Fiabilidad y Tolerancia a Fallos

- Los sistemas empotrados es usual que presenten requerimientos de fiabilidad y seguridad
- Cada vez mas se utilizan sistemas de control de tiempo real en sistemas críticos
- Fuentes de error:
  - Especificaciones inadecuadas
  - Errores de diseño en los componentes software
  - Errores introducidos por fallos hardware
  - Fallos producidos por interferencias
- Se requieren facilidades para la TF en los lenguajes
- No se requiere la tolerancia de fallos al 100%, pero si una respuesta controlada

Definiciones

- **Fiabilidad**: “Una medida de los sucesos en los cuales el sistema se ajusta al comportamiento definido en unas determinadas especificaciones que se dan como válidas”
- **Avería**: “Comportamiento del sistema que se desvía del indicado en las especificaciones”
- **Error**: “La parte del estado del sistema que lo diferencia de un estado válido”
- **Fallo**: “Defecto o imperfección, físico o de diseño, en algún elemento del sistema”

En ocasiones se usan indistintamente los conceptos de fallo y error.

Tolerancia a fallos

- Consideramos un sistema como un conjunto de estados internos y externos
- Un estado interno no especificado se puede considerar un error
- El estado de un sistema formado por un conjunto de componentes será el estado de sus componentes
- Los errores en los componentes puede que se propaguen, o no, a los componentes más externos
- Los errores en los componentes que no se traducen en averías del sistema se dice que quedan enmascarados
- La **tolerancia a fallos** consiste en detectar los errores internos y evitar que se traduzcan en averías del sistema

Técnicas de programación para la TF

- Basadas en la redundancia
  - Componentes extra para detectar y recuperar los fallos
  - Se persigue conseguir la máxima fiabilidad con la mínima redundancia
  - Nos centraremos en la redundancia software
- Tipos de redundancia
  - Estática: Programación de N-versiones
    - Ejecución simultánea
    - Mecanismo de votación
  - Dinámica: La recuperación se realiza en un paso posterior
    - Hacia atrás: Vuelta a un estado correcto
    - Hacia delante: Corrección del fallo

Bloques de Recuperación

- Redundancia dinámica hacia atrás
- Son bloques en el sentido usual de los lenguajes
  - La entrada corresponde a un punto de recuperación
  - En la salida se realiza un test de aceptación
- Funcionamiento
  - Primero se guarda el estado y ejecuta un modulo primario
  - Si falla el test se restaura el estado y se ejecuta un módulo alternativo
  - Así sucesivamente hasta el número de módulos definidos

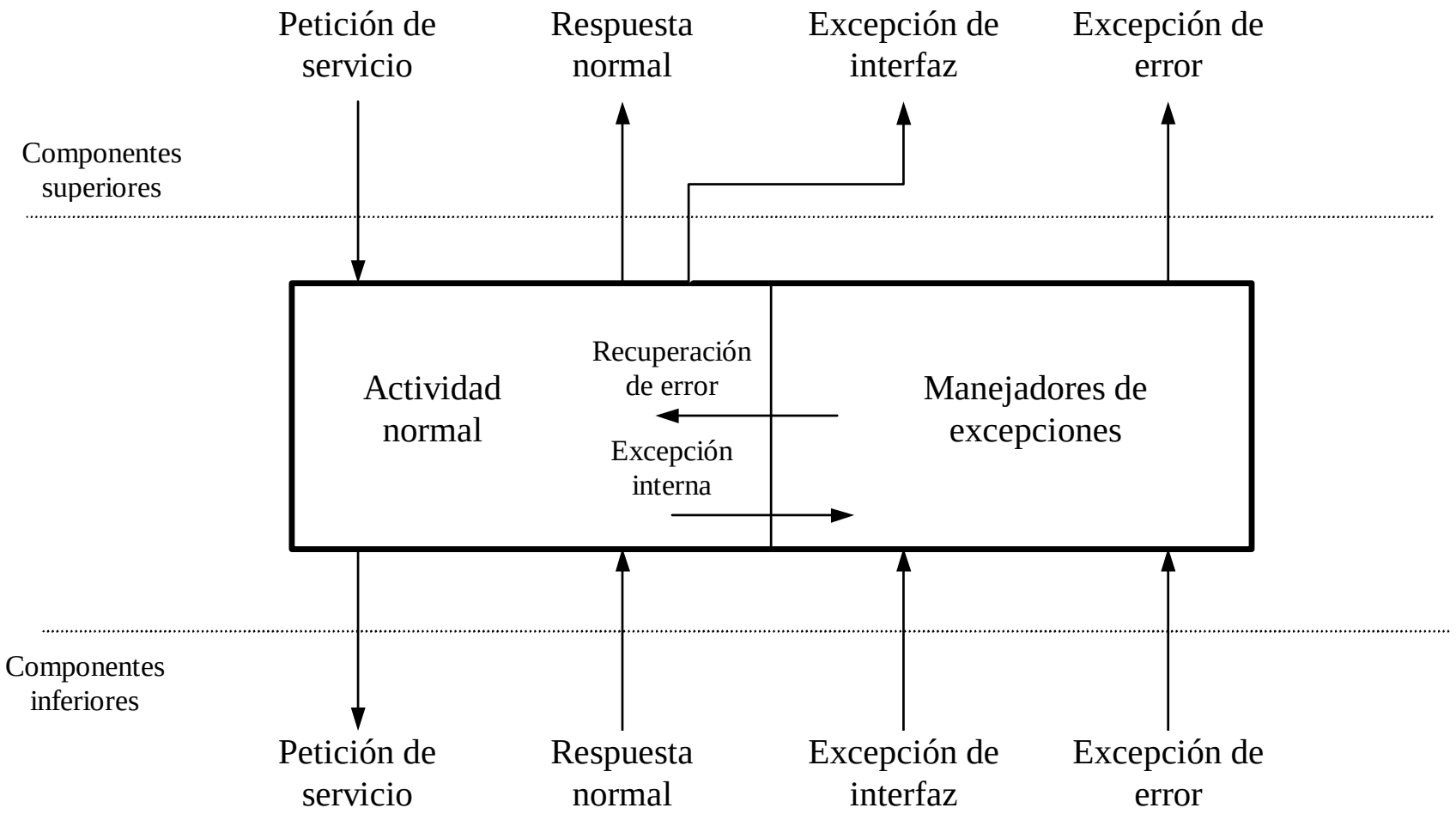
Bloques de Recuperación - Sintaxis

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

Excepciones

- Redundancia dinámica hacia delante
- Una excepción la podemos definir como la aparición de un error, que puede desencadenar una avería
- Nomenclatura:
  - Aparición: **raise** (levantar), **signal** (señalizar), **throw** (lanzar)
  - Respuesta: **handler** (manejador), **catch** (captura)
- Se pueden usar para realizar recuperación hacia atrás:
  - Las distintas alternativas se ejecutan en un bucle
  - Antes de entrar se guarda el estado
  - Si se produce una excepción se restaura el estado y se ejecuta otra alternativa

Componente Ideal Tolerante a Fallos



Manejo de excepciones (requerimientos)

- Debe ser fácil de entender y de utilizar
- El código no debe dificultar la compresión del programa funcionado sin errores
- No introducir sobrecarga si no hay excepciones
- Tratar de forma uniforme las excepciones detectadas por el entorno y las lanzadas por la aplicación
- Permitir la programación de acciones de recuperación

Representación de las excepciones

- Clasificación por el retraso introducido
  - Síncrona: Se activa inmediatamente después de producirse el error
  - Asíncrona: Se lanza algún tiempo después
- Clasificación según quién la activa
  - El propio proceso
  - Un proceso distinto o el entorno
- Las excepciones asíncronas también se denominan señales. No son adecuadas para al tolerancia a fallos
- Nos centraremos en las excepciones síncronas
- Modelos de declaración: Etiqueta (Ada), Objeto (C++)

Dominio de un manejador

- Una excepción puede tener varios manejadores
- El dominio de un manejador de excepción es la zona de código en que se activa si se produce en ella la excepción que controla
- En los lenguajes estructurados, el dominio coincide con un bloque de programación
- Un manejador se puede definir para una excepción o para varias

Dominio de un manejador (ejemplos)

En ADA

```
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
```

En C++

```
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
}
```

Propagación de las excepciones

- Define que ocurre cuando una excepción se produce en un bloque que no tiene definido su manejador
  - El compilador produce un error irrecuperable
  - Buscar el manejador en la cadena de llamadas a la función o procedimiento (propagación dinámica)
  - Buscar el manejador en la jerarquía de bloques (propagación estática)
- Si no se encuentra el manejador el programa aborta
- Se puede permitir definir un manejador por defecto que capture las excepciones sin manejador (catch all)

Modelo de reanudación y terminación

- Reanudación (o notificación): La ejecución continua donde se ha producido la excepción
- Terminación (o escape): Al terminar el manejador la ejecución continua a la salida del bloque en que se produce
- Algunos lenguajes permiten repetir la ejecución del bloque
- Con la reanudación se debe reparar el error. Si no es posible se debe usar el modelo de terminación
- Modelo híbrido: el manejador puede decidir si continuar la ejecución con el modelo de reanudación

Uso del tiempo

- Se necesitan facilidades para poder tener percepción del tiempo y usarlo en la aplicación
- La noción del tiempo es necesaria para:
  - Representar los requerimientos temporales
  - Satisfacer los requerimientos temporales
  - Interacción con el tiempo
- Propiedades matemáticas del tiempo:
  - 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: $\forall\, x, y : x < y \Rightarrow \exists\, z : (x < z < y)$
- Se suele asumir un tiempo discreto en lugar de denso

Acceso a un reloj

- Medida del paso del tiempo
  - Accediendo al marco temporal del entorno
  - Utilizando un reloj interno
- Se usa más el reloj interno, basado en una interrupción periódica, aunque ofrece un modelo discreto
- El periodo de la interrupción será la resolución temporal del sistema
- Se puede mejorar la medida del tiempo accediendo al contador interno del timer que provoca la interrupción

El reloj en Ada

- Reloj calendario en la primera versión de Ada
  - Paquete `Calendar`
  - Tipo abstracto para el tiempo `Time` y la función `Clock` para obtener el tiempo
  - Subprogramas para convertir el tiempo a la hora real: `Years, Months, Days` y `Seconds`
  - El tipo `Duration` es un tipo real de coma fija que permite realizar cálculos de tiempo. Su granularidad es `Duration'Small` que no debe ser mayor de 20 mseg
  - Ofrece operadores aritméticos entre los tipos `Time` y `Duration` , así como operadores de comparación para el tipo `Time`

El reloj en Ada (continuación)

- Reloj para tiempo real en Ada-95
  - Paquete `Real_Time`
  - Ofrece mayor granularidad pero no tiene relación con la hora real (reloj monotónico)
  - Usa los tipo `Time` y `Time_Span`
  - La constante `Time_Unit` es la cantidad menor de tiempo que se puede representar y su valor no debe ser mayor que 1 ms. El rango de `Time` debe ser mayor de 5 años
  - El tiempo avanza a intervalos de la constante `Tick`

El reloj en C y POSIX

- Define un tipo de tiempo básico `time_t` y varias rutinas para manejar variables de este tipo
- Permite usar varios relojes en una aplicación, distinguidos por un identificador del tipo `clockid_t`. Al menos debe existir el `CLOCK_REALTIME`
- La representación del tiempo se hace con la estructura que contiene la hora en segundos desde el 1/1/1970 y la fracción de segundo en nanosegundos
- La resolución mínima del reloj debe ser de 20 mseg

El reloj en C y POSIX (interfaz)

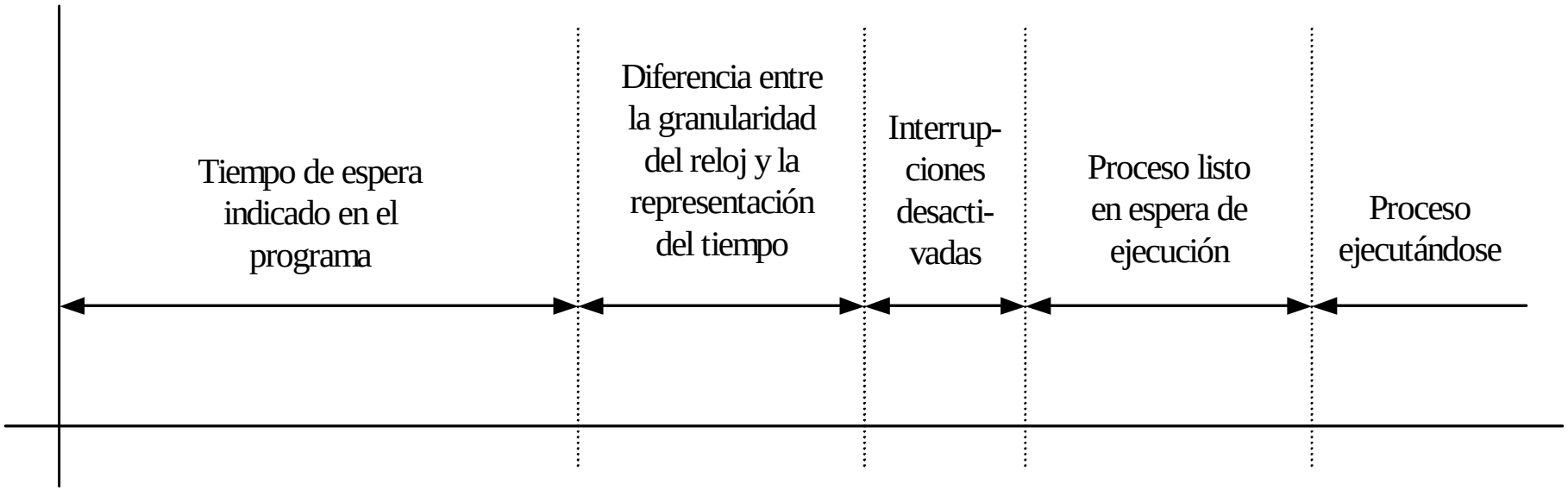
```
#define CLOCK_REALTIME ...;
struct timespec {
    time_t tv_sec;        /* Numero de segundos */
    time_t tv_nsec;       /* Numero de nanosegundos */
};
typedef ... clockid_t;

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);
```

Retardos

- Tipos de retardos:
  - Relativos. Suspensión durante un tiempo
  - Absolutos. Suspensión hasta una hora
- La mayoría de los lenguajes ofrecen un retardo absoluto. Atención al retardo (hora absoluta – hora actual)
- Factores que pueden alterar el tiempo de retardo
  - Diferencia entre la expresión y la precisión del tiempo
  - Tiempo de inhibición de interrupciones
  - Tiempo para pasar del estado Listo a Ejecutando
- Los retardos relativos en bucles puede producir errores acumulativos
- Ada dispone de las sentencias `delay` y `delay until`

Factores que afectan a los retardos



Time-Outs

- Permiten controlar la no ocurrencia de un suceso que está esperando un proceso
- En POSIX los time-out se pueden controlar utilizando señales
- También se pueden usar en las variables condición de los pthreads
- En el lenguaje Ada se usa una construcción select

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