PARALLELISM IN Ada:

The execution of an Ada program consists of the execution of one or more tasks. Each task is a thread of control that proceeds independently and concurrently between the points where it interacts with other tasks.

Task interaction takes various forms and include:

- The activation and termination of a task
- One task killing another task (via the use of an **`abort`** statement)
- Synchronous communication (by calling an **`entry`** of another task)
- Synchronous communication (by accepting an **`entry`** call of another task)

Every task must be declared and specified before its use.

- Declaration defines WHAT a task can do;
- Specification defines HOW a task does what it is supposed to do.

Examples of declarations of single tasks:

```
task User;  -- has no entries

task Parser is
  entry Next_Lexeme(L : in  Lexical_Element);
  entry Next_Action(A : out Parser_Action);
end;

task Controller is
  entry Request(Level)(D : Item);  -- a family of entries
end Controller;
```


Examples of declarations of task types:

```
task type Server is
  entry Next_Work_Item(WI : in Work_Item);
  entry Shut_Down;
end Server;


task type Keyboard_Driver(ID : Keyboard_ID := New_ID) is
  entry Read (C : out Character);
  entry Write(C : in  Character);
end Keyboard_Driver;
```

Examples of task objects:

```
Agent    : Server;
Teletype : Keyboard_Driver(TTY_ID);
Pool     : array(1 .. 10) of Keyboard_Driver;

Example of access type designating task objects:

type Keyboard is access Keyboard_Driver;
Terminal : Keyboard := new Keyboard_Driver(Term_ID);
```

On task activation:

```
procedure P is
  A, B : Server;    -- elaborate the task objects A, B
  C    : Server;    -- elaborate the task object C
begin
  -- the tasks A, B, C are activated together before the
first statement
  ...
end; -- procedure P will quit only after A, B, C are dead
```


RENDEZVOUS:

an ADA technique for enforcing mutual exclusion,
task synchronization and intertask communication

RENDEZVOUS PROTOCOL:

- exactly two tasks may rendezvous: **a caller** and **a server**
- the caller calls an **entry** in the server
- the server, when it is ready, issues the **accept statement** to receive the call
- if the caller calls an entry for which the server did not issue as yet an accept, the caller is made to wait
- if the server issues an accept for an entry which the caller did not call yet, the server is made to wait (at this accept) for a caller to call the entry
- rendezvous begins when the call is accepted
- during rendezvous, the caller waits while the server processes the accept statement. Information may be exchanged the caller and the server via the **parameters** of the entry call
- rendezvous ends when the server completes processing of the accept statement

ASPECTS OF RENDEZVOUS:

- the caller(s) must know the existence of the server(s), and the various server entries
- the server(s) need not to know any caller(s)
  - they just accept calls from any caller
- many callers may attempt to call one server
- only one caller at a time may rendezvous with a given server
- other callers attempting to rendezvous with a server are kept waiting
- after a rendezvous, any waiting callers are served on a first come, first serve basis

ACCEPT STATEMENT

Example of use, showing how to control access to a shared resource:

```
task RESOURCE_CONTROLLER is  -- task specification
  entry GET_CONTROL;
  entry RELINQUISH_CONTROL;
end RESOURCE_CONTROLLER;
.
.
task body RESOURCE_CONTROLLER is   -- task body
begin
  loop
    accept GET_CONTROL;
    accept RELINQUISH_CONTROL;
  end loop;
end RESOURCE_CONTROLLER;
.
.
RESOURCE_CONTROLLER.GET_CONTROL;   -- example of use
.....; -- statement(s) using the resource
RESOURCE_CONTROLLER.RELINQUISH_CONTROL;
```

MODUS OPERANDI:

Tasks voluntarily cooperate with `RESOURCE_CONTROLLER` to ensure mutual exclusion. If several tasks call `GET_CONTROL` at once, only one will be accepted, all other clients' requests will be queued FIFO

CAVEATS:

This is essentially the same as a binary semaphore. If one task violates the "gentlemen' s agreement", mutual exclusion cannot be guaranteed:

Example of erroneous or malicious use:

```
RESOURCE_CONTROLLER.RELINQUISH_CONTROL;
RESOURCE_CONTROLLER.GET_CONTROL;
....; -- statements for illegal manipulation of resource
```

EXAMPLE: PRODUCER - CONSUMER RELATIONSHIP

- A producer task deposits an 80-character card image in a buffer;
- a consumer task removes the characters from the buffer one at a time until the buffer is empty.

Issues of cooperation:

- a producer may not deposit a next line until the buffer empty;
- a consumer may not begin removing characters until a line has been deposited;
- after all characters have been removed, a consumer must wait for the producer to deposit a new line.

```
type CARDIMAGE is array (1..80) of CHARACTER;

task CONVERTCARDIMAGE is
      entry DEPOSITCARD (CARD: in CARDIMAGE);
      entry READCHARACTER (NEXTCHARACTER: out CHARACTER);
end;

----------------------------------------------------------

task body CONVERTCARDIMAGE is
      CARDBUFFER: CARDIMAGE;
begin
      loop
          accept DEPOSITCARD (CARD: in CARDIMAGE) do
              CARDBUFFER := CARD;
          end DEPOSITCARD;
          for POSITION in 1..80 loop
              accept READCHARACTER (NEXTCHARACTER : out
                                    CHARACTER) do
                  NEXTCHARACTER := CARDBUFFER(POSITION);
              end READCHARACTER;
          end loop;
      end loop;
end;
```

Producer and consumer tasks are unaware of each other. They are aware only of the existence of the **CONVERTCARDIMAGE** task, which coordinates their work, viz.:

```
task PRODUCER; -- specification (normally in one file)

----------------------------------------------------------

task body PRODUCER is -- implementation (in another file)
    NEWCARD: CARDIMAGE;
begin
    loop
        -- statements to create NEWCARD
        CONVERTCARDIMAGE.DEPOSITCARD (NEWCARD);
    end loop;
end;

----------------------------------------------------------

task CONSUMER; -- specification (normally in one file)

----------------------------------------------------------

task body CONSUMER is -- implementation (in another file)
    NEWCHARACTER: CHARACTER;
begin
    loop
        CONVERTCARDIMAGE.READCHARACTER (NEWCHARACTER);
        -- statements processing NEWCHARACTER
    end loop;
end;
```

THE SELECT STATEMENT:

Entry calls need not be accepted in a prescribed, rigid fashion. A task may be willing to accept several entry calls, one at a time but in indefinite order:

```
select
    when CONDITION1 = > accept ENTRY1;
        sequence of statements;
    or when CONDITION2 = > accept ENTRY2;
        sequence of statements;
    or . . .
    else
        sequence of statements;
end select;
```

Rules of selection:

- Each of the conditions (called *guards*) is evaluated once to be `TRUE` or `FALSE`. If found `TRUE`, then the following `accept` statement is considered open;
- There may be several open `accept` statements. In particular, an `accept` statement not preceded by a guard is always open;
- If there is an `else` part and no entry call to one of the open `accept` statements has been made, then the `else` part is immediately executed. If there is no `else` part, the task waits for an entry call.
- If there are no open accepts, the `else` part is executed. If there is no `else` part, then a `TASKING_ERROR` exception is raised.

EXAMPLES:

Selective accept:

```
task body Server is
  Current_Work_Item : Work_Item;
begin
  loop
    select
      accept Next_Work_Item(WI : in Work_Item) do
        Current_Work_Item := WI;
      end;
      Process_Work_Item(Current_Work_Item);
    or
      accept Shut_Down;
      exit;       -- Premature shut down requested
    or
      terminate;  -- Normal shutdown at end of scope
    end select;
  end loop;
end Server;
```

Timed entry calls:

```
select
  Controller.Request(Medium)(Some_Item);
or
  delay 45.0;
  -- controller too busy, try something else
end select;
```

Conditional entry calls:

```
select
  Controller.Request(Medium)(Some_Item);
or
  delay 45.0;
  -- controller too busy, try something else
end select;
```

Time-limited calculation:

```
select
  delay 5.0;
  Put_Line("Calculation does not converge");
then abort
  -- This calculation should finish in 5.0 seconds;
  -- if not, it is assumed to diverge.
  Horribly_Complicated_Recursive_Function(X, Y);
end select;
```


EXAMPLE: THE RING BUFFER

The select statement allows the buffer task to service appropriate entry calls. In particular:

- The guard **BUFFERSINUSE < BUFFERS** allows a call to **WRITEPACKET** to be accepted whenever space is available,
- The guard **BUFFERSINUSE > 0** allows a call to **READPACKET** to be accepted whenever the buffer contains data.

```
type DATAPACKET is array (1..80) of CHARACTER;
----------------------------------- task specification:
task RINGBUFFER is
  entry READPACKET  (PACKET: out DATAPACKET);
  entry WRITEPACKET (PACKET:  in DATAPACKET);
end;
----------------------------------- task implementation:
task body RINGBUFFER is
  BUFFERS: constant INTEGER := 20;
  RING: array (1..BUFFERS) of DATAPACKET;
  BUFFERSINUSE: INTEGER range 0..BUFFERS := 0;
  NEXTIN, NEXTOUT: INTEGER range 1..BUFFERS := 1;
begin
  loop
    select
      when BUFFERSINUSE < BUFFERS = >
          accept WRITEPACKET(PACKET: in DATAPACKET) do
              RING (NEXTIN) := PACKET;
          end;
          BUFFERSINUSE := BUFFERSINUSE + 1;
          NEXTIN := NEXTIN mod BUFFERS + 1;
      or when BUFFERSINUSE > 0 = >
          accept READPACKET(PACKET: out DATAPACKET) do
              PACKET : = RING(NEXTOUT);
          end;
          BUFFERSINUSE :=  BUFFERSINUSE - 1;
          NEXTOUT:= NEXTOUT mod BUFFERS + 1;
  end select;
  end loop;
end RINGBUFFER;
```