Resúmenes SCD

Manuel Bolaños Quesada
Carmen Azorín Martí

Índice

1. Tema 1: Introducción

1.1. Conceptos básicos y motivación

- **Programación secuencial.** Declaraciones de datos + conjunto de instrucciones ejecutables en secuencia.
- **Programa concurrente.** Conjunto de "programas" secuenciales que se pueden ejecutar lógicamente en paralelo.
- **Proceso.** Entidad software abstracta, dinámica, que ejecuta sus instrucciones y alcanza diferentes estados. Un proceso se caracteriza por una zona de memoria estructurada en varias secciones (figura de abajo), que incluye:
  - Secuencia de instrucciones que está ejecutando.
  - Espacio de datos de tamaño fijo ocupado por variables globales y estáticas.
  - La pila, que representa un espacio de tamaño variable ocupado por variables locales a procedimientos y sus parámetros.
  - Memoria dinámica o heap que contiene variables dinámicas que son creadas, asignadas y destruidas antes de la terminación del proceso.
- **Concurrencia.** Potencial para la ejecución paralela de código: entremezclamiento de instrucciones de varios programas.
- **Programación paralela.** Enfocada a acelerar la resolución de problemas de eficiencia de los cálculos, aprovechando la capacidad de ejecutar procesos paralelos del hardware.
- **Programación concurrente.** Conjunto de notaciones y técnicas de programación utilizadas para expresar el paralelismo potencial y resolver problemas de sincronización y comunicación $\implies$ + eficiencia y calidad.
- **Programación distribuida.** Enfocada a hacer que varios componentes software localizados en distintos ordenadores, trabajen juntos.
- **Programación de tiempo real.** Resuelve la programación de sistemas que funcionan continuamente, y que han de cumplir plazos estrictos de tiempo para realizar las tareas del programa.

La PC (Programación Concurrente) permite aprovechar mejor los recursos hardware existentes en nuestro ordenador:

- Si tiene un procesador $\implies$ nunca está ocioso
- Si tiene +1 procesador $\implies$ se reparte la ejecución de las tareas entre los distintos procesadores $\implies$ acelera cálculos

1.2. Modelo abstracto de la PC

La PC es una abstracción que nos ayuda a programar según un modelo "paralelo" (aunque realmente no depende de la implementación a bajo nivel). Los programas que desarrollaremos serán transportables ya que son independientes de una máquina concreta, que es uno de los objetivos que busca la concurrencia.

El modelo abstracto de la PC se define a partir de 5 hipótesis o axiomas:

I **Atomicidad y entrelazamiento** de las instrucciones

II **Coherencia** en el acceso concurrente a los datos

III **Irrepetibilidad** de la secuencia de instrucciones atómicas

IV **Independencia de la velocidad** de ejecución de los procesos

V Progreso de todos los procesos en **tiempo finito**

1.2.1. Atomicidad y entrelazamiento

- El *funcionamiento de una instrucción* es el efecto en el estado de ejecución del programa justo cuando esta acaba.
- El *estado de ejecución* de un programa concurrente está formado por los valores de las variables y de los registros de todos los procesos.

Se dice que una sentencia de un proceso en un programa concurrente es **atómica** si **siempre** se ejecuta de principio a fin **sin verse afectada** por las sentencias en ejecución de otros procesos. Es decir, cuando el funcionamiento de dicha instrucción no dependa nunca de cómo se estén ejecutando otras instrucciones.

La mayoría de sentencias de los lenguajes de alto nivel generalmente no son atómicas, aunque las instrucciones máquinas (LOAD, ADD, STORE,...), sí.

Ejemplo. Consideremos la instrucción $x := x + 1$.

El compilador utiliza una secuencia de tres sentencias $\Longrightarrow$

1. Leer el valor de x y almacenarlo en un registro r
2. Incrementar el valor del registro r
3. Escribir el valor del registro r en la variable x

Cada una de las instrucciones máquinas son atómicas, pero la instrucción total ($x := x + 1$), no, ya que entre la sentencia 1 y 2, otro proceso podría cambiar el valor del registro r, y así cambiar el valor final esperado. Decimos en este caso que **existe indeterminación**.

Supongamos ahora que tenemos un programa concurrente, C, compuesto por 2 procesos secuenciales: P_A y P_B, que se ejecutan a la vez, y que ejecutan las instrucciones atómicas $A_1, \ldots, A_5$ y $B_1, \ldots, B_5$, respectivamente, y en ese orden. Entonces, la ejecución de C puede dar lugar a cualquier combinación de instrucciones de los dos procesos, con la única condición de que las instrucciones de un proceso se ejecutan en orden. (no se puede ejecutar A_3 antes que A_1). Esto es el **entrelazamiento de instrucciones atómicas**.

Al conjunto de todas las secuencias posibles de entrelazamiento de instrucciones atómicas se le denomina **comportamiento**. No podemos saber cuál de estas secuencias se producirá finalmente cada vez ni tampoco influir para que una de ellas sea la que se produzca en particular. Esta característica fundamental se denomina **no determinismo** de la ejecución de los programas concurrentes.

1.2.2. Coherencia

Supongamos que tenemos un programa P que se compone de las dos instrucciones atómicas I_0 y I_1, que se ejecutan concurrentemente (esto se pone $P \equiv I_0 \| I_1$). Puede ser que las dos

instrucciones no accedan al mismo registro (en cuyo caso el orden de ejecución no afecta al resultado final), o que sí lo hagan. Veamos el segundo caso. Supongamos lo siguiente:

- $I_0 \equiv M \leftarrow 1$
- $I_1 \equiv M \leftarrow 2$

Lo razonable es que $M = 1$, o $M = 2$, pero nunca puede alcanzar otro valor. Esto es lo que se conoce como **coherencia**. La coherencia de los datos de las variables, tras un acceso concurrente a una misma dirección, es asegurada por el **controlador de memoria** de los computadores.

1.2.3. Irrepetibilidad

La **historia** o **traza** de un programa concurrente es la secuencia de estados del programa producida por una secuencia concreta de entrelazamiento ($s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_n$).

La probabilidad de que 2 trazas de ejecución seguidas del mismo programa coincidan es prácticamente 0 (a no ser que sea un programa con muy pocas instrucciones) $\Longrightarrow$ **irrepetibilidad**

1.2.4. Independencia de la velocidad

No se puede hacer ninguna suposición acerca de las velocidades de ejecución de los procesos. Un programa concurrente solo presta atención a sus procesos y las interacciones entre ellos, sin tener en cuenta el entorno de ejecución (velocidad de los procesadores, etc).

1.2.5. Tiempo finito

No puede existir ninguna configuración aceptable de un programa en el que un proceso podría detener arbitrariamente su ejecución durante un tiempo no definido en el tiempo de ejecución del programa concurrente.

- **Punto de vista local.** Cuando un proceso de un programa concurrente comienza la ejecución de una sentencia, completará su ejecución en un intervalo de tiempo finito.
- **Punto de vista global.** Si existe al menos 1 proceso preparado para ejecutarse, es decir, si el programa no ha entrado en interbloqueo, entonces en algún momento se permitirá la ejecución de algún proceso del programa.

1.3. Consideraciones sobre el hardware

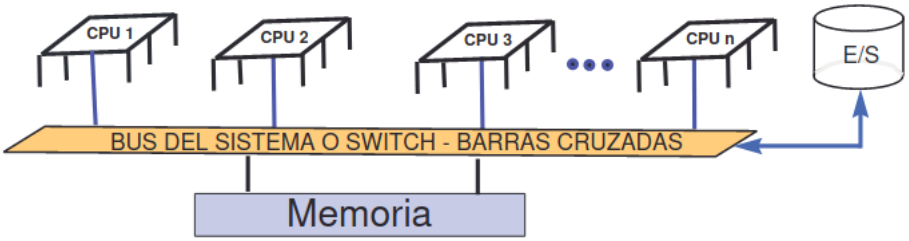
La implementación de la concurrencia en los programas depende de la arquitectura del procesador, pero el hardware y el tipo de paralelismo (monoprocesador, multiprocesador, sistema distribuido o multicomputador) nunca afectará a la corrección del programa.

1.3.1. Concurrencia en sistemas monoprocesador

- El SO gestiona el reparto de los ciclos del procesador entre los procesos
- Se aprovecha mejor la CPU
- Sincronización y comunicación mediante variables compartidas

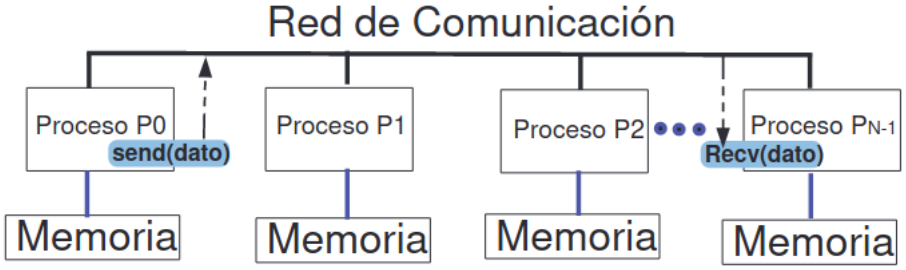
1.3.2. Concurrencia en multiprocesadores o sistemas multinúcleo

- Los procesadores pueden compartir o no físicamente la misma memoria, pero **comparten algún espacio de direcciones común**
- Los núcleos **comparten una memoria** de acceso muy rápido
- La interacción entre procesos se puede implementar con **variables compartidas**



1.3.3. Concurrencia en sistemas distribuidos

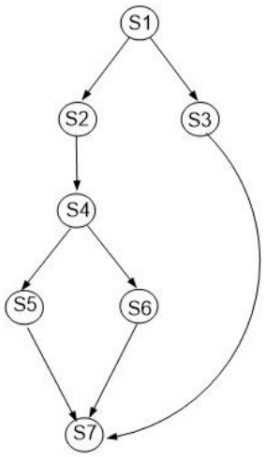
- **No existe memoria común**
- La interacción entre procesos es a través de una **red de interconexión**
- Hay que usar programación distribuida



1.4. Notaciones para PC

Existen sistemas **estáticos**, en los que el número de procesos se fija en el código fuente del programa, y sistemas **dinámicos**, en los que el número de procesos/hebras que se usan cambia durante la ejecución del programa.

El **grafo de sincronización** es un grado dirigido acíclico donde cada nodo representa una secuencia de sentencias del programa. Dadas dos actividades, S_1 y S_2, una arista desde S_1 hacia S_2 significa que S_2 no puede comenzar su ejecución hasta que S_1 no haya finalizado.



1.4.1. Definición estática (estructurada) de procesos: cobegin-coend

El par de instrucciones (`cobegin`,`coend`) de creación de procesos se considera una sentencia estructurada porque tiene un punto de comienzo (`cobegin`) y uno de finalización (`coend`), que supone la terminación de todas las instrucciones del bloque antes de ejecutarse `coend`.

Se lanza la ejecución concurrente de ambos procesos en la última linea. El programa acaba cuando acaban de ejecutarse todas las instrucciones de los dos procesos. Las variables compartidas se inicializan antes de comenzar la ejecución concurrente de los procesos.

También se pueden crear vectores de procesos.

```
var ... \\variables compartidas

process Uno;
var ... \\variables locales
begin
  ... \\codigo
end;
process Dos;
var ... \\variables locales
begin
  ... \\codigo
end;
... \\otros procesos
cobegin  Uno || Dos coend;
```

(a) Definición estática de procesos

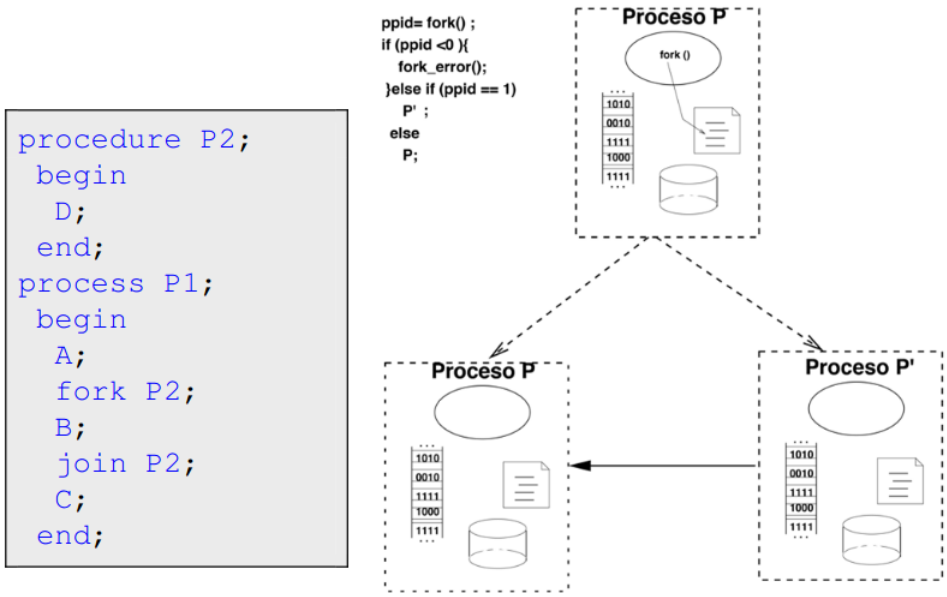
```
var ... \\variables compartidas

process NomP[ind : a.. b];
var ... \\variables locales
begin
  ... \\codigo
  ...\\ (ind vale a, a+1, ... b)
end;
... \\otros procesos
```

(b) Definición estática de vector de procesos

1.4.2. Definición no estructurada de procesos: fork-join

- **fork:** sentencia que especifica que la rutina nombrada puede comenzar su ejecución, al mismo tiempo comienza la sentencia siguiente
- **join:** sentencia que espera la terminación de la rutina nombrada, antes de comenzar la sentencia siguiente



En la figura anterior puede verse que la llamada `fork()` produce una segunda copia del programa (P') totalmente independiente del proceso P (programa que realiza la llamada) que continuará ejecutándose.
La instrucción `join()` ejecutada en el texto de un proceso P permite a este esperar a que otro proceso P' termine su ejecución, antes de proseguir con la siguiente instrucción. P es el proceso que invoca la operación de unión de flujos y P' es el proceso objetivo.

La ventaja de utilizar el mecanimso `fork/join` frente a programar un bucle de espera ocupada en el proceso P hasta que P' haya terminado, consiste en que el proceso que hace la llamada a `join()` se suspenderá y no consumirá ciclos de procesador hasta que P' haya terminado.

1.5. Exclusión mutua y sincronización

No todas las secuencias de entrelazamiento son posibles para llegar al resultado que queremos (los procesos no son independientes, colaboran entre ellos). Por tanto, hay que imponer condiciones de **sincronización** (restricciones en el orden en el que se puede entremezclar las instrucciones), y **exclusión mutua** (secciones de código que se ejecutan de principio a fin por un único proceso).

Al conjunto de secuencias comunes de instrucciones consecutivas que aparecen en el texto de varios procesos se le llama **sección crítica** (SC). Hay exclusión mutua cuando en cada instante hay, como mucho, un proceso ejecutando cualquier instrucción de la SC.

Ejemplo. Si x es una variable entera en memoria compartida, la sección crítica estará formada por todas las secuencias de instrucciones que lean o escriban la variable (como $x := x + 1$). Si varios procesos ejecutan estas instrucciones de forma simultánea, se produce una **condición de carrera** $\Longrightarrow$ resultados indeterminados (el valor final de x depende de la secuencia de instrucciones). Un ejemplo de intedeterminación sería el siguiente:
Como vimos antes, la instrucción $x := x + 1$ se compone de tres instrucciones máquina (que son atómicas):

1. LOAD $r_i \leftarrow x$
2. ADD $r_i, 1$
3. STORE $r_i \rightarrow x$

Supongamos que tenemos dos procesos, P_0 y P_1, que ejecutan a la vez esa instrucción. Cada proceso tiene su registro (r_0, r_1). Estos son dos entrelazamientos posibles:

P_0	P_1	x	P_0	P_1	x
load r0←x		0	load r0←x		0
add r0, 1		0		load r1 ←x	0
store r0→x		1	add r0, 1		0
	load r1←x	1		add r1, 1	0
	add r1, 1	1	store r0→x		1
	store r1→x	2		store r1→x	1

Con el primero, el valor final de x es 2, y con el segundo, el valor final de x es 1.

En el código, se puede forzar que una instrucción se ejecute de forma atómica, rodeándola con $<$ y $>$.

```
begin
x := 0 ;
cobegin
x:= x+1 ;
x:= x-1 ;
coend
end
```

(a) Definición de instr. no atómicas

```
begin
x := 0 ;
cobegin
< x:= x+1 > ;
< x:= x-1 > ;
coend
end
```

(b) Definición de instr. atómicas

En el primer código, x puede acabar con cualquier valor del conjunto $\{-1, 0, 1\}$, mientras que en el segundo, solo puede pasar que $x = 0$.

En un programa concurrente, una **condición de sincronización** sirve para asegurar que todos los posibles entrelazamientos son correctos. Por ejemplo, que en un punto concreto de la ejecución, uno o varios procesos deban esperar a que se cumpla una determinada condición.

El **paradigma del Productor-Consumidor**: se trata de dos procesos cooperantes, de forma que uno de ellos genera una secuencia de valores (productor), y el otro los utiliza (consumidor).

Abreviemos diciendo que E es una sentencia elemental de escritura (como STORE), y L, una de lectura (como LOAD). Entonces, los procesos sólo funcionan como se espera si el orden en el que se entrelazan estas sentencias es: $E, L, E, L, E, L, \ldots$

Esta secuencia asegura la condición de sincronización:

- El consumidor no lee hasta que el productor escribe un nuevo valor en x
- El productor no escribe un nuevo valor hasta que el valor anterior no se ha leído (así no se pierde ningún valor)

Ejemplo. Trazas incorrectas en el programa Productor-Consumidor:

- $L, E, L, E, \ldots$: se hace una lectura previa a cualquier escrita, así que se lee un valor indeterminado
- $E, L, E, E, L, \ldots$: hay dos escrituras sin ninguna lectura entre ellas, así que hay un valor que no se lee, y se pierde
- $E, L, L, E, L, \ldots$: hay dos lecturas de un mismo valor, así que se usa dos veces.

1.6. Propiedades de los sistemas concurrentes

En un programa concurrente, una **propiedad** es algo se puede afirmar del programa que es cierto para **todas** las trazas del programa.

Hay dos tipos de propiedades: propiedad de **seguridad**, y propiedad de **vivacidad**

1.6.1. Propiedad de seguridad

Son condiciones que deben cumplirse en cada instante de la traza del programa. Son del tipo: *nunca pasará nada malo.*

- Son especificaciones estáticas del programa
- Son fáciles de demostrar
- Se suele impedir que se den determinadas trazas

Ejemplo.

- 2 procesos no pueden entrelazar ciertas subsecuencias de operaciones.
- Ausencia de Interbloqueo: nunca va a ocurrir que un proceso se quede esperando a algo que no es posible. Por ejemplo, nunca va a pasar que un proceso se quede esperando al momento en el que $1 = -1$.
- Propiedad de seguridad en Productor-Consumidor. El consumidor debe consumir todos los datos producidos por el productor en el orden en el que se van produciendo $(E, L, E, L, \ldots)$.

1.6.2. Propiedades de vivacidad

Son propiedades que deben cumplirse en algún momento futuro y no especificado del programa. Son del tipo: *realmente sucede algo bueno.*

- Son propiedades dinámicas
- Más difíciles de demostrar

Ejemplo.

- Ausencia de inanición: un proceso no puede ser indefinidamente pospuesto. Es decir, en algún momento podrá avanzar
- Equidad: un proceso que pueda avanzar debe hacerlo con justicia relativa con respecto a los demás procesos del programa (que avancen todos a velocidades más o menos equitativa).

Está más ligado a la implementación.

1.7. Verificación de programas concurrentes

La definición más general de corrección del software es la siguiente: un sistema es correcto si satisface las propiedades previamente especificadas.

Para la demostración de que un código es correcto, es decir, para llevar a cabo la verificación del software, se pueden emplear diferentes métodos:

- **Depuración.** Ejecutar diferentes ejecuciones del programa y comprobar que se verifican las propiedades. En realidad, este método no sirve para la verificación del software porque no puede demostrar la ausencia de errores transitorios en un programa concurrente.
- **Razonamiento operacional.** Es un análisis de casos exhaustivo, es decir, se exploran todas las posibles secuencias de ejecución de un código.
- **Razonamiento asertivo.** Es un análisis abstracto basado en la Lógica Matemática que permite obtener una representación abstracta de los estados concretos que un programa va alcanzando durante su ejecución.

1.7.1. Enfoque operacional: análisis exhaustivo

Se hace un análisis exhaustivo de los casos: se comprueba la corrección de todas las posibles trazas.

Esto, para programas concurrentes en los que hay pocas instrucciones, está bien, pero para programas más grandes es más difícil, ya que el número de entrelazamientos crece exponencialmente con el número de instrucciones.

Por ejemplo, para un programa formado por 2 procesos y 3 sentencias atómicas por proceso, habría que estudiar 20 historias diferentes.

1.7.2. Enfoque axiomático

Se define un sistema lógico formal (SLF) que permite establecer propiedades de programas en base a axiomas y reglas de inferencia.

- Se usan fórmulas lógicas (*asertos* o *predicados*) para caracterizar un conjunto de estados (los aceptables). Es decir, un aserto $\{P\}$ caracteriza un estado aceptable, esto es, un estado que podría ser alcanzado por el programa.

  El aserto $\{V\}$ caracteriza a todos los estados del programa (precondición más débil), y el aserto $\{F\}$ no se cumple por parte de ningún estado del programa (precondición más fuerte).
- Las sentencias atómicas actúan como transformadoras de los asertos. Los **teoremas** (o triples) se escriben en la forma:

$$\{P\}S\{Q\} \qquad (\textbf{Notación de Hoare})$$

- Menor complejidad que el enfoque operacional. El trabajo que conlleva esta prueba de corrección es proporcional al número de sentencias atómicas.

Se dice que un SLF es:

- seguro (o que verifica la *propiedad de seguridad*) si se cumple que {asertos} $\subset$ {hechos ciertos}
- completo (o que verifica la *propiedad de complección*) si se cumple que {hechos ciertos} $\subset$ {asertos}

Los SLF para demostración de programas no suelen poseer la propiedad de complección.

1.7.3. Significado de la notación de Hoare

$\{P\}C\{Q\}$ es cierto si siempre que C es ejecutado en un estado en el que P es cierto, y si la ejecución de C termina, entonces el estado en que C termina satisface Q.

Ejemplo.

- $\{X == 1\}\ X = X + 1\ \{X == 2\} \longrightarrow$ cierto
- $\{X == 1\}\ X = X + 1\ \{X == 3\} \longrightarrow$ falso
- $\{X == 1\}$ WHILE T DO NULL $\{Y == 3\} \longrightarrow$ cierto (WHILE T DO NULL no acaba)

1.7.4. Estructura de las demostraciones

Una **demostración** es una secuencia de líneas (o triples) de la forma $\{P\}C\{Q\}$, cada una de las cuales es un axioma o deriva de los anteriores aplicando una regla de inferencia de la lógica.

1.7.5. Fórmulas proposicionales válidas

- Leyes distributivas:
  - $P \vee (Q \wedge R) = (R \vee Q) \wedge (P \vee R)$
  - $P \wedge (Q \vee R) = (R \wedge Q) \vee (P \wedge R)$
- Leyes de De Morgan:
  - $\neg(P \wedge Q) = \neg P \vee \neg Q$
  - $\neg(P \vee Q) = \neg P \wedge \neg Q$
- Elminación-AND: $(P \wedge Q) \to P$
- Elminación-OR: $P \to (P \vee Q)$

1.7.6. Axiomas y reglas de inferencia

Cada línea de la demostración es un axioma, o se deriva de la línea anterior mediante una regla de inferencia.

- **Axioma**: fórmulas que sabemos que son ciertas en cualquier estado del programa
- **Regla de inferencia**: regla para derivar fórmulas ciertas a partir de otras que se han demostrado ser ciertas

Notaremos una regla de inferencia de la siguiente forma:

$$(\text{nombre de la regla})\frac{H_1, H_2, \ldots, H_n}{C}$$

donde $H_1, H_2, \ldots, H_n$ son hipótesis, y C es la consecuencia.

Además, $\{P_e^x\}$ es el resultado de sustituir la expresión e en cualquier aparición de la variable x en P.

Veamos los axiomas con los que vamos a trabajar:

1. **Axioma de la sentencia nula**: $\{P\}$NULL$\{P\}$: si el aserto es cierto antes de ejecutarse esa sentencia, también lo será cuando la sentencia acabe
2. **Axioma de asignación:** $\{P_e^x\}x = e\{P\}$: una asignación cambia solo el valor de la variable objetivo (x), el resto de las variables conservan los mismos valores

Ejemplo.

- $\{V\}x = 5\{x == 5\}$ es un aserto, ya que si ponemos $\{x == 5\} \equiv \{P\}$, entonces $P_5^x \equiv \{x == 5\} \equiv \{5 == 5\} = \{V\}$.
- $\{x > 0\}x = x + 1\{x > 1\}$ es un aserto, ya que si ponemos $\{x > 1\} \equiv \{P\}$, entonces

$$P^x_{x+1} \equiv \{x > 1\}^x_{x+1} \equiv \{x + 1 > 1\} \equiv \{x > 0\}.$$

Veamos ahora reglas de inferencia:

1. **Regla de la consecuencia (1)**: $\dfrac{\{P\}S\{Q\}, \{Q\} \to \{R\}}{\{P\}S\{R\}}$
2. **Regla de la consecuencia (2)**: $\dfrac{\{R\} \to \{P\}, \{P\}S\{Q\}}{\{R\}S\{Q\}}$
3. **Regla de la composición**: $\dfrac{\{P\}S_1\{Q\}, \{Q\}S_2\{R\}}{\{P\}S_1; S_2\{R\}}$
4. **Regla del IF**: $\dfrac{\{P\} \wedge \{B\}S_1\{Q\}, \{P \wedge \neg B\}S_2\{Q\}}{\{P\} \text{ if } B \text{ then } S_1 \text{ else } S_2 \text{ fi } \{Q\}}$
5. **Regla de la iteración**: $\dfrac{\{I \wedge B\}S\{I\}}{\{I\} \text{ while } B \text{ do } S \text{ enddo } \{I \wedge \neg B\}}$

1.7.7. Verificación de sentencias concurrentes

En la verificación de los programas concurrentes se produce el problema concoido como de la **interferencia**, que cuando ocurre invalida las demostraciones individuales de los procesos. Se debe a que un proceso puede ejecutar una instrucción atómica que haga falsa la precondición (o poscondición) de una sentencia de otro proceso mientras está siendo ejecutada concurrentemente en el primero.

Supongamos que $\{P\}S\{Q\}$ es un triple demostrable. La sentencia T no interfiere con la demostración del triple anterior, si:

1. El triple $\{\text{pre}(T) \wedge P\}T\{P\}$ es demostrable
2. El triple $\{\text{pre}(T) \wedge Q\}T\{Q\}$ es demostrable

Ejemplo. El siguiente programa es un ejemplo de interferencia entre procesos de un programa concurrente.

```
y=0; z=0;
cobegin
 x= y+z || y=1; z=2
coend;
```

Recordatorio `x:= y + z` ≡ `load y; add z; store x`.

La ejecución de este programa puede producir como resultado final $x \in \{0, 1, 2, 3\}$, dependiendo de cuando se cargue `y` y se incremente `z` en el registro.

Para evitar las interferencias y las condiciones de carrera, se suelen usar secciones críticas y sincronización entre los procesos utilizando condiciones.

Además, se verifica la siguiente **Regla de la Concurrencia:**

$$\frac{\{P_i\}S_i\{Q_i\} \text{ son teoremas libres de interferencia, } 1 \leq i \leq n}{\{P_1 \wedge P_2 \wedge \cdots \wedge P_n\} \texttt{ cobegin } S_1||S_2||\cdots||S_n \texttt{ coend } \{Q_1 \wedge Q_2 \wedge \cdots \wedge Q_n\}}$$

2. Tema 2: Sincronización en memoria compartida

Hay varios inconvenientes en usar mecanismos como los semáforos. Por ejemplo, las llamadas a las funciones necesarias para utilizarlos (`sem_wait`, `sem_signal`) quedan repartidas en el código del programa, haciendo difícil corregir errores y asegurar el buen funcionamiento de los algoritmos. Además, están basados en variables globales, algo que normalmente queremos evitar.

Por tanto, es necesario un nuevo mecanismo de programación que permita la encapsulación de la información y de la sincronización entre procesos: los monitores.

2.1. Definición de monitor

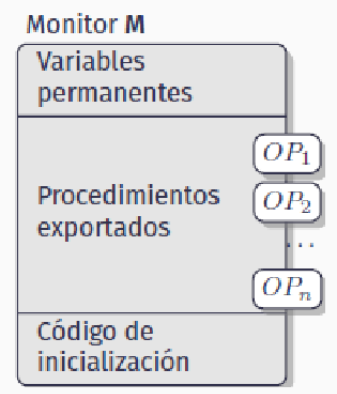
Un monitor es un mecanismo de programación de alto nivel que permite definir objetos abstractos compartidos entre los procesos concurrentes. Las variables permanentes y los procedimientos de los monitores garantizan acceso en **exclusión mutua** y **encapsulación** de su sincronización.

Sus principales características son:

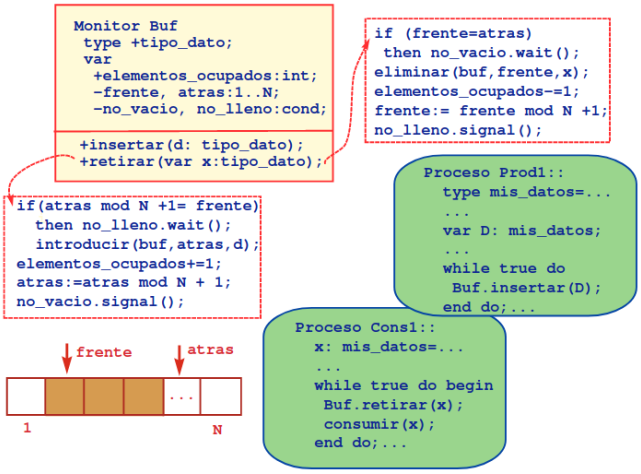
- **Modularidad** en el desarrollo de programas
- Programa = {Monitores, Procesos}
- **Estructuración** en el acceso a tipos de datos, variables compartidas, etc.
- **Capacidad de modelado** de interacciones cooperativas y competitivas entre procesos concurrentes
- **Ocultación** a los procesos de las operaciones de sincronización sobre datos compartidos
- **Reusabilidad** basada en parametrización de los módulos monitor
- **Verificación** mediante reglas más simples que las de los semáforos

Los componentes de un monitor son los siguientes:

- **Variables permanentes**: son el estado interno del monitor
- **Procedimientos**: modifican el estado interno (garantizando la exclusión mutua durante dicho cambio)
- **Código de inicialización**: fija el estado interno inicial



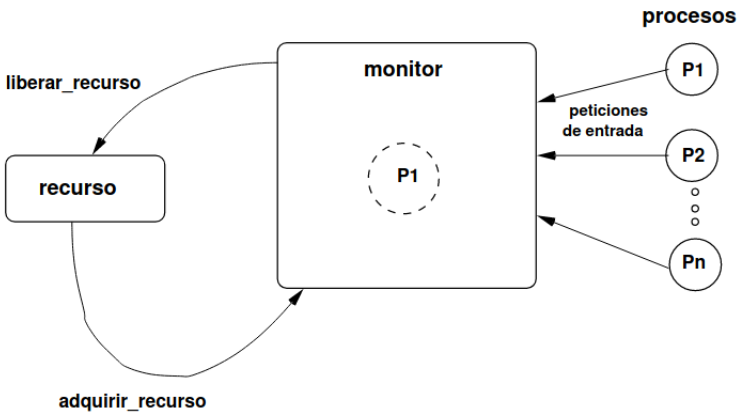
Un esbozo de cómo sería un programa con el paradigma Productor-Consumidor es el siguiente:



2.2. Funcionamiento de los monitores

Las principales características de la programación con monitores son:

- **Centralización de recursos críticos**
- Solo hay 1 procedimiento ejecutado por **un solo proceso**
- Los procedimientos pueden **interrumpirse**
- Posibilidad de **ejecución concurrente de monitores** que no estén relacionados



Cada instancia de un monitor tiene sus **variables permanentes propias**. Además, la exclusión mutua ocurre en cada instancia por separado. El control de la exclusión mutua se basa en la existencia de una cola **FIFO** de entrada al monitor, de forma que si un proceso ejecuta una llamada a uno de los procedimientos del monitor, entrará en esa cola.

La condición para que un lenguaje de programación pueda compilar monitores instanciables es que el código de los procedimientos de los monitores ha de ser reentrante (puede ser utilizado concurrentemente por varios threads sin interferencias entre ellos).

```
class monitor VariableProtegida(entr,salid : integer);
var x, inc : integer;
// incremento(), valor(); son los procedimientos llamables
     por los procesos
procedure incremento( );
begin
  x := x+inc ;
end;
procedure valor(var v : integer);
begin
  v := x ;
end;
begin
x:= entr ; inc := salid ;
end
var mv1 : VariableProtegida(0,1);
mv2 : VariableProtegida(10,4);
i1,i2 : integer ;
begin
mv1.incremento() ;
mv1.valor(i1) ; { i1==1 } //permanentes (x,inc) distintas
mv2.incremento() ; //para cada instancia del monitor
mv2.valor(i2) ; { i2==14 }
end
```

Figura 3: Instanciación de un monitor

Los siguientes diagramas representan los estados de un proceso, y el de un monitor:

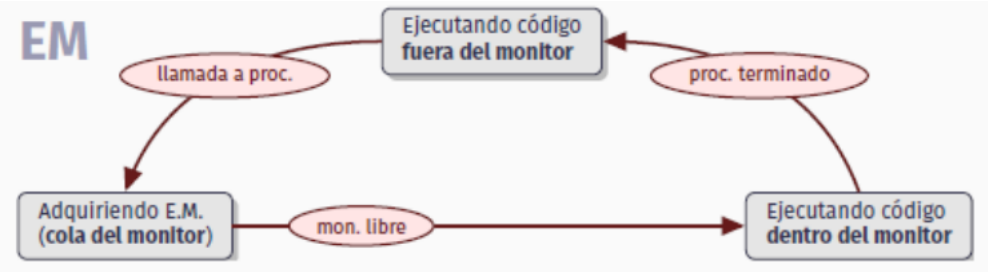


Figura 4: Estados de un proceso

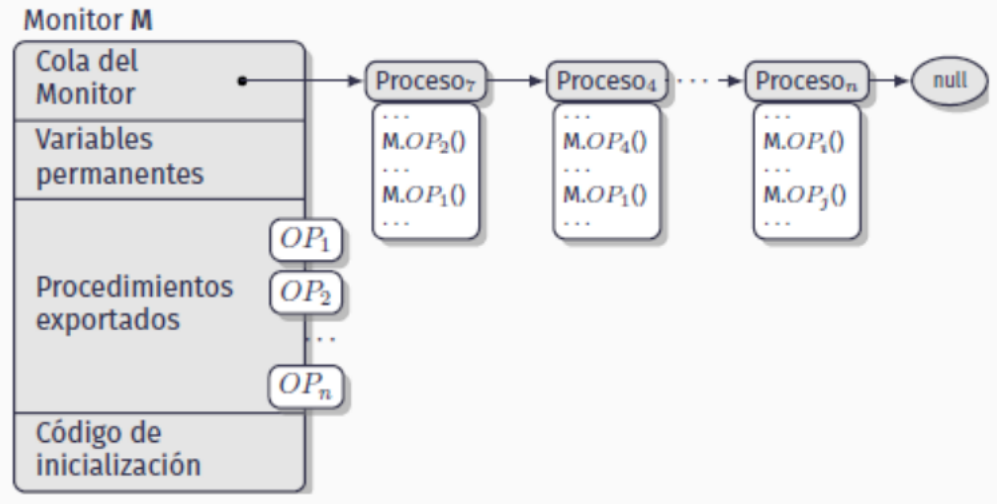


Figura 5: Estado de un monitor

2.3. Sincronización entre monitores

Para implementar la sincronización, necesitamos alguna funcionalidad para que los procesos hagan esperas bloqueadas ("se suspendan") hasta que se cumpla una determinada condición.

En el caso de los semáforos, tenemos la posibilidad de bloquear un proceso (`sem_wait`) y activarlo (`sem_signal`). Además, usamos un valor entero (el valor del semáforo). En los monitores, sin embargo, solo disponemos de **sentencias de bloqueo** y **activación**.

La sincronización debe ser explícitamente programada dentro de los procedimiento del monitor, utilizándose para ello un nuevo tipo de dato (`cond`) denominado *variable-condición*. Se definen dos operaciones fundamentales, que son:

- `wait()`: bloquea los procesos hasta que se cumple la condición. El desbloqueo es FIFO.
- `signal()`: reanuda la ejecución de un proceso cuando se cumple la condición.

También se define otra operación para comprobar si hay procesos bloqueados esperando a que la condición se haga cierta: `queue()`.

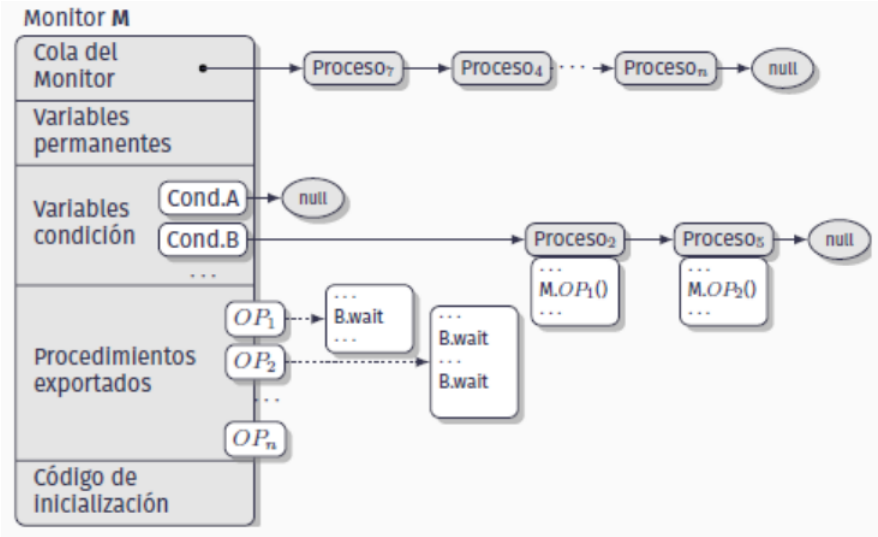


Figura 6: Los procesos 2 y 5 ejecutan las operaciones 1 y 2, que las dos producen esperas de la condición B

2.4. Verificación de programas con monitores

A priori, el programador no puede conocer la traza concreta de llamadas a los procedimientos del monitor. Por ello, vamos a buscar un **Invariante de un Monitor** (IM), que involucrará los valores permitidos de las variables permanentes del monitor. Esto es una propiedad que el monitor cumple siempre, pero que es específica para cada monitor diseñado por un programador.

El **esquema general** para la demostración de programas que usan monitores es:

1. Probar la **corrección parcial** de los procesos secuenciales
2. Comprobar que los **invariantes** de los monitores se mantienen durante toda la ejecución del programa
3. Aplicar la **regla de concurrencia** para demostrar que los procesos secuenciales cooperan sin interferirse

2.4.1. Axioma de inicialización de las variables del monitor

El código de inicialización de las variables del monitor, se denomina **cuerpo** del monitor, y debe incluir la asignación de todas las variables permanentes del monitor, antes de que los procedimientos sean llamados por primera vez por los procesos. Tras ejecutarse el cuerpo, se debe establecer el IM:

$$\{\ \texttt{V}\ \}\ \texttt{inicialización();}\ \{\ \texttt{IM}\ \}$$

2.4.2. Axioma de los procedimientos del monitor

El invariante del monitor debe ser cierto antes y después de cada llamada a un procedimiento del monitor. Además, se deben cumplir las precondiciones y poscondiciones propias de cada procedimiento. Es decir,

$$\{\ \texttt{PRE} \wedge \texttt{IM}\ \}\ \texttt{procedimiento();}\ \{\ \texttt{POS} \wedge \texttt{IM}\ \}$$

2.4.3. Axioma de la operación `c.wait()` para señales desplazantes

Supongamos que tenemos una variable-condición, `c`. Es una buena práctica asociarle un predicado, `C`, que represente la condición de desbloqueo de los procesos suspendidos (que están en la cola).

Ejemplo. Queremos programar un programa productor-consumidor, utilizando monitores. Entonces, la condición de desbloqueo para los procesos productores es: {`C: n` $\neq$ `N`} (el búfer no está lleno), y la condición de desbloqueo para los procesos consumidores es: {`C: n` $\neq$ `0`} (el búfer no está vacío).

Además, un procedimiento tendrá su propio invariante, definido a partir de las variables locales que use. Llamemos a este invariante `L`. Entonces, se tiene que cumplir:

$$\{\texttt{IM} \wedge \texttt{L}\}\ \texttt{c.wait()}\ \{\texttt{C} \wedge \texttt{L}\}$$

2.4.4. Axioma de la operación `c.signal()` para señales desplazantes

La operación `c.signal()` ha de producir la reanudación inmediata de un proceso que está esperando bloqueado en la cola de variable `c`, según orden FIFO. Para que esto ocurra, la condición `C` se debe cumplir. Además, la cola de procesos de `c` no puede ser vacía. Por último, cualquier condición definida a partir de las variables del monitor, que sea cierta antes de ejecutar `c.signal()`, seguirá siendo cierta después. Esto lo resumimos de la siguiente forma:

$$\{\neg\texttt{vacio(c)} \wedge \texttt{L} \wedge \texttt{C}\}\ \texttt{c.signal()}\ \{\texttt{IM} \wedge \texttt{L}\}$$

2.4.5. Regla de la concurrencia

Supongamos que tenemos las hipótesis:

H1 $\{P_i\}S_i\{Q_i\},\ 1 \leq i \leq n$

H2 ninguna variable libre en P_i o en Q_i es modificada por S_j, $i \neq j$

H3 todas las variables en IM_k son locales al monitor m_k

Entonces se verifica lo siguiente:

$$\frac{H_1, H_2, H_3}{\{IM_1 \wedge \cdots \wedge IM_n \wedge P_1 \wedge \cdots \wedge P_n\}\ \texttt{cobegin}\ S_i \mid\mid S_2 \mid\mid \cdots \mid\mid S_n\ \texttt{coend}\ \{IM_1 \wedge \cdots \wedge IM_n \wedge Q_1 \wedge \cdots \wedge Q_N\}}$$

2.5. Patrones de solución con monitores

Vamos a estudiar los patrones de solución para tres problemas sencillos:

- Espera única (EU)
- Exclusión mutua (EM)
- Problema del Productor-Consumidor (PC)

2.5.1. Espera única

Un proceso, antes de ejecutar una sentencia, debe esperar a que otro proceso complete otra sentencia (ocurre típicamente cuando un proceso debe leer una variable escrita por otro proceso, por ejemplo, el paradigma productor-consumidor).

El monitor podría implementarse y usarse de la siguiente forma (para un modelo de productor-consumidor, escribiendo y leyendo una sola variable):

```
monitor EU;
var terminado:boolean;
    //terminado==true si se ha terminado la sentencia E,
        si no: terminado==false
    leer_autorizado: boolean //variable auxiliar
    cola:condition;
    //cola consumidor esperando a que terminado==true
    export esperar, notificar;
  //Invariante: terminado= false => lee= false
  procedure esperar();//para llamar antes de sentencia  L
  begin
   if (not terminado) then //si no se ha terminado W
       cola.wait();//esperar hasta que termine
   //Condición de sincronización terminado= true
   leer_autorizado:= true;
  end;
  procedure termina();
  begin
    leer_autorizado:= false; terminado:= false;
  end;
  procedure notificar();//para llamar después de E
  begin
    terminado:=true;//Condición de sincronización
    cola.signal();//reactivar al otro proceso, si esa
        esperando
  end
```

(a) Implementación del monitor EU

```
//variables compartidas
var x: integer; //contiene cada valor producido
process Productor;//escribe x
  var a:integer ;
  begin
   a:=ProducirValor();
   x:=a; //sentencia E
   EU.notificar();//sentencia N
process Consumidor//lee x
  var b:integer;
  begin
  EU.esperar();//sentencia W
  b:=x;//sentencia L
  EU.termina();
  UsarValor(b) ;
end
```

(b) Uso del monitor EU en Productor-Consumidor

Verifiquemos la corrección del monitor EU. El invariante del monitor en este caso es

$$\text{terminado} = \text{false} \implies \text{leer\_autorizado} = \text{false}$$

```
procedure esperar();
{Invariante, terminado= false, leer_autorizado= false}
if (not terminado) then
  begin
    {terminado= false, leer_autorizado= false, Invariante}
    cola.wait();
  endif;
  {Condicion de sincronizacion: terminado= true}
  leer_autorizado:= true
  {Invariante}
end;
```

(a) Corrección proceso `esperar()`

```
procedure notificar();
begin
  {terminado= false, leer_autorizado=false, Invariante}
  terminado:= true;
  {Condicion de sincronizacion: terminado= true}
  cola.signal();
  {Invariante}
end;
```

(b) Corrección proceso `notificar()`

En estas imágenes se comprueba la corrección:

- Proceso `esperar()`: al inicio del proceso, se cumple el aserto {IM $\wedge$ terminado = false $\wedge$ leer_autorizado = false}, que también será cierto antes de ejecutar `cola.wait()`. Tras ejecutar esta sentencia, la condición de sincronización, `C`, será true (en este caso, `C`: {terminado = true}). Después de esto, se ejecuta `leer_autorizado = true`, y el invariante del monitor se mantiene cierto.

  Por tanto, se verifica el axioma de la operación `c.wait()` (2.4.3),

- Proceso `notificar()`: al inicio del proceso, se cumple el aserto {IM $\wedge$ terminado = false $\wedge$ leer_autorizado = false}. Además, la cola de los procesos esperando a ejecutar es no vacía (hay uno esperando). Antes de ejecutar `cola.signal()` hacemos que la condición de sincronización sea true. Tras ejecutar esto, el invariante del monitor sigue siendo cierto.

  También se verifica el axioma de la operación `c.signal()` (2.4.4)

2.5.2. Exclusión mutua

Queremos acceso en exclusión mutua a una sección crítica por parte de un número arbitrario de procesos.

El monitor que hace esto puede implementarse y usarse de la siguiente forma:

```
monitor EM ;
  var ocupada:boolean;
      //ocupada==true hay un proceso en SC, si no: ocupada
          ==false
      cola:condition;
      //cola de procesos esperando a que ocupada==false
      export entrar,salir;
      //nombra procedimientos públicos
  procedure entrar();//protocolo de entrada (sentencia E)
   begin
    if ocupada then//si hay un proceso en la SC
        cola.wait();//esperar hasta que termine
    ocupada:=true;//indicar que la SC está ocupada
   end
  procedure salir();//protocolo de salida (sentencia S)
   begin
    ocupada := false;//marcar la SC como libre
    //si al menos un proceso espera, reactivar al primero
    cola.signal();
   end
  begin//inicializacion:
   ocupada:=false;//al inicio no hay procesos en SC
  end
```

(a) Implementación del monitor EM

```
process Usuario[ i : 0..n ]
  begin
   while true do begin
    EM.entrar();//esperar SC libre, registrar SC ocupada
    ..... //sección crítica (SC)
    EM.salir(); //registrar SC libre, señalar
    ..... //otras actividades (RS)
  end
end
```

(b) Uso del monitor EM con varios procesos

Si denotamos por num_sc el número de procesos que están ejecutando la sección crítica, el invariante del monitor es:

$$(\text{ocupada} == \text{false} \iff \text{num\_sc} == 0) \wedge (0 \leq \text{num\_sc} \leq 1)$$

Es fácil ver que el IM es cierto durante todo el programa.

2.5.3. Productor-Consumidor

Es similar a la espera única, pero de forma repetida en bucle: un proceso productor escribe sucesivos valores en una variable, y cada uno de ellos debe ser leído una única vez por otro proceso consumidor.

El monitor se puede implementar y usar en un programa de la siguiente forma:

```
Monitor PC ;
 var valor_com:integer;//valor compartido
     pendiente:boolean;//true: valor escrito y no leido
     cola_prod:condition;
     //espera productor hasta que pendiente == false
     cola_cons:condition;
     //espera consumidor hasta que pendiente == true
  procedure escribir( v:integer );//sentencia E
   begin
    if pendiente then
      cola_prod.wait();
      valor_com:=v;
      pendiente:=true;
      cola_cons.signal();
   end;
  function leer():integer;//sentencia L
   begin
   if (not pendiente) then
     cola_cons.wait();
     result:=valor_com;
     pendiente:=false;
     cola_prod.signal();
   end;
   begin // inicializacion }
     pendiente := false ;
   end;
```

(a) Implementación del monitor PC

```
process Productor;//calcula x
  var a:integer;
  begin
   while true do begin
    a:=ProducirValor();
    PC.escribir(a);//copia a en valor
   end
end
process Consumidor //lee x
  var b : integer ;
  begin
   while true do begin
    PC.leer(b); //copia valor en b
    UsarValor(b);
   end
end
```

(b) Uso del monitor PC

Si llamamos $\#E$ al número de llamadas a `escribir()` completadas, y $\#L$ al número de llamadas a `leer()` completadas, el IM es:

$$\#E - \#L = \begin{cases} 0 \text{ si pendiente} == \text{false} \\ 1 \text{ si pendiente} == \text{true} \end{cases}$$

2.6. Colas de prioridad

Para determinadas clases de aplicaciones podría existir la necesidad de utilizar variables condición que planifiquen el desbloqueo de los procesos según un orden prioritario, en lugar de desbloquearlos en un orden FIFO.

Para poder obtener un orden de desbloqueo de los procesos según su prioridad se introduce una nueva operación definida para las variables condición, cuya definición es la siguiente:

- `c.wait(prioridad)`: bloquea los procesos en la cola `c`, pero ordenándolos de forma automática, con respecto al valor que tenga su argumento.
- `prioridad`: número no negativo que indica mayor importancia del proceso para valores números más pequeños.

Ejemplo. Vamos a ver un ejemplo de monitor para implementar un despertador. Con un procedimiento `tick()` (llamado por el sistema) se permite disparar una alarma a diferentes horas. Los usuarios hacen peticiones para crear alarmas, y el programa crea un proceso por cada petición. Cuando se cumple el tiempo, el monitor envía una señal al proceso correspondiente, para que se inicie la alarma. Los procesos entran en la cola prioritaria `despertar` después de realizar la llamada al procedimiento `despiertame()`.

```
monitor despertador;
  var
    ahora: Long_integer;
    despertar: cond; --prioritaria

  procedure despiertame(n: integer);
    var alarma: Long_integer;
    begin
      alarma:= ahora + n;
      while ahora< alarma do
        despertar.wait(n);
      end do;
      despertar.signal();
    end;
```

(a) Implementación del monitor `despertador`

```
  procedure tick();//cableada a INT CLK
    begin
      ahora:= ahora +1;
      despertar.signal();
    end;
begin
  ahora:= 0;
end;
```

(b) Procedimiento `tick()`

2.7. Semántica de las señales de los monitores

Las señales de *semántica desplazantes*, que son con las que hemos estado trabajando hasta ahora, suponen la reanudación inmediata del proceso señalado (por ejemplo, el primero de la cola) y evitan que pueda sufrir un robo de señal por parte de otro proceso. Sin embargo, esta semántica no es la única que utilizan los lenguajes de programación para implementar la sincronización con condiciones en los monitores. Por ejemplo, podríamos pensar en una implementación de señales en la que el proceso que llama a `c.signal()` no abandone inmediatamente el monitor, y el proceso señalado se bloqueara temporalmente hasta que el señalador termine la ejecución del procedimiento.

Existen 5 tipos de mecanismo que son utilizados por diferentes lenguajes concurrentes con monitores. Son los siguientes:

Acrónimo	Nombre del tipo de señal	Descripción del comportamiento y características de uso e implementación interna
SA	Señales automáticas	Señal implícita
SC	Señalar y continuar	Señal explícita, no desplazante
SS o SX	Señalar y salir	Señal explícita, desplazante. El proceso ha de salir del monitor después de ejecutar la operación de señalar
SE o SW	Señalar y esperar	Señal explícita, desplazante. El proceso señalador espera en la cola de entrada al monitor después de salir
SU	Señales urgentes	Señal explícita, desplazante. El proceso señalador espera en la cola de procesos urgentes

▶ Señales SA

Las señales se envían de forma implícita, es decir, las señales son incluidas por el compilador cuando se genera el código del programa. Por tanto, el programador solo tiene que incluir en el texto de los procedimientos del monitor las sentencias `c.wait()` necesarias.

▶ Señales SC, SS, SE, SU

Supone que la operación `c.signal()` ha de programarse de forma explícita en los procedimientos del monitor, es decir, el programador tiene que incluir las señales que haga falta para que cuando las condiciones de sincronización se cumplan, los procesos bloqueados en las colas de condición sean notificados.

▶ Señales SA, SC

Tienen una semántica *no desplazante*, es decir, el proceso que ejecuta la operación `c.signal()` sigue ejecutándose y, por tanto, no resulta desplazado del uso del procesador. Como el proceso señalador no deja libre el monitor inmediatamente, hay que suponer que se sigue ejcutando hasta que termine el procedimiento o él mismo se bloquee (ejecutando una operación `c.wait()`).

▶ Señales SS, SE, SU

Tienen una semántica *desplazante*, es decir, todo proceso que ejecute la operación `c.signal()` es obligado a ceder el monitor al primer proceso bloqueado, según un orden FIFO, en la cola de condición `c`. Si no hay ningún proceso bloqueado, el proceso señalador sigue ejecutándose.

La condición de desbloqueo de los procesos tiene que ser cierta cuando se ejecuta la operación `c.signal()`, y gracias a la semántica desplazante, sigue siendo cierta cuando el proceso señalado empiece su ejecución.

Notemos que si suponemos una semántica diferente, esto último no sería cierto, ya que otro proceso podría entrar en el monitor antes de que lo hiciera el proceso señalado. Para arreglar esto, basta con volver a comprobar que se cumple la condición, programando un bucle:

```
while !condicion_sincronizacion do cond.wait();
```

▶ Señales SU

El proceso señalado entra de forma inmediata en el monitor. El proceso señalador entra en una nueva cola de procesos del monitor llamada *cola de procesos urgentes*. de forma que los procesos en esta cola tienen preferencia cuando se queda libre el monitor.

2.7.1. Comparación entre las semánticas de señales

Características:

- **Potencia expresiva:** todas las semánticas son capaces de resolver los mismo problemas.
- La semántica SS puede llevar a aumentar de forma artificial el número de procedimientos

- Eficiencia:
  - Las semánticas SE y SU resultan ineficientes cuando no haya código tras `signal()` (porque se meten en la cola y cuando se desbloqueen no van a ejecutar ninguna línea de código).
  - La semántica SC también es un poco ineficiente, ya que al usar semántica no desplazante, hay que usar un bucle de comprobación para evitar el robo de señal.

Ejemplo. Supongamos que queremos construir un monitor (una barrera parcial) que cumpla con las siguientes características:

- Tiene un único procedimiento público llamado `cita`
- Hay p procesos ejecutando un bucle infinito. En cada iteración del bucle, realizan una actividad de duración arbitraria y después llaman a `cita`.
- Ningún proceso termina `cita` antes de que haya al menos n de ellos que la hayan iniciado ($1 < n < p$). Después de esperar en `cita`, pero antes de terminarla, el proceso imprime un mensaje.
- Cada vez que un grupo de n procesos llegan a la cita, n procesos imprimen su mensaje antes de que lo haga ningún otro proceso que haya llegado después de todos ellos a dicha cita (que sea del siguiente grupo de n procesos).

Una posible implementación del monitor es la siguiente:

```
Monitor BP //monitor Barrera Parcial }
  var cola : {\bf cond}ition;//procesos esperando contador
      ==n
      contador : integer;//numero de procesos ejecutando
          cita
procedure cita() ;
  begin
   contador := contador+1;//registrar un proceso mas
       ejecutando cita
   if (contador<n) then
       cola.wait();//esperar a que haya n procesos
           ejecutando
      else begin //si ya hay n procesos ejecutando la cita
         for i := 1 to n-1 do //para cada uno de estos
           cola.signal();//despertalo
         contador := 0;//volver a poner el contador a 0
      end
      print("salgo de cita");//mensaje de salida
  end
begin//inicializacion del monitor
contador := 0 ; { inicialmente, no hay procesos en cita }
```

Veamos el comportamiento de este monitor para las distintas semánticas de señales.

- Señalar y Continuar (SC). Los $n-1$ procesos señalados abandonan el `wait`, pero pasan a la cola del monitor. Los procesos del grupo de la siguiente cita se podrían adelantar.
- Señalar y Salir (SS). El último proceso del grupo abandona el monitor, y no pone `contador` a 0.
- Señalar y Esperar (SE). Parecido a SS; tampoco es correcta esta solución.
- Señalar y Espera Urgente (SU). El último proceso del grupo ejecuta todos los `signals`. Entre cada dos de ellos, espera en la cola de urgentes a que el proceso recién señalado abandone el monitor. La semántica SU de señales hace que esta solución sea **correcta**.

Ejemplo. Supongamos otra vez que queremos construir el mismo monitor del ejemplo anterior. Otra posible implementación sería la siguiente:

```
Monitor BP
var cola:{\bf cond}ition;//procesos esperando contador==n
    contador:integer;//numero de procesos esperando en la
        cola
  procedure cita() ;
   begin
    contador:=contador+1;//registrar un proceso mas
        esperando
    if (contador<n) then//todavia no hay n procesos:
      cola.wait();//esperar a que los haya
    contador:=contador-1;//registrar un proceso menos
        esperando
    print("salgo de la cita");//mensaje de salida
    if (contador>0) then//si hay otros procesos en la cola
      cola.signal();//despertar al siguiente
   end
begin//inicialización:
  contador:=0;//inicialmente, no hay procesos en la cola
end;
```

En este caso, sigue sin funcionar correctamente para la semántica SC, pero sí que funciona con el resto de semánticas (SE, SS, SU).
En general, hay que tener cuidado con la semántica que estemos usando, especialmente si el monitor tiene código tras `signal`. Generalmente, la semántica SC puede complicar mucho los diseños.