Concurrent & Distributed Systems 2010

Uwe R. Zimmer - The Australian National University

Concurrent & Distributed Systems 2010

Organization & Contents

Uwe R. Zimmer - The Australian National University

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

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

who are these people? – introduction

This course will be given by

Felix Schill & Uwe R. Zimmer

Tutoring and labs by

xxx

Organization & Contents

how will this all be done?

☞ Lectures:

- 3 per week … all the nice stuff
  Monday, 15:00 (PHYS-T1); Wednesday 9:00 (COP-G031); Friday 14:00 (CHEM-T)

☞ Laboratories:

- 2 hours per week … all the rough stuff
  time slots: on our web-site – all in CSIT N114 laboratory
  -enrolment: https://cs.anu.edu.au/streams/

☞ Resources:

- Introduced in the lectures and collected on the course page:
  http://cs.anu.edu.au/student/comp2310/ … as well as schedules,
  slides, sources, forums, etc. pp. … keep an eye on this page!

☞ Assessment:

- Exam at the end of the course (70%) plus two assign-
  ments (2 x 15%), and mid-term check (0%)

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.*

Organization & Contents

Topics

1. *Concurrency [3]*
2. *Mutual exclusion [3]*
3. *Condition synchronization [4]*
4. *Non-determinism in concurrent systems [2]*
5. *Scheduling [2]*
6. *Safety and liveness [3]*
7. *Architectures for CDS [3]*
8. *Distributed systems [8]*

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 [3]*
3. *Condition synchronization [4]*
4. *Non-determinism in concurrent systems [2]*
5. *Scheduling [2]*
6. *Safety and liveness [3]*
7. *Architectures for CDS [3]*
8. *Distributed systems [8]*

Organization & Contents

Topics

1. ***Concurrency [3]***
2. ***Mutual exclusion [3]***

2.1. by shared variables [2]

- 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

3. ***Condition synchronization [4]***
4. ***Non-determinism in concurrent systems [2]***
5. ***Scheduling [2]***
6. ***Safety and liveness [3]***
7. ***Architectures for CDS [3]***
8. ***Distributed systems [8]***

Organization & Contents

Topics

1. *Concurrency [3]*
2. *Mutual exclusion [3]*
3. *Condition synchronization [4]*

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

4. *Non-determinism in concurrent systems [2]*
5. *Scheduling [2]*
6. *Safety and liveness [3]*
7. *Architectures for CDS [3]*
8. *Distributed systems [8]*

Organization & Contents

Topics

1. *Concurrency [3]*
2. *Mutual exclusion [3]*
3. *Condition synchronization [4]*
4. *Non-determinism in concurrent systems [2]*

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

5. *Scheduling [2]*
6. *Safety and liveness [3]*
7. *Architectures for CDS [3]*
8. *Distributed systems [8]*

Organization & Contents

Topics

1. *Concurrency [3]*
2. *Mutual exclusion [3]*
3. *Condition synchronization [4]*
4. *Non-determinism in concurrent systems [2]*
5. *Scheduling [2]*

5.1. Problem definition and design space [1]

- Which problems are addressed / solved by scheduling?

5.2. Basic scheduling methods [1]

- Assumptions for basic scheduling
- Basic methods

6. *Safety and liveness [3]*
7. *Architectures for CDS [3]*
8. *Distributed systems [8]*

Organization & Contents

Topics

1. *Concurrency [3]*
2. *Mutual exclusion [3]*
3. *Condition synchronization [4]*
4. *Non-determinism in concurrent systems [2]*
5. *Scheduling [2]*
6. *Safety and liveness [3]*

6.1. Safety properties
- Essential time-independent safety properties

6.2. Livelocks, fairness
- Forms of livelocks
- Classification of fairness

6.3. Deadlocks
- Detection
- Avoidance
- Prevention (& recovery)

6.4. Failure modes

6.5. Idempotent & atomic operations
- Definitions

7. *Architectures for CDS [3]*
8. *Distributed systems [8]*

Organization & Contents

Topics

1. *Concurrency [3]*
2. *Mutual exclusion [3]*
3. *Condition synchronization [4]*
4. *Non-determinism in concurrent systems [2]*
5. *Scheduling [2]*
6. *Safety and liveness [3]*
7. *Architectures for CDS [3]*

7.1. **Hardware architecture**

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

7.2. **Operating system Architecture**

- Definitions
- Desktop OS
- Embedded OS
- Realtime OS
- Distributed OS

7.3. **Language architecture**

- Chapel
- Occam
- Go

8. *Distributed systems [8]*

Organization & Contents

Topics

1. *Concurrency [3]*
2. *Mutual exclusion [3]*
3. *Condition synchronization [4]*
4. *Non-determinism in concurrent systems [2]*
5. *Scheduling [2]*
6. *Safety and liveness [3]*
7. *Architectures for CDS [3]*
8. *Distributed systems [8]*

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
- Dynamical groups

8.5. Distributed safety and liveness [1]
- Distributed deadlock detection

8.6. Forms of distribution/ redundancy [1]
- computation
- memory
- operations

8.7. Transactions [2]

Organization & Contents

28 Lectures

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 [3]

2.1. by shared variables [2]
- 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

3. Condition synchronization [4]

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

4. Non-determinism in concurrent systems [2]

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

5. Scheduling [2]

5.1. Problem definition and design space [1]
- Which problems are addressed / solved by scheduling?

5.2. Basic scheduling methods [1]
- Assumptions for basic scheduling
- Basic methods

6. Safety and liveness [3]

6.1. Safety properties
- Essential time-independent safety properties

6.2. Livelocks, fairness
- Forms of livelocks
- Classification of fairness

6.3. Deadlocks
- Detection
- Avoidance
- Prevention (& recovery)

6.4. Failure modes

6.5. Idempotent & atomic operations
- Definitions

7. Architectures for CDS [3]

7.1. Hardware architecture
- From switches to registers and adders
- CPU architecture
- Hardware concurrency

7.2. Operating system Architecture
- Definitions
- Desktop OS
- Embedded OS
- Realtime OS
- Distributed OS
- Parallel OS

7.3. Language architecture
- Chapel
- Occam
- Ada
- Go

8. Distributed systems [8]

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
- Dynamical groups

8.5. Distributed safety and liveness [1]
- Distributed deadlock detection

8.6. Forms of distribution/redundancy [1]
- computation
- memory
- operations

8.7. Transactions [2]

Laboratories & Assignments

Laboratories

1. Concurrency language support basics (in Ada95) [3]

1.1. Structured, strongly typed programming
- Program structures
- Data structures

1.2. Generic, re-usable programming
- Generics
- Abstract types

1.3. Concurrent processes:
- Creation
- Termination
- Rendezvous

2. Concurrent programming [3]

2.1. Synchronization
- Protected objects

2.2. Remote invocation
- Extended rendezvous

2.3. Client-Server architectures
- Entry families
- Requeue facility

3. Concurrency in a multi-core system[3]

3.1. Multi-core process creation, termination

3.2. Multi-core process communication

Assignments

1. Concurrent programming [20%]

Ada95 programming task involving:
- Mutual exclusion
- Synchronization
- Message passing

2. Concurrent programming in multi-core systems [10%]

Multi-core programming task involving:
- Process communication

Examinations

1. Mid-term check [10%]
- Test question set with supplied answers [not marked]

2. Final exam [60%]
- Examining the complete lecture

Marking

The final mark is based on
the assignments [30%]
plus the mid-term [10%]
plus the final examination [70%]

Ada refresher / introduction course

Uwe R. Zimmer - The Australian National University

References for this chapter

[Cohen96]
Norman H. Cohen
Ada as a second language
McGraw-Hill series in computer science, 2nd edition, 1996

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

[Ada 2005 Reference manual]
see course pages or http://www.adaic.org/standards/ada05.html

Languages explicitly supporting concurrency: e.g. Ada2005

Ada2005 is an ISO standardized (ISO/IEC 8652:1995/Amd 1:2007) 'general purpose' language which "promotes reliability and simplify maintenance" while keeping maximal efficiency and provides core language primitives for:

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

… as well as standardized language-annexes for

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

Ada2005

A crash course

… *refreshing for some, second-language introduction for others*:

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

… not mentioned here: basic object orientation (‘tagged types’), language interfaces, marshalling, basics of imperative programming, …

Ada2005

Basics

… introducing:

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

A simple queue specification

```ada
package Queue_Pack_Simple is

   QueueSize : constant Positive := 10;

   type Element      is new Positive range 1_000..40_000;
   type Marker       is mod QueueSize;
   type List         is array (Marker'Range) of Element;

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

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

end Queue_Pack_Simple;
```

A simple queue implementation

```
package body Queue_Pack_Simple is

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

   begin
      Queue.Elements (Queue.Free) := Item;
      Queue.Free := Queue.Free - 1;
   end Enqueue;

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

   begin
      Item := Queue.Elements (Queue.Top);
      Queue.Top := Queue.Top - 1;
   end Dequeue;

end Queue_Pack_Simple;
```


A simple queue test program

```ada
with Queue_Pack_Simple; use Queue_Pack_Simple;
procedure Queue_Test_Simple is

   Queue : Queue_Type;
   Item  : Element;

begin
   Enqueue (2000, Queue);
   Dequeue (Item, Queue);
   Dequeue (Item, Queue); -- will produce an unpredictable result!
end Queue_Test_Simple;
```

Ada2005

Exceptions

… introducing:

- **exception** handling
- **enumeration** types
- type **attributed operators**

A queue specification with proper exceptions

```ada
package Queue_Pack_Exceptions is

   QueueSize : constant Integer := 10;

   type Element     is (Up, Down, Spin, Turn);
   type Marker      is mod QueueSize;
   type List        is array (Marker'Range) of Element;
   type Queue_State is (Empty, Filled);

   type Queue_Type  is record
      Top, Free : Marker      := Marker'First;
      State     : Queue_State := Empty;
      Elements  : List;
   end record;

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

   Queueoverflow, Queueunderflow : exception;

end Queue_Pack_Exceptions;
```

A queue *implementation* with proper exceptions

```
package body Queue_Pack_Exceptions is

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

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

end Queue_Pack_Exceptions;
```

A queue test program with proper exceptions

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

procedure Queue_Test_Exceptions is
   Queue : Queue_Type;
   Item  : Element;

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

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

end Queue_Test_Exceptions;
```

Ada2005

Information hiding

… introducing:

- **private** ☞ assignments and comparisons are allowed
- **limited private** ☞ entity cannot be assigned or compared

A queue specification with proper information hiding

```ada
package Queue_Pack_Private is

   QueueSize : constant Integer := 10;

   type Element is new Positive range 1..1000;
   type Queue_Type is limited private;

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

   Queueoverflow, Queueunderflow : exception;

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

end Queue_Pack_Private;
```

A queue implementation with proper information hiding

```
package body Queue_Pack_Private is

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

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

end Queue_Pack_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;
```

Ada2005

Generic packages

… introducing:

- specification of **generic** packages
- instantiation of **generic** packages

A generic queue *specification*

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

A generic queue implementation

```
package body Queue_Pack_Generic is

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

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

end Queue_Pack_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;
```

Ada2005

Tasks & Monitors

… introducing:

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

A protected queue *specification*

```
package Queue_Pack_Protected is

   QueueSize : constant Integer := 10;

   subtype Element is Character;
   type Queue_Type is limited private;

   protected type Protected_Queue is
      entry Enqueue (Item: in  Element);
      entry Dequeue (Item: out Element);
   private
      Queue : Queue_Type;
   end Protected_Queue;

private
   type Marker      is mod QueueSize;
   type List        is array (Marker'Range) of Element;
   type Queue_State is (Empty, Filled);

   type Queue_Type is record
      Top, Free : Marker      := Marker'First;
      State     : Queue_State := Empty;
      Elements  : List;
   end record;

end Queue_Pack_Protected;
```

A protected queue *implementation*

```
package body Queue_Pack_Protected is

   protected body Protected_Queue is

      entry Enqueue (Item: in Element) when
         Queue.State = Empty or Queue.Top /= Queue.Free is

      begin
         Queue.Elements (Queue.Free) := Item;
         Queue.Free  := Queue.Free - 1;
         Queue.State := Filled;
      end Enqueue;

      entry Dequeue (Item: out Element) when Queue.State = Filled is

      begin
         Item := Queue.Elements (Queue.Top);
         Queue.Top := Queue.Top - 1;
         if Queue.Top = Queue.Free then
            Queue.State := Empty;
         end if;
      end Dequeue;

   end Protected_Queue;

end Queue_Pack_Protected;
```

A protected queue test program

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

procedure Queue_Test_Protected is
   Queue : Protected_Queue;

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

   task body Producer is
      Item   : Element;
      Got_It : Boolean;

   begin
      loop

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

      end loop;
   end Producer; (...)
```

A protected queue test program (cont.)

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

Ada2005

Abstract types & dispatching

… introducing:

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

An abstract queue specification

```
package Queue_Pack_Abstract is

   subtype Element is Character;

   type Queue_Type is abstract tagged limited private;

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

private
   type Queue_Type is abstract tagged limited null record;

end Queue_Pack_Abstract;
```

… obviously this does not require an implementation package (as all procedures are `abstract`)

A concrete queue specification

```
with Queue_Pack_Abstract; use Queue_Pack_Abstract;

package Queue_Pack_Concrete is

   QueueSize : constant Integer := 10;

   type Real_Queue is new Queue_Type with private;

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

   Queueoverflow, Queueunderflow : exception;

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

A concrete queue *implementation*

```
package body Queue_Pack_Concrete is

   procedure Enqueue (Item: in Element; Queue: in out Real_Queue) is

   begin
      if Queue.State = Filled and Queue.Top = Queue.Free then
         raise Queueoverflow;
      end if;
      Queue.Elements (Queue.Free) := Item;
      Queue.Free  := Queue.Free - 1;
      Queue.State := Filled;
   end Enqueue;

   procedure Dequeue (Item: out Element; Queue: in out Real_Queue) is

   begin
      if Queue.State = Empty then
         raise Queueunderflow;
      end if;
      Item := Queue.Elements (Queue.Top);
      Queue.Top := Queue.Top - 1;
      if Queue.Top = Queue.Free then
         Queue.State := Empty;
      end if;
   end Dequeue;

end Queue_Pack_Concrete;
```

A dispatching test program

```
with Queue_Pack_Abstract; use Queue_Pack_Abstract;
with Queue_Pack_Concrete; use Queue_Pack_Concrete;

procedure Queue_Test_Dispatching is

   type Queue_Class is access all Queue_Type'class;

   task Queue_Holder is -- could be on an individual partition / computer
      entry Queue_Filled;
   end Queue_Holder;

   task Queue_User is    -- could be on an individual partition / computer
      entry Transmit_Queue (Remote_Queue: in Queue_Class);
   end Queue_User;
(...)
```

A dispatching test program (cont.)

```
task body Queue_Holder is

   Local_Queue : Queue_Class := new Real_Queue; -- any Queue_Type'class instance
   Item        : Element;

begin
   Queue_User.Transmit_Queue (Local_Queue); -- entry call between tasks
   accept Queue_Filled do
      Dequeue (Item, Local_Queue.all);      -- Item will be 'r'
   end Queue_Filled;
end Queue_Holder;

task body Queue_User is

   Local_Queue : Queue_Class := new Real_Queue; -- any Queue_Type'class instance
   Item : Element;

begin
   accept Transmit_Queue (Remote_Queue: in Queue_Class) do
      Enqueue ('r', Remote_Queue.all); -- potentially a remote procedure call
      Enqueue ('l', Local_Queue.all);  -- local procedure call
   end Transmit_Queue;
   Queue_Holder.Queue_Filled;          -- entry call between tasks
   Dequeue (Item, Local_Queue.all);    -- Item will be 'l'
end Queue_User;

begin null; end Queue_Test_Dispatching;
```

Ada2005 language status

- Established language standard with free and commercial 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 language standard
  ☞ Ravenscar profile.

☞ Used in many large scale and/or high integrity projects
  – frequently in the avionics industry, high speed trains, metro-systems, space programs …
  – but also increasingly on small platforms / micro-controllers

Ada refresher / intro course

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

Concurrent & Distributed Systems 2010

1

Introduction to Concurrency

Uwe R. Zimmer - The Australian National University

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

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.

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 kernel computer architectures

  … *but* almost all current processor architectures have **concurrent elements**

  … and *most* computer systems are part of a **concurrent network**
- Strict sequential processing is suggested by the most widely used programming languages

  … which is a reason why you might believe that concurrent computing is rare/exotic/hard

☞ sequential programming delivers some *fundamental components* for concurrent programming

☞ *but we need to add a number of further crucial concepts*

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* where sequential and/or concurrent programming is **required**
  … or: to understand where sequential or concurrent programming can be **chosen freely**
- ☞ … to *enhance* the **reactivity** of a system
- ☞ …

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

A computer scientist's view on concurrency

Terminology for real parallel machines architectures:

- SISD

  [singe instruction, single data]

  ☞ standard sequential processors

- SIMD

  [singe instruction, multiple data]

  ☞ vector processors

- MISD

  [multiple instruction, single data]

  ☞ pipelines

- MIMD

  [multiple instruction, multiple data]

  ☞ multiprocessors or computer networks

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**.

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** ☞ 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

☞ Control & monitor where required (and do it right), but not more …

Concurrency on different abstraction levels/perspectives

☞ **Networks**

- Multi-CPU network nodes and other specialized sub-networks
- Single-CPU network nodes – still including buses & I/O sub-systems
- Single-CPUs
- Operating systems (& distributed operating systems)

☞ **Processes & threads**

☞ **High-level concurrent programming**

☞ **Assembler level concurrent programming**

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

The concurrent programming abstraction

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

   ☞ e.g. low-level concurrent I/O drivers, 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:

   ☞ e.g. Multi-processing systems, which are executed on a single, sequential CPU

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:

  "*Concurrent programming abstraction* **is the study of interleaved execution sequences of the atomic instructions of sequential processes.**"
  (Ben-Ari)

Models and Terminology

The concurrent programming abstraction

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

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.

Models and Terminology

The concurrent programming abstraction

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

☞ 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

Models and Terminology

The concurrent programming abstraction

Time-line or Sequence?

Consider time (durations) explicitly:

☞ Real-time systems ☞ join the appropriate courses

Consider the sequence of interaction points only:

☞ Non-real-time systems ☞ stay here

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 on all possible sequences of interaction points

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’

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.

Models and Terminology

The concurrent programming abstraction

Standard concepts of correctness:

- **Partial correctness**:

$$(P(I) \wedge terminates(Program(I,O) \Rightarrow Q(I,O))$$

- **Total correctness**:

$$P(I) \Rightarrow (terminates(Program(I,O) \wedge Q(I,O))$$

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?

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:

$$(P(I) \wedge Processes(I, S)) \Rightarrow \Box Q(I, S)$$

where $\Box Q$ means that Q does *always* hold

Liveness properties:

$$(P(I) \wedge Processes(I, S)) \Rightarrow \Diamond Q(I, S)$$

where $\Diamond Q$ means that Q does *eventually* hold (and will then stay true)
and S is the current state of the concurrent system

The concurrent programming abstraction

Safety properties:

$$(P(I) \wedge Processes(I,S)) \Rightarrow \Box Q(I,S)$$

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)

Models and Terminology

The concurrent programming abstraction

Liveness properties:

$$(P(I) \wedge Processes(I,S)) \Rightarrow \Diamond Q(I,S)$$

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

Introduction to processes and threads

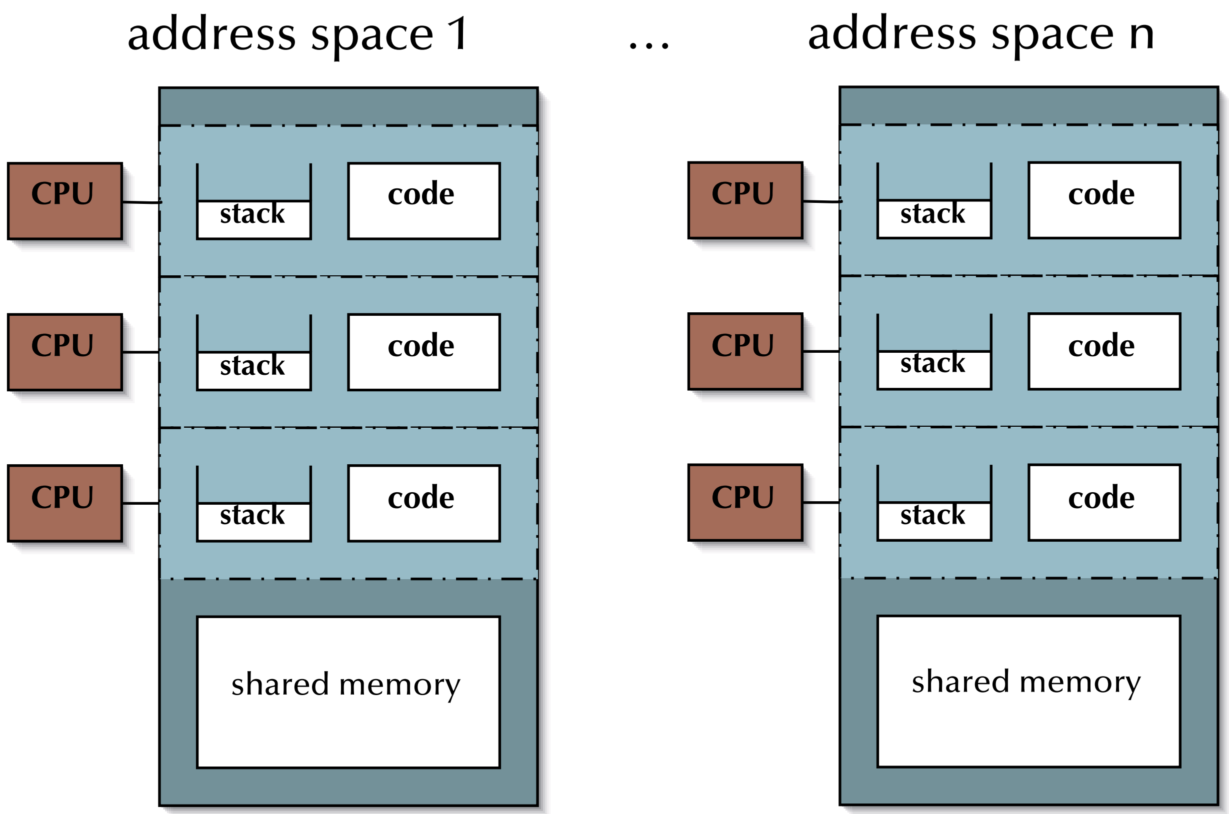
1 CPU per control-flow

for specific configurations only:

- distributed μcontrollers
- physical process control systems:

  1 cpu per task,
  connected via a typ. fast bus-system (VME, PCI)

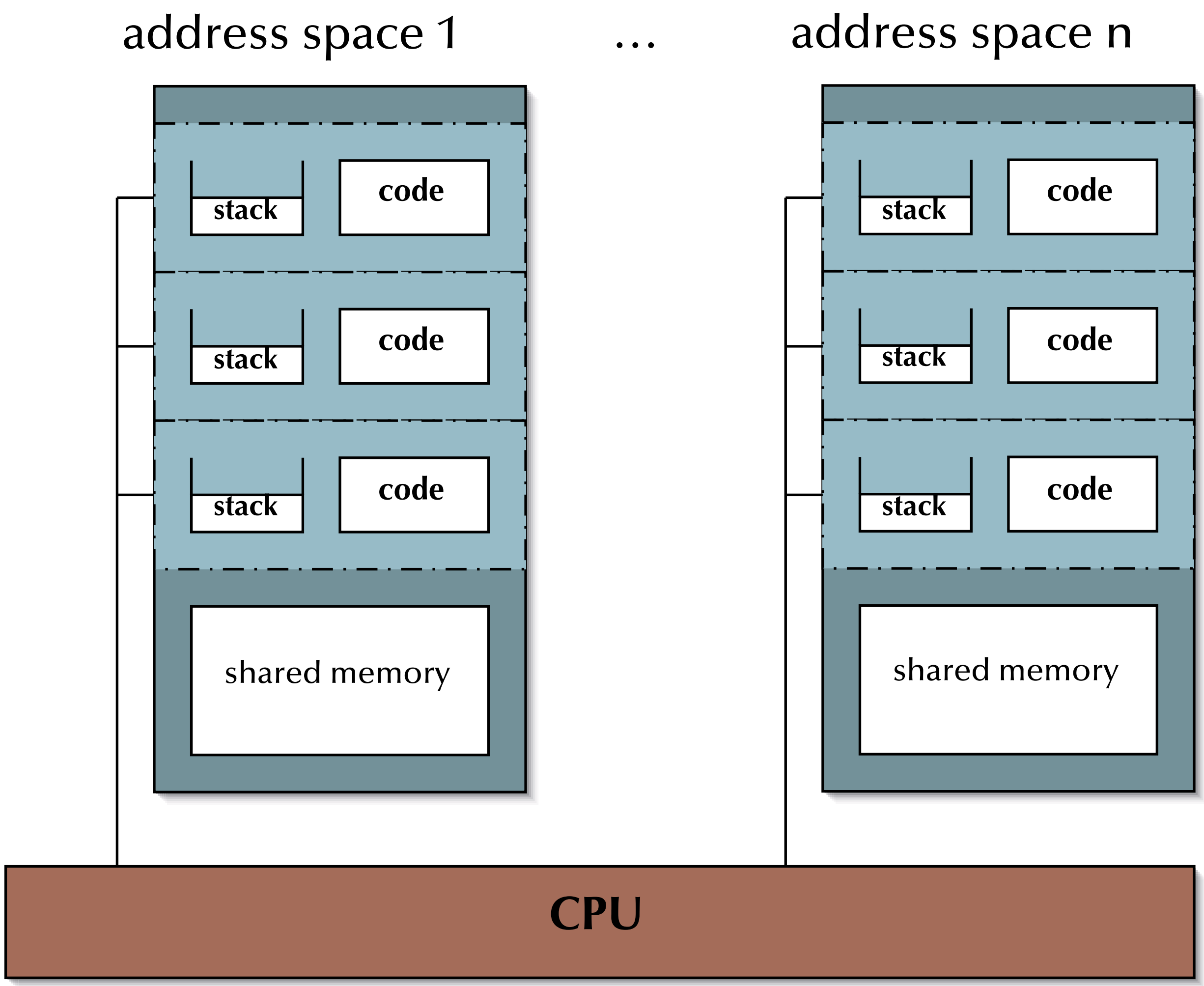
☞ no need for process management



Introduction to processes and threads

1 CPU for all control-flows

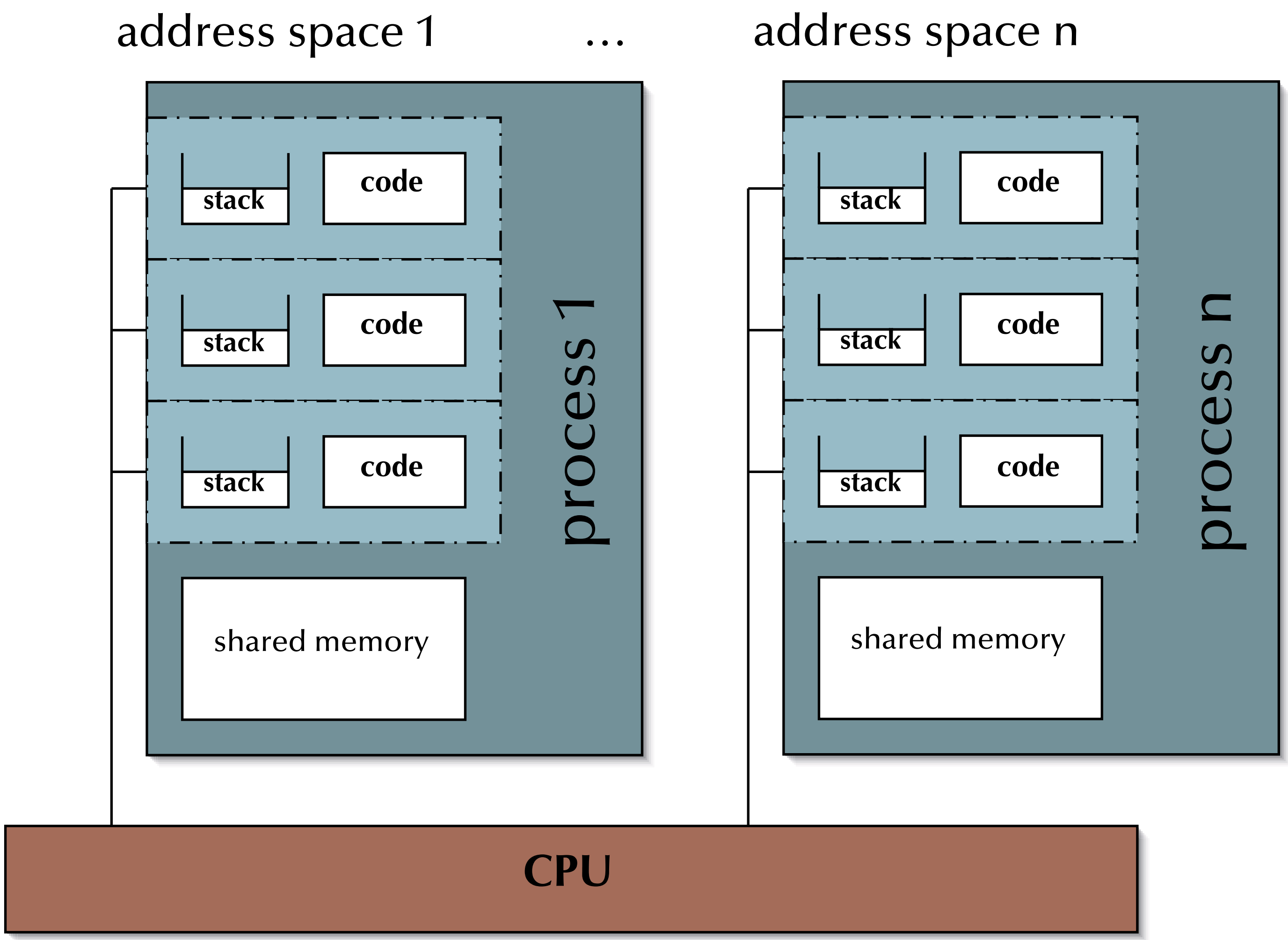
- OS: emulate one CPU for every control-flow
  - ☞ Multi-tasking operating system
- Support for memory protection becomes essential



Introduction to processes and threads

Processes

- Process ::=
  address space
  + control flow(s)
- Kernel has full knowledge about all processes as well as their requirements and current resources (see below)

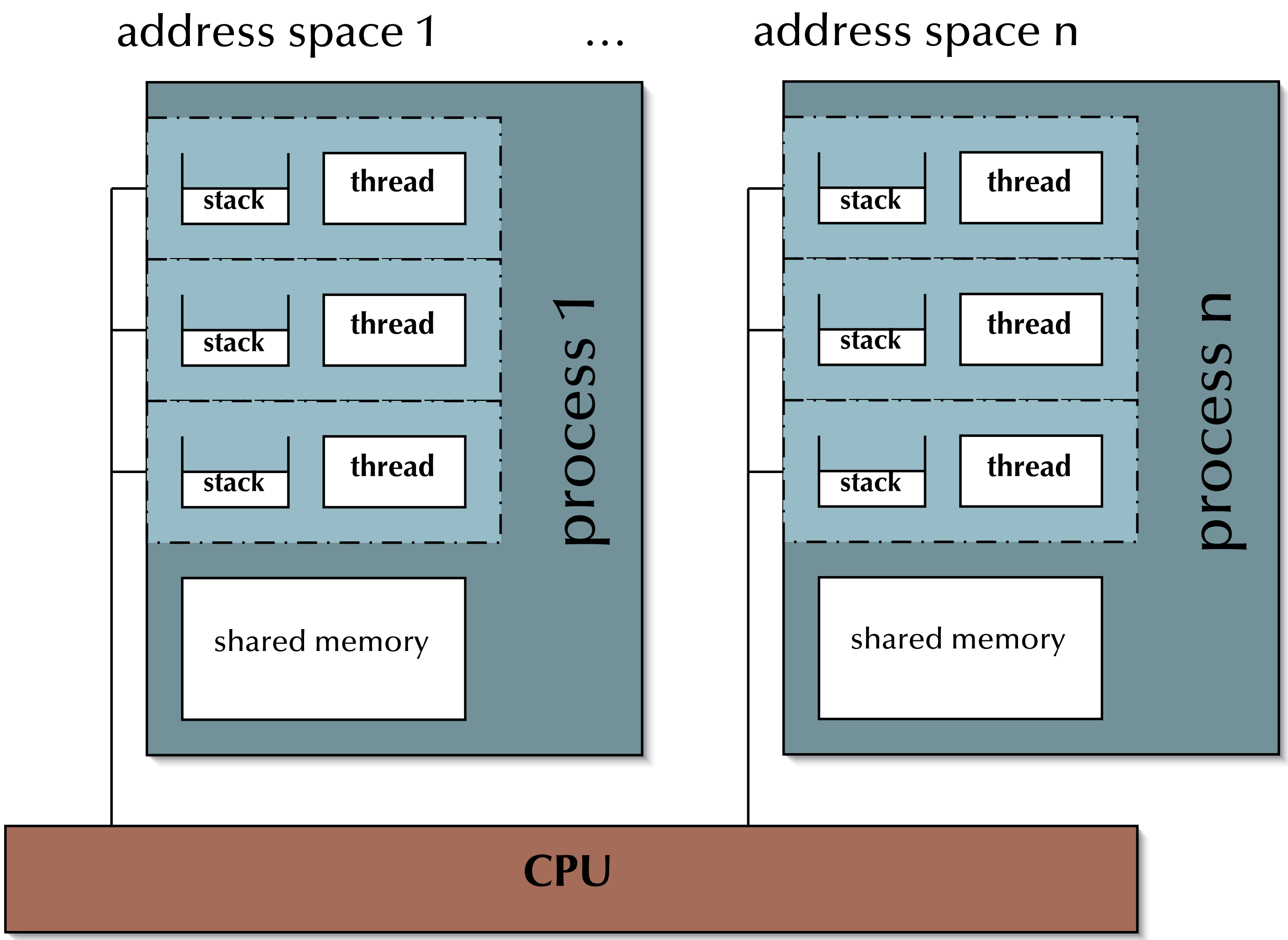


Introduction to processes and threads

Threads

Threads (individual control-flows) can be handled:

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

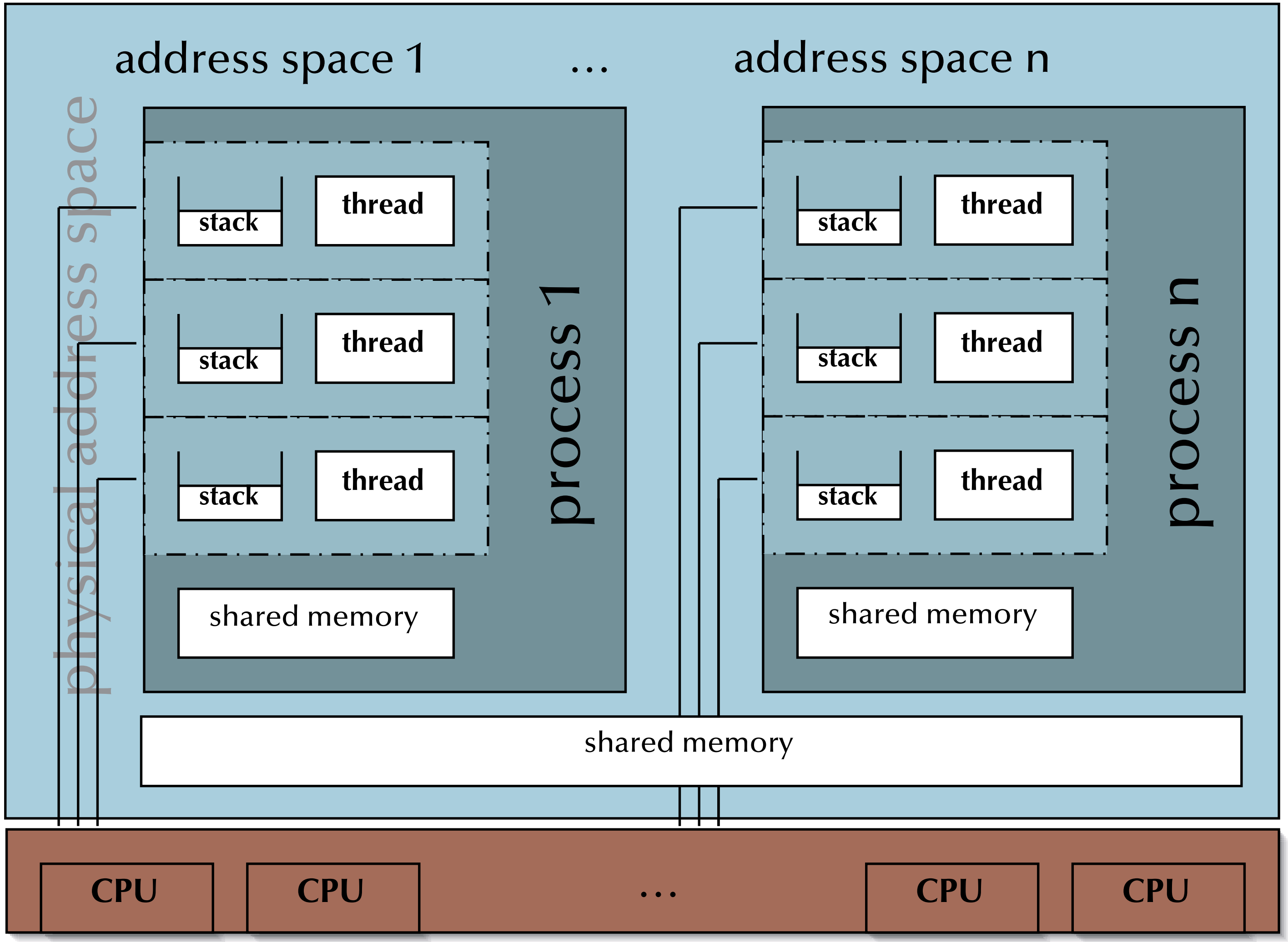


Introduction to processes and threads

Symmetric Multiprocessing (SMP)

- all CPUs share the same physical address space (and access to resources)

☞ processes/threads can be executed on any available CPU



Introduction to processes and threads

Processes ↔ Threads

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

- ☞ Threads can be regarded as a group of processes, which share some resources (☞ process-hierarchy).
- ☞ Due to the overlap in resources, the attributes attached to threads are less than for ‘first-class-citizen-processes’.
- ☞ Thread switching and inter-thread communications can be more efficient than 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.

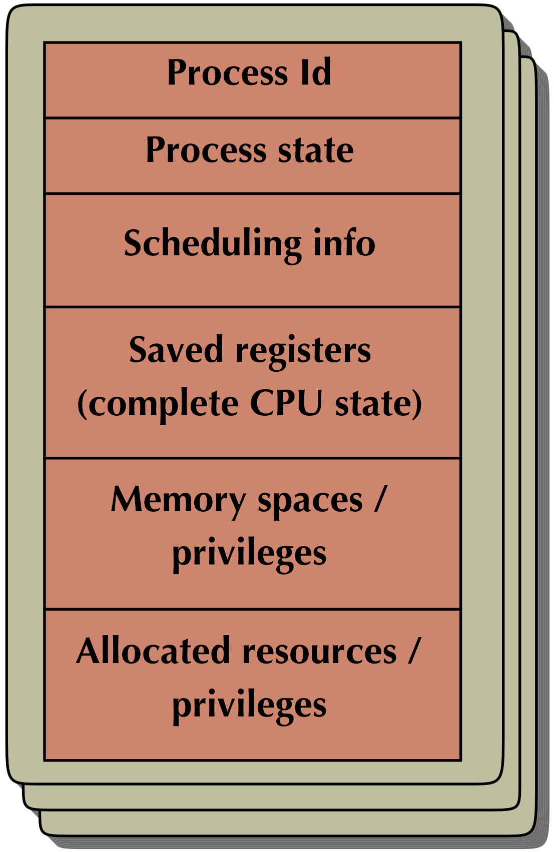
Introduction to processes and threads

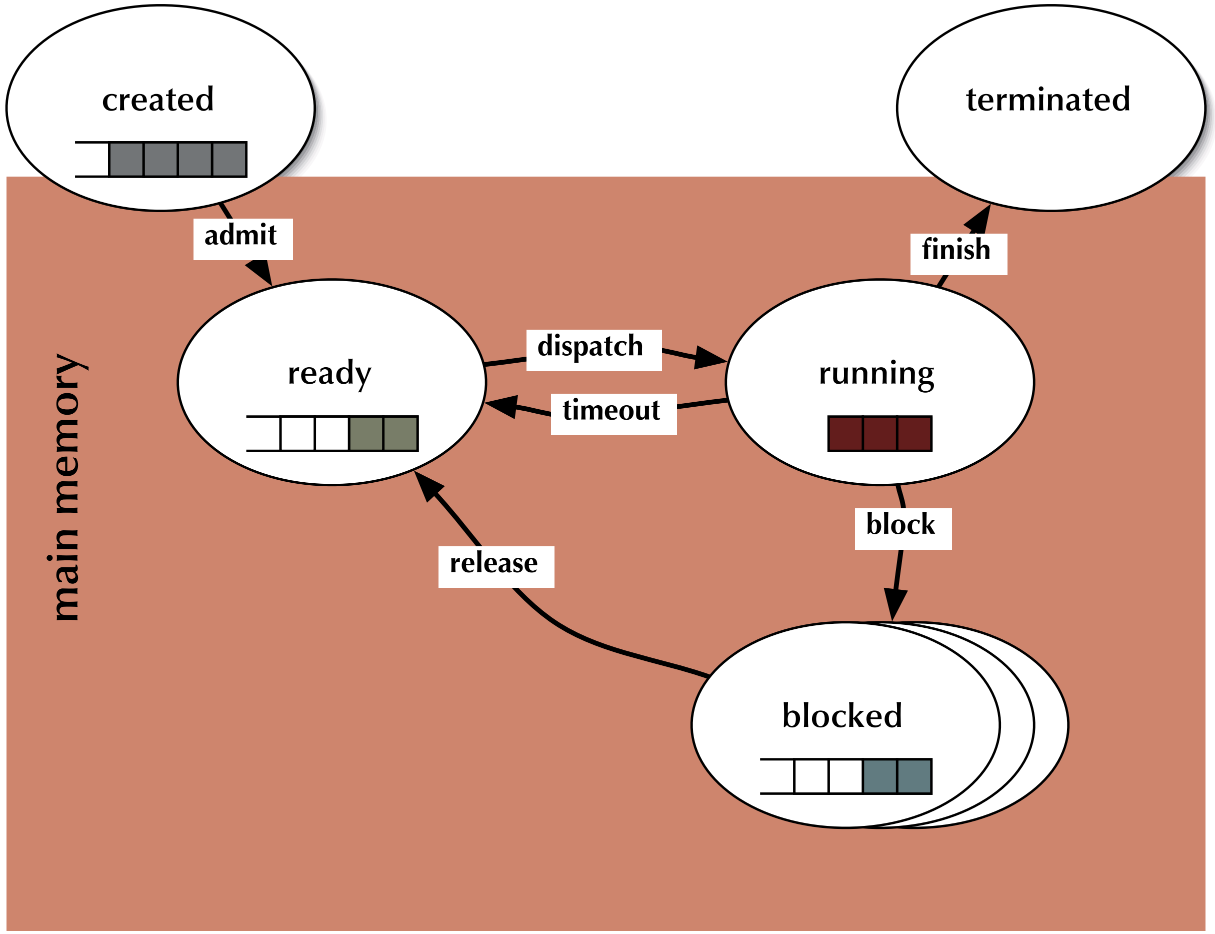
Process Control Blocks

- **Process Id**
- **Process state**:
  {created, ready, executing, blocked, suspended, …}
- **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)

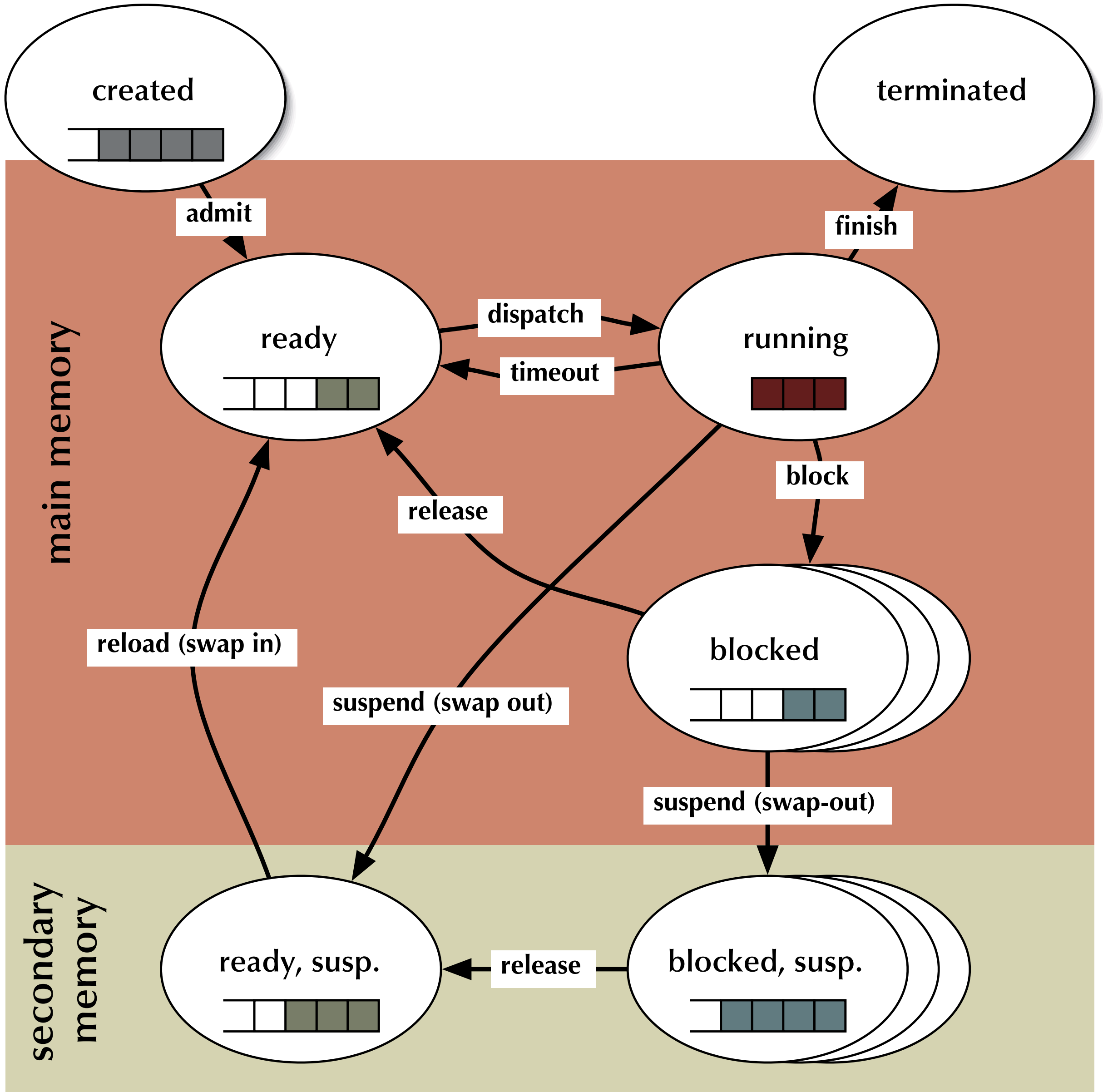
Process Control Blocks (PCBs)





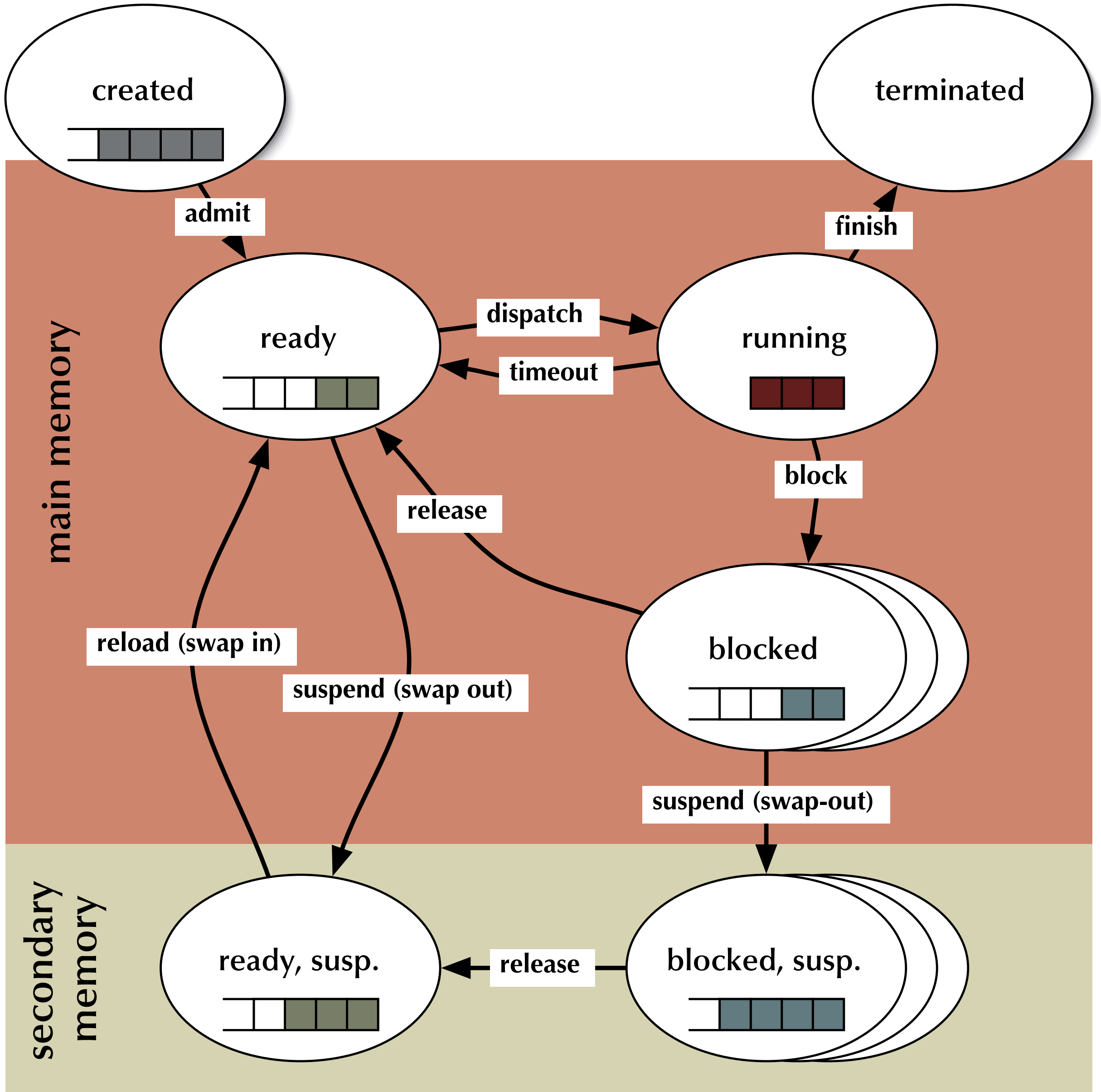
Process states

- **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



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)

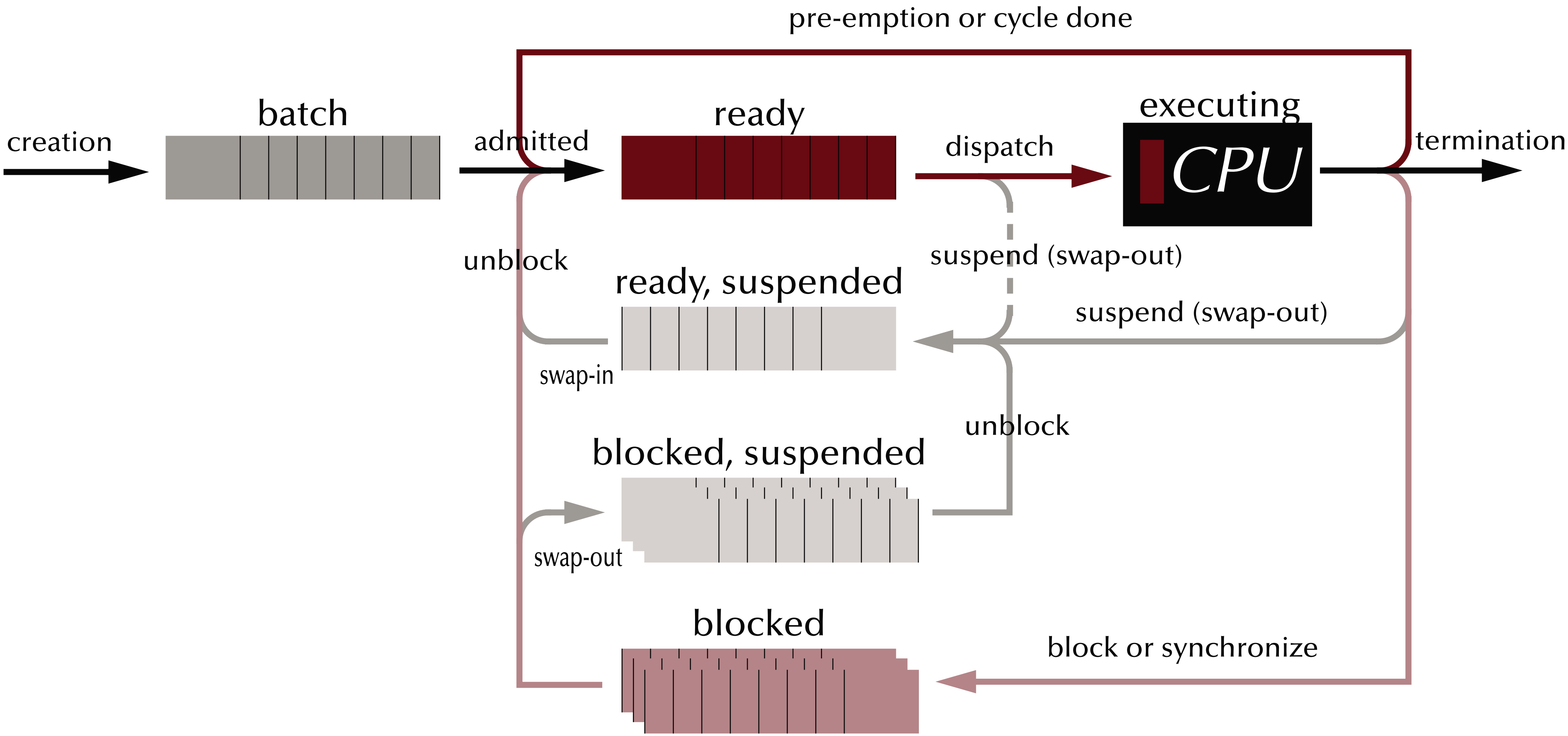


Process states

- **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

Process states



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 */
}
```

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 ();
}
```


Concurrent programming languages

Requirement

- Concept of **tasks**, **threads** or other **potentially concurrent entities**

Frequently requested essential elements

- Support for **management** or 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, …)

Language candidates

☞ Explicit concurrency

- Ada2005, Chill, Erlang
- Chapel, X10
- Occam, CSP
- Java, C#
- Modula-2, Modula-3
- …
- (Ruby, Stackless Python) [broken due to global interpreter locks]

☞ Implicit concurrency

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

☞ No support:

- Eiffel, Pascal
- C, C++
- Fortran, Cobol, Basic…

☞ Libraries & interfaces (outside language definitions)

- POSIX
- MPI (message passing interface)
- …

Languages explicitly supporting concurrency: e.g. Ada2005

Ada2005 is an ISO standardized (ISO/IEC 8652:1995/Amd 1:2007) 'general purpose' language which "promotes reliability and simplify maintenance" while keeping maximal efficiency and provides core language primitives for:

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

… as well as standardized language-annexes for

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

A protected generic queue *specification*

```
generic
   type Element is private;
package Queue_Pack_Protected_Generic is
   QueueSize : constant Integer := 10;
   type Queue_Type is limited private;
   protected type Protected_Queue is
      entry Enqueue (Item: in Element);
      entry Dequeue (Item: out Element);
   private
      Queue : Queue_Type;
  end Protected_Queue;
private
   type Marker is mod QueueSize;
   type List is array (Marker'Range) of Element;
   type Queue_State is (Empty, Filled);
   type Queue_Type is record
      Top, Free : Marker := Marker'First;
      State : Queue_State := Empty;
      Elements : List;
   end record;
end Queue_Pack_Protected_Generic;
```

A protected generic queue implementation

```
package body Queue_Pack_Protected_Generic is
   protected body Protected_Queue is
      entry Enqueue (Item: in Element) when
         Queue.State = Empty or Queue.Top /= Queue.Free is

      begin
         Queue.Elements (Queue.Free) := Item;
         Queue.Free := Queue.Free - 1;
         Queue.State := Filled;
      end Enqueue;

      entry Dequeue (Item: out Element) when
         Queue.State = Filled is

      begin
         Item := Queue.Elements (Queue.Top);
         Queue.Top := Queue.Top - 1;
         if Queue.Top = Queue.Free then
            Queue.State := Empty;
         end if;
      end Dequeue;

   end Protected_Queue;
end Queue_Pack_Protected_Generic;
```

A protected generic queue test task set

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

procedure Queue_Test_Protected_Generic is

   package Queue_Pack_Protected_Character is
      new Queue_Pack_Protected_Generic (Element => Character);
   use Queue_Pack_Protected_Character;

   Queue : Protected_Queue;

   task Producer is entry shutdown; end Producer;
   task Consumer is end Consumer;
(...)
```

… what's left to do: implement the tasks '`Producer`' and '`Consumer`'

A protected generic queue *test task set (producer)*

(...)

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

(...)

A protected generic queue test task set (consumer)

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

Concurrent programming languages

Language candidates

☞ Explicit concurrency

- Ada2005, Chill, Erlang
- Chapel, X10
- Occam, CSP
- Java, C#
- Modula-2, Modula-3
- …
- (Ruby, Stackless Python) [broken due to global interpreter locks]

☞ Implicit concurrency

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

☞ No support:

- Eiffel, Pascal
- C, C++
- Fortran, Cobol, Basic…

☞ Libraries & interfaces (outside language definitions)

- POSIX
- MPI (message passing interface)
- …

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]
```

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

☞ Parameters can be evaluated independently ☞ 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 the evaluation of g(n) and h(n) can be stopped (or not even be started)

☞ concurrent program parts need to be interruptible in this case

(Lazy) sub-expression evaluations in imperative languages still assume sequential execution:

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

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
- **First examples of concurrent programming languages:**
  - Explicit concurrency: e.g. Ada2005, Chapel, X10
  - Implicit concurrency: functional programming – e.g. Lisp, Haskell, Caml, Miranda

2

Mutual Exclusion

Uwe R. Zimmer - The Australian National University

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

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.

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: Atomic load & store operations

Atomic load & 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
begin
  G := 1
  G := G + G;
end P1;
```

```
task body P2 is
begin
  G := 2
  G := G + G;
end P2;
```

```
task body P3 is
begin
  G := 3
  G := G + G;
end P3;
```

☞ After the first global initialisation, `G` can have **many values** between `0` and 24

☞ After the first global initialisation, `G` will have **exactly one** value between `0` and 24

Mutual exclusion: first attempt

```
type Task_Token is mod 2;
Turn: Task_Token := 0;
```

```
task body P0 is

begin
  loop
    –– non_critical_section_0;
    loop exit when Turn = 0; end loop;
      –– critical_section_0;
    Turn := Turn + 1;
  end loop;
end P0;
```

```
task body P1 is

begin
  loop
    –– non_critical_section_1;
    loop exit when Turn = 1; end loop;
      –– critical_section_1;
    Turn := Turn + 1;
  end loop;
end P1;
```

☞ Mutual exclusion!

☞ No deadlock!

☞ No starvation!

☞ Locks up, if there is no contention!

Mutual exclusion: first attempt

```
type Task_Token is mod 2;
Turn: Task_Token := 0;

task body P0 is
begin
  loop
    -- non_critical_section_0;
    loop exit when Turn = 0; end loop;
      -- critical_section_0;
    Turn := Turn + 1;
  end loop;
end P0;
```

```
task body P1 is
begin
  loop
    -- non_critical_section_1;
    loop exit when Turn = 1; end loop;
      -- critical_section_1;
    Turn := Turn + 1;
  end loop;
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: 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!

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!

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!

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;
    end loop;
    -- critical section
    CSS (this_Task) := Out_CS;
    Turn := other_Task;
  end One_Of_Two_Tasks;
```

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;
                                                end loop;
                                                -- critical section
                                                CSS (this_Task) := Out_CS;
                                                Turn := other_Task;
                                              end One_Of_Two_Tasks;
```

☞ Mutual exclusion!
☞ No deadlock!
☞ No starvation!
☞ No livelock!

☞ Two tasks only!

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
  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;
```

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
    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;
```

- ☞ Mutual exclusion!
- ☞ No deadlock!
- ☞ No starvation!
- ☞ No livelock!

- ☞ Two tasks only!

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.

The idea of the Bakery Algorithm

A set of N Processes $P_1 \ldots P_N$ competing for mutually exclusive execution of their critical regions. Every process P_i out of $P_1 \ldots P_N$ 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: $\forall j \neq i : t_i < t_j$ or $t_j = 0$
- After a process left a critical section:
  - P_i resets its $t_i = 0$

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?

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_critcal_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;
     loop
      exit when
       Ticket (id) = 0
      or else
       Ticket (this_id) < Ticket (id)
      or else
       (Ticket (this_id) = Ticket (id)
        and then this_id < id);
     end loop;
    end if;
   end loop;
   -- critical_section_1;
   Ticket (this_id) := 0;
  end loop;
end P;
```

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_critcal_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;

     loop
      exit when
       Ticket (id) = 0
       or else
       Ticket (this_id) < Ticket (id)
       or else
       (Ticket (this_id) = Ticket (id)
        and then this_id < id);
     end loop;
    end if;
   end loop;
   –– critical_section_1;
   Ticket (this_id) := 0;
  end loop;
 end P;
```

- ☞ Mutual exclusion!
- ☞ No deadlock!
- ☞ No starvation!
- ☞ No livelock!
- ☞ Works for N processes!

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

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** $:\overset{R}{=}$ **C**; – read by using a *special instruction*, which puts a ‘reservation’ on C
- … calculate a <new value> for C …
- **C** $:\overset{T}{=}$ **<new value>;**
  – succeeds iff C was not manipulated by other processors or devices since the reservation

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
  -- non_critical_section_i;
  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
  -- non_critical_section_j;
  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!

Mutual exclusion: atomic exchange operation

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

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

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

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

Mutual exclusion: memory cell reservation

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

```
task body Pi is
L : Flag;
begin
 loop
  -- non_critical_section_i;
  loop
   L :R= C; C :T= 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
  -- non_critical_section_j;
  loop
   L :R= C; C :T= 1;
   exit when Untouched and 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!

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 'passeren' (Dutch for 'pass'):

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

- an atomic operation **V** on S — for 'vrygeven' (Dutch for 'to release'):

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

☞ *then* the variable **S** is called a **Semaphore**.

Beyond atomic hardware operations

Semaphores

… as supplied by operating systems and runtime environments

- a set of processes $P_1 \ldots P_N$ 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 Pi on S]
```

- an atomic operation **Signal** on S: (aka 'Set_True', 'sem_post', …)

  Process P_i : **Signal** (S):

```
[if ∃Pj suspended on S then release Pj
                       else S := S + 1]
```

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

Beyond atomic hardware operations

Semaphores

Types of semaphores:

- **Binary semaphores**: restricted to [0, 1];
  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 with a non-negative number: often the number of processes which are allowed inside a critical section, i.e. '1'.

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 Pi;
```

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

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 Pi;
```

☞ Mutual exclusion!, No global live-lock!

☞ Works for any dynamic number of processes

☞ Individual starvation possible! Deadlock possible!

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

Concurrent & Distributed Systems 2010

3

Synchronization

Uwe R. Zimmer - The Australian National University

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

[NN2006]
Ada Reference Manual - Language and Standard Libraries
Ada Reference Manual ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1 2006

[NN2009]
Chapel Language Specification 0.782
2009

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

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 ☞ Ada2005
- Atomic blocks ☞ Chapel, X10

Message based synchronization

- Asynchronous messages ☞ e.g. POSIX, …
- Synchronous messages ☞ e.g. Ada2005, CHILL, Occam2, …
- Remote invocation, remote procedure call ☞ e.g. Ada2005, …

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.

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}
```

Are those operations atomic?

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}
```

Depending on the hardware and the compiler, it might be atomic, it might be not:

☞ Handling a 64-bit integer on a 8- or 16-bit controller will not be atomic

… 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.

Assuming that all 'perhapses' apply: how to expand this code?

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: synchronization by employing knowledge about the atomicity of CPU-operations and interrupt structures is nevertheless possible and done frequently.

In anything higher than assembler level on small, predictable μ-controllers:

☞ *Measures for synchronization are required!*

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    |    x := 500
```

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

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:

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

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!

☞ More powerful synchronization operations are required for critical sections

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 'passeren' (Dutch for 'pass'):

  P(S): `[as soon as S > 0 then S := S - 1]` ☞ 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', …

☞ *then* the variable **S** is called a **Semaphore**.

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

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:

$$A \rightharpoonup B \rightharpoonup C; X \rightharpoonup Y \rightharpoonup Z; [X,Z \mid A,B,C]; [A,C \mid X,Y,Z]; \neg[B \mid Y]$$

Towards synchronization

Semaphores in Ada2005

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

only one task can be blocked at `Suspend_Until_True`!
(`Program_Error` will be raised with a second task trying to suspend itself)

☞ no queues! ☞ minimal run-time overhead

Towards synchronization

Semaphores in Ada2005

```
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;
```

☞ for special cases only … otherwise:

medieval

only one task can be blocked at Suspend_Until_True!
(Program_Error will be raised with a second task trying to suspend itself)

☞ no queues! ☞ minimal run-time overhead

Towards synchronization

Malicious use of 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;
```

☞ 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)

Towards synchronization

Malicious use of 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)

Towards synchronization

Malicious use of 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 Ada2005.

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

Towards synchronization

Semaphores in POSIX

```
sem_t mutex, cond[2];
typedef emun {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?

Towards synchronization

Semaphores in Java

```
Semaphore (int permits, boolean fair)
```

"wait":

```
          void    acquire                 ()
          void    acquire                 (int permits)
          void    acquireUninterruptibly  (int permits)
          boolean tryAcquire              ()
          boolean tryAcquire              (int permits, long timeout, TimeUnit unit)
```

'mess with it':

```
          int     availablePermits        ()
protected void    reducePermits           (int reduction)
          int     drainPermits            ()
```

"signal":

```
          void    release                 ()
          void    release                 (int permits)
```

administration:

```
protected Collection <Thread> getQueuedThreads ()
          int     getQueueLength          ()
          boolean hasQueuedThreads        ()
          boolean isFair                  ()
          String  toString                ()
```


Towards synchronization

Review of semaphores

- Semaphores are *not bound* to any resource or method or region
  - ☞ Semaphore operations are programmed as individual operations.
  - ☞ Adding or deleting a single semaphore operation might stall the whole system.
- Semaphores are *scattered* all over the code
  - ☞ Hard to read code, highly error-prone.

☞ Semaphores are generally considered inadequate for non-trivial systems.

(all concurrent languages and environments offer efficient higher-level synchronization methods)

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

Distributed synchronization

Conditional Critical Regions

```
buffer : buffer_t;
resource critial_buffer_region : buffer;
```

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

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

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 uses conditional critical regions for synchronization in a multi-processor environment (each process is associated with exactly one processor).)

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.

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 realize conditional 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.

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; …
```

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!

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, Java:
  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.

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`.

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.
```

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);
…
```

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
- …

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

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);
```

pthread_cond_signal: unblocks *'at least one'* thread

pthread_cond_broadcast: unblocks *all* threads

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

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

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;
}
```

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;
```

> *need* to be called with a locked mutex

```
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;
}
```

> better to be called *after* unlocking all mutexes (as it is itself potentially blocking)

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);
    }
  }
```

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);
    }
  };
```

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
```

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 to the monitor by convention only

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.

Centralized synchronization

Monitors in Java

(by means of language primitives)

Considerations:

1. Synchronized methods and code blocks:

- In order to implement a monitor *all* methods in an object need to be synchronized.
  - ☞ any other standard method can break 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> {…}

Centralized synchronization

Monitors in Java

(by means of language primitives)

Considerations:

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

- `wait` suspends the thread and releases the local lock only
  - ☞ nested `wait`-calls will keep all enclosing locks.
- `notify` and `notifyAll` 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

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.

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 ();
...
```


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 ();
    }
  } …
```


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;
               }
   } } } } …
```

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 ();
     }
  } …
```

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 ();
            }
        }
    }
  }
}
```

Centralized synchronization

Object-orientation and synchronization

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

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

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

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

☞ **Inheritance anomaly** (Matsuoka & Yonezawa '93)

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

Centralized synchronization

Monitors in Java

Per Brinch Hansen 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.

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** instead of 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 monitors

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. Ada95

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.

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)
- **entry guards** are *attached* to operations (no condition variables inside operations)
- *no* protected data is accessible (other than by the defined operations)
- processes are **queued** (according to their priorities)
- processes can be **requeued** to other guards

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 : in Data_Item);

private
   The_Data : Data_Item := Initial;
end Shared_Data_Item;
```

- **protected functions** can have '`in`' parameters only and are not allowed to alter the private data (enforced by the compiler).

☞ **protected functions** allow *simultaneous access* (but mutual exclusive with other operations).

… there is no defined priority between functions and other protected operations in Ada95.

Synchronization by protected objects

(Condition synchronization: entries & barriers)

Condition synchronization is realized in the form of **protected procedures** combined with boolean conditional variables (**barriers**): ☞ **entries** in Ada2005:

```
Buffer_Size : constant Integer := 10;

type    Index    is mod Buffer_Size;
subtype Count    is Natural range 0 .. Buffer_Size;
type    Buffer_T is array (Index) of Data_Item;

protected type Bounded_Buffer is

   entry Get (Item : out Data_Item);
   entry Put (Item : in Data_Item);

private
   First : Index := Index'First;
   Last : Index := Index'Last;
   Num : Count := 0;
   Buffer : Buffer_T;
end Bounded_Buffer;
```

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 : in Data_Item) when Num < Buffer_Size is
      begin
         Last           := Last + 1;
         Buffer (Last) := Item;
         Num            := Num + 1;
      end Put;
end Bounded_Buffer;
```

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
   delay 10.0;
then abort
   Buffer.Put (Some_Data);
   - try to enter for 10 s.
end select;

select
   Buffer.Get (Some_Data);
then abort
   - meanwhile try something else
end select;
```

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*.

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;
```

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: in  Message);
private
   New_Message : Message;
   Arrived     : Boolean := False;
end Broadcast;
```

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

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.

Centralized synchronization

Synchronization by protected objects

(Entry families)

```
package Modes is

  type Mode_T is
    (Takeoff, Ascent, Cruising,
     Descent, Landing);

  protected Mode_Gate is
   procedure Set_Mode (Mode: in 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: in Mode_T) is

  begin
    Current_Mode := Mode;
  end Set_Mode;

  entry Wait_For_Mode
      (for Mode in Mode_T)
      when Current_Mode = Mode is

    begin null;
    end Wait_For_Mode;

  end Mode_Gate;

end Modes;
```

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 : in Urgency);
private
   entry Server (Server_Farm) (U : in Urgency);
end Pre_Filter;
```

Centralized synchronization

Synchronization by protected objects

(Entry families, requeue & private entries)

```
protected body Pre_Filter is

   entry Reception (U : in 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 : in Urgency) when true is

   begin null; - might try something even more useful
   end Server;

end Pre_Filter;
```

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

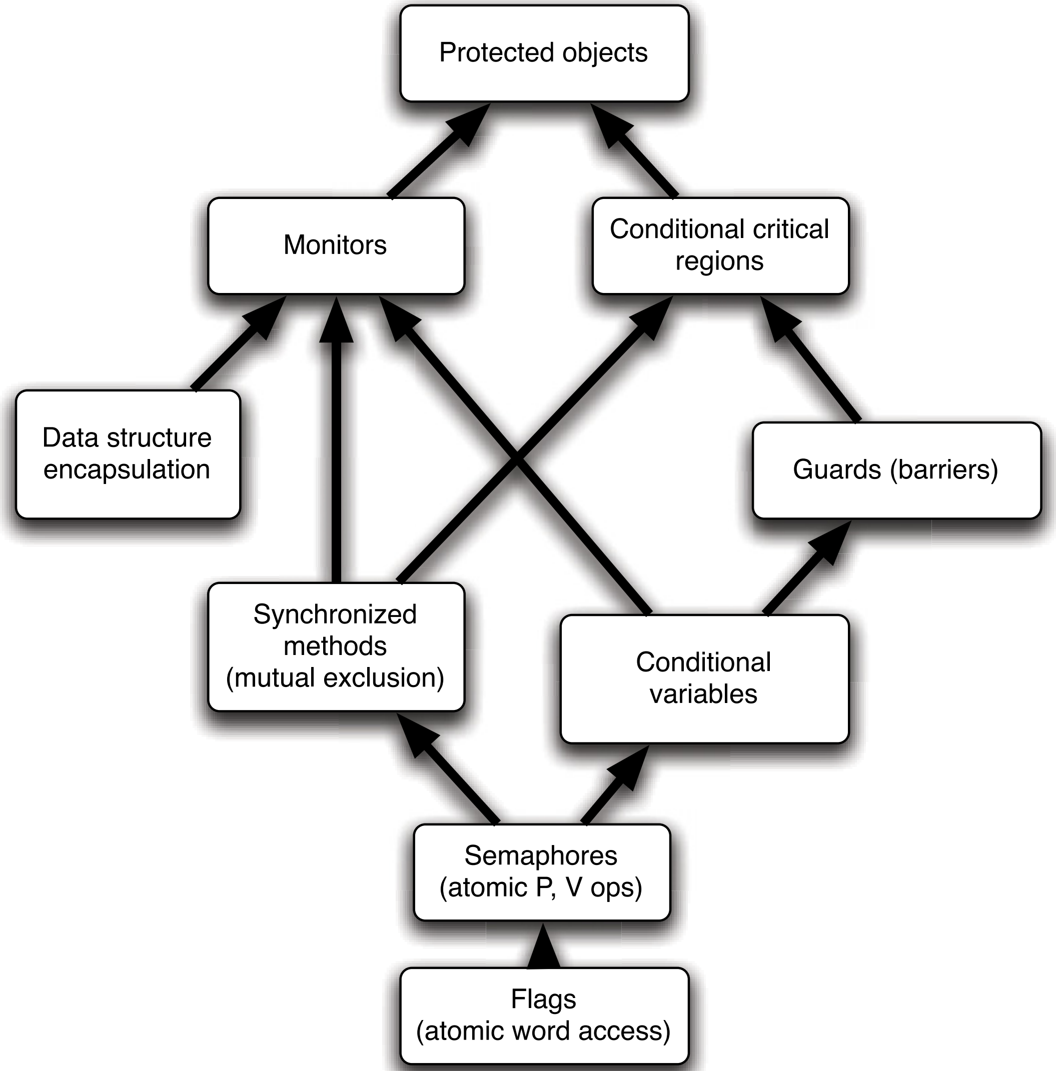
☞ The `requeue` facility allows for a potentially blocking operation, *and* releases the current lock!

Shared memory based synchronization

General

Criteria:

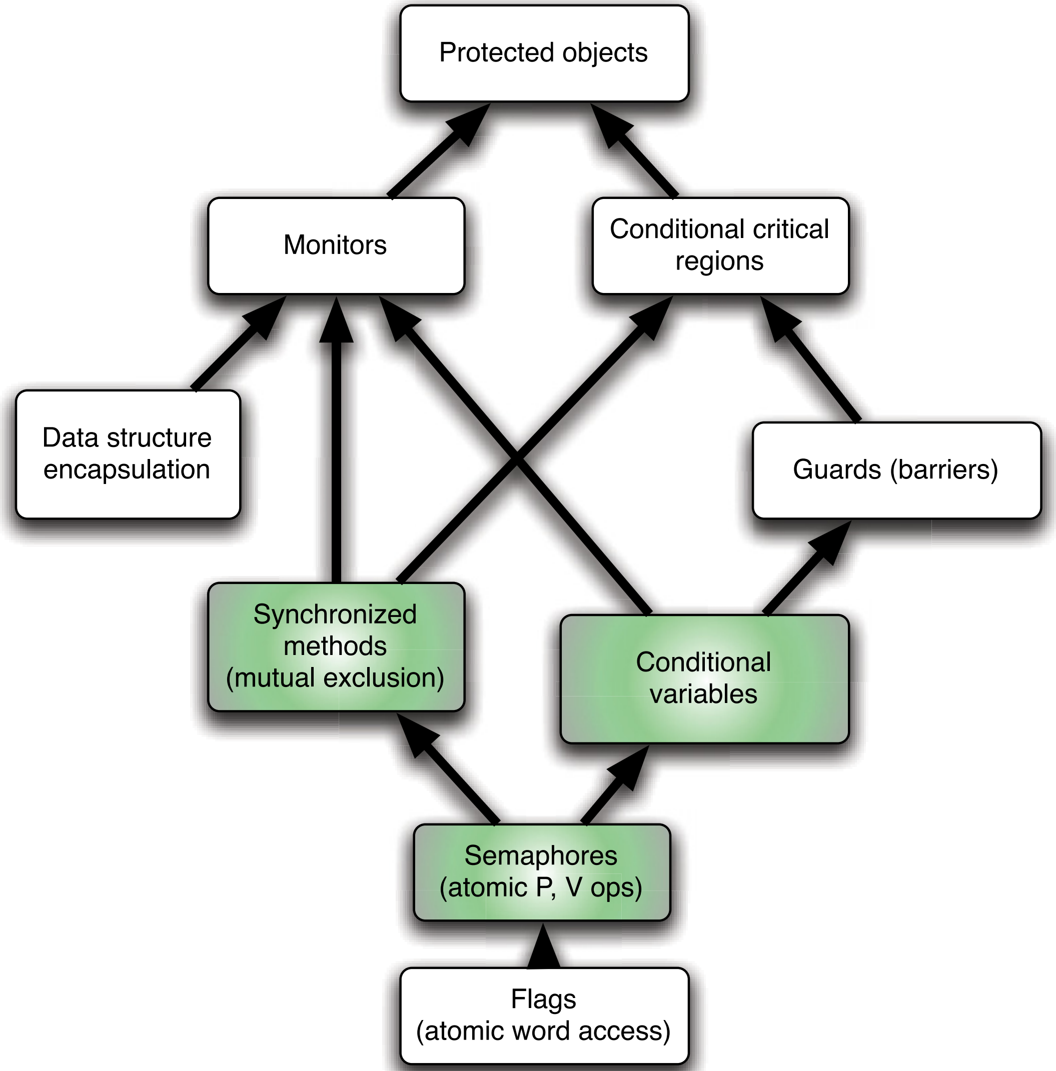
- *Levels of abstraction*
- *Centralized versus distributed*
- *Support for automated (compiler based) consistency and correctness validation*
- *Error sensitivity*
- *Predictability*
- *Efficiency*



Shared memory based synchronization

POSIX

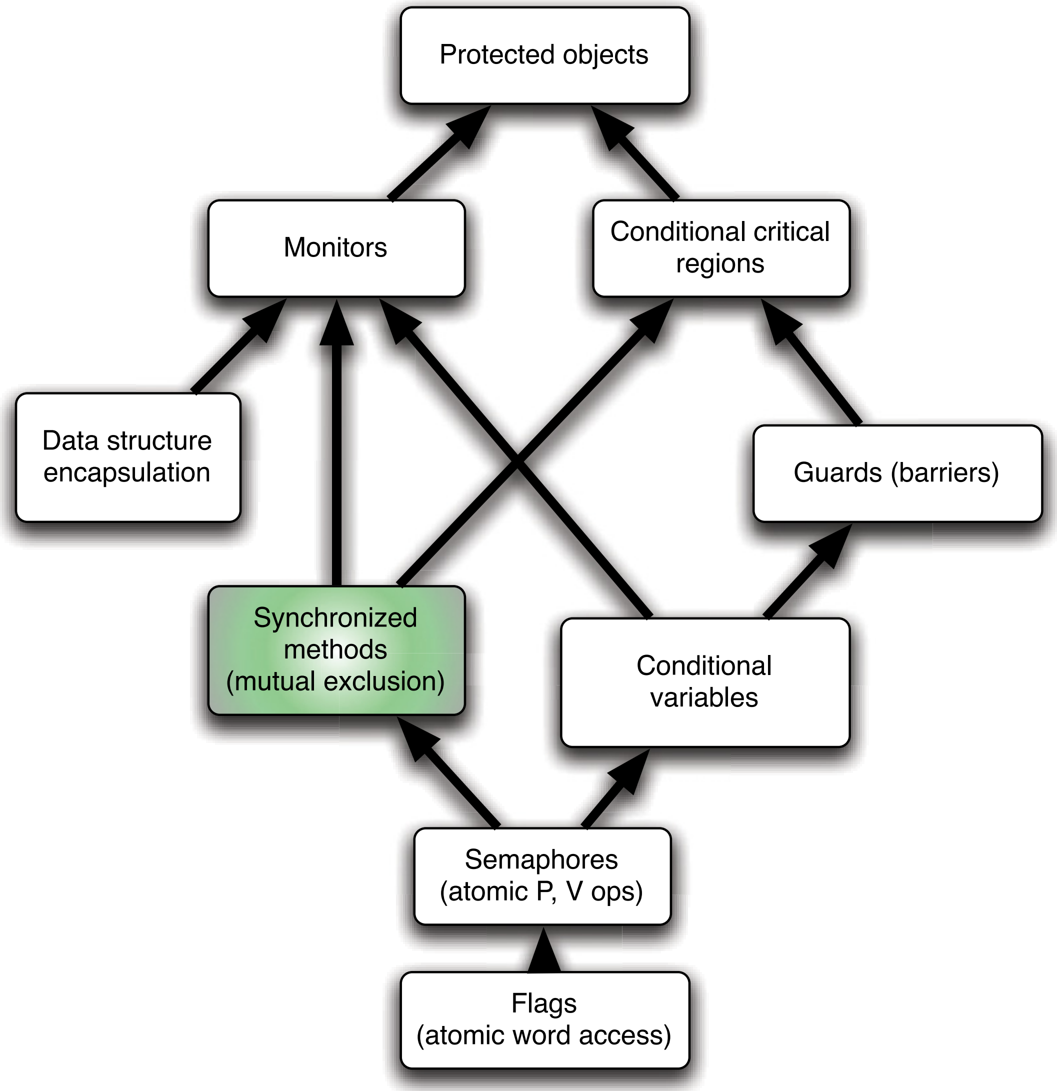
- All low level constructs available
- Connection with the actual data-structures by means of convention only
- Extremely error-prone
- Degree of non-determinism introduced by the 'release some' semantic
- 'C' based
- Portable



Shared memory based synchronization

Java

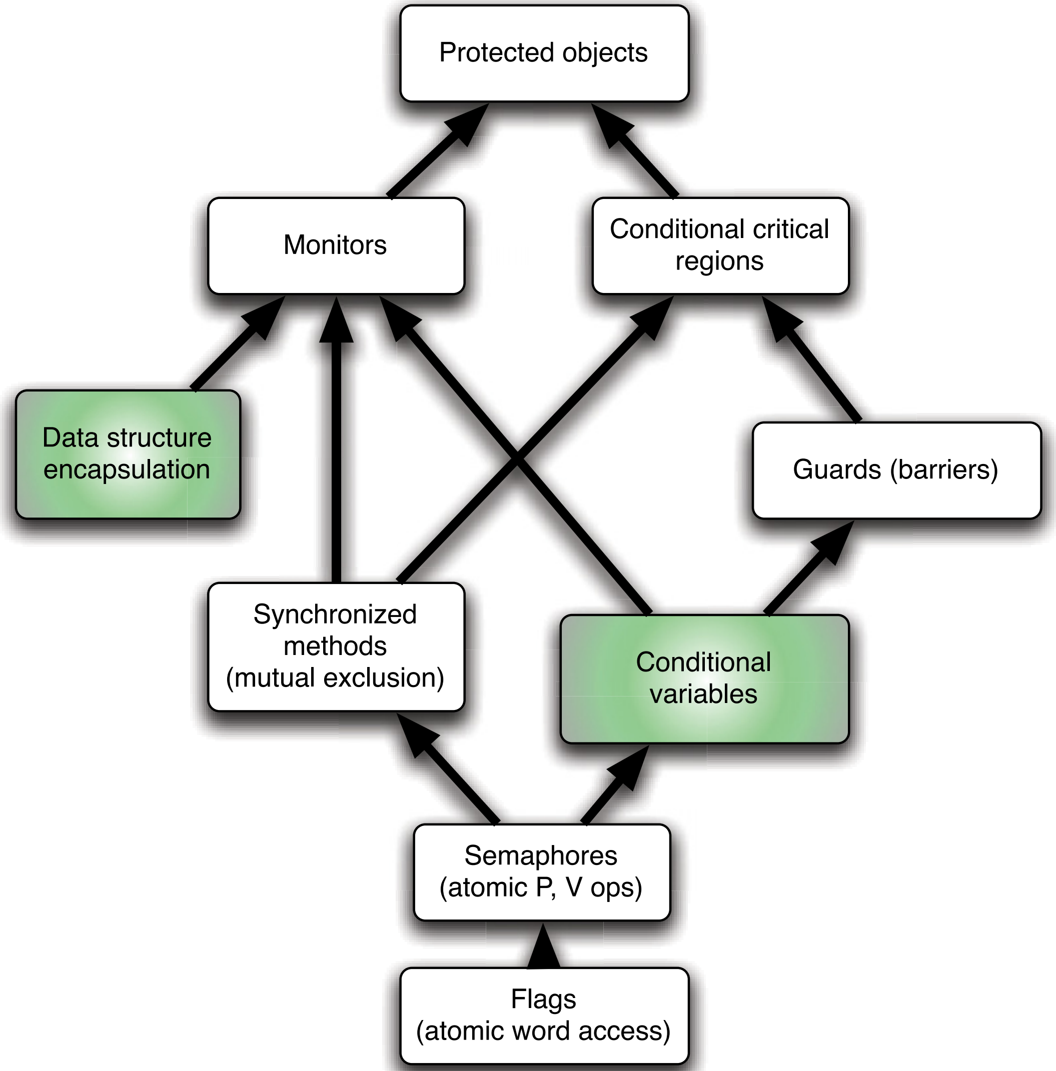
- Mutual exclusion available.
- General notification feature (not connected to other locks, hence not a conditional variable)
- Universal object orientation makes local analysis hard or even impossible
- Mixture of high-level object oriented features and low level concurrency primitives



Shared memory based synchronization

C#, Visual C++, Visual Basic

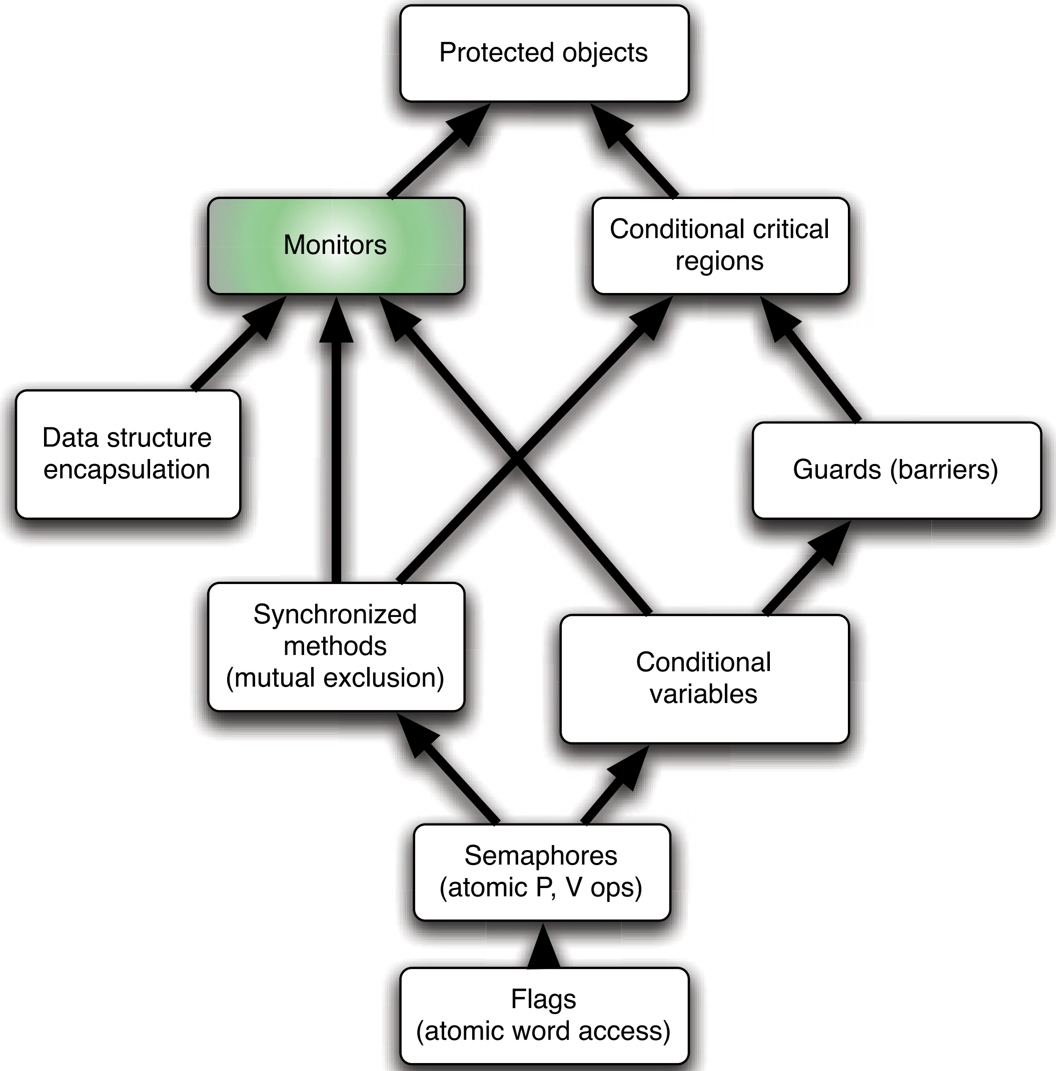
- *Mutual exclusion via library calls (convention)*
- *Data is associated with the locks to protect it*
- *Condition variables related to the data protection locks*
- *Mixture of high-level object oriented features and low level concurrency primitives*



Shared memory based synchronization

Modula-1, Chill, Parallel Pascal, …

- *Full implementation of the Dijkstra / Hoare monitor concept*

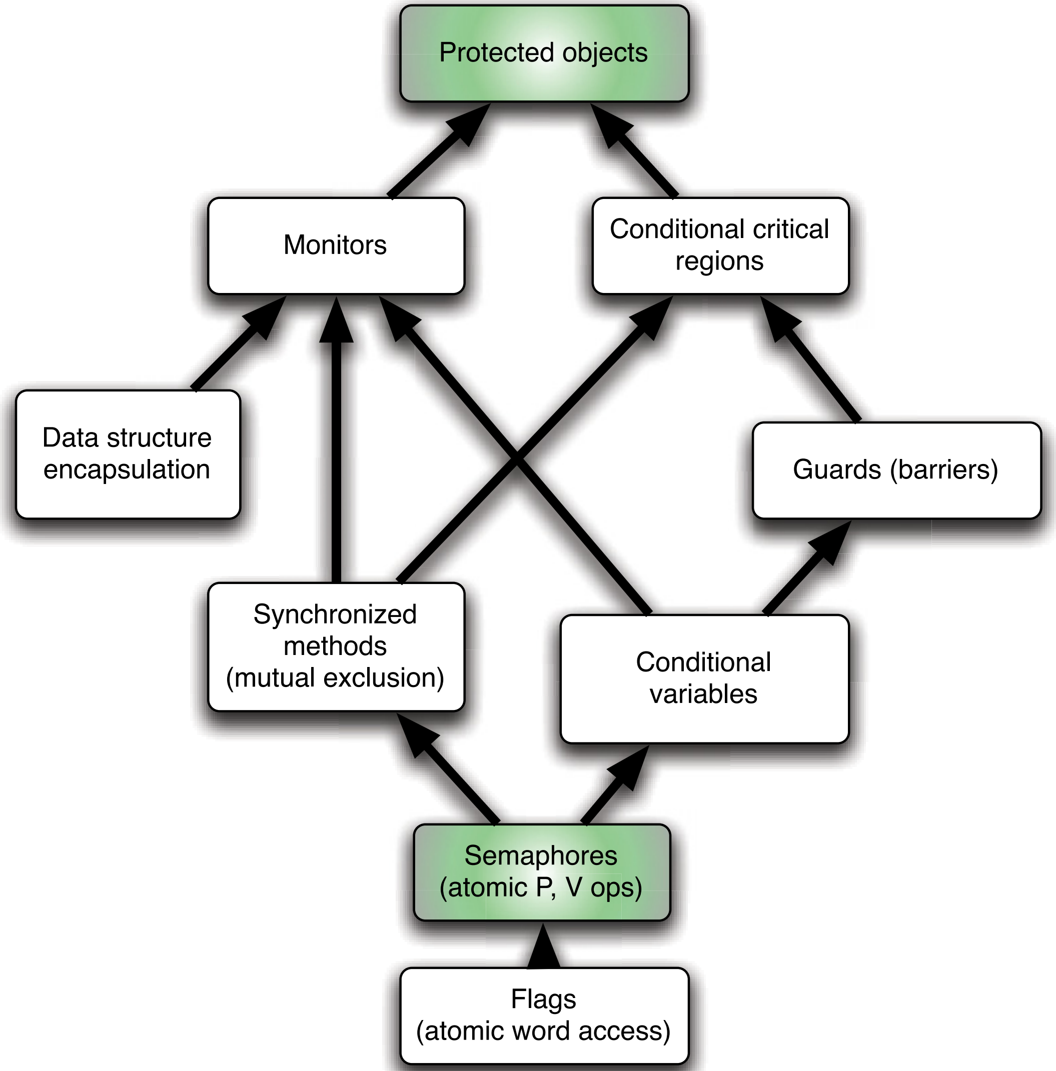


Shared memory based synchronization

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*

Ada2005 has still
no mainstream competitor in this field



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", i.e. there is 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. ☞ X10, Chapel, Fortress
- ☞ Not all algorithms break down into independent computation slices and so there is a need for memory integrity mechanisms in shared/partitioned address spaces.

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 this means that they cannot be nested, or 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 currently 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.

X10 is currently still under development and the atomic block semantic is likely to be amended while the current semantic is implemented in placeholder form only.

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

Most of the Chapel semantic is still forthcoming … so there is still hope for a stronger shared memory synchronization / memory integrity construct.

Message-based synchronization

Synchronization model

- Asynchronous
- Synchronous
- Remote invocation

Addressing (name space)

- direct communication
- mail-box communication

Message structure

- arbitrary
- restricted to 'basic' types
- restricted to un-typed communications

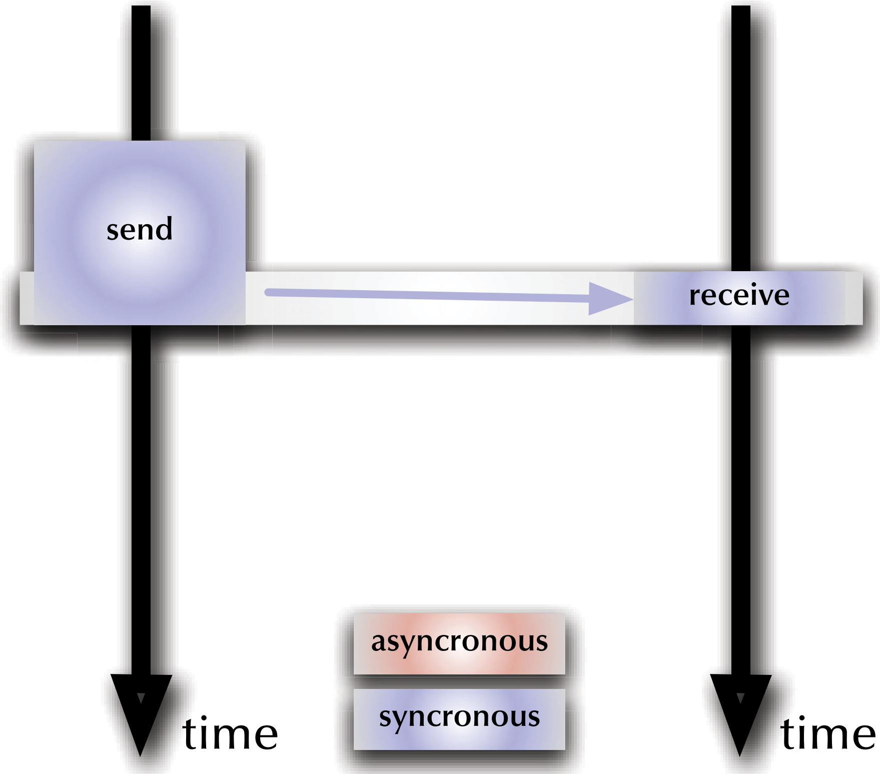
Message-based synchronization

Message protocols

Synchronous message
(sender waiting)

Delay the sender process until

- Receiver becomes available
- Receiver acknowledges reception



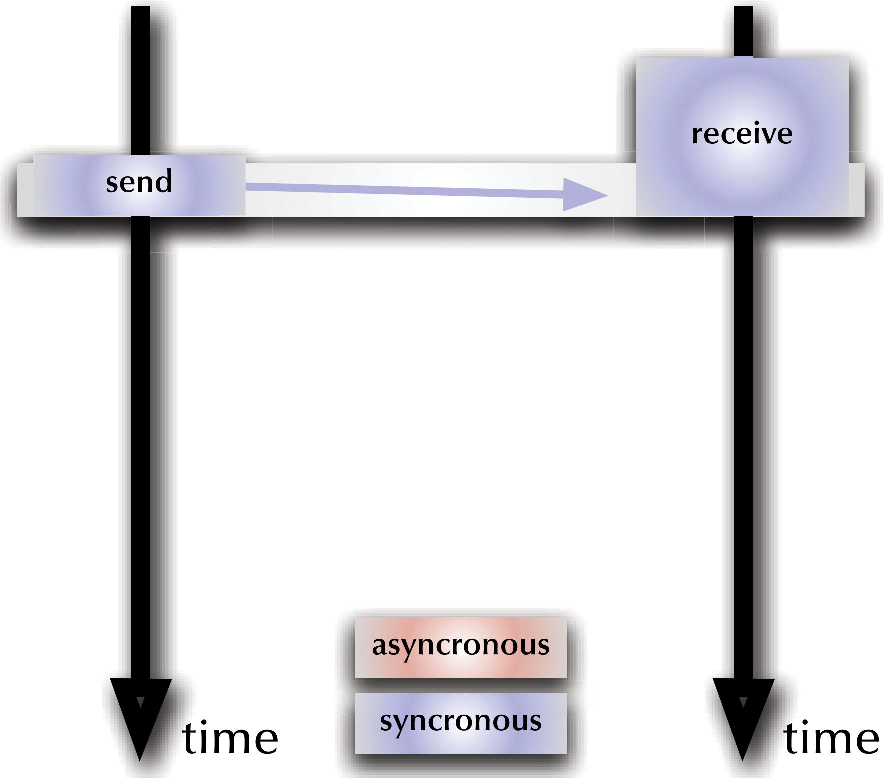
Message-based synchronization

Message protocols

Synchronous message
(receiver waiting)

Delay the receiver process until

- Sender becomes available
- Sender concludes transmission



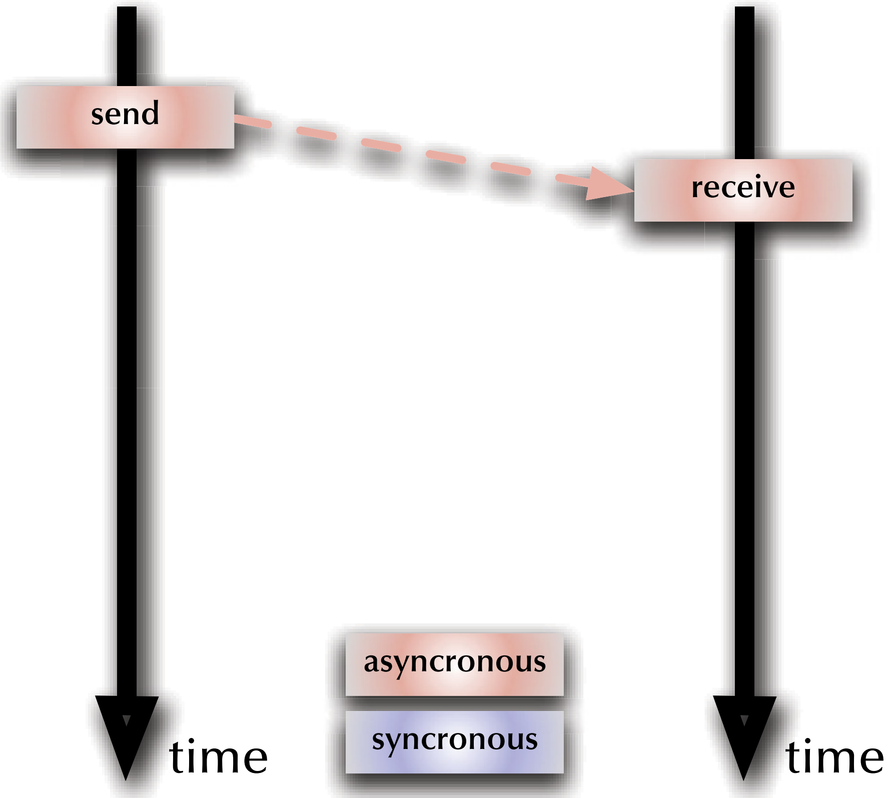
Message-based synchronization

Message protocols

Asynchronous message

Neither the sender nor the receiver is blocked:

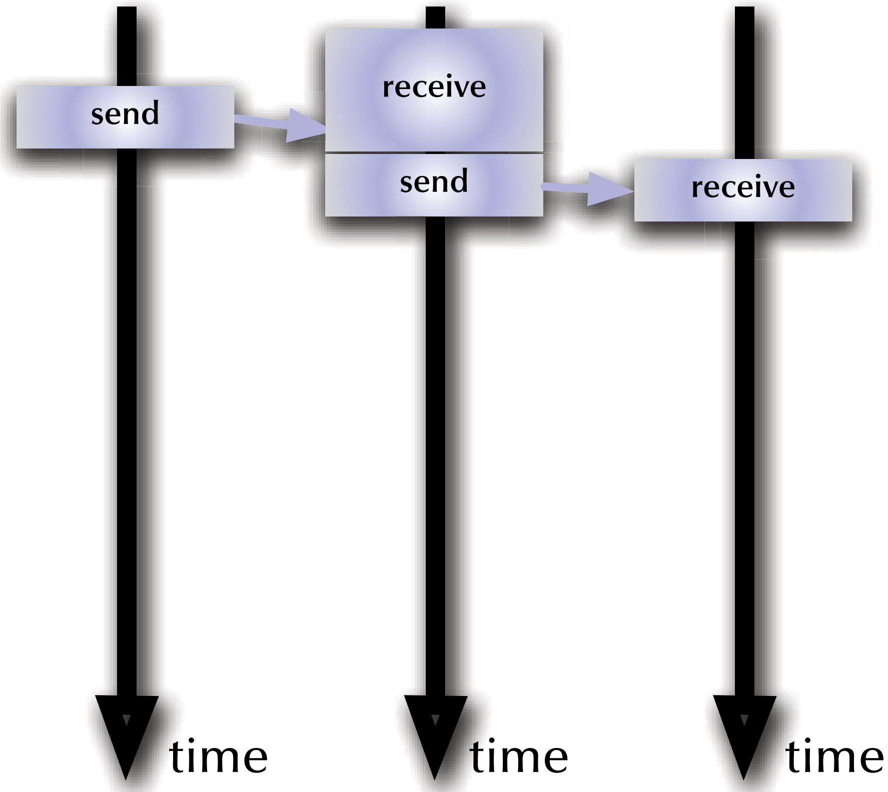
- Message is not transferred directly
- A buffer is required to store the messages
- Policy required for buffer sizes and buffer overflow situations



Message-based synchronization

Message protocols

Asynchronous message (simulated by synchronous messages)



Introducing an intermediate process:

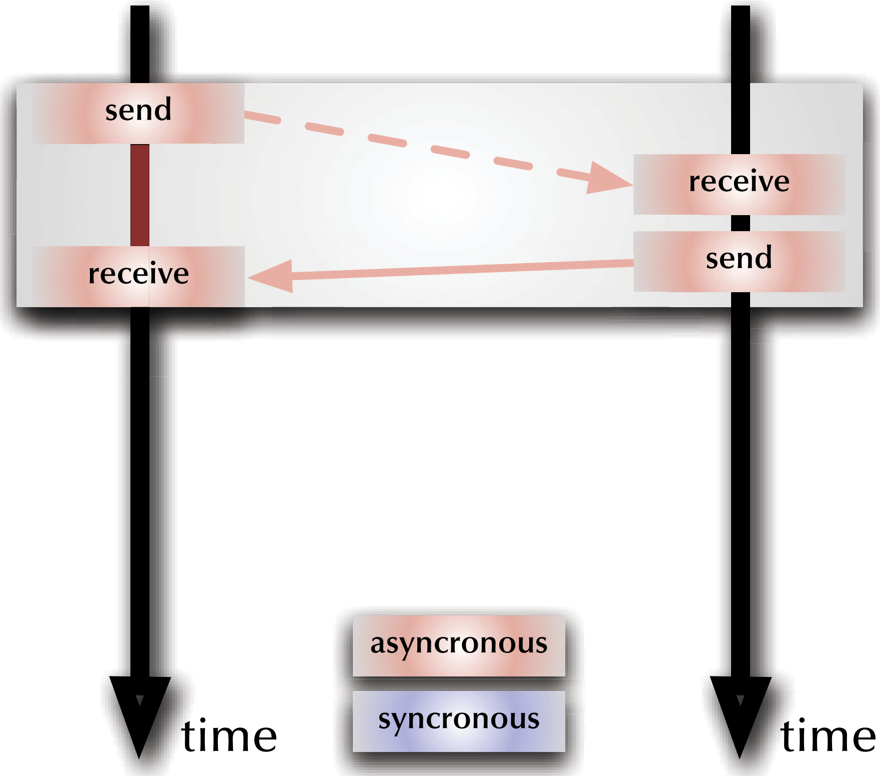
- Intermediate needs to be accepting messages at all times.
- Intermediate also needs to send out messages on request.

☞ While processes are blocked in the sense of synchronous message passing, they are not actually delayed as the intermediate is always ready.

Message-based synchronization

Message protocols

Synchronous message
(simulated by asynchronous messages)



Introducing two asynchronous messages:

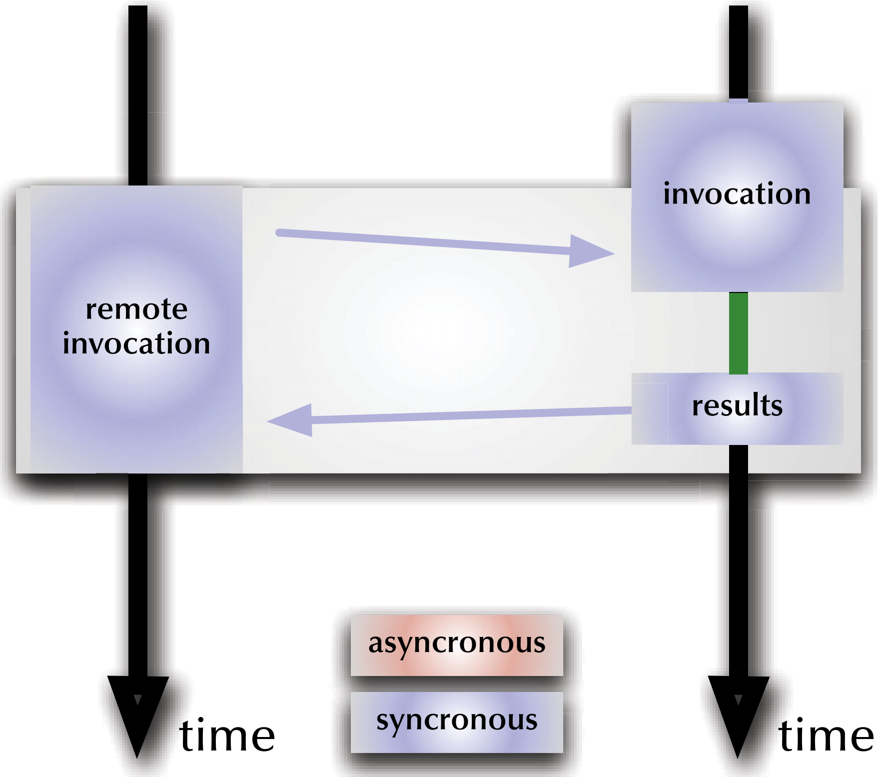
- Both processes voluntarily suspend themselves until the transaction is complete.
- As no immediate communication takes place, the processes are never actually synchronized.
- The sender (but not the receiver) process knows that the transaction is complete.

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

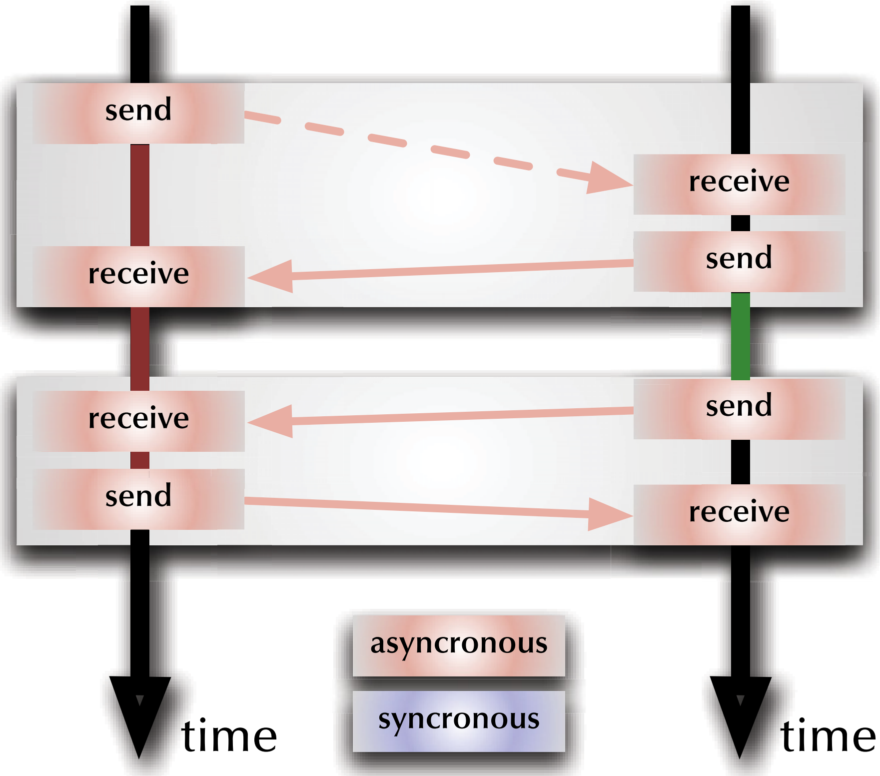


Message-based synchronization

Message protocols

Remote invocation (simulated by asynchronous messages)

- Simulate two synchronous messages
- Processes are never actually synchronized



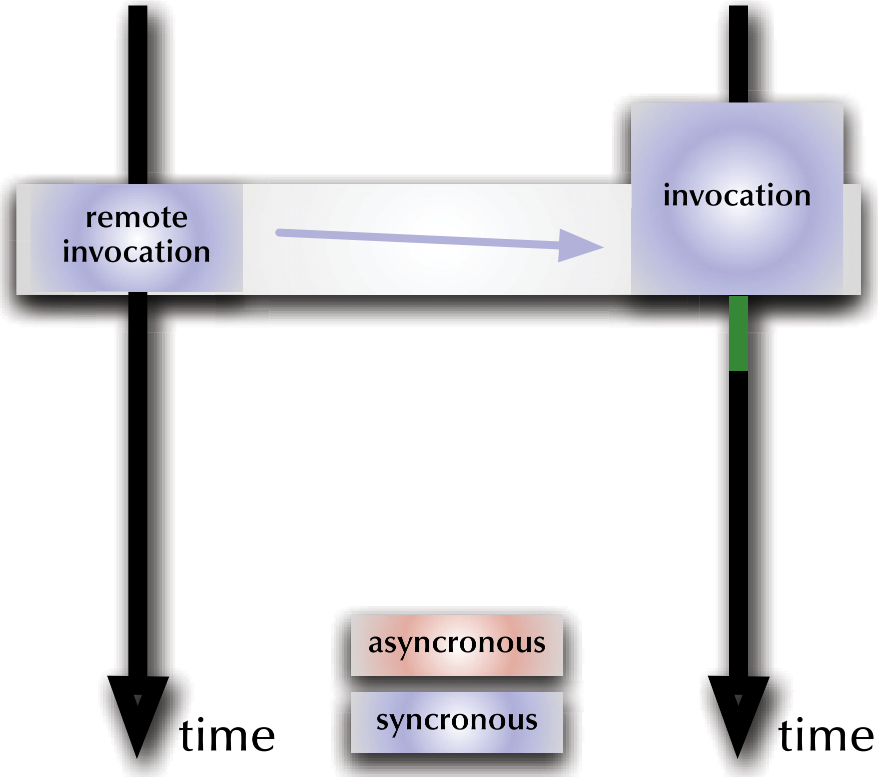
Message-based synchronization

Message protocols

Remote invocation (no results)
(simulated by asynchronous messages)

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.

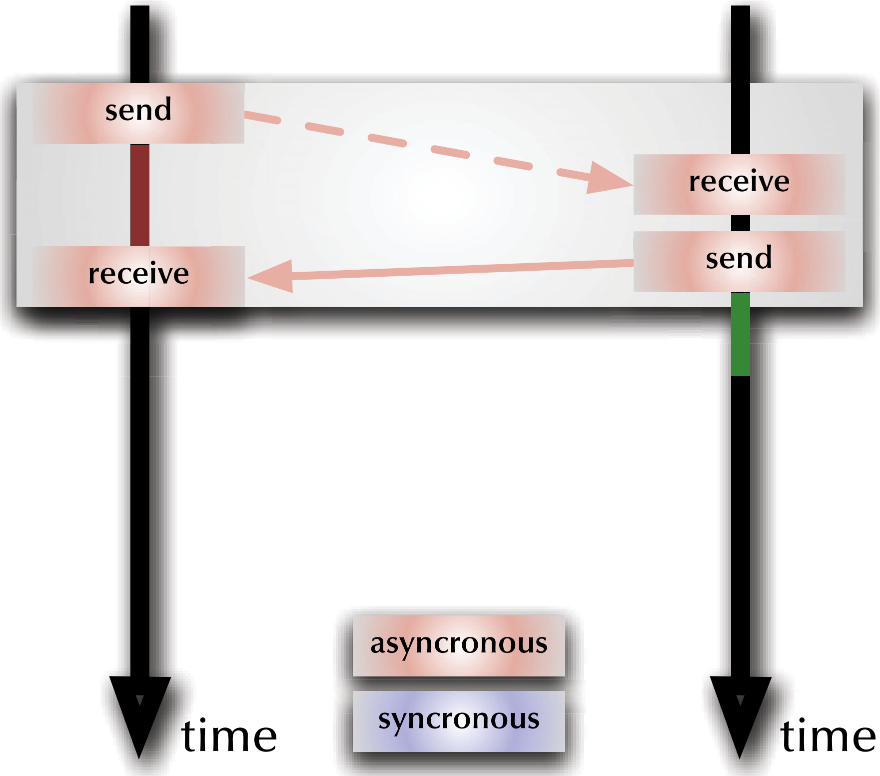


Message-based synchronization

Message protocols

Remote invocation (no results)
(simulated by asynchronous messages)

- Simulate one synchronous messages
- Processes are never actually synchronized



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 hard-ware 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).

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

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

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 then the basic element type are supplied …

… manually (POSIX, ‘C/C++’, Java)

… semi-automatic (CORBA)

… automatic (compiler-generated) and typed-persistent (Ada, CHILL, Occam2)

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;
```


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;
```

Message-based synchronization

Message-passing systems examples:

POSIX:	"**message queues**": ☞ ordered indirect [asymmetrical | symmetrical] asynchronous byte-level many-to-many message passing
MPI:	"**message passing**": ☞ ordered [direct | indirect] [asymmetrical | symmetrical] asynchronous memory-block-level [one-to-one | one-to-many | many-to-one | many-to-many] message passing
CHILL:	"**buffers**", "**signals**": ☞ ordered indirect [asymmetrical | symmetrical] [synchronous | asynchronous] typed [many-to-many | many-to-one] message passing
Occam2:	"**channels**": ☞ indirect symmetrical synchronous fully-typed one-to-one message passing
Ada:	"**(extended) rendezvous**": ☞ ordered direct asymmetrical [synchronous | asynchronous] fully-typed many-to-one remote invocation
Java:	☞ no message passing system defined

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:	✔	✔	✔		✔		✔	byte-stream			✔	message queues
MPI:	✔	✔	✔	✔	✔	✔	✔	memory-blocks	✔	✔	✔	message passing
CHILL:	✔	✔	✔	✔	✔		✔	typed		✔	✔	message passing
Occam2:		✔		✔			✔	fully typed	✔			message passing
Ada:	✔		✔	✔	✔	✔		fully typed		✔		remote invocation

Java: ☞ no message passing system defined

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

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);                receive case
                                              (SensorBuffer in data) : …
                                            esac;
```

asynchronous

```
signal SensorChannel = (int) to consumertype;
…
send SensorChannel (reading)                receive case
   to consumer                                (SensorChannel in data): …
                                            esac;
```

synchronous

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**’)

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

Message-based synchronization

Message-based synchronization in Ada

(Extended rendezvous)

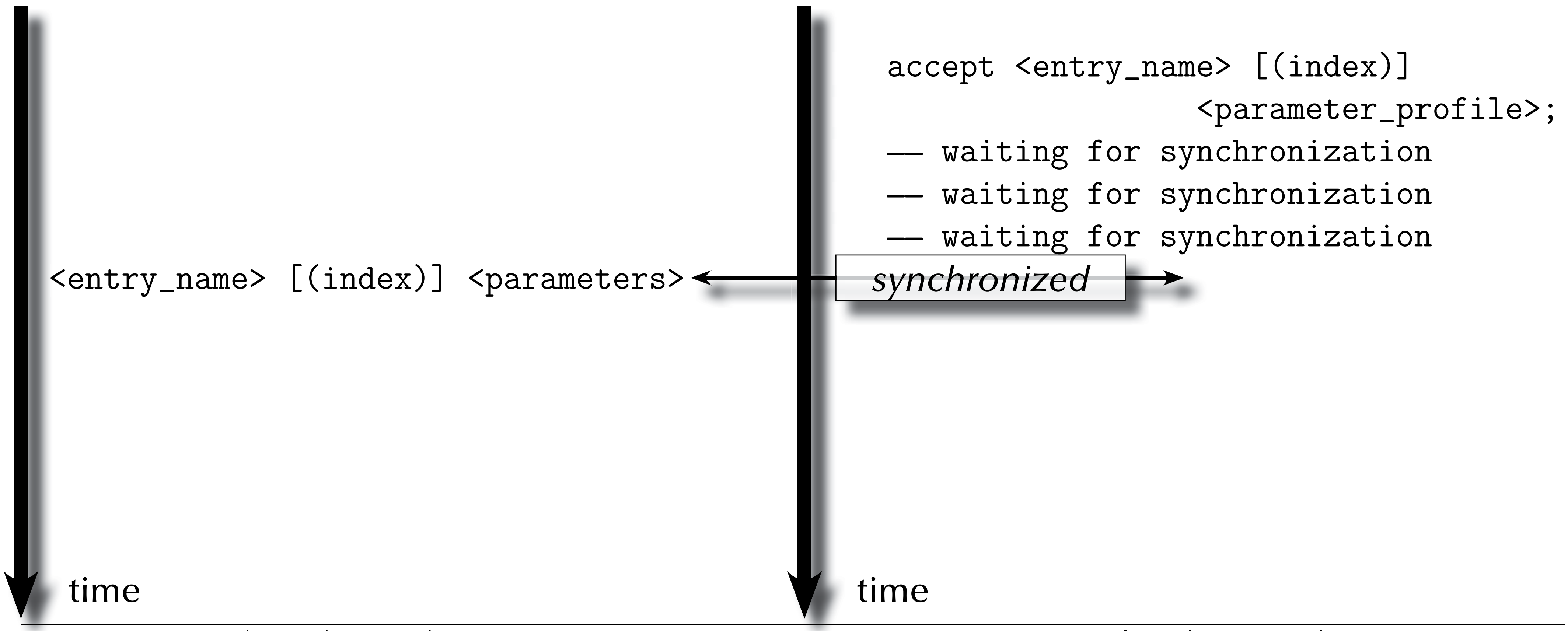
```
<entry_name> [(index)] <parameters>
 -- waiting for synchronization
 -- waiting for synchronization
 -- waiting for synchronization
 -- waiting for synchronization
 -- <-------- synchronized -------->  accept <entry_name> [(index)]
                                               <parameter_profile>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

time                                  time
```

Message-based synchronization

Message-based synchronization in Ada

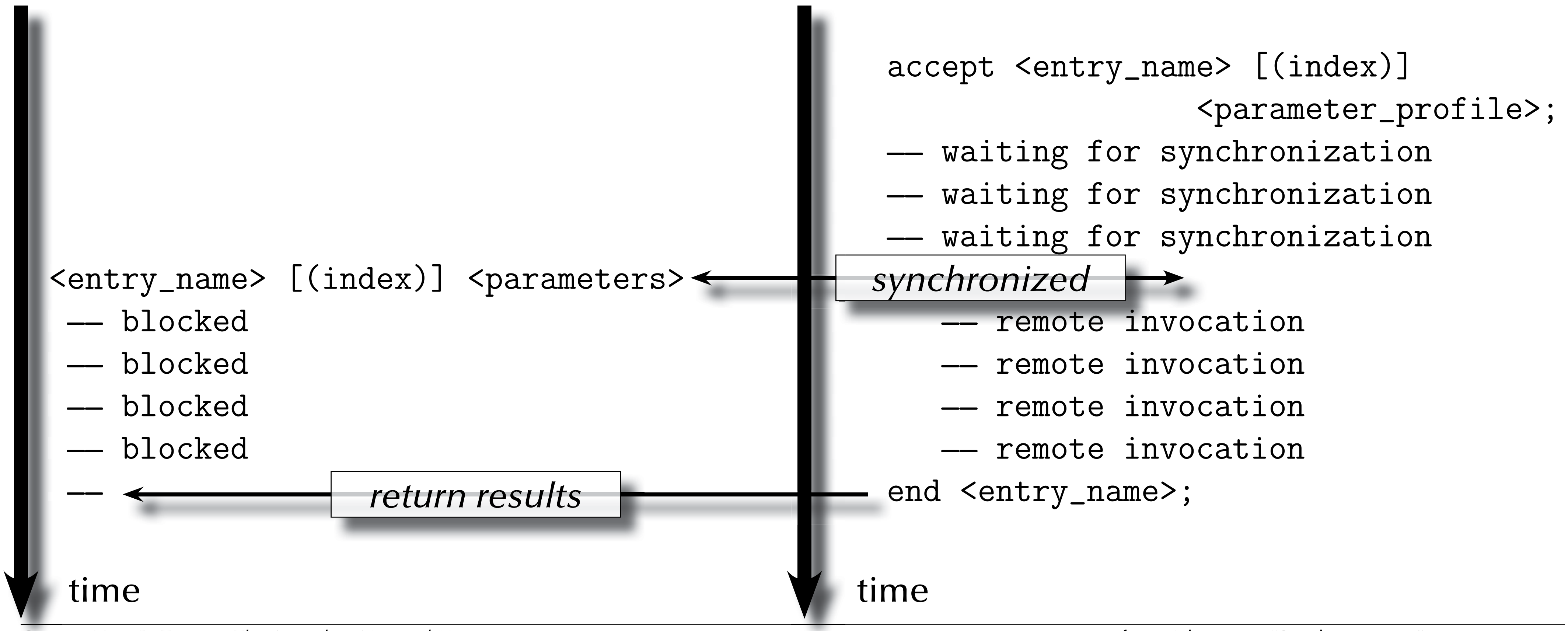
(Rendezvous)



Message-based synchronization

Message-based synchronization in Ada

(Extended rendezvous)



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 rendez-vous 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.

Summary

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

4

Non-determinism

Uwe R. Zimmer - The Australian National University

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

[NN2006]
Ada Reference Manual
- Language and Standard Libraries
Ada Reference Manual ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1 2006

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.

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-deterministc 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-deterministc depending on numeric type

☞ The programmer needs to understand the numerical implications of out-of-order expressions.

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, …

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: `PRI ALT`).

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 (Hoare) also supports non-deterministic selective waiting

Non-determinism by interaction

Select function in POSIX

```
int pselect(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
                   const struct timespec *ntimeout, 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 occured.

¬ 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.

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

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;
```

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 attached to the selected entry (if any) is executed and the select statement completes.

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.

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.

This form never suspends the task.

Selective Synchronization

Basic forms of selective synchronization

(select-guarded-accept-delay)

```
select
   when <condition> => accept …
or
   when <condition> => accept …
or
   when <condition> => accept …
…
or
   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.
- *If no open entries have waiting tasks immediately*
  ☞ The task is suspended until a call arrives on the open entries, but no longer than the deadline specified by the earliest open `delay` alternative

Selective Synchronization

Basic forms of selective synchronization

(select-guarded-accept-terminate)

```
select
   when <condition> => accept …
or
   when <condition> => accept …
or
   when <condition> => accept …
…
or
   when <condition> => terminate;
…
end select;
```

`terminate` cannot be mixed with `else` or `delay`

- *If none of the open entries have waiting calls and none of them can ever be called again*
  ☞ This `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-for-termination tasks are terminated as well.

Selective Synchronization

Conditional entry-calls

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

Example:

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

- *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 never a need in Ada to do so, as better alternatives are available.

There is only *one* entry-call and *one* `else` alternative.

Selective Synchronization

Timed entry-calls

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

Example:

```
select
   Controller.Request (Some_Item);
   -- process data
or
   delay 45.0; -- seconds
   -- try something else
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.

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*.

Non-determinism

Correctness of non-deterministic programs

Partial correctness:

$$(P(I) \wedge terminates(Program(I,O) \Rightarrow Q(I,O)$$

Total correctness:

$$P(I) \Rightarrow (terminates(Program(I,O) \wedge Q(I,O))$$

Safety properties:

$$(P(I) \wedge Processes(I,S)) \Rightarrow \Box Q(I,S)$$

where $\Box Q$ means that Q does *always* hold

Liveness properties:

$$(P(I) \wedge Processes(I,S)) \Rightarrow \Diamond Q(I,S)$$

where $\Diamond Q$ means that Q does *eventually* hold (and will then stay true) and S is the current state of the concurrent system

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 the correctness predicates need to be based on logical **invariants** with re-spect to the variations introduced by the different potential execution sequences.

For example (in verbal form):
"Mutual exclusion accessing a specific resource holds true,
for all possible numbers or requests, sequences of requests and concurrent requests to it"

- ☞ Those invariants are the only practical way to guarantee (in a logical sense) correctness in concurrent / non-deterministc systems.

  (as enumerating all possible cases and proving them individually is in general not feasible)

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-determinstic statement like the one on the left you need to formulate an **invariant** which hold true whichever alternative will actually be chosen.

This is very similar to finding **loop invariants** in sequential programs

Summary

Non-Determinism

- **Non-determimimism 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

Concurrent & Distributed Systems 2010

5

Scheduling

Uwe R. Zimmer - The Australian National University

References for this chapter

[Ben2006]
Ben-Ari, M
Principles of Concurrent and Distributed Programming
second edition, Prentice-Hall 2006

[NN2006]
Ada Reference Manual - Language and Standard Libraries
Ada Reference Manual ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1 2006

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

Purpose of scheduling

Two scenarios for scheduling algorithms:

1. Ordering resource assignments (CPU time, network access, …).
   ☞ live, on-line application of scheduling algorithms.
2. 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, …

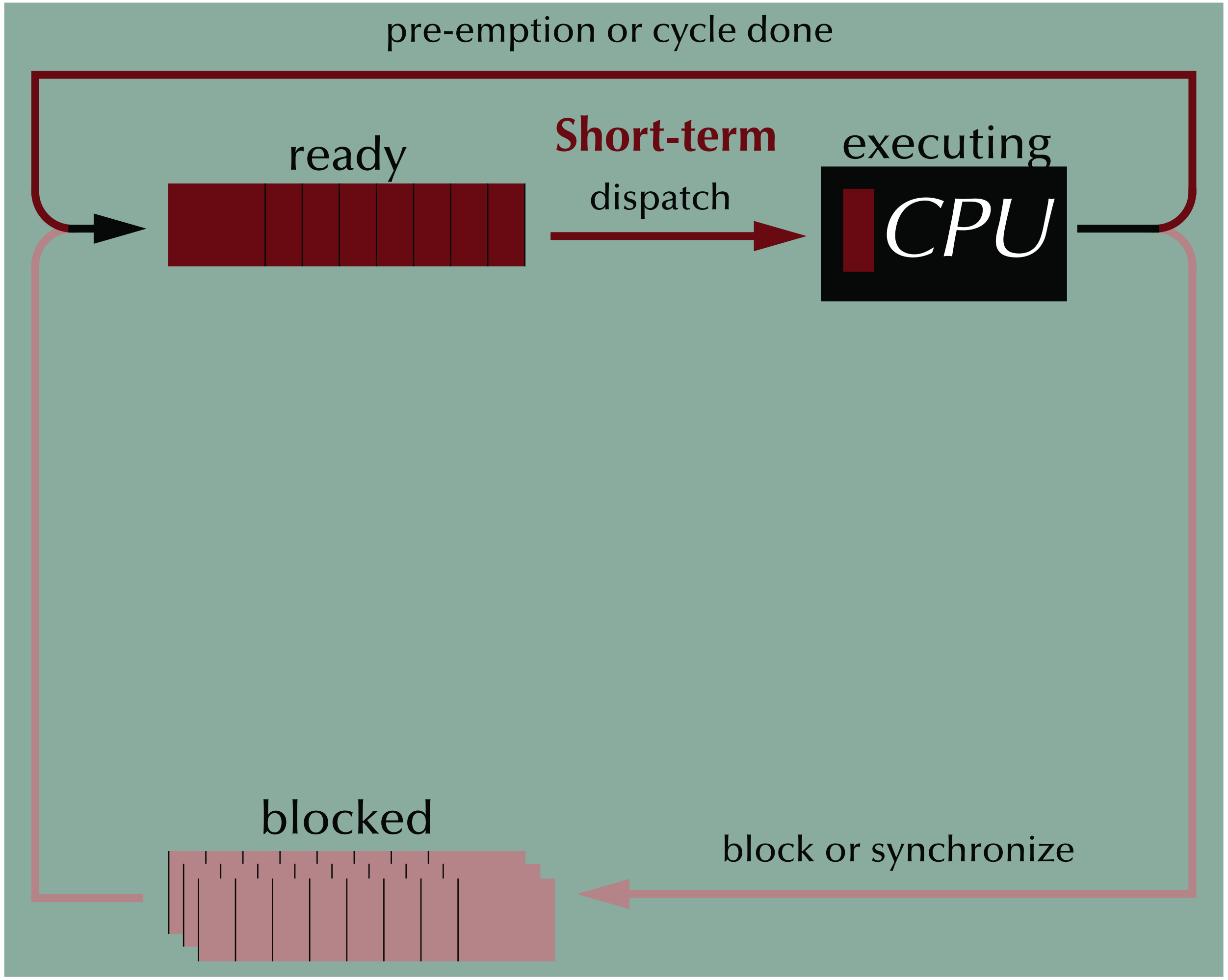
Motivation and definition of terms

Criteria

	Performance criteria:	**Predictability** criteria:
Process / user perspective:		
	minimize the …	minimize diversion from given …
Waiting time	minima / maxima / average / variance	minima / maxima
Response time	minima / maxima / average / variance	minima / maxima / deadlines
Turnaround time	minima / maxima / average / variance	minima / maxima / deadlines
System perspective:		
	maximize the …	
Throughput	minima / maxima / average	
Utilization	CPU busy time	

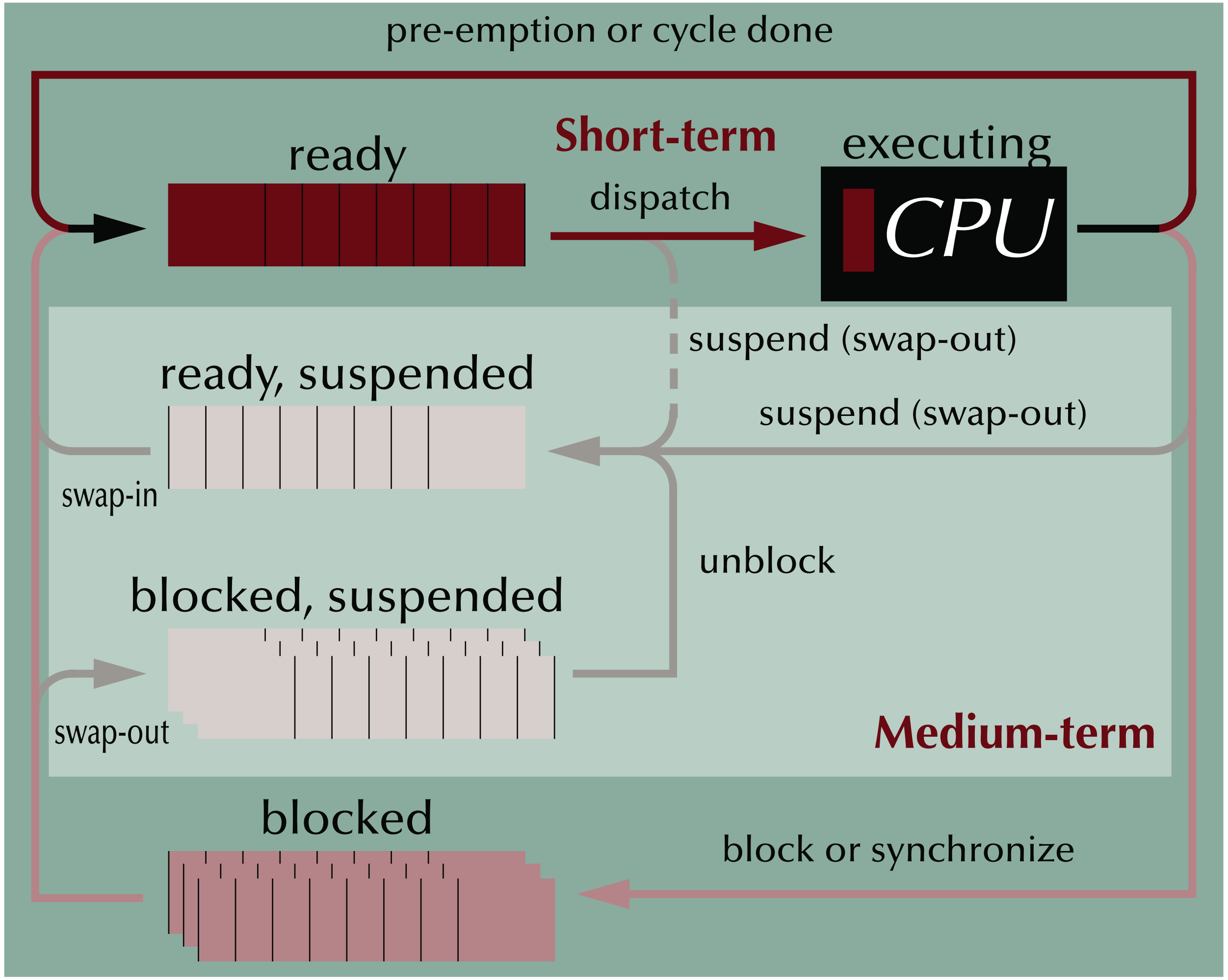
Definition of terms

Time scales of scheduling



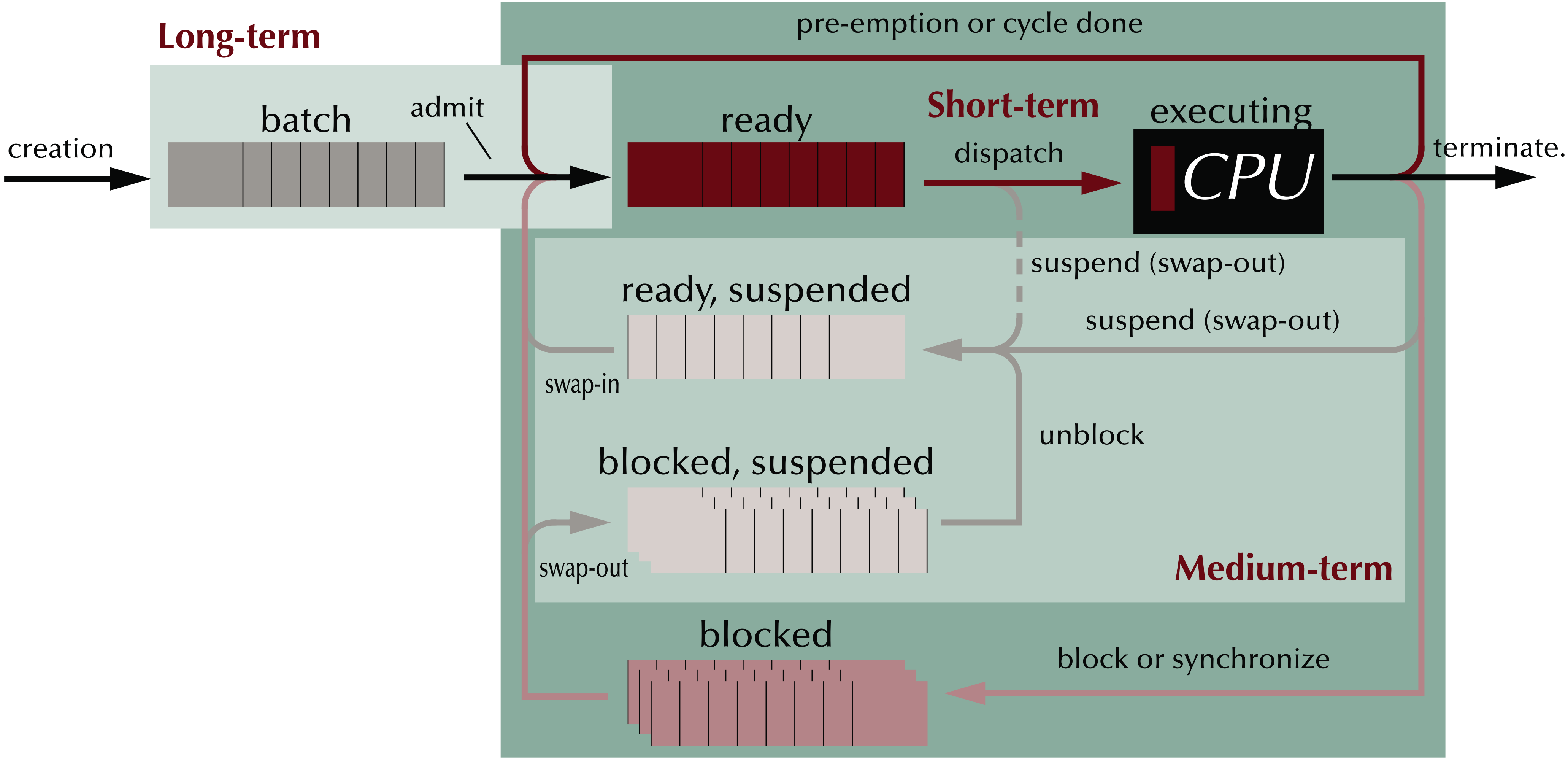
Definition of terms

Time scales of scheduling



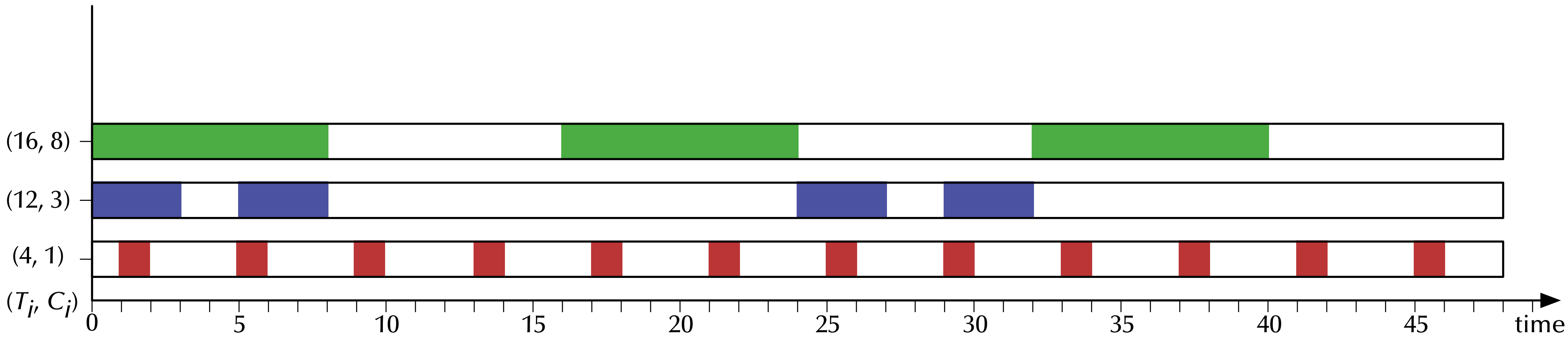
Definition of terms

Time scales of scheduling



Performance scheduling

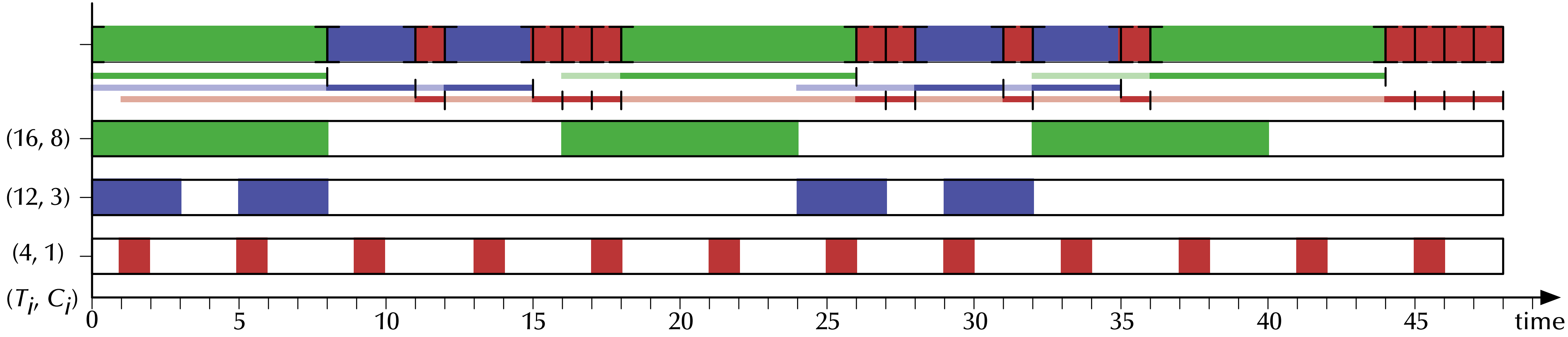
Requested resource times



Tasks have an **average time between instantiations** of T_i
and a constant **computation time** of C_i

Performance scheduling

First come, first served (FCFS)



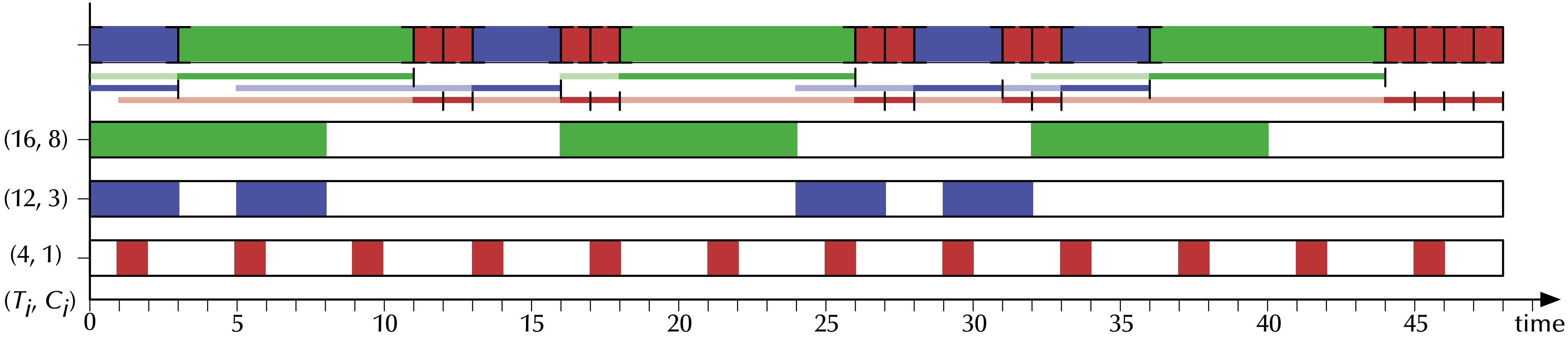
Waiting time: 0..11, average: 5.9 – **Turnaround time:** 3..12, average: 8.4

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.

Performance scheduling

First come, first served (FCFS)



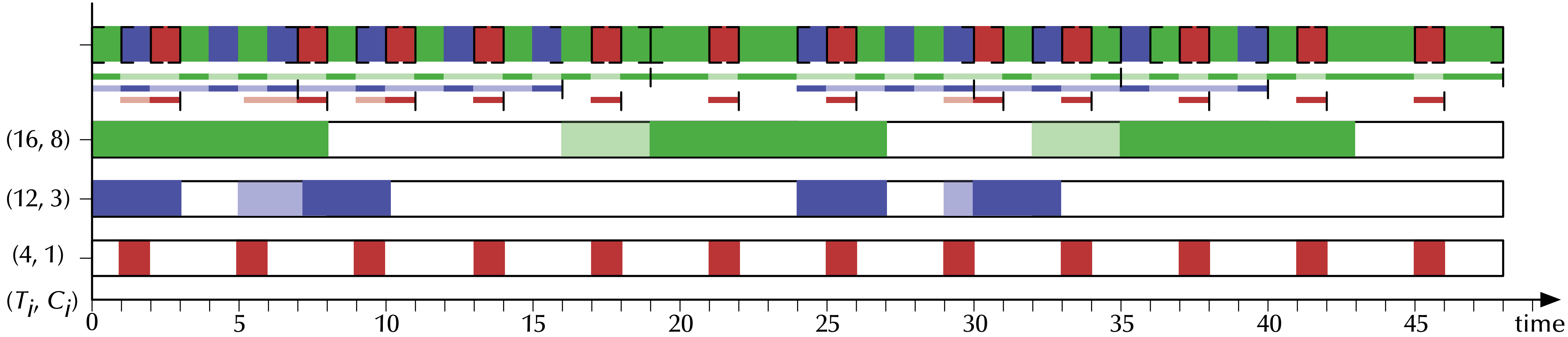
Waiting time: 0..11, average: 5.4 – **Turnaround time:** 3..12, average: 8.0

☞ 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!**

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)

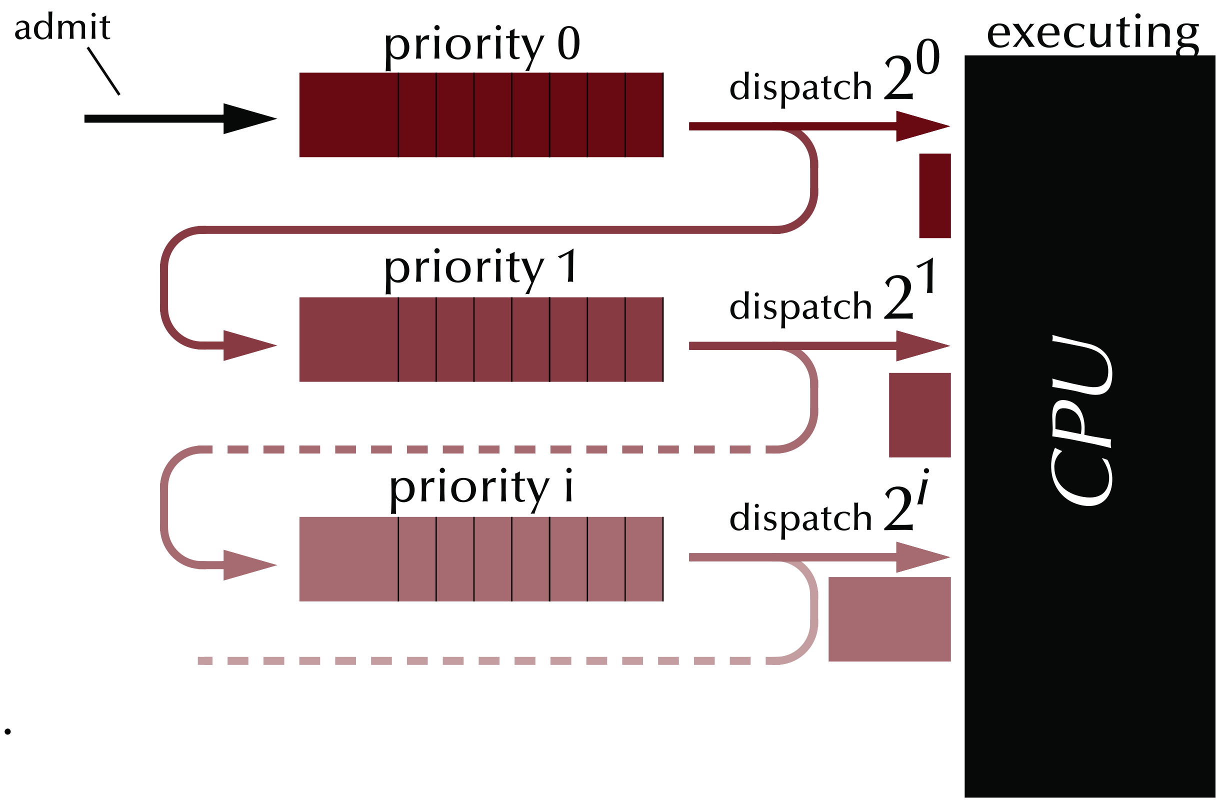
Performance scheduling

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).

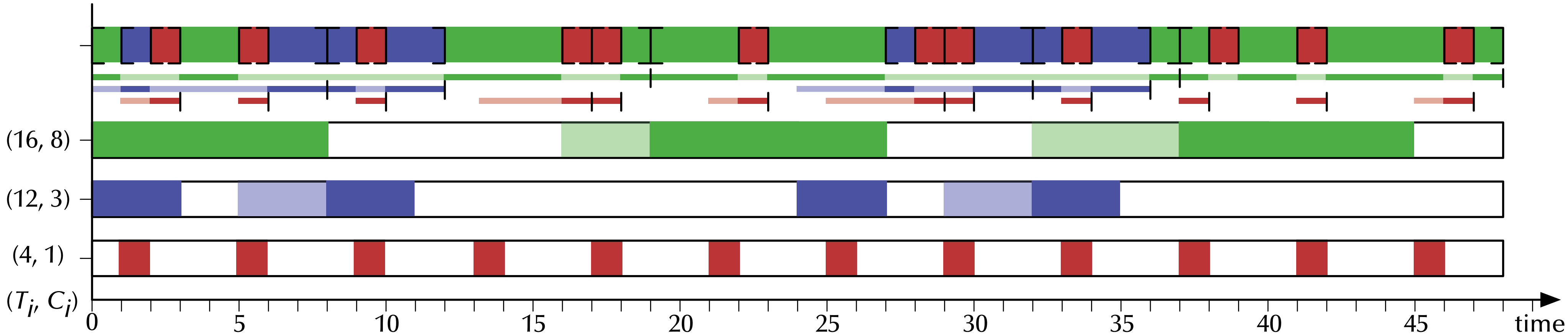
☞ Processes on lower ranks may suffer **starvation**.

☞ New and short tasks will be preferred.



Performance scheduling

Feedback with 2^i pre-emption intervals - sequential



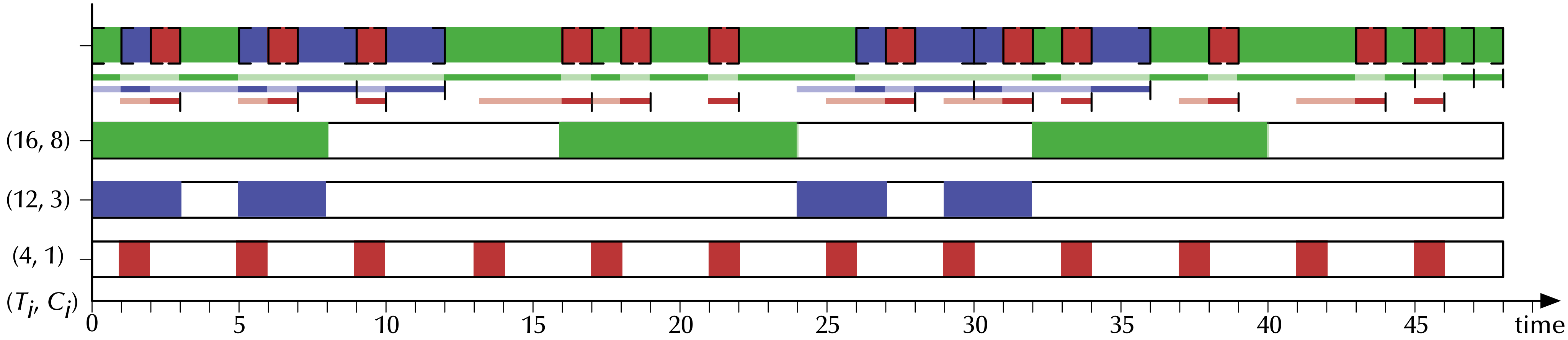
Waiting time: 0..5, average: 1.5 – **Turnaround time**: 1..21, average: 5.7

- ☞ Optimized for swift initial responses.
- ☞ Prefers short tasks and long tasks can suffer starvation.

- ☞ **Very short initial response times!** and good average turnaround times.

Performance scheduling

Feedback with 2^i pre-emption intervals - overlapping



Waiting time: 0..3, average: 0.9 – **Turnaround time**: 1..45, average: 7.7

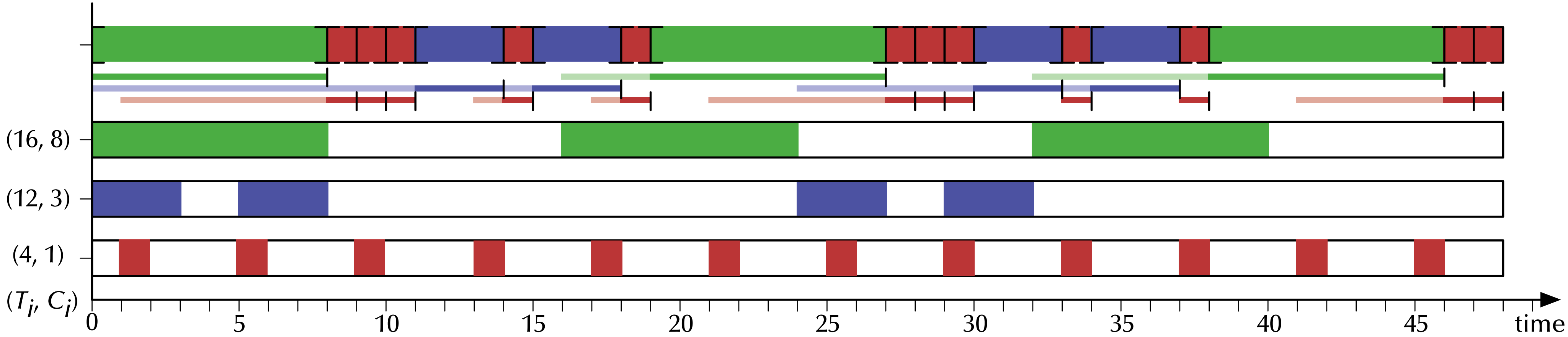
☞ Optimized for swift initial responses.

☞ Prefers short tasks and long tasks can suffer **starvation**.

☞ **Long tasks are delayed until all queues run empty!**

Performance scheduling

Shortest job first



Waiting time: 0..11, average: 3.7 – **Turnaround time**: 1..14, average: 6.3

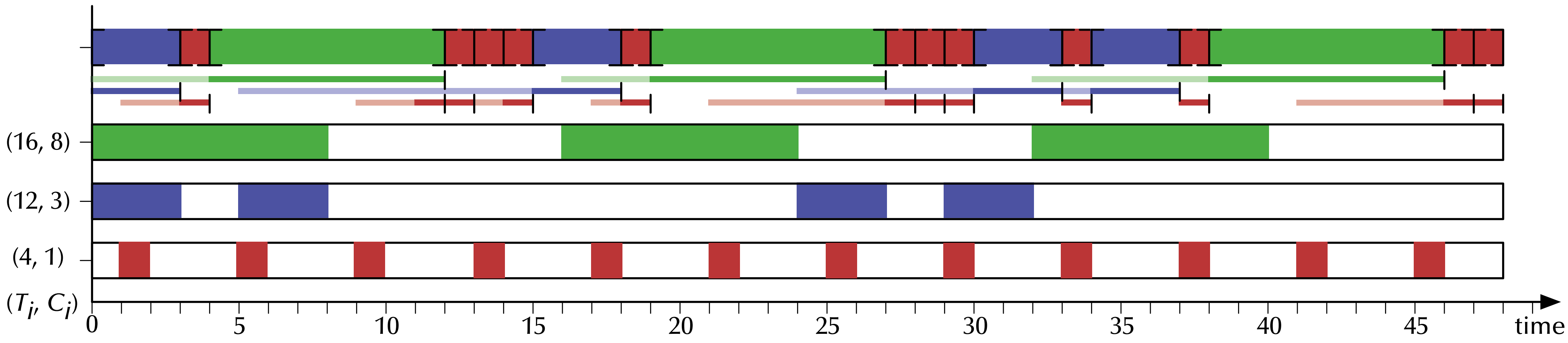
☞ Optimized for good average performance with minimal task-switches.

☞ Prefers short tasks but all tasks will be handled.

☞ **Good choice if computation times are known and task switches are expensive!**

Performance scheduling

Shortest job first

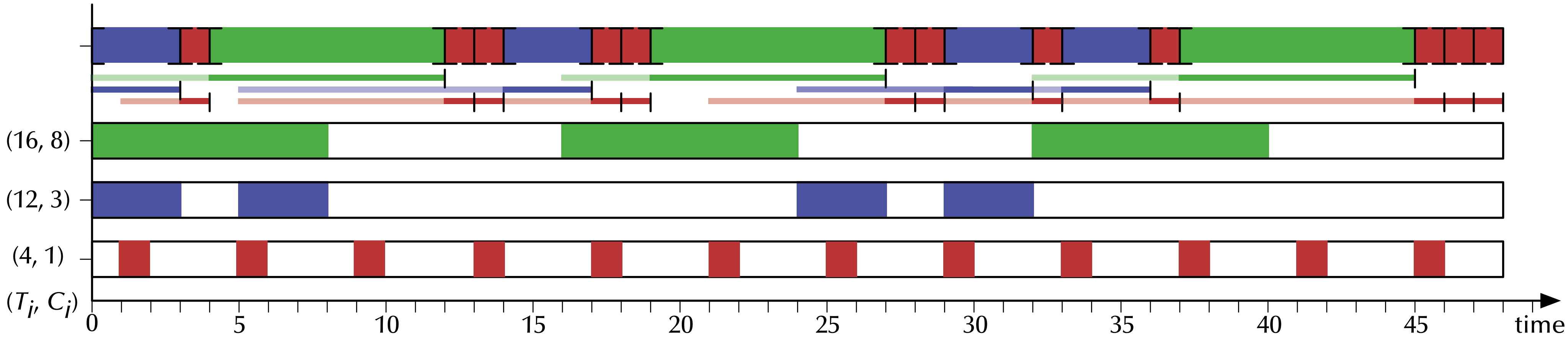


Waiting time: 0..10, average: 3.4 – **Turnaround time:** 1..14, average: 6.0

☞ Can be sensitive to non-deterministic arrival sequences.

Performance scheduling

Highest Response Ration $\frac{W_i + C_i}{C_i}$ First (HRRF)



Waiting time: 0..9, average: 4.1 – **Turnaround time**: 2..13, average: 6.6

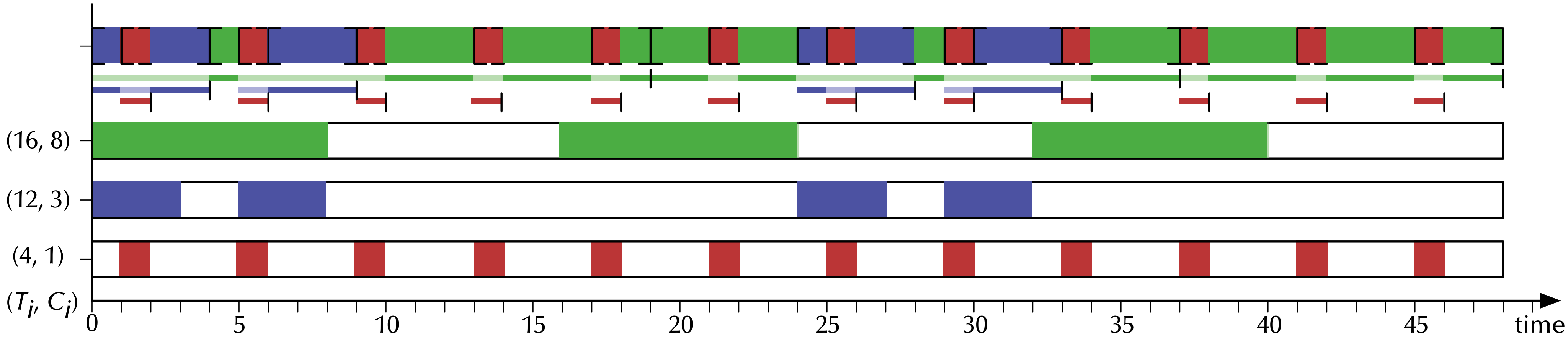
☞ Blend between Shortest-Job-First and First-Come-First-Served.

☞ Prefers short tasks but long tasks gain preference over time.

☞ **More task switches and worse averages than SJF but better upper bounds!**

Performance scheduling

Shortest Remaining Time First (SRTF)



Waiting time: 0..6, average: 0.7 – **Turnaround time**: 1..21, average: 4.4

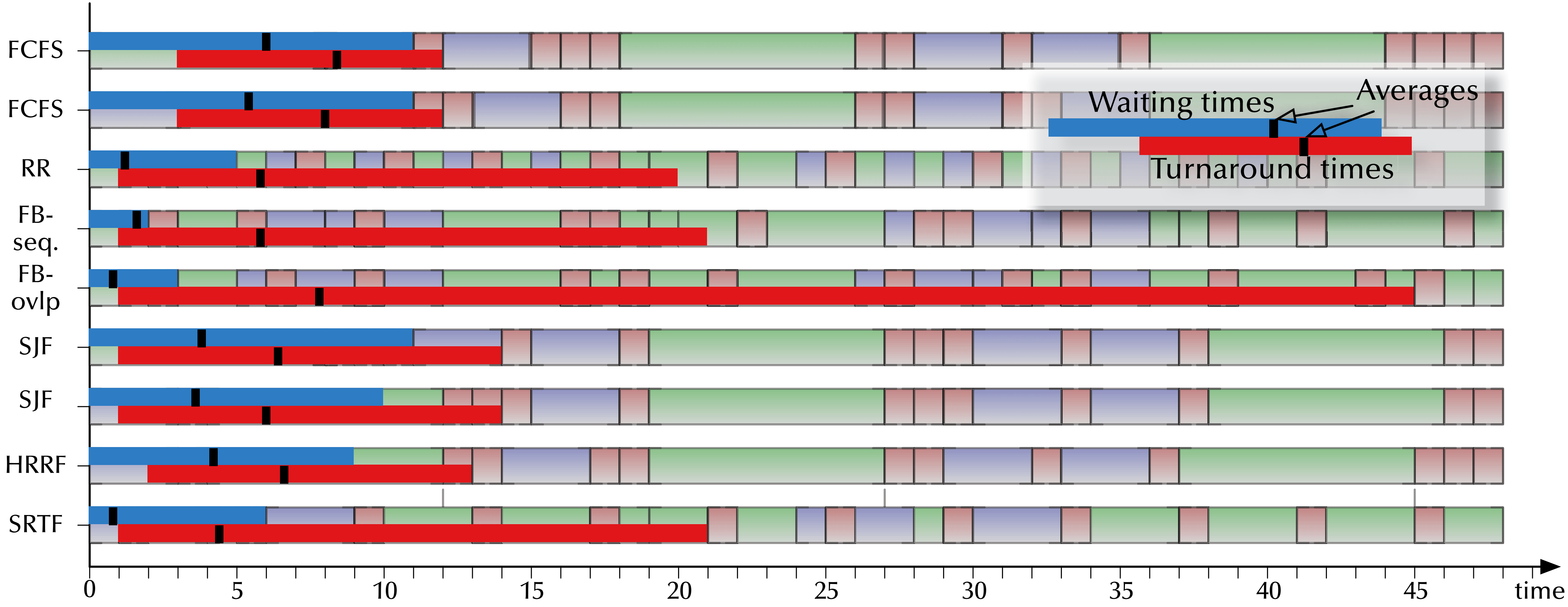
☞ Optimized for good averages.

☞ Prefers short tasks and long tasks can suffer **starvation**..

☞ **Better averages than Feedback scheduling but with longer absolute waiting times!**

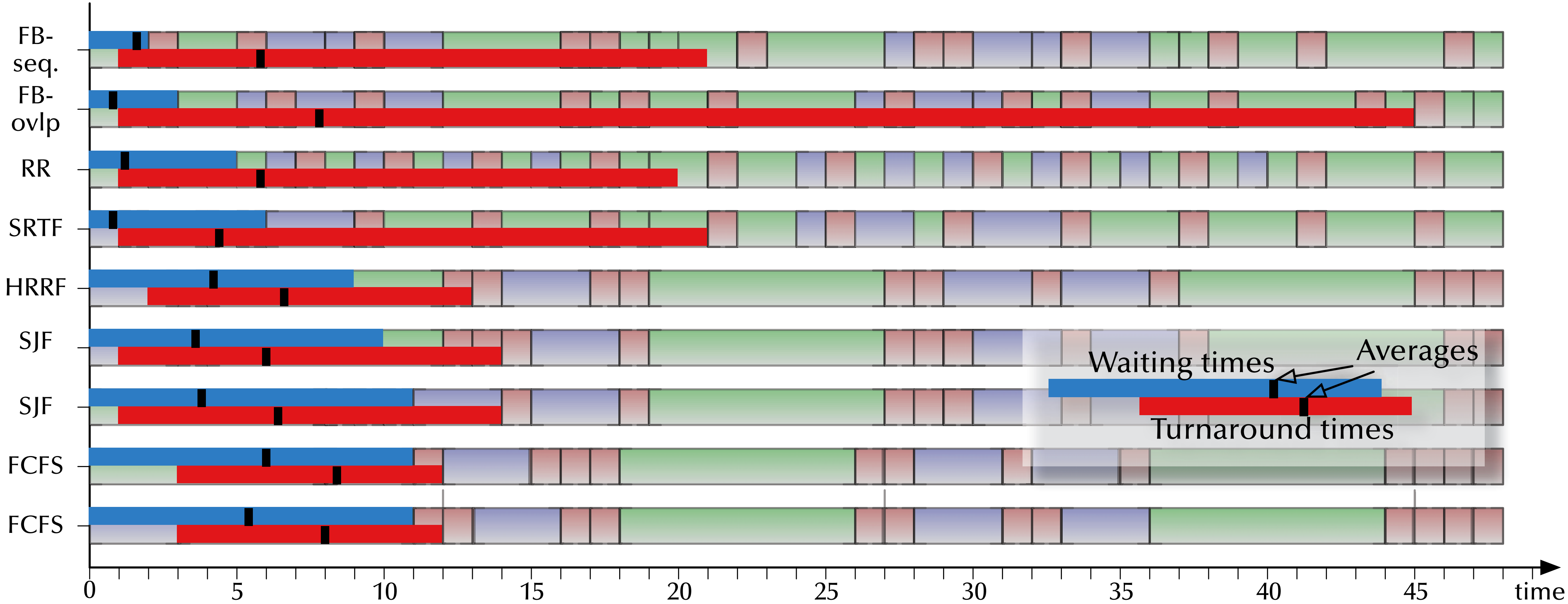
Performance scheduling

Comparison (in order of appearance)



Performance scheduling

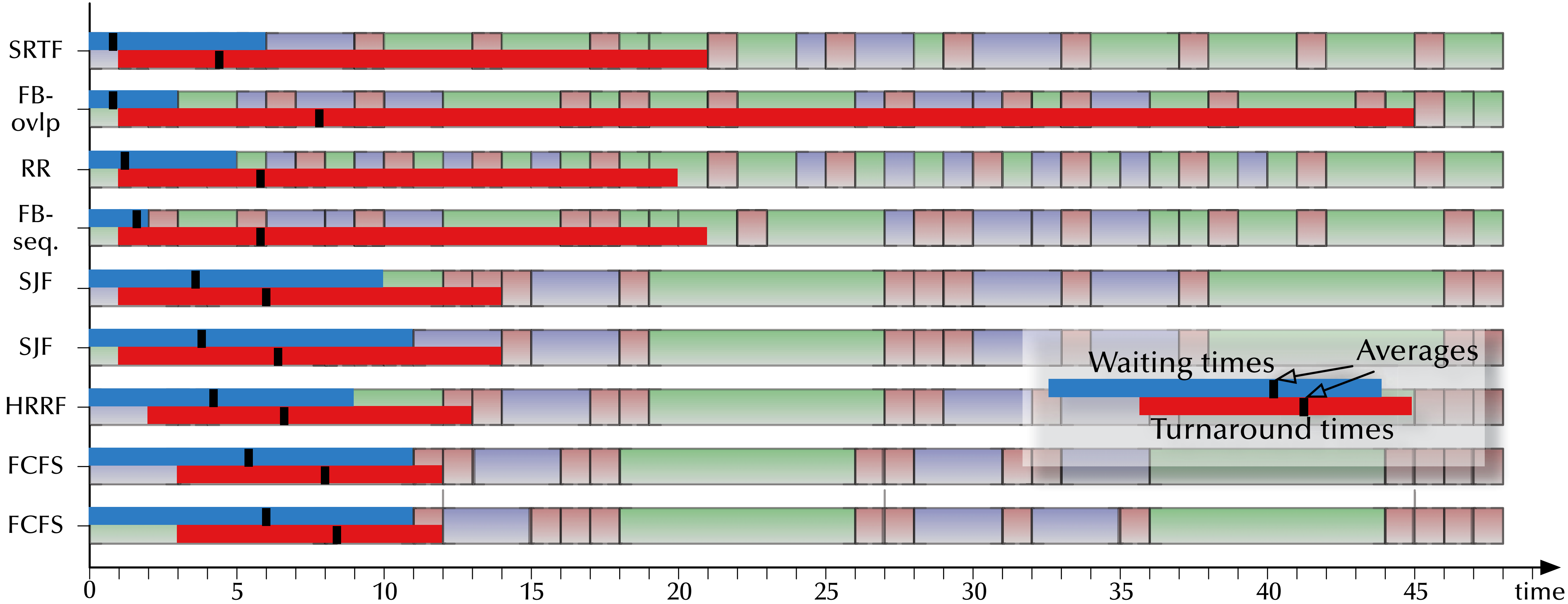
Comparison by shortest maximal waiting



☞ Providing upper bounds to waiting times ☞ Swift response systems

Performance scheduling

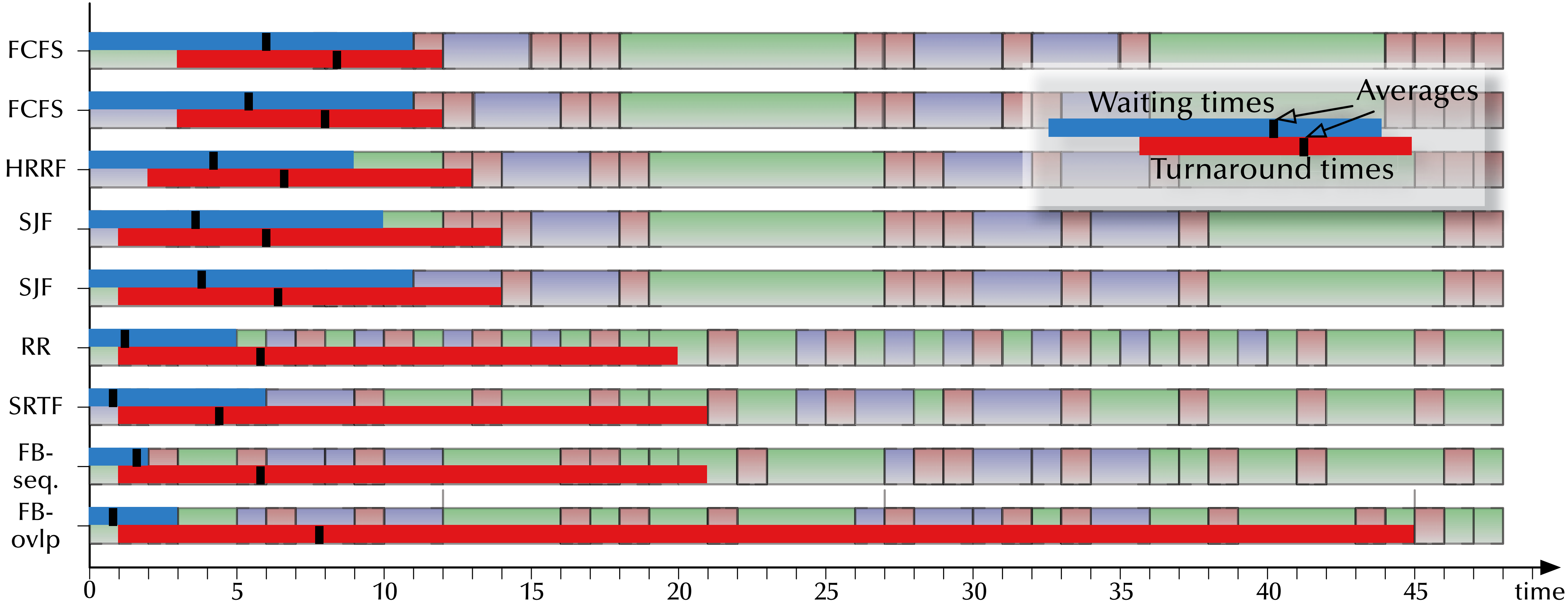
Comparison by shortest average waiting



☞ Providing short average waiting times ☞ Very swift response in most cases

Performance scheduling

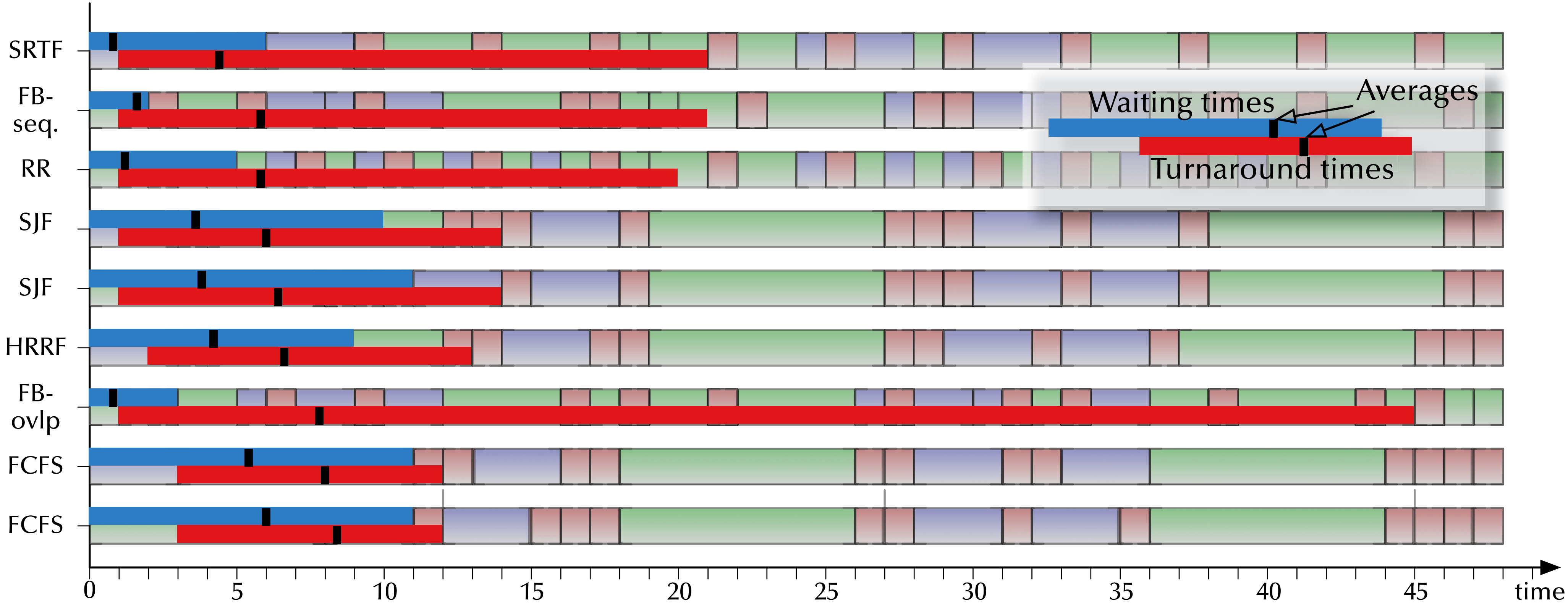
Comparison by shortest maximal turnaround



☞ Providing upper bounds to turnaround times ☞ No tasks are left behind

Performance scheduling

Comparison by shortest average turnaround



☞ Providing good average performance ☞ High throughput systems

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	$\max\left(\frac{W_i + C_i}{C_i}\right)$	no	controllable compromise	controllable compromise	controllable	no
SRTF	$\min(C_i - E_i)$	yes	very short	wide variance	short	yes

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:

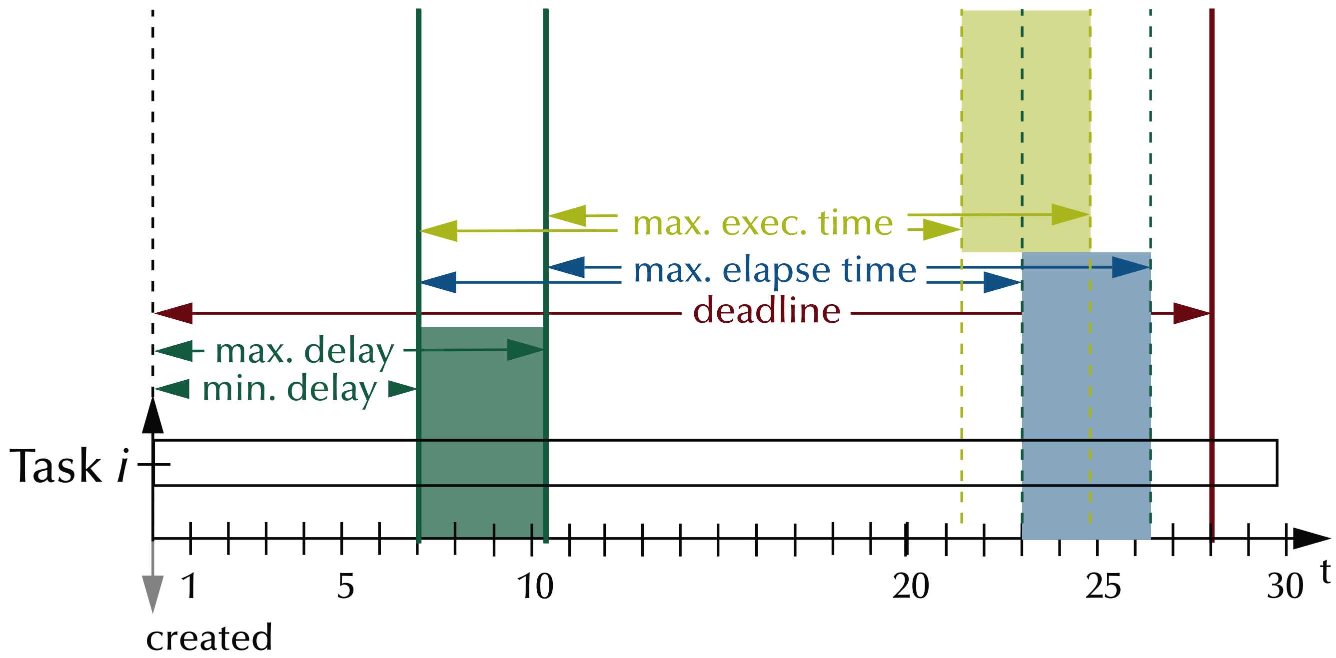
- Streaming media broadcasts, playing HD videos, live mixing audio/video, …
- Reacting to users, Reacting to alarm situations, …
- Delivering a signal to the physical world at the required time, …

Predictable scheduling

Temporal scopes

Common attributes:

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

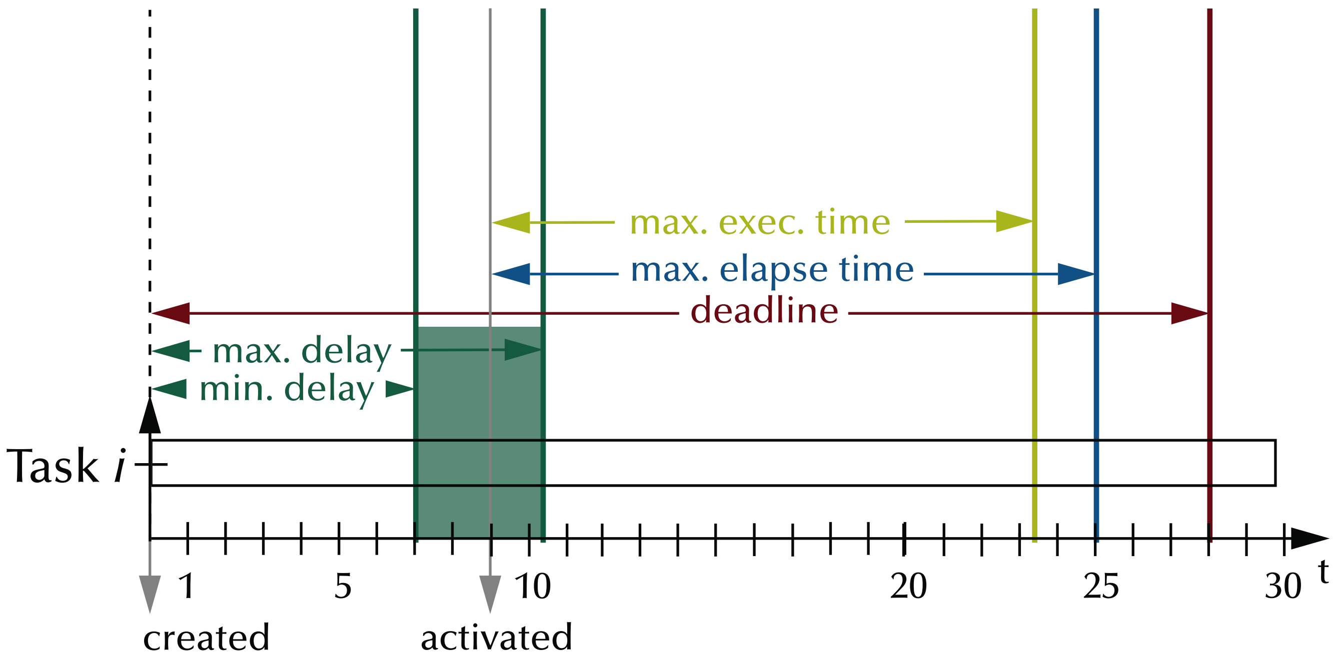


Predictable scheduling

Temporal scopes

Common attributes:

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

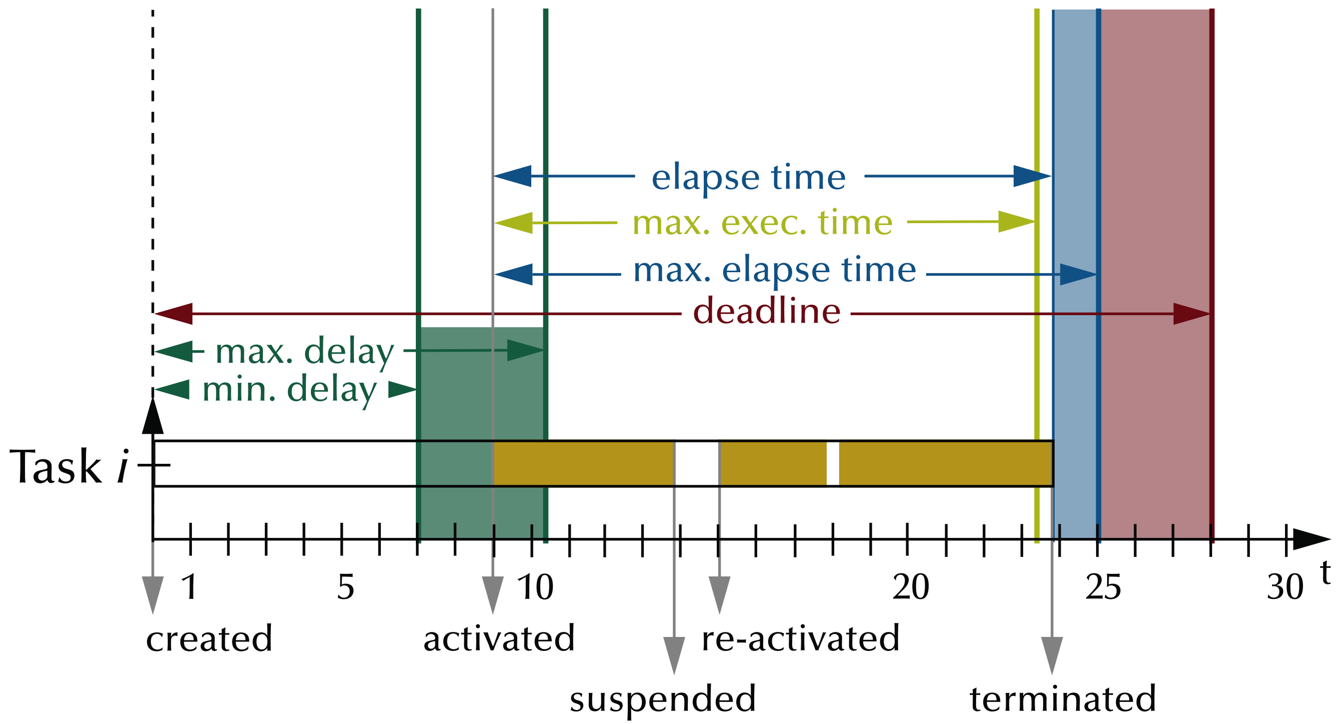


Predictable scheduling

Temporal scopes

Common attributes:

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

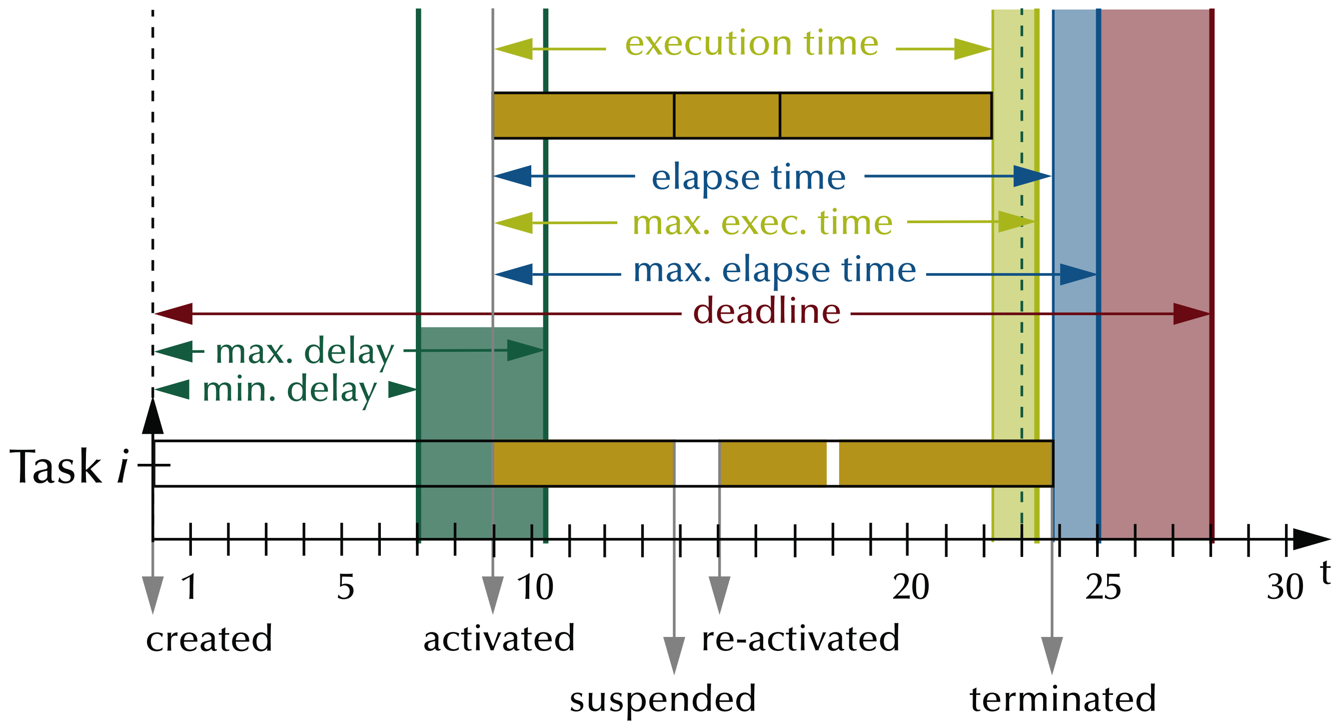


Predictable scheduling

Temporal scopes

Common attributes:

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



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:

"Hard"	☞ single failure leads to severe malfunction and/or disaster
"Firm"	☞ results are meaningless after the deadline
	☞ only multiple or permanent failures lead to malfunction
"Soft"	☞ results are still useful after the deadline

Semantics defined by application

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

6

Safety & Liveness

Uwe R. Zimmer - The Australian National University

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

Repetition

Correctness concepts in concurrent systems

Extended concepts of correctness in concurrent systems:

$\neg$ Termination is often not intended or even considered a failure

Safety properties:

$$(P(I) \land Processes(I,S)) \Rightarrow \Box Q(I,S)$$

where $\Box Q$ means that Q does *always* hold

Liveness properties:

$$(P(I) \land Processes(I,S)) \Rightarrow \Diamond Q(I,S)$$

where $\Diamond Q$ means that Q does *eventually* hold (and will then stay true)
and S is the current state of the concurrent system

Repetition

Correctness concepts in concurrent systems

Liveness properties:

$$(P(I) \wedge Processes(I,S)) \Rightarrow \Diamond Q(I,S)$$

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

Liveness

Fairness

Liveness properties:

$$(P(I) \wedge Processes(I,S)) \Rightarrow \Diamond Q(I,S)$$

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).

Revisiting

Correctness concepts in concurrent systems

Safety properties:

$$(P(I) \wedge Processes(I,S)) \Rightarrow \square Q(I,S)$$

where $\square Q$ means that Q does *always* hold

Examples:

- Mutual exclusion (no resource collisions) ☞ ▯
- 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)

Deadlocks

Most forms of synchronization may lead to

Deadlocks

(avoidance / prevention of those 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?

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: $A \rightharpoonup B \rightharpoonup C$; $X \rightharpoonup Y \rightharpoonup Z$; $[X, Z \mid A, B, C]$; $[A, C \mid X, Y, Z]$; $\neg[B \mid Y]$
or: $[A \mid X]$ followed by a deadlock situation.

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: $A \rightharpoonup B \rightharpoonup C; X \rightharpoonup Y \rightharpoonup Z; K \rightharpoonup L \rightharpoonup M;$

$[X, Z \mid A, B, C \mid K, M]; [A, C \mid X, Y, Z \mid K, M]; [A, C \mid K, L, M \mid X, Z]; \neg[B \mid Y \mid L]$

or: $[A \mid X \mid K]$ followed by a deadlock situation.

Deadlocks

Necessary deadlock conditions:

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

☞ A system *may* become deadlocked, if *all* these conditions apply!

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

Deadlock prevention

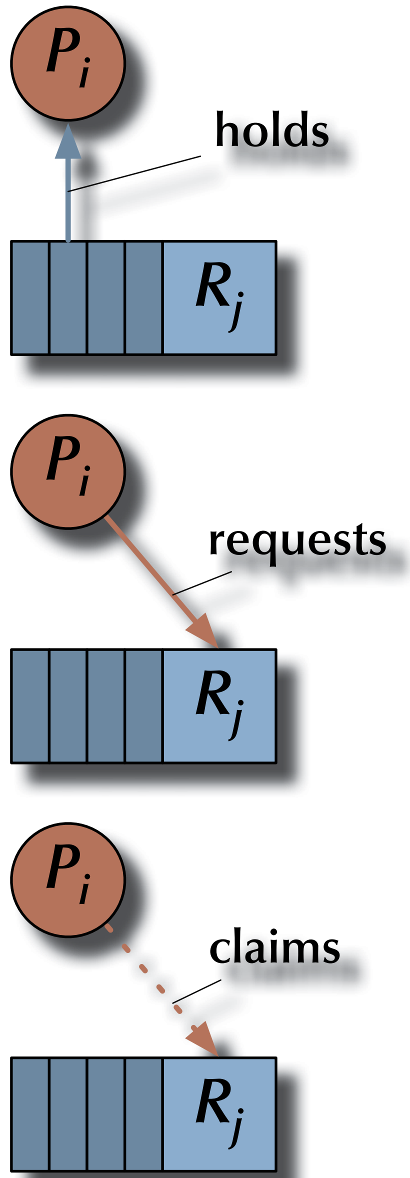
(remove one of the four necessary deadlock conditions)

1. **Break mutual exclusion**:
   By replicating critical resources, mutual exclusion becomes unnecessary (only applicable in very specific cases).
2. **Break hold and wait**:
   Allocation of all required resources in one request.
   Processes can either hold none or all of their required resources.
3. **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.
4. **Break circular waits**:
   Order all resources globally and restrict processes to request resources in that order only.

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 $P = \{P_1, \ldots, P_n\}$
and resources types $R = \{R_1, \ldots R_k\}$

$E = E_c \cup E_r \cup E_a$; Edges E can be claims E_c, requests E_r or assignments E_a
with claims $E_c = \{P_i \rightarrow R_j, \ldots\}$
and requests $E_r = \{P_i \rightarrow R_j, \ldots\}$
and assignment $E_a = \{P_i \rightarrow R_j, \ldots\}$

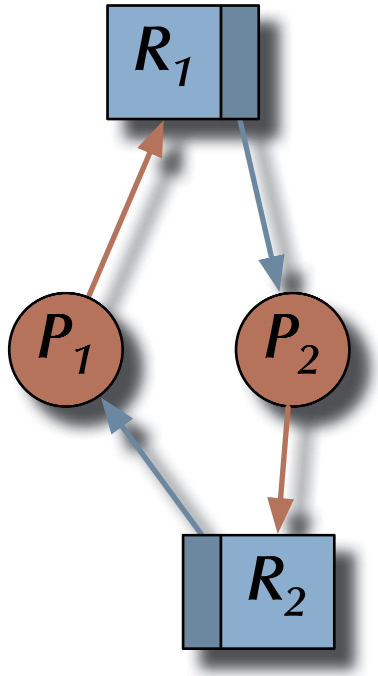
Note: any resource type R_j can have more than one instance of a resource.

Deadlocks

Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

☞ Two process, reverse allocation deadlock:

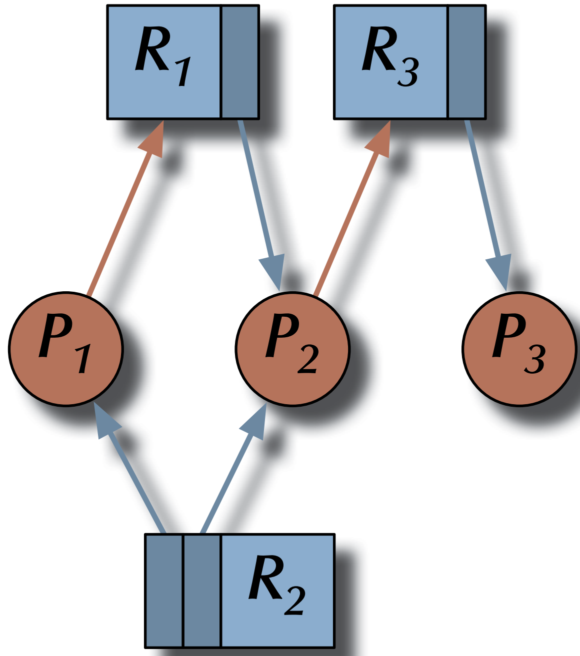


Deadlocks

Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

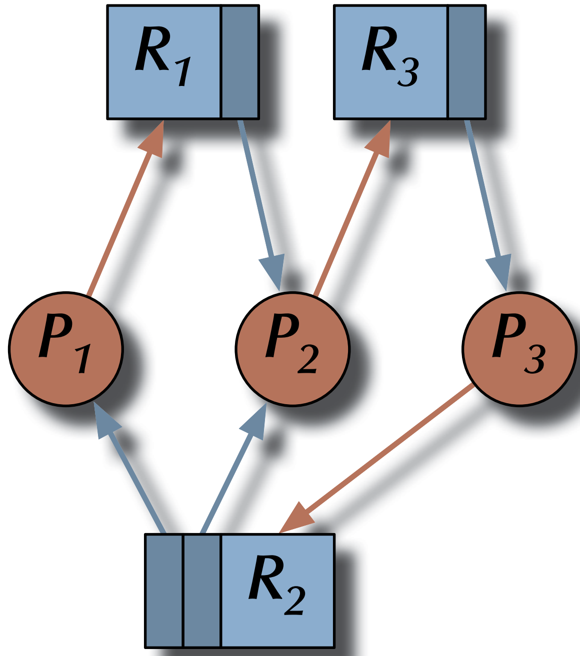
☞ No circular dependency ☞ no deadlock:



Deadlocks

Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)



☞ Two circular dependencies ☞ deadlock:

$P_1 \to R_1 \to P_2 \to R_3 \to P_3 \to R_2 \to P_1$

as well as: $P_2 \to R_3 \to P_3 \to R_2 \to P_2$

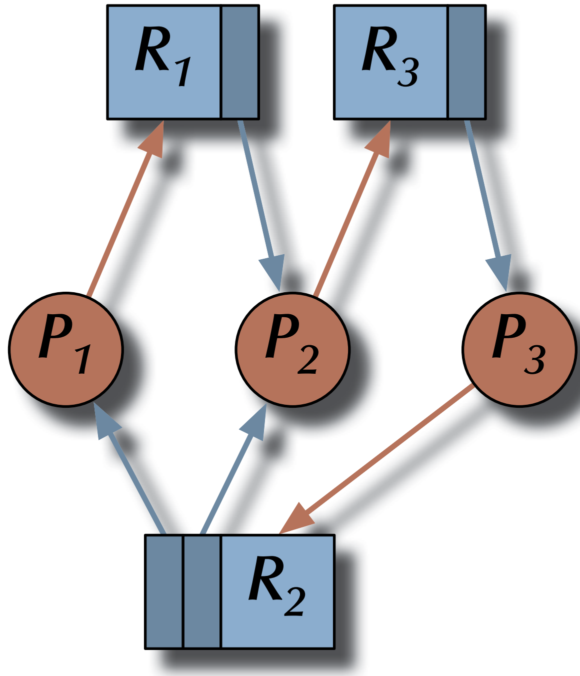
Derived rule:

If some processes are deadlocked **then** there are cycles in the resource allocation graph.

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:

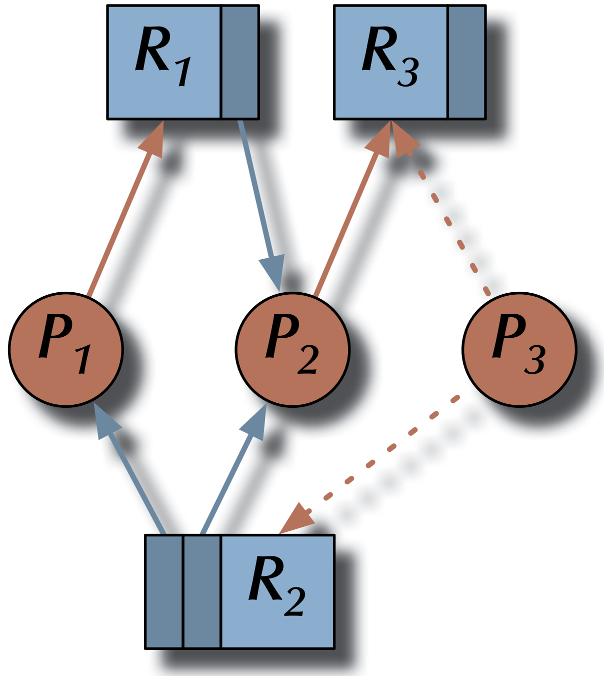
(blocked-id = targeted-id)

☞ *Circular dependency detected.*

Deadlocks

Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)



☞ Knowledge of claims:

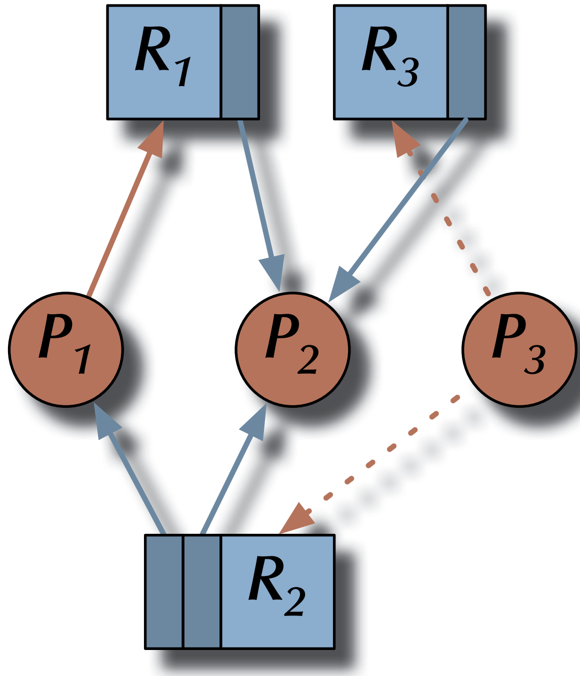
Claims are potential future requests which have no blocking effect on the claiming process – while actual *requests* are blocking.

Deadlocks

Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

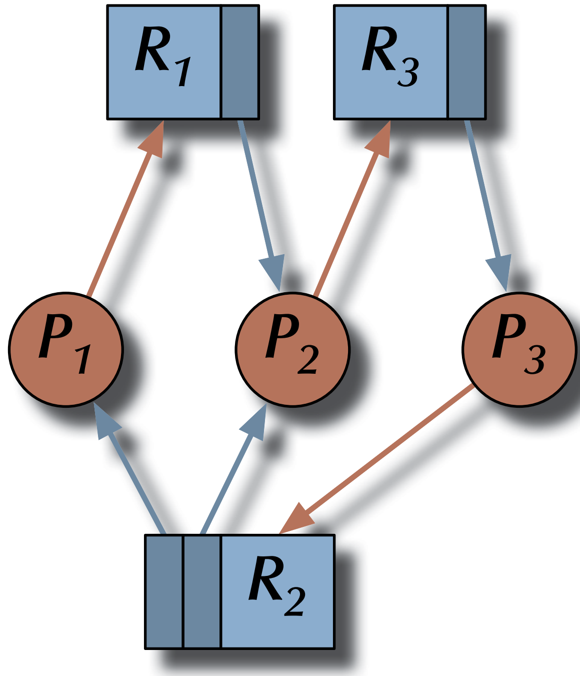
☞ Assignment of resources such that circular dependencies are avoided:



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.

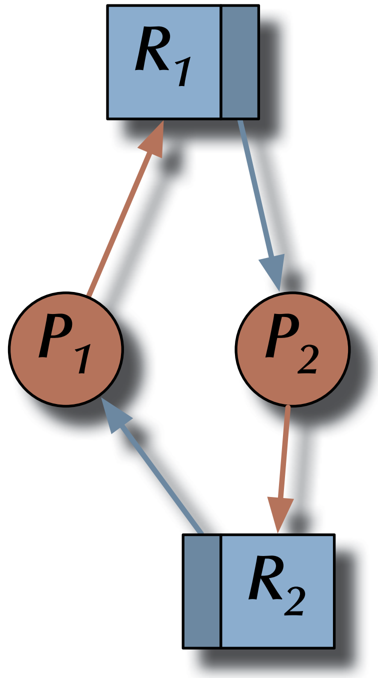
☞ Reverse rule for single instances:

If there are cycles in the resource allocation graph
and there is *exactly one* instance per resource
then the involved processes are *actually* deadlocked.

Deadlocks

Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)



Reverse rule for single instances:

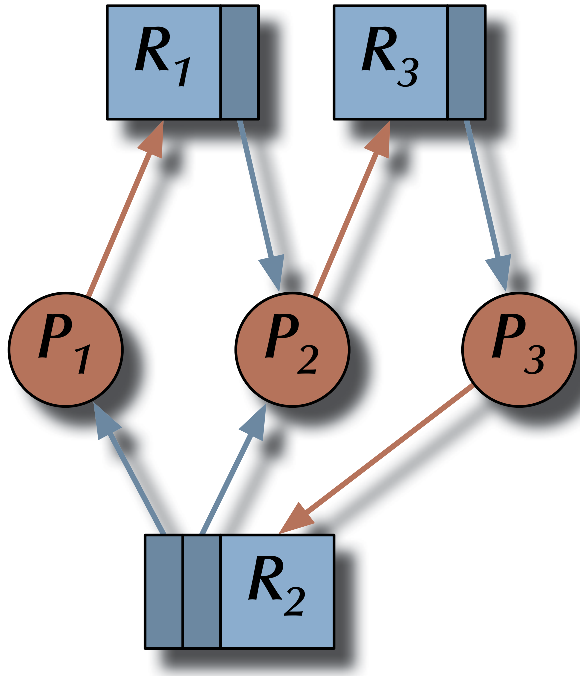
If there are cycles in the resource allocation graph
and there is *exactly one* instance per resource
then the involved processes are *actually* deadlocked.

☞ Actual deadlock identified

Deadlocks

Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)



Reverse rule for multiple instances:

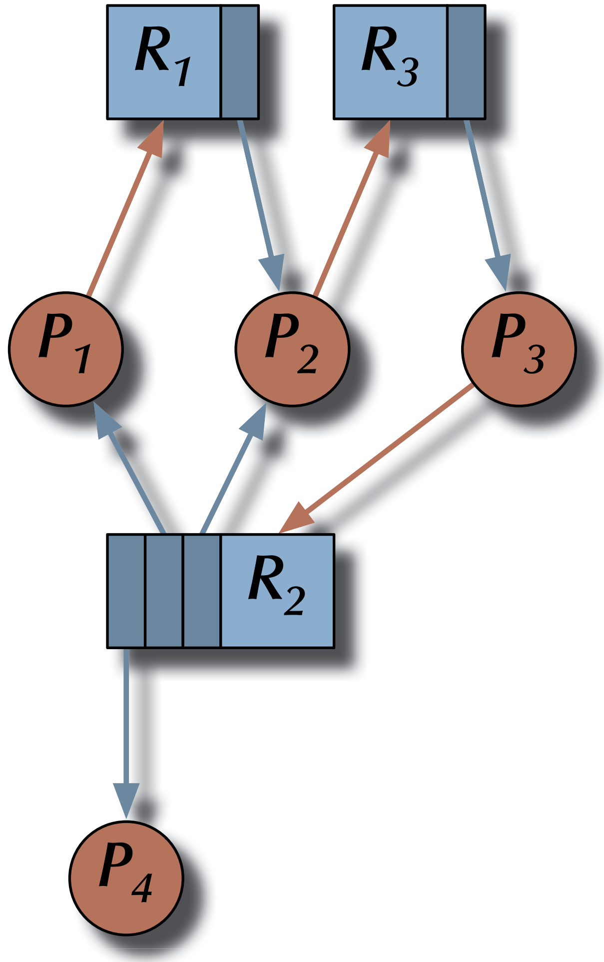
If there are cycles in the resource allocation graph
and there are *multiple* instances per resource
then the involved processes are *potentially* deadlocked.

☞ Potential deadlock identified

Deadlocks

Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)



Reverse rule for multiple instances:

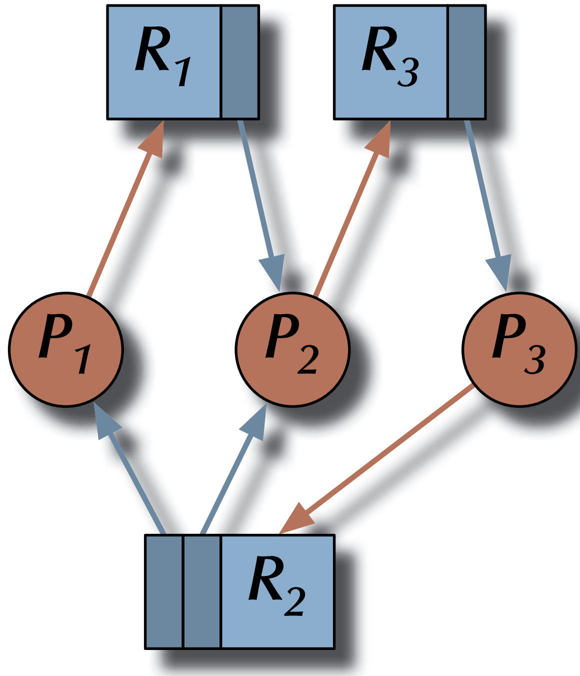
If there are cycles in the resource allocation graph
and there are *multiple* instances per resource
then the involved processes are *potentially* deadlocked.

☞ Potential deadlock identified
– yet clearly not an actual deadlock here

Deadlocks

Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)



How to detect actual deadlocks in the general case?

(multiple instances per resource)

Deadlocks

Banker's Algorithm

There are processes $P_i \in \{P_1, \ldots, P_n\}$ and resource types $R_j \in \{R_1, \ldots, R_m\}$ 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 de-allocate their resources.

Deadlocks

Banker's Algorithm

1. $\texttt{Simulated\_Free} \Leftarrow \texttt{Free};\ \forall\, \texttt{i}: \texttt{Completed [i]} \Leftarrow \texttt{False};$

2. While $\exists\, \texttt{i}: \neg\texttt{Completed [i]}$
   and $\forall\, \texttt{j}: \texttt{Requested [i, j]} < \texttt{Simulated\_Free [j]}$ do:

   $\forall\, \texttt{j}: \texttt{Simulated\_Free [j]} \Leftarrow \texttt{Simulated\_Free [j]} + \texttt{Allocated [i, j]};$
   $\texttt{Completed [i]} \Leftarrow \texttt{True};$

3. If $\forall\, \texttt{i}: \texttt{Completed [i]}$ then the system is currently **deadlock-free!**
   else all processes $\texttt{i}$ with $\neg\texttt{Completed [i]}$ are involved in a **deadlock!**.

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.

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;
```

☞ System state check by Banker's Algorithm

```
   If System_is_safe then
```

☞ Actually grant request

```
   else

      -- restore former system state: (Free, Claimed, Allocated)

   end if;
end if;
```

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*.

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)

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 computation 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.**

Atomic & idempotent operations

Atomic operations

Definitions of atomicity:

An operation is atomic if the processes performing it …

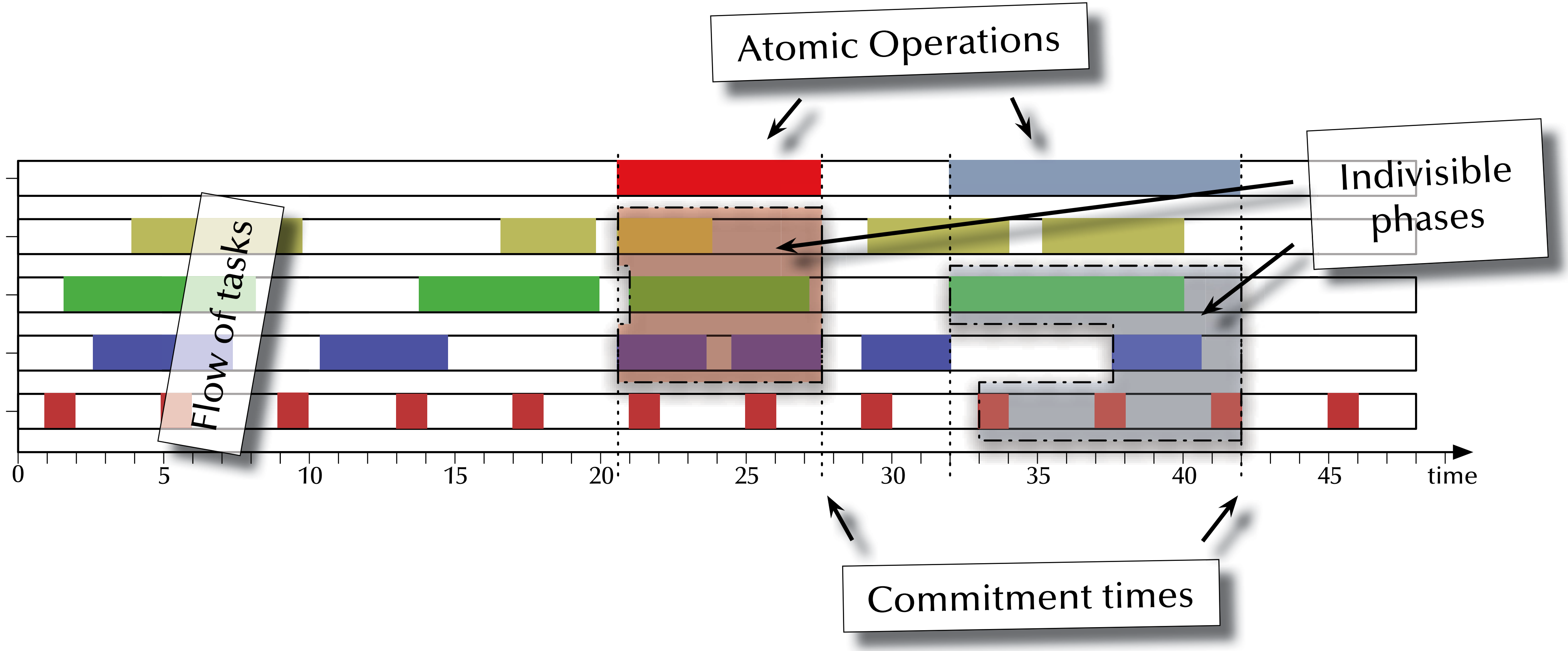
- **(by 'Awareness')** … are not aware of the existence of any other active process, and no other active process is aware of the activity of the processes during the time the processes are performing the atomic operation.
- **(by communication)** … do not communicate with other processes while the atomic 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.**

Atomic & idempotent operations

Atomic operations



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.

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.

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.

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

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

Reliability, failure & tolerance

Faults in the time domain

- Transient faults

  ☞ Single ‘glitches’, interference, … very hard to handle

- Intermittent faults

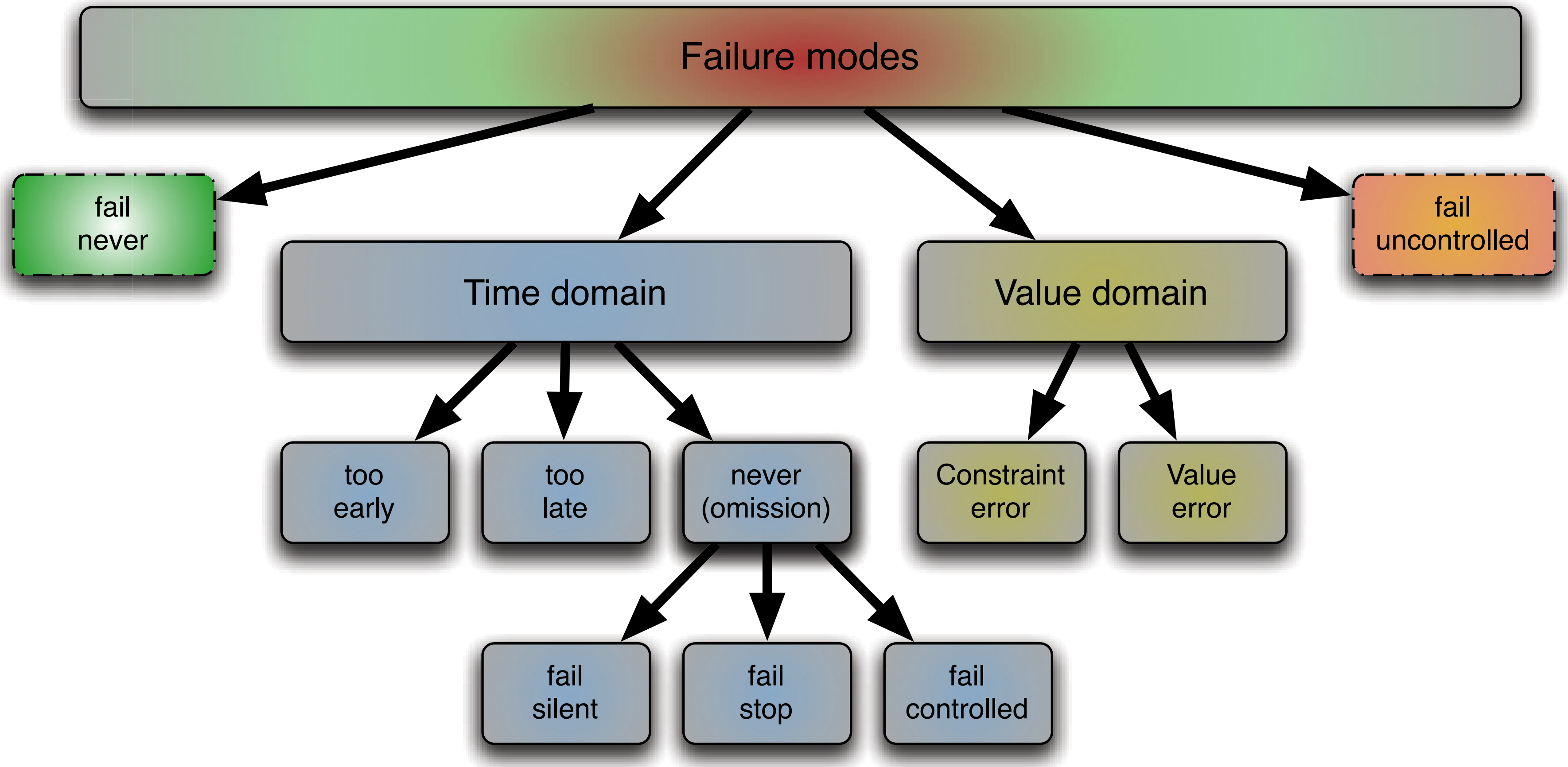
  ☞ Faults of a certain regularity … require careful analysis

- Permanent faults

  ☞ Faults which stay … the easiest to find

Reliability, failure & tolerance

Observable failure modes



Reliability, failure & tolerance

Fault prevention, avoidance, removal, …

and / or

☞ **Fault tolerance**

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.

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

7

Architectures

Uwe R. Zimmer - The Australian National University

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/

References for this chapter

[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/

In this chapter

- Hardware architectures:
    - From simple logic to multi-core CPUs
    - Concurrency on different levels
- Operating system architectures:
    - What is an operating system?
    - OS architectures
- Language architectures

Architectures

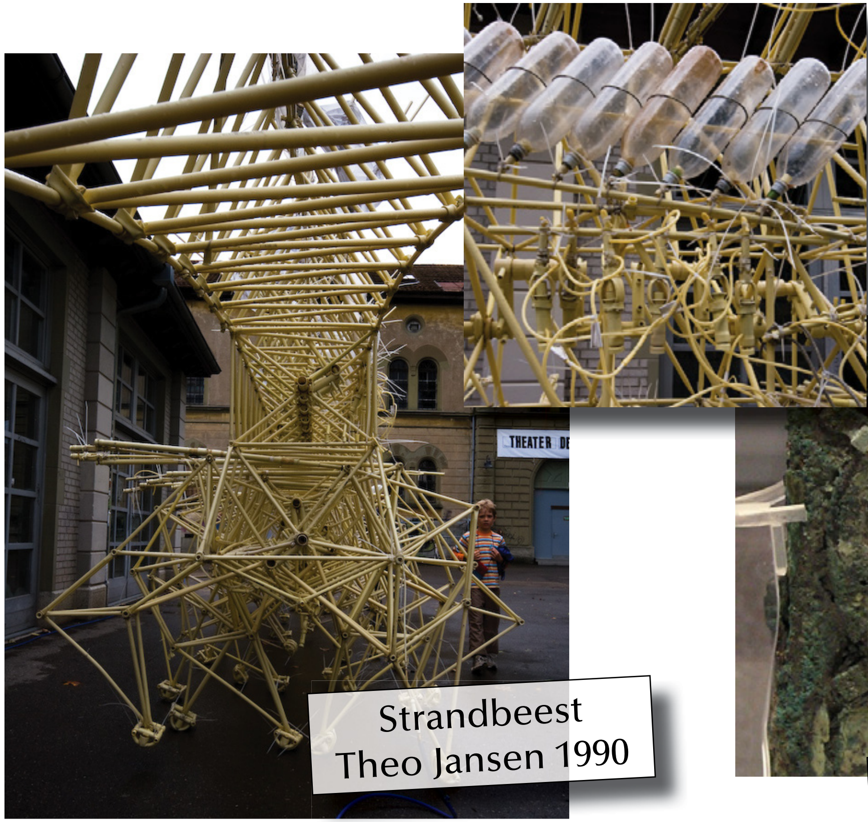
Layers of abstraction

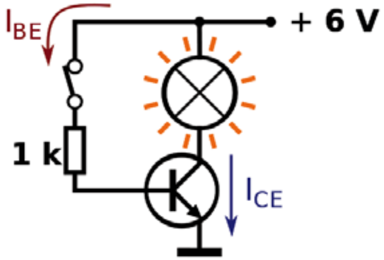
Layer	Form of concurrency
Application level (user interface, specific functionality...)	Distributed systems, servers, web services, "multitasking" (popular understanding)
Language level (data types, tasks, classes, API, ...)	Process libraries, tasks/threads (language), synchronisation, message passing,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

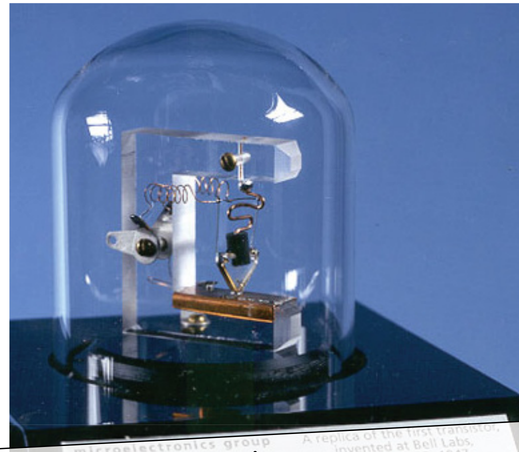
Logic - the basic building blocks

Controllable Switches & Ratios

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



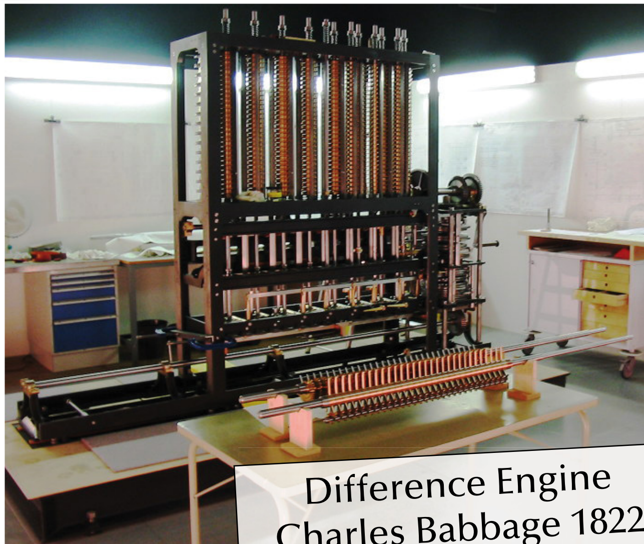




First transistor
John Bardeen and Walter Brattain 1947

Strandbeest
Theo Jansen 1990

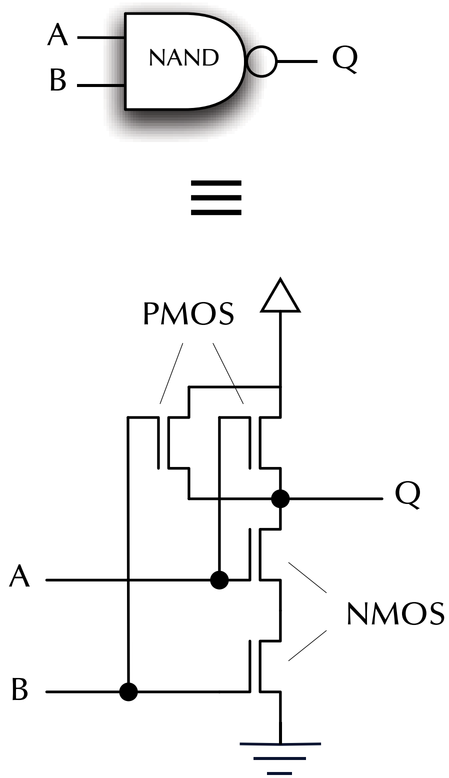
Antikythera Mechanism
Greek 150-100 BC
Credit: Wikipedia



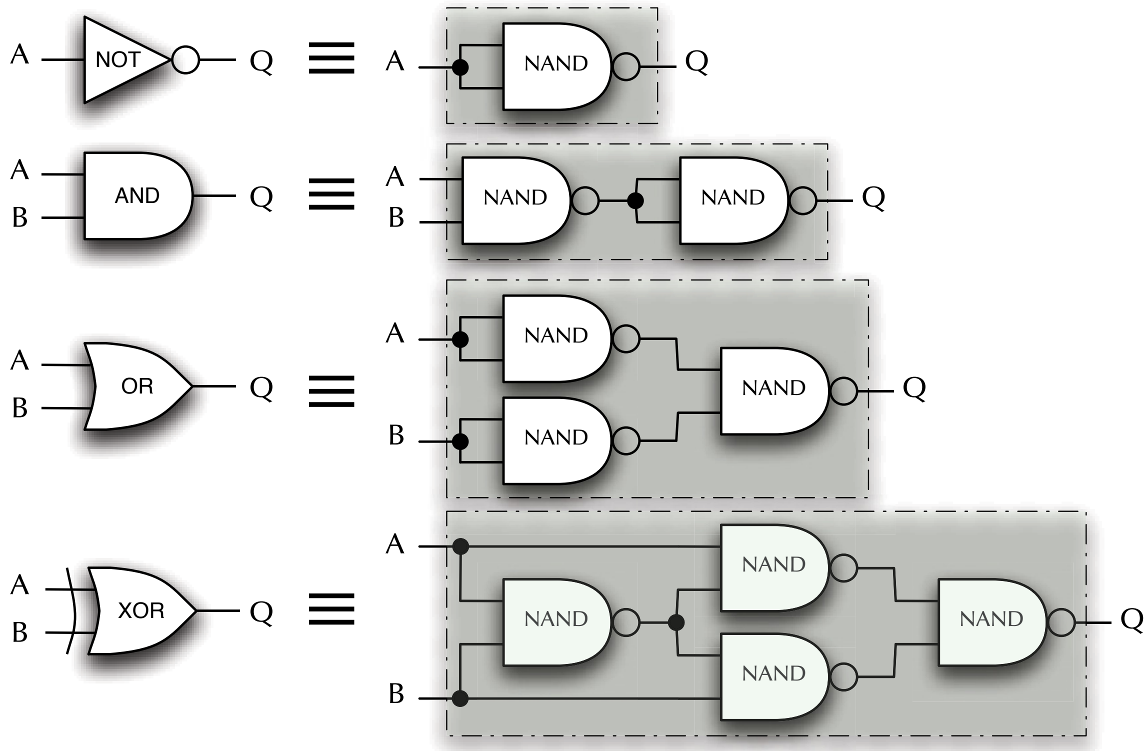
Difference Engine
Charles Babbage 1822

Logic - the basic building blocks

Constructing logic gates – for instance NAND in CMOS:

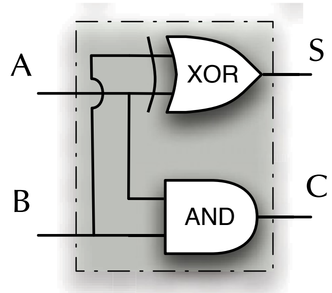


… and subsequently all other logic gates:

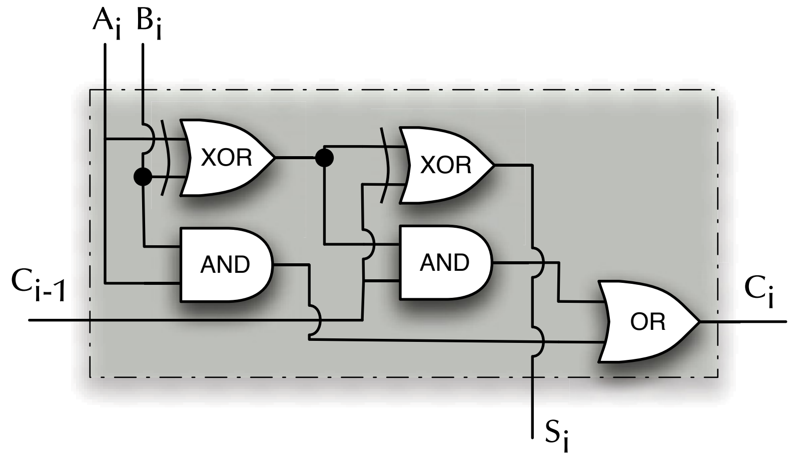


Logic - the basic building blocks

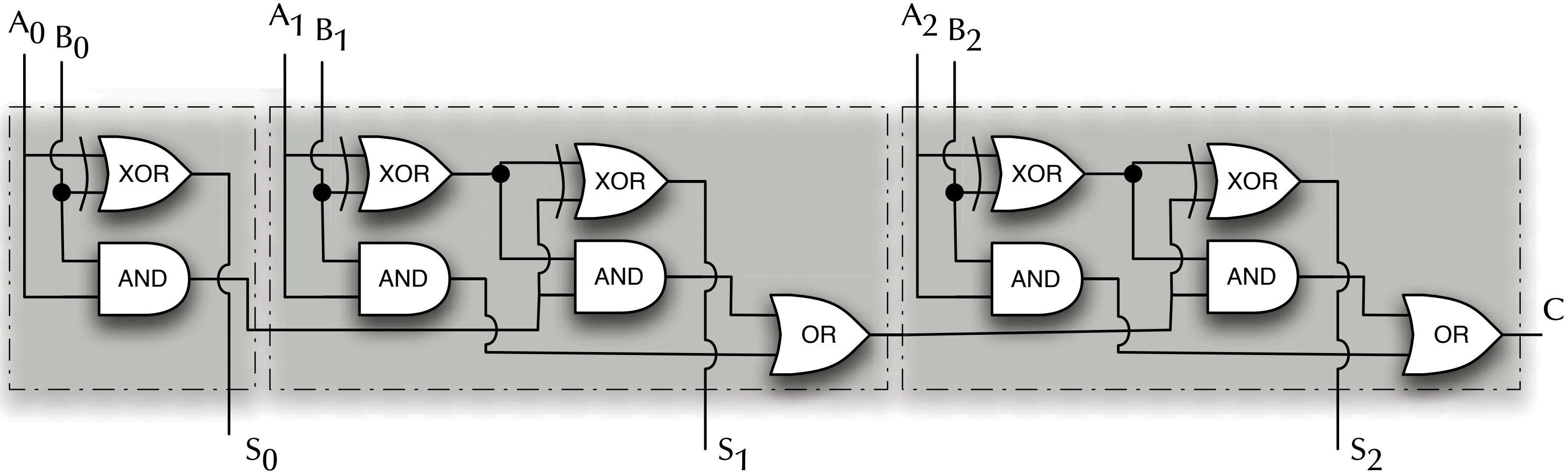
Half adder:



Full adder:

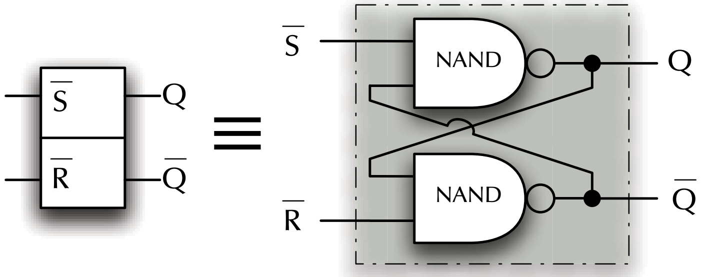


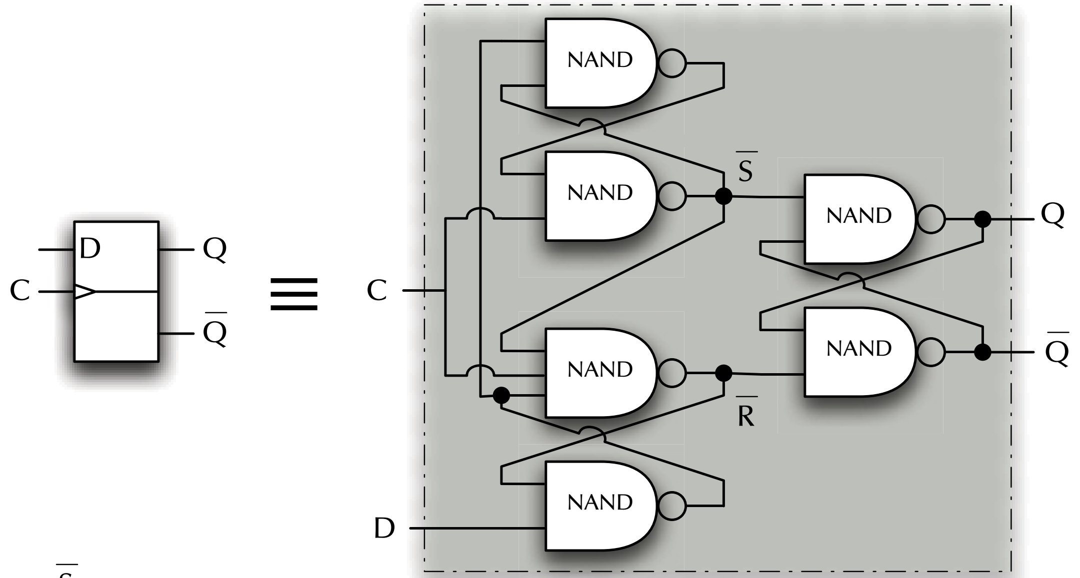
Ripple carry adder:

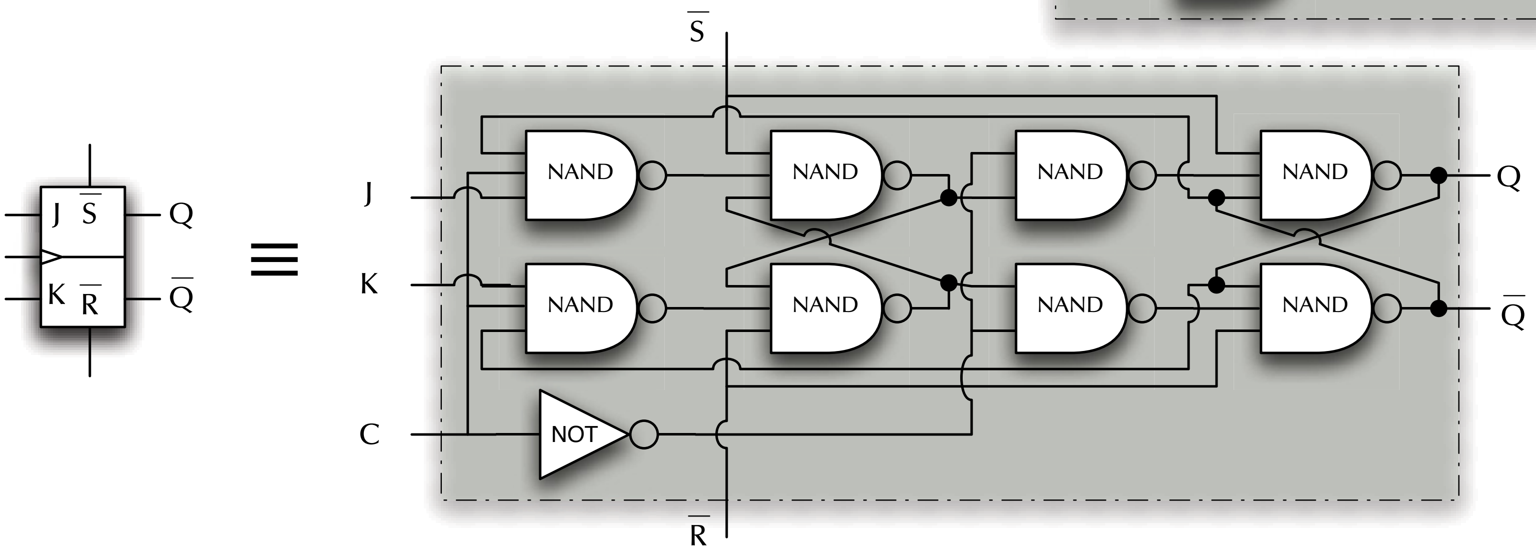


Logic - the basic building blocks

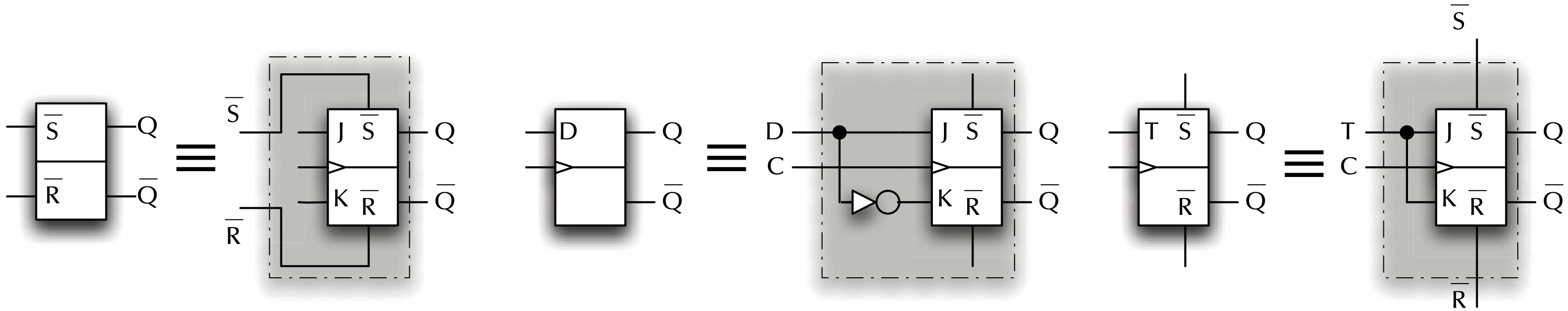
Basic Flip-Flops



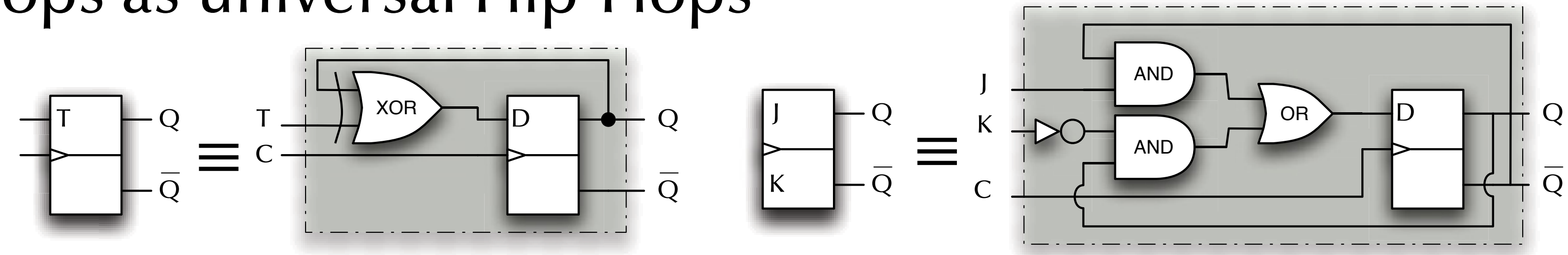




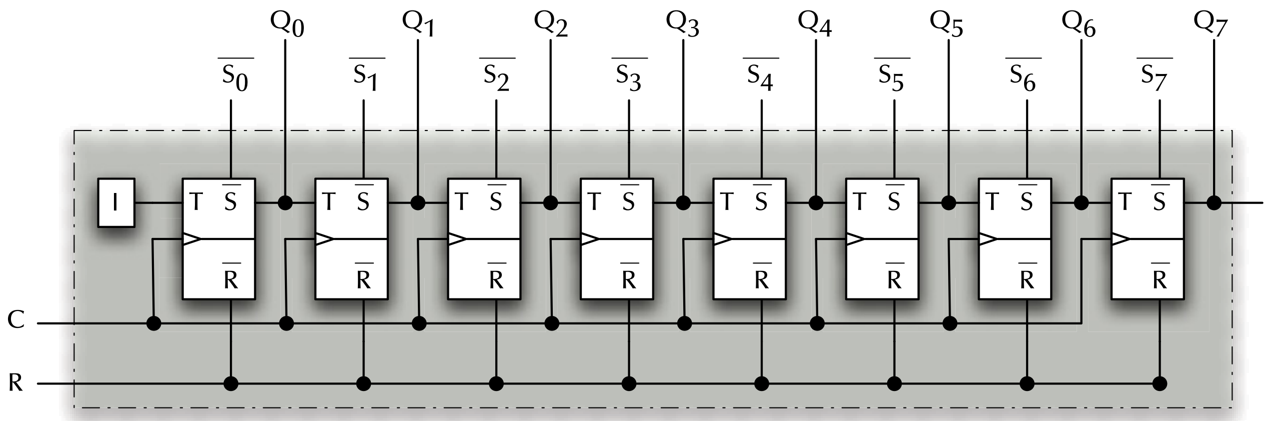
Logic - the basic building blocks



JK- and D- Flip-Flops as universal Flip-Flops

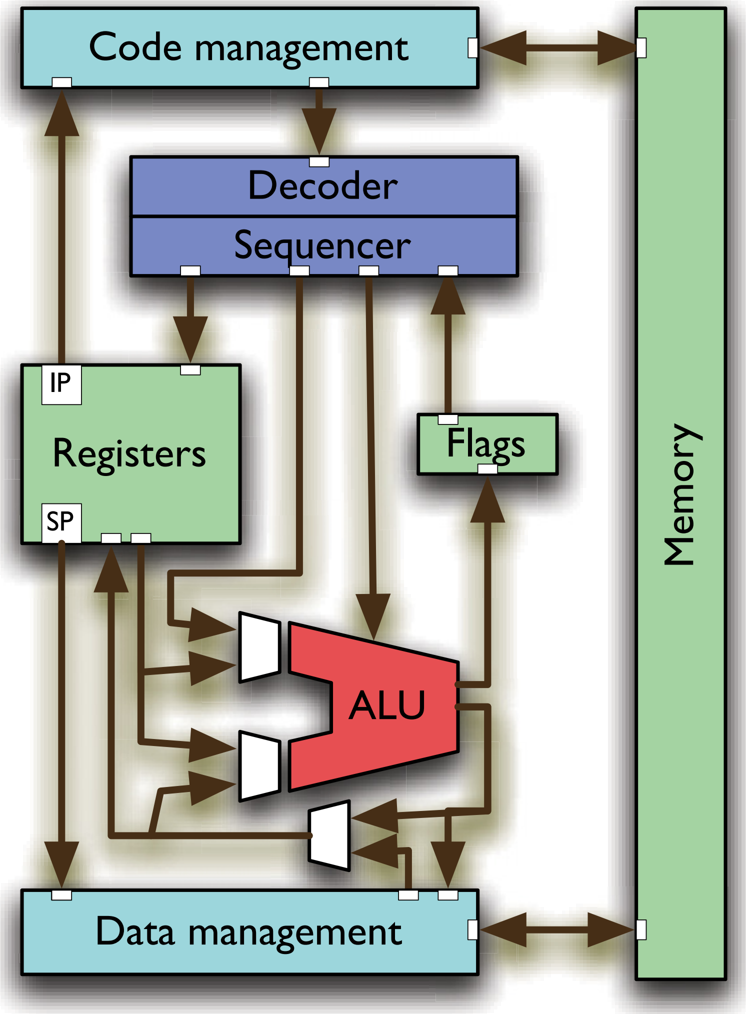


Counting register:



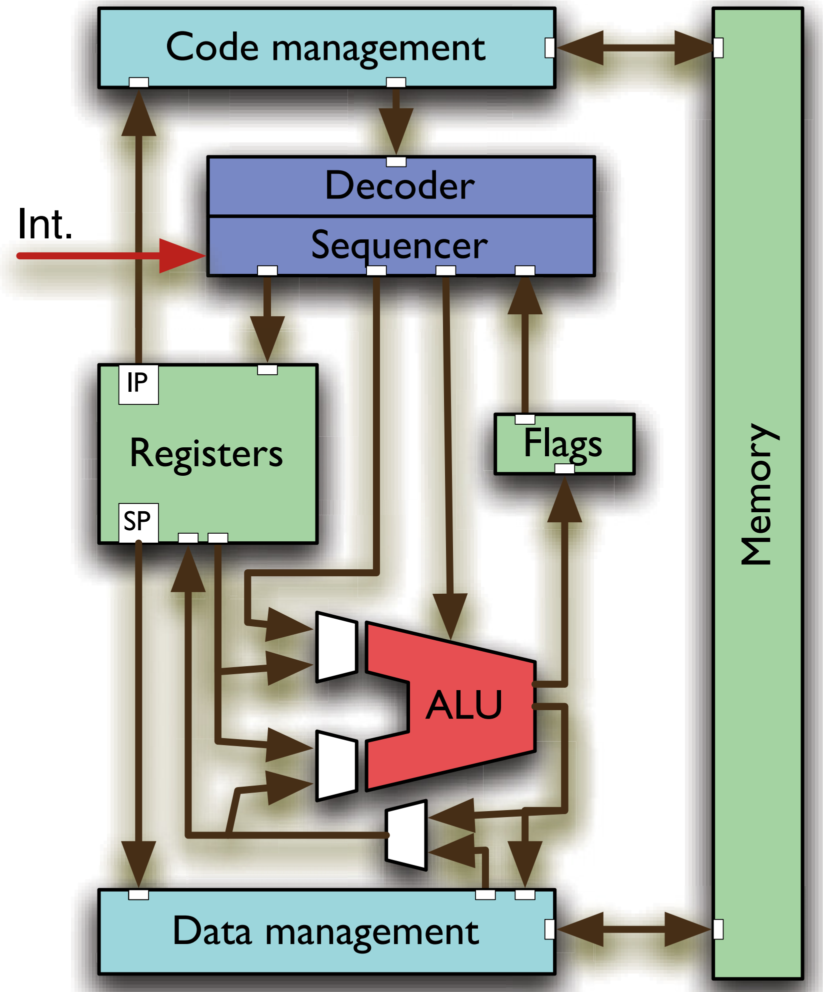
Processor Architectures

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

Processor Architectures



Interrupts

- One or multiple lines wired directly into the sequencer

☞ Precondition for:

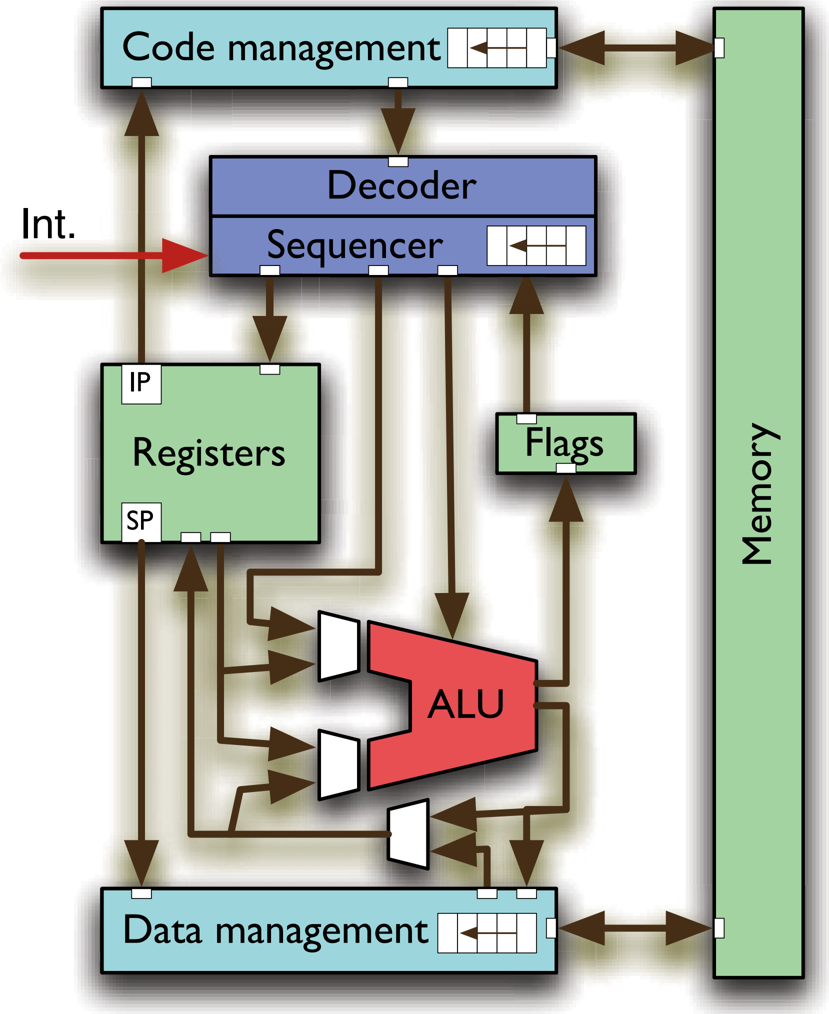
  - Pre-emption, timer driven actions, transient hardware interaction, …

☞ Usually preceded by an external logic ("interrupt controller") which accumulates and encodes all external requests.

On interrupt:

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

Processor Architectures



Pipeline

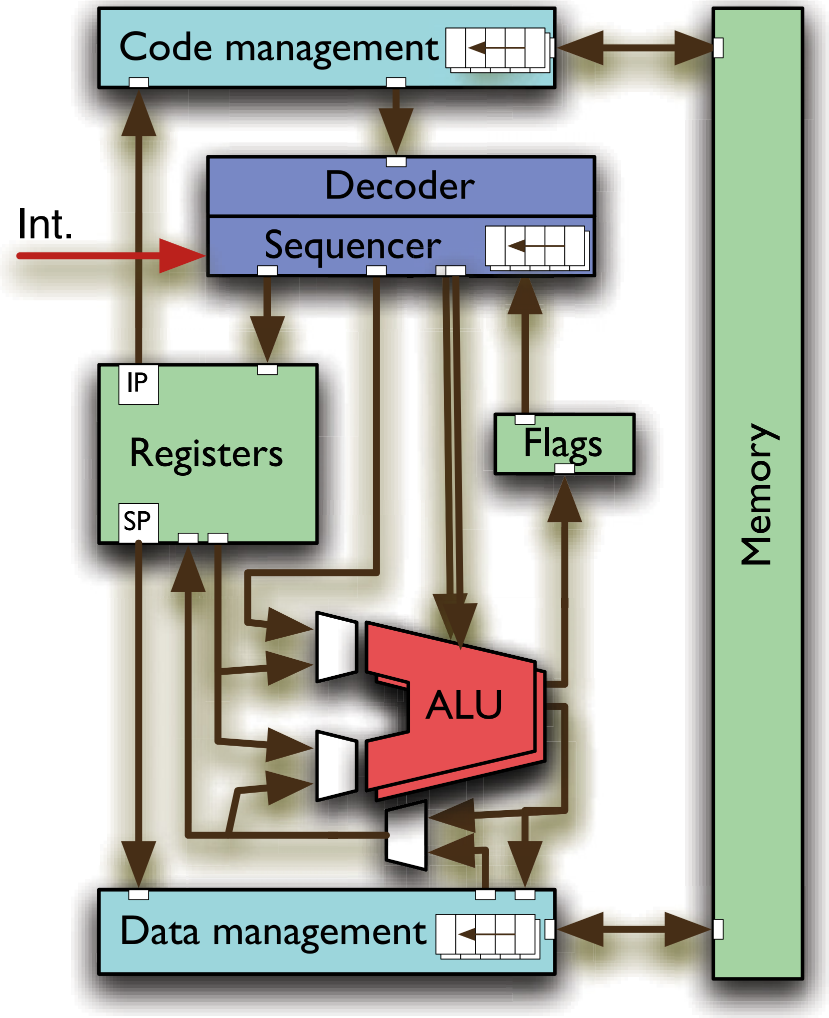
Some CPU actions are naturally sequential (e.g. instructions need to be first loaded, then decoded before they can be executed).

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

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

Processor Architectures

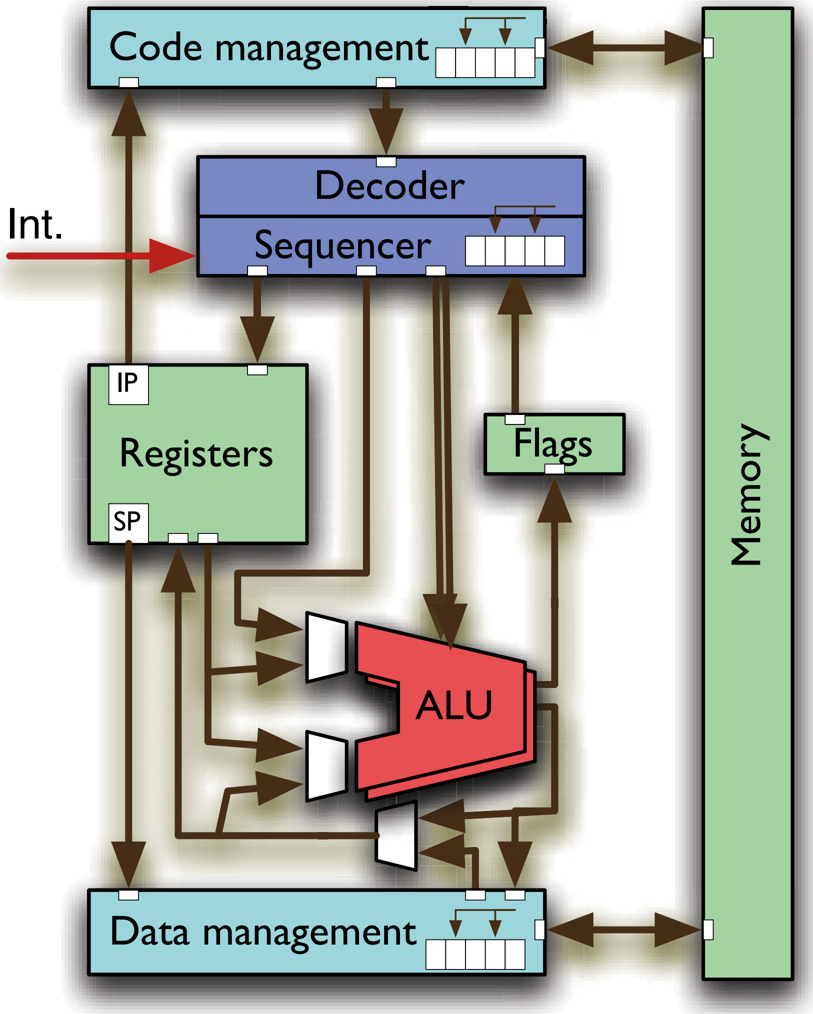
Parallel pipelines



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

- (Conditional) branches might again break the pipelines.
- Cross-dependencies might limit the degree of concurrency.
- Same latency, yet even higher throughput.
- This hardware might require code optimization to be fully utilized.

Processor Architectures

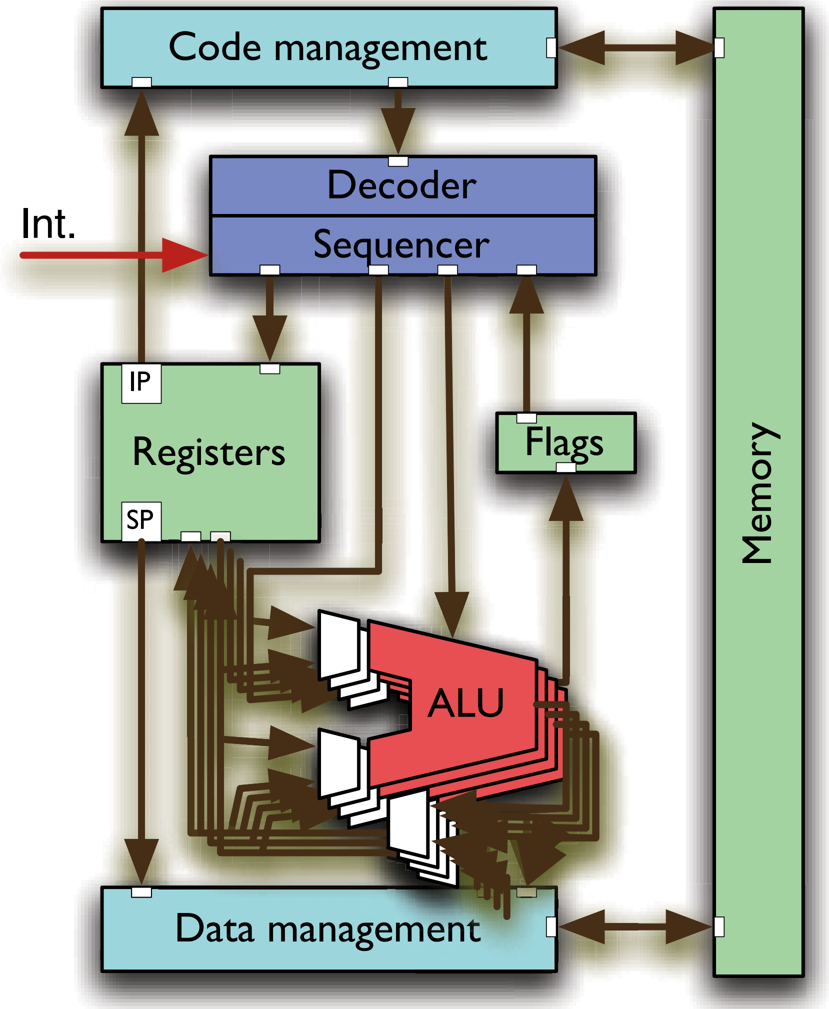


Out of order execution

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

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

Processor Architectures



SIMD ALU units

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

Examples: Altivec, MMX, SSE[2|3|4], …

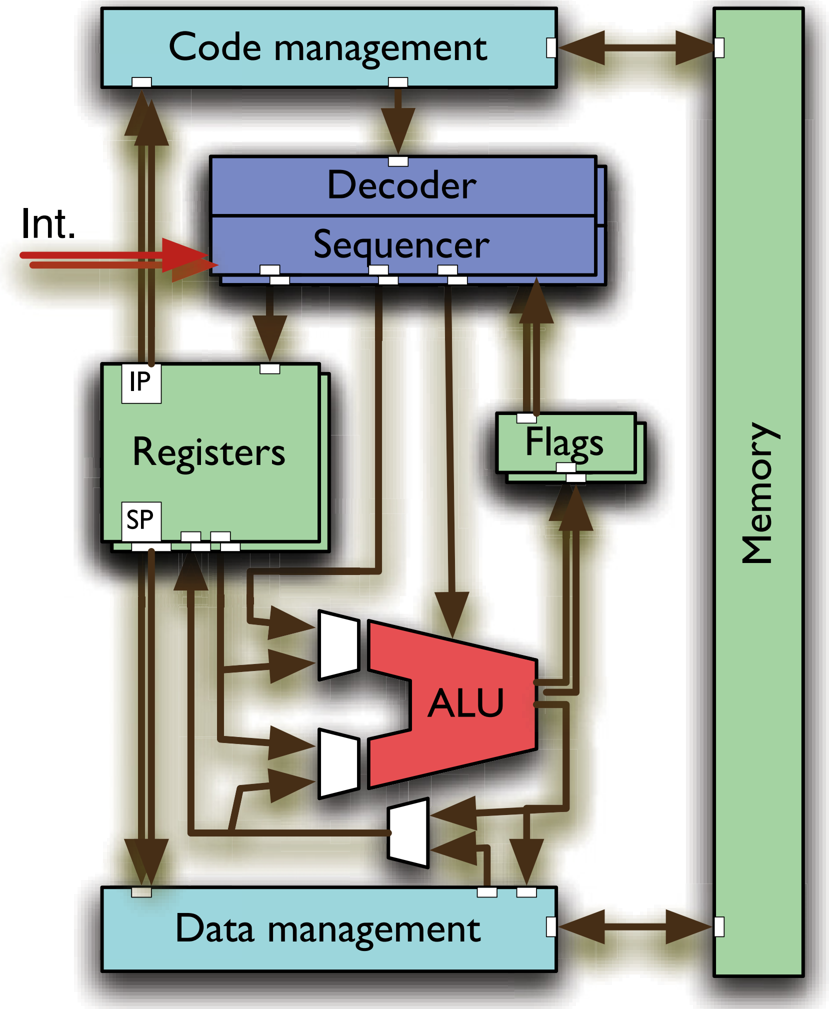
GPU processing

Employs an external “graphics processor” as a vector unit.

☞ Requires specialized compilers.

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

Processor Architectures



Hyper-threading

Emulates multiple virtual CPU cores by means of replication of:

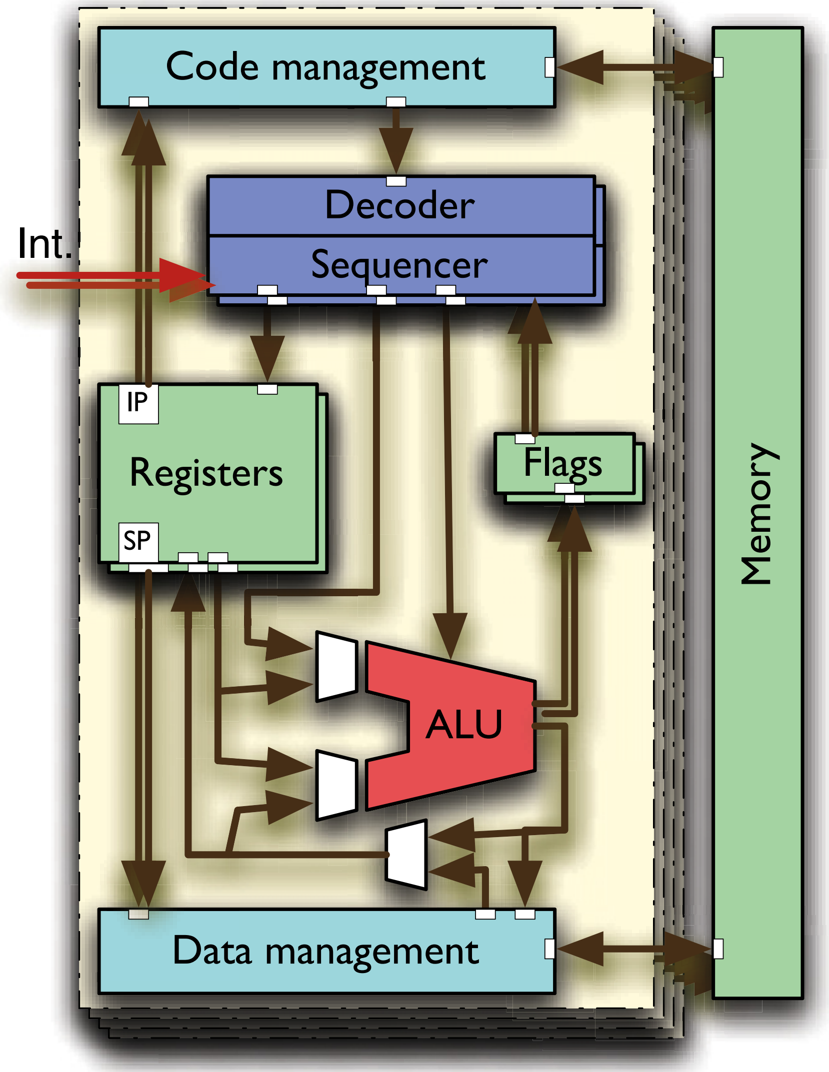
- Register sets
- Sequencer
- Flags
- Interrupt logic

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

☞ **Requires concurrency (processes or threads) on the program level.**

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

Processor Architectures



Multi-core CPUs

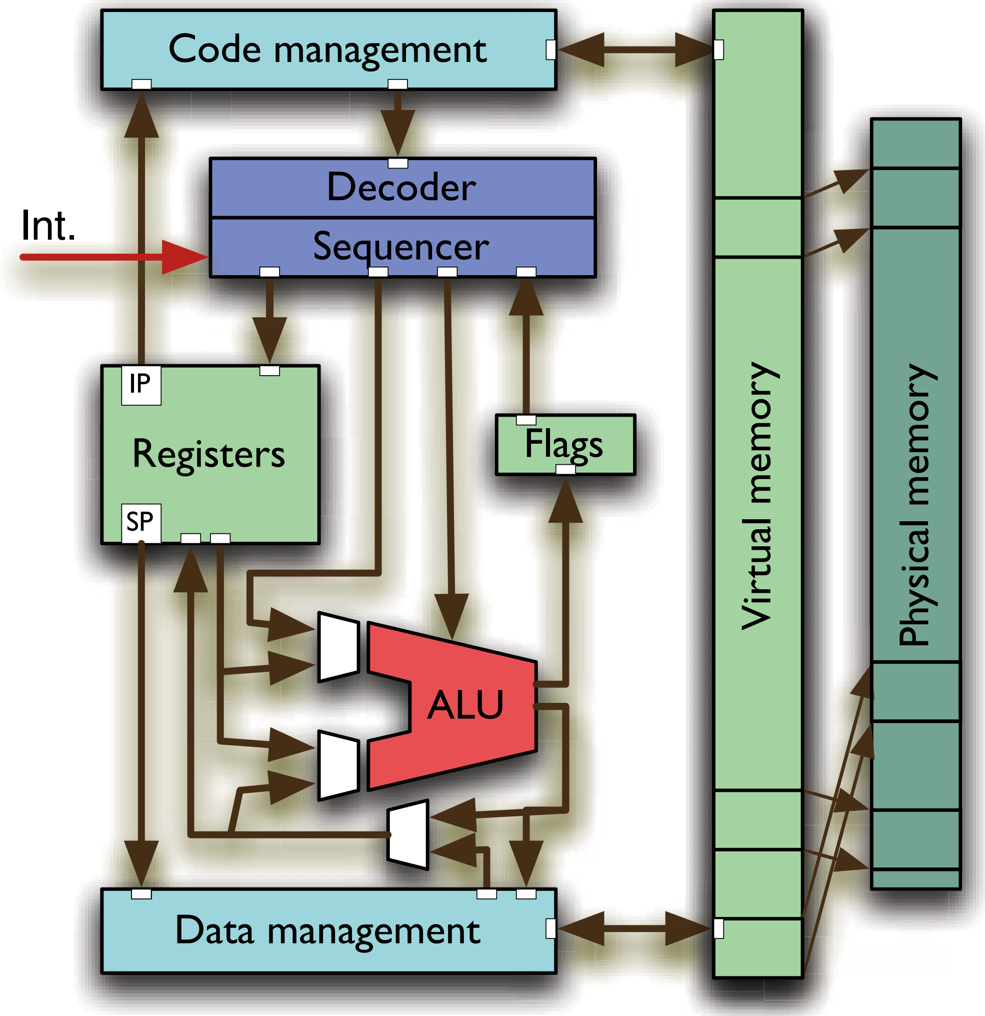
Full replication of multiple CPU cores on the same chip package.

- 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 concurrency (processes or threads) on the program level.**

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

Processor Architectures



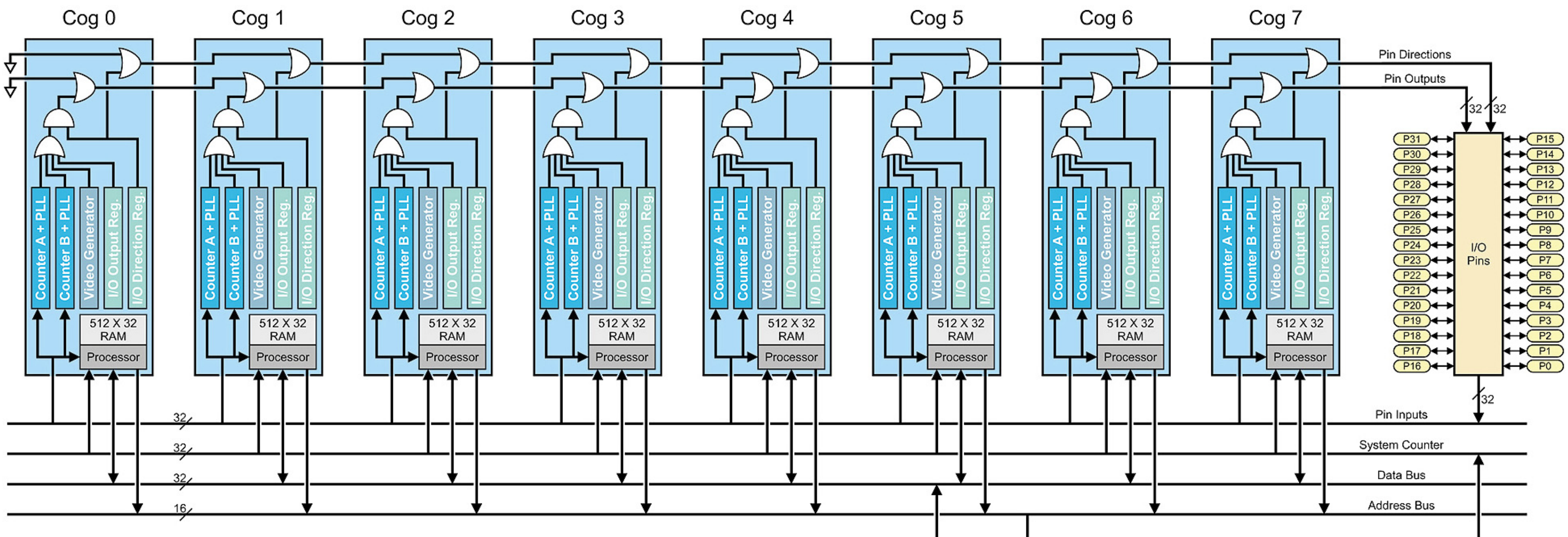
Virtual memory

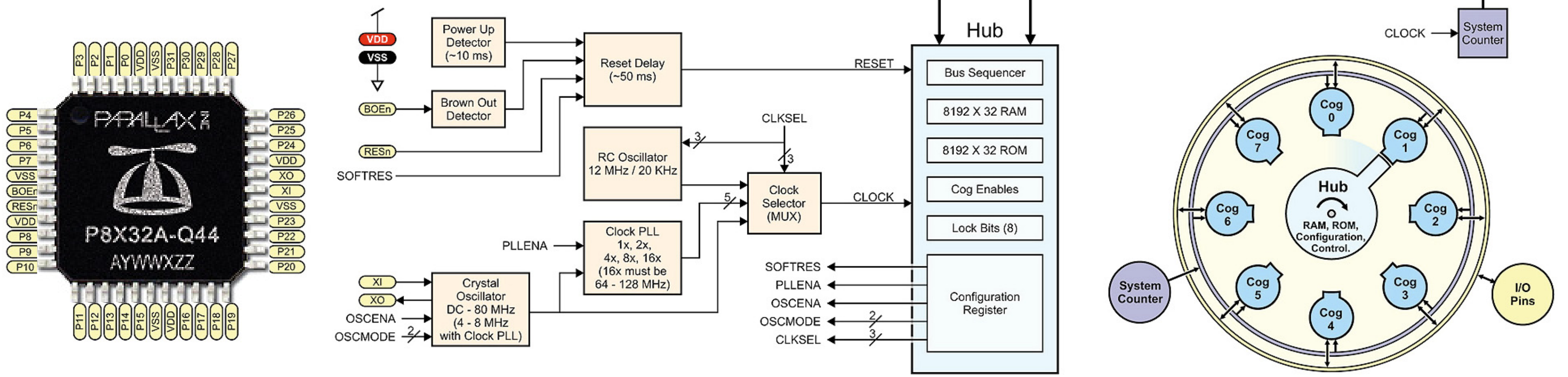
Translates logical memory addresses
into physical memory addresses
and provides memory protection features.

- Does not introduce concurrency by itself.

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

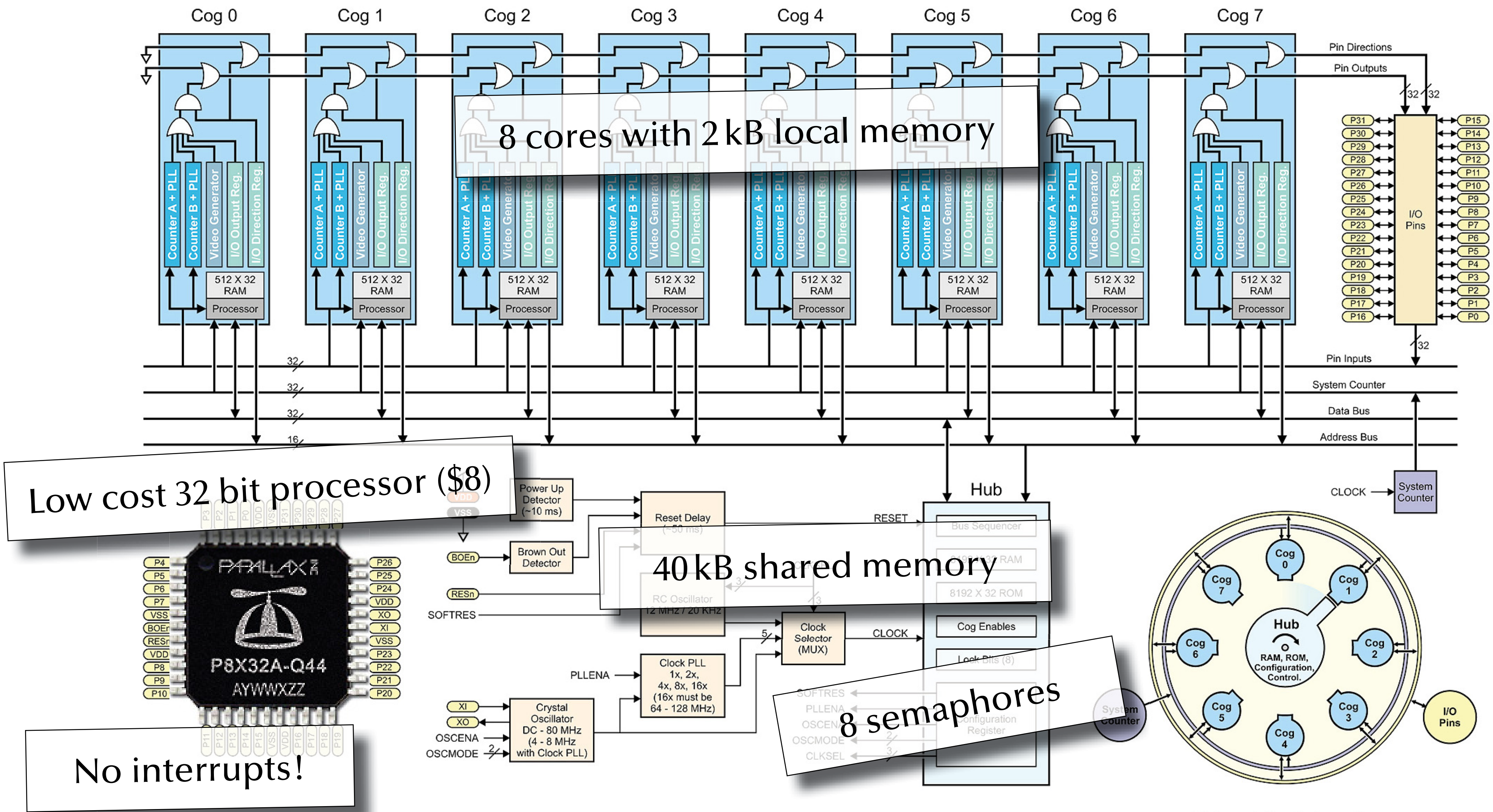
Alternative Processor Architectures: Parallax Propeller





Hub and Cog Interaction

Alternative Processor Architectures: Parallax Propeller (2006)



Hub and Cog Interaction

8 cores with 2 kB local memory

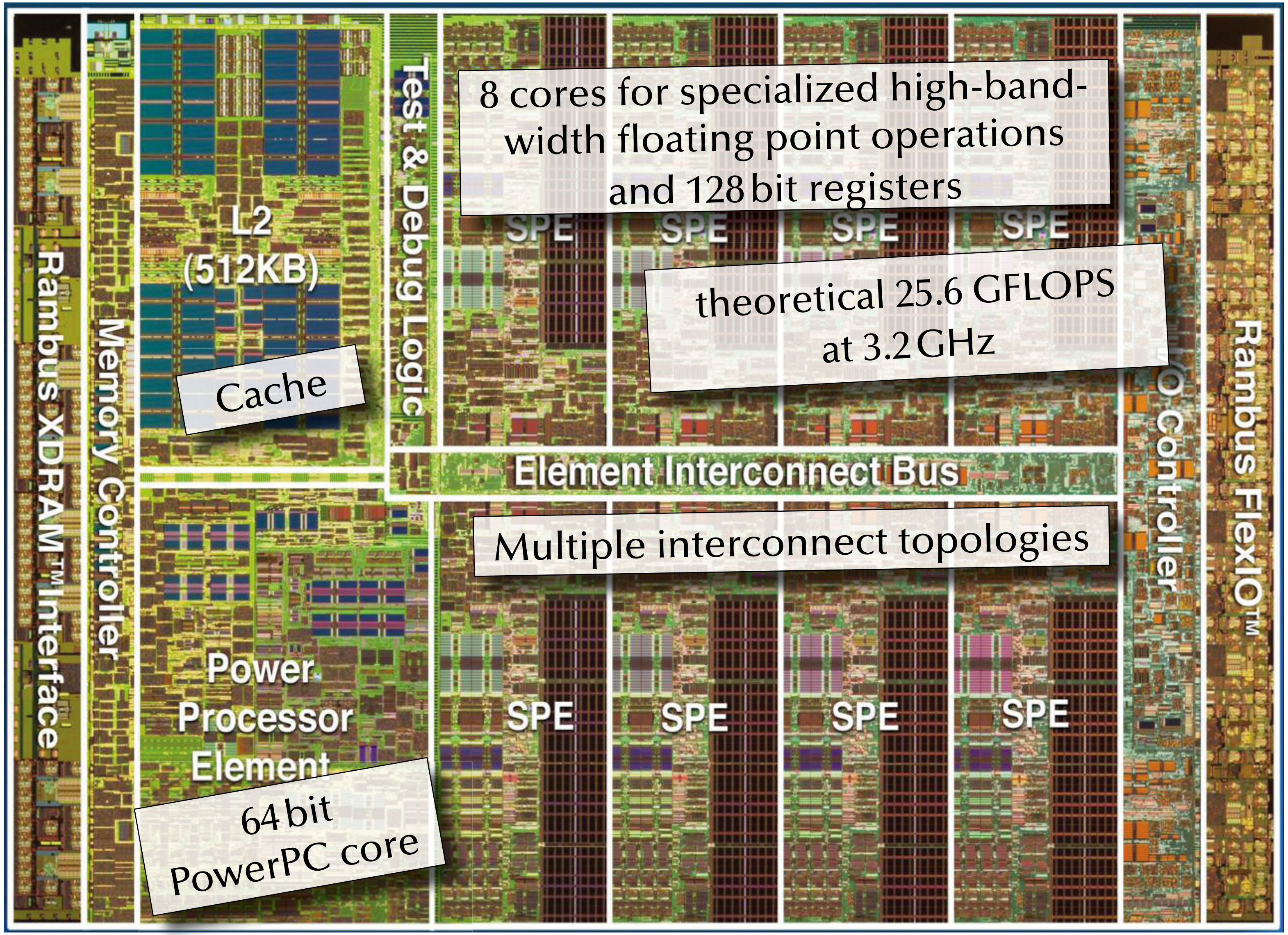
Low cost 32 bit processor ($8)

40 kB shared memory

8 semaphores

No interrupts!

Alternative Processor Architectures: IBM Cell processor (2001)



Multi-CPU systems

Scaling up:

- Multi-CPU on the same memory

  multiple CPUs on same motherboard and memory bus, e.g. servers, workstations

- Multi-CPU with high-speed interconnects

  various supercomputer architectures, e.g. Cray XE6:

  - 12-core AMD Opteron, up to 192 per cabinet (2304 cores)
  - 3D torus interconnect (160 GB/sec capacity, 48 ports per node)

- Cluster computer (Multi-CPU over network)

  multiple computers connected by network interface,

  e.g. Sun Constellation Cluster at ANU:

  - 1492 nodes, each: 2x Quad core Intel Nehalem, 24 GB RAM
  - QDR Infiniband network, 2.6 GB/sec

Operating Systems

What is an operating system?

What is an operating system?

1. A virtual machine!

... offering a more comfortable, more flexible and safer environment

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

What is an operating system?

1. A virtual machine!

... offering a more comfortable, more flexible and safer environment

Typ. general OS   Typ. real-time system   Typ. embedded system

What is an operating system?

2. A resource manager!

... coordinating access to hardware resources

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!

The evolution of operating systems

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

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 come with:

- High speed network connectors (e.g. GB-ethernet)
- Wireless LAN (e.g. IEEE802.11g)
- Local device bus-system (e.g. Firewire 800 or USB 3.0)
- Wireless local device network (e.g. Bluetooth)
- Infrared communication (e.g. IrDA)
- Modem/ADSL

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:

- 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.

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.

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)

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?

Types of current operating systems

Real-time operating systems

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

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

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!

What is an operating system?

Is there a standard set of features for operating systems?

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.

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?

What is an operating system?

Is there a standard set of features for operating systems?

☞ **no,**

the term ‘operating system’ covers 4 kB microkernels,

as well as > 1 GB installations of desktop general purpose operating systems.

Is there a minimal set of features?

☞ **almost,**

memory management, process management *and* ***inter-process communication/synchronisation***

will be considered essential in most systems

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?

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

Typical features of operating systems

Process management:

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

☞ context switch:

☞ needs to...

- ‘remove’ one process from the CPU while preserving its state
- choose another process (scheduling)
- ‘insert’ the new process into the CPU, restoring the CPU state

Some CPUs have hardware support for context switching, otherwise:

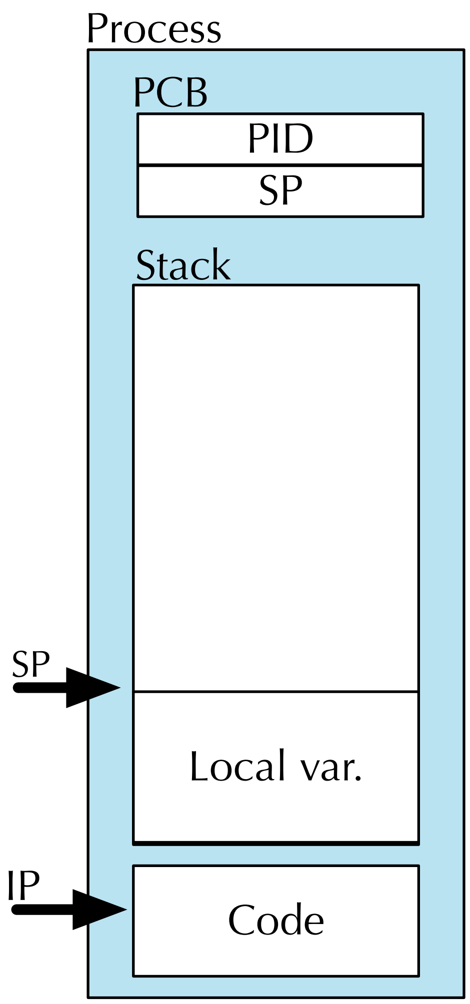
☞ use interrupt mechanism

Typical features: Context switch

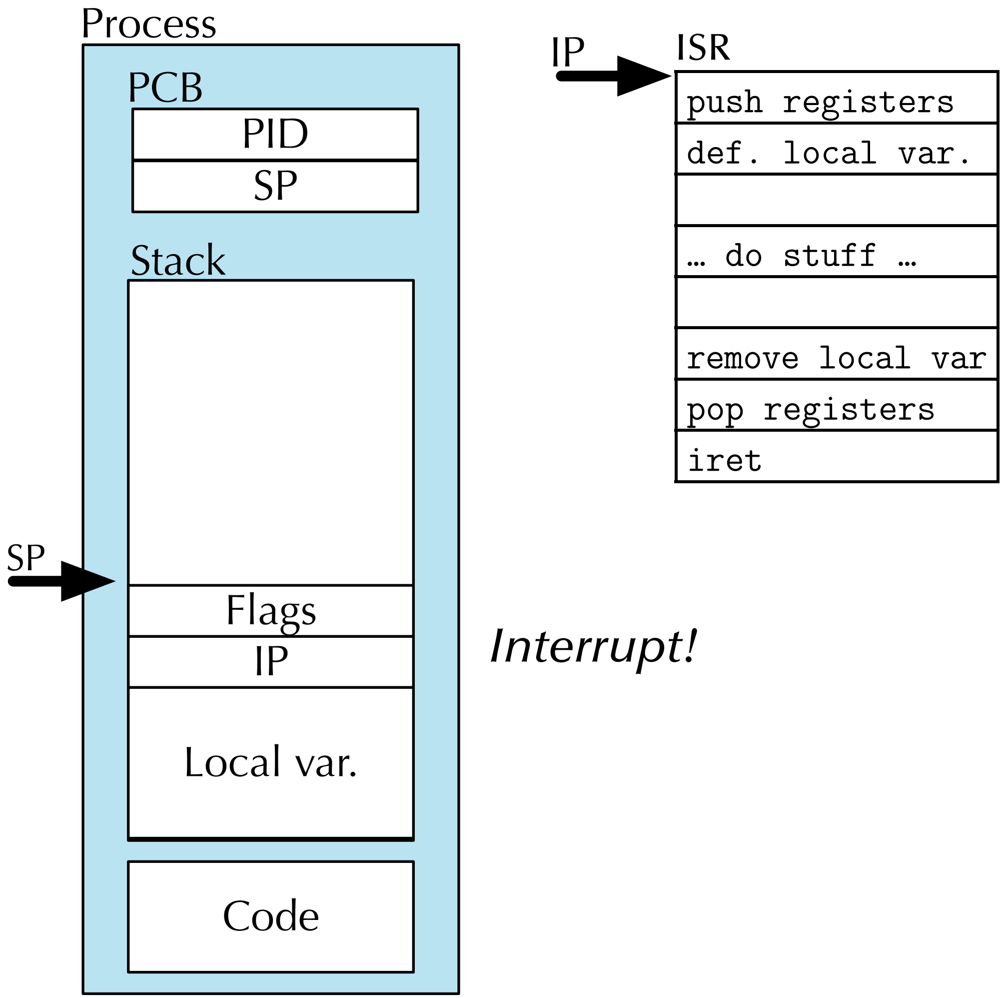
Interrupt mechanism: already stores IP and CPU state (flags) to stack, and restores it at the end (`iret`)

☞ How can we make it 'return' to a different process?

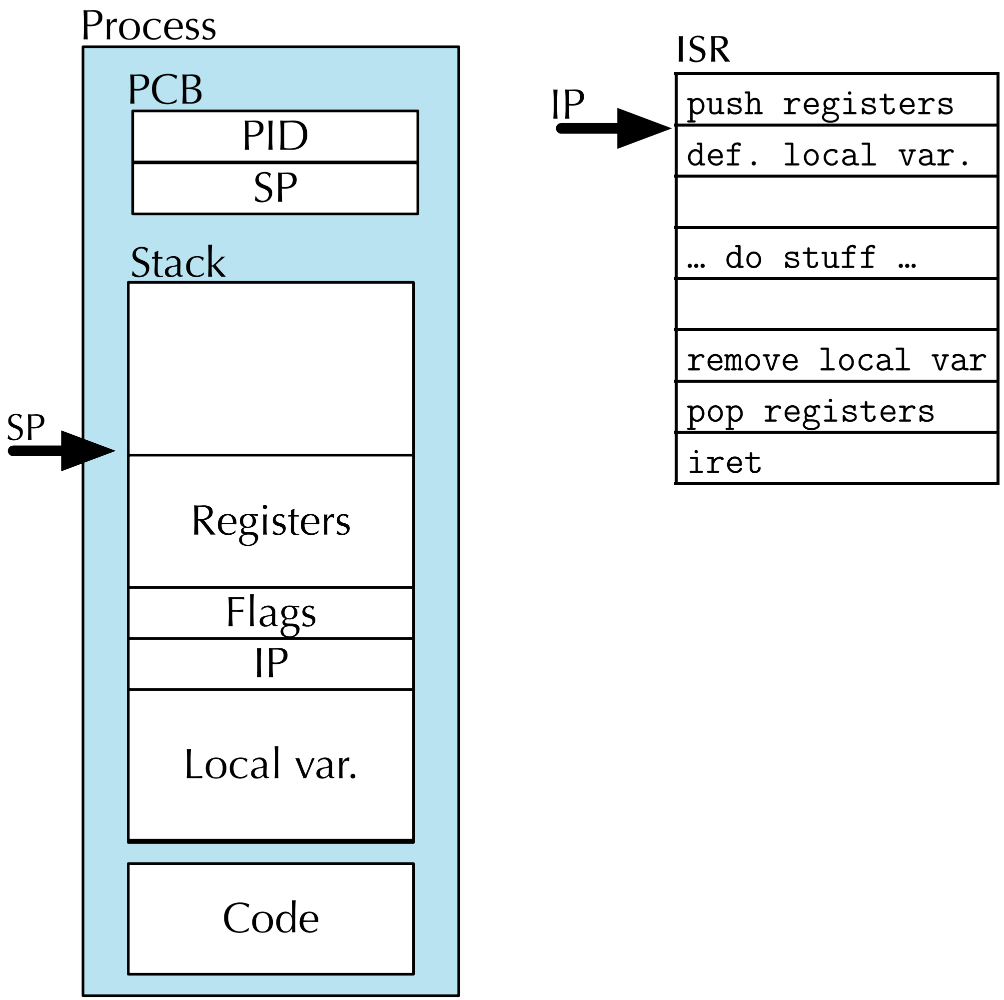
Typical features: Context switch



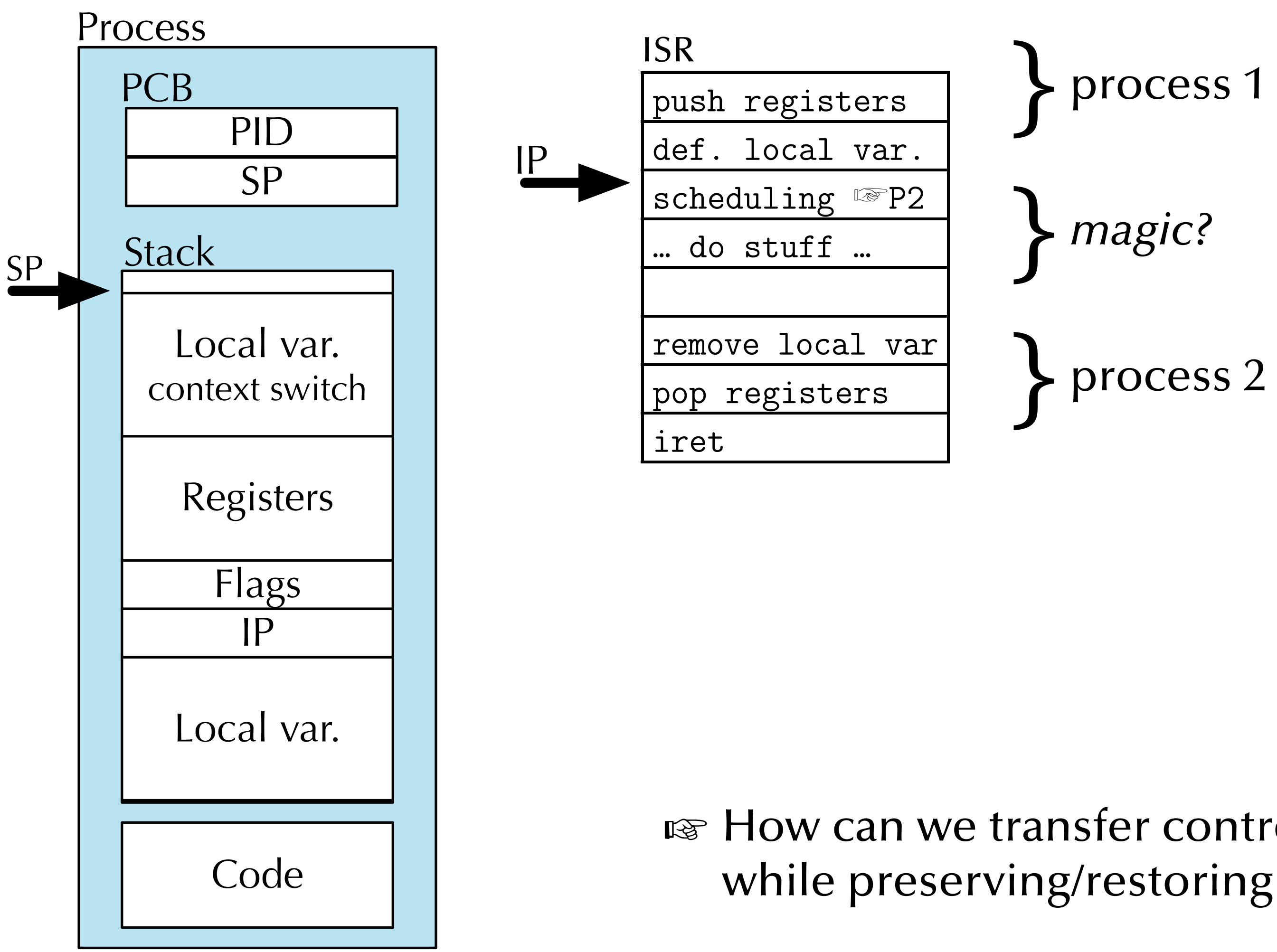
Typical features: Context switch



Typical features: Context switch

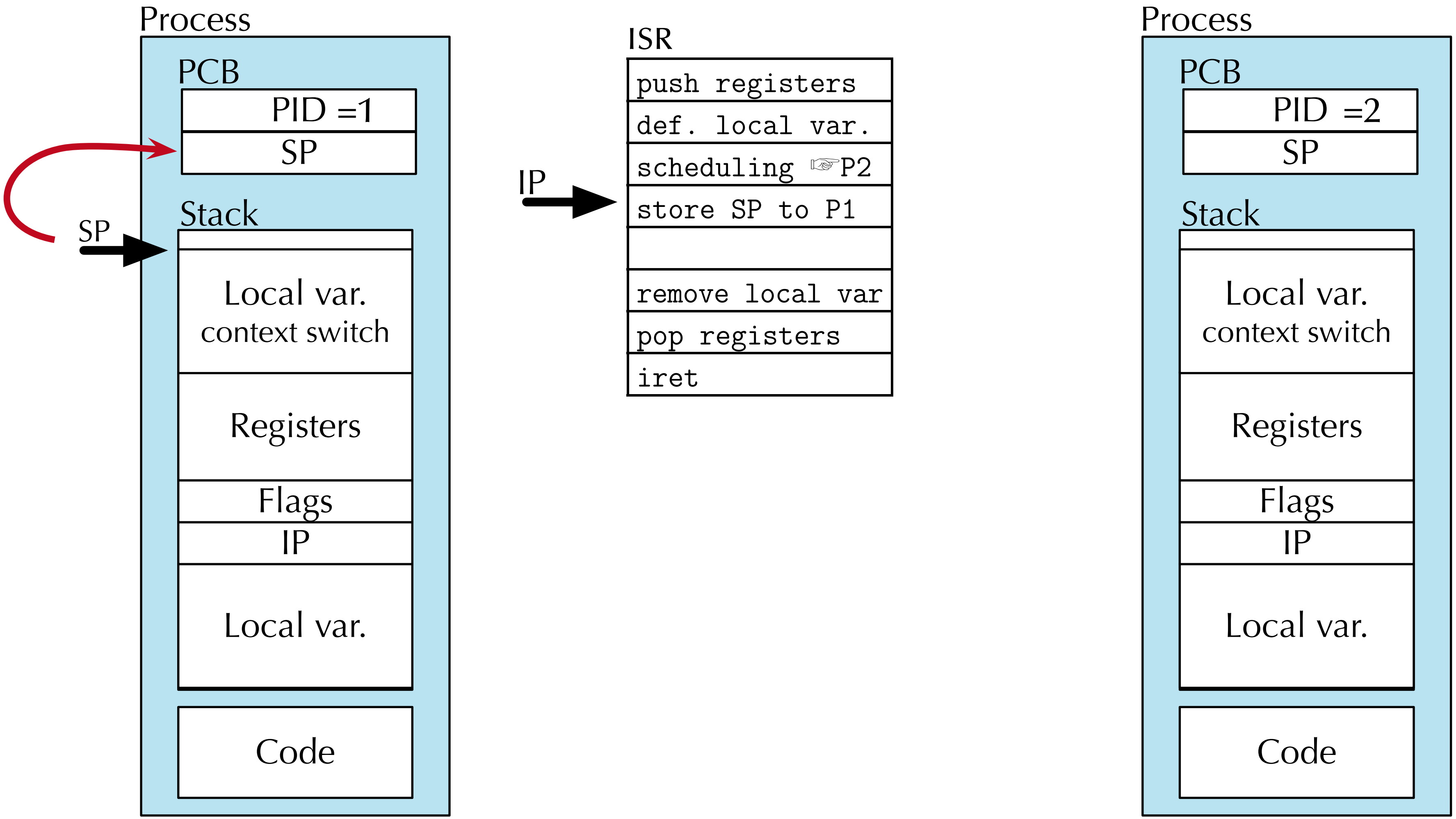


Typical features: Context switch

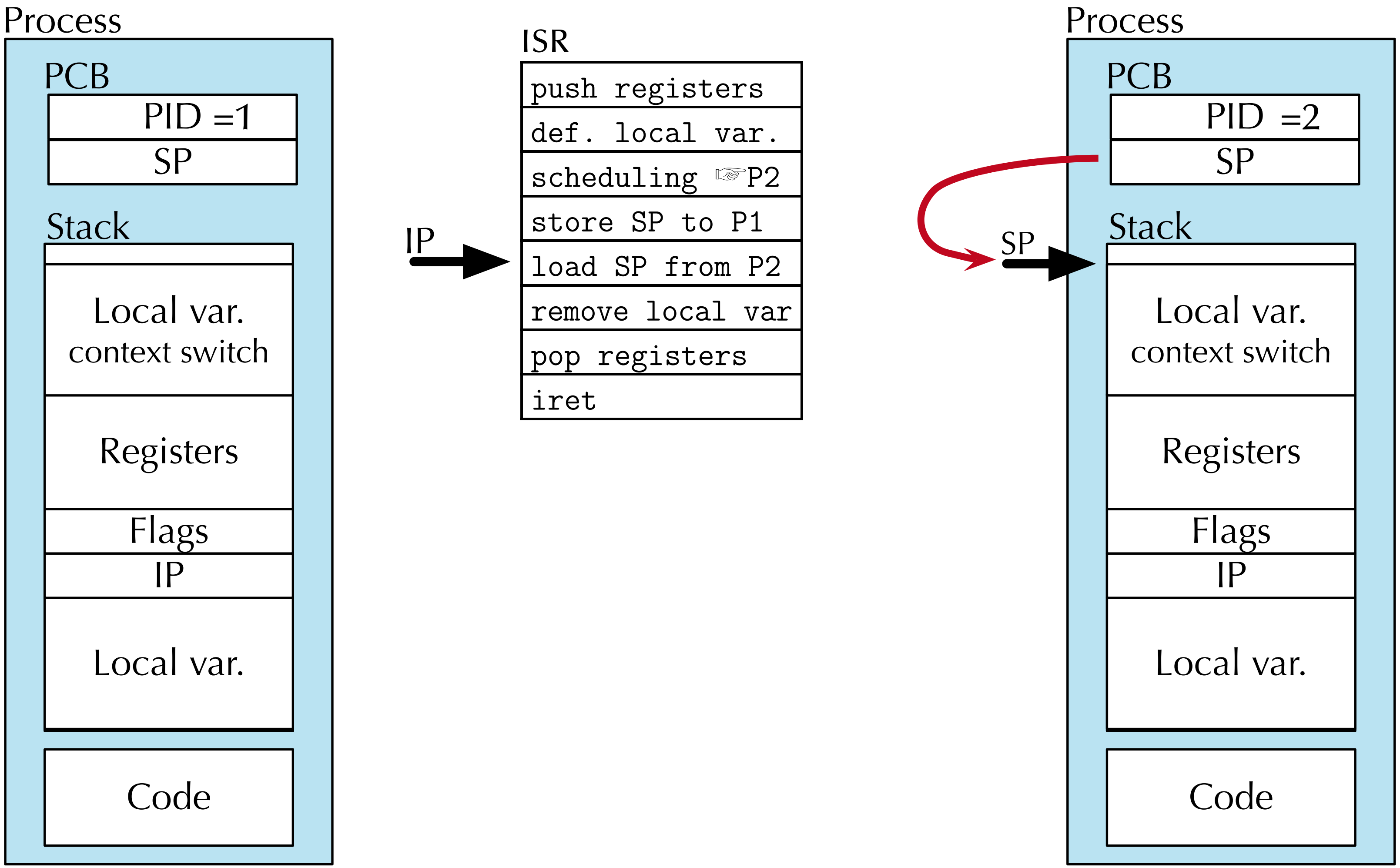


☞ How can we transfer control to another process while preserving/restoring all relevant states?

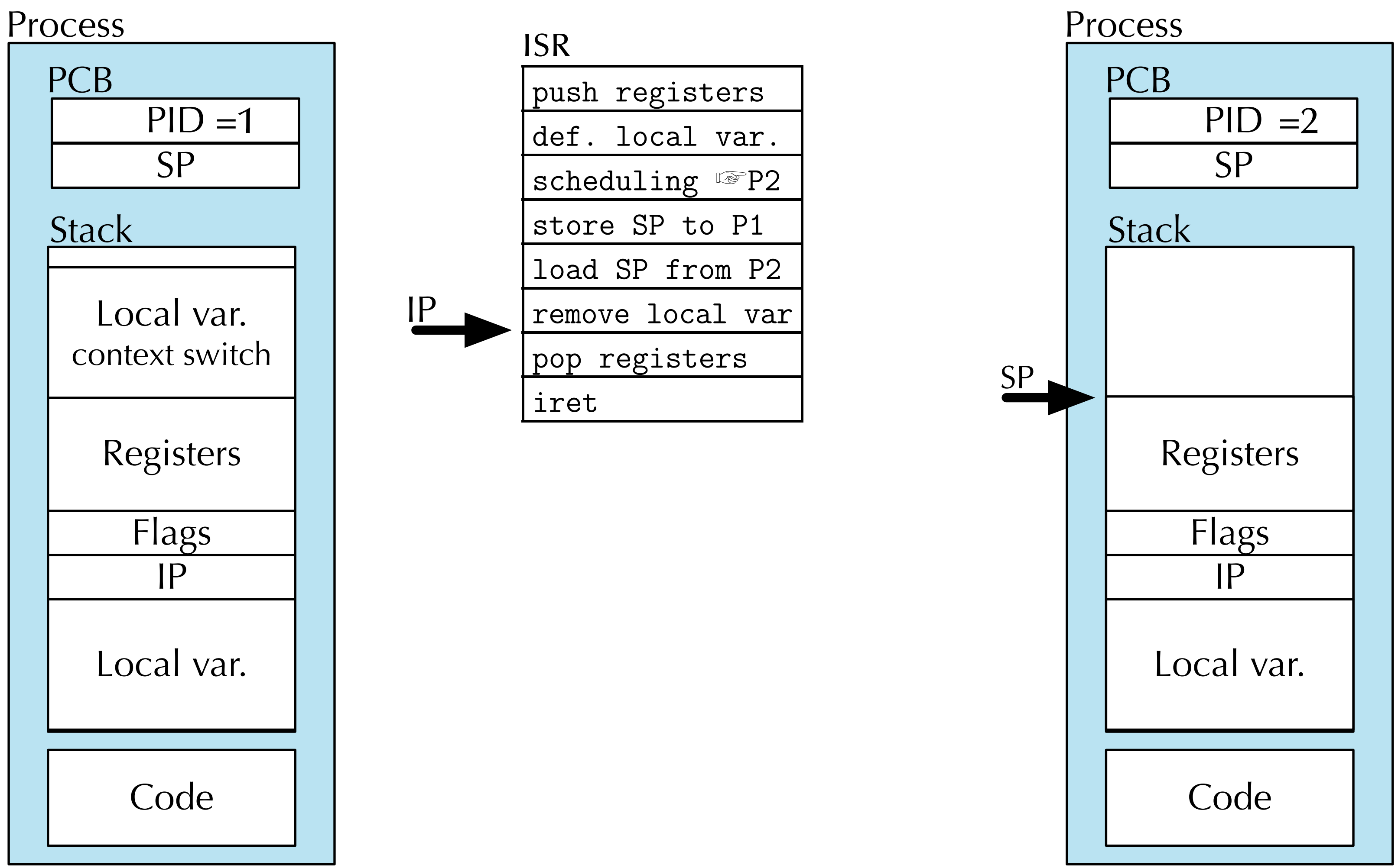
Typical features: Context switch



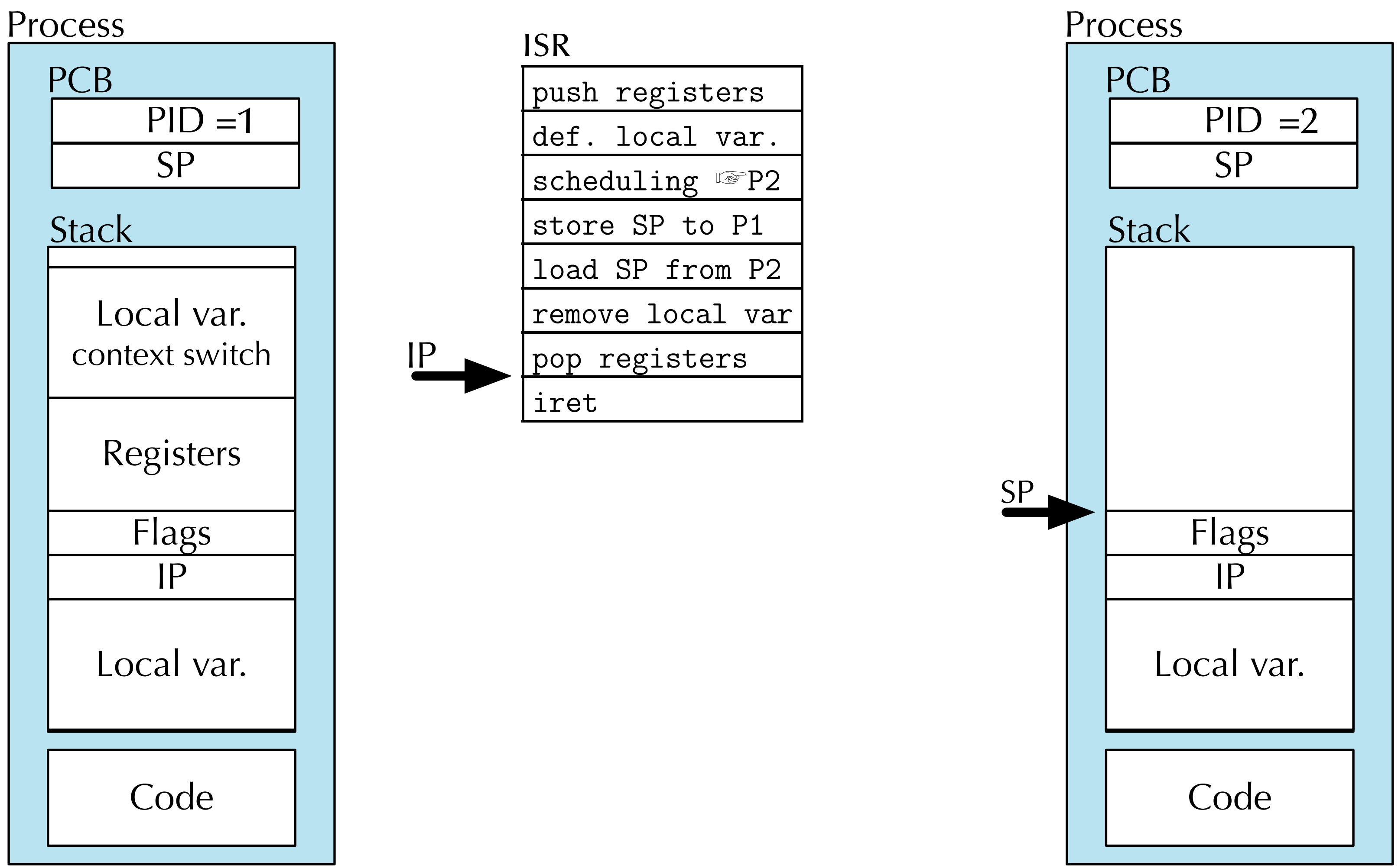
Typical features: Context switch



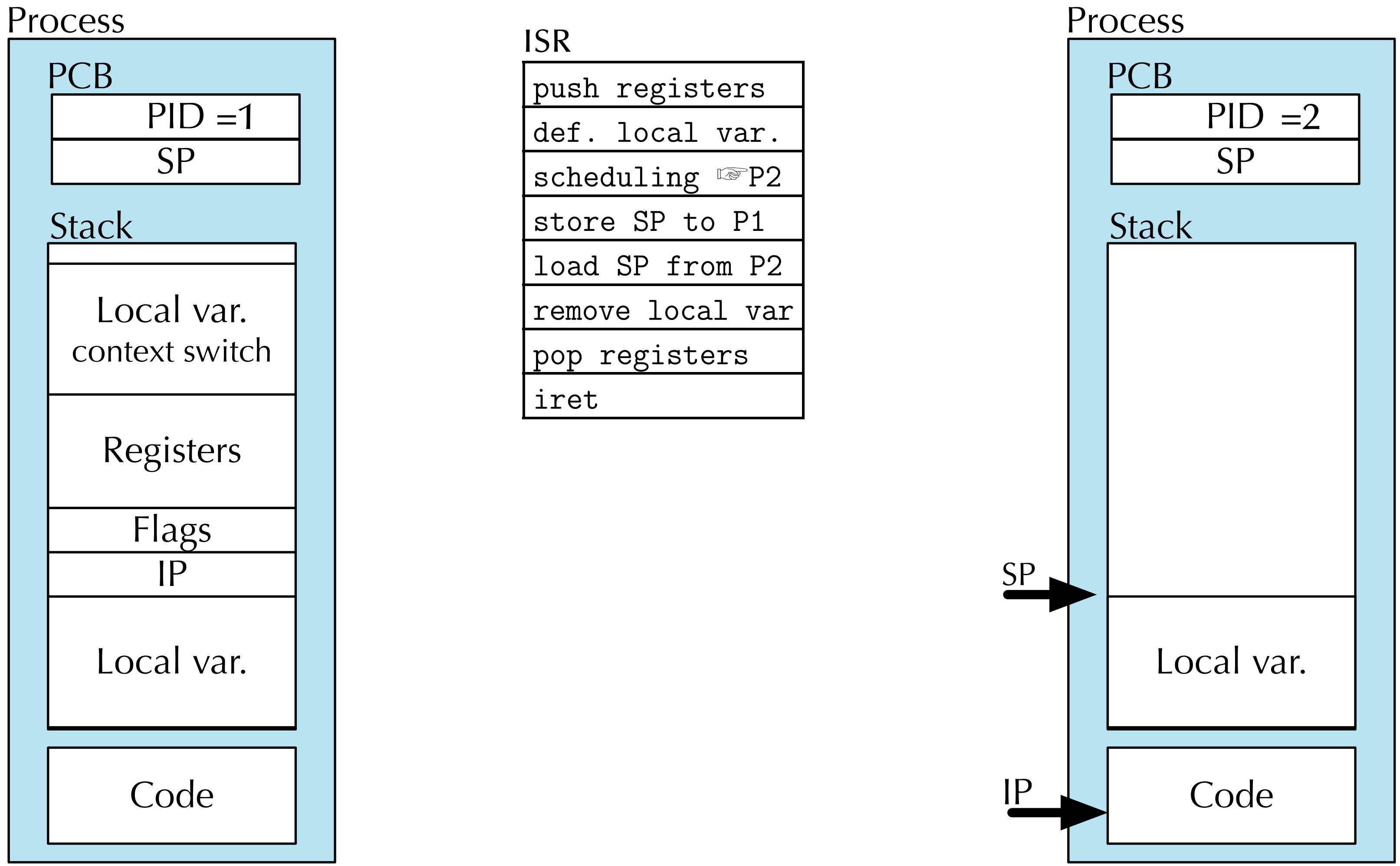
Typical features: Context switch



Typical features: Context switch



Typical features: Context switch



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...

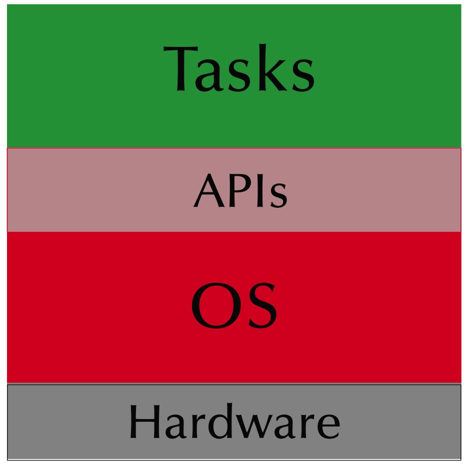
Typical structures of operating systems

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



Monolithic

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

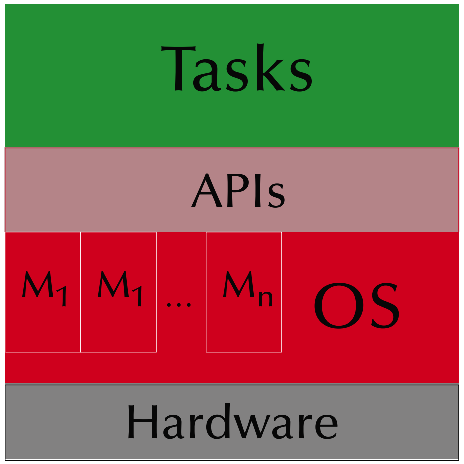
Typical structures of operating systems

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

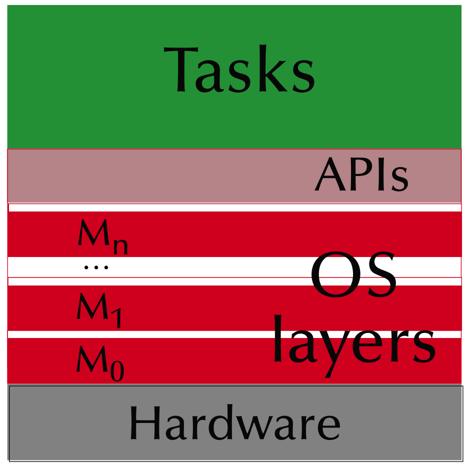


Modular

Typical structures of operating systems

Monolithic & layered

- easily portable
- significantly easier to maintain
- crashing layers do not necessarily stop the whole OS
- possibly reduced efficiency through many interfaces
- rigorous implementation of the stacked virtual machine perspective on OSs



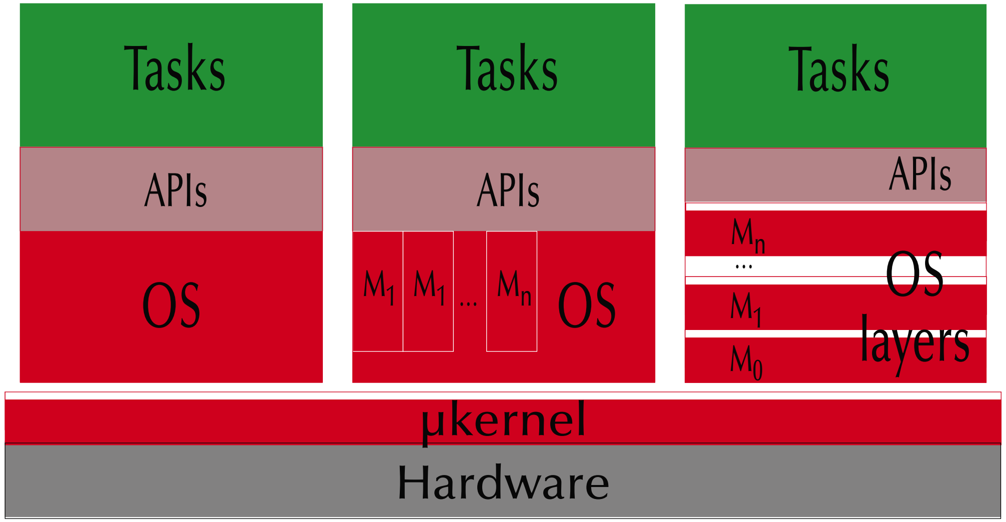
Layered

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

Typical structures of operating systems

µKernels & virtual machines

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



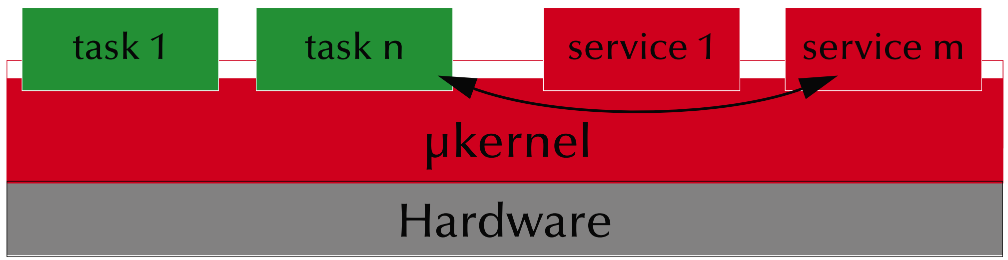
µkernel, virtual machine

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

Typical structures of operating systems

µKernels & client-server models

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



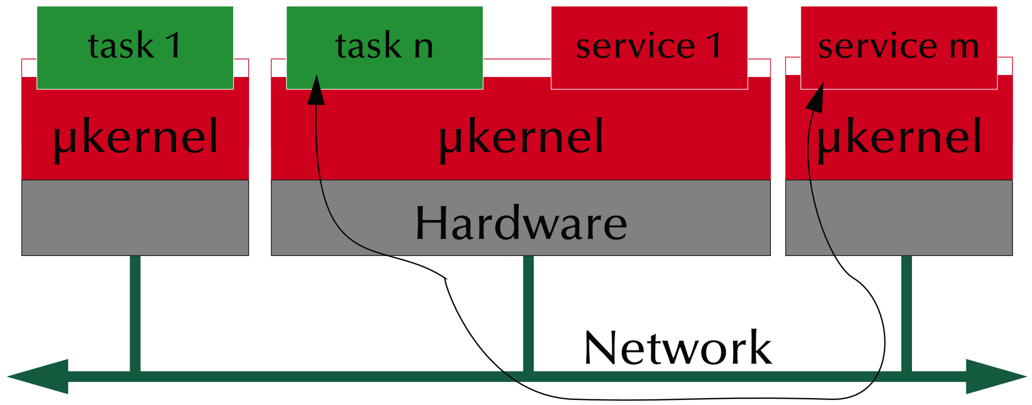
µkernel, client server structure

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

Typical structures of operating systems

µKernels & client-server models

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



µkernel, distributed systems

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

UNIX

UNIX features

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

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

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

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 */
}
```

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 ();
}
```

UNIX

Message passing in UNIX ☞ Pipes

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


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

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]`

POSIX - some of the relevant standards...

1003.1 12/01	OS Definition	single process, multi process, job control, signals, user groups, file system, file attributes, file device management, file locking, device I/O, device-specific control, system database, pipes, FIFO, ...
1003.1b 10/93	Real-time Extensions	real-time signals, priority scheduling, timers, asynchronous I/O, prioritized I/O, synchronized I/O, file sync, mapped files, memory locking, memory protection, message passing, semaphore, ...
1003.1c 6/95	Threads	multiple threads within a process; includes support for: thread control, thread attributes, priority scheduling, mutexes, mutex priority inheritance, mutex priority ceiling, and condition variables
1003.1d 10/99	Additional Real-time Extensions	new process create semantics (spawn), sporadic server scheduling, execution time monitoring of processes and threads, I/O advisory information, timeouts on blocking functions, device control, and interrupt control
1003.1j 1/00	Advanced Real-time Extensions	typed memory, nanosleep improvements, barrier synchronization, reader/writer locks, spin locks, and persistent notification for message queues
1003.21 -/-	Distributed Real-time	buffer management, send control blocks, asynchronous and synchronous operations, bounded blocking, message priorities, message labels, and implementation protocols

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

POSIX - other languages

POSIX is a 'C' standard...

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

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

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

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

Languages

High level programming languages...

- provide abstractions beyond assembly level (e.g. blocks: if, while, for..., methods, typing...)
- let the programmer concentrate on problems instead of implementation details
- offer more powerful data structures (types, containers, arrays, classes, etc.)
- make code more portable, more readable, safer, re-usable, etc.

Concurrent languages...

- offer abstractions for concurrency, communication, synchronisation, protection, etc.
- go beyond the *primitives* offered by the operating system
- may follow a particular paradigm of concurrency

Occam

named after William of Ockham (Philosopher and Franciscan monk, 1280-1349)

Occam's Razor:

"Pluralitas non est ponenda sine neccesitate"
or "plurality should not be posited without necessity"

Minimalist approach supplying all means for

- ☞ Concurrency & communication
- ☞ Distributed systems
- ☞ Realtime / predictable systems

Origins: CSP (Communicating Sequential Processes) by Tony Hoare, EPL

Occam

Characteristics: (...everything is a process)

- Primitive processes are
  - assignments
  - input or output statements (channel operations)
  - SKIP or STOP (elementary processes)
- Constructors are
  - SEQ (sequence) + replication
  - PAR (parallel) + replication
  - ALT (alternation) + replication + priorities

  - IF (conditional) + replication
  - CASE (selection)
  - WHILE (conditional loop)

Occam

Characteristics: (...everything is a process and static)

☞ no dynamic process creation

☞ no unlimited recursion

Syntax structure:

- Indentation is used for block indication (instead of ‘begin... end’ or brackets)

Scope of names:

- strictly local, indicated by indentation
- no ‘forward declarations’, ‘exports’, ‘global variables’ or ‘shared memory’

Go

currently under development by Google

- compiled, garbage-collected language
- fast compilation
- ‘C-like’ syntax, but omits a number of ‘dangerous’ constructs (e.g. pointer arithmetic, etc.)
- has built-in concurrency support, inspired by CSP (similarities to Occam)

Concurrency concept:

- “Go-routines”: concurrent procedure calls, runs procedure in a separate lightweight thread
- channels: synchronous (no capacity given) or asynchronous (if buffer size specified)
- select statements

☞ not “verifiable concurrency” in contrast to Occam/CSP

Chapel

currently under development by Cray / DARPA

☞ targeted at massively parallel supercomputers, multicores, clusters, …

built-in support for

- fine-grained parallelism:
  - ☞ concurrent loops and blocks (`cobegin, coforall`)
- task parallelism: creation, synchronisation, single-assignment variables, …
- atomic sections
- data parallelism:
  - ☞ domains
  - ☞ parallel iteration (`forall`)
  - ☞ SIMD statements (`scan, reduce`)

Ada

compiled, strongly-typed language for reliable, large scale systems

- strong built-in support for task-oriented concurrency
  - ☞ tasks
  - ☞ protected objects
  - ☞ task entries (synchronous message passing, remote invocation)
  - ☞ select statements
- support for realtime systems
  - ☞ precision timers, fixed-priority scheduling, 'delay until', high-reliability profiles (RAVENSCAR)
- support for distributed systems (distributed systems annex)
  - ☞ remote procedure calls (transparent)

Summary

Architectures

- **Hardware architectures - from simple logic to supercomputers**
  - logic, CPU architecture, pipelines, out-of-order execution, multithreading, ...
- **Operating systems**
  - basics: context switch, memory management, IPC
  - structures: monolithic, modular, layered, µkernels
  - UNIX, POSIX
- **Concurrency in languages**
  - some examples: CSP, Occam, Go, Chapel, Ada

8

Distributed Systems

Uwe R. Zimmer - The Australian National University

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

Network protocols & standards

OSI network reference model

Standardized as the

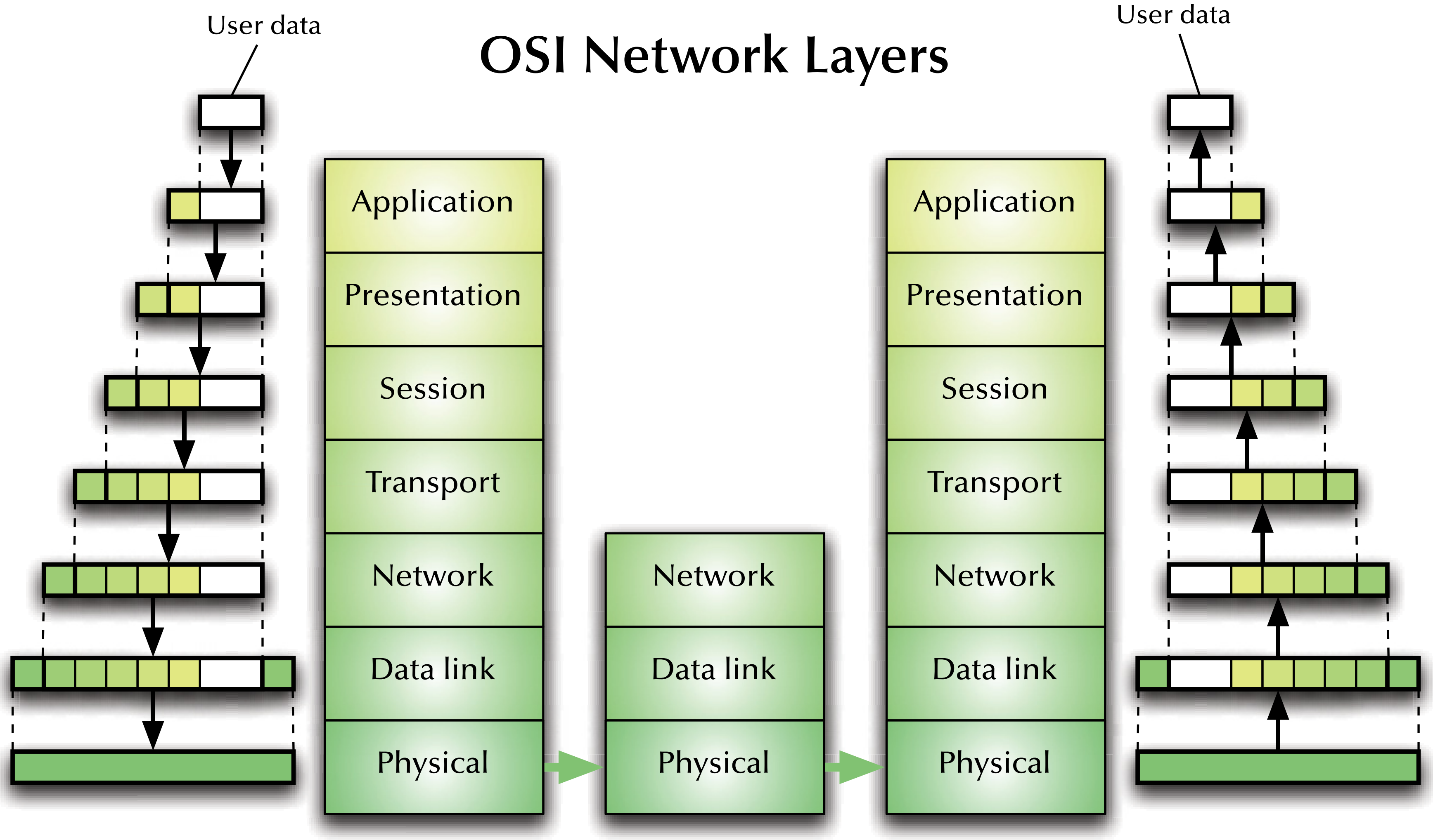
Open Systems Interconnection (**OSI**) reference model by the International Standardization Organization (ISO) in 1977

- 7 layer architecture
- Connection oriented

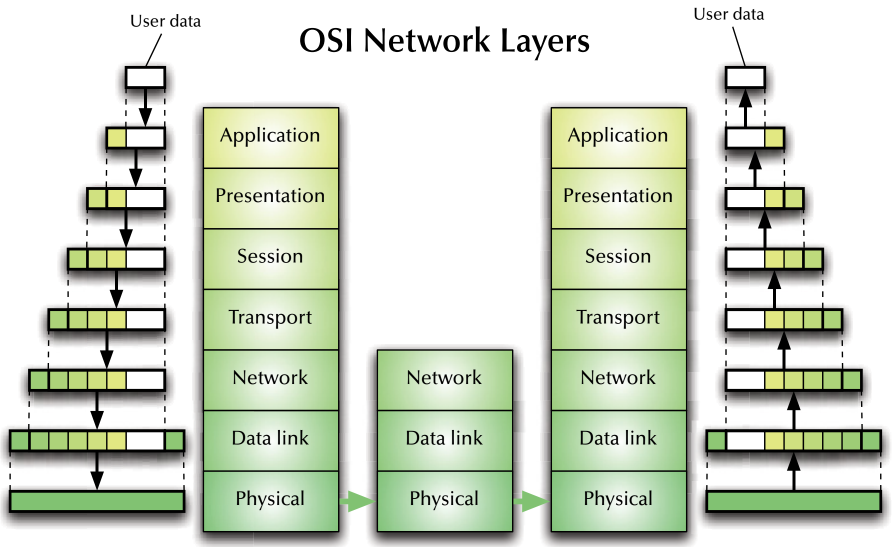
Hardy implemented anywhere in full …

…but its **concepts and terminology** are *widely used*, when describing existing and designing new protocols …

Network protocols & standards



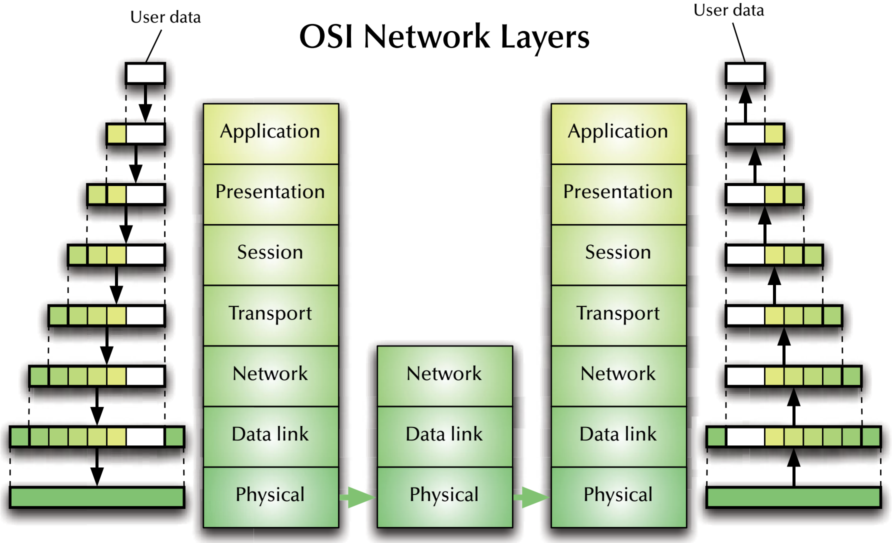
Network protocols & standards



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)

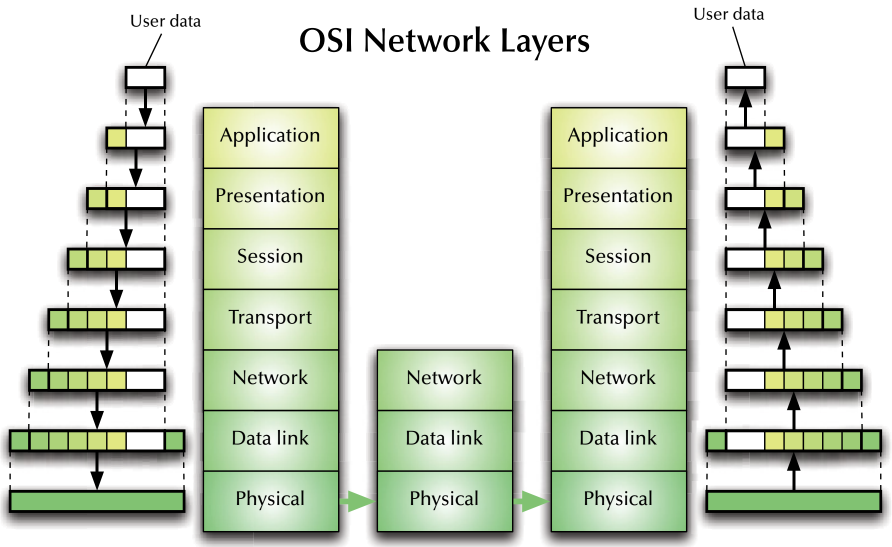
Network protocols & standards



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), …

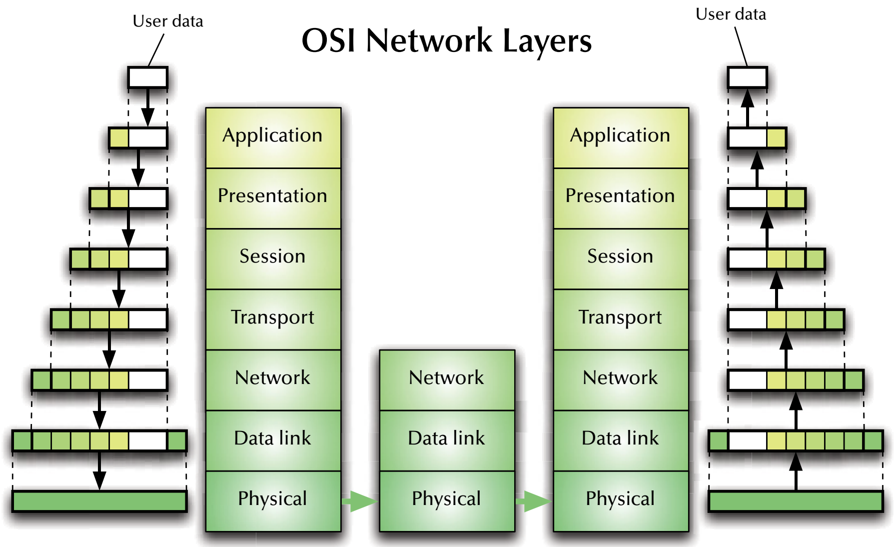
Network protocols & standards



3: Network Layer

- *Service*: Transfer of packets inside the network
- *Functions*: Routing, addressing, switching, congestion control
- *Examples*: IP, X.25

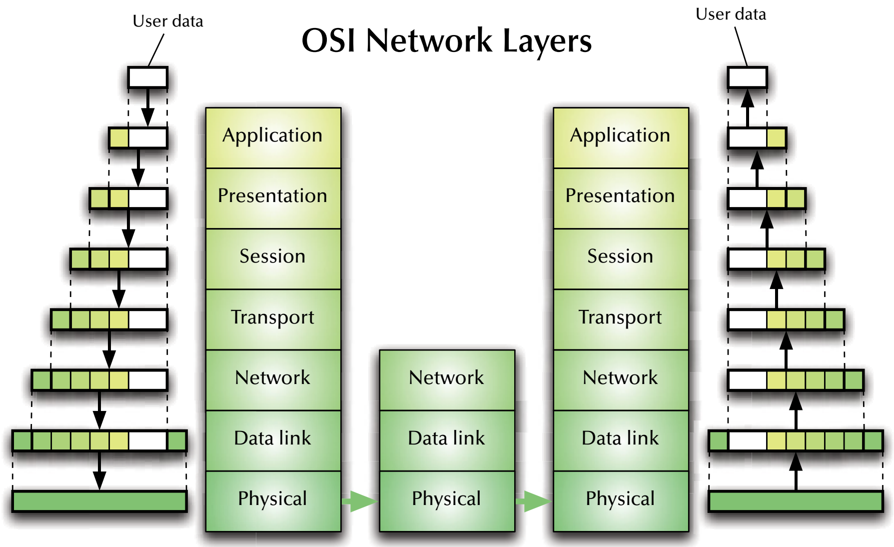
Network protocols & standards



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

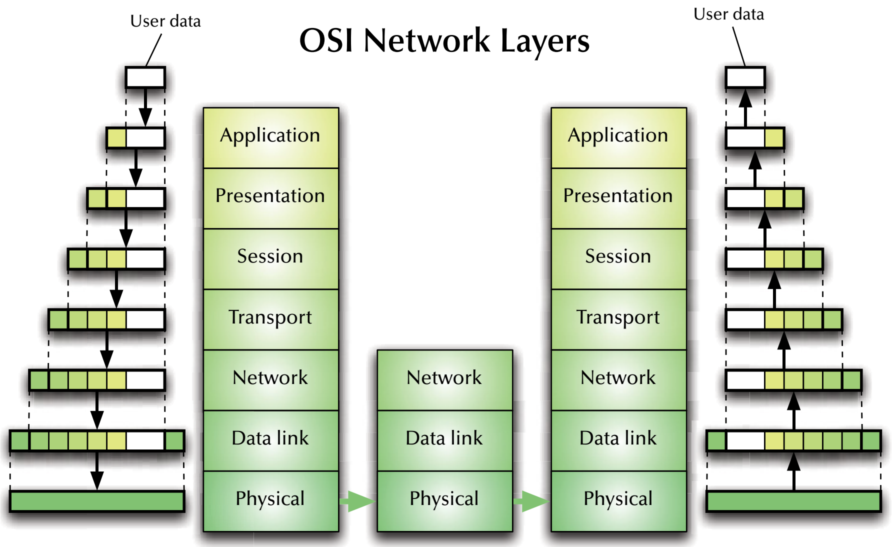
Network protocols & standards



5: Session Layer

- *Service*: Coordination of the dialogue between application programs
- *Functions*: Session establishment, management, termination
- *Examples*: RPC

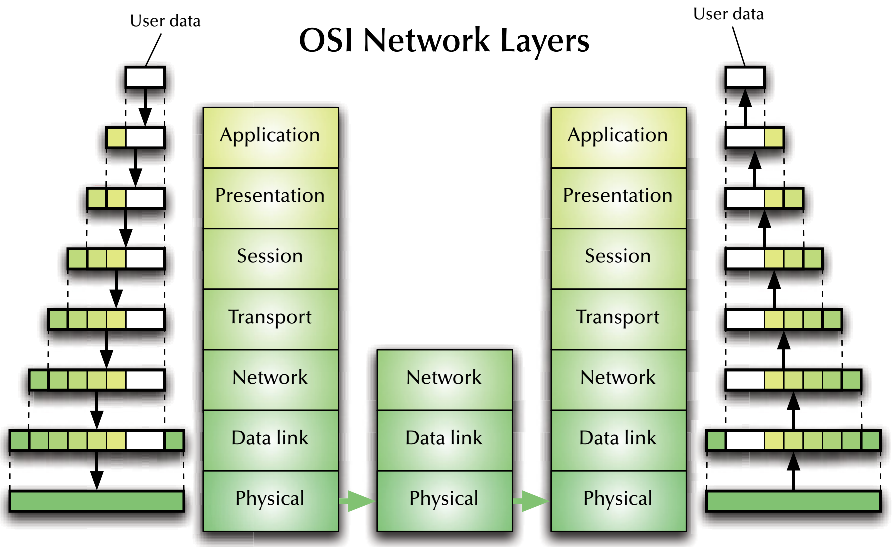
Network protocols & standards



6: Presentation Layer

- *Service*: Provision of platform independent coding and encryption
- *Functions*: Code conversion, encryption, virtual devices
- *Examples*: ISO code conversion, PGP encryption

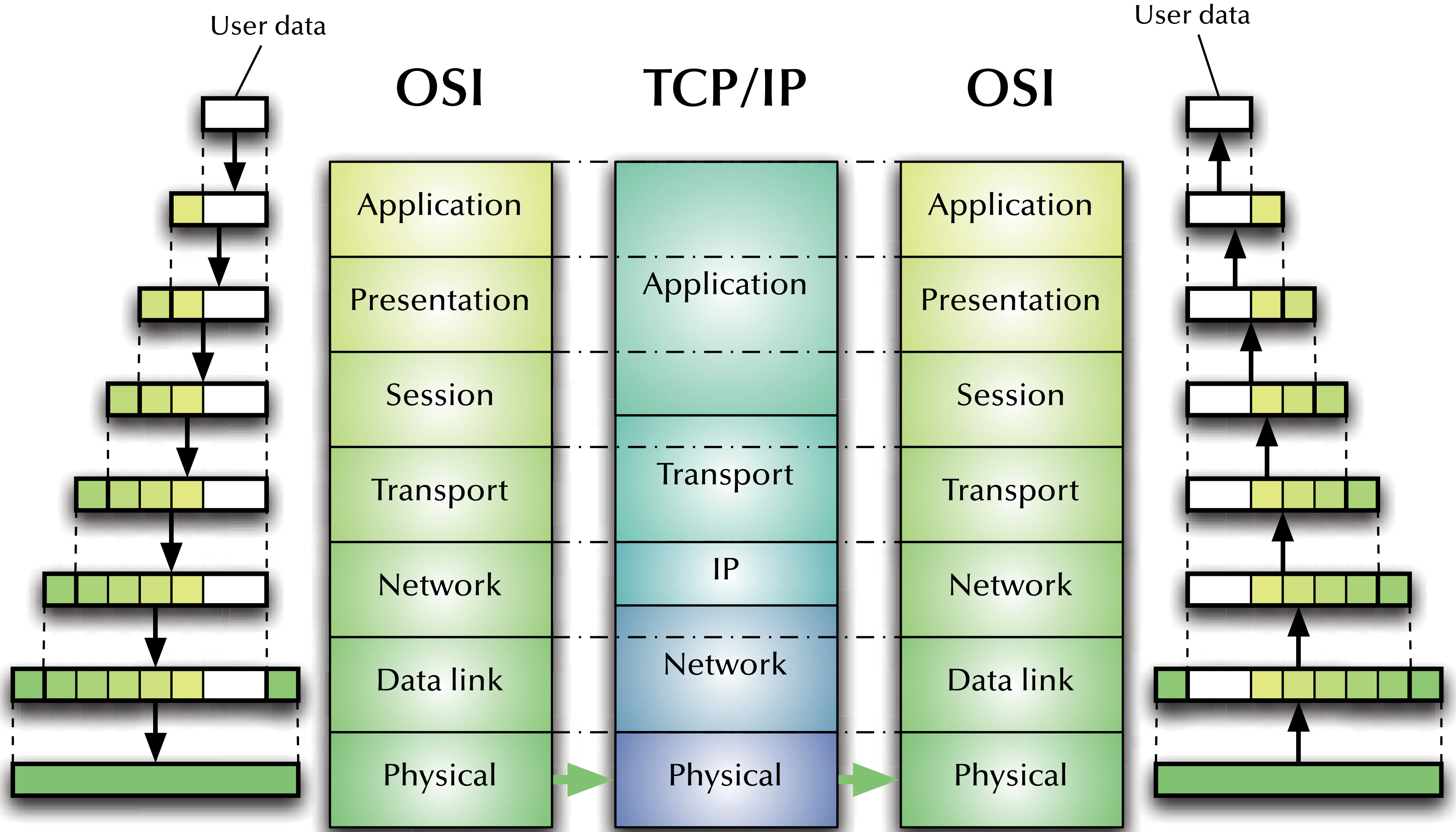
Network protocols & standards



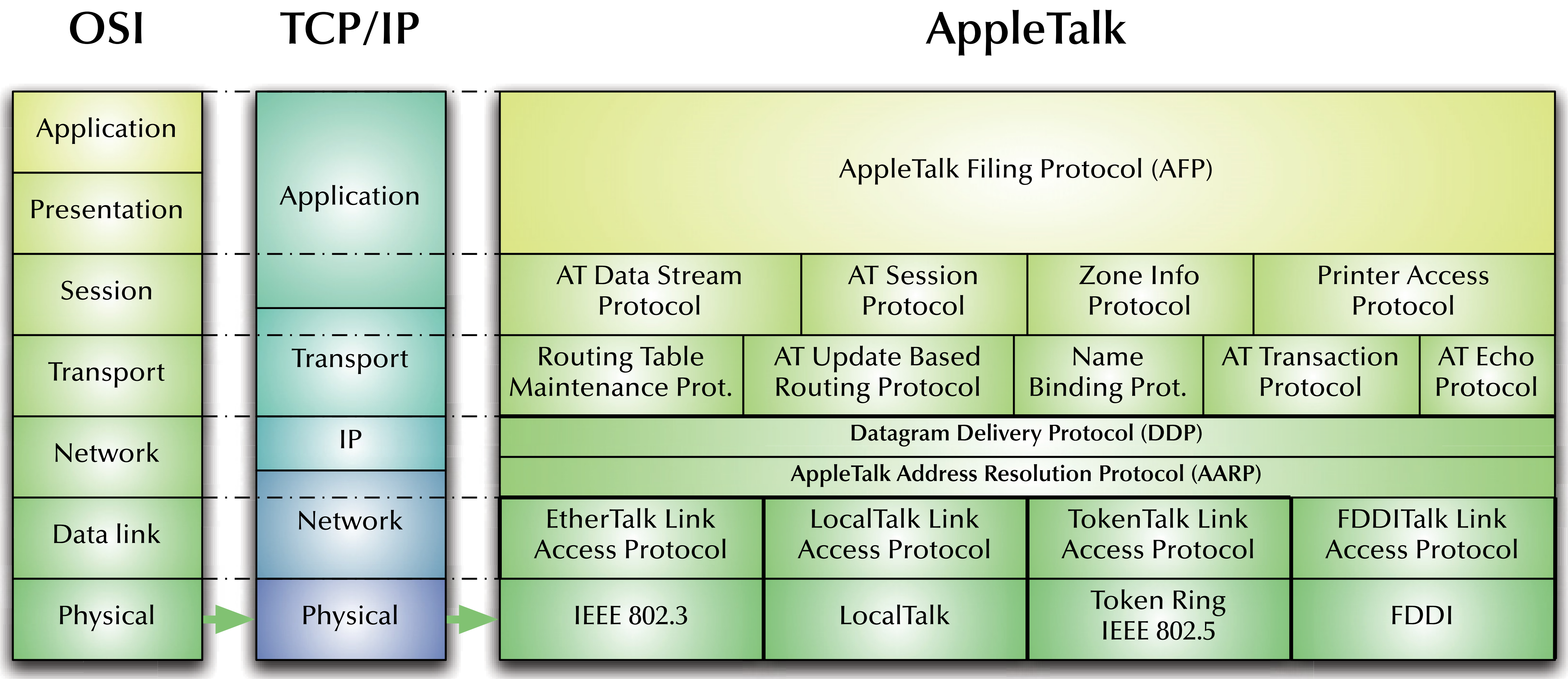
7: Application Layer

- *Service*: Network access for application programs
- *Functions*: Application/OS specific
- *Examples*: APIs for mail, ftp, ssh, scp, discovery protocols …

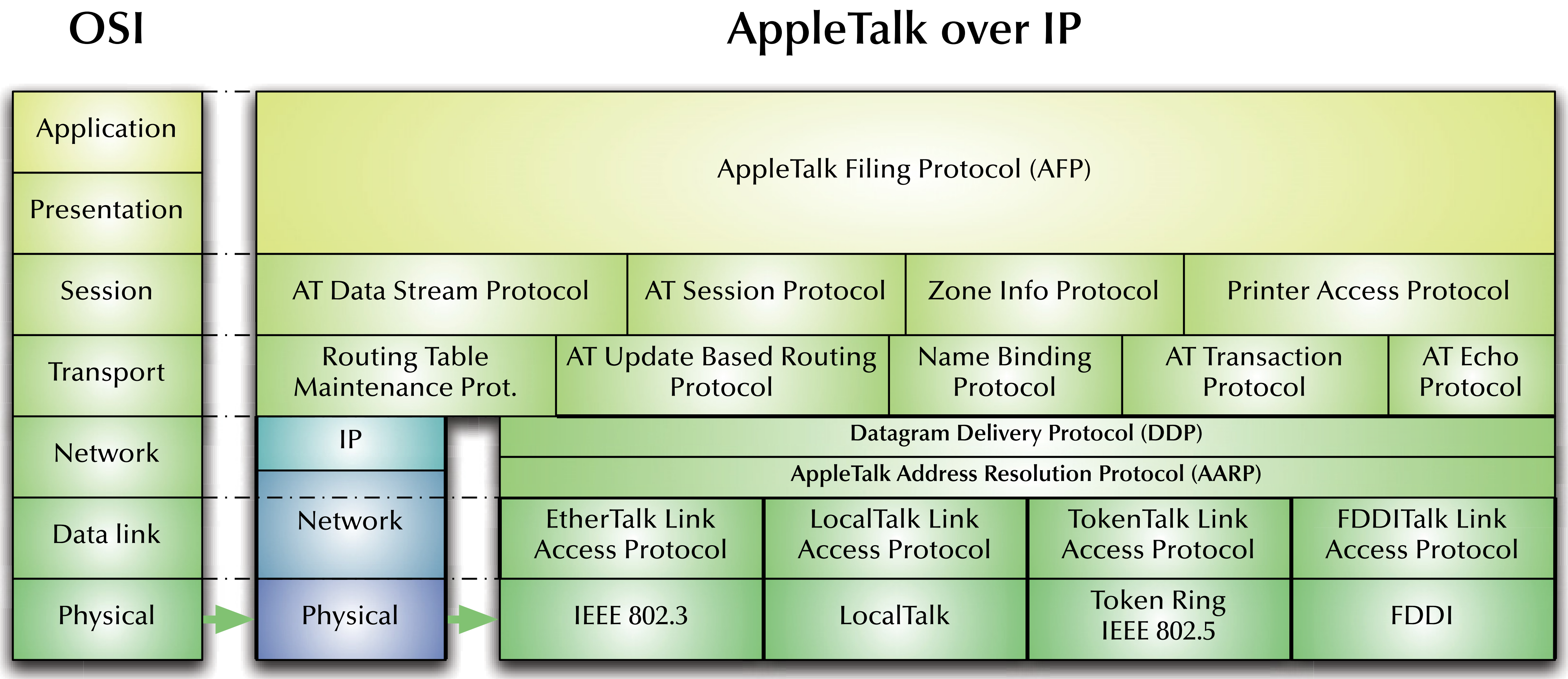
Network protocols & standards



Network protocols & standards



Network protocols & standards



Network protocols & standards

Ethernet / IEEE 802.3

Local area network (LAN) developed by Xerox in the 70's

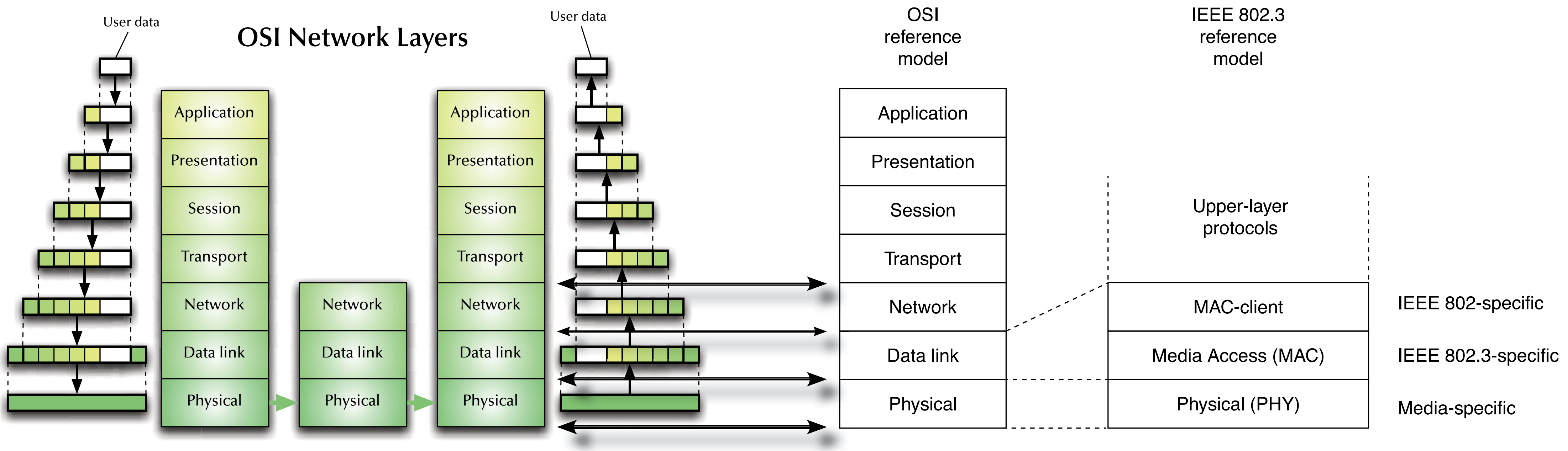
- 10 Mbps specification 1.0 by DEC, Intel, & Xerox in 1980.
- First standard as IEEE 802.3 in 1983 (10 Mbps over thick co-ax cables).
- currently 1 Gbps (802.3ab) copper cable ports used into most desktops and laptops.
- currently standards up to 100 Gbps (IEEE 802.3ba 2010).
- more than 85 % of current LAN lines worldwide (according to the International Data Corporation (IDC)).

☞ **Carrier Sense Multiple Access** with **Collision Detection** (**CSMA/CD**)

Network protocols & standards

Ethernet / IEEE 802.3

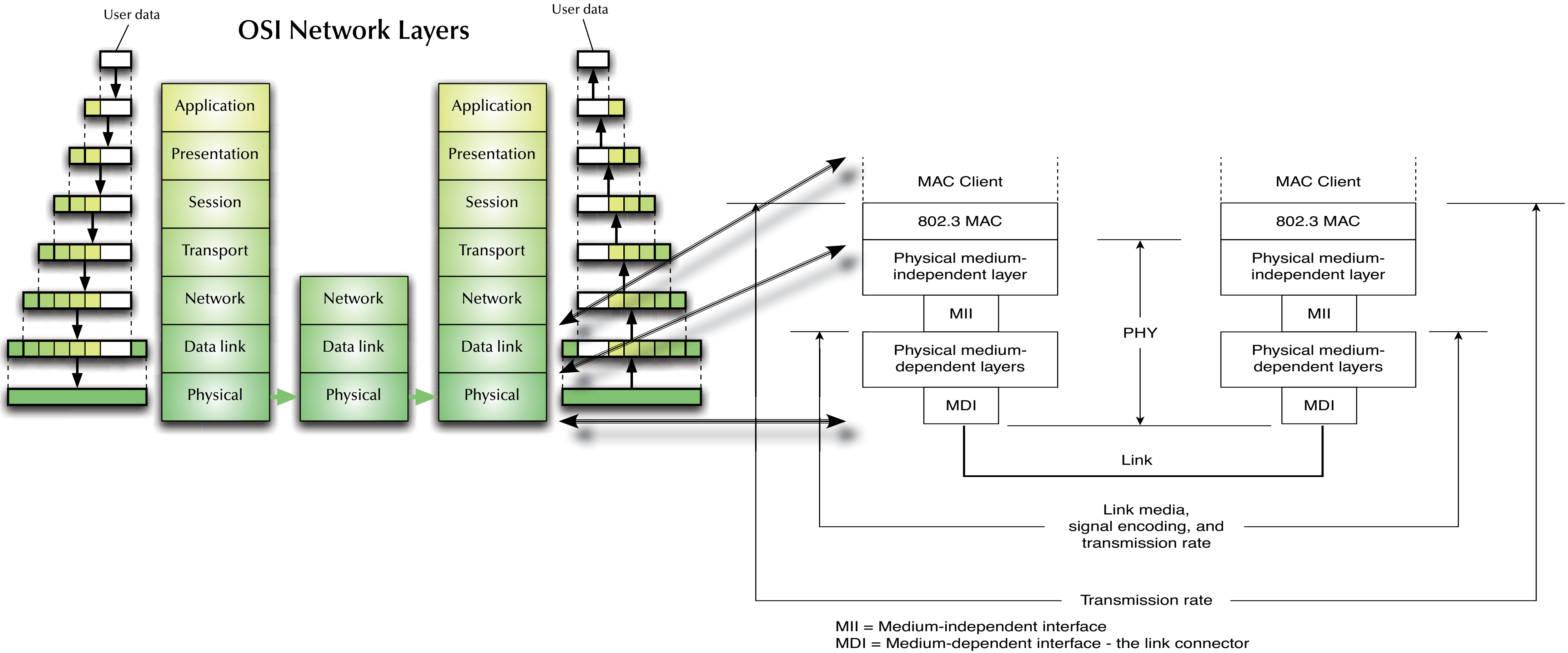
OSI relation: PHY, MAC, MAC-client



Network protocols & standards

Ethernet / IEEE 802.3

OSI relation: PHY, MAC, MAC-client



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 (1-2 Mbps over 2.4 GHz).
- Current typical usage at 54 Mbps over 2.4 GHz carrier at 20 MHz bandwidth.
- Standards up to 150 Mbps (802.11n) over 5 GHz carrier at 40 MHz bandwidth.
- 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)**

☞ No frequency hopping

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.

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.

Network protocols & standards

Fibre Channel

- Developed in the late 80's.
- ANSI standard since 1994.
- Current standards allow for > 5 Gbps per link.

- Allows for three different topologies:

  ☞ **Point-to-point**: 2 addresses

  ☞ **Arbitrated loop** (similar to token ring): 127 addresses ☞ deterministic, real-time capable

  ☞ **Switched fabric**: 2^{24} addresses, many topologies and concurrent data links possible

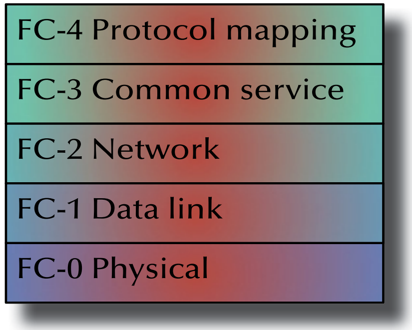
- Defines OSI equivalent layers up to the session level.

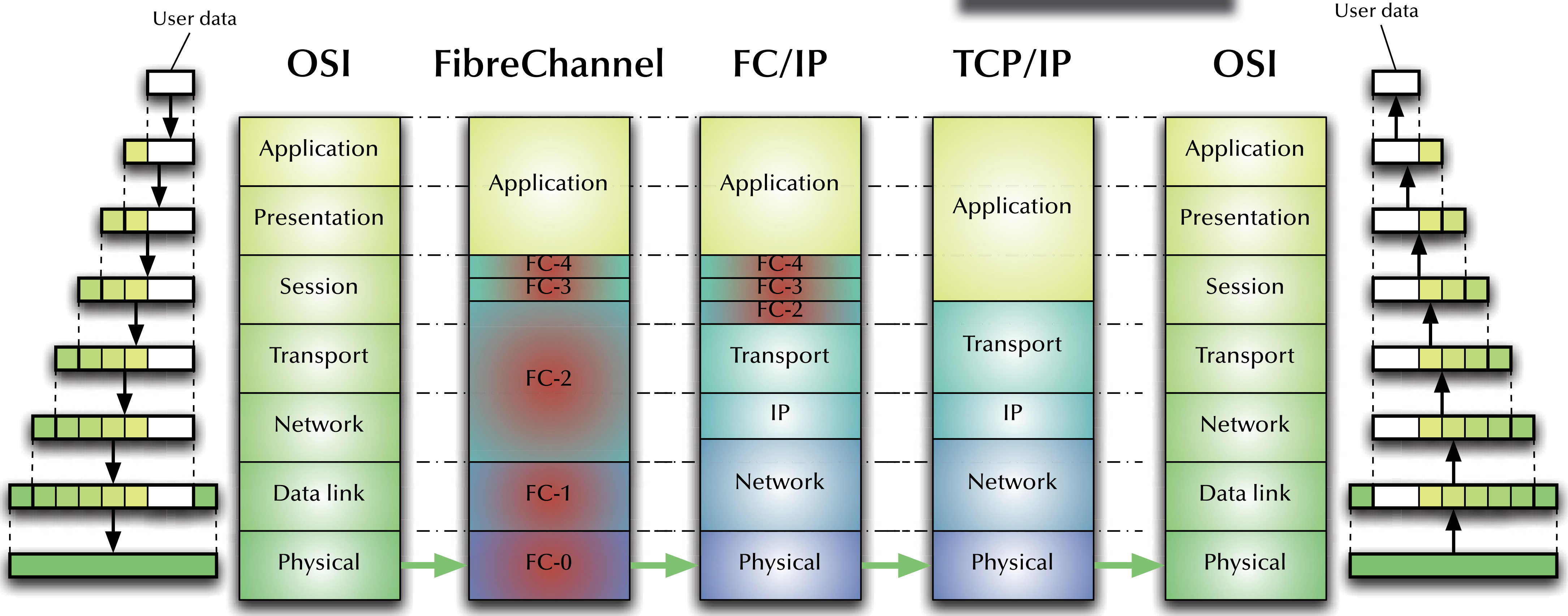
☞ Mostly used in storage arrays,
but applicable to super-computers and high integrity systems as well.

Network protocols & standards

Fibre Channel

Mapping of Fibre Channel to OSI layers:





Network protocols & standards

InfiniBand

- Developed in the late 90's
- Defined by the InfiniBand Trade Association (IBTA) since 1999.
- Current standards allow for > 10 Gbps per link.
- Switched fabric topologies.
- Concurrent data links possible.
- 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.

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.

Distributed Systems

What can be distributed?

- **State** ☞ common methods on distributed data
- **Function** ☞ distributed methods on central data
- **State & Function** ☞ client/server clusters
- **none of those** ☞ pure replication, redundancy

Distributed Systems

Common design criteria

- ☞ Achieve **De-coupling** / high degree of local autonomy
- ☞ **Cooperation** rather than central control
- ☞ Consider **Reliability**
- ☞ Consider **Scalability**
- ☞ Consider **Performance**

Distributed Systems

Some common phenomena in distributed systems

1. **Unpredictable delays** (communication)
   - ☞ Are we done yet?

2. **Missing or imprecise time-base**
   - ☞ Causal relation or temporal relation?

3. **Partial failures**
   - ☞ Likelihood of individual failures increases
   - ☞ Likelihood of complete failure decreases (in case of a good design)

Distributed Systems

Time in distributed systems

Two alternative strategies:

Based on a shared time ☞ **Synchronize clocks!**

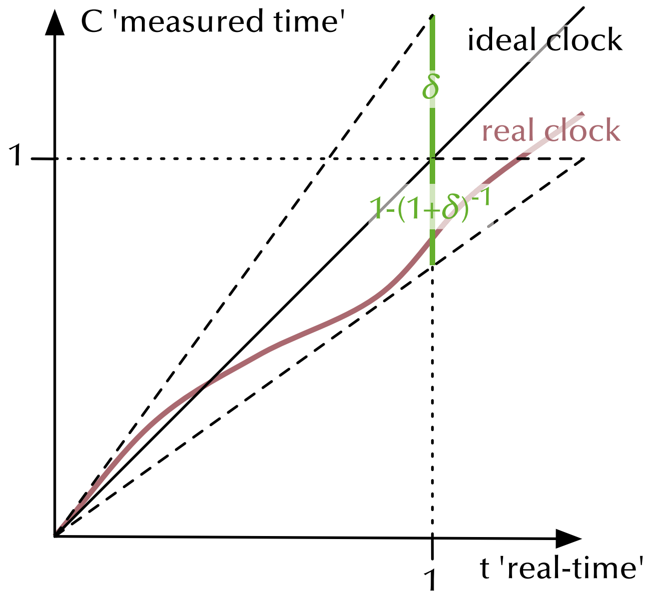
Based on sequence of events ☞ **Create a virtual time!**

Distributed Systems

'Real-time' clocks

are:

- **discrete** – i.e. time is *not* dense and there is a minimal granularity
- **drift affected**:



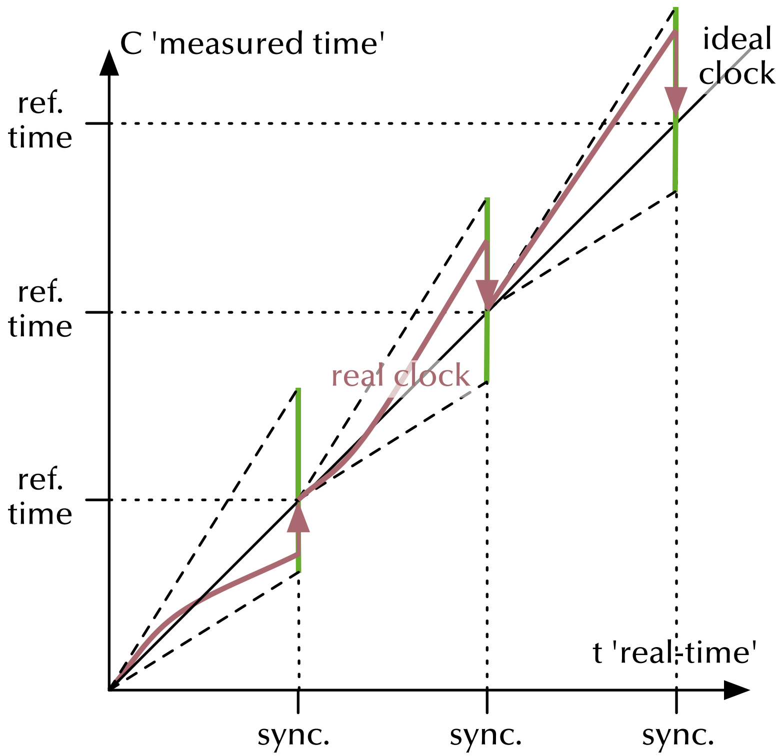
Maximal clock drift δ defined as:

$$(1+\delta)^{-1} \leq \frac{C(t_2)-C(t_1)}{t_2-t_1} \leq (1+\delta)$$

Distributed Systems

Synchronize a 'real-time' clock (bi-directional)

Resetting the clock drift by regular reference time re-synchronization:



Maximal clock drift δ defined as:

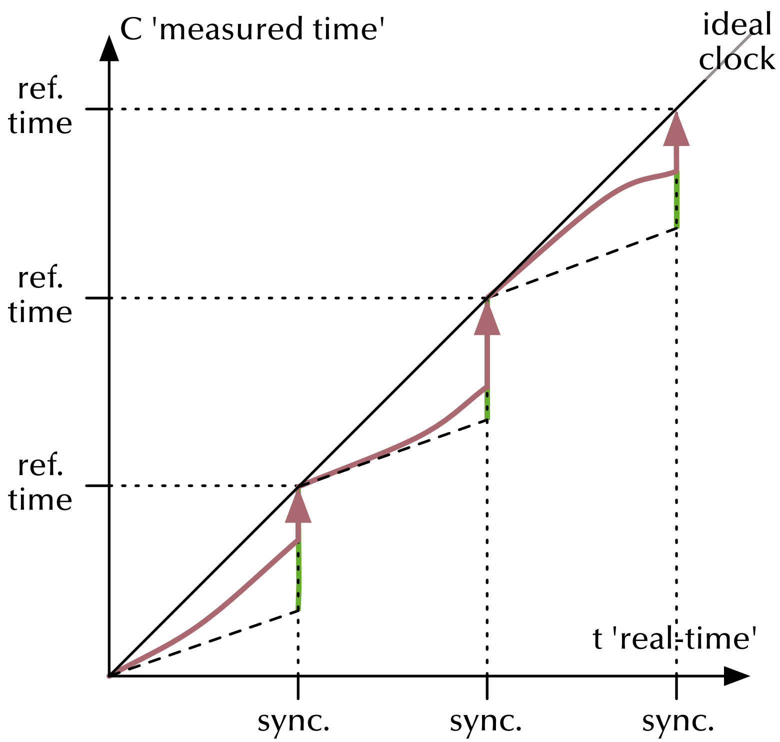
$$(1+\delta)^{-1} \leq \frac{C(t_2) - C(t_1)}{t_2 - t_1} \leq (1+\delta)$$

'real-time' clock is adjusted *forwards & backwards*

Distributed Systems

Synchronize a 'real-time' clock *(forward only)*

Resetting the clock drift by regular reference time re-synchronization:



Maximal clock drift δ 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

Distributed Systems

Distributed critical regions with synchronized clocks

- ∀ times:
  ∀ received *Requests*: **Add** to local *RequestQueue* (ordered by time)
  ∀ 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 2*L* (*L* being the time it takes for a message to reach all network nodes)
3. **While** Top (*RequestQueue*) ≠ *OwnRequest*: **delay** until new message
4. **Enter** and **leave** critical region
5. **Send** *Release*-message to *all* processes.

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.

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

$a \rightarrow b$ *holds true when:*

- a happens **earlier than** b in the *same sequential* control-flow.
- a denotes the **sending event** of message m,
  while b denotes the **receiving event** of the same message m.
- There is a transitive causal relation between a and b: $a \rightarrow e_1 \rightarrow \dots \rightarrow e_n \rightarrow b$

Notion of concurrency:

$$a \parallel b \Rightarrow \neg(a \rightarrow b) \wedge \neg(b \rightarrow a)$$

Distributed Systems

Virtual (logical) time

$$a \rightarrow b \Rightarrow C(a) < C(b)$$

Implications:

$$C(a) < C(b) \Rightarrow (a \rightarrow b) \vee (a \parallel b)$$

$$C(a) = C(b) \Rightarrow a \parallel b$$

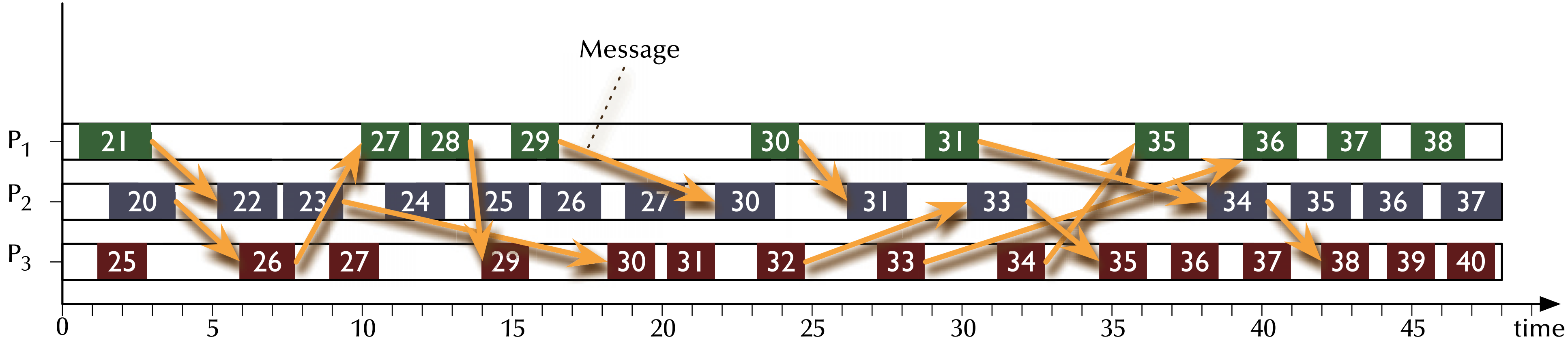
$$C(a) = C(b) < C(c) \Rightarrow (a \rightarrow c) \vee (a \parallel c)$$

$$C(a) < C(b) < C(c) \Rightarrow (a \rightarrow c) \vee (a \parallel c)$$

Distributed Systems

Virtual (logical) time

Time as derived from causal relations:



☞ Events in concurrent control flows are not ordered.

☞ No global order of time.

Distributed Systems

Implementing a virtual (logical) time

1. $\forall P_i$: $C_i = 0$

2. $\forall P_i$:

   $\forall$ local events: $C_i = C_i + 1$;

   $\forall$ send events: $C_i = C_i + 1$; Send (message, C_i);

   $\forall$ receive events: Receive (message, C_m); $C_i = \max(C_i, C_m) + 1$;

Distributed critical regions with logical clocks

- ∀ times: ∀ received *Requests*:
  Add to local *RequestQueue* (ordered by time)
  Reply with *Acknowledge* or *OwnRequest*
- ∀ times: ∀ 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. **Wait for** Top (*RequestQueue*) = *OwnRequest* & no outstanding replies
3. **Enter** and **leave** critical region
4. **Send** *Release*-message to *all* processes.

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.

Distributed Systems

Distributed critical regions with a token ring structure

1. **Organize** all processes in a logical or physical **ring** topology

2. **Send** one *token* message to one process

3. ∀ times, ∀ processes: **On receiving** the *token* message:
   1. If required the process
      enters and **leaves** a critical section (while holding the token).
   2. 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)

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.

Distributed Systems

Electing a central coordinator (the Bully algorithm)

Any process *P* which notices that the central coordinator is gone, performs:

1. *P* **sends** an *Election*-message
   to all processes with *higher* process numbers.
2. *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

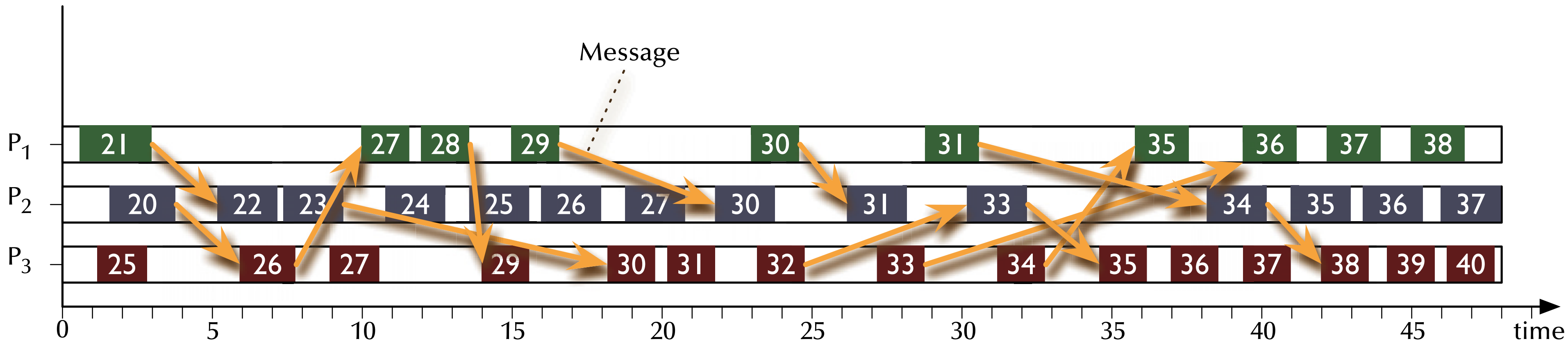
All processes P_i perform at all times:

- 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).

Distributed Systems

Distributed states

☞ How to read the current state of a distributed system?

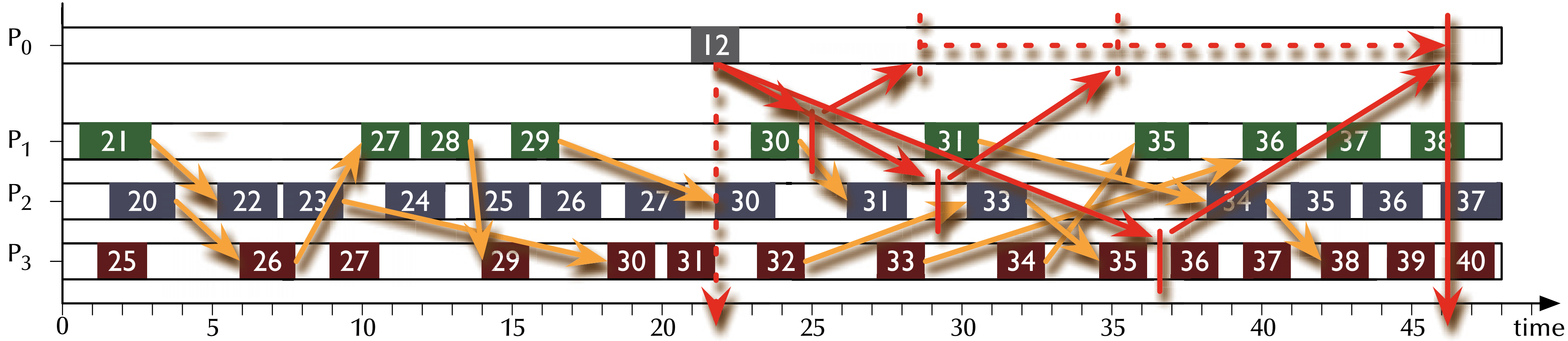


This "god's eye view" does in fact not exist.

Distributed Systems

Distributed states

☞ How to read the current state of a distributed system?



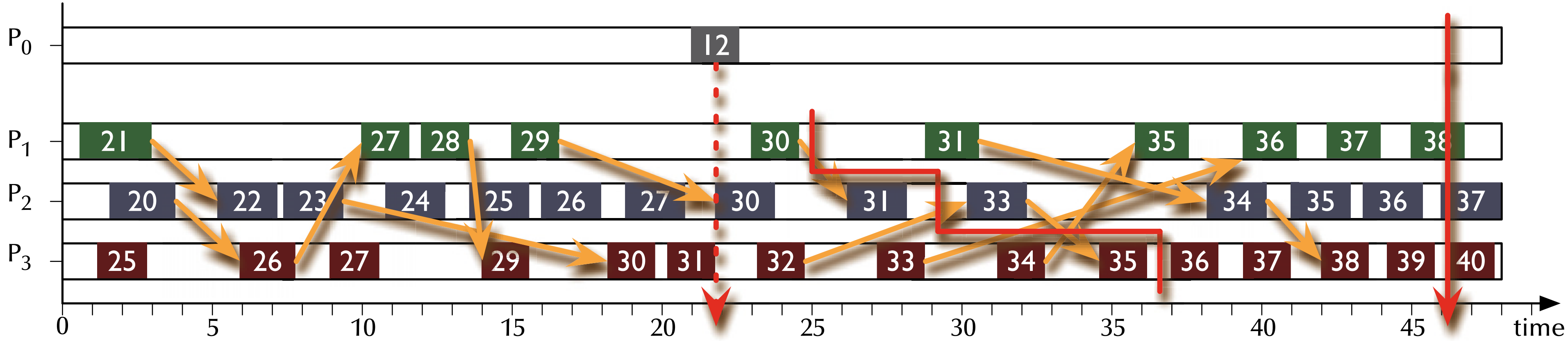
Instead: some entity probes and collects local states.

☞ What state of the global system has been accumulated?

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.

Distributed Systems

Distributed states

A consistent global state (snapshot) is define by a unique division into:

- "The Past" P (events before the snapshot):

$$(e_2 \in P) \wedge (e_1 \rightarrow e_2) \Rightarrow e_1 \in P$$

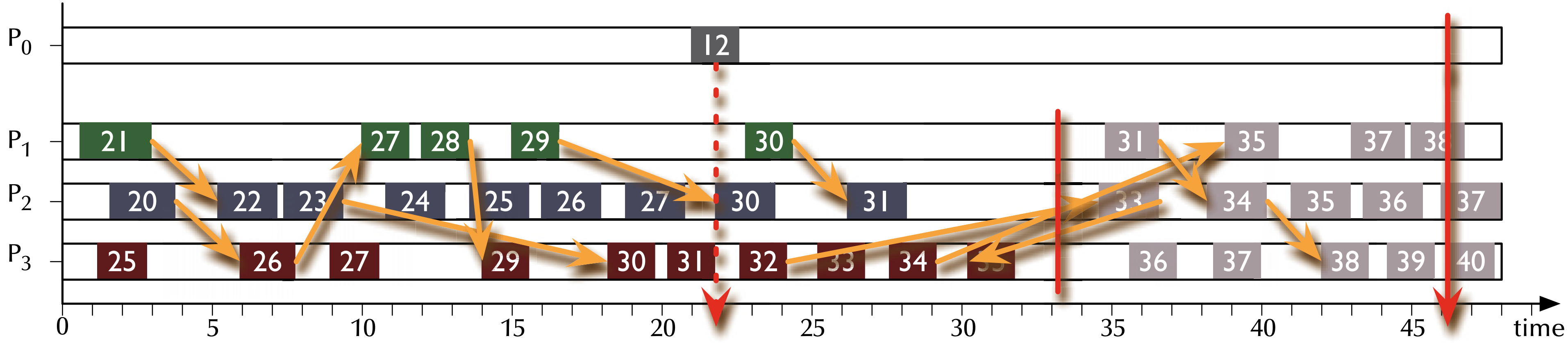
- "The Future" F (events after the snapshot):

$$(e_1 \in F) \wedge (e_1 \rightarrow e_2) \Rightarrow e_2 \in F$$

Distributed Systems

Distributed states

☞ How to read the current state of a distributed system?



Instead: some entity probes and collects local states.

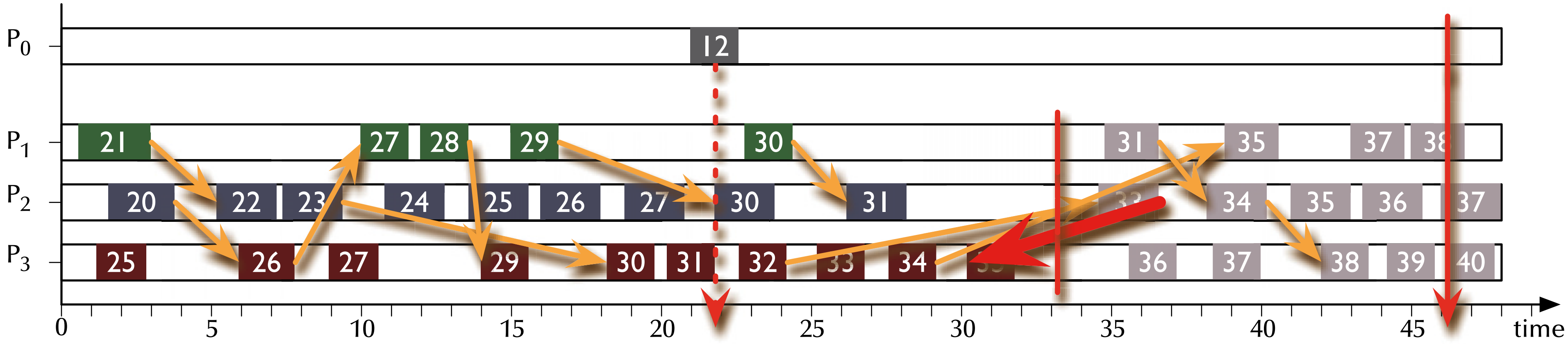
☞ What state of the global system has been accumulated?

☞ Sorting the events into past and future events.

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!

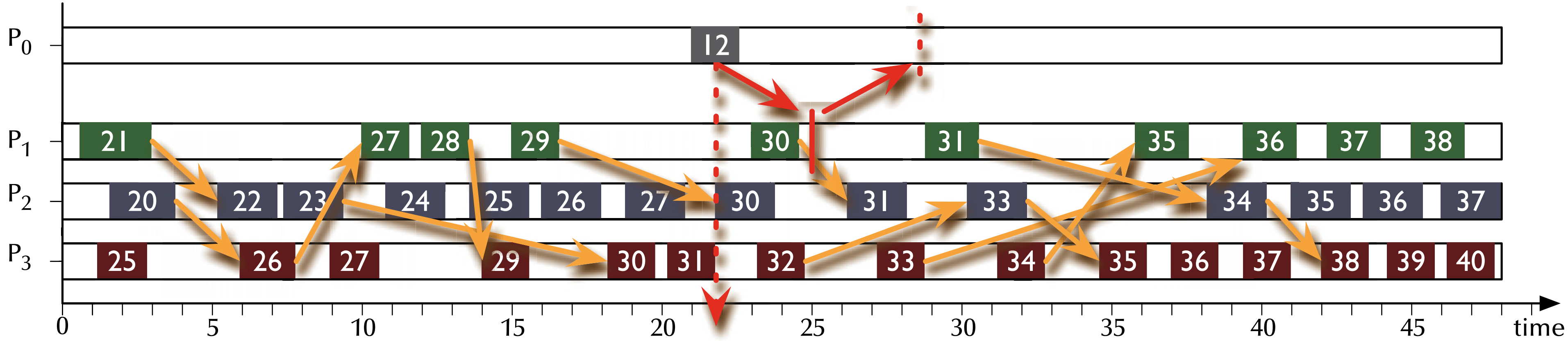
Distributed Systems

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).

Distributed states

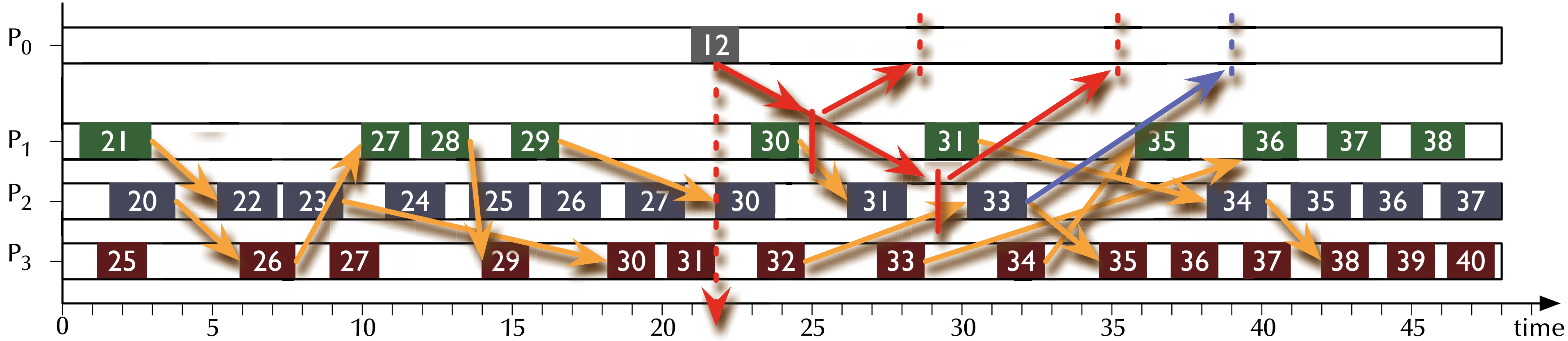
☞ Running the snapshot algorithm:



- 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.

Distributed states

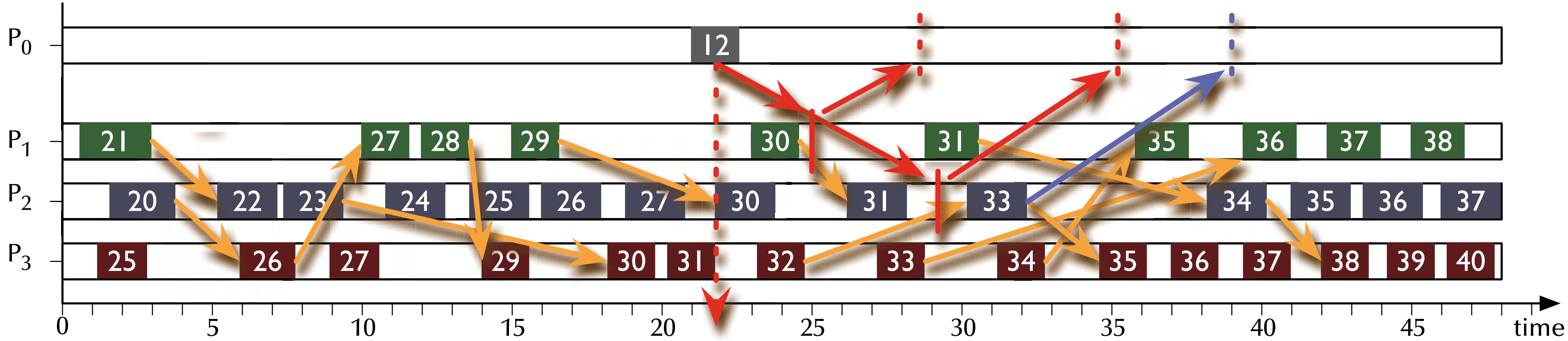
☞ Running the snapshot algorithm:



- $\forall P_i$ 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.

Distributed states

☞ Running the snapshot algorithm:

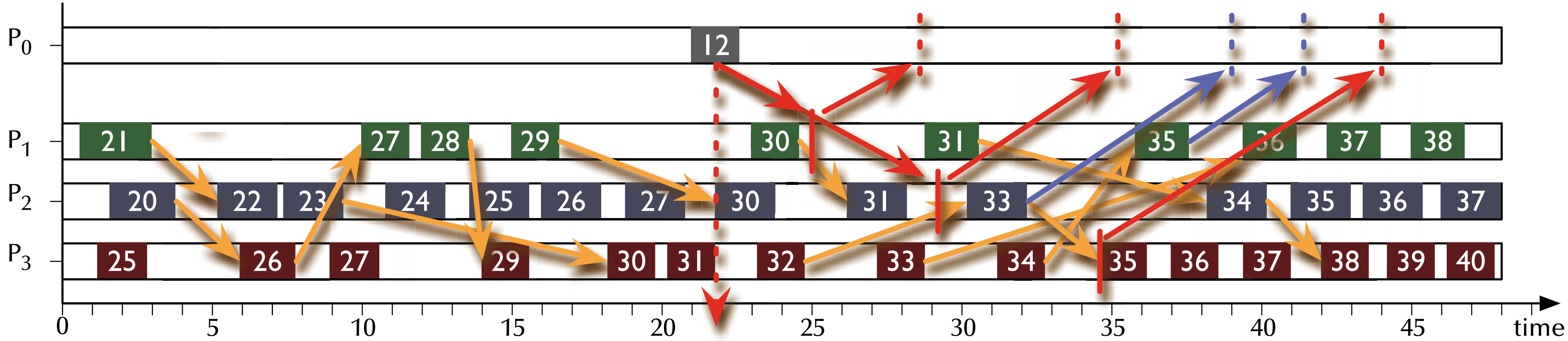


- $\forall P_i$ which previously received t_s and **receive** a message m without t_s:
  - **Forward** m to P_0 (this message belongs to the snapshot).

Distributed Systems

Distributed states

☞ Running the snapshot algorithm:

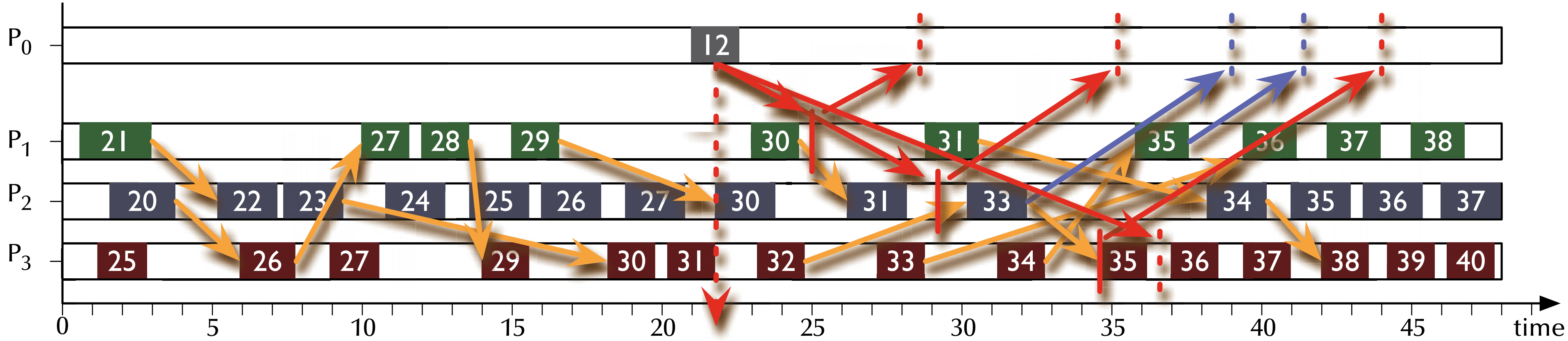


- $\forall P_i$ which **receive** t_s (as an individual token-message, or *as part of another message*):
  - **Save** local state s_i and **send** s_i to P_0.
  - **Attach** t_s to all further messages, which are to be sent to other processes.
  - **Save** t_s and **ignore** all further incoming t_s's.

Distributed Systems

Distributed states

☞ Running the snapshot algorithm:

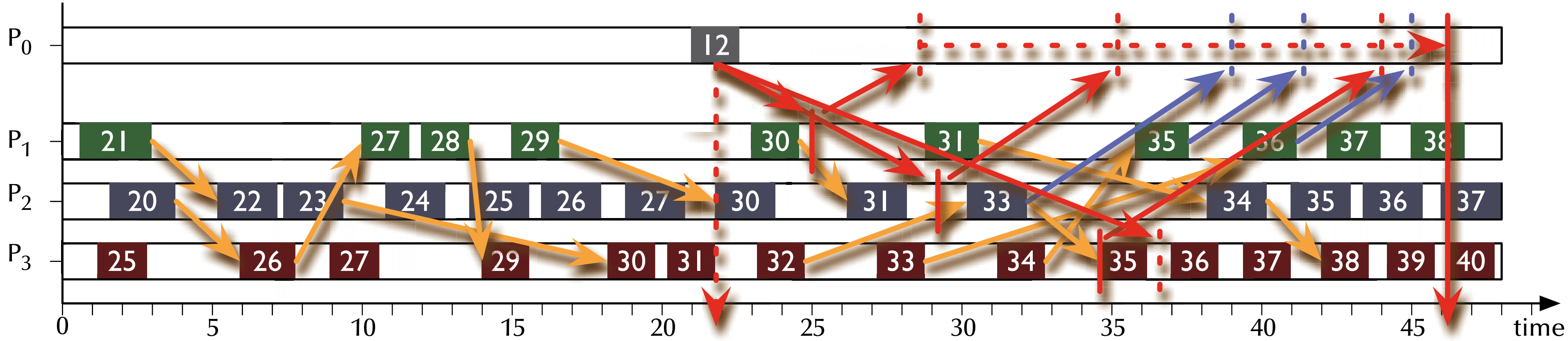


- **Save** t_s and **ignore** all further incoming t_s's.

Distributed Systems

Distributed states

☞ Running the snapshot algorithm:

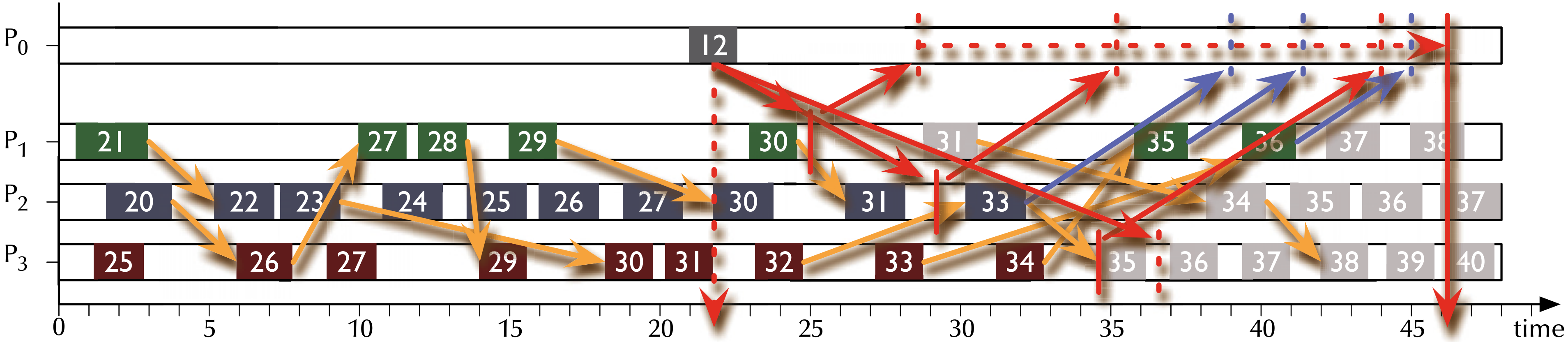


- **Finalize** snapshot

Distributed Systems

Distributed states

☞ Running the snapshot algorithm:



☞ Sorting the events into past and future events.

☞ Past and future events uniquely separated ☞ Consistent state

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.

Consistent distributed states

Why would we need that?

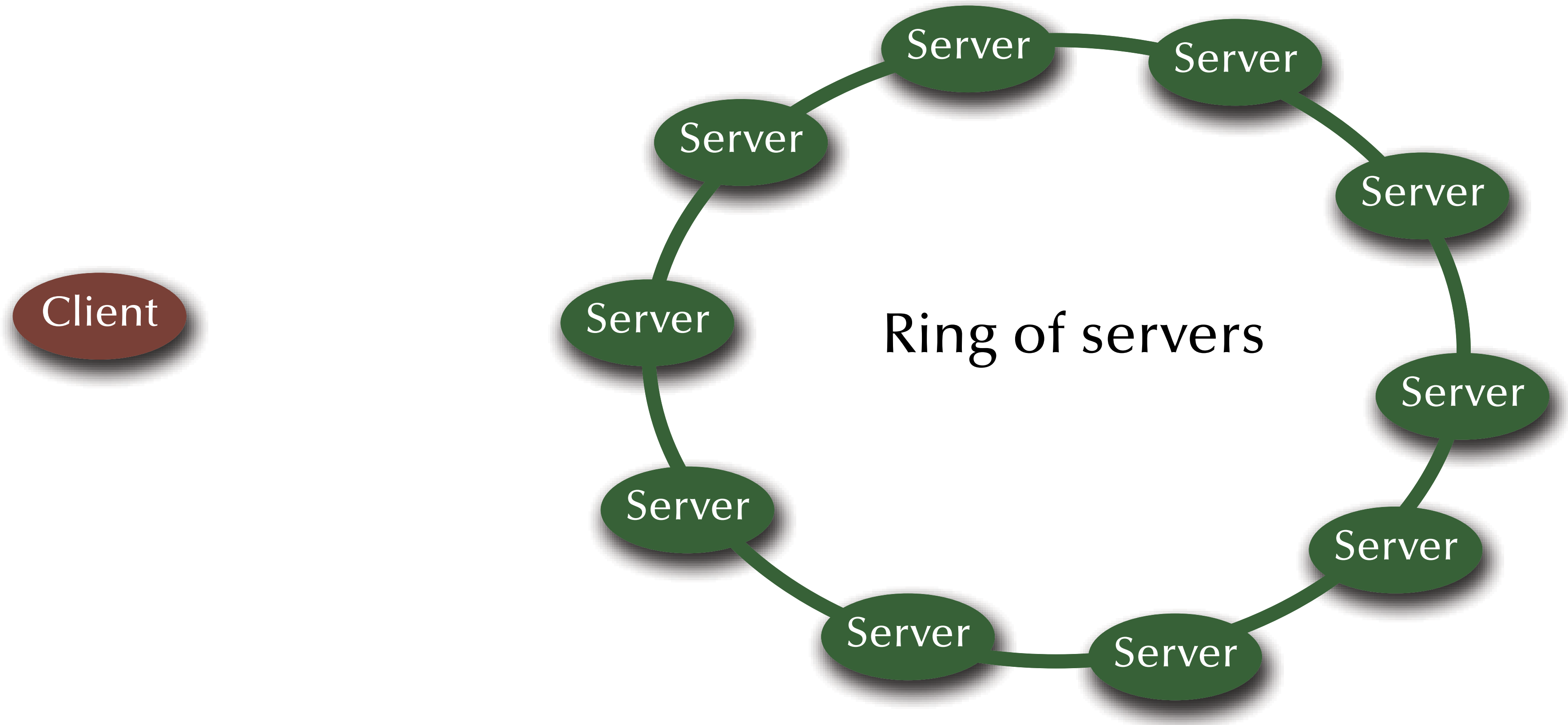
- Find deadlocks.
- Find termination / completion conditions.
- … any other global safety of liveness property.
- Collect a consistent system state for system backup/restore.
- Collect a consistent system state for further processing (e.g. distributed databases).
- …

Distributed Systems

A distributed server (load balancing)

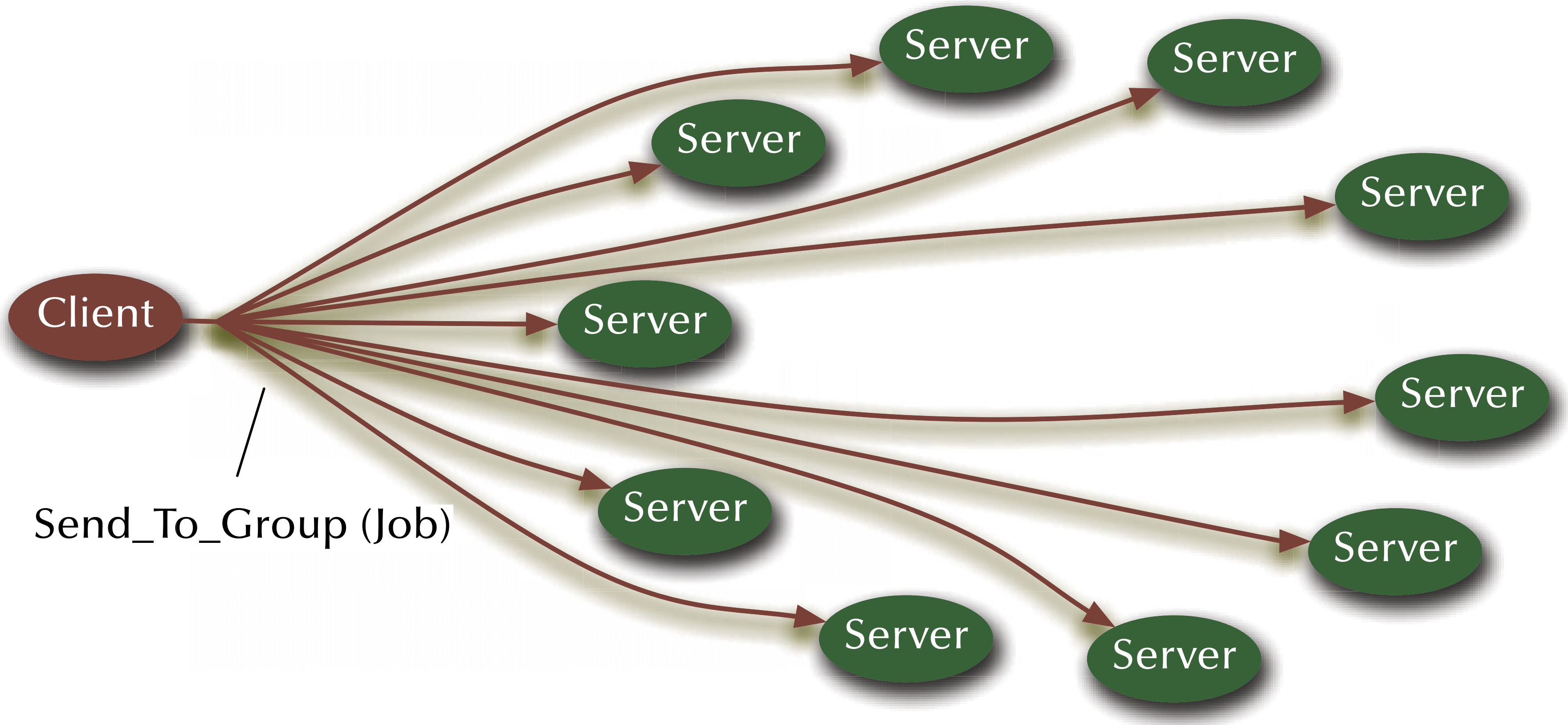
Distributed Systems

A distributed server (load balancing)



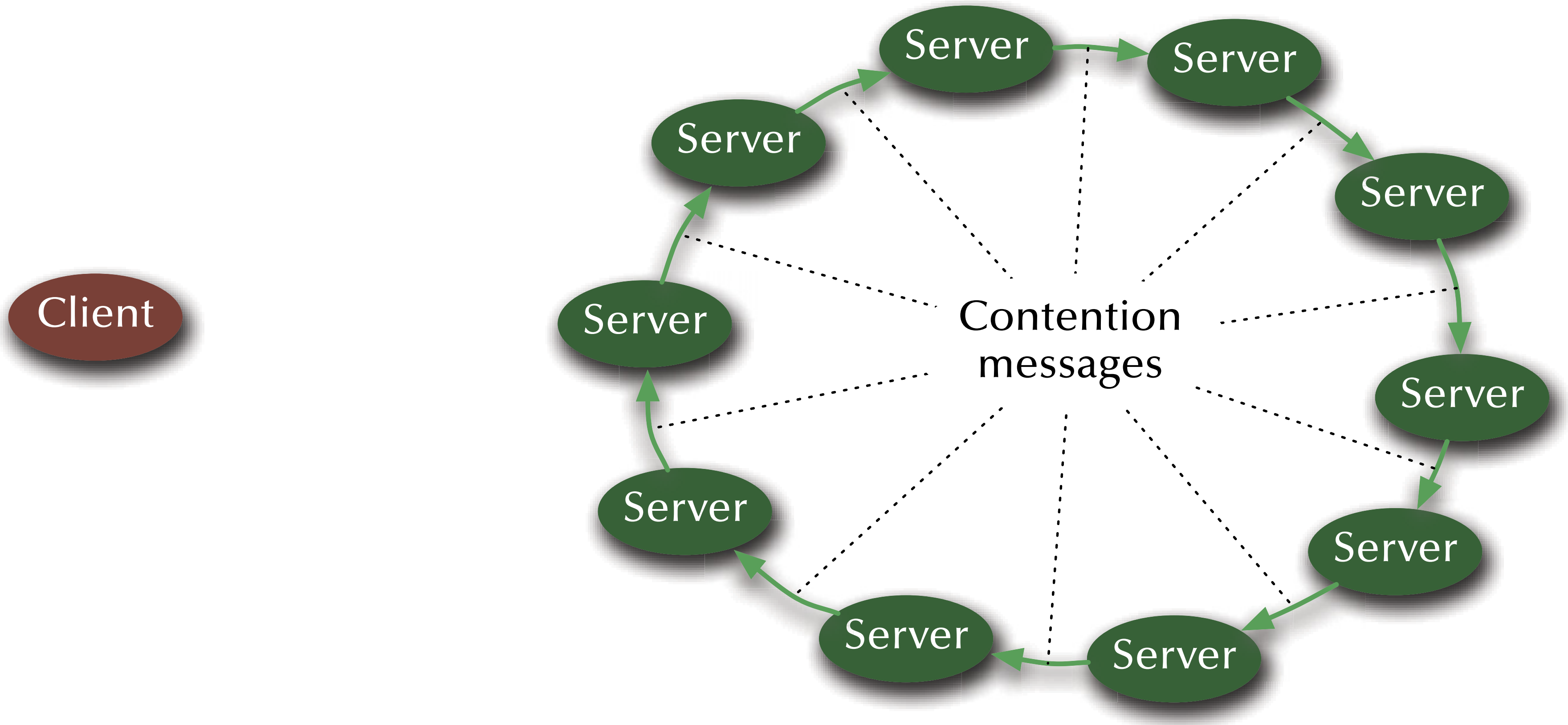
Distributed Systems

A distributed server (load balancing)



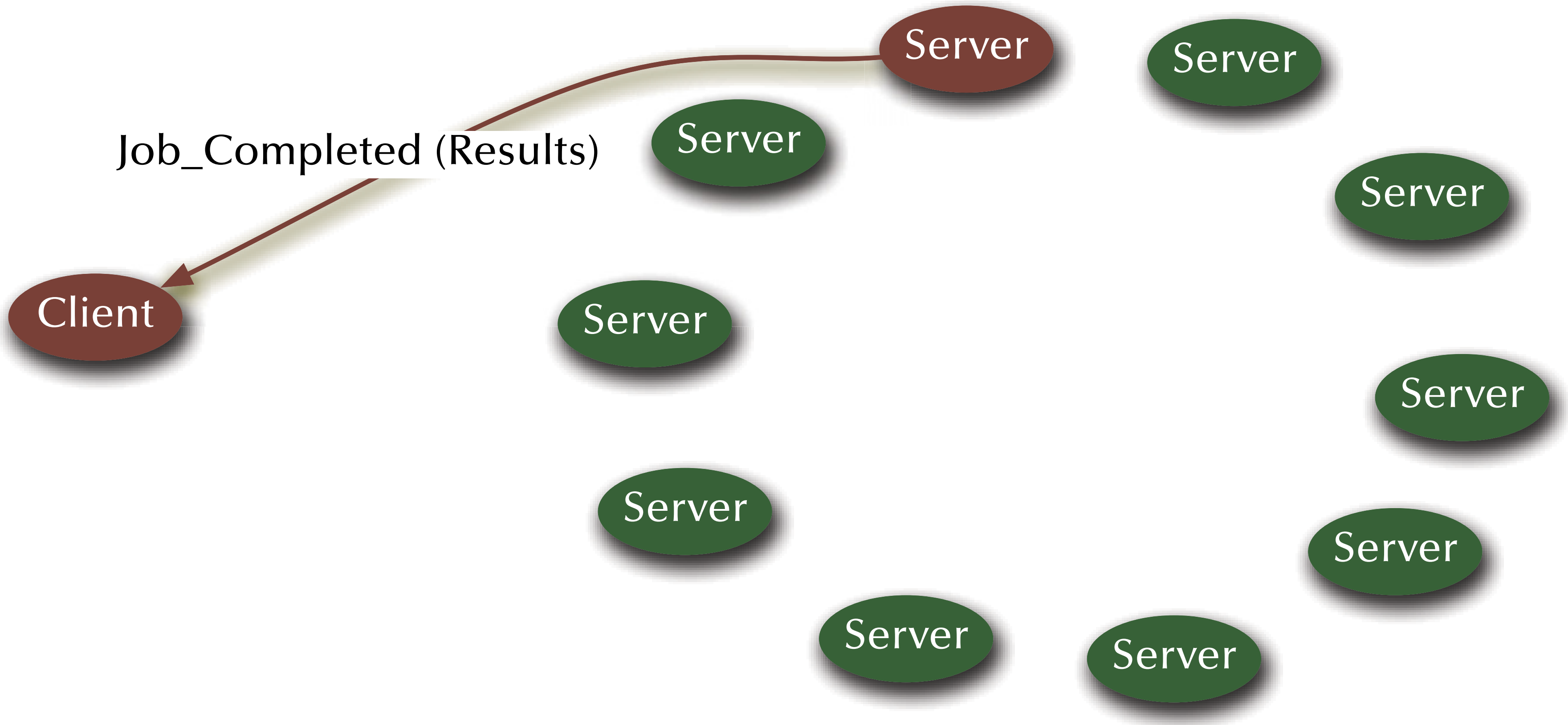
Distributed Systems

A distributed server (load balancing)



Distributed Systems

A distributed server (load balancing)



Distributed Systems

A distributed server (load balancing)

```
with Ada.Task_Identification; use Ada.Task_Identification;

task type PrintServer is

   entry SendToServer (PrintJob : in Job_Type; JobDone  : out Boolean);
   entry Contention   (PrintJob : in Job_Type; ServerId : in  Task_Id);

end PrintServer;
```

Distributed Systems

A distributed server (load balancing)

```
task body PrintServer is
   begin
      loop
         select
            accept SendToServer (PrintJob : in Job_Type; JobDone : out Boolean) do
               if not PrintJob in TurnedDownJobs then
                  if not_too_busy then
                     AppliedForJobs := AppliedForJobs + PrintJob;
                     NextServerOnRing.Contention (PrintJob, Current_Task);
                     Requeue InternalPrintServer.PrintJobQueue;
                  else
                     TurnedDownJobs := TurnedDownJobs + PrintJob;
                  end if;
               end if;
            end SendToServer;
```

(...)

```
      or
         accept Contention (PrintJob : in Job_Type; ServerId : in Task_Id) do
            if PrintJob in AppliedForJobs then
               if ServerId = Current_Task then
                  InternalPrintServer.StartPrint (PrintJob);
               elsif ServerID > Current_Task then
                  InternalPrintServer.CancelPrint (PrintJob);
                  NextServerOnRing.Contention (PrintJob; ServerId);
               else
                  null; - removing the contention message from ring
               end if;
            else
               TurnedDownJobs := TurnedDownJobs + PrintJob;
               NextServerOnRing.Contention (PrintJob; ServerId);
            end if;
         end Contention;
      or
         terminate;
      end select;
   end loop;
end PrintServer;
```

Distributed Systems

Transactions

☞ Concurrency and distribution in systems with multiple, interdependent interactions?

☞ Concurrent and distributed client/server interactions beyond single remote procedure calls?

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.

Distributed Systems

Transactions

Definition (ACID properties):

Atomic operations spanning multiple processes?

How to ensure consistency in a distributed system?

- **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.

Shadow copies?

What hardware do we need to assume?

Actual isolation and efficient concurrency?

Actual isolation or the appearance of isolation?

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**.

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**.

Distributed Systems

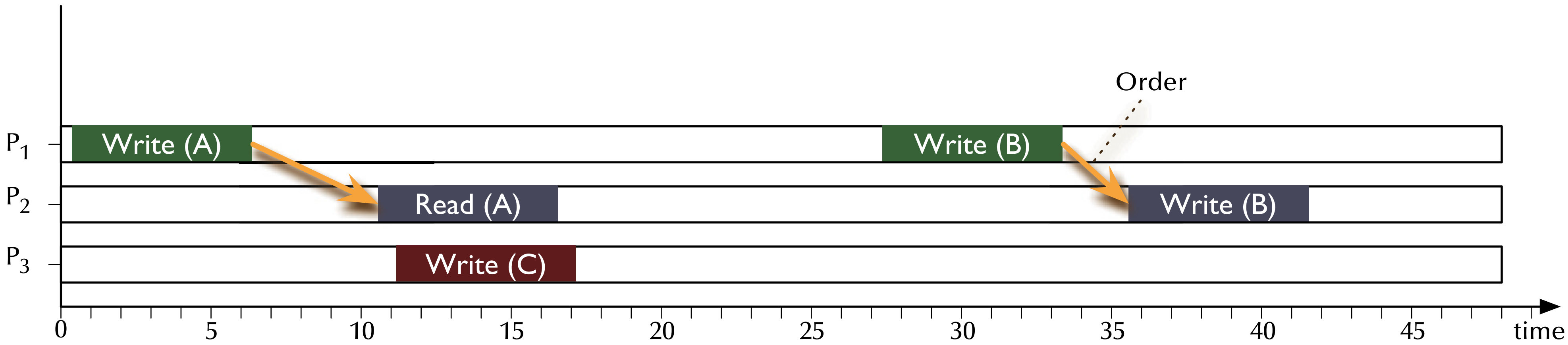
Achieving serializability

☞ 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.)

Distributed Systems

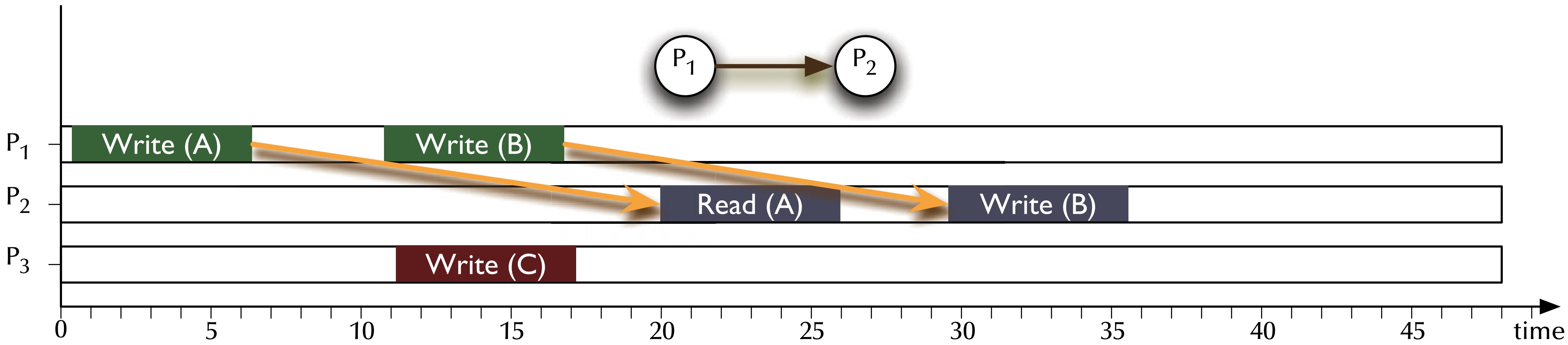
Serializability



- Two conflicting pairs of operations with the same order of execution.

Distributed Systems

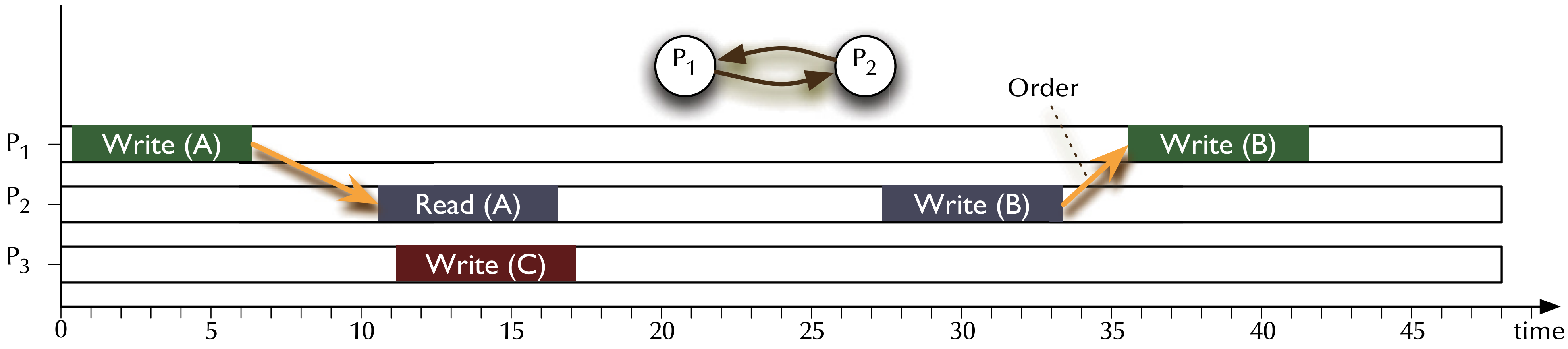
Serializability



☞ Serializable

Distributed Systems

Serializability

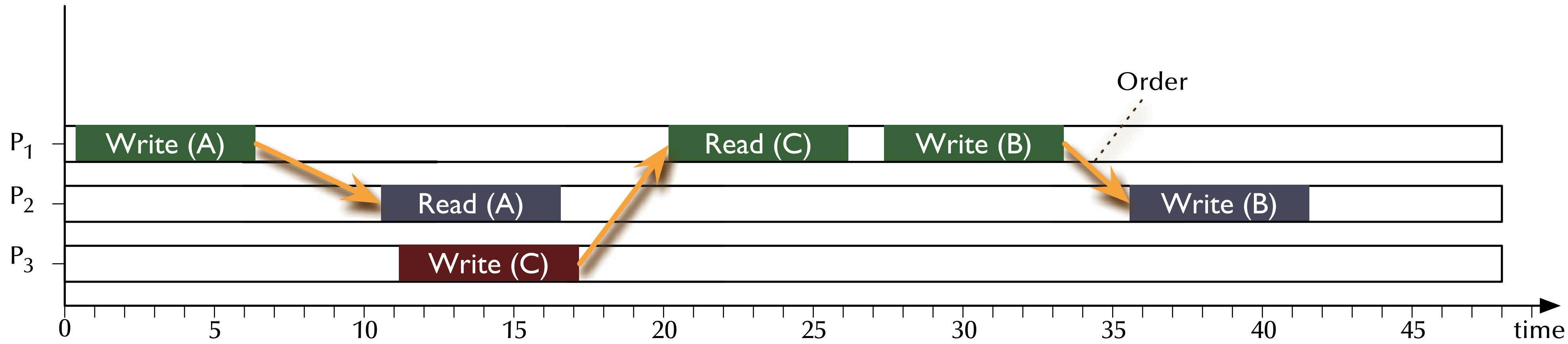


- Two conflicting pairs of operations with different orders of executions.

☞ Not serializable.

Distributed Systems

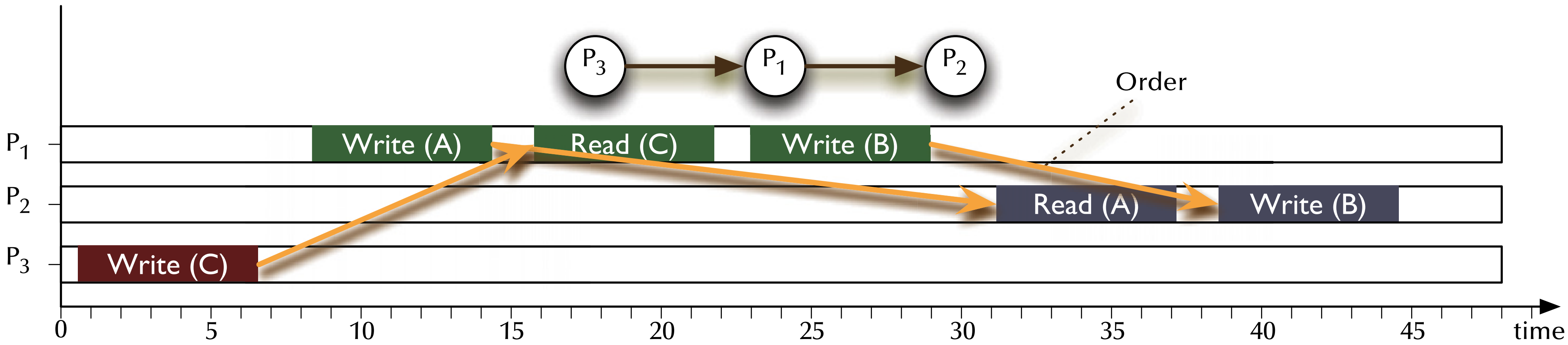
Serializability



- 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.

Distributed Systems

Serializability

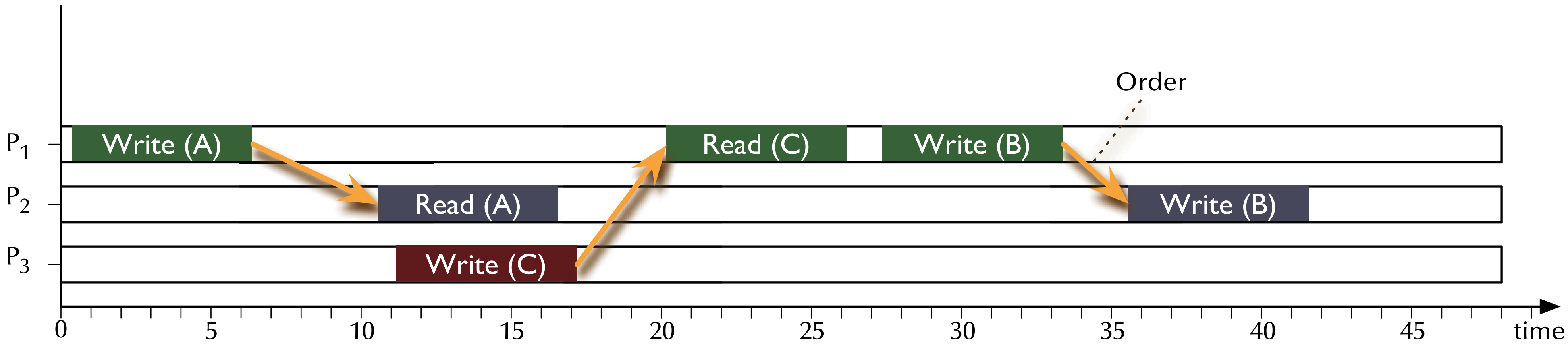


- Three conflicting pairs of operations with the same order of execution (pair-wise between processes).
- The order between processes also leads to a global order of processes.

☞ Serializable

Distributed Systems

Serializability

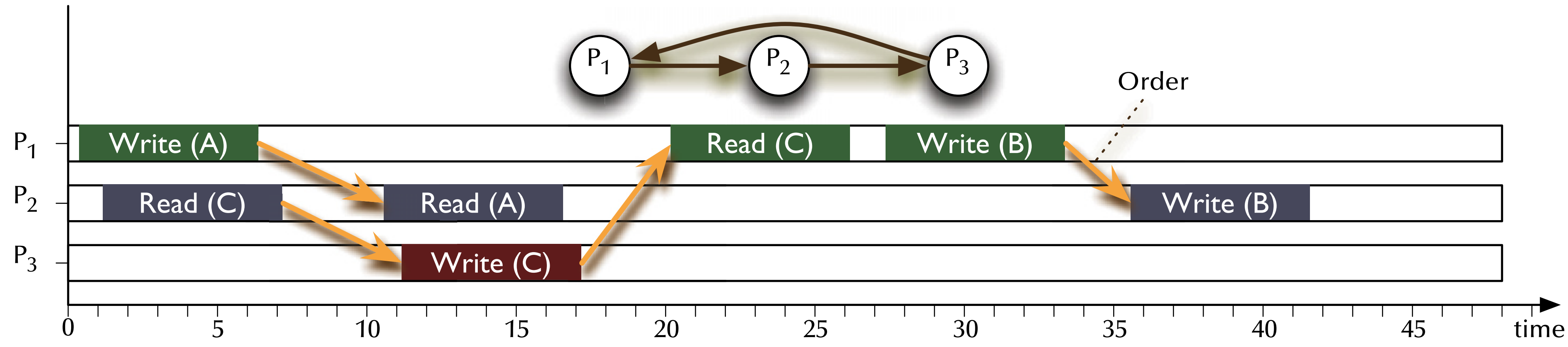


- Three conflicting pairs of operations with the same order of execution (pair-wise between processes).
- The order between processes also leads to a global order of processes.

☞ Serializable

Distributed Systems

Serializability



- Three conflicting pairs of operations with the same order of execution (pair-wise between processes).
- The order between processes *does no longer lead to a global order* of processes.

☞ Not serializable

Distributed Systems

Achieving serializability

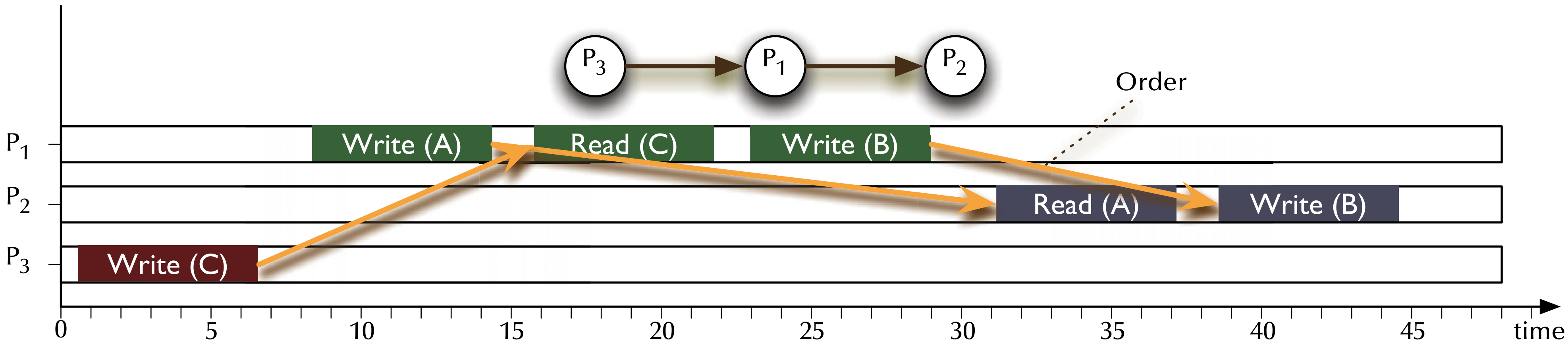
☞ For the **serializability** of *two* transactions it is **necessary** and **sufficient** for the *order* of their invocations of all conflicting pairs of operations to be *the same* *for all* the objects which are invoked by both transactions.

- 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*.

Distributed Systems

Serializability



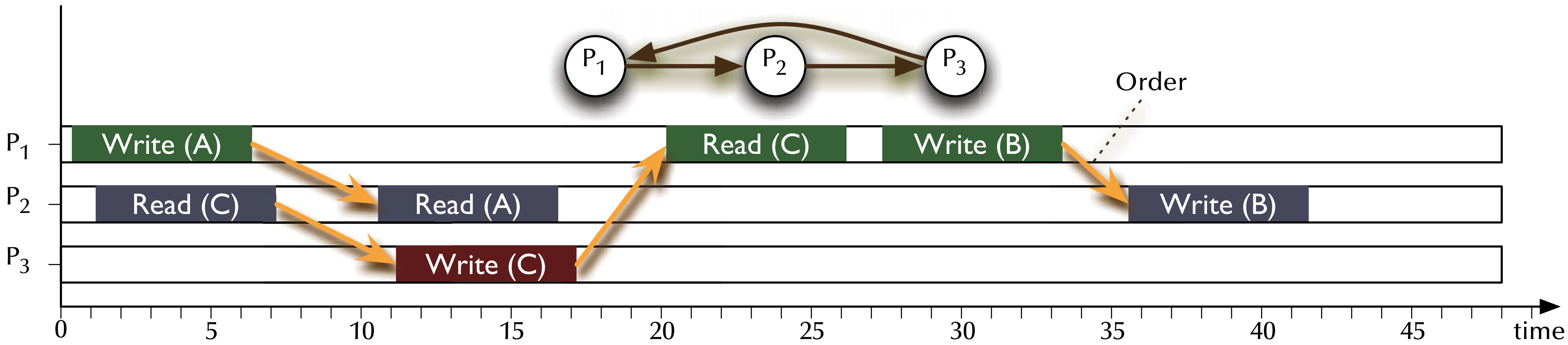
- Three conflicting pairs of operations with the same order of execution (pair-wise between processes).

☞ Serialization graph is acyclic.

☞ Serializable

Distributed Systems

Serializability



- Three conflicting pairs of operations with the same order of execution (pair-wise between processes).

☞ Serialization graph is cyclic.

☞ Not serializable

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.

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)

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.

Distributed Systems

Transaction schedulers – Optimistic control

Three sequential phases:

1. **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).
2. **Validate**:
   After local commit, **check** all occurred interleavings **for serializability**.
3. **Update or abort**:
   - 3a. 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*.
   - 3b. Otherwise: **destroy** shadow copies and **start over** with the failed transactions.

Distributed Systems

Transaction schedulers – Optimistic control

Three sequential phases:

How to create a consistent copy?

Full isolation and maximal concurrency!

1. **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).
2. **Validate**:
   After local commit, **check** all occurred interleavings **for serializability**.
3. **Update or abort**:

   How to update all objects consistently?

   3a. 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*.

   3b. Otherwise: **destroy** shadow copies and **start over** with the failed transactions.

Aborts happen after everything has been committed locally.

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.

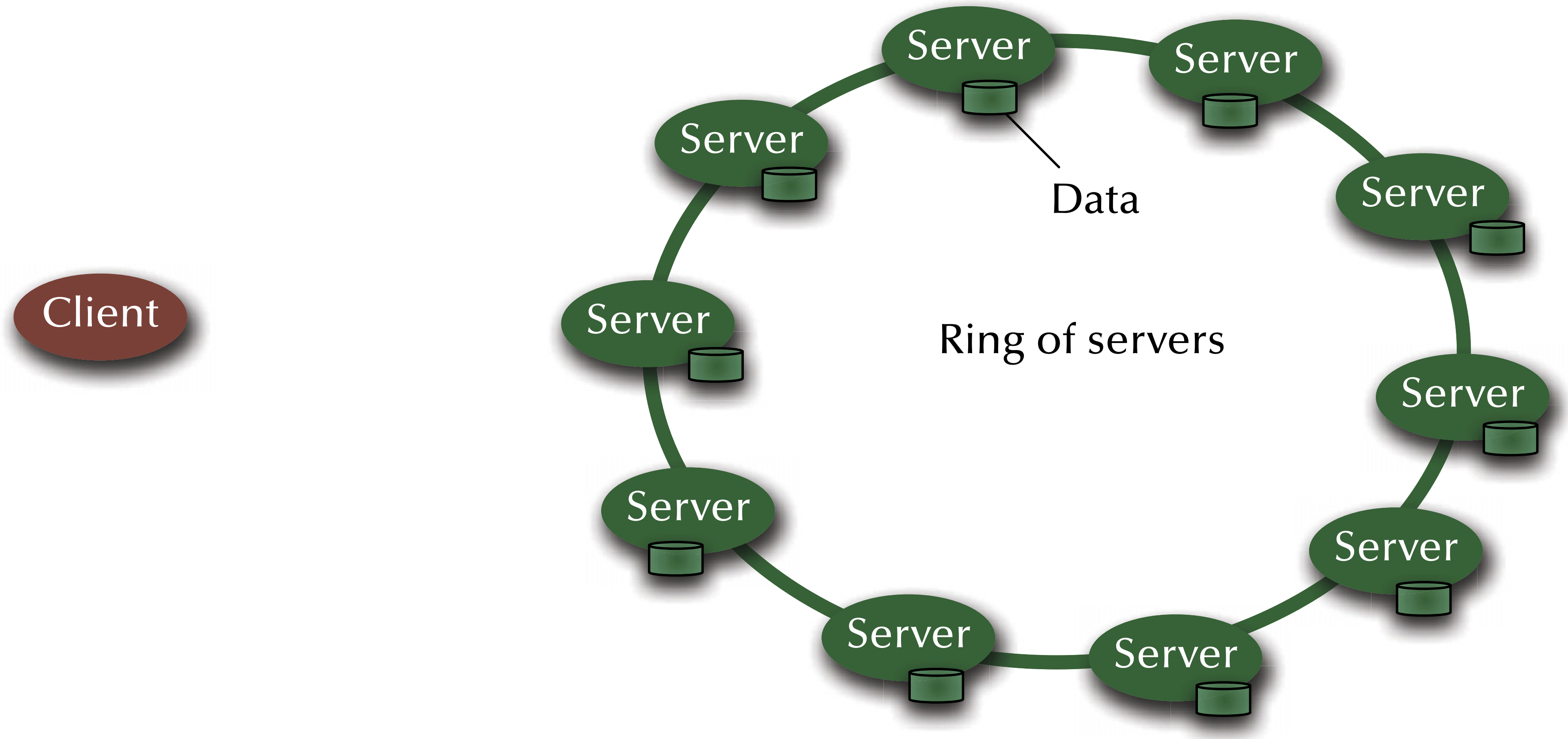
- **"Optimistic" methods**: ☞ aborts or commits at the very end
  Go ahead until a conflict is observed – then roll back.

☞ How to implement "commit" and "abort" operations in a distributed environment?

Distributed Systems

Two phase commit protocol

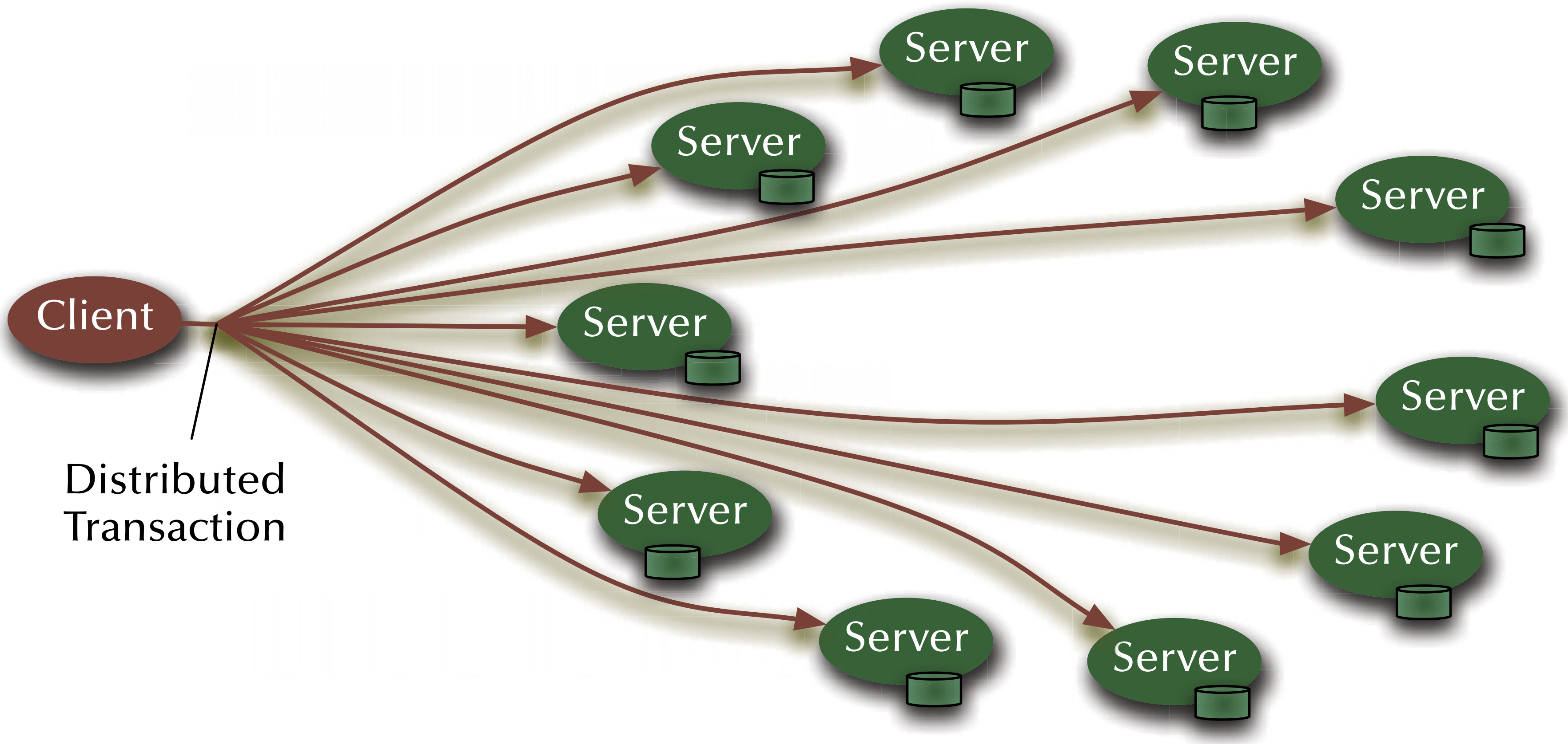
Start up (initialization) phase



Distributed Systems

Two phase commit protocol

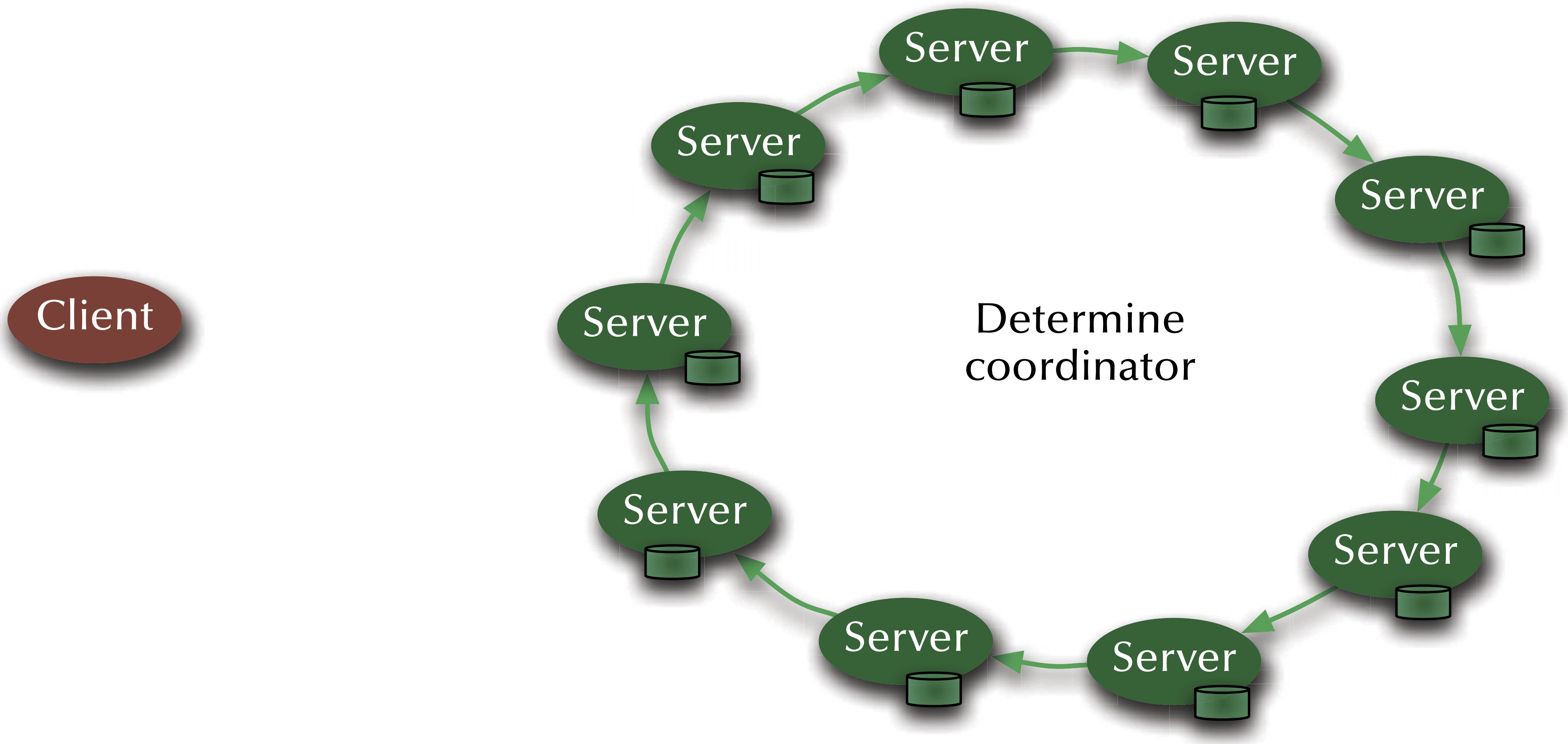
Start up (initialization) phase



Distributed Systems

Two phase commit protocol

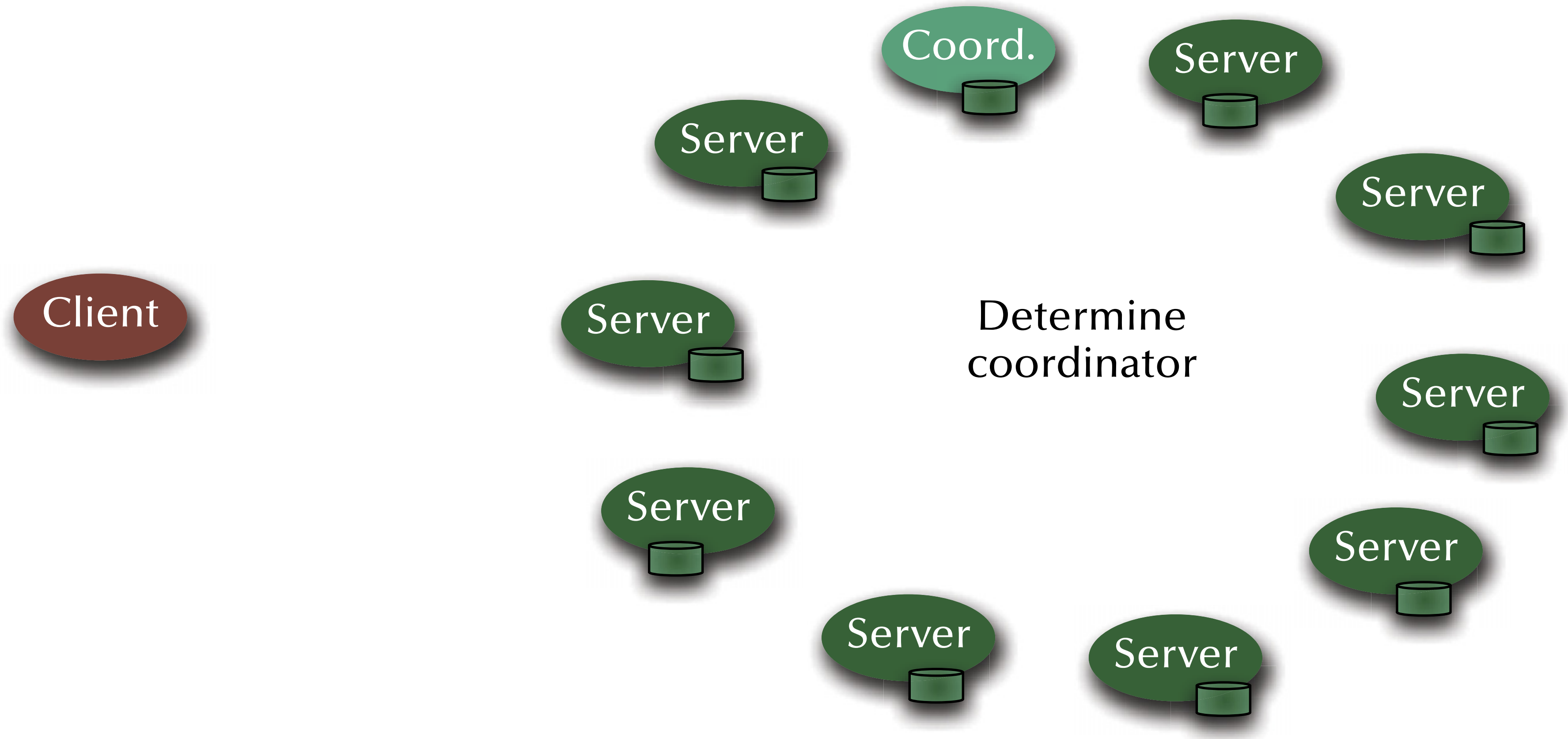
Start up (initialization) phase



Distributed Systems

Two phase commit protocol

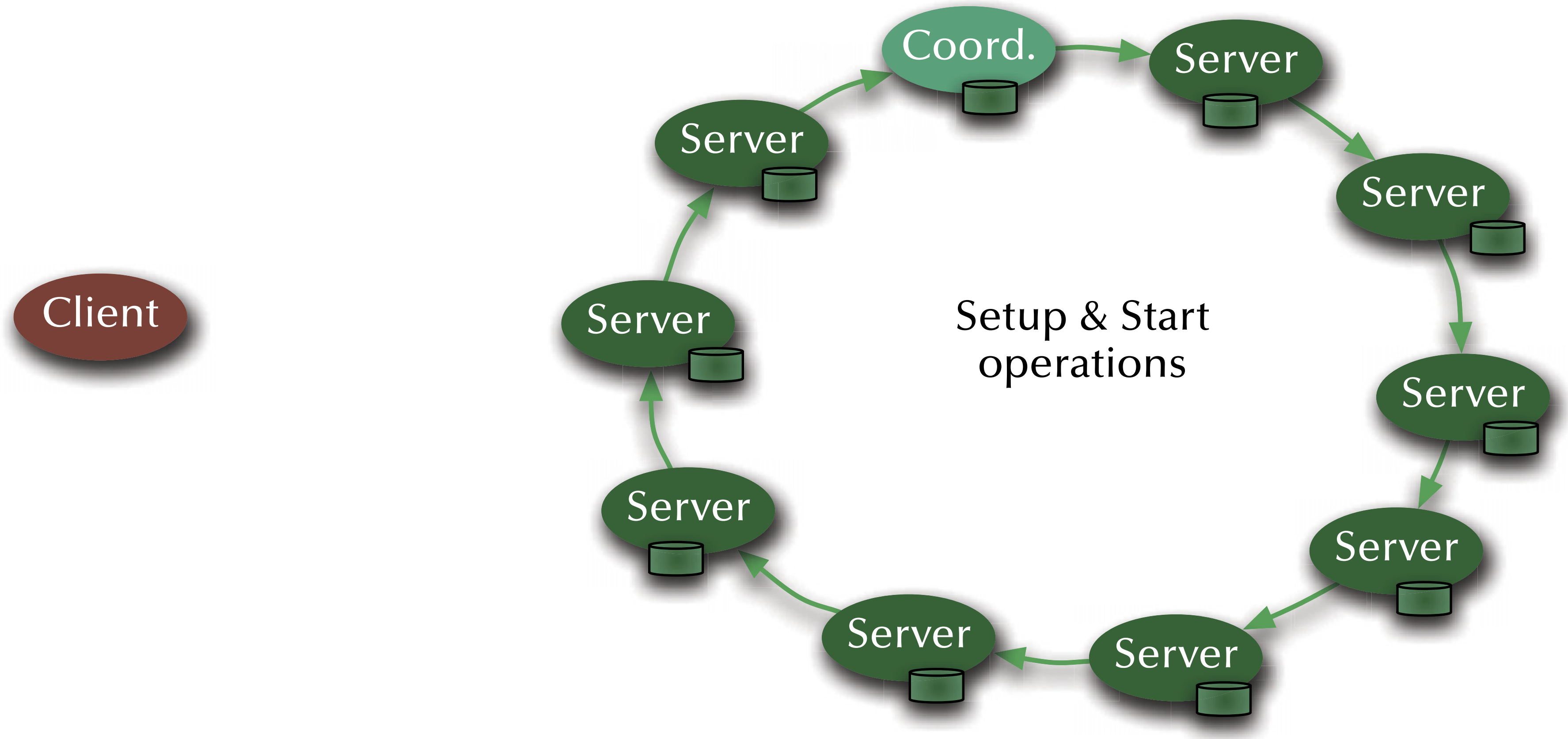
Start up (initialization) phase



Distributed Systems

Two phase commit protocol

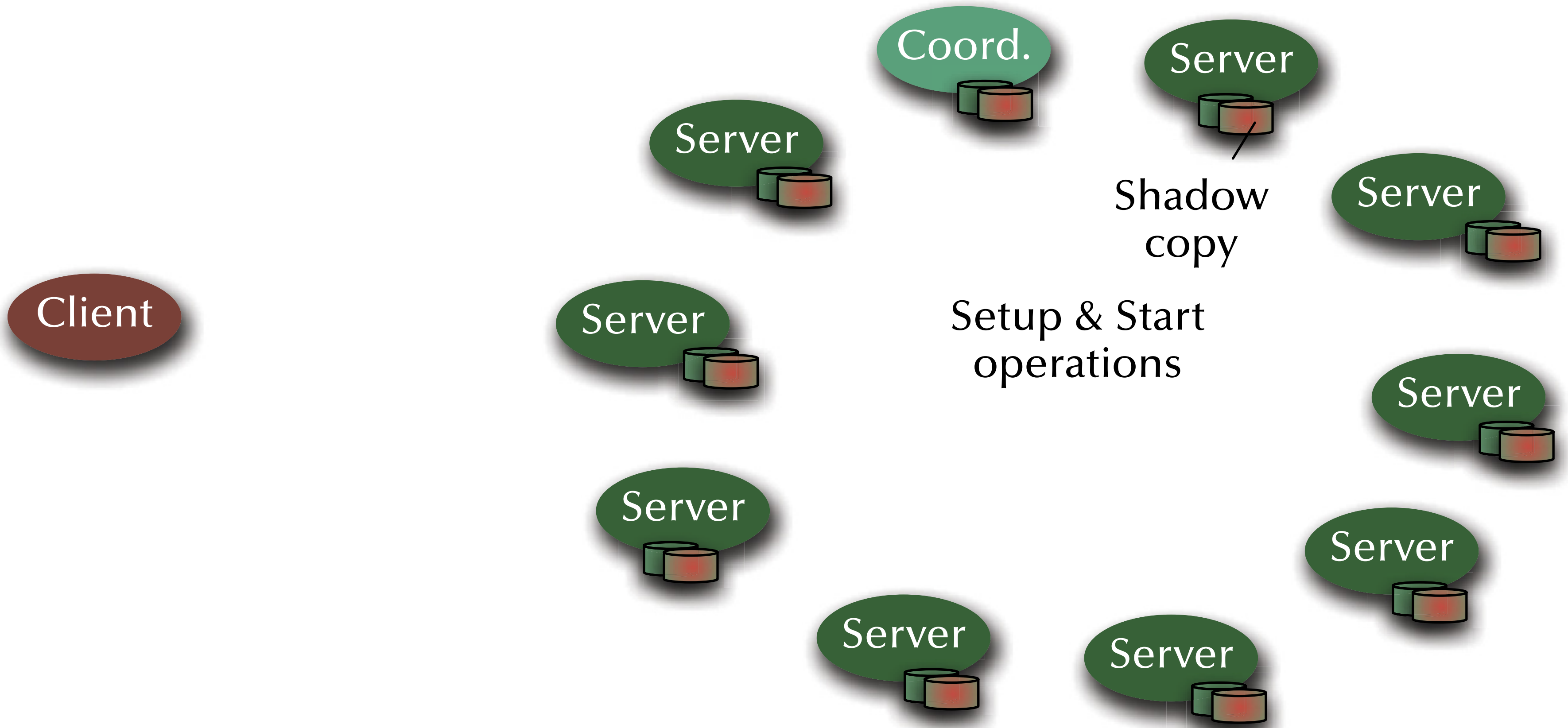
Start up (initialization) phase



Distributed Systems

Two phase commit protocol

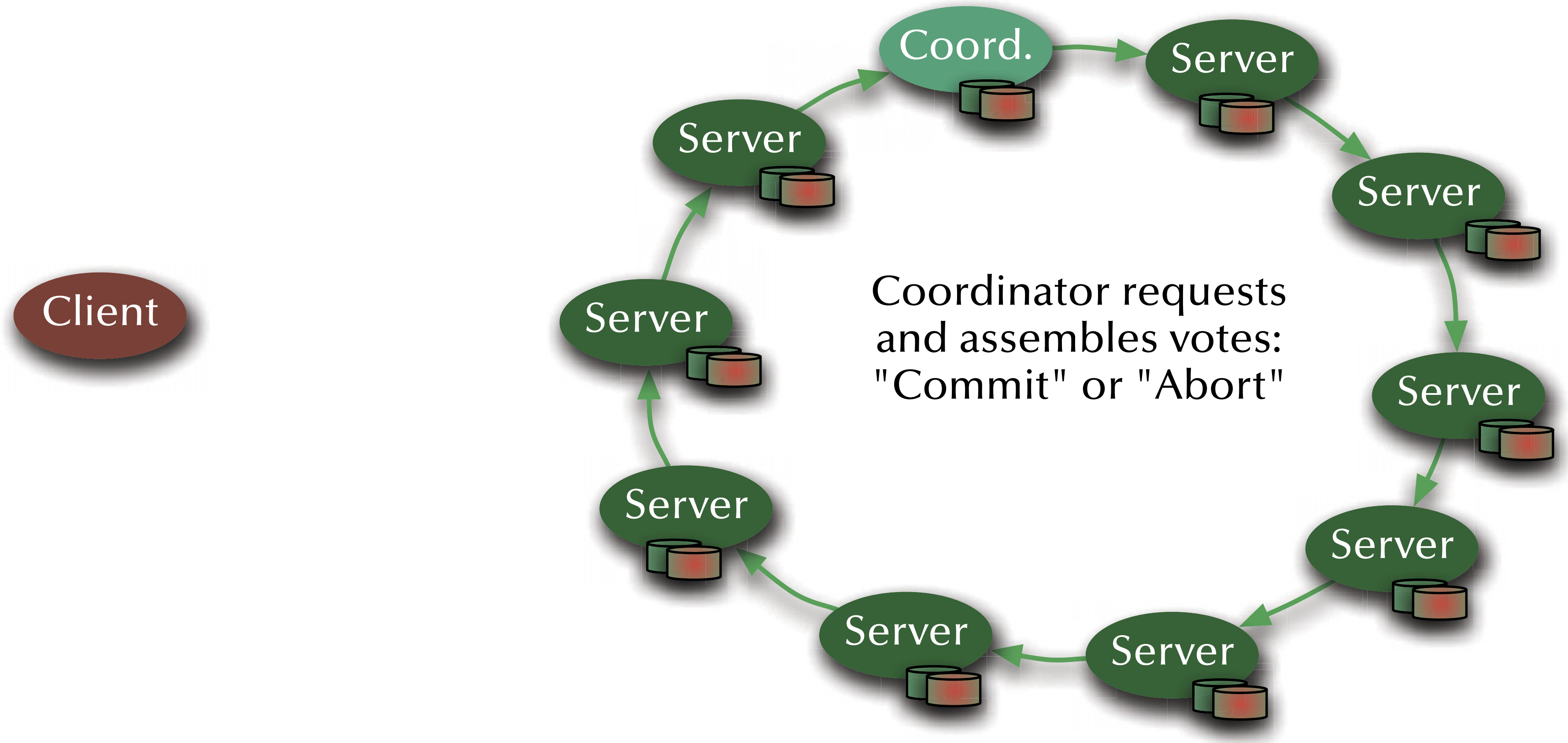
Start up (initialization) phase



Distributed Systems

Two phase commit protocol

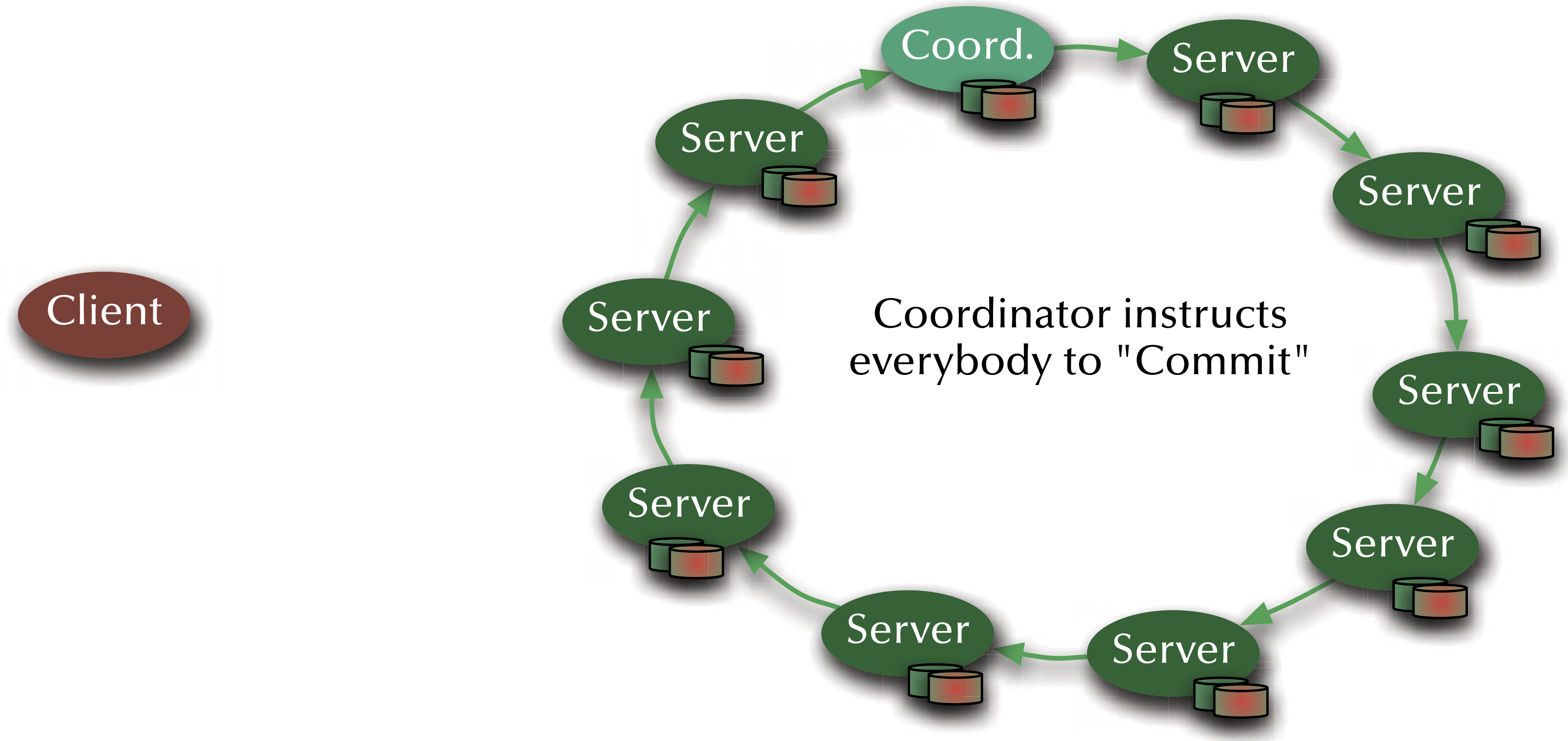
Phase 1: Determine result state



Distributed Systems

Two phase commit protocol

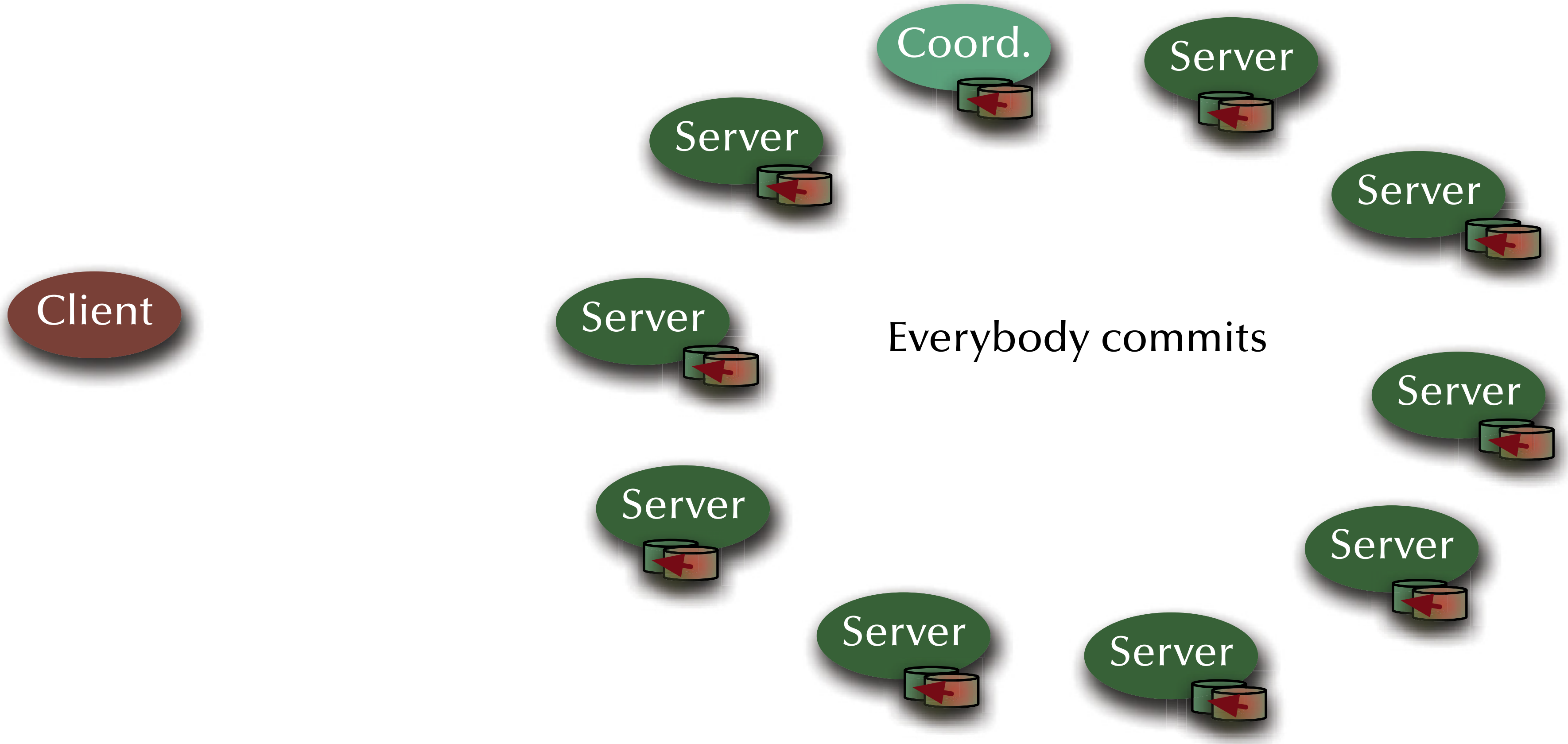
Phase 2: Implement results



Distributed Systems

Two phase commit protocol

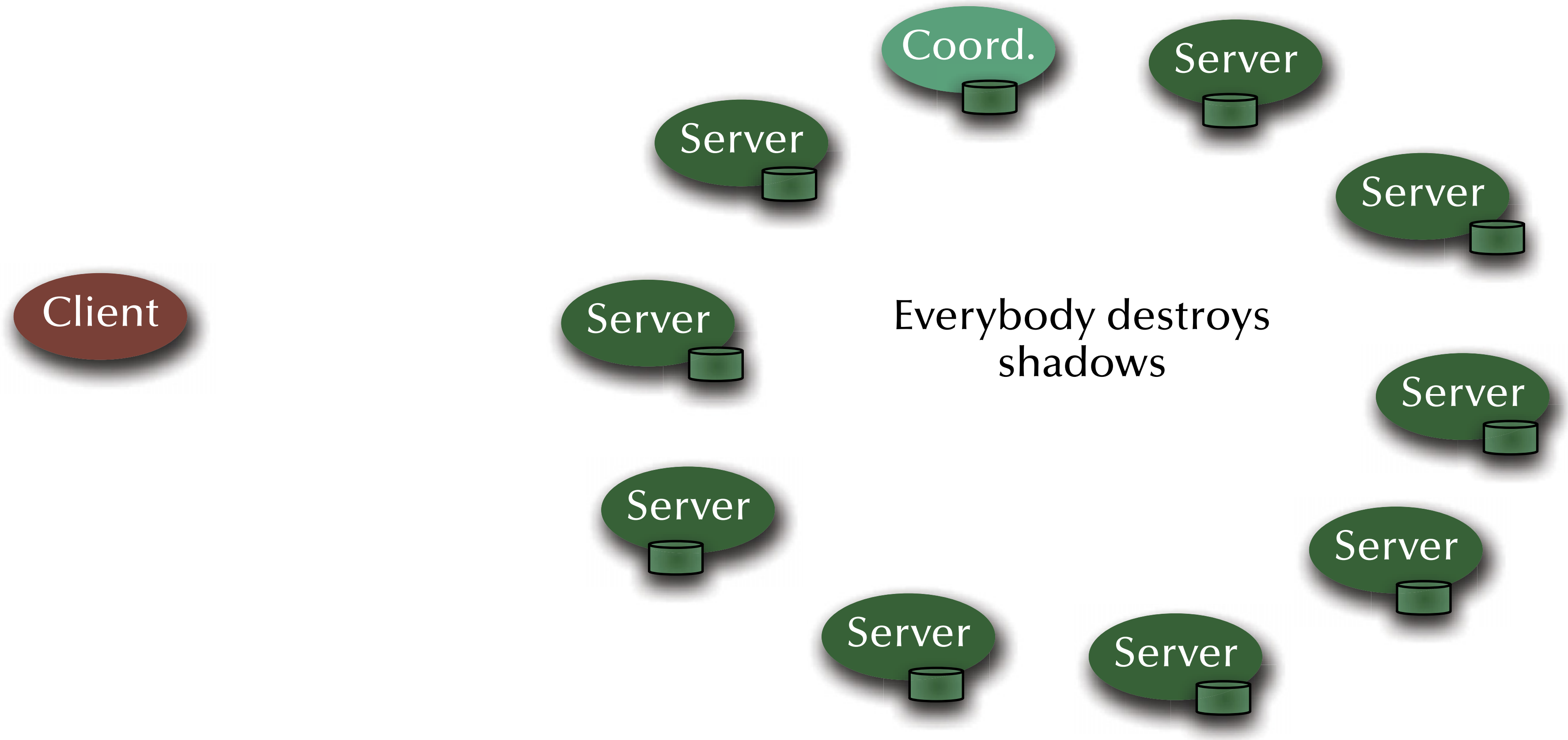
Phase 2: Implement results



Distributed Systems

Two phase commit protocol

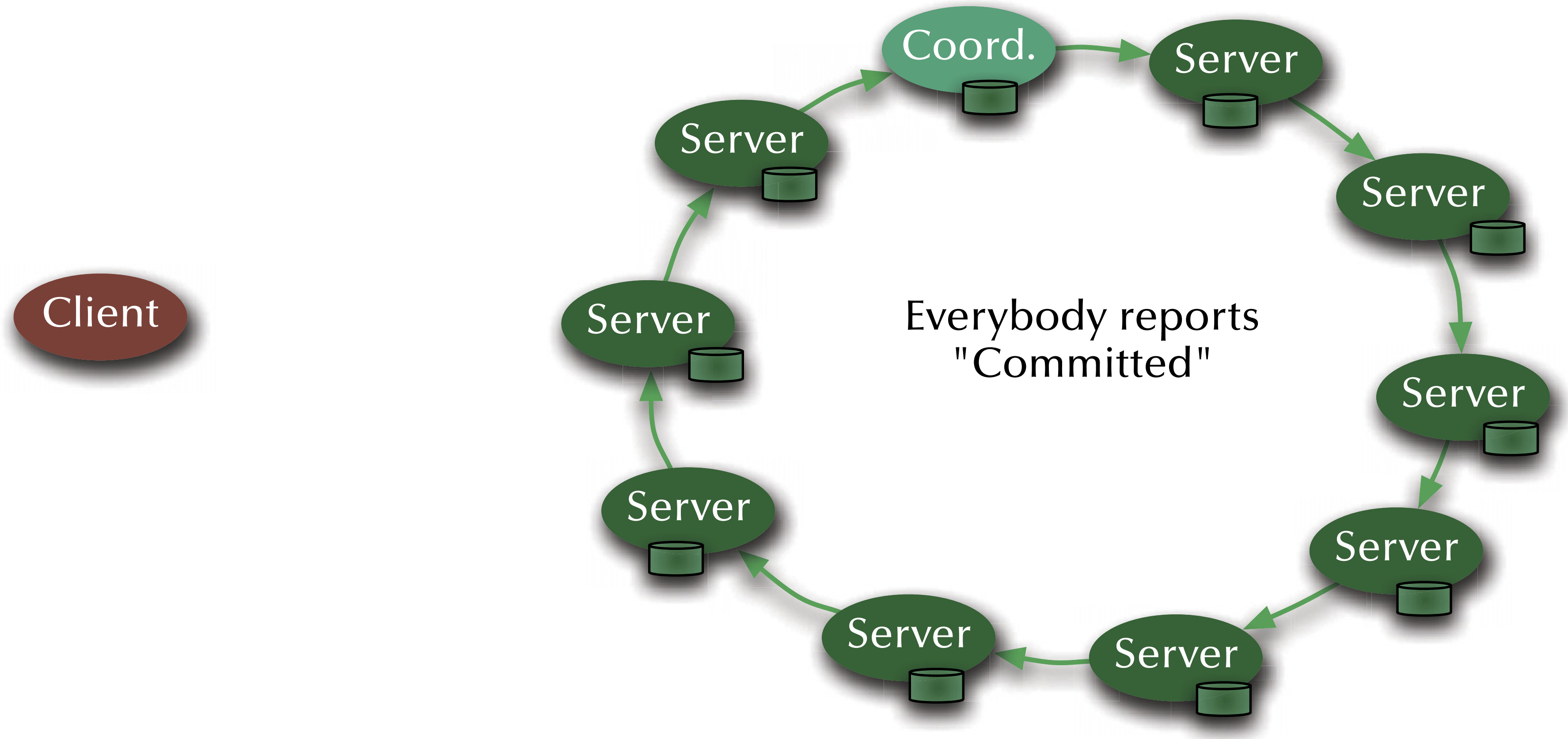
Phase 2: Implement results



Distributed Systems

Two phase commit protocol

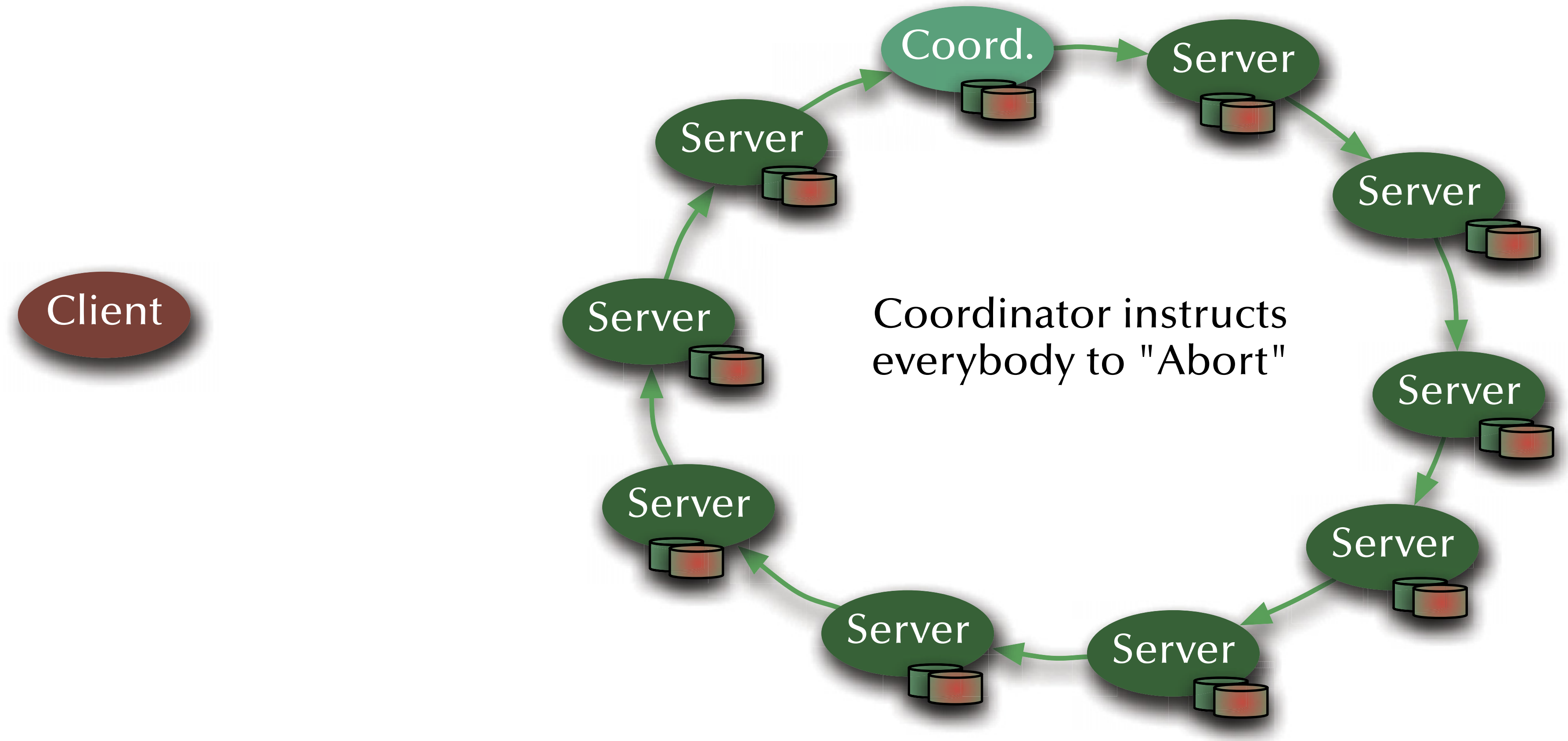
Phase 2: Implement results



Distributed Systems

Two phase commit protocol

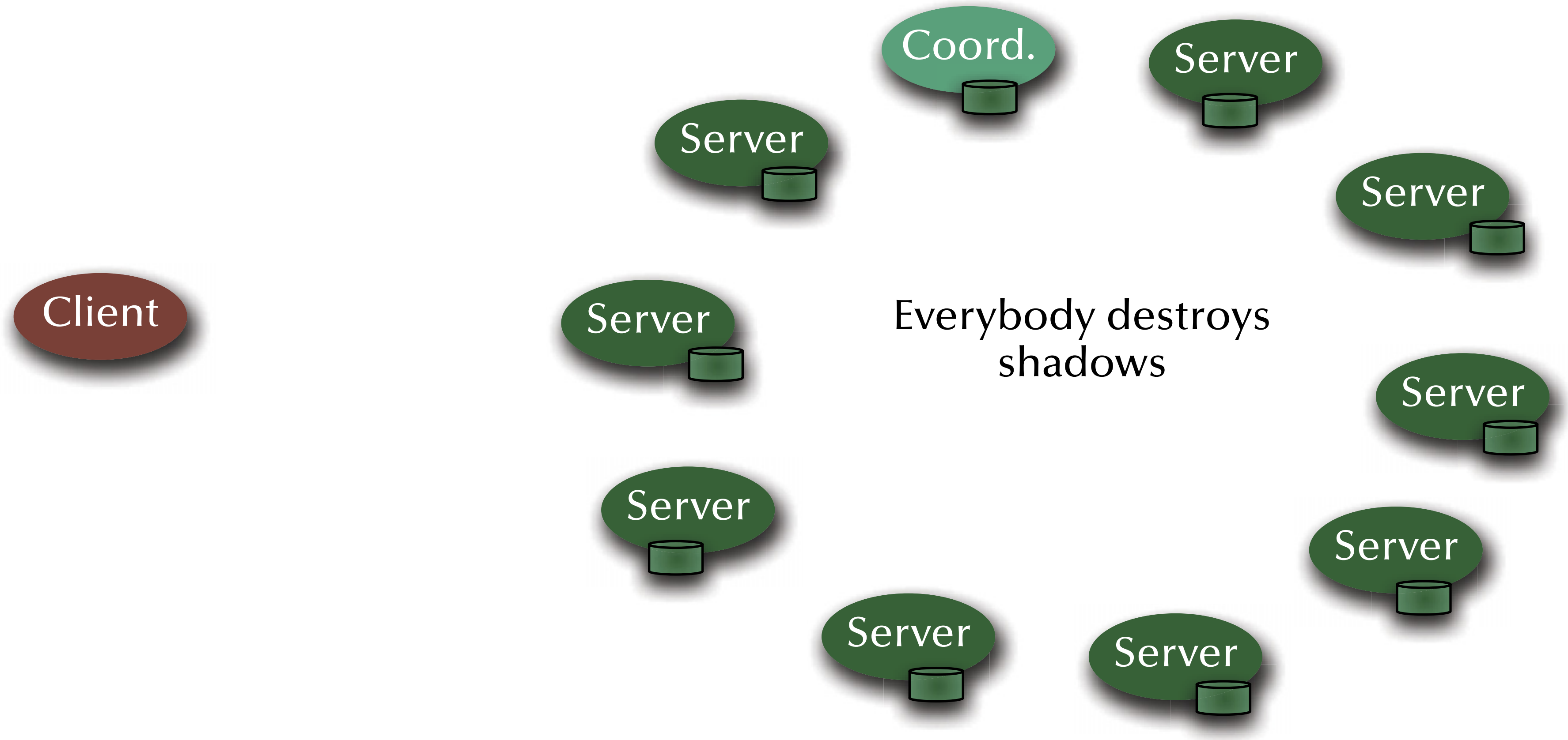
or Phase 2: Global roll back



Distributed Systems

Two phase commit protocol

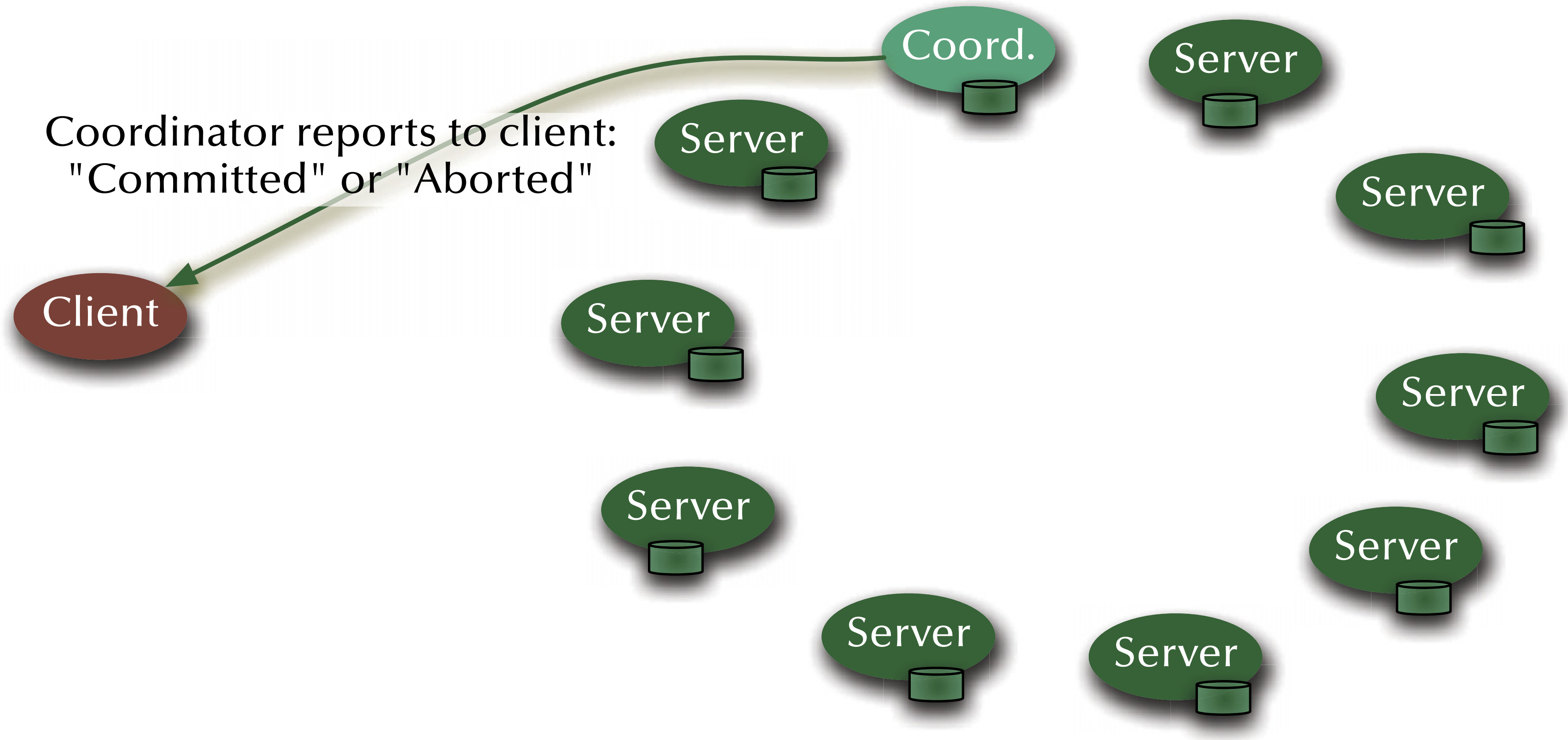
or Phase 2: Global roll back



Distributed Systems

Two phase commit protocol

Phase 2: Report result of distributed transaction



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, …)

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.

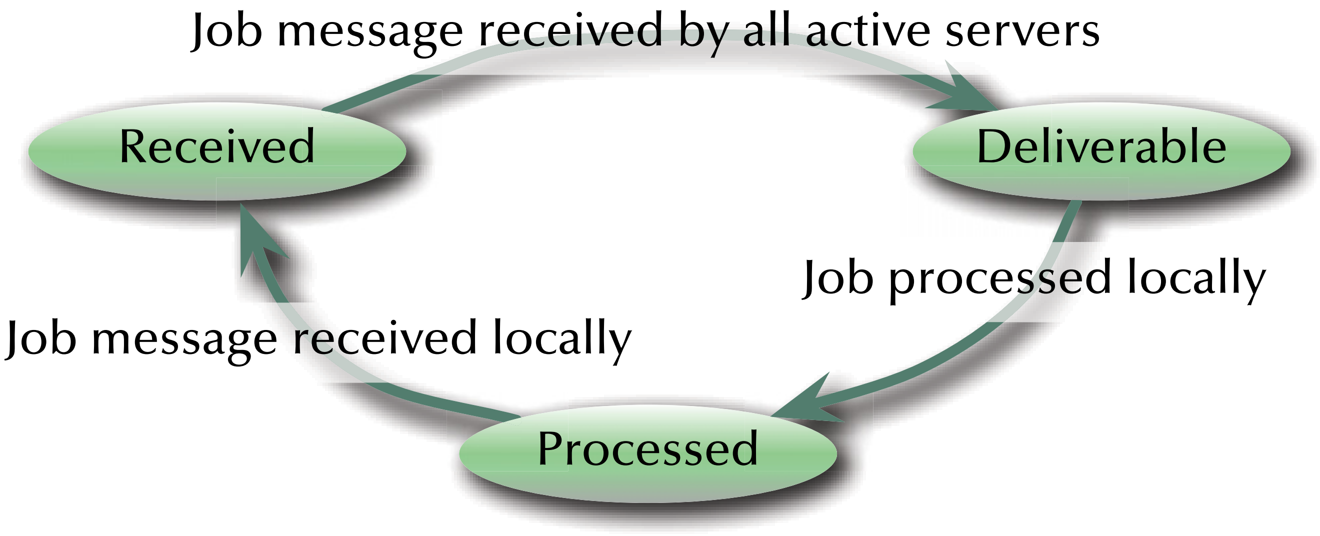
- The server is described fully by the current state and the sequence of messages received.
  ☞ State machines: we have to implement consistent state adjustments (re-organization) and consistent message passing (order needs to be preserved).

[Schneider1990]

Distributed Systems

Redundancy (replicated servers)

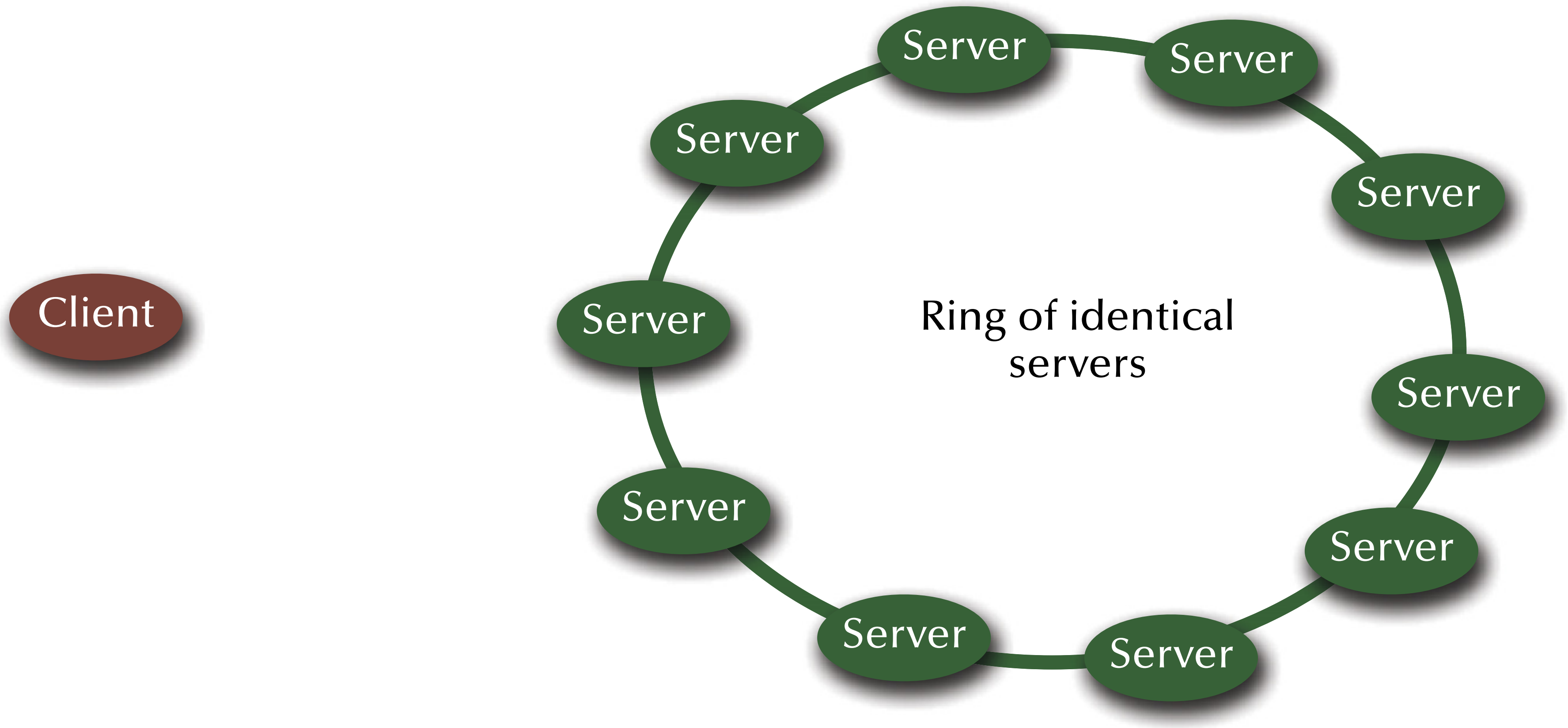
Stages of each server:



Distributed Systems

Redundancy (replicated servers)

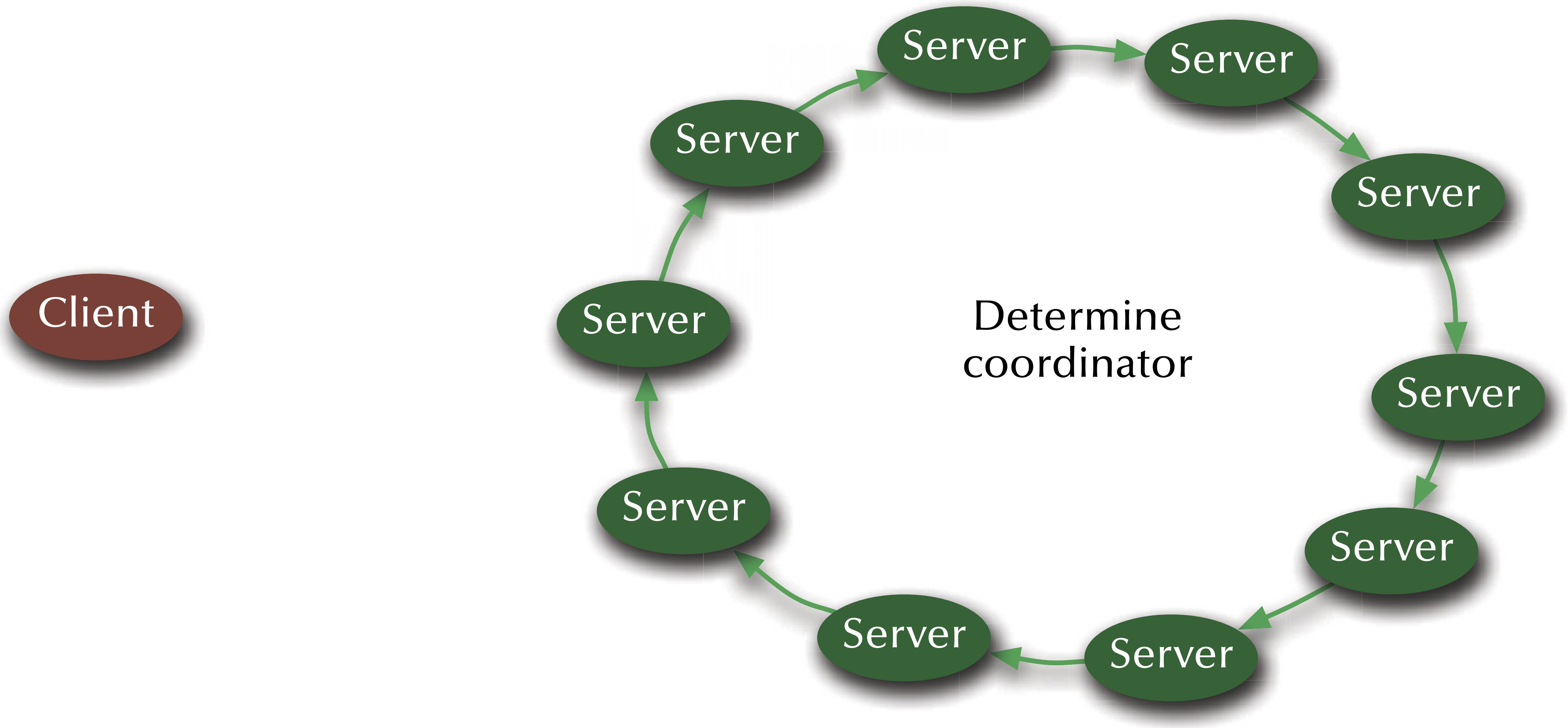
Start-up (initialization) phase



Distributed Systems

Redundancy (replicated servers)

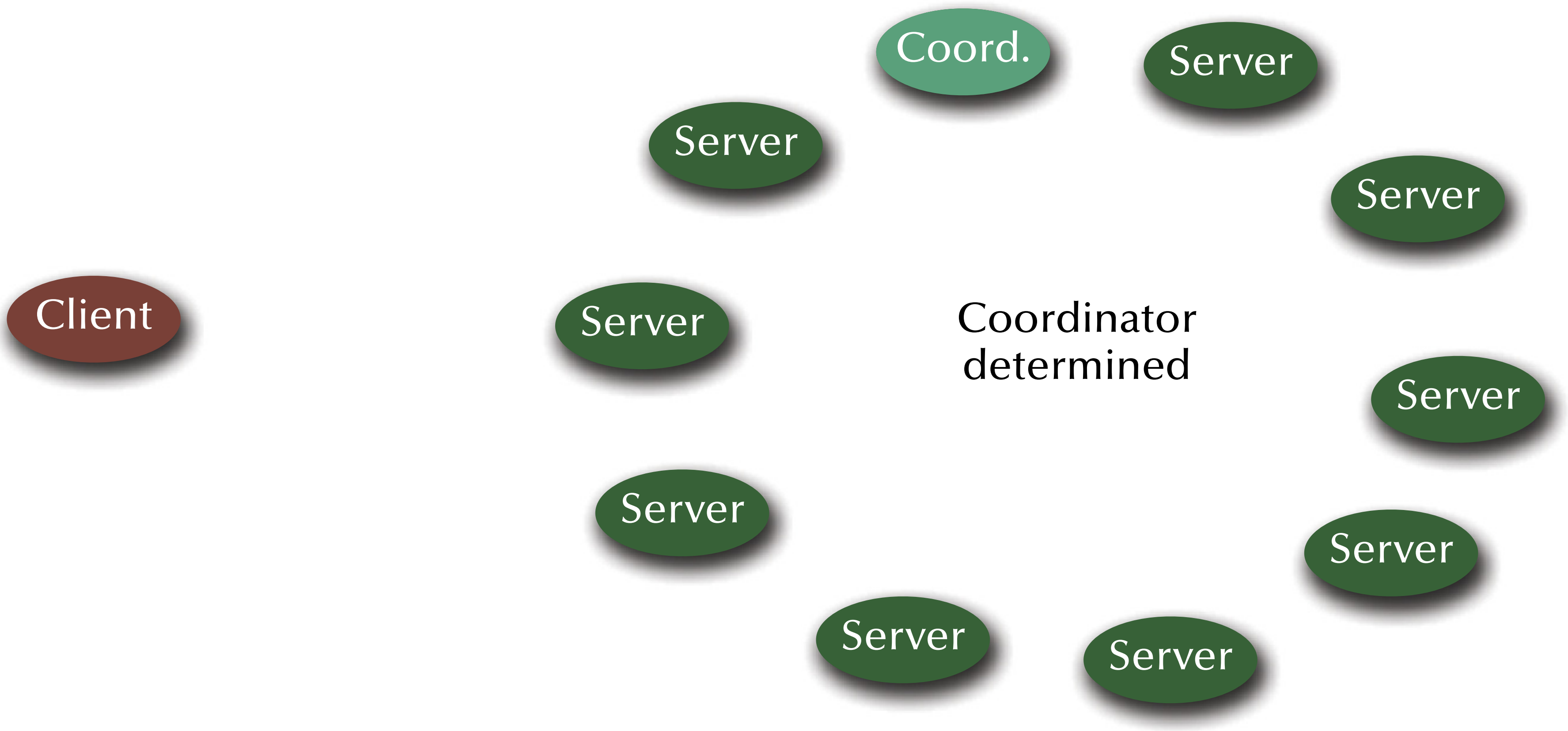
Start-up (initialization) phase



Distributed Systems

Redundancy (replicated servers)

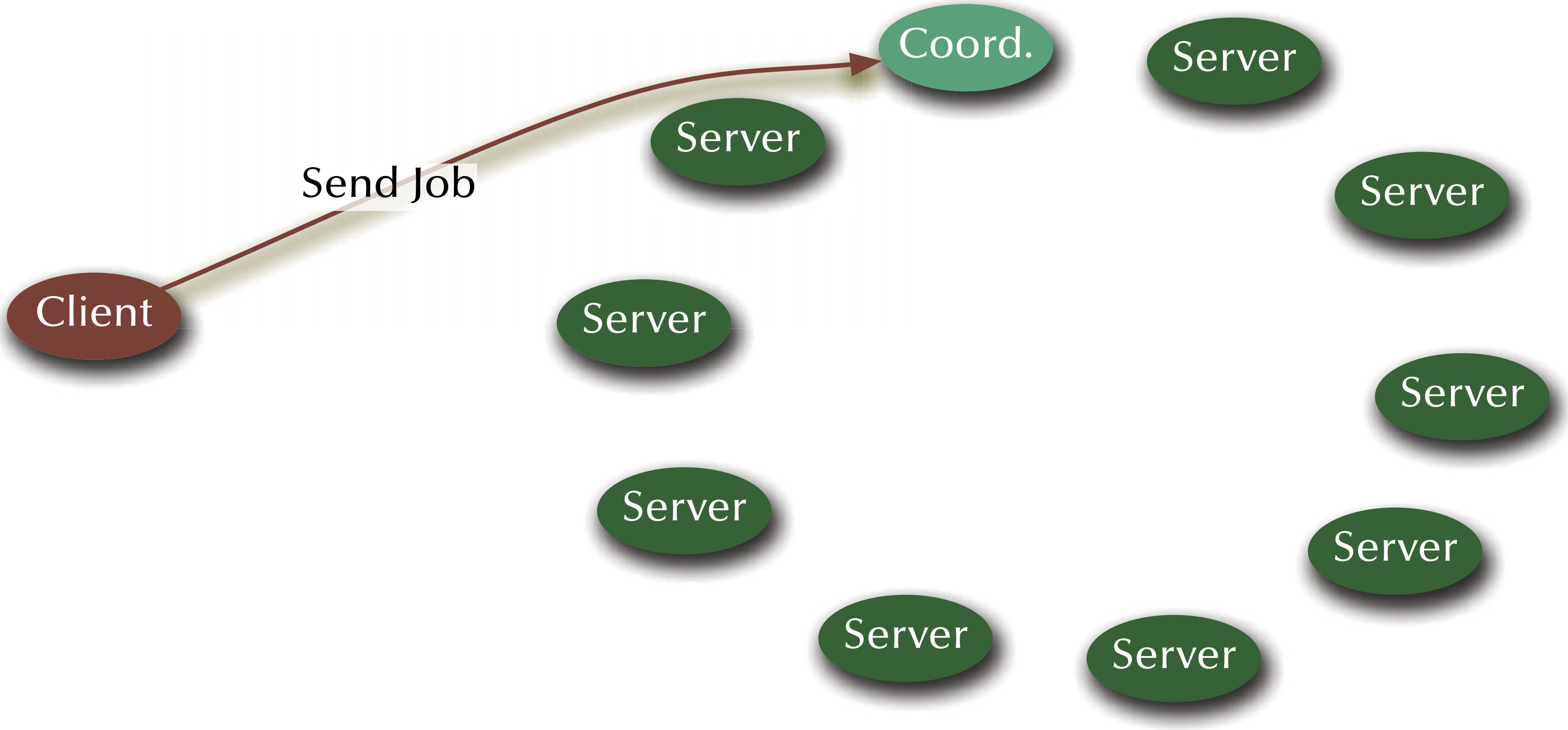
Start-up (initialization) phase



Distributed Systems

Redundancy (replicated servers)

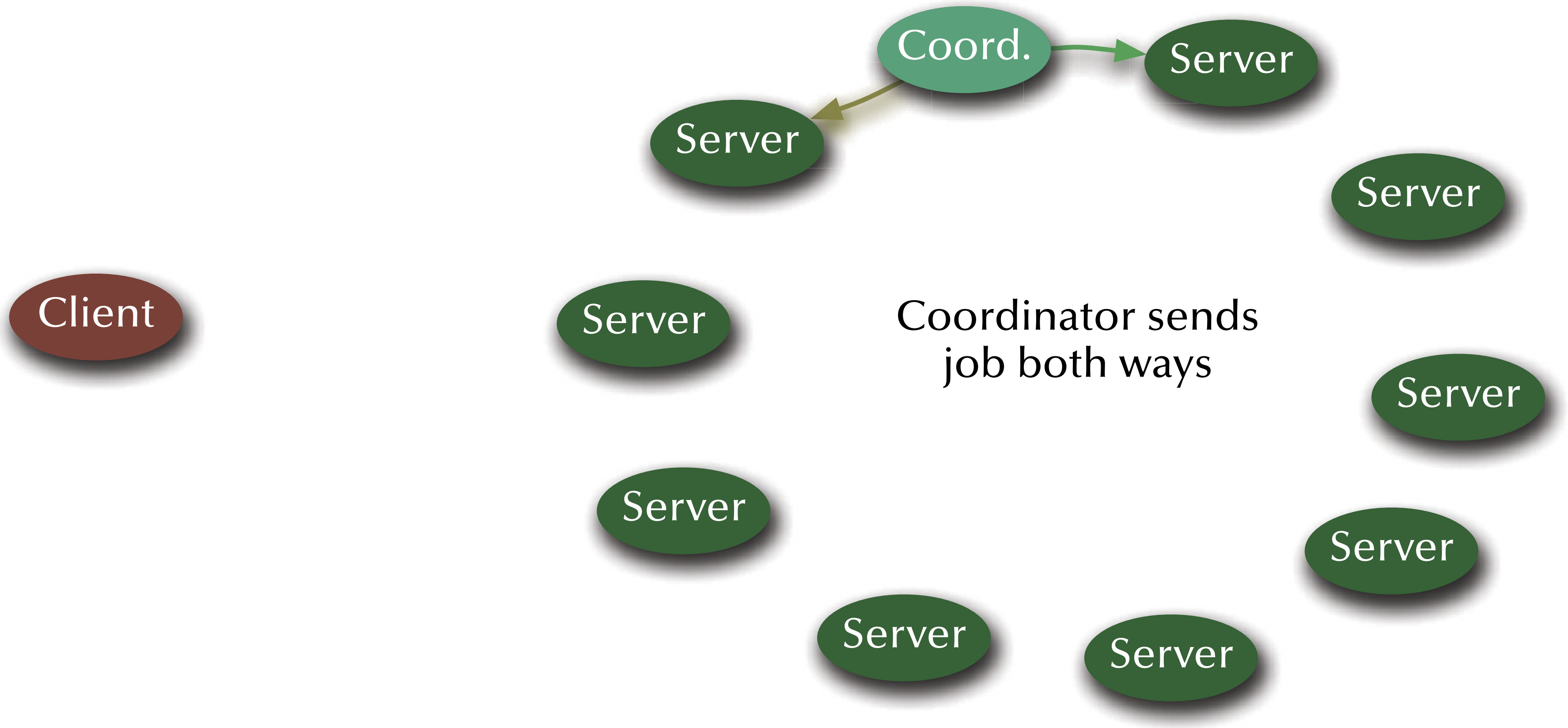
Coordinator receives job message



Distributed Systems

Redundancy (replicated servers)

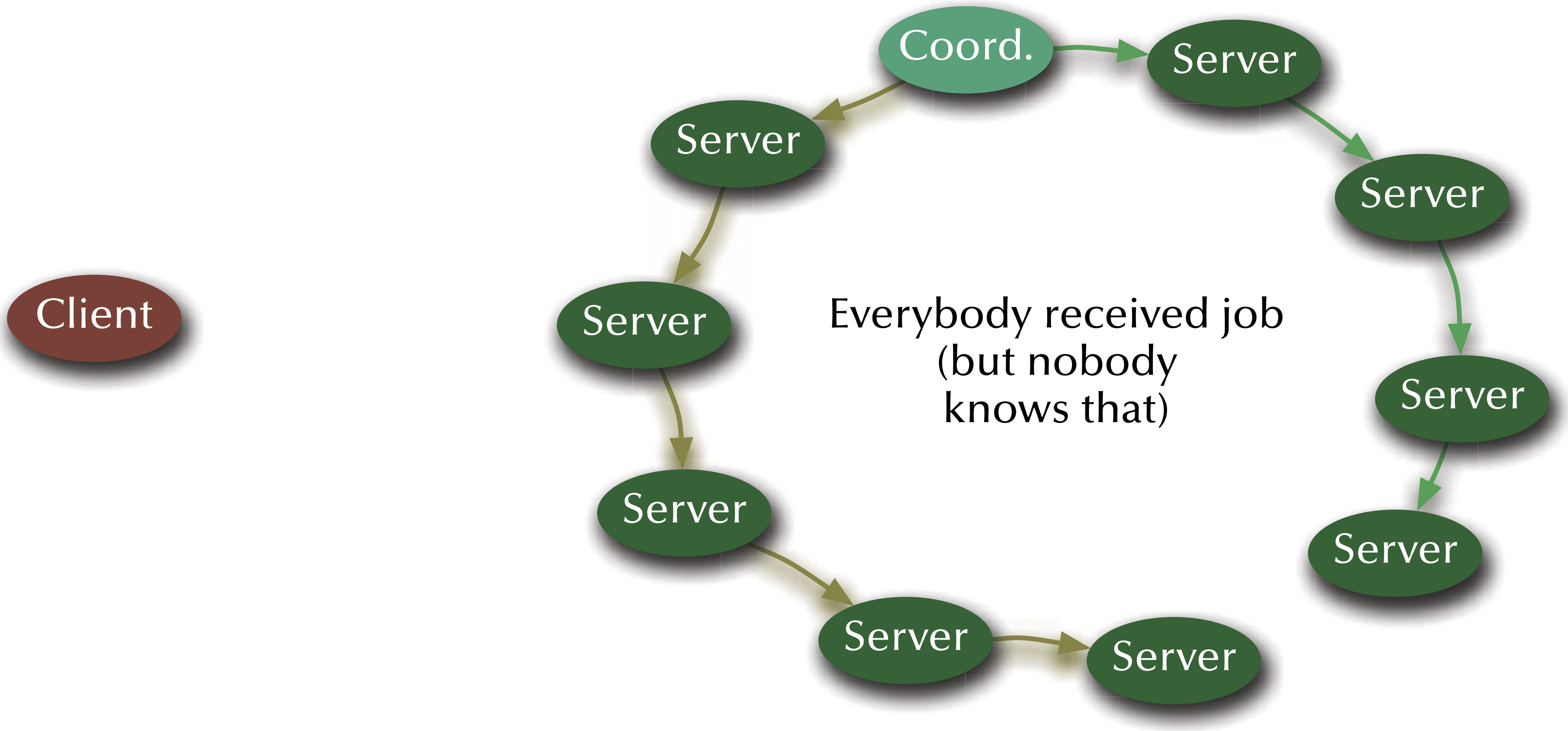
Distribute job



Distributed Systems

Redundancy (replicated servers)

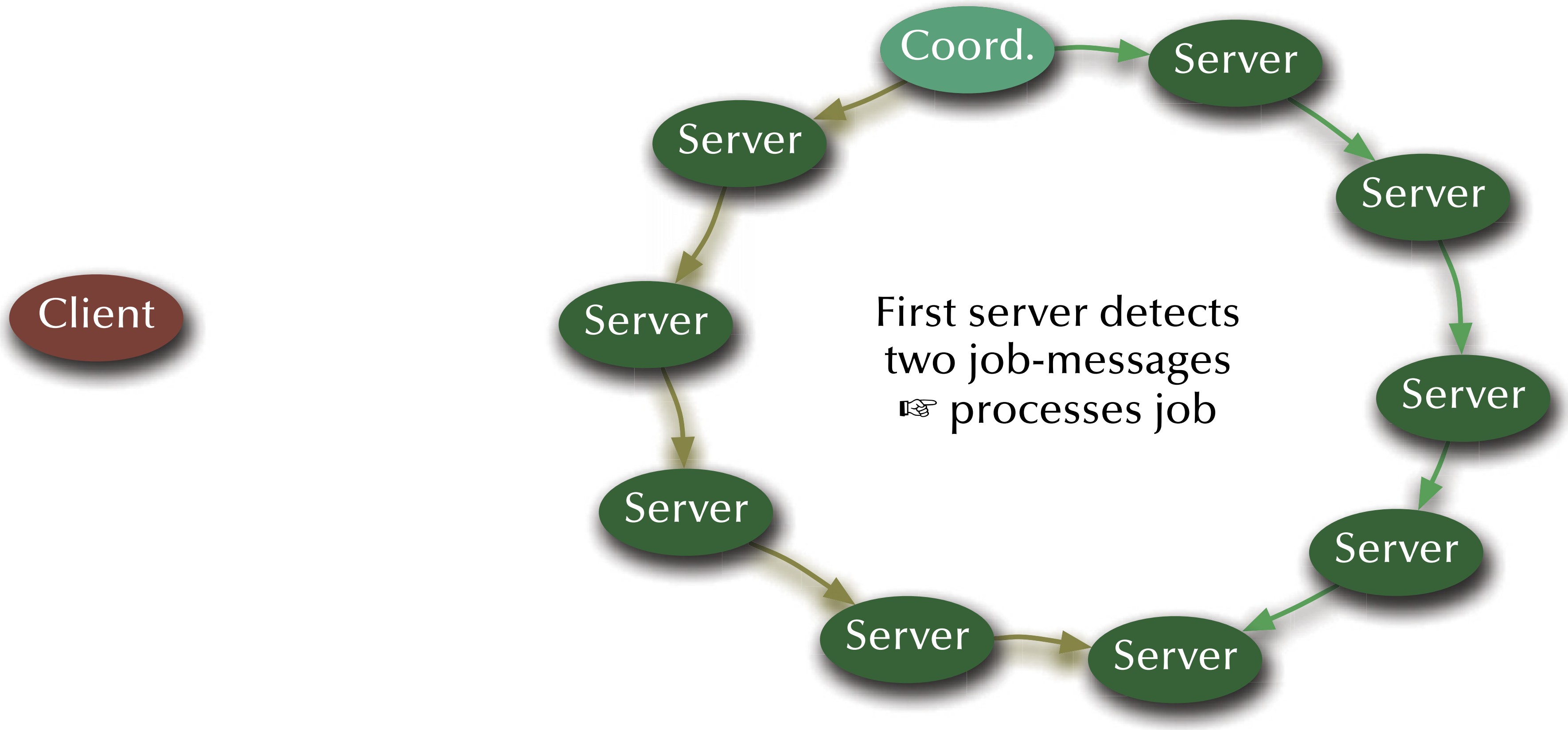
Distribute job



Distributed Systems

Redundancy (replicated servers)

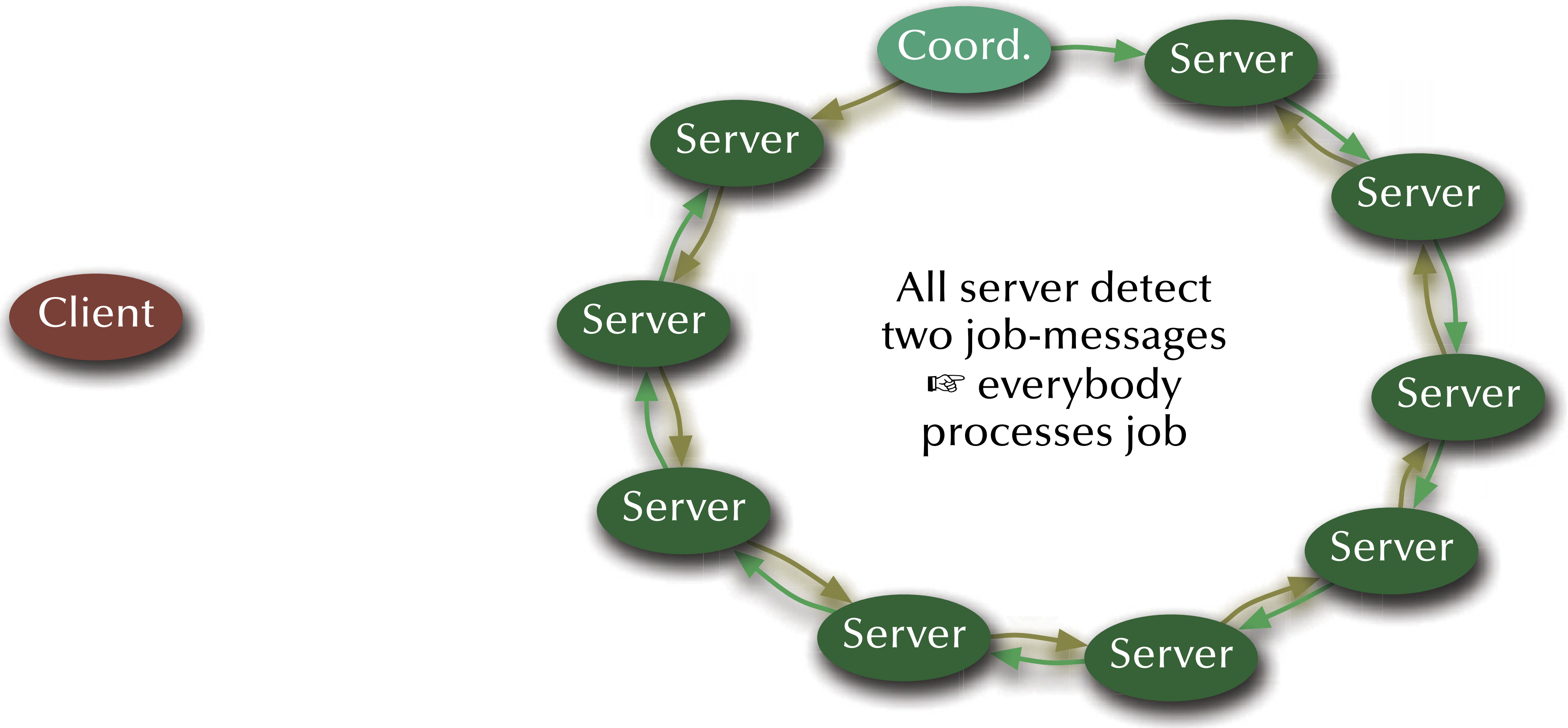
Processing starts



Distributed Systems

Redundancy (replicated servers)

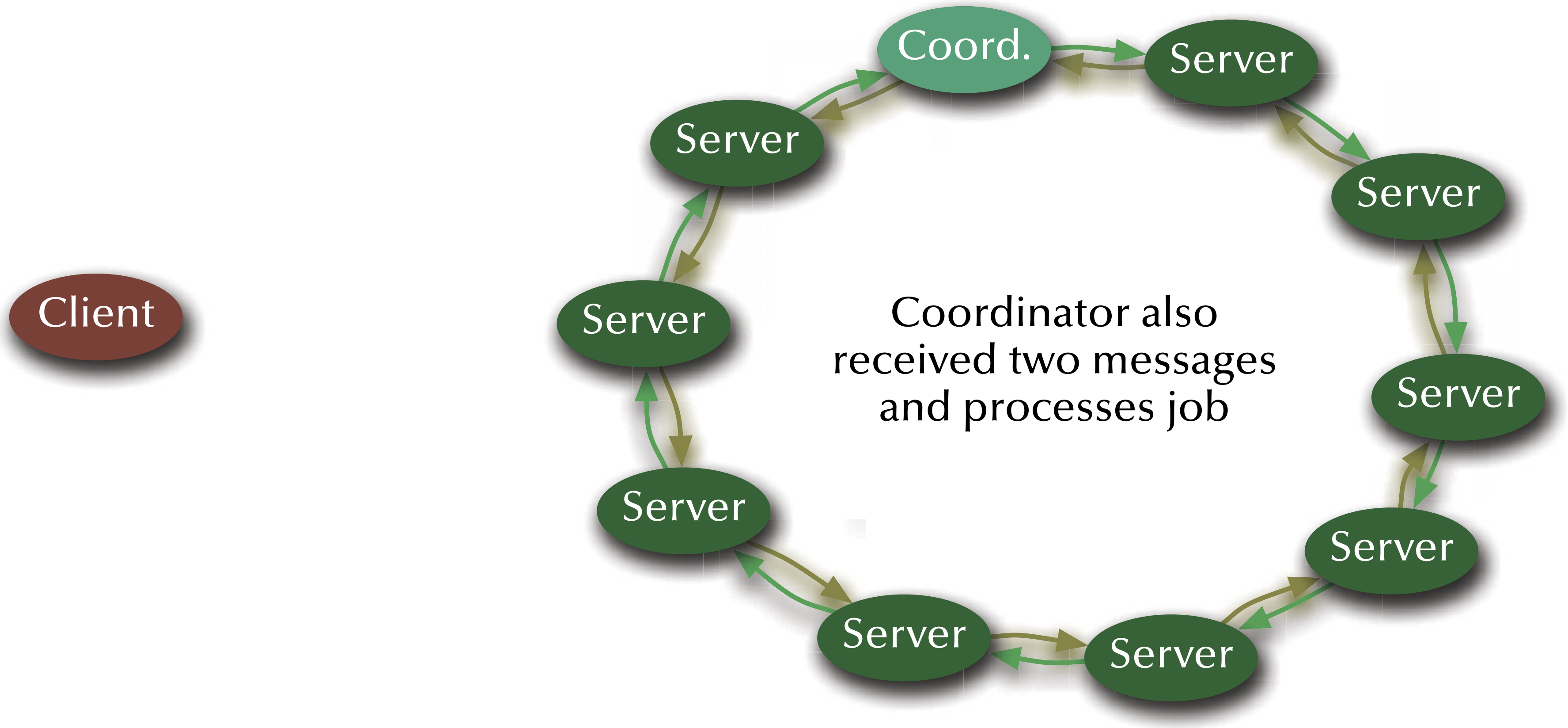
Everybody (besides coordinator) processes



Distributed Systems

Redundancy (replicated servers)

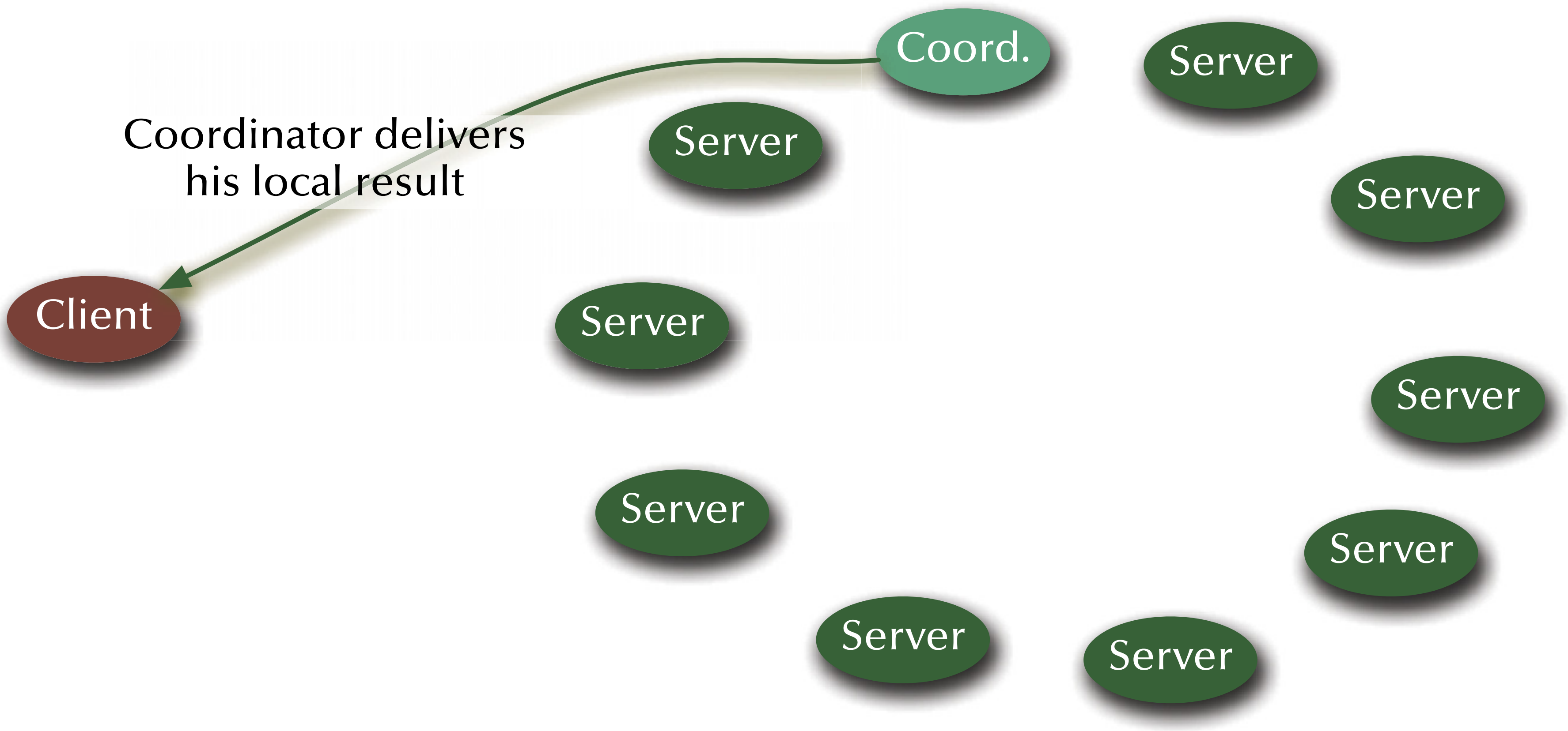
Coordinator processes



Distributed Systems

Redundancy (replicated servers)

Result delivery



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’

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)

9

Summary

Uwe R. Zimmer - The Australian National University

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
- **First examples of concurrent programming languages:**
  - Explicit concurrency: e.g. Ada2005, Chapel, X10
  - Implicit concurrency: functional programming – e.g. Lisp, Haskell, Caml, Miranda

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

Summary

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

Summary

Non-Determinism

- **Non-determimism 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

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

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

Summary

Architectures

- **Hardware architectures - from simple logic to supercomputers**
    - logic, CPU architecture, pipelines, out-of-order execution, multithreading, …
- **Operating systems**
    - basics: context switch, memory management, IPC
    - structures: monolithic, modular, layered, μkernels
    - UNIX, POSIX
- **Concurrency in languages**
    - some examples: CSP, Occam, Go, Chapel, Ada

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)

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 Ada95 / Unix / Posix / Occam / sockets reference manual page by page.