1

Introduction

Uwe R. Zimmer – International University Bremen

References for this chapter

[Silberschatz01]
Abraham Silberschatz, Peter Bear Galvin, Greg Gagne
Operating System Concepts
John Wiley & Sons, Inc., 2001

[Stallings2001]
William Stallings
Operating Systems
Prentice Hall, 2001

[Tanenbaum97]
Andrew S. Tanenbaum, Albert S. Woodhull
Operating Systems: Design and Implementation
Prentice Hall, 1997

[Tanenbaum95]
Andrew S. Tanenbaum
Distributed Operating Systems
Prentice Hall, 1995

all references and some links are available on the course page

What are operating system based on?

Hardware environments / configurations:

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

What is the common ground for operating systems?

What is an operating system?

What is an operating system?

1. A virtual machine!

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

What is an operating system?

2. A resource manager!

… dealing with all sorts of devices and coordinating access

Operating systems deal with

- processors,
- memory
- mass storage
- communication channels
- devices
  (timers, special purpose processors, interfaces, …)

☞ and many tasks/processes/programs, which are applying for access to these resources

What is an operating system?

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

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

Is there a minimal set of features?

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

Is there always an explicit operating system?

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

The evolution of operating systems

- in the beginning: single user, single program, single task, serial processing ☞ *no OS*
- 50s: System monitors / batch processing
  ☞ the monitor ordered the sequence of jobs and triggered their sequential execution
- 50s-60s: Advanced system monitors / batch processing:
  ☞ the monitor is handling interrupts and timers
  ☞ first support for memory protection
  ☞ first implementations of privileged instructions (accessible by the monitor only).
- early 60s: Multiprogramming systems:
  ☞ employ the long device I/O delays for switches to other, runable programs
- early 60s: Multiprogramming, time-sharing systems:
  ☞ assign time-slices to each program and switch regularly
- early 70s: Multitasking systems – multiple developments resulting in UNIX (besides others)
- early 80s: single user, single tasking systems, with emphasis on user interface (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 and workstations:

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

☞ last 20 years: evolving and expanding into current general purpose OSs:

  - Solaris (based on SVR4, BSD, and SunOS)
  - LINUX (open source UNIX re-implementation for x86 processors and others)
  - current Windows (proprietary, partly based on Windows NT, which is 'related' to VMS)
  - MacOS X (Mach kernel with BSD Unix and an proprietary user-interface)

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

Types of current operating systems

Parallel operating systems

- support for a large number of processors, either:
    - symmetrical:
      each CPU has a full copy of the operating system

  or

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

Types of current operating systems

Distributed operating systems

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

Types of current operating systems

Real-time operating systems

- ~~Fast context switches?~~ ☞ 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!

Types of current operating systems

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

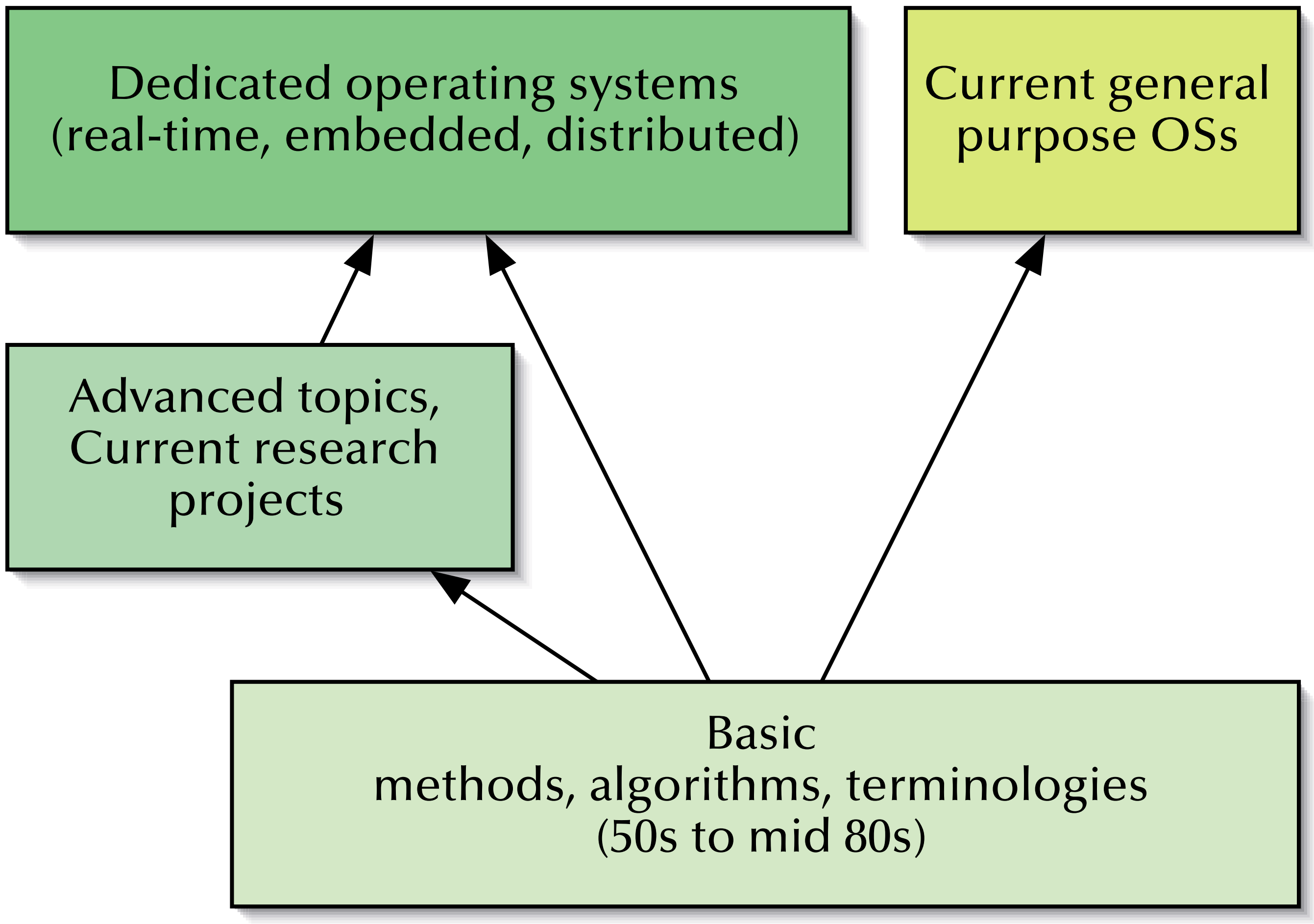
Types of current operating systems

Embedded operating systems

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

☞ 90-95% of all processors are working here!

Roots of current commercial operating systems

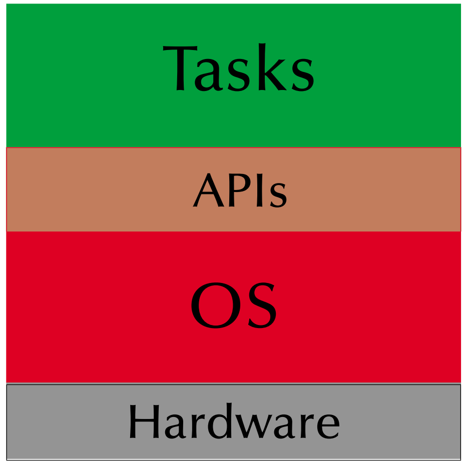


Typical structures of operating systems

'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



Monolithic

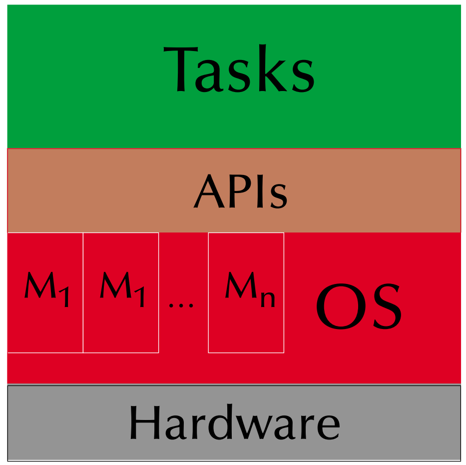
e.g. most early UNIX implementations (70s),
MS-DOS (80s), Windows (basically all versions besides NT and NT-based editions),
MacOS (until version 9),

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



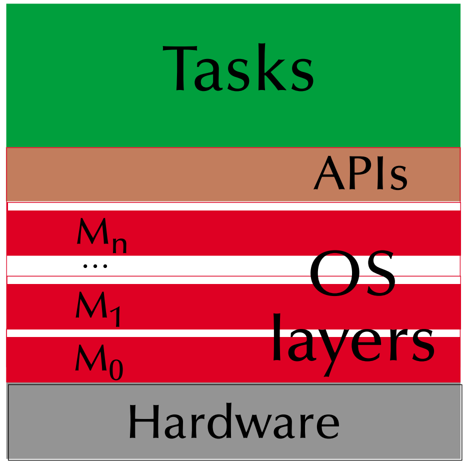
Modular

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

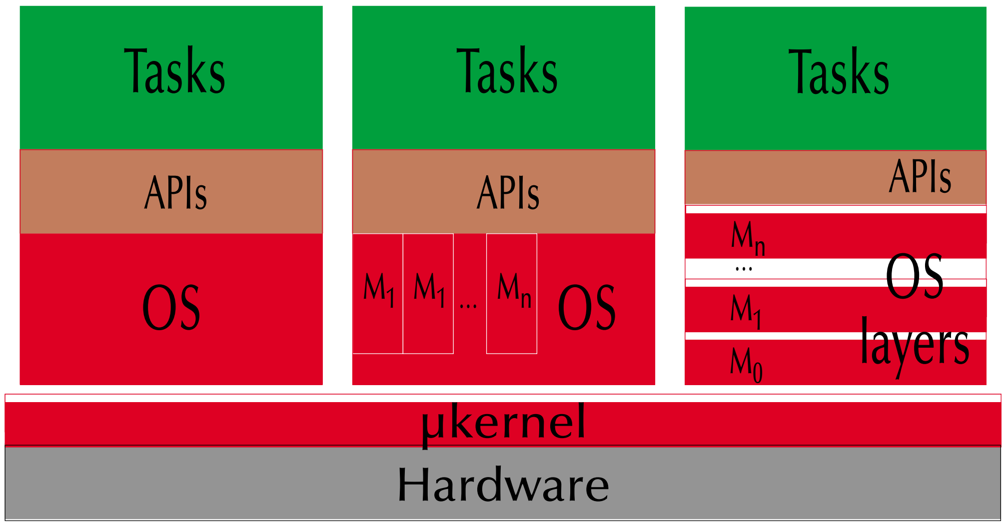


Layered

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

‘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



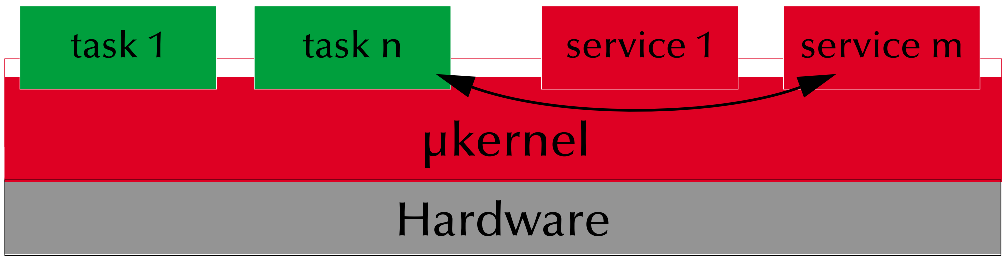
μkernel, virtual machine

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

'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

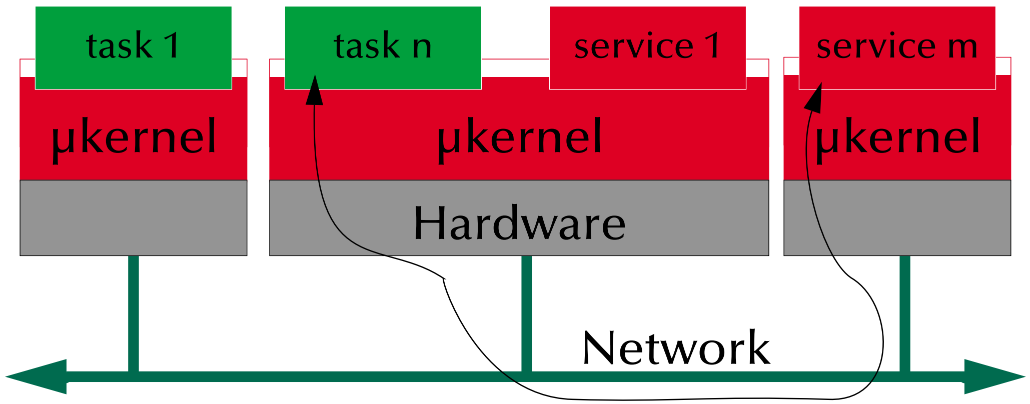
e.g. current μkernel research projects



μkernel, client server structure

‘μkernels and distributed systems’

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



μkernel, distributed systems

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

Basic programming styles

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

Programming styles alternatives

Imperative ↔ Functional ↔ Declarative ↔ Data-flow ↔ Finite state machines

Static ↔ Dynamic

Modular ↔ Concurrent ↔ Distributed

Synchronous ↔ Continuous time

Control oriented ↔ Data oriented

What makes a language suitable for operating systems?

- **Precise expressions on machine level** ☞ address physical memory + I/O
- **Concurrency** ☞ support for tasking/threading
- **Distribution** ☞ support for message passing or rpc
- **Reliability** ☞ detect errors at compile-time or in the run-time environment
- **Large systems** ☞ scalable, modular, or object-oriented + separate compilation
- **Predictability**

  ☞ no operations which will lead to unforeseeable timing behaviours (e.g. garbage collection)

Languages considered in this course

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

… others in places

Ada95

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

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

… and **standardized** language-**annexes** for

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

A crash course

… refreshing:

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

Basics

… introducing:

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

A simple queue specification

```
package Queue_Pack_Simple is

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

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

end Queue_Pack_Simple;
```

A simple queue implementation

```
package body Queue_Pack_Simple is

   procedure Enqueue (Item: in Element; Queue: in out Queue_Type) is
   begin
      Queue.Elements (Queue.Free) := Item;
      Queue.Free := Queue.Free - 1;
   end Enqueue;

   procedure Dequeue (Item: out Element; Queue: in out Queue_Type) is
   begin
      Item       := Queue.Elements (Queue.Top);
      Queue.Top := Queue.Top - 1;
    end Dequeue;

end Queue_Pack_Simple;
```

A simple queue test program

```
with Queue_Pack_Simple; use Queue_Pack_Simple;

procedure Queue_Test_Simple is

   Queue : Queue_Type;
   Item  : Element;

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

Exceptions

… introducing:

- exception handling
- enumeration types
- functional type attributes

A queue specification with proper exceptions

```
package Queue_Pack_Exceptions is

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

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

   Queueoverflow, Queueunderflow : exception;

end Queue_Pack_Exceptions;
```

A queue implementations with proper exceptions

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

```
package body Queue_Pack_Private is

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

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

end Queue_Pack_Private;
```

identical

A queue test program with proper information hiding

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

procedure Queue_Test_Private is

   Queue, Queue_Copy : Queue_Type;
   Item              : Element;

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

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

Generic packages

… introducing:

- specification of generic packages
- instantiation of generic packages

A generic queue specification

```
generic
   type Element is private;

package Queue_Pack_Generic is

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

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

   Queueoverflow, Queueunderflow : exception;

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

A generic queue *implementation*

```
package body Queue_Pack_Generic is

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

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

end Queue_Pack_Generic;
```

A generic queue test program

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

procedure Queue_Test_Generic is

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

   Queue : Queue_Type;
   Item  : Positive;

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

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

Object oriented programming I

… introducing:

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

An open queue base class *specification*

```
package Queue_Pack_Object_Base is

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

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

   Queueoverflow, Queueunderflow : exception;

end Queue_Pack_Object_Base;
```

An open queue base class implementation

```
package body Queue_Pack_Object_Base is

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

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

end Queue_Pack_Object_Base;
```

identical

A derived open queue class *specification*

```
with Queue_Pack_Object_Base; use Queue_Pack_Object_Base;

package Queue_Pack_Object is

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

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

end Queue_Pack_Object;
```

A derived open queue class implementation

```
package body Queue_Pack_Object is

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

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

end Queue_Pack_Object;
```

An open class test program

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

procedure Queue_Test_Object is

   Queue : Ext_Queue_Type;
   Item  : Element;

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

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

Object oriented programming II

… introducing:

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

An encapsulated queue base class *specification*

```
package Queue_Pack_Object_Base_Private is

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

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

   Queueoverflow, Queueunderflow : exception;

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

end Queue_Pack_Object_Base_Private;
```

An encapsulated queue base class implementation

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

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

end Queue_Pack_Object_Base_Private;
```

A derived encapsulated queue class specification

```
with Queue_Pack_Object_Base_Private; use Queue_Pack_Object_Base_Private;

package Queue_Pack_Object_Private is

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

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

private
   type Ext_Queue_Type is new Queue_Type with null record;

end Queue_Pack_Object_Private;
```

A derived encapsulated queue class implementation

```
package body Queue_Pack_Object_Private is

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

      Storage     : Queue_Type;
      ShuffleItem : Element;

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

(…)

```
  Read_The_Rest:
      begin
         for I in 1..QueueSize - Depth loop
            Deequeue (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;
```

bad

An encapsulated class test program

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

procedure Queue_Test_Object_Private is

   Queue : Ext_Queue_Type;
   Item  : Element;

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

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

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

```
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);

   Queueoverflow, Queueunderflow : exception;

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

A concrete queue implementation

```
package body Queue_Pack_Concrete is

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

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

end Queue_Pack_Concrete;
```

A multitasking dispatching test program

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

procedure Queue_Test_Dispatching is

   type Queue_Class is access all Queue_Type'class;

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

   task Queue_User is   -- could be on an individual partition
      entry Send_Queue (Remote_Queue: in Queue_Class);
   end Queue_User;
(…)
```

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

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

begin null; end Queue_Test_Dispatching;
```

Ada95 language status

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

☞ has been used and is in use in numberless large scale projects

(e.g. in the international space station, and in some spectacular crashes: e.g. Ariane 5)

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!)

POSIX – some of the real-time relevant standards

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

POSIX – 1003.1b

Frequently employed POSIX features include:

- **Timers**: delivery is accomplished using POSIX signals
- **Priority scheduling**: fixed priority, 32 priority levels
- **Real-time signals**: signals with multiple levels of priority
- **Semaphore**: named semaphore
- **Memory queues**: message passing using named queues
- **Shared memory**: memory regions shared between multiple processes
- **Memory locking**: no virtual memory swapping of physical memory pages

POSIX – support in some OSs

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

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)

POSIX – example: setting a timer

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

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

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

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

POSIX – example: setting a timer (cont.)

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

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

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

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

POSIX – example: setting a timer (cont.)

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

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

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

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

remember the Pearl timers?

AFTER 30 MIN ALL 5 MIN DURING 1 HRS ACTIVATE Help;

Languages used in this course

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

Summary

Introduction to operating systems

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