The Tasking Facility In Ada

It is used to allow concurrent tasking, where tasks can be of non-communicating and communicating natures:

A) **Non-communicating Tasks:**

procedure P **is**
  task T_1; … (SPECS interface of task T_1)… ; **end** T_1;
  task body T_1 **is**
    begin … end T_1;
        ***
  task T_n; … (SPECS interface of task T_n)… ; **end** T_n;
  task body T_n **is**
    begin … end T_n;

begin -- code for P **(It is very important to notice that the Ada's tasking facility is unique in the domain of Ada's peers HLLs)**

- tasks T_1 … T_n will start once the code of P is entered,
- without any explicit ***call*** statement for any task.
- P will not terminate until all of its tasks terminate, last terminating task could be any of the *n* tasks!

end P;

- **Tasks can not be a lib unit**, they must be declared within some program unit.
- **Ada will prevent any subprogram that hosts any task(s) from terminating until all of its hosted tasks terminate**. The **reason** is related to the disappearance (upon return/termination) of an environment (that of the hosting subprogram) that is needed by the task(s), otherwise they become **orphan tasks**.

B) Communicating Tasks Communicate and Synchronize via "Rendezvous":

Two tasks (***client*** and ***server***) will rendezvous via an "***entry***" (procedure like construct) which is to be declared in the server and "called" by the client upon requesting some service (from such server). The client "**call**" on some server "***entry***" must be "accepted" by the server, pending on the server availability, before any service is granted.
At the client: the format of an ***entry***'s **call** is

<server-name>.<entry-name>

At the server side there will be an "***accept***" statement that protect some "critical" code, which manages a shared memory space among a group of communicating client's tasks, to guarantee **mutual exclusion access** to such common memory space. Hence, the server will guarantee one task only executing at a time the critical code that manipulate the shared memory.

Entry declaration at the *server* task:

accept <**entry-name**> **do** <body-critical code> **end;**

In case of more than one client's tasks are asking for the same service entry, all requests are to be queued (FCFS), whenever the service is not ready (busy at other entries), until the server is ready (accepting from such entry) to accept requests for that entry again. A ready (free) server will pick the task-request at the head of the "queue of waiting requests". Hence, the entry "**call**" is like a **message** and the "***entry***" itself is like a ***mailbox***.

RENDEZVOUS-BASED CONCURRENCY: USE OF TASKS IN ADA

GOAL: To guarantee mutual exclusive access to the shared buffer between the "SCANNER" and "PARSER" modules of the "COMPILER".

```
procedure COMPILER () is

  task  SCANNER;                          task  PARSER;
  task body SCANNER is                    task body PARSER is
     t: token;                               x: token;
      ...                                     ...
   begin                                   begin
     ...                                     ...
     PUT (t); --wait until served            GET (x);--wait until served
     ...                                     ...
   end;                                    end;

  task BUFFERING is
     entry PUT (X: in ITEM);  --queue of requests to put an item in the buffer
     entry GET (X: out ITEM); --queue of requests to get an item from the buffer
  end;

  task body BUFFERING is
     N:constant:=8;
     A:array(1..N) of ITEM;          -- array of tokens
     I,J:INTEGER range 1..N:=1;
     COUNT:INTEGER range 0..N:=0;

  begin
    loop          -- do one clause at a time
       select     -- select a clause with "true" condition and non-empty queue.
          when  COUNT < N =>                -- clause 1
          accept PUT (X:in ITEM) do  -- waits if the PUT queue is empty
                A(I):=X;
          end;       -- end of PUT; the caller now continues (end of rendezvous)
          I:=I mod N+1; COUNT:=COUNT+1;
       or
          when COUNT>0 =>                   -- clause 2
          accept GET (X:out ITEM) do  -- waits if the GET queue is empty
                X:=A(J);
          end;       -- end of GET; the caller now continues (end of rendezvous)
          J:=J mod N+1; COUNT:=COUNT-1;
       or
         terminate;--terminate clause, when all other depending tasks terminate
       end select;
    end loop;
  end BUFFERING;

  ....
begin    -- COMPILER starting code where all tasks are to start executing
         -- concurrently.

  -- do nothing (usually) waiting for all tasks to terminate, then terminate.

end COMPILER;
```

The Rendezvous protocol:

1) If the client issues a call (message) to a server *entry* before the server is ready to accept it, then the **client** task will be **blocked** (forced to wait) until its call (request/message) is accepted, leaving its call message in the server entry mailbox (a major difference from a "regular" procedure call).
2) If the server reaches an accept statement (mailbox) and find no client waiting at it (i.e., the mailbox is empty), it blocks until a client calls; otherwise it serves any waiting call, given it is feasible to do so (e.g., no meaning of getting an element from an empty buffer).

In fact, the use of **only one server** that guards some resource and provides many entries (services) access to different client tasks to manipulate **guarantees a mutual exclusive access to such shared resource(s)** (e.g., buffers). The reason is that such server can not be serving more than one request at a time!

The "***select***" clause:

- It allows the **server to serve more than one client tasks asynchronously and feasibly**.
- It groups a number of accept clauses and places a conditional feasibility check on each, where no clause is to be executed unless the condition is true.
- Hence, it looks at all of its hosted accept statements and selects those with "true" conditions ("**open**") and nonempty mailboxes ("**ready**"); then arbitrarily (non-deterministically) selects one only to be executed (if more than one are open).
- If none of the accept clauses are **open,** or **open** but with empty mailbox, then the ***select*** clauses must have either an open ***delay*** or ***terminate*** clauses to force a wait, at the select entry, for some specified ***delay*** or until one of the accept clauses becomes open and ready, then loops again.
- The ***select*** clause ***terminates*** when ***all*** depending tasks terminate.

Exception Handling in Ada

WHY?

Ada applications include many embedded critical military systems that must act reasonably under wide variety of conditions, i.e., must be **robust**. Hence, in case of failure, they must respond and take a positive action instead of quitting, think embedded Ada control code within a nuclear missile!

HOW?

1) **Define exception flags** (one for each critical exceptional event at run time).
   At the interface declare the flag with type "**exception**":

   e.g., QUE_IS_FULL : **exception ;**

   There are also **predefined exceptions' flags in the system package "STANDARD"**:

   NUMERIC_ERROR : **exception;** -- division by zero
   PROGRAM_ERROR: **exception;**
   -- existing a function not via the "return"
   STORAGE_ERROR : **exception;**
   -- allocating object via **new** with no enough memory
   TASKING_ERROR : **exception;**
   -- error in communicating tasks
   CONSTRAINT_ERROR: **exception;**
   - attempting to access a ***null*** *access* (pointer) value object,
   - assigning a value outside its recipient subtype range,
   - referring a non-existent component of a variant record object, and
   - indexing an array with an index outside its subtype

They are raised automatically upon the occurrence of the exception anywhere in the code by the system, yet the programmer can still raise any of them, explicitly (as next).

2) **Signal the exception occurrences** upon detection:
(similar to a software "trap")

e.g., if en-queuing an element but the target queue is full, the code of the "*enque*" will have:

```
if  IS_FULL(Q) then
   raise QUE_IS_FULL;  --  the flag has to be visible via
                       --  the above declaration
end
```

3) **Respond to the raised exception** by jumping to and executing the appropriate *exception handling code* (**EHC**):

For example, write the EHC in the body of the ***caller*** (the raiser of the exception flag) or its caller, or in any of its ancestors, in the calling chain, until the main program.

The EHC might be general to handle more than one exception flags:

```
begin                          -- FILL_QUEU code
   < regular code >
exception

    when  QUE_IS_FULL →
          [declare  <decl-seq> ]
           begin  <CODE> end;
     when <OTHER EXCEPTION NAME> →
          ***
     when  others → <HANDLE ALL OTHER EXCEPTION>
            -- only one other is allowed and it must be the last
end   FILL_QUEU;
```

Exception handlers are bounded dynamically to the environment of the caller (or its callers' descendent) of the subprogram which issued the "raise" statement, i.e., following the dynamic chain.

- Hence, the system will look first locally in the current subprogram for the exception handler, if not there then it follows the dynamic chain looking for the exception handler with the same flag name that has been raised, and stops at the first subprogram that finds matching the searched EHC definition, otherwise the program terminates. Every looked subprogram **without the target handler** will be **abundant** deleting its AR from the stack.