Tema 5. Concurrencia en los sistemas en tiempo real

5.1 Introducción

En el mundo físico y real es preciso controlar acciones que se están dando de forma simultánea. Una forma de enfrentarse a este cometido es preparar un conjunto de programas que se comuniquen entre si, de tal manera que cada uno controle un aspecto de la funcionalidad del sistema. De esta forma sería más fácil elaborar un conjunto de programas sencillos que uno grande y complejo. Si además, como es frecuente en los sistemas con los que trabajamos, la respuesta del sistema debe estar condicionada a eventos temporales nos encontramos con una dificultad enorme para satisfacer las necesidades del sistema mediante un único programa.

Denominamos programación concurrente a la notación y técnicas de programación que expresan el paralelismo potencial y que resuelven los problemas resultantes de la sincronización y la comunicación. (Ben-Ari 1982)

Un programa "ordinario" consiste en un conjunto de declaraciones de datos y de instrucciones, escrito en un lenguaje de programación. Las instrucciones son ejecutadas secuencialmente (una tras otra). El camino de ejecución a través del conjunto de instrucciones puede diferir de una ejecución a otra, debido a variaciones en las entradas, pero, para unos datos determinados, sólo hay un camino de ejecución posible. Este tipo de programa se conoce como programa secuencial.

Un programa concurrente está formado por un conjunto de "*programas secuenciales*", llamados **procesos o hilos**, que son ejecutados en un paralelismo abstracto.

El paralelismo es abstracto porque no es necesario utilizar un procesador físico para ejecutar cada proceso. Aunque el programa concurrente sea ejecutado en un único procesador, podemos suponer que los procesos están siendo ejecutados simultáneamente, sin preocuparnos por los detalles del paralelismo físico que puede proporcionar o no nuestra computadora.

Una propiedad fundamental de la programación concurrente es el **indeterminismo**: dado un instante de tiempo, no es conocido qué va a ocurrir en el instante siguiente. Para la implementación sobre un único procesador, no puede saberse si va a ocurrir o no una interrupción que cause un intercambio del proceso que está siendo ejecutado. En el caso de un sistema multiprocesador, las velocidades de los procesadores no están sincronizadas, por lo que no puede saberse qué procesador va a ser el primero en ejecutar su siguiente instrucción. Habitualmente es deseable que el estado final de una computación esté determinado, aunque la computación misma no lo esté. Por ejemplo, es deseable que la suma de dos subexpresiones sea la misma, independientemente de cual de ellas sea evaluada en primer lugar.

Todos los sistemas en tiempo real son inherentemente concurrentes. Los lenguajes para sistemas en tiempo real tendrán un mayor poder expresivo si proporcionan al programador las primitivas para realizar una programación concurrente.

Se asume en este tema que los estudiantes tienen ciertos conocimientos de programación concurrente, al menos en cuanto a conceptos, adquiridos en las asignaturas de sistemas operativos. Por tanto los conceptos de proceso, hilo, semáforo, colas de mensajes, monitor, problema de la exclusión mutua y sincronización condicional no serán desarrollados aquí.

5.2 Objetivos

Comprender la importancia de la programación concurrente dentro de los sistemas de tiempo real.

Crear tareas en Ada con diversos métodos.

Crear tareas anónimas

Crear tareas pertenecientes a un tipo.

Crear tares de forma dinámica

Comunicar y sincronizar tareas mediante memoria compartida en Ada.

Comunicar y sincronizar tareas mediante paso de mensajes en Ada

Resolución de problemas de exclusión mutua y sincronización condicional

Saber realizar programas concurrentes.

Apartados del tema

5.3 Generalidades

En este tema vamos a hablar de concurrencia, que no debe entenderse por paralelismo. Físicamente, los procesos pueden:

a) Multiplexar su ejecución en un único procesador.

b) Multiplexar su ejecución en un sistema multiprocesador de memoria compartida.

c) Multiplexar su ejecución en varios procesadores que no comparten memoria (sistema distribuido).

d) Ejecutarse según un modelo híbrido de los tres anteriores.

Sólo en los casos b), c) y d) se puede hablar de una ejecución verdadera en paralelo. El término concurrente indica paralelismo potencial. Los lenguajes para programación concurrente permiten al programador expresar actividades lógicamente paralelas, independientemente de la forma de ejecución posterior.

Aunque el concepto de proceso sea común para todos los lenguajes de programación concurrente, hay grandes variaciones en cuanto al modelo de concurrencia que adoptan. Estas variaciones se dan, entre otros aspectos, en:

a) Estructura.

- Estática: El número de procesos del programa es fijo, y conocido en tiempo de compilación.
- Dinámica: Los procesos pueden ser creados en cualquier momento. El número de procesos existentes sólo puede ser conocido en tiempo de ejecución.

b) Nivel de paralelismo.

- Anidado: Un proceso puede ser definido dentro de otro proceso.
- Plano: Los procesos sólo pueden ser definidos en el nivel más externo del programa.

c) Relaciones entre procesos.

Con niveles de paralelismo anidados se pueden crear jerarquías de procesos, e interrelaciones entre ellos:

- Relación padre/hijo. Un proceso (o bloque), el padre, es el responsable de la creación de otro, el hijo. El padre ha de esperar mientras el hijo está siendo creado e inicializado.
- Relación guardián/dependiente. El proceso (o bloque) guardián no puede terminar hasta que todos los procesos dependientes hayan terminado. En lenguajes que permiten estructuras dinámicas (caso del Ada), el padre y el guardián pueden no ser el mismo.

d) Granularidad.

Se puede hacer la división paralelismo de grano fino - paralelismo de grano gordo. Un programa concurrente de grano gordo contiene relativamente pocos procesos, cada uno con una labor significativa que realizar. Programas de grano fino contienen un gran número de procesos, algunos de los cuales pueden incluir una única acción.

	Escala	Paralelismo
Mayor granularidad	1	Programas independientes
	2	Subprogramas/Segmentos de programa
	3	Bucles (procesamiento vectorial)
Menor granularidad	4	Sentencias (múltiples unidades funcionales)

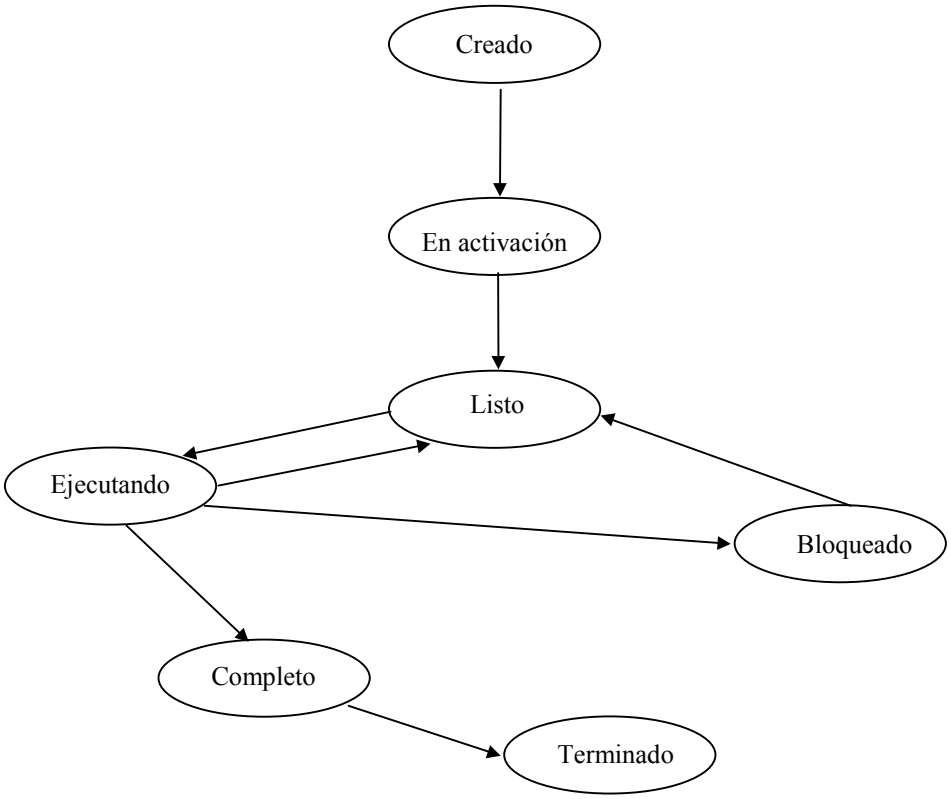
La concurrencia de grano fino requiere soporte del *hardware*. Un ejemplo lo constituye un computador con unidades de procesamiento separadas para multiplicación y suma. Una sentencia como E:=(A*B)+(C+D) puede ejecutarse en paralelo en dicho computador. Mientras una unidad realiza (A*B), la otra puede operar (C+D). Un procesador vectorial puede efectuar en paralelo una operación sobre todos los elementos de un *array*:

```
for I in 1..N loop
   A(I):=B(I)+C(I);
end loop;
```

La mayoría de los lenguajes concurrentes, Ada por ejemplo, permiten realizar paralelismo de grano gordo. Los sistemas en tiempo real habitualmente se construyen utilizando programación concurrente de grano gordo; los procesos cooperantes, además, deberán cumplir requisitos temporales.

5.3.1 Procesos, sus estados y operaciones

Durante su tiempo de existencia, un proceso se encuentra en uno de varios estados posibles: elaborado, ejecutando (*running*), listo (*ready*), bloqueado, completo y terminado. Aunque en la bibliografía se puede encontrar diferente terminología y conjunto de estados, adoptamos aquí el conjunto de estados que se utiliza en el estudio de los procesos en Ada.



- Un proceso está en activación cuando está siendo elaborada su parte declarativa.

- Un proceso está ejecutando (running) si sus sentencias están siendo ejecutadas en un procesador.
- Un proceso está listo (ready) si está esperando a ser asignado a un procesador.
- Un proceso está bloqueado si está esperando que ocurra algún evento, tal como una sincronización con otro proceso. Algunas veces se utiliza el término dormido como sinónimo.
- Un proceso está completo si ha acabado de ejecutar su secuencia de sentencias, pero no puede terminar porque está activo alguno de sus procesos dependientes.
- Un proceso está terminado si ya no está (o nunca estuvo) activo.

Los procesos llevan asociadas diversas operaciones: creación y activación, terminación, sincronización, comunicación y planificación (*scheduling*).

1. Creación y activación.

Distinguimos entre creación y activación porque en algunos sistemas y lenguajes, los procesos pueden ser declarados (creados) pero no comienzan (se activan) hasta que se alcanza un punto determinado en el programa creador. En algunos casos, la activación es implícita cuando se alcanza ese punto; en otros la activación ocurre sólo cuando se ejecuta una operación explícita.

2. Terminación.

La forma en que un proceso termina difiere entre sistemas. Un proceso puede terminar implícitamente, cuando acaba de ejecutar sus labores y no tiene procesos dependientes, o explícitamente, mediante la ejecución de una operación como kill (UNIX) o abort (Ada).

3. Planificación.

Por planificación entendemos los métodos por los que los procesos son asignados a los procesadores disponibles. En otras palabras, los medios por los que un proceso pasa del estado listo (*ready*) al estado ejecutando (*running*). La planificación de procesos es, por supuesto, una parte importante del diseño de sistemas operativos. Con la llegada de lenguajes como Ada, diseñados para soportar la concurrencia de un modo independiente del sistema sobre el que funcionen, la planificación se ha convertido en un capítulo importante tanto para los diseñadores de lenguajes como para los programadores de aplicaciones.

La planificación implica decidir sobre aspectos tales como:

- Si el programador ha de poder efectuar una asignación explícita de un proceso a un procesador.
- Cómo ha de ser la asignación de prioridades a los procesos, de modo que un proceso con prioridad más alta que el resto sea asignado a un procesador.
- Si estas prioridades han de ser estáticas o dinámicas.
- Dados varios procesos listos (ready) con la misma prioridad, qué proceso asignar a un procesador, y si el programador ha de poder controlar esta elección.
- Si implementar o no un reparto del tiempo de cómputo de un procesador (time-slicing) entre los distintos procesos, de modo que cada proceso disponga de una cantidad fija de tiempo, transcurrido el cual es interrumpido.
- Si un proceso de prioridad alta, que pase del estado bloqueado al estado listo, puede interrumpir inmediatamente a otro proceso que se está ejecutando, o si debe esperar a que éste se bloquee o termine.

Trataremos las operaciones de sincronización y comunicación entre procesos separadamente, en un apartado posterior.

5.3.2Medios de creación y manejo de procesos

Se puede diferenciar entre tres formas de soporte de procesamiento concurrente:

1. **Llamadas al sistema operativo** (*system calls*). Los proponentes de este método de conseguir concurrencia argumentan que es la forma por la que se consigue mayor eficiencia en la ejecución. Por

el contrario, los opositores entienden que los programas que utilizan servicios del sistema son más complejos, menos obvios y con costosa adaptación a otro sistema.

2. **Nuevas estructuras especiales en los lenguajes de programación, independientes del sistema operativo.** Ejemplos de lenguajes que incorporen estructuras de soporte de concurrencia son Ada, Occam, Concurrent Pascal y Concurrent C. Razones a favor de este método:

- La concurrencia se expresa en un modo claramente visible para el escritor y el lector de los programas fuente. Estos programas serán más fáciles de comprobar y mantener.
- Se mejora el poder de abstracción; la modelización del mundo físico será más clara y natural.
- Los programas serán menos dependientes de la máquina (sistema operativo y plataforma hardware), y mejorará así su portabilidad.

Los objetores a este método argumentan la menor eficiencia asociada con la independencia de la máquina. Estas razones pierden peso con el paso del tiempo, debido a las continuas mejoras de los compiladores y a la evolución del *hardware*.

3. **Bibliotecas para creación y operaciones con procesos, independientes del sistema operativo.** Compromiso entre los dos extremos dados por las aproximaciones 1 y 2. Según este método, el lenguaje no conlleva estructuras sintácticas especiales. En su lugar, un módulo o biblioteca soporta la concurrencia mediante la definición de tipos y operaciones en la forma de subprogramas, que pueden ser invocados por una aplicación. La interfaz entre el programador y dichos módulos es independiente de la máquina y del sistema. El ejemplo más conocido de esta aproximación está en la forma de concurrencia del lenguaje Modula 2.

5.4 Programación concurrente en Ada

Ada proporciona primitivas para la programación concurrente. La unidad convencional de paralelismo, el proceso, es denominada ***task*** en Ada (en adelante, utilizaremos los términos *task* o tarea).

Un programador puede introducir tareas en la parte declarativa de un bloque, subprograma, cuerpo de otra tarea, o en un *package* (paquete). Una tarea no puede estar sola. Siempre ha de estar declarada dentro de otra unidad (unidad padre), y ha de haber una tarea *master* (guardián, tutor).de la que dependa.

Al igual que los paquetes, las tareas constan de especificación y cuerpo. En la especificación se incluyen los **puntos de cita** de esa tarea mientras que en el cuerpo puede haber una parte declarativa con variables o subprogramas y las sentencias que deberá ejecutar la tarea.

Ejemplo:

```
procedure EJEMPLO1 is
   task A;
   task B;
   task body A is
      -- declaraciones locales para A
      ...
   begin
      -- sentencias de A
      ...
   end A;
   task body B is
      -- declaraciones locales para B
      ...
   begin
      -- sentencias de B
      ...
   end B;
begin
   -- sentencias de EJEMPLO1
   ...
end EJEMPLO1;
```

Cada tarea es realmente un objeto de un tipo *task*. En el ejemplo anterior, el programador simplemente declara dos tareas individuales, y el tipo de cada una es declarado implícitamente por el compilador. Se dice que estas tareas son de **tipo anónimo** ya que no hay ninguna declaración explícita de tipo (pasa lo mismo que con los arrays). En el siguiente ejemplo se muestra un ejemplo de la **definición de un tipo task**, y de la declaración de diversos objetos de ese tipo:

```
   ...
   task type T;    --declaración de un tipo tarea, especificación vacia
   A,B: T;         --declaración de objetos pertenecientes al tipo T

   type LONG is array(1..100) of T;   -- array de tareas de tipo T

   type MIXTURE is
   record
      INDEX: INTEGER;
      ACTION: T;   -- un elemento del registro es una tarea
   end record;
   L: LONG;
   M: MIXTURE;

   task body T is  -- cuerpo del tipo tarea T
      ...
   end T;
   ...
```

En Ada95 la declaración de tareas puede contener discriminantes. Estos son parámetros discretos o access que pueden ser pasados en la creación de las tareas:

```
task type TAREA(P:INTEGER; M:INTEGER:=100);
...
task body TAREA is
...
T1: TAREA(4,50);
T2: TAREA(2); -- el segundo parámetro tomará en T2 el valor por defecto
(100)
```

Otra forma de crear una tarea es mediante asignación dinámica. Es posible declarar un tipo acceso que apunte a un objeto task. Fijémonos en el siguiente fragmento de programa donde suponemos que existe un tipo *task* T. En este caso, una llamada al operador **new** crea dinámicamente un objeto del tipo *task*, y retorna un acceso al mismo. La tarea será Q.all

```
procedure EJEMPLO2 is
   task type T;
   type A is access T;  -- definición de un acceso al tipo tarea
   Q: A;                -- crear un objeto P perteneciente al tipo acceso

begin
   ...
   Q:= new T;           -- crear
   ...
end EJEMPLO2;
```

5.4.1Creación y activación de tareas

Una tarea en Ada se crea de una de estas dos formas: por declaración o por asignación dinámica.

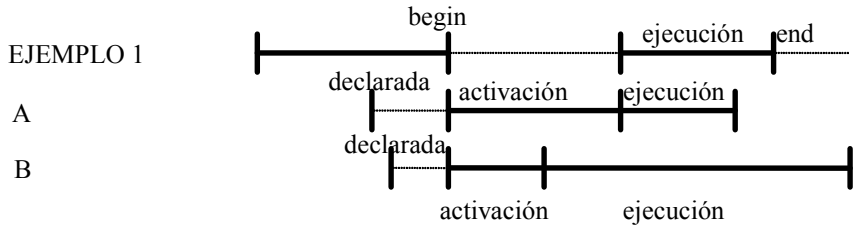
La simple declaración de una o más variables de un tipo task (o un objeto task de tipo anónimo), crea las tareas. Un objeto así declarado es activado (implícitamente) y comienza la ejecución justo antes que las sentencias del bloque en el que está declarado (padre). La ejecución de una tarea puede entenderse como un proceso que consta de dos fases. La primera conocida como activación consiste en la elaboración de las declaraciones del cuerpo de la tarea. La segunda es la ejecución de las sentencias del cuerpo de la tarea.

Durante la fase de activación no se permite continuar a la tarea progenitora

Si en un bloque se declara más de una tarea el lenguaje estándar no define el orden de activación. Durante la activación, el proceso padre estará esperando. Cuando concluye la activación, la tarea pasa a la ejecución (al estado *running* o *ready*) en paralelo con el padre.

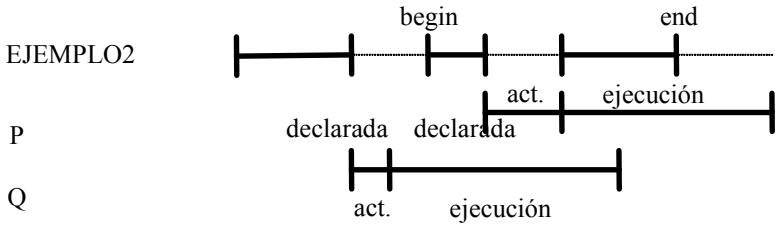
Es también posible declarar un tipo puntero que apunte a un objeto task. En este caso, una llamada al operador **new** crea dinámicamente un objeto del tipo task, y retorna un puntero al mismo. El objeto es activado justo después de su creación.

El procedure EJEMPLO1 es padre de las tareas A y B. Estas se activan y comienzan su ejecución justo antes de la ejecución de la primera sentencia del procedure




```
procedure EJEMPLO2 is
   task type T;
   type A is access T;
   P: A;
   Q: A:= new T;
   ...

begin
   ...
   P:= new T;
   ...
end EJEMPLO2;
```



La declaración de Q asocia una localización para una nueva tarea de tipo T. Esta declaración crea una tarea, que inmediatamente se activa. La tarea se designa como Q.all. Durante la ejecución del procedure, se crea y activa una nueva tarea, señalada por P. Hay ahora activos 3 procesos: las dos tareas y el procedure padre.

5.4.2Terminación de tareas

Una tarea siempre tiene una tarea guardián o *master*, de la que depende. Una tarea guardián no puede terminar hasta que todos sus dependientes hayan terminado.

Una tarea que llega al *end* de su secuencia de sentencias se dice que está completa. Una tarea termina si se cumple alguna de las siguientes condiciones:

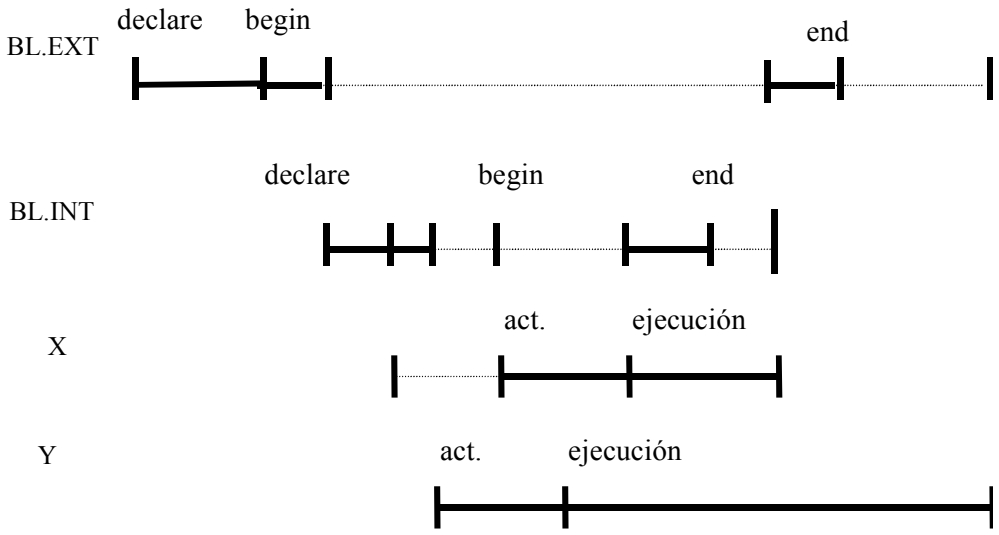
- Está completa y no tiene tareas dependientes.

- Está completa y todas sus tareas dependientes terminadas.
- Está esperando en una alternativa terminate de una sentencia select (se estudiará posteriormente), su guardián está completo, y todas las tareas dependientes del mismo guardián están, o bien terminadas, o bien esperando en una alternativa select.

Una tarea activada dinámicamente (mediante el asignador new) tiene la propiedad de que el bloque que actúa como guardián NO es el bloque padre, en el que se crea, sino el que contiene la declaración del tipo access. Para ilustrar esto consideraremos el siguiente ejemplo:

```
declare
      task type T;
      type A is access T;
      ...
   begin
      ...
      declare
         X: T;
         Y: A:= new T;
      begin
         ...
      end;
   end;
   ...
```

X e Y.all tienen el mismo padre, el bloque interno. Este bloque es, además, el guardián de X, sin embargo, Y.all es dependiente del bloque externo, y, por tanto, no afecta a la terminación del bloque interno.



Si una tarea falla al ser activada, en el padre de dicha tarea se alza la excepción TASKING_ERROR. Una excepción que se produzca durante la ejecución de las sentencias de una tarea, y que no sea tratada localmente, hace que la tarea sea abandonada, pero el error no se propaga a la unidad padre. Una tarea puede abortar (hacer que termine anormalmente) a cualquier otra tarea que esté en su alcance. Cuando una tarea es abortada, todas sus descendientes son también abortadas.

Los tipos *task* en Ada son limited private => no es posible realizar asignaciones o comparaciones entre objetos task. Si en el ejemplo anterior declaráramos Z: A; las asignaciones X:=Y.all; Y.all:=X; Z.all:=Y.all; no serían legales. Sin embargo, sí se podría utilizar el nombre del objeto access en una asignación: Y:=Z;

Otro pequeño ejemplo, comparación del procesamiento secuencial y concurrente:

```
begin          -- bloque secuencial
   PROC1;
   PROC2;
end;
```

```
-- bloque concurrente
declare
   task UNO, DOS;
task body UNO is
   begin
      PROC1;
   end UNO;
   task body DOS is
   begin
      PROC2;
   end DOS;
begin
   null;
end;
```

El bloque es padre y guardián de UNO y DOS. Al llegar al *begin* se activan las dos tareas, y el bloque no termina hasta que las dos tareas estén terminadas. Aquí tenemos otra versión concurrente

```
declare
   task UNO;
   task body UNO is
   begin
      PROC1;
   end UNO;
begin
   PROC2;
end;
```

Se pierde la simetría del algoritmo. No recomendable.

Ejemplos de creación de tareas, estudio de la activación de tareas y de la relación guardián/dependiente.

```
Creación de tareas anónimas. (solo_tareas1.adb, solo_tareas2.adb).
Creación de tipos de tareas con discriminantes (tipo_tareas.adb).
Creación de tareas de manera dinámica (access_tareas.adb).
```

5.4.3Planificación de procesos

El modelo de tareas de Ada es suficientemente rico como para ser implementado sobre cualquier arquitectura: procesadores incrustados simples, ordenadores personales, o sobre grandes sistemas distribuidos. Por consiguiente, los algoritmos precisos para distribuir las tareas sobre los diferentes procesadores, o para planificar las tareas sobre un único procesador, no están definidos por el núcleo del lenguaje. Un anexo del Ada95, denominado Real-Time Systems (ver Manual de referencia del Ada95), cubre el tema de planificación, tema que será estudiado en un próximo capítulo.

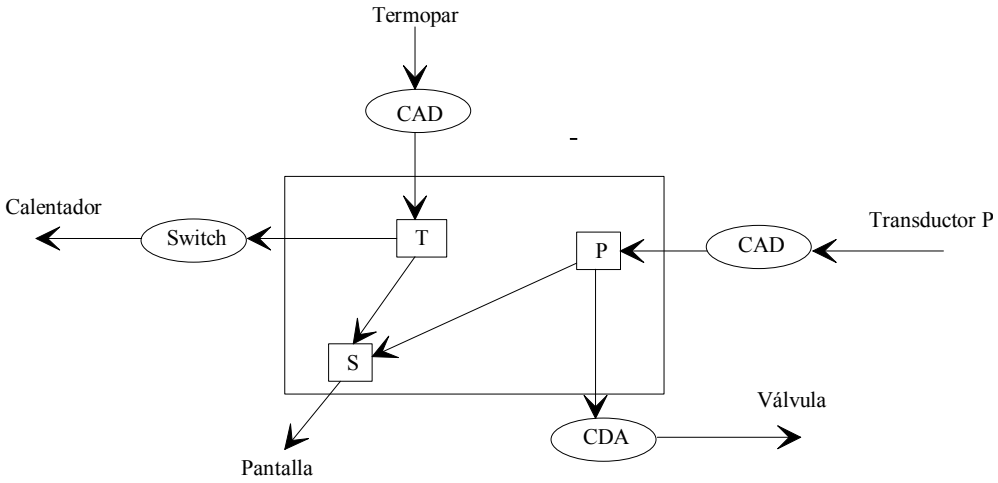
Ada soporta un esquema de prioridades: a una tarea se le puede asignar una prioridad base en tiempo de compilación, mediante una directiva del compilador. La especificación de una prioridad es una indicación dada para asistir a la asignación de recursos de procesamiento. Si dos tareas con diferentes prioridades están listas para la ejecución y van a utilizar los mismos recursos de procesamiento, no puede darse el caso de que la tarea de menor prioridad sea ejecutada y no lo sea la de mayor prioridad. Para tareas con igual prioridad, o sin prioridad explícita, el orden de planificación no está definido por el lenguaje. Las prioridades deberían ser utilizadas solamente para indicar grados relativos de urgencia, y no para sincronizar tareas. La prioridad se suele indicar mediante un número entero. Un valor mayor indica mayor urgencia.

Hasta aquí hemos visto la estructura básica del modelo de tareas en Ada. Ampliaremos este estudio posteriormente, una vez hayamos visto los modelos de sincronización y comunicación entre procesos.

5.4.4Ejemplo de un sistema empotrado

Completaremos este apartado ilustrando con un ejemplo de un sencillo sistema empotrado algunas de las ventajas de la programación concurrente. La figura adjunta muestra el sistema. La función global de este

sistema es mantener los valores de presión y temperatura de un proceso químico dentro de unos límites definidos.



Un proceso T lee los valores de temperatura señalados por un termopar, vía un convertidor AD, los compara con los deseados, y realiza los cambios oportunos sobre el calentador, mediante un *switch* digital. El proceso P realiza una función similar, pero para el control de la presión: lee el valor de la presión, y controla, mediante un convertidor DA, la apertura de una válvula. Ambos procesos deben comunicar los datos al proceso S, encargado de presentar las medidas en pantalla.

Supongamos que disponemos de los siguientes paquetes, ya implementados:

```
package DATA_TYPES is
   type TEMP_READING is new INTEGER range 10..500;
   type PRES_READING is new INTEGER range 0..750;
   type HEATER_SETTING is (ON,OFF);
   type PRES_SETTING is new INTEGER range 0..9;
end DATA_TYPES;

with DATA_TYPES; use DATA_TYPES;
package IO is
   procedure READ(TR: out TEMP_READING); -- del conversor AD
   procedure READ(PR: out PRES_READING); -- del conversor AD
   procedure WRITE(HS: in HEATER_SETTING); -- al switch
   procedure WRITE(PS: in PRES_SETTING); -- al conversor DA
   procedure WRITE(TR: in TEMP_READING); -- a la pantalla
   procedure WRITE(PR: in PRES_READING); -- a la pantalla
end IO;

with DATA_TYPES; use DATA_TYPES;
package CONTROL-PROCEDURES is
   procedure TEMP(TR: in TEMP_READING; HS: out HEATER_SETTING);
   procedure PRES(PR: in PRES_READING; PS: out PRES_SETTING);
end CONTROL_PROCEDURES;
```

Un programa secuencial simple de control podría tener la siguiente estructura:

```
with DATA_TYPES; use DATA_TYPES;
with IO; use IO;
with CONTROL_PROCEDURES; use CONTROL_PROCEDURES;
procedure CONTROLADOR is
   TR: TEMP_READING;
   HS: HEATER_SETTING;
   PR: PRES_READING;
   PS: PRES_SETTING;
begin
   loop
```

```
      READ(TR);
      TEMP(TR,HS);
      WRITE(HS);
      WRITE(TR);
      READ(PR);
      PRES(PR,PS);
      WRITE(PS);
      WRITE(PR);
   end loop;
end CONTROLADOR;
```

Inconvenientes:

- Las medidas de T y P han de tomarse a la misma 'velocidad' (1 medida por vuelta), lo que puede no estar en concordancia con los requisitos (quizás una necesite un menor período de muestreo que otra). Esto podría solucionarse con una serie de contadores y sentencias if, aunque esta solución conlleva procesamiento adicional.
- Mientras se está esperando para leer una temperatura no se puede atender a la presión, y viceversa. Si, por ejemplo, hay un problema y la llamada READ(TR) no retorna, en adición a este problema se presentará otro: perderemos también el control sobre la presión. Para evitar esto, podríamos incluir dos funciones booleanas en el package IO, TEMP_LISTA y PRES_LISTA, para indicar la disponibilidad para leer un dato:

```
loop
   if TEM_LISTA then
      READ(TR);
      ...
   end if;
   if PRES_LISTA then
      READ(PR);
      ...
   end if;
end loop;
```

Ahora, el programa perderá mucho tiempo en una espera ocupada (*'busy loop'*). Estas esperas activas son, en general, inaceptablemente deficientes, pues ocupan innecesariamente el tiempo del procesador.

La mayor crítica que se le puede hacer a este programa secuencial es que no reconoce el hecho de que las operaciones de los procesos P y T son independientes.

Si hacemos una programación concurrente:

```
with DATA_TYPES; use DATA_TYPES;
with IO; use IO;
with CONTROL_PROCEDURES; use CONTROL_PROCEDURES;
procedure CONTROLADOR is
   task T;
   task P;
   task body T is
      TR: TEMP_READING;
      HS: HEATER_SETTING;
   begin
      loop
         READ(TR);
         TEMP(TR,HS);
         WRITE(HS);
        WRITE(TR);
      end loop;
   end T;
   task body P is
      PR: PRES_READING;
      PS: PRES_SETTING;
   begin
      loop
```

```
         READ(PR);
         PRES(PR,PS);
         WRITE(PS);
         WRITE(PR);
      end loop;
   end P;
begin
   null;
end CONTROLADOR;
```

Mientras una tarea está detenida esperando por un READ, la otra puede estar ejecutándose. Si ambas están esperando, no se ejecuta ninguna espara ocupada (*busy loop*).

Aun nos quedaría por estudiar cómo envían T y P los datos a la pantalla, de forma que no colisionen (sincronización y comunicación entre tareas).

5.5 Comunicación y sincronización entre procesos

Las mayores dificultades asociadas a la programación concurrente se derivan de la interacción entre procesos. El correcto funcionamiento de un programa concurrente depende críticamente de la sincronización y comunicación entre los procesos. En su sentido más amplio, la sincronización es el cumplimiento de ciertas condiciones en el orden de ejecución de las acciones de diferentes procesos. En un sentido más limitado, se puede entender como el control de la ejecución de 2 o más procesos interactuantes de modo que a) realicen la misma operación simultáneamente, o b) no realicen la misma operación simultáneamente. Comunicación es el paso de información de un proceso a otro. Ambos conceptos están unidos, puesto que algunas formas de comunicación requieren sincronización, y la sincronización puede ser considerada como una comunicación sin contenidos.

La comunicación de datos se basa usualmente en uno de estos dos modelos:

Variables compartidas: son objetos a los que tienen acceso varios procesos

Paso de mensajes: implica el intercambio explícito de datos entre dos procesos por medio de un mensaje que se envía de un proceso a otro.

Las variables compartidas son implementables fácilmente sobre un sistema de memoria compartida (acceso de múltiples procesadores a una memoria común a través de un mismo bus). Este modelo podría ser usado incluso si el sistema incorpora otro medio de comunicación, por ejemplo, a través de una red, aunque de forma más dificultosa (habría que mantener múltiples copias físicas de una única variable lógica, y asegurarse de actualizar adecuadamente esas copias). Similarmente, el modelo de paso de mensajes puede ser implementado sobre un sistema de memoria compartida, o sobre un sistema distribuido conectado por una red.

Vamos en primer lugar a revisar brevemente las comunicaciones en sistemas de memoria compartida, con el modelo de variables compartidas, y en particular los términos exclusión mutua, sección crítica, semáforo y monitor.

5.5.1 Exclusión mutua y sincronización condicional

Aunque el modelo de variables compartidas parece una forma clara y sencilla de pasar información entre procesos, puede presentar serios problemas en las actualizaciones.

Por ejemplo, considerar 2 procesos actualizando una variable compartida, según la operación X:=X+1;

En la mayoría del *hardware*, esta operación no será atómica (indivisible), sino que se realizará mediante tres instrucciones:

1. Cargar el valor de X en algún registro
2. Incrementar el valor del registro

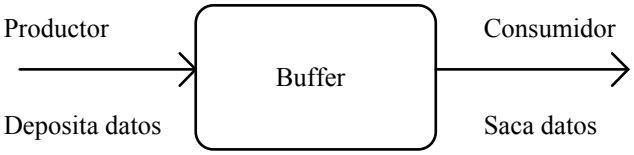
3. Almacenar el valor del registro en X

Asumimos que las instrucciones LOAD y STORE implicadas sí son atómicas.

Un proceso podría ejecutar la instrucción 1 después que el otro haya ejecutado la 1, pero antes de que ejecute la 3. Si el valor inicial de X es 5, entonces ambos cargarían el valor 5, incrementarían y almacenarían el valor 6, valor final de X, cuando dicho valor debería ser 7. Esto se puede simular en Ada como se ve en el ejemplo mal_compartido que muestra el acceso a una variable compartida sin ningún tipo de restricción. Este ejemplo aparece referenciado más adelante.

Una secuencia de sentencias que deben ser ejecutadas como si se tratara de una operación atómica es llamada **sección crítica (hay muchas otras definiciones ver [1])** . El mecanismo de sincronización requerido para evitar problemas en los accesos a una sección crítica se conoce como **exclusión mutua**.

El problema de la exclusión mutua fue descrito por Dijkstra. Es de gran interés tanto teórico como práctico, pero no es la única forma de sincronización de importancia; si dos procesos no comparten recursos, no hay necesidad de exclusión mutua. Otra sincronización habitualmente requerida es la conocida como **sincronización condicional**. Es necesaria cuando un proceso desea realizar una operación que sólo puede ser llevada a cabo si otro proceso ha realizado previamente alguna otra determinada operación. Ejemplo: sistema Productor-Consumidor.



Dos sincronizaciones condicionales:

a) El productor no puede depositar datos si el buffer está lleno.

b) El consumidor no puede sacar datos si el buffer está vacío.

Si hay varios productores o varios consumidores, hay que tener también en cuenta exclusiones mutuas.

La confección de cualquier forma de sincronización implica que los procesos en ocasiones deben ser detenidos hasta que sea apropiado que prosigan. Vamos a ver a continuación algunas de las formas de implementar la sincronización.

5.5.2Semáforos

Los semáforos representan un mecanismo simple de programación de la exclusión mutua y la sincronización condicional. Fueron originalmente diseñados por Dijkstra, y tienen las siguientes ventajas:

1. Simplifican los protocolos para sincronización.
2. Eliminan la necesidad de los bucles de espera activa.

Un semáforo es una variable S entera no negativa sobre la que están definidas dos operaciones: **wait**(S) y **signal**(S). Las acciones de estas dos operaciones han de ser atómicas.

```
wait(S):        si S>0 entonces
                  S:=S-1
                sino
                  Incrementar el número de procesos suspendidos (NS:=NS +1)
                  Suspender el proceso
                fin si
```

Cuando un proceso queda suspendido (nuevo estado a añadir al diagrama de estados de un proceso), éste sale del procesador y pasa a la cola de procesos suspendidos por ese semáforo en particular (una cola por semáforo).

signal(S): si número de procesos suspendidos > 0 entonces
Decrementar el número de suspensos (NS:= NS-1)
Mandar a ejecución uno de los procesos suspendidos
sino
S:=S+1
fin si

El lenguaje Ada no incluye semáforos, pero pueden ser fácilmente implementados a partir de sus primitivas de programación concurrente.

La sincronización condicional y la exclusión mutua pueden ser programadas fácilmente utilizando semáforos. Para la sincronización condicional:

```
S : SEMAFORO :=0;
task body C is
begin
   ...
   wait(S)          -- esto no es código Ada
   ...
end C;


task body P is
begin
   ...
   signal(S)        -- esto no es código Ada
   ...
end P;
```

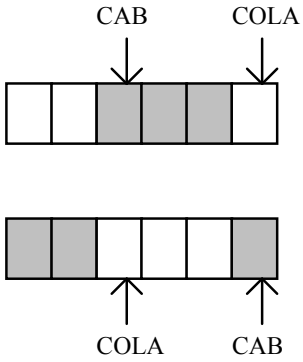
Una solución al problema de exclusión mutua para dos procesos, utilizando un semáforo:

```
S : SEMAFORO :=1;

task body P1 is
begin
   loop
      wait(S);
      Seccion_crítica_1;
      signal(S);
      Seccion_no_crítica_1;
   end loop;
end P1;

task body P2 is
begin
   loop
      wait(S);
      Seccion_crítica_2;
      signal(S);
      Seccion_no_crítica_2;
   end loop;
end P2;
```

Como wait es atómica, un proceso completará su ejecución antes de que otro empiece. El número máximo de ejecuciones concurrentes de la sección encerrada entre wait(S) y signal(S) es igual al valor inicial de S. Si S=1, sólo podrá entrar un proceso (exclusión mutua). Ejemplo de utilización de semáforos: problema del productor/consumidor con un buffer circular.




```
B: array(0..N-1) of INTEGER;
CAB,COLA: INTEGER:= 0;
ELEMENTOS: SEMAFORO:= 0;
ESPACIOS: SEMAFORO:= N;
task body PRODUCTOR is
   I: INTEGER;
begin
   loop
      PRODUCE(I);
      wait(ESPACIOS);
      B(COLA):= I;
      COLA:= (COLA+1) mod N;
      signal(ELEMENTOS);
   end loop;
end PRODUCTOR;

task body CONSUMIDOR IS
   I: INTEGER;
begin
   loop
      wait(ELEMENTOS);
      I:= B(CAB);
      CAB:= (CAB+1) mod N;
      signal(ESPACIOS);
      CONSUME(I);
   end loop;
end CONSUMIDOR;
```

El uso de primitivas de sincronización también puede dar lugar a condiciones de error. ***Deadlock*** es la condición de error más seria, y conlleva que una serie de procesos se encuentran en un estado del que ninguno puede salir. Es similar al *livelock*, pero en este caso los procesos están suspendidos, en lugar de estar en un '*busy loop*'. Como ejemplo ilustrativo, considerar dos procesos P1 y P2, que desean acceder a dos recursos no concurrentes (que no pueden ser accedidos por más de un proceso simultáneamente), recursos que están protegidos por dos semáforos S1 y S2. Si ambos procesos desean acceder a los dos recursos en el mismo orden, no se presentan problemas.

```
P1:                          P2:
  wait(S1);   --1              wait(S1);
    wait(S2); --2                wait(S2);
      ...                          ...
    signal(S2);                  signal(S2);
  signal(S1);                  signal(S1);
```

Sin embargo, si uno de los procesos quiere usar los recursos en el orden inverso:

```
P2:
  wait(S2);    --3
    wait(S1); --4
      ...
    signal(S1);
  signal(S2);
```

Si el orden de ejecución es 1324, se producirá un bloqueo *deadlock*. Otras formas de establecer la sincronización pueden ser el uso de instrucciones tsl, primitivas suspend/resume, uso de variables de condición y mutex, empleo de monitores etc. Todos estos métodos sueles aparecer en los libros de sistemas operativos y también están descritos en [1] cap 8.

5.5.3 Modelos de paso de mensajes

Según estos modelos, los procesos se comunican mediante el envío de un mensaje de un proceso a otro. Estos modos de comunicación pueden ser implementados sobre un sistema distribuido, razón por la que también se les conoce como modelos de programación distribuida.

En el diseño de las primitivas de programación concurrente mediante paso de mensajes hay que seleccionar entre las distintas alternativas que se presentan en los siguientes aspectos:

- El modelo de sincronización.
- El método para identificar un proceso.
- El flujo de datos.

1. Sincronización de procesos

Sincronización implícita en la recepción de un mensaje: un proceso no puede recibir un mensaje antes de que haya sido enviado. Si un proceso ejecuta una operación incondicional de recepción de un mensaje, y el mensaje no está disponible, el proceso queda suspendido hasta que el mensaje llegue.

La operación de envío de un mensaje puede ser clasificada como:

a) Asíncrona; el proceso envía el mensaje y continúa inmediatamente, independientemente de que el mensaje haya sido recibido o no.

b) Síncrona; el emisor continúa sólo cuando el mensaje ha sido recibido. Si el receptor no está preparado para recibir el mensaje, el emisor queda bloqueado. La comunicación sincroniza de este modo la secuencia de ejecución de los procesos. Se utiliza habitualmente el término cita (*rendezvous*) para evocar la imagen de dos personas que han de encontrarse en un punto acordado. El primero que llegue ha de esperar por el otro.

La diferencia más importante entre los dos esquemas es la necesidad de almacenar los mensajes. En la comunicación asíncrona, el emisor puede enviar muchos mensajes sin que el receptor los retire del canal de comunicación. Este canal debe estar preparado para almacenar un número potencialmente ilimitado de mensajes. Si el número máximo de mensajes en el canal es limitado, el emisor puede quedar bloqueado. En la comunicación síncrona, sólo existe un mensaje en el canal en cada instante de tiempo, por lo que no hay necesidad de buffer.

Ada y Occam presentan primitivas de comunicación síncronas. Si se necesitan buffers, deben ser explícitamente programados sobre el sistema síncrono, de modo análogo a los contestadores automáticos para las llamadas telefónicas, construidos para permitir que el emisor almacene un mensaje. El lenguaje Linda dispone de comunicación asíncrona.

2. Identificación de procesos

Una compañía telefónica puede instalar una línea que conecte directamente dos teléfonos. Esta línea dedicada será de mayor calidad y velocidad que otra que se active temporalmente mediante un sistema de conmutadores, pero tendrá como desventajas la complejidad de la instalación y el precio.

Un equipo de conmutadores puede permitir que cualquier teléfono llame a cualquier otro. Para llamar, el emisor debe conocer la identificación (nº de teléfono) del destinatario. El destinatario o receptor, en cambio, no tiene por qué conocer la identidad del llamador, a no ser que éste explícitamente decida pasarle esta información. En otras palabras, el emisor puede ser anónimo, pero no el receptor.

Otra forma de comunicación más flexible es aquella en la que el emisor envía el mensaje a una entidad intermedia, tal como un buzón (un apartado de correos). Ambas partes pueden permanecer anónimas: un mensaje sin destinatario directo puede ser leído o retirado por cualquiera.

- Occam presenta canales dedicados que conectan pares de procesos.
- Ada: un proceso llama a otro proceso por su identificador, sin divulgar su propia identidad.
- Linda: se emiten mensajes sin identificador del receptor.

Los canales dedicados son los más eficientes, puesto que los mensajes pueden ser repartidos sin la sobrecarga que conlleva la decodificación de identificadores, o direcciones. En contrapartida, cualquier modificación del sistema conllevará cambios en el código del programa.

El sistema del Ada es el más apropiado para escribir procesos servidores. Cualquier proceso que conozca el nombre del servidor puede pedir y recibir el servicio que dicho proceso proporciona.

3. Flujo de datos

Una comunicación entre dos procesos puede fluir en un solo sentido o en ambos. Una carta representa un flujo de datos en un sentido. Una respuesta requiere una nueva comunicación. Una llamada telefónica permite una comunicación en los dos sentidos; aunque sea necesario que una de las partes efectúe la llamada, una vez establecida la comunicación, el flujo de datos puede ir en los dos sentidos.

Sistemas asíncronos (Linda) usan flujo en un solo sentido, puesto que el emisor puede enviar el mensaje y continuar con el resto de sus operaciones. En sistemas síncronos se establece un canal entre los dos procesos, y puede hacerse que la comunicación sea en un sentido (Occam) o en ambos (Ada). Como siempre, toda alternativa tiene pros y contras. Enviar un mensaje por un canal de sentido único puede ser muy eficiente, puesto que el emisor no necesita estar bloqueado mientras el receptor procesa el mensaje y decide si es necesaria una respuesta. Sin embargo, si la mayoría de los mensajes necesitan respuesta, puede ser más eficiente bloquear al emisor en lugar de iniciar la labor de enviar un nuevo mensaje.

En líneas generales se considera que el empleo de paso de mensajes como medio de obtener exclusivamente sincronización es menos eficiente, consumen más tiempo, que el empleo de semáforos.

Ciertos sistemas operativos o lenguajes de programación pueden proveer de algunas de estas primitivas de sincronización mientras que otras no se suministran. Es posible construir unas a partir de otras, tal como se verá en los ejemplos con paso de mensajes mediante cita se pueden construir semáforos binarios. Con semáforos y memoria compartida pueden construirse colas de mensajes etc.

5.6 Comunicación y sincronización entre procesos en Ada

En Ada95 hay 2 formas posibles de comunicación entre tareas:

a) Indirecta, mediante objetos protegidos.

b) Directa por paso de mensajes

En Ada83 solo existía el modelo de paso de mensajes. En realidad hay una tercera forma de hacerlo que es sin protección ninguna. Un objeto declarado dentro de un bloque o subprograma que sea visible por otra tarea declarada en ese bloque podrá ser accedido por ella. Sería un caso similar al de las variables globales en C cuando se tiene una programación multihilo. Si no se toman precauciones para realizar accesos

seguros la actualización de esa variable puede ser peligrosa y tener consecuencias no previstas. Este método queda totalmente desaconsejado.

EJEMPLOS
Ejecución del programa mal_compartido que muestra el acceso a una variable compartida sin ningún tipo de restricción.

5.6.1Tipos protegidos

Un tipo protegido en Ada proporciona protección a los datos compartidos por múltiples tareas. Un objeto de un tipo protegido encierra datos a los que las tareas pueden acceder solamente mediante un grupo de operaciones protegidas. El lenguaje garantiza que estas operaciones serán ejecutadas de modo que se asegure la actualización de los datos con exclusión mutua. Los tipos protegidos pueden ser considerados como una forma avanzada de monitores. En los monitores se encapsulan los datos junto con las operaciones que deben manejarlos garantizándose que solo un proceso puede estar activo dentro del monitor. Los objetos protegidos mejoran esta solución dotando a las operaciones que se deben ejecutar de manera condicionada de barreras, mucho mas estructuradas y fáciles de manejar que las variables de condición de los monitores.

Las operaciones protegidas son 3:

a) Funciones protegidas. Permiten el acceso sólo de lectura a los datos internos. Varias tareas pueden llamar simultáneamente a una *function* protegida.

b) Procedures protegidos. Acceso exclusivo de lectura/escritura a los datos internos. Cuando una tarea está ejecutando un *procedure* protegido ninguna otra tarea podrá ejecutar otra operación protegida sobre los datos internos.

c) *Entries* protegidas. Similares a los procedures, pero con una barrera (*barrier*) asociada. Una barrera es una expresión booleana. Si cuando una tarea hace una llamada a la ***entry*** la barrera es FALSE, la tarea se coloca en la cola asociada a la *entry* a esperar que la barrera sea TRUE y que no haya otras tareas ejecutando operaciones protegidas. Las *entries* de los objetos protegidos, al igual que las *entries* de las tareas, tienen definido el atributo COUNT, que da el número actual de tareas en la cola de la *entry* especificada.

Las operaciones protegidas deberían ser cortas y rápidas. Deberían dedicarse a cosas tales como incrementar, decrementar o fijar el valor de una variable compartida. Hay un conjunto de sentencias de carácter bloqueante que no deberían usarse en una operación protegida (por ejemplo sentencias *delay*).

Especificación de un tipo protegido:

```
protected type IDENTIFICADOR[DISCRIMINANTE] is
     {declaración de operación protegida}  --una o más declaraciones
private  -- la parte privada es opcional
     {declaración de elemento protegido}]
end IDENTIFICADOR;
```

Un objeto protegido puede ser también de tipo anónimo:

```
protected IDENTIFICADOR is
     {declaración de operación protegida}
private
     {declaración de elemento protegido}]
end IDENTIFICADOR;
```

Cuerpo del tipo u objeto protegido:

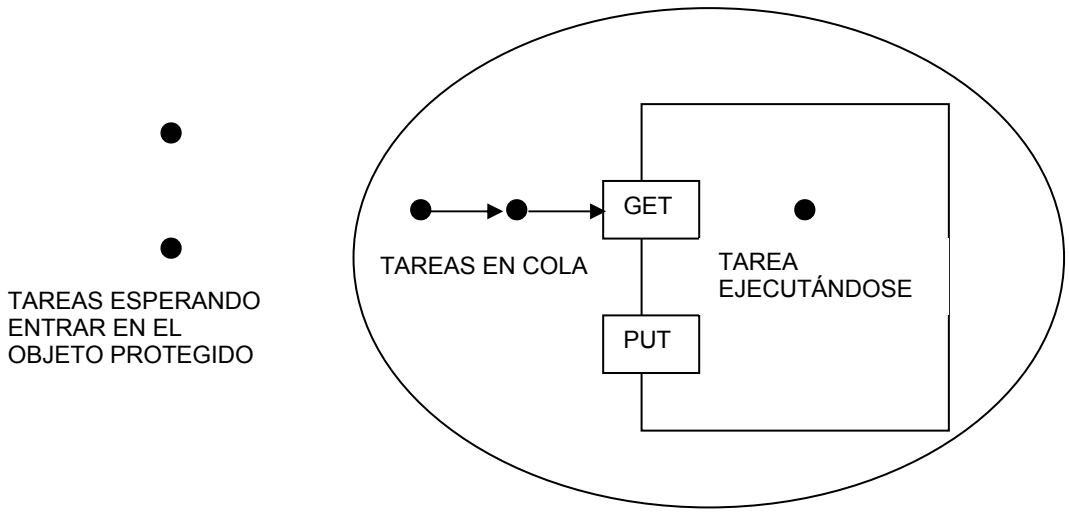
```
protected body IDENTIFICADOR is
     {cuerpo de la operación protegida}
end IDENTIFICADOR;
```

Obsérvese que en cuerpo de un objeto protegido no aparece detallado el discriminante que si puede aparecer en su especificación.

El comportamiento de los objetos protegidos en las entradas es controlado por las barreras. Cuando una tarea llama a una *entry* se evalúa la barrera que tiene asociada. Ésta es una expresión booleana por que puede resultar:

- Cierta: La tarea ejecuta el código asociado a la entrada protegida. Mientras esté activa dentro del objeto protegido impedirá que otras tareas puedan ejecutar cualquier tipo de acción dentro del objeto protegido.
- Falsa: La tarea se bloquea quedando en la cola asociada a la entrada. Permanecerá en esta situación hasta que la barrera vuelva a ser evaluada y resulte cierta. Si una tarea está dentro de un objeto protegido pero se encuentra bloqueada en una entrada protegida no impide el acceso de otras tareas al objeto protegido. Las barreras son evaluadas nuevamente cuando una tarea abandona la ejecución de una entrada o procedimiento protegido dentro de ese objeto protegido. Si la barrera resulta cierta se desbloqueará una de las tareas que pudieran estar esperando en la cola asociada a esa entrada. Puede darse el caso de que al evaluar las expresiones se hagan cierta 2 o más barreras, en tal caso el Ada no especifica a que tarea se atenderá en primer lugar.

El uso de barreras da dos niveles de protección, que en la bibliografía aparece como el modelo de la cascara de huevo (*eggshell*). Esta doble protección consiste en que cuando una tarea abandona la ejecución de una entrada o procedimiento protegido se evalúan nuevamente las barreras, si alguna de ellas se hace cierta las tareas que pudieran estar encoladas esperando por esa barrera tiene precedencia sobre otras tareas que pudieran estar fuera del objeto protegido esperando para entrar en él. Una tarea que quiera acceder a una entrada del objeto no podrá evaluar su barrera mientras haya una tarea activa dentro del objeto.



La cáscara solamente puede ser penetrada cuando el objeto está en reposo, por lo tanto hay una espera en dos niveles.

Ejemplos.

1.- Un objeto protegido que proporciona exclusión mutua en el acceso a un dato:

```
protected type ENTERO_COMPARTIDO(VALOR_INICIAL:Integer) is
     function READ return Integer;
     procedure WRITE(NUEVO_VALOR:Integer);
     procedure INCREMENT(INC:Integer);
private
     DATO:Integer:=VALOR_INICIAL;
end ENTERO_COMPARTIDO;
protected body ENTERO_COMPARTIDO(VALOR_INICIAL:Integer) is
     function READ return Integer is
     begin
          return DATO;
     end READ;

     procedure WRITE(NUEVO_VALOR:Integer) is
```

```
     begin
          DATO:=NUEVO_VALOR;
     end WRITE;

     procedure INCREMENT(INC:Integer) is
     begin
          DATO:=DATO+INC;
     end INCREMENT;
end ENTERO_COMPARTIDO;

MI_DATO:ENTERO_COMPARTIDO(42);
```

Llamadas:

```
MI_DATO.WRITE(28);
MI_DATO.INCREMENT(5);
```

```
Ejemplo de sincronización condicional. Productor/consumidor con un buffer
     circular implementado como un objeto protegido (prodcons.adb)
```

5.6.2Modelo por paso de mensajes (Rendezvous)

Esta es la forma tradicional de compartir información en Ada. El paso de mensajes entre tareas en el lenguaje Ada es síncrona (sin *buffers*). Para comunicarse dos tareas deben concertar una cita (*rendezvous*). El lugar de la cita pertenece a una de las tareas, llamada la **tarea receptora**. La otra tarea, o **tarea llamadora** (o emisora), debe conocer la identidad de la receptora, y el nombre del lugar de la cita o entrada (*entry*). Ambas tareas se ejecutarán concurrentemente y la única sincronización entre ambas será en el *rendezvous*.

Una tarea en Ada se divide en dos secciones: especificación y cuerpo. La especificación sólo puede contener declaraciones de entradas (*entry*). Sintácticamente, una ***entry*** es similar a una declaración de un procedure.

```
task BUFFER is
   entry APPEND(I: in INTEGER);
   entry TAKE(I: out INTEGER);
end BUFFER;
```

Llamada: BUFFER.APPEND(I);

No es aplicable la cláusula **use** para evitar escribir el nombre de la tarea. La llamada puede ser realizada desde cualquier punto en que se permita una llamada a un subprograma. Si una tarea trata de llamarse a sí misma caerá en un bloqueo (*deadlock*) puesto que la cita será imposible. Si APPEND fuese un *procedure* el control sería transmitido inmediatamente al cuerpo del procedure. Una llamada a una ***entry*** debe cumplir el *rendezvous* con una sentencia ***accept*** en el cuerpo de la tarea receptora, propietaria de la ***entry***.

```
task body BUFFER is
begin
   ...
   accept APPEND(I: in INTEGER) do
      ...  -- sentencias
   end APPEND;
   ...
end BUFFER;
```

La tarea receptora (BUFFER) es un proceso secuencial ordinario. Como otras instrucciones, la sentencia *accept* se ejecuta en secuencia cuando el contador de programa llega a ella. Se diferencia del resto de las sentencias en que el *accept* necesita sincronizarse con la llamada en la tarea emisora. La primera tarea que llegue al *rendezvous* (tarea emisora a la llamada, o tarea receptora al *accept*) quedará bloqueada esperando a la otra.

La semántica del *rendezvous* es como sigue:

1. La tarea llamadora, que invoca a la ***entry***, pasa sus parámetros a la tarea receptora, y queda bloqueada esperando al final del *rendezvous*.

2. La tarea receptora ejecuta las sentencias del cuerpo del accept.
3. Los parámetros out (de salida) son pasados a la tarea llamadora.
4. El *rendezvous* está completo, y ambas tareas pueden continuar.

Si sólo es necesario sincronizar ambas tareas, sin realizar otra acción, la sentencia accept puede estar vacía de la forma

```
accept NOMBRE;            --  (sin do ... end NOMBRE;)
```

Veamos un ejemplo con una solución para el problema del productor consumidor empleando citas. La información, el buffer, ahora estará declarada dentro de una tarea que funcionará como un servidor atendiendo las solicitudes de clientes. El productor y el consumidor serán sendas tareas que funcionarán como clientes solicitando un servicio al servidor. Este servicio se programa por medio de una cita donde la parte de la tarea receptora, el *accept* lo ejecuta la tarea servidora mientras que la llamada a la *entry* le corresponde a los clientes. Cada cliente llamará a una *entry* distinta ya sea una tarea consumidora o productora.

```
task body BUFFER is
   B: array(0..N-1) of INTEGER;
   CAB, COLA: INTEGER:= 0;
   CONT: INTEGER:= 0;
begin
   loop
      accept APPEND(I: in INTEGER) do      -- para productor
         B(COLA):=I;
      end APPEND;
      CONT:=CONT+1;
      COLA:=(COLA+1) mod N;
      accept TAKE(I: out INTEGER) do       -- para consumidor
         I:=B(CAB);
      end TAKE;
      CONT:=CONT-1;
      CAB:=(CAB+1) mod N;
   end loop;
end BUFFER;
```

```
task body PRODUCTOR is
   I: INTEGER;
begin
   ...
   BUFFER.APPEND(I);
   ...
```

```
task body CONSUMIDOR is
   I: INTEGER;
begin
   ...
   BUFFER.TAKE(I);
   ...
```

Varias tareas pueden llamar a la misma ***entry*** de otra tarea (varias tareas productoras pueden llamar BUFFER.APPEND concurrentemente). Estas tareas irán a una cola en el orden de llegada. Cada ejecución de la sentencia accept de la tarea receptora irá extrayendo una tarea llamadora de la cola, y efectuará un *rendezvous* con ella.

Atributo COUNT: Entry_E'COUNT, dentro del cuerpo de la tarea receptora, devuelve el número de llamadas esperando en la Entry_E.

En resumen, el *rendezvous* del Ada es una primitiva con las siguientes características:

- Comunicación síncrona, sin almacenamiento de mensajes.
- Identificación asimétrica: el emisor conoce la identidad del receptor, pero no a la inversa.
- Flujo de datos en los dos sentidos durante el rendezvous.

5.6.2.1 Select en la tarea receptora

La solución dada al buffer en el apartado anterior presenta varios problemas:

- No se tiene en cuenta que el buffer esté vacío para el accept TAKE, o que esté lleno en el accept APPEND. Tal como está escrito implementa la exclusión mutua pero no la ejecución condicional.
- Presenta una alternancia estricta entre producir y consumir, que no se corresponde con la definición de uso de un buffer.

En esta sección se presenta la sentencia ***select,*** que permite que una tarea seleccione una ***entry*** a ejecutar entre varias alternativas. Antes de dar una definición completa de la sentencia *select*, vamos a ver una mejor aproximación a la tarea buffer modificada.

```
task body BUFFER is
   B: array(0..N-1) of INTEGER;
   CAB, COLA: INTEGER:= 0;
   CONT: INTEGER:= 0;
begin
   loop
      select
         when CONT < N =>
            accept APPEND(I: in INTEGER) do
               B(COLA):=I;
            end APPEND;
            CONT:=CONT+1;
            COLA:=(COLA+1) mod N;
      or
         when CONT > 0 =>
            accept TAKE(I: out INTEGER) do
               I:=B(CAB);
            end TAKE;
            CONT:=CONT-1;
            CAB:=(CAB+1) mod N;
      end select;
   end loop;
end BUFFER;
```

La sentencia *select* permite al buffer escoger entre varias alternativas, guardadas opcionalmente por expresiones booleanas. Si la expresión es TRUE la alternativa se denomina **alternativa abierta**, y se permite el *rendezvous*. Si la expresión es FALSE la alternativa es cerrada y no se permite el *rendezvous*. Si 0<CONT<N, ambas alternativas serán abiertas. Si no hay tareas esperando en las colas de las entradas, la tarea receptora (BUFFER) esperará por la primera tarea que llame a una ***entry*** con su alternativa abierta.

Si hay tareas esperando en una sola cola, la tarea receptora establecerá un *rendezvous* con la primera tarea de la cola. Si hay tareas esperando en más de una cola, la tarea receptora escogerá la primera tarea de una de las colas, la elección de la cola dependerá del algoritmo seguido en la implementación del compilador y no está especificado por el lenguaje.

Es importante destacar que solo se atenderá a las alternativas que estén abiertas, nunca a las cerradas a pesar de que pueda haber tareas esperando en ellas

Sintácticamente, la sentencia *select* consta de un número arbitrario de sentencias *accept* guardadas opcionalmente por expresiones *booleanas*, y separadas por la palabra reservada ***or***. Si en una alternativa no hay condición de guarda se asume valor TRUE. La última alternativa del *select* puede ser una de las siguientes:

- else seguido por una secuencia de sentencias, o bien
- delay T seguido por una secuencia de sentencias, o bien
- terminate.

Semánticamente, en *select*:

1. Se evalúan las expresiones guardianas, dando como resultado un conjunto de alternativas abiertas (condiciones que dan como resultado TRUE). Si este conjunto está vacío, se produce un error (PROGRAM_ERROR), salvo si hay una alternativa else al final (un else siempre da una opción cierta).

2. Si hay tareas esperando alternativas abiertas en las colas de las entradas, comienza un *rendezvous* con la primera tarea de una de esas colas.

3. Si todas las colas están vacías, la tarea receptora queda bloqueada, hasta que una tarea emisora llame a una entrada del conjunto de alternativas abiertas. Este conjunto no ha cambiado desde que la tarea receptora llegó al *select* porque las condiciones guardianas no son reevaluadas durante la ejecución de la sentencia *select*.

4. Se completa el *rendezvous*, con la semántica vista en el apartado anterior.

5. La tarea receptora ejecuta el resto de las sentencias que siguen a la sentencia *accept*.

6. *Select* con alternativa *else*. Si no hay alternativas abiertas, o si hay alternativas abiertas, pero no hay tareas en las colas de las entradas, se ejecuta la secuencia de sentencias de la alternativa *else*.

7. *Select* con alternativa *delay*. Si no hay tareas en las colas ***entry*** de las alternativas abiertas, la tarea receptora esperará según el punto 3, durante el intervalo de tiempo dado en la cláusula delay. Si pasado este tiempo no llegó ninguna llamada, se ejecuta la secuencia de sentencias de la alternativa delay.

8. *Select* con alternativa *terminate*. Si esta sentencia select está suspendida, todas sus tareas dependientes están terminadas, la tarea guardián está completa y todas las dependientes del mismo guardián están terminadas o esperando en una alternativa *terminate*, el conjunto de tareas terminará. Una tarea está lista para terminar cuando está esperando en un *terminate* o en un *end* final.

Ejemplos programa prodcons_cita.adb: solución al problema del productor consumidor empleando citas y una tarea servidora

6.1.1.1 Select en la tarea llamadora

Al igual que la tarea receptora, la tarea llamadora puede utilizar alternativas *else* o *delay* para evitar quedar suspendida indefinidamente esperando un *rendezvous*. Una diferencia esencial es que la tarea llamadora sólo puede esperar en la cola de una ***entry***. No hay llamadas simultáneas por parte de una sola tarea a múltiples entradas.

Una tarea llamadora solamente puede tener 2 ramas de ejecución dentro de una sentencia *select.* Sin embargo puede anidar sentencias *select* una dentro de otra

Ejemplos:

```
task body T is      -- llamadora
begin
   loop
      select
         SENSOR.LEE(...);
      or
         delay 10.0;     -- segundos
         NOTIFICAR_OPERADOR;
      end select;
   end loop;
end T;
```

Si la llamada no es aceptada en el intervalo de tiempo dado en delay, se abandona el intento de comunicación, y se ejecuta el resto de las sentencias tras el delay.

```
task body T is      -- llamadora
begin
   loop
      select
         SERVIDOR_1.OP(...);
      else
         select
            SERVIDOR_2.OP(...);
         else
            null;
         end select;
      end select;
```

```
      ...
   end loop;
end T;
```

Si el SERVIDOR_N no está inmediatamente disponible, se abandona el intento, y se intenta la comunicación con el SERVIDOR_N+1.

Ejemplos cliente.adb y cliente2.adb: Desde la tarea emisora (el cliente) se ejecuta un select para citarse con una servidora, si no es atendida en un tiempo prueba con otra servidora.

Veamos ahora un ejemplo con pseudo-código con un caso de la vida diaria modelado a través de un programa.

Un empleado de banco y un cliente pueden representar tareas independientes que se sincronizan en algunos puntos temporales. Durante el día, el cliente del banco realizará actividades tales como levantarse, desayunar, ir a trabajar, y, quizás, ir al banco a hacer un depósito. El empleado también se levantará y desayunará, pero no al mismo tiempo ni en el mismo lugar que el cliente. Más tarde, irá a trabajar y esperará por clientes. Estas dos entidades sólo interactúan cuando el cliente va al banco a hacer una transacción.

Supongamos inicialmente que el banquero está completamente dedicado a la atención al cliente: una vez en el banco, su labor consistirá en esperar y atender a los clientes.

```
task body CLIENTE is
   ...
   BANQUERO.HACER_DEPOSITO(NUM_COD=>1234, CANT=>10000.0);
task body BANQUERO is
   ...
   loop
      accept HACER_DEPOSITO(NUM_COD: in INTEGER; CANT: in FLOAT) do
         BALANCE(NUM_COD):=BALANCE(NUM_COD)+CANT;
      end HACER_DEPOSITO;
      ...
   end loop;
```

Si el CLIENTE hace la llamada a la ***entry*** HACER_DEPOSITO antes de que el BANQUERO llegue al accept, esperará (tarea dormida, o bloqueada) hasta que el BANQUERO acepte el mensaje. Si el BANQUERO llega al accept antes que el CLIENTE, esperará hasta que un CLIENTE venga a hacer un depósito.

Según lo visto, el banquero no es muy eficiente. Debemos darle más trabajo, por ejemplo, atender a los ingresos que se efectúen mediante un cajero automático.

task body BANQUERO is

```
   ...
   loop
      select
         accept HACER_DEPOSITO(NUM_COD: in INTEGER; CANT: in FLOAT) do
            BALANCE(NUM_COD):=BALANCE(NUM_COD)+CANT;
         end HACER_DEPOSITO;
      or
         accept DEPOSITO_CAJERO(NUM_COD: in INTEGER; CANT: in FLOAT) do
            BALANCE(NUM_COD):=BALANCE(NUM_COD)+CANT;
         end DEPOSITO_CAJERO;
      end select;
      ...
   end loop;
   ...
```

Todavía no serán suficientes labores para el empleado. Podemos hacer que, caso de que no haya ningún tipo de depósito, por ejemplo rellene documentos.

```
select
```

```
   accept HACER_DEPOSITO(NUM_COD: in INTEGER; CANT: in FLOAT) do
      BALANCE(NUM_COD):=BALANCE(NUM_COD)+CANT;
   end HACER_DEPOSITO;
or
   accept DEPOSITO_CAJERO(NUM_COD: in INTEGER; CANT: in FLOAT) do
      BALANCE(NUM_COD):=BALANCE(NUM_COD)+CANT;
   end DEPOSITO_CAJERO;
else
   RELLENAR_DOCS;
end select;
```

Si no hay clientes ni depósitos en el cajero, el empleado inmediatamente se pondrá a rellenar documentos. Una vez en la alternativa else, no puede ser interrumpido. Aunque llegue un cliente, el banquero acabará de rellenar el documento, y atenderá al cliente en la siguiente vuelta del bucle.

Si los depósitos de los clientes sólo pueden hacerse a determinadas horas de servicio:

```
select
   when HORAS_SERVICIO =>
      accept HACER_DEPOSITO(NUM_COD: in INTEGER; CANT: in FLOAT) do
         BALANCE(NUM_COD):=BALANCE(NUM_COD)+CANT;
      end HACER_DEPOSITO;
or
   accept DEPOSITO_CAJERO(NUM_COD: in INTEGER; CANT: in FLOAT) do
      BALANCE(NUM_COD):=BALANCE(NUM_COD)+CANT;
   end DEPOSITO_CAJERO;
else
   RELLENAR_DOCS;
end select;
```

Otra posibilidad es que, en lugar de ponerse a rellenar papeles inmediatamente, el banquero espere durante un intervalo de tiempo a que se haga un depósito. Si durante ese tiempo no llega ningún mensaje, el banquero tomará un descanso (y/o hará otra labor).

```
select
   when HORAS_SERVICIO =>
      accept HACER_DEPOSITO(NUM_COD: in INTEGER; CANT: in FLOAT) do
         BALANCE(NUM_COD):=BALANCE(NUM_COD)+CANT;
      end HACER_DEPOSITO;
or
   accept DEPOSITO_CAJERO(NUM_COD: in INTEGER; CANT: in FLOAT) do
      BALANCE(NUM_COD):=BALANCE(NUM_COD)+CANT;
   end DEPOSITO_CAJERO;
or
   delay 10*MINUTOS;
   TOMAR_DESCANSO;
   RELLENAR_DOCS;
end select;
```

Para el cliente:

```
task body CLIENTE is
...
select
   BANQUERO.HACER_DEPOSITO(NUM_COD=>1234, CANT=>10000.0);
or
   delay 10*MINUTOS;
   DAR_UNA_VUELTA;
end select;
...
```

Si el banquero no está listo para el *rendezvous* en 10 minutos, el cliente se retira de la cola de la entrada HACER_DEPOSITO.

Si el cliente es muy impaciente:

```
select
   BANQUERO.HACER_DEPOSITO(NUM_COD=>1234, CANT=>10000.0);
else
   DAR_UNA_VUELTA;
end select;
```

5.6.3Atributos predefinidos para objetos y tipos task.

Las tareas, al igual que otros objetos del Ada como pueden ser los enteros, tipos enumerados, arrays etc. tienen sus propios atributos. Alguos de ellos aparecen reflejados aquí. Existe un paquete Package Task_Attributes que usado con Ada.Task_Identification permite manejar la información concerniente a las tareas (estado, direcciones, identificación et). Algunos de los atributos de las tareas son:

- T'ADDRESS Dirección inicial del código de la *task*. El valor de este atributo es de tipo ADDRESS, definido en el package SYSTEM.
- T'CALLABLE FALSE si la ejecución de T está completa o terminada.
- T'SIZE Número mínimo de bits necesario para alojar objetos de tipo T, o número de bits asociados a la tarea T.
- T'STORAGE_SIZE Espacio reservado (bytes de stack + posiblemente espacio adicional, fijado por la implementación) para cada activación de la tarea T, o de las tareas de tipo T.
- T'TERMINATED TRUE si T está terminada.

5.6.4La sentencia abort

Se puede hacer que una tarea termine (aborte) explícitamente utilizando la sentencia abort

abort t1, t2, ..., t2;

Esta sentencia hace que todas las tareas especificadas que no han terminado previamente pasen a ser ‘anormales’. Una tarea que depende de otra tarea anormal también se vuelve anormal. Una vez que todas las tareas indicadas están marcadas como anormales, la sentencia abort está completa, y la tarea que ejecutó el abort puede continuar. Si una tarea que está llamando a una entrada se vuelve anormal durante el *rendezvous* puede terminar dicho *rendezvous* antes de haber acabado. La tarea llamada no se ve afectada. Si se vuelve anormal durante el *rendezvous* la tarea receptora, se alza la excepción TASKING_ERROR en el punto de llamada de la tarea emisora. El mismo error se alza en todas las tareas que están esperando establecer un *rendezvous* con una tarea anormal.

El atributo CALLABLE dará el valor FALSE para una tarea anormal, de modo que ninguna tarea podrá comunicarse con ella.

Idealmente, una tarea anormal termina inmediatamente si está detenida esperando en una llamada a una entrada, en una sentencia accept, select o delay, o bien termina en cuanto alcance uno de esos puntos. Sin embargo, algunas implementaciones no son capaces de facilitar esta terminación inmediata. Lo que el Ada Reference Manual especifica es que la tarea anormal termine sin interaccionar con otras tareas.

La sentencia abort sólo debe ser utilizada en casos de ‘emergencia’.

5.6.5Excepciones

Las excepciones se alzan y se manejan en las tareas de modo similar a como se hace en los subprogramas, con una diferencia importante: las excepciones pueden ser propagadas fuera de un subprograma, pero no fuera de una tarea (si se estaba ejecutando ya). Puntos donde puede alzarse una excepción en una tarea:

1. En la declaración.
2. En la activación.

3. Durante la ejecución.
4. Durante la comunicación entre tareas, esperando o ejecutando un *rendezvous*.

Cada uno de estas alternativas tiene sus particularidades. Cuando aparece una excepción puede suceder:

1. La elaboración de la declaración es abandonada, y la excepción se alza de nuevo inmediatamente después de la declaración de la tarea en la parte declarativa que la encierra. De esta manera si se propagaría a su proceso guardian.
2. Se alza la excepción TASKING_ERROR en el punto de activación de la tarea: en las tareas estáticas tras el begin del padre; en las dinámicas en la sentencia new del padre. La tarea se puede propagra en el padre.
3. Igual que en la ejecución de los subprogramas, con la diferencia de que si no hay un manejador adecuado, la tarea queda en estado completa, y no se propaga el error.
4. Casos posibles:
    a) La tarea llamada (receptora) está completa antes de aceptar una llamada a una ***entry***. La excepción TASKING_ERROR se alza en el punto de llamada de la tarea emisora.
    b) La tarea llamada se vuelve anormal durante el *rendezvous*. La excepción TASKING_ERROR se alza como en 4.1.
    c) Se alza una excepción durante la ejecución del accept, y no existe un manejador local (un bloque dentro del accept, con el manejador adecuado). Entonces, la excepción se alza en las dos tareas: en la que está ejecutando la sentencia accept, (la receptora) y en el punto de llamada de la tarea emisora.

Una terminación anormal de la tarea que está llamando a una ***entry*** no alza una excepción en la tarea receptora: si el *rendezvous* no ha empezado se cancela, sino el *rendezvous* termina normalmente sin afectar a la tarea receptora.

Ejemplo: El siguiente programa es ilustrativo de la propagación de excepciones en Ada durante la fase de activación programaTask_error.adb

Sumario

5.7 Sumario

Los sistemas en tiempo real de una cierta complejidad son inherentemente concurrentes, es decir, hay varias actividades que transcurren de forma simultánea y a las que hay que controlar y responder adecuadamente. En este capítulo se han sentado las bases de la multiprogramación empleando el lenguaje Ada. Para ello se ha realizado un breve repaso a conceptos ya conocidos de la programación concurrente y de los problemas que conlleva. En primer lugar aparecen los estados que en los que un proceso puede estar: creado, inicializado, en ejecución, en espera, terminado.

También se han analizado diferentes variaciones en la organización de los procesos atendiendo a su estructura, nivel, modo de inicializarse, granularidad forma de terminación etc.

Seguidamente se ha mostrado el modelo de concurrencia del Ada, basado en tareas que pueden ser creadas de manera estática o dinámica. Es importante destacar que las tareas pueden ser de tipo anónimo o pertenecer a un tipo creado previamente. En este modelo se detallan las formas de inicializarse las tareas y de su terminación. También se explica, incluyendo ejemplos, el funcionamiento de la relación guardián dependiente.

Un a vez establecido cómo crear tareas se impone intercomunicarlas. Para ello se dispone de dos alternativas: empleo de variables compartidas y paso de mensajes.

Cuando se opta por la compartición de variables la herramienta que proporciona el Ada es el uso de objetos protegidos. En estos objetos se encapsulan las variables que se van a compartir junto con las únicas operaciones que pueden acceder a ellas. La ejecución de estas operaciones está garantizada que se hará de manera exclusiva resolviendo así el problema de exclusión mutua. Mediante el uso de barreras en las entradas de las operaciones protegidas del objeto es posible establecer también la ejecución condicional con lo que los objetos protegidos sirven para resolver estos problemas de sincronización.

Otra forma de comunicación es el paso de mensajes, que en el Ada toma forma de cita. En este modelo tenemos una forma de comunicación síncrona, no buffereada, con formato de información totalmente libre y en ambos sentidos. El direccionamiento de los mensajes es directo indicando quien es el receptor y en que punto la tarea emisora quiere citarse con él.

Para dotar de mayor flexibilidad al mecanismo de cita se introduce la sentencia select, ejecutable desde la tarea receptora y también la emisora. Se detallan las distintas alternativas que pueden estar presentes (else, delay, terminate) y se destaca el diferente funcionamiento que presenta ya se ejecute desde la tarea emisora o la receptora.

5.8 BIBLIOGRAFIA

[1] Burns,A., Wellings,A.; ‘Sistemas de Tiempo Real y lenguajes de Programación ·3ª edición Addison-Wesley (2001); caps 7,8,9

[2] Burns,A., Wellings,A.; ‘Concurrency in Ada’; ed. Cambridge University Press (1995); caps 1-11

[3] Barnes,J.G.P.; 'Programming in Ada95'; ed. Addison-Wesley (1996); cap 18

[4] Booch,G.; 'Software Engineering with Ada';ed. Addison-Wesley (1994); caps 16,17

[5] Ben-Ari,M.; 'Principles of Concurrent and Distributed Programming'; ed. Prentice-Hall (1990); caps 1..5,7,8,A,B

[6] Gehani,N.; 'Ada Concurrent Programming';ed. Prentice-Hall (1991)