Explicación Práctica de ADA (Rendezvous)

Ejercicios

Links a los archivos con audio

El archivo en formato MP4 de la explicación con audio se encuentra comprimido en el siguiente link:

- https://drive.google.com/u/1/uc?id=1tqVK1jHUUdzDSvXgYgxTcQGIqWivEOUm&export=download

Lenguaje ADA (Rendezvous)

- Una aplicación en ADA es un único programa con un cuerpo principal, y para que trabaje concurrentemente se desarrollan TASK (tareas) dentro de ese programa que pueden ejecutar independientemente y contienen primitivas de sincronización/comunicación.

- Se pueden implementar directamente TASK (es una única instancia de esa tarea) o se puede declarar un TASK TYPE para crear varias instancias iguales de ese tipo (arreglo, puntero, instancia simple).

- Cada TASK o TASK TYPE tienen una especificación donde se declaran las operaciones exportadas (llamadas ENTRYs) y un BODY donde se implementa la tarea.

Sintaxis – estructura del programa

```
Procedure nombre is
        --Especificación de un TASK
        TASK nombreT1 is
            Declaración de los ENTRYs de la tarea
        End nombreT1;
        --Especificación de un tipo TASK
        TASK TYPE nombreTipoT is
            Declaración de los ENTRYs del tipo tarea
        End nombreTipoT;
        Declaración de variables del tipo nombreTipoT
        --Cuerpo de las tareas
        TASK BODY nombreT1 is
            Implementación de la tarea
        End nombreT1;
        TASK BODY nombreTipoT is
            Implementación del tipo tarea
        End nombreTipoT;
Begin
    Cuerpo principal del programa
End nombre;
```

Sintaxis – especificación de las tareas

- Las tareas pueden exportan operaciones o no, si lo hacen deben definir los ENTRYs correspondientes en la especificación.
- Los ENTRYs pueden o no tener parámetros, si los tiene se debe declarar el nombre, el tipo de datos y el tipo de parámetro (IN, OUT o IN OUT).

- Especificación de tarea y tipo tarea sin ENTRYs

```
TASK nombreT1;

TASK TYPE nombreTipo1;
```

- Especificación de tareas con ENTRYs sin parámetros

```
TASK nombreT2 IS
      ENTRY nombreE1;
      ENTRY nombreE2;
 END nombreT2;
```

- Especificación de tareas con ENTRYs con parámetros

```
TASK nombreT3 IS
      ENTRY nombreE3(A, B: IN integer; C: OUT char; D: IN OUT boolean);
END nombreT3;
```

- Declaración de Variables de tipo Tarea (nombreTipo1)

```
arregloT1: ARRAY (1..10) OF nombreTipo1;
nombreVariable: nombreTipo1
```

Sintaxis – cuerpo de las tareas

- El cuerpo de las tareas es donde se implementa la tarea. En esta parte no hay diferencia entre las TASK y los TASK TYPE (no se pone TYPE).

```
TASK BODY nombreT1 IS
    variables
BEGIN
     implementación de la tarea
END nombreT1;
```

- Si se hubiese definido un arreglo de un tipo tarea (por ejemplo ***arregloT1*** de la página anterior), se generaran varias tareas idénticas de ese tipo (en el ejemplo ***10*** tareas). La única manera de identificar una de otra es por su posición en el arreglo, pero la tarea por si misma no conoce ese dato. Si lo necesita saber, otra tarea o el cuerpo principal del programa se lo deberá comunicar para que guarde ese "identificador" en una variable.
  La forma de hacerlo lo veremos en alguno de los ejemplos.

Sintaxis – comunicación/sincronización

- ***Rendezvous*** (encuentro) es el principal mecanismo de sincronización en Ada y también es el mecanismo de comunicación primario.

- ADA tiene otros mecanismos de comunicación QUE NO PUEDEN ser utilizados en la materia, como por ejemplo el uso de variables compartidas.

- La comunicación (el encuentro) entre dos tareas se realiza por medio de un ***Entry Call*** por parte del proceso que realiza el "pedido" y un ***Accept*** por parte del que "sirve" (resuelve) ese "pedido".
  Ambas tareas deben esperar a que el ***Accept*** termine su ejecución para poder continuar (se mantienen ***sincronizados*** durante el ***rendezvous***).

- La comunicación es bidireccional, y es fundamental que esta característica se aproveche (siempre que sea posible y no reduzca la concurrencia) al resolver los problemas en ADA.

Sintaxis – entry call

Hay 3 formas opciones de ***Entry Call*** que se diferencian por el tiempo de demora.

- ***Entry Call*** simple: la ejecución demora al llamador hasta que la otra tarea termine de ejecutar el ***accept*** correspondiente.

Ejemplos de *Entry Call* a las tareas de la página 4:

- Tarea2.nombreE1;
- Tarea3.nombreE3(5, x, y, z);
  --Los parámetros de tipo IN pueden ser variables, constantes o expresiones; los de tipo OUT o IN OUT deben ser variables
- ArregloT2(3).nombreE4;
  -- Siendo ArregloT2 un arreglo de tareas (no está en los ejemplos de la página 4).

- ***Entry Call Condicional (select – else):*** no espera a que le acepten el pedido, si la otra tarea no está lista para realizar el ***accept*** a su pedido inmediatamente, entonces lo cancela y realiza otra cosa.

- ***Entry Call Temporal (select – or delay):*** espera a lo sumo un tiempo a que la otra tarea realice el ***accept*** a su pedido, pasado el tiempo cancela el entry call y realiza otra cosa.

Sintaxis – entry call

- ***Entry Call Condicional (select – else):*** no espera a que le acepten el pedido, si la otra tarea no está lista para realizar el ***accept*** a su pedido inmediatamente, entonces lo cancela y realiza otra cosa.

Ejemplos de *Entry Call Condicional* a una tarea de la página 4:

select
Tarea3.nombreE3(5, x, y, z);
sentencias **S1** a realizar sólo si se completo el *entry call*
else
sentencias **S2** que se harán si se cancelo el *entry call*
end select;

Intenta hacer el ***entry call nombreE3****, y si* ***Tarea3*** *inmediatamente acepta su pedido, se demora hasta que se termina el accept; luego (ya terminado la sincronización con* ***Tarea3****) realiza el conjunto de sentencia S1 (puede no estar este conjunto de sentencias S1); al terminar finaliza el select.*
Si ***Tarea3*** *no acepta inmediatamente su pedido, cancela el entry call y realiza el conjunto de sentencias* ***S2*** *que está después del* ***else*** *(si no hubiese que hacer nada en el* ***else*** *igual se debe poner alguna sentencia, en ese caso* ***S2*** *sería sentencia nula* ***null*** *que no hace nada); al terminar finaliza el select****.***

Sintaxis – entry call

- ***Entry Call Temporal (select – or delay):*** espera a lo sumo un tiempo a que la otra tarea realice el ***accept*** a su pedido, pasado el tiempo cancela el *entry call* y realiza otra cosa.

Ejemplos de *Entry Call Temporal* a una tarea de la página 4:

select
Tarea3.nombreE3(5, x, y, z);
sentencias **S1** a realizar sólo si se completo el *entry call*
or delay *tiempo*
sentencias **S2** que se harán si se cancelo el *entry call*
end select;

*Intenta hacer el **entry call nombreE3** y espera a lo sumo **tiempo** a que **Tarea3** acepte su pedido. Si antes de ese tiempo le aceptan el pedido, entonces se demora hasta que se termina el accept; luego (ya terminado la sincronización con **Tarea3**) realiza el conjunto de sentencia S1 (puede no estar este conjunto de sentencias S1); al terminar finaliza el select.*
*Si pasado **tiempo** la **Tarea3** aún no acepta su pedido, cancela el entry call y realiza el conjunto de sentencias **S2** que está después del **or delay** (si no hubiese que hacer nada en el **or delay** igual se debe poner alguna sentencia, en ese caso **S2** sería sentencia nula **null** que no hace nada); al terminar finaliza el select.*

Sintaxis – accept

- Por cada ***entry*** declarado en la especificación de una tarea, en el cuerpo se debe hacer al menos un ***accept*** para dicho ***entry***.

- Por cada ***entry*** existe en la tarea una cola implícita (no se puede manipular por la tarea) de entry calls pendientes.

- El *accept* puede tener un ***cuerpo***, en cuyo caso hasta que no se termine de ejecutar el mismo no se pierde la sincronización con el *entry call*. O puede no tener un ***cuerpo***, en cuyo caso inmediatamente termina la sincronización con el *entry call*.

- El *accept* demora la tarea hasta que haya al menos un *entry call* en la cola implícita asociada; saca al primero de ellos y, dependiendo del tipo de accept hace:
  - ***Si no tiene cuerpo:*** termina inmediatamente el accept y ambas tareas continúan con su ejecución.
  - ***Si tiene cuerpo:*** copia los parámetros reales del llamado en los parámetros formales (si el *entry* no tuviese parámetros no haría este paso); ejecuta las sentencias que están en el cuerpo; cuando termina los parámetros formales de salida son copiados a los parámetros reales (si tuviese parámetros de tipo OUT o IN OUT). Luego ambas tareas continúan con su ejecución.

Sintaxis – accept

Ejemplos de *ACCEPT* de las tareas de la página 4:

- Ejemplo de accept sin cuerpo en la tarea nombreT2

ACCEPT nombreE1;

- Ejemplo de accept con cuerpo de un entry sin parámetros en la tarea nombreT2

ACCEPT nombreE2 **IS**
sentencias **S1** que forman el ***cuerpo del accept***
END nombreE2;

- Ejemplo de accept con cuerpo de un entry con parámetros en la tarea nombreT3

ACEPT nombreE3 (A, B: IN integer; C: OUT char; D: IN OUT boolean) **IS**
sentencias **S1** que forman el ***cuerpo del accept***
END nombreE3;

*Los parámetros en el accept se ponen completos (igual que en la especificación), y SÓLO son conocidos y se pueden usar dentro del **cuerpo del accept**, fuera de este los parámetros ya no existen. Si alguno de esos parámetros se debe usar fuera del cuerpo del accept, entonces dentro de él se debe copiar el valor en una variable de la tarea, y luego se usa esa variable.*

Las sentencias que forman el conjunto S1 puede ser cualquier sentencia válida de ADA (incluso otros ACCEPT o ENTRY CALL de cualquiera de los 3 tipos)

Sintaxis – accept

- ADA brinda la posibilidad de implementar ***wait selectivos*** por medio de ***comunicación guardada*** (como la de PMS). ***Select para los Accept.*** Permite tener varias alternativas de ***ACCEPT*** que pueden o no tener asociada una condición booleana (no se puede utilizar los parámetros del *entry* como parte de la condición) utilizando la clausula ***when***.

Ejemplos de SELECT de *ACCEPT* de las tareas de la página 4:

```
SELECT
        ACCEPT nombreE1;
        sentencias S1 a realizar sólo si se completo el accept nombreE1
OR
        WHEN (condición) => ACCEPT nombre E2 IS
                                    sentencias S2 que forman el cuerpo del accept
                                END nombreE2;
                                sentencias S3 a realizar sólo si se completo el accept nombreE2
END SELECT;
```

*En este ejemplo el **Select** tiene dos alternativas de ACCEPT, la primera de ellas no tiene condición asociada (se considera TRUE) y el ACCEPT no tiene cuerpo. En cambio la segunda alternativa tiene una condición booleana y tiene un cuerpo (**sentecias S2**). Cuando la tarea llega a este select, espera hasta que al menos una de las alternativas sea exitosa (condición boolena TRUE y que haya entry calls pendientes para ese entry) y selecciona una de ellas (de las exitosas) de forma no determinística para ejecutarla. Si todas las alternativas fallan (todas tuviesen condición booleana y fuese falsa) termina el select sin ejecutar nada.*
*Al elegir una alternativa para ejecutar (supongamos la segunda en este caso), primero ejecuta el ACCEPT (de la forma que se vio en la página anterior) y luego se ejecuta el conjunto de **sentencias S3** (si lo tuviese) pero en este punto ya libero a la tarea que hizo el entry call.*

Sintaxis – accept

- A su vez se le puede agregar al final un **ELSE** o un **OR DELAY** que actúa de forma similar al de los ***entry call:***
  - En el caso del ELSE, si no se puede ejecutar ninguna alternativa en forma inmediata, entonces se ejecuta el conjunto de sentencias asociadas al ELSE.
  - En el caso del OR DELAY, espera a los sumo un tiempo a poder aceptar alguna alternativa, si no lo logra ejecuta el conjunto de sentencias asociadas al OR DELAY.

Ejemplos de SELECT de *ACCEPT* de las tareas de la página 4:

SELECT
ACCEPT nombreE1;
sentencias **S1** a realizar sólo si se completo el accept *nombreE1*
OR
WHEN (condición) **=> ACCEPT** nombre E2 **IS**
sentencias **S2** que forman el ***cuerpo del accept***
END nombreE2;
sentencias **S3** a realizar sólo si se completo el accept *nombreE2*
OR DELAY tiempo
sentencias **S4** a realizar si paso *tiempo* sin poder aceptar ninguna alternativa
END SELECT;

Sintaxis – accept

- Cada ***entry*** tiene asociado un atributo que puede ser consultado SÓLO por la tarea a la que pertenece el *entry* que indica la cantidad de entry call pendientes (cantidad de elementos de la cola implícita asociada al entry): **NombreDelEntry'count**

- Este atributo puede ser usado en los ***when***, por ejemplo para dar prioridad a un entry sobre otro. Como veremos en alguno de los ejercicios.

- Algo importante para tener en cuenta es que NO TIENE SENTIDO usarlo en el *when* de un *accept* dentro de un *select* para comprobar que haya un *entry call* pendiente. Por ejemplo:

  when (nombreE1'count > 0) => ACCEPT nombreE1;

  Ya que si no hay un pedido pendiente (es decir que *nombreE1'count = 0*) esta alternativa no podrá ser elegida por el momento ya que no se puede hacer el *accept* inmediatamente. Y además, esto podría llevar a producir BUSY WAITING si se hace lo mismo en todas las alternativas del *select*.

EJERCICIO 1

Se debe modelar la atención en un banco por medio de un único empleado. Los clientes llegan y son atendidos de acuerdo al orden de llegada.

Lo primero es definir la estructura del programa: que tareas y tipos de tareas se necesitar.

En este problema hay una TASK ***empleado*** y arreglo de un TASK TYPE para los ***clientes***.

```
Procedure Banco1 is

Task Empleado ….

Task Type Cliente ....

arrClientes: array (1..N) of Cliente;

Begin
    …
End Banco1;
```

EJERCICIO 1

En primer instancia vamos a realizar una solución donde el cliente envía el pedido y el empleado le responde.

Procedure Banco1 is

```
Task empleado is
    Entry Pedido (Datos: IN texto);
End empleado;

Task type cliente is
     Entry Respuesta (Res: IN texto);
End cliente;

arrClientes: array (1..N) of Cliente;

Task Body cliente is
     Resultado: texto;
Begin
     Empleado.Pedido ("datos");
     Accept Respuesta (Res: IN texto) do
        Resultado := Res;
     End Respuesta;
End cliente;
```

No aprovecha la comunicación bidireccional de rendezvous

```
Task Body Empleado is
     R, D: texto;
Begin
   loop
      Accept Pedido (Datos: IN texto) do
           D := Datos;
      End Pedido;
      R := resolverPedido(D);
      arrClientes(…).Respuesta(R);
   end loop;
End empleado;

Begin
   null;
End Banco1;
```

EJERCICIO 1

El cliente no debe hacer nada entre que hace el pedido y recibe la respuesta. Por el otro lado, el empleado cuando acepta un pedido inmediatamente lo resuelve y le envía el resultado al cliente. Por lo tanto la resolución del pedido se debe hacer EN EL CUERPO DEL ACCEPT, y evitar así una comunicación.

```
Procedure Banco1 is

  Task empleado is
      Entry Pedido (D: IN texto; R: OUT texto);
  End empleado;

  Task type cliente;

  arrClientes: array (1..N) of Cliente;

  Task Body cliente is
       Resultado: texto;
  Begin
       Empleado.Pedido ("datos", Resultado);
  End cliente;

  Task Body empleado is
  Begin
     loop
        accept Pedido (D: IN texto; R: OUT texto) do
             R := resolverPedido(D);
        end Pedido;
     end loop;
  End empleado;

Begin
   null;
End Banco1;
```

¿Cómo se asegura la atención en orden de llegada?

Los *entry call* a un *entry* se almacenan en la cola implícita del mismo de acuerdo al orden de llegada.

EJERCICIO 2

Se debe modelar la atención en un banco por medio de un único empleado. Los clientes llegan y esperan a lo sumo 10 minutos a ser atendido de acuerdo al orden de llegada. Pasado ese tiempo se retira sin ser atendido.

Partimos de la solución anterior

```
Procedure Banco2 is
  Task empleado is
      Entry Pedido (D: IN texto; R: OUT texto);
  End empleado;
  Task type cliente;
  arrClientes: array (1..N) of Cliente;
  Task Body cliente is
       Resultado: texto;
  Begin
       Empleado.Pedido ("datos", Resultado);
  End cliente;

  Task Body empleado is
  Begin
     loop
        accept Pedido (D: IN texto; R: OUT texto) do
             R := resolverPedido(D);
        end Pedido;
     end loop;
  End empleado;

Begin
   null;
End Banco2;
```

¿Cómo resolvemos lo del tiempo de espera?

EJERCICIO 2

Para modelar la espera máxima de 10 minutos debemos utilizar un *entry call* temporal en el cliente.

```
Procedure Banco2 is
  Task empleado is
     Entry Pedido (D: IN texto; R: OUT texto);
  End empleado;
  Task type cliente;
  arrClientes: array (1..N) of Cliente;
  Task Body cliente is
       Resultado: texto;
  Begin
     SELECT
        Empleado.Pedido ("datos", Resultado);
     OR DELAY 600.0
         NULL;
     END SELECT;
  End cliente;
  Task Body empleado is
  Begin
     loop
        accept Pedido (D: IN texto; R: OUT texto) do
              R := resolverPedido(D);
        end Pedido;
     end loop;
  End empleado;
Begin
   null;
End Banco2;
```

EJERCICIO 3

Se debe modelar la atención en un banco por medio de un único empleado. Los clientes llegan y esperan a ser atendidos; pueden ser Regulares o Prioritarios. El empleado los atiende de acuerdo al orden de llegada pero dando prioridad a los clientes Prioritarios

Partimos de la solución anterior del ejercicio 1.

```
Procedure Banco3 is
   Task empleado is
      Entry Pedido (D: IN texto; R: OUT texto);
   End empleado;
   Task type cliente;
   arrClientes: array (1..N) of Cliente;
   Task Body cliente is
      Resultado: texto;
   Begin
      Empleado.Pedido ("datos", Resultado);
   End cliente;
   Task Body empleado is
   Begin
      loop
         accept Pedido (D: IN texto; R: OUT texto) do
            R := resolverPedido(D);
         end Pedido;
      end loop;
   End empleado;
Begin
   null;
End Banco3;
```

¿Cómo resolvemos prioridad de los clientes?

EJERCICIO 3

Pasamos el tipo de cliente como parámetros en el entry call.

```
Procedure Banco3 is
   Task empleado is
      Entry Pedido (Tipo: IN texto; D: IN texto; R: OUT texto);
   End empleado;
   Task type cliente;
   arrClientes: array (1..N) of Cliente;
   Task Body cliente is
        Resultado: texto;
   Begin
        if (“es cliente prioritario”) then
             Empleado.Pedido (“PRIORITARIO”, “datos”, Resultado);
        else
             Empleado.Pedido (“REGULAR”, “datos”, Resultado);
        end if;
   End cliente;
```

¿Cómo se implementaría el Body del empleado?

EJERCICIO 3

Agregamos el parámetro *Tipo* en el *accept*.

```
Task Body empleado is
Begin
   loop
      accept Pedido (Tipo: IN texto; D: IN texto; R: OUT texto) do
            R := resolverPedido(Tipo, D);
      end Pedido;
   end loop;
End empleado;
```

NO RESPETARÁ LA PRIORIDAD. Si el primer pedido es un cliente *Regular*, lo atenderá aunque haya clientes *Prioritarios* en la cola del *entry*

Para manejar el tipo de cliente como un parámetro del *entry* deberíamos aceptar todos los pedidos para ver a cual se debe atender. Y en realidad el empleado no debe perder tiempo en eso → en este caso es mejor tener *entrys* diferentes para cada tipo de cliente

EJERCICIO 3

El empleado tendrá 2 *entrys*, uno para cada tipo de cliente. Y el empleado usará un *select* para determinar cual de los dos *entrys* aceptar.

```
Task empleado is
    Entry Prioritario (D: IN texto; R: OUT texto);
    Entry Regular (D: IN texto; R: OUT texto);
End empleado;

Task Body empleado is
Begin
   loop
      SELECT
          accept Prioritario (D: IN texto; R: OUT texto) do
                R := resolverPedidoPrioritario(D);
          end Prioritario;
       OR
          accept Regular (D: IN texto; R: OUT texto) do
                R := resolverPedidoRegular(D);
          end Regular;
       END SELECT;
   end loop;
end empleado;
```

¿Tiene más prioridad el primer *accept* por estar como primera alternativa del *select*?

EJERCICIO 3

NO. El orden de las alternativas no afecta en la elección a ejecutar. Para seleccionar el *accept Regular* se debe estar seguro que no hay pedidos pendientes de *Prioritario* → Usar el *when* en el *accept Regular* consultando por el *count* de *Prioritario*.

```
Task Body empleado is
Begin
   loop
      SELECT
           accept Prioritario (D: IN texto; R: OUT texto) do
                R := resolverPedidoPrioritario(D);
           end Prioritario;
       OR
           when (Prioritario'count = 0) => accept Regular (D: IN texto; R: OUT texto) do
                R := resolverPedidoRegular(D);
           end Regular;
       END SELECT;
   end loop;
end empleado;
```

EJERCICIO 3

```
Procedure Banco3 is
   Task empleado is
      Entry Prioritario (D: IN texto; R: OUT texto);
      Entry Regular (D: IN texto; R: OUT texto);
   End empleado;

   Task type cliente;

   arrClientes: array (1..N) of Cliente;

   Task Body cliente is
        Resultado: texto;
   Begin
        if ("es cliente prioritario") then
             Empleado.Prioritario ("datos", Resultado);
        else
             Empleado.Regular ("datos", Resultado);
        end if;
   End cliente;

   Task Body empleado is
   Begin
      loop
         SELECT
              accept Prioritario (D: IN texto; R: OUT texto) do
                        R := resolverPedidoPrioritario(D);
              end Prioritario;
         OR
              when (Prioritario'count = 0) =>
                   accept Regular (D: IN texto; R: OUT texto) do
                        R := resolverPedidoRegular(D);
                   end Regular;
         END SELECT;
      end loop;
   end empleado;
Begin
     null;
End Banco3;
```

EJERCICIO 4

Se debe modelar la atención en un banco por medio de DOS empleados. Los clientes llegan y son atendidos de acuerdo al orden de llegada por cualquiera de los dos.

Partimos de la solución anterior del ejercicio 1. Pero *Empleado* será un tipo y declararemos dos variables: *EmpleadoA* y *EmpleadoB*.

```
Procedure Banco4 is
   Task Type empleado is
      Entry Pedido (D: IN texto; R: OUT texto);
   End empleado;
   EmpleadoA, EmpleadoB: empleado;
   Task type cliente;
   arrClientes: array (1..N) of Cliente;
   Task Body cliente is
        Resultado: texto;
   Begin
        Empleado.Pedido ("datos", Resultado);
   End cliente;
   Task Body empleado is
   Begin
      loop
         accept Pedido (D: IN texto; R: OUT texto) do
               R := resolverPedido(D);
         end Pedido;
      end loop;
   End empleado;
Begin
    null;
End Banco4;
```

El *entry call* debe ser a una tarea en particular y no a un tipo tarea ¿A cúal de los dos se lo hace?

EJERCICIO 4

El cliente no determina que empleado lo debe atender, cuando un empleado está libre debe atender al primer cliente que esté esperando. Necesitaremos una tarea ***Administradora*** que almacena los pedidos de los clientes en orden, y los empleados le piden al siguiente.

Comencemos con los ***Clientes***

```
Task Administrador …

Task Type empleado …

Task type cliente …

EmpleadoA, EmpleadoB: empleado;

arrClientes: array (1..N) of Cliente;

Task Body cliente is
      Resultado: texto;
Begin
      Administrador.Pedido (“datos”, Resultado);
End cliente;
```

En este caso el Administrador no puede resolver el pedido del cliente para devolverle el resultado.

EJERCICIO 4

Necesitamos dos comunicaciones. El cliente le hace el pedido al administrador, y luego espera a que un empleado le envíe la respuesta.

```
Task Administrador is
    entry Pedido (D: IN texto);
End Administrador;

Task type cliente is
    entry Respuesta (R: IN texto);
End cliente;

arrClientes: array (1..N) of Cliente;

Task Body cliente is
     Resultado: texto;
Begin
     Administrador.Pedido ("datos");
     Accept Respuesta (R: IN texto) do
          Resultado := R;
     end Respuesta;
End cliente;
```

Cómo sabrá a quien contestarle el empleado, debería saber su identificador (posición en el vector aaClientes).

EJERCICIO 4

Para esto el cliente debería conocer ese identificador y enviarlo como parámetro en el *Pedido* hecho al *Administrador*. Pero él no lo conoce, alguien se lo tiene que enviar por medio de un *entry call*, por ejemplo en el cuerpo principal del programa.

```
Task type cliente is
    entry Ident (Pos: IN integer);
    entry Respuesta (R: IN texto);
End cliente;

arrClientes: array (1..N) of Cliente;

Task Body cliente is
    Resultado: texto;
    id: integer;
Begin
    Accept Ident (Pos: IN integer) do
        id := Pos;
    end Ident;
    Administrador.Pedido (id, "datos");
    Accept Respuesta (R: IN texto) do
        Resultado := R;
    end Respuesta;
End cliente;
```

```
Begin
    for i in 1..N loop
        arrClientes(i).Ident(i);
    end loop;
End Banco4;
```

EJERCICIO 4

Seguimos con los empleado

Cada empleado le debe pedir al administrador el siguiente pedido (cliente) a atender, luego resuelve el pedido y le da la respuesta al cliente correspondiente.

```
Task Administrador is
    entry Pedido (IdC: IN integer; D: IN texto);
    entry Siguiente (Id: OUT integer; DC: OUT texto);
End Administrador;

Task Type Cliente is
    entry Ident (Pos: IN integer);
    entry Respuesta (R: IN texto);
End Cliente;

Task Type Empleado;

EmpleadoA, EmpleadoB: Empleado;

arrClientes: array (1..N) of Cliente;
```

```
Task Body Empleado is
      Res, Dat: texto;
      idC: integer;
Begin
   loop
      Administrador.Siguiente (idC, Dat);
      Res := resolverPedido(Dat);
      arrClientes(idC).Respuesta(Res);
   end loop;
End Empleado;
```

EJERCICIO 4

Seguimos con el Administrador

Para evitar tener que encolar en forma explicita los pedidos de los clientes podemos esperar hasta que haya un pedido de un empleado para aceptar el pedido del cliente.

```
Task Administrador is
     entry Pedido (IdC: IN integer; D: IN texto);
     entry Siguiente (Id: OUT integer; DC: OUT texto);
End Administrador;

Task Body Administrador is
Begin
   loop
        Accept Siguiente (Id: OUT integer; DC: OUT texto) do
             Accept Pedido (IdC: IN integer; D: IN texto) do
                  Id := IdC;
                  DC := D;
             End Pedido;
        End Siguiente;
   end loop;
End Empleado;
```

EJERCICIO 4

```
Procedure Banco4 is

   Task Administrador is
      entry Pedido (IdC: IN integer; D: IN texto);
      entry Siguiente (Id: OUT integer; DC: OUT texto);
   End Administrador;

   Task Type Cliente is
      entry Ident (Pos: IN integer);
      entry Respuesta (R: IN texto);
   End Cliente;

   arrClientes: array (1..N) of Cliente;

   Task Body Administrador is
   Begin
      loop
        Accept Siguiente (Id: OUT integer; DC: OUT texto) do
              Accept Pedido (IdC: IN integer; D: IN texto) do
                    Id := IdC;
                    DC := D;
              End Pedido;
        End Siguiente;
      end loop;
   End Empleado;

   Task Type Empleado;

   EmpleadoA, EmpleadoB: Empleado;

   Task Body Empleado is
         Res, Dat: texto;    idC: integer;
   Begin
      loop
          Administrador.Siguiente (idC, Dat);
          Res := resolverPedido(Dat);
          arrClientes(idC).Respuesta(Res);
      end loop;
   End Empleado;

   Task Body cliente is
         Resultado: texto;    id: integer;
   Begin
         Accept Ident (Pos: IN integer) do
              id := Pos;
         end Ident;
         Administrador.Pedido (id, "datos");
         Accept Respuesta (R: IN texto) do
              Resultado := R;
         end Respuesta;
   End cliente;
```

```
Begin
    for i in 1..N loop
        arrClientes(i).Ident(i);
    end loop;
End Banco4;
```

EJERCICIO 5

Se debe modelar un buscador para contar la cantidad de veces que aparece un número dentro de un vector distribuido entre las ***N tareas contador***. Además existe un ***administrador*** que decide el número que se desea buscar y se lo envía a los N contadores para que lo busquen en la parte del vector que poseen, y calcula la cantidad total.

```
Procedure ContadorOcurrencias is
  Task Admin;
  Task type Contador is
       entry Contar (num: in integer; res: out integer);
  End contador;

  ArrC: array (1..N) of Contador;

  Task body Admin is
       num: integer := elegirNumero;
       parcial, total: integer := 0;
  Begin
       for i in 1..N loop
            ArrC(i).Contar (num, parcial);
            total:= total + parcial;
       end loop;
  End Admin;
```

SECUENCIAL. No puede indicarle que numero buscar a un Contador hasta que el anterior no le devolvió su resultado

```
  Task body Contador is
       vec: array (1..V) of integer := InicializarVector;
       cant: integer :=0;
  Begin
       Accept Contar(num: in integer; res: out integer) do
            for i in 1..V loop
                 if (vec(i) = num) then
                      cant:=cant+1;
                 end if;
            end loop;
            res := cant;
       end contar;
  End contador;

  Begin
     null;
  End ContadorOcurrencias;
```

EJERCICIO 5

Se debe distribuir el envío del número con la recepción de los resultados.

```
Procedure ContadorOcurrencias is
   Task Admin is
        entry Resultado (res: in integer);
   End admin;
   Task type Contador is
        entry Contar (num: in integer);
   End contador;
   ArrC: array (1..N) of Contador;
   Task body Admin is
        num: integer := elegirNumero;
        total: integer := 0;
   Begin
        for i in 1..N loop
            ArrC(i).Contar (num);
        end loop;
        for i in 1..N loop
            accept Resultado (res: in integer) do
                  total:= total + res;
            end Resultado;
        end loop;
   End Admin;

   Task body Contador is
        vec: array (1..V) of integer := InicializarVector;
        valor, cant: integer :=0;
   Begin
        Accept Contar(num: in integer) do
              valor := num;
        end contar;
        for i in 1..V loop
            if (vec(i) = valor) then
                cant:=cant+1;
            end if;
        end loop;
        Admin.Resultado(cant);
   End contador;

   Begin
      null;
   End ContadorOcurrencias;
```

Posible demora si hay demora en unos de las tareas

La solución es ACEPTABLE. Pero se puede mejorar evitando las demoras mencionadas.

EJERCICIO 5

Para evitar el asignar en orden el número a buscar generando demoras si uno está demorado, se puede optar por la opción donde el Contador pide el número cuando está listo (intercambiamos quien hace el *entry call* y el *accept)*.

```
Procedure ContadorOcurrencias is
   Task Admin is
        entry Valor (num: out integer);
        entry Resultado (res: in integer);
   End admin;
   Task type Contador;
   ArrC: array (1..N) of Contador;
   Task body Contador is
      vec: array (1..V) of integer := InicializarVector;
      valor, cant: integer :=0;
   Begin
      Admin.valor(valor);
      for i in 1..V loop
          if (vec(i) = valor) then
              cant:=cant+1;
          end if;
      end loop;
      Admin.Resultado(cant);
   End contador;

   Task body Admin is
      numero: integer := elegirNumero; total: integer := 0;
   Begin
       for i in 1..2*N loop
           select
               accept Valor (num: out integer) do
                     num := numero;
               end Valor;
           or
               accept Resultado (res: in integer) do
                   total:= total + res;
               end Resultado;
           end select;
       end loop;
   End Admin;

Begin
   null;
End ContadorOcurrencias;
```