16.070

Ada and Real-Time

Prof. Lars Asplund

lars.asplund@mdh.se

Mälardalen University, Computer Science

History

- Software Engineering – first conference -69
- Strawman -> Steelman
- Ada (ANSI standard 1983); “Ada-83”
- ISO 1987
- Revision 1995, “Ada-95”, ISO (ANSI, IEEE)
  - First standardized Object Oriented Language
  - Full Real-Time
  - Safety Critical

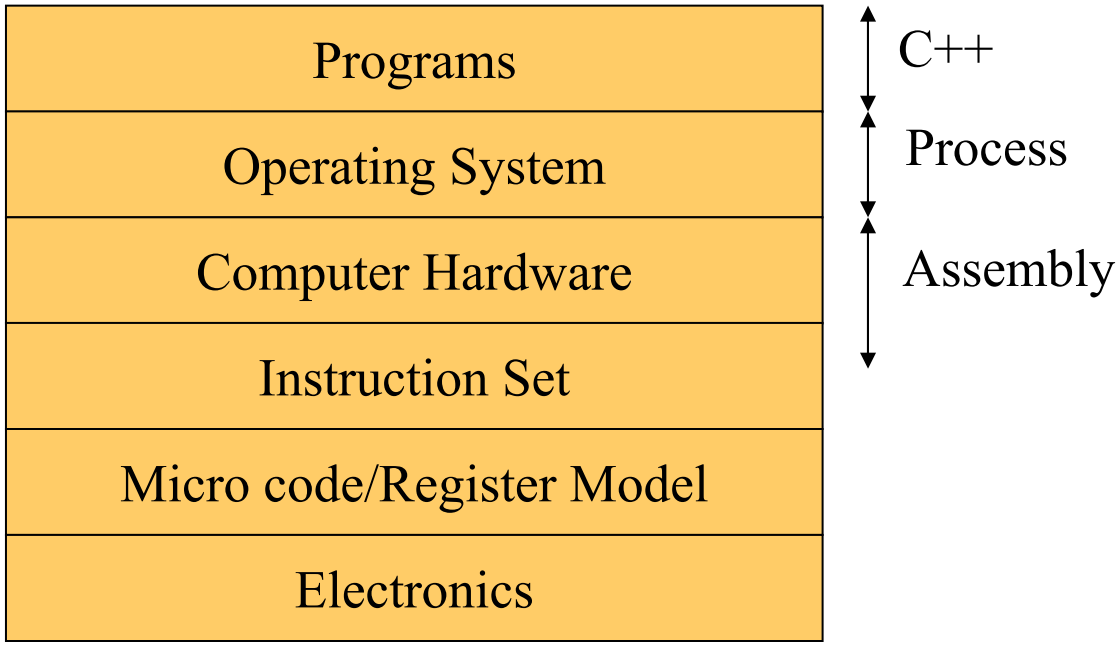
Concurrency

- Different activities that can run simultaneously
- Not necessary in parallel

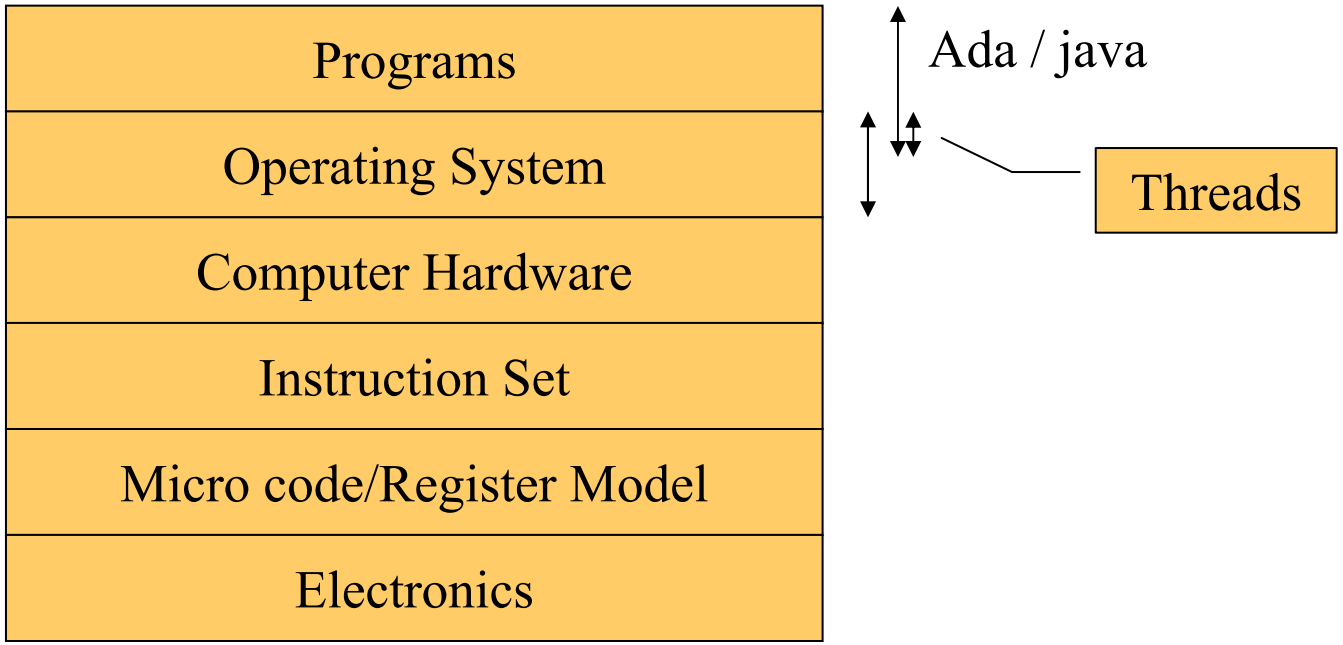
Example

- Flight Control
  - One task to control the engine
  - One task per rudder to implement a control loop
  - One task for main control

Computer model



Computer model

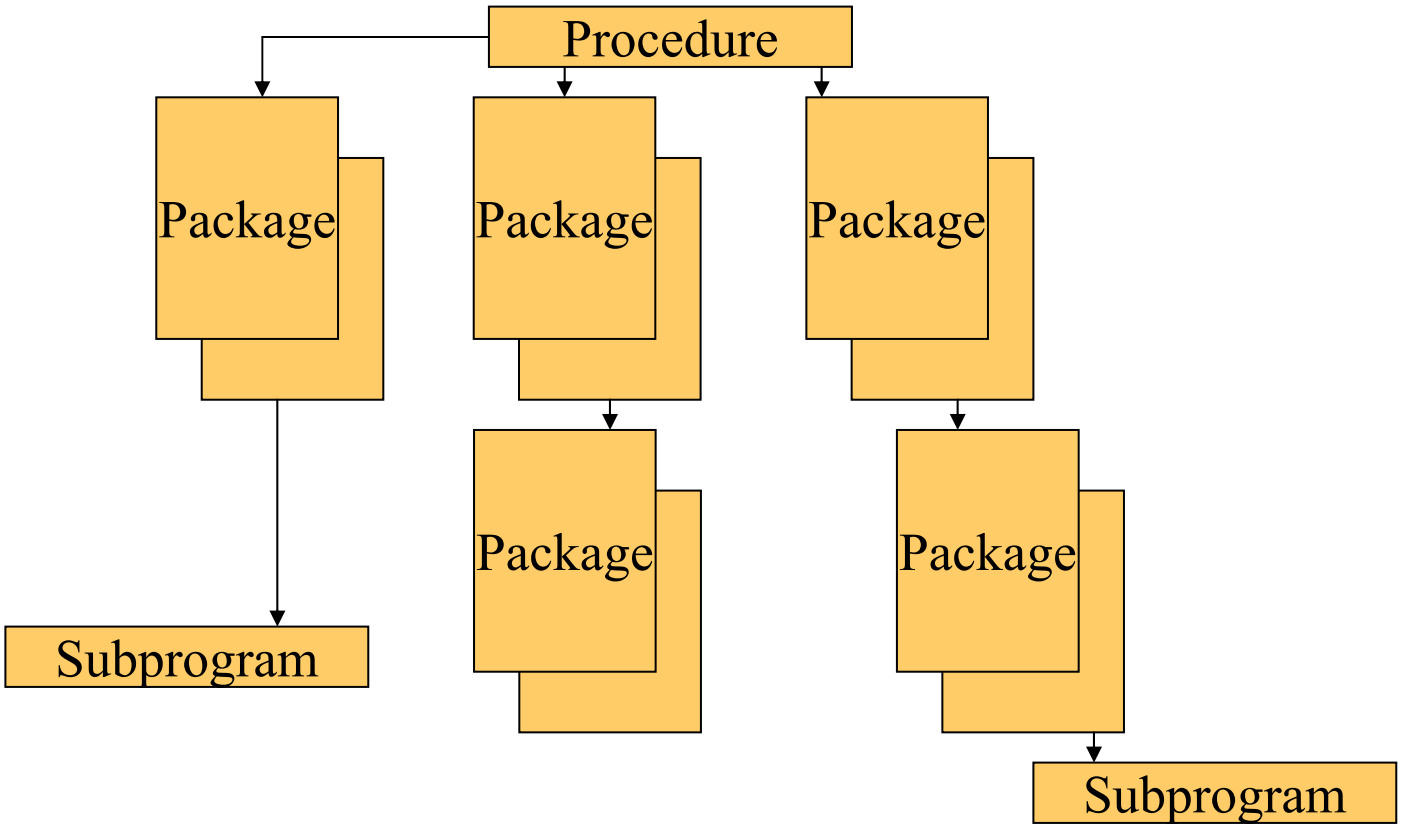


Thread vs Process

	Thread	Process
Communication	Semaphores, memory	Pipes, signals
OS-resources	Share	Own
Share data	Yes	No

Building an Ada System (SE)

Compilation units



Subprogram

function *function_name* (params…) **return** *type* **is**
  declarative part
begin
  statements
  return …;
end *function_name;*

procedure *procedure_name* (params…) **is**
  declarative part
begin
  statements
end *procedure_name;*

Packages

package *Package_Name* **is**
  declarative part
[private
  private_part]
end *Package_Name;*

package body *Package_Name* **is**
  declarative part
  subprogram implementations
[begin
  statement_part]
end *Package_Name;*

Task specification

```
task task_name is
   entry Entry_Name_1 (Params);
   entry Entgry_Name_2 (Params);
   pragma Priority (4);
end task_Name;
```

Task type and object specification

```
task type task_type_name is
   entry Entry_Name_1 (Params);
   entry Entgry_Name_2 (Params);
   pragma Priority (4);
end task_type_Name;

Task_Name : Task_Type_Name;
Another_Task : Task_Type_Name;
```

Task body

```
task body task_name is
   declarative_part
begin
   loop
      Standard_Code
      delay Duration_Expr; delay until Time_Expr;
      task_T.Entry_Name (…);
      select
         …
      end select;
   end loop;
[exception]
end task_Name;
```

Typical instructions for Real-Time

delay *Duration_Expr; -- delta time*

delay *Time_Expr; -- absolute time*

entry calls

accept – statements

select - statements

abort *Task_Name;*

requeue *Entry_Name;*

task and **protected object** handling

Synchronization and Communication

- There are two models
- Rendez Vous
  - Was introduced in Ada-83
  - A French expression for meeting your secret lover in the evening, or docking space ships
- Protected Object
  - Was introduced in Ada-95
  - Like a monitor (critical section)

Rendez Vous

- Two tasks
- The callee declares an **entry**, which can have formal arguments
- The caller calls the **entry**

The Callee

```
task Buffer is
  entry Put (C : Integer);
  entry Get (C : out Integer);
  pragma Priority (4);
end Buffer;
```

```
task body Buffer is
  Data : Integer;
begin
  loop
    accept Put (C : Integer) do
      Data := C;
    end Put;
    accept Get (C : out
     Integer) do
      C := Data;
    end Get;
  end loop;
end Buffer;
```

One Caller

```
task Producer;
```

```
task body Producer is
begin
  for I in 1..1_000_000 loop
    Buffer.Put (I);
  end loop;
end Producer;
```

Another Caller

```
task Consumer;
```

```
with Ada.Text_IO;
use Ada.Text_IO;
package/procedure ….
task body Consumer is
  My_Value : Integer;
begin
  loop
    Buffer.Get (My_Value);
    Put_Line
     (Integer'Image(My_Value));
  end loop;
end Consumer;
```

Some Action…

```
task body Producer is
begin
  for I in 1..1_000_000 loop
    Buffer.Put (I);
  end loop;
end Producer;
```

```
task body Consumer is
  My_Value : Integer;
begin
  loop
    Buffer.Get (My_Value);
    Put_Line
      (Integer'Image(My_Value));
  end loop;
end Consumer;
```

```
task body Buffer is
  Data : Integer;
begin
  loop
    accept Put (C : Integer) do
      Data := C;
    end Put;
    accept Get (C : out Integer) do
      C := Data;
    end Get;
  end loop;
end Buffer;
```

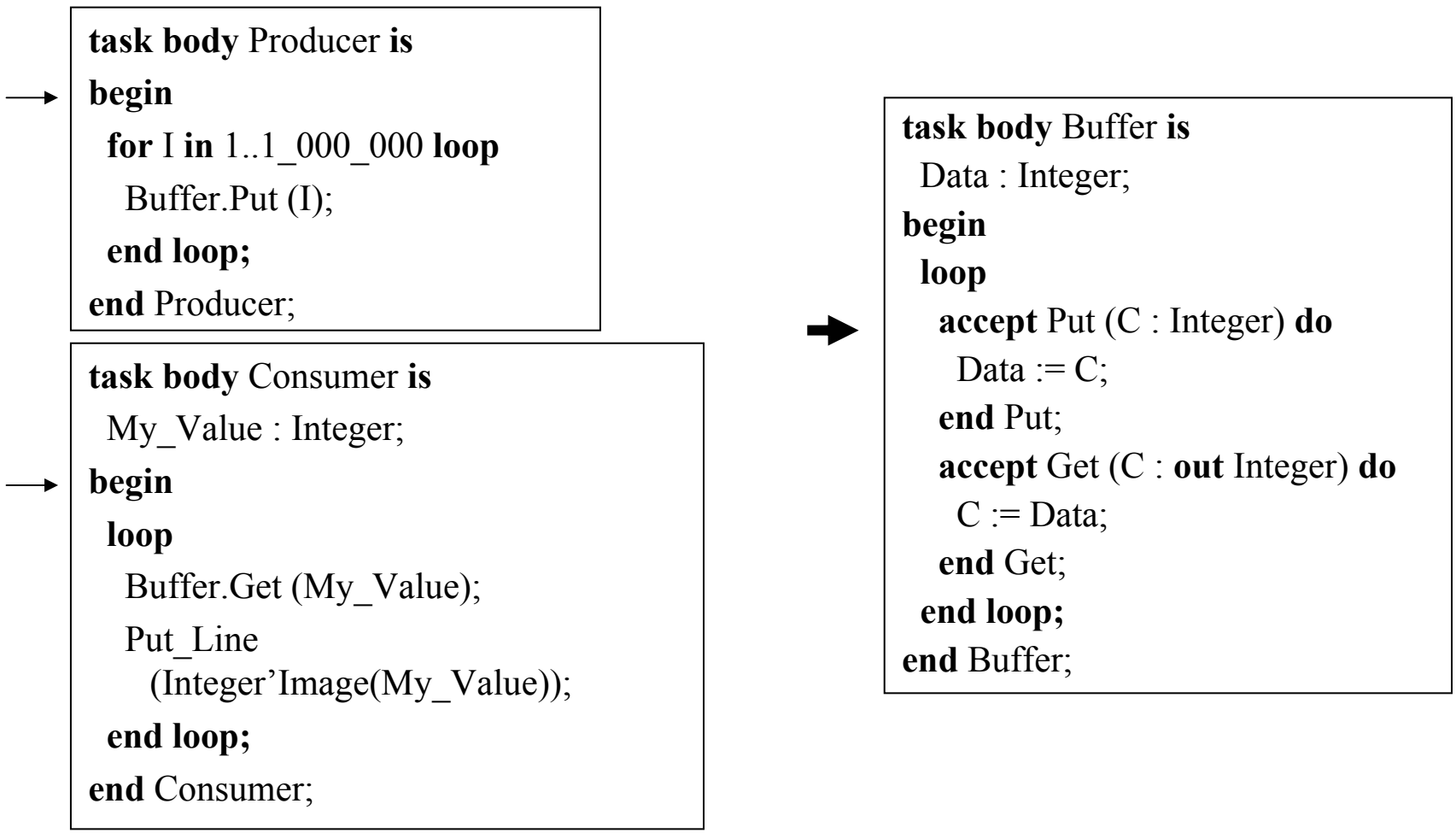
Buffer is running

```
task body Producer is
begin
  for I in 1..1_000_000 loop
    Buffer.Put (I);
  end loop;
end Producer;
```

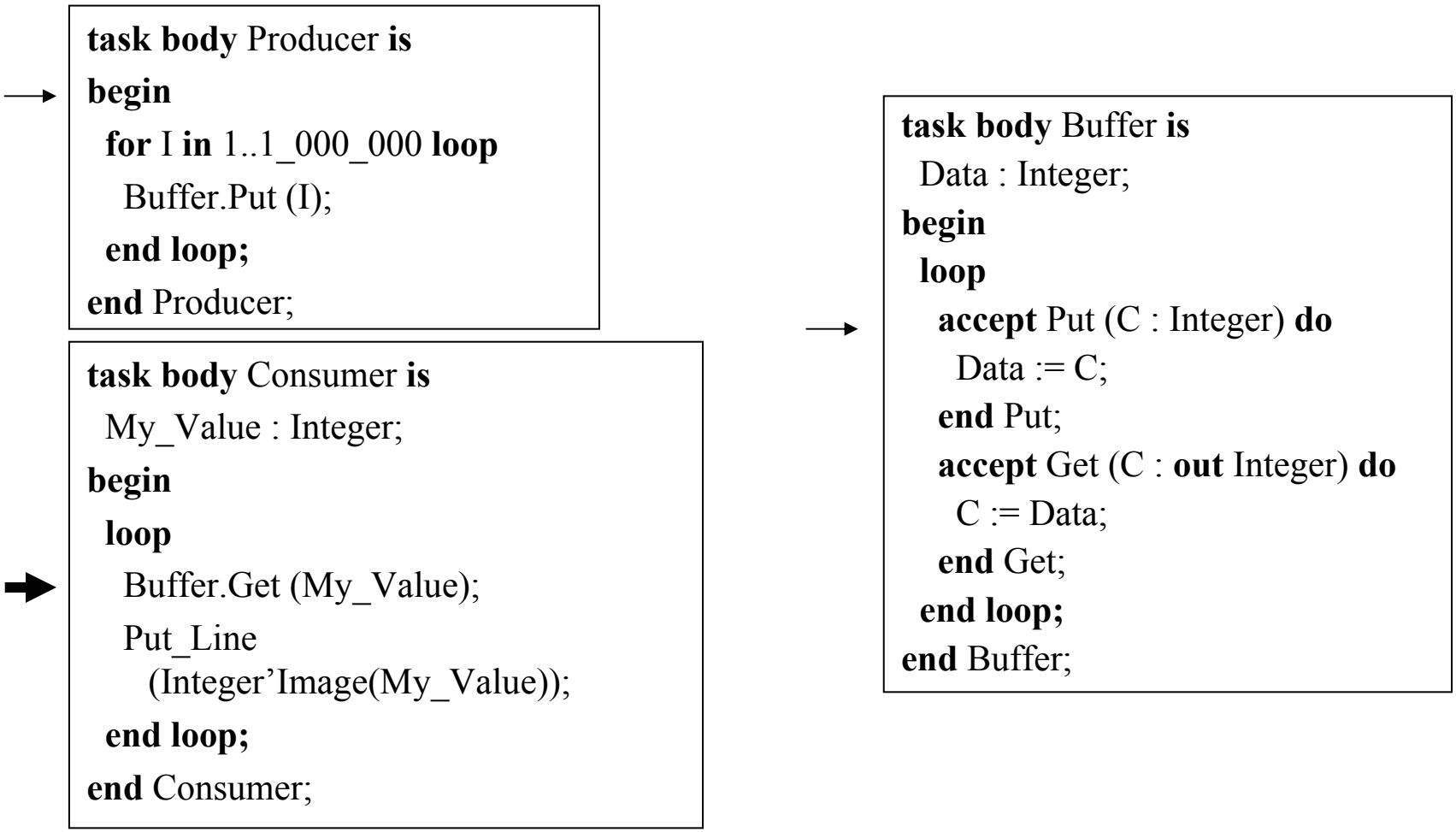
```
task body Consumer is
  My_Value : Integer;
begin
  loop
    Buffer.Get (My_Value);
    Put_Line
      (Integer'Image(My_Value));
  end loop;
end Consumer;
```

```
task body Buffer is
  Data : Integer;
begin
  loop
    accept Put (C : Integer) do
      Data := C;
    end Put;
    accept Get (C : out Integer) do
      C := Data;
    end Get;
  end loop;
end Buffer;
```

Buffer is running / suspends



Consumer is running / suspends

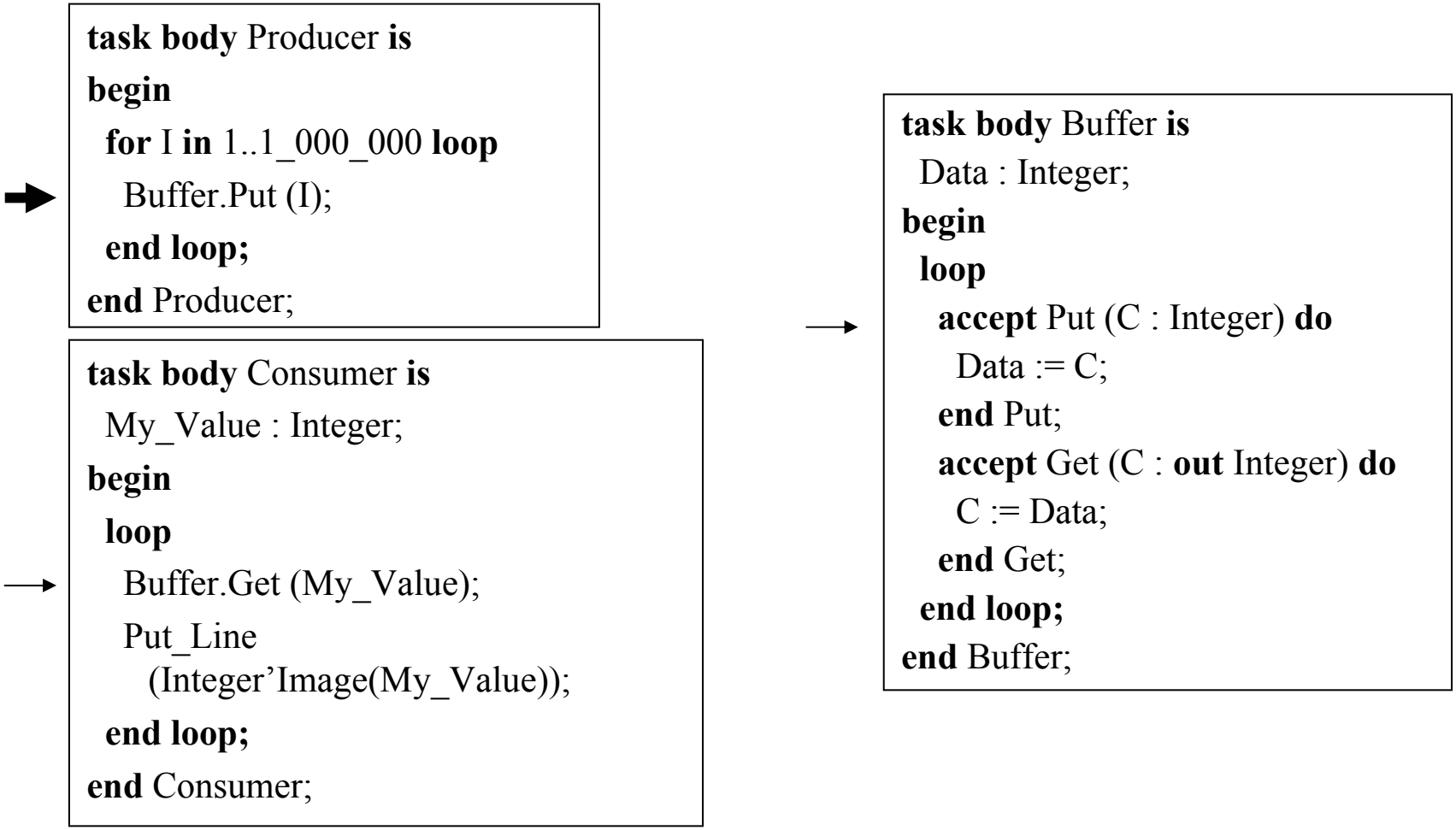


```
task body Producer is
begin
  for I in 1..1_000_000 loop
    Buffer.Put (I);
  end loop;
end Producer;
```

```
task body Consumer is
  My_Value : Integer;
begin
  loop
    Buffer.Get (My_Value);
    Put_Line
      (Integer'Image(My_Value));
  end loop;
end Consumer;
```

```
task body Buffer is
  Data : Integer;
begin
  loop
    accept Put (C : Integer) do
      Data := C;
    end Put;
    accept Get (C : out Integer) do
      C := Data;
    end Get;
  end loop;
end Buffer;
```

Producer is running / Rendez Vous



```
task body Producer is
begin
  for I in 1..1_000_000 loop
    Buffer.Put (I);
  end loop;
end Producer;
```

```
task body Consumer is
  My_Value : Integer;
begin
  loop
    Buffer.Get (My_Value);
    Put_Line
      (Integer'Image(My_Value));
  end loop;
end Consumer;
```

```
task body Buffer is
  Data : Integer;
begin
  loop
    accept Put (C : Integer) do
      Data := C;
    end Put;
    accept Get (C : out Integer) do
      C := Data;
    end Get;
  end loop;
end Buffer;
```

Producer/Buffer is running / Rendez Vous

```
task body Producer is
begin
  for I in 1..1_000_000 loop
    Buffer.Put (I); -- i equals 1
  end loop;
end Producer;
```

```
task body Consumer is
  My_Value : Integer;
begin
  loop
    Buffer.Get (My_Value);
    Put_Line
      (Integer'Image(My_Value));
  end loop;
end Consumer;
```

```
task body Buffer is
  Data : Integer;
begin
  loop
    accept Put (C : Integer) do
      Data := C;
    end Put;
    accept Get (C : out Integer) do
      C := Data;
    end Get;
  end loop;
end Buffer;
```

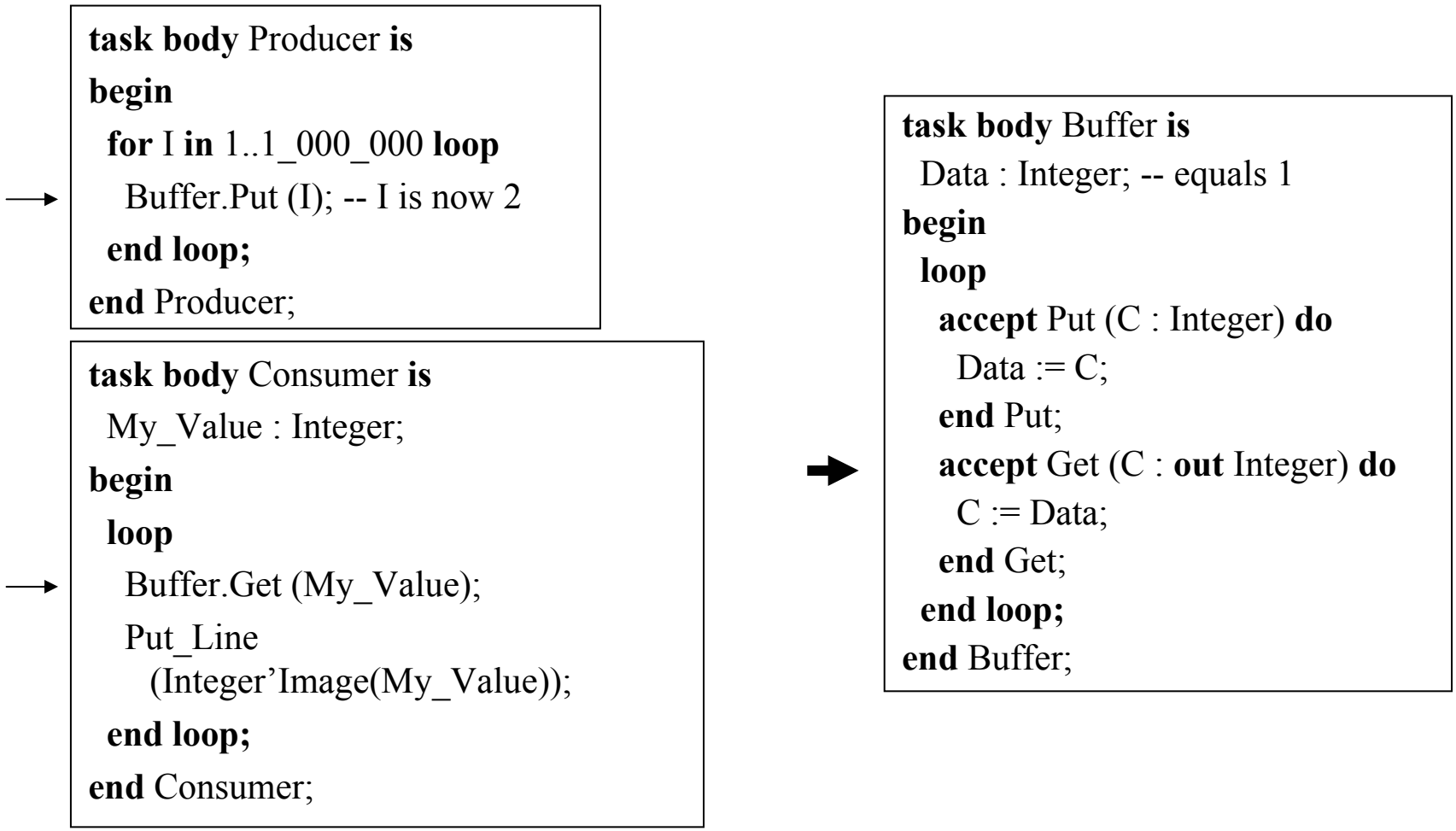
Producer is running / Suspends

```
task body Producer is
begin
  for I in 1..1_000_000 loop
    Buffer.Put (I); -- I is now 2
  end loop;
end Producer;
```

```
task body Consumer is
  My_Value : Integer;
begin
  loop
    Buffer.Get (My_Value);
    Put_Line
      (Integer'Image(My_Value));
  end loop;
end Consumer;
```

```
task body Buffer is
  Data : Integer; -- equals 1
begin
  loop
    accept Put (C : Integer) do
      Data := C;
    end Put;
    accept Get (C : out Integer) do
      C := Data;
    end Get;
  end loop;
end Buffer;
```

Buffer is running / Suspends

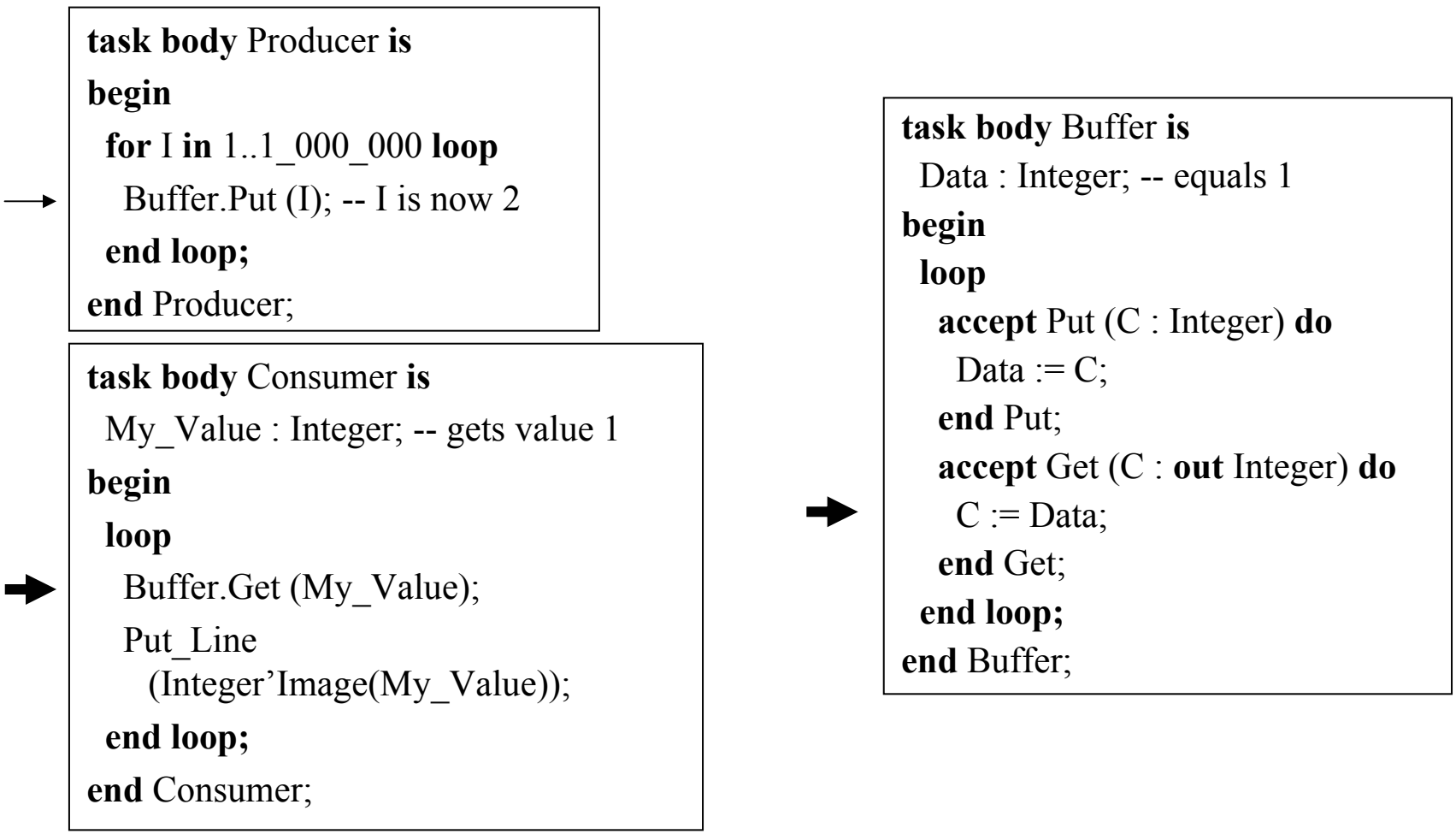


```
task body Producer is
begin
  for I in 1..1_000_000 loop
    Buffer.Put (I); -- I is now 2
  end loop;
end Producer;
```

```
task body Consumer is
  My_Value : Integer;
begin
  loop
    Buffer.Get (My_Value);
    Put_Line
      (Integer'Image(My_Value));
  end loop;
end Consumer;
```

```
task body Buffer is
  Data : Integer; -- equals 1
begin
  loop
    accept Put (C : Integer) do
      Data := C;
    end Put;
    accept Get (C : out Integer) do
      C := Data;
    end Get;
  end loop;
end Buffer;
```

Consumer/Buffer is running / Rendez Vous

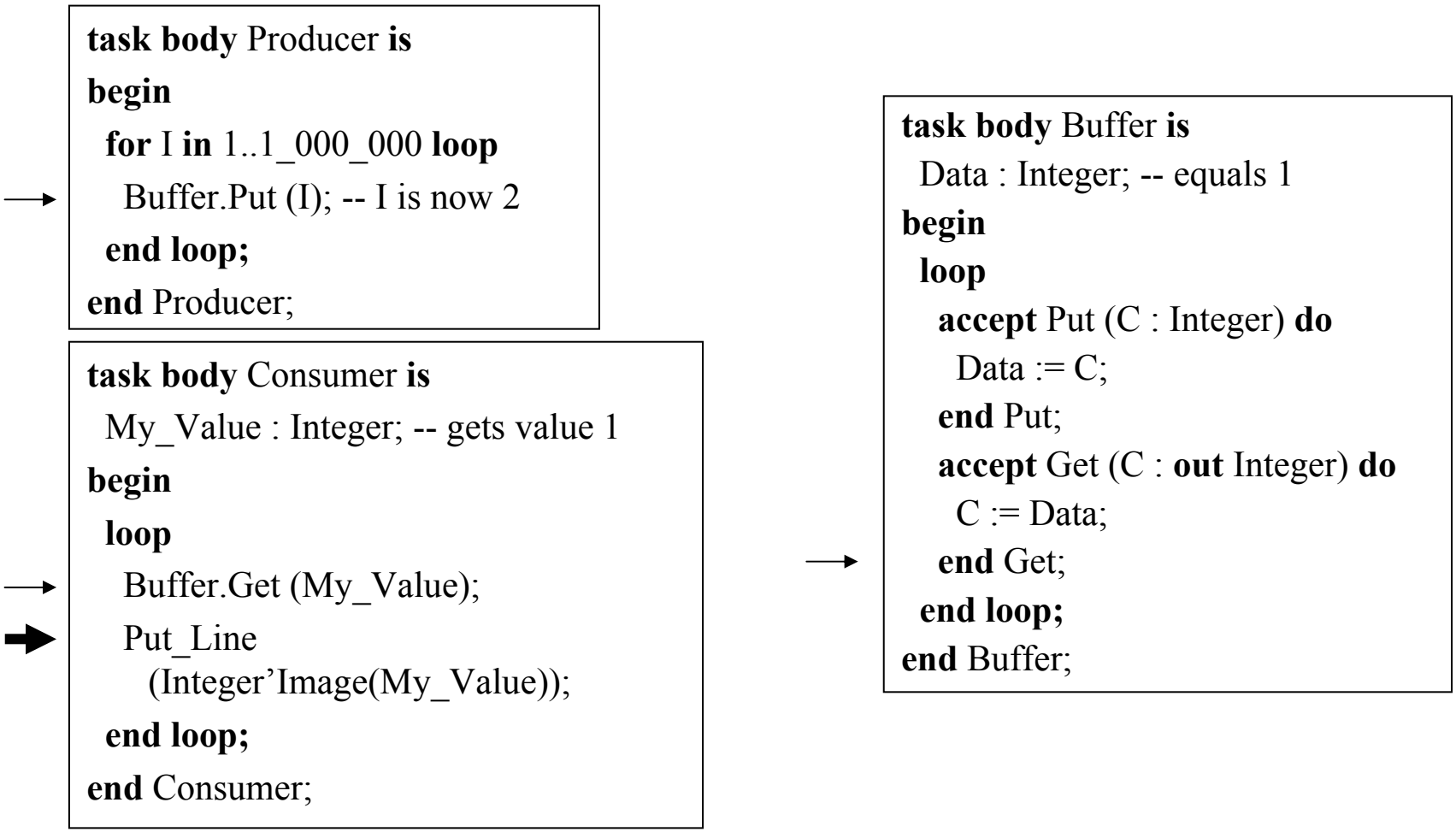


```
task body Producer is
begin
  for I in 1..1_000_000 loop
    Buffer.Put (I); -- I is now 2
  end loop;
end Producer;
```

```
task body Consumer is
  My_Value : Integer; -- gets value 1
begin
  loop
    Buffer.Get (My_Value);
    Put_Line
      (Integer'Image(My_Value));
  end loop;
end Consumer;
```

```
task body Buffer is
  Data : Integer; -- equals 1
begin
  loop
    accept Put (C : Integer) do
      Data := C;
    end Put;
    accept Get (C : out Integer) do
      C := Data;
    end Get;
  end loop;
end Buffer;
```

Consumer is running / Suspends



Extend the Buffer

```
task body Buffer is
   type Index is range 0..100;
   Data : array (Index range 1..Index'last) of Integer;
   Size : Index := 0;   In_P, Out_P : Index := 1;
begin
  loop

    select
      accept Put (C : Integer) do
        In_P := In_P mod Index'last + 1; Size := Size + 1;
        Data  (In_P) := C;
      end Put;
    or
      accept Get (C : out Integer) do
        Out_P := Out_P mod Index'last + 1; Size := Size -1;
        C := Data (Out_P);
      end Get;
    end select;

  end loop;
end Buffer;
```

There is a potential ERROR here

Final solution of task Buffer

```
task body Buffer is
   type Index is range 0..100;
   Data : array (Index range 1..Index'last) of Integer;
   Size : Index := 0;   In_P, Out_P : Index := 1;
begin
  loop
    select
      when size < Index'last =>
        accept Put (C : Integer) do
         In_P := In_P mod Index'last + 1;       Size := Size + 1;
          Data  (In_P) := C;
        end Put;
    or
      when size > 0 =>
        accept Get (C : out Integer) do
         Out_P := Out_P mod Index'last + 1;   Size := Size -1;
          C := Data (Out_P);
        end Get;
    end select;
  end loop;
end Buffer;
```

The select statement (1)

```
select
  [when logical_Expr =>]
    accept Entry_Name do
      Statements
    end Entry_Name;
or
  [when logical_Expr =>]
    accept Entry_Name do
      Statements
    end Entry_Name;
or
  delay Duration_Expr;
  Statements
or
  delay Duration_Expr;
  Statements
or
  delay until Time_Expr;
  Statements
[or
  terminate;]
end select;
```

The select statement (2)

```
select
  [when logical_Expr =>]
    accept Entry_Name do
      Statements
    end Entry_Name;
or
  [when logical_Expr =>]
    accept Entry_Name do
      Statements
    end Entry_Name;
else
  Statements
end select;
```

The select statement (3)

select

Task.Entry_Name;

else

Statements

end select;

select

Task.Entry_Name;

or

delay *Duration_Expr;*

Statements

end select;

The select statement (4)

select

Task.Entry_Name;

then abort

Statements

end select;

select

delay [until] *Dur/Time;*

then abort

Statements

end select;

Protected Objects

```
protected Buffer is
  entry Put (C : Integer);
  entry Get (C : out Integer);
  pragma Priority (5);
private
  …
  Data : Buffer;
  Size : Index;
end Buffer;
```

```
protected type Buffer_T is
  entry Put (C : Integer);
  entry Get (C : out Integer);
  pragma Priority (5);
private
  …
  Data : Buffer;
  Size : Index;
end Buffer_T;

Buffer : Buffer_T;
Another_Buffer : Buffer_T;
```

The Protected Object (2)

```
protected body Buffer is
  entry Put (C : Integer) when size < 100 is
  begin
     In_P := In_P mod Index'last + 1;
     Data (In_P) := C;
  end Put;
  entry Get (C : out Integer)  when  size > 0  is
  begin
    Out_P := Out_P  mod Index'last + 1;
    C := Data (Out_P);
  end Get;
end Buffer;
```

Same behaviour but no context switch

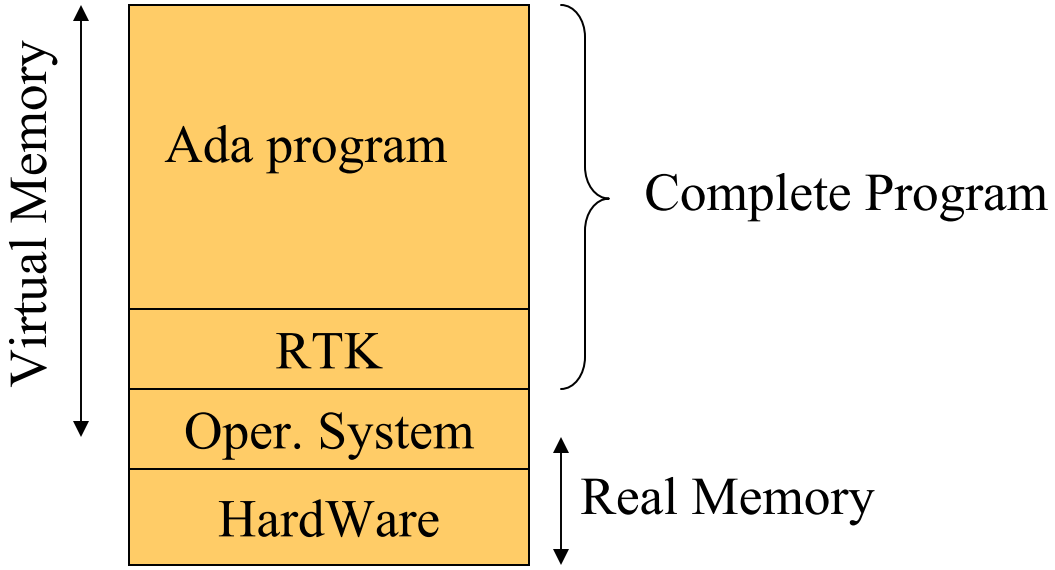
```
task Producer is
begin
  for I in 1..1_000_000 loop
    Buffer.Put (I);
  end loop;
end Producer;
```

```
task body Consumer is
  My_Value : Integer;
begin
  loop
    Buffer.Get (My_Value);
    Put_Line
      (Integer'Image(My_Value));
  end loop;
end Consumer;
```

```
protected body Buffer is
  entry Put (C : Integer) when size < 100
      is
  begin
     In_P := In_P mod Index'last + 1;
     Data (In_P) := C;
  end Put;
  entry Get (C : out Integer)  when  size
      > 0  is
  begin
    Out_P := Out_P  mod Index'last + 1;
    C := Data (Out_P);
  end Get;
end Buffer;
```

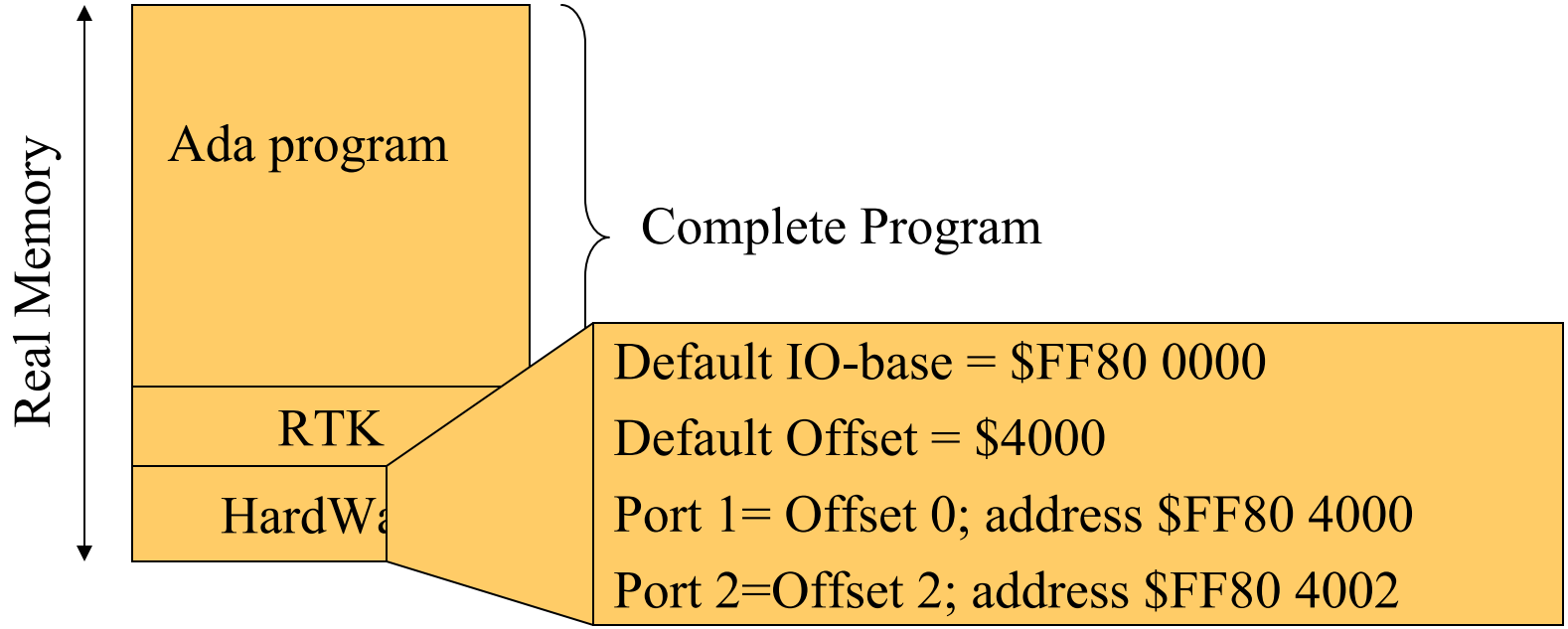
‘Workstation’ Systems

- Using an operating system (Windows NT, VxWorks, ...)
  Ada tasking can be on top or using Threads in the OS.
  Restrictions to the use of memory, instructions and hardware



Embedded Systems

- In an embedded system (no OS)
  - Ada tasking (RTK) is linked with the code
  - No restrictions in access to underlying hardware
  - There is an environment

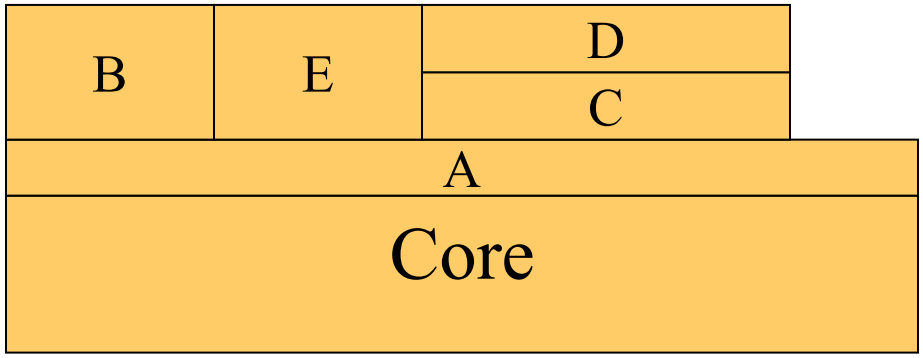


Embedded Systems

- High end system: A Pentium with 128Mbyte of RAM, Windows NT
- Medium sized system: VME-board with dRAM, VxWorks, VRTX or Lynx
- Small sized systems: Bare but RT-OS may work, but only one process (VxWorks)
- Eight bit microcontrollers: No Ada - yet

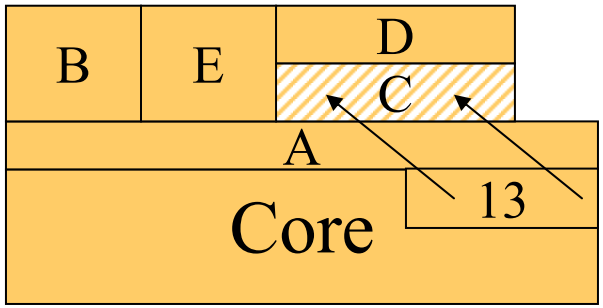
Core and Annexes

- A Predefined Language Environment
- B Interface to Other Languages
- C Systems Programming
- D Real-Time Systems
- E Distributed Systems
- F Information Systems
- G Numerics
- H Safety and Security



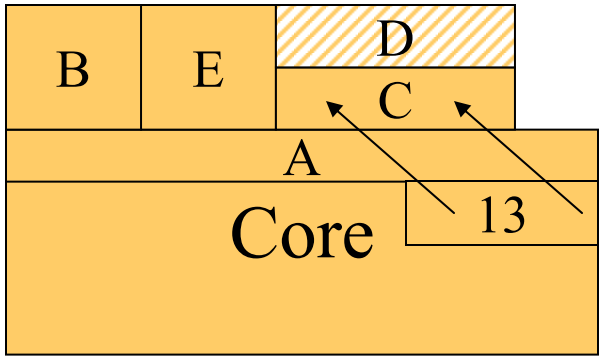
Systems Programming (C)

- The annex starts with chapter 13
  - Representation specifications
  - Package System
  - Unchecked conversions
  - Storage Management
- Annex C
  - Interrupt support
  - Task Identification and Attributes



Real-Time Systems (D)

- Priorities
- Scheduling Policies
- Entry Queuing Policies
- Dynamic Priorities
- Monotonic Time
- Synchronous Task Control
- Asynchronous Task Control
- ...



Systems Programming

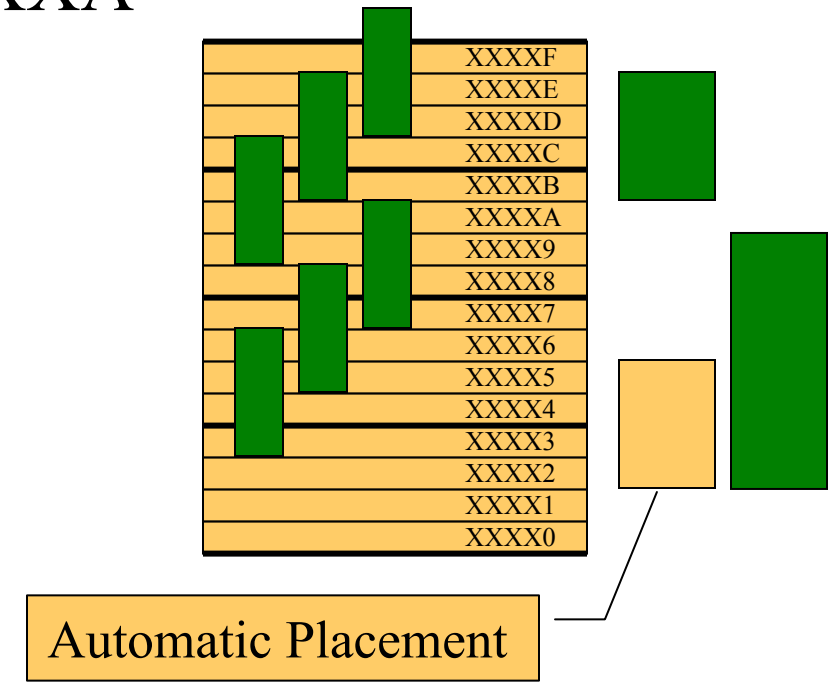
- Specification of Representation of Data
  - Layout
  - Address
  - Size
- Address manipulation
- Very low level programming (ports, machine code)
- Interrupt Handling

How to reach the bits

- A register in an IO-device has several bits (bit-fields).
- Low Level C or assembler use **and**/**or** operations
- Ada use records and rep-specs

Representation Attributes

- X'Address=XXXXA
- X'Alignment=2
- X'Size=8*8

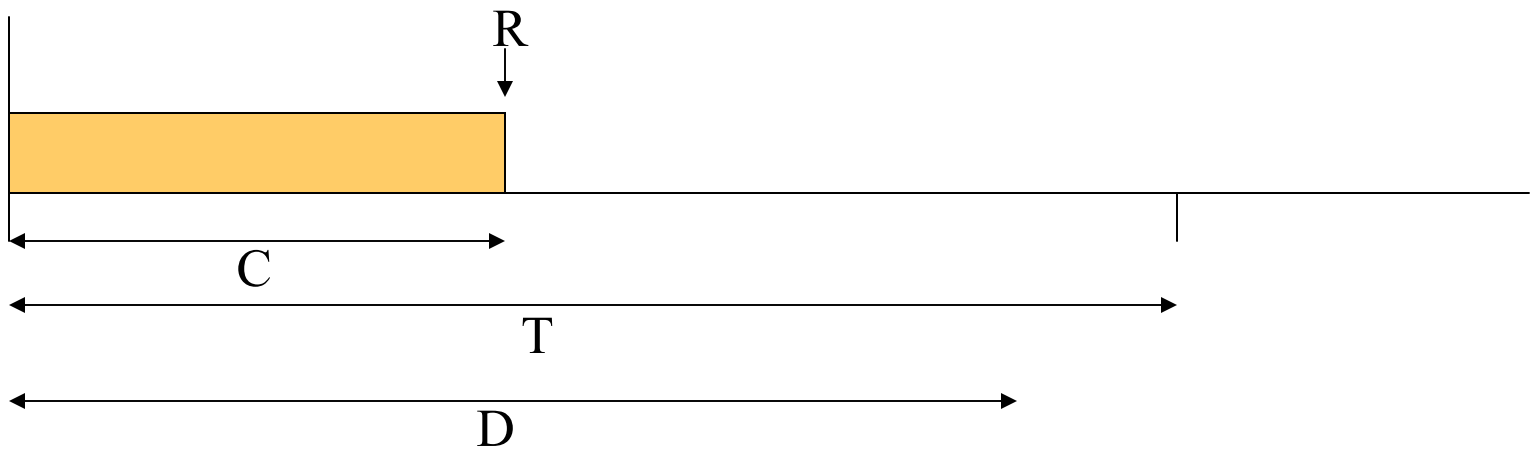


τ

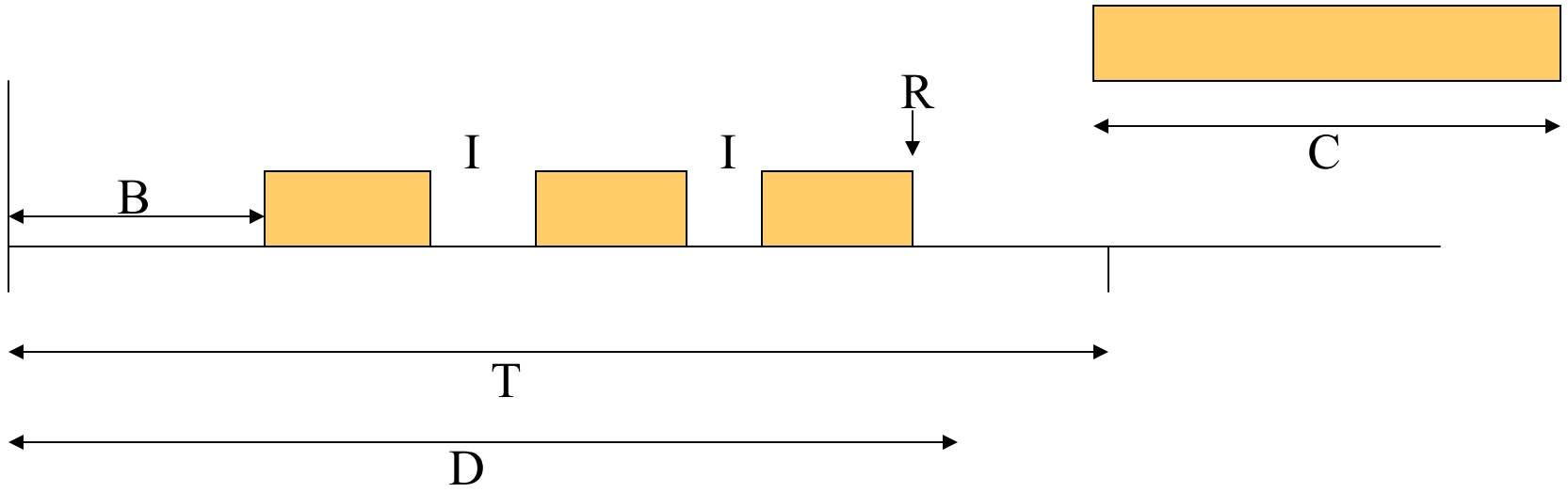
Deadline monotonic

- Like Rate monotonic, but deadline (D)
- Set of tasks
  - T Period
  - C computational time
  - D deadline
  - B Blocking
  - J Jitter
  - R Response
  - P Priority
- Low D => higher P

Basic concepts (1)



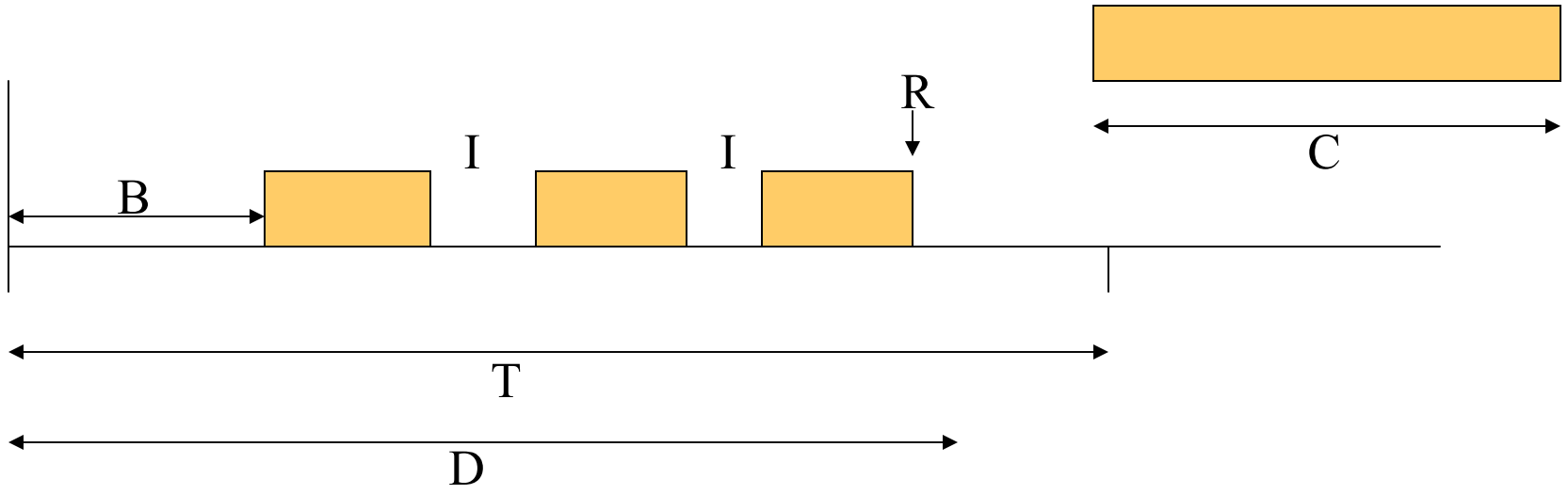
Basic concepts (2)



I Interference (interrupted by high priority tasks)

B Blocking (low priority tasks)

Basic concepts (3)



I Interference (interrupted by high priority tasks)

B Blocking (low priority tasks)

$$R_I = C_I + B_I + \sum_{J \in hp(I)} \left\lceil \frac{R_I + J_J}{T_J} \right\rceil C_J$$

τ Priority Ceiling and Inheritance

- Priority of a Task T ≤ Priority of a PO used by T
- Inheritance Task T inherits the ceiling of PO when entering the PO

```
protected type PO is
  entry …;
  pragma Priority (5);
end PO;
```

```
task type T is
  entry …;
  pragma Prioriy (4);
end T;
```

Immediate Inheritance

- As soon as you enter a PO you recieve the PO-priority – cf when PO requested
- => all blocking before start
- => no priority inversion

T3 uses PO4; T2 uses PO2 T1 uses PO4 and PO2

T1 is running takes PO4; is preempted by T2

T2 is running takes PO2; is preempted by T3

T3 needs PO4; T1 starts to run; needs PO2 before releasing PO4; **run T2**

Ada for Safety Critical Applications

- Today's safety critical systems use cyclic executives.
- Research take for granted that a system consists of processes (scheduling, priorities), and that there is communications between these.
- Process based safety critical systems - formal methods (Raven, Enea …)

Ada 83

Ada 83

- Tasking
- Rendez-vous
- Dynamic
- Hierarchy
- Termination
- ...

- Complex Run-Time
- No Formal Proofs

- For High Integrity Systems
- Subsets:
  - SPARK (No tasking)
  - Boeing
  - ...

Ada 95

Subsets:

Ada 95	
Ada 83	
•Tasking •Rendez-vous •Dynamic •Hierarchy •Termination •...	•Protected objects •ATC •Requeue •Delay until •New interrupts •Task discriminants •...

GNORT (Gnat NO Run-Time)

SPARK-95

Ravenscar
•Tasking •Protected objects •Delay until •New interrupts •Task discriminants

Ravenscar

- The Ravenscar profile has been proposed as a possible standard runtime support system suitable for safety critical real-time Ada 95 applications.
- The subset provides enough functionality for targeted systems.

Ravenscar - tasking

- Library level
- No dynamic creation
- No unchecked deallocation
- Non-terminating
- No entries
- No user defined attributes
- Keep task discriminants
- No ATC

Ravenscar - Protected Objects

- Single Entry
- Barrier a single Boolean
- Only one task in the entry queue

Ravenscar - Communication

- No Rendez vous
- No requeue
- No select statement
- Interrupts are mapped only to PO procedures

Ravenscar - Real Time

- *delay until* for delays
- No Calendar
- Clock from Real-Time package
- No dynamic priorities
- Immediate Ceiling Priority