# Tema 6. Programación en Tiempo Real

## 6.1 Introducción

Un lenguaje para programar sistemas empotrados ha de proporcionar facilidades para realizar control en tiempo real. El término 'tiempo real' se utiliza habitualmente como sinónimo de este tipo de sistemas. Dada la importancia del tiempo en la mayoría de los sistemas empotrados puede parecer extraño el haber pospuesto este tema hasta ahora. La razón es que las facilidades para la programación de sistemas en tiempo real se cimientan generalmente sobre la concurrencia que el lenguaje proporciona, razón por la que hemos tratado la programación concurrente en primer lugar.

Una de las principales características de los sistemas empotrados es el requisito de que interactúen con dispositivos I/O especiales. La programación de los controladores (*drivers)* para estos dispositivos requiere:

- La capacidad para enviar y recibir datos e información de control a/de estos dispositivos.
- La habilidad para manejar interrupciones.

Lenguajes como Ada, Modula y Occam proporcionan mecanismos de alto nivel para programar estas funciones de bajo nivel. Con ello se consigue que las rutinas de manejo de los dispositivos y de las interrupciones sean más fáciles de leer, escribir y mantener.

La introducción de la noción del tiempo en un lenguaje de programación puede ser descrita en términos de 4 requisitos:

a) Acceso a un reloj, de modo que se pueda medir el paso del tiempo.

b) Retraso de un proceso, de forma que pueda ser suspendido durante un periodo de tiempo dado.

c) Programación de límites temporales (***timeouts***), de forma que la no ocurrencia de un evento dentro de un periodo de tiempo especificado pueda ser reconocida y tratada.

d) Asignación de prioridades a los procesos, de forma que se pueda influir en el orden de ejecución de los mismos.

Cabe añadir que cuando se trabaja con sistemas empotrados, de naturaleza reactiva, es conveniente que el lenguaje nos facilite esquemas para la ejecución de acciones asíncronas ligadas a eventos, y también para el abandono de actividades en caso de exceder un tiempo determinado.

También hay que considerar que en un entorno donde las tareas comparten y compiten por los recursos será preciso tener mecanismos que nos faciliten la asignación segura de los recursos e incluso que nos permita establecer la organización de las colas de procesos pendientes.

No hay que olvidar que en los sistemas en tiempo real se establecen plazos que deben ser cumplidos. Los sistemas en tiempo real, dada su naturaleza concurrente, son indeterministas, por lo que cuando hay varias tareas es difícil saber si todas ellas cumplen con los plazos. Veremos diversos algoritmos de planificación y de análisis de los tiempos de respuesta.

Ada95 presenta dos anexos: *Systems Programming* y *Real-Time Programming*, que contienen especificaciones y facilidades para la programación de sistemas empotrados y de sistemas en tiempo real en general.

# Objetivos

## Objetivos

Conocer los principales aspectos relacionados con

a) Capacidades para gestionar el tiempo:

- Medición del paso del tiempo.
- Programación de retardos, de actividades periódicas y de límites temporales.
- Especificación de requisitos temporales.

b) Planificación de procesos:

- Modelos de procesos; algoritmos de planificación.
- Análisis de planificabilidad.
- Planificación de tareas en Ada.

c) Control de recursos.

d) Programación de bajo nivel:

- Controladores; especificaciones de representación.
- Interfaz con otros lenguajes.
- Interrupciones.

## 6.2 Acceso a un reloj

Si un programa ha de cumplir determinados requisitos temporales, ha de tener algún medio de medir el paso del tiempo. Esto puede hacerse de dos modos diferentes:

a) Incluyendo una primitiva reloj en el lenguaje.

b) Programando un *driver* para el reloj del sistema asociado al procesador (reloj interno), o para un reloj externo, o radio receptor que utilice una señal de tiempo internacional (UTC, *Coordinated Universal Time*).

### 6.2.1 El paquete CALENDAR del Ada

El acceso a un reloj en Ada lo proporciona el package predefinido CALENDAR. Este paquete implementa el tipo de datos TIME. Proporciona una función CLOCK para leer el tiempo, y varios subprogramas para hacer operaciones, y conversiones entre TIME y otras unidades más comprensibles, entre ellos las funciones YEAR, MONTH, DAY y SECONDS. Las tres primeras funciones retornan un dato de un subtipo INTEGER. SECONDS devuelve un valor de un subtipo del tipo DURATION. Básicamente, un valor de tipo DURATION debe ser interpretado como un valor en segundos. DURATION es un tipo predefinido real de punto fijo. El rango debe ser, al menos, 0.0..86400.0, que es el número de segundos de un día. El rango y la precisión (nº de dígitos) son dependientes de la implementación. Se garantiza una precisión mínima de 20 ms. Si se necesita un reloj de gran precisión debe programarse un *driver* sobre el reloj del sistema.

```
package CALENDAR is
   type Time is private;
   subtype Year_Number  is Integer range 1901 .. 2099;
   subtype Month_Number is Integer range 1 .. 12;
   subtype Day_Number   is Integer range 1 .. 31;
   subtype Day_Duration is Duration range 0.0 .. 86_400.0;

   function Clock return Time;
   function Year    (Date : Time) return Year_Number;
   function Month   (Date : Time) return Month_Number;
   function Day     (Date : Time) return Day_Number;
   function Seconds (Date : Time) return Day_Duration;
   procedure Split
     (Date    : Time;
      Year    : out Year_Number;
      Month   : out Month_Number;
      Day     : out Day_Number;
      Seconds : out Day_Duration);
```

```
   function Time_Of
     (Year    : Year_Number;
      Month   : Month_Number;
      Day     : Day_Number;
      Seconds : Day_Duration := 0.0)
      return    Time;

   function "+" (Left : Time;     Right : Duration) return Time;
   function "+" (Left : Duration; Right : Time)     return Time;
   function "-" (Left : Time;     Right : Duration) return Time;
   function "-" (Left : Time;     Right : Time)     return Duration;

   function "<"  (Left, Right : Time) return Boolean;
   function "<=" (Left, Right : Time) return Boolean;
   function ">"  (Left, Right : Time) return Boolean;
   function ">=" (Left, Right : Time) return Boolean;

   Time_Error : exception;

private
   ...
end CALENDAR;
```

### 6.2.2 El paquete Ada.Real-Time del Ada95

En el anexo Real-Time Programming del Ada95 se incluye un segundo paquete de medición del tiempo. Ambos paquetes manejan el mismo reloj hardware. Se utilizará el que se adapte mejor a las necesidades de la aplicación.

```
...
package Ada.Real_Time is

   type Time is private;
   Time_First : constant Time;
   Time_Last  : constant Time;
   Time_Unit  : constant := 10#1.0#E-9;

   type Time_Span is private;
   Time_Span_First : constant Time_Span;
   Time_Span_Last  :  constant Time_Span;
   Time_Span_Zero  :  constant Time_Span;
   Time_Span_Unit  :  constant Time_Span;

   Tick : constant Time_Span;
   function Clock return Time;

   function "+" (Left: Time; Right: Time_Span) return Time;
   function "+" (Left: Time_Span; Right: Time) return Time;
   function "-" (Left: Time; Right: Time_Span) return Time;
   function "-" (Left: Time; Right: Time) return Time_Span;

   function "<"  (Left, Right : Time) return Boolean;
   function "<=" (Left, Right : Time) return Boolean;
   function ">"  (Left, Right : Time) return Boolean;
   function ">=" (Left, Right : Time) return Boolean;

   function "+" (Left, Right: Time_Span) return Time_Span;
   function "-" (Left, Right :Time_Span) return Time_Span;
   function "-" (Right :Time_Span) return Time_Span;
```

```
      function "*" (Left: Time_Span; Right: Integer) return  Time_Span;
      function "*"  (Left : Integer;   Right : Time_Span) return Time_Span;
      function "/"  (Left, Right : Time_Span)             return Integer;
      function "/"  (Left : Time_Span; Right : Integer)   return Time_Span;
      function "abs" (Right : Time_Span) return Time_Span;

      function "<"  (Left, Right : Time_Span) return Boolean;
      function "<=" (Left, Right : Time_Span) return Boolean;
      function ">"  (Left, Right : Time_Span) return Boolean;
      function ">=" (Left, Right : Time_Span) return Boolean;
      function To_Duration  (TS : Time_Span) return Duration;
      function To_Time_Span (D : Duration)   return Time_Span;

      function Nanoseconds  (NS : integer) return Time_Span;
      function Microseconds (US : integer) return Time_Span;
      function Milliseconds (MS : integer) return Time_Span;

      type Seconds_Count is new integer range -integer'Last .. integer'Last;
      procedure Split (T : Time; SC : out Seconds_Count; TS : out
Time_Span);
      function Time_Of (SC : Seconds_Count; TS : Time_Span) return Time;

   private
      ...
   end Ada.Real_Time;
```

## 6.3 Retraso de un proceso

Los procesos deben ser capaces de bloquearse a sí mismos durante un periodo de tiempo. Esto permite que el proceso espere, en lugar de hacer un *busy wait* con llamadas continuas al reloj:

```
NOW:=CLOCK;
loop          -- busy loop
   exit when (CLOCK-NOW)>10.0;
end loop;
```

Para eliminar estas esperas activas, Ada introduce la sentencia delay:

**delay T**;

T, de tipo DURATION, es el tiempo mínimo, a partir del actual, que el proceso va a estar bloqueado.

En algunos casos, el tiempo que el proceso tardará en volver a ejecutar será considerablemente más largo. Por ejemplo, si una tarea de mayor prioridad está lista para ejecución cuando expire el tiempo T.

Analizar la diferencia entre estos dos bloques:

```
declare
   INTERVALO: constant DURATION:=7.0;
begin
   loop
      ACCION;
      delay INTERVALO;
   end loop;
end;
```

```
declare
   use CALENDAR;
   INTERVALO: constant DURATION:=7.0;
```

```
      SIG: TIME;
   begin
      SIG:=CLOCK+INTERVALO;
      loop
         ACCION;
         delay SIG-CLOCK;
         SIG:=SIG+INTERVALO;
      end loop;
   end;
```

Suponer que es el único proceso listo para ejecución, y que la ejecución de ACCION en cada vuelta requiere 1.0, 1.5, 2.0 segundos.

En Ada95 podemos programar un 'retraso absoluto', es decir, un bloqueo mínimo hasta que se cumpla un tiempo:

**delay until T**; -- T de tipo time

La instrucción **delay until T** se puede aproximar mediante **delay (T-clock)**, pero el efecto puede no ser el mismo. Para que fueran equivalentes, delay (t-clock) debería ejecutarse de forma atómica.

### 6.3.1 Programación de actividades periódicas

```
task body TAREA is
   INTERVALO: constant DURATION:=5.0;
begin
   loop
      ACCION;
      delay INTERVALO;
   end loop;
end TAREA;
```

Este código nos asegura que ACCION no va a ser llamada durante intervalos de 5 segundos. Dos llamadas consecutivas pueden estar separadas por cualquier valor del tiempo > 5 segundos, dependiendo de

- La duración de ACCION. Si la ejecución de ACCION dura 2 segundos, la siguiente llamada se efectuará, como mínimo, tras 7 segundos + un pequeño *overhead* por la sentencia loop (una instrucción *jump*)
- Si hay tareas listas para ejecución con mayor prioridad que TAREA, compitiendo por el mismo procesador, TAREA habrá de esperar todo el tiempo necesario.

Si ACCION debe ser llamada periódicamente cada 5 segundos (y suponiendo que TAREA no compite con otras tareas para utilizar el procesador) podemos utilizar la sentencia delay until:

```
task body TAREA is
   INTERVALO: constant Duration :=5.0;
   SIG: Time;
begin
   SIG:=CLOCK+INTERVALO;
   loop
      ACCION;
      delay until SIG;       -- en Ada83, utilizar delay(SIG-clock);
      SIG:=SIG+INTERVALO;
   end loop;
end TAREA;
```

## 6.4 Programación de límites temporales (*timeouts*)

Quizás, la condición primaria que un RTS debe cumplir, es la de reconocer la no-ocurrencia de algún evento externo, y de actuar en consecuencia. Por ejemplo, supongamos que un sensor de temperatura debe proporcionar una

medida cada segundo. Si en un intervalo de 10 segundos no da ningún valor consideraremos que se produjo un fallo. En general, un *timeout* es una restricción sobre el tiempo que un proceso debe esperar por una comunicación.

En Ada nos serviremos de la sentencia select para programar *timeouts*.

**-Aceptación temporizada.** En la tarea receptora:

```
task CONTROLADOR is
   -- es llamada por la tarea DRIVER_SENSOR, que le
   -- pasa un valor de la temperatura.
   entry LLAMADA(T: TEMPERATURA);
end CONTROLADOR;


task body CONTROLADOR is
...  -- declaraciones
begin
   loop
      select
         accept LLAMADA(T:TEMPERATURA) do
            NUEVA_TEMP:=T;
         end LLAMADA;
         ...
      or
         delay 10.0;
         ...  -- acciones si se propasa el tiempo límite
      end select;
      ...
   end loop;
end CONTROLADOR;
```

**Llamada temporizada.** En la tarea llamadora:

Como el *driver* del sensor está continuamente leyendo temperaturas, no tendría sentido que el driver esté esperando mucho tiempo para pasar un valor de T al controlador, cuando podría pasarle un valor más nuevo, leído posteriormente.

```
loop
   LEE_T;  .. nuevo valor de T del sensor
   select
      CONTROLADOR.LLAMADA(T);
   or
      delay 0.5;
      null;
   end select;
end loop;
```

Las llamadas temporizadas se pueden utilizar también para limitar el tiempo de espera de una **entry protegida**:

```
select
   P.E(...);  -- P es un objeto protegido
or
   delay 0.5;
end select;
```

Ejemplos cliente.adb y cliente2.adb propuesto en el tema anterior: 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.

**-Acciones temporizadas.**

Se puede utilizar una transferencia asíncrona de control (ATC, mecanismo que permite que una tarea pueda responder rápidamente a determinados sucesos) para limitar el tiempo de ejecución de una acción.

```
select
    -- llamada a una entry, o delay
    -- sentencias (opcionales) a ser ejecutadas
    -- una vez finalizada la anterior
then abort
   -- acciones, que serán abortadas si la parte tras select
   -- finaliza antes que ellas
end select;

select
   delay 0.1;
then abort
   acción;    --Pasados 100 ms, si la acción no ha terminado, es abortada
end select;
```

**Ejemplo**. Una tarea debe finalizar antes de un tiempo dado. Si le sobra tiempo, se puede utilizar un algoritmo opcional que mejora el resultado.

```
begin
   --parte obligatoria de cálculo
   --...
   select
      delay until Tiempo_Limite;
   then abort
      loop
         --acciones de refinamiento del resultado
      end loop;
   end select;
end;
```

Si la acción tras abort es una llamada a un objeto protegido, el efecto del abort se postpone hasta que la tarea abandone el objeto protegido.

**Ejemplos atc1.adb, atc2.adb, atc3.adb, atc4.adb Son ejemplos extraídos de [1] para mostrar ditintas posibilidades en cuanto a la ejecución de tareas según como se vayan produciente las llamadas en una transferencia asíncrona de control**

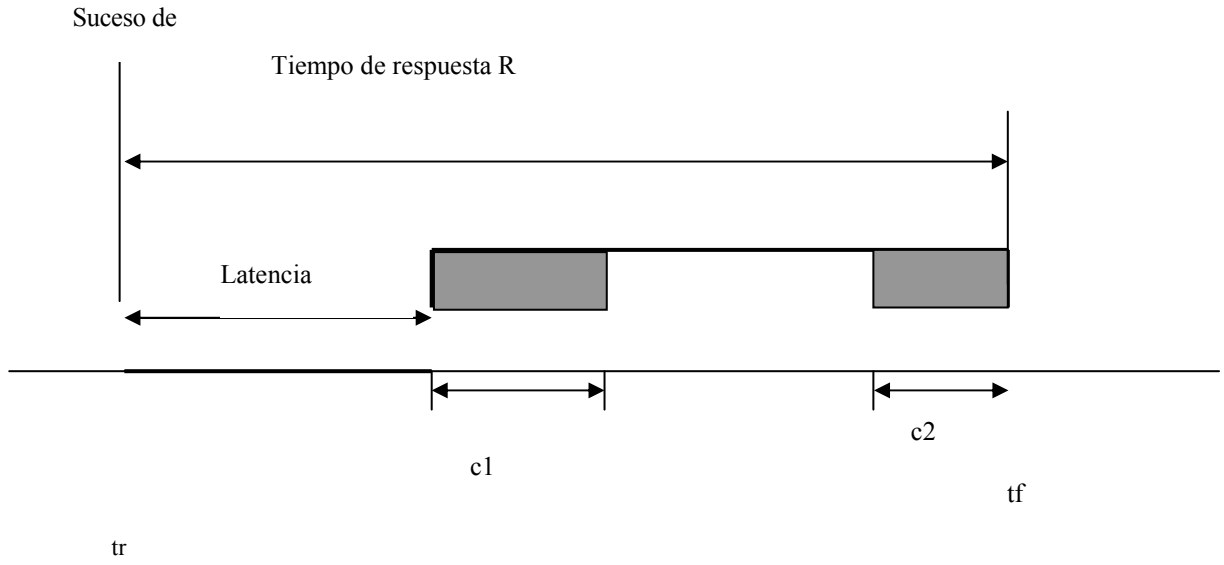
# 6.5 Especificación de requisitos temporales

Para muchos sistemas en tiempo real, el software, además de tener una lógica correcta, debe cumplir unos requisitos temporales impuestos por el sistema físico con el que se relaciona.

Es muy común que se especifique, diseñe y construya un sistema correcto lógicamente, y que luego se testee para ver si cumple los requisitos temporales. Si no los cumple, han de realizarse modificaciones que darán lugar a un software más difícil de entender y de mantener. Es por tanto necesario un tratamiento sistemático del tiempo. Un modo de enfocar este tema consiste en analizar las propiedades temporales mediante estudios de planificabilidad: hay que analizar si, con los recursos *hard* y *soft* disponibles, y fijados los tiempos límite de cada tarea, es posible que el sistema cumpla los requisitos temporales en cualquier tiempo futuro.

## Atributos temporales

Los atributos temporales de una secuencia de instrucciones (o tarea) a tener en cuenta a la hora de estudiar su planificabilidad son (ver gráfico):




**C:** Tiempo de cómputo = c1 + c2

**D:** Plazo de respuesta (R<D)

**L:** Tiempo límite (tf<L)

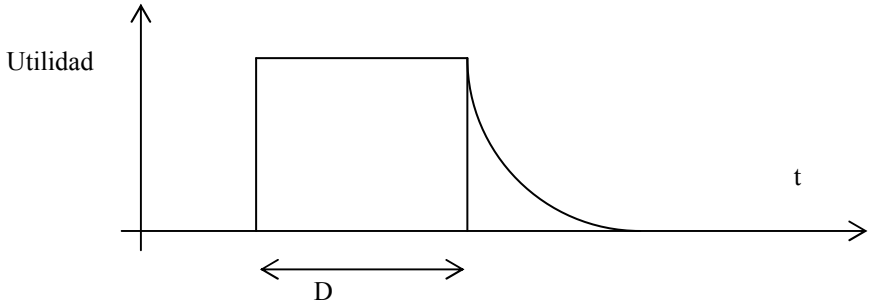
**Jmin:** Latencia mínima

**Jmax:** Latencia máxima

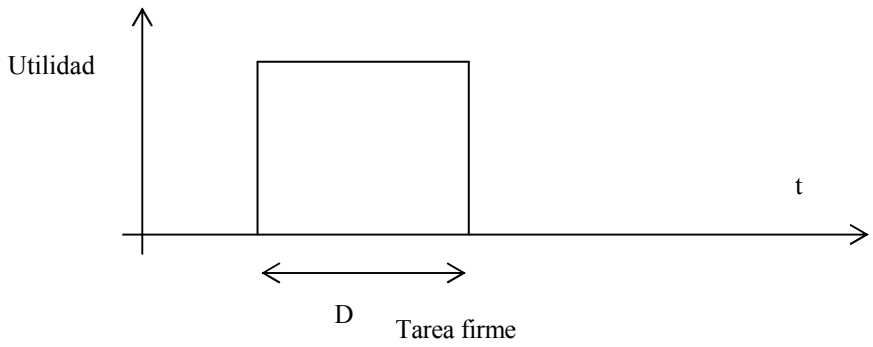
La activación que origina la ejecución de la secuencia de sentencias puede ser: periódica, aperiódica o esporádica.

Típicamente, la activación periódica corresponde a muestreos o bucles de control, con tiempos límite definidos. Las activaciones aperiódicas o esporádicas se deben a la aparición de eventos asíncronos externos, a los que las tareas han de responder dentro de un plazo dado. Las tareas esporádicas son casos particulares de las aperiódicas, en los que podemos definir un periodo mínimo entre dos eventos (sabemos que los eventos ocurrirán con un espaciado temporal >= que el periodo definido). Esta 'periodicidad' permitirá estudiar sus planificaciones.

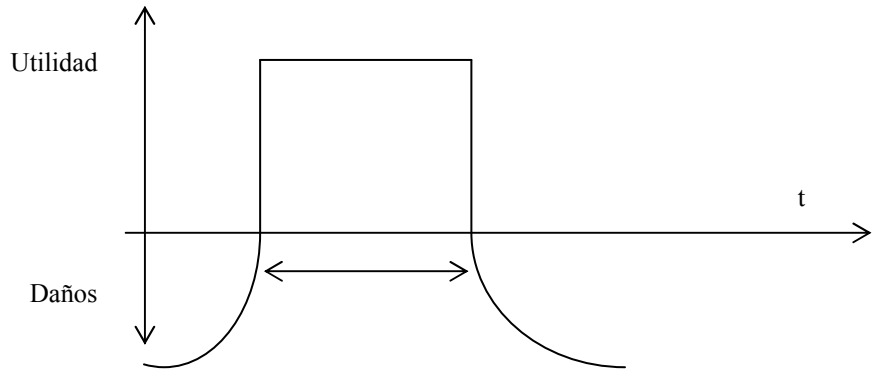
Se dice que una tarea es **crítica (*hard*)** cuando no se puede admitir que sobrepase el plazo de respuesta especificado ni una sola vez. Es **acrítica (*soft*)** si es admisible que se sobrepase el plazo ocasionalmente. Es **firme (*firm*)** si el plazo no es crítico, pero la respuesta tardía no es válida. Es **interactiva (*interactive*)** si no se especifican plazos de respuesta, sino tiempos de respuesta adecuados.




Tarea *soft*




Tarea firme




Tarea *hard*

### 6.5.1 Tareas periódicas en Ada.

Ya las hemos visto en un apartado anterior.

```
task body TPeriodica is
   Periodo: constant Duration :=...; -- o constant Time_Span
   Sig: Time := ...;
begin
   ...
   loop
      delay until Sig;
      -- ACCION PERIODICA;
      Sig:=Sig+Periodo;
   end loop;
end TPeriodica;
```

Tarea periódica con acción temporizada:

```
task body TPeriodica is
   Periodo: constant Duration :=...; -- o constant Time_Span
   TiempoL: constant Duration :=...; -- o constant Time_Span
   Sig, Limite: Time := ...;
   Overrun: exception;
begin
   ...
   loop
      delay until Sig;
      Limite:=clock+TiempoL;
      select
         delay until Limite;
```

```
            raise Overrun;
         then abort
      --ACCION PERIODICA;
         end select;
         Sig:=Sig+Periodo;
      end loop;
   exception
      when Overrun => ...;
   end TPeriodica;
```

### 6.5.2 Tareas esporádicas en Ada.

Las tareas esporádicas responden frente a eventos interno o externos. En esencia están bloqueadas hasta que el evento por el que esperan se produce. El suceso de activación se implementa mediante un objeto protegido. La tarea se activa cuando ocurre el evento que se traducirá en que una barrera de una *entry* se volverá cierta y la tarea podrá eructar una entrada protegida en la que estaba esperando.

```
protected Evento is
   procedure Signal;    --asociado al evento asíncrono
   entry Wait;
private
   Activacion: boolean:= False;
end Evento;

protected body Evento is
   procedure Signal is
   begin
      Activacion:=True;
   end Signal;
   entry Wait when Activacion is
   begin
      Activacion:=False;
   end Wait;
end Evento;

task body TEsporadica is
begin
   --Inicialización
   loop
      Evento.Wait;
      --ACCION ESPORADICA;
   end loop;
end TEsporadica;
```

Ejemplo En el programa aperiodica.adb y aperiodica2.adb hay una tarea calcular que se activa esporádicamente desde el programa principal. Esta tarea realiza el cálculo de una sucesión y el principal la lanza según intervalos de tiempo aleatorios, en el primer caso, en el segundo es la pulsación de una tecla quien activa la tarea calcular.

## 6.6 Planificación

Supongamos un programa concurrente formado por 5 procesos independientes. En un sistema monoprocesador, y si los procesos, una vez en ejecución, no abandonan el procesador hasta su finalización, nos encontramos con 120 formas distintas en el orden de ejecución de los procesos (ABCDE, ABCED, ABDCE, ABDEC, ...). Si los procesos, antes de finalizar, ceden a otro su puesto en el procesador, o si se tarta de un sistema multiprocesador, los órdenes de ejecución son casi infinitos. Aunque el resultado del programa concurrente sea el mismo, independientemente del orden de

ejecución de los procesos (sistemas no deterministas), su comportamiento temporal puede variar considerablemente. Si uno de los procesos es crítico, quizás sean válidas sólo aquellas formas de ejecución en las que dicho proceso se ejecute el primero. Un sistema en tiempo real necesita restringir el indeterminismo de los sistemas concurrentes para poder garantizar el cumplimiento de los requisitos temporales. Esto es lo que se conoce como planificación.

Un método de **planificación** tiene dos aspectos importantes:

- Un **algoritmo de planificación**, que determina el orden de acceso de los procesos a los recursos del sistema (particularmente, al procesador).
- Un **método de análisis**, que permite calcular el comportamiento temporal del sistema. En general, se estudia el peor comportamiento posible, y se comprueba si el sistema cumple, en ese caso, todos sus requisitos temporales. Si los cumple, queda garantizado que puede cumplirlos en todos los casos.

Un método de planificación puede ser **estático** (se realiza antes de la ejecución), o **dinámico** (se realiza durante la ejecución, y depende de ella).

Métodos estáticos de planificación muy utilizados son los que conllevan prioridades fijas y desalojo (*pre-emption*). La prioridad de una tarea o proceso es un parámetro relacionado con la urgencia o importancia del mismo. Si 2 tareas con diferentes prioridades están listas para ejecutar en el mismo procesador, la tarea con prioridad más alta debe ejecutarse primero. Si la prioridad es igual, el orden vendrá determinado por la implementación. Además, el sistema debe ser ***pre-emptive***: una tarea en ejecución debe abandonar el procesador cuando otra tarea de mayor prioridad pasa al estado *ready*, es decir, está lista para ejecución. Un método de planificación incluirá un algoritmo de asignación de prioridades y un test de planificabilidad.

### 6.6.1 Modelo simple

En esta sección vamos a presentar un modelo de sistema concurrente muy simple, con el fin de describir los métodos de planificación más habituales. Posteriormente generalizaremos este modelo. El modelo básico consta de las siguientes características:

- El sistema consta de un número fijo de procesos.
- Todos los procesos son periódicos, con periodos conocidos.
- Los procesos son independientes unos de otros.
- Los plazos de respuesta de los procesos son iguales a sus periodos respectivos.
- El tiempo de ejecución máximo de cada proceso es conocido.
- Se asume que las operaciones del sistema son instantáneas (coste cero en tiempo).

**Parámetros de planificación:**

| | |
|---|---|
| N | Número de procesos |
| T | Periodo de activación |
| C | Tiempo de ejecución máximo |
| D | Plazo de respuesta |
| R | Tiempo de respuesta máximo |
| P | Prioridad |

En el modelo básico, para cada tarea se cumple C <= D = T. Se trata de asegurar que R <= D.

En este modelo, el mínimo común múltiplo de los periodos de activación de todas las tareas se conoce como **hiperperiodo** del sistema. El comportamiento temporal del sistema se repite cada hiperperiodo.

## 6.6.2 Planificación por prioridades fijas con desalojo

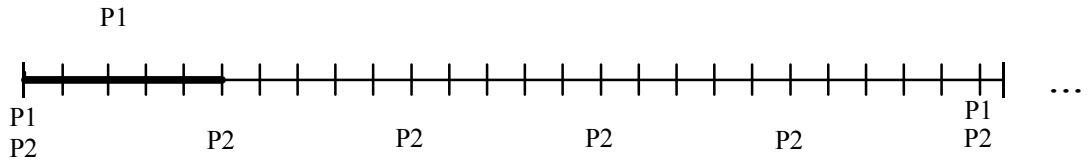
Si utilizamos un método con desalojo (*pre-emptive*) en cada momento se ejecutará la tarea con mayor prioridad de todas las tareas ejecutables. En un método *non pre-emptive*, una tarea podrá completarse aunque haya tareas con mayor prioridad listas para ejecución. Existen otras alternativas intermedias, llamadas de **planificación cooperativa**, que permiten que un proceso de baja prioridad ejecute durante un tiempo limitado, pero no entraremos en ellas. A partir de ahora, a lo largo del tema, asumiremos planificación *pre-emptive.*

### 6.5.2.1. Algoritmo de planificación por preferencias (PPSA. *Pre-emptive preference scheduling algorithm*)

Es el algoritmo más simple. A todos los procesos se les asignan las prioridades según su importancia cara al sistema. El planificador siempre ejecutará el proceso de prioridad más alta. Esto garantizará que el proceso de más alta prioridad cumplirá sus restricciones de tiempos de ejecución (suponiendo que el procesador es suficientemente rápido como para ejecutar, dentro del límite de tiempo, un ciclo de este proceso). Desafortunadamente, es lo único que este algoritmo nos asegura. Considerar, por ejemplo, lo siguiente:

| Proceso ($P_i$) | Periodo ($T_i$) | Ejecución ($C_i$) | Utilización |
|---|---|---|---|
| P1 | 50 | 10 | 20% |
| P2 | 10 | 2 | 20% |

Supongamos que P1 tiene mayor prioridad que P2, puesto que es más importante. Si ambos procesos comienzan al mismo tiempo, P1 se ejecutará durante 10 segundos, y luego será suspendido por otros 40 segundos. Aunque durante estos 40 segundos el procesador está libre para ejecutar P2, es demasiado tarde, puesto que debería ejecutarse cada 10 segundos, y periodo ya transcurrió.




### 6.6.2.1 Algoritmo de planificación por frecuencia (RMSA. *Rate monotonic scheduling algorithm*)

A cada proceso se le asigna una prioridad fija basándose en su periodo: cuanto más corto el periodo o intervalo de ejecución (mayor la frecuencia) mayor la prioridad. Este algoritmo es óptimo para el modelo simple: se puede probar que si existe una asignación de prioridades factible para un grupo de procesos, la asignación mediante RMSA también será factible.

### 6.5.2.3. Test de planificabilidad basado en la utilización

Test de planificabilidad muy simple, aunque no exacto.

**Factor de utilización** del procesador: es una medida de la carga del procesador para un conjunto de tareas. Se define como:

$$U = \sum_{i=1}^{N} \frac{Ci}{Ti}$$

En un sistema monoprocesador debe ser U<=1

Para el modelo simple, con el algoritmo RMSA, los plazos están garantizados si se cumple la siguiente condición:

$$U < U0 = N(2^{1/N} - 1)$$

| N | U0 |
|---|---|
| 1 | 1.000 |
| 2 | 0.828 |
| 3 | 0.779 |
| 4 | 0.756 |
| ... | ... |
| ∞ | Log 2 = 0.693 |

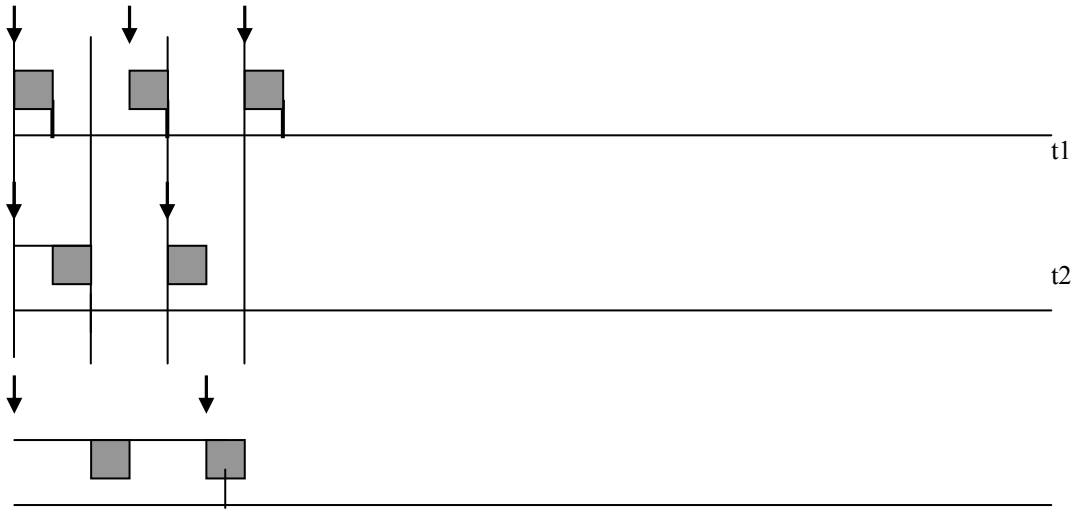
De la tabla anterior se puede deducir que un sistema con una utilización menor del 69.3% será planificable utilizando el algoritmo RMSA

**- Ejemplo 1:**

| **Tarea** | **T** | **C** | **P** | **C/T** |
|---|---|---|---|---|
| t1 | 30 | 10 | 3 | 0.333 |
| t2 | 40 | 10 | 2 | 0.250 |
| t3 | 50 | 12 | 1 | 0.240 |
| | | | | U=0.823 |

El sistema no cumple el test de utilización (U > 0.779).

Según vemos en el cronograma, la tarea t3 falla en T=50, puesto que sólo ha consumido 10 unidades de tiempo de ejecución y hubiera necesitado 12.




0 20 40 60 ...

El cronograma se puede utilizar para comprobar si se cumplen los plazos de respuesta. Para la comprobación, hay que trazar el cronograma para un hiperperiodo completo. Sin embargo, si todas las tareas se activan a la vez, bastaría comprobar el cronograma para el periodo más largo de todos.

- **Ejemplo 2:**

| **Tarea** | **T** | **C** | **P** | **C/T** |
|---|---|---|---|---|
| t1 | 16 | 4 | 3 | 0.250 |
| t2 | 40 | 5 | 2 | 0.125 |
| t3 | 80 | 32 | 1 | 0.400 |
| | | | | U=0.775 |

Este sistema está garantizado (U < 0.779). Dibujar el cronograma.

- **Ejemplo 3:**

| **Tarea** | **T** | **C** | **P** | **C/T** |
|---|---|---|---|---|
| t1 | 20 | 5 | 3 | 0.250 |
| t2 | 40 | 10 | 2 | 0.250 |
| t3 | 80 | 40 | 1 | 0.500 |
| | | | | U=1.000 |

El sistema no cumple el test de utilización (U > 0.779). Sin embargo, es planificable (se cumplen los plazos). Comprobar dibujando el cronograma.

El test de utilización es condición suficiente pero no necesaria: si un sistema pasa el test, cumplirá los plazos. Si no lo pasa, puede ser o puede no ser planificable.

### 6.6.2.2 Test de planificabilidad basado en el análisis del tiempo de respuesta de cada tarea

Para el proceso de más alta prioridad, el tiempo máximo de respuesta va a ser igual al tiempo máximo de computación (R = C). El resto de procesos sufrirán interferencias por la ejecución de procesos más prioritarios. Para ellos, el tiempo de respuesta será la suma de su tiempo de cómputo más la interferencia: Ri = Ci + Ii, donde Ii representa la máxima interferencia que el proceso i experimenta en cualquier intervalo de tiempo [t, t+Ri). Supongamos que todos los procesos son activados simultáneamente en el tiempo 0. Consideremos un proceso j de mayor prioridad que otro i. Dentro del intervalo [0, Ri), el número de veces que el proceso j es activado vendrá dado por la expresión:

$$NAj = \left\lceil \frac{Ri}{Tj} \right\rceil$$

Que devuelve el entero más pequeño mayor que el cociente. Si Ri =15 y Tj=6, hay 3 activaciones del proceso j, en los instantes 0, 6, 12. Cada activación del proceso j dará lugar a una interferencia para el proceso i igual a Cj. Como cada proceso de mayor prioridad que i interferirá con él, llegamos a la siguiente expresión:

$$Ii = \sum_{j \in hp(i)} \left\lceil \frac{Ri}{Tj} \right\rceil \cdot Cj$$

Donde hp(i) es el conjunto de los procesos con mayor prioridad que el proceso i. La ecuación del tiempo de respuesta será por tanto:

$$Ri = Ci + \sum_{j \in hp(i)} \left\lceil \frac{Ri}{Tj} \right\rceil \cdot Cj$$

La ecuación anterior tiene el término Ri en ambos lados. Habrá muchos valores de Ri que sean solución de la ecuación, pero el que nos interesa es el más pequeño. Una forma de resolver la ecuación es mediante la relación de recurrencia:

$$Ri^{n+1} = Ci + \sum_{j \in hp(i)} \left\lceil \frac{Ri^{n}}{Tj} \right\rceil \cdot Cj$$

Un valor inicial aceptable es

$$Ri^{0} = Ci + \sum_{j \in hp(i)} Cj$$

Se termina la recurrencia cuando Ri(n+1) = Ri(n), o bien cuando Ri(n+1) > Ti, lo que implica que no se cumple el plazo.

**- Ejemplo 4:**

| **Tarea** | **T** | **C** | **P** |
|---|---|---|---|
| t1 | 7 | 3 | 3 |
| t2 | 12 | 3 | 2 |
| t3 | 20 | 5 | 1 |

**R1 = 3**

R2(0) = 3 + 3 = 6

R2(1) = 6 => **R2 = 6**

R3(0) = 5 + 3 + 3 = 11

R3(1) = 5 + 2*3 + 1*3 = 14

R3(2) = 5 + 2*3 + 2*3 = 17

R3(3) = 5 + 3*3 + 2*3 = 20

R3(4) = 5 + 3*3 + 2*3 = 20 => **R3 = 20**

Todas las tareas tienen sus plazos garantizados. Comprobar que para el ejemplo 3 obtenemos la siguiente tabla:

| **Tarea** | **T** | **C** | **P** | **R** |
|---|---|---|---|---|
| t1 | 20 | 5 | 3 | 5 |
| t2 | 40 | 10 | 2 | 15 |
| t3 | 80 | 40 | 1 | 80 |

Y que por tanto el sistema cumple sus plazos.

Los cálculos del tiempo de respuesta dan condiciones necesarias y suficientes (si el sistema pasa el test, cumplirá sus plazos. Si no, no es planificable).

#### 6.6.2.3 Algoritmo de planificación por tiempo límite (DDSA. *Deadline Driven scheduling algorithm*)

Se asigna dinámicamente la prioridad a un proceso, de forma que el proceso que tenga su tiempo límite (final del periodo) más próximo recibe la mayor prioridad. Cuando éste se completa, se vuelve a hacer una nueva asignación de prioridades. Este es evidentemente un mecanismo más complejo que requiere de una ejecución de un proceso que establece las prioridades basándose en los tiempos límites de las tareas del sistema. Si un conjunto de procesos es planificable según RMSA, o por cualquier otro algoritmo, también lo es con este método. Suele ofrecer resultados óptimos en cuanto a dar mayor cumplimiento a los requisitos temporales de las tareas.

En la industria se aplica mucho el algoritmo RMSA por ser más sencillo de implantar y por resultar muy estable respecto a los procesos rígidos. Al dar prioridades fijas, y funcionando de manera apropiativa, se garantiza que los procesos de tiempo real rígidos (siempre tendrán las prioridades más altas) podrán ejecutarse y el incumplimiento de plazos recaerá sobre los procesos menos prioritarios. Con un algoritmo DDSA las prioridades son cambiantes, una tarea de poca importancia para el sistema podría en un momento dado estar muy cerca de su tiempo límite y alcanzar una prioridad muy alta, por lo que se podría planificar por delante de tareas rígidas que podrían así ver incumplidas sus especificaciones temporales.

### 6.6.3 Procesos esporádicos

Para expandir el modelo simple de la sección 6.5.1 de modo que incluya tareas esporádicas, el valor de T se considera como la separación mínima entre dos activaciones de una tarea. Un proceso esporádico con un valor de T de 20 ms no se activará más de una vez en cada intervalo de 20 ms. En realidad, se activará con mucha menor frecuencia.

El modelo simple asume que D = T. Para procesos esporádicos, esto no es apropiado. Uno de estos procesos puede ser invocado muy infrecuentemente, pero una vez que se invoca, puede requerir una respuesta urgente (a menudo se crean tareas esporádicas para responder ante situaciones de error). El modelo debe permitir que D < T.

#### 6.6.3.1 Algoritmo de planificación por plazos (*Deadline Monotonic Priority Ordering*, DMPO)

Cuando los plazos D son menores que los periodos, se asigna una prioridad a cada tarea inversamente proporcional a su plazo de respuesta (a mayor D, menor P).

El test basado en el análisis del tiempo de respuesta de cada tarea sigue siendo válido.

Se termina la recurrencia cuando Ri(n+1) = Ri(n), o bien cuando Ri(n+1) > Di, lo que implica que no se cumple el plazo.

**- Ejemplo 5:**

| **Tarea** | **T** | **C** | **D** | **P** | **R(DMPO)** | **R (RMSA)** |
|---|---|---|---|---|---|---|
| t1 | 20 | 3 | 5 | | | |
| t2 | 15 | 3 | 7 | | | |
| t3 | 10 | 4 | 10 | | | |
| t4 | 20 | 3 | 20 | | | |

Comprobar que con el algoritmo DMPO se cumplen los plazos, y que no se cumplen con el algoritmo RMSA.

### 6.6.4 Interacción entre procesos

Una de las características más simplistas del modelo simple expuesto es la de que los procesos son independientes. En la mayoría de los casos, existirá una interacción entre los procesos. Esta interacción se puede producir mediante el acceso a datos comunes (protegidos) o directamente mediante citas o mensajes. En estos casos existe la posibilidad de que un proceso de alta prioridad tenga que esperar por un suceso de otro menos prioritario (ejecución de un rendezvous, ejecutar una operación de un objeto protegido, ocupación de un semáforo,...).

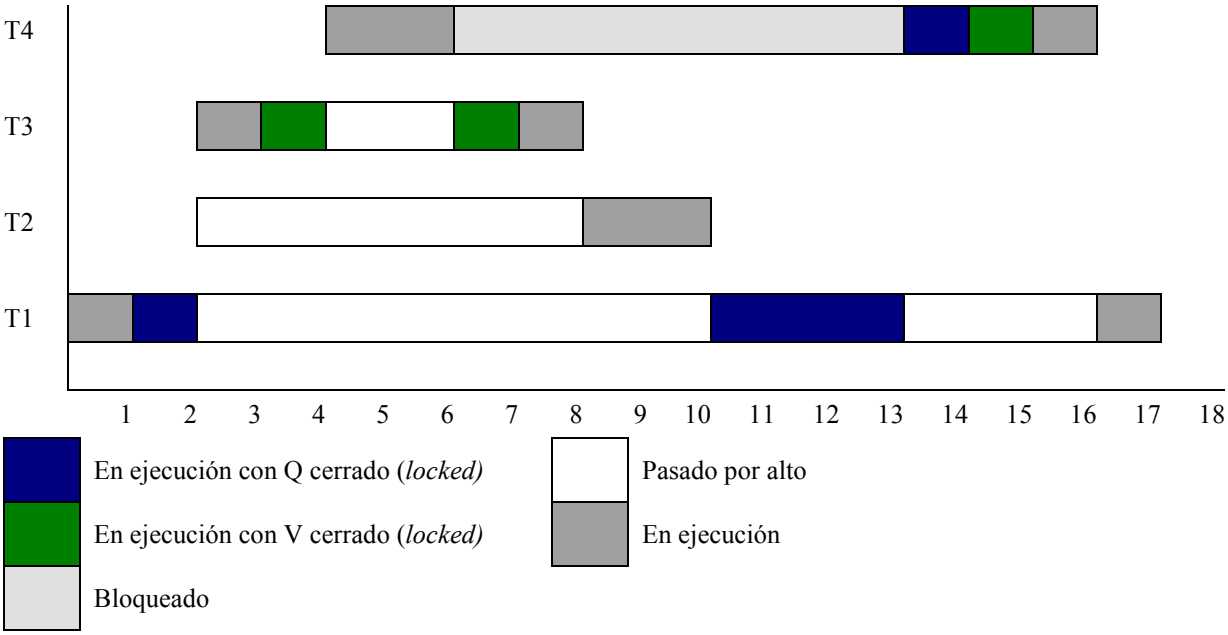
Si un proceso queda suspendido esperando que otro de menor prioridad complete una computación, se dice que está **bloqueado** y que se produjo una **inversión de prioridad**.

- **Ejemplo 6:**

Sean 4 tareas T1 a T4 que pueden acceder a dos recursos Q y V de manera exclusiva según el orden que se detalla en la siguiente tabla. E representa la ejecución durante un tic y Q y V representa la ejecución durante un tic utilizando los respectivos recursos.

| Tarea | Prioridad | Secuencia Ejecución | Tiempo Activación |
|---|---|---|---|
| T4 | 4 | EEQVE | 4 |
| T3 | 3 | EVVE | 2 |
| T2 | 2 | EE | 2 |
| T1 | 1 | EQQQQE | 1 |

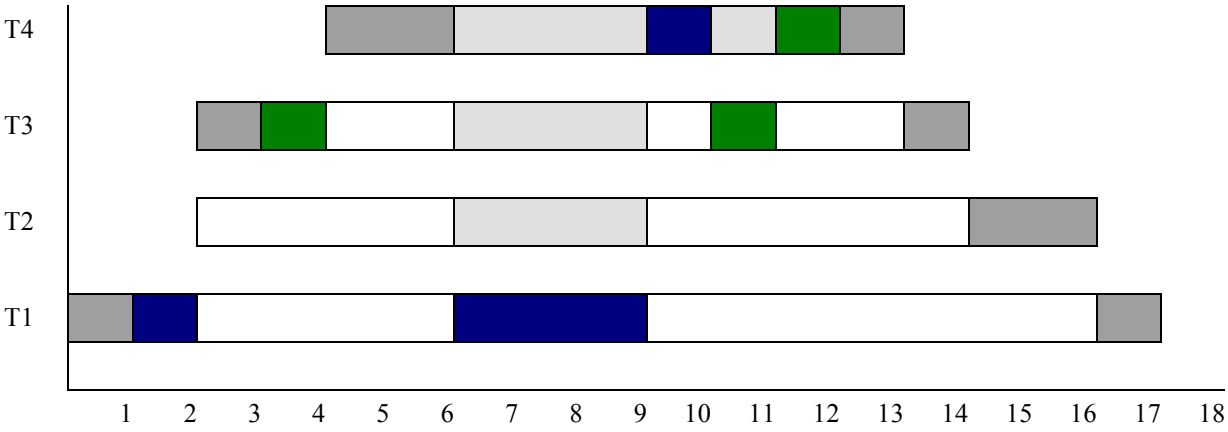
A continuación se ilustra la secuencia de ejecución de los procesos desde el instante t=0.




T1 comienza a ejecutarse. En **1** toma el recurso Q de forma exclusiva haciendo una operación de *locked.* En **2** se lanzan los procesos T2 y T3. Como T3 es más prioritario pasa por encima de T1, que se detiene, y de T2 que no ha comenzado aun su ejecución. En **3** la tarea T3 toma el recurso V, también de forma exclusiva, T1 y T2 permanecen en espera. En **4** se lanza la tarea T4 que es la más prioritaria y pasa por alto a T3. T4 continúa ejecutándose hasta el instante **6**. En este punto T4 desea acceder al recurso Q, que está siendo utilizado de manera exclusiva por T1, por tanto no puede emplearlo. Se bloquea a la espera de que el recurso está disponible, y el procesador pasa por tanto a T3. T3 continúa su ejecución con el proceso V, que libera en **7.** Finalmente T3 termina en **8**. En ese momento T2 puede comenzar con su ejecución que llevará hasta el final sin consumir recursos. T2 acaba en **10**. En este instante tenemos a T4 bloqueado esperando por Q y a T1 que estaba pasado por alto y ahora

puede ejecutarse por no haber ninguna tarea preparada con prioridad más alta. Se ejecuta usando Q hasta el instante **13** que libera el recurso. En este momento T4, que estaba a la espera de Q, se desbloquea y pasa por alto a T1. T4 ejecuta un tic con V y otro sin recursos concluyendo su trabajo en **16**. Cuando T4 finaliza T1 puede acabar consumiendo el tic que le resta. Los tiempos de respuesta de cada proceso (t_conclusión – t_lanzamiento) son respectivamente 12, 6, 8, y 17.

Hay un problema de **inversión de prioridad** ya que T4 se ve bloqueada no sólo por T1, necesario para mantener la integridad de los datos al usar un recurso compartido, sino que también ve perturbado su funcionamiento por T3 y T2. El problema se presenta por utilizar recursos compartidos y mantener un esquema de prioridades fijo. Una medida para limitar su efecto es el empleo de **herencia de prioridades**. Bajo este esquema si un proceso A se bloquea (no que sea pasado por alto) esperando a que otro proceso B finalice algún cálculo, entonces B pasa a tomar la prioridad de A. En el ejemplo anterior cuando T4 (proceso A) se bloquease esperando a que T1 (B) acabe de usar Q se produciría la herencia de prioridades por la cual T1 pasaría a tomar la prioridad de T4, y sería por tanto de mayor prioridad que T2 y T3. Con esta regla la prioridad de un proceso es la máxima entre su propia prioridad y la de otros procesos que dependan de él. Con la herencia de prioridad el esquema de ejecución quedaría como se muestra en la siguiente figura.

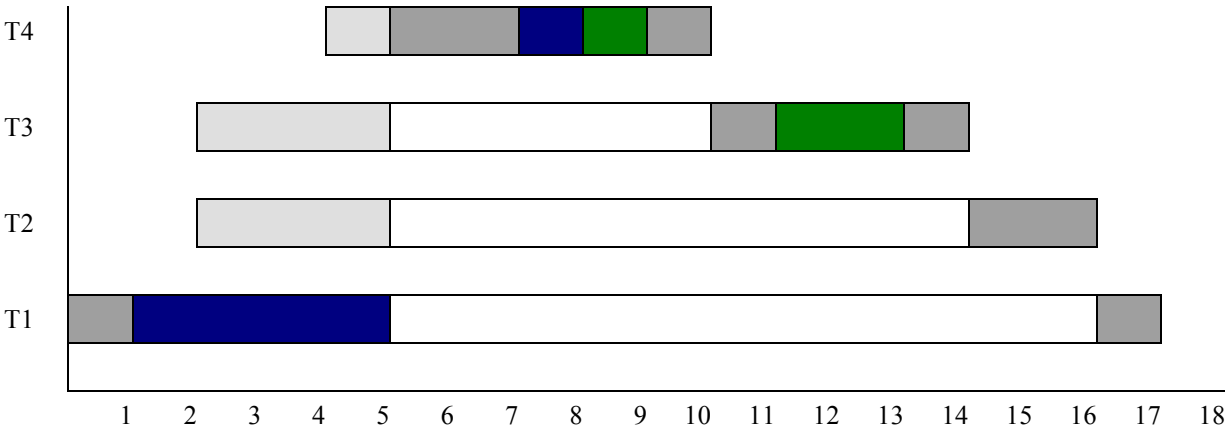



Los tiempos de respuesta son 9, 12, 14 y 17 frente a los obtenidos anteriormente 12, 6, 8, y 17. El proceso más prioritario ha reducido su tiempo de respuesta, incrementándose el de los procesos intermedios. La justificación de cada transición en la ejecución de los procesos queda propuesta como ejercicio.

Los esquemas de herencia de prioridad no tienen por qué limitarse a un único paso: si T4 espera por T3, pero T3 no puede avanzar porque espera por T2, entonces a T2 también se le puede dar la prioridad de T4.

Uno de los protocolos de herencia de prioridad más eficientes, que se llama *Immediate Ceiling Priority Protocol* (ICPP), se define de la siguiente forma:

- Cada proceso tiene una prioridad base, asignada según algún esquema de planificación.
- Cada recurso tiene un valor de tope (*ceiling*) definido, que es la máxima de la de los procesos que lo usan.
- Un proceso tiene una prioridad dinámica que es la máxima de su prioridad base y el valor tope de cualquiera de los procesos que tiene asignados.
-



En **1** T1 se asigna el proceso Q. Durante los siguientes 4 tics se ejecutará con prioridad 4, por tanto cuando en **2** se lancen T2 y T3 van a verse bloqueados por un proceso de más alta prioridad. Lo mismo pasa con T4 en **4**. En el instante **5** T1 deja el recurso Q, e inmediatamente ve decrementada su prioridad al nivel 1, por lo que puede ejecutarse T4 que es la tarea preparada de mayor nivel. T4 se ejecuta ininterrumpidamente hasta **10**, tomando los recursos (que ahora están libres) cuando los precisa. El tiempo de respuesta para T4 ahora es sólo de 6 tics. El resto de tiempos son 12, 14 y 17.

Con este protocolo, un proceso se puede bloquear, como máximo, una vez por cada activación. La duración máxima del bloqueo será:

$$Bi = \max_{j \in lp(i), k \in lc(i)} C_{j,k}$$

donde lp(i) es el conjunto de procesos de menor prioridad que el proceso i, lc(i) es el conjunto de recursos compartidos, y Cj,k es el tiempo durante el que el proceso j accede al recurso k.

La ecuación del tiempo de respuesta incluirá este término Bi:

$$Ri = Ci + Bi + \sum_{j \in hp(i)} \left\lceil \frac{Ri}{Tj} \right\rceil$$

El problema de la inversión de prioridad no es un mero ejercicio teórico, se ha presentado en ocasiones con efectos funestos. Podéis ver lo que le paso al vehículo Pathfinder en el documento Pathfinder.Doc

# 6.7 Planificación de tareas en Ada

En el anexo de tiempo real se define un modelo de planificación con prioridades y desalojo.

El package System incluye declaraciones para las prioridades y lo podemos encontrar al final de este capítulo. Abreviadamente la prioridad en Ada es un subtipo entero donde los valores más altos denotan mayores prioridades.

El programador puede fijar la prioridad inicial de una tarea mediante la inclusión de un pragma en su especificación:

```
task CONTROLADOR is
   pragma priority(10);
end CONTROLADOR;
```

La forma del pragma priority es:

**pragma PRIORITY(expression);**

Donde *expression* es un valor de un subtipo entero. Este *pragma* debe estar incluido en la parte especificativa de una tarea o en la parte declarativa del cuerpo del procedimientoprincipal.

En Ada83, la prioridad de una tarea debe tener un valor estático, y es siempre igual a la prioridad base.

En Ada95, las tareas de un mismo tipo pueden tener prioridades distintas:

```
task type SERVIDORES(PRIORIDAD:System.Priority) is
-- el discriminante utilizado en la definición de una task
-- ha de ser un escalar
   entry ...
   entry ...
```

```
      pragma priority(PRIORIDAD);
   end SERVIDORES;
   ...
   S1: SERVIDORES(System.Priority'FIRST);
   S2: SERVIDORES(System.Priority'LAST);
```

Si el procedure principal no lleva pragma priority, su prioridad toma el valor por defecto definido en system (System.Default_Priority). Cualquier otra tarea que no lleve este pragma tendrá la misma prioridad base que su tarea padre.

**Ejemplos tareas1.adb, tareas2.adb, tareas3.adb. Son ejemplos con 2 tareas periódicas más el programa principal, donde se ve como funciona el planificador apropiativo del Ada, desalojando de la CPU a tareas menos prioritarias.**

**TAREAS1 define dos tareas y las arranca mediante una llamada. A partir de ese instante, ambas van a entrar en un bucle durante 30 segundos, dando avisos cada 5 segundos. Como en estas tareas no hay puntos de intercambio, la primera que entre en el bucle, (o la de mayor prioridad) se ejecutará hasta que termine. Probar su funcionamiento variando las prioridades.**

**TAREAS2 hace lo mismo que TAREAS1, pero utiliza sentencias delay en lugar de bucles de espera activa, con lo cual se introducen puntos de intercambio. Probar su funcionamiento asignando prioridades.**

**TAREAS3 es una prueba de si el sistema es pre-emptive o no. La tarea de menor prioridad no tiene puntos de intercambio. Si el sistema no es pre-emptive, esta tarea, una vez empiece a ejecutarse, lo hará hasta que termine. Si el sistema es pre-emptive, cuando la tarea de mayor prioridad se desbloquee y esté lista para ejecución, sacará del procedure a la otra.**

### 6.7.1 Herencia de prioridades

Se puede utilizar el protocolo de herencia de prioridades definido en el apartado anterior incluyendo el siguiente pragma:

**pragma Locking_Policy(Ceiling_Locking);**

Si no se incluye, la forma de actuación dependerá de la implementación.

Al tipo protegido se le asignará una prioridad mediante el pragma priority. Si una tarea con prioridad base mayor que la prioridad del objeto protegido intenta utilizar una operación del mismo, se alzará la excepción PROGRAM_ERROR.

Además de la herencia de prioridades del tipo protegido, hay otras herencias:

- Durante la activación: la tarea que se está activando hereda la prioridad del padre.
- Durante el rendezvous: la tarea que ejecuta la sentencia accept heredará la prioridad de la tarea llamadora, si es mayor que su propia prioridad.

**Prioridad activa**: Prioridad que tiene una tarea en un momento determinado, máximo de la prioridad base y la heredada.

### 6.6.2. Prioridades dinámicas

En Ada95 la prioridad base puede ser cambiada. Para ello, el lenguaje incluye el package Ada.Dynamic_Priorities:

```
package Ada.Task_Identification is
   type Task_ID is private;
   Null_Task_ID : constant Task_ID;
   function  "=" (Left, Right : Task_ID) return Boolean;
```

```
   function  Image(T : Task_ID) return String;
   function  Current_Task return Task_ID;
   procedure Abort_Task(T : in out Task_ID);
   function  Is_Terminated(T : Task_ID) return Boolean;
   function  Is_Callable(T : Task_ID) return Boolean;
   private
      ... -- not specified by the language
end Ada.Task_Identification;


with System;
with Ada.Task_Identification;
package Ada.Dynamic_Priorities is
   procedure Set_Priority(Priority : in System.Any_Priority; T : in
     Ada.Task_Identification.Task_ID :=
da.Task_Identification.Current_Task);
   function Get_Priority (T : Ada.Task_Identification.Task_ID :=
Ada.Task_Identification.Current_Task) return System.Any_Priority;
end Ada.Dynamic_Priorities;
```

Get_Priority devuelve la prioridad base actual de la tarea.

Set_Priority permite cambiar la prioridad base de la tarea.

### 6.7.2 Prioridades de colas

Un programador en Ada95 puede escoger también la política de colas para una entry de una tarea o de un tipo protegido.

**pragma QUEUING_POLICY(Policy_Identifier);**

Policy_Identifier puede tomar uno de estos dos valores: FIFO_QUEUING o PRIORITY_QUEUING;

FIFO_QUEUING es el valor por defecto, y el único en Ada83. Las colas son atendidas según el orden de llegada de las tareas. Si se escoge PRIORITY_QUEUING, las llamadas quedarán en la cola de la entrada ordenadas por las prioridades activas de las tareas llamadoras.

Otro modo de dar prioridades a las entries:

```
task SERVIDOR is
   entry PETICION(...);
end SERVIDOR;
```

Todos los clientes de esta tarea deben llamar a la entrada PETICION, y serán atendidos en orden FIFO. Si los clientes pueden ser divididos en tres niveles de prioridad, serán necesarias tres entradas PETICION diferentes, de forma que cada nivel de prioridad tenga su propia cola FIFO. Se pueden programar las llamadas **familias de entradas**:

```
type PRIORIDADES is (ALTA, MEDIA, BAJA);
task SERVIDOR is
   entry PETICION(PRIORIDADES)(...);
end SERVIDOR;

task body SERVIDOR is
begin
   loop
      select
         accept PETICION(ALTA)(...) do
          ...
         end PETICION;
      or
         when PETICION(ALTA)'COUNT=0 =>
```

```
                accept PETICION(MEDIA)(...) do
                 ...
                end PETICION;
          or
             when PETICION(ALTA)'COUNT=0 and PETICION(MEDIA)'COUNT=0 =>
                accept PETICION(BAJA)(...) do
            ...
                end PETICION;
          end select;
       end loop;
    end SERVIDOR;
```

Llamada: `SERVIDOR.PETICION(ALTA)(...);`

El Atributo COUNT: da el número de tareas que está esperando en la cola de la entrada dada.

**Ejemplos sin_pri_entries.adb y pri_entries.adb: Una tarea servidora cuenta con tres puntos de cita, se crean tareas clientes con diferentes prioridades que pueden citarse en alguno de esos puntos. En el primer caso veríamos que no hay preferencia a la hora de atender a los clientes. En el segundo caso se atiende a las tareas no según su prioridad sino según el punto de cita de la servidora en el que se quieran citar (familias de entradas).**

### 6.7.3 Planificación en Ada

Vamos a ver las reglas que determinan qué tarea es seleccionada para ejecución cuando hay más de una ready. El modelo de despacho de tareas según el manual de referencia del Ada especifica planificación ***preemptive*** basada en el concepto de colas de tareas ready ordenadas por prioridades: **una tarea debe abandonar el procesador cuando otra tarea de mayor prioridad queda lista para ejecución.**

La selección de la tarea que va a ser ejecutada se efectúa en determinadas partes llamadas **puntos de intercambio**. Una tarea alcanza un punto de intercambio siempre que se bloquea y siempre que pasa al estado ready. Para la tarea que está ejecutando también son puntos de intercambio la finalización de una sentencia accept y el paso al estado completa.

Cada procesador tiene una cola de tareas ready para cada valor de la prioridad. En un sistema multiprocesador una tarea puede estar en varias colas ready. En cualquier instante, cada cola contiene el grupo de tareas de su prioridad que están listas para ejecución en ese procesador, y que no están ejecutando en ningún procesador. Cuando una tarea que se está ejecutando en un procesador alcanza un punto de intercambio, se selecciona una tarea para ser ejecutada en dicho procesador; la tarea que se selecciona es la que está en la cabeza de la cola ready de mayor prioridad no vacía. Esa tarea se borra de todas las colas a las que pertenezca. También se selecciona una nueva tarea para ejecución en el caso de que haya una cola ready no vacía con mayor prioridad que la tarea en ejecución.

Las colas ready son puramente conceptuales. No es necesario que tales listas existan físicamente en una implementación.

**pragma Task_Dispatching_Policy(Policy_Identifier);**

Policy_Identifier: FIFO_WITHIN_PRIORITY definido por el lenguaje, u otro definido por la implementación. Este pragma establece la colocación de las tareas en las colas ready. Si se especifica FIFO_WITHIN_PRIORITY las tareas se colocan en las colas en orden FIFO según van quedando listas para ejecución. Se da una excepción en el caso de que una tarea sea sacada del procesador por otra de mayor prioridad. Dicha tarea se colocará en la cabeza de la cola de tareas listas de su nivel de prioridad. Si se especifica FIFO_WITHIN_PRIORITY se debe especificar también CEILING_LOCKING en el pragma Locking_Policy.

**Ejemplos fifo_ques.adb. Mediante un par de pragmas presentes al comienzo del código se puede escoger como deseamos que se organicen la tareas pendientes en las entradas.**

## 6.8 Control de recursos

Cuando se dispone de una serie de procesos que compiten por la utilización de recursos compartidos no hay una comunicación directa entre los procesos. Esto es, no se transmiten información del resultado de sus respectivos códigos, sin embargo si se pueden comunicar para compartir los recursos.

La implementación de entidades recursos precisa de la creación de algún tipo de agente. Si este es pasivo, no tiene por si vida, entonces se dice que el recurso está protegido o sincronizado (objetos protegidos, monitores etc.). Si por el contrario son activos (procesos), se requiere un agente para su sincronización, entonces el control del recurso es un servidor.

Veremos ahora parte de la problemática de la compartición de recursos de manera segura y como las acciones que los procesos realizan interfieren las acciones de los demás, en particular el fallo en un proceso puede impedir la asignación de un recurso al resto de los procesos.

El controlador del recurso no tiene porque ejecutarse, en sí mismo, de manera atómica. Eso si, debe garantizar la aceptabilidad global de cualquier cambio que se produzca entre sus datos. Donde si será preciso realizar una ejecución atómica es en el código que los diferentes procesos deban ejecutar para comunicarse con el agente.

### 6.8.1 Potencia expresiva y facilidad de uso.

Bloom estableció unas líneas para evaluar la calidad de las primitivas de sincronización, para ello se fijaba en:

- La facilidad con la que se expresaban las restricciones de sincronización
- La facilidad con la que se podían combinar para poder establecer otras más complejas.
- Las restricciones que era preciso expresar son:
- El tipo de servicio requerido
- El orden en el que son recibidas las solicitudes.
- El estado del servidor y cualquier objeto que él maneje.
- Los parámetros de la solicitud.

Adicionalmente se puede incluir la prioridad del cliente.

En general hay dos aproximaciones lingüísticas para restringir el acceso a los recursos. La primera es la sincronización condicional, todas las solicitudes son aceptadas pero cualquier proceso al que no se pueda atender es suspendido en una cola, este es el comportamiento de los monitores. La otra opción es la evitación (*avoidance synchronization*), donde las solicitudes no se aceptan a menos que puedan ser satisfechas. Estas condiciones son expresadas como guardas.

Es conveniente destacar que el número de elementos de un recurso necesitados por un proceso puede, como resulta lógico, expresarse como parámetro en la llamada a un agente (activo o pasivo). A este respecto es preciso aclarar que el Ada no permite que un parámetro de una llamada a una *entry* pueda aparecer dentro de una condición de guarda.

#### 6.8.1.1 Tipo de servicio requerido

En un monitor u objeto protegido pueden agruparse distintos *procedimientos* para acceder a los recursos. Cada entrada o procedimiento puede estar orientada a dar el acceso a un recurso concreto. La semántica de los lenguajes en muchas ocasiones no especifica cual va a ser el orden de atención de las solicitudes y esto puede tener importancia. En Ada mediante el atributo *Count* es posible saber el número de tareas pendientes en una *entry* ya sea de una tarea o de un objeto protegido. Con esta información se puede dar prioridad a una solicitud frente a otra examinando el valor de este atributo en la condición de guarda o barrera.

### 6.8.1.2 Orden de la solicitud

Habitualmente el orden de atención de las solicitudes recibidas es del tipo FIFO, en el Ada esto es así dentro de una misma entry o subprograma protegido. El lenguaje no especifica cuál será la alternativa escogida si hay varias disponibles. Esta arbitrariedad introduce indeterminismo, nuevamente el atributo *Count* nos ayuda a fijar el orden de una solicitud, también se puede recurrir a familias de prioridades.

### 6.8.1.3 Estado del servidor

Ciertas operaciones deben ser permitidas solamente cuando el servidor y los objetos que administra se encuentren en un determinado estado. Por ejemplo, un recurso puede estar asignado si se encuentra libre, o un productor puede introducir un elemento si el buffer no está lleno. Las guardas en el caso del Ada o las variables de condición en los monitores se ajustan a esta finalidad.

### 6.8.1.4 Parámetros de la petición

El orden de las operaciones de un servidor puede estar regulado por la información contenida en los parámetros de las solicitudes. Tales informaciones típicamente identifican el recurso y su cantidad. Por ejemplo, si una solicitud para un conjunto de recursos determinados encuentra que no hay bastantes disponibles el proceso llamador debe ser suspendido, cuando se vayan liberando recursos los procesos suspendidos se despertarán para ver si ahora se pueden ver satisfechas sus demandas. En Ada, con lo visto hasta ahora, se presenta un problema debido a que no se pueden examinar los parámetros de la petición en las guardas. Este problema se puede resolver de manera sencilla con la llamada a *requee* que estudiaremos próximamente.

### 6.8.1.5 Prioridad de las solicitudes

Habitualmente la política de colas en las *entrys* es de tipo FIFO, así por ejemplo se evitan problemas de inanición, pero algunos sistemas permites establecer diferentes criterios. En Ada mediante pragmas se puede cambiar la política de colas. Estos pragmas están definidos en el anexo de tiempo real.

### 6.8.1.6 Nombramiento asimétrico y seguridad

En los lenguajes donde hay una nominación simétrica u proceso servidor (u objeto protegido) siempre conoce la identidad del cliente al que está sirviendo. Cuando el nombramiento no es simétrico podemos tener poca seguridad a la hora de manejar los recursos. Podría ser interesante conocer la identidad del cliente para :

- Poder rechazar una solicitud y así evitar una condición de estancamiento (*deadlock)*.
- Garantizar que el recurso es liberado por el propio proceso que lo adquirió.

En Ada mediante el paquete de identificación de tareas Ada.Task.Identification se tiene el tipo Task_Id, hay un atributo para las *entrys* de tareas y objetos protegidos que es Allocated'Caller y permite obtener el Task:Id de la tarea llamadora.

```
package Recursos is
   type Rango_Solicitud is range 1..Max;
   procedure Asignacion (R : out Recurso ; Cantidad : Rango_Solicitud);
   procedure Liberacion (R : Recurso ; Cantidad : Rango_Solicitud);

end Recursos;

package body Recursos is


protected Controlador_Recursos is
   entry Solicitar (R : out Recurso; Cantidad : Rango_Solicitud)
   procedure Liberar (R : Recurso ; Cantidad : Rango_Solicitud);
private
   entry Asignar (R: out Recurso; Cantidad : Rango_Solicitud);
   Liberados : Rango_Solicitud := Rango_Solicitud´Last;
```

```
   Nuevos_Recursos_Liberados : Boolean := False;
   Intentar : Natural := 0;
end Controlador_Recursos;

protected body Controlador_Recursos is
   entry Solicitar (R: out Recurso ; Cantidad : Rango_Solicitud)when
Liberados > 0 is
   begin
      if Cantidad <= Liberados then
         Liberados := Liberados - Cantidad;  -- Asignar
      else
         requeue Asignar;
      end if;
   end Solicitar;

   entry Asignar (R : out Recurso; Cantidad : Rango_Solicitud)when
Nuevos_Recursos_Liberados is
   begin
      Intentar := Intentar - 1;
      if Intentar = 0 then
         Nuevos_Recursos_Liberados := False;
      end if;
      if Cantidad < = Liberados then
         Liberados := Liberados - Cantidad; -- asignar
      else
         requeue Asignar;
      end if;
   end Asignar;

   procedure Liberar(R: Recurso; Cantidad : Rango_Solicitud) is
   begin
      Liberados := Liberados + Cantidad;     -- liberar recursos
      if Asignar ´Count  > 0 then
         Intentar := Asignar'Count;
         Nuevos_Recursos_Liberados := True;
      end if;
   end Liberar;
end Controlador_Recursos;

procedure Asignacion(R: out Recurso; Cantidad:Rango_Solicitud) is
begin
   Controlador_Recursos.Solicitar(R,Cantidad);
end Asignacion;

procedure Liberacion(R: Recurso; Cantidad:Rango_Solicitud) is
begin
   Controlador_Recursos.Liberar(R,Cantidad);
end Liberacion;

end Recursos;
```

## 6.9 La facilidad requeue

La principal característica de *requeue* es el movimiento de una tarea de una cola de espera en una guarda a otra. Para poner un símil supongamos a una persona (tarea) que pretende entrar en una habitación (objeto protegido) que tiene una o más puertas (entradas con guardas). Si esa persona es capaz de flanquear una puerta puede ser conducida a otra diferente donde puede tener que esperar, esta sería la acción de *requeue.*

Ada permite el reencolamiento en entradas de tareas y de objetos protegidos. La nueva cola puede ser la misma de antes, otra diferente dentro del mismo objeto o tarea o bien una que se encuentre en una unidad diferente. Se permiten reencolamientos de una tarea a un objeto protegido y viceversa. La ejecución de una sentencia *requeue* termina con la ejecución de la entrada llamada originalmente.

Para ver el funcionamiento de esta llamada plantearemos un problema de control de recursos. Supongamos que de cierto recurso R disponemos de una cantidad finita de elementos. Cuando una tarea pretenda usar el recurso debe indicar cuantos elementos va a utilizar. Cuando haya acabado deberá liberarlos para que otras los puedan utilizar. Si no hay suficientes recursos disponibles para satisfacer la demanda de una tarea no se le otorgarán ninguno y la tarea esperará hasta que haya suficientes.

### 6.9.1 Semántica de requeue

La llamada a *requeue* no es una simple llamada a procedimiento. Si un subprograma P llama a otro Q tras la finalización de Q se continúa la ejecución en P. Si una entrada X llama con *requeue* a otra Y el control no pasa nuevamente a X. Una vez que finaliza la entrada en Y el control pasa a la tarea que había llamado a X.

La entrada llamada en la sentencia requeue (entrada objetivo) no debe tener parámetros, o si los tiene deben coincidir en número y tipo con los de la llamadora. Por ejemplo, en el caso anterior del asignador de recursos los parámetros de Asignar coinciden con los de Solicitar, debido a esto no es preciso dar los valores reales de los parámetros en la llamada. Está incluso prohibido cambiar el valor de los parámetros de una llamada a otra.

Es posible incorporar a la llamada *requeue* una cláusula opcional *with abort*. Cuando una tarea realiza una llamada a una *entry* permanecerá en ella hasta que sea atendida, a no ser que hubiera sido dentro de una sentencia *selec*t con alternativa *o*r o *else*. Otras opciones son que pueda ser eliminada con una sentencia abort o programar una Transferencia Asíncrona de Control (ATC) empleando una cláusula *with abort* con una *entry* o un *requeue*, por el momento no estudiaremos esta alternativa

**Ejemplos allocate.adb. Mediante un par de pragmas presentes al comienzo del código se puede escoger como deseamos que se organicen la tareas pendientes en las entradas.**

## 6.10 DRIVERS

Un programa controlador de dispositivo (*device driver*) es un *software* dedicado a controlar el acceso a un dispositivo *hardware*, tal como un disco, impresora, etc. El driver aísla al resto del programa de las peculiaridades del dispositivo, como son la localización de sus registros *hardware*, las direcciones de interrupción, etc. Típicamente, los programas no interactúan directamente con el *hardware*, sino que lo hacen con los drivers correspondientes. De este modo, los cambios en los dispositivos sólo afectarán a sus drivers. La programación de drivers de dispositivos requiere facilidades para acceder a los registros de los dispositivos y para el manejo de interrupciones.

Tradicionalmente, los drivers han sido escritos en lenguajes ensambladores, aunque el resto del programa estuviera escrito en un lenguaje de alto nivel. Esto es debido a que muchos de estos lenguajes no proporcionan facilidades para acceder al *hardware*.

Un dispositivo puede ser considerado como un procesador que ejecuta un proceso o tarea fija, y que opera en paralelo con otros elementos del sistema. Es lógico por tanto considerar que el dispositivo es una tarea *hardware*. El sistema puede ser entonces modelado como formado por varios procesos paralelos que necesitan sincronizarse y comunicarse. La sincronización será proporcionada por una interrupción

### 6.10.1 Especificaciones de representación

Las especificaciones de representación indican cómo los tipos y otras entidades del lenguaje pueden ser mapeadas sobre el hardware subyacente. Esta especificación es opcional. Se debe aplicar solamente por motivos de eficiencia, y para permitirnos hacer referencia a elementos de bajo nivel con la terminología normal del lenguaje de alto nivel, Ada en nuestro caso.

Ada proporciona cuatro clases de especificaciones de representación: especificación de atributos, de tipos enumerados, de tipos record, y de direcciones.

## 6.10.2 Definición de atributos

Permite fijar varios atributos de un objeto, tarea o subprograma: tamaño en bits de los objetos, espacio de almacenamiento reservado para una tarea.

Toma la forma:

```
for NOMBRE'ATRIBUTO use EXPRESION_SIMPLE;
ATRIBUTO:  SIZE, STORAGE_SIZE
```

P'SIZE: número de bits para almacenar un objeto del tipo o subtipo P, o un objeto P

```
type BYTE is range 0..255;
for BYTE'SIZE use 8;
```

Los objetos de tipo BYTE ocuparán sólo 8 bits. Esta especificación afectará no solo a los objetos de tipo BYTE, sino también a los arrays y registros que contengan componentes de este tipo.

Por razones de direccionabilidad, los objetos simples ocuparán N bytes, siendo N un nº entero positivo.

El espacio (en unidades de almacenamiento) para almacenar tareas y grupos (pools) de objetos se indica mediante el atributo STORAGE_SIZE. La constante STORAGE_UNIT, definida en el package SYSTEM, contiene el número de bits de la unidad de almacenamiento. Valor típico: 8 (1 byte).

Si por ejemplo queremos asegurar que el espacio para un objeto de `type CELL_PTR is access CELL` acomode 500 objetos de tipo CELL podemos escribir

```
for CELL_PTR'STORAGE_SIZE use 500*CELL'SIZE/System.STORAGE_UNIT;
```

El espacio para tareas se fija mejor colocando el pragma STORAGE_SIZE(Tamaño) en la especificación de la tarea:

```
task type T(ESPACIO:Integer) is
   pragma STORAGE_UNIT(ESPACIO);
   ...
end T;
```

Para un prefijo X que denote un objeto o unidad de programa, X'ADDRESS indica la dirección del primero de los elementos direccionables asociado a X.

```
ONE_WORD: constant:=16;   --bits en una palabra
ONE_BYTE: constant:=8;      -- bits en un byte
type MY_INT is range -100..100;
for MY_INT'SIZE use ONE_BYTE;
type REG is
record
   ...
end record;
for REG'SIZE use ONE_WORD;
```

### 6.10.2.1 Especificación de tipos enumerados

Indica los códigos internos para las literales del tipo enumerado. Estos códigos internos deben tener valores distintos (únicos). Pueden ser valores no contiguos, aunque sí deben estar ordenados. No influyen en el uso de los atributos POS, VAL, PRED y SUCC.

Un uso común de la representación de tipos enumerados es el de dar nombres a secuencias de bits, utilizando identificadores Ada regulares. Esto permite hacer referencia a construcciones de bajo nivel mediante expresiones de alto nivel, lo que aumentará la legibilidad de las operaciones.

Supongamos un dispositivo periférico (un controlador de disco, por ejemplo) que se maneja mediante una serie de comandos definidos como secuencias de bits. Estos comandos podrían ser representados de forma más legible:

```
type COMMAND is (HOME, SEEK, STEP, WHERE, READ, WRITE);
```

```
          for COMMAND'SIZE use 6;   -- bits
          for COMMAND use (HOME    => 8#00#,
                           SEEK  => 8#04#,
                           STEP  => 8#06#,
                           WHERE => 8#10#,
                           READ  => 8#50#,
                           WRITE => 8#70#);
```

Cuando utilizamos uno de los comandos, estamos realmente manejando una cadena particular de bits.

### 6.10.2.2 Representación de tipos RECORD

Podemos describir el registro en términos del número de unidades de almacenamiento, más la localización de cada componente del registro dentro de esas unidades de almacenamiento. Una utilización típica de esta especificación es la descripción a alto nivel de las estructuras hardware direccionables de un dispositivo.

Supongamos un registro de estado de un dispositivo I/O con la siguiente estructura:

bits

- 12-15 Errores (0001: READ_ERROR, 0010: WRITE_ERROR, 0100: POWER_FAIL, 1000: OTHER
- 11 Busy (ocupado)
- 8-10 Nº unidad seleccionada (caso de varios dispositivos a controlar)
- 7 Done. (I/O completo)
- 6 Activa/desactiva interrupciones
- 3-5 Reservados para uso futuro
- 1-2 Función a realizar por el dispositivo (01: READ, 10: WRITE, 11: SEEK)
- 0 Activo/inactivo

```
     ONE_WORD: constant:=16;   -- bits
     WORD: constant:=2;          -- nº de unidades de almacenamiento en una
palabra
                           -- suponiendo que SYSTEM.STORAGE_UNIT es 8 (1 byte)
     type TIPO_ERROR is (READ_ERROR, WRITE_ERROR, POWER_FAIL, OTHER);
     type FUNCION is (READ, WRITE, SEEK);
     type UNIDAD is range 0..7;
     type REGISTRO is
     record
        ERROR: TIPO_ERROR;
        BUSY: BOOLEAN;
        UNIT: UNIDAD;
        DONE: BOOLEAN;
        IENABLE: BOOLEAN;
        FUNC: FUNCION;
        DENABLE: BOOLEAN;
     end record;
     for REGISTRO'Bit_Order use Low_Order_First;
          -- tipo Bit_Order definido en el package System

     for FUNCION use (READ => 2#01#, WRITE => 2#10#, SEEK => 2#11#);
     for TIPO_ERROR use (READ_ERROR => 2#0001#, WRITE_ERROR => 2#0010#,
                         POWER_FAIL => 2#0100#, OTHER => 2#1000#);
     for REGISTRO use
        record
       -- componente at posicion range primer_bit..ultimo_bit;
```

```
-- posicion, primer_bit y ultimo_bit: valores estáticos enteros
    DENABLE at 0*WORD range 0..0;
    FUNC at 0*WORD range 1..2;
    IENABLE at 0*WORD range 6..6;
    DONE at 0*WORD range 7..7;
    UNIT at 0*WORD range 8..10;       -- o bien UNIT at 1 range 0..2;
    BUSY at 0*WORD range 11..11;    -- o bien BUSY at 1 range 3..3;
    ERROR at 0*WORD range 12..15; -- o bien ERROR at 1 range 4..7;
 end record;
```

### 6.10.2.3 Especificación de direcciones

```
D: constant address := to_address(8#177566#);
CSR: REGISTRO;
for CSR'ADDRESS use D;
```

Esta especificación puede ser utilizada para indicar la dirección absoluta del primer elemento de un objeto, como en el ejemplo, o para indicar la dirección de comienzo de una unidad de programa (subprograma, package o task). En este caso, será la dirección de comienzo del código del cuerpo de la unidad.

#### Ejercicio

Considerar el registro de estado de una controladora de disco:

| **ERROR** | **HARD_ERROR** | **...** | **RDY** | **IDE** | **EXT** | **EXT** | **MD** | **MD** | **MD** | **GO** |
|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | ... | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

ERROR, HARD_ERROR, RDY (Ready), IDE (Interrupt enable) y GO son variables con 2 estados, y se representan con los flags Off (0), On(1).

MD (Modo de operación) ocupa 3 bits, pero solo tiene 4 operaciones: Reset, Write, Read, Seek, con códigos internos 0,1,2,4.

EXT (Memory extension bit) ocupa 2 bits, y puede tener los valores 0,1,2,3.

Escribir la declaración y representación del registro

**Package SYSTEM (Ada95)**

```
package System is
    pragma Preelaborate(System);

    type Name is implementation-defined-enumeration-type;
    System_Name : constant Name := implementation-defined;

    -- System-Dependent Named Numbers:

    Min_Int         : constant := root_integer'First;
    Max_Int         : constant := root_integer'Last;

    Max_Binary_Modulus    : constant := implementation-defined;
    Max_Nonbinary_Modulus : constant := implementation-defined;

    Max_Base_Digits    : constant := root_real'Digits;
    Max_Digits      : constant := implementation-defined;

    Max_Mantissa       : constant := implementation-defined;
    Fine_Delta      : constant := implementation-defined;

    Tick         : constant := implementation-defined;

    -- Storage-related Declarations:

    type Address is implementation-defined;
    Null_Address : constant Address;
    Storage_Unit : constant := implementation-defined;
    Word_Size    : constant := implementation-defined * Storage_Unit;
    Memory_Size  : constant := implementation-defined;

    -- Address Comparison:
    function "<" (Left, Right : Address) return Boolean;
    function "<="(Left, Right : Address) return Boolean;
    function ">" (Left, Right : Address) return Boolean;
    function ">="(Left, Right : Address) return Boolean;
    function "=" (Left, Right : Address) return Boolean;
    -- function "/=" (Left, Right : Address) return Boolean;
    -- "/=" is implicitly defined
    pragma Convention(Intrinsic, "<");
    ... -- and so on for all language-defined subprograms in this package

    -- Other System-Dependent Declarations:
    type Bit_Order is (High_Order_First, Low_Order_First);
    Default_Bit_Order : constant Bit_Order;

    -- Priority-related declarations (see D.1):
    subtype Any_Priority is Integer range implementation-defined;
    subtype Priority is Any_Priority range Any_Priority'First ..
         implementation-defined;
    subtype  Interrupt_Priority is  Any_Priority  range  Priority'Last+1  ..
              Any_Priority'Last;
    Default_Priority : constant Priority := (Priority'First +
                                  Priority'Last)/2;
    private
    ... -- not specified by the language
end System;
```

**OTROS PACKAGES RELACIONADOS (Ada95)**

```
package System.Storage_Elements is
   pragma Preelaborate(System.Storage_Elements);
   type Storage_Offset is range implementation-defined;
   subtype Storage_Count is Storage_Offset range 0..Storage_Offset'Last;
   type Storage_Element is mod implementation-defined;
   for Storage_Element'Size use Storage_Unit;
   type Storage_Array is array(Storage_Offset range <>) of aliased
     Storage_Element;
   for Storage_Array'Component_Size use Storage_Unit;

   -- Address Arithmetic:
   function "+"(Left : Address; Right : Storage_Offset) return Address;
   function "+"(Left : Storage_Offset; Right : Address) return Address;
   function "-"(Left : Address; Right : Storage_Offset)  return Address;
   function "-"(Left, Right : Address)  return Storage_Offset;
   function "mod"(Left : Address; Right : Storage_Offset) return
         Storage_Offset;

   -- Conversion to/from integers:
   type Integer_Address is implementation-defined;
   function To_Address(Value : Integer_Address) return Address;
   function To_Integer(Value : Address) return Integer_Address;

   pragma Convention(Intrinsic, "+");
   -- ...and so on for all language-defined subprograms declared in this

   --    package.
end System.Storage_Elements;


generic
   type Object(<>) is limited private;
package System.Address_To_Access_Conversions is
   pragma Preelaborate(Address_To_Access_Conversions);

   type Object_Pointer is access all Object;
   function To_Pointer(Value : Address) return Object_Pointer;
   function To_Address(Value : Object_Pointer) return Address;

   pragma Convention(Intrinsic, To_Pointer);
   pragma Convention(Intrinsic, To_Address);
end System.Address_To_Access_Conversions;
```

**Ejemplo de uso de estos packages**

```
with System, Text_IO, System.Storage_Elements;
with System.Address_to_Access_conversions;
use System, Text_IO, System.Storage_Elements;

procedure PRUREP2 is

   type BYTE is range 0..255;
   for BYTE'size use 8;
   type INT_15 is range 0..15;
   for INT_15'SIZE use 4;
```

```
        type LOGICO is (F,T);
        for LOGICO'SIZE use 1;
        for LOGICO use (F =>0, T=>1);

        type REGISTRO is
        record
           PRIORIDAD:INT_15;
           CERO,NEGATIVO,ACARREO,OVERF: LOGICO;
        end record;

        for REGISTRO use
        record
           PRIORIDAD at 0 range 0..3;
           CERO at 0 range 4..4;
           NEGATIVO at 0 range 5..5;
           ACARREO at 0 range 6..6;
           OVERF at 0 range 7..7;
        end record;

        for REGISTRO'SIZE use STORAGE_UNIT;

        package IIO is new Integer_IO(Integer); use IIO;
        package BIO is new Integer_IO(BYTE); use BIO;
        package LIO is new Enumeration_IO(LOGICO); use LIO;
        package QIO is new Integer_IO(INT_15); use QIO;
        package  CONV  is  new  system.address_to_access_conversions(BYTE);  use
CONV;

        X:BYTE;
        A:address;
        V:Object_Pointer;
        REG1:REGISTRO;

     begin
        put("prioridad (0..15)");
        get(REG1.PRIORIDAD);
        new_line;
        put("bit CERO (F,T)");
        get(REG1.CERO);
        new_line;
        put("bit NEGATIVO (F,T)");
        get(REG1.NEGATIVO);
        put("bit ACARREO (F,T)");
        get(REG1.ACARREO);
        new_line;
        put("bit OVERFLOW (F,T)");
        get(REG1.OVERF);
        new_line;
        put_line("OVERF(bit7)  ACARREO(bit6)  NEG(bit5)  CERO(bit4)  PRI(bits0-
3)");

        put("unidad de almacenamiento (bits):");
        iio.put(storage_unit);
        new_line;

        put("direccion de la variable X:");
        iio.put(item=>integer(to_integer(x'address)),width=>10,base=>16);
        new_line;
```

```
   put("direccion de REG1:");
   iio.put(integer(to_integer(REG1'address)),10,16);
   new_line;

   put("tamanio de REG1:");
   iio.put(REG1'size);
   new_line;

   A:=REG1'address;
   V:=To_Pointer(A); -- puntero a REG1
   X:=V.all; -- X = contenido del byte que representa REG1

   put("contenido de X = contenido del registro = ");
   put(X);
   new_line;
   put("En binario = ");
   put(X,10,2);
   new_line;

end PRUREP2;
```

# 6.11 Interfaz con otros lenguajes

Un programa en Ada puede utilizar código escrito en otro lenguaje. El siguiente ejemplo utiliza una función escrita en C, llamada c_double.

```
function DOBLE(N: in Integer) return Integer is
   function CDOBLE(N: in Integer) return Integer;
   pragma IMPORT(C, CDOBLE, "c_double");
begin
   return CDOBLE(N);
end DOBLE;

with Text_IO, DOBLE;
use Text_IO;
procedure EJEMPLO is
   package IIO is new Integer_IO(Integer); use IIO;
begin
   put("El doble de 3 es ");
   put(DOBLE(3));
   new_line;
end EJEMPLO;
```

Pragma IMPORT se utiliza para indicar al compilador que se debe utilizar una rutina externa escrita en C. La función C correspondiente será

```
int c_double(int n)
{
   return n+n;
}
```

En este ejemplo no se requiere conversión de tipos porque la representación del tipo Integer es la misma en Ada y en C. Si hiciera falta conversión, habría que utilizar las funciones to_ada y to_c del package Interfaces.C.

**Ejemplo de acceso a los puertos I/O del PC con gnat3.05**

Para el acceso a puertos HW del PC en programas Ada (Gnat 3.05) se ha seguido el siguiente procedimiento:

En la especificación del paquete IO_Ada se define un procedimiento para poner un dato en un puerto (procedure Sacar) y una función que devuelve el dato leído de un puerto (function Entrada). En el cuerpo del paquete se definen, dentro de los anteriores, otros ocultos junto con el pragma IMPORT que establece la correspondencia entre éstos y las rutinas predefinidas de C "outportb" e "inportb", para escribir y leer datos, respectivamente.

Como las rutinas de C a utilizar son predefinidas, para construir el programa ejecutable basta compilar el paquete con el resto de unidades + gnatbind +gnatlink, sin ninguna peculiaridad (abreviadamente gnatmake principal).

```
     with System.Unsigned_Types;                        -- SOPORTA TIPOS DE DATOS

     package IO_Ada is
     -- PAQUETE PARA EL ACCESO A PUERTOS HW DEL PC
        subtype byte is integer range 0..255;         -- RANGO 8 BITS
        procedure           Sacar(puerto:System.Unsigned_Types.Short_Unsigned;
dato:byte);
        function  Entrada(puerto:System.Unsigned_Types.Short_Unsigned)  return
byte;
     end IO_Ada;

     package body IO_Ada is

      procedure  Sacar(puerto:System.Unsigned_Types.Short_Unsigned;  dato:byte)
is
           procedure      Sacar_Oculto(p:System.Unsigned_Types.Short_Unsigned;
d:byte);
           pragma  Import  (C,  Sacar_Oculto,  "outportb");     -- LLAMADA RUTINA
EN C
      begin
          Sacar_Oculto(puerto,dato);
      end Sacar;
      function   Entrada(puerto:System.Unsigned_Types.Short_Unsigned)   return
byte is

           function     Lectura_Oculta(p:System.Unsigned_Types.Short_Unsigned)
return        byte;
           pragma  Import(C,  Lectura_Oculta,  "inportb");     -- LLAMADA FUNCION
EN C
      begin
          return Lectura_Oculta(puerto);
      end Entrada;

     end IO_Ada;
```

**EJERCICIO.** Ejemplo del uso de puertos. Temporización y generación de sonidos.

El programa ha de desarrollar la escala cromática a partir de DO4.

El canal 2 del chip temporizador 8253 (8254) está conectado al altavoz del ordenador, produciendo sencillas señales de onda cuadrada que se traducen en sonidos. El puerto de este canal, de dirección 42h, conduce a un registro de 8 bits que envía y recibe datos por el canal. Cuando se programa un canal se envía un valor de 2 bytes a través del puerto, con el byte bajo en primer lugar. El número enviado pasa a un registro contador, que se ve decrementado en 1 cada vez que llega al canal un pulso proveniente del reloj del sistema. Cuando este número llega

a 0 el canal produce un pulso de salida. Así, el valor de la frecuencia del sonido a emitir será igual a la frecuencia del reloj (1193180 Hz) dividida por el valor enviado al canal 2.

El sonido se activa poniendo a 1 los 2 bits menos significativos del puerto del altavoz, de 8 bits y de dirección 61h, y se desactiva volviéndolos a poner a 0.

Para hacer una escala cromática a partir de DO4 (f=261 Hz) hay que ir semitono a semitono hasta llegar a DO5. Para ir al semitono siguiente, la frecuencia se incrementa en 22 Hz.

El programa deberá por tanto

1. Activar el altavoz.
2. Tener un bucle en el que se incremente la frecuencia 14 veces. En cada vuelta, se calcula el valor correspondiente del contador, y se manda al temporizador. Antes de pasar a la nota siguiente se hace un retardo para poder apreciar la variación de los sonidos.
3. Desactivar el altavoz.

Siguiendo con el Ada95 de gnat, si las rutinas en C a utilizar no son predefinidas, sino elaboradas por el programador en un fichero.c, se debe compilar dicho fichero mediante gcc -c fichero.c, compilar el resto de unidades (gcc -c principal.adb, etc.), y luego ejecutar gnatbind -x principal.ali y gnatlink principal.ali fichero.c

**Ejemplo**. El siguiente procedure utiliza el package RAW_IO para leer y escribir caracteres de modo inmediato. Este paquete, a su vez, utiliza unas funciones en C, almacenadas en el fichero fichero.c:

```
with Text_IO, RAW_IO;
use Text_IO, RAW_IO;

procedure PRUEBA is
   C:character;

begin
   put_line("Escribe caracteres, . para acabar");
   READKEY(C);
   while C/='.' loop
      PUTKEY(C);
      READKEY(C);
   end loop;
   new_line;
   put_line("FIN");
end PRUEBA;
```

RAW_IO exporta un procedure READKEY, que devuelve el carácter pulsado, y un procedure PUTKEY quemuestra en pantalla el carácter pasado como parámetro:

```
package RAW_IO is
   procedure READKEY(CH: out Character);
   procedure PUTKEY(CH: in Character);
end RAW_IO;
```

RAW_IO utiliza dos funciones C escritas por el usuario, y almacenadas en fichero.c. Utiliza también las funciones to_ada y to_c del package Interfaces.C para conversión de tipos entre C y Ada:

```
with Interfaces.C; use Interfaces.C;
package body RAW_IO is
   procedure READKEY(CH: out Character) is
      function GET_CHAR return Char;
```

```
         pragma IMPORT(C,GET_CHAR,"c_get_char");
      begin
         CH:=to_ada(GET_CHAR);
      end READKEY;

   procedure PUTKEY(CH: in Character) is
         procedure PUT_CHAR(Ch: in Char);
         pragma IMPORT(C,PUT_CHAR,"c_put_char");
      begin
         PUT_CHAR(to_c(CH));
      end PUTKEY;
   end RAW_IO;

   /* fichero.c  */
   #include <pc.h>
   #include <keys.h>
   #include <stdio.h>
   typedef enum {false, true} bool;
   char c_get_char();
   void c_put_char(char ch);

   /* Lectura y escritura inmediata de caracteres */
   char c_get_char()
   {
   /* Las teclas de funcion y las flechas devuelven dos caracteres
    * Por ejemplo, la flecha derecha devuelve char(0) 'M'
    *              la flecha izquierda devuelve char(0) 'K'
    * el resto de las teclas devuelven un caracter
    */

      int c;
      static char the_ch;
      static bool prev_char = false;       /* Si true, the_ch es el segundo
*/
      if (prev_char) {                      /* caracter de la tecla pulsada
*/
         prev_char=false;
         return the_ch;
      }
      c=getkey();              /* funcion especifica de djgpp C para PC   */
      if (c & 0x100) {                     /* tecla de funcion o flecha   */
         prev_char=true;
         the_ch=(char) (c & 0xFF);
         return (char) 0;
      }
      return (char) (c & 0xFF);           /* cualquier otra tecla         */
   }
   void c_put_char(char ch)
   {
      fputc(ch, stdout);
      fflush(stdout);
   }
```

Para construir el ejecutable PRUEBA.EXE:

a) Compilar las tres unidades

```
gcc -c fichero.c
gcc -c raw_io.adb
```

gcc -c prueba.c

b) Ejecutar el binder

gnatbind -x prueba.ali

c) Linkar

gnatlink prueba.ali fichero.o

# 6.12 Interrupciones

El papel de las interrupciones en el control de I/O es muy importante. Permiten que las operaciones I/O sean llevadas a cabo asíncronamente, evitando así '*busy waiting*', o un chequeo constante al estado de los dispositivos.

Cuando ocurre una interrupción, el estado actual del procesador debe ser almacenado, y debe activarse la rutina de servicio apropiada. Una vez que la interrupción ha sido atendida, el proceso inicial continúa su ejecución. Alternativamente, como consecuencia de la interrupción, puede ser seleccionado un nuevo proceso para ejecución.

El estado del proceso en ejecución incluye:

- Valor del registro contador de programa, es decir, dirección de la siguiente instrucción.
- Información acerca de: modo de procesamiento, prioridad, interrupciones permitidas,...
- Contenido de los registros.

Los dispositivos difieren en la forma en que deben ser controlados. Consecuentemente, requerirán diferentes rutinas de manejo de las interrupciones. Uno de los mecanismos para identificar el dispositivo que genera la interrupción es el llamado vectorizado (**interrupciones vectorizadas**). Consiste en un conjunto de posiciones de memoria, habitualmente contiguas, llamadas vectores de interrupción, y un mapeado de las direcciones *hardware* de los dispositivos en los vectores de interrupción. Estos deben estar asociados con las rutinas de servicio de las interrupciones, es decir, cada vector debe contener o hacer referencia a la dirección de la rutina correspondiente.

## 6.12.1 Manejo de interrupciones en Ada

En Ada95, los manejadores de interrupciones (*handlers*) se representan mediante procedures protegidos. La asociación del procedure manejador a la interrupción se puede hacer mediante uno de los siguientes pragmas:

a) **pragma Attach_Handler**(NOMBRE_MANEJADOR,EXPRESION);

Este pragma permite la asociación estática del manejador a la interrupción identificada por EXPRESION. La asociación se produce cuando se crea el objeto protegido que contiene el manejador.

b) **pragma Interrupt_Handler**(NOMBRE_MANEJADOR);

Permite la asociación dinámica del manejador a una o más interrupciones.

El package siguiente define el soporte para la identificación de interrupciones y la asociación dinámica de manejadores según lo especificado en el anexo *Systems Programming*:

```
with System;
with Interfaces.C.System_Constants;
package Ada.Interrupts is
   type Interrupt_ID is new integer
range 0 .. Interfaces.C.System_Constants.NSIG;
   type Parameterless_Handler is access protected procedure;
   function Is_Reserved (Interrupt : Interrupt_ID) return Boolean;
   function Is_Attached (Interrupt : Interrupt_ID) return Boolean;
   function Current_Handler (Interrupt : Interrupt_ID)
                             return Parameterless_Handler;
   procedure Attach_Handler(New_Handler : Parameterless_Handler;
```

```
                                Interrupt   : Interrupt_ID);
   procedure Exchange_Handler(Old_Handler : out Parameterless_Handler;
         New_Handler: Parameterless_Handler; Interrupt: Interrupt_ID);
   procedure Detach_Handler(Interrupt : Interrupt_ID);
   function Reference (Interrupt : Interrupt_ID) return System.Address;
end Ada.Interrupts;
with System;
--  This implementation-define package spec contains interrups
--  supported by DJGPP v2
--  Usable Interrupts for Ada.Interrupts and Interrupts Entries are
--  marked with "--  ***" in the following constant definition.
package Ada.Interrupts.Names is
   SIGABRT     : constant Interrupt_ID := 288;
   SIGFPE      : constant Interrupt_ID := 289;
   SIGILL      : constant Interrupt_ID := 290;
   SIGSEGV     : constant Interrupt_ID := 291;
   SIGTERM     : constant Interrupt_ID := 292;
   SIGALRM     : constant Interrupt_ID := 293;
   SIGHUP      : constant Interrupt_ID := 294;
   SIGINT      : constant Interrupt_ID := 295;
   SIGKILL     : constant Interrupt_ID := 296;
   SIGPIPE     : constant Interrupt_ID := 297;
   SIGQUIT     : constant Interrupt_ID := 298;
   SIGUSR1     : constant Interrupt_ID := 299;
   SIGUSR2     : constant Interrupt_ID := 300;
   SIGNOFP     : constant Interrupt_ID := 301;
   SIGTRAP     : constant Interrupt_ID := 302;
   SIGTIMR     : constant Interrupt_ID := 303;
   SIGPROF     : constant Interrupt_ID := 304;
end Ada.Interrupts.Names;
```

El objeto protegido que representa el manejador tiene el formato siguiente:

```
package MANEJADORES is
   protected MANEJADOR1 is
      procedure RESPUESTA;
      pragma Interrupt_Handler(RESPUESTA);
   end MANEJADOR1;
   ...
end MANEJADORES;

package body MANEJADORES is
   protected body MANEJADOR1 is
      procedure RESPUESTA is
         ...
with Ada.Interrupts, Ada.Interrupts.Names, MANEJADORES;
use Ada.Interrupts, Ada.Interrupts.Names, MANEJADORES;
procedure EJEMPLO is
   ...
begin
   ...
   Attach_Handler(MANEJADOR1.RESPUESTA'ACCESS, NOMBRE_INT);
   ...
```

# 6.13 BIBLIOGRAFÍA

[1] Burns, A., Wellings, A.; Sistemas de Tiempo Real y Lenguajes de Programación, 3ª ed.; Ed. Addison-Wesley (2003); caps 11, 12,13,15

[2] Burns, A., Wellings, A.; Concurrency in Ada; Ed. Cambridge University Press (1995); caps 2,7,11,12

[3] Ben-Ari, M.; Principles of Concurrent and Distributed Programming; Ed. Prentice-Hall (1990); cap 16

[4] Barnes, J.; Programming in Ada95; Ed. Addison-Wesley (1996); caps 21,22

[5] Smith, M.A.; Object Oriented Software in Ada95; International Thomson Computer Press (1996); caps 20,24

[6] Ada Reference Manual (ARM) (1995)