Ada - Protected

Protected

- allows encapsulation and safe usage of shared variables without the need for any explicit mutual exclusion and signaling mechanisms
- tasks that safely manage locally-declared data with regard to multiple clients
- **protected** modules, based on the well-known computer science concept of *monitor*
- a protected module encapsulates a data structure and exports subprograms that operate on it under automatic mutual exclusion
- it also provides automatic, implicit signaling of conditions between client tasks

Protected

- a protected module can be either a single protected object or a protected type, allowing many protected objects to be created.
- a protected module can export only procedures, functions and entries
- its body may contain only the bodies of procedures, functions and entries
- the protected data is declared after **private** in its specification, but is accessible only within the protected module's body

Protected

- protected procedures and entries may read and/or write its encapsulated data, and automatically exclude each other
- protected functions may only read the encapsulated data
- multiple protected function calls can be concurrently executed in the same protected object, with complete safety
- but protected procedure calls and entry calls exclude protected function calls, and vice versa
- exported entries and subprograms of a protected object are executed by its calling task

Protected

- like a task entry, a protected entry can use a guard to control admission
- this provides automatic signaling, and ensures that when a protected entry call is accepted, its guard condition is True, so that a guard provides a reliable precondition for the entry body

```
protected type Protected_Buffer_Type is
entry Insert (An_Item : in Item);
entry Remove (An_Item : out Item);
private Buffer : Item;
        Empty : Boolean := True;
end Protected_Buffer_Type;
```

Protected

```
protected body Protected_Buffer_Type is
entry Insert (An_Item : in Item) when
Empty is
begin Buffer := An_Item;
Empty := False; end Insert;
entry Remove (An_Item : out Item) when
not Empty is
begin An_Item := Buffer; Empty := True;
end Remove;
end Protected_Buffer_Type;
```

Protected

- the state variable Empty ensures that messages are alternately inserted and removed,
- no attempt can be made to take data from an empty buffer
- this is achieved without explicit signaling or mutual exclusion constructs, whether in the calling task or in the protected type itself
- calling a protected entry or procedure is exactly the same as that for calling a task entry
- easy to replace one implementation of the abstract type by the other, the calling code being unaffected

Protected

```
task type Semaphore_Task_Type is
entry Initialize (N : in Natural);
entry Wait;
entry Signal;
end Semaphore_Task_Type; ...

task body Semaphore_Task_Type is
Count : Natural;
begin
accept Initialize (N : in Natural) do Count := N; end Initialize;
loop
select
when Count > 0 => accept Wait do Count := Count - 1; end Wait;
or
accept Signal; Count := Count + 1;
end select;
end loop;
end Semaphore_Task_Type;
```

Protected

The task can be used like:

Full, Free : Semaphore_Task_Type; …

Full.Initialize (0); Free.Initialize (nr_Slots); …

Free.Wait; Full.Signal;

- Initialize and Signal operations of this protected type are unconditional, so they are implemented as protected procedures
- the Wait operation must be guarded and is therefore implemented as an entry.

Reimplemented with protected:

Protected

```
protected type Semaphore_Protected_Type is
procedure Initialize (N : in Natural);
entry Wait;
procedure Signal;
private Count : Natural := 0;
end Semaphore_Protected_Type;
```

- Unlike the task type above, this does not ensure that Initialize is called before Wait or Signal, and Count is given a default initial value instead

Protected

```
protected body Semaphore_Protected_Type is
procedure Initialize (N : in Natural) is
begin Count := N;
end Initialize;
entry Wait when Count > 0 is
begin Count := Count - 1; end Wait;
procedure Signal is
begin Count := Count + 1; end Signal;
end Semaphore_Protected_Type;
```

Exception

- `Tasking_Error`, when a task cannot be activated because the operating system has not enough resources, e.g.
- when calling a terminated task it raises `Tasking_Error` in the caller

Attributes

- X'Callable is an Ada attribute where X is any task object. If the task is completed or has been terminated, this attribute is false. Otherwise, this attribute is true (i.e. the task is callable).
- calling X'Callable can result in a race condition. X'Callable may be true at the time the attribute value is read, but it may become false at the time action is taken based on the value read. Once X'Callable is false, however, it can be expected to stay false.

Attributes

- X'Terminated is an Ada attribute where X is any task object. This attribute indicates if X has terminated (true) or not (false).
- calling X'Terminated can result in a race condition. X'Terminated may be false at the time the attribute value is read, but it may become true at the time action is taken based on the value read. Once X'Terminated is true, however, it can be expected to stay true.