Resumen-T1-y-T2-SCD.pdf

LuCaS732

Sistemas Concurrentes y Distribuidos

2º Grado en Ingeniería Informática

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

Sistemas Concurrentes y Distribuidos

TEMA 1

Definiciones:

- Programa secuencial, el que estamos acostumbrados a usar en MP y FP, en secuencia.
- Programa concurrente, está relacionado, consta de varios programas que se ejecutan en paralelo y que tienen relación.
- Proceso, ejecución de un programa secuencial.
- Concurrencia, potencial para ejecución paralela, solapamiento real o virtual de varias actividades en el tiempo.
- Programación concurrente, 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.

Conceptos:

- 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. Se utiliza cuando hay que hacer cálculos enormes para optimizarlos utilizando los recursos que hay.

- Programación distribuida: (T3) 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). Es preferible una mala respuesta pero pronto que la buena pero tarde.

Beneficios de la programación concurrente:

- Mejora la eficiencia
    - En sistemas con un solo procesador:
        - Al tener varias tareas, cuando la tarea que tiene el control del procesador necesita realizar una E/S cede el control a otra (modelo de los estados), evitando la espera ociosa del procesador.
        - También permite que varios usuarios usen el sistema de forma interactiva (actuales sistemas operativos multiusuario).
    - En sistemas con varios procesadores

        - Es posible repartir las tareas entre los procesadores reduciendo el tiempo de ejecución.
        - Fundamental para acelerar complejos cómputos numéricos.

- Mejoras en la calidad
    - Muchos programas se entienden mejor en términos de varios procesos secuenciales ejecutándose concurrentemente que como un único programa secuencial.
    - Servidor web para reserva de vuelos: Es más natural, considerar cada petición de usuario como un proceso e implementar políticas para evitar situaciones conflictivas (permitir superar el límite de reservas en un vuelo).
    - Simulador del comportamiento de una gasolinera: Es más sencillo considerar los surtidores, clientes, vehículos y empleados como procesos que cambian de estado al participar en diversas actividades comunes, que considerarlos como entidades dentro de un único programa secuencial.

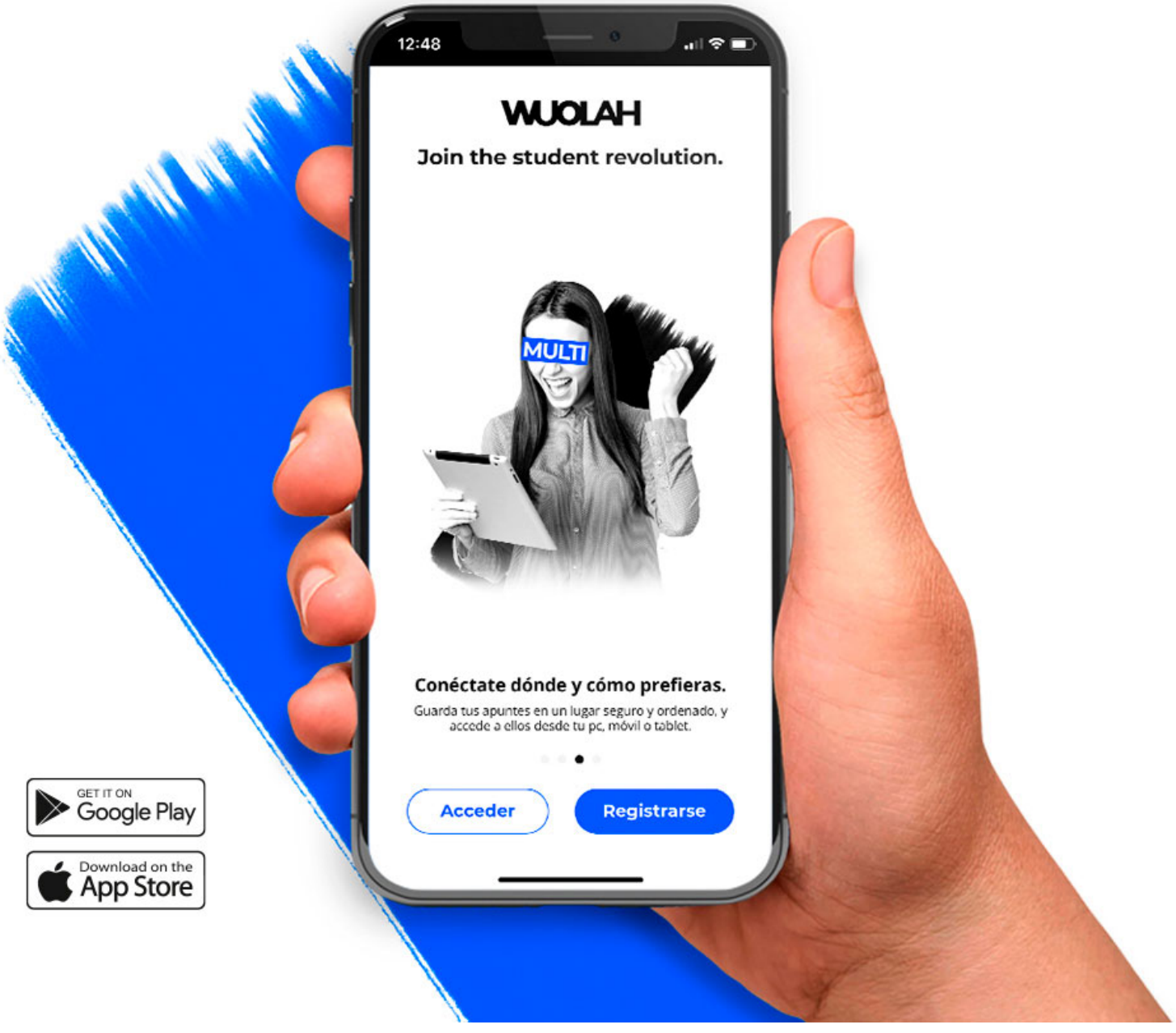
Consideraciones sobre el hardware

- Paralelismo Virtual
    - Monoprocesador
- Paralelismo Real o Híbrido
    - Multiprocesador de memoria compartida
    - Sistema Distribuido

Subsección 2.2 | Modelo Abstracto de Concurrencia

Sentencia atómica, lo que tiene que hacer lo hace sin verse afectada.
Prácticamente cualquier sentencia en lenguaje de alto nivel es no atómica.
Todo en última instancia es una sentencia atómica.
El orden de la interfoliación de sentencias atómicas es siempre secuencial. El número de posibles interfoliaciones es muy elevado incluso para programas cortos.
Progreso Finito (**importante**) No se puede hacer ninguna suposición acerca de las velocidades absolutas/relativas de ejecución de los procesos, salvo que es mayor que cero.
Si vemos en el código "*process x*" y "*process y*" significa que se ejecutan concurrentemente.

Exámenes, preguntas, apuntes.



GET IT ON Google Play

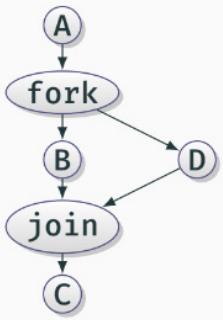
Download on the App Store

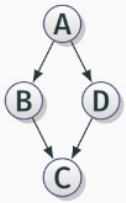
Fork-Join

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

```
procedure P1 ;
begin
   A ;
   fork P2 ;
   B ;
   join P2 ;
   C ;
end
```

```
procedure P2 ;
begin
   D ;
end
```





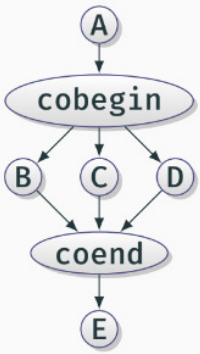
- **Ventajas:** práctica y potente, creación dinámica.
- **Inconvenientes:** no estructuración, difícil comprensión de los programas.

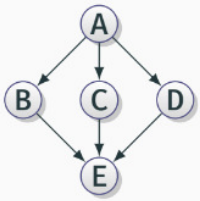
Cobegin-Coend

Las sentencias en un bloque delimitado por **cobegin-coend** comienzan su ejecución todas ellas a la vez:

- en el **coend** se espera a que se terminen todas las sentencias.
- Hace explícito qué rutinas van a ejecutarse concurrentemente.

```
begin
   A ;
   cobegin
      B ; C ; D ;
   coend
   E ;
end
```





- **Ventajas**: impone estructura: 1 única entrada y 1 única salida $\Longrightarrow$ más fácil de entender.
- **Inconveniente**: menor potencia expresiva que **fork-join**.

Subsección 3.1 | Exclusión Mutua

Cuando los procesos concurrentes ejecutan sus instrucciones atómicas, algunas de las posibles formas de combinar las secuencias no son válidas, es lo que se llama **condición de sincronización**, es decir, que hay alguna restricción sobre el orden en el que se pueden mezclar las instrucciones de distintos procesos.

La exclusión mutua, son secuencias finitas de instrucciones (**sección crítica**) que deben ejecutarse de principio a fin por un único proceso, sin que a la vez otro proceso las esté ejecutando también.

Es importante también que en el pseudocódigo utilizamos “< >” para indicar que son sentencias que se deben ejecutar de forma atómica.

Subsección 3.2 | Condición de Sincronización

No son correctas todas las posibles interfoliaciones de las secuencias de instrucciones atómicas de los procesos.

Sección 4 | Propiedades de los Sistemas Concurrentes

Hay de seguridad y de vivacidad.

Ausencia Interbloqueo (Deadlock-freedom): Nunca ocurrirá que los procesos se encuentren esperando algo que nunca sucederá.

Sección 5 | Verificación de Programas Concurrentes

Enfoque axiomático

TEMA 2

Sección 1 | Introducción a la sincronización en memoria compartida.

Soluciones de bajo nivel:

- Soluciones software: se usan operaciones estándar sencillas de lectura y escritura de datos simples (típicamente valores lógicos o enteros) en la memoria compartida.
- Soluciones hardware (cerrojos): basadas en la existencia de instrucciones máquina específicas dentro del repertorio de instrucciones de los procesadores involucrados.

Soluciones de alto nivel: se ofrecen interfaces de acceso a estructuras de datos y además se usa bloqueo de procesos en lugar de espera ocupada.

- Semáforos
- Regiones críticas condicionales
- Monitores.

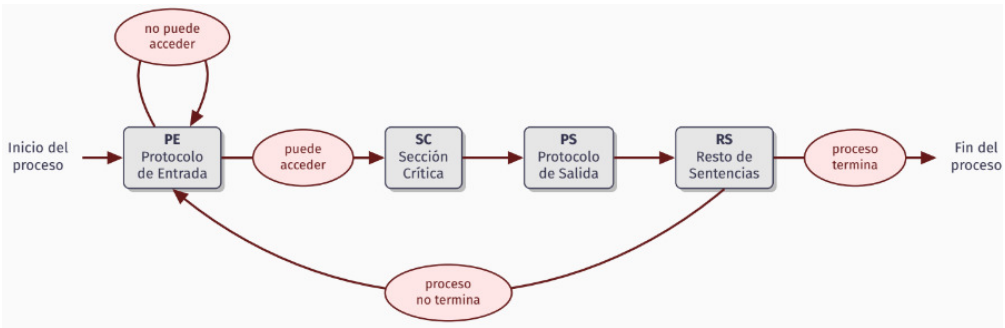
Sección 2 | Soluciones software con espera ocupada para E.M.

Veremos Algoritmo de Dekker (2 procesos) y Algoritmo de Peterson (para 2 y para un número arbitrario de procesos).

Subsección 2.1 | Estructura de los procesos con secciones críticas.

Un bloque considerado como sección crítica (SC) tendrá dicho bloque estructurado en tres etapas:

- **Protocolo de entrada (PE)**: instrucciones de espera.
- **Sección crítica (SC)**
- **Protocolo de salida (PS)**: instrucciones de que el proceso ha terminado la SC.
- El resto que no forma parte de ninguna de estas tres partes se llama **Resto de Sentencias (RS)**.



El código del Protocolo de Entrada introduce esperas (mediante bucles) para asegurar exclusión mútua en la Sección Crítica. Durante el tiempo en que un proceso se encuentra en una sección crítica nunca:

- Finaliza o aborta.
- Es finalizado o abortado externamente.
- Entra en un bucle infinito.
- Es bloqueado o suspendido indefinidamente de forma externa.

Es deseable que el tiempo empleado en SC sea lo menos posible.

Subsección 2.2 | Propiedades para exclusión mutua.

Para que un algoritmo para EM sea correcto, se deben cumplir cada una de estas tres propiedades mínimas:

- Exclusión mutua
- Progreso
- Espera limitada

Además, hay propiedades deseables adicionales que también deben cumplirse:

- Eficiencia
- Equidad

Vamos a explicar cada una brevemente.

Propiedad de Exclusión Mutua: En cada instante de tiempo, y para cada sección crítica existente, habrá como mucho un proceso ejecutando alguna sentencia de dicha región crítica.

Propiedad de Progreso: Consideremos una SC en un instante en el cual no hay ningún proceso ejecutándola, pero sí hay procesos en el protocolo de entrada compitiendo por entrar a la SC. La propiedad de progreso establece:

Un algoritmo de EM debe estar diseñado de forma tal que:

1. Después de un intervalo de tiempo finito desde que ingresó el primer proceso al PE, uno de los procesos en el mismo podrá acceder a la SC.

2. La selección del proceso anterior es completamente independiente del comportamiento de los procesos que durante todo ese intervalo no han estado en SC ni han intentado acceder.

Cuando la condición (1) no se da, se dice que ocurre un **interbloqueo**, ya que todos los procesos en el PE quedan en espera ocupada indefinidamente sin que ninguno pueda avanzar.

Espera Limitada: Supongamos que un proceso emplea un intervalo de tiempo en el PE intentando acceder a una SC. Durante ese intervalo de tiempo, cualquier otro proceso activo puede entrar un número arbitrario de veces *n* a ese mismo PE y lograr acceso a la SC (incluyendo la posibilidad de que n = 0). La propiedad de espera limitada establece que:

Un algoritmo de exclusión mutua debe estar diseñado de forma que *n* nunca será superior a un valor máximo determinado.

Esto implica que las esperas en el PE siempre serán finitas (suponiendo que los procesos emplean un tiempo finito en la SC).

Eficiencia: Los protocolos de entrada y salida deben emplear poco tiempo de procesamiento (excluyendo las esperas ocupadas del PE), y las variables compartidas deben usar poca cantidad de memoria.

Equidad: En los casos en que haya varios procesos compitiendo por acceder a una SC (de forma repetida en el tiempo), no debería existir la posibilidad de que sistemáticamente se perjudique a algunos y se beneficie a otros.

Subsección 2.3 | Refinamiento sucesivo de Dijkstra.

(Pueden caer ejercicios de los diferentes algoritmos para que comprobemos si se cumplen o no las propiedades requeridas para las soluciones para la Exclusión Mutua)

El Refinamiento sucesivo de Dijskstra 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. La versión final correcta se denomina Algoritmo de Dekker.

Ver las 5 versiones con sus correcciones, transparencias 15-24.

En la versión 2 no se cumple la propiedad de progreso, ya que el proceso P1 si ya ha terminado su parte del “resto de sección” pero en teoría le toca al P0 la sección crítica, aunque este esté aún en “resto de sección”, el P1 no puede repetir el bucle pues tiene que esperar a P0.

En la versión 4 se produce interbloqueo, quiere decir que uno espera por el otro y el otro por el uno en un bucle infinito.

En la versión 5 podría darse el caso de que se diese un interbloqueo continuo.

Subsección 2.4 | Algoritmo de Dekker.

Es un algoritmo correcto, es decir, cumple exclusión mutua, espera limitada y condición de progreso. Se puede interpretar como el resultado final del refinamiento sucesivo de Dijkstra. La gran diferencia es que la espera de cortesía solo la hace uno de los procesos. Solo se hace si ambos procesos entran a la vez, si no, no es necesario.

```
{ variables compartidas y valores iniciales }
var p0sc    : boolean := falso ; { true solo si proc.0 en PE o SC }
    p1sc    : boolean := falso ; { true solo si proc.1 en PE o SC }
    turno0  : boolean := true  ; { true ==> pr.0 no hace espera de cortesia }
```

```
1  process P0 ;
2  begin
3     while true do begin
4        p0sc := true ;
5        while p1sc do begin
6           if not turno0 then begin
7              p0sc := false ;
8              while not turno0 do
9                 begin end
10             p0sc := true ;
11          end
12       end
13       { sección crítica }
14       turno0 := false ;
15       p0sc := false ;
16       { resto sección }
17    end
18 end
```

```
1  process P1 ;
2  begin
3     while true do begin
4        p1sc := true ;
5        while p0sc do begin
6           if turno0 then begin
7              p1sc := false ;
8              while turno0 do
9                 begin end
10             p1sc := true ;
11          end
12       end
13       { sección crítica }
14       turno0 := true ;
15       p1sc := false ;
16       { resto sección }
17    end
18 end
```

Subsección 2.5 | Algoritmo de Peterson.

Misma idea que Dekker pero simplificado. Al ser la asignación de turno0 atómica (línea 5), si los dos procesos coinciden a la vez en el segundo while, y p0sc y p1sc valen true, turno0 o vale true en uno y false en el otro o viceversa, y al ser con and, solo puede entrar uno de los dos procesos. Cumple exclusión mutua, espera limitada y condición de progreso (En las diapositivas 29-32 se ve la justificación de las 3 condiciones).

```
{ variables compartidas y valores iniciales }
var p0sc    : boolean := falso ; { true solo si proc.0 en PE o SC }
    p1sc    : boolean := falso ; { true solo si proc.1 en PE o SC }
    turno0  : boolean := true  ; { true ==> pr.0 no hace espera de cortesia }
```

```
1  process P0 ;
2  begin
3     while true do begin
4        p0sc := true ;
5        turno0 := false ;
6        while p1sc and not turno0 do
7           begin end
8        { sección crítica }
9        p0sc := false ;
10       { resto sección }
11    end
12 end
```

```
1  process P1 ;
2  begin
3     while true do begin
4        p1sc := true ;
5        turno0 := true ;
6        while p0sc and turno0 do
7           begin end
8        { sección crítica }
9        p1sc := false ;
10       { resto sección }
11    end
12 end
```

Sección 3 | 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 solucionar el problema de la exclusión mutua.

- La espera ocupada constituye un bucle que se ejecuta hasta que ningún otro proceso esté ejecutando instrucciones de la sección crítica.
- Existe un valor lógico en una posición de memoria compartida (llamado cerrojo) que indica si algún proceso está en la sección crítica o no.
- En el protocolo de salida se actualiza el cerrojo de forma que se refleje que la SC ha quedado libre.

La instrucción TestAndSet: la mayoría de los procesadores la siguen incorporando, o alguna instrucción similar. Es una instrucción máquina. Las tres acciones que realiza las ejecuta de forma atómica, es decir, hasta que no completa las tres acciones no es interrumpida. Admite como argumento la dirección de memoria de la variable lógica que actúa como cerrojo. Se invoca como una función desde LLPP de alto nivel, y ejecuta estas acciones:

1. Lee el valor anterior del cerrojo.
2. Pone el cerrojo a true.
3. Devuelve el valor anterior del cerrojo.

```
{ variables compartidas y valores iniciales }
var sc_ocupada : boolean := false ; { true solo si la SC esta ocupada }

{ procesos }
process P[ i : 1 .. n ];
begin
   while true do begin
      while TestAndSet( sc_ocupada ) do begin end
      { seccion critica  }
      sc_ocupada := false ;
      { resto seccion }
   end
end
```

Cuando hay más de un proceso intentando entrar en SC (estando SC libre), solo uno de ellos (el primero en ejecutar TestAndSet) ve el cerrojo (sc_ocupada) a false, lo pone a true y logra entrar a SC.

Las desventajas de los cerrojos son:

- Las esperas ocupadas consumen tiempo de CPU que podría dedicarse a otros procesos para hacer trabajo útil.
- Se puede acceder directamente a los cerrojos y por tanto un programa erróneo o escrito malintencionadamente puede poner un cerrojo en un estado incorrecto, pudiendo dejar a otros procesos indefinidamente en espera ocupada.
- En la forma básica que hemos visto no se cumplen ciertas condiciones de equidad.

Uso de los cerrojos:

- Por seguridad, normalmente solo se usan desde componentes software que forman parte del sistema operativo, librerías de hebras, de tiempo real o similares (estas componentes suelen estar bien comprobadas y por tanto libres de errores o código malicioso).
- Para evitar la pérdida de eficiencia que supone la espera ocupada, se usan solo en casos en los que la ejecución de la SC conlleva un intervalo de tiempo muy corto (por tanto las esperas ocupadas son muy cortas, y la CPU no se desaprovecha).

Sección 4 | Semáforos para sincronización.

Vamos a estudiar la herramienta semáforo para la exclusión mutua

Subsección 4.1 | Estructura y operaciones de los semáforos.

Los semáforos constituyen un mecanismo que soluciona o aminora los problemas de las soluciones de bajo nivel. En general:

- No se usa espera ocupada, sino bloqueo de procesos (uso mucho más eficiente de la CPU).
- Resuelven fácilmente el problema de exclusión mutua con esquemas de uso sencillos.
- Se pueden usar para resolver problemas de sincronización (aunque en ocasiones los esquemas de uso son complejos).
- El mecanismo se implementa mediante instancias de una estructura de datos a las que se accede únicamente mediante subprogramas específicos. Esto aumenta la seguridad y simplicidad.

Los semáforos exigen que los procesos en espera no ocupen la CPU, esto implica que:

- Un proceso en ejecución debe poder solicitar quedarse bloqueado.
- Un proceso bloqueado no puede ejecutar instrucciones en la CPU.
- Un proceso en ejecución debe poder solicitar que se desbloquee (se reanude) algún otro proceso bloqueado.
- Deben poder existir simultáneamente varios conjuntos de procesos bloqueados.
- Cada petición de bloqueo o desbloqueo se debe referir a alguno de estos conjuntos.

Estructura: Un semáforo es un instancia de una estructura de datos (un registro) que contiene los siguientes elementos:

- Un conjunto de procesos bloqueados (de estos procesos decimos que están esperando al semáforo).
- Un valor natural (un valor entero no negativo), al que llamaremos por simplicidad 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.

Operaciones: Además de la inicialización, sólo hay dos operaciones básicas que se pueden realizar sobre una variable u objeto cualquiera de tipo semáforo (que llamamos s):

- sem_wait(s)
    - Si el valor de s es 0, bloquear el proceso, que será reanudado después en un instante en que el valor ya es 1.
    - Decrementar el valor del semáforo en una unidad.

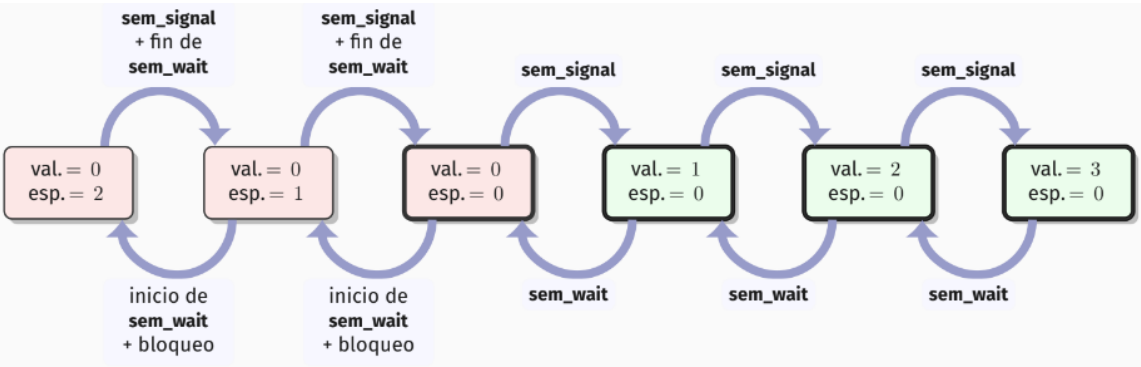
- sem_signal(s)
    - Incrementar el valor de s en una unidad.
    - Si hay procesos esperando en s, reanudar o despertar a uno de ellos (ese proceso pone el semáforo a 0 al salir).

Este diseño implica que el valor del semáforo nunca es negativo, ya que antes de decrementar se espera a que sea 1. Además, solo puede haber procesos esperando cuando el valor es 0.

Invariante de un semáforo: Dado un semáforo s, en un instante de tiempo cualquiera t su valor (vt) será el valor inicial (v0) más el número de llamadas a sem_signal completadas (ns), menos el número de llamadas a sem_wait completadas (nw). Ese valor nunca es negativo. Es decir:

$$vt = v0 + ns - nw \geq 0$$

Vemos algunos posibles estados de un semáforo, según su número de procesos esperando, (esp.) y su valor (val.), y las posibles transiciones atómicas (en E.M.) entre esos estados, provocadas por hebras que invocan sem_wait o sem_signal



Subsección 4.2 | Uso de semáforos: patrones sencillos.

Vamos a ver el uso de semáforos en algunos problemas (ver en las diapositivas 48 en adelante).

Sección 5 | Monitores como mecanismo de alto nivel

Vamos a ver una solución, de mayor alto nivel que los semáforos, que son los monitores.

Subsección 5.1 | Introducción. Definición de monitor.

Problemas de los semáforos:

- Están basados en variables globales
- El uso y función de las variables no se hace explícito en el programa.
- Las operaciones se encuentran dispersas y no protegidas, hay más posibilidad de errores.

Se pide una herramienta de más alto nivel, que son los **monitores**, para **resolver los mismos problemas de programación concurrente que resuelven los semáforos**. El **monitor** es un mecanismo de alto nivel que permite definir **objetos abstractos compartidos** por todas las hebras del programa concurrente. Incluyen:

- Colección de variables encapsuladas (datos) que representan un recurso compartido por varios procesos. No son accesibles desde el exterior.
- Un conjunto de procedimientos para manipular el recurso: afectan a las variables encapsuladas.

En los monitores:

- Se garantiza el acceso en exclusión mutua a las variables encapsuladas.
- Se implementan la sincronización requerida por el problema mediante esperas bloqueadas.

Las propiedades son iguales que las de cualquier objeto, solo se puede acceder al recurso mediante un conjunto de operaciones y el usuario solo conoce la interfaz. La exclusión mutua en el acceso a los procedimientos del monitor está garantizada por definición.
Las ventajas respecto a los semáforos son:

- Las variables están protegidas.
- La exclusión mutua está garantizada.
- Las operaciones de esperas bloqueadas y de señalización se programan exclusivamente dentro del monitor.

Normalmente todos los procedimientos del monitor van a ser públicos.

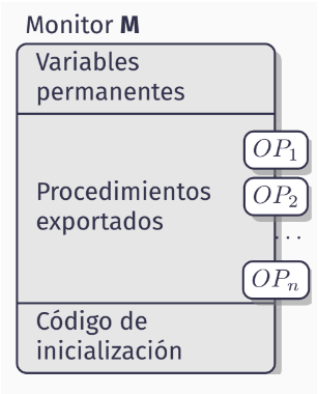
Los componentes de un monitor son:

- Variables permanentes: son el estado interno del monitor.
  - Sólo pueden ser accedidas dentro del monitor (en el cuerpo de los procedimientos y código de inicialización).
  - Permanecen sin modificaciones entre dos llamadas consecutivas a procedimientos del monitor.

- Procedimientos: modifican el estado interno (en E.M.)
  - Pueden tener variables y parámetros locales, que toman un nuevo valor en cada activación del procedimiento.
  - Algunos (o todos) constituyen la interfaz externa del monitor y podrán ser llamados por los procesos que comparten el recurso.

- Código de inicialización: fija estado interno inicial (opcional)
  - Se ejecuta una única vez, antes de cualquier llamada a procedimientos del monitor.

Diagrama de los componentes del monitor:



- El uso que se hace del monitor se hace exclusivamente usando los procedimientos exportados (constituyen el interfaz con el exterior).
- Las variables permanentes y los procedimientos no exportados no son accesibles desde fuera.
- Ventaja: la implementación de las operaciones se puede cambiar sin modificar su semántica.

Ejemplo de monitor sencillo:

```
{ declaracion del monitor }
monitor VarCompartida ;

 var x : integer; { permanente }
 export incremento, valor;

 procedure incremento( );
 begin
   x := x+1 ; {incrementa valor }
 end;
 function valor() : integer ;
 begin
   return x; { devuelve valor }
 end;

begin { código de inicialización}
  x := 0 ; { inicializa valor }
end { fin del monitor}
```

```
{ ejemplo de uso del monitor  }
{ (debe aparecer en el ámbito }
{ de la declaración)          }

{ incrementar valor: }
VarCompartida.incremento();

{ copiar en k el valor: }
k := VarCompartida.valor() ;
```

Los procedimientos que devuelven un valor usan **return**, y se declaran con **function** en lugar de **procedure**. Se especifica el tipo del valor devuelto.

Se pueden crear múltiples instancias en un monitor. Cada una con sus variables permanentes propias.

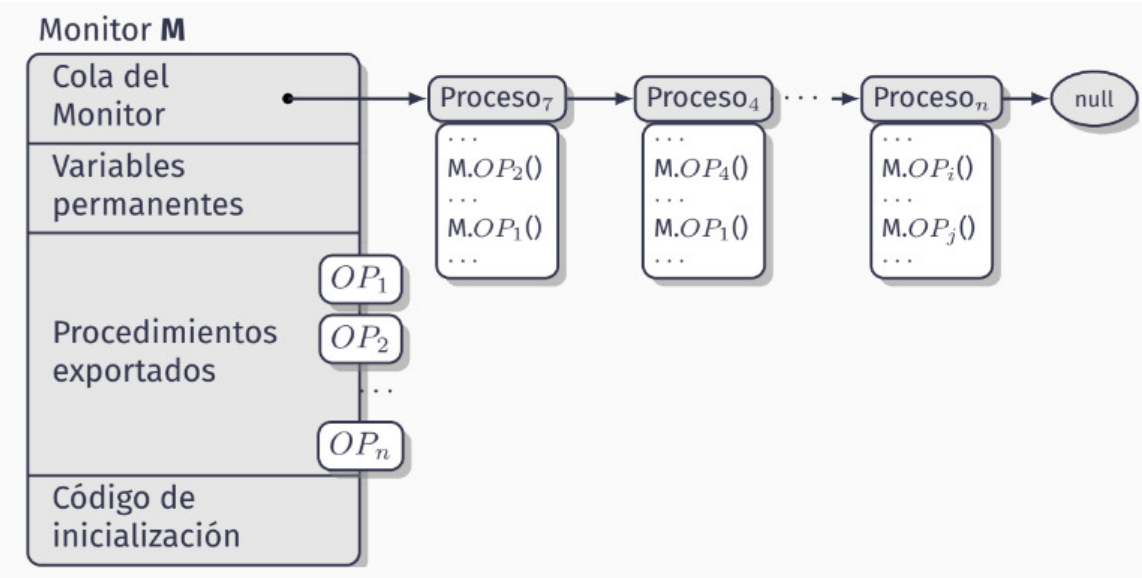
Subsección 5.2 | Funcionamiento de los monitores

Mientras el proceso está ejecutando algún procedimiento del monitor decimos que el proceso está dentro del monitor.
Exclusión mutua: Si un proceso P está dentro de un monitor, cualquier otro proceso Q que llame a un procedimiento de ese monitor deberá esperar hasta que P salga del mismo.
El acceso exclusivo entre los procedimientos del monitor debe estar garantizado en la implementación de los monitores.
Los monitores son objetos pasivos, el código de sus procedimientos sólo se ejecuta cuando estos son invocados por los procesos (hebras).

El control de la exclusión mutua se basa en la existencia de la cola del monitor:

- Si un proceso está dentro del monitor y otro proceso intenta ejecutar un procedimiento del monitor, éste último proceso queda bloqueado y se inserta en la cola del monitor.
- Cuando un proceso abandona el monitor (finaliza la ejecución del procedimiento), se desbloquea un proceso de la cola, que ya puede entrar al monitor.
- Si la cola del monitor está vacía, el monitor está libre y el primer proceso que ejecute una llamada a uno de sus procedimientos, entrará en el monitor.
- Para garantizar la vivacidad del sistema, la planificación de la cola del monitor debe seguir una política FIFO.

Por tanto, el estado del monitor incluye también la cola de procesos esperando a comenzar a ejecutar el código del mismo. Se puede representar por este diagrama:



Subsección 5.3 | Sincronización en monitores.

En semáforos, existe:

- La posibilidad de bloqueo (sem_wait) y activación (sem_signal)
- Un valor entero (el valor del semáforo), que indica si la condición se cumple (> 0) o no (== 0).

En monitores, sin embargo:

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

Para cada condición distinta que los procesos pueden eventualmente tener que esperar en un monitor, se debe de declarar una variable permanente de tipo **condition**, la llamaremos señales o variable condición. Cada variable condición tiene asociado una lista de procesos bloqueados esperando por esa condición. Para cada una de esas variables un proceso puede invocar dos operaciones:

- **wait** (estoy esperando a que alguna condición ocurra)
- **signal** (estoy señalando que una condición ocurre).

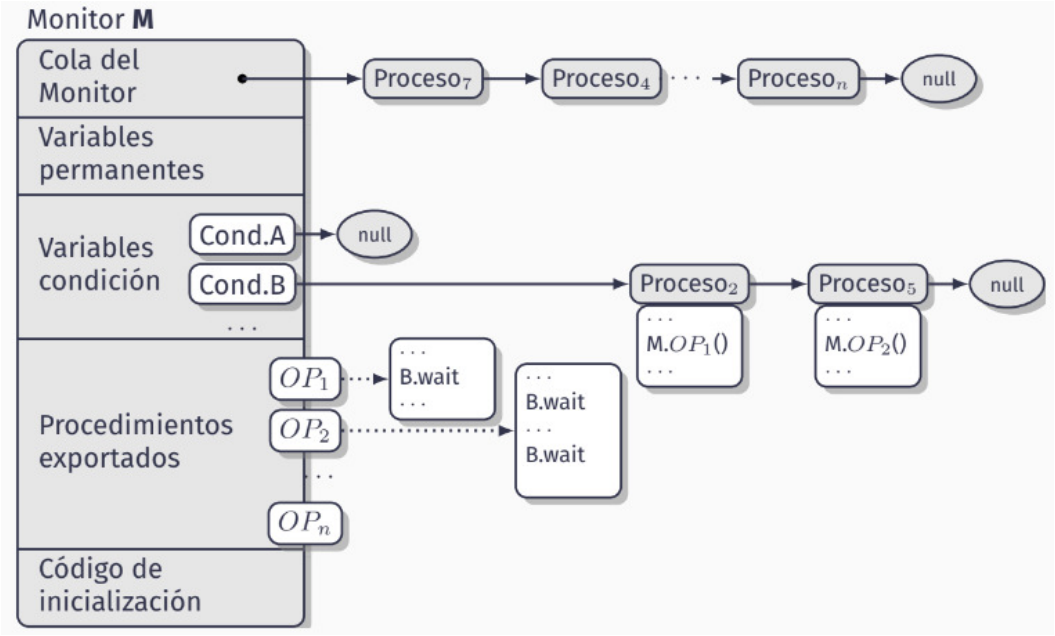
Ejemplos de uso con una variable cond:

- **cond.wait()**: bloquea incondicionalmente al proceso que la llama y lo introduce en la cola de la variable condición.
- **cond.signal()**: si hay procesos bloqueados por esa condición, libera uno de ellos. Si no hay ninguno esperando no hace nada. Si hay más de uno, se sigue una política FIFO (first-in first-out):
    - Se reactivará el proceso que lleve más tiempo esperando.
    - Esto evita la inanición: Cada proceso en cola obtendrá eventualmente su turno.
- **cond.queue()**: Función lógica que devuelve true si hay algún proceso esperando en la cola de cond, y false en caso contrario.

Imaginamos el caso de que un proceso que está dentro del monitor se bloquee, el problema sería caótico, porque no pueden entrar más procesos. En este caso se debe liberar la E.M. del monitor y así puede entrar otro. Por así decirlo **podemos tener varios procesos "dentro" del monitor, pero solo 1 ejecutándose**, es decir, podemos tener varios procesos bloqueados pero solo uno, como máximo, en ejecución.

Un proceso que está en la cola de variables de condición, está dentro del monitor pero está bloqueado esperando, mientras que la cola del monitor son procesos que NO están dentro del monitor sino esperando para entrar. Un proceso que no esté en el monitor no puede acceder a las operaciones de las variables condición.

Se ve claramente en el siguiente diagrama:



Subsección 5.4 | Verificación de monitores

La verificación de la corrección de un programa concurrente con monitores requiere:

- Probar la corrección de cada monitor.
- Probar la corrección de cada proceso de forma aislada.
- Probar la corrección de la ejecución concurrente de los procesos implicados.

El enfoque de verificación que vamos a seguir utiliza un invariante de monitor, que es una propiedad que el monitor cumple siempre.

- El invariante del monitor es una función lógica que se puede evaluar como true o false en cada estado del monitor a lo largo de la ejecución del programa concurrente.
- Su valor depende de la traza (*La traza del monitor es la secuencia ordenada en el tiempo de llamadas a procedimientos del monitor ya completadas, desde el inicio del programa hasta llegar al estado indicado*) del monitor y de los valores de las variables permanentes de dicho monitor.
- El IM debe ser cierto en cualquier estado del programa concurrente, excepto cuando un proceso está ejecutando código del monitor, en E.M. (está en proceso de actualización de los valores de las variables permanentes).

Subsección 5.5 | Patrones de solución con monitores

Vamos a estudiar el uso de monitores para los siguientes problemas:

- Espera única
- Exclusión mutua
- Problema del Productor/Consumidor

Ver las diapositivas con los ejemplos de pseudocódigo (*diapositivas 84 - 96*).

Subsección 5.6 | El problema de los Lectores/Escritores

Ver vídeo | Diapositivas 97 - 100.

Subsección 5.7 | Semántica de las señales de los monitores

Cuando un proceso hace signal en una cola no vacía, se denomina proceso **señalador**. El proceso que esperaba en la cola y que se reactiva se denomina **señalado**.

Planteamos el problema. Tenemos 2 hebras, una de ellas está bloqueada con un wait. La otra cuando se ejecuta llega a un signal, que desbloquea la primera hebra, pero claro no detiene su ejecución (suponiendo que hay código debajo del wait y del signal), luego el problema sería que tendríamos dos hebras en ejecución.

P1 (señalado)	P2 (señalador)
---	---
---	---
---	cond.signal()
cond.wait()	---
---	---
---	---

Se denomina semántica de señales a la política que establece la forma concreta en que se resuelve el conflicto tras hacerse un signal en una cola no vacía. Las posibles soluciones serían (truco: los nombres de las soluciones dependen del proceso señalador):

- El proceso señalador continua la ejecución tras el signal. El señalado espera bloqueado **en la cola del monitor** hasta que puede adquirir la E.M. de nuevo (**SC: señalar y continuar**).
- El proceso señalado se reactiva inmediatamente. El señalador:
  - abandona el monitor tras hacer signal sin ejecutar el código que haya después de dicho signal (**SS: señalar y salir**).
  - queda bloqueado a la espera en:
    - la cola del monitor, junto con otros posibles procesos que quieren comenzar a ejecutar código del monitor (**SE: señalar y esperar**).
    - una cola específica para esto, con mayor prioridad que esos otros procesos (**SU: señalar y espera urgente**).

Características **SC**:

- Tanto el señalador como otros procesos pueden hacer falsa la condición después de que el señalado abandone la cola condición.
- Por tanto, en el proceso señalado no se puede garantizar que la condición asociada a cond es cierta al terminar cond.wait(), y lógicamente es necesario volver a comprobarla entonces.

- Esta semántica obliga a programar la operación wait en un bucle, de la siguiente manera:
    - *while not condicion_lógica_desbloqueo do*
      cond.wait() ;

Características **SS**:

- En ese caso, la operación signal conlleva:
    - Liberar al proceso señalado.
    - Terminación del procedimiento del monitor que estaba ejecutando el proceso señalador.
- Está asegurado el estado que permite al proceso señalado continuar la ejecución del procedimiento del monitor en el que se bloqueó (la condición de desbloqueo se cumple).
- Esta semántica condiciona el estilo de programación ya que obliga a colocar siempre la operación signal como última instrucción de los procedimientos de monitor que la usen.

Características **SE**:

- Está asegurado el estado que permite al proceso señalado continuar la ejecución del procedimiento del monitor en el que se bloqueó.
- El proceso señalador entra en la cola de procesos del monitor, por lo que está al mismo nivel que el resto de procesos que compiten por la exclusión mutua del monitor.
- Puede considerarse una semántica injusta respecto al proceso señalador ya que dicho proceso ya había obtenido el acceso al monitor por lo que debería tener prioridad sobre el resto de procesos que compiten por el monitor.

Características **SU**:

- El proceso señalador se bloquea justo después de ejecutar la operación signal.
- El proceso señalado entra de forma inmediata en el monitor.
- Está asegurado el estado que permite al proceso señalado continuar la ejecución del procedimiento del monitor en el que se bloqueó.
- El proceso señalador entra en una nueva cola de procesos que esperan para acceder al monitor, que podemos llamar cola de procesos urgentes.
- Los procesos de la cola de procesos urgentes tienen preferencia para acceder al monitor frente a los procesos que esperan en la cola del monitor.
- Es la semántica que se supone en los ejemplos vistos.

<u>Comparativa de las diferentes semánticas</u>:

- **Potencia expresiva**: todas las semánticas son capaces de resolver los mismos problemas.
- **Facilidad de uso**: La semántica SS condiciona el estilo de programación y puede llevar a aumentar de forma artificial el número de procedimientos.
- **Eficiencia**:

    - Las semánticas SE y SU resultan ineficientes cuando no hay código tras signal, ya que en ese caso implican que el señalador emplea tiempo en bloquearse y después reactivarse, pero justo a continuación abandona el monitor sin hacer nada.
    - La semántica SC también es un poco ineficiente al obligar a usar un bucle para cada instrucción wait.

Ejemplo monitor **barrera parcial** (ver diapositivas 113 - 119).

En general, hay que ser cuidadoso con la semántica en uso, especialmente si el monitor tiene código tras signal. Generalmente, la semántica SC puede complicar mucho los diseños.

Subsección 5.8 | Colas de prioridad

Por defecto se usan colas de espera FIFO. Pero si queremos darle prioridad utilizamos una variable “p” que es un entero no negativo que refleja la prioridad.

- La sintaxis sería: *cond.wait(p)*.
- *cond.signal()* reanudará un proceso que especificó el valor mínimo de p de entre todos los que esperan (si hay más de uno con prioridad mínima, se usa política FIFO).
- Se deben evitar riesgos como la inanición.
- No tiene ningún efecto sobre la lógica del programa: el funcionamiento es similar con y sin colas de prioridad.
- Sólo mejoran las características dependientes del tiempo.

Subsección 5.9 | Implementación de monitores con semáforos

Es posible implementar cualquier monitor usando exclusivamente semáforos. Cada cola tendrá un semáforo asociado:

- **Cola del monitor**: se implementa con un semáforo de exclusión mutua (vale 0 si algún proceso está ejecutando código, 1 en otro caso).
- **Colas de variables condición**: para cada variable condición será necesario definir un semáforo (que está siempre a 0) y una variable entera que indica cuántos procesos hay esperando.
- **Cola de procesos urgentes**: en semántica SU, debe haber un entero y un semáforo (siempre a 0) adicionales.

Limitación: esta implementación no permite llamadas recursivas a los procedimientos del monitor y no asegura orden FIFO en las colas.

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