Mike0418
www.wuolah.com/student/Mike0418

Resumen SCD Temas 1 y 2.pdf

Resúmenes

2° Sistemas Concurrentes y Distribuidos

Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
UGR - Universidad de Granada

Resumen SCD Temas 1 y 2

Tema 1: Introducción

1. Conceptos básicos y motivación.

1.1. Conceptos básicos relacionados con la concurrencia.

- **Programa secuencial:** Declaraciones de datos + Conjunto de instrucciones sobre dichos datos que se deben ejecutar en secuencia.
- **Programa concurrente:** Conjunto de programas secuenciales ordinarios que se pueden ejecutar lógicamente en paralelo.
- **Proceso:** Ejecución de un programa secuencial.
- **Concurrencia:** Describe el potencial para ejecución paralela, es decir, el solapamiento real o virtual de varias actividades en el tiempo.
- **Programación Concurrente (PC):** Conjunto de notaciones y técnicas de programación usadas para expresar paralelismo potencial y resolver problemas de sincronización y comunicación. Es independiente de la implementación del paralelismo. Es una abstracción
- **Programación paralela:** Su principal objetivo es acelerar la resolución de problemas concretos mediante el aprovechamiento de la capacidad de procesamiento en paralelo del hardware disponible.
- **Programación distribuida:** Su principal objetivo es hacer que varios componentes software localizados en diferentes ordenadores trabajen juntos.
- **Programación de tiempo real:** Se centra en la programación de sistemas que están funcionando continuamente, recibiendo entradas y enviando salidas a/desde componentes hardware (sistemas reactivos), en los que se trabaja con restricciones muy estrictas en cuanto a la respuesta temporal (sistemas de tiempo real).

1.2. Motivación de la Programación Concurrente.

Mejora de la eficiencia: permite aprovechar mejor los recursos hardware existentes.

- **En sistemas con un solo procesador:**

Cuando la tarea que tiene el control del procesador necesita realizar una E/S cede el control a otra para evitar esperas. SO Multiusuario.

- **En sistemas con varios procesadores:**

Reparte las tareas entre los procesadores. Acelerar operaciones numéricas muy complejas.

Mejora de la calidad: muchos programas se entienden mejor como varios procesos secuenciales ejecutándose concurrentemente que como un único programa secuencial.

2. Modelo abstracto y consideraciones sobre el hardware.

2.1. Consideraciones sobre el hardware.

Modelos de arquitecturas para programación concurrente.

Dependen fuertemente de la arquitectura.
Base homogénea para la ejecución de los procesos concurrentes.
Paralelismo afecta a la eficiencia pero no a la corrección.

INESEM
BUSINESS SCHOOL

Ponle nombre a lo que quieres ser

Master BIM Management

udima
UNIVERSIDAD A DISTANCIA DE MADRID

60 Créditos ECTS

Dynamo AUTODESK® REVIT® AUTODESK® NAVISWORKS®

Jose María Girela
Bim Manager.

Concurrencia en sistemas monoprocesador.
Multiprogramación. Mejor aprovechamiento CPU. Multiusuario. Soluciones de diseño concurrente. Sincronización y comunicación con variables compartidas.

Concurrencia en multiprocesadores de memoria compartida.
Procesadores no comparten espacio de direcciones, pueden compartir físicamente la misma memoria. Interacción entre procesos mediante variables compartidas.

Concurrencia en sistemas distribuidos.
No hay memoria común, cada procesador con su propio espacio de direcciones. Procesadores interaccionan a través de paso de mensajes. Programación distribuida.

2.2. Modelo Abstracto de concurrencia.

- **Sentencias atómicas y no atómicas**

Sentencia atómica: siempre se ejecuta de principio a fin sin ver afectada su ejecución por otras sentencias en ejecución de otros procesos del programa.
No depende de la ejecución de otras.
El estado al acabar está determinado.
El estado de ejecución: valores de las variables y registros de todos los procesos.

Sentencia no atómica: su ejecución puede verse afectada por otras sentencias en ejecución de otros procesos del programa.
Hay indeterminación (no se puede predecir el estado final a partir del inicial).

- **Conceptos de interfoliación y abstracción**

Interfoliación: suceso que ocurre al ejecutar procesos secuenciales a la vez, se pueden producir diferentes mezclas de las sentencias atómicas de cada uno de ellos, cada una de esas mezclas se denomina interfoliación.

Abstracción: está constituida por el estudio de todas las posibles interfoliaciones.
Se consideran sólo las características relevantes que determinan el resultado del cálculo: simplificar el análisis y diseño de los programas concurrentes.
Los detalles no relevantes para el resultado son ignorados.

- **Independencia del entorno de ejecución**

El entrelazamiento preserva la consistencia. El resultado de una instrucción individual sobre un dato no depende de las circunstancias de la ejecución.

- **Velocidad de ejecución. Hipótesis del progreso finito**

Progreso finito: No se puede hacer ninguna suposición acerca de las velocidades absolutas/relativas de ejecución de los procesos, salvo que es mayor que 0.

- **Estados e historias de ejecución de un programa concurrente**

Estado de un programa concurrente: valores de las variables del programa en un momento dado. Incluye también variables con información de estado oculta.
Historia o traza de un programa concurrente: secuencia de estados producida por una secuencia concreta de interfoliación.

- **Notaciones para expresar ejecución concurrente**

Sistemas Estáticos: número de procesos fijado en el fuente del programa. Se activan al lanzar el programa.
Sistemas Dinámicos: número variable de procesos/hebras que se pueden activar en cualquier momento de la ejecución.

- **Grafo de sincronización**

Es un Grafo Dirigido Acíclico donde cada nodo representa una secuencia de sentencias.

- **Definición estática de procesos**

Número de procesos y código que ejecutan no cambian entre ejecuciones. Programa acaba cuando acaban todos los procesos. Antes de comenzar proceso, inicialización var. comp. process → asocia cada proceso con su identificador y su código.

- **Definición estática de vectores de procesos**

Grupos de procesos similares que sólo se diferencian en el valor de una constante.

- **Creación de procesos no estructurada con fork-join**

fork: la rutina nombrada puede comenzar su ejecución a la vez que la sentencia siguiente.
join: espera la terminación de la rutina nombrada antes de comenzar la siguiente.
Ventajas: práctica y potente, creación dinámica.
Inconvenientes: no estructuración, difícil comprensión de los programas.

- **Creación de procesos estructurada con cobegin-coend**

Las sentencias comienzan su ejecución a la vez y se espera a que terminen todas.
Ventajas: impone estructura (1 entrada, 1 salida) → más fácil de entender.
Inconvenientes: menor potencia expresiva que fork-join.

3. Exclusión mutua y sincronización.

En un conjunto de procesos que no son independientes entre sí (cooperativos), algunas posibles combinaciones de las secuencias no son válidas.
Hay condición de sincronización cuando hay alguna restricción sobre el orden en que se pueden mezclar las instrucciones.

3.1. Concepto de Exclusión Mutua.

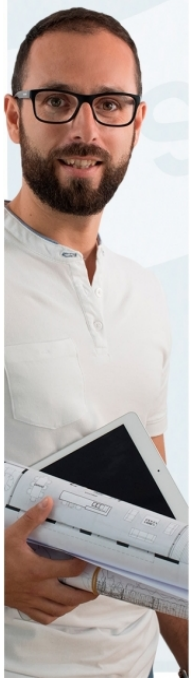
Secuencias finitas de instrucciones que deben ejecutarse de principio a fin por un único proceso, sin que a la vez otro proceso las esté ejecutando también.
Sección crítica: conjunto de secuencias de instrucciones ejecutadas en exclusión mutua.
El solapamiento de instrucciones debe ser tal que cada secuencia de instrucciones de la SC se ejecuta como mucho por un proceso de principio a fin, sin que otros ejecuten ninguna de esas instrucciones ni otras de la misma SC.
Notación pseudocódigo, escribimos las instrucciones en SC con < y >:
cobegin < x := x+1 > ; < x := x-1 > ; coend

3.2. Condición de sincronización.

Establece que: no son correctas todas las posibles interfoliaciones de las secuencias de instrucciones atómicas de los procesos.
Ej: uno o varios procesos deben esperar a que se cumpla una determinada condición global.

4. Propiedades de los sistemas concurrentes.

Propiedad de un programa concurrente: Atributo del programa que es cierto para todas las posibles secuencias de interfoliación. Hay propiedad de seguridad y de vivacidad.



4.1. Propiedades de Seguridad

Son condiciones que deben cumplirse en cada instante, del tipo: nunca pasará nada malo.

- Requeridas en especificaciones estáticas del programa.
- Son fáciles de demostrar y para cumplirlas se suelen restringir las posibles interfoliaciones

4.2. Propiedades de Vivacidad

Son propiedades que deben cumplirse eventualmente: realmente sucede algo bueno.

- Son propiedades dinámicas, más difíciles de probar.

5. Verificación de programas concurrentes.

5.1. Introducción.

¿Cómo demostrar que un programa cumple una determinada propiedad?

Pruebas simples. Limitaciones.

- **Posibilidad:** realizar diferentes ejecuciones del programa y comprobar que se verifica la propiedad.
- **Problema:** sólo permite considerar un número limitado de historias (interfoliación) de ejecución y no demuestra que no existan casos indeseables.

Enfoque operacional (análisis exhaustivo)

- **Enfoque operacional:** análisis exhaustivo de casos, comprueba la corrección de todas las posibles interfoliaciones.
- **Problema:** su utilidad está muy limitada cuando se aplica a programas concurrentes complejos porque el número de interfoliaciones crece exponencialmente con el número de instrucciones de los procesos.

5.2. Enfoque axiomático.

- Se define un sistema lógico formal que permite establecer propiedades de programas en base a axiomas y reglas de inferencia.
- Se usan fórmulas lógicas (asertos) para caracterizar un conjunto de estados.
- Las sentencias atómicas actúan como transformadores de predicados (asertos). Los teoremas en la lógica tienen la forma:

$$\{ P \} S \{ Q \}$$

"Si la ejecución de la sentencia S empieza en algún estado en el que es verdadero el predicado P (precondición), entonces el predicado Q (poscondición) será verdadero en el estado resultante."

- **Menor Complejidad:** El trabajo que conlleva la prueba de corrección es proporcional al número de sentencias atómicas en el programa.
- **Invariante global:** Predicado que referencia variables globales siendo cierto en el estado inicial de cada proceso y manteniéndose cierto ante cualquier asignación dentro de los procesos.
  En una solución correcta del Productor-Consumidor, un invariante global sería:

$$\text{consumidos} \leq \text{producidos} \leq \text{consumidos} + 1$$

Tema 2: Sincronización en memoria compartida

1. Introducción a la sincronización en memoria compartida.

1.1. Soluciones de bajo nivel con espera ocupada.

Basadas en programas que contienen instrucciones de bajo nivel para lectura y escritura directamente a la memoria compartida y bucles para realizar las esperas.

Espera ocupada: debe esperar a que ocurra un evento o sea cierta una condición (bucle).

- Soluciones software: operaciones estándar sencillas de lectura y escritura de datos simples en la memoria compartida.
- Soluciones hardware (cerrojos): existen instrucciones máquina específicas.

1.2. Soluciones de alto nivel.

Se diseña capa software por encima para las aplicaciones. La sincronización se consigue bloqueando un proceso cuando deba esperar.

Las soluciones de bajo nivel producen algoritmos complicados y tienen un efecto negativo en la eficiencia de uso de la CPU (por bucles), en las de alto nivel se ofrecen interfaces de acceso a estructuras de datos y se usa bloqueo de procesos en lugar de espera ocupada.

Semáforos: se construyen sobre las soluciones de bajo nivel, pueden bloquear y reactivar procesos.

Regiones críticas condicionales: de más alto nivel que los semáforos, se pueden implementar sobre ellos.

Monitores: de más alto nivel que los anteriores, se pueden implementar en lenguajes orientados a objetos.

2. Semáforos para sincronización.

2.1. Introducción.

Los semáforos solucionan o aminoran los problemas de soluciones de bajo nivel, tienen un ámbito de uso amplio:

- No usa espera ocupada, sino bloqueo de procesos.
- Problema de exclusión mutua se resuelve con esquemas de uso sencillos.
- El mecanismo implementado mediante instancias de una estructura de datos a las que se accede sólo con subprogramas, lo que aumenta la seguridad y simplicidad.

2.1.1. Bloqueo y desbloqueo de procesos.

Los semáforos exigen que los procesos en espera no ocupen la CPU por lo que:

- Un proceso bloqueado no puede ejecutar instrucciones en la CPU.
- Un proceso en ejecución puede desbloquear otro proceso bloqueado.
- Pueden existir simultáneamente varios conjuntos de procesos bloqueados.

2.2. Estructura de un semáforo.

Un semáforo es una instancia de una estructura de datos que contiene:

- Un conjunto de procesos bloqueados.
- Un valor entero no negativo → valor del semáforo.

Estas estructuras de datos residen en memoria compartida. Al inicio de un programa que los usa debe poder inicializarse cada semáforo:

- El conjunto de procesos asociados estará vacío
- Se deberá indicar un valor inicial del semáforo

2.3. Operaciones sobre los semáforos.

sem_wait(semaphore): Reduce en una unidad el valor del semáforo, si el valor es 0, bloquea el proceso hasta que el valor sea mayor que 0.
sem_signal(semaphore): Incrementa el valor del semáforo en una unidad, si había procesos esperando, reanuda uno de ellos.

- El valor del semáforo nunca es negativo (no podemos hacer un wait cuando es 0)
- Sólo puede haber procesos esperando cuando el valor es 0.

2.3.1. Invariante de un semáforo.

Dado un semáforo **s**, en un instante de tiempo cualquiera **t**, su valor $\mathbf{v_t}$ será el valor inicial $\mathbf{v_0}$ más los sem_signal completados $\mathbf{n_s}$, menos los sem_wait completados $\mathbf{n_w}$. Nunca es negativo.

$$v_t = v_0 + n_s - n_w \geq 0$$

2.4. Uso de semáforos: patrones sencillos.

Cada patrón es un esquema que permite solucionar la sincronización de un problema típico sencillo, estos son:

- **Espera única:** un proceso, antes de ejecutar una sentencia, debe esperar a que otro proceso complete otra sentencia (leer variable escrita por otro proceso)
- **Exclusión mutua:** acceso en exclusión mutua a una sección crítica por parte de un número arbitrario de procesos.
- **Problema del Productor/Consumidor:** como la espera única, pero de forma repetida en un bucle (un proceso escribe sucesivos valores y cada uno de ellos debe ser leído una sola vez por otro proceso).

2.4.1. Limitaciones de los semáforos.

Los semáforos resuelven de una forma eficiente y sencilla el problema de la exclusión mutua y problemas sencillos de sincronización, pero:

- Problemas más complejos de sincronización se resuelven de forma más compleja.
- Programas erróneos o malintencionados pueden provocar interbloqueo.

3. Monitores como mecanismo de alto nivel.

3.1. Introducción. Definición de monitor.

Limitaciones de los semáforos.

- Basados en variables globales: impide diseño modular y reduce escalabilidad.
- Uso y función de las variables no se hace explícito, dificulta verificar los programas.
- Las operaciones se encuentran dispersas y no protegidas.

Estructura y funcionalidad de un monitor.

Un monitor es un mecanismo de alto nivel que permite definir objetos abstractos compartidos, incluyen una colección de variables encapsuladas (datos) que son compartidas y un conjunto de procedimientos para manipular el recurso.

- Garantiza el acceso en exclusión mutua a las variables encapsuladas
- La sincronización requerida se implementa mediante esperas bloqueadas.

Propiedades del monitor.
Un monitor es un recurso compartido que se usa como un objeto al que se accede concurrentemente.

- Acceso estructurado y encapsulación: usuario sólo puede acceder al recurso mediante determinadas operaciones e ignora las variables y la implementación.
- Exclusión mutua en el acceso a los procedimientos: EM garantizada por definición y nunca dos procesos ejecutarán simultáneamente algún procedimiento del monitor.

Ventajas sobre los semáforos.
Facilitan el diseño e implementación de programas libres de errores:

- Las variables están protegidas, sólo pueden leerse y modificarse desde el monitor.
- La exclusión mutua está garantizada.
- Las operaciones de esperas bloqueadas y de señalización se programan exclusivamente dentro del monitor.

Componentes de un monitor.

- **Variables permanentes:** estado interno del monitor. Sólo pueden accederse dentro del monitor, no se modifican entre dos llamadas consecutivas a procedimientos.
- **Procedimientos:** modifican el estado interno. Pueden tener variables y parámetros locales, algunos constituyen la interfaz externa del monitor.
- **Código de inicialización:** fija estado interno inicial. Se ejecuta una única vez y antes de cualquier llamada a procedimientos del monitor.

Variables permanentes y los procedimientos no exportados no son accesibles desde fuera.

3.2. Funcionamiento de los monitores.

Comunicación monitor-mundo exterior: Se necesita operar sobre un recurso compartido controlado por un monitor → proceso realiza llamada a uno de los procedimientos exportados por el monitor.

- Mientras ejecute algún procedimiento del monitor, el proceso está dentro del monitor.

Exclusión mutua: asegura que el acceso de las variables permanentes nunca sea concurrente.
Los monitores son objetos pasivos: cuando se ejecuta el código de inicialización, el código de sus procedimientos sólo se ejecuta cuando estos son invocados por los procesos.
Instanciación de monitores: cada instancia tiene sus propias variables, EM en cada instancia ocurre por separado, facilita mucho escribir código reentrante.

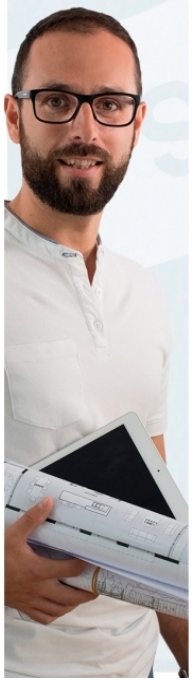
3.2.1. Cola del monitor para exclusión mutua.

El control de la EM se basa en la cola del monitor:

- Sigue una política FIFO
- Si un proceso está dentro del monitor y otro intenta acceder, este queda bloqueado y se mete en la cola
- Cuando un proceso sale del monitor, se desbloquea un proceso de la cola
- Si la cola está vacía y el monitor libre, el primer proceso que haga una llamada entra

3.3. Sincronización en monitores.

Las variables permanentes del monitor determinan si la condición se cumple o no.
Sólo se dispone de sentencias de bloqueo y activación.



3.3.1. Variables condición.

Para cada condición distinta que los procesos tengan que esperar se debe declarar una variable de tipo condition, se les llama señales o variables condición.

- Cada variable condición tiene asociada una cola de procesos en espera hasta que se haga cierta.
- Podemos usar wait y signal sobre una variable condición.:

cond.wait(): bloquea incondicionalmente e introduce el proceso en la cola de la variable c.
cond.signal(): si hay procesos bloqueados por esa condición, despierta uno, si hay varios sigue una política FIFO y si no hay no hace nada.
cond.queue(): devuelve true si hay algún proceso en la cola y false en caso contrario.

3.3.2. Esperas bloqueadas y EM en el monitor.

Más de un proceso puede estar dentro del monitor, aunque sólo uno de ellos estará ejecutándose, el resto estarán bloqueados en variables condición.
Cuando un proceso llama a wait se bloquea y libera la exclusión mutua del monitor, cuando es reactivado adquiere de nuevo la EM hasta ejecutar el siguiente wait.

3.4. Verificación de monitores.

El programador no puede conocer a priori la interfoliación concreta de llamadas a los procedimientos del monitor. El enfoque de verificación utiliza un invariante de monitor:

- Es una propiedad que el monitor cumple siempre que unido a las propiedades de los procesos concurrentes que invocan al monitor, facilita la verificación.
- Su valor depende de la traza del monitor (secuencia de llamadas a procedimientos ya completadas) y de los valores de las variables permanentes.
- Se puede evaluar como true o false en cada estado del monitor.

3.5. Patrones de solución con monitores.

Espera única: un proceso antes de ejecutar una sentencia debe esperar a que otro proceso complete otra sentencia. Se puede usar una variable compartida, el que va a necesitar que se complete la sentencia debe esperar, así aseguramos la interfoliación correcta.
Exclusión mutua: un número aleatorio de procesos acceden en EM a una sección crítica.
Problema del Productor/Consumidor: similar a la espera única pero en bucle.

3.6. Colas de prioridad.

Por defecto las colas de espera son FIFO, pero podemos cambiar esto dándole un valor de prioridad de proceso como un parámetro entero de wait. Sintaxis: cond.wait(p)

- cond.signal() reanudará un proceso con el valor mínimo de p, si hay más de uno se usa política FIFO.
- No afecta a la lógica del programa, el funcionamiento es similar con o sin prioridad.
- Sólo mejoran las características que dependen del tiempo.

3.7. Semántica de las señales de los monitores.

Un **proceso señalador** es aquel que hace signal sobre una cola no vacía.
Un **proceso señalado** es el que esperaba en la cola y se reactiva.
La **semántica de señales** es la política que establece la forma concreta en que se resuelve el conflicto tras hacerse un signal en una cola no vacía.

3.7.1. Señalar y Continuar (SC).

El proceso **señalador continua la ejecución tras el signal**. El **señalado espera bloqueado** hasta que puede adquirir EM de nuevo.

- El proceso **señalado** tiene que **volver a comprobar su condición** al terminar el wait, por eso la semántica obliga a programar la condición wait en un **bucle**.

3.7.2. Señalar y Salir (SS).

El **proceso señalado se reactiva inmediatamente** y el **señalador abandona el monitor** tras hacer signal **sin ejecutar el código después del signal**.

- El **estado** que permite al proceso señalado continuar la ejecución está **garantizado**.
- Esta semántica obliga a colocar siempre el signal como **última instrucción**.

3.7.3. Señalar y Esperar (SE).

El proceso **señalado se reactiva inmediatamente** y el **señalador queda bloqueado** en la cola del monitor junto a otros procesos que quieres comenzar a ejecutar código del monitor.

- El **estado** que permite al proceso señalado continuar la ejecución está **garantizado**.
- El proceso **señalador** vuelve a estar al **mismo nivel** que otros procesos en la cola.

3.7.4. Señalar y espera Urgente (SU).

El proceso **señalado se reactiva inmediatamente** y el **señalador queda bloqueado** en una cola específica con **mayor prioridad** que otros procesos en la cola normal.

- El **estado** que permite al proceso señalado continuar la ejecución está **garantizado**.
- El proceso **señalador** entra en una nueva cola llamada **cola de procesos urgentes**, los procesos de esta cola tienen preferencia frente a los de la cola del monitor.

3.7.5. Análisis comparativo de las diferentes semánticas.

Todas las semánticas son **capaces de resolver los mismos problemas**.
Facilidad de uso: la semántica **SS** condiciona el estilo de programación, puede llevar a aumentar el número de procedimientos.
Eficiencia: las semánticas **SE y SU** resultan ineficientes cuando no hay código tras signal ya que el señalador emplea tiempo en bloquearse y reactivarse para salir sin hacer nada. La semántica **SC** es también un poco ineficiente al obligar a usar un bucle para cada signal.

3.8. Implementación de monitores con semáforos.

Se puede implementar cualquier monitor usando sólo semáforos, asociando uno por cola:

- **Cola del monitor:** tipo mutex que vale 0 si se está ejecutando código y 1 si no.
- **Colas de variables condición:** para cada variable condición se define un semáforo que está siempre a 0 y un entero con el número de procesos esperando.
- **Cola de procesos urgentes:** semántica SU, debe haber un entero y un semáforo.

Los semáforos y monitores son equivalentes en potencia expresiva pero los monitores facilitan el desarrollo.

4. Soluciones software con espera ocupada para EM.

4.1. Propiedades para la EM.

Para que un algoritmo para EM sea correcto tiene que cumplir estas tres **propiedades mínimas**: **EM, progreso y espera limitada**. Además deben cumplirse: **eficacia y equidad**.

Exclusión Mutua:
Propiedad fundamental para la sección crítica. En cada instante de tiempo para cada SC existente habrá como mucho un proceso ejecutando en dicha región.

Propiedad de Progreso:
Un algoritmo de EM debe estar diseñado de forma que después de un tiempo finito desde que ingresó el primer proceso al protocolo de entrada, uno de los procesos en el mismo podrá acceder a la SC. Cuando esta condición no se da, se produce un interbloqueo (todos los procesos en el PE quedan en espera indefinidamente).

Espera Limitada:
Un algoritmo de EM debe estar diseñado de forma que las esperas en el PE siempre serán finitas.

Eficacia:
Protocolos E/S deben usar poco tiempo de procesamiento y las variables compartidas poca cantidad de memoria.

Equidad:
Cuando haya varios procesos queriendo entrar en una SC no se debería perjudicar a algunos para beneficiar a otros.

4.2. Refinamiento sucesivo de Dijkstra.

El **Refinamiento sucesivo de Dijkstra** hace referencia a una serie de algoritmos que intentan resolver el problema de la exclusión mutua.

- Se comienza desde una **versión muy simple, incorrecta** (no cumple alguna de las propiedades), y se hacen sucesivas mejoras para intentar cumplir las tres propiedades. Esto ilustra muy bien la importancia de dichas propiedades.
- La **versión final correcta** se denomina **Algoritmo de Dekker**.
- El **algoritmo de Peterson** es otro **algoritmo correcto para EM**, que además es más simple que el algoritmo de Dekker, pues a diferencia del de Dekker, el PE no usa dos bucles anidados, sino que los unifica en uno solo.

5. Soluciones hardware con espera ocupada (cerrojos) para E.M.

Los cerrojos constituyen una solución hardware basada en espera ocupada que puede usarse en procesos concurrentes con memoria compartida para el problema de la EM.

- Existe un valor lógico en una posición de memoria compartida (cerrojo) que indica si algún proceso está en la sección crítica o no.
- Los cerrojos consumen poca memoria y son eficientes en tiempo.

Podemos usar instrucciones máquinas atómicas para acceso a la zona de memoria donde se aloja el cerrojo, una de ellas es TestAndSet.
TestAndSet: lee el valor anterior del cerrojo, pone el cerrojo a true y devuelve el valor anterior del cerrojo. Se ejecuta de forma atómica.

5.1. Desventajas de los cerrojos.

Las esperas ocupadas consumen tiempo de CPU que podría dedicarse a otros procesos.
Se puede acceder directamente a los cerrojos, puede dejar a otros procesos en interbloqueo
Estas desventajas restringen el uso de los cerrojos:

- Por seguridad, sólo se usan desde componentes software del SO, librerías…
- Para evitar pérdida de eficiencia se usan sólo cuando la SC tiene un tiempo corto.