TEMA 1. INTRODUCCIÓN A LOS STR.

1.1. Definición de un sistema de tiempo real.
1.2. Características de los sistemas de tiempo real.
1.3. Tiempo compartido y tiempo real.
1.4. Planificación.
1.5. Sistemas críticos y no críticos.

1.1. Definición de un sistema de tiempo real.

Un **Sistema de Tiempo Real** (STR) puede definirse como (Young 1982):

> "Cualquier actividad de proceso de información o sistema que tiene que responder a estímulos generados externamente dentro de un **plazo especificado y finito**".

Por tanto, la corrección de un STR dependerá de:

1.- El **resultado lógico** del procesamiento (como en cualquier sistema informático).

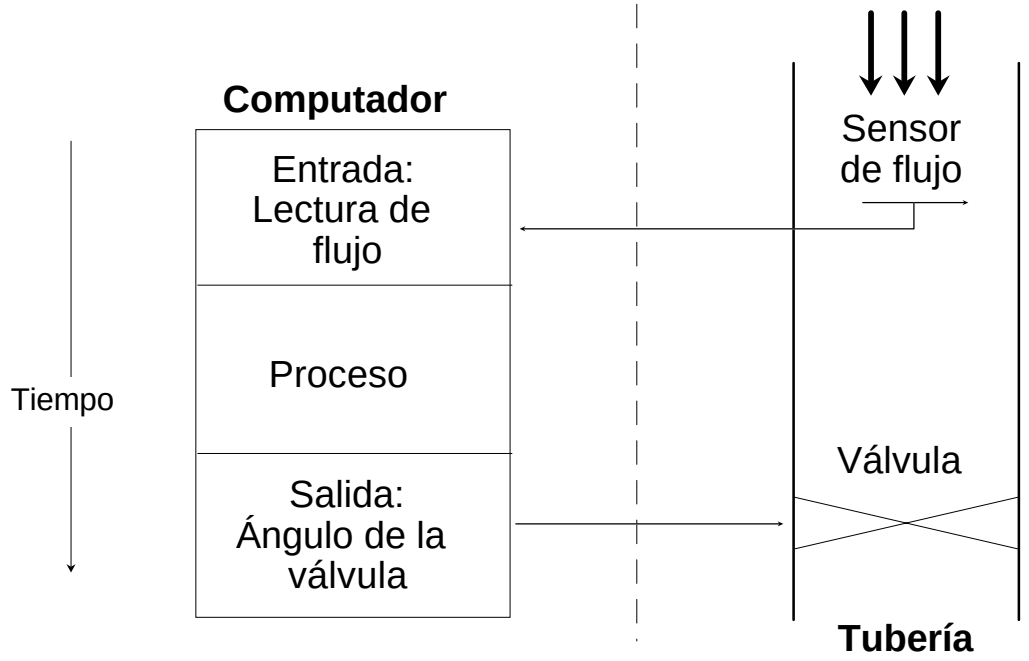
2.- Del **instante** en que se genera el resultado.

Computación de tiempo real ≠ Computación Rápida, sino que un STR ha de ser **predecible.**

Un STR forma parte de un sistema más grande (al cual controla), por lo que a veces se les denomina **Sistemas Empotrados** (embedded systems).

1.1. Definición de un sistema de tiempo real.

Ejemplo de STR:



1. El sensor es utilizado por el computador para detectar las variaciones en el flujo.

2. La respuesta en la válvula ha de ser lo suficientemente rápida para asegurar un flujo constante

1.2. Características de los STR.

• **Gran tamaño y complejidad**:

– La complejidad de una aplicación va ligada a la **variedad** de la misma, no al número de instrucciones.

– Variedad grande ⇒ la aplicación responde a un entorno diverso.

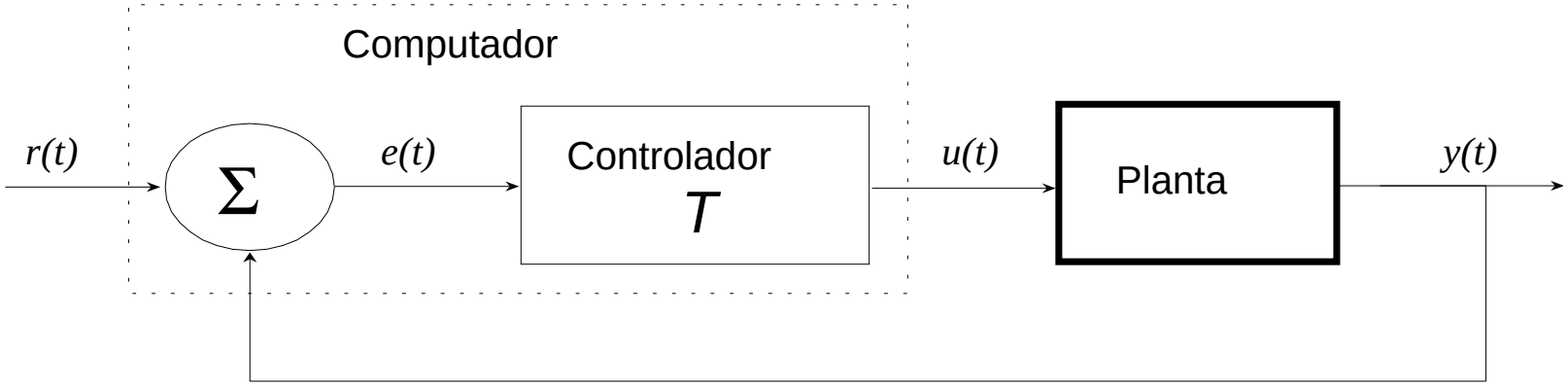
– Un tamaño grande es consecuencia de variedad.

– Los STR responden a eventos del mundo real.

– El entorno del STR es continuamente cambiante, por lo que la aplicación debe evolucionar continuamente ⇒ STR deben ser **extensibles**.

1.2. Características de los STR.

- **Manipulación de números reales:**



– El sistema físico controlado, la **planta**, produce *y(t)* dada *u(t)*.

– El objetivo del control es que *y(t)* sea lo más parecido a *r(t)* (esta última es una señal de referencia) ⇒ minimizar *e(t)* (donde, *e(t) = r(t) - y(t)*).

– Para ello es necesario aplicar a la planta la señal apropiada *u(t)*.

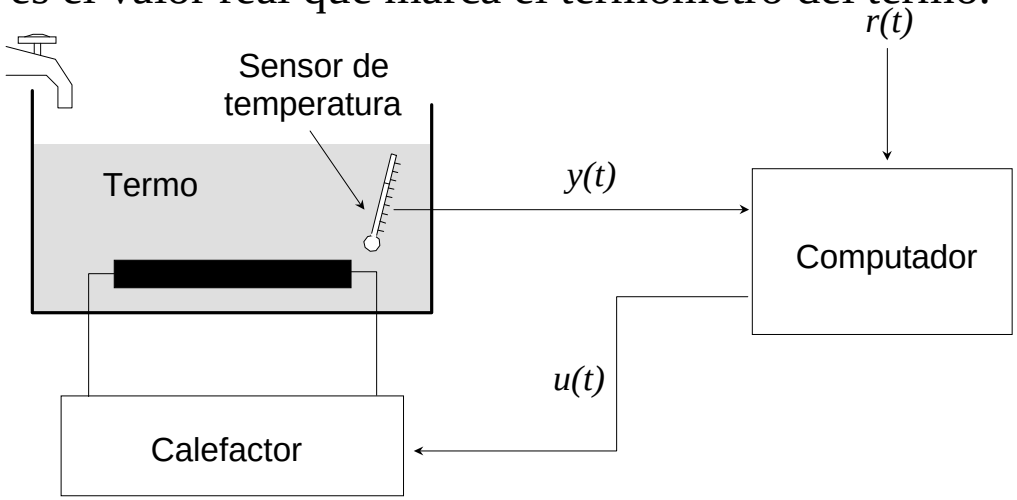
– Esto es lo que se conoce como **feedback controller**.

1.2. Características de los STR.

Por ejemplo:

– *r(t)* podría ser la temperatura que ha de tener el agua de un termo.

– *y(t)* es el valor real que marca el termómetro del termo.



– A partir de *y(t)* y de *r(t)* el controlador genera *u(t)*, con el fin de minimizar su diferencia.

– El controlador dispone de un **modelo matemático** que relaciona el calor aplicado e incremento de la temperatura experimentado (usa número reales).

1.2. Características de los STR.

- **Fiabilidad y seguridad**:

  – Hardware y software de los computadores deben ser fiables y seguros.

- **Interacción con el hardware**:

  – La naturaleza de los STR requiere una interacción del computador con el mundo externo (monitorizando sensores y activando actuadores).

  – En el pasado, la interacción con estos dispositivos se realizaba:

    1) Dejando el control al sistema operativo.

    2) Directamente la aplicación utilizando lenguaje ensamblador.

  – Hoy día,

    1) Se aconseja un control directo, no a través de una capa de funciones de SO.

    2) Se desaconseja el uso del ensamblador por la necesidad de fiabilidad.

    3) El **lenguaje de tiempo real** proporciona primitivas de acceso a dispositivos y soporte de interrupciones.

1.2. Características de los STR.

• **Determinismo temporal**:

– No es fácil diseñar e implementar sistemas que garanticen que la salida apropiada se generará en el tiempo adecuado en cualquier circunstancia.

– Para ello podemos:

1) Usar procesadores con potencia de cómputo bien sobrada, de forma que aseguramos que el comportamiento en el peor caso no produce un retraso en periodos críticos.

2) Exigir al lenguaje de tiempo real que proporcione al programador características que hagan al sistema predecible.

– A estas características se les llama **facilidades de tiempo real**. Son:

a) Especificar los tiempos en que las operaciones han de ejecutarse.

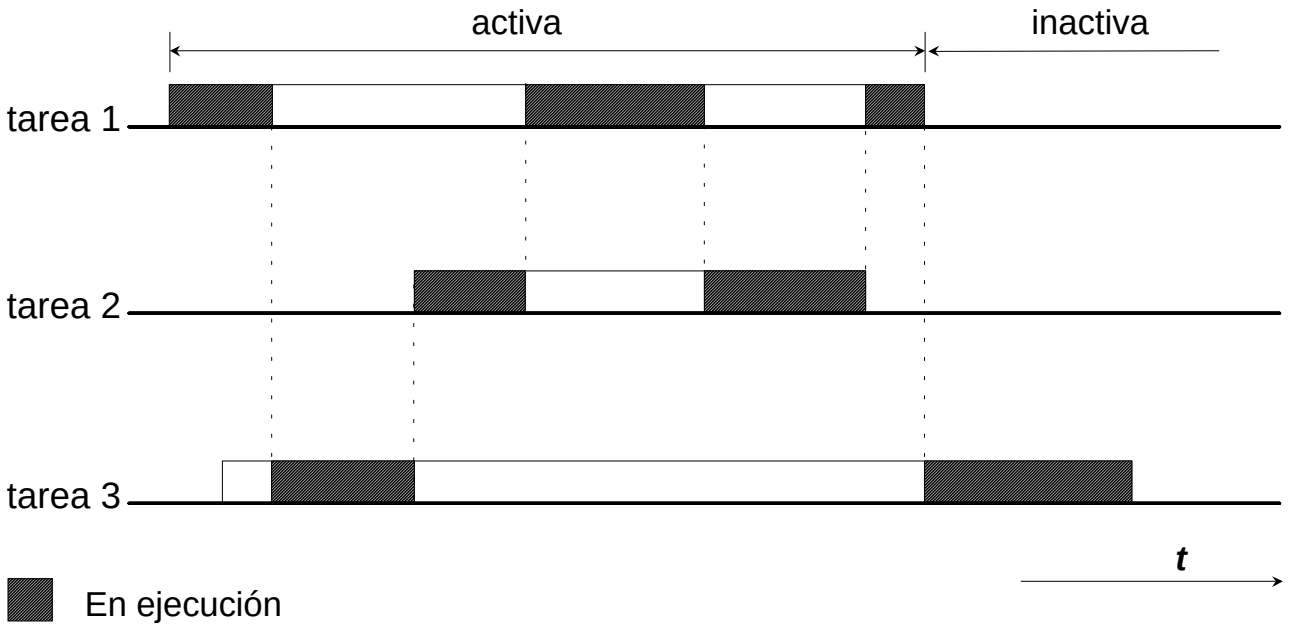
b) Especificar los tiempos en que las operaciones deben completarse.

c), d) Responder a los casos en los que no se pueden cumplir todos los requisitos temporales o éstos pueden cambiar dinámicamente.

1.3. Tiempo compartido y tiempo real.

• Los conceptos de *planificación* y *concurrencia* son muy importantes en los STR.

• Cada estímulo del entorno activa una o más *tareas*. Una tarea es una secuencia de instrucciones que se ejecuta en concurrencia con otras tareas. La ejecución de las tareas se multiplexa en el el tiempo en uno o más procesadores.



1.3. Tiempo compartido y tiempo real.

• En un STR la planificación de las tareas concurrentes (al contrario que en un STC) debe asegurar:

1. **Garantía de plazos**: Un STR funciona correctamente cuando se garantizan los plazos de todas las tareas. En un STC lo importante es asegurar un flujo lo más elevado posible.

2. **Estabilidad**: Caso de sobrecarga, un STR debe garantizar que al menos un subconjunto de tareas cumplen sus plazos (tareas *críticas*). En un STC, hay que repartir equitativamente el tiempo de ejecución.

3. **Tiempo de respuesta máximo**: En un STR se trata de acotar el tiempo de respuesta máximo de las tareas. En un STC se trata de minimizar el tiempo de respuesta medio.

• Para asegurar estas propiedades hay que utilizar un **método de planificación de tareas** adecuado. Para lo que hay que tener en cuenta que las tareas pueden ser:

periódicas, **aperiódicas** y **esporádicas**.

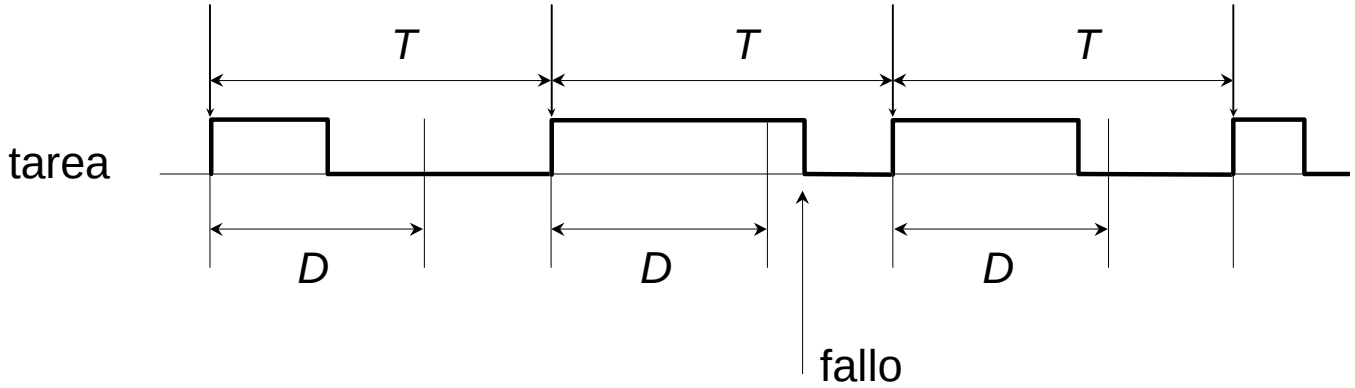
1.3. Tiempo compartido y tiempo real.

• **Tarea periódica:** se ejecuta en instantes de tiempo espaciados regularmente (ciclos).

– ***T*** Periodo de activación de la tarea.

– ***D*** Plazo de respuesta de la tarea (*deadline*). Tiempo máximo que puede transcurrir entre la activación de la tarea y la finalización de la misma.

– ***C*** Tiempo de CPU que, en el peor de los casos, necesita la tarea en cada ciclo para completarse.



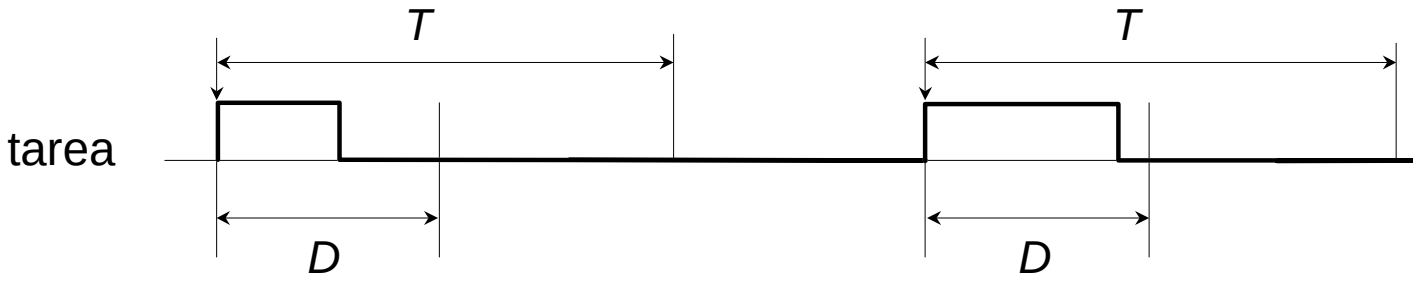
1.3. Tiempo compartido y tiempo real.

• **Tarea esporádica:** se ejecuta en respuesta a eventos que ocurren de forma asíncrona imposibles de prever.

– ***T*** Separación mínima entre dos eventos consecutivos. Cada evento *activa* la tarea que tiene asociada.

– ***D*** Plazo de respuesta de la tarea (*deadline*). Tiempo máximo que puede transcurrir entre la activación de la tarea y la finalización de la misma.

– ***C*** Tiempo de CPU que, en el peor de los casos, necesita la tarea en cada activación para completarse.



1.3. Tiempo compartido y tiempo real.

• **Tarea aperiódica:** generalización del caso anterior.

– ***D*** Plazo de respuesta de la tarea (*deadline*). Tiempo máximo que puede transcurrir entre la activación de la tarea y la finalización de la misma.

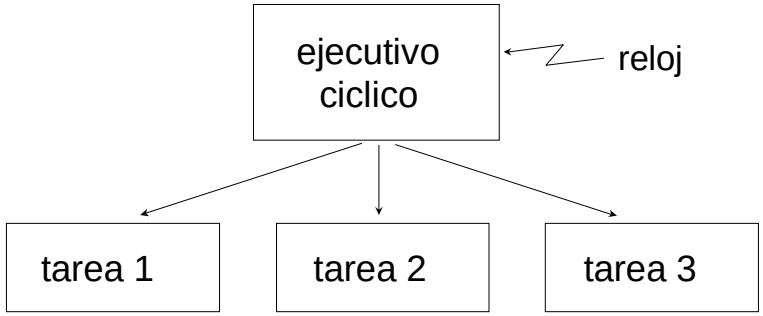
– ***C*** Tiempo de CPU que, en el peor de los casos, necesita la tarea en cada activación para completarse.

• **Ejemplo:** Control de un conjunto de subsistemas en un automóvil.

Tarea	Tipo	C	T	D
Control de inyección	Periódica	40	80	80
Medida de la velocidad	Periódica	4	20	5
Control de frenado (ABS)	Periódica	10	40	40

1.4. Planificación.

• **Ejecutivo cíclico:**



• Es un proceso que ejecuta cada una de las tareas de tiempo real siempre en la misma secuencia, de forma que se cumplan las restricciones temporales de las mismas.

• Se emplea en sistemas muy críticos pero de reducido tamaño.

• Si las tareas tienen periodos no relacionados por un común denominador, la codificación se vuelve compleja.

• Desarrollo laborioso y difícil de mantener.

1.4. Planificación.

• **Ejecutivo cíclico:**

Ejemplo: Sistema de control del automóvil

```
procedure Sistema_Control_Automovil is
  type Indice is range 1..4;
  Marco: Indice := 1;
begin
  loop
    Espera_Tick_Reloj; -- cada 20 ms.
    case Marco is
      when 1 => Mide_Velocidad;
                Controla_ABS;
                Controla_Inyeccion_1;
      when 2 => Mide_Velocidad;
                Controla_Inyeccion_2;
      when 3 => Mide_Velocidad;
                Controla_ABS;
                Controla_Inyeccion_3;
      when 4 => Mide_Velocidad;
                Controla_Inyeccion_4;
    end case;
  end loop;
end Sistema_Control_Automovil;
```

1.4. Planificación.

- **Tareas concurrentes:**

  – Las tareas se programan de forma independiente.

  – El SO o el núcleo de ejecución del lenguaje es el que reparte el tiempo de CPU entre las tareas activas.

```
task Medida_Velocidad          task Control_Freno          task Control_Inyeccion
loop                           loop                        loop
  Mide_Velocidad;                Controla_ABS;               Controla_Inyeccion;
  Sig := Sig + 0.020;            Sig := Sig + 0.040;         Sig := Sig + 0.080;
  delay until Sig;               delay until Sig;            delay until Sig;
end loop;                      end loop;                   end loop;
```

  – Más fácil de desarrollar y mantener.

  – La teoría de la planificación de tiempo real proporciona algoritmos y métodos de análisis que permiten determinar si se garantizan los plazos.

  – Asignando prioridades a las tareas se indica al planificador el orden en que dichas tareas tienen que ser ejecutadas.

1.4. Planificación.

– Asignación de prioridades a las tareas:

a) Dinámica.

b) **Estática**.

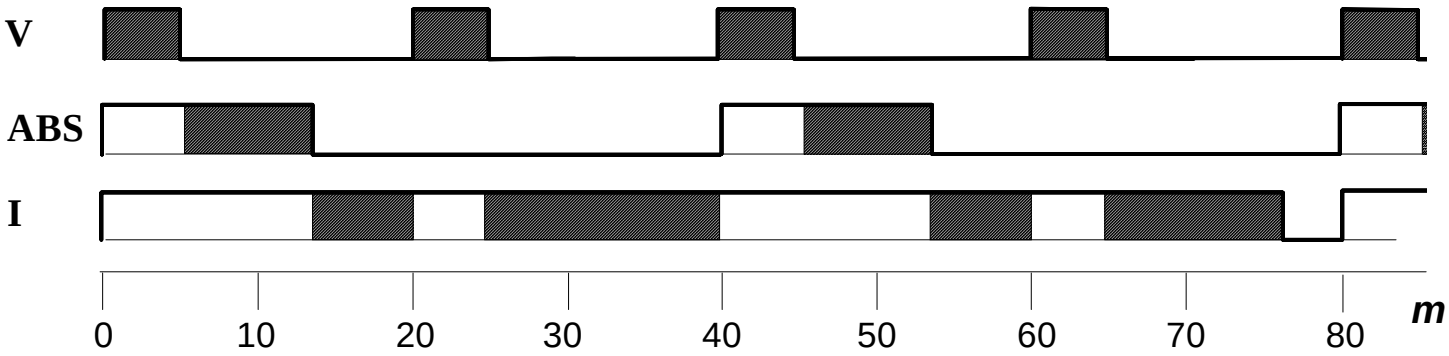
b.1) **Monótona en frecuencia (Rate Monotonic)**: prioridad más alta a la tarea más frecuente.

b.2) **Monótona en plazo (Deadline Monotonic)**: prioridad más alta a la tarea más urgente.

Para ambas existen herramientas matemáticas que permiten analizar el sistema y comprobar si se cumplen los plazos.

1.4. Planificación.

– Ejemplo: Asignación **Rate Monotonic** para el sistema de control del automóvil.



Tarea	Tipo	C	T	D
Control de inyección	Periódica	40	80	80
Medida de la velocidad	Periódica	4	20	5
Control de frenado (ABS)	Periódica	10	40	40

1.5. Sistemas críticos y no críticos.

• Podemos distinguir dos tipos de sistemas de tiempo real:

– **Crítico**: El incumplimiento de un plazo tiene consecuencias catastróficas.

Ejemplo: Sistema de frenado de un automóvil.

– **No crítico**: Se puede tolerar que ocasionalmente se incumpla un plazo de respuesta.

Ejemplo: Sistema de adquisición de datos del entorno.

TEMA 2. PROGRAMACIÓN DE STR.

2.1. Introducción.
2.2. Léxico.
2.3. Tipos de datos.
2.4. Instrucciones.
2.5. Subprogramas.
2.6. Estructura de programas.
2.7. Aspectos avanzados.
2.8. Facilidades para la programación de sistemas grandes.

2.1. Introducción.

- Ada es un lenguaje de programación descendiente de Pascal y diseñado para la construcción de sistemas software altamente fiables y de “larga vida”.

- En Ada se intentan detectar lo antes posible los errores de los programas, con el fin de reducir el tiempo de depuración de los mismos.

- Ada fue desarrollado originalmente por iniciativa y bajo la supervisión del Departamento de Defensa (DoD) de los EEUU para la realización de sistemas de tiempo real (empotrados).

- Existen dos versiones normalizadas:
  - Ada 83
  - Ada 95 (ó 9X)

2.2. Léxico.

- Los **identificadores** pueden contener letras, dígitos o el carácter separador '_'.
  Ejemplos: Sensor
  Tiempo_de_activación

- En Ada no se distingue entre mayúsculas y minúsculas.

- Las **palabras reservadas** no pueden ser usadas como identificadores.

- Se puede usar el carácter separador con los números para mayor claridad.
  Ejemplos: 123 150_000 3.141_592_654 1.7E-5

- Se usa el carácter '#' para indicar otra base. Ejemplo: 2#10001100#
- Los comentarios empiezan con '--' y van hasta el final de la línea.

2.3. Tipos de datos.

- Un **tipo de datos** es un conjunto de **valores** con un conjunto de **operaciones primitivas** asociadas.

- **Tipado fuerte**
  - No se pueden usar valores de un tipo en operaciones de otro tipo sin efectuar una **conversión de tipo** explícita.
  - Las operaciones dan siempre resultados del tipo correcto.

- **Tipos enumerados**:
  - Boolean -- *predefinido*
  - Character -- *predefinido*
  - **type** Modo **is** (Manual, Automático);

2.3. Tipos de datos.

- **Números enteros**:
  - **Con signo**:
    - Integer -- *predefinido*
    - **type** Index **is range** 1 .. 10;
  - **Modulares** (sin signo):
    - **type** Octeto **is mod** 256; -- *podrá tomar valores entre 0 y 255*
- **Números reales**:
  - **Coma flotante**: especifican el número de dígitos significativos a representar, pero no garantizan ningún grado de exactitud sobre todo el rango de representación.
    - Float -- *predefinido*

      (Por ejemplo: Float'Digits = 6, Float'First = -3.40282E+38
      Float'Last - Float'Pred(Float'Last) = 2.02824E+31
      Float'Succ(Float'Succ(0.0)) - Float'Succ(0.0) = 1.40130E-45)
    - **type** Longitud **is digits** 5 **range** 0.0 .. 100.0;
  - **digits** indica el número mínimo de dígitos decimales significativos.

2.3. Tipos de datos.

- **Coma fija**: Especifica un grado de exactitud que ha de mantenerse durante todo el rango del tipo.
  - **Ordinarios**:
    - Duration -- *predefinido*

      (Por ejemplo: Duration'First = -9223372036.854775810
      Duration'Last - Duration'Pred(Duration'Last) = 0.000000001
      Duration'Succ(Duration'Succ(0.0)) - Duration'Succ(0.0) = 0.000000001)

    - **type** Punto_fijo **is delta** 0.1 **range** -1.0 .. 1.0;

  - **Decimales**: Combina la definición de la exactitud y la del grado de exactitud.
    - **type** Decimal **is delta** 0.01 **digits** 10;

    Especifica un tipo punto fijo de 10 dígitos (dos de los cuales son decimales). El rango de la representación:

    -99_999_999.99 .. +99_999_999.99

2.3. Tipos de datos.

- Ejemplos:

```
type Indice is range 1 .. 100;                  -- entero
type Longitud is digits 5 range 0.0 .. 100.0;   -- coma flotante
first, last : Indice;
Front, Side : Longitud;

last:= first + 15;                              -- correcto
side:= 2.5 * front;                             -- correcto
side:= 2 * front;                               -- incorrecto
side:= front + 2*first;                         -- incorrecto
side:= front + 2.0*Longitud(first);             -- correcto
```

2.3. Tipos de datos.

- Tipos compuestos:
  - **String**
  - **Array**
    - **type** voltajes **is array** (Indice) **of** Voltaje;
    - **type** matriz **is array** (1 .. 10, 1 .. 10) **of** Float;
    - Elementos:

      v : voltajes;

      v(5):= 1.0;

      v:= (1.0, 0.0, 0.0, 0.0, 2.5, 0.0, 0.0, 0.0, 0.0, 0.0);

      v:= (1 => 1.0, 5 => 2.5, **others** => 0.0);

2.3. Tipos de datos.

- Tipos compuestos:
  - **Registros**

```
type estado is record
    modo_operacion : modo;
    referencia     : voltaje;
end record;

Elementos:
    x : estado;
    x.referencia:= 0.0;
    x:= (automatico, 0.0);
    x:= (modo_operacion => automatico, referencia => 0.0);
```

2.3. Tipos de datos.

- Tipos acceso:
  - Designan valores de otros tipos (equivalen a los punteros C):
    type Acceso_estado **is access** estado;
  - Los objetos a los que se accede a través de ellos se crean dinámicamente:
    s : Acceso_estado:= **new** estado;
    Las variables acceso se inicializan a **null** si no se dice nada.
  - Acceso al objeto dinámico:
    s.modo_operacion:= manual;
    s.**all**:= (modo_operacion => manual, referencia => 0.0);
    s := **new** estado'(modo_operacion=>manual, referencia=> 0.0);

    - No se puede operar con los valores de los tipo acceso

2.3. Tipos de datos.

- Declaraciones: Asocian nombres con definiciones de
  - tipos

    type real **is digits** 8;
  - objetos

    x : real:= 0.0;

    j : **constant** complejo:= (0.0, -1.0);
  - números

    pi : **constant**:= 3.14_15_92_65;
  - subprogramas y otras entidades.
  - Se colocan en las **zonas declarativas.**
  - Al entrar en la zona declarativa se **elaboran**, es decir, se crea la entidad declarada y se realizan las operaciones iniciales asociadas a la misma.

    P. Ejemplo: iniciación del valor de una variable ó asignación de memoria para crear un objeto dinámico.

2.4. Instrucciones.

- **Simples**:
  - **Asignación**: U:= 2.0*v(5) + U0;
    - El tipo de la expresión y de la variable tienen que coincidir.
    - La conversión de tipos es explícita. P.ej:

      a : float:= 10.5; -- *a = 10.5*

      b : integer:=20; -- *b = 20*

      a:= a + float(b); -- *a = 30.5*
  - **Llamada a procedimientos**: borrar(v);
  - **Nula**: **null**;
- **Compuestas**:
  - **Secuencia**: 1 o más instrucciones.

    borrar(v);

    u:= 2.0*v(65) + u0;

    Sintaxis: secuencia → instrucción {instrucción}

2.4. Instrucciones.

- **Compuestas**:
  - **Bloque**:

```
declare
    a : integer;          -- variable local al bloque
begin
    leer(a);
    imprimir(a);
end;                      -- a deja de existir aquí
```

Sintaxis:

bloque → [identificador :]
[**declare** {declaración}] -- parte declarativa
begin secuencia **end** [identificador];

2.4. Instrucciones.

- **Compuestas**:
  - **Selección**:

```
if a > b then max:= a;
elsif a < b then max:= b;
else max:= 0;
end if;
```

Sintaxis:

selección_if → **if** expresión_booleana **then** secuencia
{**elsif** expresión_booleana **then** secuencia}
[**else** secuencia]
end if;

2.4. Instrucciones.

- **Compuestas**:
  - **Selección por casos**:

```
case dia is
   when lunes                => principio_semana;
   when martes .. viernes    => mitad_semana;
   when sabado | domingo     => fin_de_semana;
   when others               => no_hay;
end case;
```

¡Hay que cubrir todos los valores del tipo discreto!

Sintaxis:

selección_case → **case** expresión_discreta **is** alternativa {alternativa}
end case;

alternativa → **when** lista **=>** secuencia

lista → opción {| opción}

opción → expresión | intervalo | **others**

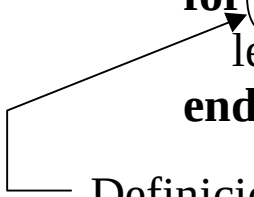
2.4. Instrucciones.

- **Compuestas**:
  - **Iteración**: Existen 3 posibilidades

```
for i in 1..10 loop
    leer(v(i));
end loop;
```

```
while a < b loop
    a:= a+1;
end loop;
```

```
loop
    leer(d);
    exit when d=0;
end loop;
```



Definición implícita del índice

Sintaxis:

bucle → [identificador :]
[esquema_de_iteración] **loop** secuencia **end loop** [identificador];

esquema_de_iteración → **for** índice **in** [**reverse**] intervalo_discreto
| **while** expresión_booleana

En cuanto a **exit**:

Sintaxis: exit [identificador] [**when** expresión_booleana];

- La ejecución continúa inmediatamente después del bucle.
- En bucles anidados, se sale del interior o del indicado por el identificador.

2.4. Instrucciones.

- **Compuestas**:
  - **Transferencia de control**:

    <<label>> i:= i + 1;

    Ada.Integer_Text_IO.put(i);

    goto label;

    Sintaxis: **goto** etiqueta1;

    etiqueta1 → identificador

    etiqueta2 → <<identificador>>

    instrucción_etiquetada → etiqueta2 instrucción

    - Limitaciones para aumentar la seguridad
      - La instrucción etiquetada estará en la misma secuencia y con el mismo grado de anidamiento que la de transferencia.
      - No se puede salir de un subprograma ni de otras entidades de programa

2.5. Subprogramas.

- Hay dos tipos de **subprogramas**:
  - ***procedimientos***
  - ***funciones***

  } Ambos pueden tener parámetros

  - Un subprograma tiene dos partes
    - **Especificación**: define la interfaz (nombre y parámetros).
    - **Cuerpo**: define la acción o el algoritmo que se ejecuta cuando se invoca el subprograma.

  - A veces se puede omitir la especificación, en este caso la interfaz se define al declarar el cuerpo.

2.5. Subprogramas.

- Especificación de subprogramas:
  - La especificación se declara en una zona declarativa:

    procedure Reset; -- *Sin parámetros*

    procedure Incrementar(valor : **in out** integer; cantidad: **in** integer:= 1);

    function Maximo(a, b : integer) **return** integer;

  - *Sintaxis*

    declaración_de_subprograma → especificación_de_subprograma;

    especificación_de_subprograma → **procedure** nombre [parámetros]

    | **function** nombre [parámetros] **return** tipo

    parámetros → (definición{, definición})

    definición → lista : modo tipo [:= valor por defecto]

    lista → identificador{, identificador}

    modo → [**in**] | **out** | **in out**

    valor_por_defecto → expresión

2.5. Subprogramas.

- Tres modos de parámetros:
  - **in** : no se modifica al ejecutar el subprograma. Se toma por defecto. Puede tener un valor por defecto.
  - **out** : el valor del parámetro será asignado por el subprograma.
  - **in out** : el valor del parámetro es tanto de entrada como de salida.

  - Los parámetros de las funciones sólo pueden ser de modo **in**.

- Cuerpo de subprogramas:
  - Se coloca en una zona declarativa

```
procedure Incrementar( valor : in out integer; cantidad: in integer:= 1) is
begin
   valor:= valor + cantidad;
end Incrementar;
```

2.5. Subprogramas.

- Cuerpo de subprogramas:

```
function Maximo (a, b : integer) return integer is
begin
  if a > b then return a;
  else return b;
  end if;
end Maximo;
```

  – *Sintaxis*

  cuerpo_de_subprograma → especificación_de_subprograma **is**
  {declaración} **begin** secuencia; **end** nombre;

  - El cuerpo comienza con una zona declarativa (variables locales).
  - El cuerpo de una función debe contener, al menos, una instrucción **return**.

2.5. Subprogramas.

- Llamada a subprogramas:
  - Es una instrucción simple:

    Incrementa(x, 2); -- *asociación posicional*

    Incrementar(valor => x, cantidad => 2) -- *asociación nombrada*

    Incrementar(x); -- *cantidad => 1 (por defecto)*

    w:= 2*Maximo(u,v);
  - *Sintaxis*

    llamada_a_subprograma → nombre; | nombre parámetros_reales;

    parámetros_reales → (asociación{, asociación})

    asociación → [nombre_formal =>] parámetro_real

    parámetro_real → expresión | nombre_de_variable
    - Los parámetros formales de modo **in** se asocian a expresiones.
    - Los de modo **out** ó **in out** se asocian a variables.
    - Con asociación nombrada da igual el orden.

2.6. Estructura de programas.

- Un programa Ada está formado por un conjunto de *unidades de programa*.
- Existen distintas clases de unidades de programa:
  - Subprogramas: Definen algoritmos. Son procedimientos y funciones.
  - Paquete: unidad básica para definir una colección de entidades lógicamente relacionadas.
  - Otras: Tarea, Objeto protegido, Unidad genérica.
- En general, una unidad de programa consta de dos partes: una *declaración* (también denominada *especificación*) y un *cuerpo*.
- Los compiladores de Ada compilan *unidades de compilación*.
- Una unidad de compilación es tanto la parte declarativa como el cuerpo de una unidad de programa, precedidos de una *cláusula de contexto*.
- Un programa Ada está formado por
  - un *procedimiento principal* (normalmente sin parámetros).
  - otros subprogramas o paquetes escritos por el programador.
  - subprogramas o paquete predefinidos (y precompilados).

2.6. Estructura de programas.

- Cuando se usan elementos de un paquete hay que importar el paquete con una cláusula **with**:

  with nombre_de_paquete{, nombre_de_paquete};

- Para hacer referencia directa a los nombres declarados en los paquetes importados se usa la cláusula **use**.

  use nombre_de_paquete{, nombre_de_paquete};

- Paquetes predefinidos:
  - Operaciones numéricas:

    Ada.Numerics, Ada.Numerics.Generic_Elementary_Functions
  - Operaciones con caracteres y tiras:

    Ada.Characters, Ada.Strings
  - Entrada y salida:

    Ada.Text_IO, Ada.Integer_Text_IO, Ada.Float_Text_IO
  - Interface con otros lenguajes . . .

2.6. Estructura de programas.

- Ejemplo: Ordenación de un array por el método de la burbuja.

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO;
use Ada.Integer_Text_IO;

procedure Burbuja is
 type m_array is array (0..9) of
                 integer;
 a : m_array:=
      (8, 4 ,2 ,5 ,3, 0, 9, 7, 6, 1);
 aux : integer;
 procedure Imprime_array(m : m_array) is
 begin
   for i in 0 .. 9 loop
     put(m(i),2);
   end loop;
 end Imprime_array;
begin
  Put_Line("Array de partida:");
  Imprime_array(a);
  for i in 0 .. 9 loop
   for j in reverse 0..(i-1) loop
    if (j>=0 and a(j)>a(j+1)) then
     aux:= a(j);
     a(j):= a(j+1);
     a(j+1):= aux;
    end if;
   end loop;
  end loop;
  New_line;
  Put_line("Después de la
            ordenación:");
  Imprime_array(a);
end Burbuja;
```

2.7. Aspectos avanzados.

- Un **subtipo** es un subconjunto de valores de un tipo, definido por una **restricción.**
  - La forma más simple de restricción es un intervalo de valores:

    subtype Indice_inferior **is** Indice **range** 1 .. 30;

    subtype Indice_superior **is** Indice **range** 15 .. 100;

    subtype Tensión_baja **is** Voltaje **range** 0.0 .. 2.0;
  - Hay dos subtipos predefinidos:

    subtype Natural **is** Integer **range** 0 .. Integer'Last;

    subtype Positive **is** Integer **range** 1 .. Integer'Last;
  - Las operaciones con valores de distintos subtipos de un mismo tipo están permitidas.
  - En tiempo de compilación ó ejecución se comprueban las restricciones.
  - Ejemplo: a : Indice_inferior; b : Indice_superior;

    a:= b + 10; -- *error si b > 20*

2.7. Aspectos avanzados.

- Un **tipo derivado** es una copia de otro tipo (con los valores y operaciones de este último).

  type edad **is new** Integer;

  – Se puede imponer una restricción en el conjunto de valores:

  type edad_persona **is new** Integer **range** 0 .. 150;

  type edad_gato **is new** Integer **range** 0 .. 20;

  a : Integer; b : edad_persona; c : edad_gato; -- tipos incompatibles

  – Los valores no son compatibles, aunque se pueden hacer conversiones:

  a:= b*c; -- incorrecto

  a:= Integer(b)*Integer(c); -- correcto

  – En tiempo de compilación ó ejecución se comprueban las restricciones.

2.7. Aspectos avanzados.

- Se pueden declarar tipos formación con un intervalo de índice indefinido (**formaciones irrestringidas**).

  type Muestras **is array** (Positive **range** <>) **of** Integer;

  – El tipo **String** está predefinido:

  type String **is array** (Positive **range** <>) **of** Character;

  – En la declaración de los objetos hay que restringir los índices:

  a : Muestras; -- incorrecto

  a : Muestras (1 .. 10); -- correcto

  a : Muestras := (1 ..10 => 0); -- correcto

  s : String (1 .. 25); -- correcto

  s : String := “Mi cadena”; -- correcto

  – También se pueden declarar formaciones con intervalos dinámicos (evaluados en tiempo de ejecución, al elaborar la declaración):

  type Muestras **is array** (1 .. n_entradas) **of** Integer;

  Tiene que estar definida al elaborar

2.7. Aspectos avanzados.

- Un **discriminante** es un componente de un registro que permite parametrizar los objetos del tipo

```
type Exactitud is (alta, baja, media);
type Medida (tipo : Exactitud) is record
        valor : integer;
end record;

a : Medida(alta);
a : Medida:= (tipo => alta, valor => 10);

subtype Medida_exacta is Medida (alta);
b : Medida_exacta;
```

  - El discriminante tiene que ser de un tipo discreto o acceso y no se puede cambiar una vez asignado.

2.7. Aspectos avanzados.

- Se puede usar un discriminante para declarar tipos registro con partes **variantes**:

```
type Medida (tipo : Exactitud) is record
  valor : integer;
  case tipo is
     when alta => valor_auxiliar : integer:= 0; when others => null;
  end case;
end record;
a : Medida(baja);
b : Medida:= (alta, 10, 15);
```

- Se pueden hacer conversiones:
  - Entre tipos numéricos
  - Entre formaciones con la misma estructura y tipo de elementos
  - Entre registros con los mismos discriminantes y tipo de componentes
  - Entre un tipo y sus derivados (pero no éstos entre sí).

2.8. Facilidades para la programación de SG.

- Los **módulos** son utilizados con el fin de dominar la complejidad de los grandes sistema empotrados.
- Un **módulo** puede ser descrito, informalmente, como una colección de objetos y operaciones lógicamente relacionadas. Es lo que en Ada se denomina **paquete**.

- Con una estructura de módulos es posible soportar:
  - **Ocultamiento de información**: permite ocultar detalles de la implementación declarándolos en el cuerpo del paquete.
  - **Compilaciones separadas**: uso de la cláusula de contexto **with**. Permite comprobar la consistencia lógica de un programa antes de su construcción detallada (haciendo uso de la parte de especificación de los paquetes).
  - **Tipos abstractos de datos**: un módulo nombrará un nuevo tipo y dará las operaciones que pueden ser aplicadas al mismo. La estructura del TDA es ocultada dentro del módulo y son posibles más de una instancia del tipo.

2.8. Facilidades para la programación de SG.

- Ejemplo TAD:

```
package Buffer_TAD is
  type Buffer is private;
  procedure Put(B : out Buffer; S : in String);
  procedure Get(B : in out Buffer; C : out Character);
private
  Max : constant integer := 80;
  type Buffer is record
     Data : String(1..80);
     Start : Integer := 1;
     Finish : Integer := 0;
   end record;
end Buffer_TAD;
package body Buffer_TAD is
  procedure Put(B : out Buffer; S : in String) is begin . . . end Put;
  procedure Get(B : in out Buffer; C : out Character) is  begin . . . end Get;
end Buffer_TAD;
```

```
Mi_Buffer : Buffer;
. . .

Put(Mi_Buffer, nombre);

. . .
```

2.8. Facilidades para la programación de SG.

- **Reusabilidad**: A diferencia de lo que ocurre con el software, el hardware se construye en base a componentes bien establecidos en el mercado que pueden ser combinados para formar un sistema más amplio y complejo. En Ada, uno de los mecanismos para lograr la reusabilidad son los paquetes con parámetros genéricos.
- Ejemplo: Pila genérica.

```
generic
  type Elemento is private;
package Pila_generica is
  type pila is private;
  procedure crear (p : in out pila);
  procedure push (p : in out pila; e : in Elemento);
  procedure pop (p : in out pila; e : out Elemento);
private                                  -- declaración del tipo pila
  . . .
end Pila_generica;
```

Instanciación: en la parte declarativa de nuestro programa pondremos:

```
package Pila_Enteros is new Pila_generica(Integer);
```

TEMA 3. FIABILIDAD Y TOLERANCIA DE FALLOS.

3.1. Introducción.

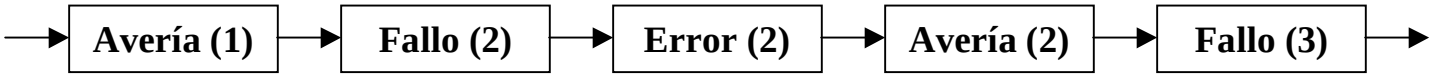
- Uno de los requisitos más importantes de los STR es que deben ser altamente **fiables y seguros**.
- En 1986, Hetch y Hetch estudiaron sistemas software grandes y concluyeron:
  - Por cada millón de líneas de código se introducen 20.000 errores.
  - Normalmente, el 90% de estos errores se encuentran en la fase de prueba.
  - De los que no se detectan, unos 200 se manifiestan en el primer año de operación.
  - Quedan 1800 no detectados.
  - El mantenimiento corregiría unos 200 errores e introduciría otros tantos.
- Los fallos de funcionamiento de un STR pueden tener su origen en:
  - Una especificación inadecuada.
  - Errores de diseño de los componentes software. ←
  - Averías en los componentes hardware.
  - Interferencias transitorias o permanentes en los sistemas de comunicaciones.

3.1. Introducción.

- Definiciones:
  - **Fiabilidad** (reliability) de un sistema:
    - Es una medida de conformidad con una especificación autorizada de su comportamiento.
    - Idealmente, esta especificación debería ser completa, consistente y no ambigua.
    - Los **tiempos de respuesta** son una parte importante de la especificación de un sistema.
  - **Avería** (failure) de un sistema:
    - Es una desviación del comportamiento de un sistema respecto de su especificación.
    - Altamente fiable = tasa de averías baja.
  - Las definiciones anteriores se refieren al *comportamiento* del sistema (su apariencia externa).
  - Las averías son el resultado de problemas internos no esperados, **errores**, que al final se manifiestan en el comportamiento externo del sistema.

3.1. Introducción.

- Definiciones:
  - Las causas mecánicas o algorítmicas de los errores son denominadas **Fallos** (faults).
  - Un sistema está formado normalmente por varios componentes, cada uno de los cuales puede ser visto como un sistema aislado. Por lo tanto, una avería en un sistema puede provocar un fallo en otro, lo cual dará lugar a un error y a una avería potencial en este último sistema.
  - Cadena fallo, error, avería, fallo.

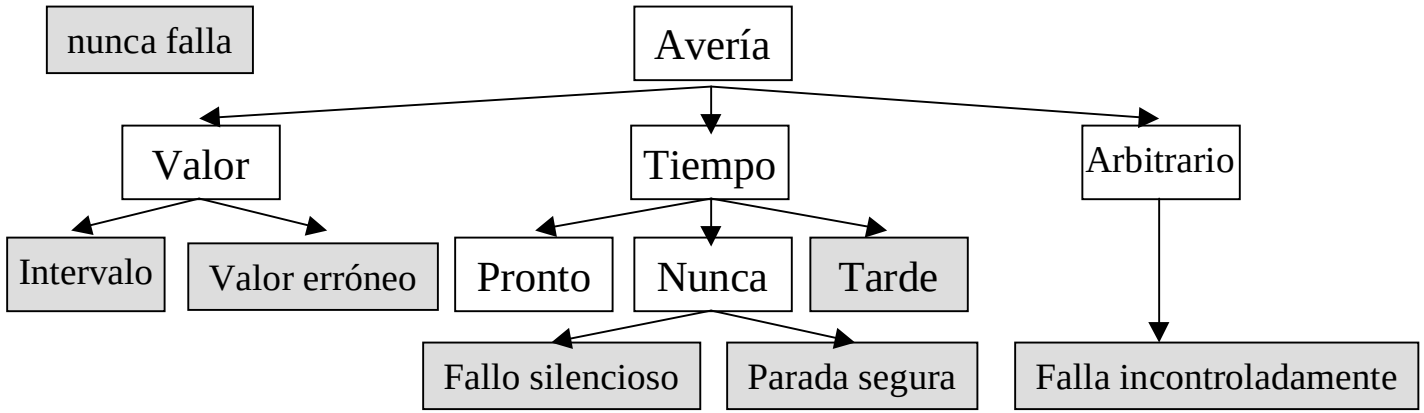


- Podemos distinguir 3 tipos de fallos:
  - **Transitorios**: Comienzan en un determinado instante, continúan en el sistema durante algún tiempo y después desaparecen. Ej.: interferencias.
  - **Permanentes**: Permanecen en el sistema hasta que se reparan. Ej.: error de diseño software o componente hardware roto.
  - **Intermitentes**: Fallos transitorios que ocurren de vez en cuando. Ej: componente hardware sensible al calor.

3.1. Introducción.

- Para crear sistemas fiables debe impedirse que todos estos tipos de fallos causen averías.
- Modos de avería: Puesto que un sistema proporciona servicios es, por tanto, posible clasificar los modos de avería de un sistema en función de las consecuencias que tienen sobre los servicios prestados. Podemos identificar dos dominios generales de modos de avería:
  - Valor: el valor asociado con el servicio es erróneo.
  - Tiempo: el servicio no es prestado en el instante correcto.

  → Combinación de ambos tipos de avería: avería arbitraria.



3.2. Prevención y tolerancia de fallos.

- Existen dos formas que permiten mejorar la fiabilidad de un sistema:
  - **Prevención de fallos** (fault prevention): Se trata de evitar que se introduzcan fallos en el sistema antes de que entre en funcionamiento.
  - **Tolerancia de fallos** (fault tolerance): Se trata de conseguir que el sistema siga funcionando aunque se produzcan fallos.
- Ambas aproximaciones tratan de producir sistemas con modos de fallo bien definidos.

Prevención de fallos.

- La prevención de fallos se realiza en 2 etapas:
  - **Evitar fallos**: Impide que se introduzcan fallos durante la construcción del sistema. Hay que considerarla tanto para hardware como para software.
    - **Evitar fallos hardware:**
      - Usar componentes fiables.
      - Técnicas rigurosas de ensamblaje de subsistemas.
      - Apantallamiento de hardware.

3.2. Prevención y tolerancia de fallos.

Prevención de fallos (evitar fallos).

- **Evitar fallos software**: Actualmente, los componentes software de los STR grandes son mucho más complejos que los componentes hardware.
  - Especificación rigurosa o formal de requisitos.
  - Reutilización de componentes software fiables.
  - Usar lenguajes con facilidades para abstracción de datos y modularidad.
  - . . .

– **Eliminación de fallos**: Consiste en encontrar y eliminar los fallos que inevitablemente se encuentran en el sistema una vez construido. Algunas técnicas que se pueden usar:

- Revisiones del diseño.
- Verificación del programa e inspección del código.
- Pruebas.

3.2. Prevención y tolerancia de fallos.

Prevención de fallos (eliminación de fallos).

- Las pruebas no sirven para eliminar todos los errores, ya que:
  - Únicamente pueden mostrar la presencia de fallos, pero no su ausencia.
  - Algunas veces es imposible hacer pruebas bajo condiciones realistas.
  - Los errores de especificación no aparecen hasta que el sistema está en funcionamiento.

- Los componentes hardware fallarán a pesar de las técnicas de prevención $\Rightarrow$ la prevención de fallos no tendrá éxito cuando la frecuencia o duración de los tiempos de reparación sea inaceptable, o el sistema sea inaccesible. Se hace necesario, por tanto, la utilización de técnicas de **tolerancia de fallos.**

3.2. Prevención y tolerancia de fallos.

Tolerancia de fallos.

- Debido a las inevitables limitaciones de la aproximación de prevención de fallos, los diseñadores de los STR deben considerar el uso de la tolerancia de fallos.
- Prevención y tolerancia de fallos han de usarse conjuntamente (nos centraremos en esta última).
- Un sistema puede proporcionar diferentes niveles de tolerancia de fallos:
  - **Tolerancia de fallos completa** (*fail operational*): El sistema sigue funcionando en presencia de errores, al menos durante un tiempo, sin perder funcionalidad ni prestaciones.
  - **Degradación aceptable** (*fail soft*): El sistema sigue funcionando en presencia de errores con una pérdida parcial de funcionalidad o prestaciones hasta la reparación del fallo.
  - **Parada segura** (*fail safe*): El sistema se detiene en un estado que asegura su integridad y la del entorno hasta que se repare el fallo.
- El grado de tolerancia de fallos requerido depende de la aplicación.

3.3. Redundancia estática y dinámica.

- Todas las técnicas para lograr tolerancia de fallos se basan en la introducción de elementos extra en el sistema para detectar fallos y recuperarse de los mismos.
- Estos componentes extra son los componentes redundantes ya que no son requeridos en un sistema perfecto.
- El objetivo de la tolerancia de fallos es el de minimizar la redundancia y maximizar la fiabilidad proporcionada teniendo en cuenta las restricciones de costo y tamaño del sistema.
- Esto aumenta la complejidad del sistema y puede introducir fallos adicionales.
- A la hora de estructurar el sistema, lo mejor es separar los componentes redundantes del resto del sistema.
- De nuevo tenemos que considerar tanto la tolerancia de fallos hardware como la software, aunque nos centraremos más en esta última.

3.3. Redundancia estática y dinámica.

Redundancia para el hardware: Existen dos tipos:

- **Redundancia estática**
  - Los componentes redundantes están siempre activos.
  - Las decisiones se toman por mayoría.
  - Se utiliza para enmascarar los fallos.
  - Ejemplo: Redundancia modular n-ésima (RMN):
    - *n* subsistemas idénticos que operan en paralelo.
    - Un circuito de votación de mayoría.
    - Con n = 3 se enmascara el fallo de 1 componente.
    - Con n = 2 sólo se pueden detectar fallos, pero no enmascararlos.
- **Redundancia dinámica**
  - Los componentes redundantes sólo se activan cuando se detecta un fallo.
  - Se basa en la **detección** y posterior **recuperación** de los fallos.
  - Ejemplo: Checksums y bits de paridad en memoria.

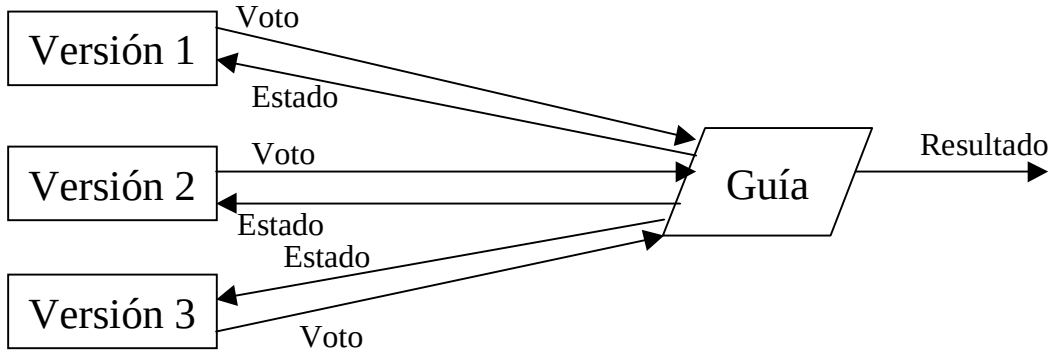
3.3. Redundancia estática y dinámica.

Redundancia para el software:

- Técnicas para detectar y corregir errores de diseño del software.
- Existen dos aproximaciones generales:
  - **Redundancia estática**:
    - Programación con N versiones.
  - **Redundancia dinámica**:
    - Dos etapas: detección y recuperación de fallos.
    - Bloques de recuperación:
      - » Proporcionan recuperación hacia atrás
    - Excepciones:
      - » Proporcionan recuperación hacia adelante.

3.3.1. Programación de N versiones.

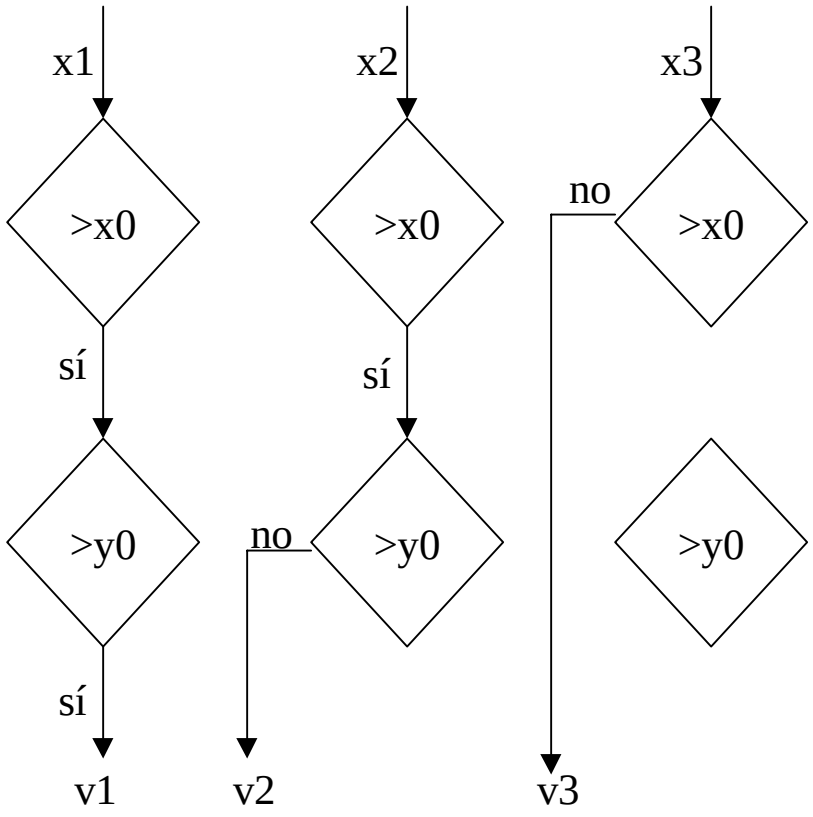
- Aplicación de la técnica NMR al software.
- Dado que el software no se deteriora con el tiempo, esta técnica se aplica para detectar fallos de diseño.
- Consiste en la generación **independiente** de N programas con idéntica funcionalidad a partir de la misma especificación inicial. N tiene que ser mayor que 2.
- Los programas se ejecutan concurrentemente con las mismas entradas y sus resultados son comparados por un proceso **guía**.



- **Granularidad** de la tolerancia a fallos = frecuencia con la que se realizan las comparaciones.

3.3.1. Programación de N versiones.

- Problema de la comparación consistente:



- La comparación de valores no es exacta.
- Cada versión produce un resultado correcto, pero diferente de las otras.
- No se arregla comparando $x0+\Delta$ y $y0+\Delta$

3.3.1. Programación de N versiones.

- **Problemas de la Programación con N versiones**: La correcta aplicación de este método depende de:
  - Especificación inicial.
    - Un error de especificación aparecerá en todas la versiones implementadas.
  - Desarrollo independiente.
    - No debe existir contacto entre los equipos.
    - No está claro que distintos programadores comentan errores independientes.
  - Presupuesto suficiente.
    - Los costes de desarrollo se multiplican.
    - El mantenimiento es también más costoso.

- Se ha utilizado en sistemas de aviónica críticos.

3.3. Redundancia estática y dinámica.

Redundancia dinámica

- Los componentes redundantes operan sólo cuando se detecta un error.
- Consta de 4 etapas:
  1. **Detección de errores**: Ningún esquema de tolerancia de fallos puede emplearse hasta que el error asociado se detecta.
  2. **Evaluación y confinamiento de daños**: Intenta delimitar la extensión del sistema que ha sido corrompida por el error.
  3. **Recuperación de errores**: Lleva a un sistema que ha sido corrompido hasta un estado en el que pueda continuar su operación normal (quizás con funcionalidad degradada).
  4. **Reparación de fallos**: Aunque el sistema funcione, el fallo puede persistir y hay que repararlo.

3.3. Redundancia estática y dinámica.

1. Detección de errores

- La efectividad de cualquier sistema de tolerancia de fallos depende de la efectividad de las técnicas de detección de errores.
- Podemos identificar 2 clases de técnicas de detección de fallos**:**

  1. Por el entorno de ejecución:
    - Hardware (Ej: overflow aritmético).
    - Núcleo de ejecución o el sistema operativo (Ej: valor fuera de rango).

  2. Por el software de aplicación:
    - Duplicación (redundancia con dos versiones).
    - Comprobaciones de tiempo:
      - Se asume que un componente está en un estado de error si no reinicia un proceso temporizador (*watchdog timer*).
      - Detección de plazos no cumplidos por el núcleo de ejecución.
    - Inversión de funciones en componentes en los que la relación entre entrada y salida es uno a uno.

3.3. Redundancia estática y dinámica.

1. Detección de errores

2. Por el software de aplicación:

- Códigos detectores de error.
- Validación de estado.
- Validación estructural para comprobar la integridad de objetos de datos como listas o colas.

2. Evaluación y confinamiento de daños

- Desde que ocurre un fallo hasta que se detecta transcurre tiempo, durante el que puede extenderse información errónea por el sistema y su entorno.
- El confinamiento de daños tiene que ver con la estructuración del sistema de forma que minimice el daño causado por un componente averiado (también conocido como *firewalling*).
- Dos técnicas para estructurar el sistema:
  - Descomposición modular (confinamiento estático): el sistema se estructura en módulos con interfaces bien definidos, con lo que se hace más difícil que un error se transmita entre componentes.

3.3. Redundancia estática y dinámica.

2. Evaluación y confinamiento de daños

- Acciones atómicas (confinamiento dinámico): durante una acción atómica un componente no interacciona con el sistema. Sirven para mover al sistema desde un estado consistente hasta otro.

3. Recuperación de errores

- Probablemente constituye la fase más importante.
- Se trata de situar al sistema en un estado correcto desde el que pueda seguir funcionando normalmente, aunque quizás con un servicio degradado.
- Se han propuesto dos aproximaciones:

  1. Recuperación directa (hacia adelante).

  - Consiste en avanzar desde el estado erróneo hacia un nuevo estado correcto (hay que asegurar también el sistema controlado).
  - Es específica y depende de una predicción correcta de los posibles fallos y su situación.
  - Ejemplos: punteros redundantes en estructuras de datos, código de Hamming

3.3. Redundancia estática y dinámica.

3. Recuperación de errores

2. Recuperación inversa (hacia atrás).

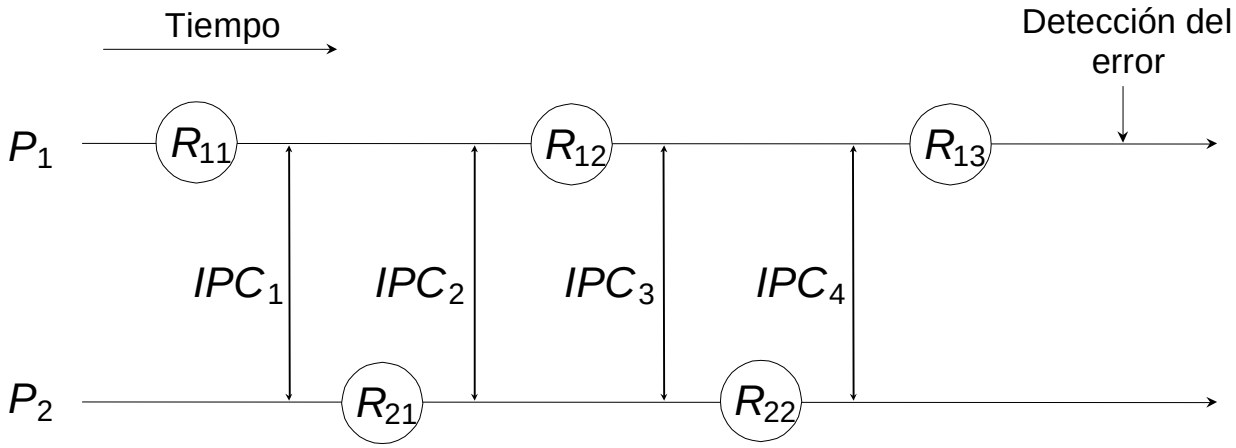
- Consiste en retroceder desde el estado erróneo a un estado anterior correcto (**punto de recuperación**), a partir del cual se ejecuta un segmento de programa alternativo con idéntica funcionalidad que el que falló pero distinto algoritmo.
- Tiene la ventaja de que el estado erróneo se elimina y de que no depende de encontrar la localización o causa del fallo.
- Sirve para recuperar el sistema ante fallos imprevistos, pero tiene como desventaja que no puede deshacer los efectos que el fallo puede haber tenido sobre el entorno del sistema controlado.
- Para establecer un punto de recuperación es necesario guardar información del estado en tiempo de ejecución.

3.3. Redundancia estática y dinámica.

3. Recuperación de errores

2. Recuperación inversa (hacia atrás).

- La recuperación se complica cuando hay tareas concurrentes que se comunican.



- Si P_1 detecta un error en el instante T_e, regresa al estado dado por su punto de recuperación R_{13} y el sistema vuelve a un estado consistente.
- ¿Y si es P_2 el que detecta el error en el instante T_e? $\Rightarrow$ **Efecto dominó**

3.3. Redundancia estática y dinámica.

4. Reparación de fallos

- Un error es la manifestación de un fallo y aunque la fase de recuperación de errores haya llevado al sistema a un estado no erróneo, el error podría reaparecer ⇒ la fase final de la tolerancia de fallos es eliminar el fallo del sistema.
- La reparación de fallos se divide en dos etapas:

  1. La localización del fallo: Se pueden usar técnicas de detección de errores.

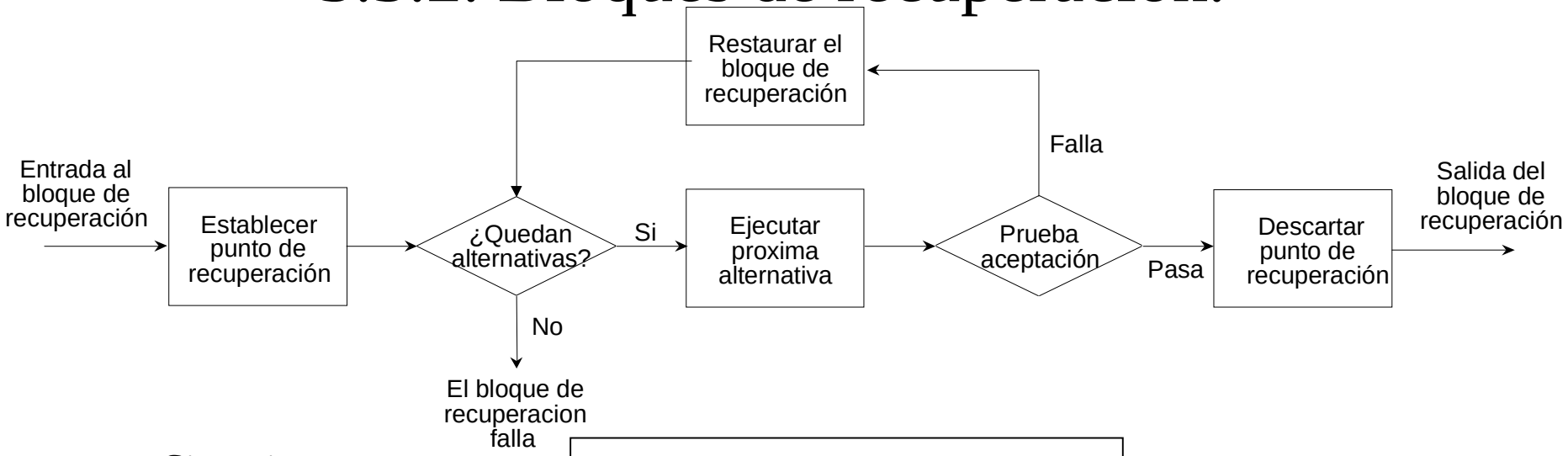
  2. La reparación del sistema:
  - Los componentes de hardware se sustituyen.
  - Los componentes de software se reparan haciendo una nueva versión.
  - En algunos casos puede ser necesario reemplazar el componente defectuoso sin parar el sistema.

- La recuperación automática es difícil y depende del sistema.

3.3.2. Bloques de recuperación.

- Es una técnica de recuperación inversa integrada en el lenguaje de programación.
- Un **bloque de recuperación** es un bloque tal que
  - Su entrada es un **punto de recuperación**.
  - A su salida se efectúa un **prueba de aceptación**:
    - Sirve para comprobar si el **módulo primario** del bloque termina en un estado correcto.
  - Si la prueba de aceptación falla,
    - Se restaura el estado inicial en el punto de recuperación.
    - Se ejecuta un **módulo alternativo** del mismo bloque.
  - Si vuelve a fallar, se siguen intentando alternativas.
  - Cuando no quedan más alternativas, el bloque falla y hay que intentar la recuperación en un nivel más alto.

3.3.2. Bloques de recuperación.



- Sintaxis:

```
ensure <condición_de_aceptación>
by <módulo_primario>
else by <módulo_alternativo>
else by <módulo_alternativo>
. . .
else by <módulo_alternativo>
else error;
```

- Puede haber bloques anidados: si falla el bloque interior, se restaura el punto de recuperación del bloque exterior.

3.3.2. Bloques de recuperación.

- Ejemplo:

```
ensure error <= tolerance
by
    Explicit_Runge_Kutta;
else by
    Implicit_Runge_Kutta;
else error;
```

- El método explícito es más rápido, pero no es adecuado para algunos tipos de ecuaciones.
- El método implícito sirve para todas las ecuaciones, pero es más lento.
- Este esquema sirve para todos los casos.

3.3. Redundancia estática y dinámica.

- Comparación entre la programación con N versiones y los bloques de recuperación:

N versiones	Bloques de recuperación
° Redundancia estática	° Redundancia dinámica
° Diseño – Algoritmos alternativos – Proceso guía	° Diseño – Algoritmos alternativos – Prueba de aceptación
° Ejecución – Múltiples recursos	° Ejecución – Puntos de recuperación
° Detección de errores – Votación	° Detección de errores – Prueba de aceptación

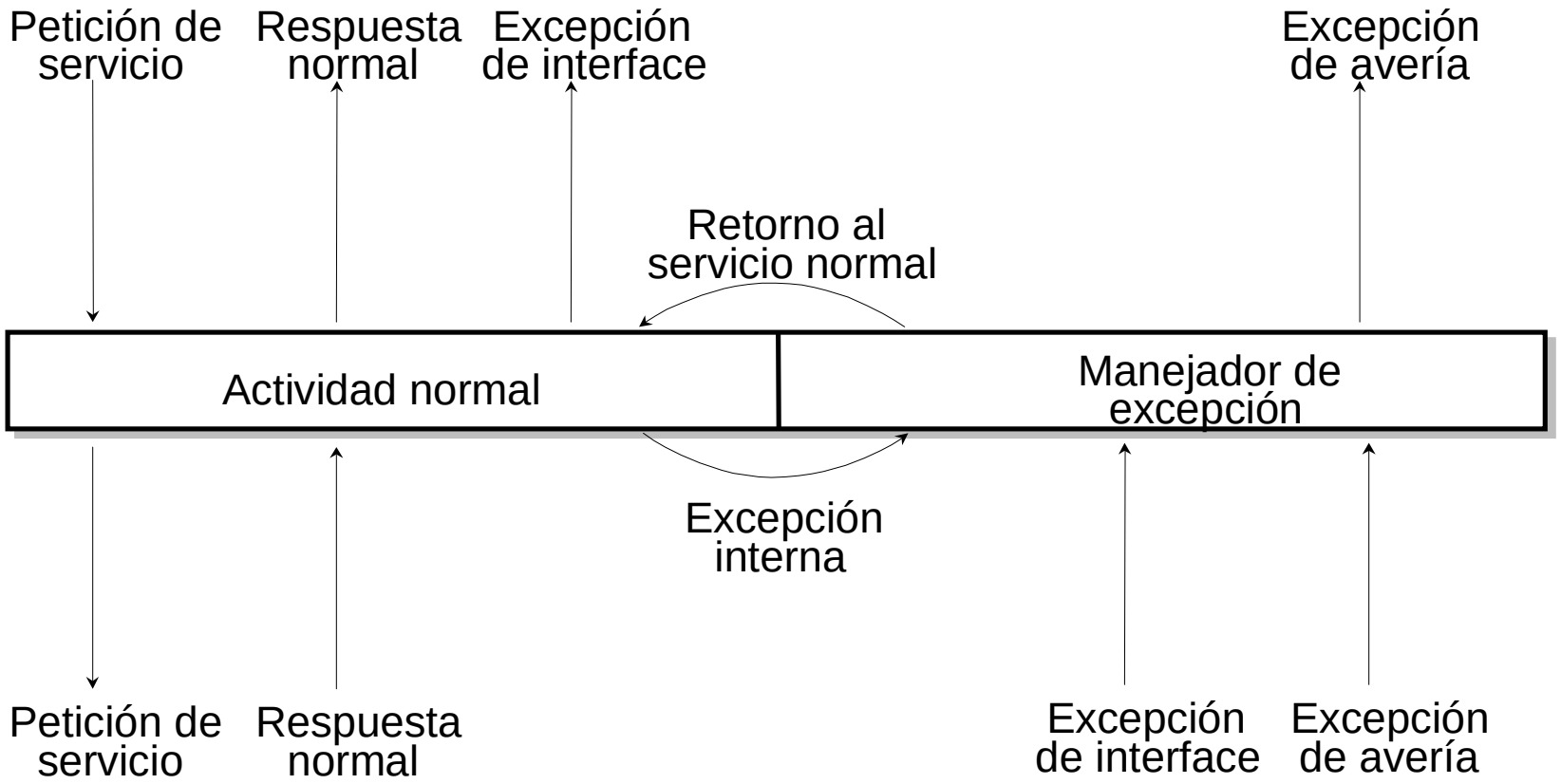
- Tanto programación con N versiones como bloques de recuperación son sensibles a los errores de especificación.

3.4. Redundancia dinámica y excepciones.

- Una **excepción** es una manifestación de un cierto tipo de error.
- Cuando se produce un error, se **eleva** la excepción correspondiente en el contexto donde se ha invocado la actividad errónea.
- Esto permite **manejar** la excepción en este contexto.
- Se trata de un mecanismo de recuperación directa de errores, aunque se puede utilizar también para la recuperación inversa de errores.
- Las excepciones pueden ser utilizadas por el programador para:
  - Tratar situaciones anormales en el sistema controlado.
  - Tolerar fallos de diseño del software.
  - Facilitar un mecanismo generalizado de detección y corrección de errores.

3.4. Redundancia dinámica y excepciones.

• Componente ideal de un sistema con tolerancia de fallos:



TEMA 4. EXCEPCIONES Y MANEJO DE EXCEPCIONES.

4.1.Introducción.

- Un mecanismo de manejo de excepciones debe cumplir una serie de requerimientos generales:

  - **(R1**) El mecanismo debe ser simple de usar y entender.
  - **(R2)** Separación del código para el manejo de las excepciones del código normal, de otro modo el código resultaría difícil de entender y mantener, lo cual podría conducir a sistemas menos fiables.
  - **(R3)** Las sobrecargas de tiempo de ejecución se producen sólo cuando se maneja una excepción.
  - **(R4)** Tratamiento uniforme de las excepciones detectadas por el entorno y por la propia aplicación. Por ejemplo, un overflow aritmético debería ser manejada del mismo modo que una excepción disparada por la aplicación.
  - **(R5)** Debe permitir que las acciones de recuperación sean programadas.

4.2. Tratamiento de excepciones.

4.2.1. Excepciones en lenguajes tradicionales.

- **Retorno de un valor inusual:**
  - Ejemplo C:

```
if (function_call(parameters) == AN_ERROR) {
    /* Código para el manejo del error */
} else {
    /* Código de retorno normal */
};
```

  - Veamos las propiedades anteriores:
    - Cumple la propiedad de la simplicidad (**R1**).
    - Permite que las acciones de recuperación sean programadas (**R5**).
    - El código no está separado de la actividad normal (no cumple **R2**).
    - Hay sobrecarga aunque no se produzcan errores (no cumple **R3**).
    - No está claro la forma de manejar los errores detectados por el entorno (no cumple **R4**).

4.2. Tratamiento de excepciones.

4.2.1. Excepciones en lenguajes tradicionales.

- **Bifurcación forzada:**
  - Es una técnica de muy bajo nivel que se utiliza en lenguaje ensamblador.
  - La rutina (bloque) manipula el PC (contador de programa) en función de los errores encontrados.

```
call    rutina
jmp     error_1
jmp     error_2
; Procesamiento normal
```

  - Veamos las propiedades anteriores:
    - Difícil de comprender y usar (no cumple **R1**).
    - El código no está separado de la actividad normal (no cumple **R2**).
    - Incurre en una sobrecarga muy pequeña (**R3**).
    - No está claro la forma de manejar los errores detectados por el entorno (no cumple **R4**).
    - Permite programar las acciones de recuperación (**R5**).

4.2. Tratamiento de excepciones.

4.2.1. Excepciones en lenguajes tradicionales.

- **Otros métodos:** Versión en los lenguajes de alto nivel del método de la bifurcación forzada.

Salto incondicional no local

```
svc data rrerr;
  label erl;
  ...
enddata;
proc WhereErrorIsDetected();
  ...
  goto erl;
  ...
endproc;
proc Caller();
  ...
 WhereErrorIsDetected();
  ...
endproc;
proc main();
  ...
  erl := restart;
  ...
  Caller();
  ...
  restart:
  ...
end proc;
```

Variable de procedimiento

```
svc data rrerr;
  label erl;
  proc(int) erp;
enddata;
proc recover(int); ... endproc;
proc WhereErrorIsDetected();
  ...
  if recoverable then erp(n)
  else goto erl end;
  ...
endproc;
proc Caller();
  ...
 WhereErrorIsDetected();
  ...
endproc;
proc main();
  ...
  erl := fail;
  erp := recover;
  ...
  Caller();
  ...
fail: ...
end proc;
```

4.2. Tratamiento de excepciones.

4.2.2. Tratamiento de excepciones moderno.

- Las anteriores aproximaciones para el tratamiento de las excepciones provocan que el código de recuperación se mezcle con el código de ejecución normal (código no estructurado).
- La aproximación moderna consiste en introducir el mecanismo de manejo de excepciones directamente en el lenguaje (código estructurado).
- Tipos de excepciones:
  - Detectadas por el entorno y elevadas síncronamente. Por ejemplo, una división por cero.
  - Detectadas por la aplicación y elevadas síncronamente. Por ejemplo, al hacer un chequeo la aplicación obtiene un valor no esperado.

Señales (signals):

  - Detectadas por el entorno y elevadas asíncronamente. Por ejemplo, una excepción elevada como resultado de un fallo de alimentación.
  - Detectas por la aplicación y elevadas asíncronamente. Por ejemplo, un proceso podría reconocer que se ha producido una condición de error, la cual ha provocado que otro proceso no cumpla sus plazos.

4.2. Tratamiento de excepciones.

4.2.2. Tratamiento de excepciones moderno.

- En Ada las excepciones tienen que ser declaradas como constantes.
- Por ejemplo, las excepciones que puede elevar el núcleo de ejecución (excepciones predefinidas) están declaradas en el paquete *Standard*:

```
package Standard is
    . . .
    Constraint_Error : exception;
    Program_Error : exception;
    Storage_Error : exception;
    Tasking_Error : exception;
    . . .
end Standard;
```

- Dentro de un programa podría haber varios manejadores para una determinada excepción. Asociado a cada manejador hay un **dominio** que especifica la región de código en la que será activado cuando se detecta la excepción. En Ada el dominio es el bloque.
- La exactitud con la que se puede especificar un dominio determinará la precisión con la que se puede localizar la fuente de la excepción.

4.2. Tratamiento de excepciones.

4.2.2. Tratamiento de excepciones moderno.

- Por ejemplo, supongamos un sensor de temperatura con valores comprendidos entre 0 y 100ºC. Si el valor de temperatura calculado cae fuera de este rango, el núcleo de ejecución de Ada eleva la excepción *Constraint_Error*:

```
declare
    subtype Temperatura is Integer range 0..100;
begin
    -- Leer la temperatura del sensor y calcular su valor
    . . .
exception
    -- manejador para Constraint_Error
end;
```

- Algunas veces no es posible determinar la causa de la excepción:

```
declare
    subtype Temperatura is Integer range 0..100;
    subtype Presión is Integer range 0..50;
begin
    -- Leer el sensor de temperatura y calcular su valor
    -- Leer el sensor de presión y calcular su valor
    . . .
exception
    -- manejador para Constraint_Error
end;
```

4.2. Tratamiento de excepciones.

4.2.2. Tratamiento de excepciones moderno.

- Solución (1): reducir el tamaño de los bloques y aumentar el número de los mismos.

```
declare
    subtype Temperatura is Integer range 0..100;
    subtype Presión is Integer range 0..50;
begin
    begin
       -- Leer el sensor de temperatura y calcular su valor
    exception
       -- Manejador para Constraint_Error para la temperatura
    end;
    begin
       -- Leer el sensor de presión y calcular su valor
    exception
       -- Manejador para Constraint_Error para la presión
    end;
    . . .
exception
    -- manejador para otras excepciones
end;
```

- Problema: la realización es larga y pesada.

4.2. Tratamiento de excepciones.

4.2.2. Tratamiento de excepciones moderno.

- Solución (2): permitir manejadores de excepciones asociados a instrucciones.
  No se puede realizar en Ada.

```
declare
    subtype Temperatura is Integer range 0..100;
    subtype Presión is Integer range 0..50;
begin
    Leer_sensor_temperatura;
       exception -- Manejador para Constraint_Error para la temperatura
    Leer_sensor_presión;
       exception -- Manejador para Constraint_Error para la presión
    . . .
end;
```

- Permite localizar de forma más precisa la causa de la excepción.
- Problema: No cumple R2.

- La mejor solución es que el entorno pase parámetros indicando el punto donde se han elevado las excepciones a los manejadores.

4.2. Tratamiento de excepciones.

4.2.2. Tratamiento de excepciones moderno.

Propagación de excepciones

- Cuando no existe un manejador local para una excepción, se busca un manejador a lo largo de la cadena de invocadores en tiempo de ejecución.
- Problema: Una excepción se puede propagar fuera de su ámbito de visibilidad.
- Solución: Manejadores por defecto para excepciones desconocidas (*catch all)*.
- Una excepción no tratada provoca que un programa secuencial sea abortado. Si el programa contiene más de un proceso, alguno de los cuales no trata una excepción elevada por él mismo, entonces dicho proceso es abortado.

Reanudación y terminación

- Cuando el manejador termina se pueden hacer dos cosas:
  - **Reanudar** la ejecución del bloque.
  - **Terminar** la ejecución del bloque y devolver el control al punto de invocación.

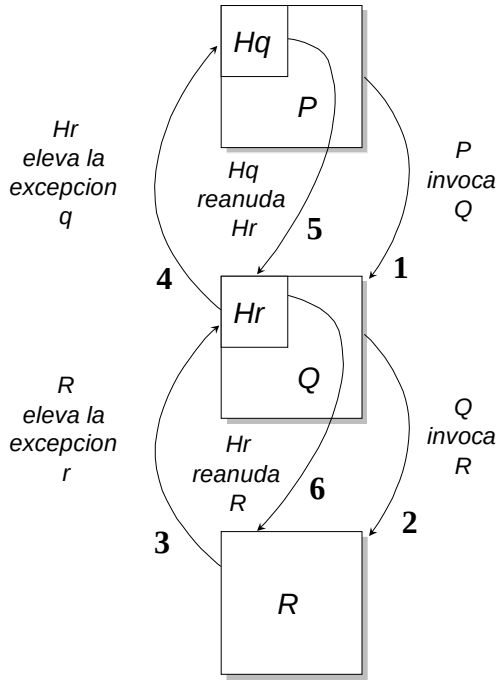
→ Podría adoptarse un modelo híbrido

4.2. Tratamiento de excepciones.

4.2.2. Tratamiento de excepciones moderno.

a) Modelo de reanudación

- El manejador de excepción puede ser visto como un procedimiento que es llamado cuando la excepción es elevada.



Problema: En ocasiones es difícil reparar los errores detectados por el entorno de ejecución. Ejemplo: un desbordamiento aritmético al evaluar expresiones complejas que utilizan algunos registros para almacenar resultados parciales. Como consecuencia de la llamada al manejador puede modificarse el valor de esos registros.

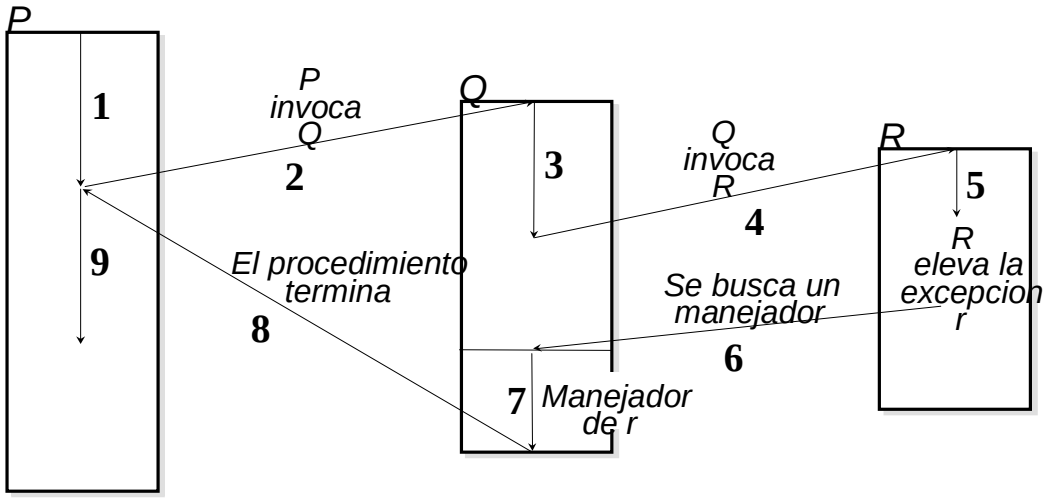
Aunque una implementación del modelo de reanudación estricto es difícil, un compromiso es volver a ejecutar el bloque desde el principio.

4.2. Tratamiento de excepciones.

4.2.2. Tratamiento de excepciones moderno.

b) Modelo de terminación

- Cuando una excepción ha sido elevada y el manejador ha sido llamado, el control no es devuelto al punto en el que ocurrió la excepción, sino que el bloque o procedimiento que contiene el manejador es finalizado y el control es pasado al bloque o procedimiento llamador.



- Un procedimiento invocado podría terminar normalmente o como consecuencia de una excepción.

4.2. Tratamiento de excepciones.

4.2.2. Tratamiento de excepciones moderno.

Manejo de excepciones y sistemas operativos.

- En muchos casos, los programas escritos utilizando un lenguaje con la facilidad de las excepciones (como Ada), serán ejecutados en un sistema operativo como DOS o POSIX.
- Estos sistemas operativos detectarán ciertas condiciones de error síncronas, como por ejemplo instrucción ilegal o violación de memoria. Esto provocará, en sistemas operativos convencionales, que el proceso en ejecución sea finalizado.
- POSIX permite que la aplicación maneje las excepciones (mediante **señales**).
- Cuando el manejador termina se reanuda el proceso (modelo de reanudación).
- En lenguajes como Ada, en los que se proporciona el modelo de terminación, es responsabilidad del núcleo de ejecución del lenguaje el capturar el error y realizar la manipulación necesaria del estado del programa de forma que el programador pueda usar este modelo.

4.3. Tratamiento de excepciones en Ada.

- El lenguaje de programación Ada soporta:
  - Declaración explícita de excepciones.
  - El modelo de terminación de manejo de excepciones con propagación de excepciones no manejadas.
  - Una forma limitada de paso de parámetros de excepción.

Declaración y elevación de excepciones

```
package body Pila is
   Max : constant:= 100;
   p : array (1..Max) of Integer;
   top : Integer range 0..Max;
   procedure Push( x : Integer) is begin
            top:= top+1;
            p(top):= x;
   end Push;
   function Pop return Integer is begin . . . end Pop;
begin
   Top:= 0;
end Pila;
```

4.3. Tratamiento de excepciones en Ada.

- Si creamos un procedimiento llamado *Principal* que invoque a Push cuando la pila esté llena, el incremento de *top* producirá *Constraint_Error*. Ya que Push no maneja excepciones, la excepción se propagará al contexto del invocante.
- Podemos manejar la excepción en *Principal*:

```
with Pila;
procedure Principal is
   . . .
begin
   . . .
exception
   when Constraint_Error => -- ¿desbordamiento de pila?
end Principal;
```

- ¿Podemos asegurar que cuando se eleve *Constraint_Error* en *Principal* es siempre debido a un desbordamiento de pila?
- Mejor declarar nuevas excepciones.

4.3. Tratamiento de excepciones en Ada.

```
package Pila is
  Error : exception;
  procedure Push(X : Integer);
  function Pop return Integer;
end Pila;
```

```
Package body Pila is
  . . .
  procedure Push(X : Integer) is
  begin
    if Top = Max then
      raise Error;
    end if;
    . . .
  end Push;
  function Pop return Integer is
  begin
    if Top = 0 then
      raise Error;
    end if;
     . . .
  end Pop;
begin
  Top := 0;
end Pila;
```

```
with Pila;
procedure Principal is
  ...
begin
  ...
exception
  when Pila.Error => -- Pila incorrecta
    Put("Desbordamiento de Pila");
  when others => -- Otra cosa fue mal
    Put("Excepción no prevista en Pila");
end Principal;
```

- Usamos la cláusula **raise** para elevar la excepción.
- Hemos aislado el manejo de errores en la manipulación de la pila de otros que pudieran presentarse en una operación del paquete.

4.3. Tratamiento de excepciones en Ada.

Manejo de excepciones

- En Ada cada bloque puede contener una colección opcional de manejadores de excepciones y son declarados al final del bloque.
- Cada manejador es una secuencia de instrucciones. Esta secuencia aparece precedida de:
  - Palabra clave **when** + un parámetro opcional denominado **ocurrencia** terminado en dos puntos + los nombres de las excepciones manejadas + el símbolo =>.
- Ejemplo:

```
declare
    Sensor_Alto, Sensor_Bajo, Sensor_Muerto : exception;
begin
    -- sentencias que podrían elevar las excepciones anteriores.
exception
    when E: Sensor_Alto | Sensor_Bajo => . . . -- E contiene la ocurrencia
    when Sensor_Muerto => . . .
    when others => . . .
end;
```

4.3. Tratamiento de excepciones en Ada.

Propagación de excepciones

- Si no hay manenador en el bloque o cuerpo donde se eleva una excepción, se termina éste y se propaga al nivel superior (bloque exterior o punto de invocación).

- También se propagan las excepciones elevadas en los manejadores.

- Las excepciones de la secuencia inicial de un paquete se propagan a la tarea de entorno que las invoca antes de llamar al programa principal.

- Si una excepción se propaga fuera de su ámbito de visibilidad puede ser capturada utilizando un manejador por defecto (*when others*...).

- Si queda alguna excepción sin manejar en la tarea de entorno, el programa será finalizado.

4.3. Tratamiento de excepciones en Ada.

Re-elevación de excepciones

- En algunas ocasiones, las acciones que se realizan como consecuencia de una excepción necesitan ser estructuradas en 2 niveles:
  - Parte en el contexto en el que ocurrió la excepción.
  - Parte en el contexto del invocante.
- Para ello se emplea la cláusula **raise** sin nombrar la excepción.
- Ejemplo:

```
procedure A is
  E, F : exception;
  procedure B is  begin
    ...
    -- Bloque donde se eleva E
    ...
    -- Bloque donde se eleva F
    ...
  exception
    when E | F =>
      -- Acción común a E y F
      raise;
  end B;
begin
  ...
  B; -- En B puede ocurrir E y F
  ...
exception
  when E => -- Acción relativa a E
  when F => -- Acción relativa a F
end A;
```

4.3. Tratamiento de excepciones en Ada.

Ocurrencia de una excepción

- Ada proporciona la información sobre una determinada excepción en un objeto del tipo *Exception_Ocurrence* denominado **ocurrencia** de la excepción:

  When Capturada: **Others** => . . .

```
package Ada.Exceptions is
  type Exception_Id is private;
  Null_Id : constant Exception_Id;
  function Exception_Name(Id : Exception_Id) return String;
  type Exception_Occurrence is limited private;
  Null_Occurrence : constant Exception_Occurrence;
  procedure Raise_Exception(E : in Exception_Id; Message : in  String = "");
  function Exception_Message(X : Exception_Occurrence) return String;
  procedure Reraise_Occurrence(X : in Exception_Occurrence);
  function Exception_Identity(X : Exception_Occurrence) return Exception_Id;
  function Exception_Name(X : Exception_Occurrence) return String;
  function Exception_Information(X : Exception_Occurrence) return String;
  procedure Save_Ocurrence(Target : out Exception_Occurrence; Source : in Exception_Occurrence);
  function  Save_Ocurrence(Source : Exception_Occurrence) return Exception_Occurrence_Access;
private  ... -- No especificado por el lenguaje end Ada.Exceptions;
```

4.3. Tratamiento de excepciones en Ada.

- *Exception_Name* : Devuelve el nombre del tipo de la excepción en notación de puntos completa en letras mayúsculas. Ejemplo: PILA.ERROR.
- *Exception_Message* : Devuelve la cadena pasada pasada con *Raise_Exception*. Si la excepción fue elevada con **raise**, la cadena contiene información definida por la implementación sobre la excepción.
- *Exception_Information* : Devuelve una cadena que contiene más información definida por la implementación que la anterior.
- Toda excepción declarada usando la cláusula **exception** tiene un *Exception_Id* asociado que puede obtenerse a través del atributo predefinido *Identity*.
- Ejemplo:

```
declare
    Stuck_Valve : exception;
    id : Ada.Exception.Exception_Id:= Stuck_Valve'Identity;
begin
    ... Raise_Exception(id, "parámetros");    ...
exception
    when capturada : Stuck_Valve =>
        put_line(Exception_Message(capturada));
end;
```

4.3. Tratamiento de excepciones en Ada.

Críticas al modelo de excepciones de Ada

- **Excepciones y paquetes**: Las excepciones elevadas dentro de un paquete se declaran en la especificación del mismo, pero no se dice nada acerca de las operaciones que pueden elevar cada una de las excepciones. Será el creador del paquete el que deberá indicar mediante comentarios las excepciones elevadas por cada operación.

- **Paso de parámetros**: Únicamente se puede pasar como parámetro una cadena de caracteres.

- **Ámbito y propagación**: Es posible propagar una excepción fuera de su ámbito, de manera que sólo puede ser manejada de forma anónima por una cláusula *when others*.

4.4. Tratamiento de excepciones en C.

- En C no hay ningún mecanismo de excepciones.

- Es posible, sin embargo, simular el mecanismo de excepciones (pero de forma limitada) haciendo uso de macros.

- Para implementar el modelo de terminación en C es necesario almacenar el estado de los registros de un programa cuando entramos en el ámbito de una excepción, y restaurarlos si se produce una excepción.

- Se puede hacer utilizando las rutinas POSIX:
  - *setjmp*: guarda el estado del programa y devuelve 0.
  - *longjmp*: restaura el estado del programa y hace que el programa se ejecute desde donde se llamó a *setjmp*. Esta vez *setjmp* devolverá los valores pasados por *longjmp*.

4.4. Tratamiento de excepciones en C.

- **Esquema**:

```
#define SENSOR_AVERIADO   1
 int current_exception;
 jmp_buf save_area;
. . .
if ((current_exception = setjmp(save_area)) == 0) {
    /*guarda los registros en save_area y devuelve 0 */

    /* región de código protegida por el manejador */

    /* cuando se identifica que el sensor no funciona*/
    longjmp(save_area, SENSOR_AVERIADO);
}
else {
    if (current_exception == SENSOR_AVERIADO) { /* manejador*/ }
    else { /* re-eleva la excepción en el ámbito superior */ }
}
```

4.4. Tratamiento de excepciones en C.

- Podemos proporcionar un conjunto de macros que ayuden a estructurar el programa:

```
#define BEGIN_GUARDED . . .
        /* Entrar en el dominio de una excepción */
#define EXCEPTION . . .
        /* Comienzo de los manejadores */
#define END_GUARDED . . .
        /* Salir del dominio de una excepción */
#define RAISE(id) . . .
        /* Código para elevar una excepción */
#define WHEN(id) . . .
        /* Código para el manejador */
#define WHEN_OTHERS . . .
        /* Código para capturar todas las excepciones*/
```

4.5. Excepciones y bloques de recuperación.

- Es posible implementar los bloques de recuperación haciendo uso de excepciones y manejadores de excepciones.
- Partimos de la siguiente estructura:

```
ensure <acceptance_test>
by
    <primary_module>
else by
    <alternative_module>
. . .
else by
    <alternative_module>
else error;
```

- Mediante la prueba de aceptación se realiza la detección de errores.
- Necesitamos guardar y recuperar el estado en el punto de recuperación: Suponemos un paquete de biblioteca con los procedimientos **save** (almacena el estado) y **restore** (recupera el estado).

4.5. Excepciones y bloques de recuperación.

```
procedure Recovery_Block is
  f_prim, f_sec, f_terc, f_block : exception;
  type Module is (primary, secondary, tertiary);
  function Acceptance_Test return boolean is begin
    -- Código para el test de aceptación
  end Acceptace_Test;

  procedure Primary is begin
    -- código para el algoritmo primario
    if not Acceptance_Test then
      raise f_prim;
    end if;
  exception
    when f_prim => -- asegurar el entorno
      raise;
    when others => -- asegurar el entorno
      raise f_prim;
  end Primary;

  procedure Secondary is begin
    -- Igual que Primary pero eleva f_sec
  end Secondary;

  procedure Tertiary is begin
    -- Igual que Primary pero eleva f_terc
  end Tertiary;
begin
  Recovery_Cache.Save;
  for try in Module loop
    begin
      case try is
        when Primary => Primary; exit;
        when Secondary => Secondary; exit;
        when Tertiary => Tertiary;
      end case;
    exception
      when f_prim => Recovery_Cache.Restore;
      when f_sec => Recovery_Cache.Restore;
      when f_terc => Recovery_Cache.Restore;
                     raise f_block;
      when others => Recovery_Cache.Restore;
                     raise f_block;
    end;
  end loop;
end Recovery_Block;
```

TEMA 5. PROGRAMACIÓN CONCURRENTE.

5.1. Ejecución concurrente.

5.2. Representación de procesos.

5.3. Tareas en Ada.

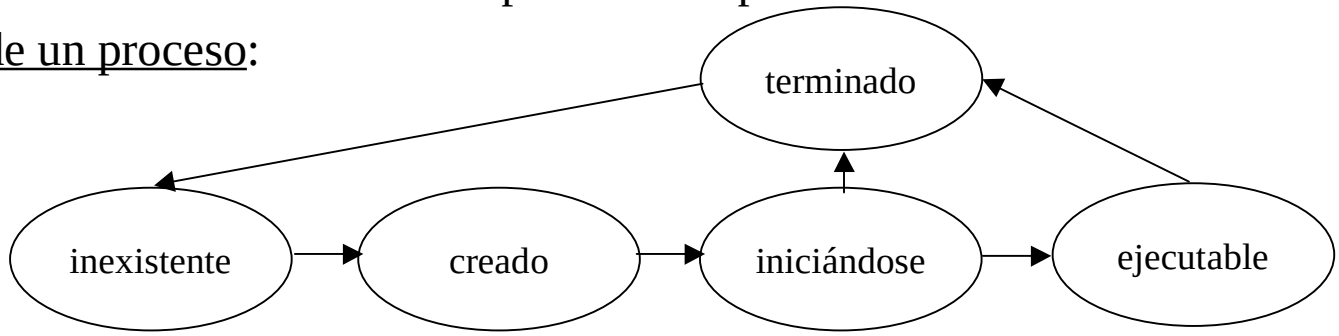
5.4. Ejemplo.

5.5. Comunicación y sincronización con variables comunes.

5.6. Comunicación y sincronización mediante mensajes.

5.1. Ejecución concurrente.

- Un programa concurrente está formado por una colección de procesos secuenciales autónomos que se ejecutan (aparentemente) en paralelo.
- Podemos distinguir tres formas de ejecutar una colección de procesos concurrentes:
  1. Los procesos multiplexan sus ejecuciones sobre un único procesador (multiprogramación).
  2. Los procesos multiplexan sus ejecuciones sobre un sistema multiprocesador de memoria compartida (multiproceso).
  3. Los procesos multiplexan sus ejecuciones en varios procesadores que no comparten memoria (procesamiento distribuido).
- El término **concurrencia** indica paralelismo potencial.
- Vida de un proceso:



5.1. Ejecución concurrente.

- Los procesos concurrentes se ejecutan con ayuda de un **núcleo de ejecución** (*Run-Time Kernel*) que posee muchas de las propiedades del planificador en un sistema operativo y que lógicamente está situado entre el hardware y el software de la aplicación.
- El núcleo se encarga de la creación, terminación y el multiplexado de los procesos.
- El núcleo puede tomar varias formas:
  - Núcleo desarrollado como parte de la aplicación.
  - Núcleo incluido en el entorno de ejecución del lenguaje.
  - Núcleo de un sistema operativo tiempo real.
  - Núcleo microprogramado dentro del procesador para mayor eficiencia.
- El algoritmo de planificación utilizado por el núcleo de ejecución afectará al comportamiento temporal del programa.
- Desde el punto de vista del programador, el núcleo de ejecución planifica los procesos de una forma no determinista.

5.1. Ejecución concurrente.

- Cuando hablamos de procesos debemos distinguir:
  - Procesos *pesados*: cada uno se ejecuta en su máquina virtual. Cada proceso puede ser visto como un programa independiente.
  - Procesos *ligeros* (**threads**): todos los *threads* de un mismo proceso comparten la misma máquina virtual. Tienen acceso al mismo espacio de memoria.
- La concurrencia puede estar soportada por el lenguaje de programación o sólo por el sistema operativo. El soportar la concurrencia en los lenguajes de programación tiene una serie de argumentos a favor, como por ejemplo
  - Conduce a programas más legibles y fáciles de mantener.
  - Un STR podría no disponer de un sistema operativo.

  Pero también algunos en contra:
  - Dificultad al implementar de forma eficiente el modelo de concurrencia del lenguaje sobre el modelo de concurrencia del sistema operativo.
- Estamos interesados en el estudio de la concurrencia soportada por el lenguaje de programación.

5.1. Ejecución concurrente.

Lenguajes concurrentes

- Los lenguajes de programación concurrentes proporcionan elementos para:
  - La expresión de la ejecución concurrente a través de la noción de proceso.
  - La sincronización de procesos.
  - La comunicación entre procesos.
- Teniendo en cuenta la relación entre los procesos, estos pueden:
  - Ser independientes: no se comunican ni sincronizan.
  - Cooperar para un fin común: regularmente se comunican y sincronizan sus actividades para realizar alguna operación común.
  - Competir por el uso de recursos: se comunican y sincronizan con el fin de obtener los recursos compartidos.
- Los modelos de concurrencia de los distintos lenguajes de programación difieren respecto a:
  - Estructura
  - Nivel
  - Granularidad
  - Inicialización
  - Terminación
  - Representación

5.1. Ejecución concurrente.

Características de los lenguajes concurrentes

- Estructura
  - Estática: número de procesos fijo y conocido en tiempo de compilación.
  - Dinámica: número de procesos conocido sólo en tiempo de ejecución.
- Nivel de paralelismo
  - Anidado (nested): procesos definidos a cualquier nivel del programa.
  - Plano (flat): procesos definidos sólo al nivel *externo* del programa.
- Granularidad del paralelismo
  - Grano grueso (coarse grain): pocos procesos pero de *larga vida*.
  - Grano fino (fine grain): gran número de procesos simples.
- Inicialización

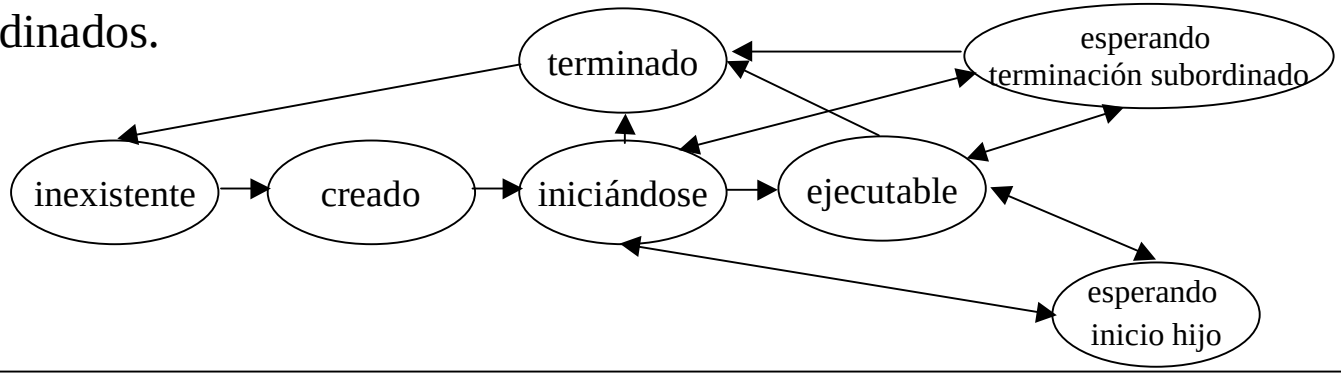
  Paso de parámetros o comunicación explícita.
- Terminación

- al completar la ejecución
- suicidio
- aborto
- excepción sin manejar
- nunca
- cuando no se necesita más

5.1. Ejecución concurrente.

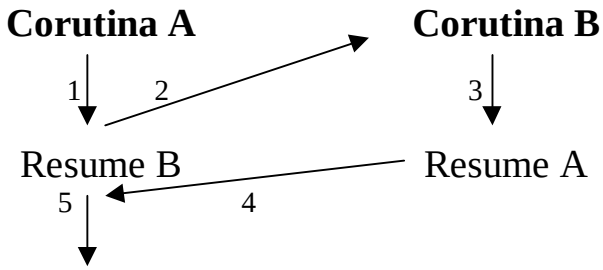
Jerarquía de procesos

- Con niveles de procesos anidados se pueden crear jerarquías de procesos.
- Para cualquier proceso es útil distinguir:
  - El proceso (o bloque) que es el responsable de su creación. Relación **padre/hijo**.
  - El proceso (o bloque) que se ve afectado por su terminación. Relación **guardián/subordinado**.
- El guardián puede ser:
  - el padre, otro proceso o un bloque interno de uno de los anteriores
- El guardián no puede terminar hasta que no lo hayan hecho todos sus subordinados.



5.2. Representación de procesos.

- La ejecución concurrente en un lenguaje de programación puede representarse:
  - Corrutinas



  - fork/join

```
pid = fork();      /* Se crea un proceso idéntico al original */
   . . .
c = join();        /* El primero que llega espera al otro */
   . . .
```

  - Cobegin/coend

```
cobegin
  S1; S2; S3; . . . Sn
coend
```

  - Declaración explícita de procesos
    - Los procesos son unidades de programa (como los procedimientos).
    - Permite una mejor estructura.
    - Ejemplo: Ada.

5.3. Tareas en Ada.

- **Tarea**: es el nombre que reciben las actividades concurrentes en Ada.
- La tarea, como unidad de programa que es, tiene dos partes:
  - Especificación: define la interfaz visible de la tarea.
  - Cuerpo: define la actividad de la tarea.
- Las tareas se crean implícitamente cuando se entra en el ámbito de su declaración, y comienzan a ejecutarse antes que el cuerpo correspondiente a la parte declarativa donde se declaran.

```
Procedure Ejemplo1 is
    Task A;
    Task body A is
   --declaraciones
    begin    ...
    end A;
begin    -- A comienza su ejecución aquí antes que la primera
    ...   -- sentencia de Ejemplo_1
end Ejemplo_1; -- Ejemplo1 no retorna mientras no termine A
```

5.3. Tareas en Ada.

El tipo tarea

- En ocasiones es útil tener un grupo de tareas similares.
- La declaración de un tipo tarea permite disponer de una plantilla para crear tareas similares.
- Las tareas que se declaran directamente (como antes) son de un **tipo anónimo**.

```
task type Tipo_A;
A, B : Tipo_A;
type Long is array (1..100) of Tipo_A;
type Mixture is record
    Indice : Integer;
    Accion : Tipo_A;
end record;
L : Long; M : Mixture;
task body Tipo_A is
     . . .
end Tipo_A;
```

5.3. Tareas en Ada.

El tipo tarea

- Ejemplo

```
procedure Robot is
    type Dimension is (Plano_X, Plano_Y, Plano_Z);
    task type Control (Dim : Dimension);
    C1 : Control(Plano_X); C2 : Control(Plano_Y); C3 : Control(Plano_Z);
    task body Control is
      Posicion, Setting : Integer;          -- Posición absoluta y movimiento relativo
    begin
      Position:= 0;              -- Posición de reposo
      loop
        Calcula_Movimiento(Dim, Setting);
        Posicion:= Posicion + Setting;
        Mueve_Brazo(Dim, Posicion);
      end loop;
    end Control;
begin
    null;
end Robot;
```

5.3. Tareas en Ada.

Tareas dinámicas

- La creación dinámica de tareas puede ser efectuada explícitamente utilizando el operador **new** sobre un tipo acceso (acceso a un tipo tarea):

```
procedure Ejemplo2 is
    task type T;
    type A is access T;
    P : A;
    Q : A:= new T;
begin
    . . .
    P:= new T;
    Q:= new T;
    . . .
end Ejemplo2;
```

- Existen tres tareas activas: P.**all**, Q.**all** y la primera tarea que fue creada.
- Esta tarea se convierte en anónima al reasignar su puntero.

5.3. Tareas en Ada.

Tareas dinámicas

- Para las tareas creadas con **new** el bloque que actúa como guardián (o maestro, que es como se denomina en Ada) es aquel que contiene la declaración del tipo acceso.

```
declare
    task type T;
    type A is access T;
begin
 . . .
 declare     -- bloque interno
  X : T;
  Y : A:= new T;
 begin
  -- secuencia de instrucciones
 end;       -- Tiene que esperar a que termine X pero no Y.all
 . . .      -- Y.all podría estar activa todavía, aunque el nombre Y está fuera de ámbito
end;        -- Tiene que esperar a que termine Y.all
```

5.3. Tareas en Ada.

Tareas y excepciones

- Si una tarea falla mientras está siendo inicializada el padre de la tarea recibe la excepción *Tasking_Error*.
- Cuando la tarea comienza su ejecución recibe cualquier excepción que eleve.

Identificación de tareas

- El tipo **task** es **limited private**:

  ~~Robot_Arm.**all**:= New_Arm.**all**;~~ ⟶ Robot_Arm:= New_Arm;

- El anexo de programación de sistemas de Ada proporciona un mecanismo por el que cada tarea puede obtener su propia identificación y ésta puede ser pasada a otras tareas.

```
package Ada.Task_Identification is
  type Task_Id is private;
  Null_Task_Id : constant Task_Id;
  function Current_Task return Task_Id;   --Devuelve un identificador único
  ...  private   ...
end Ada.Task_Identification;
```

- El paquete soporta los atributos *T'Identity* y *E'Caller*.

5.3. Tareas en Ada.

Terminación de tareas

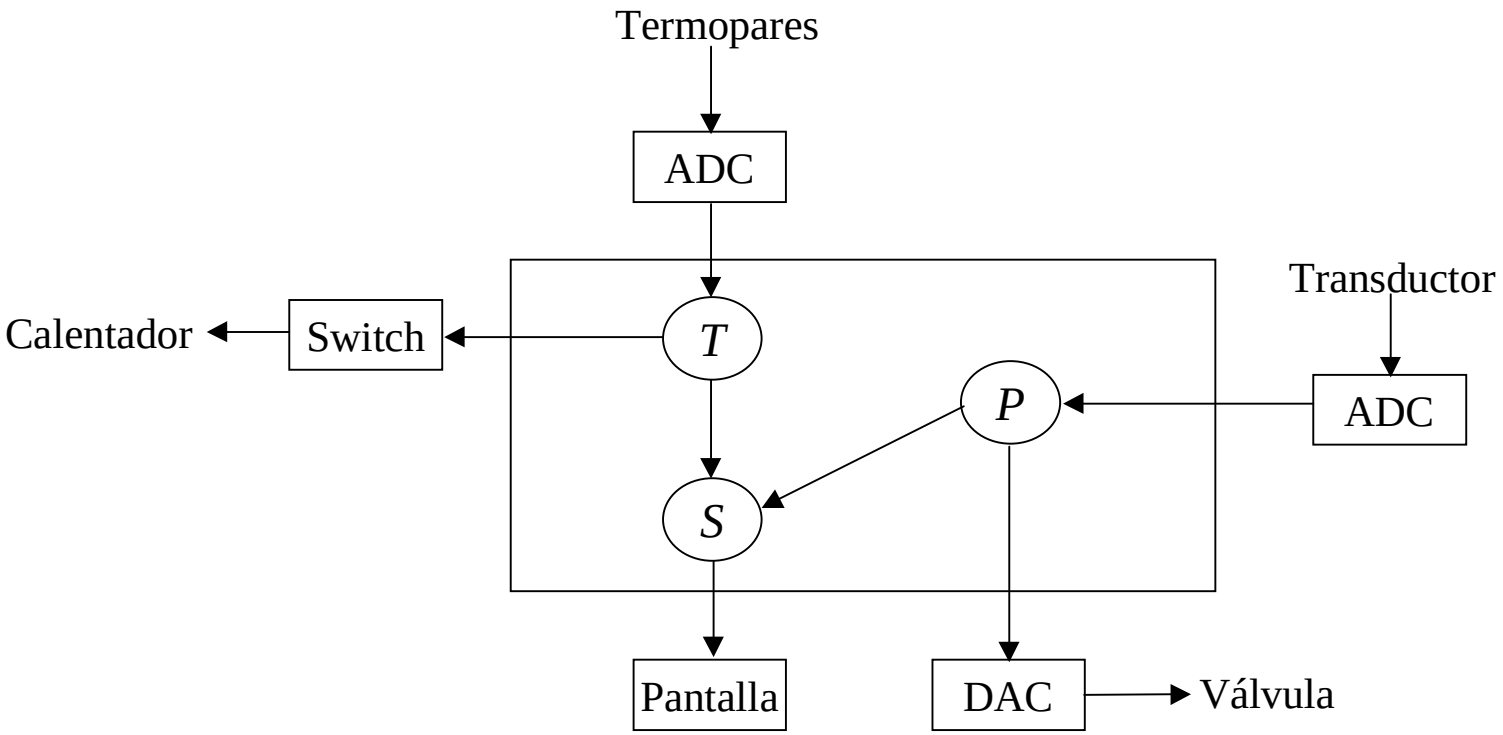
- Una tarea termina cuando:
  - Completa la ejecución de su cuerpo (o bien normalmente o bien como resultado de una excepción no manejada).
  - Ejecuta una alternativa **terminate** en una instrucción **select** (se verá después).
  - Es abortada.
- Una tarea puede comprobar si otra ha terminado:

```
if T'Terminated then        -- para alguna tarea T
   . . .
end if;
```

- Pero no puede saber si terminó normalmente o como consecuencia de un error.
- Una tarea puede abortar otra cuyo nombre es visible: **abort** T;
- Cuando se aborta una tarea, se abortan también todos sus subordinados.
- Las tareas anónimas no se pueden abortar.
- Se puede abortar una tarea identificada mediante *Ada.Task_Identification.Abort_Task*.

5.4. Ejemplo.

Sistema empotrado sencillo



- El objetivo global del sistema es el de mantener la temperatura y presión de algún proceso químico dentro de unos límites preestablecidos.

5.4. Ejemplo.

Sistema empotrado sencillo

- Solución 1: Se utiliza un único programa que ignora la concurrencia lógica de T, P y S => No es necesario sistema operativo.

```
procedure Controlador is
  TR : Temp_Leida;
  PR : Presion_Leida;
  HS : Valor_Calentador;
  PS : Valor_Presion;
  Lista_Temp, Lista_Presion : Boolean;
begin
  loop
     . . .
     if Lista_Temp then
        Leer(TR);
        Convertir_Temp(TR, HS);
        Escribir(TR);
        Escribir(HS);
     end if;
     if Lista_Presion then
        Leer(PR);
        Convertir_Presion(PR, PS);
        Escribir(PR);
        Escribir(PS);
     end if;
  end loop;
end Controlador;
```

- Problema: Se basa en una espera ocupada. Además ambas actividades deberían estar desacopladas.

5.4. Ejemplo.

Sistema empotrado sencillo

- Solución 2: Tres procedimientos secuenciales ejecutados como procesos de un sistema operativo.

```
procedure Controlador is
  Controlador_Temp : Thread_ID;
  Controlador_Pres : Thread_ID;
  procedure Controlador_Temperatura is
    TR : Temp_Leida; HS : Valor_Calentador;
  begin
    loop
      Leer(TR);
      Convertir_Temp(TR, HS);
      Escribir(TR);
      Escribir(HS);
    end loop;
  end Controlador_Temperatura;
  procedure Controlador_Presion is
    PR : Presion_Leida; PS : Valor_Presion;
  begin
    loop
      Leer(PR);
      Convertir_Temp(PR, PS);
      Escribir(PR);
      Escribir(PS);
    end loop;
  end Controlador_Presion;
begin
  -- Crear los dos procesos
end Controlador;
```

- Problema: No hay diferencia entre procedimientos y procesos concurrentes => dificulta la creación y mantenimiento de sistemas grandes.

5.4. Ejemplo.

Sistema empotrado sencillo

- Solución 3: Con un único programa en un lenguaje concurrente => no hace falta sistema operativo pero sí un núcleo de ejecución.

```
procedure Controlador is
  task Controlador_Temp;
  task Controlador_Pres;
  task body Controlador_Temp is
    TR : Temp_Leida; HS : Valor_Calentador;
  begin
    loop
      Leer(TR);
      Convertir_Temp(TR, HS);
      Escribir(TR);
      Escribir(HS);
    end loop;
  end Controlador_Temp;
  task body Controlador_Pres is
    PR : Presion_Leida; PS : Valor_Presion;
  begin
    loop
      Leer(PR);
      Convertir_Temp(PR, PS);
      Escribir(PR);
      Escribir(PS);
    end loop;
  end Controlador_Pres;
begin
  null;
end Controlador;
```

- Problema: Ambas tareas hacen uso de la pantalla => hace falta gestionar el acceso a este recurso compartido.

5.5. Comunicación y sincronización con variables comunes.

- Hay veces en las que los procesos **cooperan** para un fin común o **compiten** por la utilización de recursos.
- Para ello es necesario realizar operaciones de **comunicación** y **sincronización** entre procesos:
  - Dos procesos se **comunican** cuando hay una transferencia de información de uno a otro.
  - Dos procesos están **sincronizados** cuando hay restricciones en el orden en que ejecutan algunas de sus acciones.
- La forma más sencilla de comunicación entre dos procesos o más procesos consiste en la compartición de variables comunes.
- El acceso incontrolado a variables comunes
  - Hace que el resultado de la ejecución dependa del orden en que se intercalan las instrucciones de dos o más procesos: **condición de carrera** (*race condition*).
  - Es una situación anómala que hay que evitar

5.5. Comunicación y sincronización con variables comunes.

Mecanismos de sincronización

- Sección crítica: secuencia de instrucciones que debe ejecutarse de forma indivisible.
- Exclusión mutua: forma de sincronización que se usa para proteger una sección crítica.
- **Espera ocupada**

```
while Test_and_Set(Flag) loop
  null;
end loop;
-- Sección Crítica
Flag:= False;
```

  - La operación Test_and_Set es atómica.
  - Ineficiente.
  - Los protocolos que usan esperas ocupadas son difíciles de diseñar, entender y demostrar su correctitud.

5.5. Comunicación y sincronización con variables comunes.

Mecanismos de sincronización

- **Semáforos**
  - Un semáforo es una variable que toma valores enteros no negativos.
  - Además de asignarle un valor inicial, sólo se pueden hacer dos operaciones (indivisibles) sobre un semáforo: **wait** y **signal**.
  - Ejemplo: Mutex : Semaforo; -- Inicializado a 1 por defecto

```
task t1;
task body t1 is begin
  loop
    . . .
    Wait(Mutex);
    -- Sección Crítica 1
    Signal(Mutex);
    . . .
  end loop;
end t1;
```

```
task t2;
task body t2 is begin
  loop
    . . .
    Wait(Mutex);
    -- Sección Crítica 2
    Signal(Mutex);
    . . .
  end loop;
end t2;
```

5.5. Comunicación y sincronización con variables comunes.

Mecanismos de sincronización

- **Regiones críticas condicionales**
  - Una región crítica es una secuencia de instrucciones que se ejecuta en exclusión mutua.
  - La sincronización condicional se consigue por medio de **guardas** en las regiones críticas: sólo se puede entrar en la región crítica cuando la guarda es verdadera.
  - Ejemplo (no Ada)

```
V : Shared T;

region V when condición is
  . . .
end region;
```

5.5. Comunicación y sincronización con variables comunes.

Mecanismos de sincronización

- **Monitores**
  - Un monitor es un módulo cuyas operaciones se ejecutan en exclusión mutua.
  - La sincronización condicional se consigue por medio de **variables de condición** y las operaciones **wait** y **signal**.
  - Ejemplo (no Ada)

```
monitor buffer;
  export append, take;
  var . . .
  procedure append (i : integer);
  begin
    if Buffer_Lleno then wait(hay_espacio);
    . . . signal(hay_item);
  end append;
  procedure take (var i : integer); . . .  end take;
begin -- incicialización de las variables del monitor
end buffer;
```

5.5. Comunicación y sincronización con variables comunes.

Objetos protegidos (Ada)

- Un objeto protegido encapsula datos que sólo podrán ser accedidos a través de subprogramas protegidos o entradas protegidas en exclusión mutua.
- La condición de sincronización se alcanza mediante expresiones booleanas en las entradas (son guardas, pero en Ada se llaman **barreras**), que deben evaluarse antes de la ejecución de la entrada.
- Una unidad protegida podría ser declarada como un tipo o como una única instancia y, como toda unidad de programa, consta de una especificación y un cuerpo.
- Ejemplo:

```
protected type Shared_Integer(Initial_Value : Integer) is
      function Read return Integer;
      procedure Write(New_Value : Integer);
      procedure Increment(By : Integer);
private
      The_Data : Integer:= Initial_Value;
end Shared_Integer;

My_Data : Shared_Integer(42);
```

5.5. Comunicación y sincronización con variables comunes.

Objetos protegidos (Ada)

- Un procedimiento protegido proporciona acceso exclusivo a los datos encapsulados tanto para escritura como para lectura.
- Una función protegida proporciona acceso concurrente de lectura a los datos encapsulados.
- Una entrada protegida es un procedimiento protegido guardado por una expresión booleana (barrera). Si esta barrera es *false* al llamar a la entrada, se suspende la tarea llamadora hasta que la barrera valga *true* y no exista ninguna otra tarea activa dentro del objeto protegido.

```
protected body Shared_Integer(Initial_Value : Integer) is
      function Read return Integer is begin
        return The_Data;
      end Read;
      procedure Write(New_Value : Integer) is begin
        The_Data:= New_Value;
      end Write;
      procedure Increment(By : Integer) is begin
        The_Data:= The_Data + By;
      end Increment;
end Shared_Integer;
```

5.5. Comunicación y sincronización con variables comunes.

Objetos protegidos (Ada)

- Ejemplo: Buffer compartido.

```
Buffer_Size : constant Integer:= 10;
type Index is mod Buffer_Size;
subtype Count is Natural range 0..Buffer_Size;
type Buffer is array (Index) of Data_Item;

protected type Bounded_Buffer is
  entry Get(Item : out Data_Item);
  entry Put(Item : in Data_Item);

  private
    First, : Index:= Index'First;
    Last : Index:= Index'Last;
    Number_In_Buffer : Count := 0;
    Buf : Buffer;
end Bounded_Buffer;
```

```
protected body Bounded_Buffer is
  entry Get(Item : out Data_Item)
    when Number_In_Buffer > 0 is
  begin
    Item:= Buf(First);
    First:= First + 1;
    Number_In_Buffer:= Number_In_Buffer - 1;
  end Get;
  entry Put(Item : in Data_Item)
    when Number_In_Buffer < Buffer_Size is
  begin
    Last:= Last + 1;
    Buf(Last):= Item;
    Number_In_Buffer:= Number_In_Buffer + 1;
  end Put;
end Bounded_Buffer;
```

5.6. Comunicación y sincronización mediante mensajes.

- La alternativa al paradigma de memoria compartida lo constituye el paradigma de paso de mensajes.
- Las variaciones en el modelo de sincronización de procesos surgen a partir de la semántica de la operación **send**:
  - **Asíncrono**: el emisor no espera a que el mensaje se reciba.
  - **Síncrono**: el emisor continúa después de que el mensaje se reciba.
  - **Invocación Remota**: el emisor continúa sólo cuando el receptor manda una respuesta. Modela el paradigma de comunicación petición-respuesta. Es el que incluye Ada.

Modelo de Ada

- Para que una tarea pueda recibir un mensaje debe definir una entrada (**entry**).
- Ésta entrada es invocada por otra tarea para enviar un mensaje.

```
task Tarea_Calculadora is
  entry Suma(S1, S2 : Integer; Resultado : out Integer);
  entry Resta(M, S : Integer; Resultado : out Integer);
end Tarea_Calculadora;
```

En otra tarea: Tarea_Calculadora.Suma(10, 15, c);

5.6. Comunicación y sincronización mediante mensajes.

Modelo de Ada

- Se pueden definir entradas privadas: sólo pueden ser llamadas por tareas locales al cuerpo de la tarea en la que se definen.
- También pueden haber entradas con igual nombre pero distintos parámetros.

```
task type Telephone_Operator is
  entry Directory_Enquiry(Person : in Name; Addr : in Address; Num : out Number);
  entry Directory_Enquiry(Person : in Name; Zip : in Postal_Code; Num : out Number);
  entry Report_Fault(Num : Number);
private
  entry Allocate_Repair_Worker(Num : out Number);
end Telephone_Operator;
```

- Se pueden definir conjuntos de entradas con un discriminante discreto.

```
type Channel_Number is new Integer range 1..7;
task Multiplexor is
  entry Channels(Channel_Number) (Data : Input_Data);
end Multiplexor;
```

Se podría usar : Multiplexor.Channels(3)(Dato);

5.6. Comunicación y sincronización mediante mensajes.

Modelo de Ada

- En Ada, entradas protegidas y entradas de tareas son idénticas desde la perspectiva del llamador.
- Si se llama a la entrada de una tara inactiva, se eleva la excepción *Tasking_Error* en el punto de llamada.
- Para poder recibir un mensaje, la tarea receptora tendrá que aceptar la llamada a la entrada correspondiente.

```
accept Suma(S1, S2 : Integer; Resultado : out Integer) do
  Resultado:= S1 + S2;
end Suma;
accept Channels(3)(Data : Input_Data) do
  -- Almacena el dato para el tercer canal
end Channels;
```

- Todas las entradas deben tener *accept*'s asociados con ellas.
- La cláusula **accept** se puede poner en cualquier lugar del cuerpo de una tarea. En particular, podría ponerse dentro de otro **accept** (siempre que sea de una entrada diferente). No se puede poner en un procedimiento.

5.6. Comunicación y sincronización mediante mensajes.

Modelo de Ada

- Existe la posibilidad de que se eleve una excepción dentro de una instrucción **accept**:
  - Si hay un manejador válido dentro de **accept**, éste termina normalmente.
  - Caso contrario, el **accept** es terminado y se re-eleva la excepción en las tareas llamadora (después de la llamada a la entrada) y llamada (después del **accept**).
- Hasta ahora hemos visto como las dos tareas tienen que estar listas para realizar la comunicación, la que llega primero a la cita se suspende hasta que la otra ejecuta la acción complementaria (llamada ó aceptación).

- Hay ocasiones en las que no es posible prever el orden en que se van a invocar las distintas entradas de un tarea. En estos caso se usa **espera selectiva**.
- Esto ocurre cuando una tarea servidora acepta llamadas de varios clientes, pero no sabe el orden en que los clientes van a realizar las llamadas.
- Es necesario que una tarea pueda esperar simultáneamente llamadas en varias entradas.

5.6. Comunicación y sincronización mediante mensajes.

Modelo de Ada

- Para ello, Ada dispone de la cláusula **select**.

```
task body Tarea_Calculadora is  ...
begin
  -- Inicialización del servicio
  loop
    select
      accept Suma (S1, S2: integer; S : out integer) do
       . . .
      end Suma;
    or
      accept Resta (M, S: integer; R : out integer) do
      . . .
      end Resta;
    end select;
  end loop;
end Tarea_Calculadora;
```

```
select
  when <Expresión_Booleana> =>
    accept <entrada> do . . . end <entrada>;
    -- cualquier secuencia de instrucciones
  or
    -- similar
          . . .
end select;
```

- Además de las alternativas **accept**, pueden existir otras: una cláusula **else**, una cláusula **terminate** o una cláusula **delay**.

5.6. Comunicación y sincronización mediante mensajes.

Modelo de Ada

- **Else** se ejecuta si no hay otra alternativa inmediatamente ejecutable.
- **Delay** será estudiada en el tema siguiente.
- **Terminate**: sólo se puede elegir si no hay tareas que puedan invocar al **select**, y la tarea que lo ejecuta será terminada.

```
procedure P is
  task server is
    entry E1;
  end server;
  task body server is begin
    loop
      select
        accept E1 do ... end E1;
      or
        terminate;
      end select;
    end loop;
  end server;
begin
  server.E1;
-- Sin terminate, P nunca terminaría
-- debido al bucle de la tarea server
end P;
```

TEMA 6. FACILIDADES DE TIEMPO REAL

6.1. Introducción.
6.2. Acceso a un reloj.
6.3. Retardos en procesos.
6.4. Plazos temporales.
6.5. Requisitos temporales.

6.1. Introducción.

- El tiempo es una magnitud física fundamental cuya unidad en el Sistema Internacional es el segundo.
- La introducción de la noción del tiempo en un lenguaje de programación puede ser descrita en términos de tres aspectos independientes:
  1. Incluir funciones temporales en un programa para:
     - Acceso a relojes para medir el paso del tiempo.
     - Retrasar procesos hasta un tiempo futuro.
     - Programar plazos temporales de forma que pueda reconocerse y tratarse la no ocurrencia de una evento.
  2. Representar los requerimientos temporales: Por ejemplo, especificar plazos y periodos de ejecución.
  3. Satisfacer los requerimientos temporales.

- 1 y 2 serán estudiados en este tema, 3 será el objetivo del tema siguiente.

6.2. Acceso a un reloj.

Relojes en Ada

- El acceso a un reloj en Ada puede realizarse a través del paquete *Calendar*, que proporciona el tipo abstracto *Time*.
- *Calendar* proporciona una función *Clock* para leer el tiempo y varios subprogramas para hacer conversiones entre *Time* y otras unidades entendibles por los humanos como años, meses, días y segundos.
- Además de *Calendar*, Ada ofrece otro paquete de medida del tiempo más adecuado para la realización de sistemas empotrados: *Real_Time*.
- Los segundos se representan como un subtipo del tipo *Duration* (es un tipo predefinido de punto fijo).
- Rango y precisión de *Duration* dependen de la implementación, aunque el rango mínimo debe comprender ±1 día (-86400.0 .. 86400.0) y su precisión (también llamada resolución) debe ser de al menos 20 milisegundos.
- La granularidad de un reloj software se define como el intervalo de tiempo entre dos interrupciones de reloj físico. A menor granularidad mayor precisión.

6.2. Acceso a un reloj.

```
package Ada.Calendar
  type Time is private
  subtype Year_Number is Integer range 1901..2099; --Idem con Month_Number y Day_Number
  subtype Day_Duration is Duration range 0.0..86400.0;
  function Clock return Time;
  function Year(Date:Time) return Year_Number; -- Idem con Month y Day
  function Seconds(Date:Time) return Day_Duration;
  procedure Split(Date:in Time; Year:out Year_Number; Month:out Month_Number;
                  Day:out Day_Number; Seconds:out Day_Duration; );
  function "+"(Left:Time;     Rigth:Duration) return Time;
  function "+"(Left:Duration; Rigth:Time)     return Time;
  function "-"(Left:Time;     Rigth:Duration) return Time;
  function "-"(Left:Time;     Rigth:Time)     return Duration;
  function "<" (Left,         Right:Time)     return Boolean;       -- Idem con <=, >, >=
  Time_Error: exception;   -- TIME_ERROR es elevada por Time_Of, split, "+", y "-" cuando se
                              intenta crear tiempos y duraciones fuera de rango.
  private   -- Detalles de la implementación
end Ada.Calendar;
```

6.2. Acceso a un reloj.

Relojes en Ada

- El paquete *Real_Time* consigue una granularidad más fina y un tiempo monótono (el tiempo proporcionado por *Calendar* puede pasar, por ejemplo, de las 3h a las 2h).
- En el paquete *Real_Time*, *Time* representa el tiempo transcurrido desde el instante en que arrancó el programa, *Clock*, por lo tanto, es monótona.
- La constante *Time_Unit* es la unidad de tiempo *Time* más pequeña.
- *Real_Time* define los intervalos de tiempo en términos del tipo *Time_Span*.
- Las funciones *Nanoseconds*, *Microseconds* y *Milliseconds* toman un parámetro entero y lo convierten a *Time_Span*.
- Por tanto, el código requerido para medir el tiempo empleado en un cálculo:

```
Old_Time:= Clock;
-- Cálculo a medir
New_Time:= Clock;
Interval:= New_Time - Old_Time;
```

Donde:

Old_Time, New_Time : Time;

Ada.Calendar ⇒ Interval : Duration;

Ada.Real_Time ⇒ Interval : Duration;

6.2. Acceso a un reloj.

```
package Ada.Real_Time is
  type Time is private ;
  Time_First, Time_Last: constant Time;
  Time_Unit: constant := ...
  type Time_Span is private ;
  Time_Span_First, Time_Span_Last, Time_Span_Zero, Time_Span_Unit : constant Time_Span;
  Tick: constant Time_Span;
  function Clock return Time;
  function "+" (Left: Time; Right: Time_Span) return Time;
  function "<" (Left, Right: Time) return Boolean;
  function "+" (Left, Right: Time_Span) return Time_Span;
  function "<" (Left, Right: Time_Span) return Boolean;
  function "abs"(Right : Time_Span) return Time_Span;
  function To_Duration (Ts : Time_Span) return Duration;
  function To_Time_Span (D : Duration) return Time_Span;
  function Nanoseconds (Ns: Integer) return Time_Span; --Idem para Microseconds y Milliseconds
  type Seconds_Count is range                  -- Dependiente de la implementación
  procedure Split(T : in Time; Sc: out Seconds_Count; Ts : out Time_Span);
  private    -- Detalles de la implementación
end Ada.Real_Time;
```

6.3. Retardos en procesos.

- Un **retardo** suspende la ejecución de una tarea durante un cierto tiempo. Distinguimos entre:
  - Retardos relativos: la ejecución se suspende durante un intervalo de tiempo relativo al instante actual.
  - Retardos absolutos: la ejecución se suspende hasta que se llega a un instante de tiempo absoluto.

Retardos relativos

- Ada dispone de la instrucción **delay** que suspende la ejecución de la tarea que la invoca durante un intervalo de tiempo.
- Recibe como parámetro un valor de tipo *Duration* que especifica el intervalo de tiempo de suspensión.
- Ejemplo: **delay** 10.0; -- *Suspende a la tarea que lo ejecuta 10 seg.*
- Garantiza que el proceso es *ejecutable* una vez que el periodo de suspensión ha finalizado, pero no que *se vaya a ejecutar*.

6.3. Retardos en procesos.

Retardos absolutos

- Ada dispone de la instrucción **delay until** que suspende la ejecución de la tarea que la invoca hasta que el valor del reloj sea igual al especificado.
- Recibe como parámetro un valor de tipo *Time* (tanto de *Ada.Calendar* como de *Ada.Real_Time*) que especifica el valor del reloj a esperar.
- Ejemplo: *Second_Action* se ejecuta 10 seg. tras el comienzo de *First_Action*.

```
Start:= Clock;
First_Action;
delay until Start + 10.0;
Second_Action;
```

Atómico

No son equivalentes

```
Start:= Clock;
First_Action;
delay 10.0 - (Clock - Start);
Second_Action;
```

No atómico

- Ejemplo: La acumulación de la sobrecarga introducida por las primitivas de retardo tanto relativo como absoluto puede evitarse. →

```
sig:= Clock + 7.0;
loop
  Acción;
  delay until sig;
  sig:= sig + 7.0;
end loop;
```

6.4. Plazos temporales.

- Un **plazo** (timeout) es una restricción sobre el tiempo que un proceso puede esperar la ocurrencia de un evento.
- En STR, los mecanismos de comunicación entre procesos deben tener plazos.

Plazos y comunicación basada en memoria compartida

- Cuando un proceso intenta acceder a una **sección crítica** que está siendo utilizada por otro proceso, es bloqueado.
- La duración de este bloqueo está delimitado por el tiempo que tarda en ejecutarse la sección crítica y el resto de procesos que también desean acceder.
- En el caso de la **sincronización condicional** la duración del bloqueo no está delimitada. Por ejemplo: Un proceso productor que intenta almacenar un dato en un buffer lleno tiene que esperar un tiempo indefinido hasta que un proceso consumidor retire un dato del mismo, por tanto, es importante que los procesos especifiquen plazos de espera para la sincronización condicional.
- En Ada y con objetos protegidos:

```
select
  P.E;  -- E es una entrada de P (OBJETO PROTEGIDO)
or
  delay 2.0;
end select;
```

6.4. Plazos temporales.

Plazos y comunicación basada en paso de mensajes

- En la cita Ada, la introducción de plazos se realiza a través de variantes de la instrucción **select** con alternativas especiales:
  - **delay**
  - **else**
  - **terminate**

  } Son mútuamente excluyentes.
- Problema 1: **Recepción con plazo**: Tarea controladora que recibe un valor de temperatura regularmente y que debe realizar una acción si se rebasa un plazo de espera sin recibir dicho valor.

  Sin considerar el plazo:

```
task Controller is
  entry Call (T : Temperature);
end Controller;
```

```
task body Controller is
  -- Declaraciones
begin
  loop
    accept Call (T : Temperature) do
      New_Temp:= T;
    end Call;
    -- Otras acciones
  end loop;
end Controller;
```

6.4. Plazos temporales.

Plazos y comunicación basada en paso de mensajes

- Considerando el plazo y usando primitivas ya conocidas: solución compleja.

```
task Controller is
  entry Call (T : Temperature);
private
  entry Timeout;
end Controller;
task body Controller is
  task Timer is
    entry Go (D : Duration);
  end Timer;   -- Otras declaraciones
  task body Timer is
    Timeout_Value : Duration;
  begin
    accept Go (D : Duration) do
      Timeout_Value:= D;
    end Go;
    delay Timeout_Value;
    Controller.Timeout;
end Timer;
begin
  loop
    Timer.Go(10.0);
    select
      accept Call (T : Temperature) do
        New_Temp:= T;
      end Call;
    or
      accept Timeout; -- Acción para Timeout
    end select;
    -- Otras acciones
  end loop;
end Controller;
```

6.4. Plazos temporales.

Plazos y comunicación basada en paso de mensajes

- Usando recepción con plazo: alternativa *delay* en **select**.

```
task Controller is
  entry Call (T : Temperature);
end Controller;
```

```
task body Controller is
-- Declaraciones
begin
  loop
    select
      accept Call (T : Temperature) do
        New_Temp:= T;
      end Call;
    or
      delay 10.0; -- Acción para Timeout
    end select;
    -- Otras acciones
  end loop;
end Controller;
```

- **select** puede tener más de una alternativa de retardo, pero todas del mismo tipo (retardos absolutos o relativos). La opertativa es la que tiene menor plazo.

6.4. Plazos temporales.

Plazos y comunicación basada en paso de mensajes

- Ejemplo: Uso de la alternativa *delay until*.

```
task Ticket_Agent is
  entry Registration (. . .);
end Ticket_Agent;
```

```
task body Ticket_Agent is
  Shop_Open : Boolean:= True;
begin
  while Shop_Open loop
    select
      accept Registration (. . .) do
        -- Detalles del registro
      end Registration;
    or
      delay until Closing_Time;
        Shop_Open:= False;
    end select;
    -- Procesa los registros
  end loop;
end Ticket_Agent;
```

6.4. Plazos temporales.

Plazos y comunicación basada en paso de mensajes

- Problema 2: **Envío con plazo.**

```
loop
  -- Adquiere la nueva temperatura T
  select
    Controller.Call(T);
  or
    delay 2.0; -- Si la llamada no es aceptada en 2 seg. se ejecuta el código que le sigue
    -- Código a ejecutar cuando expira el plazo de envío
  end select;
end loop;
```

- Si lo que queremos es hacer una llamada temporizada a una entrada sólo en el caso de que la tarea receptora esté lista podemos usar *delay 0.0;* o bien

```
select
  T.E  -- Entrada E en la tarea T
else    -- Si E no puede atender la llamada inmediatamente
  -- otras acciones
end select;
```

6.4. Plazos temporales.

Plazos y comunicación basada en paso de mensajes

- Problema 3: **Poner plazo a la ejecución de un conjunto de instrucciones.**

  Supongamos que una tarea debe completar una acción en 100 milisegundos.

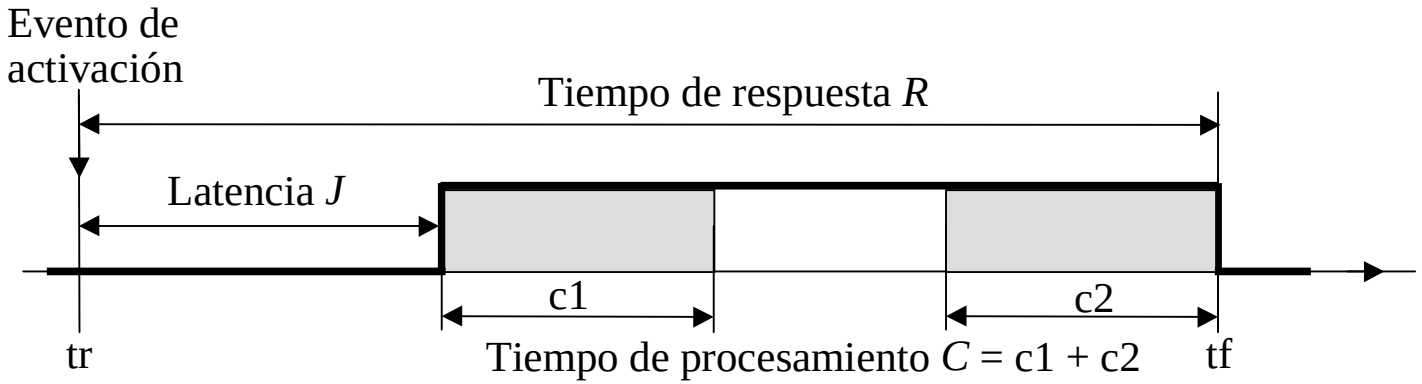
```
select
  delay 0.1;
then abort
  -- acción a ejecutar en un máximo de 100 ms.
end select;
```

- Ejemplo: una tarea con un componente obligatorio que calcula el resultado en un plazo dado y otro opcional de mejora que se ejecuta si sobra tiempo.

```
Completion_Time.= . . .;
-- Parte obligatoria
Results.Write(. . .);  -- Llamada al procedimiento en un objeto protegido externo
select
  delay until Completion_Time;
then abort
  loop
    -- Mejora el resultado
    Results.Write(. . .);
  end loop;
end select;
```

6.5. Requisitos temporales.

- Un **marco temporal** (TS) es un conjunto de instrucciones con una serie de restricciones temporales asociadas. **Atributos**:
  - **Latencia mínima J_{min}**: Cantidad mínima de tiempo desde que se produce el evento hasta que se ejecuta el TS asociado.
  - **Latencia máxima J_{max}**: Cantidad máxima de tiempo desde que se produce el evento hasta que se ejecuta el TS asociado.
  - **Plazo de respuesta *D***: Instante de tiempo en que la ejecución de un TS debe haber finalizado.
  - **Tiempo de respuesta *R***: Tiempo que transcurre desde que se produce el evento hasta que finaliza la ejecución del TS asociado.
  - **Tiempo de procesamiento *C***: Tiempo de uso del procesador del TS.



6.5. Requisitos temporales.

- Los marcos temporales pueden clasificarse en función de su activación en:
  - **Periódicos**: Generalmente muestrean datos o forman parte de un bucle de control. **T** representa el periodo de activación.
  - **Aperiódicos**: Se activan en respuesta a eventos externos al STR que se producen de forma aleatoria.
  - **Esporádicos**: Se activan en respuesta a eventos externos al STR. **T** representa la separación mínima entre dos eventos consecutivos.
- En muchos STR, los marcos temporales suelen ir asociados a los procesos que los contienen $\Rightarrow$ hablamos de procesos periódicos, aperiódicos o esporádicos.
- El problema de satisfacer los atributos de los marcos (restricciones temporales) se traduce en el de planificar los procesos asociados de manera adecuada. Es lo que se denomina **planificación con plazos**.
- En función de la importancia del tiempo, podemos clasificar los STR:
  - **Crítico** (*hard*): Es inadmisible que se pierda algún plazo.
  - **Acrítico** (*soft*): Ocasionalmente puede perderse algún plazo.
  - **Interactivo**: No se especifican plazos sino tiempos de respuesta medios.
  - **Firme** (*firm*): El plazo no es crítico, pero una respuesta tardía no sirve.

6.5. Requisitos temporales.

Esquema de un proceso periódico

```
process p_periodico
  . . .
begin
  loop
    Ocioso(T);
    Marco_Temporal;
  end;
end;
```

Esquema de un proceso esporádico

```
process p_esporádico
  . . .
begin
  loop
    Espera_Interrupción;
    Marco_Temporal;
  end;
end;
```

Tareas periódicas en Ada

```
task body  tarea_periodica is
    Intervalo_Activacion : Time_Span := Milliseconds( . . . );
begin
    -- Leer el reloj y calcular el siguiente instante de activación.
    loop
      -- Muestrear datos (por ejemplo) o calcular y enviar una señal de control
      delay until Siguiente_Activacion;
      Siguiente_Activacion:= Siguiente_Activacion + Intervalo_Activacion;
    end loop;
end tarea_periodica;
```

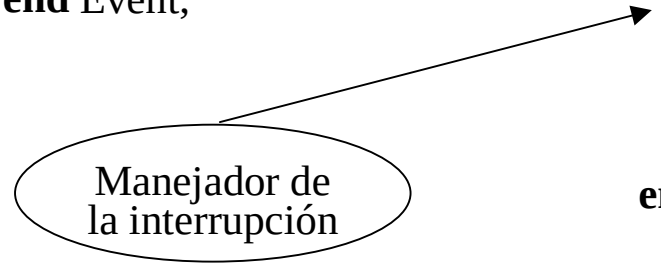
6.5. Requisitos temporales.

Tareas esporádicas en Ada

- El suceso de activación se implementa mediante un **objeto protegido**.

```
protected Event is
   entry Wait;
   procedure Signal;
private
   Occurred : Boolean:= False;
end Event;
```

```
protected  body Event is
   entry Wait when Occurred is
   begin
      Occurred:= False;
   end Wait;
   procedure Signal is
   begin
      Occurred:= True;
   end Signal;
end Event;
```



- La tarea esporádica se activa cuando ocurre el evento:

```
task body tarea_esporadica is
begin
   -- Inicialización
   loop
      Event.Wait;
      -- Acción esporádica
   end loop;
end tarea_esporadica;
```

TEMA 7. PLANIFICACIÓN PARA TIEMPO REAL

7.1. Introducción.

- Un STR necesita restringir el indeterminismo encontrado en los sistemas concurrentes: este proceso se conoce como **planificación** (*Scheduling*).
- Un **método de planificación** presenta dos aspectos importantes:
  1. Un algoritmo de planificación que ordena el uso de los recursos del sistema (en particular, la CPU).
  2. Un método de análisis que permita predecir el comportamiento temporal del sistema en el peor caso cuando se aplica el algoritmo de planificación.
- Un método de planificación puede ser:
  1. **Estático**: el análisis puede realizarse antes de la ejecución.
  2. Dinámico: se utilizan decisiones tomadas en tiempo de ejecución.
- Un método estático muy utilizado es el de **planificación con prioridades fijas y desalojo** (*preemtive priority-based schemes*): a cada proceso se le asigna una prioridad y en cada instante se ejecuta el proceso con prioridad mayor.
- Necesitamos, por lo tanto, un algoritmo de asignación de prioridades y una prueba de planificabilidad.

7.1. Introducción.

- Inicialmente consideraremos un modelo de tareas muy simple para describir algunos métodos estándar de planificación. Tiene las siguientes características:
  - La aplicación está formada por un conjunto fijo de tareas.
  - Todas las tareas son periódicas, con periodos conocidos.
  - Las tareas son completamente independientes entre sí.
  - Todas las sobrecargas del sistema, duración de los cambios de contexto... son ignoradas (se asume que tienen coste cero).
  - Cada tarea tiene un plazo igual a su periodo.
  - Todos las tareas tienen un tiempo de ejecución máximo conocido.
- Como consecuencia de la independencia entre tareas, puede asumirse que en algún instante de tiempo (**instante crítico)** las tareas serán liberadas juntas.
- Parámetros de planificación:
  - *N* : Número de tareas.
  - *T* : Periodo de activación.
  - *D* : Plazo de respuesta.
  - *C* : Tiempo de ejecución máximo.
  - *R* : Tiempo de respuesta máximo.
  - *P* : Prioridad.

7.1. Introducción.

- En el modelo de tareas simple descrito con anterioridad:

  Para toda tarea t_i se cumple:

  $$C_i \leq D_i = T_i$$

  Se trata, por lo tanto, de asegurar que:

  $$R_i \leq D_i$$

  El valor $H = \text{mcm}(T_i)$ se denomina **hiperperiodo** del sistema.

- El comportamiento temporal se repite cada hiperperiodo.

7.2. Planificación con ejecutivos cíclicos.

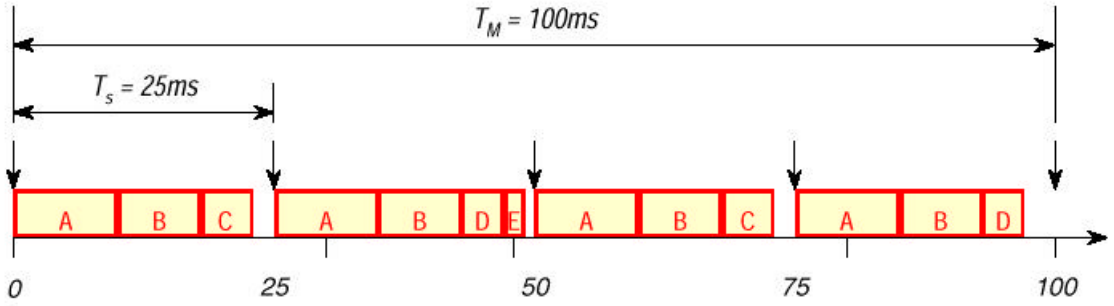
- Es posible confeccionar un plan de ejecución para un conjunto fijo de tareas periódicas, de forma que la ejecución repetida del plan asegure que todas las tareas se ejecutan en el tiempo correcto.
- Se trata de un esquema que se repite cada T_M (= mcm(T_i)) $\Rightarrow$ **Ciclo principal**.
- El ciclo principal se divide en **ciclos secundarios** con periodo T_S tal que:

$$\forall i : T_S \leq D_i,\ \forall i : T_S \geq max(C_i),\ T_M = k \cdot T_S$$

- En cada ciclo secundario se ejecutan las actividades asociadas a determinadas tareas.
- Ejemplo:

Proceso	T	C
A	25	10
B	25	8
C	50	5
D	50	4
E	100	2

- El ciclo principal dura 100 ms
- Se compone de 4 ciclos secundarios de 25 ms



7.2. Planificación con ejecutivos cíclicos.

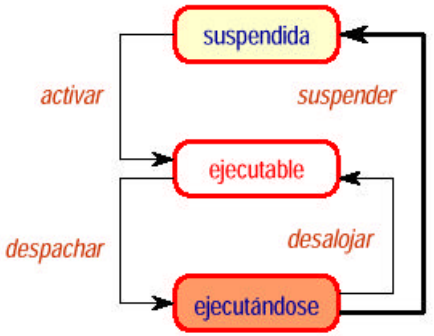
- Ejemplo: Código Ada.

```
procedure Ejecutivo_Cíclico is
  type Ciclo is mod 4;
  Marco : Ciclo:= 0;
begin
  loop
    Espera_Interrupción;
    case Marco is
      when 0 => A; B; C;
      when 1 => A; B; D; E;
      when 2 => A; B; C;
      when 3 => A; B; D;
    end case;
    Marco:= Marco + 1;
  end loop;
end Ejecutivo_Cíclico;
```

- Características:
  - En tiempo de ejecución no existen procesos, cada ciclo secundario es una secuencia de llamadas a procedimientos.
  - Los procedimientos pueden compartir datos. Estos datos no necesitan ser protegidos debido a que no es posible acceso concurrente.
  - Todos los periodos de los *procesos* deben ser múltiplo de la duración del ciclo secundario.
  - El sistema es correcto por construcción.

7.3. Planificación con prioridades.

- Las tareas se realizan como procesos concurrentes y se usa un atributo de prioridad para determinar qué proceso debe ejecutarse en cada instante.
- Una tarea puede estar en varios estados:



- Las tareas ejecutables se despachan para su ejecución en orden de prioridad.
- El despacho puede hacerse:
  - Con desalojo: se producirá un cambio inmediato al llegar un proceso con mayor prioridad que el que se está ejecutando.
  - Sin desalojo: el proceso de menor prioridad podrá completar su ejecución antes de que la de mayor prioridad comience a ejecutarse.
- En este tema supondremos **prioridades fijas con desalojo**.

7.3. Planificación con prioridades.

7.3.1. Tareas periódicas independientes.

- Asignación de prioridades monótona en frecuencia (*rate monotonic scheduling*).
- A cada tarea se le asigna una prioridad en función de su periodo, las tareas de menor periodo tendrán mayor prioridad ⇒ Asignación óptima para el modelo de tareas simple (cualquier conjunto de tareas que pueda ser planificado con un esquema de prioridades fijas, podrá también serlo con la asignación monótona en frecuencia).

Test de planificabilidad basado en el factor de utilización.

- La cantidad $U = \sum_{i=1}^{N} \left(\frac{C_i}{T_i} \right)$ es el factor de utilización del procesador.
- Es una medida de la carga del procesador para un conjunto de tareas.
- En un sistema monoprocesador debe ser $U \leq 1$.
- Para el modelo simple, con asignación de prioridades monótona en frecuencia, los plazos están garantizados si se cumple:

$$U = \sum_{i=1}^{N} \left(\frac{C_i}{T_i} \right) \leq N(2^{1/N} - 1)$$

- Para valores grandes de N, la cota se aproxima asintóticamente a 69.3%.

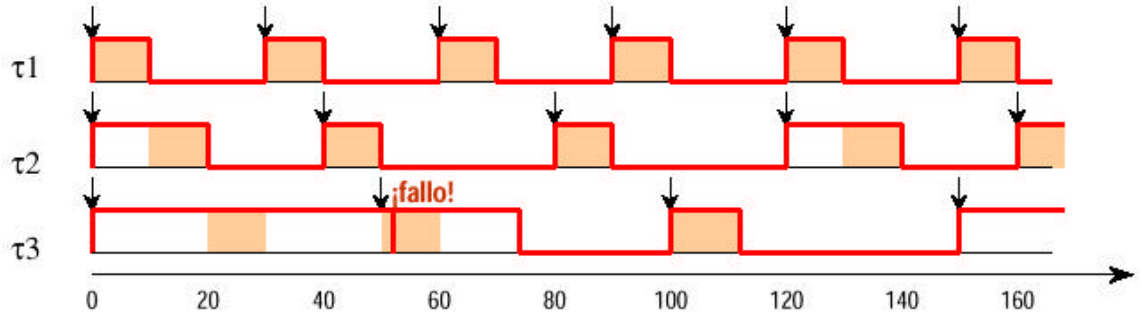
7.3. Planificación con prioridades.

7.3.1. Tareas periódicas independientes.

- Ejemplo 1:

	T	*C*	*P*	*U*
Tarea 1	50	12	1	0.24
Tarea 2	40	10	2	0.25
Tarea 3	30	10	3	0.33

El sistema no cumple la prueba de utilización: U =0.24+0.25+0.33=0.82>0.779
La tarea 3 falla en t=50.



- Se pueden usar cronogramas para comprobar si se cumplen los plazos: hay que trazar el cronograma durante un hiperperido completo.
- En el instante crítico el tiempo de respuesta es máximo.
- Si el instante inicial es crítico basta comprobar el primer ciclo de cada tarea.

7.3. Planificación con prioridades.

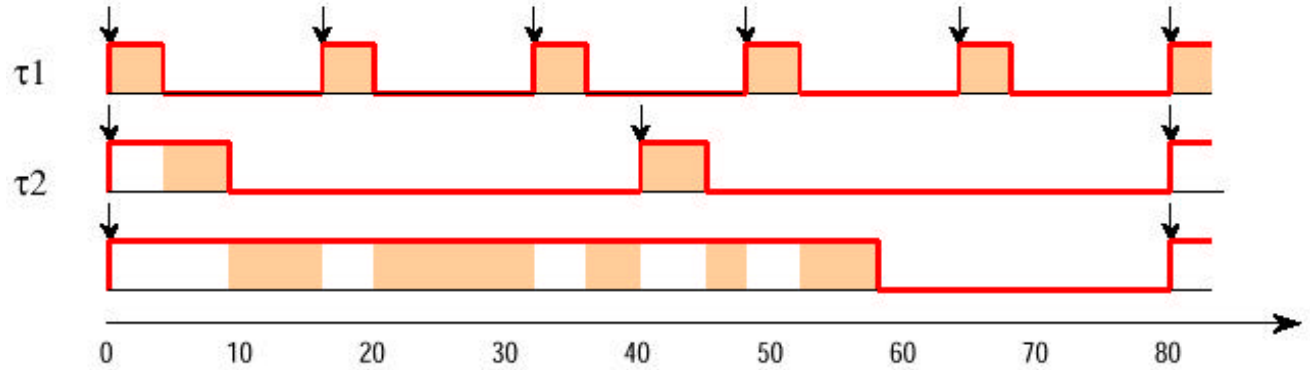
7.3.1. Tareas periódicas independientes.

- Ejemplo 2:

	T	*C*	*P*	*U*
Tarea 1	80	32	1	0.400
Tarea 2	40	5	2	0.125
Tarea 3	16	4	3	0.250

El sistema cumple la prueba de utilización:

$U = 0.4+0.125+0.25=0.775<0.779$



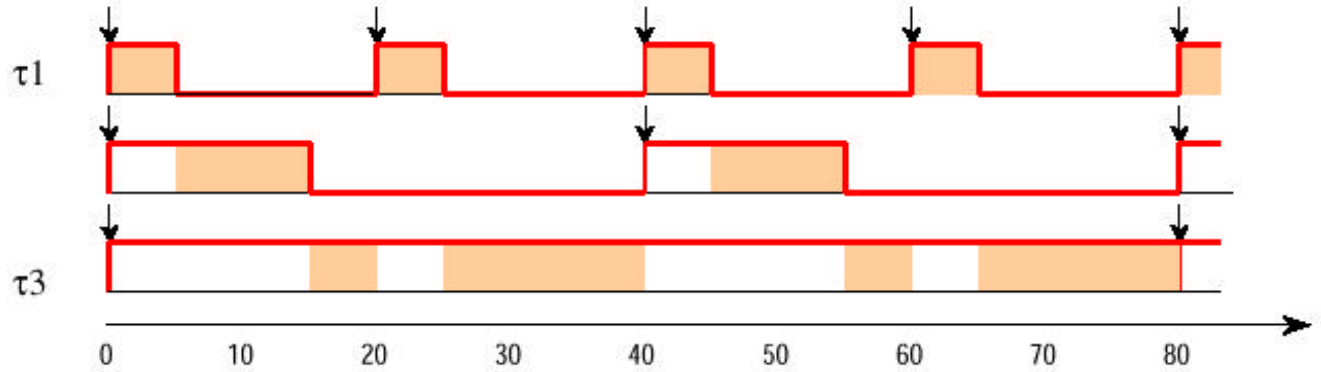
7.3. Planificación con prioridades.

7.3.1. Tareas periódicas independientes.

- Ejemplo 3:

	T	*C*	*P*	*U*
Tarea 1	80	40	1	0.50
Tarea 2	40	10	2	0.25
Tarea 3	20	5	3	0.25

El sistema no cumple la prueba de utilización: U =0.5+0.25+0.25=1<0.779



- El conjunto de tareas no pasa el test basado en el factor de utilización, pero en tiempo de ejecución no se pierde ningún plazo ⇒ este test impone una condición suficiente pero no necesaria.
- Este test no informa de los tiempos de respuesta de las tareas.

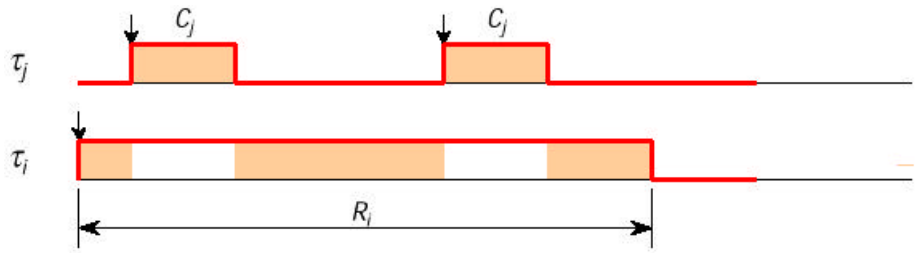
7.3. Planificación con prioridades.

7.3.1. Tareas periódicas independientes.

Análisis del tiempo de respuesta.

- El test basado en el factor de utilización tiene dos importantes problemas:
  1. No es exacto.
  2. No es aplicable a un modelo de tareas más general.
- Como alternativa veremos un test basado en el cálculo del tiempo de respuesta en el peor caso para cada tarea. Este valor es comparado después con el plazo asociado a cada tarea.
- El tiempo de respuesta en el peor caso para la tarea de prioridad más alta es igual a su propio tiempo de cálculo ($R=C$). El resto de las tareas sufrirán ciertas **interferencias** por parte de las tareas de mayor prioridad. Para la tarea *i*:

  $R_i=C_i+I_i$ donde I_i es la máxima interferencia para la tarea *i*



7.3. Planificación con prioridades.

7.3.1. Tareas periódicas independientes.

Análisis del tiempo de respuesta.

- Cálculo de la interferencia máxima: la máxima interferencia ocurre, obviamente, cuando todas las tareas de mayor prioridad son liberadas al mismo tiempo que la tarea *i* (es decir, en el instante crítico).
  - Consideremos una tarea *j* de mayor prioridad que *i*: dentro de $[0, R_i)$ será liberada una o varias veces ⇒

$$Number\_Of\_Releases = \left\lceil \frac{R_i}{T_j} \right\rceil$$

$$Maximum\_Interference = \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

  - Cada tarea de mayor prioridad interfiere con *i*, por tanto:

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \qquad\qquad R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

7.3. Planificación con prioridades.

7.3.1. Tareas periódicas independientes.

Análisis del tiempo de respuesta.

- La ecuación del tiempo de respuesta puede resolverse mediante la relación de recurrencia:

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

- Empezamos con $w_i^0 = C_i + \sum_{j \in hp(i)} C_i$
- Se termina cuando $w^{n+1} = w^n$, o bien $w^{n+1} > T_i$ (no se cumple el plazo)
- Ejemplo 4:

	T	*C*	*P*	*R*
Tarea 1	7	3	3	3
Tarea 2	12	3	2	6
Tarea 3	20	5	1	20

$w_1^0 = 3$ ← Tarea 1

Tarea 2:

$$w_2^0 = 3+3 = 6$$
$$w_2^1 = 3 + \left\lceil \frac{6}{7} \right\rceil \cdot 3 = 6$$

Tarea 3:

$$w_3^0 = 5+3+3 = 11$$
$$w_3^1 = 5 + \left\lceil \frac{11}{7} \right\rceil \cdot 3 + \left\lceil \frac{11}{12} \right\rceil \cdot 3 = 14$$
$$w_3^2 = 5 + \left\lceil \frac{11}{7} \right\rceil \cdot 3 + \left\lceil \frac{14}{12} \right\rceil \cdot 3 = 17$$
$$w_3^3 = 5 + \left\lceil \frac{11}{7} \right\rceil \cdot 3 + \left\lceil \frac{17}{12} \right\rceil \cdot 3 = 20$$
$$w_3^4 = 5 + \left\lceil \frac{11}{7} \right\rceil \cdot 3 + \left\lceil \frac{20}{12} \right\rceil \cdot 3 = 20$$

7.3. Planificación con prioridades.

7.3.1. Tareas periódicas independientes.

Análisis del tiempo de respuesta.

- Da una condición **suficiente** y **necesaria**: si un conjunto de tareas cumple el test se cumplirán sus plazos, si no en tiempo de ejecución alguna perderá su plazo.
- El resto del tema lo dedicaremos a extender la aplicabilidad del método.
- Pero, ¿cómo podemos calcular el valor C para cada tarea?. Hay dos formas:
  1. Medida del tiempo de ejecución:
     - » El código es compilado y ejecutado con dispositivos de medida conectados.
     - » En las ejecuciones se utilizan muchos datos de entrada.
     - » Tiempo más largo requerido = tiempo más largo medido
     - » Problema: no hay garantía de encontrar realmente el tiempo de ejecución más largo (es decir, en el peor caso).
  2. Análisis del código ejecutable:
     - » Descompone el código en un grafo de bloques secuenciales.
     - » Calcula tiempo de ejecución de cada bloque y busca camino más largo.

7.3. Planificación con prioridades.

7.3.2. Tareas esporádicas y aperiódicas.

- Las tareas esporádicas y aperiódicas responden a eventos asíncronos que llegan en grupos, podrían definirse frecuencias de activación medias y máximas.
- Inconveniente: en muchas ocasiones el peor caso es considerablemente mayor que la media ⇒ Medir la planificabilidad con tiempos en el peor caso podría llevar a utilizaciones muy bajas del procesador ⇒ Asegurar requerimientos mínimos:
  - Todas las tareas deben ser planificables usando tiempos de ejecución y frecuencias de llegada medios ⇒ pueden perderse algunos plazos.
  - Todas las tareas *hard* deben ser planificables usando tiempos de ejecución y frecuencias de llegada en el peor caso ⇒ asegura los plazos de estas tareas.
- Para incluir tareas esporádicas hay que eliminar la restricción $D=T$ del modelo básico de tareas. Ahora $D \leq T$.
- Necesitamos otra forma de asignar prioridades: **Asignación monótona en plazos** (*Deadline monotonic priority assigment*):
  - Cuando los plazos son menores o iguales que los periodos, la asignación de mayor prioridad a las tareas de menor plazo de respuesta es óptima.

7.3. Planificación con prioridades.

7.3.2. Tareas esporádicas y aperiódicas.

- El análisis del tiempo de respuesta se realiza de igual forma que antes.
  - se termina cuando $w_i^{n+1} = w_i^n$ o cuando $w_i^{n+1} > D_i$
- Ejemplo:

				RMPO		***DMPO***	
	T	***D***	***C***	***P***	***R***	***P***	***R***
Tarea1	20	5	3	2	***10***	4	3
Tarea2	15	7	3	3	7	3	6
Tarea3	10	10	4	4	4	2	10
Tarea4	20	20	3	1	20	1	20

Con asignación de prioridades monótona en frecuencia (RMPO) los plazos no están garantizados.

7.3. Planificación con prioridades.

7.3.3. Interacción entre tareas y bloqueos.

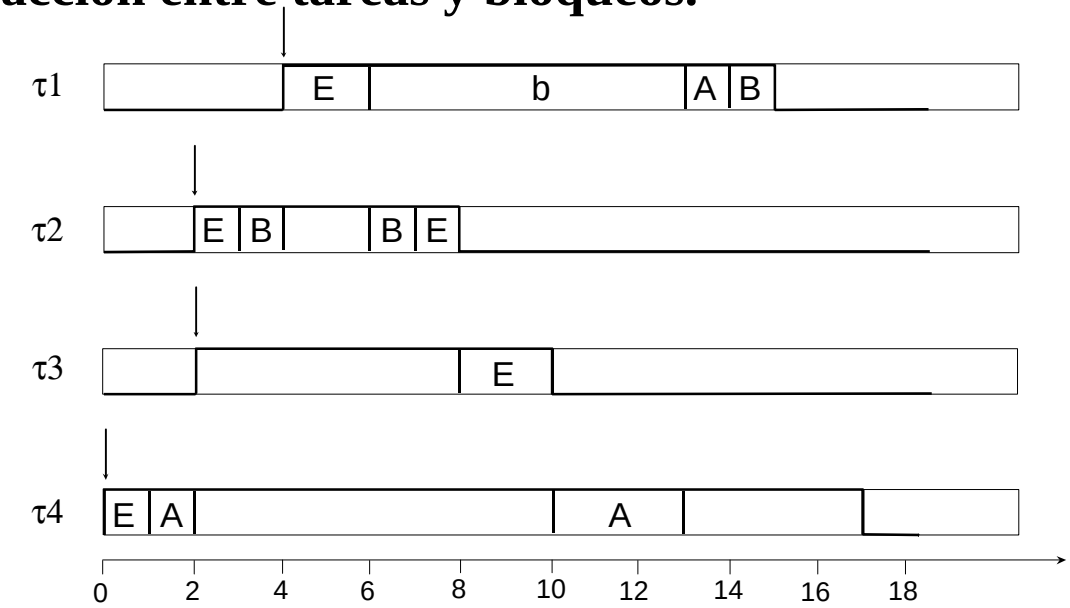
- En el modelo de tareas básico suponíamos que las tareas eran independientes.
- En la mayoría de los sistemas de interés práctico no se cumple esta restricción, sino que las tareas interaccionan mediante datos comunes o paso de mensajes.
- En estos casos, es posible que una tarea tenga que esperar un suceso de otra de menor prioridad. Se dice que la tarea de mayor prioridad está **bloqueada**.
- Esta situación da lugar a una **inversión de prioridades** que no debería existir.
- No podemos evitar la inversión de prioridades, pero sí minimizarla.
- Para poder comprobar la planificabilidad de un conjunto de tareas es necesario acotar y medir los bloqueos.
- Ejemplo:

	P	**Secuencia de ejecución**	**Instante de liberación**
Tarea1	4	EEABE	4
Tarea2	3	EBBE	2
Tarea3	2	EE	2
Tarea4	1	EAAAAE	0

E = Tick de ejecución.
A = Tick de ejecución con acceso a la S.C. A.
B = Tick de ejecución con acceso a la S.C. B.

7.3. Planificación con prioridades.

7.3.3. Interacción entre tareas y bloqueos.



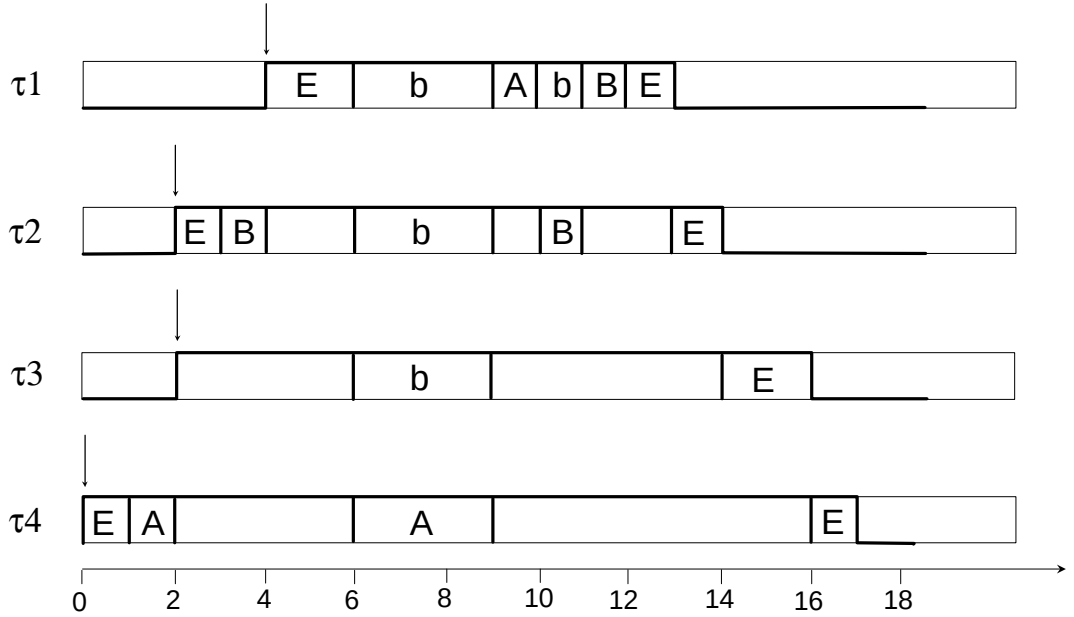
- Los bloqueos son debidos a la inversión de prioridades y comprometen la planificabilidad del sistema.
- Para minimizar el efecto de la inversión de prioridades permitimos que las prioridades de las tareas puedan cambiar dinámicamente ⇒
- **Herencia de prioridades**: si una tarea *p* está bloqueada por una tarea *q* de menor prioridad, entonces *q* cambiará su prioridad por la de *p* hasta que termine.

7.3. Planificación con prioridades.

7.3.3. Interacción entre tareas y bloqueos.

- Para el ejemplo anterior:

Con el protocolo de herencia de prioridades, la tarea 1 sufre 2 bloqueos.
Con prioridades fijas, la tarea 1 sufre 1 bloqueo.
Sin embargo, hemos reducido el tiempo de respuesta de la tarea 1 de 11 a 9 ticks de reloj.



- Con esta simple norma, la prioridad de una tarea es el máximo entre su propia prioridad y la de todas aquellas tareas a las que está bloqueando.
- Hemos de extender nuestro análisis de tiempo de respuesta para tener en cuenta la máxima cantidad de tiempo que una tarea puede verse bloqueada.

7.3. Planificación con prioridades.

7.3.3. Interacción entre tareas y bloqueos.

- Con el protocolo de herencia de prioridad, puede calcularse una cota superior del número de veces que una tarea se puede bloquear:
  - Si tiene *m* secciones críticas ⇒ Numero máximo de bloqueos = *m*.
  - Si hay *n* procesos de menor prioridad y $n < m$, entonces puede reducirse a *n*.
- Sea *K* el número de secciones críticas (recursos) de un conjunto de tareas. La cantidad máxima de tiempo que la tarea *i* puede ser bloqueada puede obtenerse:

$$B_i = \sum_{k=1}^{K} usage(k,i) \cdot CS(k)$$

- Donde *usage(k,i)* es una función binaria que vale 1 cuando el recurso *k*:
  - Es usado al menos por una tarea de prioridad menor a la de la tarea *i*, y
  - Es usado al menos por una tarea de prioridad mayor o igual a la de la tarea *i*
- La función *CS(k)* representa el coste computacional más desfavorable en el uso de la sección crítica *k*.
- Ejemplo: $B_3 = Usage(A,3) \cdot CS(A) + Usage(B,3) \cdot CS(B) = 1 \cdot 4 + 0 \cdot 2 = 4$

7.3. Planificación con prioridades.

7.3.3. Interacción entre tareas y bloqueos.

- Una tarea puede ser bloqueada por recursos a los que no accede (ej. tarea 2).
- Una tarea puede ser bloqueada aunque no acceda a ningún recurso (ej. tarea 3).
- La tarea de menor prioridad, como es evidente, nunca sufrirá ningún bloqueo.
- Considerando bloqueos, la ecuación para el cálculo del tiempo de respuesta que se obtiene

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

y la relación de recurrencia para poder resolverla

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

7.3. Planificación con prioridades.

7.3.3. Interacción entre tareas y bloqueos.

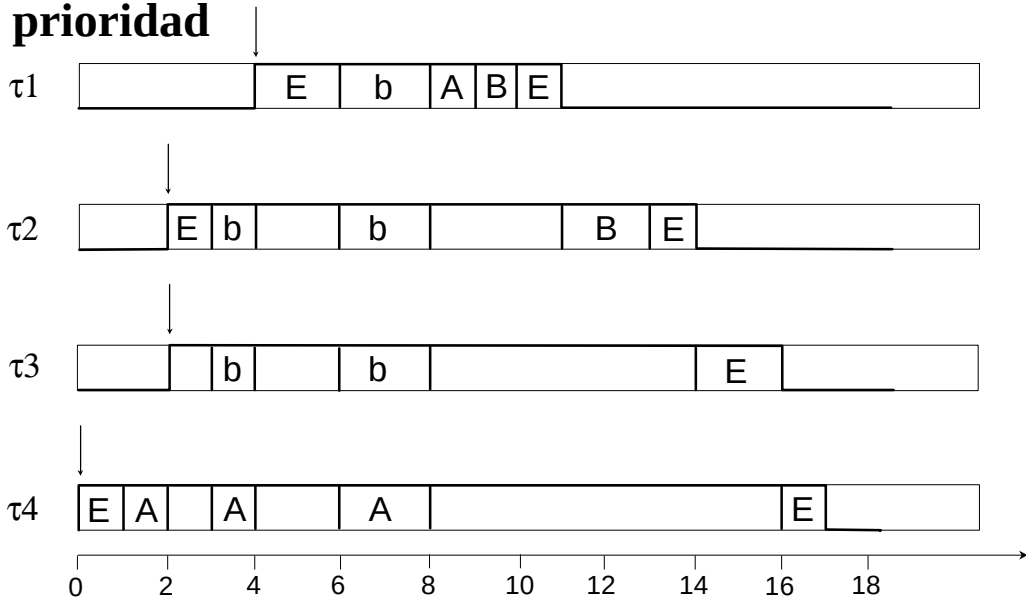
Protocolos de techo de prioridad

- Techo de prioridad de un recurso: Máxima prioridad de las tareas que lo usan.
- El protocolo del techo de prioridad original consiste en:
  - La prioridad dinámica de la tarea es el máximo de su prioridad básica y las prioridades de las tareas a las que bloquea.
  - Una tarea sólo puede usar un recurso si su prioridad dinámica es mayor que el techo de todos los recursos en uso por otras tareas.
- Aseguran que si un recurso es poseído por una tarea, T_1, y esta posesión puede conducir al bloqueo de una tarea de mayor prioridad T_2, entonces no se permite asignar a ninguna tarea distinta de T_1 recurso alguno que pudiera bloquear T_2.
- Propiedades en un sistema monoprocesador:
  - Cada tarea se puede bloquear una vez en cada ciclo, como máximo.
  - No puede haber interbloqueos.
  - No puede haber bloqueos transitivos (T_3 bloqueado por T_2 y éste por T_1).
  - Se consigue exclusión mutua sin necesidad de mecanismos de protección.

7.3. Planificación con prioridades.

7.3.3. Interacción entre tareas y bloqueos.

Protocolos de techo de prioridad



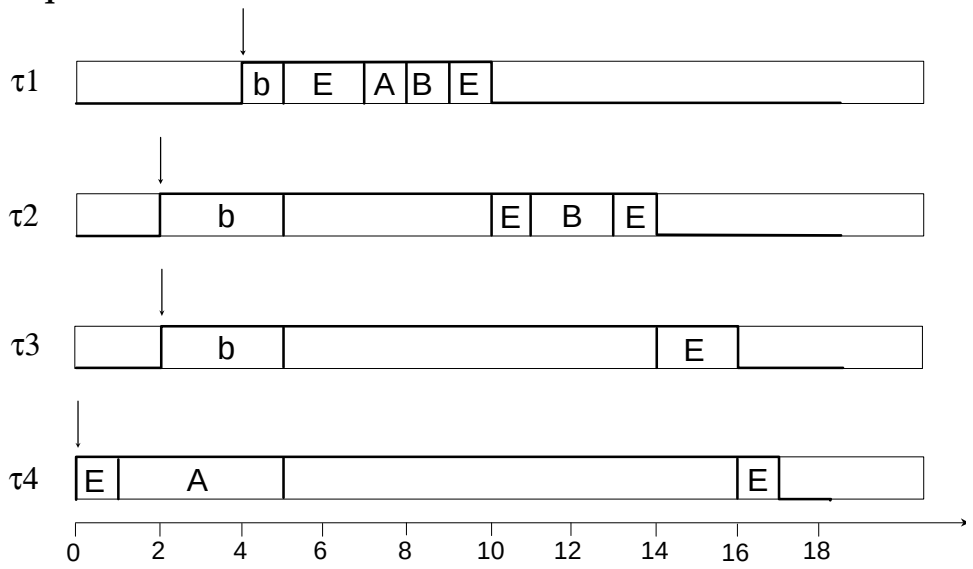
- La primera reserva de un recurso es siempre permitida. El objetivo del protocolo es asegurar que un segundo recurso pueda ser reservado sólo si no existe una tarea más prioritaria que pueda usar ambos recursos.
- La duración máxima de bloqueo es ahora $$B_i = \max_{k=1}^{K} usage(k,i) \cdot CS(k)$$

7.3. Planificación con prioridades.

7.3.3. Interacción entre tareas y bloqueos.

Protocolo de techo de prioridad inmediato

- En este caso, una tarea que reserva un recurso hereda inmediatamente el techo de prioridad del recurso ⇒ la prioridad dinámica de una tarea es el máximo de su prioridad y los techos de prioridad de los recursos que usa.
- Las propiedades son las mismas que para el protocolo original, pero además se cumple que si una tarea se bloquea, lo hace sólo al comienzo del ciclo.
- Es más fácil de implementar que el protocolo original y produce menos cambios de contexto (con lo que resulta más eficiente).



7.3. Planificación con prioridades.

7.3.4. Activación irregular.

- El problema de la activación irregular surge cuando dejamos de suponer que cada tarea es activada tan pronto como llega.
- El *jitter*, J_i, es el tiempo máximo que puede transcurrir desde que llega una tarea hasta que se activa.
- Ejemplo: Con lo visto hasta ahora no es suficiente.

Tarea	C	T	D	J
Tarea1	3	12	8	4
Tarea2	6	20	10	0

• La tarea 1 tiene más prioridad que la 2, queremos comprobar si esta última cumple sus plazos.

• La tarea 1 experimenta un bloqueo externo debido a que necesita un mensaje antes de que pueda empezar, y se garantiza que no tarda en llegar más de 4 ticks.

  - Si se ignora el *jitter*, R_2 (= 9) ≤ D_2 (= 10) ⇒ Parece que cumple los plazos.
  - Sin embargo, si se considera el *jitter* hay ocasiones en las que la tarea 2 no cumple los plazos: T1 llega en el instante 0 y queda esperando el mensaje que llega en el instante 4, el mismo en el que T2 es activada. En la siguiente liberación de T1, el mensaje llega inmediatamente (no hay que esperar) ⇒ la tarea 2 pierde un plazo.

7.3. Planificación con prioridades.

7.3.4. Activación irregular.

- El tiempo de respuesta de la tarea *i* se obtiene mediante la ecuación:

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j$$

- Las tareas periódicas no suelen tener *jitter*, sin embargo éste puede aparecer si la planificación se hace con una granularidad apreciable.
- Ejemplo: Un valor de *T* de 10 en un sistema con granularidad de 8 implica un *jitter* máximo de 6 (en el instante 16 el proceso periódico será activado para la invocación correspondiente al instante 10).
- Si se mide el tiempo de respuesta en relación al tiempo de activación real, entonces hay que añadir el *jitter* al valor previamente calculado:

$$R'_i = R_i + J_i$$

7.4. Realización de sistemas con prioridades.

- En el anexo de tiempo real se define un modelo de planificación con prioridades y desalojo.
- La prioridad de una tarea es de un subtipo definido en el paquete *System*.

```
package System is   ...
  subtype Any_Priority is Integer range definido por la implementación;
  subtype Priority is Any_Priority
        range Any_Priority'First .. definido por la implementación;
  subtype Interrupt_Priority is Any_Priority
        range Priority'Last+1 .. Any_Priority'Last;
  Default_Priority : constant Priority :=
        (Priority'First + Priority'Last)/2;
private ...
end System;
```

- Los valores mayores denotan prioridades más altas.
- La prioridad básica de una tarea o un tipo tarea se indica con un *pragma* **priority** en la especificación:

```
task Controller is
    pragma Priority(10);
end Controller;
```

7.4. Realización de sistemas con prioridades.

- Si una definición de tipo tarea contiene este pragma, entonces todas las tareas de ese tipo tendrán la misma prioridad a no ser que se utilice un discriminante:

```
task type Servers(Task_Priority : System.Priority) is
  entry Service1(. . .);
  entry Service2(. . .);
  pragma Priority(Task_Priority);
end Servers;
```

- Si no se hace la inclusión, se utiliza como prioridad el valor por defecto *Default_Priority*.
- Cualquier tarea que no usa el pragma *Priority* hereda la prioridad de la tarea que la creo.
- El planificador de Ada, por defecto, ejecuta las tareas dispuesta en orden de activación. Ahora bien, si se usa el pragma

```
Pragma Task_Dispatching_Police(FIFO_Within_Priorities);
```

  la ejecución de las tareas se realiza en función de su prioridad, y en caso de igualdad de prioridad se aplica el orden de llegada.

7.4. Realización de sistemas con prioridades.

- En una cola de entrada de tareas, como la que puede aparecer en una sentencia *accept* de un servidor, por defecto, el núcleo de Ada las ejecuta en orden de llegada, sin embargo las tareas pueden ser despachadas por orden de prioridad si se incluye el pragma:

  Pragma Queuing_Police(Priority_Queuing);
- Uno de los problemas asociados con la planificación con prioridades es el de la inversión de prioridades.
- Para tratar este problema, Ada permite asignar prioridades también a los objetos protegidos (recursos) con el objetivo de implementar protocolos de techo de prioridades.
- Para utilizar el protocolo de techo de prioridades, un programa Ada tiene que incluir el pragma:

  Pragma Locking_Policy(Ceiling_Locking);
- La prioridad definida mediante el pragma *Priority* se denomina en Ada **prioridad base**, y corresponde a la prioridad estática.
- Cuando una tarea entra en un objeto protegido, adquiere el techo de prioridad del objeto si éste es mayor que la prioridad base de la tarea, esta prioridad heredada se denomina en Ada **prioridad activa**.

7.4. Realización de sistemas con prioridades.

- Hay otras formas de heredar una prioridad más alta, además del uso de un objeto protegido:
  - Una tarea se activa con la prioridad dinámica del padre.
  - Cuando ejecuta una instrucción *accept* hereda la prioridad de la tarea que llama al punto de entrada.

7.5. Otros métodos de planificación.

- Hasta ahora la prioridad de una tarea es estática (aparte de la operación de un protocolo de herencia de prioridades) y puede determinarse a partir de un simple algoritmo como, por ejemplo, una asignación monótona en frecuencia. Se pueden conseguir mejores resultados realizando la planificación **dinámicamente** en función de los requisitos temporales y de los recursos disponibles.
- Dos métodos dinámicos interesantes son:
  - Primero el más urgente (*Earliest Deadline First, EDF*): Se basa en la ejecución de la tarea con el plazo más cercano siempre, haciendo una elección no determinista en caso de empate.
  - Primero el menos holgado (*Least Laxity First, LLF*): Este algoritmo se basa en la ejecución de la tarea (entre aquellas que hubieran llegado) con menos holgura, y al igual que antes se hace una elección no determinista caso de empate. La holgura de una tarea en el instante t es $L(t) = D - C(t) - t$
- Ambos son óptimos para tareas independientes:
  - Se comportan bien cuando hay muchas tareas esporádicas.
- Problemas:
  - En caso de sobrecarga, el comportamiento es imprevisible.
  - No está bien resuelta la interacción entre tareas.

7.6. Ejemplos.

- Ejemplo 1: Dadas la siguientes tareas *P* (T=3, C=1), *Q* (T=6, C=2) y *S* (T=18, C=5):

  a) ¿Cumplen los plazos suponiendo una asignación de prioridades monótona en frecuencia?

  Sol.: Con RM las prioridades se asignarían: $P = 3$, $Q = 2$ y $S = 1$.

  1- Con RM podemos aplicar el test basado en el factor de utilización:

  $U = 1/3 + 2/6 + 5/18 = 17/18 = 0.945 > 0.78$ (no pasa el test)

  Recordar que este test impone una condición **suficiente** pero **no necesaria**.

  2- Aplicamos el análisis de tiempo de respuesta:

  $R_P = 1$; $R_Q = 3$; $R_S = 17 \Rightarrow$ Se cumplen los plazos!!

  b) Construir un ejecutivo cíclico para la planificación.

  Sol.:

  $T_M = mcm(3, 6, 18) = 18$

  $T_S = T_P / k = 3$

  Por tanto, el ciclo principal cuenta con 6 ciclos secundarios.

Ciclo	***Tareas***
1	P; Q
2	P; S
3	P; Q
4	P; S
5	P; Q
6	P; S

7.6. Ejemplos.

- Ejemplo 2: Un programa consta de cinco tareas A, B, C, D y E (en orden de prioridad) y seis recursos compartidos R1, ..., R6 (protegidos por semáforos e implementando el protocolo de techo de prioridad). Calcular el máximo bloqueo para cada tarea.

Tarea	Utiliza
A	R3
B	R1, R2
C	R3, R4, R5
D	R1, R5, R6
E	R2, R6

Recurso	Tiempo
R1	50
R2	150
R3	75
R4	300
R5	250
R6	175

Sol.: Recordar $B_i = \max_{k=1}^{K} usage(k,i) \cdot CS(k)$

$B_A = max(50 \cdot 0 + 150 \cdot 0 + 75 \cdot 1 + 300 \cdot 0 + 250 \cdot 0 + 175 \cdot 0) = 75$

$B_B = max(50 \cdot 1 + 150 \cdot 1 + 75 \cdot 1 + 300 \cdot 0 + 250 \cdot 0 + 175 \cdot 0) = 150$

$B_C = max(50 \cdot 1 + 150 \cdot 1 + 75 \cdot 0 + 300 \cdot 0 + 250 \cdot 1 + 175 \cdot 0) = 250$

$B_D = max(50 \cdot 0 + 150 \cdot 1 + 75 \cdot 0 + 300 \cdot 0 + 250 \cdot 0 + 175 \cdot 1) = 175$

$B_E = max(50 \cdot 0 + 150 \cdot 0 + 75 \cdot 0 + 300 \cdot 0 + 250 \cdot 0 + 175 \cdot 0) = 0$

TEMA 8. SISTEMAS OPERATIVOS PARA TIEMPO REAL.

8.1. Introducción.

- Un sistema operativo para tiempo real es un sistema operativo capaz de garantizar los requisitos temporales de los procesos que controla.
- Los sistemas operativos convencionales no son apropiados para la realización de sistemas de tiempo real, debido a que
  - no tienen un comportamiento determinista y
  - no permiten garantizar los tiempos de respuesta.
- Un sistema operativo para tiempo real debe ofrecer las siguientes facilidades:
  - concurrencia: procesos ligeros (*threads*) con memoria compartida.
  - temporización: medida de tiempos y ejecución periódica.
  - planificación: prioridades fijas con desalojo, acceso a recursos con protocolos de herencia de prioridad.
  - dispositivos de E/S: acceso a recursos hardware e interrupciones.
- Existen varios sistemas operativos para tiempo real: QNX, Lynx, VxWorks, RT-Linux, KURT,... e incluso soluciones para usar Windows NT como RTOS.
- Nos centraremos en el estudio de RT-Linux y mostraremos las extensiones de tiempo real ofrecidas por el estándar POSIX (estándar IEEE 1003.1b).

8.2. Problemas de los sistemas operativos convencionales.

- Existen algunas características de los sistemas operativos convencionales que impiden su uso como RTOS (*Real-Time Operating System*).
- Lo veremos en el caso particular de un sistema operativo UNIX.
- Problemas:
  1. Planificación para tiempo compartido:
     - Uso de planificadores que aseguran un uso equitativo del tiempo de CPU entre todos los procesos.
     - Es conveniente para un usuario que usa el sistema desde una terminal.
     - No para procesamiento de tiempo real, ya que la ejecución de cualquier proceso depende de forma compleja e impredecible de la carga del sistema y el comportamiento del resto de procesos.
  2. Baja resolución del temporizador:
     - Históricamente a los procesos de usuario se les proporcionaban señales de alarma y la llamada al sistema *sleep()* con sólo 1 segundo de resolución $\Rightarrow$ No es suficiente para procesamiento de tiempo real.
     - Las versiones más modernas proporcionan medios de especificar intervalos con mayor precisión.

8.2. Problemas de los sistemas operativos convencionales.

- Problemas (cont.):
  3. Núcleo no desalojable:
     - Los procesos que se ejecutan en modo núcleo no pueden ser desalojados.
     - Una llamada al sistema podría tardar demasiado tiempo para poder admitirlo en procesamiento de tiempo real.
  4. Deshabilitación de interrupciones:
     - Muy cercano al anterior está el problema de la sincronización.
     - Para proteger los datos que podrían ser accedidos asíncronamente, algunos diseñadores optan por inhibir las interrupciones durante las secciones críticas $\Rightarrow$ más eficiente que los semáforos.
     - Sin embargo, pone en peligro la posibilidad de responder a eventos externos de forma adecuada.
  5. Memoria Virtual:
     - En STR introduce un nivel de impredecibilidad intolerable.

8.2. Problemas de los sistemas operativos convencionales.

- Algunas soluciones:
  1. MINIX OS:
     - Cambiar el planificador *round-robin* de tiempo compartido por uno basado en prioridades.
     - No hay problemas con la paginación ni el *swapping* (no se usan).
     - Es una solución aceptable para sistemas sencillos.
  2. POSIX.1b-1993:
     - Estándar para introducir características de tiempo real en UNIX.
     - Define planificación con prioridades, bloqueo de páginas de memoria del usuario en memoria, señales para tiempo real, IPC, timers . . .
  3. QNX:
     - Cumple el estándar POSIX.1b.
     - Arquitectura de microkernel $\Rightarrow$ el núcleo implementa sólo cuatro servicios: planificación de procesos, comunicación entre procesos, comunicación de red de bajo nivel y manejo de interrupciones.
     - El resto de servicios se implementan como procesos de usuario.

8.2. Problemas de los sistemas operativos convencionales.

- Algunas soluciones (cont.):
  4. VxWorks:
     - RTOS patentado orientado hacia la aproximación *host*/*target*.
     - Un *host* UNIX es usado para el desarrollo del software y para la ejecución de las partes de la aplicación que no son de tiempo real.
     - El núcleo de VxWorks (llamado *wind*) ejecuta las tareas de tiempo real en la máquina *target*.
     - Las máquinas se comunican utilizando TCP/IP.
     - Proporciona algunas funciones POSIX.1b.
  5. REAL/IX:
     - Es un sistema UNIX (a partir del UNIX System V) al que se le han añadido capacidades para procesamiento de tiempo real.
     - Cumple el estándar POSIX.
  6. Windows NT:
     - Tendencia por usar un SO para todo (incluido tiempo real).
     - Núcleo de Windows NT no sirve para procesamiento de tiempo real.

8.2. Problemas de los sistemas operativos convencionales.

- Algunas soluciones (cont.):

  6. Windows NT (cont.):

     - Varias soluciones proporcionadas por distintas compañías:
       - Intime de RadiSys: modifica la capa Hardware Abstraction Level para atrapar los intentos de Windows para desactivar las interrupciones o resetear el reloj.
       - Los diseñadores de QNX implementan la API de Win32 sobre su SO POSIX-*compliant*.

- Direcciones de interés:
  - QNX: *http://www.qnx.com*
  - VxWorks: *http://www.wrs.com/products/html/vxwks52.html*
  - REAL/IX: *http://209.249.130.161/real_time/c_and_c/c_and_c.html*
  - Real-Time & Windows NT:
    - *http://www.radisys.com/products/rtos/nt_prod.html*

8.3. Sistemas operativos para tiempo real basados en Linux.

- Existen varias posibilidades para la realización de STR utilizando el sistema operativo Linux:
  - Real-Time Linux: Proporciona un núcleo de tiempo real basado en un planificador con desalojo y prioridades fijas. Está preparado para la gestión de tareas críticas.
  - KU Real-Time Linux: Se trata de la realización de un sistema para planificación cíclica de procesos acríticos. Es útil para procesos acríticos como los relacionados con la grabación y reproducción de imágenes en movimiento.
  - POSIX: La versión del núcleo estándar de Linux 2.0.36 implanta algunas llamadas requeridas por la especificación POSIX 1003.1b.

8.3.1. Real-Time Linux (RTL).

- Es una modificación de código de Linux para gestionar tareas críticas.
- Está disponible la versión 1.1 para la versión 2.0.36 del kernel de Linux.
- Por ahora sólo está disponible para la arquitectura del PC.
- Dirección Web: *http://rtlinux.cs.nmt.edu/~rtlinux/*

8.3. Sistemas operativos para tiempo real basados en Linux.

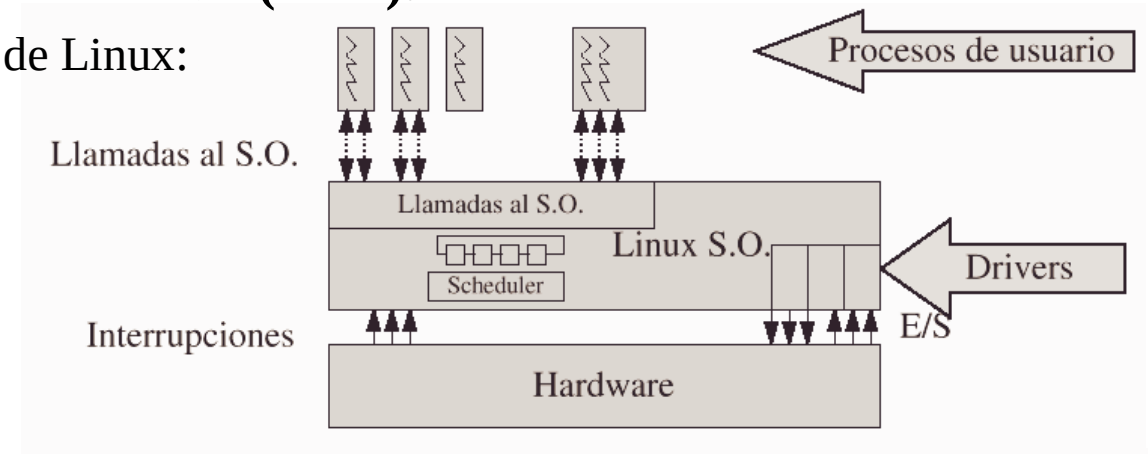
8.3.1. Real-Time Linux (RTL).

- Características:
  - Incluye un planificador con desalojo y prioridades fijas, para la ejecución de tareas críticas de tiempo real.
  - Las tareas pueden ser periódicas o bien activadas mediante una interrupción (esporádicas o aperiódicas).
  - Incorpora mecanismos para la comunicación con los procesos no críticos, que son los de Linux *normal*. Estos mecanismos son colas FIFO.
  - Las tareas de tiempo real se ejecutan con la CPU en modo *supervisor* (pueden acceder a puertos E/S, reprogramar interrupciones, etc...).
  - Convierte al núcleo de Linux en una tarea más, pero de segundo plano (de prioridad mínima).
- RT-Linux es como un microkernel que realiza operaciones muy básicas: gestionar todas las interrupciones y planificar las tareas de tiempo real.
- Linux deja de disponer de las instrucciones *cli*, *sti* e *iret* para tener en su lugar llamadas al propio microkernel de RT-Linux, que las simula.
- Linux pierde el control del sistema y no se ejecutará si las tareas críticas ocupan toda la CPU $\Rightarrow$ El sistema puede bloquearse aparentemente.

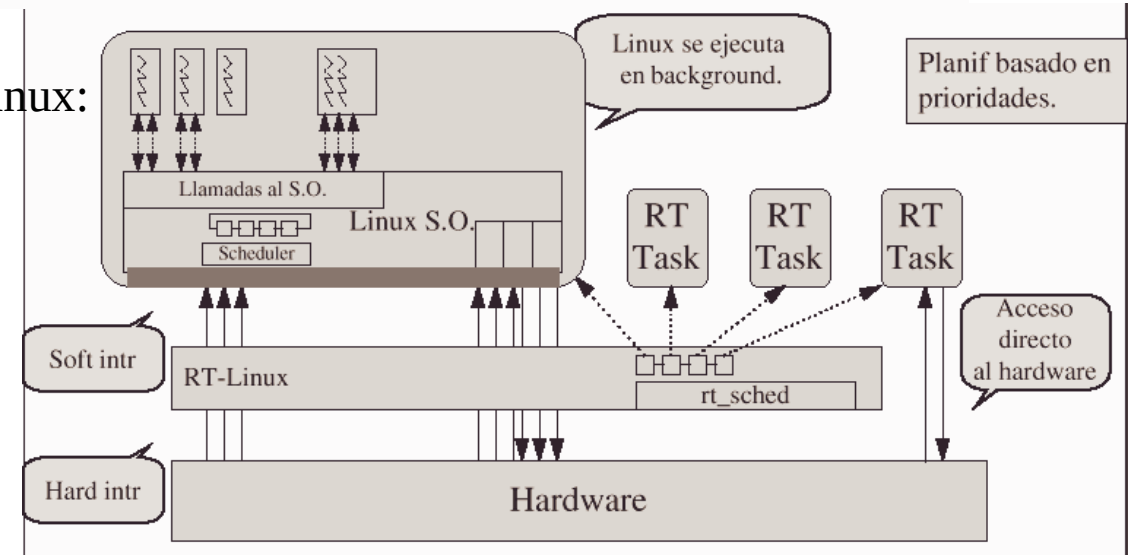
8.3. Sistemas operativos para tiempo real basados en Linux.

8.3.1. Real-Time Linux (RTL).

- Estructura de Linux:



- Estructura de RT-Linux:



8.3. Sistemas operativos para tiempo real basados en Linux.

8.3.1. Real-Time Linux (RTL).

- Las tareas de tiempo real en RT-Linux son código que se ejecuta en modo supervisor de la CPU, para tener acceso directo a los dispositivos de E/S.
- Para ejecutar procesos en modo supervisor no basta con hacerlo desde la cuenta *root*, sino que además es necesario que sea parte del código del núcleo.
- El código del núcleo no se pagina ⇒ un tarea no puede ser expulsada a disco.
- Es necesario programar las taras como módulos de carga (el propio RT-Linux se ha hecho como un módulo).
- Un módulo de carga es un programa que podemos realizar en C en el que:
  - Carece de función *main*.
  - Tiene una función para iniciar el módulo *init_module()* ⇒ se ejecuta al cargarlo y desde ella llamaremos a otras.
  - Tiene una función para finalizar el módulo *cleanup_module()* ⇒ se ejecuta al descargar el módulo.
- El núcleo no dispone de salida estándar, por lo que no podremos usar las funciones de E/S por pantalla habituales, en su lugar usaremos *printk*.

8.3. Sistemas operativos para tiempo real basados en Linux.

8.3.1. Real-Time Linux (RTL).

- Ejemplo:

```
#define MODULE
#include <linux/module.h>
static int x,y;
int init_module (void) { printk("Iniciando módulo..."); return(0); }
int cleanup_module (void) { printk("Finalizando módulo..."); return(0); }
```

- Todo lo que imprimimos con la función *printk* va al anillo de mensajes de Linux, que es explorado por *syslog* para su registro y presentación por pantalla.
- Para compilar el módulo: *gcc -O2 -Wall -D__KERNEL__ -c mimodulo.c*
- Para cargar el módulo: *insmod mimodulo.o* ó *modprobe mimodulo.o*
- Para descargar el módulo: *rmmod mimodulo.o*
- Para ver los módulos cargados: *lsmod*
- Es posible pasar parámetros al módulo, para ello se han definido las variables globales estáticas *x* e *y*, para darles valor: *insmod mimodulo.o x=5 y=6*

8.3. Sistemas operativos para tiempo real basados en Linux.

8.3.1. Real-Time Linux (RTL).

La API de RT-Linux:

- Gestión de tareas según un esquema de prioridades fijas:

rt_task_init	rt_task_suspend
rt_task_delete	rt_task_wait
rt_task_make_periodic	rt_task_wakeup

- Asignación de manejadores de interrupción:

request_RTint	free_RTint

- Comunicación con aplicaciones Linux de usuario:

rtf_create	rtf_get
rtf_destroy	rtf_put
rtf_create_handler	rtf_resize

- Alta resolución de medida de tiempos:

  rt_get_time

8.3. Sistemas operativos para tiempo real basados en Linux.

8.3.1. Real-Time Linux (RTL).

- Ejemplo: Tarea periódica.

gcc -O2 -Wall -D__RT__ -D__KERNEL__ -c ejemplo.c

```
#define MODULE
#include <linux/module.h>
#include <linux/rt_sched.h>
RT_TASK mi_tarea;
void accion(int no_usado) {
  static int activacion;
  while (1) {  printk("Activación número %d\n", cont++);
               rt_task_wait();  } }
int init_module (void) {
  rt_task_init(&mi_tarea, accion, 1, 1000, 1);
  rt_task_make_periodic(&mi_tarea, (RTIME) rt_get_time() + (RTIME) RT_TICKS_PER_SEC,
                                   (RTIME) RT_TICKS_PER_SEC);
  return(0); }
int cleanup_module (void) {
  rt_task_delete(&mi_tarea);
  return(0); }
```

Constituye la tarea periódica

Registra la tarea en el planificador

Instante de la primera activación

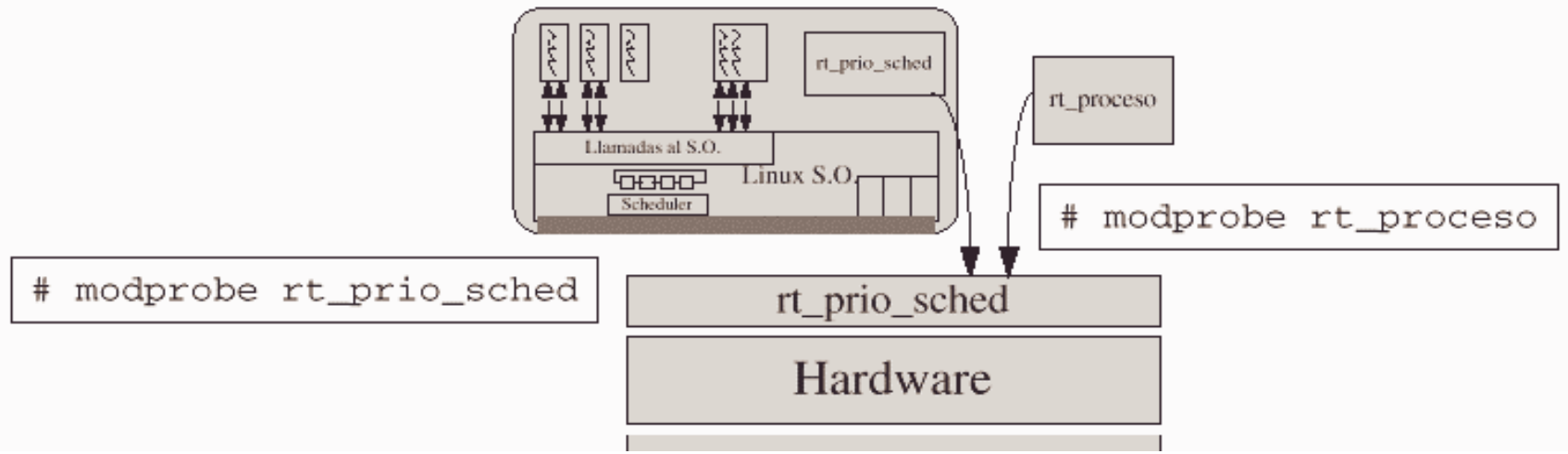
Periodo

Borra la tarea del planificador

8.3. Sistemas operativos para tiempo real basados en Linux.

8.3.1. Real-Time Linux (RTL).

- Para poder ejecutar tareas de tiempo real tenemos que cargar el módulo del planificador. Al hacerlo, Linux pasa a ser una tarea de tiempo real (RT_TASK) con la prioridad más baja.
- Si durante la ejecución de otras tareas de tiempo real se produce una interrupción que debe atender Linux, se guarda temporalmente hasta el momento en que Linux sea la tarea activa.
- Una tarea RT_TASK se carga igual que cualquier módulo.



8.3. Sistemas operativos para tiempo real basados en Linux.

8.3.1. Real-Time Linux (RTL).

- La idea de RT-Linux es la de dividir una aplicación de tiempo real en dos partes:
  - Parte de tiempo real: Incluye el código que es crítico en tiempo y debe mantenerse lo más simple posible.
  - Parte de no-tiempo real: Realiza el procesamiento de los datos, incluyendo interfaces de usuario así como el archivo y distribución de los datos.

¿Cómo se comunican?

- RT-Linux proporciona colas de tiempo real (RT-FIFO) para la comunicación de ambas partes así como entre tareas de tiempo real. Son similares a las tuberías UNIX.
- Las FIFO son globales a todo el sistema y se identifican por un número.
- Las tareas RT_TASK hacen uso de las FIFO usando las funciones de la API para comunicación con tareas de usuario.
- Las tareas de usuario acceden empleando mecanismos estándar de UNIX sobre ficheros (funciones *open*, *read*, *write* y *close*). Concretamente a las FIFO se accede mediante los ficheros especiales */dev/rtf*?, donde ? = 0, 1, 2 ...

8.3. Sistemas operativos para tiempo real basados en Linux.

8.3.1. Real-Time Linux (RTL).

- Otra forma de comunicación es memoria compartida

- RT-Linux se instala como un parche (*patch*) sobre el código fuente de Linux:
  - Este parche modifica el código fuente de Linux para evitar que éste acceda directamente a las interrupciones y a algunas instrucciones máquina importantes (*cli*, *sti*, *iret*).
  - También se instalan los ficheros fuente del planificador de tiempo real y de otras facilidades de tiempo real (*rt-fifo*, *rt-time*).

    patch -p1 < donde_esta/parche
- Hay que recompilar el núcleo de Linux de la forma habitual:

make config; make dep; make clean; make zImage; make modules; make modules_install;

- Se reinicia la máquina.
- En este momento Linux se comporta normalmente, hasta que no carguemos el planificador no tendremos el sistema de tiempo real.

8.3. Sistemas operativos para tiempo real basados en Linux.

8.3.2. KU Real-Time Linux.

- Es útil para sistemas de tiempo real *firm*: sistemas que tienen unos requisitos de temporización de grano fino (típicos de los críticos) junto con los requerimientos de los servicios de los acríticos.
- Por ejemplo: aplicaciones multimedia de vídeo tienen requisitos temporales típicos de un sistema crítico y necesitan muchos servicios del sistema.
- Al contrario que en RT-Linux, las tareas pueden hacer uso de todas las facilidades de Linux.
- Las modificaciones que se han llevado a cabo sobre el núcleo son:
  - Mejorar la granularidad del reloj del sistema: en Linux-i386 la frecuencia con la que se interrumpe el reloj es de 10 ms (100 veces por segundo), y es con esta resolución temporal con la que se toman las acciones de control y se mide el tiempo. KURT programa el chip de reloj para que genere interrupciones bajo demanda, en vez de periódicamente. Se logran una resolución superior al microsegundo.
  - Se ha modificado el planificador para incluir una nueva política de planificación (SCHED_KURT) además de las que Linux ya implementa.
  - Se han añadido nuevas llamadas al sistema para poder hacer uso de las nuevas funcionalidades de tiempo real.

8.3. Sistemas operativos para tiempo real basados en Linux.

8.3.2. KU Real-Time Linux.

- Las tareas de tiempo real son módulos de carga dinámica.
- Se ha implementado un planificador cíclico que se basa en el uso de una tabla (plan) en la que están anotadas todas las acciones de planificación (instante de activación, tarea a ejecutar, duración ...).
- La tabla se construye durante la fase de diseño del sistema y el trabajo del planificador consiste en leer secuencialmente la tabla y seguir sus indicaciones ⇒
  - El planificador es muy sencillo de implementar y eficiente.
  - Dificultad a la hora de construir el plan.

- Más información: *http://hegel.ittc.ukans.edu/projects/kurt/index.html*