3. Programación de los Sistemas de Tiempo Real

Contenido

3.1 Introducción

En la programación de tiempo real existen unas cuantas nociones que se utilizan en la mayoría de los sistemas, por lo que merecen una atención especial. Estas son:

- Programación concurrente
- Comunicación y sincronización
- Fiabilidad y tolerancia a fallos
- Manejo de excepciones
- Uso del tiempo

Algunas ya se han nombrado en capítulos anteriores. Ahora las vamos a estudiar con mayor detalle, muchas de ellas apoyándonos en los lenguajes de programación adecuados para tiempo real, en particular el Ada y el C++.

3.2 Programación concurrente

Virtualmente, todos los sistemas de tiempo real son en sí concurrentes. Los lenguajes que tienen incorporada la programación concurrente da al programador grandes posibilidades de expresión utilizando las primitivas que ofrece el lenguaje.

Cualquier lenguaje, natural o de ordenador, tiene la doble propiedad de facilitar una serie de expresiones y, al mismo tiempo, limitar el marco en el que se pueden utilizar sus expresiones (por ejemplo, el lenguaje natural y el lenguaje matemático). Si el lenguaje no soporta un determinado concepto o noción, entonces el uso de ese lenguaje para expresar esta noción puede ser inadecuado.

Los lenguajes como el Pascal, el C, el FORTRAN o el COBOL son lenguajes secuenciales. En ellos la ejecución comienza en único punto y, a partir de ahí continúa ejecutándose una instrucción por vez hasta que termina. El camino o hilo por el que transcurre el flujo del programa puede variar según los datos de entrada, pero siempre hay un único camino. Por tanto, estos lenguajes, en sí, no son adecuados para la programación en tiempo real.

Definición:

Un lenguaje de programación concurrente es aquel que permite expresar una colección de procesos secuenciales autónomos, ejecutándose "lógicamente" en paralelo.

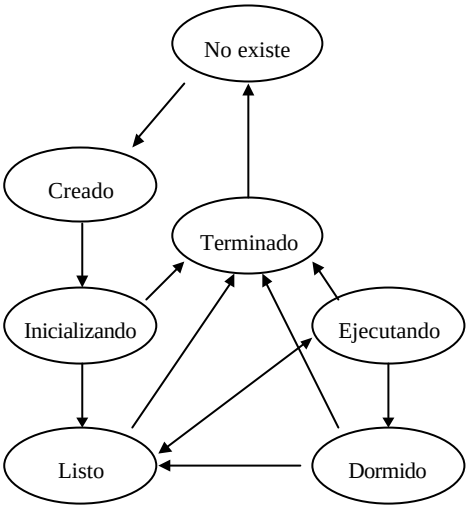
La implementación, es decir, la ejecución, de una colección de procesos normalmente se puede realizar de tres formas distintas. Los procesos pueden ser:

1) Con ejecución multiplexada en un único procesador.
2) Con ejecución multiplexada en un sistema multiprocesador con acceso a memoria compartida.
3) Con ejecución multiplexada en varios procesadores que carecen de memoria compartida. Este es el caso de los sistemas distribuidos.

También es posible encontrar sistemas híbridos de los tres anteriores.

Solamente en los casos 2) y 3) es posible la ejecución paralela real de más de un proceso. El término **concurrente** indica paralelismo potencial. Los lenguajes de programación concurrente permiten expresar de forma lógica distintas actividades paralelas, sin considerar su implementación.

La vida de un proceso va variando tomando distintos estados, como se ve el siguiente diagrama:



Estados de un proceso

Un proceso es creado, inicializado y puesto en ejecución hasta que termina. Hay que tener en cuenta que algunos procesos puede que no finalicen nunca, o que terminen antes de ponerse en ejecución por fallar su inicialización.

El estado más importante de un proceso es cuando se está ejecutando, ahora bien, cuando hay más procesos ejecutándose simultáneamente que el número CPU del sistema, habrá procesos que estarán listos para ser ejecutados pero en espera de CPU (listos).

Otro estado normal de un proceso es el de dormido, esperando a que se produzca un evento, a que esté libre algún recurso que necesita, o que le llegue el tiempo de su ejecución.

Con esta consideración de un proceso, está claro que la ejecución de un programa concurrente no es tan sencilla como la ejecución de un programa secuencial. Los procesos deben ser creados y finalizados, y activados cuando el procesador está disponible. Estas actividades las lleva a cabo el Run - Time Support System (RTSS) o Run - Time Kernel. El RTSS tiene muchas de las propiedades de un planificador en un Sistema Operativo y desde el punto de vista lógico está situado entre el hardware y la aplicación software. En realidad, puede tomar varias formas:

- Un programa incorporado como parte de la aplicación, esto es, un componente del programa concurrente. Este es el mecanismo utilizado por el lenguaje Modula-2.

- Un sistema software estándar generado con el código del programa objeto por el compilador. Esta es la estructura normal utilizada por el lenguaje Ada.

- Un microcódigo incorporado en el microprocesador para mejorar la eficiencia. Los sistemas que funcionan con tránsputers programados con occam2 utilizan esta técnica.

El algoritmo utilizado para la planificación por el RTSS, esto es, para decidir que proceso se ejecutará después del actual, afectará al comportamiento temporal del programa. Para los sistemas de tiempo real, las características del RTSS son importantes.

Una alternativa para implementar la programación concurrente es utilizar las facilidades que ofrecen la mayoría de los sistemas operativos para crear y administrar varios procesos.

Usualmente cada proceso se ejecuta en una máquina virtual independiente para evitar interferencias, aunque los sistemas operativos más modernos permiten crear procesos que compartan el espacio de direcciones de variables (los llamados procesos ligeros o threads).

La utilización de los threads para implementar la concurrencia mejora la eficiencia del RTSS. Además hace posible que se utilicen desde una tarea las llamadas al sistema bloqueantes sin que se detengan todos los procesos de un programa.

3.2.1 Construcciones para la programación concurrente

Aunque las construcciones para la programación concurrente varían de uno a otro lenguaje y sistemas operativos, hay tres facilidades fundamentales que se deben ofrecer. Estas son:

- La expresión de ejecución concurrente por medio de la noción de proceso.
- Sincronización entre procesos
- Comunicación entre procesos

Con relación a la comunicación y sincronización entre los procesos, es útil distinguir entre tres tipos de comportamiento:

- **Independiente**: Los procesos no se comunican o sincronizan entre ellos, ejecutándose de forma totalmente independiente.
- **Cooperativo**: Se comunican o sincronizan regularmente entre ellos, realizando conjuntamente una determina operación. Un ejemplo sería un sistema en que procesos distintos realizan la medida de los sensores, elaboran unos resultados y modifican el estado de los actuadores.
- **Competitivo**: Varios procesos utilizan unos recursos del sistema que son limitados y, por tanto, la utilización por parte de unos procesos provoca la espera de otros a que estén libres. Ejemplos de estos recursos pueden ser periféricos o memoria.

El tipo de relación entre los procesos es importante a la hora de estudiar el comportamiento temporal del sistema y asegurar que se van a cumplir las restricciones temporales.

3.2.2 Procesos y objetos

La Programación Orientada a Objetos ofrece a los programadores la visión de un sistema como una colección de objetos que colaboran entre ellos. En este paradigma, es conveniente considerar varios tipos de objetos: Activos, pasivos y reactivos.

Los objetos **activos** asumen la posibilidad de realizar acciones de forma espontánea: estos autorizan la ejecución de un proceso. Los objetos activos también se denominan agentes activos.

Los objetos **reactivos** sólo realizan acciones cuando son invocados por un objeto activo. Estos objetos se pueden asociar a los datos del programa. Las acciones que pueden realizar los datos consisten en la modificación o lectura de estos. Los objetos reactivos también se denominan agentes pasivos.

Los objetos reactivos sólo permiten que los utilice un agente a la vez, de lo contrario las operaciones que pueden realizar podrían ser incorrectas. Un ejemplo podría ser un buffer de datos que implemente una cola FIFO. Estos objetos se llaman **recursos protegidos**, en oposición a los objetos **pasivos** que no presentan restricciones en el número de accesos simultáneos.

Si el acceso a un recurso está controlado por una agente activo, a este se le denomina **servidor**. Cuando es utilizado un servidor, bloquea al agente solicitante y toma el control del procesador hasta que finaliza la petición, para despertar al solicitante y se bloquea esperando otra petición.

3.2.3 Representación de los procesos concurrentes

Existen tres mecanismos básicos de representación de la ejecución de procesos concurrentes. Estos son:

- Fork y Join
- Cobegin y Coend
- Declaración explícita.

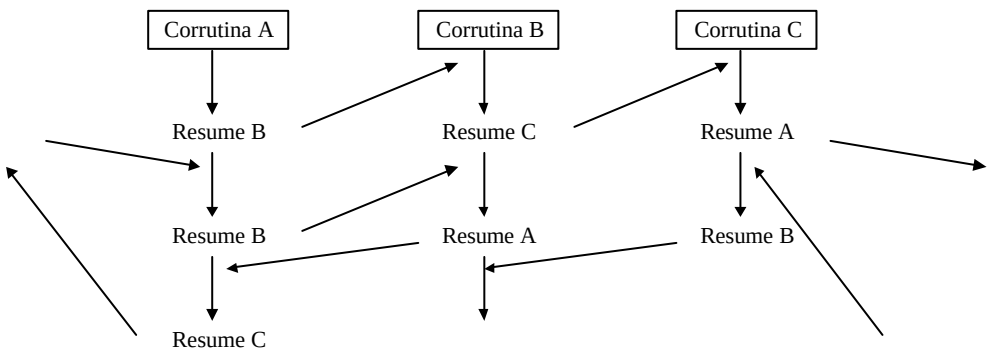
Las corrutinas en ocasiones se incluyen en los mecanismos para expresar la ejecución concurrente, aunque por sí no constituyen procesos.

3.2.3.1 Corrutinas

Las corrutinas son como subrutinas con la diferencia que entre ellas se pueden pasar el control de una manera simétrica, en lugar de una manera jerárquica. El control se pasa de una corrutina a otra con una sentencia **resume**, que significa que se reanude la ejecución de una corrutina en el lugar donde había suspendido su ejecución, suspendiéndose la corrutina actual hasta que sea activada por otra.

Cada corrutina se puede ver como la implementación de un proceso, no obstante no es necesario RTSS pues las mismas corrutinas realizan el paso de una a otra.

Es evidente que el modelo de corrutinas no es adecuado para el modelo de proceso paralelo, en el que, en principio, cada proceso se ejecuta de forma independiente. Modula-2 es un ejemplo de lenguaje que soporta corrutinas. El lenguaje C también permite el uso de corrutinas con las funciones `setjmp` y `longjmp`.



Flujo de control en corrutinas

3.2.3.2 Fork y Join

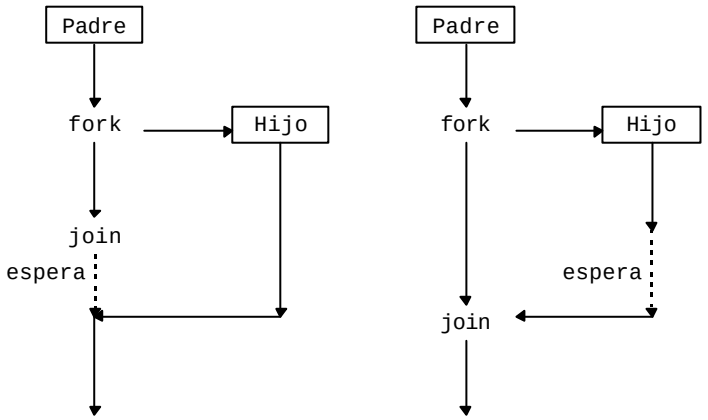
La sentencia `fork` especifica que una determinada rutina comenzará a ejecutarse de forma concurrente con quien invoca a `fork`. La sentencia `join` permite que el invocador se sincronice con la finalización de la rutina que ha lanzado, esperando a que ella termine si no lo ha hecho.

```
function F return ...
.
.
end F;

procedure P
  ...
  C := fork F;
  .
  .
  J := join C;
end P
```

Entre la ejecución del `fork` y del `join` del procedimiento P, la función F se ejecuta de forma concurrente.

En el estándar POSIX existe una versión equivalente al `fork` y al `join`. En éste el `fork` se utiliza para crear una copia del programa invocador que se ejecutará concurrentemente, y la llamada al sistema `wait` realiza las funciones del `join`.



Uso del `fork` y `join`

`fork` y `join` permiten la creación dinámica de procesos y ofrece un mecanismo para pasar información al proceso hijo a través de parámetros. Normalmente, la función `join` solo permite devolver un único parámetro usado para indicar el estado de error al finalizar el proceso hijo.

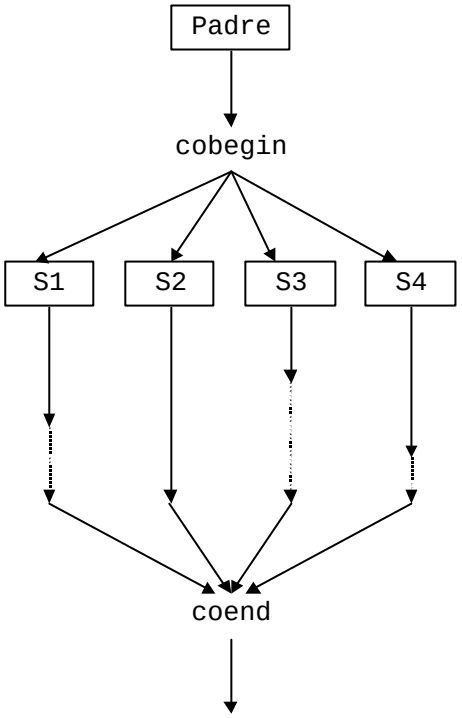
3.2.3.3 Cobegin y Coend

El `cobegin` (o `copar` o `par`) es una forma estructurada de indicar la ejecución concurrente de un grupo de sentencias:

```
cobegin
```

```
    S1;
    S2;
    S2;
    .
    .
    Sn;
coend
```

Esta construcción hace que todas las sentencias `S1`, `S2` … se ejecuten concurrentemente. La sentencia `cobegin` finaliza sólo cuando han terminado todas las sentencias.



Uso del ***`cobegin`*** *y* ***`coend`***

Las sentencias `Sn` pueden ser cualquier construcción que permita el lenguaje, desde simples asignaciones hasta llamadas a subprogramas. Si se emplean llamadas a subprogramas se les puede pasar información a los nuevos procesos utilizando el paso de parámetros.

Una de estas sentencias podría contener a su vez otra construcción cobegin/coend, creándose una jerarquía de procesos.

3.2.3.4 Declaración explícita.

Aunque varias rutinas secuenciales se pueden ejecutar concurrentemente utilizando las construcciones ***`fork/join`*** y ***`cobegin/coend`***, la estructura de un programa puede quedar más clara si las rutinas se declaran ellas mismas como concurrentes. La declaración explícita de procesos ofrece esta posibilidad.

El lenguaje Ada soporta la declaración explícita de procesos con la declaración de tareas. En Ada, todas las tareas declaradas en un mismo bloque comienzan su ejecución de forma concurrente con las sentencias propias del bloque que figuran al final de la parte declarativa. La ejecución del bloque no finalizará hasta que terminen todas las tareas definidas en él.

```
procedure Tareas_concurrentes
    task A;
    task B;

    task body A is
        -- declaraciones locales de la tarea A
    being
        -- secuencia de instrucciones de la tarea A
    end;

    task body B is
        -- declaraciones locales de la tarea B
    being
        -- secuencia de instrucciones de la tarea B
    end;

begin
    -- Las tareas A y B comienzan a ejecutarse
    -- antes de la primera sentencia del programa
    .
    .
end Tareas_concurrentes;
    -- el programa no termina hasta que no terminen
    -- las tareas A y B
```

3.3 Comunicación y sincronización

La mayor dificultad asociada con la programación concurrente se debe a la interacción entre procesos. Muy raramente se utilizan procesos independientes. El comportamiento correcto depende en gran medida de la sincronización y comunicación entre procesos.

En un sentido amplio, la **sincronización** corresponde al cumplimiento de las dependencias temporales entre diferentes procesos (por ejemplo, una determinada acción de un proceso sólo puede ocurrir después de otra acción específica de otro proceso). El término también se utiliza en el caso de que dos o más proceso coincidan en el tiempo en unos estados predefinidos.

Estos dos casos de sincronización corresponden a los modelos cooperativo y competitivo respectivamente referente a la relación entre procesos.

La **comunicación** es el paso de información de un proceso a otro.

Los dos conceptos están relacionados. La comunicación, de alguna manera, requiere la sincronización, y la sincronización se puede interpretar como una forma de comunicación entre los procesos.

La comunicación de datos usualmente está basada en dos mecanismos, las **variables compartidas** y el **paso de mensajes**.

Las variables compartidas son objetos a los que puede acceder a ellas más de un proceso. La comunicación se puede llevar a cabo depositando en ellas información un determinado proceso, la cual será leída por otro.

El paso de mensajes se refiere al intercambio de datos entre dos procesos por medio de un mensaje que se envían de alguna manera.

La elección entre el uso de las variables compartidas y el paso de mensajes depende de la implementación del lenguaje, aunque lo normal es que un lenguaje de programación para tiempo real ofrezca ambos mecanismos. En tal caso, la utilización de una u otra dependerá del programador.

Las variables compartidas son fáciles de utilizar si existe memoria compartida entre los procesos, pero se puede simular si el hardware ofrece algún medio de comunicación, aunque carezca de memoria compartida.

De forma similar, el paso de mensajes se puede implementar tanto utilizando memoria compartida o bien por el mecanismo de envío de mensajes de una red.

3.3.1 Comunicación basada en memoria compartida

El mecanismo de paso de mensajes suele incorporar la sincronización entre el proceso que envía y e que recibe. La utilización de memoria compartida requiere la utilización de métodos de sincronización extra. Existen multitud de mecanismos que permiten distintos tipos de sincronización entre procesos. Vamos a estudiar algunos de ellos.

3.3.1.1 Exclusión mutua y sincronización condicional

Aunque la memoria compartida aparece como un mecanismo sencillo de pasar información entre distintos procesos, se debe utilizar con cuidado para evitar múltiples problemas. Por ejemplo, consideremos la operación:

```
X := X + 1
```

donde la variable X está en una zona de memoria compartida a la que pueden acceder varios procesos.

En muchas plataformas hardware esta operación no se puede hacer de forma indivisible (supóngase por ejemplo que el tamaño de la variable es superior al de los registros que maneja la CPU). Esta operación se implementa entonces en tres instrucciones distintas.

1) Leer el valor de la variable X en algún registro (o en la cima de la pila).

2) Incrementar el valor del registro en 1

3) Guardar el resultado de la operación en la variable X.

Como las tres operaciones no son indivisibles, puede coincidir que dos procesos intenten incrementar a la vez la variable X. Si esto ocurre, y uno de los procesos es expulsado de la CPU después de leer el valor y antes de guardar el resultado, cuando reanude su ejecución deshará el incremento de la variable que haya realizado el otro proceso. Por tanto, tras dos operaciones de incrementar en uno la variable X, su valor solo habrá aumentado una unidad, lo que resulta ser incorrecto.

Si ocurre que la variable X representa un valor que precisa de varias operaciones para ser leída y guardada en memoria por tener un tamaño superior a los registros que maneja la CPU, aparecerían errores hasta en el caso en el que un proceso modifica el valor de la variable y otro simplemente intenta leer su valor. En nuestro caso ocurrirá en los valores que suponen acarreo.

La secuencia de instrucciones que deben ejecutarse en una operación indivisible se llama **sección crítica**. La sincronización requerida para proteger una sección crítica es conocida como **exclusión mutua**.

La exclusión mutua no es el único mecanismo de sincronización importante. En efecto, si dos procesos no comparten variables no necesitan utilizar exclusión mutua. La **sincronización condicional** también en necesaria. Esta se necesita cuando un proceso quiere realizar una operación que sólo es posible, o segura, cuando otro proceso a realizado otra acción o está en un estado determinado.

Un ejemplo de la sincronización condicional es la utilización de buffers. La utilización de buffers es muy común en los sistemas de tiempo real pues permite intercambiar información entre dos procesos de forma desacoplada, permitiendo pequeñas fluctuaciones en la velocidad de funcionamiento de dos procesos.

Supongamos en caso de un proceso consumidor que lee de un buffer y otro productor que escribe en él. Se necesitan dos sincronizaciones condicionales. El consumidor no puede realizar la lectura del buffer mientras el consumidor no ha escrito en él, debiendo quedar bloqueado hasta que escriba el productor. Por otro lado, el productor no podrá escribir en el buffer si éste está lleno, debiendo quedar bloqueado hasta que el consumidor deje espacio suficiente.

En el caso de utilización de buffer en el que existan más de un productor o más de un consumidor, se necesita además realizar exclusión mutua para impedir que varias operaciones simultáneas del mismo tipo corrompan la estructura interna del buffer.

Existen varios mecanismos para implementar la exclusión mutua y sincronización condicional. Algunos de estos son:

- **Espera ocupada**. El proceso que se bloquea realiza un bucle en espera que en una variable se indique que ha finalizado su situación de espera. Resulta poco eficiente pues los procesos en espera permanecen en un bucle inútil consumiendo tiempo de procesador, en especial cuando puede haber más de un proceso en espera.

  Esta es la única posibilidad cuando la comunicación es entre distintos procesadores con memoria compartida, sin posibilidad de interrumpirse uno al otro. En estos casos se utiliza una operación del estilo TAS (Test And Save) que se realiza en un ciclo de bus indivisible (se obtiene el valor de la memoria y se modifica y se guarda de forma indivisible).

- **Suspender y reanudar**. Cuando un proceso debe esperar realiza una llamada a la función 'suspender' que lo deja en estado de bloqueo. Cuando otro proceso decide que ya puede continuar realiza una llamada a la función 'reanudar' sobre el proceso anterior. Si pueden haber más de un proceso en espera se debe mantener una cola.

  Una variante útil de este mecanismo es la operación **reanudar con memoria**. En el caso de reanudar un proceso que no está suspendido, no se devuelve un fallo sino que la operación queda memorizada hasta que se realiza un suspender (en tal caso no se suspendería).

  Estos mecanismos de sincronización también se conocen como señales transitorias y señales persistentes.

- **Semáforos**. Son elementos que facilitan la sincronización y evitan los bucles de la espera ocupada. Se pueden considerar como un contador sobre el que se pueden hacer dos

operaciones. La operación P (espera) decrementa el contador y bloquea al proceso si el contador toma un valor negativo. Y la operación V (señalizar) que incrementa el contador, despertando al primer proceso en espera si hubiera alguno.

En un semáforo, el contador inicial indica el número de procesos con acceso simultáneo al recurso.

En el uso de la exclusión mutua hay que prestar atención a la posibilidad de bloqueos anidados. Es posible que un proceso que tiene acceso exclusivo a una sección crítica intente tomar de nuevo acceso a la misma región crítica. Si el mecanismo de exclusión no prevé esta posibilidad el proceso quedaría bloqueado indefinidamente.

3.3.1.2 Regiones críticas condicionales

Una región crítica es una sección de código en la que se garantiza que se ejecuta con exclusión mutua.

La variables que deben estar protegidas de una utilización concurrente se agrupan juntas en regiones a las que se da un nombre que se etiquetan como recursos. Un proceso no podrá entrar en una región crítica si otro proceso se encuentra en la misma región crítica. Se dispone también de una condición de sincronización para acceder a la sección crítica. Cuando un proceso quiere entrar en una sección crítica evalúa la condición de acceso (bajo exclusión mutua). Si la condición de acceso se evalúa a verdadero puede entrar, pero, si se evalúa a falso queda en espera.

La exclusión permite hacer un uso competitivo de un recurso, mientras que la condición permite hacer un uso cooperativo en una determinada operación.

Para ilustrar el uso de una región crítica, veamos un ejemplo de un programa que utiliza un buffer al que pueden acceder varios procesos (con sintaxis tipo Ada):

```
program buffer_example
    type buffer_t is record
        slots: array(1..N) of character;
        size: integer range 0..N;
        head, tail: integer range 1..N;
    end record;

    buffer: buffer_t;
    resource buf: buffer;

    process producer;
        ...
        loop
            region buf when buffer.size < N do
                -- place char in buffer
            end region
            ...
        end loop;
    end

    process consumer;
        ...
```

```
            loop
                region buf when buffer.size > 0 do
                    -- take char in buffer
                end region
                ...
            end loop;
        end
    end
```

Un problema de rendimiento que puede aparecer en las regiones críticas es que las condiciones de acceso de todos los procesos que están suspendidos se ha de evaluar cada vez que un proceso sale de la región crítica. Si la condición de acceso se evaluara a falso debería seguir bloqueado.

3.3.1.3 Monitores

El principal problema de las regiones críticas es que su utilización puede quedar desperdigada a lo largo del código de los distintos procesos que forman un programa. Los monitores pretenden reducir este problema ofreciendo un control de la región crítica más estructurado.

La idea es escribir las secciones de código en las que se accede a las regiones críticas encapsuladas juntas en un único módulo llamado **monitor**. Al igual que en un módulo, todas las variables que deben ser accedidas bajo exclusión mutua quedan ocultas. Además, como en las regiones críticas, todas las llamadas a los procedimientos garantizan que se ejecutan en exclusión mutua.

Como ejemplo, un monitor para manejar un buffer de enteros podría tener la forma:

```
monitor buffer;
    export append, take;
    var (* declaracion de las variables utilizadas *)

    procedure append (I: integer);
        ...
    end;

    procedure take (I: integer);
        ...
    end;

begin
    (* Inicializacion de las variables del monitor *)
end
```

Aunque los monitores ofrecen exclusión mutua, aun se hace necesarias las condiciones de acceso dentro del monitor. Para esto se podrían utilizar semáforos, pero otra posibilidad es la utilización de las **variables condición**, disponibles en algunas implementaciones de los monitores.

Las variables condición se utilizan con dos operaciones, similares a las de los semáforos, que son `wait` y `signal`.

Cuando un proceso realiza la operación `wait` sobre una variable condición, queda bloqueado en una cola de espera. Cuando se bloquea el proceso libera el monitor permitiendo que accedan a él otros procesos.

Cuando un proceso realiza la operación `signal` liberara un proceso bloqueado (si no lo hubiera esta operación no tendría ningún efecto).

La semántica de las operaciones `wait` y `signal` no es completa, tal como se ha dicho hasta ahora. Pues podría ocurrir que más de un proceso estuvieran activos simultáneamente dentro del monitor. Estro podría ocurrir a continuación de una operación `signal` en la que se despierta un proceso bloqueado. Para evitar esta acción indeseable, se debe modificar la semántica de la operación `signal`. En distintos lenguajes aparecen cuatro enfoques diferentes:

1) Una operación `signal` sólo se permite si es la última acción que realiza un proceso antes de salir del monitor

2) Una operación `signal` tiene como efecto secundario la ejecución de una instrucción `return` del procedimiento del monitor, es decir, saldría del procedimiento.

3) Una operación `signal` que desbloquea otro proceso tiene como efecto secundario el bloquear al proceso llamante. La ejecución del proceso se reanudará solo cuando se libere el monitor.

4) Una operación `signal` que desbloquea otro proceso no lo desbloqueará hasta que salga del procedimiento actual y libere el monitor

Los monitores suelen presentar problemas si se realizan llamadas anidadas, es decir, llamar a un procedimiento de un monitor desde dentro de un procedimiento del mismo monitor. En este caso habrá que prestar atención a la implementación que hace el lenguaje de los monitores.

Algunos lenguajes que implementan los monitores son Modula-2, Pascal concurrente y Mesa. Un comportamiento equivalente a los monitores se puede conseguir con los mutex y las variables condición del estándar POSIX.

3.3.1.4 Objetos protegidos

Los objetos protegidos son unas estructuras más evolucionadas que los monitores que pretenden eliminar las pegas que tiene, sobre todo en lo referente al uso de las variables condición. El lenguaje Ada es el único que ofrece este mecanismo de sincronización.

Un objeto protegido en Ada encapsula unos datos y permite el acceso a ellos solamente a través de subprogramas protegidos o entradas protegidas. El lenguaje garantiza que estos subprogramas y entradas se ejecutan de forma que los datos son modificados bajo exclusión mutua.

Las condiciones de sincronización se ofrecen por medio de expresiones booleanas en las entradas (estas son las condiciones de acceso que en Ada se llaman **barreras**) las cuales deben evaluarse a verdadero antes que se le permite a un proceso el acceso a una entrada. En consecuencia, los objetos protegidos realizan la función de los monitores y de las regiones críticas.

En Ada una unidad protegida se puede declarar como una única instancia o como un tipo de datos para declarar múltiples instancias. Tiene una especificación y un cuerpo (similar a la definición de las tareas). Su especificación puede contener funciones, procedimientos y entradas.

El siguiente ejemplo que implementa un semáforo con contador ilustra como los objetos protegidos ofrecen exclusión mutua para el acceso a los datos, en este caso el contador del semáforo.

```
protected type Semaforo is
  function get_cont return integer;
  procedure set_cont(val: integer);
  entry wait;
  procedure signal;
private
  Counter : integer := 1;
end Semaforo;

protected body Semaforo is
  function get_cont return integer is
  begin
    return Counter;
  end get_cont;

  procedure set_cont(val: integer) is
  begin
    Counter := val;
  end set_cont;

  entry wait when Counter > 0 is
  begin
    Counter := Counter - 1;
  end wait;

  procedure signal is
  begin
    counter := counter + 1;
  end signal;
end Semaforo;
```

Los procedimientos y las entradas del objeto protegido ofrecen exclusión mutua de acceso de lectura / escritura a los datos encapsulados. En nuestro caso, llamadas concurrentes a `set_cont`, `wait` o `signal` se ejecutarán en exclusión mutua, es decir, sólo una se ejecutará a la vez.

Las entradas, aparte de la exclusión mutua en lectura / escritura tienen una expresión booleana (la barrera) que se evalúa en la entrada, ya dentro de la zona de exclusión. Si la expresión se evalúa a verdadero se continúa de la misma forma que en un procedimiento, pero si se evalúa a falso se libera la exclusión mutua de lectura / escritura y sé bloque el proceso. Esto se repetirá hasta que la barrera se evalúe a verdadero.

Las funciones ofrecen acceso concurrente de solo lectura a los datos encapsulados. En el ejemplo anterior se podrán ejecutar simultáneamente múltiples llamadas a la función `get_cont`, pero una llamada a la función `get_cont` no se podrá ejecutar concurrentemente con una llamada a un procedimiento o a una entrada.

Las situaciones que provocan que se repita la evaluación de una barrera en los procesos que han quedado suspendidos por haberse evaluado anteriormente a falso son:

(1) Un proceso llama a una entrada de un objeto protegido y la barrera asociada hace referencia a una variable o atributo que puede haber cambiado desde que la barrera se evaluó por última vez.

(2) Una tarea abandona un procedimiento protegido o barrera protegida y hay tareas en la cola de espera de las entradas en las que se hace referencia a variables o atributos que pueden haber cambiado desde que la barrera se evaluó por última vez.

Las barreras no se evalúan como resultado de las llamadas a las funciones protegidas. Hay que tener en cuenta que en las funciones protegida sólo se tiene acceso de lectura a las variables encapsuladas y, por tanto, no se puede modificar el resultado de ninguna barrera.

Del mismo modo que los semáforos, utilizando los objetos protegidos se pueden construir distintos mecanismos de sincronización. Algunos de ellos son:

- **Señales transitorias y persistentes**: Son el equivalente al suspend / resume y wait / wakeup.
- **Eventos**: Los eventos son banderas que pueden tener el valor subido o bajado. Las tareas pueden quedar esperando a cualquiera de estos estados
- **Buffers**: El mecanismo de sincronización va ligada a un paso de información. Existen las operaciones 'Put(Mensaje)' y 'Get(Mensaje)'. La tarea que envía se bloquea si el buffer está lleno y la que recibe se bloquea si la cola está vacía.
- **Pizarras**: Son similares a los buffers salvo que la lectura no borra el mensaje, con lo que puede ser leído varias veces por distintos procesos. Existe una operación especial 'Clear' para borrar el mensaje.
- **Boradcast**: Se envía un mensaje que es leído por todos los procesos que están bloqueados en lectura.
- **Multicast**: Los mensajes enviados son leídos por un grupo de procesos que se registran en el grupo.
- **Barreras**: Bloquea un grupo de tareas esta que estén todas bloqueadas. Tras llegar la última se desbloquean todas a la vez.

Veamos un ejemplo: las señales transitorias:

```
package senyales is
  protected type senyal is
     procedure enviar;
     entry esperar;
     procedure enviar_todos;
  private
     senyal_activada: Boolean := False;
     liberar_todos: Boolean;
  end senyal;
end senyales;

package body senyales is
  protected body senyal is
     procedure enviar is
     begin
        if esperar'Count > 0 then
           senyal_activada := True;
           liberar_todos := False;
```

```
            end if
         end enviar;
         entry esperar when senyal_activada is
            if esperar'Count = 0 or not liberar_todos then
               senyal_activada := False;
            end if
         end esperar;
         procedure enviar_todos is
            if esperar'Count > 0 then
               senyal_activada := True;
               liberar_todos := True;
            end if
         end enviar_todos;
      end senyal;
   end senyales;
```

3.3.2 Comunicación basada en paso de mensajes

La comunicación por paso de mensajes es el mecanismo alternativo a la comunicación por variables compartidas. Esta planteamiento se caracteriza por ser una construcción que incluye tanto mecanismos de comunicación como de sincronización.

Existen gran variedad tipos de comunicación por paso de mensajes. Cada lenguaje proporciona uno u otro. Esta variedad está reflejada en tres elementos que intervienen en el paso de mensajes, que son:

- El modelo de sincronización
- El método para indicar el destino
- La estructura de los mensajes

3.3.2.1 Sincronización entre procesos

En todos los sistemas basados en mensajes, existe una sincronización implícita que hace que un proceso no pueda recibir un mensaje antes que éste haya sido enviado. Aunque esto aquí parece obvio, es similar al paso de información por variables compartidas; en este caso, el proceso receptor no debe leer una variable antes que el emisor haya escrito en ella. Si un proceso realiza una lectura incondicional de un mensaje antes que haya un mensaje disponible, debe quedar suspendido hasta que llegue el mensaje.

Atendiendo al modelo de sincronización, podemos hacer la siguiente clasificación de la comunicación por paso de mensajes:

- **Asíncrona** (o **sin espera**). El proceso que envía procesa el envío inmediatamente, de forma independiente a que el mensaje sea leído en el momento del envío o no.
- **Síncrono**. El proceso que envía no finaliza el envío hasta que el mensaje es leído por el receptor.
- **Invocación remota**. El proceso que envía no finaliza el envío hasta que el proceso receptor invoca una operación de respuesta después de haber leído el mensaje.

En el caso de comunicación síncrona, como el proceso que envía y el que espera reanudan su ejecución juntos en el momento del paso del mensaje, al instante de la sincronización se le llama **cita** (`rendezvous`). De forma análoga, la sincronización en el caso de invocación remota se llama cita extendida.

Existe una relación entre estos modelos de sincronización. De hecho, con el modelo asíncrono se puede construir el modelo síncrono si el proceso que envía queda a continuación esperando un mensaje de confirmación:

```
        P1                          P2
envio_asinc(mensaje)        espera(mensaje)
espera(confirmacion)        envio_asinc(confirmacion)
```

De forma análoga, el modelo con invocación remota se puede construir combinando dos envíos síncronos:

```
        P1                          P2
envio_sinc(mensaje)         espera(mensaje)
espera(respuesta)                   ...
                                    construccion de la respuesta
                                    ...
                            envio_sinc(respuesta)
```

Como el modelo asíncrono se puede utilizar para construir el resto de modelos, se puede decir que es el modelo de sincronización más general, y es el que suelen ofrecer los lenguajes y los sistemas operativos. No obstante, este modelo tiene algunos inconvenientes:

- En principio se necesitan buffers de tamaño infinito para guardar todos los mensajes que se pueden enviar (por ejemplo porque el receptor ha muerto, o por que lee a menor velocidad que se producen los mensajes)

- Como en el envío asíncrono no se sabe si el mensaje es leído, muchos envíos se programan esperando a continuación un mensaje de reconocimiento (se acaba utilizando el modelo síncrono)

- Se utilizan más envíos en el modelo asíncrono (cuando se quiere utilizar el síncrono o el de invocación remota), por tanto, los programas se hacen más complejos.

- Resulta más difícil verificar la exactitud de los sistemas y depurar los errores, sobre todo si los sistemas son complejos (puede desaparecer la relación en el tiempo de la causa - efecto).

También hay que tener en cuenta que si se desea comunicación asíncrona en el modelo síncrono, no es complicado construir una comunicación utilizando buffers. Aunque hay que tener en cuenta que la utilización excesiva de buffers puede llegar a disminuir el rendimiento de una aplicación (el envío de un mensaje supone dos copias, mientras que el envío asíncrono se puede realizar con una sola copia, y con invocación remota se puede acceder al mensaje origen por referencia, sin copiar el mensaje), además del problema que supone el correcto dimensionado de los buffers y el consiguiente gasto de memoria que puede no ser despreciable.

3.3.2.2 Indicación del destino

La indicación del destino del mensaje introduce dos tipos de envío, el envío **directo** y el **indirecto**, y el concepto de **simetría**.

En el envío del tipo **directo**, el proceso que envía indica directamente el proceso receptor.

```
enviar <mensaje> a <nombre-de-proceso>
```

En el envío de tipo **indirecto**, el proceso que envía hace referencia a una entidad intermedia (conocida por diversos nombres según el sistema: canal, buzón, enlace, puerto, tubería):

```
enviar <mensaje> a <buzon>
```

Hay que tener en cuenta que en la utilización de un buzón, el envío puede seguir siendo síncrono.

El envío directo tiene la ventaja que es más simple, mientras que el envío indirecto ayuda a la descomposición del software; se aíslan los procesos de envío y lectura por medio de una entidad intermedia que proporciona un interfaz.

El tipo de envío es **simétrico** si tanto el proceso que envía como el que recibe hacen referencia al otro:

```
enviar <mensaje> a <nombre-de-proceso>
recibir <mensaje> de <nombre-de-proceso>

enviar <mensaje> a <buzon>
recibir <mensaje> de <buzon>
```

El tipo de envío será **asimétrico** si el receptor no hace referencia a la fuente del mensaje, bien sea un proceso (envío directo) o un buzón (envío indirecto). En este caso se estarán aceptando mensajes que provengan desde cualquier proceso o que estén en cualquier buzón:

```
recibir <mensaje>
```

El tipo asimétrico es adecuado para la construcción del modelo de comunicación cliente - servidor, en el cual un proceso “servidor” ofrece un determinado servicio como respuesta a mensajes que provengan de cualquier proceso “cliente”.

Si la indicación del destino es de **tipo indirecta**, existen algunos detalles que convienen considerar. La entidad intermedia puede tener:

- Una estructura de muchos a uno. Esto es, muchos procesos pueden escribir pero sólo uno puede leer. Esta estructura se adapta bien al modelo de comunicación cliente - servidor.
- Una estructura de muchos a muchos. Al igual que pueden escribir muchos procesos, también pueden leer muchos.
- Estructura de uno a uno. Solo existe un proceso que escribe y es otro concreto el que lee.
- Estructura de uno a muchos. Esta estructura es útil cuando un proceso desea enviar una petición a un grupo de procesos, y no es relevante el proceso que sirve la petición.

En las estructuras en que son muchos los que leen caben dos posibilidades:

1) Que el mensaje lo lea sólo uno de los procesos que leen.

2) Que todos los procesos que leen reciban una copia del mensaje.

La primera se puede utilizar para el modelo cliente - servidor en el que existen varios servidores idénticos y, por tanto, se podrán servir varias peticiones simultáneamente.

La segunda se puede utilizar para implementar mecanismos de tolerancia a fallos en la que varios servidores procesan el mismo mensaje y, en caso de fallar uno, se podrá utilizar el proceso de otro servidor sin que se pierda la petición.

3.3.2.3 Estructura de mensajes

Un lenguaje debería permitir transmitir por medio de un mensaje cualquier objeto de datos de cualquier tipo definido por el usuario o predefinido. El mantener este caso ideal es difícil, más aún si en la representación del objeto se utilizan punteros que construyen estructuras dinámicas. Debido a esta dificultad, algunos lenguajes restringen el envío de mensajes a tipos sin estructura, objetos de tamaño fijo o tipos predefinidos. Algunos lenguajes más modernos han eliminado esta restricción; no obstante, los sistemas operativos modernos que no saben de los tipos definidos en el lenguaje necesitan que los objetos sean convertidos a cadenas de bytes antes de ser enviados.

3.3.2.4 Espera selectiva

En todos los tipos de envíos de mensajes que se han visto, el proceso receptor debe esperar la llegada del mensaje hasta que un proceso específico o buzón lo haga disponible.

Esto es, en general, demasiado restrictivo. Un proceso receptor que hace de servidor puede, en un momento dado, desear quedar a la espera sólo de un determinado tipo de mensajes, y dejar sin tratar los mensajes de otros tipos.

Este tipo de lectura se puede refinar aun más. Puede que en un momento determinado interese leer unos tipos de mensajes y en otro momento en el que ha cambiado el estado del servidor, interese leer otros grupo de tipos de mensajes. Lo normal es admitir un tipo de mensajes que tratará normalmente el servidor y, en algunos casos excepcionales, se permiten o evita el leer un número reducido de tipos de mensajes.

Para facilitar la construcción de este tipo de estructuras, se permite la lectura de un tipo determinado de mensaje solamente si se cumple una condición de control. La condición de control es una expresión booleana que hará que se lee un determinado tipo de mensaje solamente si en el momento de la lectura se evalúa a verdadero. Este tipo de estructura para realizar la lectura de un mensaje se llama **espera selectiva**.

El lenguaje Ada, por medio de la construcción `select`, permite la lectura de mensajes con espera selectiva.

En UNIX existe una lectura selectiva en las colas de mensajes más restrictiva indicando el tipo de mensaje a leer (se leen todos los tipos mayores).

También existe en las versiones modernas de UNIX la función `select`, que permite leer de varios descriptores de ficheros simultáneamente.

3.3.2.5 Llamada a procedimientos remotos. RPC

El mecanismo de llamada a procedimientos remotos se construye por medio del paso de mensajes. Es una forma muy usual de paso de información entre procesos en entornos distribuidos.

La llamada a procedimientos remotos es una extensión de llamada a un procedimiento dentro de un proceso, a realizar la llamada ejecutándose desde otro proceso o, incluso, en otro procesador.

La ejecución del procedimiento remoto puede incluir comunicación y sincronización con el proceso que reside en la máquina remota, bien por la llamada a métodos de variables compartidas (por ejemplo, monitores) o por el paso de mensajes (por ejemplo, en una cita).

La ejecución de procedimientos remotos se puede implementar bien utilizando el tipo de envío con invocación remota. En el mensaje que se envía se debe incluir, con una determinada estructura, un identificador de la función que se quiere llamar y los parámetros que se pasan en la llamada con sus tipos asociados. En la respuesta debe ir el resultado, si se trata de una función, o los parámetros que se pasan por referencia (en los que el procedimiento puede devolver información).

3.4 Fiabilidad y tolerancia a fallos

Las exigencias de fiabilidad y la seguridad son mucho más rigurosas en los sistemas empotrados que en otros sistemas informáticos. Por ejemplo, si se produce un fallo en una aplicación que calcula la solución de un problema científico, es admisible abortar el programa pues lo único que se habrá perdido es tiempo de cálculo. Sin embargo, en un sistema de tiempo real esto no suele ser una solución aceptable. Un sistema de control por ordenador, por ejemplo, responsable de mantener la temperatura en un horno industrial, no es aceptable detenerlo en cuanto se produce un fallo, pues se podrían alcanzar temperaturas fuera de control que podrían ocasionar la destrucción de la factoría y hasta poner en peligro vidas humanas. Lo mismo ocurre con un sistema que controle un proceso en un reactor nuclear o el sistema de navegación en un avión.

Hoy en día, cada vez más se sustituyen funciones realizadas previamente por operadores humanos utilizando sistemas de control de tiempo real por ordenador por lo que el tema de la fiabilidad y la seguridad son cada vez más importantes.

Hecht and Hecht (1986) efectuaron estudios de software en grandes sistemas y llegaron a la conclusión que, de promedio, por cada millón de líneas de código, se introducían en el software unos 20.000 fallos. Normalmente, el 90% de estos fallos se detectaban y corregían durante las pruebas. Posteriormente, unos 200 fallos surgían durante el primer año de vida del sistema, quedando unos 1800 fallos por detectar. El software desarrollado para el mantenimiento introduce alrededor de 200 nuevos fallos que son detectados y 200 más sin detectar.

La historia está llena de ejemplos de los efectos catastróficos producidos por fallos de software no detectados. Un ejemplo que todos conocemos es el telescopio espacial Huble, en el que un fallo en el programa encargado de pulido de las lentes introdujo una aberración importante en un determinado espectro.

Algunas empresas renuncian a un control automático de sus sistemas vitales por el hecho de que estos sistemas no pueden fallar (por ejemplo la Central Nuclear de Cofrentes).

En general, existen cuatro fuentes de errores que pueden hacer que un sistema empotrado falle:

- Especificaciones inadecuadas, incorrectas o incompletas.
- Errores de diseño en los componentes de software.
- Errores introducidos por fallos en uno o más componentes del procesador o del sistema empotrado.
- Fallos introducidos por la interferencia, transitoria o permanente, en el subsistema que soporta las comunicaciones.

A la programación de los sistemas en tiempo real les afecta sólo los tres últimos supuestos. En el primer caso los elementos que inducen a errores son impredecibles e intratables, aunque los producidos por el procesador o por la red de comunicaciones son, en algún sentido, predecibles. Por consiguiente, uno de los principales requerimientos de cualquier lenguaje de programación para tiempo real es que facilite la construcción de sistemas altamente fiables.

La fiabilidad no exige que el sistema continúe funcionando tras un fallo al 100% como si no hubiera ocurrido. Pero si que es necesario que el fallo sea detectado y el sistema reaccione con una respuesta aceptable y controlada, aunque sea en un modo degradado.

3.4.1 Fiabilidad, averías, errores y fallos

Conviene que definamos de forma más precisa la nomenclatura que se utiliza para hablar de la tolerancia a fallos.

Primero vamos a definir la **fiabilidad** de un sistema como:

> *"Una medida de los sucesos en los cuales el sistema se ajusta al comportamiento definido en unas determinadas especificaciones que se dan como válidas."*

Idealmente, estas especificaciones deben ser completas, consistentes, compresibles y no ambiguas. Hay que tener en cuenta también que el **comportamiento temporal** del sistema es una parte importante de las especificaciones en los sistemas de tiempo real, de ahí que la tolerancia a fallos juegue un papel muy importante en los sistemas de tiempo real críticos.

La definición anterior de fiabilidad se puede utilizar para definir una **avería** de un sistema:

> *"Una avería ocurre cuando el comportamiento del sistema se desvía del indicado en las especificaciones."*

Es evidente pues, con estas definiciones, que un sistema será altamente fiable cuando en el se produzcan un número muy reducido de averías.

Las dos definiciones anteriores hacen referencias al comportamiento externo de un sistema. Pero el hecho de que en un momento determinado surja al exterior una avería es debido a la existencia de un

problema interno. Estos problemas internos se denominan **errores**, y son las consecuencias directas o indirectas de los **fallos**.

Con esto, podemos formular las siguientes definiciones:

> *"Un error es la parte del estado del sistema que lo diferencia de un estado válido."*

> *"Un fallo es un defecto o imperfección, físico o de diseño, en algún elemento del sistema"*

Desde el punto de vista del estado, un sistema se puede considerar como un conjunto de estados externos y un conjunto de estados internos. Un estado interno que no esté especificado se puede considerar como un error.

De la misma forma, un sistema está formado por un conjunto de componentes, cada uno caracterizado por sus estados internos y externos. El conjunto de estados de estos componentes constituye el estado interno del sistema. Si se produce un error (estado interno no previsto) en uno de los componentes, se puede traducir en una avería que posiblemente ocasione un error en otro componente relacionado, y así hasta que, posiblemente, se produzca una avería en el sistema.

Los errores en los componentes que no se traducen en averías del sistema se dice que quedan enmascarados. Puede ocurrir que un error quede enmascarado en unas determinadas circunstancias pero puede ocurrir que en otras se traduzca en una avería visible.

Un ejemplo podría ser el efecto de la utilización incorrecta de un índice en una tabla de una variable local que altere el espacio de direcciones de las variables locales del procedimiento llamante.

En este caso el fallo es la utilización de un índice incorrecto, y el error la variable que se altera al utilizar este índice. Según que la llamada que produce el error se haga mientras se utiliza la variable que se altera, o se utilice antes o después, el error afectará o no al uso de la variable.

Existe una correspondencia entre fallo y error, consecuencia de este fallo. Por ello, en ocasiones estos conceptos se utilizan indistintamente.

Con estas definiciones queda abierto el mecanismo para tolerar los fallos: Los fallos son detectados por los errores que producen. Las técnicas de tolerancia a fallos no corrigen estos, sino que pretenden evitar que sus errores no se traduzcan en averías.

3.4.2 Técnicas de programación para la tolerancia a fallos

Todas las técnicas de tolerancia a fallos se basan de alguna forma en la **redundancia**, es decir, en la introducción de componentes extras en el sistema que permitan la detección de los fallos y su recuperación. Estos componentes son redundantes en el sentido que no son necesarios en un funcionamiento normal. La meta que persiguen las técnicas de tolerancia a fallos es conseguir la máxima fiabilidad con la mínima redundancia.

La redundancia se puede introducir tanto en el hardware como en el software. Nosotros haremos mayor hincapié en la redundancia del software.

En cuanto al hardware podemos distinguir (Anderson and Lee, 1990) entre redundancia estática y dinámica. En la **redundancia estática** los componentes redundantes se utilizan dentro de un sistema para ocultar el efecto de los errores. Un ejemplo es el Triple Modular Redundancy (TMR), que consiste en tres componentes idénticos y un circuito de votación que compara la salida de todos los componentes y, si una difiere de las otras dos, la enmascara. La generalización de esta técnica es la N Modular Redundancy (NMR)

La **redundancia dinámica** es la ofrecida por un componente la cual le permite indicar que su salida es incorrecta. Ofrece, por tanto una facilidad de detección del error en lugar de su enmascaramiento. En este caso, la recuperación se debe realizar utilizando otro componente. Un ejemplo de la redundancia dinámica es el bit de paridad en las memorias RAM.

Para la tolerancia a fallos en errores del software existen dos aproximaciones equivalentes. La primera se denomina programación con N-versiónes. La segunda se basa en la detección de errores y su recuperación; esta es similar a la redundancia dinámica en el sentido que la recuperación se realiza en un paso posterior a la detección.

La programación N-versiónes se define como la generación independiente de N ($N \geq 2$) programas funcionalmente equivalentes a partir de las mismas especificaciones. Estas técnicas tienen la dificultad de que la ejecución de cada versión se debe sincronizar y debe existir un mecanismo de votación que permita decidir cual es el resultado correcto en caso de divergencia.

Debido a las dificultades que entraña la ejecución de varias réplicas nosotros nos centraremos en las técnicas de redundancia dinámica.

En la redundancia dinámica existen dos alternativas una vez se ha detectado el fallo:

- Hacia atrás: Vuelta a un estado correcto previamente guardado.
- Hacia delante: Realizando una serie de acciones para corregir el error.

3.4.2.1 Bloques de recuperación

Los bloques de recuperación son una forma de representar la redundancia dinámica en los lenguajes. Son bloques en el sentido usual de los lenguajes de programación, excepto que la entrada se corresponde a un punto de recuperación y en la salida se realiza un test de aceptación.

Al entrar el programa en un bloque se ejecuta un módulo primario. Si al final falla el test de aceptación, el programa se restaura al punto de recuperación correspondiente y se pasa a ejecutar un módulo alternativo. Si con el módulo alternativo también falla el test de aceptación se ejecuta otro, y así consecutivamente hasta el número de módulos alternativos que haya previstos en el bloque. Si fallan todos los módulos se determina que el bloque ha fallado y la recuperación se debe realizar en un nivel superior.

Una sintaxis posible para los bloques de recuperación en un lenguaje de programación podría ser:

```
ensure <test de aceptación>
by
     <modulo primario>
else by
     <modulo alternativo>
else by
     <modulo alternativo>
     ...
else by
     <modulo alternativo>
else error
```

Los bloques de recuperación pueden anidarse, de forma que si en un bloque interno falla el test de aceptación en todos sus módulos, se pasaría a ejecutar un módulo alternativo en el nivel superior.

Un ejemplo de utilización de los bloques de recuperación podría ser un programa que resolviera ecuaciones utilizando varios métodos numéricos de resolución. En este programa, el test de aceptación tendría que determinar si la solución encontrada utilizando un determinado módulo, o método de resolución, es aceptable o hay que probar con otro.

```
ensure resultado_aceptable
by
     metodo_iterativo_1
else by
     metodo_iterativo_2
else error
```

3.4.2.2 Redundancia dinámica y excepciones

Vamos a ver en este apartado como se puede utilizar el tratamiento de excepciones que ofrecen muchos lenguajes para implementar tolerancia a fallos del software basada en la redundancia dinámica.

Una excepción la podemos definir como la aparición de un error, entendiendo como error algún suceso que puede desencadenar una avería.

El hecho de producirse la condición que lanza una excepción se denomina, según el lenguaje empleado, **raise** (levantar), **signal** (señalizar) o **throw** (lanzar). La respuesta que se produce cuando ocurre la excepción se denomina **handler** (manejador) o **catch** (captura) de la excepción.

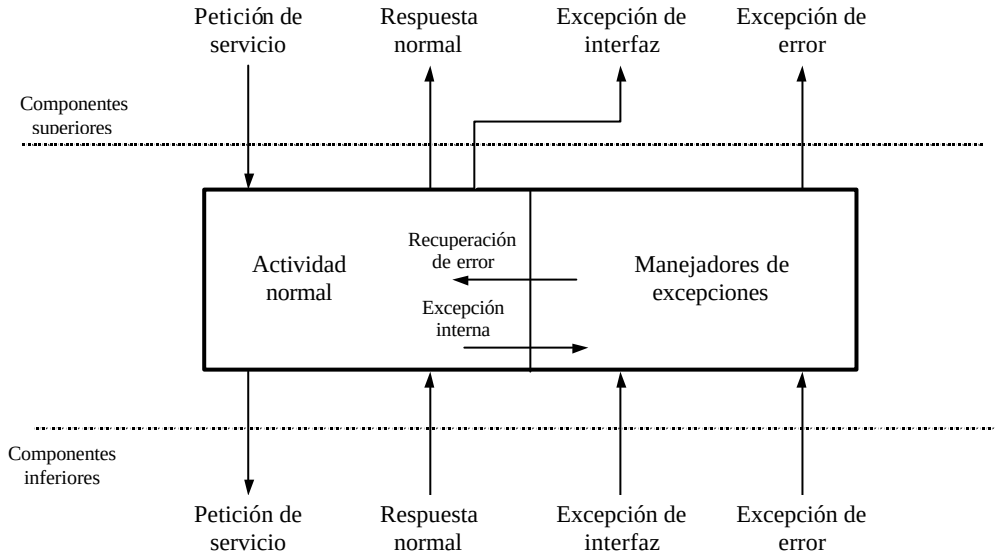
El manejo de excepciones funciona como un mecanismo de recuperación de errores hacia adelante (*forware error recovery*), pues cuando se produce una excepción el sistema no regresa a un estado previo sino que se da paso a un procedimiento de recuperación del error. No obstante, el manejo de excepciones se puede utilizar para implementar mecanismos de recuperación de errores hacia atrás (*backward error recovery*).

La recuperación hacia atrás se puede realizar con un bucle, de forma que a la entrada del bucle el sistema debe estar en una situación estable. Si en un paso del bucle el sistema falla, se producirá una excepción (recuperación hacia adelante) cuyo manejador debe encargarse de volver al sistema a una situación estable y, al finalizar, se realizará otro paso del bucle con un procedimiento alternativo. El bucle terminará cuando un procedimiento finalice sin error o cuando ya no queden más procedimientos alternativos.

3.4.2.3 Componente ideal tolerante a fallos

Con el mecanismo de excepciones podríamos describir el componente ideal para construir sistemas tolerantes a fallos. Este componente aceptaría peticiones de servicios y, si fuese necesario, podría invocar servicios de otros componentes antes de devolver una respuesta.

Se podría producir una respuesta normal o una respuesta de excepción. En este componente ideal se podrían producir dos tipos de respuesta de error: por un lado un error debido a una petición de servicio incorrecta, denominada **excepción de interfaz**, y por otro debido a un error en el funcionamiento interno del componente o de los componentes a los que se llaman que no es posible tolerarlo internamente por los mecanismos propios de recuperación de fallos (hacia adelante o hacia atrás). La respuesta en este último caso se denomina **excepción de error**.



Componente ideal

Antes de responder con una excepción el componente debe regresar a una situación estable para poder seguir sirviendo peticiones.

3.5 Manejo de excepciones

Existen una serie de requerimientos generales que deben cumplir el manejo de excepciones en un lenguaje de programación. Estas son:

1) Como el resto de características del lenguaje, el manejo de excepciones debe ser fácil de entender y de utilizar.

2) El código para describir la utilización de las excepciones no debe dificultar la comprensión del programa en su funcionamiento normal sin errores. Un mecanismo que intercale código para verificar la existencia de un error y lanzar su manejador dificulta la compresión del código y su mantenimiento.

3) El mecanismo debe estar diseñado de forma que la sobrecarga que conlleva durante la ejecución solo afecte cuando se captura una excepción.

4) Debe permitir un tratamiento uniforme de las excepciones detectadas por el entorno y por el programa.

5) El mecanismo de excepciones debe permitir la programación de acciones de recuperación de fallos.

3.5.1 Representación de las excepciones

Los lenguajes de programación modernos suelen incluir en el propio lenguaje facilidades que ofrecen mecanismos estructurados para manejar excepciones. La forma exacta de estas facilidades depende del lenguaje.

Podemos clasificar las excepciones en dos tipos dependiendo del retraso admitido entre que se produce el error y se lanza excepción: síncronas o asíncronas. Una **excepción síncrona** se llama inmediatamente después de una sección del código que ha intentado realizar una operación incorrecta. Por el contrario, una **excepción asíncrona** es lanzada algún tiempo después de la operación que ha causado el error.

A su vez, las excepciones pueden ser lanzadas por el **propio proceso** que las trata o por otro **proceso distinto**.

Dependiendo de la forma de lanzar una excepción y de sí esta es detectada por el entorno o por la aplicación, existen cuatro clases de excepciones:

- Síncrona detectada por el entorno. Algunos ejemplos son la violación de los límites en los índices de un array o una división por cero.

- Síncrona detectada por la aplicación. Es el caso de las 'assert' típicos en los que se detecta un error en alguna condición crítica de un programa.

- Asíncrona detectada por el entorno. Por ejemplo, la detección del fallo de la alimentación o de exceso de temperatura.

- Asíncrona detectada por la aplicación. Es el caso de los procesos que detectan una condición de error producida por el incumplimiento de las restricciones temporales de otro proceso.

Las excepciones asíncronas a menudo se denominan **señales** y se suelen utilizar en el contexto de la programación concurrente.

Las excepciones asíncronas presentan problemas para la recuperación de fallos pues está desconectado el fallo que provoca la excepción con su tratamiento. Nosotros vamos a centrarnos por ahora en las excepciones síncronas.

En las excepciones síncronas existen varios modelos para su declaración. Por ejemplo, se pueden ver de las siguientes formas:

- Un nombre de etiqueta que tiene que ser declarado de forma explícita. Este es el modelo que toma Ada.

– Un objeto de un tipo particular que puede o no necesitar ser declarado previamente. Es el modelo que utiliza el C++. En este caso el objeto puede pasar información al manejador de la excepción.

3.5.2 Dominio de un manejador de excepción

Dentro de un mismo programa pueden haber varios manejadores asociados a una misma excepción. Esto es de esperar pues es normal que, por ejemplo, ante un valor fuera de rango, se deban hacer cosas distintas en diferentes partes del programa.

A cada manejador se le asocia un dominio el cual especifica la zona de ejecución durante la cual, si ocurre su excepción asociada, se activará. La precisión con la que se especifique el dominio de un manejador determina la precisión con la que se puede localizar el error. En un lenguaje estructurado basado en bloques, como es el Ada, el dominio normalmente es un bloque.

Un ejemplo sencillo sería:

```
procedure resultado_aceptable is
declare
     subtype temperatura is integer range 0 .. 100;
begin
     --  lectura  del  sensor  de  temperatura  y  calculo  del
valor
exception
     -- manejador de la excepción de fuera de rango
end
```

Dentro del manejador de excepción, se puede distinguir entre varios tipos de excepciones utilizando sentencias del tipo `when` *`etiqueta`* `=>`.

En algunos lenguajes, como es el caso del C++, las excepciones no se pueden utilizar directamente en todos los bloques, sino que hay que indicar expresamente cuando se quieren utilizar y el dominio al que afectan. En C++ los bloques que corresponden con el dominio de las excepciones se indican con `try` y los manejadores se colocan a continuación del bloque con una o más sentencias `catch`. El ejemplo anterior quedaría de la forma.

```
trye {
     // Lectura de la temperatura y calculo del valor
     // Control del rango y lanzamiento de la excepcion
     // usando throw
}
catch ( excepcio_fuera_de_rango ) {
     // Manejador de la excepcion
}
```

En C++ los tipos de excepciones que se da como parámetro a `catch` son clases de objetos. Al contrario que en Ada, cuando se produce una excepción el manejador recibe un objeto de esa clase que puede contener información adicional de la excepción.

3.5.3 Propagación de las excepciones

El concepto de propagación de las excepciones está muy ligado al concepto de dominio de las excepciones.

Ya hemos visto que cuando se produce una excepción en un bloque y existe un manejador para esta excepción, se pasa a ejecutar este manejador. La cuestión está en que ocurre cuando en el bloque que se produce la excepción no existe un manejador definido para ella.

Una primera aproximación es que el compilador de un error cuando se intenta llamar a una excepción que no está definida en su dominio.

Normalmente ocurre que una excepción es manejada en el contexto del subprograma que llama al bloque donde se produce la excepción. En este caso el compilador no puede comprobar que el manejador no existe, a no ser que se indique en el subprograma las excepciones que son manejadas desde el exterior, es decir, que no son capturadas localmente. En este sentido, el C++ permite definir en una función las excepciones que puede lanzar al exterior, es decir, las que no son capturadas localmente.

La segunda aproximación que se utiliza, consiste en buscar el manejador a través de la cadena de llamadas a procedimientos que ha dado lugar a la ejecución del subprograma donde se ha producido la excepción, comenzando por el más interno hasta el bloque donde comienza el programa. A esto se le llama propagación de excepciones. Tanto el lenguaje C++ como el Ada permite la propagación de excepciones.

Un problema que puede surgir en la propagación de excepciones es que al recorrer la cadena de llamadas no se encuentre un manejador para la excepción que se ha producido. Si en un determinado contexto es necesario que sean capturadas todas las excepciones esto puede resultar difícil de evitar, pues habría que prever manejadores para todas las excepciones que se pueden producir en los procedimientos llamados en ese contexto, directa o indirectamente. Para facilitar esta tarea muchos lenguajes ofrecen el manejador 'catch all' que permite capturar cualquier excepción que no disponga de un manejador específico. Este manejador es útil para evitar al programador la enumeración de múltiples excepciones en las que se ha de tomar una acción por defecto.

Cuando una excepción no encuentra su manejador, lo normal es que aborte el programa. Si se trabaja con más de un proceso lo normal es que aborte el proceso que ha causado la excepción. No está claro en que condiciones la excepción se debe propagar hacia el proceso padre.

Otra forma de considerar la propagación de excepciones es en términos si las posibles excepciones son asociadas a manejadores de forma estática (en tiempo de compilación) o de forma dinámica. Aunque la propagación dinámica es más flexible, esta implica mayor sobrecarga en el tiempo de proceso al tener que buscar el manejador. En la propagación estática al compilar se genera directamente la dirección del manejador que va a tratar cada excepción.

3.5.4 Modelos de reanudación y terminación

Una consideración muy importante en la forma de manejar excepciones es si el invocador de la excepción puede reanudar su ejecución una vez a finalizado el manejador. Si el invocador puede continuar, es posible que el manejador de la excepción pueda reparar el error que ha producido la excepción y continuar la ejecución como si no hubiera ocurrido nada. Esto se llama modelo de **reanudación** o de **notificación**.

Si no existe la posibilidad de reanudar la ejecución donde se había interrumpido, lo normal es que el control se devuelva al bloque inmediatamente superior al que tiene definido el manejador que ha tratado la excepción, es decir, el bloque donde es manejada la excepción se da por finalizado. A este se le llama modelo de **terminación** o de **escape**.

El problema que aparece en el modelo de reanudación es que, en la mayoría de excepciones, por ejemplo una división por cero, es imposible de reparar y, por tanto, no es posible reanudar la ejecución como si no hubiera ocurrido nada.

Aunque la implementación estricta del modelo de recuperación es complicada, una alternativa es volver a ejecutar el bloque asociado al manejador de la excepción. El lenguaje Eiffel ofrece esta posibilidad usando la directiva **replay**. En tal caso el manejador puede modificar un flag para indicar que se ha producido una excepción.

El modelo de recuperación presenta ventajas cuando la excepción es lanzada de forma asíncrona. En este caso la reanudación del proceso resulta sencilla pues no hay nada que reparar en el proceso que trata la excepción, pues no la ha causado directamente.

En el modelo de terminación, al terminar de ejecutarse el manejador, se da por concluido el bloque donde es manejada la excepción. Si el bloque corresponde al más externo de un subprograma, la ejecución continua en el subprograma que lo ha llamado. En el caso de que el bloque sea parte de un subprograma, la ejecución continua en la primera instrucción que sigue al bloque.

Los lenguajes Ada y C++ ofrecen ambos el modelo de terminación.

Se puede considerar también un modelo **híbrido**, En este, el propio manejador decide si la ejecución se ha de reanudar o hay que dar el bloque por finalizado.

Cuando un programa se ejecuta en una máquina con un determinado sistema operativo, algunas excepciones síncronas son controladas por el propio sistema operativo, como por ejemplo el acceso ha una zona de memoria protegida. Normalmente se toma la acción de abortar el proceso.

Algunos sistemas tienen la posibilidad de reanudar la ejecución del proceso realizando una acción de recuperación (es el caso de las señales definidas en POSIX, asociando manejadores a señales). En este caso, debe ser la implementación del lenguaje quien capture estas excepciones y realice las acciones oportunas para localizar el manejador definido para esta excepción y lanzarlo, definiendo el retorno del manejador según el modelo utilizado en el lenguaje.

3.6 Uso del tiempo

Un lenguaje para la programación de sistemas de tiempo real necesita facilidades para poder tener percepción del tiempo y poderlo utilizar en los algoritmos utilizados. Generalmente, los mecanismos para controlar el tiempo van incluidos en el modelo de concurrencia, pero de alguna forma han de ser accesibles desde el lenguaje, si este no incorpora la concurrencia.

La utilización de la noción del tiempo en los Sistemas de Tiempo Real puede aparecer en varios aspectos independientes:

- Representar los requerimientos temporales. Por ejemplo, especificar la frecuencia de ejecución y los límites temporales.

- Satisfacer los requerimientos temporales. Poder verificar que efectivamente los requerimientos temporales se cumplen.

- Interacción con el tiempo. Por ejemplo, acceso a un reloj para poder medir el paso del tiempo, retardar un proceso hasta un tiempo futuro y programar timeouts para poder detectar la no ocurrencia de algún evento.

3.6.1 La noción de tiempo

En la vida cotidiana estamos muy acostumbrados a manejar los conceptos de pasado, presente y futuro. Sin embargo la respuesta a la pregunta ¿que es el tiempo? sigue siendo una cuestión filosófica que no está del todo resuelta.

Como curiosidad, podemos considerar distintas teorías y nociones del tiempo:

- Reduccionismo. El tiempo establece un orden entre la ocurrencia de los sucesos. La medida del tiempo tiene un origen un final

- Platonismo. El tiempo es un continuo sin límites, no tiene ni origen ni final

- Relativismo. El tiempo no es absoluto y universal, sino que está ligado al sistema de referencia en que se observa. Introducen el concepto de orden causal (Un suceso A puede causar otro B solo si todos los posibles observadores ven A antes que B. Se postula que una señal que transmita información (usada para observar) no puede viajar a una velocidad superior a la de la luz.

A nosotros no nos interesa resolver filosóficamente esta pregunta sino utilizar un concepto matemático del tiempo que nos permita definir el concepto de sucesos simultáneos, que sirva para introducir un orden en la ocurrencia de los sucesos y para medir de la distancia 'temporal' entre sucesos no simultáneos.

Desde un punto de vista matemático, existen numerosas topologías para el tiempo. La más común entiende el paso del tiempo como una línea real. Las propiedades del tiempo con la operación '<' son, en este sentido:

- Lineal: $\forall\, x, y : x < y$ o $y < x$ o $x = y$

- Transitivo: $\forall\, x, y, z : (x < y$ y $y < z) \Rightarrow x < z$

- Irreflexivo: $\forall\, x$: no $(x < x)$

- Denso: $\forall\, x, y : x < y \Rightarrow \exists\, z : (x < z < y)$

Un sistema empotrado de tiempo real necesita coordinar su ejecución con el tiempo de su entorno. El término real se utiliza para marcar una distinción con el tiempo del ordenador. Para realizar la correspondencia entre el tiempo real y el del ordenador, la consideración de la teoría reduccionista o la platonista es irrelevante. Igualmente, los efectos relativista se pueden despreciar prácticamente en todos los casos.

En términos de la topología matemática, existen discusiones en cuanto a considerar el tiempo denso o discreto, pero como el ordenador maneja el tiempo de forma discreta, resulta más sencillo asumir un

modelo de tiempo discreto. De todas formas, si la aplicación lo requiere, se puede usar un modelo denso de tiempo o, lo que es más común, un modelo híbrido.

Existen varias medidas estándar del tiempo que son asumidas de forma general. La medida universal más difundida es la hora GMT (Greenwitch Meridian Time). Algunas emisiones de radio y el sistema GPS permiten obtener la hora UTC, basada en un reloj atómico de cesio, que corresponde a la hora GMT (UT0) con algunas correcciones debidas a la rotación de los polos (UT1) y variaciones en la velocidad de rotación de la tierra (UT2)

3.6.2 Acceso a un reloj

Si un programa va a interactuar, de alguna manera, en un marco temporal con respecto a su entorno, entonces debe poder acceder a algún método que le permita 'obtener la hora' o, al menos, alguna forma de medir el paso del tiempo. Esto se puede hacer de dos formas:

- Accediendo directamente al marco temporal del entorno.
- Utilizando un reloj interno que de una buena aproximación al paso del tiempo en el entorno.

El método más utilizado es el segundo. La forma más simple es utilizando una interrupción periódica que es utilizada como base de tiempo. Esto ofrece un modelo discreto de tiempo. El periodo de esta interrupción nos dará la máxima resolución de tiempo del sistema. Si se tiene acceso al contador interno del timer que provoca las interrupciones, y una cuenta de este contador es más rápida que la instrucción más breve de nuestro procesador (por ejemplo un tick de reloj de la CPU) se puede asumir que el modelo de tiempo así obtenido es denso.

3.6.2.1 El reloj en Ada

El lenguaje Ada ofrece un paquete predefinido llamado `Calendar` como una opción para tiempo real. Este implementa un tipo de dato abstracto para el tiempo: `Time`. Este paquete dispone de la función `Clock` para leer el tiempo, y varios subprogramas para convertir este tiempo en unidades más usadas en la vida real: `Years`, `Months`, `Days`, y `Seconds`. Los tres primeros están definidos como subtipos enteros. El `Seconds` está definido como un subtipo del tipo primitivo `Duration`.

El tipo `Duration` es un tipo real de como fija que permite realizar cálculos de tiempo. Tanto la precisión como el rango dependen de la implementación pero, al menos, su rango debe ser -86_400.0 .. 86_400.0 que cubre el número de segundos en un día. Su granularidad (`Duration'Small`) no debe ser mayor de 20 milisegundos, aunque debe interpretarse como segundos.

El paquete `Calendar` ofrece operadores aritméticos entre los tipos `Time` y `Duration`, así como operadores de comparación para el tipo `Time`.

En la versión de ADA95 se ha introducido un nuevo paquete para manejar el reloj más apropiado para el tiempo real que es el `Real_Time`. Este es similar al paquete `Calendar` pero ofrece una granularidad más fina. La constante `Time_Unit` es la cantidad de tiempo menor que se puede representar con el tipo `Time` y su valor no debe ser mayor de un milisegundo. El rango de `Time` debe ser de al menos 5 años.

El equivalente a `Duration` es el tipo `Time_span` (modelo denso). El tiempo avanza a intervalos de duración `Tick` (modelo discreto).

El reloj de `Real_Time` debe ser monotónico, es decir, avanza de forma continua (nunca puede retroceder, ajustándose variando su velocidad). Por el contrario, el reloj de `Calendar` pretende ser una abstracción de un reloj de pared, y está sujeto a ajustes de años y de segundos.

3.6.2.2 El reloj en C y POSIX

El estándar ANSI C posee una librería para interactuar con un tiempo del tipo 'calendario'. Define un tipo de tiempo básico `time_t`, y varias rutinas para manipular objetos de este tipo. POSIX permite la manipulación de varios relojes en una misma aplicación, cada uno distinguido por un identificador del tipo `clockid_t`. El estándar IEEE obliga a que exista al menos un reloj con el identificador `CLOCK_REALTIME`.

El interfaz para el reloj en tiempo real de POSIX es:

```
#define CLOCK_REALTIME ...;
struct timespec {
     time_t tv_sec;       /* Numero de segundos */
     time_t tv_nsec;      /* Numero de nanosegundos */
};
typedef ... clockid_t;
int clock_settime(clockid_t clock_id, const struct timespec *tp)
int clock_gettime(clockid_t clock_id, struct timespec *res)
int nanosleep(const struct timespec *rqtp, struct timespec *rmtd);
```

La estructura `timespec` se utiliza para obtener el valor que devuelve un reloj (a través de la función `clock_gettime`); `tv_sec` representa el número de segundos transcurridos desde el 1 de Enero de 1970, y `tv_nsec` el número de nanosegundos de la fracción de segundo. POSIX obliga a que la resolución mínima del reloj de tiempo real sea de 50 Hz. (20 mseg.). La resolución de la implementación se puede obtener con la función `clock_getres`.

La función `nanosleep` deja suspendido el proceso. Si el proceso es despertado por una señal antes del tiempo indicado devuelve –1, dejando en `rmtd` el tiempo que falta para cumplir el retardo.

3.6.3 Retardos y timeouts

Una de las mayores utilidades del hecho de disponer de un reloj es poder retardar un proceso o suspender su ejecución hasta una determinada hora. Se utilizan dos tipos de retardos:

- Relativos. La ejecución se suspende durante un cierto tiempo
- Absolutos. La ejecución se suspende hasta una determinada hora.

La mayoría de los lenguajes ofrecen un retardo relativo. Si no se dispone de la facilidad del retardo absoluto, se puede realizar utilizando un retardo relativo y la hora actual del sistema (hora absoluta - hora actual), aunque puede ocurrir que el proceso sea interrumpido justo entre el cálculo de la diferencia y la llamada al retardo, con lo que el tiempo que transcurre hasta que el proceso se reanude retrasaría la hora de puesta en marcha.

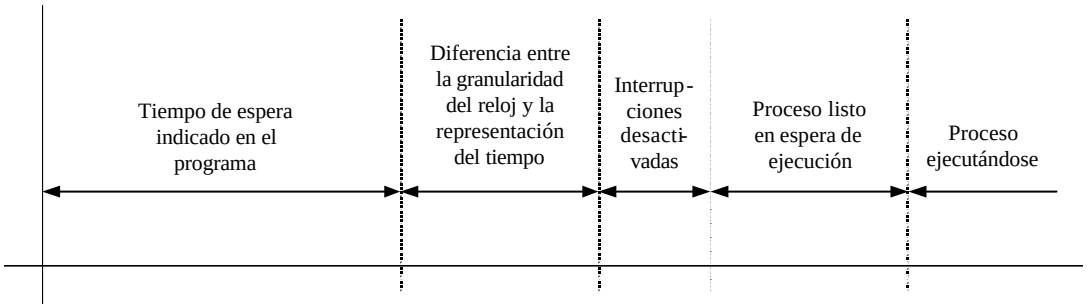
Hay que tener en cuenta que existen varios factores en un sistema multiproceso que pueden alterar el tiempo en que se reanuda un proceso después de un retardo. Estos son:

- La diferencia entre la granularidad en la expresión del tiempo y la real del reloj. Normalmente el reloj se implementa utilizando una interrupción periódica que suele tener una granularidad mayor que la soportada por el reloj.

- El tiempo en que están inhibidas las interrupciones en el sistema. Esto puede hacer que la interrupción que despertaría nuestro proceso se retardara si coincidiera con un espacio con interrupciones inhibidas, hasta que se habilitaran de nuevo, o se terminaran de procesar interrupciones de mayor prioridad.

- El tiempo para pasar de listo a ejecutable. Puede ocurrir que cuando se active el proceso estén listos para ejecutarse procesos de mayor prioridad. Hasta que no finalizara el trabajo de estos no entraría nuestro proceso.

Todos estos factores hacen que el tiempo de retardo sufra un cierto desplazamiento.



Desplazamiento del retardo

La utilización de retardos relativos en bucles puede hacer que el desplazamiento en el tiempo de retardo se acumule y haga que la tarea se desincronice con el resto del sistema. Pare evitar esto conviene utilizar siempre retardos absolutos en los bucles.

Ada dispone de las sentencias `delay` y `delay until` para efectuar retardos relativos y absolutos respectivamente.

En un sistema de tiempo real es importante controlar la no ocurrencia de un suceso que esté esperando un proceso, como por ejemplo la lectura de una sensor, llegada de un mensaje o la liberación de un semáforo. Se trata pues poder introducir restricciones temporales a la ocurrencia de sucesos en un determinado proceso. A estas restricciones de le llaman **timeouts**.

En POSIX los timeouts se pueden programar utilizando señales pues, cuando un proceso recibe una señal en un estado de espera, este es despertado devolviendo una condición de error. En el caso de la función `nanosleep`, si se recibe una señal durante su ejecución, además de devolver un valor de error indicando que no ha terminado la espera, en el parámetro `rmtd` devuelve el tiempo que falta para finalizar la espera.

En las variables condición de POSIX se puede utilizar la función `pthread_cond_timedwait` para realizar una espera sobre la variable condición con timeout.

El lenguaje Ada acepta timeouts en la lectura de mensajes y en los objetos protegidos, permitiendo en las sentencias `select` la sentencia `delay` como una posibilidad más. Si transcurrido en tiempo

indicado en el `delay` no se ha producido ninguna de las posibilidades del `select` se ejecuta las sentencias que siguen al `delay`.

Otra utilidad de los timeouts es controlar el tiempo de ejecución de un determinado cáculo, y si éste supera un determinado deadline finalizarlo para realizar un proceso alternativo.

En Ada esto se puede expresar de la forma:

```
select
     delay 0.1;
then abort
     -- sentencias del calculo
end select;
```

En el código anterior, si las sentencias del cálculo no se completan en 100 mseg, se abortaría su ejecución y se pasaría a ejecutar la sentencia siguiente al `end select`.

En C esta funcionalidad se puede implementar utilizando las señales y las funciones `setjmp` y `longjmp`.

Los timeouts son una característica importante en los sistemas de tiempo real pero, como veremos al hablar de planificación, no son suficientes para abordar todas las necesidades que aparecen en estos sistemas (prioridades, comunicación, sincronización, etc.).