Parallel Programming Tasks

Tasks

- A *task unit* is a program unit that is running concurrently with the main program and other tasks of an Ada program
- is called a *task* in Ada terminology, and is similar to a *thread, e.g. in Java*
- the execution of the main program is also a task, the anonymous environment task (parent task)
- a task unit has both a declaration and a body (mandatory) a task body may be compiled separately as a subunit, but a task may not be a library unit, nor a generic.
- every task depends on a *master (or parent)*, which is the surrounding declarative region - a block, a subprogram, another task, or a package - that declared the task

Tasks

- The execution of a master does not complete until all its dependent tasks have terminated.
- The environment task is the master of all other tasks; it terminates only when all other tasks have terminated.
- Task units are similar to packages in that a task declaration defines entities exported from the task, whereas its body contains local declarations and statements of the task.

Tasks

```
task Action is
declarations of exported identifiers
end Action; ...
task body Action is local
declarations and statements
end Action;
```

A task declaration can be simplified, if nothing is exported:

```
task No_Exports_Actions;
```

Tasks

```
procedure Housework is
task Clean;
task Cook;
task body Clean is ... end Clean;
task body Cook is ... end Cook;
-- the two tasks are automatically
created and begin their execution
begin -- Housework
null; -- Housekeeping waits here for
them to terminate
end Housework;
```

Tasks

- It is possible to declare task types, allowing task units to be created dynamically, and placed in data structures

```
task type TaskTypeName is
...
end TaskTypeName; ...
Task_1, Task_2 : TaskTypeName;
...
task body TaskTypeName is
...
end TaskTypeName;
```

- Task types are **limited**, i.e. they are restricted in the same way as limited private types, so assignment and comparison are not allowed

Tasks

- **Rendezvous** – communication via entry points
- The only entities that a task may export are entries
  An **entry** looks like a procedure. It has a name identifier and may have **in**, **out** or **in out** parameters.
- Communication from task to task: by the *entry calls*
- Information passes between tasks through the actual parameters of the entry call
- Tasks encapsulate data structures within and operate on them by entry calls, in a way analogous to the use of packages for encapsulating variables

Tasks

- an entry is executed by the called task, not the calling task, which is suspended until the call completes
- if the called task is not ready to service a call on an entry, the calling task waits in a (FIFO) queue associated with the entry
- the interaction between calling task and called task is known as a *rendezvous*
- the calling task requests rendezvous with a specific named task by calling one of its entries
- a task accepts rendezvous with any caller of a specific entry by executing an **accept** statement for the entry.
- If no caller is waiting, it is held up
- the entry call and accept statement behave symmetrically

Tasks – buffer example

```
task type Buffer_Task_Type is
entry Insert (An_Item : in Item);
entry Remove (An_Item : out Item);
end Buffer_Task_Type; ...
Buffer_Pool: array (0..10) of Buffer_Task_Type;
Item1 : Item; ...
task body Buffer_Task_Type is
Datum : Item;
begin
loop
accept Insert (An_Item : in Item) do Datum := An_Item;
end Insert;
accept Remove (An_Item : out Item) do An_Item := Datum;
end Remove;
end loop;
end Buffer_Task_Type; ...
Buffer_Pool(1).Remove (Item1);
Buffer_Pool(2).Insert (Item1);
```

Tasks – selective wait

- To avoid being held up when it could be doing productive work, a task often needs the freedom to accept a call on any one of a number of alternative entries
- the *selective wait* statement allows a task to wait for a call on any of two or more entries
- if only one of the alternatives in a selective wait statement has a pending entry call, then that one is accepted
- if two or more alternatives have calls pending, the implementation is free to accept any one of them, it chooses one at random, introduces *bounded non-determinism* into the program

Tasks

```
task type Variable_Task_Type is

entry Store (An_Item : in Item);
entry Fetch (An_Item : out Item);

end Variable_Task_Type;

task body Variable_Task_Type is
Datum : Item;
begin
… *
end Variable_Task_Type;

x, y : Variable_Task_Type; -- x, y two tasks
```

Tasks

```
accept Store (An_Item : in Item) do
Datum := An_Item;
end Store;
loop
select
accept Store (An_Item : in Item) do
Datum := An_Item;
end Store;
or
accept Fetch (An_Item : out Item) do An_Item :=
Datum;
end Fetch;
end select;
end loop;
```

Tasks

```
item1 : Item; ...
x.Store(An_Expression); ...
x.Fetch (item1); y.Store (item1);
```

- a task of Variable_Task_Type must be given an initial value by a first Store operation before any Fetch operation can be accepted
- the acceptance of any alternative can be conditional by a *guard*, which is *Boolean* precondition for acceptance
- it makes easy to write monitor-like tasks
- no need for an explicit signaling mechanism, nor for mutual exclusion
- an alternative with a True guard is said to be *open*
- it is an error if no alternative is open when the selective wait statement is executed, and raises Program_Error exception

Tasks

```
task Cyclic_Buffer_Task_Type is
entry Insert (An_Item : in Item);
entry Remove (An_Item : out Item);
end Cyclic_Buffer_Task_Type; ...
task body Cyclic_Buffer_Task_Type is Q_Size : constant := 100;
subtype Q_Range is Positive range 1 .. Q_Size; Length : Natural
range 0 .. Q_Size := 0;
Head, Tail : Q_Range := 1;
Data : array (Q_Range) of Item;
begin  .....
end Cyclic_Buffer_Task_Type;
```

Tasks

```
select when Length < Q_Size =>
accept Insert (An_Item : in Item) do
Data(Tail) := An_Item;
end Insert;
Tail := Tail mod Q_Size + 1;
Length := Length + 1;
or
when Length > 0 =>
accept Remove (An_Item : out Item) do
An_Item := Data(Head);
end Remove;
Head := Head mod Q_Size + 1; Length :=
Length - 1;
end select;
```