Parte I: Elementos del lenguaje Ada

1. Introducción a los computadores y su programación
2. Elementos básicos del lenguaje
3. Modularidad y programación orientada a objetos
4. Estructuras de datos dinámicas
5. Tratamiento de errores
6. Abstracción de tipos mediante unidades genéricas
7. Entrada/salida con ficheros
8. Herencia y polimorfismo
9. Programación concurrente y de tiempo real

9.1. Concurrencia

UNIVERSIDAD DE CANTABRIA

Muchos problemas se expresan de forma natural mediante varias actividades concurrentes:

- **sistemas de control atendiendo a múltiples subsistemas y eventos**
- **sistemas multicomputadores o distribuidos**
- **para el uso concurrente de múltiples recursos**

La concurrencia implica prever la sincronización:

- **para la utilización de recursos y datos compartidos**
- **y para el intercambio de eventos e información**

Notas:

UNIVERSIDAD DE CANTABRIA

Tradicionalmente, existen tres entornos de aplicación en los que se utilizan programas concurrentes:

- En aplicaciones que controlan entornos físicos o interaccionan con ellos, en los que múltiples eventos y subsistemas deben ser controlados a la vez. En estos sistemas una solución concurrente lleva a un mayor grado de modularidad y a un diseño más claro.
- En sistemas multicomputadores o distribuidos, donde el paralelismo físico se puede aprovechar al máximo ejecutando diferentes tareas en diferentes procesadores.
- En sistemas de cálculo convencional, si se desea utilizar múltiples recursos a la vez. Por ejemplo, en un sistema de ventanas, puede ser útil asociar tareas a determinadas clases de ventanas, de forma que puedan ejecutar diferentes actividades concurrentemente.

Los programas concurrentes representan el modelo más natural para resolver muchos problemas del mundo real que son de naturaleza concurrente.

También se pueden usar (a veces) aproximaciones no concurrentes para la programación de problemas de naturaleza concurrente, pero el problema es más difícil de abordar y, sobre todo, más difícil de modificar y extender, tal como se muestra en el ejemplo de control del automóvil que aparece a continuación.

Ejemplo: Control de un Automóvil

UNIVERSIDAD DE CANTABRIA

Actividades a controlar:

Medida de Velocidad
C=4 ms.
T=20 ms.
D=5 ms.

Control de Frenos ABS
C=10 ms.
T=40 ms.
D=40 ms.

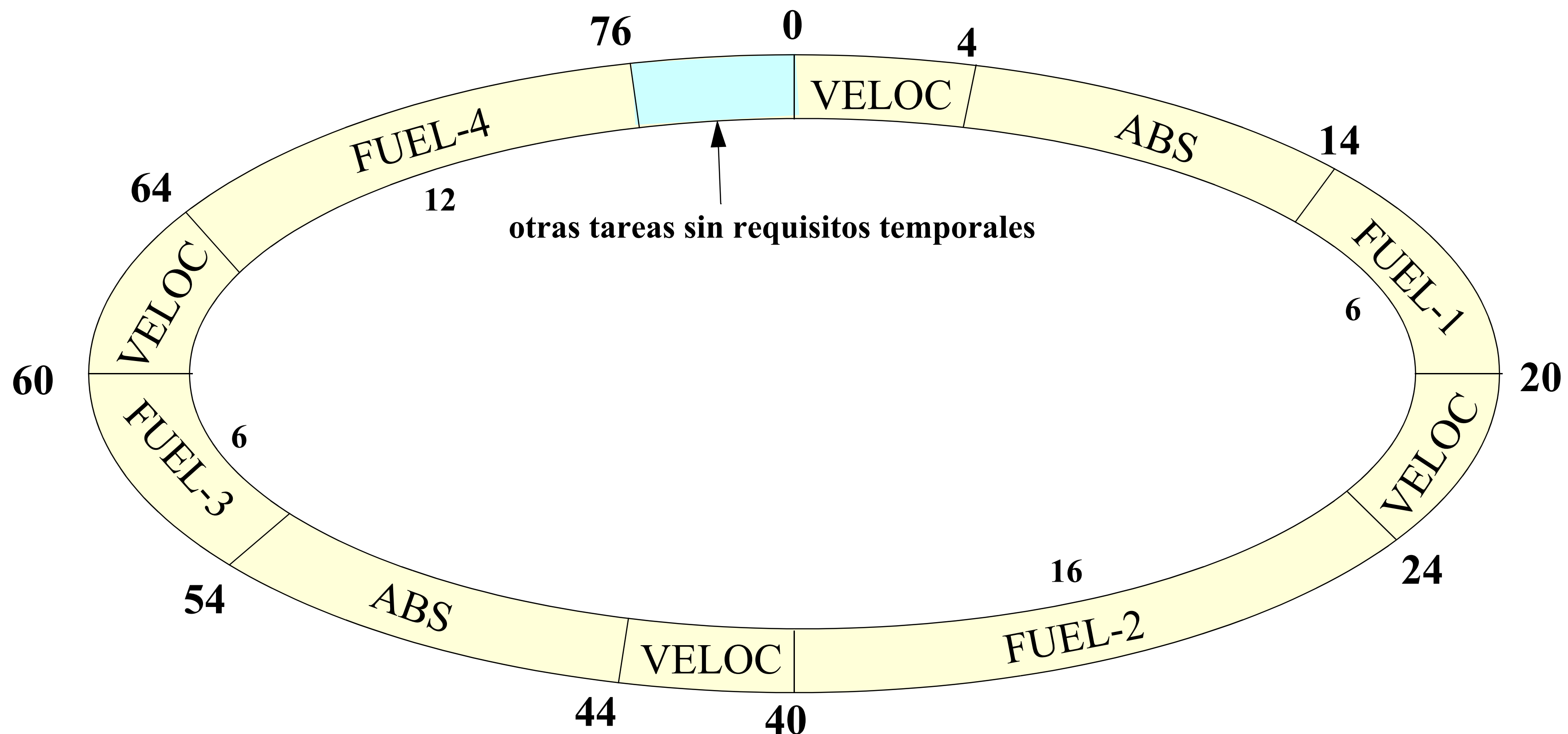
Control de Inyección
C=40 ms.
T=80 ms.
D=80 ms.

C = Tiempo de ejecución
T = Periodo
D = Plazo máximo de finalización

Solución no concurrente: cíclica

UNIVERSIDAD DE CANTABRIA

Una solución cíclica implica partir la actividad más larga en varias partes:



© J. Javier Gutiérrez, Michael González Harbour

Solución no concurrente: cíclica (cont.)

UNIVERSIDAD DE CANTABRIA

```
procedure Control_del_Automóvil is
   Tiempo : Integer:=0;
begin
   loop
      Acciones de Medida_de_Velocidad;
      if Tiempo=0 or Tiempo=40 then
         Acciones de Control de Frenos ABS;
      end if;
      if Tiempo=0 then
         Primera parte de control de inyección; --(<6 ms)
      elsif Tiempo=20 then
         Segunda parte de control de inyección; --(<16 ms)
      elsif Tiempo=40 then
         Tercera parte de control de inyección; --(<6 ms)
      else
         Cuarta parte de control de inyección;  --(<16 ms)
      end if;
      Espera hasta el próximo periodo; --mediante el reloj
```

Solución no concurrente: cíclica (cont.)

UNIVERSIDAD DE CANTABRIA

```
      Tiempo:=Tiempo+20;
      if Tiempo=80 then
          Tiempo=0;
      end if;
   end loop;
end Control_del_Automovil;
```

El programa generado depende del plan cíclico diseñado:

- para un plan diferente habría que trocear el código de otra manera además de modificar el propio programa que ejecuta el plan

Concurrencia en Ada

UNIVERSIDAD DE CANTABRIA

El lenguaje Ada soporta la programación de procesos concurrentes mediante las tareas (*tasks*).

Las tareas constan de especificación y cuerpo:

- la especificación de una tarea declara sus puntos de entrada (*entries*), si los hay
- el cuerpo define una actividad que se ejecuta independientemente; tiene declaraciones e instrucciones

Las tareas van en la parte de declaraciones de un módulo:

- se arrancan nada más ser visibles
- el módulo que las declara no termina si no han terminado todas sus tareas

Solución concurrente

UC UNIVERSIDAD DE CANTABRIA

```
procedure Control_del_Automóvil is

   task Medida_Velocidad; -- Tareas sin entries

   task Control_ABS;

   task Control_Inyeccion;

   task body Medida_Velocidad is
   begin
      loop
         Acciones de Medida_de_Velocidad;
         Esperar al próximo periodo (20 ms);
      end loop;
   end Medida_Velocidad;
```

Solución concurrente (cont.)

UNIVERSIDAD DE CANTABRIA

```
task body Control_ABS is
begin
   loop
      Acciones de Medida_de_Velocidad;
      Esperar al próximo periodo (40 ms);
   end loop;
end Control_ABS;

task body Control_Inyección is
begin
   loop
      Acciones de Control de Inyección;
      Esperar al próximo periodo (80 ms);
   end loop;
end Control_Inyección;

begin
   null; -- el programa principal es otra tarea
end Control_del_Automovil;
```


© J. Javier Gutiérrez, Michael González Harbour

9.2. Sincronización de espera

UNIVERSIDAD DE CANTABRIA

La sincronización de espera entre tareas en Ada se realiza mediante el mecanismo del *rendezvous*, o punto de entrada, o punto de encuentro entre dos tareas (*entry*):

- **Hay una tarea que llama, y otra que acepta el encuentro**
- **En la llamada se pueden pasar parámetros, como en un procedimiento**
- **La tarea que "llama" se queda esperando hasta que la tarea que acepta la llamada termine de ejecutarla**

Los puntos de entrada se declaran en la especificación

En el ejemplo que se muestra a continuación una tarea (consumidora) debe esperar a un dato que le suministra la otra tarea (productora)

© J. Javier Gutiérrez, Michael González Harbour

Ejemplo de sincronización de espera

```
-- Especificación de la tarea Consumidor
task Consumidor;

-- Especificación de la tarea Productor con un punto de entrada
-- que tiene un parámetro
task Productor is
     entry Espera_Dato(El_dato : out Dato);
end Productor;

-- Cuerpo de la tarea Consumidor: llama al punto de entrada de la
-- tarea Productor como si fuera un procedimiento
task body Consumidor is
     Copia_del_dato : Dato;
begin
     -- hace cosas
     Productor.Espera_Dato(Copia_Del_Dato);
     -- hace más cosas
end Consumidor;
```

Ejemplo de sincronización de espera

```
-- Cuerpo de la tarea Productor: el punto de entrada tiene su
-- punto de aceptación en el cuerpo
task body Productor is
    Dato_Producido : Dato;
begin
    -- calcula valor de Dato_Producido
    accept Espera_Dato (El_Dato : out Dato)
    do
         El_Dato:=Dato_Producido;
    end Espera_Dato;
    -- hace más cosas
end Productor;
```

Notas:

UNIVERSIDAD
DE CANTABRIA

Las tareas que declaran un punto de entrada en la especificación deben tener su correspondiente punto de aceptación *accept* en el cuerpo.

La tarea que llama a un punto de entrada de otra tarea se queda esperando hasta que el código asociado al *accept* finaliza (este código es opcional).

Esta es la base de la sincronización de tareas en Ada, pero esta sincronización es mucho más rica existiendo las siguientes posibilidades:

- Si una tarea declara varios puntos de entrada puede quedarse esperando a cualquiera de ellos con la sentencia *select*. Además, esta sentencia permite diferentes versiones temporizadas y con la opción de abortar la espera selectiva.
- Una tarea que acepta la sincronización con otra puede transferir esta sincronización a un punto de entrada de una tercera tarea con *requeue*.

9.3. Sincronización de datos

UNIVERSIDAD DE CANTABRIA

La sincronización de datos, para compartir información de manera mutuamente exclusiva, se realiza mediante objetos protegidos

El objeto protegido tiene una especificación con:

- *parte visible*: tiene funciones, procedimientos y puntos de entrada
- *parte privada*: contiene los datos protegidos

Con las funciones y procedimientos se garantiza un acceso mutuamente exclusivo a la información protegida:

- las operaciones de un objeto protegido se ejecutan en exclusión mutua respecto a sí mismas y a las demás operaciones protegidas

Ejemplo de sincronización de datos

UNIVERSIDAD DE CANTABRIA

```
type Coordenadas is record
    X,Y,Z : Float;
end record;


protected Datos_Avion is
    function Posicion return Coordenadas;
    procedure Cambia_Posicion (La_Posicion : Coordenadas);
private
    Posicion_Avion : Coordenadas;
end Datos_Avion;
```


© J. Javier Gutiérrez, Michael González Harbour

Ejemplo de sincronización de datos

UNIVERSIDAD DE CANTABRIA

```
protected body Datos_Avion is

    function Posicion return Coordenadas is
    begin
         return Posicion_Avion;
    end Posicion;

    procedure Cambia_Posicion (La_Posicion : Coordenadas) is
    begin
        Posicion_Avion:=La_Posicion;
    end Cambia_Posicion;

end Datos_Avion;
```

Notas:

UNIVERSIDAD DE CANTABRIA

En el ejemplo anterior se ha definido el objeto protegido `Datos_Avion`. También se podría haber definido un tipo de objeto protegido para poder crear posteriormente nuevos objetos:

```
protected type Tipo_Datos_Avion is
    function Posicion return Coordenadas;
    procedure Cambia_Posicion (La_Posicion : Coordenadas);
private
    Posicion_Avion : Coordenadas;
end Tipo_Datos_Avion;

Datos_Avion: Tipo_Datos_Avion;
```

Puntos de entrada en objetos protegidos

UNIVERSIDAD DE CANTABRIA

Los *puntos de entrada* proporcionan la misma protección que los procedimientos protegidos, pero además:

- permiten a una tarea *esperar*, hasta que se cumpla una determinada condición
- la evaluación de esta condición también está *protegida*

El ejemplo que se muestra a continuación implementa una cola con datos:

- utiliza el tipo abstracto de datos "Cola" (ver Tema 11)
- las operaciones de la cola (`Insertar`, `Extraer`, etc.) están protegidas
- la operación de `Extraer` hace que la tarea espere hasta que haya datos disponibles

Notas:

UNIVERSIDAD DE CANTABRIA

La sincronización de datos es necesaria para garantizar que la modificación o lectura de datos compuestos es consistente. Si por ejemplo una tarea está modificando un dato compuesto, y cuando sólo ha modificado una parte es interrumpida por otra tarea, esta segunda puede leer un dato inconsistente.

La solución es la sincronización para acceso mutuamente exclusivo. En Ada esto se logra con los objetos protegidos. Los objetos protegidos proporcionan acceso mutuamente exclusivo mediante funciones, procedimientos o puntos de entrada protegidos. Los procedimientos protegidos funcionan de modo que no pueden ser interrumpidos por otro procedimiento o función protegido del mismo objeto.

Los puntos de entrada, además de la protección equivalente a la de un procedimiento protegido, permiten que la tarea que los llama espere a que se cumpla una determinada condición. Esta condición se evalúa de forma protegida, es decir mutuamente exclusiva. Mientras la tarea espera, otras tareas pueden ejecutarse.

Ejemplo con punto de entrada

UNIVERSIDAD DE CANTABRIA

```
with Colas, Elementos;
package Cola_Protegida is

    subtype Dato is Elementos.Elemento;

    protected Cola is
       procedure Inserta (El_Dato : Dato);
       entry Extrae (El_Dato : out Dato); -- si no hay datos, espera
       procedure Haz_Nula;
    private
       La_Cola : Colas.Cola;
    end Cola;

end Cola_Protegida;
```

Ejemplo con punto de entrada (cont.)

UNIVERSIDAD DE CANTABRIA

```
package body Cola_Protegida is
    protected body Cola is
       procedure Inserta (El_Dato : Dato) is
       begin
            Colas.Inserta(El_Dato,La_Cola);
       end Inserta;

       entry Extrae (El_Dato:out Dato)
           when not Colas.Esta_Vacia(La_Cola) is
       begin
            Colas.Extrae(El_Dato,La_Cola);
       end Extrae;

       procedure Haz_Nula is
       begin
            Colas.Haz_Nula(La_Cola);
       end Haz_Nula;
    end Cola;
end Cola_Protegida;
```

9.4. Programación de Tiempo Real

UNIVERSIDAD DE CANTABRIA

El Ada soporta la programación de sistemas empotrados y de sistemas de tiempo real, mediante los siguientes mecanismos:

- Representación del hardware
- Interrupciones
- Gestión del tiempo
- Prioridades
- Sincronización libre de inversión de prioridad

Estos mecanismos están descritos en anexos opcionales:

- Anexo de programación de sistemas
- Anexo de sistemas de tiempo real

© J. Javier Gutiérrez, Michael González Harbour

Notas:

UNIVERSIDAD DE CANTABRIA

Un computador empotrado es aquel que forma parte de un sistema más grande. Por ejemplo, el computador que controla un robot o un avión. Generalmente son computadores con un hardware restringido en algunos aspectos (por ejemplo sin disco duro) y ampliado en otros aspectos de relación con el entorno (convertidores A/D y D/A, entrada/salida digital, etc.). Las necesidades especiales de programación se derivan precisamente de la necesidad de acceder directamente al hardware y de responder a interrupciones externas.

Los sistemas de tiempo real son aquellos en los que no sólo importa la realización de unos determinados cálculos, sino el instante en el que se produce la salida de los resultados de estos cálculos. Ello se debe a que el sistema de tiempo real interacciona con un entorno cambiante con el tiempo.

Normalmente, un sistema de tiempo real debe realizar diversas actividades de forma concurrente, y cada una de estas actividades presenta unos requerimientos de tiempo de respuesta que es preciso cumplir. Ello implica la necesidad de disponer de facilidades de programación tales como gestión del tiempo, poder dar más prioridad a aquellas tareas más urgentes, etc.

Muchas de las facilidades que presenta el lenguaje Ada para sistemas empotrados y de tiempo real son opcionales, ya que se hallan contenidas en Anexos del lenguaje.

9.5. Representación del Hardware

UNIVERSIDAD DE CANTABRIA

En muchos sistemas los programas deben interaccionar directamente con el hardware

Muchas veces se utiliza para ello lenguaje ensamblador

El Ada proporciona mecanismos para establecer la correspondencia entre estructuras lógicas y representaciones físicas del hardware. Por ejemplo:

- asociación entre campos de registros y posiciones de bits
- posicionamiento de datos en direcciones de memoria concretas
- control de optimización

Notas:

UNIVERSIDAD DE CANTABRIA

En general, si se programa en Ada no es necesario el uso del lenguaje ensamblador para acceder al hardware. El Ada permite:

- Expresar la correspondencia entre los campos de un registro y las posiciones de determinados bits en una palabra de la memoria. Está especialmente indicado para acceder a los registros de control y estado de dispositivos hardware.
- Es posible expresar la posición de memoria concreta en la que se pueden colocar una o más variables del programa. Esto permite acceder directamente a dispositivos hardware que están mapeados en memoria.
- Cuando los dispositivos hardware no están mapeados en memoria, el Ada permite expresar operaciones de entrada/salida de bajo nivel, para poder acceder a estos dispositivos a través del bus de entrada/salida.
- En Ada 95 se permite indicar al computador que determinadas variables corresponden a memoria física, bien porque son dispositivos hardware o bien porque son posiciones de memoria que van a ser compartidas en un multiprocesador. Esto permite al optimizador del programa saber que estas variables no se pueden optimizar.

9.6 Interrupciones

UNIVERSIDAD DE CANTABRIA

En los sistemas empotrados es preciso atender a las interrupciones provocadas por el hardware externo.

En Ada 95 se permite especificar un procedimiento protegido como elemento de manejo de interrupciones.

- **en este modelo, el hardware invoca directamente la ejecución del procedimiento protegido**
- **sólo funciona si el sistema operativo lo permite**

Notas:

UNIVERSIDAD DE CANTABRIA

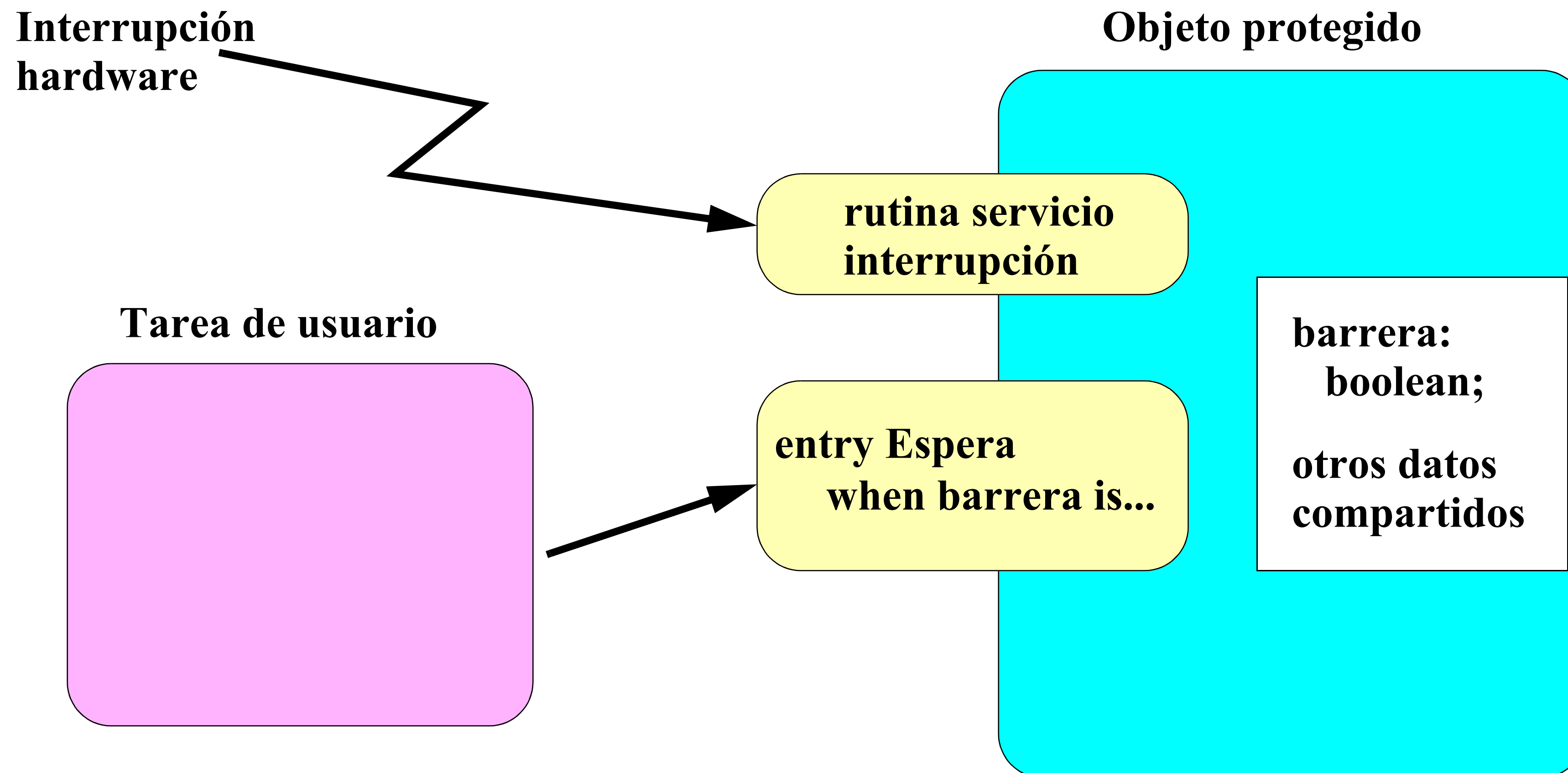
Un requisito imprescindible de la programación de sistemas empotrados es el poder atender a las interrupciones que llegan a un sistema, generadas por el hardware externo.

El Ada 95 define como mecanismo para manejo de interrupciones un procedimiento protegido.

En este caso, cuando el hardware provoca una interrupción, el sistema se encarga de llamar de forma automática al procedimiento protegido que se ha asignado a esa interrupción.

Esto sólo funciona si el sistema operativo lo permite. En los sistemas operativos protegidos, como Linux o Windows, no se permite a la aplicación gestionar interrupciones. En los sistemas operativos específicos para sistemas empotrados, habitualmente sí.

Ejemplo: Tarea que espera a una interrupción



Notas:

UNIVERSIDAD DE CANTABRIA

En el ejemplo de arriba hay un objeto protegido que tiene un procedimiento y un punto de entrada (*entry*):

- El procedimiento que hace las veces de rutina de servicio de interrupción se ha declarado para que atienda a una interrupción hardware, usando el pragma **`Attach_Handler`**. Cada vez que llega la interrupción se ejecuta este procedimiento. Lo que hace es poner la barrera a True; además puede hacer más cosas, como interaccionar con el hardware de interrupción y preparar datos para la tarea.
- La *entry* sólo se puede ejecutar cuando la barrera es True, es decir, cuando la interrupción ha llegado y ha sido procesada. Debe poner la barrera a False. Además, puede compartir datos del objeto protegido con la rutina de servicio de interrupción, y pasárselos a la tarea que llame a esta *entry*.

Las tareas de usuario pueden llamar a la *entry* para esperar a que llegue la interrupción. Cuando llega, la *entry* se ejecuta (porque la barrera se pone a True). Por medio de los parámetros de la *entry* la tarea puede pasar datos al objeto protegido, y obtener datos de él.

9.7 Gestión del Tiempo

UNIVERSIDAD DE CANTABRIA

El Ada permite diversas formas de manejo del tiempo:

- Package `Ada.Calendar`, con funciones para saber la hora, fecha, etc.
- Package `Ada.Real_Time` para acceder a un reloj monótono, que se incrementa siempre
- Instrucciones `delay until` y `delay`, que permiten dormir a una tarea hasta que transcurra el instante o intervalo indicado
  - mientras una tarea está dormida, otras pueden ejecutar

© J. Javier Gutiérrez, Michael González Harbour

Notas:

UNIVERSIDAD DE CANTABRIA

El package **Ada.Calendar** permite acceder a procedimientos y funciones para consultar o cambiar la fecha y hora. Presenta el inconveniente de que la fecha y la hora normalmente dependen de factores sociales tales como el horario invierno/verano, etc.

Con objeto de que las aplicaciones de tiempo real puedan disponer de un reloj que se incrementa siempre, se define el package **Ada.Real_Time** que permite saber la hora (desde un instante arbitrario). De esta forma se dispone de un reloj fiable para las aplicaciones de tiempo real.

Las instrucciones **delay** y **delay until** permiten a una tarea suspenderse hasta que transcurra un determinado intervalo de tiempo, o se alcance un determinado instante de tiempo, respectivamente. Esta facilidad es de gran importancia en programas de tiempo real. Mientras la tarea se halla suspendida, otras tareas que estén activas (es decir no suspendidas) pueden ejecutar.

Paquete Ada.Calendar

UNIVERSIDAD DE CANTABRIA

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


© J. Javier Gutiérrez, Michael González Harbour

Paquete Ada.Calendar

UNIVERSIDAD DE CANTABRIA

```
   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;
   ...

end Ada.Calendar;
```


© J. Javier Gutiérrez, Michael González Harbour
24/ma/10

Ejemplo

UNIVERSIDAD DE CANTABRIA

```ada
-- Imprimir en pantalla el día, mes y año actuales,
-- y luego las horas y minutos

with Ada.Calendar, Ada.Text_Io;
use Ada.Calendar; use Ada.Text_Io;
procedure Muestra_Dia_Y_Hora is
   Instante : Time:=Clock;
   Hora     : Integer := Integer(Seconds(Instante))/3600;
   Minuto   : Integer := (Integer(Seconds(Instante))-Hora*3600)/60;
begin
   Put_Line("Hoy es "&Integer'Image(Day(Instante))&" del "&
      Integer'Image(Month(Instante))&" de "&
      Integer'Image(Year(Instante)));
   Put_Line("La hora es : "&Integer'Image(Hora)&
      ":"&Integer'Image(Minuto));
end Muestra_Dia_Y_Hora;
```

Notas:

UNIVERSIDAD DE CANTABRIA

El paquete `Ada.Calendar` contiene el tipo `Time`, que representa un instante de tiempo. Se puede combinar en operaciones con el tipo predefinido `Duration`, que representa un intervalo relativo de tiempo.

La función `Clock` permite obtener la fecha y hora actual, del tipo `Time`. Para obtener a partir de este dato el año, mes, día, o segundos dentro del día, (y al revés) existen también funciones en `Ada.Calendar`.

Tareas periódicas

UNIVERSIDAD DE CANTABRIA

Con la instrucción `delay until` es posible hacer tareas periódicas:

```
task body Periodica is
    Periodo : constant Duration:=0.050; -- en segundos
    Proximo_Periodo : Time := Clock;
begin
    loop
        -- hace cosas
        Proximo_Periodo:=Proximo_Periodo+Periodo;
        delay until Proximo_Periodo;
    end loop;
end Periodica;
```

Tareas periódicas

UNIVERSIDAD DE CANTABRIA

También se puede hacer lo mismo (*pero mal*) con la orden delay:

```
task body Periodica is
    Periodo : constant Duration:=0.050; -- en segundos
    Proximo_Periodo : Time := Clock;
begin
    loop
        -- hace cosas
        Proximo_Periodo:=Proximo_Periodo+Periodo;
        delay Proximo_Periodo-Clock;
    end loop;
end Periodica;
```

En este caso la tarea puede ser interrumpida por otra entre la lectura del reloj (con `Clock`) y la ejecución del `delay`.

- Esto provocaría un retraso mayor que el deseado.

9.8 Prioridades

UNIVERSIDAD DE CANTABRIA

El Ada utiliza el sistema de tareas expulsoras (o desalojantes) por prioridad fija (mayor número mayor prioridad):

- cuando una tarea de alta prioridad se activa, desaloja a otra tarea de menor prioridad que se esté ejecutando

A cada tarea se le puede asignar una prioridad con:

```
pragma Priority(13);
```

Esta prioridad se puede modificar, usando las operaciones del paquete `Ada.Dynamic_Priorities`

Mediante la planificación de tareas por prioridades fijas, es posible garantizar la respuesta en tiempo real de las tareas.

- La teoría RMA permite realizar el análisis del sistema

Notas:

UNIVERSIDAD DE CANTABRIA

El sistema de planificación expulsora de tareas por prioridad fija es muy comúnmente utilizado en la mayoría de los sistemas de tiempo real construidos en la práctica. En este tipo de planificación, el planificador de tareas, que es la parte del software encargada de decidir qué tarea es la que se va a ejecutar, ejecuta la tarea de más alta prioridad de todas las que estén activas. La expulsión implica que si una tarea está siendo ejecutada y en ese momento otra tarea de prioridad más alta se activa para ejecutar, la tarea de prioridad más alta "expulsa" a la de prioridad baja de la CPU, y pasa a ser ejecutada.

Para asignar una prioridad a una tarea se usa el pragma `Priority` en la especificación de la tarea. También se puede asignar prioridad al programa principal, con el mismo pragma.

En el Ada 95 se permite que la prioridad de las tareas cambie. Para ello se cuenta con el paquete `Ada.Dynamic_Priorities`, que tiene las operaciones `Set_Priority` y `Get_Priority`.

El rango de prioridades que utiliza un sistema para las tareas se encuentra entre `Priority'First` y `Priority'Last` (atributos aplicados sobre el subtipo `Priority`)

La teoría RMA (Rate Monotonic Analysis) ayuda a hacer una buena asignación de prioridades, y muestra cómo es posible predecir matemáticamente si el sistema va a cumplir todos sus requerimientos temporales. (Ver "A Practitioner's Handbook for Real-Time Analysis", por M.Klein, T. Ralya, B. Pollak, R. Obenza, y M. González, Kluwer Academic Pub., 1993).

Inversión de prioridad

UNIVERSIDAD DE CANTABRIA

En la sincronización de datos puede aparecer la inversión de prioridad:

- una tarea de alta prioridad puede verse forzada a esperar a que una de menor prioridad termine
- esto provoca retrasos enormes, inaceptables en sistemas de tiempo real

Para evitarla existe el protocolo de techo de prioridad, asociado a los objetos protegidos:

- a cada objeto protegido se le asigna un techo de prioridad
- con el `pragma Priority`
- debe ser igual o superior a las prioridades de las tareas que usan ese objeto protegido

Notas:

UNIVERSIDAD DE CANTABRIA

Cuando las tareas de un sistema de tiempo real se sincronizan pueden ocurrir inversiones de prioridad, que causan que una tarea de alta prioridad esté esperando a que otra tarea de baja prioridad termine.

Esto puede provocar que el tiempo de respuesta sea muy alto, y por tanto es inaceptable en sistemas de tiempo real.

Para evitar este problema, el Ada cuenta con un mecanismo especial, llamado el techo de prioridad, por el que las tareas modifican su prioridad al sincronizarse.

Para que funcione correctamente este mecanismo, debemos asignar a cada objeto protegido un techo de prioridad, usando el pragma `Priority`. Este techo debe calcularse de manera que sea igual o superior (mejor igual) a las prioridades de las tareas que pueden usar ese objeto compartido.