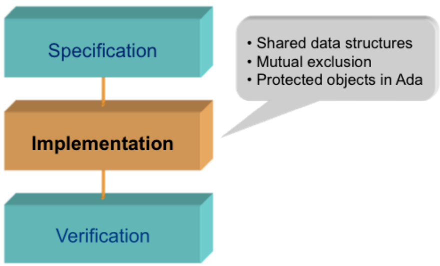
CHALMERS

Real-Time Systems



CHALMERS

Mutual exclusion

Systems with cooperating concurrent tasks require automatic handling of shared data structures.

- An important problem that has to be solved is how to guarantee that a data structure is always kept in a consistent state.
- A working solution is achieved if one makes sure that only one task at a time receive access to the data structure.
- Exclusive access to a data structure can be achieved by making sure that program code that manipulates the data structure executes under so-called mutual exclusion, that is, the code execution cannot be preempted in the most critical moment.

CHALMERS

Mutual exclusion

Program constructs that provide mutual exclusion:

- Ada 95 uses server tasks (rendezvous) or protected objects.
- Other (mainly older) programming languages (e.g. Modula-1, Concurrent Pascal, Mesa) use monitors.
- Java uses synchronized methods, a simplified version of monitors.
- Real-time kernels and operating systems use semaphores. When programming in languages (e.g. C and C++) that do not provide the constructs mentioned above, such semaphores must be used.

To guarantee mutual exclusion in the implementation itself for the constructs mentioned above, special methods must be used.

CHALMERS

Example: circular buffer

Problem: Write a server task `Circular_Buffer` in Ada that handles a circular buffer with room for 8 data records of type `Data`.

- The server task should have two entries, `Put` and `Get`.
- Producer tasks should be able to insert data records in the buffer via entry `Put`. If the buffer is full, a task that calls `Put` should be blocked.
- Consumer tasks should be able to remove data records from the buffer via entry `Get`. If the buffer is empty, a task that calls `Get` should be blocked.

We solve this on the blackboard!

CHALMERS

Mutual exclusion

When a shared data structure is handled by a server task in Ada, mutual exclusion is obtained because a rendezvous can only occur between two tasks at a time. This means that all statements between **do** and **end** must be executed before a new rendezvous can take place.

```
accept Put(D : in Data) do
  A(I) := D;
end Put;
```

The remaining statements, e.g.

```
I := (I mod N) + 1;
Count := Count + 1;
```

are only executed by one task (`Circular_Buffer`), and it is therefore no risk for data inconsistency.

CHALMERS

Mutual exclusion

Observe that, if the operations `Put` and `Get` had been implemented as ordinary procedures, mutual exclusion would not have been obtained.

Instead, the buffer data structure could very easily become corrupt and give rise to data inconsistencies.

The following example demonstrates one such case ...

CHALMERS

Mutual exclusion

Assume that **`Put`** is implemented as a procedure.

```
procedure Put(D : in Data) is
  -- declaration of A, I and Count
  ...

begin
  A(I) := D;
  I := (I mod N) + 1;
  Count := Count + 1;
end Put;
```

Now, assume that **`Put`** is called by two task:

```
task body P1 is          task body P2 is
begin                    begin
  loop                       loop
  ...                        ...
  Put(X);                    Put(Y);
  ...                        ...
  end loop;                  end loop;
end P1;                  end P2;
```

CHALMERS

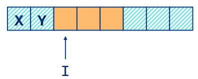
Mutual exclusion

The following execution order causes data inconsistency:

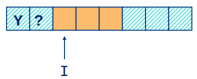
```
Put(X):                Put(Y):                Fault:

A(I) := X;
                       A(I) := Y;             -- X is overwritten
I := (I mod N)+1;
                       I := (I mod N)+1;      -- old value in
                       Count := Count + 1;    -- last data record
Count := Count + 1;
```

Consistent data:



Inconsistent data:



CHALMERS

Protected objects

Using server tasks to handle shared data structures is often an inefficient solution as task switches will occur every time a data structure has to be manipulated.

Ada 95 therefore provides an alternate solution with a language construct called protected objects.

- A protected object is an entirely passive object that offers protected operations for data being shared by multiple tasks.
- A protected object consists (similar to packages and tasks) of a specification and a body.

CHALMERS

Protected objects

A protected operation can be a function, a procedure or an entry.

- Protected procedures and entries are regarded as writers and mutual exclusion is guaranteed for these operations.
- Protected functions are regarded as readers and are therefore not allowed to modify the data of the protected object. Mutual exclusion among tasks calling protected functions is thus not required.

CHALMERS

Example: integers as protected objects

Specification:

```
protected type Shared_Integer(Initial_Value : Integer) is

  function Read return Integer;
  procedure Write(New_Value : Integer);
  procedure Increment(By : Integer);

private
  The_Data : Integer := Initial_Value;

end Shared_Integer;
```

Declaration of protected variable:

```
My_Data : Shared_Integer(42);
```

CHALMERS

Example: integers as protected objects

Body:

```
protected body Shared_Integer is

  function Read return Integer is
  begin
    return The_Data;
  end Read;

  procedure Write(New_Value : Integer) is
  begin
    The_Data := New_Value;
  end Write;

  procedure Increment(By : Integer) is
  begin
    The_Data := The_Data + By;
  end Increment;

end Shared_Integer;
```

Observe that the protected object is entirely passive, i.e. lacks active code.

CHALMERS

Protected objects

Protected entries are guarded by a Boolean expression called a barrier. This barrier must evaluate to "true" to allow the entry body code to be executed.

```
entry E1(X : in Data) when boolean expression is
begin
  ...
end E1;
```

Ada 95 requires that a Boolean expression (barrier) is given for each protected entry.

CHALMERS

Protected objects

Semantics of protected entries:

- A task that calls an entry whose barrier evaluates to "false" is queued (blocked).
- A barrier is evaluated ...
  1. … in conjunction with calls to the protected entry if the barrier may have changed since it was last evaluated.
  2. … every time a call to a protected procedure/entry completes, if there are tasks being queued at the entry and the barrier may have changed since it was last evaluated.
- Barriers are not evaluated at calls to protected functions as such functions cannot change the state of a protected object.
- Queued tasks that are waiting for barriers have priority over other callers when the protected object becomes available.

CHALMERS

Protected objects

Restrictions of protected entries:

- A task executing the code of a protected procedure/entry may not use any operation that can block.
- This means that the following operations may not be used in the body of a protected procedure/entry:
  - Call to an entry
  - Delay statement
  - Call to a sub-program whose code contains a potentially blocking operation
- Exception: A tasks executing the code of an entry body may execute a **requeue** statement.
  - **Requeue** places the task executing the statement in a given entry queue. Because the task using **requeue** releases the protected object at the same time, there is no conflict.

CHALMERS

Example: circular buffer

Problem: Write a protected object **Circular_Buffer** that handles a circular buffer with room for 8 data records of type **Data**.

- The protected object should have two entries, **Put** and **Get**.
- Producer tasks should be able to insert data records in the buffer via entry **Put**. If the buffer is full, a task that calls **Put** should be blocked.
- Consumer tasks should be able to remove data records from the buffer via entry **Get**. If the buffer is empty, a task that calls **Get** should be blocked.

We solve this on the blackboard!