Sistemas Concurrentes Distribuidos
Tema 1

1.Conceptos básicos y motivación

1.1.Conceptos básicos relacionados con la concurrencia

Programa secuencial: son unas declaraciones de datos y estructuras de datos + un conjunto de instrucciones sobre dichos datos que se deben ejecutar en secuencia.

Programa concurrente: son un conjunto de programas secuenciales ordinarios que se pueden ejecutar lógicamente en paralelo. (Lógicamente, ya que aunque un programa sea concurrente si no se disponen de los recursos necesarios se va a seguir ejecutando secuencialmente tal vez dando sensación de que se ejecutan virtualmente varias instrucciones). Ejemplo en un monoprocesador con una única CPU.

Proceso: Es la 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. La PC es independiente de la implementación del paralelismo. Es una abstracción acerca del hardware subyacente.

Programación paralela: Subconjunto de la programación concurrente, es decir, un subconjunto de ejercicios de la programación concurrente el cual 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: Toda lo relativo a la computación entre ordenadores que no comparten una zona de memoria común. 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

La programación concurrente es más compleja que la programación secuencial y hay dos motivos para el desarrollo de esta: La mejora de la eficiencia y Mejoras en la calidad.

Mejora de la eficiencia:
La programación concurrente permite aprovechar mejor los recursos hardware existentes.
-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, evitando la espera ociosa del procesador. También permite que varios usuarios usen el sistema de forma interactiva (actuales sistemas operativos multisusuario).
-En sistemas con varios procesadores es posible repartir las tareas entre los procesadores, reduciendo el tiempo de ejecución al igual que la carga de cada uno de estos. También es fundamental para acelerar complejos cómputos numéricos.

Mejora de la calidad:
Muchos programas se entienden mejor en términos de varios procesos secuenciales ejecutándose concurréntemente que en un único programa secuencial.

Ejemplos:

-Servidor web de reservas de vuelos de manera que es natural considerar que cada petición de usuario como un proceso e implementar políticas para evitar situaciones conflictivas (permitir superar el límites de reservas en un vuelo)

-Simulador del comportamiento de una gasolinera es muy 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.

2. Modelo abstracto y consideraciones sobre el hardware

2.1. Consideraciones sobre el hardware

Vamos a hacer dos distinciones, la primera va a ser en relación al número de CPU que dispongamos (1 unidad o varias) y la segunda en relación a la memoria (un solo espacio de memoria o va a estar distribuido en distintas CPUs).

2.2. Modelo Abstracto de concurrencia

Sentencia atómica (indivisible) :Una sentencia o instrucción de un proceso en un programa concurrente es atómica si siempre se ejecuta de principio a fin sin verse *afectada* (durante su ejecución) por otras sentencias en ejecución de otros procesos del programa.
Esta no se verá afectada cuando el funcionamiento de dicha instrucción no dependa nunca de como se estén ejecutando otras instrucciones.
El funcionamiento de una instrucción se define por su efecto en el estado de ejecución del programa justo cuando acaba.
El estado de ejecución esta formado por los valores de las variables y los registros de todos los procesos.
El resultado de estas instrucciones no depende nunca de otras intrucciones que se estén ejecutando concurrentemente. Al finalizar, la celda de memoria o el registro (donde se escribe) tomará un valor concreto predecible siempre a partir del estado al inicio (el estado justo al acabar está determinado).

La mayoría de las sentencias en lenguajes de alto nivel son no atómicas de forma que el en un programa concurrente el resultado dependera de que haya o no otras sentencias ejecutandose a la vez y escribiendo simultaneamente una variable, produciendo una indeterminación.

El modelo basado en el estudio de todas las posibles secuencias de ejecución entrelazadas de los procesos constituye una abstracción:
- Se consideran exclusivamente las características relevantes que determinan el resultado del cálculo
- Esto permite simplificar el análisis y diseño de los programas concurrentes.
Se ignoran los detalles no relevantes para el resultado, como por ejemplo las áreas de memoria asignadas a los procesos, los registros particulares que están usando, el costo de los cambios de contexto entre procesos, la política del S.O. relativa a asignación de CPU, las diferencias entre entornos multiprocesador o monoprocesador.

Aunque se produzca entrelazamiento se preserva la consistencia, es decir, el resultado de una instrucción individual sobre un dato no depende de las circunstancias de la ejecución. En caso contrario no se podría razonar acerca de la correción de programas concurrentes.

La consistencia secuencial estricta dice que una instrucción atómica de de lectura de una variable siempre leerá el valor escrito en dicha variable por la última instrucción atómica previa de escritura en esa variable. Esto supone:

- Implica que un valor escrito de forma atómica es inmediatamente legible o visible para todos los procesos sin retrasos.
- Hace más sencillo el análisis de l comportamiento de las interfoliaciones
- Es complejo asegurarlo en la practica por los retrasos en las arquitecturas modernas de CPUs y memorias.

Progreso Finito quiere decir que no se puede hacer ninguna suposición de la velocidad absoluta o relativa de ejecución de los procesos salvo que es mayor que 0
Si se hicieran suposiciones sería difícil detectar y corregir fallos. Además de que la corrección dependería de la configuración de ejecución, que puede cambiar.
Existen dos consecuencias del progreso finito:

- De manera global durante la ejecución de un PC en cualquier momento existirá al menos un proceso preparado (eventualmente se permitirá la ejecución de algún proceso)
-De forma local cuando un proceso comienza la ejecución de una sentencia, completará la ejecución de la sentencia en un intervalo de tiempo finito.

El estado de un PC es el valor de las variables del programa en un momento dado. Incluyen variables declaradas explícitamente y variables con información de estado oculta (contador de programa, registros...).
Un PC comienza en un estado inicial y los procesos van modificando el estado conforme ejecutan sus sentencias atómicas.
Historia o traza de un PC es la secuencia de estados producida por una secuencia concreta de interfoliación.

Distinguimos dos tipos de sistemas concurrentes en función de la posibilidad para especificar cuales son sus procesos.

- Sistemas estáticos (Número de procesos fijados en el fuente del programa y se activan al lazar el programa).
- Sistemas dinámicos (Número de procesos se pueden activar en cualquier momento de la ejecución)

El grafo de sincronización es un Grafo dirigido Acíclico (DAG) donde cada nodo representa una secuencia de sentencias del programa, mostrándose las restricciones que determinan cuando una actividad puede empezar un programa.

En la definición estática de los procesos el número de procesos y el código que ejecutan no cambian entre ejecuciones. Cada proceso se asocia con su identificador y su código mediante la palabra clave procesos.
El programa acaba cuando acaban todos los procesos. Las variables compartidas se inicializan antes de que comiencen los procesos.
Se pueden usar definiciones estáticas de grupos de procesos similares que solo se diferencia en el valor de una constante(vectores de procesos)

La creación de procesos no estrucutada fork-join:

- Fork: sentencia que especifica el comienzo de una rutina
- Join: sentencia que especifica la terminación de una rutina

Es práctica y potente pero al no tener estructuración la comprensión de los programas es complicada.

La creación de procesos estructurada con cobegin-coend:

- Coend: se espera a que terminen todas las sentencias. Además hace explicito que rutinas van a ejecutarse concurrentemente

Tiene la ventaja de que impone estructura: 1 entrada- 1 salida pero tiene el inconveniente de que la potencia expresiva es de menor potencia.

3. Exclusión mutua y sincronización

Según el modelo abstracto, los procesos concurrentes ejecutan sus instrucciones atómicas de forma que es completamente arbitrario el entremezclado en el tiempo de sus respectivas secuencias de instrucciones. Sin embargo, en un conjunto de procesos que no son independientes entre sí (es decir, son cooperativos), algunas de las posibles formas de combinar las secuencias no son válidas.
En general, se dice que hay una condición de sincronización cuando esto ocurre (alguna restricción en el orden de mezcla de las instrucciones de distintos procesos)
Un caso particular es la exclusión mutua, son 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.

3.1. Concepto de exclusión mutua

La restricción se refiere a una o varias secuencias de instrucciones consecutivas que aparecen en el texto de uno o varios procesos. Al conjunto de dichas secuencias de instrucciones se le denomina sección crítica (SC).
Ocurre exclusión mutua (EM) cuando los procesos solo funcionan correctamente si, en cada instante de tiempo, hay como mucho uno de ellos ejecutando cualquier instrucción de la sección crítica. Es decir, el solapamiento de las 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 (durante ese tiempo) otros procesos ejecuten ninguna de esas instrucciones ni otras de la misma SC.

3.2. Condición de sincronización
En general, en un programa concurrente con varios procesos, una condición de sincronización establece que no son correctas todas las posibles interfoliaciones de las instrucciones atómicas de los procesos. Esto ocurre cuando, en un punto concreto de su ejecución, uno o varios procesos deben esperar a que se cumpla una determinada condición global (depende de varios procesos).
La secuencia válida asegura la condición de sincronización.

4. Propiedades de los sistemas concurrentes
Porpiedad de un programa concurrente se entiende como el atributo del programa que es cierto para todas kas posibles secuencias de interfoliación (historias del programa).
Hay dos tipos: propiedad de seguridad (safety) y propiedad de vivacidad (liveness).

Safety: son condiciones que deben cumplirse en cada instante. Son requeridas en especificaciones estáticas del programa, son fáciles de demostrar y para cumplirlas se suelen restringir las posibles interfoliaciones. Ej:EM, Ausencia de interbloqueo (esperar algo que nunca pasará), propiedad de seguridad...

Liveness: son propiedades que deben cumplirse eventualmente. Porpiedades dinámicas, mas difíciles de probar. Ej: Ausencia de inanición (un proceso no puede ser indefinidamente pospuesto), Equidad (un proceso que desee progresar debe hacerlo con justicia relativa con respecto a los demás).
5. Verificación de programas concurrentes
5.1 Introducción
Como se puede demostrar que un programa cumple determinada propiedad?

- Posibilidad: realizar diferentes ejecuciones del programa y comprobar que se verifica la propiedad
  Problema: solo permite considerar un número limitado de historias de ejcución y no demuestra que no existan casos indeseables

- Enfoque operacional: análisis exhaustivo de casos. Se chequea la corrección de todas las posibles historias.
  Problema: su utilidad es limitada ya que con PC complejos el numero de interfoliaciones crece exponencialmente.

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."
Tiene una menor Complejidad ya que 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.

Tema 2
Sincronización en memoria compartida

1.Introducción a la sincronización en memoria compartida

Vamos a estudiar soluciones para exclusión mutua y sincronización basadas en el uso de memoria compartida entre los
procesos involucrados. Este tipo de soluciones se pueden dividir en dos categorías:

- Soluciones de bajo nivel con espera ocupada están basadas en programas que contienen explícitamente instrucciones de bajo nivel para lectura y escritura directamente a la memoria compartida, y bucles para realizar las esperas.
- Soluciones de alto nivel partiendo de las anteriores, se diseña una capa software por encima que ofrece un interfaz para las aplicaciones. La sincronización se consigue bloqueando un proceso cuando deba esperar.

Soluciones de bajo nivel:

Cuando un proceso debe esperar a que ocurra un evento o sea cierta determinada condición, entra en un bucle indefinido donde continuamente comprueba si la situación ya se da o no (a esto se le
llama espera ocupada). Este tipo de soluciones se pueden dividir en dos categorías:

- 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:

Las soluciones de bajo nivel con espera ocupada se prestan a errores, producen algoritmos complicados y tienen un impacto negativo en la eficiencia de uso de la CPU (por los bucles). En las 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. Veremos algunas de estas soluciones:

- Semáforos: se construyen sobre las soluciones de bajo nivel, usando servicios del SO que dan la capacidad de bloquear y reactivar procesos.
- Regiones críticas condicionales: son soluciones de más alto nivel que los semáforos, y que se pueden implementar sobre ellos.
- Monitores: son soluciones de más alto nivel que las anteriores y se pueden implementar en algunos lenguajes orientados a objetos como Java o Python.

2. Soluciones software con espera ocupada para EM

2.1. Introducción

Veremos diversas soluciones para lograr exclusión mutua en una sección crítica usando variables compartidas entre los procesos o hebras involucrados.

- Estos algoritmos usan dichas variables para hacer espera ocupada cuando sea necesario en el protocolo de entrada.
- Los algoritmos que resuelven este problema no son triviales, y menos para más de dos procesos. En la actualidad hay distintas soluciones con distintas propiedades. Veremos estos dos algoritmos:
  - Algoritmo de Dekker (para 2 procesos)
  - Algoritmo de Peterson (para 2 y para un número arbitrario de procesos).
- Previamente a esos algoritmos, veremos la estructura de los procesos con secciónes críticas y las propiedades que deben cumplir los algoritmos.

2.2. Estructura de los procesos con secciones críticas

Para analizar las soluciones a EM asumimos que un proceso que incluya un bloque considerado como sección crítica (SC) tendrá dicho bloque estructurado en tres etapas:

1. Protocolo de entrada (PE): una serie de instrucciones que incluyen posiblemente espera, en los casos en los que no se pueda conceder acceso a la sección crítica.
2. Sección crítica (SC): instrucciones que solo pueden ser ejecutadas por un proceso.
3. Protocolo de salida (PS): instrucciones que permiten que otros procesos puedan conocer que este proceso ha terminado la sección crítica.

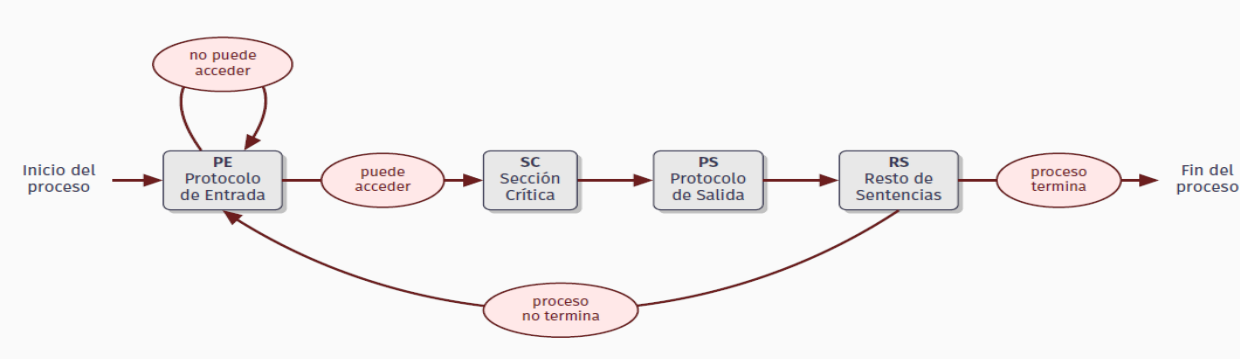
Todas las sentencias que no son parte de estas tres etapas se llaman Resto de Sentencias (RS) .

Un proceso puede tener más de una sección crítica, y cada una puede estar desglosada en varios bloques de código separados. Para simplificar el análisis, suponemos:

- Cada proceso tiene una única sección crítica (SC).
- La SC es un único bloque contiguo de instrucciones.
- Se ejecuta un bucle infinito, con dos pasos:
  - Sección crítica (con el PE antes y el PS después)
  - Resto de sentencias: se emplea un tiempo arbitrario no acotado, e incluso el proceso puede finalizar en esta sección, de forma prevista o imprevista.

Es el caso más general: no se supone nada acerca de cuantas veces un proceso puede intentar entrar en una SC.

Un proceso cualquiera pasa por los siguientes estados:



- El proceso únicamente puede terminar cuando se encuentra en Resto de sentencias,
- El código del Protocolo de Entrada introduce esperas (mediante bucles) para aseguar exclusión mútua en la Sección Crítica.

Para que se puedan implementar soluciones correctas para la EM suponemos:
Los procesos siempre terminan una sección crítica y emplean un intervalo de tiempo finito desde que la comienzan hasta que la terminan. Es decir, 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.

2.2.Propiedades para exclusión mútua

Para que un algoritmo para EM sea correcto, deben cumplir las tres propiedades mínimas: (Exclusión mutua, progreso y espera limitada)
Además, hay propiedades adicionales que también deben cumplirse:
(Eficiencia, Equidad)
Si bien consideramos correcto un algoritmo que no sea muy eficiente o para el que no pueda demostrarse claramente la equidad.

Propiedad de exclusión mútua:
Es la propiedad fundamental para el problema de la sección crítica. Establece que en cada instante de tiempo, y para cada sección crítica existente, habrá un proceso ejecutando alguna sentencia de dicha región crítica.
En esta sección veremos soluciones de memoria compartida que permiten un único proceso en una sección crítica.
Si bien esta es la propiedad fundamental, no puede conseguirse de cualquier forma, y para ello se establecen las otras dos condiciones mínimas que vemos a continuación.

Propiedad de progreso:
Consideremos una SC en un instante en que no hay ningún proceso ejecutándola, pero sí hay procesos en el PE compitiendo por entrar a la SC. La propiedad de progreso establece que un algoritmo de EM debe estar diseñado de forma que:
- 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.
- La selección del proceso anterior es 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, cualquier otro proceso activo puede entrar un número arbitrario de veces n a ese mismo PE y lograr acceso a la SC. La propiedad de espera limitada establece que: Un algoritmo de EM debe estar diseñado de forma que n no será superior a un valor máximo determinado, es decir, las esperas en el PE serán finitas (suponiendo que los procesos emplean un tiempo finito en la SC).

Las propiedades deseables son estas dos:
- 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 caso de que haya varios procesos compitiendo por acceder a una SC (de forma repetida en el tiempo), no debería existir la posibilidad de que se perjudique a algunos y se beneficie a otros.

2.3.Refinamiento sucesivo de Dijkstra

El Refinamiento sucesivo de Dijskstra hace referencia a una serie de algoritmos que intentan resolver el problema de la EM.
- Se comienza desde una versión muy simple, aunque no cumple alguna de las propiedades, y se hacen 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.
- Por simplicidad, veremos algoritmos para 2 procesos únicamente.
  - Se asume que hay dos procesos, denominados P0 y P1, cada uno de ellos ejecuta un bucle infinito conteniendo: protocolo de entrada, sección crítica, protocolo de salida y otras sentencias del proceso.

Versión 1

```
{ variables compartidas y valores iniciales }
var p01sc : boolean := false ; { indica si la SC esta ocupada }
```

```
process P0 ;
begin
   while true do begin
      while p01sc do begin end
      p01sc := true ;
      { sección crítica }
      p01sc := false ;
      { resto sección }
   end
end
```

```
process P1 ;
begin
   while true do begin
      while p01sc do begin end
      p01sc := true ;
      { sección crítica }
      p01sc := false ;
      { resto sección }
   end
end
```

No cumple la EM, pues ambos procesos pueden estar en SC

Versión 2

```
{ variables compartidas y valores iniciales }
var turno0 : boolean := true ; { podría ser también |false| }
```

```
process P0 ;
begin
   while true do begin
      while not turno0 do begin end
      { sección crítica }
      turno0 := false ;
      { resto sección }
   end
end
```

```
process P1 ;
begin
   while true do begin
      while turno0 do begin end
      { sección crítica }
      turno0 := true ;
      { resto sección }
   end
end
```

Se cumple la propiedad de Em ya que si un proceso está en SC ha logrado pasar el bucle del PE y la variable turno0 tiene un valor que forzosamente hace esperar al otro, pero no se cumple la propiedad de progreso en la ejecución. Se debe aque obliga a los procesos a acceder de forma alterna a la sección crítica. En caso de que un proceso quiera acceder dos veces seguidas sin que el otro intente acceder más, la segunda vez quedará esperando indefinidamente.

Versión 3

```
{ variables compartidas y valores iniciales }
var p0sc  : boolean := false ; { verdadero solo si proc. 0 en SC }
    p1sc  : boolean := false ; { verdadero solo si proc. 1 en SC }
```

```
process P0 ;
begin
   while true do begin
      while p1sc do begin end
      p0sc := true ;
      { sección crítica }
      p0sc := false ;
      { resto sección }
   end
end
```

```
process P1 ;
begin
   while true do begin
      while p0sc do begin end
      p1sc := true ;
      { sección crítica }
      p1sc := false ;
      { resto sección }
   end
end
```

Se cumple la propiedad de progreso: los procesos no tienen que entrar de forma

alterna, al usar dos variables independientes pero no se cumple la EM, por motivos parecidos a la primera versión.

Versión 4

```
{ variables compartidas y valores iniciales }
var p0sc  : boolean := falso ; { verdadero solo si proc. 0 en PE o SC }
    p1sc  : boolean := falso ; { verdadero solo si proc. 1 en PE o SC }
```

```
process P0 ;
begin
   while true do begin
      p0sc := true ;
      while p1sc do begin end
      { sección crítica }
      p0sc := false ;
      { resto sección }
   end
end
```

```
process P1 ;
begin
   while true do begin
      p1sc := true ;
      while p0sc do begin end
      { sección crítica }
      p1sc := false ;
      { resto sección }
   end
end
```

Sí se cumple la E.M, también se permite el entrelazamiento con regiones no críticas, ya que si un proceso accede al PE cuando el otro no está en el PE ni en el SC, el primero logrará entrar a la SC. pero no se cumple el progreso en la ejecución, ya que puede ocurrir interbloqueo.

Versión 5

```
var p0sc : boolean := false ; { true solo si proc. 0 en PE o SC }
    p1sc : boolean := false ; { true solo si proc. 1 en PE o SC }
```

```
process P0 ;
begin
   while true do begin
     p0sc := true ;
     while p1sc do begin
       p0sc := false ;
       { espera durante un tiempo }
       p0sc := true ;
     end
     { sección crítica }
     p0sc := false ;
     { resto sección }
   end
end
```

```
process P1 ;
begin
   while true do begin
     p1sc := true ;
     while p0sc do begin
       p1sc := false ;
       { espera durante un tiempo }
       p1sc := true ;
     end
     { sección crítica }
     p1sc := false ;
     { resto sección }
   end
end
```

Se cumple exclusión mutua pero no es imposible que se produzca interbloqueo en el PE: no se cumple la propiedad de progreso. Esta es pequeña, y depende de cómo se seleccionen las duraciones de los tiempos de la *espera de cortesía*, de cómo se implemente dicha espera, y de la metodología usada para asignar la CPU a los procesos o hebras
a lo largo del tiempo.
Por ejemplo, el interbloqueo podría ocurrir si ocurre que:

- Hay una CPU y la espera de cortesía es espera ocupada.
- Los dos procesos acceden al bucle de las líneas 4-8 (ambas variables están a verdadero).
- Cuando un proceso está en la CPU y ha terminado de ejecutar la asignación de la línea 8, la CPU se le asigna al otro

2.4.Algoritmo de Dekker

El algoritmo de Dekker es correcto (cumple las propiedades mínimas), y se puede interpretar como el resultado final del refinamiento sucesivo de Dijkstra:

- Cada proceso incorpora una espera de cortesía(Versión 5) durante la cual le cede al otro la posibilidad de entrar en SC, cuando ambos coinciden en el PE.
- Para evitar interbloqueos, la espera de cortesía solo la realiza uno de los procesos, de forma alterna, mediante la variable turno (parecido a versión 2)
- La variable de turno permite saber cuando acabar la espera de cortesía, implementada mediante un bucle (espera ocupada).

```
{ 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 cortes
```

```
process P0 ;
begin
   while true do begin
      p0sc := true ;
      while p1sc do begin
         if not turno0 then begin
            p0sc := false ;
            while not turno0 do
               begin end
            p0sc := true ;
         end
      end
      { sección crítica }
      turno0 := false ;
      p0sc := false ;
      { resto sección }
   end
end
```

```
process P1 ;
begin
   while true do begin
      p1sc := true ;
      while p0sc do begin
         if turno0 then begin
            p1sc := false ;
            while turno0 do
               begin end
            p1sc := true ;
         end
      end
      { sección crítica }
      turno0 := true ;
      p1sc := false ;
      { resto sección }
   end
end
```

2.5.Algoritmo de Peterson

Este algoritmo es tambien correcto para EM y más simple que el algoritmo de Dekker.

- Usa 2 variables lógicas que expresan la presencia del proceso en el PE o la SC y una variable de turno para romper el interbloqueo en caso de acceso simultáneo al PE.
- La asignación de turno se hace al inicio del PE en lugar de en el PS, con lo cual, en caso de acceso simultáneo al PE, el segundo proceso en ejecutar la asignación (atómica) al turno da preferencia al otro (el primero en llegar).
- A diferencia del algoritmo de Dekker, el PE no usa dos bucles anidados, sino que unifica ambos en uno solo.

```
{ 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 cortes
```

```
process P0 ;
begin
   while true do begin
      p0sc := true ;
      turno0 := false ;
      while p1sc and not turno0 do
         begin end
      { sección crítica }
      p0sc := false ;
      { resto sección }
   end
end
```

```
process P1 ;
begin
   while true do begin
      p1sc := true ;
      turno0 := true ;
      while p0sc and turno0 do
         begin end
      { sección crítica }
      p1sc := false ;
      { resto sección }
   end
end
```

3.Soluciones Hardware con espera ocupada (cerrojos) para EM

3.1.Introducción

Los cerrojos son una solución basada en espera ocupada usados en procesos concurrentes con memoria compartida para solucionar el problema de la EM.

- 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

Veremos una solución elemental que sin embargo es incorrecta e ilustra la necesidad de instrucciones hardware específicas (u otras soluciones más elaboradas).

TestAndSet es una instrucción máquina disponible en el repertorio de algunos procesadores.
-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

-Durante su ejecución, ninguna otra instrucción ejecutada por otro proceso puede leer ni escribir la variable lógica: por tanto, se ejecuta de forma atómica

Los cerrojos constituyen una solucion válida para EM que consume poca memoria y es eficiente en tiempo (excluyendo las esperas ocupadas), sin embargo:

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

Los cerrojos por tanto tienen un uso restringido por razones de:

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

4.Semáforos para sincronización

4.1.Estructura y operaciones de los semáforos

Los semáforos son un mecanismo que aminora los problemas de las soluciones de bajo nivel, y tienen un ámbito de uso más amplio:

- 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 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.

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

- Un conjunto de procesos bloqueados (procesos esperando al semáforo).
- Un valor natural (valor entero no negativo), al que llamamos 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

Solo hay dos operaciones básicas que se realizan sobre una variable u objeto 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, 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.

En un semáforo cualquiera, estas operaciones se ejecutan en exclusión mutua sobre cada semáforo, es decir, no puede haber dos procesos distintos ejecutando

estas operaciones a la vez sobre un mismo semáforo (excluyendo el período de bloqueo que potencialmente conlleva la llamada a sem_wait).

(LEER PDF PAGINAS 60-71)

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

- los problemas más complejos de sincronización se resuelven de forma más compleja (difícil verificar su validez y es fácil que sean incorrectos).
- al igual que los cerrojos, programas erróneos pueden provocar que haya procesos bloqueados indefinidamente o en estados incorrectos.

5.Monitores como mecanismo de alto nivel

5.1.Introducción. Definición de monitor

Hay algunos inconvenientes de usar mecanismos como los semáforos:

- Basados en variables globales, es decir, impide un diseño modular y reduce la escalabilidad (incorporar más procesos al programa requiere revisión de las variables globales).
- El uso y función de las variables no se hace explícito en el programa, lo cual dificulta razonar sobre la corrección de los programas.
- Las operaciones se encuentran dispersas y no protegidas (posibilidad errores).

Por tanto, es necesario un mecanismo que permita el acceso estructurado y la encapsulación, y que además proporcione herramientas para garantizar la exclusión mutua e implementar condiciones de sincronización.

El Monitor es un mecanismo de alto nivel que permite definir objetos abstractos compartidos, que incluyen:

- Una colección de variables encapsuladas (datos) que representan un recurso compartido por varios procesos.
- Un conjunto de procedimientos para manipular el recurso: afectan a las variables encapsuladas.

Ambos conjuntos de elementos permiten al programador invocar a los procedimientos de forma que en ellos

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

Acceso estructurado y encapsulación:

- El proceso solo puede acceder al recurso mediante un conjunto de operaciones.
- El proceso ignora las variables que representan al recurso y la implementación de las operaciones asociadas.

Exclusión mutua en el acceso a los procedimientos

- La exclusión mutua en el acceso a los procedimientos del monitor está garantizada por definición.
- La implementación del monitor garantiza que nunca dos procesos estarán ejecutando simultáneamente algún procedimiento del monitor.

Los monitores con preferibles respecto de los semáforos ya que facilita el diseño e implementación de programas libres de errores

- Las variables están protegidas: solo pueden leerse o modificarse desde el código del monitor, no desde cualquier punto de un programa que puede tener miles de líneas de código.
- La exclusión mutua está garantizada:no hay que usar mecanismos explícitos de exclusión mutua en el acceso a las variables compartidas.
- Las operaciones de esperas bloqueadas y de señalización se programan dentro del monitor (es más fácil verificar que el diseño es correcto)

Componentes:
Variables permanentes: son el estado interno del monitor.

- Sólo pueden ser accedidas dentro del monitor (en el cuerpo delos 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.

Ventaja: la implementación de las operaciones se puede cambiar sin modificar su semántica.
En algunos casos es conveniente crear múltiples instancias independientes de un monitor

- Cada instancia tiene sus variables permanentes propias.
- La E.M. ocurre en cada instancia por separado.
- Esto facilita mucho escribir código reentrante.

5.2.Funcionamiento de los monitores

Cuando un proceso necesita operar sobre un recurso compartido controlado por un monitor deberá realizar una llamada a uno de los procedimientos exportados por el monitor usando los parámetros actuales apropiados:

- Mientras el proceso está ejecutando algún procedimiento del monitor decimos que el proceso está dentro del monitor.

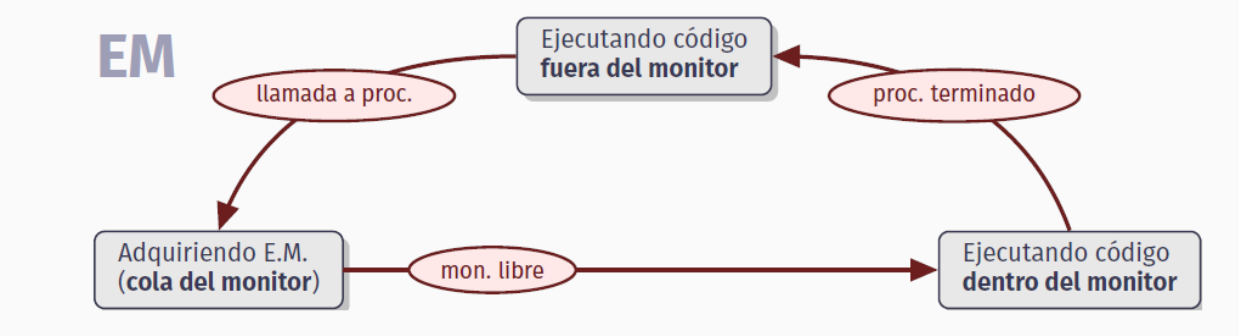
Exclusión mutua: Si un proceso P está dentro de un monitor, cualquier otro proceso que llame a un procedimiento de ese monitor deberá esperar hasta que P salga:

- Esta política de acceso asegura que las variables permanentes nunca son accedidas concurrentemente.
- El acceso exclusivo entre los procedimientos del monitor debe estar garantizado en la implementación de los monitores

Después de ejecutarse el código de inicialización, un monitor es un objeto pasivo y el código de sus procedimientos sólo se ejecuta cuando son invocados por los procesos.

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

- Si un proceso está en el monitor y otro proceso intenta ejecutar un procedimiento de este, el último queda bloqueado y se inserta en la cola del monitor
- Cuando un proceso abandona el monitor (fin 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 la cola de procesos esperando a comenzar a ejecutar el código del mismo

5.3.Sincronización en monitores

Para implementar la sincronización, se requiere de una facilidad para que los procesos hagan esperas bloqueadas, hasta que sea cierta determinada condición:

- 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 sólo se dispone de sentencias de bloqueo y activación y 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. A esas variables las llamamos señales o variables condición:

- Cada variable condición tiene asociada una lista o cola (inicialmente vacía) de procesos en espera hasta que la condición se haga cierta.
- Para una cualquiera de estas variables, un proceso puede invocar:
  - wait: Estoy esperando a que alguna condición ocurra.
  - signal: Estoy señalando que una condición ocurre.

Dada una variable condición cond, se definen al menos las siguientes operaciones asociadas a 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*) evitando la inanición: Cada proceso en cola obtendrá su turno.
- cond.queue(): Función lógica que devuelve true si hay algún proceso esperando en la cola de cond, y false en caso contrario.

Dado que los procesos pueden estar dentro del monitor, pero bloqueados:

- Cuando un proceso llama a wait y queda bloqueado, se debe liberar la exclusión mutua del monitor, si no se hiciese, se produciría interbloqueo con seguridad (ese proceso quedaría quedaría bloqueado y el resto también cuando intentasen después entrar al monitor).
- Cuando un proceso es reactivado después de una espera, adquiere de nuevo la exclusión mutua antes de ejecutar la sentencia siguiente a wait.
- Más de un proceso podrá estar dentro del monitor, aunque solo uno de ellos estará ejecutándose, el resto estarán bloqueados en variables condición.

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 programador no conoce a priori la interfoliación concreta de llamadas a los procedimientos del monitor. El enfoque de verificación que vamos a seguir utiliza un invariante de monitor:

- Es una propiedad que el monitor cumple siempre (parecido al invariante de un semáforo, pero específico de cada monitor diseñado por un programador
- Unido a las propiedades de los procesos concurrentes que invocan al monitor, facilita la verificación de los programas

El Invariante del Monitor (IM)

- Es un 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 del monitor y de los valores de las variables permanentes de dicho monitor.
- La traza del monitor es la secuencia (ordenada en el tiempo) de llamadas a procedimientos del monitor ya completadas, desde el inicio hasta llegar al estado indicado (se incluyen las llamadas de todos los procesos del 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).

El invariante del monitor se escribe como un predicado

- Determina que trazas y/o estados son posibles en el monitor.
- Puede incluir referencias a las variables y/o a la traza, por ejemplo, al número de veces que se ha ejecutado un procedimiento, o al último procedimiento ejecutado.

El invariante del monitor debe ser cierto

- justo después de la inicialización de las variables permanentes.
- antes y después de cada llamada a un procedimiento del monitor
- antes y después de cada operación wait.
- antes y después de cada operación signal.

Justo antes de signal sobre una variable condición, además, debe ser cierta la condición lógica asociada a dicha variable.

5.5.Ejemplos de monitores

(Ver diapositivas 101-140)

5.6.Colas de prioridad

Por defecto, se usan colas de espera FIFO. Sin embargo a veces resulta útil disponer de un mayor control sobre la estrategia de planificación, dando la prioridad del proceso en espera como un parámetro entero de wait

- La sintaxis de la llamada es: cond.wait(p), donde p es un entero no negativo que refleja la prioridad.
- cond.signal() reanudará un proceso que especificó el valor mínimo de p de todos los que esperan (si hay más de uno con prioridad mínima, se usa 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

5.7.Implementación de monitores

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 cuantos 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.