Concurrent & Distributed Systems 2005

Uwe R. Zimmer & Alistair Rendell
The Australian National University
what is offered here?

Fundamentals & Overview
as well as perspectives, paths, methods, implementations

of/into/for/about
Concurrent & Distributed Systems
who could be interested in this?

anybody who …

... works 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 know what you do not know about concurrent systems
who are these people? – introduction

This course will be given by

Uwe R. Zimmer
and
Alistair Rendell
how will this all be done?

Lectures:

- 3 per week ... all the nice stuff and theory
  Monday, 14:00 (CHEM-T1); Tuesday 14:00 (FSTY-T); Wednesday 14:00 (RSBS-T)

Laboratories:

- 2 hours per week ... all the rough stuff and practice
  Monday 15:00; Tuesday 15:00; Wednesday 11:00 – all in CSIT N113/N114
  laboratory-enrolment: https://cs.anu.edu.au/streams/

Resources:

- introduced in the lectures and collected on the course page:
  ... as well as schedules, slides, sources, etc. pp. ... keep an eye on this page!

Assessment:

- exam at the end of the course (70%) plus two assignments (15% each), and mid-term check (0%)
Useful Literature

[Ben-Ari90]
M. Ben-Ari
*Principles of Concurrent and Distributed Programming*
1990
Prentice-Hall,

Main technical textbook for this course.

- Many algorithms and basic concepts will be found here

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

Major practical extensions

- Some practical examples
- Less technical aspects

* references for specific aspects of the course will be given at appropriate places
### Lectures 2005

[number of lectures] - total: ≈28

#### 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


3.1. Shared memory synchronization [2]
   - Semaphores
   - Cond. variables
   - Conditional critical regions
   - Monitors
   - Protected objects

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

   - in concurrent systems

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


6.1. Safety properties
   - Examples for 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
   - Examples

7. Architectures for CDS [3]

7.1. Academic
   - CSP
   - occam

7.2. Production
   - Ada95
   - JAVA

7.3. Historical roots: UNIX [1]
   - UNIX processes
   - UNIX communication schemes

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]

---

© 2005 Uwe R. Zimmer, The Australian National University
Concurrent & Distributed Systems

Laboratories & Assignments 2005

[number of labs] - total: 9

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 UNIX [3]
   3.1. UNIX process creation, termination
   3.2. UNIX process communication
       • Pipes
       • Sockets

Assignments

1. Concurrent programming [15%]
   Ada95 programming task involving:
   • Mutual exclusion
   • Synchronization
   • Message passing

2. Concurrent programming in UNIX [15%]
   UNIX programming task involving:
   • Semaphores
   • Process communication

Examination & Checkpoints

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

2. Final exam – [70%]
   • Examining the complete lecture

Marking

The final mark is based on the assignments [30%] plus the final examination [70%]
Ada refresher course

Uwe R. Zimmer & Alistair Rendell
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

[Ada 95 Reference manual]
(see lab pages or web)
Ada95 is a **standardized** (ISO/IEC 8652:1995(E)) ‘general purpose’ language with **core** language primitives for

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

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

- additional real-time features, distributed programming, system-level programming, numeric, informations systems, safety and security issues.
Ada95

A crash course

... refreshing:

- specification and implementation (body) parts, basic types
- exceptions
- information hiding in specifications (‘private’)
- generic programming
- class-wide programming (‘tagged types’)
- abstract types and dispatching
Concurrent & Distributed Systems

Ada95

Basics

... introducing:

• specification and implementation (body) parts
• constants
• some basic types (integer specifics)
• some type attributes
• parameter specification
package Queue_Pack_Simple is

    QueueSize : constant Positive := 10;
    type Element is new Positive range 1_000..40_000;
    type Marker is mod QueueSize;
    type List is array (Marker'Range) of Element;
    type Queue_Type is record
        Top, Free : Marker := Marker'First;
        Elements : List;
    end record;

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

end Queue_Pack_Simple;
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;
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;
... introducing:

- exception handling
- enumeration types
- functional type attributes
A queue specification with proper exceptions

package Queue_Pack_Exceptions is

    QueueSize : constant Integer := 10;
    type Element is (Up, Down, Spin, Turn);
    type Marker is mod QueueSize;
    type List is array (Marker'Range) of Element;
    type Queue_State is (Empty, Filled);
    type Queue_Type is record
        Top, Free : Marker      := Marker'First;
        State     : Queue_State := Empty;
        Elements  : List;
    end record;

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

    Queueoverflow, Queueunderflow : exception;

end Queue_Pack_Exceptions;
A queue implementations with proper exceptions

package body Queue_Pack_Exceptions is

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

end Queue_Test_Exceptions;
Information hiding (private parts)

... introducing:

- private assignments and comparisons are allowed
- limited private entity cannot be assigned or compared
A queue specification with proper information hiding

package Queue_Pack_Private is

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

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

    Queueoverflow, Queueunderflow : exception;

private

    type Marker is mod QueueSize;
    type List is array (Marker'Range) of Element;
    type Queue_State is (Empty, Filled);
    type Queue_Type is record
        Top, Free : Marker := Marker'First;
        State     : Queue_State := Empty;
        Elements  : List;
    end record;
end Queue_Pack_Private;
A queue implementations 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;
    Enqueue (Item => 1, Queue => Queue);
    Dequeue (Item, Queue);
    Dequeue (Item, Queue); -- will produce a 'Queue underflow'

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

end Queue_Test_Private;
Ada95

Generic packages

... introducing:

- specification of generic packages
- instantiation of generic packages
A generic queue specification

generic
  type Element is private;

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

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

  Queueoverflow, Queueunderflow : exception;

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

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

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

end Queue_Pack_Generic;
A generic queue test program

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

procedure Queue_Test_Generic is

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

use Queue_Pack_Positive;

Queue : Queue_Type;
Item  : Positive;

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

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

- tagged types ☞ the Ada-way to say that this type can be extended
- derivation of tagged types
- method overwriting
- usage of parent entities
package Queue_Pack_Object_Base is

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

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

    Queueoverflow, Queueunderflow : exception;

end Queue_Pack_Object_Base;
package body Queue_Pack_Object_Base is

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

    procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
    begin
        if Queue.State = Empty then
            raise Queueunderflow;
        end if;
        Item := Queue.Elements (Queue.Top);
        Queue.Top := Queue.Top - 1;
        if Queue.Top = Queue.Free then Queue.State := Empty; end if;
    end Dequeue;
end Queue_Pack_Object_Base;
A derived open queue class specification

with Queue_Pack_Object_Base; use Queue_Pack_Object_Base;

package Queue_Pack_Object is

   type Ext_Queue_Type is new Queue_Type with record
      Reader       : Marker      := Marker'First;
      Reader_State : Queue_State := Empty;
   end record;

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

end Queue_Pack_Object;
A derived open queue class implementation

package body Queue_Pack_Object is

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

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

end Queue_Pack_Object;
An open class test program

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

procedure Queue_Test_Object is
    Queue : Ext_Queue_Type;
    Item  : Element;

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

exception
    when Queueunderflow => Put ("Queue underflow");
    when Queueoverflow  => Put ("Queue overflow");
end Queue_Test_Object;
Object oriented programming II

… introducing:

- private tagged types
- objects which are protected against their children also
An encapsulated queue base class specification

package Queue_Pack_Object_Base_Private is

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

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

  Queueoverflow, Queueunderflow : exception;

private

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

end Queue_Pack_Object_Base_Private;
package body Queue_Pack_Object_Base_Private is

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

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

end Queue_Pack_Object_Base_Private;
A derived encapsulated queue class specification

with Queue_Pack_Object_Base_Private; use Queue_Pack_Object_Base_Private;

package Queue_Pack_Object_Private is

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

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

private

   type Ext_Queue_Type is new Queue_Type with null record;

end Queue_Pack_Object_Private;
A derived encapsulated queue class implementation

package body Queue_Pack_Object_Private is

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

        Storage     : Queue_Type;
        ShuffleItem : Element;

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

    (...)
Real-Time & Embedded Systems

Concurrent & Distributed Systems

(...)

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

Restore_The_Queue:
begin
  for I in 1..QueueSize loop
    Dequeue (ShuffleItem, Storage);
    Enqueue (ShuffleItem, Queue);
  end loop;
exception
  when Queueunderflow => null; -- restore is done
end Restore_The_Queue;

end Look_Ahead;

end Queue_Pack_Object_Private;
An encapsulated class test program

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

procedure Queue_Test_Object_Private is
    Queue : Ext_Queue_Type;
    Item  : Element;

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

exception
    when Queueunderflow => Put ("Queue underflow");
    when Queueoverflow  => Put ("Queue overflow");
end Queue_Test_Object_Private;
```
Ada95

Tasks & Monitors

... introducing:

• protected types
• tasks (definition, instantiation and termination)
• task synchronisation
• entry guards
• entry calls
• accept and selected accept statements
A protected queue specification

Package Queue_Pack_Protected is

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

Protected type Protected_Queue is

  entry Enqueue (Item: in Element);
  entry Dequeue (Item: out Element);

private
  Queue : Queue_Type;

end Protected_Queue;

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

package body Queue_Pack_Protected is

protected body Protected_Queue is

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

entry Dequeue (Item: out Element) when
  Queue.State = Filled is
begin
  Item := Queue.Elements (Queue.Top);
  Queue.Top := Queue.Top - 1;
  if Queue.Top = Queue.Free then Queue.State := Empty; end if;
end Dequeue;
end Protected_Queue;
end Queue_Pack_Protected;
A multitasking protected queue test program

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

procedure Queue_Test_Protected is
    Queue : Protected_Queue;

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

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

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

(...)

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

begin
    null;
end Queue_Test_Protected;
Abstract types & dispatching

... introducing:

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

package Queue_Pack_Abstract is

    subtype Element is Character;
    type Queue_Type is abstract tagged limited private;

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

private

    type Queue_Type is abstract tagged limited null record;

end Queue_Pack_Abstract;
A concrete queue specification

with Queue_Pack_Abstract; use Queue_Pack_Abstract;
package Queue_Pack_Concrete is

  QueueSize : constant Integer := 10;
type Real_Queue is new Queue_Type with private;

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

  Queue_overflow, Queue_underflow : 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;
package body Queue_Pack_Concrete is

procedure Enqueue (Item: in Element; Queue: in out Real_Queue) is
  begin
    if Queue.State = Filled and Queue.Top = Queue.Free then
      raise Queueoverflow;
    end if;
    Queue.Elements (Queue.Free) := Item;
    Queue.Free  := Queue.Free - 1;
    Queue.State := Filled;
  end Enqueue;

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

end Queue_Pack_Concrete;
A multitasking dispatching test program

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

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

© 2005 Uwe R. Zimmer, The Australian National University
task body Queue_Holder is
  Local_Queue : Queue_Class;
  Item        : Element;
begin
  Local_Queue := new Real_Queue;  -- could be a different implementation!
  Queue_User.Send_Queue (Local_Queue);
  accept Queue_Filled do
    Dequeue (Item, Local_Queue.all);  -- Item will be 'r'
  end Queue_Filled;
end Queue_Holder;

task body Queue_User is
  Local_Queue : Queue_Class;
  Item        : Element;
begin
  Local_Queue := new Real_Queue;  -- could be a different implementation!
  accept Send_Queue (Remote_Queue: in Queue_Class) do
    Enqueue ('r', Remote_Queue.all);  -- potentially a rpc!
    Enqueue ('l', Local_Queue.all);
  end Send_Queue;
  Queue_Holder.Queue_Filled;
  Dequeue (Item, Local_Queue.all);  -- Item will be 'l'
end Queue_User;
begin null; end Queue_Test_Dispatching;
Ada95

Ada95 language status

- Established language standard with free and commercial compilers available for all major OSs.
- Stand-alone runtime environments for embedded systems (some are only available commercially).
- Special (yet non-standard) extensions (i.e. language reductions and proof systems) for extreme small footprint embedded systems or high integrity real-time environments available - Ravenscar profile systems.

has been used and is in use in numberless large scale projects (e.g. in the international space station, and in some spectacular crashes: e.g. Ariane 5)

Ada2005 is just around the corner
Summary

Ada refresher course

• Specification and implementation (body) parts, basic types
• Exceptions
• Information hiding in specifications (‘private’)
• Generic programming
• Class-wide programming (‘tagged types’)
• Monitors and synchronisation (‘protected’, ‘entries’, ‘selects’, ‘accepts’)
  • Abstract types and dispatching
Concurrency – The Basic Concepts

Uwe R. Zimmer & Alistair Rendell
The Australian National University
References for this chapter

[Ben-Ari90]
M. Ben-Ari
Principles of Concurrent and Distributed Programming
1990
Prentice-Hall,
Forms of concurrency

What is concurrency?

Working definitions:

- literally ‘concurrent’ means:
  
  \textit{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 [Oxfo\-ds 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 parts for concurrent programming
  - but we need to add a number of further crucial concepts
Concurrent & Distributed Systems

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
Forms of concurrency

A computer scientist’s view on concurrency

Terminology for real parallel machines architectures:

- **SISD** [single instruction, single data]
  - standard sequential processors
- **SIMD** [single instruction, multiple data]
  - vector processors
- **MISD** [multiple instruction, single data]
  - pipelines
- **MIMD** [multiple instruction, multiple data]
  - multiprocessors or computer networks
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-reproducibility ✈ attempts at 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
  - ...
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 high 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
The concurrent programming abstraction

• 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 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) which are executed simultaneously

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

– even though this is usually not correct, it is an often valid assumption in the context of concurrent programming.
The concurrent programming abstraction

- No interaction between concurrent system parts means that we can analyse them individually as pure sequential programs.

Interaction points:

- **Contention:** multiple concurrent execution units compete for one shared resource
- **Communication:** Explicit passing of information **and/or** 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** ☞ *this course*
Correctness of concurrent non-real-time systems

[logical correctness]:

• does not depend on speeds / execution times / delays
• does not depend on actual interleaving of concurrent processes [scheduler]

does depend on all possible sequences of interaction points
The concurrent programming abstraction

Correctness vs. testing in concurrent systems:

Slight changes in external triggers may (and usually will) result in complete different schedules (interleaving):

- Concurrent programs which depend in any way on external influences cannot be tested easily
- Designs which are provably correct with respect to the specification and are independent of the actual timing behaviour 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.
The concurrent programming abstraction

Standard concepts of correctness:

• Partial correctness:

\[(P(I) \land \text{terminates}(\text{Program}(I, O))) \Rightarrow Q(I, O)\]

• Total correctness:

\[P(I) \Rightarrow (\text{terminates}(\text{Program}(I, O)) \land Q(I, O))\]

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?
Extended concepts of correctness in concurrent systems:

- 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
The concurrent programming abstraction

• Safety properties:

\[ (P(I) \land \text{Processes}(I, S)) \Rightarrow \Box Q(I, S) \]
where \( \Box Q \) means that \( Q \) does always hold

Examples:

• Mutual exclusion (no resource collisions)

• Absence of deadlocks
  (and other forms of ‘silent death’ and ‘freeze’ conditions)

• Specified responsiveness or free capabilities
  (typical in real-time / embedded systems or server applications)
The concurrent programming abstraction

- Liveness properties:

\[(P(I) \land \text{Processes}(I, S)) \Rightarrow \Diamond Q(I, S)\]

where \(\Diamond Q\) means that \(Q\) does eventually hold (and will then stay true)

Examples:

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

Interesting liveness properties can be extremely hard to be proven
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

address space 1

- stack
- code
- shared memory

... address space n

- stack
- code
- shared memory

CPU
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
Multi-processor-systems

- The kernel may execute multiple processes at a time.

- Address space and resource restrictions of individual CPUs and processes/threads need to be considered.

- Caching, synchronization, and memory protection need to be adapted.
Symmetric Multi-processing (SMP)

- all CPUs share the same physical address space (and access to resources)
- processes/threads can be executed on any available CPU
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 on full-process-level
- Scheduling of threads depends on the actual thread implementations:
  - e.g. user-level control-flows, which the kernel has no knowledge about at all
  - e.g. kernel-level control-flows, which are handled as processes with some restrictions
Process Control Blocks

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

… PCBs are usually enqueued at a certain state or condition
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 to become available
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 (not 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 (not time critical processes) – waiting for main memory space (and other resources)

 FileInputStream uses an interrupt handler to wake up the process...
**Process states**

- **Creation**
  - Batch
  - Admitted
  - Ready
  - Ready, suspended
  - Blocked, suspended
  - Blocked

- **Dispatch**
  - Pre-emption or cycle done
  - Suspending (swap-out)
  - Unblocking

- **Termination**
  - Block or synchronize
  - Swap-in
  - Swap-out
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 -1 for a failure

Frequent usage:

```c
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 processes

Communication between UNIX tasks (‘pipes’)

```c
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 & Distributed Systems

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, ...)
Concurrent & Distributed Systems

Concurrent programming languages

Language candidates

- Ada95, Chill, Erlang
- Occam, CSP
- Java, C#
- Modula-2
- Lisp, Haskell, Caml, Miranda
- Smalltalk, Squeak
- Prolog
- Esterel, Signal

Without any support for concurrency: Eiffel, C, C++, Pascal, Fortran, Cobol, Basic…

C-libraries & interfaces

- POSIX
- MPI (message passing interface)
Ada95 is a **standardized** (ISO/IEC 8652:1995(E)) ‘general purpose’ language with **core** language primitives for

- strong typing, separate compilation (specification and implementation), object-orientation,

- concurrency, monitors, rpcs, timeouts, scheduling, priority ceiling locks

- strong run-time environments

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

- additional real-time features, distributed programming, system-level programming, numeric, informations systems, safety and security issues.
A protected 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 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 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 queue test task set (producer)

(...)

task body Producer is

  Item   : Character;
  Got_It : Boolean;

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

(...)

© 2005 Uwe R. Zimmer, The Australian National University
A protected 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;  -- main task loop  
end if;  
end loop;  
end Consumer;  
begin  
null;  
end Queue_Test_Protected_Generic;
Ada95

Ada95 language status

• Established language standard with free and commercial compilers available for all major OSs.

• Stand-alone runtime environments for embedded systems (some are only available commercially).

• Special (yet non-standard) extensions (i.e. language reductions and proof systems) for extreme small footprint embedded systems or high integrity real-time environments available ☞ Ravenscar profile systems.

☞ has been used and is in use in numberless large scale projects (e.g. in the international space station, and in some spectacular crashes: e.g. Ariane 5)
Languages suggesting concurrency: e.g. functional programming

Implicit concurrency in some programming schemes

\[
\text{qsort} \; [\; ] \; = \; [\; ]
\]
\[
\text{qsort} \; (x:xs) \; = \; \text{qsort} \; [y \mid y \leftarrow xs, y < x] \; ++ \; [x] \; ++ \; \text{qsort} \; [y \mid y \leftarrow xs, y \geq x]
\]

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:

\[
\text{borderline} \; = \; (n \neq 0) \; \&\& \; (g \; (n) \; > \; h \; (n))
\]

☞ 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 assume sequential execution:

\[
\text{if} \; \text{Pointer} \; \neq \; \text{nil} \; \text{and then} \; \text{Pointer}.\; \text{next} \; = \; \text{nil} \; \text{then} \; \text{...}
\]
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: Ada95
  • Implicit concurrency: functional programming – Lisp, Haskell, Caml, Miranda
Mutual Exclusion

Uwe R. Zimmer & Alistair Rendell
The Australian National University
References for this chapter

[Ben-Ari90]
M. Ben-Ari
*Principles of Concurrent and Distributed Programming*
1990
Prentice-Hall,
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;

begin
G := 2
G := G + G;
end P2;

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

Turn: Positive range 1..2 := 1;

task body P1 is
begin
  loop
    -- non_critical_section_1;
    loop exit when Turn = 1; end loop;
    -- critical_section_1;
    Turn := 2;
    end loop;
end P1;

task body P2 is
begin
  loop
    -- non_critical_section_2;
    loop exit when Turn = 2; end loop;
    -- critical_section_2;
    Turn := 1;
    end loop;
end P2;

☞ Mutual exclusion!
☞ No deadlock!
☞ No starvation!
☞ Locks up, if there is no contention!
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!
Concurrent & Distributed Systems

Mutual exclusion: third attempt

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

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;

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!
Real-Time & Embedded Systems

Concurrent & Distributed Systems

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!
☞ Deadlock possible!
Mutual exclusion: fourth 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!
Mutual exclusion: fourth 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!

Individual starvation & global livelock possible!
Mutual exclusion: Decker’s Algorithm

type Critical_Section_State is (In_CS, Out_CS);
C1, C2: Critical_Section_State := Out_CS; Turn : Positive range 1..2 := 1;

task body P1 is begin
  loop
    -- non_critical_section_1;
    C1 := In_CS;
    loop
      exit when C2 = Out_CS;
      if Turn = 2 then
        C1 := Out_CS;
        loop exit when Turn = 1;
        end loop;
        C1 := In_CS;
      end if;
    end loop;
    -- critical_section_1;
    C1 := Out_CS; Turn := 2;
  end loop;
end P1;

end P2;

☞ Mutual exclusion!
☞ No deadlock!
☞ No starvation!
☞ No livelock!
Mutual exclusion: Peterson’s Algorithm

type Critical_Section_State is (In_CS, Out_CS);
C1, C2 : Critical_Section_State := Out_CS;
Last   : Positive range 1..2    := 1;

task body P1 is
begin
  loop
    -- non_critical_section_1;
    C1   := In_CS; Last := 1;
    loop
      exit when C2 = Out_CS or else Last /= 1;
    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; Last := 2;
    loop
      exit when C1 = Out_CS or else Last /= 2;
    end loop;
    -- critical_section_2;
    C2 := Out_CS;
  end loop;
end P2;

☞ Mutual exclusion!
☞ No deadlock!
☞ No starvation!
☞ No livelock!
... and it’s simpler
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.
Mutual exclusion: Bakery Algorithm

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 $P_i$ 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

type Choosing_State is (Yes, No);
Choosing: array (1..N) of Choosing_State := (others => No);
Number : array (1..N) of Natural := (others => 0);

task type P (I: Natural) is end P;

begin
loop
-- non_critical_section_1;
Choosing (I) := Yes;
Number (I) := Max (Number) + 1;
Choosing (I) := No;
end loop;

for J in 1..N loop
  if J /= I then
    loop
      exit when Choosing (J) = No;
    end loop;
    loop
      exit when
        Number (J) = 0 or
        Number (I) < Number (J) or
        (Number (I) = Number (J) and I < J);
    end loop;
  end if;
end loop;

-- critical_section_1;
Number (I) := 0;
end loop;
end P;

☞ Intensive communication with all processes, even if just one process tries to enter!
Beyond atomic memory access

Realistic hardware support

Atomic **test-and-set** operations [Motorola 68xxx; Intel 80x86]:

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

Atomic **exchange** operations [Motorola 68xxx; Intel 80x86]:

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

Memory cell **reservations** [Motorola PowerPC]:

- \(L := C\); – by using a special instruction, which puts a ‘reservation’ on \(C\)
- … calculate a <new value> for \(C\) …
- \(C := <\text{new value}>\);
  – 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;

begin
  loop
    -- non_critical_section_i;
    loop
      [L := C; C := 1] exit when L = 0;
    end loop;
    -- critical_section_i;
    C := 0;
  end loop;
end Pi;

begin
  loop
    -- non_critical_section_j;
    loop
      [L := C; C := 1] exit when L = 0;
    end loop;
    -- critical_section_j;
    C := 0;
  end loop;
end Pj;

☞ Mutual exclusion!, No deadlock!, No global live-lock! – for N 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;
      end loop;
      -- critical_section_i;
      Temp := L; L := C; C := Temp;
    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;
      end loop;
      -- critical_section_j;
      Temp := L; L := C; C := Temp;
    end loop;
  end Pj;

☞ Mutual exclusion!, No deadlock!, No global live-lock! – for N processes
☞ Individual starvation possible!
Concurrent & Distributed Systems

**Mutual exclusion: memory cell reservation**

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

```pascal
task body Pi is
  L : Flag;
  begin
    loop
      -- non_critical_section_i;
      loop
        L := C; -- reservation on C
        C := 1; -- works if untouched
        exit when Untouched and L = 0;
      end loop;
      -- critical_section_i;
      C := 0;
    end loop;
  end loop;
end Pi;
```

```pascal
task body Pj is
  L : Flag;
  begin
    loop
      -- non_critical_section_j;
      loop
        L := C; -- reservation on C
        C := 1; -- works if untouched
        exit when Untouched and L = 0;
      end loop;
      -- critical_section_j;
      C := 0;
    end loop;
  end loop;
end Pj;
```

☞ Mutual exclusion!, No deadlock!, No global live-lock! – for $N$ processes
☞ Individual starvation possible!
Synchronization

Semaphores

Basic definition (Dijkstra 1968)

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

- a set of processes agree on a variable $S$ operating as a flag to indicate synchronization conditions … and …

- an atomic operation $P$ on $S$ — $P$ stands for ‘passeren’ (Dutch for ‘pass’):
  - $P(S)$: \( \text{if } S > 0 \text{ then } S := S - 1 \)

- an atomic operation $V$ on $S$ — $V$ stands for ‘vrygeven’ (Dutch for ‘to release’):
  - $V(S)$: \( S := S + 1 \)

The variable $S$ is then called a **semaphore**.
Synchronization

Semaphores
... as supplied by operating systems

• a set of processes $P(1) \ldots P(N)$ agree on a variable $S$ operating as a flag to indicate synchronization conditions ... and ...

• an atomic operation \texttt{Wait} on $S$: — also: `Suspend\_Until\_True', `sem\_wait'

  • Process $P(i)$: \texttt{Wait} ($S$):
    \begin{verbatim}
    if $S > 0$
    then $S := S - 1$
    else “suspend $P(i)$ on $S”$
    \end{verbatim}

• an atomic operation \texttt{Signal} on $S$: — also: `Set\_True', `sem\_post'

  • Process $P(i)$: \texttt{Signal} ($S$):
    \begin{verbatim}
    if $\exists j$ : “$P(j)$ is suspended on $S”$
    then “release $P(j)$”
    else $S := S + 1$
    \end{verbatim}

  a release order is \textit{not} specified!
Types of semaphores:

- **General semaphores** *(counting semaphores)*: non-negative number; (range limited by the system) P and V increment and decrement the semaphore by one.

- **Binary semaphores**: restricted to \([0, 1]\); Multiple V calls have the same effect than 1 call.
  - binary semaphores are sufficient to create all other semaphore forms.
  - atomic ‘test-and-set’ operations support binary semaphores at hardware level.

- **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”.
Mutual exclusion: Semaphores

\[ S : \text{Semaphore} := 1; \]

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

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

☞ Mutual exclusion!, No deadlock!, No global live-lock! – for \( N \) processes
☞ Individual starvation possible!
Mutual exclusion: Semaphores

S1, S2: Semaphore := 1;

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

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

☞ Mutual exclusion!, No global live-lock!
☞ Individual starvation possible!
☞ Possible deadlock!
Summary

Concurrency – The Basic Concepts

• 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
Synchronization

Uwe R. Zimmer & Alistair Rendell
The Australian National University
References for this chapter

[Ben-Ari90]
M. Ben-Ari
Principles of Concurrent and Distributed Programming
1990
Prentice-Hall,

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

[Ada95RM] (link to on-line version)
Ada Working Group
ISO/IEC JTC1/SC 22/WG 9
Ada 95 Reference Manual
– Language and Standard Libraries
ISO/IEC 8652:1995(E) with COR.1:2000,
June 2001

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

all references and links are available on the course page
Synchronization

Synchronization methods

- Shared memory based synchronization
  - Semaphores
  - Conditional critical regions
  - Monitors
  - Mutexes & conditional variables
  - Synchronized methods
  - Protected objects

- Message based synchronization
  - Asynchronous messages
  - Synchronous messages
  - Remote invocation, remote procedure call
  - Synchronization in distributed systems

☞ ‘C’, POSIX — Dijkstra
☞ Edison (experimental)
☞ Modula-1, Mesa — Dijkstra, Hoare, …
☞ POSIX
☞ Java
☞ Ada95

☞ e.g. POSIX, …
☞ e.g. Ada95, CHILL, Occam2
☞ e.g. Ada95, …
☞ e.g. CORBA, …
Synchronization in concurrent systems

All data is declared ...

... either local (and protected by language-, os-, or hardware-mechanisms)

... or it is ‘out in the open’ and all access need to be synchronized!
Synchronization

Synchronization in concurrent systems

Synchronization: the run-time overhead?

Is the potential overhead justified for simple data-structures:

\[
\begin{align*}
\text{int } & \ i; \\
\text{......} & \\
\text{i++;} & \quad \text{if } \ i > n \ \{i=0;\} \\
\end{align*}
\]

{in one thread}    {in another thread}

• Are those operations atomic?
• Do we really need to introduce full featured synchronization methods here?
Synchronization in concurrent systems

```c
int i;
......
i++; | if i > n { i = 0; }
```

- Depending on the hardware and the compiler, it might be atomic, it might be not:
  - Handling a 64-bit integer on a 8- or 16-bit controller will not be atomic
    ... but perhaps it is an 8-bit integer.
  - Any manipulations on the main memory will usually not be atomic
    ... but perhaps it is a register.
  - Broken down to a load-operate-store cycle, the operations will usually not be atomic
    ... but perhaps the processor supplies atomic operations for the actual case.

- Assuming that all ‘perhapes’ apply: how to expand this code?
Synchronization

Synchronization in concurrent systems

```c
int i;

......

i++; | if i >= n { i = 0; }
```

Unfortunately: the chances that such programming errors turn out are usually small and some implicit by chance synchronization in the rest of the system might prevent them at all.

- Many effects stemming from asynchronous memory accesses are interpreted as (hardware) ‘glitches’, since they are usually rare but then 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 µcontrollers:

Measures for synchronization are required!
Synchronization

Synchronization by flags

Word-access atomicity:

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

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

\[
\text{Task 1: } x := 0; \quad | \quad \text{Task 2: } x := 5;
\]

will result in either \(x = 0\) xor \(x = 5\) — and no other value is ever observable.
**Synchronization**

*Condition synchronization by flags*

```plaintext
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: \([A \mid X] \Rightarrow [B \mid Y]\)
Synchronization by flags

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

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

  Memory flag method is ok for simple condition synchronization, but …
  ✘ … is not 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
Synchronization by semaphores
(Dijkstra 1968)

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

- A set of processes agree on a variable \( S \) operating as a flag to indicate synchronization conditions … and …

- An atomic operation \( P \) on \( S \) — \( P \) stands for ‘passeren’ (Dutch for ‘pass’):
  - \( P: \) \[ \text{if } S > 0 \text{ then } S := S - 1 \]  
  - Also: ‘Wait’, ‘Suspend_Until_True’

- An atomic operation \( V \) on \( S \) — \( V \) stands for ‘vrygeven’ (Dutch for ‘to release’):
  - \( V: \) \[ S := S + 1 \]  
  - Also: ‘Signal’, ‘Set_True’

☞ The variable \( S \) is then called a semaphore.

OS-level: \( P \) is usually also suspending the current task until \( S \geq 0 \).
CPU-level: \( P \) indicates whether it was successful, but the operation is not blocking.
Synchronization

Condition synchronization by semaphores

\[
\text{var sync : semaphore := 0;}
\]

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

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

Sequence of operations: [A | X] ⇒ [B | Y]
Synchronization

Mutual exclusion by semaphores

```plaintext
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 | X] ⇔ [B ⇔ Y xor Y ⇔ B] ⇔ [C | Z]
```
Synchronization

Semaphores in Ada95

package Ada.Synchronous_Task_Control is
    type Suspension_Object is limited private;
    procedure Set_True  (S : in out Suspension_Object);
    procedure Set_False (S : in out Suspension_Object);
    function Current_State (S : Suspension_Object) return Boolean;
    procedure Suspend_Until_True (S : in out Suspension_Object);

private
    ... -- not specified by the language
end Ada.Synchronous_Task_Control;

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

Semaphores in POSIX

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

Semaphores in POSIX

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

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

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

delivers the number of waiting processes as a negative integer,
if there are processes waiting on this semaphore
void allocate (priority_t P) {
    sem_wait (&mutex);
    if (busy) {
        sem_post (&mutex);
        sem_wait (&cond[P]);
    }
    busy = 1;
    sem_post (&mutex);
}

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

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

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

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

☞ could raise a Program_Error in Ada95.
☞ produces a potential deadlock when implemented with general semaphores.
☞ Deadlocks can be generated by all kinds of synchronization methods.
Criticism of semaphores

• Semaphores are not bound to any resource or method or region
  ☞ Adding or deleting a single semaphore operation some place might stall the whole system

• Semaphores are scattered all over the code
  ☞ hard to read, error-prone

☞ Semaphores are considered inadequate for non-trivial systems.

(all concurrent languages and environments offer efficient higher-level synchronization methods).
Basic idea:

- Critical regions are a set of code sections in different processes, which are guaranteed to be executed in mutual exclusion:
  - Shared data structures are grouped in named regions and are tagged as being private resources.
  - Processes are prohibited from entering a critical region, when another process is active in any associated critical region.

- Condition synchronisation is provided by guards:
  - When a process wishes to enter a critical region it evaluates the guard (under mutual exclusion). If the guard evaluates false, the process is suspended / delayed.
  - As with semaphores, no access order can be assumed.
Conditional critical regions

buffer : buffer_t;
resource critical_buffer_region : buffer;

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

process consumer;
  loop
    region critical_buffer_region
      when buffer.size > 0 do
        -- take from buffer etc.
      end region
  end loop;
end consumer
Criticism of conditional critical regions

• All guards need to be re-evaluated, when any conditional critical region is left:
  ☞ all involved processes are activated to test their guards
  ☞ there is no order in the re-evaluation phase ☞ potential livelocks

• As with semaphores the conditional critical regions are scattered all over the code.
  ☞ on a larger scale: same problems as with semaphores.

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

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.
How to realize conditional synchronization?

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

(Hoare)

Hoare-monitors:

- Condition variables are implemented by semaphores (\texttt{Wait} and \texttt{Signal}).
- Queues for tasks suspended on condition variables are realized.
- A suspended task releases its lock on the monitor, enabling another task to enter.

☞ More efficient evaluation of the guards:
  the task leaving the monitor can evaluate all guards and the right tasks can be activated.
☞ Blocked tasks may be ordered and livelocks prevented.
Monitors with condition synchronization

monitor buffer;
  export append, take;
var BUF : array [ ... ] of integer;
top, base : 0..size-1;
NumberInBuffer : integer;
spaceavailable, itemavailable : condition;

procedure append (I : integer);
  begin
    if NumberInBuffer = size then
      wait (spaceavailable);
    end if;
    BUF[top] := I; NumberInBuffer := NumberInBuffer+1;
top := (top+1) mod size;
signal (itemavailable)
  end append;  …
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;

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

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

Monitors in Modula-1

- **wait (s, r):**
  delays the caller until condition variable `s` is true (`r` is the rank (or ‘priority’) of the caller).

- **send (s):**
  If a process is waiting for the condition variable `s`,
  then the process at the top of the queue of the highest filled rank is activated
  (and the caller suspended).

- **awaited (s):**
  check for waiting processes on `s`. 
Monitors in Modula-1

INTERFACE MODULE resource_control;

DEFINE allocate, deallocate;
VAR busy : BOOLEAN; free : SIGNAL;

PROCEDURE allocate;
BEGIN
  IF busy THEN WAIT (free) END;
  busy := TRUE;
END;

PROCEDURE deallocate;
BEGIN
  busy := FALSE;
  SEND (free); -- or: IF AWAITED (free) THEN SEND (free);
END;

BEGIN
  busy := false;
END.
Synchronization

Monitors in ‘C’ / POSIX
(types and creation)

Synchronization between POSIX-threads:

```c
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);
...
```
Synchronization

Monitors in ‘C’ / POSIX
(types and creation)

Synchronization between POSIX-threads:

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

int pthread_mutex_init (      pthread_mutex_t     *mutex,
                             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
- … … …
Synchronization

Monitors in ‘C’ / POSIX
(types and creation)

Synchronization between POSIX-threads:

```c
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, if locked

Undefined, if threads are waiting
Monitors in ‘C’ / POSIX

(operators)

...
Monitors in ‘C’ / POSIX

(operators)

... int pthread_mutex_lock ( pthread_mutex_t *mutex);
int pthread_mutex_trylock ( pthread_mutex_t *mutex);
int pthread_mutex_timedlock ( pthread_mutex_t *mutex,
                              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);

unblocking ‘at least one’ thread
unblocking all threads
Monitors in ‘C’ / POSIX

(operators)

...
Synchronization

Monitors in ‘C’ / POSIX

(operators)

\[
\begin{align*}
\text{int } &\text{pthread_mutex_lock} & \left( \text{pthread_mutex_t } * \text{mutex} \right); \\
\text{int } &\text{pthread_mutex_trylock} & \left( \text{pthread_mutex_t } * \text{mutex} \right); \\
\text{int } &\text{pthread_mutex_timedlock} & \left( \text{pthread_mutex_t } * \text{mutex}, \text{const struct timespec } * \text{abstime} \right); \\
\text{int } &\text{pthread_mutex_unlock} & \left( \text{pthread_mutex_t } * \text{mutex} \right); \\
\text{int } &\text{pthread_cond_wait} & \left( \text{pthread_cond_t } * \text{cond}, \text{pthread_mutex_t } * \text{mutex} \right); \\
\text{int } &\text{pthread_cond_timedwait} & \left( \text{pthread_cond_t } * \text{cond}, \text{pthread_mutex_t } * \text{mutex}, \text{const struct timespec } * \text{abstime} \right); \\
\text{int } &\text{pthread_cond_signal} & \left( \text{pthread_cond_t } * \text{cond} \right); \\
\text{int } &\text{pthread_cond_broadcast} & \left( \text{pthread_cond_t } * \text{cond} \right); \\
\end{align*}
\]
Monitors in ‘C’ / POSIX
(example, definitions)

#define BUFF_SIZE 10
typedef struct {
    pthread_mutex_t mutex;
    pthread_cond_t buffer_not_full;
    pthread_cond_t buffer_not_empty;
    int count, first, last;
    int buf[BUFF_SIZE];
} buffer;
Monitors in ‘C’ / POSIX
(example, operations)

```c
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;
}
```

```c
int take (int *item, buffer *B) {
    PTHREAD_Mutex_Lock (&B->mutex);
    while (B->count == 0) {
        PTHREAD_Cond_Wait (    
            &B->buffer_not_empty,  
            &B->mutex);        
    }
    PTHREAD_Mutex.Unlock (&B->mutex);
    PTHREAD_Cond.Signal (    
        &B->buffer_not_full);   
    return 0;
}
```
Monitors in Java

Java provides two mechanisms to construct monitors:

- **Synchronized methods and code blocks**
  all methods and code blocks which are using the `synchronized` tag
  are mutually exclusive with respect to the addressed class.

- **Notification methods:** `wait`, `notify`, and `notifyAll`
  can be used only in synchronized regions and are waking any or all threads,
  which are waiting in the same synchronized object.
Synchronization

Monitors in Java

Considerations:

1. Synchronized methods and code blocks:

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

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

Monitors in Java

Considerations:

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

- `wait` suspends the thread and releases the local lock only
  - nested `wait`-calls will keep all enclosing locks.
- `notify` and `notifyAll` 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.
  - notified threads need to wait for the lock to be released and to re-evaluate its entry condition
Monitors in Java

*(multiple-readers-one-writer-example)*

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

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

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

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

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

Monitors in Java

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

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

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

public synchronized void StopWrite() {
    writing = false;
    notifyAll();
} …
public synchronized void StartRead() throws InterruptedException {
    while (writing || waitingWriters > 0) {
        wait();
    }
    readers++;
}

public synchronized void StopRead() {
    readers--;
    if (readers == 0) notifyAll();
}
Synchronization

Monitors in Java

Standard monitor solution:

• declare the monitored data-structures private to the monitor object (non-static).

• introduce a class `ConditionVariable`:

```java
public class ConditionVariable {
    public boolean wantToSleep = false;
}
```

• introduce synchronization-scopes in monitor-methods:
  ✗ synchronize on the adequate conditional variables **first** and
  ✗ synchronize on the monitor-object **second**.

• make sure that **all** methods in the monitor are implementing the correct synchronizations.

• make sure that **no other method** in the whole system is synchronizing on this monitor-object.
Monitors in Java

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

```java
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 ();

    ...
```java
... public void StartWrite () throws InterruptedException {
    synchronized (OkToWrite) {
        synchronized (this) {
            if (writing | readers > 0) {
                waitingWriters++;
                OkToWrite.wantToSleep = true;
            } else {
                writing = true;
                OkToWrite.wantToSleep = false;
            }
        }
        if (OkToWrite.wantToSleep) OkToWrite.wait ();
    }
} ...
```
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;
                }
            }
        }
    }
}
... 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 ();
    }
} ...
... public void StopRead ()
{
    synchronized (OkToWrite)
    {
        synchronized (this)
        {
            readers--;
            if (readers == 0 & waitingWriters > 0) {
                waitingWriters--;
                OkToWrite.notify ()
            }
        }
    }
}
Object-orientation and synchronization

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

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

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

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

☞ Inheritance anomaly (Matsuoka & Yonezawa ‘93)

Methods to design and analyse expandible synchronized systems exist, but are fairly complex and are not provided in any current object-oriented language.
flexible and universal,
but relies on conventions rather than compilers

POSIX offers conditional variables

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

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

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

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

Suggestions to solve this situation:

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

Criticism of monitors

• Mutual exclusion is solved elegantly and safely.
• Conditional synchronization is on the level of semaphores still
  ☞ all criticism on semaphores apply

☞ mixture of low-level and high-level synchronization constructs.
Synchronization

Synchronization by protected objects

Combine

- the **encapsulation** feature of monitors

with

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

to

Protected objects

- *all* controlled data and operations are encapsulated
- *all* operations are mutual exclusive
- entry guards are *attached* to operations
- the protected interface allows for operations on data
- no protected data is accessible (other than by defined operations)
- tasks are queued (according to their priorities)
Synchronization by protected objects in Ada95
(simultaneous read-access)

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

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

Buffer_Size : constant Integer := 10;

type Index is mod Buffer_Size;

subtype Count is Natural range 0 .. Buffer_Size;

type Buffer_T is array (Index) of Data_Item;

protected type Bounded_Buffer is

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

private

First : Index := Index'First;
Last : Index := Index'Last;
Num : Count := 0;
Buffer : Buffer_T;

end Bounded_Buffer;
Synchronization by protected objects in Ada95

(barriers)

protected body Bounded_Buffer is

  entry Get (Item : out Data_Item) when Num > 0 is
    begin
      Item  := Buffer (First);         First := First + 1;         Num   := Num - 1;
    end Get;

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

end Bounded_Buffer;
Synchronization

Synchronization by protected objects in Ada95

Protected entries are used like task entries:

```ada
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;
```

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

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

```ada
select
    delay 10.0;
   then abort
    Buffer.Put (Some_Data);
    -- meanwhile try something else
end select;
```
Barrier evaluations and task activations:

• on **calling a protected entry**, the associated barrier is evaluated (only those parts of the barrier which might have changed since the last evaluation).

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

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

Synchronization by protected objects in Ada95

(operations on entry queues)

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

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

protected body Blocker is
    entry Proceed when Proceed’count = 5
    or Release is
    begin
        Release := Proceed’count > 0;
    end Proceed;
end Blocker;
Synchronization by protected objects in Ada95
(operations on entry queues)

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

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

protected body Broadcast is
  entry Receive (M: out Message)
    when Arrived is
      begin
        M := New_Message
        Arrived := Receive’count > 0;
      end Proceed;
  procedure Send (M: in  Message) is
    begin
      New_Message := M;
      Arrived := Receive’count > 0;
    end Send;
end Blocker;
```
Further refinements on task control by:

- **Entry families**: a protected entry declaration can contain a discrete subtype selector, which can be evaluated by the barrier (other parameters cannot be evaluated by barriers) and implements an array of protected entries.

- **Requeue facility**: protected operations can use ‘requeue’ to redirect tasks to other internal, external, or private entries. The current protected operation is finished and the lock on the object is released.

  ‘**Internal progress first**'-rule: internally requeued tasks are placed at the head of the waiting queue!

- **Private entries**: protected entries which are not accessible from outside the protected object, but can be employed as destinations for requeue operations.
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;
Synchronization

Synchronization by protected objects in Ada95
(requeue & private entries)

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

☞ e.g. by employing two entries:

```ada
package Single_Release is
  entry Wait;
  procedure Trigger;
  private
    Front_Door, Main_Door : Boolean := False;
    entry Queue;
end Single_Release;
```
Synchronization by protected objects in Ada95
(requeue & private entries)

package body Single_Release is

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

entry Queue
  when Main_Door is
  begin
    if Queue'count = 0 then
      Main_Door := False;
    end if;
  end Queue;

procedure Trigger is
  begin
    Front_Door := True;
  end Trigger;
end Single_Release;
Synchronization

Synchronization by protected objects in Ada95
(restrictions applying to protected operations)

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

Thus the following operations are prohibited:

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

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

**Shared memory based synchronization**

**General**

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

Shared memory based synchronization

**POSIX**

- all low level constructs available.
- no connection with the actual data-structures.
- error-prone.
- non-determinism introduced by ‘release some’ semantics of conditional variables (cond_signal).
Summary

Shared memory based synchronization

Java

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

Shared memory based synchronization

Modula-1, CHILL

- full monitor implementation (Dijkstra-Hoare monitor concept).

  ... no more, no less, ...

☞ all features of and criticism about monitors apply.
**Summary**

**Shared memory based synchronization**

**Ada95**

- complete synchronization support
- low-level semaphores for very special cases.
- predictable timing (**scheduler**).

☞ most memory oriented synchronization conditions are realized by the compiler or the run-time environment directly rather than the programmer.

(Ada95 is currently without any mainstream competitor in this field)
Synchronization

Message-based synchronization

• Synchronization model
  • Asynchronous
  • Synchronous
  • Remote invocation

• Addressing (name space)
  • direct communication
  • mail-box communication

• Message structure
  • arbitrary
  • restricted to ‘basic’ types
  • restricted to un-typed communications
Synchronization

Message-based synchronization

Asynchronous messages

If there is a listener:
☞ send the message directly
Message-based synchronization

Asynchronous messages

If the receiver becomes available at a later stage:

- the message needs to be buffered
Synchronization

Message-based synchronization

Synchronous messages

Delay the receiver:
- until the message becomes available

Diagram:
- P1
  - sync. send
  - time
- P2
  - sync. receive
  - time
**Message-based synchronization**

Synchronous messages

Delay the receiver:
- until the message becomes available

Simulated by asynchronous messages:
- two asynchronous messages required
**Synchronization**

**Message-based synchronization**

**Synchronous messages**

Delay the sender until:

- a receiver is available
- a receiver got the message
Message-based synchronization

Synchronous messages

Delay the sender until:

- a receiver is available
- a receiver got the message

Simulated by asynchronous messages:
If the receiver becomes available at a later stage:
☞ message needs to be buffered
Synchronization

Message-based synchronization

Remote invocation

Delay the receiver, until:

- an invocation is available
- a receiver executed an addressed routine
Synchronization

Message-based synchronization

Remote invocation

Delay the receiver, until:

- an invocation is available
- a receiver executed an addressed routine

Simulated by asynchronous messages:

☞ four messages are required
Message-based synchronization

Remote invocation

Delay the sender, until:

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

Delay the sender, until:

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

Simulated by asynchronous messages:

- four messages are required
- message buffering required
Message-based synchronization

Asynchronous remote invocation

Delay the sender, until:

- a receiver becomes available
- a receiver got the message
Synchronization

Message-based synchronization

Asynchronous remote invocation

Delay the sender, until:

- a receiver becomes available
- a receiver got the message

Simulated by asynchronous messages:

☞ two messages are required
Synchronization

Synchronous vs. asynchronous communications

Purpose ‘synchronization’: synchronous messages / remote invocations
Purpose ‘in-time delivery’: asynchronous messages / asynchronous remote invocations

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

Can both communication modes emulate each other?

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

Direct vs. indirect:

send \langle message \rangle to \langle process-name \rangle
wait for \langle message \rangle from \langle process-name \rangle
send \langle message \rangle to \langle mailbox \rangle
wait for \langle message \rangle from \langle mailbox \rangle

Asymmetrical addressing:

send \langle message \rangle to ...
wait for \langle message \rangle

☞ Client-server paradigm
### Synchronization

**Addressing (name space)**

Communication medium:

<table>
<thead>
<tr>
<th>Connections</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>one-to-one</td>
<td>buffer, queue, synchronization</td>
</tr>
<tr>
<td>one-to-many</td>
<td>multicast</td>
</tr>
<tr>
<td>one-to-all</td>
<td>broadcast</td>
</tr>
<tr>
<td>many-to-one</td>
<td>local server, synchronization</td>
</tr>
<tr>
<td>all-to-one</td>
<td>general server, synchronization</td>
</tr>
<tr>
<td>many-to-many</td>
<td>general network- or bus-system</td>
</tr>
</tbody>
</table>
Synchronization

Message structure

- Machine dependent representations need to be taken care of in a distributed environment.
- Communication system is often outside the typed language environment.

Most communication systems are handling streams (packets) of a basic element type only.

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

... manually (POSIX, ‘C/C++’, Java)
... semi-automatic (CORBA)
... automatic and are typed-persistent (Ada95, CHILL, Occam2)
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;
Reading and writing values of any type to a stream:

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

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

```ada
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;
```
### Synchronization

**Message-based synchronization**

Practical message-passing systems:

<table>
<thead>
<tr>
<th>Environment</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSIX</td>
<td>&quot;message queues&quot;: ordered indirect [asymmetrical</td>
</tr>
<tr>
<td>CHILL</td>
<td>&quot;buffers&quot;, &quot;signals&quot;: ordered indirect [asymmetrical</td>
</tr>
<tr>
<td>Occam2</td>
<td>&quot;channels&quot;: indirect symmetrical synchronous fully-typed one-to-one message passing</td>
</tr>
<tr>
<td>Ada95</td>
<td>&quot;(extended) rendezvous&quot;: ordered direct asymmetrical [synchronous</td>
</tr>
<tr>
<td>Java</td>
<td>no communication via messages available</td>
</tr>
</tbody>
</table>
## Message-based synchronization

Practical message-passing systems:

<table>
<thead>
<tr>
<th>System</th>
<th>ordered</th>
<th>symmetrical</th>
<th>asymmetrical</th>
<th>synchronous</th>
<th>asynchronous</th>
<th>direct</th>
<th>indirect</th>
<th>one-to-one</th>
<th>many-to-one</th>
<th>many-to-many</th>
<th>method</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSIX:</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>message passing</td>
</tr>
<tr>
<td>CHILL:</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>message passing</td>
</tr>
<tr>
<td>Occam2:</td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>message passing</td>
</tr>
<tr>
<td>Ada95:</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>message passing</td>
</tr>
<tr>
<td>Java:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>remote invocation</td>
</tr>
</tbody>
</table>

Java: no communication via messages available
Message-based synchronization

Practical message-passing systems for strict synchronisation purposes:

<table>
<thead>
<tr>
<th>OS</th>
<th>ordered</th>
<th>symmetrical</th>
<th>asymmetrical</th>
<th>synchronous</th>
<th>asynchronous</th>
<th>direct</th>
<th>indirect</th>
<th>contents</th>
<th>one-to-one</th>
<th>many-to-one</th>
<th>many-to-many</th>
<th>method</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSIX:</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td>bytes</td>
<td></td>
<td></td>
<td></td>
<td>message passing</td>
</tr>
<tr>
<td>CHILL:</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td>typed</td>
<td></td>
<td></td>
<td></td>
<td>message passing</td>
</tr>
<tr>
<td>Occam2:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td>fully typed</td>
<td></td>
<td></td>
<td></td>
<td>message passing</td>
</tr>
<tr>
<td>Ada95:</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td>fully typed</td>
<td></td>
<td></td>
<td>*</td>
<td>remote invocation</td>
</tr>
<tr>
<td>Java:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>no communication via messages available</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

tasks are synchronized at these points
CHILL is the ‘CCITT High Level Language’, where CCITT is the Comité Consultatif International Télégraphique et Téléphonique. The CHILL language development was started in 1973 and standardized in 1979.

- strong support for concurrency, synchronization, and communication (monitors, buffered message passing, synchronous channels)

```chill

dcl SensorBuffer buffer (32) int;
...
send SensorBuffer (reading);
receive case
  -asynchronous
    (SensorBuffer in data) : ...
  esac;

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

Synchronization

Message-based synchronization in CHILL
Message-based synchronization in Ada95

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

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

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

☞ parameters incl. bounds and discriminants are ‘tunnelled’ through byte-stream-formats.

Synchronization:

- both tasks are synchronized at the beginning of the remote invocation (☞ ‘rendezvous’)
- the calling task if blocked until the remote routine is completed (☞ ‘extended rendezvous’)

© 2005 Uwe R. Zimmer, The Australian National University
Message-based synchronization in Ada95

Remote invocation (Rendezvous)

Delay the sender, until:

- a receiver becomes available
- a receiver got the message
- a receiver started an addressed routine
Message-based synchronization in Ada95

Remote invocation (Extended rendezvous)

Delay the sender, until:

- a receiver becomes available
- a receiver got the message
- a receiver executed an addressed routine
- a receiver passed the results
Message-based synchronization in Ada95

(Rendezvous)
Synchronization

Message-based synchronization in Ada95

(Rendezvous)

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

...<entry_name> [(index)] <parameters>

... synchronized
Synchronization

Message-based synchronization in Ada95
(Extended rendezvous)

```ada
...<entry_name>[,(index)]<parameters>...  ...
...-- waiting for synchronization...  ...
...--...
...--...
...--
synchronized
...
...-- blocked
...
...--

accept <entry_name>[,(index)]
<parameter_profile> do
...
...--
...-- remote invocation
...
...--

return results

end <entry_name>;
...
```
Message-based synchronization in Ada95

(Extended rendezvous)

```ada
accept <entry_name> [(index)] <parameter_profile> do
    -- waiting for synchronization
    ... -- remote invocation
end <entry_name>;
```

```
return results
```
Message-based synchronization in Ada95

Some things to consider for task-entries:

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

[Ben-Ari90]
M. Ben-Ari
Principles of Concurrent and Distributed Programming
1990
Prentice-Hall,

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

[Ada95RM] (link to on-line version)
Ada Working Group
ISO/IEC JTC1/SC 22/WG 9
Ada 95 Reference Manual
– Language and Standard Libraries
ISO/IEC 8652:1995(E) with COR.1:2000,
June 2001

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

all references and links are available on the course page
**Non-Determinism**

**Selective waiting**

Dijkstra’s guarded commands:

```plaintext
if x <= y -> m := x
fi
```

```plaintext
if x >= y -> m := y
fi
```

☞ the programmer needs to design the alternatives as ‘parallel’ options:
   all cases need to be covered and overlapping conditions need to lead to the same result

Extremely different philosophy: ‘C’-switch:

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

☞ the sequence of alternatives has a crucial role.
Non-Determinism

Selective waiting in Occam2

\[
\text{ALT} \\
\quad \text{Guard 1} \\
\qquad \text{Process 1} \\
\quad \text{Guard 2} \\
\qquad \text{Process 2} \\
\ldots
\]

- 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

Selective waiting in Occam2

\[
\begin{align*}
\text{ALT} & \\
\text{NumberInBuffer} < \text{Size} & \land \text{Append} ? \text{Buffer} \ [\text{Top}] \\
\text{SEQ} & \\
\text{NumberInBuffer} :&= \text{NumberInBuffer} + 1 \\
\text{Top} :&= (\text{Top} + 1) \text{ REM Size} \\
\text{NumberInBuffer} > 0 & \land \text{Request} ? \text{ANY} \\
\text{SEQ} & \\
\text{Take} ! \text{Buffer} \ [\text{Base}] & \\
\text{NumberInBuffer} :&= \text{NumberInBuffer} - 1 \\
\text{Base} :&= (\text{Base} + 1) \text{ REM Size}
\end{align*}
\]

- synchronization on input-channels only:
  - to initiate the sending of data (Take ! Buffer [Base]),
    a request need to be made first (Request ? ANY)

CSP (Hoare) also supports non-deterministic selective waiting
Selective Synchronization

Message-based selective synchronization in Ada95

Forms of selective waiting:

\[
\text{select\_statement ::= selective\_accept | conditional\_entry\_call | timed\_entry\_call | asynchronous\_select}
\]

... underlying concept: Dijkstra’s guarded commands

\text{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 Ada95

selective_accept in its full syntactical form in Ada95:

\[
\text{selective_accept ::= select} \\
\quad \text{[guard] selective_accept_alternative} \\
\quad \quad \{ \text{or [guard] selective_accept_alternative} \\
\quad \quad \quad \text{[else sequence_of_statements]} \\
\quad \quad \text{end select;}
\]

\[
\text{guard ::= when } \langle \text{condition} \rangle \Rightarrow
\]

\[
\text{selective_accept_alternative ::= accept_alternative \; | \; delay_alternative \; | \; terminate_alternative}
\]

\[
\text{accept_alternative ::= accept_statement [ sequence_of_statements ]}
\]

\[
\text{delay_alternative ::= delay_statement [ sequence_of_statements ]}
\]

\[
\text{terminate_alternative ::= terminate;}
\]
Selective Synchronization

Basic forms of selective synchronization

(select-or)

```plaintext
select
    accept ... do ...
end ...

or
    accept ... do ...
end ...

or
    accept ... do ...
end ...

or
    accept ... do ...
end ...

... end select;
```

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

Basic forms of selective synchronization

(guarded select-or)

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

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

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

...

end select;
```

- Analogue to Dijkstra’s guarded commands
- all accepts closed will raise a Program_Error
  - set of conditions need to be complete
Selective Synchronization

Basic forms of selective synchronization

(guarded select-or-else)

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

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

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

else
  <statements>

end select;
```

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

Basic forms of selective synchronization

(guarded select-or-delay)

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

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

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

... 
end select;
```

- If none of the open entries has been called before the amount of time specified in the earliest open delay alternative, this delay alternative is selected.

- There can be multiple delay alternatives if more than one delay alternative expires simultaneously, either one may be chosen.

- `delay` and `delay until` can be employed.
Selective Synchronization

Basic forms of selective synchronization

(selective select-or-terminate)

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

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

or
  [ when <condition> => ]
    terminate;
...
end select;
```

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

- all tasks which can possibly call any of the named entries are terminated.

or

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

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

Basic forms of selective synchronization

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

```
select
   | when <condition> => |
   | accept ... do ... |
ed ... or
   | when <condition> => |
   | delay ... |
   | <statements> |
... end select;

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

else
   <statements>
... end select;
```

else-delay-terminate alternatives cannot be mixed!
Selective Synchronization

Conditional & timed entry-calls

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

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

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

select Controller.Request (Medium) (Some_Item); -- process data or delay 45.0; -- try something else end select;
Selective Synchronization

Conditional & timed entry-calls

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

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

There is only one entry call and either one ‘else’ or one ‘or delay’

select  
Light_Monitor.Wait_for.  
Lux := True;  
else  
   Lux := False;  
end;
Selective Synchronization

Conditional & timed entry-calls

\[
\text{conditional\_entry\_call ::= select entry\_call\_statement [sequence\_of\_statements] else sequence\_of\_statements end select;}
\]

\[
\text{timed\_entry\_call ::= select entry\_call\_statement [sequence\_of\_statements] or delay\_alternative end select;}
\]

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

\[
\text{select Light\_Monitor.Wait\_for\_Light; Lux := True; else Lux := False; end select;}
\]

\[
\text{select Controller.Request (Medium) (Some\_Item); -- process data or delay 45.0; -- try something else end select;}
\]
Selective Synchronization

Non-determinism in selective synchronizations

- If equal alternatives are given, then the program correctness (incl. the timing specifications) must not be affected by the actual selection.

- If alternatives have different priorities, this can be expressed e.g. by means of the Ada real-time annex.

- Non-determinism in concurrent systems is or can be also introduced by:
  - non-ordered monitor or other queues
  - buffering / routing message passing systems
  - non-deterministic schedulers
  - timer quantization
  - clock drifts
  - network congestions
  - ... any other form of asynchronism
The concurrent programming abstraction

Correctness of concurrent non-real-time systems [logical correctness]:

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

does depend on all possible sequences of interaction points
remember our introduction: 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:

\[(P(I) \land \text{Processes}(I, S)) \Rightarrow \Box Q(I, S)\]

where \(\Box Q\) means that \(Q\) does always hold

• Liveness properties:

\[(P(I) \land \text{Processes}(I, S)) \Rightarrow \Diamond Q(I, S)\]

where \(\Diamond Q\) means that \(Q\) does eventually hold (and will then stay true)

and \(S\) is the current state of the concurrent system
The concurrent programming abstraction

Correctness of concurrent non-real-time systems [logical correctness]:

☞ does depend on all possible sequences of interaction points

☞ Isn’t there an actual unique sequence of interaction points, … ☞ which is determined by the system and can be calculated?

in general: NO
- due to common intrinsically non-deterministic effects
Non-Determinism

Selective waiting

Dijkstra’s guarded commands:

\[
\begin{align*}
&\text{if } x \leq y \rightarrow m := x \\
&\text{but } x \geq y \rightarrow m := y \\
&\text{fi}
\end{align*}
\]

☞ selection is non-deterministic!

☞ 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

☞ Systems based on non-deterministic alternatives extent canonically to concurrent systems
Selective Synchronization

Basic forms of selective synchronization in Ada95

(select)

```ada
select
  when <condition> =>
    accept … do …
  end …

or
  when <condition> =>
    accept … do …
  end …

or
  when <condition> =>
    accept … do …
  end …

... end select;
```

Considering all alternatives
leads to many different interleavings!

How to keep it understandable / verifiable?

☞ avoid combinatorial explosions!
☞ reunite different paths as soon as possible
☞ specify unique system-wide synchronization-(check)-points
**Summary**

**Non-Determinism**

- Selective synchronization
  - Selective accepts
  - Selective calls
  - Indeterminism in message based synchronization

- General Non-Determinism in Concurrent Systems
Scheduling

Uwe R. Zimmer & Alistair Rendell
The Australian National University
References for this chapter

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

[Stallings2001] – Chapter 3,4
William Stallings
*Operating Systems*
Prentice Hall, 2001

all references and some links are available on the course page
Purpose of scheduling

A scheduling scheme provides two features:

- **Ordering the use of resources** (e.g. CPUs, networks)
- **Predicting the worst-case behaviour** of the system when the scheduling algorithm is applied

… in case that some or all information about the expected resource requests are known

A prediction can then be used

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

<table>
<thead>
<tr>
<th>Performance criteria:</th>
<th>Predictability criteria:</th>
</tr>
</thead>
<tbody>
<tr>
<td>minimize the ...</td>
<td>minimize the diversion from given</td>
</tr>
</tbody>
</table>

#### Process / user perspective:

<table>
<thead>
<tr>
<th>Metric</th>
<th>User Perspective</th>
<th>System Perspective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waiting time</td>
<td>maximum / average / variance</td>
<td>minimal and maximal waiting times</td>
</tr>
<tr>
<td>Response time</td>
<td>maximum / average / variance</td>
<td>minimal and maximal response times</td>
</tr>
<tr>
<td>Turnaround time</td>
<td>maximum / average / variance</td>
<td>deadlines</td>
</tr>
</tbody>
</table>

#### System perspective:

<table>
<thead>
<tr>
<th>Metric</th>
<th>System Perspective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>maximum / average / variance</td>
</tr>
<tr>
<td>Utilization</td>
<td>CPU idle time</td>
</tr>
</tbody>
</table>
**Scheduling**

**Time scales of scheduling**

- Long-term
  - creation
  - batch
  - admit

- Short-term
  - ready
  - dispatch
  - CPU

- Medium-term
  - ready, suspended
  - swap-in
  - blocked, suspended
  - swap-out
  - blocked

- Terminate
  - suspend (swap-out)
  - unblock
  - block or synchronize

**Pre-emption or cycle done**
Example: Requested times

\[(16,8)\]
\[(12,3)\]
\[(4,1)\]

\[(T_i, C_i)\]
First come, first served (FCFS) – bad case: (arrival order: \( T_i, C_i \))

Waiting time: 0…11; average: 5.95 – Turnaround time: 3…12; average: 8.47
**First come, first served (FCFS) – nice case:** (arrival order: , , )

- **Waiting time:** 0…11; average: **5.47** – **Turnaround time:** 3…12; average: **8.00**

☞ The actual average waiting time for FCFS may vary here between: **5.47** and **5.95**
**Scheduling**

**Round robin (RR) – pre-emption**

Waiting time: 0…4; average: 1.21 – Turnaround time: 1…19; average: 5.63

*Waiting* and average *turnaround time* is going down, but *maximal turnaround* time is going up

... assuming that task-switching is free and always possible
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
Feedback with $2^i$ pre-emption intervals – pre-emption

Waiting time: 0…6; average: 1.79 – Turnaround time: 1…21; average 5.63

less task switches than RR,
but long processes can suffer starvation!
Scheduling

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

Waiting time: 0…10; average: 3.47 – Turnaround time: 1…14; average: 6.00

☞ on average this is doing better than FCFS
Highest response ratio first (HRRF) – $C_i$ is known

Response ratio: $(W_i + C_i) / C_i$ – Waiting time: 0…9; average: 4.11 – Turnaround time: 1…13; average 6.63

☞ on average this is doing worse than SJF, but the maximal waiting and turnaround times and variance might be reduced!
Scheduling

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

Waiting time: 0…6; average: 1.05 – Turnaround time: 1…18; average 4.42

☞ on average this is doing better than FCFS, SJF or HRRF,
but the maximal turnaround time is going up and there are many task-switches!
CPU utilization: 100% in all cases.

Pre-emptive methods perform better, assuming that the overhead is negligible.

Knowledge of $C_i$ (computation times) has a significant impact on scheduler performance.
## Non-realtime scheduling methods

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

- Task behaviours are more specified (restricted).
- Task requirements are more specific (time scopes).
- Task sets are often fully or mostly static.
- Sporadic and urgent requests (e.g. user interaction, alarms) need to be addressed.

¬ CPU-utilization and throughput (system oriented performance measures) are not important!
Specifying timing requirements

Temporal scopes

Common attributes:

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

Common attributes:

- Minimal & maximal delay after creation
- Maximal elapsed time
- Maximal execution time
- Absolute deadline
### Specifying timing requirements

#### Some common scope attributes

**Temporal Scopes** can be:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Periodic</td>
<td>e.g. controllers, samplers, monitors</td>
</tr>
<tr>
<td>Aperiodic</td>
<td>e.g. ‘periodic on average’ tasks, burst requests</td>
</tr>
<tr>
<td>Sporadic / Transient</td>
<td>e.g. mode changes, occasional services</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard</td>
<td>single failure leads to severe malfunction</td>
</tr>
<tr>
<td>Firm</td>
<td>results are meaningless after the deadline</td>
</tr>
<tr>
<td>Soft</td>
<td>only multiple or permanent failures threaten the whole system</td>
</tr>
<tr>
<td></td>
<td>results may still be useful after the deadline</td>
</tr>
</tbody>
</table>
A simple process model

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

This model can only be applied to a very specific group of systems.
(more real-world extensions to this model will be discussed in other courses).
Predictable scheduling

Introducing deadlines
Dynamic scheduling

Earliest deadline first (EDF)

1. Determine (one of) the process(es) with the closest deadline.

2. Execute this process
   2-a until it finishes
   2-b or until another process’ deadline is found closer than the current one.

☞ Pre-emptive scheme
☞ Dynamic scheme, since the dispatched process is selected at run-time, due to the current deadlines.
Dynamic scheduling: Earliest Deadline First (EDF)

Earliest deadline first

1. Schedule the earliest deadline first
2. Avoid task switches (in case of equal deadlines)
Dynamic scheduling: Earliest Deadline First (EDF)

Earliest deadline first: Response times

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

- can be close or identical to deadlines.
- small or none spare capacity, if any task misses its expected computation time.
Dynamic scheduling: Earliest Deadline First (EDF)

Earliest deadline first: Maximal utilization

\[ \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \]

Maximal possible utilization: sufficient & necessary test!

with \( C_i, T_i \) the computation and cycle times of task \( i \)

(the deadlines \( D_i \) are assumed to be identical with the cycles times \( T_i \) here)
Static scheduling

Fixed Priority Scheduling (FPS), rate monotonic

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

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

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

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

Fixed Priority Scheduling (FPS), rate monotonic

Rate monotonic ordering is **optimal**
(in the framework of fixed priority schedulers)

i.e. **if** a process set is schedulable under a FPS-scheme,
**then** it is also schedulable by applying rate monotonic priorities.
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Rate monotonic priorities

\[
\sum_{i=1}^{n} \frac{C_i}{T_i} \leq N \left( \frac{1}{2^N - 1} \right)
\]

max. utilization test: sufficient, but not necessary test!
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Rate monotonic priorities

utilization test: \( \sum_{i=1}^{n} \frac{C_i}{T_i} = 1 > 0.779 \approx N \left( \frac{1}{2^N} - 1 \right) \)  \(\Rightarrow\) not guaranteed!
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Rate monotonic priorities (reduced requests)

\[ \text{utilization: } \frac{6}{16} + \frac{3}{12} + \frac{1}{4} = 0.875 > 0.779 \approx 3 \left( \frac{1}{2^3} - 1 \right) ; \quad \sum_{i=1}^{n} \frac{C_i}{T_i} \leq N \left( \frac{1}{2^N} - 1 \right) \]

\(\Rightarrow\) not guaranteed!
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Rate monotonic priorities (further reduced requests)

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

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

Response time analysis (further reduced requests)

- calculate the worst case response times for each task individually.
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis (further reduced requests)

For the highest priority task: $R_3 = C_3$
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis (further reduced requests)

\[ R_i = C_i + l_i = \text{computation } C_i + \text{interference } l_i \]
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis (further reduced requests)

for other tasks: \( R_i = C_i + \sum_{j > i} \left[ \frac{R_i}{T_j} \right] C_j \)
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis (further reduced requests)

$R_3 = 1\checkmark; R_2 = 4\checkmark; R_1 = 10\checkmark$ and $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq N \left( \frac{1}{2^N} - 1 \right)$
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis (reduced requests)

\[
R_3 = 1; R_2 = 4; R_1 = 12 \text{ but } \sum_{i=1}^{n} \frac{C_i}{T_i} > N \left( \frac{1}{2^N} - 1 \right)
\]
Static scheduling: Fixed Priority Scheduling (FPS), rate monotonic

Response time analysis (full requests)

\[
R_3 = 1 \checkmark; \ R_2 = 4 \checkmark; \ R_1 = 19 \times \text{and} \sum_{i=1}^{n} \frac{C_i}{T_i} > N \left( \frac{1}{2^N - 1} \right) \times
\]
**Dynamic scheduling: Earliest Deadline First (EDF)**

**Response time analysis** (full requests)

- Testing all combinations in a hyper-period: LCM of \( \{ T_i \} \) — here: 48
Dynamic scheduling: Earliest Deadline First (EDF)

Response time analysis (full requests)

Testing all combinations in a hyper-period: LCM of \( \{ T_i \} \) — here: 48

\[
R_\text{−}: 16 \leq 16 \checkmark = T_\text{−}; \quad R_\text{−}: 12 \leq 12 \checkmark = T_\text{−}; \quad R_\text{−}: 4 \leq 4 \checkmark = T_\text{−}
\]
Dynamic scheduling: Earliest Deadline First (EDF)

Response time analysis (reduced requests)

relaxed task-set changes:

\[ R_\text{-} : 16 \rightarrow 12 \leq 16 \checkmark = T_\text{-} ; \quad R_\text{-} : 12 \rightarrow 8 \leq 12 \checkmark = T_\text{-} ; \quad R_\text{-} : 4 \rightarrow 1 \leq 4 \checkmark = T_\text{-} \]
Dynamic scheduling: Earliest Deadline First (EDF)

Response time analysis (further reduced requests)

Further relaxed task-set changes:

\[ R_{-}: 12 \rightarrow 10 \leq 16 \checkmark = T_{-}; \quad R_{-}: 8 \rightarrow 6 \leq 12 \checkmark = T_{-}; \quad R_{-}: 1 \rightarrow 1 \leq 4 \checkmark = T_{-} \]
Real-time scheduling

Response time analysis (comparison)

<table>
<thead>
<tr>
<th></th>
<th>Fixed Priority Scheduling</th>
<th></th>
<th>Earliest Deadline First</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>utilization test</td>
<td>response times ( {R_i} )</td>
<td>utilization test</td>
</tr>
<tr>
<td>( {(T_i, C_i)} = {(16, 8); (12, 3); (4, 1)} )</td>
<td>✗ (1.000)</td>
<td>{ ✗, 4, 1}</td>
<td>✓ (1.000)</td>
</tr>
<tr>
<td>( {(T_i, C_i)} = {(16, 6); (12, 3); (4, 1)} )</td>
<td>✗ (0.875)</td>
<td>{12, 4, 1}</td>
<td>✓ (0.875)</td>
</tr>
<tr>
<td>( {(T_i, C_i)} = {(16, 4); (12, 3); (4, 1)} )</td>
<td>✓ (0.750)</td>
<td>{10, 4, 1}</td>
<td>✓ (0.750)</td>
</tr>
</tbody>
</table>

\[
\sum_{i=1}^{n} \frac{C_i}{T_i} \leq N \left( \frac{1}{2^N} - 1 \right) \]

\[
C_i + \sum_{j>i} \left[ \frac{R_i}{T_j} \right] C_j \]

\[
\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1
\]

check full hyper-cycle
Real-time scheduling

**Fixed Priority Scheduling ↔ Earliest Deadline First**

- EDF can handle higher (full) utilization than FPS.
- FPS is easier to implement and implies less run-time overhead.

- Graceful degradation features (resource is over-booked):
  - FPS: processes with lower priorities will always miss their deadlines first.
  - EDF: any process can miss its deadline and can trigger a cascade of failed deadlines.

- Response time analysis and utilization tests:
  - FPS: $O(n)$ utilization test — response time analysis: fixed point equation
  - EDS: $O(n)$ utilization test — response time analysis: fixed point equation in hyper-cycle
### Concurrent & Distributed Systems

<table>
<thead>
<tr>
<th>Selection</th>
<th>Pre-emption</th>
<th>Waiting</th>
<th>Turnaround</th>
<th>Preferred processes</th>
<th>Starvation possible?</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCFS</td>
<td>$max(W_i)$</td>
<td>no</td>
<td>possibly long</td>
<td>possibly long</td>
<td>long</td>
</tr>
<tr>
<td>RR</td>
<td>equal share</td>
<td>yes</td>
<td>bound</td>
<td>possibly long</td>
<td>none</td>
</tr>
<tr>
<td>Feedback</td>
<td>priority queues</td>
<td>yes</td>
<td>short on average</td>
<td>very short on average, large maximum</td>
<td>short</td>
</tr>
<tr>
<td>SJF</td>
<td>$min(C_i)$</td>
<td>no</td>
<td>short on average</td>
<td>short on average</td>
<td>short</td>
</tr>
<tr>
<td>HRRF</td>
<td>$max((W_i + C_i)/C_i)$</td>
<td>no</td>
<td>short on average, lower variance</td>
<td>short on average, lower variance</td>
<td>balanced</td>
</tr>
<tr>
<td>SRTF</td>
<td>$min(C_i - E_i)$</td>
<td>yes</td>
<td>very short on average</td>
<td>very short on average, large maximum</td>
<td>short</td>
</tr>
<tr>
<td>FPS</td>
<td>$max(P_i)$</td>
<td>yes</td>
<td>priority based</td>
<td>priority based</td>
<td>higher priority</td>
</tr>
<tr>
<td>EDF</td>
<td>$min(D_i)$</td>
<td>yes</td>
<td>deadline based</td>
<td>often close to deadlines</td>
<td>most urgent</td>
</tr>
</tbody>
</table>
Summary

Scheduling

• Basic performance based scheduling
  
  • $C_i$ is not known: first-come-first-served (FCFS), round robin (RR), and feedback-scheduling
  
  • $C_i$ is known: shortest job first (SJF), highest response ration first (HRRF), shortest remaining time first (SRTF)-scheduling

• Basic predictable scheduling
  
  • Fixed Priority Scheduling (FPS) with Rate Monotonic (RMPO)
  
  • Earliest Deadline First (EDF)
Safety & Liveness
Uwe R. Zimmer & Alistair Rendell
The Australian National University
References for this chapter

[Ben-Ari90]
M. Ben-Ari
Principles of Concurrent and Distributed Programming
1990
Prentice-Hall,

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

Extended concepts of correctness in concurrent systems:

¬ Termination is often not intended or even considered a failure

• Safety properties:

\[(P(I) \land \text{Processes}(I, S)) \Rightarrow \Box Q(I, S)\]

where \(\Box Q\) means that \(Q\) does \textit{always} hold

• Liveness properties:

\[(P(I) \land \text{Processes}(I, S)) \Rightarrow \Diamond Q(I, S)\]

where \(\Diamond Q\) means that \(Q\) does \textit{eventually} hold (and will then stay true)

and \(S\) is the current state of the concurrent system
Correctness in concurrent systems

- Liveness properties:

\[(P(I) \land \text{Processes}(I, S)) \Rightarrow \diamond Q(I, S)\]

where \(\diamond Q\) means that \(Q\) does eventually hold (and will then stay true)

Examples:

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

Interesting liveness properties can be extremely hard to be proven
Models and Terminology

one central liveness property: Fairness

• Liveness properties:

\[(P(l) \land \text{Processes}(l, S)) \Rightarrow \Diamond Q(l, S)\]

where \(\Diamond Q\) means that \(Q\) does eventually hold (and will then stay true)

Fairness (as a means to avoid starvation):

• **Weak fairness**: \(\Diamond \Box R \Rightarrow \Diamond G\)
  resource will eventually be granted, if a process requests continually

• **Strong fairness**: \(\Box \Diamond R_j \Rightarrow \Diamond G\)
  resource will eventually be granted, if a process requests infinitely often

• **Linear waiting**: resource will be granted
  before any other process had the same resource granted more than once.

• **First-in, first-out**: resource will be granted
  before any other process which applied for the same resource at a later point in time.
Correctness in concurrent systems

• Safety properties:

\[(P(I) \land \text{Processes}(I, S)) \Rightarrow \square Q(I, S)\]

where \(\square Q\) means that \(Q\) does always hold

Examples:

• Mutual exclusion (no resource collisions)

• Absence of deadlocks

  (and other forms of ‘silent death’ and ‘freeze’ conditions)

• Specified responsiveness or free capabilities

  (typical in real-time / embedded systems or server applications)
Deadlocks

Synchronization may lead to

\[ \text{DEADLOCKS} \]

(avoidance / prevention of those is one central safety property)

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

Reserving resources in reverse order

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

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

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

Sequence of operations: 

\[
[A \mid X] \Rightarrow \{[B \rightarrow Y] \text{ xor } [Y \rightarrow B]\} \Rightarrow [C \mid Z]
\]

or:

\[
[A \mid X] \Rightarrow \text{deadlocked!}
\]
Deadlocks

Circular dependencies

```plaintext
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_3);
  signal (reserve_2);
  statement M;
end P3;

Sequence of operations :
[A | X | K] \rightarrow \{[B \rightarrow Y \rightarrow L] \text{xor } ...\} \rightarrow [C | Z | M]
or : [A | X | K] \Rightarrow \text{deadlocked!}
```
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

☞ system *may* be deadlocked, if *all* these conditions apply!
Deadlocks

**Deadlock strategies:**

1. Ignorance
   - Kill unresponsive processes

2. Deadlock detection & recovery
   - Find deadlocked processes and recover the system in a coordinated way

3. Deadlock avoidance
   - The resulting system state is checked before any resources are actually assigned

4. Deadlock prevention
   - The system prevents deadlocks by its structure
Deadlocks

Deadlock prevention
(remove one of the four deadlock conditions)

1. **Mutual exclusion**: Applicable to specific cases only; usually this can only be removed by replication of resources.

2. **Hold and wait**: Processes are forced to allocate all their required resources at once, often at the time of admittance to the main dispatcher – done in many static realtime-systems.

3. **No pre-emption**: If the current state of a resource can be stored and restored easily, then they can be pre-empted. Usually resources are pre-empted from processes, which are currently not ready to run.

4. **Circular wait**: A circular wait can be avoided by a global ordering of all resources, e.g. resources can only be requested in a specific order – hard to maintain in a dynamic system configuration.
Deadlocks

Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

\[ RAG = \{ V, E \} ; \text{vertices and edges} \]

\[ V = P \cup R ; \text{vertices are processes or resource types:} \]

\[ P = \{ P_1, P_2, \ldots, P_n \} ; \text{processes} \]

\[ R = \{ R_1, R_2, \ldots R_k \} ; \text{resource types} \]

\[ E = E_r \cup E_a \cup E_c ; \text{claims, requests and assignments} \]

\[ E_c = \{ P_i \rightarrow R_j, \ldots \} ; \text{claims} \]

\[ E_r = \{ P_i \rightarrow R_j, \ldots \} ; \text{requests} \]

\[ E_a = \{ R_i \rightarrow P_j, \ldots \} ; \text{assignments} \]

Note: a resource may have more than one instance
Deadlocks

Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

the two process, reverse allocation deadlock:
Deadlocks

Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

no, there is no circular dependency
Deadlocks

Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

Yes, there are circular dependencies:

\[ P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1 \]

as well as: \[ P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2 \]

\[ \text{IF some processes are deadlocked, THEN there are cycles in the resource allocation graph} \]
Deadlocks

Edge Chasing

(Chandy, Misra & Haas distributed version)

∀ blocking process:
  • send probe containing three process id’s:
    [the blocked, the sending, the receiving process]

∀ blocked process receiving a probe:
  • propagate the probe to the process holding the resource, which this process requests
    (while updating the second and third proc.-id’s.)

∀ blocking process receiving its own probe:
  ☜ possible deadlock detected!
Could the deadlock situation be avoided?

Assuming all claims of $P_3$ are known in advance,
Real-Time & Embedded Systems
Concurrent & Distributed Systems

Deadlocks

Resource Allocation Graphs
(Silberschatz, Galvin & Gagne)

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

☞ in this case: assign $R_3$ to $P_2$ (instead of $P_3$)
Concurrent & Distributed Systems

Deadlocks

Resource Allocation Graphs
(Silberschatz, Galvin & Gagne)

\[ P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1 \]
as well as: \[ P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2 \]

☞ ARE some processes deadlocked, IF there are cycles in the resource allocation graph?
Deadlocks

Resource Allocation Graphs
(Silberschatz, Galvin & Gagne)

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

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

Resource Allocation Graphs
(Silberschatz, Galvin & Gagne)

If there is more than one instance per resource type:

\[
\text{IF there are cycles in the resource allocation graph AND there is more than one instance per resource type, THEN some processes may be deadlocked!}
\]
Deadlocks

How to detect deadlocks in the general case?
(of multiple instances per resource)
Deadlocks

Banker’s algorithm

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

- **Allocated**\([i, j]\)
  - the number of resources of type \( j \) allocated by process \( i \).

- **Free**\([j]\)
  - the number of available resources of type \( j \).

- **Claimed**\([i, j]\)
  - the number of resources of type \( j \) required by process \( i \) to complete eventually.

- **Request**\([i, j]\)
  - the number of *currently* requested resources of type \( j \) by process \( i \).

Temporary variables:

- **Completed**\([i]\): boolean vector indicating processes, which may complete right now.
- **Simulated_Free**\([j]\): available resources, if some processes complete and de-allocate.
Deadlocks

Banker’s algorithm

Checking for a deadlock situation

1. $Simulated\_Free \leftarrow Free; \forall i: Completed[i] \leftarrow False$

2. $\textbf{While } \exists i: \neg Completed[i]$
   $\textbf{and } \forall j: Requested[i, j] < Simulated\_Free[j] \textbf{ do: } \{\text{request } i \text{ can be granted}\}$

   $\forall j: Simulated\_Free[j] \leftarrow Simulated\_Free[j] + Allocated[i, j]$
   $Completed[i] \leftarrow True$

3. $\textbf{If } \forall i: Completed[i] \textbf{ then the system is } \textbf{deadlock-free!}$
   (otherwise all processes $i$ with $Completed[i] = False$ are deadlocked)
Deadlocks

Banker’s algorithm

Checking the current system state

1. $\text{Simulated\_Free} \leftarrow \text{Free}; \forall i: \text{Completed}[i] \leftarrow \text{False}$

2. While $\exists i: \neg \text{Completed}[i]$ and $\forall j: \text{Claimed}[i, j] < \text{Simulated\_Free}[j]$ do: {meaning process $i$ can complete}

   $\forall j: \text{Simulated\_Free}[j] \leftarrow \text{Simulated\_Free}[j] + \text{Allocated}[i, j]$
   $\text{Completed}[i] \leftarrow \text{True}$

3. If $\forall i: \text{Completed}[i]$ then the system is safe!
   (e.g. no process is currently deadlocked and no process can be deadlocked in any future state)
Deadlocks

Banker’s algorithm

Checking the validity of a resource request

\[
\text{If } (\text{Request} < \text{Claimed}) \text{ and } (\text{Request} < \text{Free}) \text{ then}
\]

\[
\text{Free} := \text{Free} - \text{Request}; \\
\text{Claimed} := \text{Claimed} - \text{Request}; \\
\text{Allocated} := \text{Allocated} + \text{Request};
\]

☞ **Apply system state check (as above)**

\[
\text{If System.is.safe then}
\]

☞ **Actually grant request**

\[
\text{else}
\]

\[
\text{-- restore former system state (Free, Claimed, Allocated)}
\]

\[
\text{end if;}
\]

\[
\text{end if;}
\]
Deadlocks

Deadlock detection / prevention

☞ Distributed version?

• Most resources are assigned to a local group of processes.
☞ Split the system into nodes
☞ Organize them as hierarchical trees or other topologies
☞ Check for deadlocks locally
☞ find local deadlocks immediately
☞ Exchange information about blocked tasks occasionally
☞ detect global deadlocks eventually

Menasce & Muntz – Ho & Ramamoorthy
Deadlocks

Deadlock recovery

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

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

- use deadlock prevention or deadlock avoidance instead
Deadlocks

- Ignorance & recovery
  - ‘kill some seemingly persistently blocked processes from time to time’ (exasperation)

- Deadlock detection & recovery
  - multiple methods for detection, e.g. resource allocation graphs, Banker’s algorithm
  - recovery is mostly ‘ugly’

- Deadlock avoidance
  - check system safety before allocating resources, e.g. Banker’s algorithm

- Deadlock prevention
  - eliminate one of the pre-conditions for deadlocks
**Failure modes**

**Terminology**

Reliability ::= measure of success with which a system conforms to its specification or low failure rate.

Failure ::= deviation of a system from its specification

Error ::= system state which lead to failures

Fault ::= the reason for an error
Failure modes

Faults on different levels

- Inconsistent or inadequate specification  
  - frequent source for disastrous faults
- Software design errors  
  - frequent source for disastrous faults
- Component & communication system failures  
  - rare and mostly predictable
Failure modes

Faults in the logic domain

• Non-termination / completion

  ☞ systems frozen in a deadlock state, blocked for missing input, or in infinite loop

• Value overruns, other inconsistent states

  ☞ sometimes caught by the run-time environment

• Wrong results

  ☞ wrong implementation with respect to the specification
Failure modes

Faults in the time domain

• Transient faults
  ☞ many communication system failures, electric interference, etc.

• Intermittent faults
  ☞ transient errors which occur more than once (e.g. overheating effects)

• Permanent faults
  ☞ stay in the system until they are repaired by some means
**Concurrent & Distributed Systems**

**Failure modes**

**Observable failures states**

- **Time domain**
  - too early
  - too late
  - never (omission)
  - fail silent
  - fail stop
  - fail controlled

- **Value domain**
  - fail never
  - fail uncontrolled
  - Constraint error
  - Value error
Fault prevention, avoidance, removal, ...

and/or

Fault tolerance
Fault tolerance

• Full fault tolerance
  the system continues to operate in the presence of ‘foreseeable’ error conditions without any significant failures — also this might induct a reduced operation period.

• Graceful degradation (fail soft)
  the system continues to operate in the presence of ‘foreseeable’ error conditions, 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.
Atomic & idempotent operations

Atomic operations

Definitions given in different scenarios:

An operation is atomic if the processes performing it …

- ... 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 action.
- ... do not communicate with other processes while the action is being performed.
- ... cannot detect any outside state change and do not reveal their own state changes until the action is complete.

☞ ... can be considered to be indivisible and instantaneous.
Atomic & idempotent operations

Atomic operations

Important implications:

☞ An atomic operation ...

• … is either performed fully, or not at all.

• … is declared as failed, if any part of the operation fails
  (and everything is reset to the original state).
Atomic & idempotent operations

Atomic operations

Time-lines:
Idempotent operations

Definition:

An operation is idempotent if …

• … the observable effects of the operation are *identical* after executing it *once* and after executing it *multiple times*.

Observations:

• Idempotent operations are often atomic, but do not need to be.
• Atomic operations do not need to be idempotent.
Summary

Safety & Liveness

• Liveness
  • Fairness

• Safety
  • Deadlock detection
  • Deadlock avoidance
  • Deadlock prevention

• Failure modes
  • Definitions, fault sources and basic fault tolerance

• Atomic & Idempotent operations
  • Definitions & implications
Architectures

Uwe R. Zimmer & Alistair Rendell
The Australian National University
References for this chapter

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

(Some workflow languages are already introduced at this point, so we turn to another style of clean concurrent architectures here)
William of Ockham (born at Ockham in Surrey (England) in 1280 and died in Munich in 1349):

- Philosopher and Franciscan monk
- Reasoning in the frame of the school of Nominalism:
  - ... science has nothing to do directly with things, but only with concepts of them
  - ... leading to the absolute subjectivity of all concepts and universals
- Pioneer of modern Epistemology
  (will also help to develop the concept of Phenomenology 500 years later)

- ‘Occam’s razor’:
  “Pluralitas non est ponenda sine necessitate”
  or “plurality should not be posited without necessity”
  (a common place in medieval philosophy)
occam 2.1

Origins:

- EPL (Experimental Programming Language) by David May
- CSP (Communicating Sequential Processes) by Tony Hoare
- “Dijkstra-Style” programming

Goals:

- Minimalist approach (Occam’s razor) supplying all means for:
  - Concurrency & communication,
  - Distributed systems
  - Realtime / Predictable systems
occam 2.1

Implementations:

- Transputer networks as an hardware implementation of the occam architecture (inmos, now SGS-Thomson)
- spoc (Southampton Portable occam Compiler)
- KRoC (Kent Retargetable Occam Compiler)

Historical:

- 1982: First conception
- 1992: occam 3 (draft)
- 1994: latest complete version: 2.1

Current state: academic (education)
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)
Characteristics (... everything is a process and static):

☞ no dynamic process creation
☞ no unlimited recursion

Syntax structure:

• Indention is used block indication
  (instead of ‘begin-end brackets’)

Scope of names:

• strictly local, indicated by indention
• no ‘forward declarations’, ‘exports’, ‘global variables’, or ‘shared memories’
**occam 2.1**

An example

- use processes and channels to implement a simple prime sieve
occam 2.1

VAL INT n IS 50:
   -- # of primes to be generated

VAL INT limit is 1000:
   -- range to check

[n-2] CHAN of INT link:
   -- links between filters

[n-1] CHAN of INT prime:
   -- channels to Print process

CHAN OF INT display:
PLACE display AT 1:
   -- output display to device 1
PROC Starter
(CHAN OF INT out, print)
-- feed number into the chain

INT i:
SEQ
  print ! 2  -- 2 is prime
  i := 3
  WHILE i < limit
    SEQ
      out ! i
      i := i + 2:
        -- generate odd numbers

PROC Sieve
(CHAN OF INT in, out, print)
-- filter out one prime

INT p, next:
SEQ
  in ? p
  print ! p  -- p is prime
  WHILE TRUE
    SEQ
      in ? next
      IF
        (next \p) <> 0 -- remainder?
        out ! next
      TRUE
      SKIP
PROC Ender
  (CHAN OF INT in, print)
  -- consume rest of numbers

INT p:
SEQ
  in ? p
  print ! p  -- p is prime
  WHILE TRUE
    in ? p:

PROC Printer ([] CHAN OF INT value)
  -- print each prime, in order

INT p:
SEQ i = 0 FOR SIZE value
  SEQ
    value [i] ? p
    display ! p:

PAR -- main program

  Starter (link [0], prime [0])
  PAR i = 1 FOR n-2
  Sieve (link [i-1],
        link [i],
        prime [i])
  Ender (link [n-1], prime [n-1])
  Printer (prime)
### occam 2.1 versus Ada95

<table>
<thead>
<tr>
<th>Feature</th>
<th>occam 2.1</th>
<th>Ada95</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Addressing</strong></td>
<td>one-to-one</td>
<td>many-to-one</td>
</tr>
<tr>
<td>Message formats defined by</td>
<td>the channels’ profiles</td>
<td>the ‘accepting’ tasks’ parameter profiles</td>
</tr>
<tr>
<td><strong>Synchronization form</strong></td>
<td>rendezvous</td>
<td>rendezvous</td>
</tr>
<tr>
<td><strong>Data-flow</strong></td>
<td>one way</td>
<td>one way or two ways (extended rendezvous)</td>
</tr>
<tr>
<td><strong>Selection of open alternatives</strong></td>
<td>non-deterministic</td>
<td>non-deterministic</td>
</tr>
<tr>
<td><strong>Processes</strong></td>
<td>static</td>
<td>dynamic</td>
</tr>
<tr>
<td><strong>Shared memory (‘monitors’)</strong></td>
<td>-</td>
<td>yes</td>
</tr>
</tbody>
</table>

© 2005 Uwe R. Zimmer, The Australian National University
Operating System based architectures

Operating systems architectures
Concurrent & Distributed Systems

Operating System based architectures

Hardware environments / configurations:

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

What is the common ground for operating systems?

What is an operating system?
What is an operating system?

1. A virtual machine!

... offering a more comfortable, robust, reliable, flexible ... machine
What is an operating system?

2. A resource manager!

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

Operating systems deal with

• processors,
• memory
• mass storage
• communication channels
• devices
  (timers, special purpose processors, interfaces, ...)

and many tasks/processes/programs, which are applying for access to these resources
What is an operating system?

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

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

Is there a minimal set of features?

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

Is there always an explicit operating system?

☞ no,
some languages and development systems operate with stand-alone run-time-environments.
The evolution of operating systems

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

• 1901: first wireless data transmission (Morse-code from ships to shore)
• ‘56: first transmission of data through phone-lines
• ‘62: first transmission of data via satellites (Telstar)
• ‘69: ARPA-net (predecessor of the current internet)
• 80s: introduction of fast local networks (LANs): ethernet, token-ring
• 90s: mass introduction of wireless networks (LAN and WAN)

Currently: standard consumer computers come with

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

- Multiprocessing is supported by all these OSs to some extend.
- None of these OSs are suitable for embedded systems, also 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
  - asymmetrical: only one CPU carries the full operating system, the others are operated by small operating system stubs to transfer code or tasks.
Distributed operating systems

- all CPUs carry a small kernel operating system for communication services.
- all other OS-services are distributed over available CPUs
- services may migrate
- services can be multiplied in order to
  - guarantee availability (hot stand-by)
  - or to increase throughput (heavy duty servers)
Types of current operating systems

Real-time operating systems

- Fast context switches?
- Small size?
- Quick responds 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 responds to external interrupts? ☞ not ‘quick’, but predictable
- Multitasking? ☞ real time systems are often multitasking systems
- ‘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!
Real-time operating systems requesting …

☞ the logical correctness of the results as well as
☞ the correctness of the time, when the results are delivered

☞ Predictability!

(not performance!)

☞ All results are to be delivered just-in-time – not too early, not too late.

Timing constraints are specified in many different ways …
… often as a response to ‘external’ events ☞ reactive systems
Embedded operating systems

- usually real-time systems, often hard real-time systems
- very small footprint (often a few KBs)
- none or limited user-interaction

☞ 90-95% of all processors are working here!
**Typical structures of operating systems**

‘Monolithic’ or ‘the big mess’

- non-portable
- hard to maintain
- lacks reliability
- all services are in the kernel (on the same privilege level)

☞ may reach very high efficiency

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

e.g. current LINUX versions
Typical structures of operating systems

‘Monolithic & layered’

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

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

‘μkernels and virtual machines’

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

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

‘µkernels and client-server models’

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

e.g. current µkernel research projects
Typical structures of operating systems

‘µkernels and distributed systems’

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

E.g. Java engines, distributed real-time operating systems, current distributed OSs research projects
UNIX features

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

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

```c
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 -1 for a failure

Frequent usage:
```c
if (fork () == 0) {
    ... the child’s task ...
    ... often implemented as: exec ("absolute path to executable file", "args"神通);
    exit (0);                      /* terminate child process */
} else {
    ... the parent’s task ...
    pid = wait ();                  /* wait for the termination of one child process */
}
```
Synchronization in UNIX 🔄 Signals

```c
#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 ();
}
```
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 (0);
}
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 synchronization

Pipes:

- unstructured byte-stream communication, access is identical to file operations
  ➡ not sufficient to design client-server architectures or network communications
Sockets try to keep the paradigm of a universal file interface for everything and introduce:

Connectionsless 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**

**Portable Operating System Interface for Computing Environments**

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

**POSIX – some of the real-time relevant standards**

<table>
<thead>
<tr>
<th>Version</th>
<th>Description</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>1003.1</td>
<td>OS Definition</td>
<td>single process, multi process, job control, signals, user groups, file system, file attributes, file device management, file locking, device I/O, device-specific control, system database, pipes, FIFO, …</td>
</tr>
<tr>
<td>1003.1b</td>
<td>Real-time Extensions</td>
<td>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, …</td>
</tr>
<tr>
<td>1003.1c</td>
<td>Threads</td>
<td>multiple threads within a process; includes support for: thread control, thread attributes, priority scheduling, mutexes, mutex priority inheritance, mutex priority ceiling, and condition variables</td>
</tr>
<tr>
<td>1003.1d</td>
<td>Additional Real-time Extensions</td>
<td>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</td>
</tr>
<tr>
<td>1003.1j</td>
<td>Advanced Real-time Extensions</td>
<td>typed memory, nanosleep improvements, barrier synchronization, reader/writer locks, spin locks, and persistent notification for message queues</td>
</tr>
<tr>
<td>1003.21</td>
<td>Distributed Real-time</td>
<td>buffer management, send control blocks, asynchronous and synchronous operations, bounded blocking, message priorities, message labels, and implementation protocols</td>
</tr>
</tbody>
</table>
Frequently employed POSIX features include:

- **Threads**: a common interface to threading - differences to ‘classical UNIX processes’
- **Timers**: delivery is accomplished using POSIX signals
- **Priority scheduling**: fixed priority, 32 priority levels
- **Real-time signals**: signals with multiple levels of priority
- **Semaphore**: named semaphore
- **Memory queues**: message passing using named queues
- **Shared memory**: memory regions shared between multiple processes
- **Memory locking**: no virtual memory swapping of physical memory pages
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)
Summary

Architectures

• Academic
  • occam 2.1, CSP, ...

• Workfloor
  • Ada95, Java, ...

• Environments / Operating Systems
  • Operating systems architectures
  • UNIX as a concept and basic UNIX features
  • POSIX
Distributed Systems

Uwe R. Zimmer & Alistair Rendell
The Australian National University
References for this chapter

[Ben-Ari90]
M. Ben-Ari  
*Principles of Concurrent and Distributed Programming*  
Prentice Hall 1990,  

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

[Schneider90]
Fred B. Schneider  
Implementing fault-tolerant services using the state machine approach  
ACM Computing Surveys,  
Vol. 22, No. 4, 299-319; 1990

[Tanenbaum03]
Andrew S. Tanenbaum  
*Computer Networks*  

[Tanenbaum01]
Andrew S. Tanenbaum  
*Distributed Systems: Principles and Paradigms*  
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 as such ...

  ...but its concepts and terminology are widely used,
  when designing new protocols ...
Network protocols & standards

OSI Network Layers
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)
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), …
3: Network Layer

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

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

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

- **Service**: Network access to application programs
- **Functions**: application specific
- **Examples**: APIs for mail, ftp, ssh, scp, …
Network protocols & standards

OSI
- Application
- Presentation
- Session
- Transport
- Network
- Data link
- Physical

TCP/IP
- Application
- Presentation
- Session
- Transport
- IP
- Network
- Data link
- Physical

User data
## Network protocols & standards

### OSI

<table>
<thead>
<tr>
<th>Application</th>
<th>Presentation</th>
<th>Session</th>
<th>Transport</th>
<th>Network</th>
<th>Data link</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### AppleTalk over IP

- **AppleTalk Filing Protocol (AFP)**
- **AT Data Stream Protocol**
- **AT Session Protocol**
- **Zone Info Protocol**
- **Printer Access Protocol**
- **Routing Table Maintenance Prot.**
- **AT Update Based Routing Protocol**
- **Name Binding Protocol**
- **AT Transaction Protocol**
- **AT Echo Protocol**
- **Datagram Delivery Protocol (DDP)**
- **AppleTalk Address Resolution Protocol (AARP)**
- **EtherTalk Link Access Protocol**
- **LocalTalk Link Access Protocol**
- **TokenTalk Link Access Protocol**
- **FDDITalk Link Access Protocol**
- **IEEE 802.3**
- **LocalTalk**
- **Token Ring IEEE 802.5**
- **FDDI**
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
- specified by the IEEE 802.3 standard in 1983
- 10Mbps - 1 Gbps (10Gbps in preparation)
- approx. 85% of current LAN lines worldwide

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

OSI reference model classification

- **OSI reference model**
  - Application
  - Presentation
  - Session
  - Transport
  - Network
  - Data link
  - Physical

- **IEEE 802.3 reference model**
  - MAC-client
  - Media Access (MAC)
  - Physical (PHY)
  - Upper-layer protocols

- **IEEE 802-specific**
- **IEEE 802.3-specific**
- **Media-specific**
Ethernet

MAC & PHY layer

MAC Client

802.3 MAC

Physical medium-independent layer

MII

Physical medium-dependent layers

MDI

PHY

Link

Link media, signal encoding, and transmission rate

Transmission rate

MII = Medium-independent interface
MDI = Medium-dependent interface - the link connector
Network protocols & standards

Token Ring / IEEE 802.5

- 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

Unlike CSMA/CD, the token ring is deterministic (with respect to its timing behaviour)
Token Ring / IEEE 802.5

Topology (IBM)
Fiber Distributed Data Interface (FDDI)

- Designed in the 80’s as a standard for ‘backbone networks’
- American National Standards Institute (ANSI) X3T9.5 standard
- 100Mbps token passing, dual ring local area network using fiber optical cable (or copper in case of CDDI)
- Second ring is idle in normal operations

Deterministic and Failure resistant
Network protocols & standards

FDDI / ANSI X3T9.5

OSI reference model classification

- Logical link control
- Media access control
- Physical layer protocol
- Physical layer medium

FDDI standards

Station management

OSI

Presentation
Session
Transport
Network
Data link
Physical

TCP/IP

Application
Transport
Network
Data link
Physical

User data
FDDI / ANSI X3T9.5

Cable failure tolerance

Station 1

Station 2

Station 3

Station 4

MAC

B

A

Ring wrap

Failed wiring

B

A

MAC

B

A

MAC

B

A

MAC

B

A

Network protocols & standards
Station failure tolerance

Optical bypass switch "normal configuration"

Optical bypass switch "bypassed configuration"

Failed station

Ring does not wrap

Station failure tolerance

FDDI / ANSI X3T9.5

Network protocols & standards

Concurrent & Distributed Systems
finally: distribution!

What are potential benefits?

- Fits an **existing physical distribution** (e-mail system, devices in a large aeroplane, ...).
- Possible **high performance** due to potentially high degree of parallel computing.
- Possible **high reliability** due to redundancy of hardware and software.
- Possible **scalability**.
- Integration of a large number of **heterogeneous nodes/devices** tailored to specific needs.
Distributed Systems

What can be distributed?

- State
  - common methods on distributed databases, e-mail
- Function
  - distributed methods on central data
- State & Function
  - client/server clusters
- none of those
  - pure replication, redundancy
Common design criteria

- Achieve decoupling / high degree of local autonomy
- Cooperation rather than central control
- Consider reliability
- Consider scalability
- Consider performance
Common phenomena in distributed systems

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

2. Missing or imprecise time-base
   - Was there a causal relation?
   - Was there a 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 principle alternative strategies:

☞ Synchronize clocks

☞ Create a virtual time
‘Real-time’ clocks in computer systems are:

- discrete, i.e. time is not ‘dense’, there is a minimal granularity
- drift affected

\[(1 + \delta)^{-1} \geq \frac{C(t_2) - C(t_1)}{t_2 - t_1} \geq (1 + \delta)\]
Synchronize local, drift affected clocks (both ways)
Synchronize local, drift affected clocks (forward only)
Distributed critical regions with synchronized clocks

1. **Create** `OwnRequest` and **attach** current time-stamp

2. **Add** `OwnRequest` to local `RequestQueue` (ordered by time)
   - **Send** `OwnRequest` to all processes

3. **Delay** \(2L\) (\(L\) being the time it takes for a message to reach all network nodes)

4. **Add** all received `Requests` in local `RequestQueue` (ordered by time)

5. **While** \(\text{Top}(\text{RequestQueue}) \neq \text{OwnRequest}\) do

   5-a for all received release messages **delete** corresponding `Request` in local `RequestQueue`

6. **Enter** and **leave** critical region

7. **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 requesting process: $2(N - 1)$ messages
  (can be significantly improved by employing broadcast mechanisms)

Assumptions:

- $L$ is known and constant
- no messages are lost
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 \( C(a) \), \( C(b) \) the (virtual) times associated with \( a \) and \( b \)

• \( a \rightarrow b \) holds when
  
  • \( a \) happens earlier than \( b \) in the same sequential process
  • \( a \) denotes the event of sending of message \( m \), while \( b \) denotes the receiving event of \( m \)
    (in different processes)
  • there is a transitive causal relation: \( a \rightarrow e_1 \rightarrow \ldots \rightarrow e_n \rightarrow b \)

• \( a \parallel b \Rightarrow \neg(a \rightarrow b) \land \neg(b \rightarrow a) \)
Virtual (logical) time

Implications:

\[ a \rightarrow b \Rightarrow C(a) < C(b) \]

\[ C(a) < C(b) \Rightarrow (a \rightarrow b) \lor (a \parallel b) \]

\[ C(a) = C(b) \Rightarrow a \parallel b \]
Virtual (logical) time

• time is no longer global and is attached to observable causal relations

• all events in between communications are considered concurrent in different processes
Implementing a virtual (logical) time

1. \( \forall P_i : C_i = 0 \)

2. \( \forall P_i : \)
   2-a \( \forall \) local events: \( C_i = C_i + 1 \)
   2-b \( \forall \) send \( m \) operations: \( C_i = C_i + 1; \) Send \((m, C_i)\)
   2-c \( \forall \) receive \( m \) operations: Receive \((m, C_m)\); \( C_i = max(C_i, C_m) + 1 \)
Distributed critical regions with logical clocks

Concurrently:

- Request-message received:
  - Add Request in local RequestQueue (ordered by time)
  - if OwnRequest pending reply with OwnRequest else reply with Ack

- Release-message received if delete corresponding Request in local RequestQueue

- if access to critical region required:
  1. Create OwnRequest and attach current time-stamp
  2. Add OwnRequest to local RequestQueue (ordered by time)
    - Send OwnRequest to all processes
  3. Wait for Top(RequestQueue) = OwnRequest & no outstanding replies
  4. Enter and leave critical region
  5. Send Release-message to all processes
Distributed critical regions with logical clocks

Analysis

- No deadlock, no individual starvation, no livelock
- Minimal request delay: $N - 1$ request messages, $N - 1$ reply messages
- Minimal release delay: $N - 1$ release messages
- Total communications requirements per requesting process: $3(N - 1)$ messages (can be significantly improved by employing broadcast mechanisms)

Assumption:
- no messages are lost

No assumptions about:
- runtime of messages over the communication system
Distributed critical regions with a token ring structure

1. Organize all processes in a ring (physically or logically)

2. Pass a ‘token’-message along the ring

3. On receiving the token:
   
   3-a If the local process wants to enter a critical section it does so now (while storing the token)
   
   3-b The token is passed along

☞ What happens if the token is lost?

   (there are simple recovery algorithms similar to the ‘election’ scheme following)
Distributed critical regions with a central coordinator

- a global, static, central coordinator invalidates the concept of a distributed system, but enables very simple mutual exclusion algorithms, so ...

... we pronounce one processes as the central coordinator, but
... if this one fails, the rest of the processes are able to come up with a new coordinator.

☞ This is done by a distributed ‘election’ algorithm, i.e. the Bully-algorithm [Garcia-Molina 1982]
ELECTING A CENTRAL COORDINATOR (THE BULLY ALGORITHM)

Any process \( P \) which notices that the central coordinator is done, performs:

1. Sending an \textit{Election}-message to all processes with higher process numbers

2. \( P \) wait for response messages

   2-a If no one responds after a pre-defined amount of time:
      \( P \) declares itself the new coordinator and sends out a \textit{Coordinator}-message to all.

   2-b If any process responds, the election activity for \( P \) is over
      and \( P \) waits for a \textit{Coordinator}-message

All processes \( P_i \):

   If \( P_i \) receives an \textit{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

- collect all local states at a given time:
Distributed Systems

Distributed states

• collect all local states at a given time:

\[ P_0, P_1, P_2, P_3 \]
Distributed Systems

Distributed states

• collect all local states at a given time:
Distributed Systems

Distributed states

• collect all local states at a given time (snapshot):

☞ collecting all local states at an absolute, global point in time is impossible
☞ make sure that the observed distributed state (snapshot) is at least consistent
Consistent global state (snapshot):

Make sure that all events can be uniquely divided in:

1. **before** the snapshot (belonging to the past \( P \)):
   \[(e_2 \in P) \land (e_1 \rightarrow e_2) \Rightarrow e_1 \in P\]

2. **after** the snapshot (belonging to the future \( F \)):
   \[(e_1 \in F) \land (e_1 \rightarrow e_2) \Rightarrow e_2 \in F\]
Distributed Systems

Distributed states

- check for consistency: straighten out the snapshot cut
Distributed Systems

Distributed states

- check for consistency: straighten out the snapshot cut
Distributed Systems

Distributed states

- check for consistency: straighten out the snapshot cut

\[
\begin{align*}
\text{inconsistent snapshot} & \\
\text{ inconsistent snapshot} & \\
\text{inconsistent snapshot} & \\
\end{align*}
\]
Snapshot algorithm

- Observer-process $P_o$ (any process) creates a snapshot token $t_s$ and saves its local state $s_o$
- $P_o$ sends $t_s$ to all other processes.
- $\forall P_i$ which receive the $t_s$ (as a token-message, or as part of another message):
  - save local state $s_i$ and send $s_i$ to $P_o$
  - 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 $P_o$ (this message belongs to the snapshot)
Distributed Systems

Distributed states

- apply snapshot algorithm:

- $P_0$ send out snapshot token to all
Distributed Systems

Distributed states

• apply snapshot algorithm:

• $P_2$ responds with its local state
Distributed Systems

Distributed states

• apply snapshot algorithm:

• $P_2$ forwards an untagged message
Distributed Systems

Distributed states

- apply snapshot algorithm:

- \( P_1 \) responds with its local state
- \( P_3 \) responds with its local state (due to a tagged message)
Distributed Systems

Distributed states

- apply snapshot algorithm:

- $P_3$ ignores the snapshot token (token was previously received as part of a message, local state is already reported)
Distributed Systems

Distributed states

• apply snapshot algorithm:

• $P_2$ forwards an untagged message
Distributed Systems

Distributed states

• apply snapshot algorithm:

• $P_1$ ignores a tagged message (token was previously received, local state is already reported)
• apply snapshot algorithm:

the effective snapshot of the system
... which is known to the observer $P_o$ after it received all reports
Snapshot algorithm

Termination?

either

• make assumptions about the 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

or …
Consistent distributed states

Why do we need that?

- find deadlocks
- find termination / completion conditions
- any other safety of liveness property
- collect a consistent system state for further processing (distributed databases)
Distributed Systems

A distributed server
Distributed Systems

A distributed server

Client

SendToGroup (Job)

Ring of servers
Distributed Systems

A distributed server

Client

Ring of servers

Contention messages
A distributed server

Client

Ring of servers

Server

Server

Server

Server

Server

Server

JobCompleted (Results)
A distributed server

with GroupCommunication; use GroupCommunication;

task type Client is
end Client;

task body Client is
  begin
    SendToGroup (PrintServerGroup, ClientId, PrintJob);
  end Client;
A distributed server

with Ada.Task_Identification; use Ada.Task_Identification;

task type PrintServer is
  entry SendToServer ( PrintJob : in  Job_Type;
                      JobDone : out Boolean);

  entry Contention ( ServerId : in Task_Id;
                    PrintJob : in  Job_Type);

end PrintServer;
A distributed server

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 (Current_Task, PrintJob);
            Requeue InternalPrintServer.PrintJobQueue;
          else
            TurnedDownJobs := TurnedDownJobs + PrintJob;
          end if;
        end if;

      end SendToServer;
or

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

How to construct predictable client-server systems beyond a single remote procedure call / rendezvous?

Transactions:

- **Atomicity**: All or none of the sub-operations are performed. Atomicity helps achieve crash resilience. If a crash occurs, then it’s possible to roll back the system to the state before the transaction was invoked.

- **Consistency**: Transforms the system from one consistent state to another.

- **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.

☞ known as the ‘ACID’-properties
Distributed Systems

Transactions

- **Atomicity**: All or none of the sub-operations are performed. Atomicity helps achieve crash resilience. If a crash occurs, then it’s possible to roll back the system to the state before the transaction was invoked.

- **Consistency**: Transforms the system from one consistent state to another.

- **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.

☞ how to achieve *consistency* and *isolation* in a concurrent / distributed system?
Distributed Systems

Transactions

- **Atomicity**: All or none of the sub-operations are performed. Atomicity helps achieve crash resilience. If a crash occurs, then it’s possible to roll back the system to the state before the transaction was invoked.

- **Consistency**: Transforms the system from one consistent state to another.

- **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.

☞ how to achieve consistency and isolation in a concurrent / distributed system?

☞ if the transactions are not completely side-effect free, they cannot operate on the same server data-structures concurrently? …
Distributed Systems

Transactions

- **Atomicity**: All or none of the sub-operations are performed. Atomicity helps achieve crash resilience. If a crash occurs, then it’s possible to roll back the system to the state before the transaction was invoked.

- **Consistency**: Transforms the system from one consistent state to another.

- **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.

☞ how to achieve consistency and isolation in a concurrent / distributed system?

- if the transactions are not completely side-effect free, they cannot operate on the same server data-structures concurrently? …

… maybe we can implement the appearance of isolation and the full effect of consistency?
A closer look at transactions

• Transactions consist of a sequence of individual 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.

• Some of the operations out of transactions have side-effects; those are the critical operations.

• Any sequential execution of multiple transactions fulfils the ACID-properties, by definition of a single transaction.

• Some concurrent executions (interleavings) of multiple transactions might fulfil the ACID-properties.

☞ If a specific interleaving can be shown to be equivalent to a specific sequential execution of the involved transactions then this specific interleaving is called ‘serializable’.

☞ Construct an interleaving which ensures that no transaction ever encounters an inconsistent state (ensure the appearance of isolation).
Achieving serializability

- If two side-effecting operations out of two different transactions (affecting the same object) cannot be executed in any order with the same final effect then those are conflicting pairs of operations.

☞ For 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.

Order of operations needs to be determined:
☞ distributed time-stamps are required, e.g. Lamport clocks
Serialization graphs

For **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.

Above order gives also an order dependency between the transactions as a whole.

- **Serialization graph**: directed graph; vertices $i$ represent transactions $T_i$; edges $T_i \rightarrow T_j$ represent that an observer witnessed that order dependency.

A multiple transactions interleaving is serializable

$\iff$ its serialization graph is acyclic
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, but not acquired (two phase locking) or locks are released on commit (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**

- Put 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
  - 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.

☞ no isolation ☞ cascading aborts possible.

- Alternative case 1 (strict time-stamp ordering):
  ☞ the request is delayed until the currently active earlier transaction has committed

☞ simple implementation, high degree of concurrency
  – also in a distributed environment, as long as a global event order (time) can be supplied.
Transaction schedulers – Optimistic concurrency control

Premise:
If conflict is unlikely the overhead to ensure a serializable interleaving might not be justified

Idea:
• get a local copy (shadow copy) of the involved objects
• perform a subset of the required transactions locally
• check for the current state of the object again and see whether the results of the local operations can be embedded without violating consistency
• depending on the previous check:
  either delete all local results or write them back to the actual object
Transaction schedulers – Optimistic concurrency control

Three phases

1. **Read & execute:**
   generate a shadow copy of all involved objects and perform all required operations there.

2. **Validate:**
   after local commit, check all occurred interleavings for serializability

3. **Update or abort:**
   IF serializability could be ensured in step 2 then all results of involved transactions (one transaction at a time) are written to all involved objects (in dependency order of the transactions). Otherwise destroy shadow copies and possibly start over with the failed transactions.

☞ Open issue: how to gain a consistent set of shadow copies in phase one and how to update all involved objects consistently (atomically) in phase three?
Transaction schedulers – Optimistic concurrency control

Premise:
If conflict is unlikely the overhead to ensure a serializable interleaving might not be justified

Results:
- possibly many additional copies
- deadlock free
- maximum concurrency
- with more overlapping transactions this scheduler breaks down rapidly
  - starvation & live-locks
Distributed Systems

Distributed transaction schedulers

The three major designs again:

- **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.

Commit or abort operations are required in many places above

How to implement those in a distributed environment?
Two phase commit protocol

Start-up (initialization) phase
Two phase commit protocol

Start-up (initialization) phase

Client

SendToGroup (Transaction)
Distributed Systems

Two phase commit protocol

Start-up (initialization) phase

Determine coordinator
Two phase commit protocol

Start-up (initialization) phase

Determine coordinator
Two phase commit protocol

Start-up (initialization) phase
**Two phase commit protocol**

**Start-up (initialization) phase**

- **Client**
- **Coord.**
- **Server**

Setup & start operating

Shadow copy
Two phase commit protocol

Phase 1: Determine result state

Coordinator requests & assembles votes (commit or abort)
Two phase commit protocol

**Phase 1: Determine result state**

**Important:**
Commit’ means just that (even in case of a crash)

Client

Coord. Server

Server

Server

Server

Server

Server

Client
Two phase commit protocol

Phase 2: Implement results

Client

Coord. Server
Server
Server
Server
Server
Server

Shadow copy

All commit: Coordinator tells everybody to actually commit
Two phase commit protocol

Phase 2: Implement results

All commit:
Shadow copies are destroyed
Two phase commit protocol

Phase 2: Implement results

Client

Coord. Server

Server

Server

Server

Server

Server

Everybody responds with ‘done’
Two phase commit protocol

or phase 2: global roll back
Two phase commit protocol

or phase 2: global roll back

One abort: Shadow copies are destroyed (without update)
Two phase commit protocol

Phase 2: Report result of distributed transaction

Commit or abort: Coordinator reports to client
Evaluating the three major design methods in a distributed environment:

- **Locking methods:**
  Large overheads; distributed deadlock detection required.

- **Time-stamp ordering:**
  If time-stamps can be provided: Recommends itself for distributed applications, since decisions are taken locally and communication overhead is relatively small.

- **“Optimistic” methods:**
  Maximises concurrency, but also data replication; chances of aborts and roll-backs are higher.

☞ side-aspect *data replication*: large body of literature on this topic
  (see: distributed data-bases / operating systems / shared memory, cache management, …)
Redundancy (replicated servers)

Premise:
A crashing server computer should not compromise the functionality of the system (full fault tolerance)

• $k$ 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, dynamical 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).

[Schneider90]
Redundancy (replicated servers)

Message processing stages in each server:
Fault tolerance (replicated servers)

Start-up (initialization) phase
Fault tolerance (replicated servers)

Start-up (initialization) phase

Determine coordinator
Distributed Systems

Fault tolerance (replicated servers)

Start-up (initialization) phase

Coordinator determined
Fault tolerance (replicated servers)

Receive job-message at coordinator

Client

SendToCoordinator (Job)

Coordinator receives job
Distributed Systems

Fault tolerance (replicated servers)

Distribute job-message

Coordinator distributes job both ways
Fault tolerance (replicated servers)

Distribute job-message

All server received the job (but nobody knows that)
Fault tolerance (replicated servers)

Distribute job-message

First server detects two job-messages
Fault tolerance (replicated servers)

Distribute job-message

All servers detect both job messages
Fault tolerance (replicated servers)

servers decide whether this message is known to everybody else

 processo job

Coordinator distributes job both ways

2 received messages means:
“everybody else must have seen this too

process job”
Fault tolerance (replicated servers)

Coordinator processes job-message

Coordinator receives 2 messages:
process job
Fault tolerance (replicated servers)

All servers are in the same state again - Coordinator delivers response
Fault tolerance (replicated servers)

servers crash!, new servers joining, old servers leaving …

☞ somebody (either a server detecting a time-out, or an explicitly joining or leaving server)
sends a ‘FormNewGroup’ signal to all current servers
(this message passing mechanism is assumed to be part of the distributed operating system)

1. Wait for local job processing 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 ☞ $S_i := S_j$
5. Elect coordinator
6. Enter ‘Coordinator-’ or ‘Replicate-mode’
Summary

Distributes Systems

• Networks
  • OSI, topologies, 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)
Summary

Uwe R. Zimmer & Alistair Rendell
The Australian National University
Summary

Topics in this course

1. Concurrency [3]
2. Mutual exclusion [3]
5. Scheduling [2]
7. Architectures for CDS [3]
8. Distributed systems [8]
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: Ada95
  • Implicit concurrency: functional programming – 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

• Selective synchronization
  • Selective accepts
  • Selective calls
  • Indeterminism in message based synchronization

• General Non-Determinism in Concurrent Systems
Summary

Scheduling

• Basic performance based scheduling
  • $C_i$ is not known: first-come-first-served (FCFS), round robin (RR), and feedback-scheduling
  • $C_i$ is known: shortest job first (SJF), highest response ration first (HRRF), shortest remaining time first (SRTF)-scheduling

• Basic predictable scheduling
  • Fixed Priority Scheduling (FPS) with Rate Monotonic (RMPO)
  • Earliest Deadline First (EDF)
Summary

Safety & Liveness

• Liveness
  • Fairness

• Safety
  • Deadlock detection
  • Deadlock avoidance
  • Deadlock prevention

• Failure modes
  • Definitions, fault sources and basic fault tolerance

• Atomic & Idempotent operations
  • Definitions & implications
Summary

Architectures

- **Academic**
  - occam 2.1, CSP, ...

- **Workfloor**
  - Ada95, Java, ...

- **Environments / Operating Systems**
  - Operating systems architectures
  - UNIX as a concept and basic UNIX features
  - POSIX
Summary

Distributes Systems

- Networks
  - OSI, topologies, 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)