Sistemas en Tiempo Real

Grado en Ingeniería Electrónica

Capítulo 4. Programación concurrente

Capítulo 4, excepto puntos 4.7 y 4.9 de la edición Inglés (2009)

Capítulo 7 completo de la edición en Español (2003)

Tutoría

0. Programación concurrente
1. Procesos y tareas
2. Ejecución concurrente
3. Representación de tareas
4. Ejecución concurrente en Ada
5. Ejecución concurrente en Java
6. Ejecución concurrente en C/Real-Time POSIX
7. Un Sistema simple embebido

Programación concurrente

- Denominamos programación concurrente a la notación y técnicas de programación que expresan el paralelismo potencial y que resuelven los problemas resultantes de la sincronización y la comunicación.
- El diseño de programas concurrentes busca:
  - Modelar el paralelismo en el mundo real, de manera que las aplicaciones sean más fáciles de comprender y mantener, y más fiables.
  - Utilizar todas las características del procesador.
  - Utilizar varios procesadores para resolver un problema.
- Desde la perspectiva de la concurrencia, la plataforma utilizada se puede considerar irrelevante.
- Un programa concurrente se puede ver como un conjunto de procesos secuenciales que se ejecutan en paralelo.

1. Procesos y tareas

- Los programas secuenciales tiene un único hilo de control. Se puede variar la traza en función de los datos de entrada de cada ejecución pero, para una ejecución concreta existe solo una traza. En los STR esto no es lo más adecuado.
- Tradicionalmente, un programa concurrente puede verse como un conjunto de procesos secuenciales autónomos que se ejecutan en paralelo. Cada proceso tiene un hilo de control; es un programa independiente.
- Si embargo, es importante distinguir la concurrencia entre programas (procesos) de la concurrencia dentro de las tareas de un mismo programa (threads o tareas), ya que estos deben tener acceso a memoria compartida.
- Conceptos:
  - **Tarea o thread**: indica un hilo de control.
  - **Proceso**: indica una o más tareas ejecutándose en un contexto de memoria compartida.
  - *La mayoría de los lenguajes de programación concurrentes permiten implementar la abstracción de tareas más que la de procesos.*

1. Procesos y tareas

- Los programas concurrentes pueden utilizar el concepto de tarea/thread del S.O. en el que corren, o bien implementar la concurrencia de manera transparente al S.O.
- El término concurrente indica un paralelismo potencial. Los lenguajes de programación concurrentes permiten expresar actividades paralelas sin considerar su implementación.
- Un proceso puede:
  - Multiplexar sus ejecuciones en un único procesador * (no sería una ejecución paralela real)
  - Multiplexar sus ejecuciones en un sistema multiprocesador con acceso a memoria compartida
  - Multiplexar sus ejecuciones en diversos procesadores con memoria independiente (sistemas distribuidos).
  - (También puede haber mezclas de esto métodos)

1. Procesos y tareas

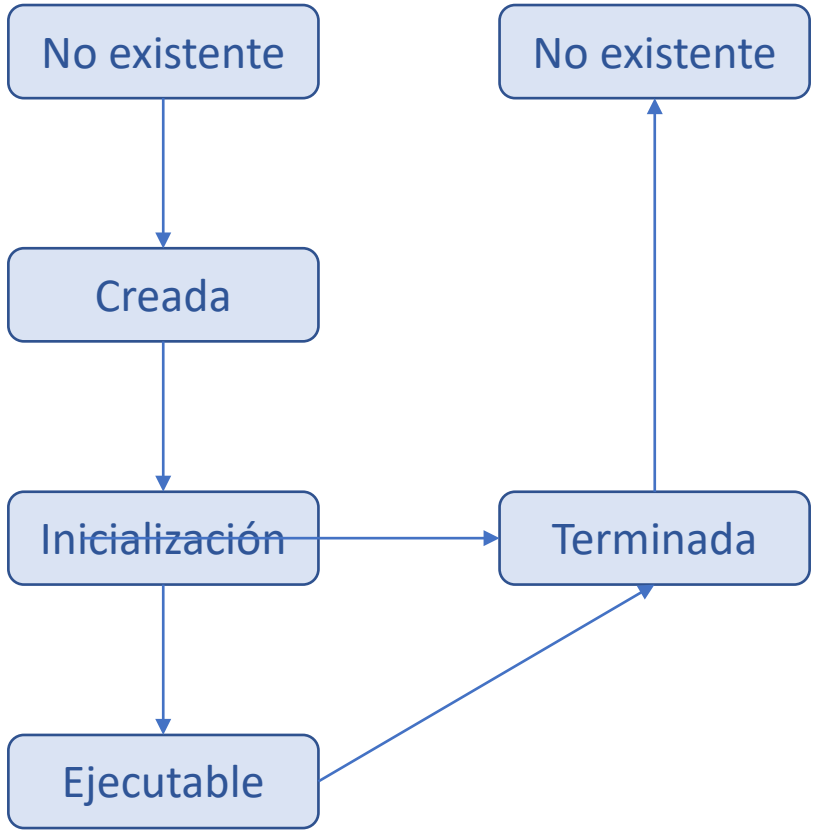


Diagrama de estados de una tarea

- El Run-Time Support System (RTSS) debe encargarse de la planificación de las tareas: creación, finalización y asignación de procesador.
- Posibilidades del RTSS:
  - Programado como una estructura dentro de la aplicación: C y C++ tienen sus propias librerías para gestionar las tareas
  - Un sistema software generado por el compilador junto con el código objeto. Lo habitual en Ada y Java.
  - Una estructura hardware microcodificada en el procesador por motivos de eficiencia. El procesador aJ100 del sistema aJile soporta directamente la ejecución del código byte de Java.
  - El algoritmo de planificación de las tareas afecta al comportamiento temporal del programa, aunque no debería afectar al comportamiento lógico del mismo.

1.1 Constructores para programación concurrente

- Servicios fundamentales:
  - La expresión de actividades concurrentes (threads, tareas o procesos)
  - Mecanismos de sincronización entre actividades concurrentes.
  - Primitivas para la comunicación entre actividades concurrentes.
- Tipos de comportamiento en la interacción de tareas:
  - **Independientes**: no se comunican.
  - **Cooperativos**: se comunican con regularidad y sincronizan sus actividades para operaciones comunes.
  - **Competitivos**: procesos y tareas independientes deben competir para utilizar determinados recursos del sistema (periféricos, memoria y procesador). Esto implica comunicación y sincronización.

2. Ejecución concurrente

- Estructura:
  - Estática: número de tareas fijo y conocido en tiempo de compilación
  - Dinámica: las tareas se deciden en tiempo de ejecución
- Nivel de paralelismo:
  - Anidado: se pueden definir tareas dentro de otras
  - Plano: las tareas se definen en el nivel más exterior del programa.
- Granularidad:
  - Grueso: pocas tareas importantes. (La mayoría de los lenguajes concurrentes, representados por Ada).
  - Fino: muchas tareas sencillas. (Occam2).
- Inicialización:
  - A través de parámetros de la tarea
  - Comunicación explícita una vez que ha comenzado la ejecución de la tarea
- Finalización:
  - Completar la ejecución de la tarea
  - Suicidio por ejecución de una sentencia de finalización
  - Aborto por la acción de otra tarea
  - Un error no controlado
  - Nunca: tareas que se ejecutan en bucles infinitos
  - Cuando ya no es necesaria
- Representación:
  - Padre/hijo. Una tarea crea a la otra. El padre debe esperar mientras el hijo se crea e inicializa.
  - Guardián/dependiente (de toda la tarea o de un bloque interno). El guardián no puede terminar un bloque hasta que todas las tareas dependientes lo hagan.

  - *El padre y el guardián pueden ser o no la misma tarea. Siempre será la misma cuando se trate de estructura estática (Occam2). En estructuras dinámicas, puede ser la misma o diferente.*

2. Ejecución concurrente

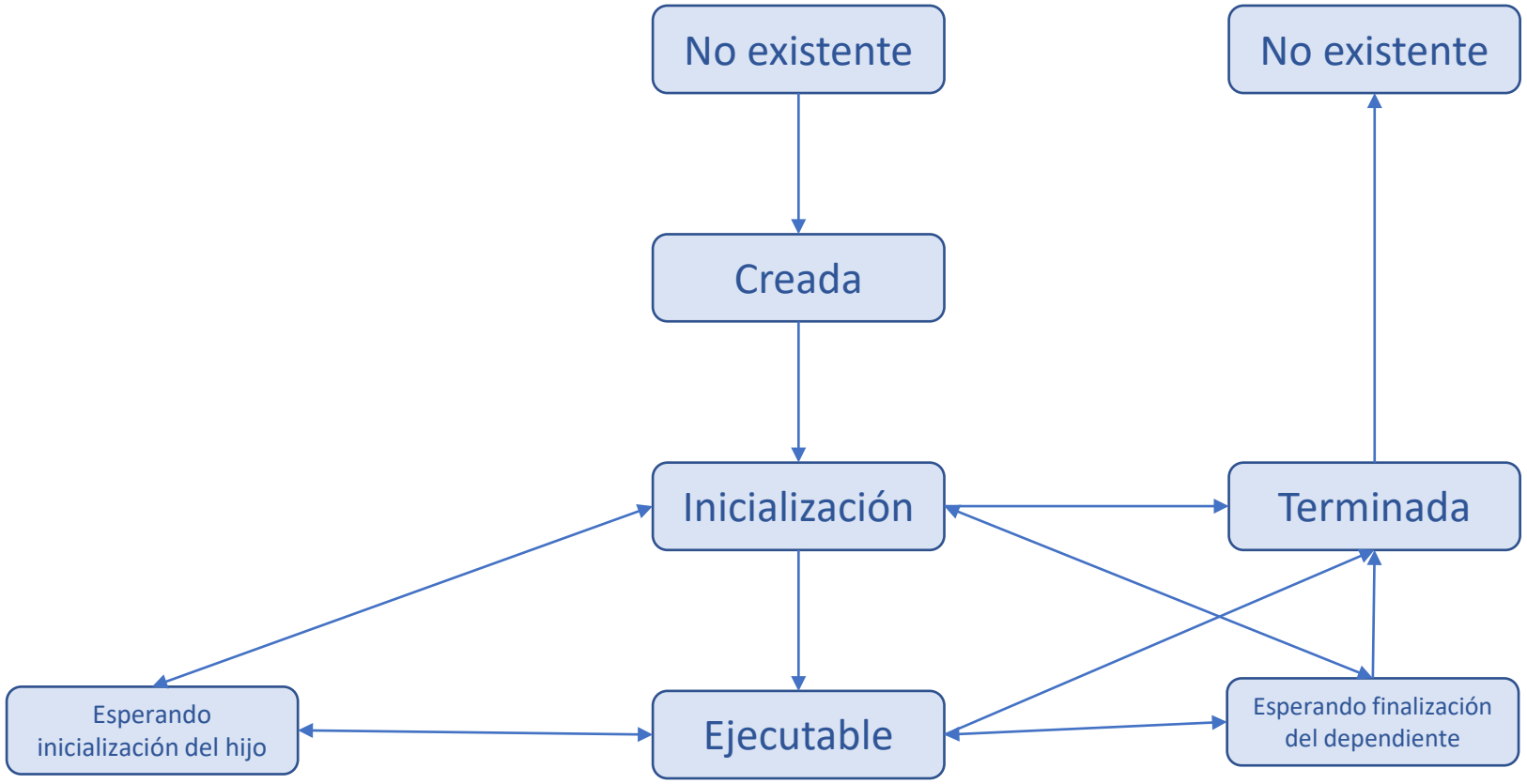


Diagrama de estados de una tarea

2.1 Tareas y objetos

- En un lenguaje concurrente, las tareas representan entidades **activas**.
  - Realizan acciones espontáneas para que la ejecución prosiga.
- Las entidades **reactivas (recursos)** pueden ser datos variables o estar encapsuladas en alguna construcción que proporcione una interfaz procedural.
  - Realizan acciones cuando son invocadas por una entidad activa.
  - Serán **pasivas** cuando permitan un acceso completo.
  - Si el control de un recurso se lleva a cabo a través de un agente pasivo (semáforo), el recurso se dice **protegido o sincronizado**.
  - Si es necesario un agente activo, se dirá que el recurso es un **servidor**.

Desde una perspectiva de POO:

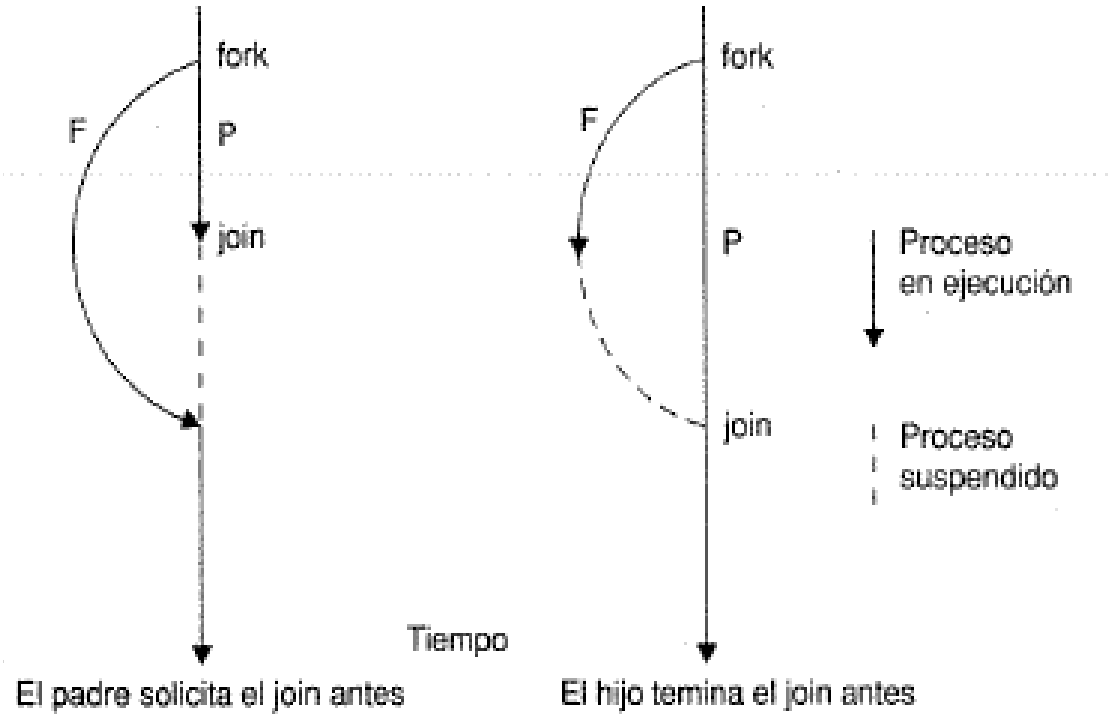
- Objeto pasivo
  - Entidad reactiva sin restricciones de sincronización. Necesita una tarea externa de control para ejecutar sus métodos.
- Objeto protegido
  - Entidad reactiva con restricciones de sincronización (compartida entre varias tareas). Necesita una tarea externa de control para ejecutar sus métodos.
- Objeto activo
  - Objeto con una tarea interna explícita o implícita.
- Objeto servidor
  - Objeto activo con restricciones de sincronización (compartido entre varias tareas).

3. Representación de tareas

Fork y join

- *Fork indica que una rutina debe ejecutarse concurrentemente con la que ha invocado el fork.*
- *Join permite al que lo invoca sincronizarse con la finalización de la invocada.*

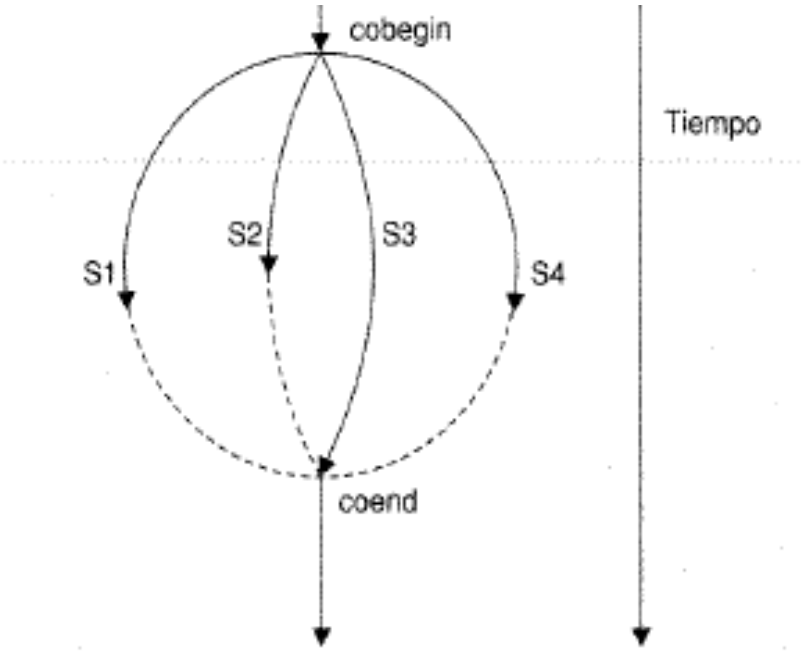
- *Estas primitivas son las más utilizadas. También pueden llamarse **spawn** y **join**.*
- *En Linux **clone** permite crear procesos hijos que compartan memoria. El mecanismo de join se hace con **wait** y **waitpid**.*
- *En C/Real-Time POSIX se usa **fork** y **vfork** para crear una copia de la tarea, junto con **wait** y **waitpid**.*



3. Representación de tareas

Cobegin

- *Permite la ejecución concurrente de las instrucciones entre cobegin y coend*
- *La instrucción cobegin termina cuando han terminado todas las instrucciones concurrentes.*
- *Estas instrucciones pueden ser de cualquier tipo. Desde asignaciones simples a llamadas a procedimientos.*
- *También se permiten anidaciones de cobegin, construyendo así una jerarquía de tareas.*



4. Ejecución concurrente en Ada

- La unidad de paralelismo se llama task (tarea)
- Se puede declarar en cualquier nivel de programa. Cada tarea tiene especificación y cuerpo.
- Se pueden crear de tipo anónimo o especificar un tipo.

- Identificación de tareas: Paquete **Ada.Task_Identification**

- Posibilidad de finalización de tareas:
  - Al completar su ejecución o como resultado de una excepción sin manejar
  - Al ejecutar la alternativa “terminate” de una instrucción select.
  - Ser abortada por otra tarea en su rango. Cuando esto sucede, todas las que dependen de ella abortarán también.

  - Una tarea puede usar el atributo Terminated para saber si otra ha finalizado.

5. Ejecución concurrente en Java

- Java tiene una clase predefinida para crear tareas: **Java.Lang.Thread**
- Y también un interfaz, para poder crear tareas de otras clases: **java.lang.Runnable** (implementar el método run())

- Como Ada, Java permite la creación dinámica de tareas.
- Como punto diferente, Java permite el paso de datos como parámetros del constructor.

- En Java no existe el concepto de guardián, porque tiene mecanismos de recolección de basura para eliminar los procesos que ya no se van a utilizar.

- Identificación de tareas: Thread **threadID**; o bien: Runnable **threadCodeID**;
- Posibilidad de finalización de tareas:
  - Al finalizar su método run o como resultado de una excepción sin manejar.
  - Las tareas pueden ser **user thread** o **daemon thread** (servicios generales que nunca finalizan – llamar al método setDaemon antes del inicio)
  - Cuando todas las user thread han finalizado, las daemon thread y el programa pueden finalizar.

- Excepciones:
  - RuntimeExceptions relacionadas con Threads: Illegal ThreadStateException, InterruptedException

6. Ejecución concurrente en C/Real-Time POSIX

- POSIX para tiempo real proporciona tres mecanismos para crear actividades concurrentes:
  - fork, junto a la llamada asociada wait , que permite crear una copia completa de un proceso.
  - Spawn (Combina fork y exec).
  - También permite que cada proceso tenga varios hilos de ejecución, con acceso al mismo espacio de memoria.
  - pthread_atfork: especifica 3 funciones que se deben ejecutar: una antes del fork, otra en el padre, y la última en el hijo.

- pthread_create: crea tarea a nivel de aplicación.
- pthread_setconcurrency: indica el nivel de concurrencia deseado.
- Pthread_self: obtiene el identificador de la tarea.

- Posibilidad de finalización de tareas:
  - Cuando acaba su start_routine o llamando a pthread_exit o pthread_cancel.
  - Puede esperar por otra con pthread_join.

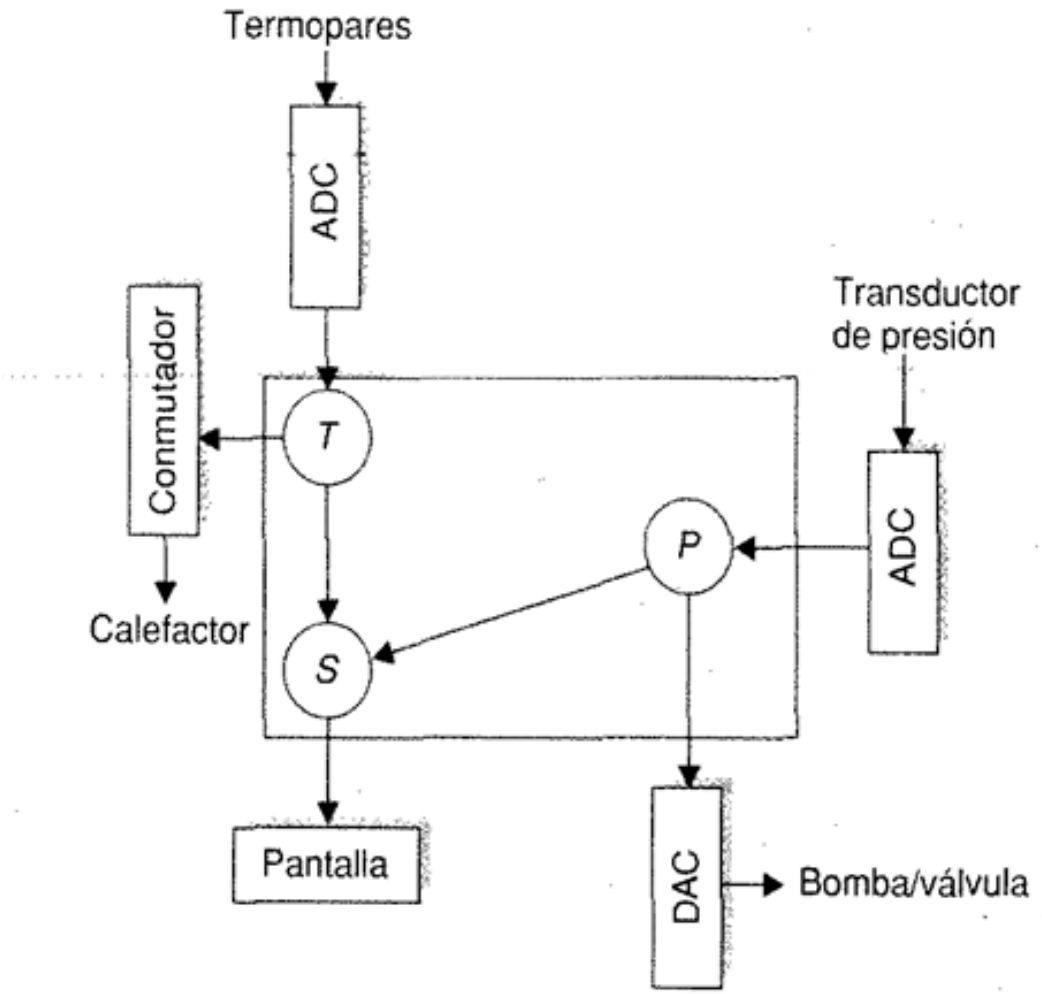
7. Un sistema embebido sencillo

P y T: Procesos activos
S: recurso (recurso protegido o servidor)

Objetivo:
Mantener temperatura y presión

Soluciones:

1. Programa sin concurrencia.
2. T, P y S secuenciales (distintos programas o procesos de un mismo programa). El S.O. gestiona su creación e interacción.
3. Programa único concurrente. No es necesario el control del S.O., pero sí un sistema de soporte de ejecución.



MÓDULO 1: CONCEPTOS BÁSICOS DE LOS SISTEMAS EN TIEMPO REAL

Este módulo explica los fundamentos de los sistemas de tiempo real y las características a tener en cuenta en su desarrollo. Se define el modelo de fallo y el concepto de tolerancia a fallos, indicando algunas técnicas para su resolución. Se termina el módulo explicando cómo se gestionan las excepciones (posibles errores) desde el punto de vista del lenguaje de programación y del propio sistema operativo (en tiempo real o no)

Autoevaluaciones de las unidades

- Auto-evaluación de la Unidad 1: Introducción a los sistemas en tiempo real
- Autoevaluación de la Unidad 2: Fiabilidad y tolerancia a fallos
- Autoevaluación de la Unidad 3: Excepciones y manejo de excepciones

MÓDULO 2: ASPECTOS DE LA CONCURRENCIA EN SISTEMAS EN TIEMPO REAL

Este módulo explica los conceptos de ejecución concurrente, que se basa en el modelo de procesos/tareas/hebras. Se indica cómo se implementa la concurrencia en los tres modelos de programación (Java RT, Ada y C/Real Time POFIX) para pasar a describir como se implementan los mecanismos de comunicación y sincronización mediante variables compartidas y mensajes. A continuación se introduce el concepto de acción atómica y la forma de implementarlas en los tres modelos de programación propuestos. Se finaliza el módulo explicando la gestión de acceso a los recursos y el modelo de interbloqueo

- Autoevaluación de la Unidad 4: Programación concurrente
- Autoevaluación de la Unidad 5: Sincronización y comunicación basada en variables compartidas
- Autoevaluación de la Unidad 6: Sincronización y comunicación basada en mensajes
- Autoevaluación de la Unidad 7: Acciones atómicas, procesos concurrentes y fiabilidad
- Autoevaluación de la Unidad 8: Control de recursos

Prueba de evaluación a distancia (PED1): Instalación/Configuración de un RTOS
Prueba de evaluación a distancia (PED2): Instalación/Configuración de RTSJ

MODULO 3: TEMPORIZACIÓN DE LAS TAREAS DE UN SISTEMA EN TIEMPO REAL

Este módulo explica los mecanismos de temporización que se pueden aplicar en un sistema de tiempo real, tomando como base el reloj del sistema en tiempo real. Una vez definidos los

Sistemas en Tiempo Real

Grado en Ingeniería Electrónica

Capítulo 4. Programación concurrente

UNED

Nieves Pedreira Souto