)$$

$$C(a) < C(b) < C(c) \Rightarrow \neg(c \rightarrow a)$$

© 2010 Dave R. Zinnec, The Australian National University page 558 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Virtual (logical) time

$$a \rightarrow b \Rightarrow C(a) < C(b)$$

Implications:

$$C(a) < C(b) \Rightarrow \neg(b \rightarrow a) = (a \rightarrow b) \vee (a \parallel b)$$

$$C(a) = C(b) \Rightarrow a \parallel b = \neg(a \rightarrow b) \wedge \neg(b \rightarrow a)$$

$$C(a) = C(b) < C(c) \Rightarrow \neg(c \rightarrow a) = (a \rightarrow c) \vee (a \parallel c)$$

$$C(a) < C(b) < C(c) \Rightarrow \neg(c \rightarrow a) = (a \rightarrow c) \vee (a \parallel c)$$

© 2020 Dave R. Zinnec, The Australian National University page 559 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Virtual (logical) time

Time as derived from causal relations:

☞ Events in concurrent control flows are not ordered.  
☞ No global order of time.

© 2020 Dave R. Zinnec, The Australian National University page 560 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Implementing a virtual (logical) time

- $\forall P_i; C_i = 0$
- $\forall P_i;$ 
  - local events:  $C_i = C_i + 1$ ;
  - send events:  $C_j = C_j + 1$ ; Send (message,  $C_i$ );
  - receive events: Receive (message,  $C_m$ );  $C_i = \max(C_i, C_m) + 1$ ;

© 2020 Dave R. Zinnec, The Australian National University page 561 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Distributed critical regions with logical clocks

- $\forall$  times:  $\forall$  received *Requests*:
  - Add to local *RequestQueue* (ordered by time)
  - Reply with *Acknowledge* or *OwnRequest*
- $\forall$  times:  $\forall$  received *Release messages*:
  - Delete corresponding *Requests* in local *RequestQueue*

- Create *OwnRequest* and attach current time-stamp. Add *OwnRequest* to local *RequestQueue* (ordered by time). Send *OwnRequest* to all processes.
- Wait for Top (*RequestQueue*) = *OwnRequest* & no outstanding replies
- Enter and leave critical region
- Send *Release-message* to all processes.

© 2020 Dave R. Zinnec, The Australian National University page 562 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Distributed critical regions with logical clocks

#### Analysis

- No deadlock, no individual starvation, no livelock.
- Minimal request delay:  $N - 1$  requests (1 broadcast) +  $N - 1$  replies.
- Minimal release delay:  $N - 1$  release messages (or 1 broadcast).
- Communications requirements per request:  $3(N - 1)$  messages (or  $N - 1$  messages + 2 broadcasts).
- Clocks are kept recent by the exchanged messages themselves.

Assumptions:

- No messages are lost ☞ violation leads to stall.

© 2020 Dave R. Zinnec, The Australian National University page 563 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Distributed critical regions with a token ring structure

- Organize all processes in a logical or physical ring topology
- Send one *token* message to one process
- $\forall$  times,  $\forall$  processes: On receiving the *token* message:
  - If required the process enters and leaves a critical section (while holding the token).
  - The *token* is passed along to the next process in the ring.

Assumptions:

- Token is not lost ☞ violation leads to stall.  
(a lost token can be recovered by a number of means – e.g. the 'election' scheme following)

© 2020 Dave R. Zinnec, The Australian National University page 564 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Distributed critical regions with a central coordinator

A global, static, central coordinator

- ☞ Invalidates the idea of a distributed system
- ☞ Enables a very simple mutual exclusion scheme

Therefore:

- A global, central coordinator is employed in some systems ... yet ...
- ... if it fails, a system to come up with a new coordinator is provided.

© 2020 Dave R. Zinnec, The Australian National University page 565 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Electing a central coordinator (the Bully algorithm)

Any process  $P$  which notices that the central coordinator is gone, performs:

- $P$  sends an *Election*-message to all processes with *higher* process numbers.
- $P$  waits for response messages.
  - ☞ If no one responds after a pre-defined amount of time:  $P$  declares itself the new coordinator and sends out a *Coordinator*-message to all.
  - ☞ If any process responds, then the election activity for  $P$  is over and  $P$  waits for a *Coordinator*-message

All processes  $P_i$  perform at all times:

- If  $P_i$  receives a *Election*-message from a process with a *lower* process number, it responds to the originating process and starts an election process itself (if not running already).

© 2020 Dave R. Zinnec, The Australian National University page 566 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Distributed states

☞ How to read the current state of a distributed system?

This "god's eye view" does in fact not exist.

© 2020 Dave R. Zinnec, The Australian National University page 567 of 758 (chapter 8: "Distributed Systems" up to page 641)

### Distributed Systems

#### Distributed Systems

#### Distributed states

How to read the current state of a distributed system?

Instead: some entity probes and collects local states.  
What state of the global system has been accumulated?

© 2020 Univ. R. Zimenc, The Australian National University page 568 of 758 (chapter 8: "Distributed Systems" up to page 641)

### Distributed Systems

#### Distributed Systems

#### Distributed states

How to read the current state of a distributed system?

Instead: some entity probes and collects local states.  
What state of the global system has been accumulated?  
Connecting all the states to a global state.

© 2020 Univ. R. Zimenc, The Australian National University page 569 of 758 (chapter 8: "Distributed Systems" up to page 641)

### Distributed Systems

#### Distributed Systems

#### Distributed states

A consistent global state (snapshot) is define by a unique division into:

- “The Past”  $P$  (events before the snapshot):  
 $(e_2 \in P) \wedge (e_1 \rightarrow e_2) \Rightarrow e_1 \in P$
- “The Future”  $F$  (events after the snapshot):  
 $(e_1 \in F) \wedge (e_1 \rightarrow e_2) \Rightarrow e_2 \in F$

© 2020 Univ. R. Zimenc, The Australian National University page 570 of 758 (chapter 8: "Distributed Systems" up to page 641)

### Distributed Systems

#### Distributed Systems

#### Distributed states

How to read the current state of a distributed system?

Instead: some entity probes and collects local states.  
What state of the global system has been accumulated?  
Sorting the events into past and future events.

© 2020 Univ. R. Zimenc, The Australian National University page 571 of 758 (chapter 8: "Distributed Systems" up to page 641)

### Distributed Systems

#### Distributed Systems

#### Distributed states

How to read the current state of a distributed system?

Instead: some entity probes and collects local states.  
What state of the global system has been accumulated?  
Event in the past receives a message from the future!  
Division not possible Snapshot inconsistent!

© 2020 Univ. R. Zimenc, The Australian National University page 572 of 758 (chapter 8: "Distributed Systems" up to page 641)

### Distributed Systems

#### Distributed Systems

#### Snapshot algorithm

- Observer-process  $P_0$  (any process) creates a snapshot token  $t_s$  and saves its local state  $s_0$ .
- $P_0$  sends  $t_s$  to all other processes.
- $\forall P_i$  which receive  $t_s$  (as an individual token-message, or as part of another message):
  - Save local state  $s_i$  and send  $s_i$  to  $P_0$ .
  - Attach  $t_s$  to all further messages, which are to be sent to other processes.
  - Save  $t_s$  and ignore all further incoming  $t_s$ 's.
- $\forall P_i$  which previously received  $t_s$  and receive a message  $m$  without  $t_s$ :
  - Forward  $m$  to  $P_0$  (this message belongs to the snapshot).

© 2020 Univ. R. Zimenc, The Australian National University page 573 of 758 (chapter 8: "Distributed Systems" up to page 641)

### Distributed Systems

#### Distributed Systems

#### Distributed states

Running the snapshot algorithm:

- Observer-process  $P_0$  (any process) creates a snapshot token  $t_s$  and saves its local state  $s_0$ .
- $P_0$  sends  $t_s$  to all other processes.

© 2020 Univ. R. Zimenc, The Australian National University page 574 of 758 (chapter 8: "Distributed Systems" up to page 641)

### Distributed Systems

#### Distributed Systems

#### Distributed states

Running the snapshot algorithm:

- $\forall P_i$  which receive  $t_s$  (as an individual token-message, or as part of another message):
  - Save local state  $s_i$  and send  $s_i$  to  $P_0$ .
  - Attach  $t_s$  to all further messages, which are to be sent to other processes.
  - Save  $t_s$  and ignore all further incoming  $t_s$ 's.

© 2020 Univ. R. Zimenc, The Australian National University page 575 of 758 (chapter 8: "Distributed Systems" up to page 641)

### Distributed Systems

#### Distributed Systems

#### Distributed states

Running the snapshot algorithm:

- $\forall P_i$  which previously received  $t_s$  and receive a message  $m$  without  $t_s$ :
  - Forward  $m$  to  $P_0$  (this message belongs to the snapshot).

© 2020 Univ. R. Zimenc, The Australian National University page 576 of 758 (chapter 8: "Distributed Systems" up to page 641)

**Distributed Systems**  
**Distributed Systems**  
**Distributed states**

Running the snapshot algorithm:

- $\forall P_i$  which receive  $t_s$  (as an individual token-message, or as part of another message):
  - Save local state  $s_i$  and send  $s_i$  to  $P_0$ .
  - Attach  $t_s$  to all further messages, which are to be sent to other processes.
  - Save  $t_s$  and ignore all further incoming  $t_s$ 's.

© 2020 Uwe R. Zimmer, The Australian National University page 577 of 758 (chapter 8: "Distributed Systems" up to page 641)

**Distributed Systems**  
**Distributed Systems**  
**Distributed states**

Running the snapshot algorithm:

- Save  $t_s$  and ignore all further incoming  $t_s$ 's.

© 2020 Uwe R. Zimmer, The Australian National University page 578 of 758 (chapter 8: "Distributed Systems" up to page 641)

**Distributed Systems**  
**Distributed Systems**  
**Distributed states**

Running the snapshot algorithm:

- Finalize snapshot

© 2020 Uwe R. Zimmer, The Australian National University page 579 of 758 (chapter 8: "Distributed Systems" up to page 641)

**Distributed Systems**  
**Distributed Systems**  
**Distributed states**

Running the snapshot algorithm:

Sorting the events into past and future events.  
 Past and future events uniquely separated  
 Consistent state

© 2020 Uwe R. Zimmer, The Australian National University page 580 of 758 (chapter 8: "Distributed Systems" up to page 641)

**Distributed Systems**  
**Distributed Systems**  
**Snapshot algorithm**

**Termination condition?**

Either

- Make assumptions about the communication delays in the system.

or

- Count the sent and received messages for each process (include this in the local state) and keep track of outstanding messages in the observer process.

© 2020 Uwe R. Zimmer, The Australian National University page 581 of 758 (chapter 8: "Distributed Systems" up to page 641)

**Distributed Systems**  
**Distributed Systems**  
**Consistent distributed states**

Why would we need that?

- Find deadlocks.
- Find termination / completion conditions.
- ... any other global safety of liveness property.
- Collect a consistent system state for system backup/restore.
- Collect a consistent system state for further processing (e.g. distributed databases).
- ...

© 2020 Uwe R. Zimmer, The Australian National University page 582 of 758 (chapter 8: "Distributed Systems" up to page 641)

**Distributed Systems**  
**Distributed Systems**  
**A distributed server (load balancing)**

© 2020 Uwe R. Zimmer, The Australian National University page 583 of 758 (chapter 8: "Distributed Systems" up to page 641)

**Distributed Systems**  
**Distributed Systems**  
**A distributed server (load balancing)**

© 2020 Uwe R. Zimmer, The Australian National University page 584 of 758 (chapter 8: "Distributed Systems" up to page 641)

**Distributed Systems**  
**Distributed Systems**  
**A distributed server (load balancing)**

© 2020 Uwe R. Zimmer, The Australian National University page 585 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### A distributed server (load balancing)

© 2010 Dave R. Zinner, The Australian National University page 586 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### A distributed server (load balancing)

© 2010 Dave R. Zinner, The Australian National University page 587 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### A distributed server (load balancing)

```

with Ada.Task_Identification; use Ada.Task_Identification;
task type Print_Server is
  entry Send_To_Server (Print_Job : in Job_Type; Job_Done : out Boolean);
  entry Contention (Print_Job : in Job_Type; Server_Id : in Task_Id);
end Print_Server;
    
```

© 2010 Dave R. Zinner, The Australian National University page 588 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### A distributed server (load balancing)

```

task body Print_Server is
begin
loop
select
  accept Send_To_Server (Print_Job : in Job_Type; Job_Done : out Boolean) do
    if not Print_Job in Turned_Down_Jobs then
      if Not_Too_Busy then
        Applied_For_Jobs := Applied_For_Jobs + Print_Job;
        Next_Server_On_Ring.Contention (Print_Job, Current_Task);
        queue Internal_Print_Server.Print_Job_Queue;
      else
        Turned_Down_Jobs := Turned_Down_Jobs + Print_Job;
      end if;
    end if;
  end Send_To_Server;
    (...)
    
```

© 2010 Dave R. Zinner, The Australian National University page 589 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

```

or
  accept Contention (Print_Job : in Job_Type; Server_Id : in Task_Id) do
    if Print_Job in Applied_For_Jobs then
      if Server_Id = Current_Task then
        Internal_Print_Server.Start_Print (Print_Job);
      elsif Server_Id > Current_Task then
        Internal_Print_Server.Cancel_Print (Print_Job);
        Next_Server_On_Ring.Contention (Print_Job; Server_Id);
      else
        null; -- removing the contention message from ring
      end if;
    else
      Turned_Down_Jobs := Turned_Down_Jobs + Print_Job;
      Next_Server_On_Ring.Contention (Print_Job; Server_Id);
    end if;
  end Contention;
or
  terminate;
end select;
end loop;
end Print_Server;
    
```

© 2010 Dave R. Zinner, The Australian National University page 590 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Transactions

☞ Concurrency and distribution in systems with multiple, interdependent interactions?

☞ Concurrent and distributed client/server interactions beyond single remote procedure calls?

© 2010 Dave R. Zinner, The Australian National University page 591 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Transactions

Definition (ACID properties):

- **Atomicity:** All or none of the sub-operations are performed. Atomicity helps achieve crash resilience. If a crash occurs, then it is possible to roll back the system to the state before the transaction was invoked.
- **Consistency:** Transforms the system from one consistent state to another consistent state.
- **Isolation:** Results (including partial results) are not revealed unless and until the transaction commits. If the operation accesses a shared data object, invocation does not interfere with other operations on the same object.
- **Durability:** After a commit, results are guaranteed to persist, even after a subsequent system failure.

© 2010 Dave R. Zinner, The Australian National University page 592 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Transactions

Definition (ACID properties):

- **Atomicity:** All or none of the sub-operations are performed. Atomicity helps achieve crash resilience. If a crash occurs, then it is possible to roll back the system to the state before the transaction was invoked.
- **Consistency:** Transforms the system from one consistent state to another consistent state.
- **Isolation:** Results (including partial results) are not revealed unless and until the transaction commits. If the operation accesses a shared data object, invocation does not interfere with other operations on the same object.
- **Durability:** After a commit, results are guaranteed to persist, even after a subsequent system failure.

Atomic operations spanning multiple processes? How to ensure consistency in a distributed system?

Shadow copies?

What hardware do we need to assume? Actual isolation and efficient concurrency? Actual isolation or the appearance of isolation?

© 2010 Dave R. Zinner, The Australian National University page 593 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Transactions

A closer look *inside* transactions:

- Transactions consist of a sequence of operations.
- If two operations out of two transactions can be performed in any order with the same final effect, they are commutative and not critical for our purposes.
- Idempotent and side-effect free operations are by definition commutative.
- All non-commutative operations are considered critical operations.
- Two critical operations as part of two different transactions while affecting the same object are called a conflicting pair of operations.

© 2010 Dave R. Zinner, The Australian National University page 594 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems Transactions

A closer look at *multiple* transactions:

- Any *sequential* execution of multiple transactions will fulfil the ACID-properties, by definition of a single transaction.
- A *concurrent* execution (or 'interleavings') of multiple transactions *might* fulfil the ACID-properties.

☞ If a specific *concurrent* execution can be shown to be *equivalent* to a specific sequential execution of the involved transactions then this specific interleaving is called **'serializable'**.

☞ If a concurrent execution ('interleaving') ensures that no transaction ever encounters an inconsistent state then it is said to ensure the **appearance of isolation**.

© 2020 Univ R. Zinnec, The Australian National University page 595 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems Achieving serializability

☞ For the **serializability** of two transactions it is **necessary and sufficient** for the **order** of their invocations of all conflicting pairs of operations to be **the same** for all the objects which are invoked by both transactions.

(Determining order in distributed systems requires logical clocks.)

© 2020 Univ R. Zinnec, The Australian National University page 596 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems Serializability

- Two conflicting pairs of operations with the same order of execution.

© 2020 Univ R. Zinnec, The Australian National University page 597 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems Serializability

☞ **Serializable**

© 2020 Univ R. Zinnec, The Australian National University page 598 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems Serializability

- Two conflicting pairs of operations with different orders of executions.

☞ **Not serializable.**

© 2020 Univ R. Zinnec, The Australian National University page 599 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems Serializability

- Three conflicting pairs of operations with the same order of execution (pair-wise between processes).
- The order between processes also leads to a global order of processes.

© 2020 Univ R. Zinnec, The Australian National University page 600 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems Serializability

- Three conflicting pairs of operations with the same order of execution (pair-wise between processes).
- The order between processes also leads to a global order of processes.

☞ **Serializable**

© 2020 Univ R. Zinnec, The Australian National University page 601 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems Serializability

- Three conflicting pairs of operations with the same order of execution (pair-wise between processes).
- The order between processes also leads to a global order of processes.

☞ **Serializable**

© 2020 Univ R. Zinnec, The Australian National University page 602 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems Serializability

- Three conflicting pairs of operations with the same order of execution (pair-wise between processes).
- The order between processes *does not* lead to a global order of processes.

☞ **Not serializable**

© 2020 Univ R. Zinnec, The Australian National University page 603 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Achieving serializability

For the **serializability** of two transactions it is **necessary and sufficient** for the **order** of their invocations of all conflicting pairs of operations to be **the same** for all the objects which are invoked by both transactions.

- Define: **Serialization graph**: A directed graph; Vertices  $i$  represent transactions  $T_i$ ; Edges  $T_i \rightarrow T_j$  represent an established global order dependency between all conflicting pairs of operations of those two transactions.

For the **serializability** of multiple transactions it is **necessary and sufficient** that the serialization graph is **acyclic**.

© 2010 Dave R. Zinnec, The Australian National University page 604 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Transaction schedulers

Three major designs:

- Locking methods:** Impose strict mutual exclusion on all critical sections.
- Time-stamp ordering:** Note relative starting times and keep order dependencies consistent.
- "Optimistic" methods:** Go ahead until a conflict is observed – then roll back.

© 2010 Dave R. Zinnec, The Australian National University page 607 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Transaction schedulers – Optimistic control

Three sequential phases:

- Read & execute:** Create a **shadow copy** of all involved objects and perform all required operations on the **shadow copy** and **locally** (i.e. in isolation).
- Validate:** After local commit, check all occurred interleavings for **serializability**.
- Update or abort:**
  - If serializability could be ensured in step 2 then all results of involved transactions are **written** to all involved objects – **in dependency order of the transactions**.
  - Otherwise: **destroy** shadow copies and **start over** with the failed transactions.

© 2010 Dave R. Zinnec, The Australian National University page 610 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Serializability

- Three conflicting pairs of operations with the same order of execution (pair-wise between processes).
- Serialization graph is **acyclic**.
- Serializable**

© 2010 Dave R. Zinnec, The Australian National University page 605 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Transaction schedulers – Locking methods

Locking methods include the possibility of deadlocks – careful from here on out ...

- Complete resource allocation** before the start and release at the end of every transaction:
  - This will impose a **strict sequential execution** of all critical transactions.
- (Strict) two-phase locking:** Each transaction follows the following two phase pattern during its operation:
  - Growing phase:** locks can be acquired, but not released.
  - Shrinking phase:** locks can be **released anytime**, but not acquired (two phase locking) or locks are released **on commit only** (strict two phase locking).
- Possible deadlocks
  - Serializable interleavings
  - Strict isolation (in case of strict two-phase locking)
- Semantic locking:** Allow for separate read-only and write-locks
  - Higher level of concurrency (see also: use of functions in protected objects)

© 2010 Dave R. Zinnec, The Australian National University page 608 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Transaction schedulers – Optimistic control

Three sequential phases:

- Read & execute:** Create a **shadow copy** of all involved objects and perform all required operations on the **shadow copy** and **locally** (i.e. in isolation).
  - How to create a consistent copy?
  - Full isolation and maximal concurrency!
- Validate:** After local commit, check all occurred interleavings for **serializability**.
- Update or abort:**
  - If serializability could be ensured in step 2 then all results of involved transactions are **written** to all involved objects – **in dependency order of the transactions**.
    - How to update all objects consistently?
  - Otherwise: **destroy** shadow copies and **start over** with the failed transactions.
    - Abort happens after everything has been committed locally.

© 2010 Dave R. Zinnec, The Australian National University page 611 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Serializability

- Three conflicting pairs of operations with the same order of execution (pair-wise between processes).
- Serialization graph is **cyclic**.
- Not serializable**

© 2010 Dave R. Zinnec, The Australian National University page 606 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

### Distributed Systems

#### Transaction schedulers – Time stamp ordering

Add a unique time-stamp (any global order criterion) on every transaction upon start. Each involved object can inspect the time-stamps of all requesting transactions.

- Case 1: A transaction with a time-stamp **later** than all currently active transactions applies:
  - the request is accepted and the transaction can go **ahead**.
  - Alternative case 1 (strict time-stamp ordering):
    - the request is **delayed** until the currently active earlier transaction has committed.
- Case 2: A transaction with a time-stamp **earlier** than all currently active transactions applies:
  - the request is not accepted and the applying transaction is to be **aborted**.

Collision detection rather than collision avoidance  
 No isolation – Cascading aborts possible.  
 Simple implementation, high degree of concurrency – also in a distributed environment, as long as a global event order (time) can be supplied.

© 2010 Dave R. Zinnec, The Australian National University page 609 of 758 (chapter 8: "Distributed Systems" up to page 641)

## Distributed Systems

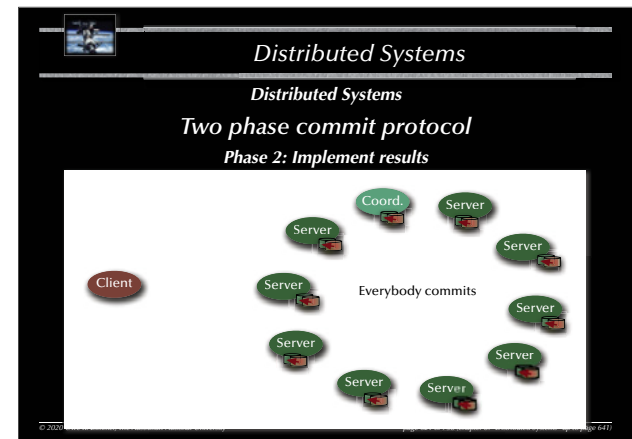
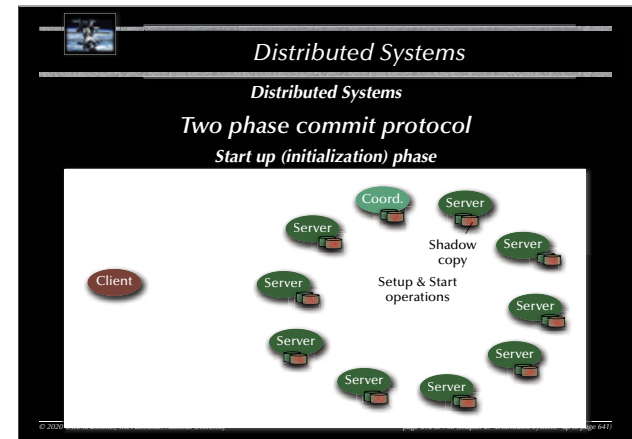
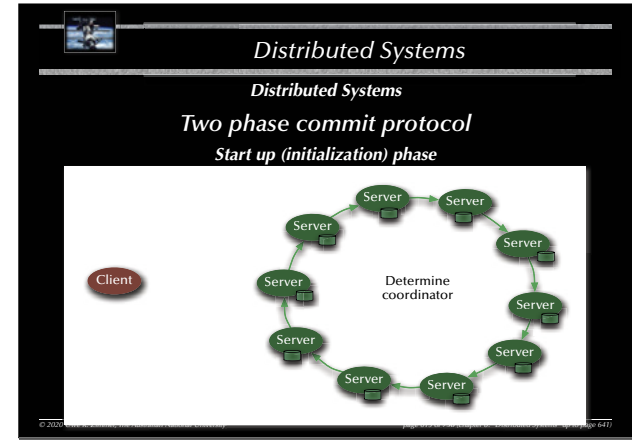
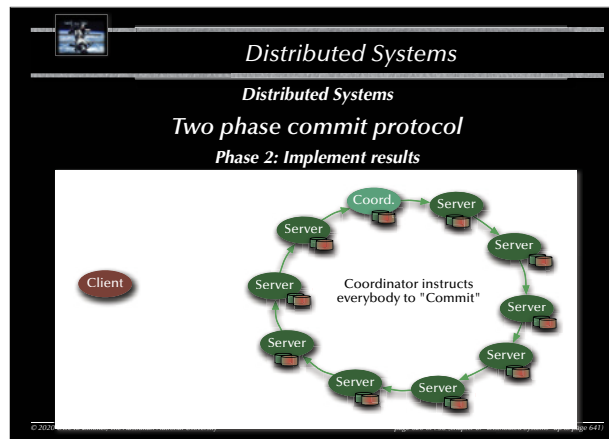
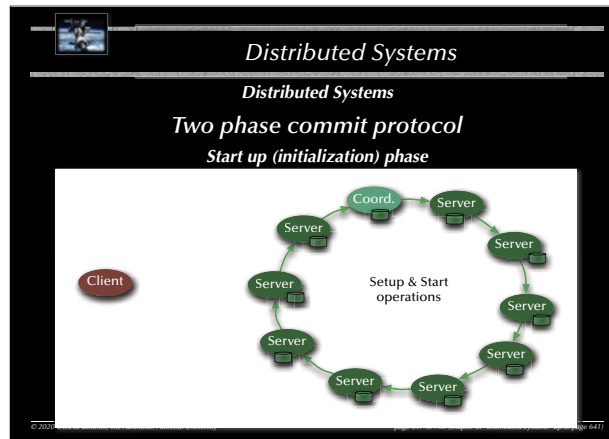
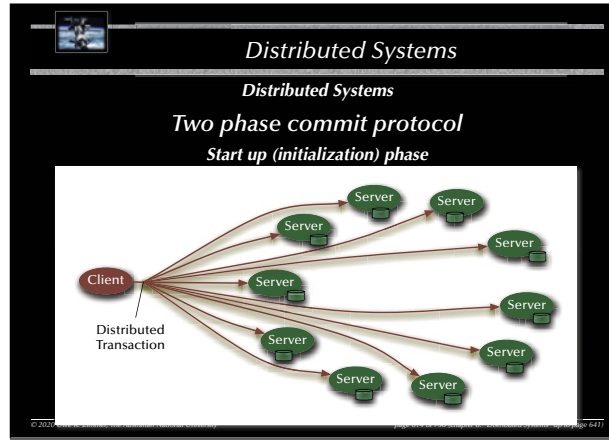
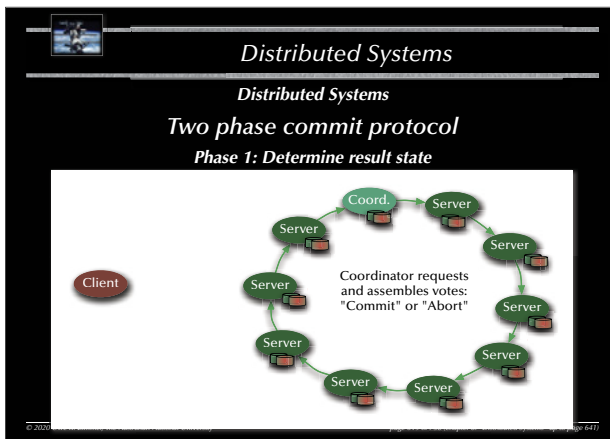
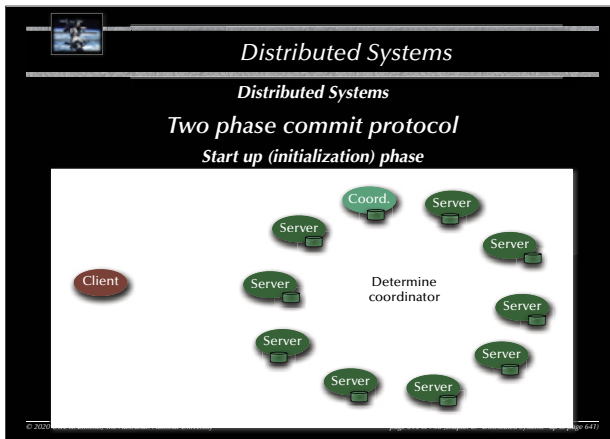
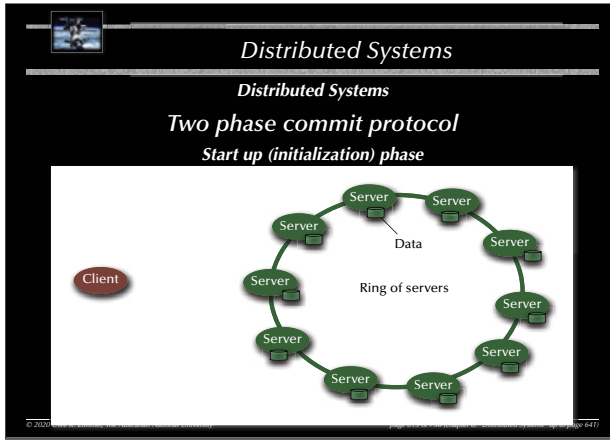
### Distributed Systems

#### Distributed transaction schedulers

Three major designs:

- Locking methods:** **no aborts** Impose strict mutual exclusion on all critical sections.
- Time-stamp ordering:** **potential aborts along the way** Note relative starting times and keep order dependencies consistent.
- "Optimistic" methods:** **abort or commit at the very end** Go ahead until a conflict is observed – then roll back.
  - How to implement "commit" and "abort" operations in a distributed environment?

© 2010 Dave R. Zinnec, The Australian National University page 612 of 758 (chapter 8: "Distributed Systems" up to page 641)





Distributed Systems

Distributed Systems

Two phase commit protocol

Phase 2: Implement results

Everybody destroys shadows

© 2020 Univ. R. Zimmer, The Australian National University page 622 of 758 (chapter 8: "Distributed Systems" - up to page 641)

Distributed Systems

Distributed Systems

Two phase commit protocol

or Phase 2: Global roll back

Everybody destroys shadows

© 2020 Univ. R. Zimmer, The Australian National University page 625 of 758 (chapter 8: "Distributed Systems" - up to page 641)

Distributed Systems

Distributed Systems

Redundancy (replicated servers)

Premise:

A crashing server computer should not compromise the functionality of the system (full fault tolerance)

Assumptions & Means:

- $k$  computers inside the server cluster might crash without losing functionality.
  - ☞ Replication: at least  $k + 1$  servers.
- The server cluster can reorganize any time (and specifically after the loss of a computer).
  - ☞ Hot stand-by components, dynamic server group management.
- The server is described fully by the current state and the sequence of messages received.
  - ☞ State machines: we have to implement consistent state adjustments (re-organization) and consistent message passing (order needs to be preserved).

[Schneider1990]

© 2020 Univ. R. Zimmer, The Australian National University page 628 of 758 (chapter 8: "Distributed Systems" - up to page 641)

Distributed Systems

Distributed Systems

Two phase commit protocol

Phase 2: Implement results

Everybody reports "Committed"

© 2020 Univ. R. Zimmer, The Australian National University page 623 of 758 (chapter 8: "Distributed Systems" - up to page 641)

Distributed Systems

Distributed Systems

Two phase commit protocol

Phase 2: Report result of distributed transaction

Coordinator reports to client: "Committed" or "Aborted"

© 2020 Univ. R. Zimmer, The Australian National University page 626 of 758 (chapter 8: "Distributed Systems" - up to page 641)

Distributed Systems

Distributed Systems

Redundancy (replicated servers)

Stages of each server:

Job message received by all active servers

Received

Deliverable

Job message received locally

Processed

Job processed locally

© 2020 Univ. R. Zimmer, The Australian National University page 629 of 758 (chapter 8: "Distributed Systems" - up to page 641)

Distributed Systems

Distributed Systems

Two phase commit protocol

or Phase 2: Global roll back

Coordinator instructs everybody to "Abort"

© 2020 Univ. R. Zimmer, The Australian National University page 624 of 758 (chapter 8: "Distributed Systems" - up to page 641)

Distributed Systems

Distributed Systems

Distributed transaction schedulers

Evaluating the three major design methods in a distributed environment:

- **Locking methods:** ☞ No aborts. Large overheads; Deadlock detection/prevention required.
- **Time-stamp ordering:** ☞ Potential aborts along the way. Recommends itself for distributed applications, since decisions are taken locally and communication overhead is relatively small.
- **"Optimistic" methods:** ☞ Aborts or commits at the very end. Maximizes concurrency, but also data replication.

☞ Side-aspect "data replication": large body of literature on this topic (see: distributed data-bases / operating systems / shared memory / cache management, ...)

© 2020 Univ. R. Zimmer, The Australian National University page 627 of 758 (chapter 8: "Distributed Systems" - up to page 641)

Distributed Systems

Distributed Systems

Redundancy (replicated servers)

Start-up (initialization) phase

Ring of identical servers

© 2020 Univ. R. Zimmer, The Australian National University page 630 of 758 (chapter 8: "Distributed Systems" - up to page 641)

**Distributed Systems**  
**Distributed Systems**  
**Redundancy (replicated servers)**  
 Start-up (initialization) phase

Determine coordinator

© 2020 Drew K. Zimmerman, The Australian National University page 631 of 758 (chapter 8: Distributed systems - up to page 641)

**Distributed Systems**  
**Distributed Systems**  
**Redundancy (replicated servers)**  
 Start-up (initialization) phase

Coordinator determined

© 2020 Drew K. Zimmerman, The Australian National University page 632 of 758 (chapter 8: Distributed systems - up to page 641)

**Distributed Systems**  
**Distributed Systems**  
**Redundancy (replicated servers)**  
 Coordinator receives job message

Coordinator receives job message

© 2020 Drew K. Zimmerman, The Australian National University page 633 of 758 (chapter 8: Distributed systems - up to page 641)

**Distributed Systems**  
**Distributed Systems**  
**Redundancy (replicated servers)**  
 Distribute job

Coordinator sends job both ways

© 2020 Drew K. Zimmerman, The Australian National University page 634 of 758 (chapter 8: Distributed systems - up to page 641)

**Distributed Systems**  
**Distributed Systems**  
**Redundancy (replicated servers)**  
 Distribute job

Everybody received job (but nobody knows that)

© 2020 Drew K. Zimmerman, The Australian National University page 635 of 758 (chapter 8: Distributed systems - up to page 641)

**Distributed Systems**  
**Distributed Systems**  
**Redundancy (replicated servers)**  
 Processing starts

First server detects two job-messages & processes job

© 2020 Drew K. Zimmerman, The Australian National University page 636 of 758 (chapter 8: Distributed systems - up to page 641)

**Distributed Systems**  
**Distributed Systems**  
**Redundancy (replicated servers)**  
 Everybody (besides coordinator) processes

All server detect two job-messages & everybody processes job

© 2020 Drew K. Zimmerman, The Australian National University page 637 of 758 (chapter 8: Distributed systems - up to page 641)

**Distributed Systems**  
**Distributed Systems**  
**Redundancy (replicated servers)**  
 Coordinator processes

Coordinator also received two messages and processes job

© 2020 Drew K. Zimmerman, The Australian National University page 638 of 758 (chapter 8: Distributed systems - up to page 641)

**Distributed Systems**  
**Distributed Systems**  
**Redundancy (replicated servers)**  
 Result delivery

Coordinator delivers his local result

© 2020 Drew K. Zimmerman, The Australian National University page 639 of 758 (chapter 8: Distributed systems - up to page 641)



## Distributed Systems

### Distributed Systems

#### Redundancy (replicated servers)

Event: Server crash, new servers joining, or current servers leaving.

☞ Server re-configuration is triggered by a message to all  
(this is assumed to be supported by the distributed operating system).

Each server on reception of a re-configuration message:

1. Wait for local job to complete or time-out.
2. Store local consistent state  $S_i$ .
3. Re-organize server ring, send local state around the ring.
4. If a state  $S_j$  with  $j > i$  is received then  $S_i \leftarrow S_j$ .
5. Elect coordinator.
6. Enter 'Coordinator-' or 'Replicate-mode'.




## Distributed Systems

### Summary

#### Distributed Systems

- **Networks**
  - OSI, topologies
  - Practical network standards
- **Time**
  - Synchronized clocks, virtual (logical) times
  - Distributed critical regions (synchronized, logical, token ring)
- **Distributed systems**
  - Elections
  - Distributed states, consistent snapshots
  - Distributed servers (replicates, distributed processing, distributed commits)
  - Transactions (ACID properties, serializable interleavings, transaction schedulers)

# Systems, Networks & Concurrency 2020



# 9

## Architectures

Uwe R. Zimmer - The Australian National University

## Architectures

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

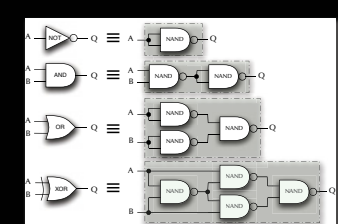
## Architectures

### Logic - the basic building blocks for digital computers

Constructing logic gates – for instance NAND in CMOS:

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

... and subsequently all other logic gates:



## Architectures

### References

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

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

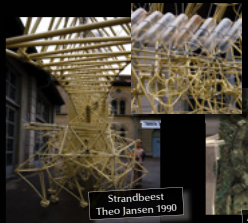
[Intel2010]  
Intel® 64 and IA-32 Architectures Optimization Reference Manual  
<http://www.intel.com/products/processor/manuals/>

## Architectures


### Logic - the basic building blocks

### Controllable Switches & Ratios


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



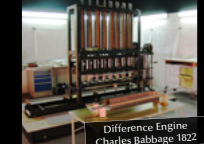
Strandbeest  
Theo Jansen 1990



Antikythera Mechanism  
Greek 150-100BC -  
Credit: Wikipedia



First transistor  
John Bardeen and Walter Brattain 1947

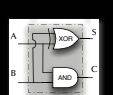


Difference Engine  
Charles Babbage 1822

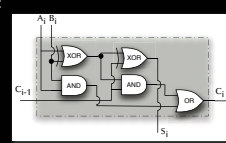
## Architectures

### Logic - the basic building blocks

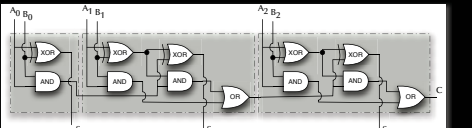
Half adder:



Full adder:



Ripple carry adder:



## Architectures

### In this chapter

Hardware architectures:

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

Software architectures:

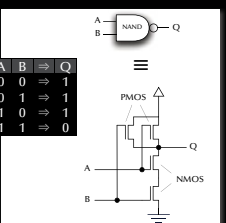
- ☞ Languages of Concurrency
- ☞ Operating systems and libraries

## Architectures

### Logic - the basic building blocks for digital computers

Constructing logic gates – for instance NAND in CMOS:

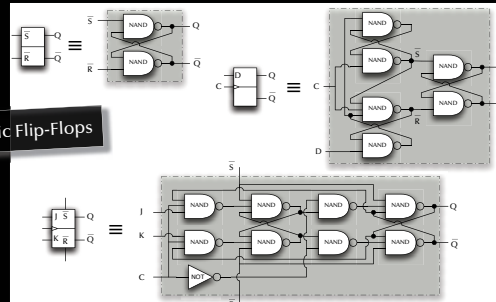
A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0



## Architectures

### Logic - the basic building blocks

Basic Flip-Flops



### Architectures

#### Logic - the basic building blocks

**JK- and D-Flip-Flops as universal Flip-Flops**

**Counting register:**

© 2020 Uwe R. Zimmer, The Australian National University page 651 of 758 chapter 9: "Architectures" up to page 746



### Architectures

#### Interrupt processing

**Push registers  
Declare local variables**

© 2020 Uwe R. Zimmer, The Australian National University page 657 of 758 chapter 9: "Architectures" up to page 746

### Architectures

#### Processor Architectures

##### A simple CPU

- Decoder/Sequencer**  
Can be a machine in itself which breaks CPU instructions into *concurrent* micro code.
- Execution Unit / Arithmetic-Logic-Unit (ALU)**  
A collection of transformational logic.
- Memory**
- Registers**  
Instruction pointer, stack pointer, general purpose and specialized registers
- Flags**  
Indicating the states of the latest calculations.
- Code/Data management**  
Fetching, Caching, Storing

© 2020 Uwe R. Zimmer, The Australian National University page 652 of 758 chapter 9: "Architectures" up to page 746

### Architectures

#### Interrupt processing

##### Interrupt handler

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

© 2020 Uwe R. Zimmer, The Australian National University page 655 of 758 chapter 9: "Architectures" up to page 746

### Architectures

#### Interrupt processing

##### Interrupt handler

**Remove local variables**

© 2020 Uwe R. Zimmer, The Australian National University page 658 of 758 chapter 9: "Architectures" up to page 746

### Architectures

#### Processor Architectures

##### Interrupts

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

On interrupt (if unmasked):

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

© 2020 Uwe R. Zimmer, The Australian National University page 653 of 758 chapter 9: "Architectures" up to page 746

### Architectures

#### Interrupt processing

##### Interrupt handler

© 2020 Uwe R. Zimmer, The Australian National University page 656 of 758 chapter 9: "Architectures" up to page 746

### Architectures

#### Interrupt processing

##### Interrupt handler

© 2020 Uwe R. Zimmer, The Australian National University page 659 of 758 chapter 9: "Architectures" up to page 746

### Architectures

#### Interrupt processing

Program

Code

Stack

PC

SP

FP

local variables

return address

control

parameters

local variables

return address

control

parameters

global variables

Interrupt handler

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

© 2020 Dave R. Zinin, The Australian National University page 660 of 758 chapter 9: "Architectures" up to page 746

### Architectures

#### Interrupt processing

Program

Code

Stack

PC

SP

FP

local variables

return address

control

parameters

local variables

return address

control

parameters

global variables

Interrupt handler

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

Baha Honda Rail Bridge Creative Commons Attribution-ShareAlike 3.0. Photography by M&S of English Wikipedia

© 2020 Dave R. Zinin, The Australian National University page 661 of 758 chapter 9: "Architectures" up to page 746

We successfully interrupted a sequence of operations ...

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

© 2020 Dave R. Zinin, The Australian National University page 662 of 758 chapter 9: "Architectures" up to page 746

### Architectures

#### Interrupt processing

Program

Code

Stack

PC

SP

FP

local variables

return address

control

parameters

local variables

return address

control

parameters

global variables

Interrupt handler

© 2020 Dave R. Zinin, The Australian National University page 663 of 758 chapter 9: "Architectures" up to page 746

### Architectures

#### Interrupt processing

Program

Code

Stack

PC

SP

FP

local variables

return address

control

parameters

local variables

return address

control

parameters

global variables

Interrupt handler

© 2020 Dave R. Zinin, The Australian National University page 664 of 758 chapter 9: "Architectures" up to page 746

### Architectures

#### Interrupt processing

Program

Code

Stack

PC

SP

FP

local variables

return address

control

parameters

local variables

return address

control

parameters

global variables

Interrupt handler

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

© 2020 Dave R. Zinin, The Australian National University page 665 of 758 chapter 9: "Architectures" up to page 746

### Architectures

#### Interrupt processing

Program

Code

Stack

PC

SP

FP

local variables

Registers

Flags

PC

local variables

return address

control

parameters

local variables

return address

control

parameters

global variables

Interrupt handler

Push registers  
Declare local variables

© 2020 Dave R. Zinin, The Australian National University page 666 of 758 chapter 9: "Architectures" up to page 746

### Architectures

#### Interrupt processing

Program

Code

Stack

PC

SP

FP

local variables

Registers

Flags

PC

local variables

return address

control

parameters

local variables

return address

control

parameters

global variables

Interrupt handler

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

© 2020 Dave R. Zinin, The Australian National University page 667 of 758 chapter 9: "Architectures" up to page 746

### Architectures

#### Interrupt processing

Program

Code

Stack

PC

SP

FP

local variables

Registers

Flags

PC

local variables

return address

control

parameters

local variables

return address

control

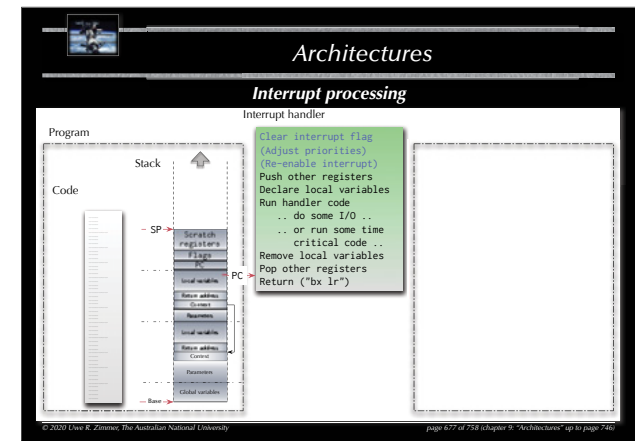
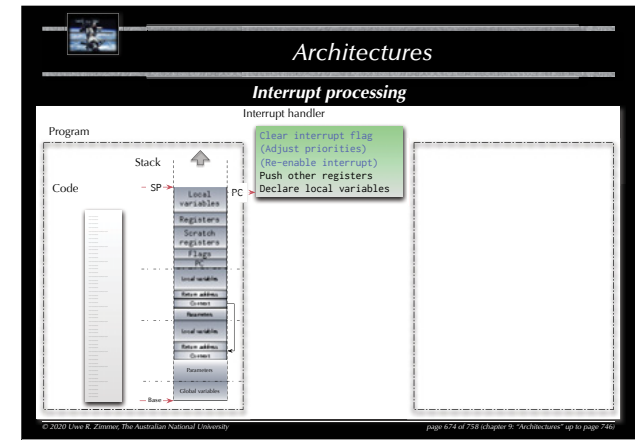
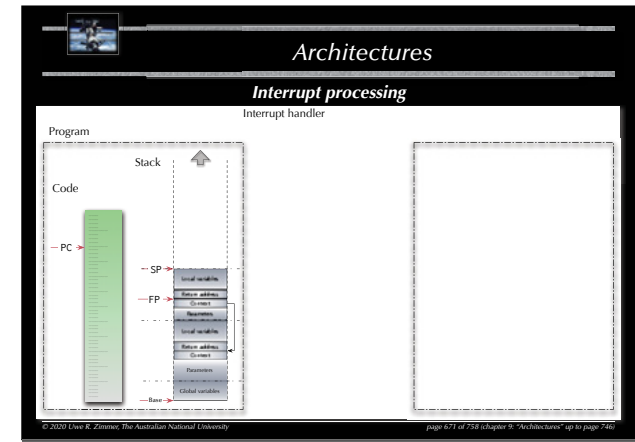
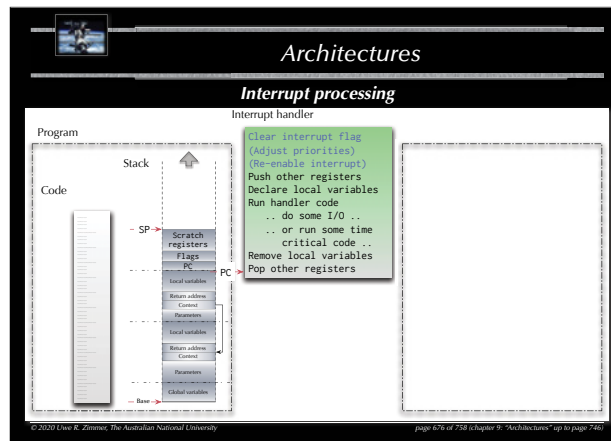
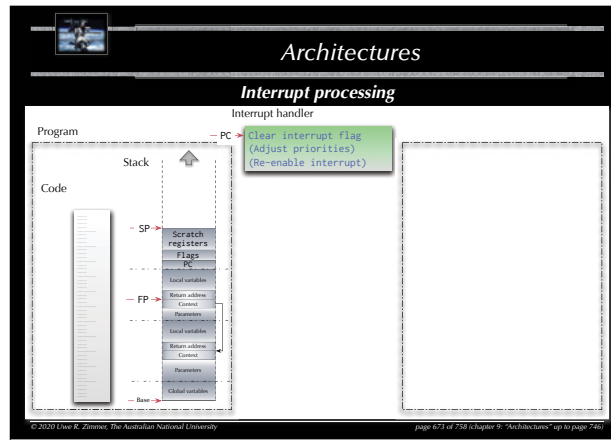
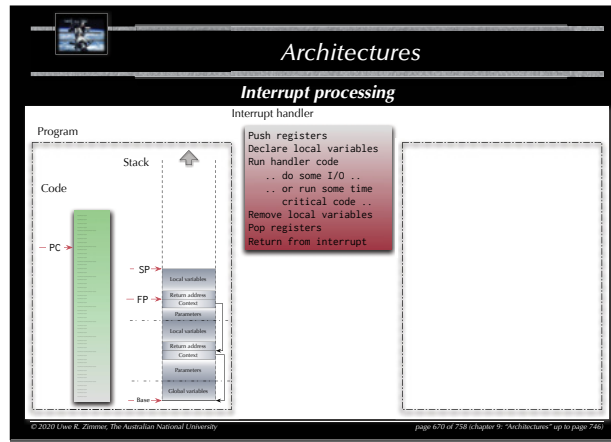
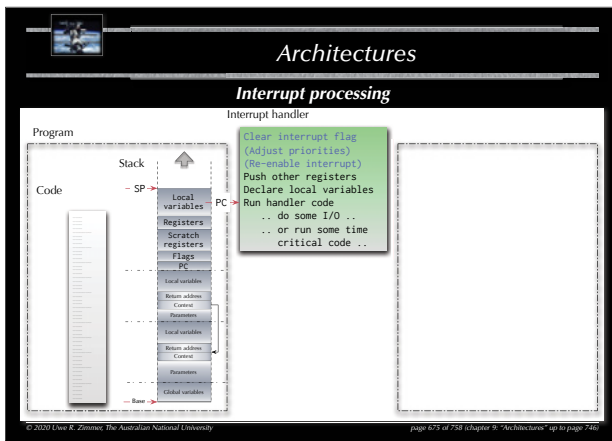
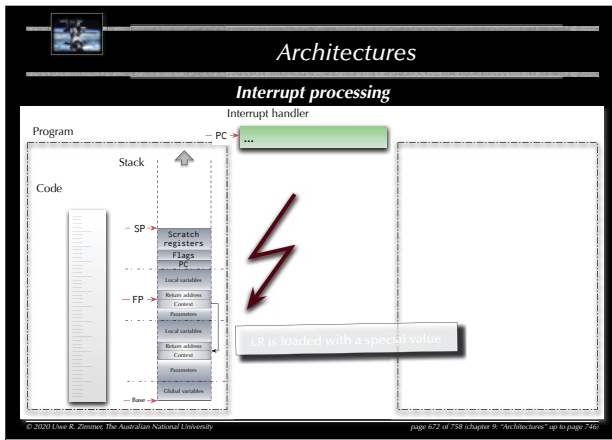
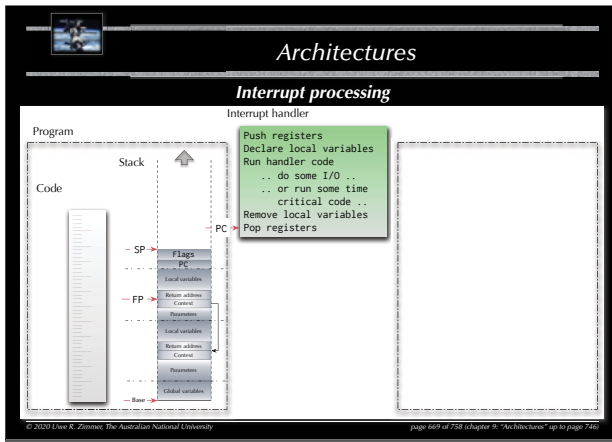
parameters

global variables

Interrupt handler

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

© 2020 Dave R. Zinin, The Australian National University page 668 of 758 chapter 9: "Architectures" up to page 746



### Architectures

#### Interrupt processing

Program

Stack

Code

PC

SP

FP

local variables

Return address

Context

Parameters

local variables

Return address

Context

Parameters

Global variables

Interrupt handler

local variables

Return address

Context

Parameters

local variables

Return address

Context

Parameters

Global variables

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

© 2010 Dave R. Zimmerman, The Australian National University page 678 of 758 chapter 9: "Architectures" up to page 746

### Architectures

#### Interrupt handler

#### Things to consider

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

© 2010 Dave R. Zimmerman, The Australian National University page 679 of 758 chapter 9: "Architectures" up to page 746

### Architectures

#### Interrupt handler

#### Things to consider

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

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

© 2010 Dave R. Zimmerman, The Australian National University page 680 of 758 chapter 9: "Architectures" up to page 746

### Architectures

#### Multiple programs

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

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

© 2010 Dave R. Zimmerman, The Australian National University page 681 of 758 chapter 9: "Architectures" up to page 746

### Architectures

#### Context switch

#### Dispatcher

Process 1

PCB

PID

Code

Stack

PC

SP

FP

local variables

Return address

Context

Parameters

local variables

Return address

Context

Parameters

Global variables

Process 2

PCB

PID

Code

Stack

PC

SP

FP

local variables

Return address

Context

Parameters

local variables

Return address

Context

Parameters

Global variables

Dispatcher

© 2010 Dave R. Zimmerman, The Australian National University page 682 of 758 chapter 9: "Architectures" up to page 746

### Architectures

#### Context switch

#### Dispatcher

Process 1

PCB

PID

Code

Stack

PC

SP

FP

local variables

Return address

Context

Parameters

local variables

Return address

Context

Parameters

Global variables

Process 2

PCB

PID

Code

Stack

PC

SP

FP

local variables

Return address

Context

Parameters

local variables

Return address

Context

Parameters

Global variables

Dispatcher

© 2010 Dave R. Zimmerman, The Australian National University page 683 of 758 chapter 9: "Architectures" up to page 746

### Architectures

#### Context switch

#### Dispatcher

Process 1

PCB

PID

Code

Stack

PC

SP

FP

local variables

Return address

Context

Parameters

local variables

Return address

Context

Parameters

Global variables

Process 2

PCB

PID

Code

Stack

PC

SP

FP

local variables

Return address

Context

Parameters

local variables

Return address

Context

Parameters

Global variables

Dispatcher

Push registers  
Declare local variables

© 2010 Dave R. Zimmerman, The Australian National University page 684 of 758 chapter 9: "Architectures" up to page 746

### Architectures

#### Context switch

#### Dispatcher

Process 1

PCB

PID

Code

Stack

PC

SP

FP

local variables

Return address

Context

Parameters

local variables

Return address

Context

Parameters

Global variables

Process 2

PCB

PID

Code

Stack

PC

SP

FP

local variables

Return address

Context

Parameters

local variables

Return address

Context

Parameters

Global variables

Dispatcher

Push registers  
Declare local variables  
Store SP to PCB 1

© 2010 Dave R. Zimmerman, The Australian National University page 685 of 758 chapter 9: "Architectures" up to page 746

### Architectures

#### Context switch

#### Dispatcher

Process 1

PCB

PID

Code

Stack

PC

SP

FP

local variables

Return address

Context

Parameters

local variables

Return address

Context

Parameters

Global variables

Process 2

PCB

PID

Code

Stack

PC

SP

FP

local variables

Return address

Context

Parameters

local variables

Return address

Context

Parameters

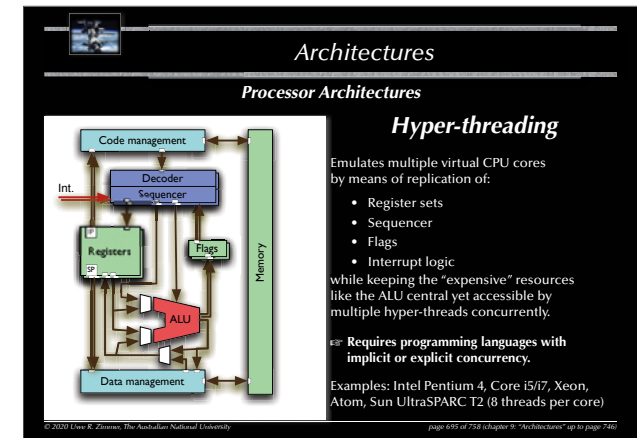
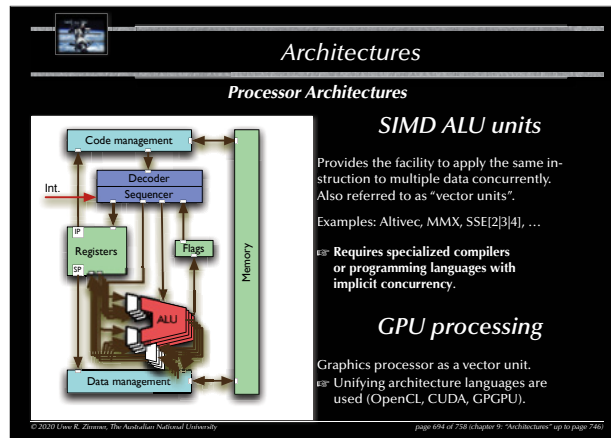
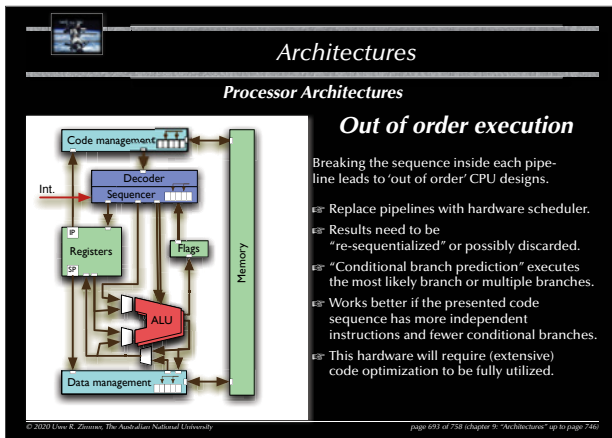
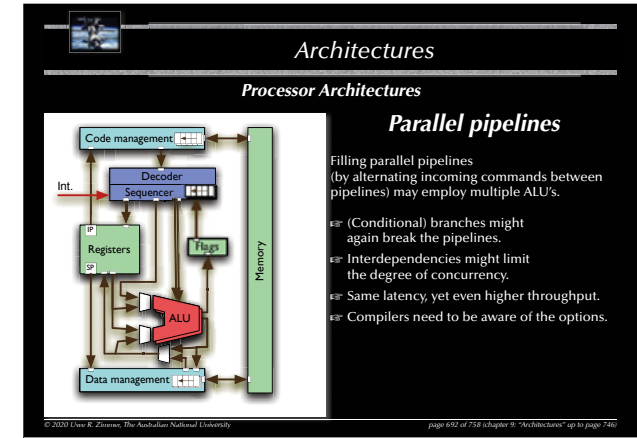
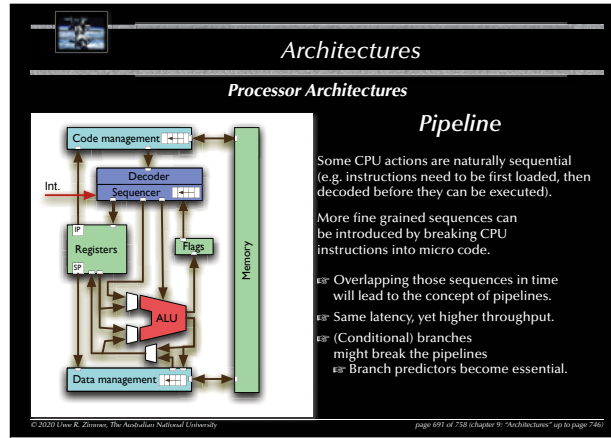
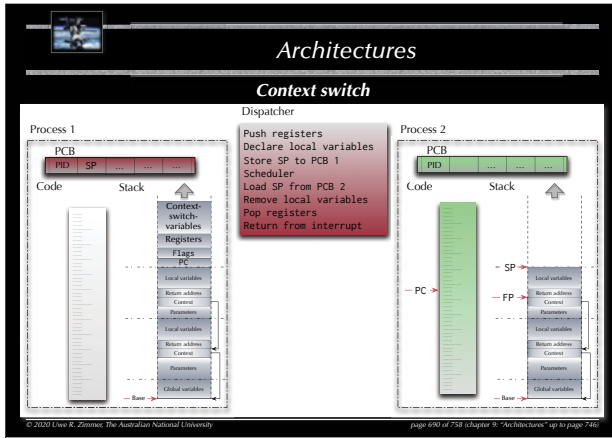
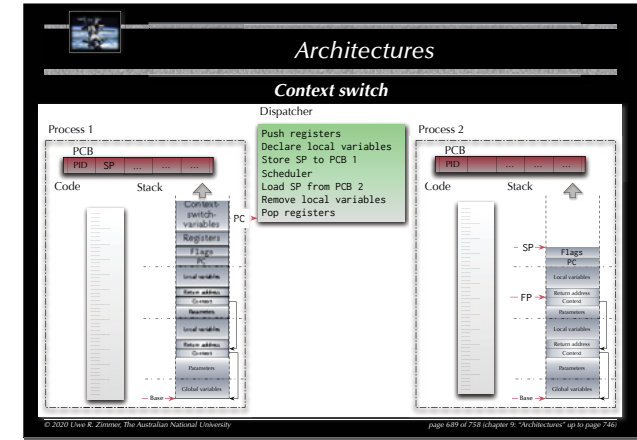
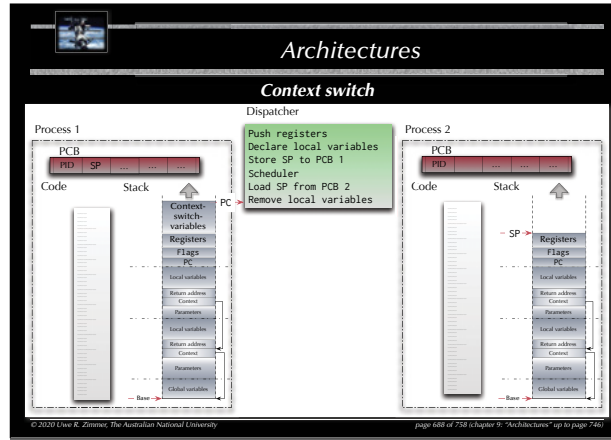
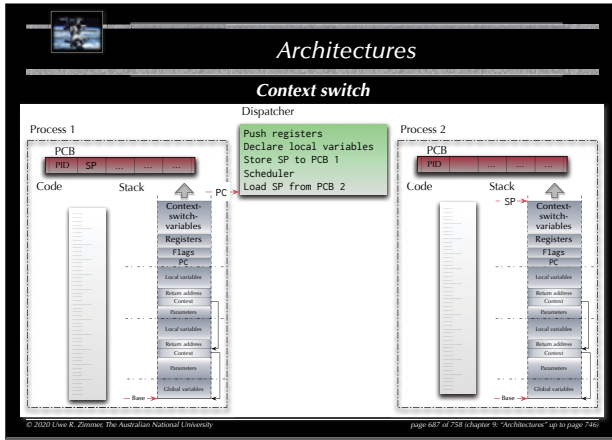
Global variables

Dispatcher

Push registers  
Declare local variables  
Store SP to PCB 1  
Scheduler

© 2010 Dave R. Zimmerman, The Australian National University page 686 of 758 chapter 9: "Architectures" up to page 746

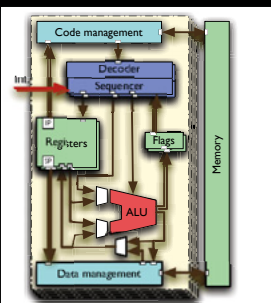




## Architectures

### Processor Architectures

#### Multi-core CPUs



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

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

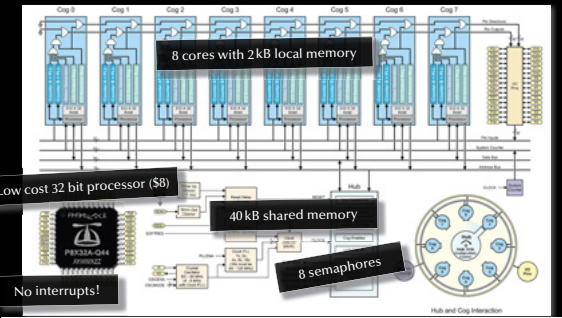
⚠ Requires synchronized atomic operations.  
⚠ Requires programming languages with implicit or explicit concurrency.

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

© 2010 Uwe R. Zimmer, The Australian National University page 696 of 758 chapter 9: "Architectures" up to page 746

## Architectures

### Alternative Processor Architectures: Parallax Propeller (2006)



8 cores with 2kB local memory

Low cost 32 bit processor (\$8)

40 kB shared memory

8 semaphores

No interrupts!

© 2010 Uwe R. Zimmer, The Australian National University page 699 of 758 chapter 9: "Architectures" up to page 746

## Architectures

### Vector Machines

#### Vectorization

**Buzzword collection:**  
AltVec, SPE, MMX, SSE, NEON, SPU, AVX, ...

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

Translates into CPU-level vector operations

```

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

```

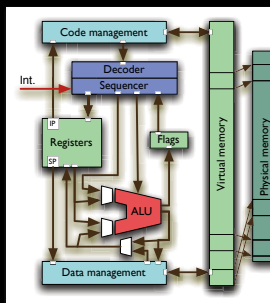
Combined with in-lining, loop unrolling and caching this is as fast as a single CPU will get.

© 2010 Uwe R. Zimmer, The Australian National University page 702 of 758 chapter 9: "Architectures" up to page 746

## Architectures

### Processor Architectures

#### Virtual memory



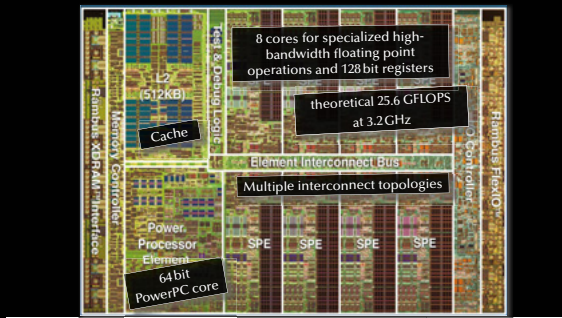
Translates logical memory addresses into physical memory addresses and provides memory protection features.

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

© 2010 Uwe R. Zimmer, The Australian National University page 697 of 758 chapter 9: "Architectures" up to page 746

## Architectures

### Alternative Processor Architectures: IBM Cell processor (2001)



8 cores for specialized high-bandwidth floating point operations and 128 bit registers

theoretical 25.6 CFLOPS at 3.2GHz

Cache

Element Interconnect Bus

Multiple interconnect topologies

Power Processor Element


64 bit PowerPC core

© 2010 Uwe R. Zimmer, The Australian National University page 700 of 758 chapter 9: "Architectures" up to page 746

## Architectures

### Vector Machines

#### Vectorization



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

Function is "promoted"

```

const Index = { 1 .. 1000000000 };
Vector_1 : [Index] real = 1.0,
Scale : real = 5.1,
Scaled : [Vector] real = Scale * Vector_1;

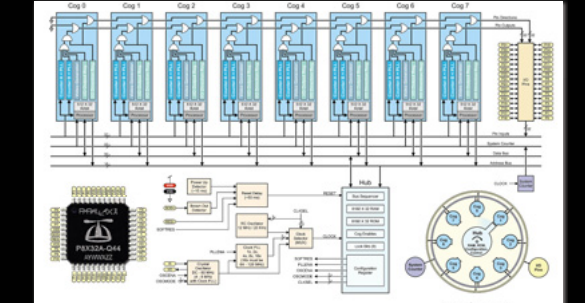
```

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

© 2010 Uwe R. Zimmer, The Australian National University page 703 of 758 chapter 9: "Architectures" up to page 746

## Architectures

### Alternative Processor Architectures: Parallax Propeller



8 cores with 2kB local memory

Low cost 32 bit processor (\$8)

40 kB shared memory

8 semaphores

No interrupts!



© 2010 Uwe R. Zimmer, The Australian National University page 698 of 758 chapter 9: "Architectures" up to page 746

## Architectures

### Multi-CPU systems

#### Scaling up:

- Multi-CPU on the same memory multiple CPUs on same motherboard and memory bus, e.g. servers, workstations
- Multi-CPU with high-speed interconnects various supercomputer architectures, e.g. Cray XE6:
  - 12-core AMD Opteron, up to 192 per cabinet (2304 cores)
  - 3D torus interconnect (160 GB/sec capacity, 48 ports per node)
- Cluster computer (Multi-CPU over network) multiple computers connected by network interface, e.g. Sun Constellation Cluster at ANU:
  - 1492 nodes, each: 2x Quad core Intel Nehalem, 24 GB RAM
  - QDR Infiniband network, 2.6GB/sec

© 2010 Uwe R. Zimmer, The Australian National University page 701 of 758 chapter 9: "Architectures" up to page 746

## Architectures

### Vector Machines

#### Reduction

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

Translates into CPU-level vector operations

^-chain is evaluated lazy sequentially.

© 2010 Uwe R. Zimmer, The Australian National University page 704 of 758 chapter 9: "Architectures" up to page 746

## Architectures

### Vector Machines

### Reduction

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

```

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

```

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

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

Function is "promoted"

© 2010 Dave R. Zimenc, The Australian National University page 705 of 758 chapter 9: "Architectures" up to page 746

## Architectures

### Operating Systems

What is an operating system?

© 2010 Dave R. Zimenc, The Australian National University page 708 of 758 chapter 9: "Architectures" up to page 746

## Architectures

### What is an operating system?

### 2. A resource manager!

... coordinating access to hardware resources

© 2010 Dave R. Zimenc, The Australian National University page 711 of 758 chapter 9: "Architectures" up to page 746

## Architectures

### Vector Machines

### General Data-parallelism

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

```

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

```

© 2010 Dave R. Zimenc, The Australian National University page 706 of 758 chapter 9: "Architectures" up to page 746

## Architectures

### What is an operating system?

### 1. A virtual machine!

... offering a more comfortable and safer environment

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

© 2010 Dave R. Zimenc, The Australian National University page 709 of 758 chapter 9: "Architectures" up to page 746

## Architectures

### What is an operating system?

### 2. A resource manager!

... coordinating access to hardware resources

Operating systems deal with

- processors
- memory
- mass storage
- communication channels
- devices (timers, special purpose processors, peripheral hardware, ...)

and tasks/processes/programs which are applying for access to these resources!

© 2010 Dave R. Zimenc, The Australian National University page 712 of 758 chapter 9: "Architectures" up to page 746

## Architectures

### Vector Machines

### General Data-parallelism

Cellular automaton transitions from a state into the next state :  
 $\rightarrow ' \Leftrightarrow \forall \in : \rightarrow ' = ( , )$ , i.e. all cells of a state transition **concurrently** into new cells by following a rule .

```

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

```

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

```

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

```

© 2010 Dave R. Zimenc, The Australian National University page 707 of 758 chapter 9: "Architectures" up to page 746

## Architectures

### What is an operating system?

### 1. A virtual machine!

... offering a more comfortable and safer environment

© 2010 Dave R. Zimenc, The Australian National University page 710 of 758 chapter 9: "Architectures" up to page 746

## Architectures

### The evolution of operating systems

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

© 2010 Dave R. Zimenc, The Australian National University page 713 of 758 chapter 9: "Architectures" up to page 746

## Architectures

### The evolution of communication systems

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

Current standard consumer computers might come with:

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

© 2010 Dave R. Zinner, The Australian National University page 714 of 758 (chapter 9: "Architectures" up to page 746)

## Architectures

### Types of current operating systems

Personal computing systems, workstations, and workgroup servers:

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

☞ last 20 years: evolving and expanding into current general purpose OSs, like for instance:

- Solaris (based on SVR4, BSD, and SunOS)
- LINUX (open source UNIX re-implementation for x86 processors and others)
- current Windows (proprietary, partly based on Windows NT, which is 'related' to VMS)
- MacOS X (Mach kernel with BSD Unix and a proprietary user-interface)

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

© 2010 Dave R. Zinner, The Australian National University page 715 of 758 (chapter 9: "Architectures" up to page 746)

## Architectures

### Types of current operating systems

Parallel operating systems

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

© 2010 Dave R. Zinner, The Australian National University page 716 of 758 (chapter 9: "Architectures" up to page 746)

## Architectures

### Types of current operating systems

Distributed operating systems

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

© 2010 Dave R. Zinner, The Australian National University page 717 of 758 (chapter 9: "Architectures" up to page 746)

## Architectures

### Types of current operating systems

Real-time operating systems

- Fast context switches?
- Small size?
- Quick response to external interrupts?
- Multitasking?
- 'low level' programming interfaces?
- Interprocess communication tools?
- High processor utilization?

© 2010 Dave R. Zinner, The Australian National University page 718 of 758 (chapter 9: "Architectures" up to page 746)

## Architectures

### Types of current operating systems

Real-time operating systems

- Fast context switches?
- Small size?
- Quick response to external interrupts?
- Multitasking?
- 'low level' programming interfaces?
- Interprocess communication tools?
- High processor utilization?

should be fast anyway  
should be small anyway  
not 'quick', but predictable  
often, not always  
needed in many operating systems  
needed in almost all operating systems  
fault tolerance builds on redundancy!

© 2010 Dave R. Zinner, The Australian National University page 719 of 758 (chapter 9: "Architectures" up to page 746)

## Architectures

### Types of current operating systems

Real-time operating systems need to provide...

☞ the logical correctness of the results as well as

☞ the correctness of the time, when the results are delivered

☞ Predictability! (not performance!)

☞ All results are to be delivered just-in-time – not too early, not too late.

Timing constraints are specified in many different ways ...  
... often as a response to 'external' events  
☞ reactive systems

© 2010 Dave R. Zinner, The Australian National University page 720 of 758 (chapter 9: "Architectures" up to page 746)

## Architectures

### Types of current operating systems

Embedded operating systems

- usually real-time systems, often hard real-time systems
- very small footprint (often a few KBs)
- none or limited user-interaction

☞ 90-95% of all processors are working here!

© 2010 Dave R. Zinner, The Australian National University page 721 of 758 (chapter 9: "Architectures" up to page 746)

## Architectures

### What is an operating system?

Is there a standard set of features for operating systems?

© 2010 Dave R. Zinner, The Australian National University page 722 of 758 (chapter 9: "Architectures" up to page 746)

## Architectures

### What is an operating system?

Is there a standard set of features for operating systems?

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

© 2020 Dave R. Zimmerman, The Australian National University page 723 of 758 (chapter 9: "Architectures" up to page 746)

## Architectures

### What is an operating system?

Is there a standard set of features for operating systems?

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

Is there a minimal set of features?

© 2020 Dave R. Zimmerman, The Australian National University page 724 of 758 (chapter 9: "Architectures" up to page 746)

## Architectures

### What is an operating system?

Is there a standard set of features for operating systems?

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

Is there a minimal set of features?

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

© 2020 Dave R. Zimmerman, The Australian National University page 725 of 758 (chapter 9: "Architectures" up to page 746)

## Architectures

### What is an operating system?

Is there a standard set of features for operating systems?

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

Is there a minimal set of features?

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

Is there always an explicit operating system?

© 2020 Dave R. Zimmerman, The Australian National University page 726 of 758 (chapter 9: "Architectures" up to page 746)

## Architectures

### What is an operating system?

Is there a standard set of features for operating systems?

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

Is there a minimal set of features?

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

Is there always an explicit operating system?

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

© 2020 Dave R. Zimmerman, The Australian National University page 727 of 758 (chapter 9: "Architectures" up to page 746)

## Architectures

### Typical features of operating systems

Process management:

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

**context switch:**

needs to...

- 'remove' one process from the CPU while preserving its state
- choose another process (scheduling)
- 'insert' the new process into the CPU, restoring the CPU state

Some CPUs have hardware support for context switching, otherwise:

use interrupt mechanism

© 2020 Dave R. Zimmerman, The Australian National University page 728 of 758 (chapter 9: "Architectures" up to page 746)

## Architectures

### Typical features of operating systems

Memory management:

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

Synchronisation / Inter-process communication

- semaphores, mutexes, cond. variables, channels, mailboxes, MPI, etc. (chapter 4)

tightly coupled to scheduling / task switching!

Hardware abstraction

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

© 2020 Dave R. Zimmerman, The Australian National University page 729 of 758 (chapter 9: "Architectures" up to page 746)

## Architectures

### Typical structures of operating systems

Monolithic  
(or 'the big mess...')

- non-portable
- hard to maintain
- lacks reliability
- all services are in the kernel (on the same privilege level)

but: may reach high efficiency

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

© 2020 Dave R. Zimmerman, The Australian National University page 730 of 758 (chapter 9: "Architectures" up to page 746)

## Architectures

### Typical structures of operating systems

Monolithic & Modular

- Modules can be platform independent
- Easier to maintain and to develop
- Reliability is increased
- all services are still in the kernel (on the same privilege level)

may reach high efficiency

e.g. current Linux versions

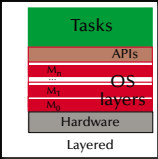
© 2020 Dave R. Zimmerman, The Australian National University page 731 of 758 (chapter 9: "Architectures" up to page 746)

## Architectures

### Typical structures of operating systems

#### Monolithic & layered

- easily portable
- significantly easier to maintain
- crashing layers do not necessarily stop the whole OS
- possibly reduced efficiency through many interfaces
- rigorous implementation of the stacked virtual machine perspective on OSs



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

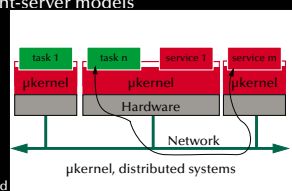
© 2020 Dave R. Ziminec, The Australian National University page 732 of 738 chapter 9: "Architectures" up to page 746

## Architectures

### Typical structures of operating systems

#### µKernels & client-server models

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



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

© 2020 Dave R. Ziminec, The Australian National University page 735 of 738 chapter 9: "Architectures" up to page 746

## Architectures

### UNIX

#### Dynamic process creation

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

resulting a *duplication of the current process*

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

Frequent usage:

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

© 2020 Dave R. Ziminec, The Australian National University page 738 of 738 chapter 9: "Architectures" up to page 746

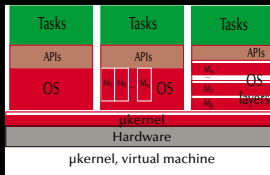
## Architectures

### Typical structures of operating systems

#### µKernels & virtual machines

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

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



© 2020 Dave R. Ziminec, The Australian National University page 733 of 738 chapter 9: "Architectures" up to page 746

## Architectures

### UNIX

#### UNIX features

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

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

© 2020 Dave R. Ziminec, The Australian National University page 736 of 738 chapter 9: "Architectures" up to page 746

## Architectures

### UNIX

#### Synchronization in UNIX ⇒ Signals

```
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
pid_t id;
void catch_stop (int sig_num)
{
    /* do something with the signal */
}

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

© 2020 Dave R. Ziminec, The Australian National University page 739 of 738 chapter 9: "Architectures" up to page 746

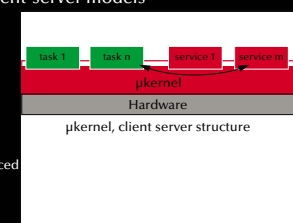
## Architectures

### Typical structures of operating systems

#### µKernels & client-server models

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

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



© 2020 Dave R. Ziminec, The Australian National University page 734 of 738 chapter 9: "Architectures" up to page 746

## Architectures

### UNIX

#### Dynamic process creation

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

resulting a *duplication of the current process*

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

© 2020 Dave R. Ziminec, The Australian National University page 737 of 738 chapter 9: "Architectures" up to page 746

## Architectures

### UNIX

#### Message passing in UNIX ⇒ Pipes

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

© 2020 Dave R. Ziminec, The Australian National University page 740 of 738 chapter 9: "Architectures" up to page 746

# Architectures

## UNIX

### Processes & IPC in UNIX

**Processes:**

- Process creation results in a duplication of address space ('copy-on-write' becomes necessary)
- inefficient, but can generate new tasks out of any user process – no shared memory!

**Signals:**

- limited information content, no buffering, no timing assurances (signals are **not** interrupts!)
- very basic, yet not very powerful form of synchronisation

**Pipes:**

- unstructured byte-stream communication, access is identical to file operations
- not sufficient to design client-server architectures or network communications

© 2010 Dave R. Zimmerman, The Australian National University page 741 of 758 chapter 9: "Architectures" up to page 746

# Architectures

## POSIX - some of the relevant standards...

1003.1 1201	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 1093	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 645	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 1099	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 1400	Advanced Real-time Extensions	typed memory, nanosleep improvements, barrier synchronization, reader/writer locks, spin locks, and persistent notification for message queues
1003.21 41-	Distributed Real-time	buffer management, send control blocks, asynchronous and synchronous operations, bounded blocking, message priorities, message labels, and implementation protocols

© 2010 Dave R. Zimmerman, The Australian National University page 744 of 758 chapter 9: "Architectures" up to page 746

# Architectures

## UNIX

### Sockets in BSD UNIX

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

**Connectionless interfaces (e.g. UDP/IP):**

- Server side: `socket -> bind -> recvfrom -> close`
- Client side: `socket -> sendto -> close`

**Connection oriented interfaces (e.g. TCP/IP):**

- Server side: `socket -> bind -> {select} [connect | listen -> accept -> read | write -> [close | shutdown]`
- Client side: `socket -> bind -> connect -> write | read -> [close | shutdown]`

© 2010 Dave R. Zimmerman, The Australian National University page 742 of 758 chapter 9: "Architectures" up to page 746

# Architectures

## POSIX - 1003.1b/c

### Frequently employed POSIX features include:

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

© 2010 Dave R. Zimmerman, The Australian National University page 745 of 758 chapter 9: "Architectures" up to page 746

# Architectures

## POSIX

### Portable Operating System Interface for Unix

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

© 2010 Dave R. Zimmerman, The Australian National University page 743 of 758 chapter 9: "Architectures" up to page 746

# Architectures


## Summary

### Architectures

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

© 2010 Dave R. Zimmerman, The Australian National University page 746 of 758 chapter 9: "Architectures" up to page 746

Systems, Networks & Concurrency 2020



# 10

Summary

Uwe R. Zimmer - The Australian National University

Summary

Summary

## Communication & 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
  - Examples

© 2020 Uwe R. Zimmer, The Australian National University page 750 of 758 (chapter 10: "Summary" up to page 758)

Summary

Summary

## Scheduling

- **Basic performance scheduling**
  - Motivation & Terms
  - Levels of knowledge / assumptions about the task set
  - Evaluation of performance and selection of appropriate methods
- **Towards predictable scheduling**
  - Motivation & Terms
  - Categories & Examples

© 2020 Uwe R. Zimmer, The Australian National University page 753 of 758 (chapter 10: "Summary" up to page 758)

Summary

Summary

## Concurrency – The Basic Concepts

- **Forms of concurrency**
- **Models and terminology**
  - Abstractions and perspectives: computer science, physics & engineering
  - Observations: non-determinism, atomicity, interaction, interleaving
  - Correctness in concurrent systems
- **Processes and threads**
  - Basic concepts and notions
  - Process states
- **Concurrent programming languages:**
  - Explicit concurrency: e.g. Ada, Chapel
  - Implicit concurrency: functional programming – e.g. Haskell, Caml

© 2020 Uwe R. Zimmer, The Australian National University page 748 of 758 (chapter 10: "Summary" up to page 758)

Summary

Summary

## Non-Determinism

- **Non-determinism by design:**
  - Benefits & considerations
- **Non-determinism by interaction:**
  - Selective synchronization
  - Selective accepts
  - Selective calls
- **Correctness of non-deterministic programs:**
  - Sources of non-determinism
  - Predicates & invariants

© 2020 Uwe R. Zimmer, The Australian National University page 751 of 758 (chapter 10: "Summary" up to page 758)

Summary

Summary

## Safety & Liveness

- **Liveness**
  - Fairness
- **Safety**
  - Deadlock detection
  - Deadlock avoidance
  - Deadlock prevention
- **Atomic & Idempotent operations**
  - Definitions & implications
- **Failure modes**
  - Definitions, fault sources and basic fault tolerance

© 2020 Uwe R. Zimmer, The Australian National University page 754 of 758 (chapter 10: "Summary" up to page 758)

Summary

Summary

## Mutual Exclusion

- **Definition of mutual exclusion**
- **Atomic load and atomic store operations**
  - ... some classical errors
  - Decker's algorithm, Peterson's algorithm
  - Bakery algorithm
- **Realistic hardware support**
  - Atomic test-and-set, Atomic exchanges, Memory cell reservations
- **Semaphores**
  - Basic semaphore definition
  - Operating systems style semaphores

© 2020 Uwe R. Zimmer, The Australian National University page 749 of 758 (chapter 10: "Summary" up to page 758)

Summary

Summary

## Data Parallelism

- **Data-Parallelism**
  - Vectorization
  - Reduction
  - General data-parallelism
- **Examples**
  - Image processing
  - Cellular automata

© 2020 Uwe R. Zimmer, The Australian National University page 752 of 758 (chapter 10: "Summary" up to page 758)

Summary


Summary

## Distributed Systems

- **Networks**
  - OSI, topologies
  - Practical network standards
- **Time**
  - Synchronized clocks, virtual (logical) times
  - Distributed critical regions (synchronized, logical, token ring)
- **Distributed systems**
  - Elections
  - Distributed states, consistent snapshots
  - Distributed servers (replicates, distributed processing, distributed commits)
  - Transactions (ACID properties, serializable interleavings, transaction schedulers)

© 2020 Uwe R. Zimmer, The Australian National University page 755 of 758 (chapter 10: "Summary" up to page 758)






## Summary

### Summary

### Architectures

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

© 2020 Dave R. Zimmerman, The Australian National University page 756 of 758 (chapter 10: "Summary" up to page 758)



## Summary

### Exam preparations

### Helpful

- Distinguish central aspects from excursions, examples & implementations.
- **Gain** full understanding of all central aspects.
- Be able to **categorize** any given example under a general theme discussed in the lecture.
- **Explain** to and **discuss** the topics with other (preferably better) students.
- Try whether you can **connect** aspects from different parts of the lecture.

### Not helpful

- Remembering the slides word by word.
- Learn the Chapel / Unix / Posix / Occam / sockets reference manual page by page.

© 2020 Dave R. Zimmerman, The Australian National University page 757 of 758 (chapter 10: "Summary" up to page 758)

