SISTEMAS OPERATIVOS II

TEMA 3. Sincronización entre Procesos

Área de Arquitectura y Tecnología de Computadores
Escuela Universitaria Politécnica de Teruel
http://????.????.es/SOII/

Sincronización entre procesos

- **Bibliografía:**
  - Silberschatz & Galvin: Sistemas Operativos
    - Capítulo 6: Sincronización de procesos

Sincronización entre procesos

- **Objetivos**:
  - Entender el problema de la sección crítica
  - Entender los requisitos que debe cumplir la solución al problema de la sección crítica
  - Conocer las soluciones hardware y software del problema de la sección crítica
  - Conocer los tres problemas clásicos de sincronización
  - Conocer algunos mecanismos de sincronización de alto nivel

Sincronización entre procesos

- Introducción
- El problema de la sección crítica
- Sincronización hardware
- Semáforos
- Problemas clásicos de sincronización
- Regiones críticas
- Monitores

Introducción

- **"El modelo de procesos"**
  - Sistema con un conjunto de procesos secuenciales cooperantes que se ejecutan asíncronamente y pueden compartir datos
  - **Procesos independientes:**
    - Son procesos que no afectan ni pueden ser afectados por otros procesos en ejecución
      - **Ejemplo**: Procesos que se ejecutan en diferentes máquinas no conectadas en red

Introducción

- **"El modelo de procesos"**
  - **Procesos cooperantes**:
    - Son procesos que pueden afectar o ser afectados por la ejecución de otros procesos
      - **Ejemplos**:
        - Proceso que comparten código y datos: threads
        - Procesos que comparten datos a través del sistema de ficheros
    - Propiedades
      - Comportamiento no determinista
      - Puede no ser reproducible su ejecución

Introducción

- **"El modelo de procesos"**
  - **Problema:** "La concurrencia"
    - El acceso concurrente a datos compartidos puede llevar a la inconsistencia de los mismos
    - **Condición de carrera**: situación en la que varios procesos acceden a, y manipulan, los mismos datos de forma concurrente, y el resultado de la ejecución depende del orden en el que se produce el acceso
  - **Solución:** "Sincronización"
    - Para mantener la consistencia de los datos debemos asegurar la ejecución ordenada de los procesos cooperantes

Introducción

- **Ejemplo:** Problema de un productor consumidor con buffer limitado

Proceso productor

```
repeat
    . . .
    produce item en var nextp
    . . .
    send(consumidor, nextp);
until false;
```

Proceso consumidor

```
repeat
    receive(productor, nextc);
    . . .
    consume item en var nextc
    . . .
until false;
```

Introducción

- **Ejemplo:** Problema de un productor consumidor con buffer limitado
- Primera solución: Memoria compartida

Datos compartidos

```
var n;
type item = … ;
var buffer : array[0 … n-1] of item;
in, out : 0 … n-1;

int := 0;
out := 0;
```


Introducción

- **Ejemplo:** Problema de un productor consumidor con buffer limitado

Proceso productor

```
repeat
        … … …
        produce un item en var nextp
        … … …
        while in+1 mod n = out do no-op;
        buffer[in] := nextp;
        in := in+1 mod n;
until false;
```


Introducción

- **Ejemplo:** Problema de un productor consumidor con buffer limitado

Proceso consumidor

```
repeat
        while in = out do no-op;
        nextc := buffer[out];
        out := out+1 mod n;
        … … …
        consume un item en var nextc
        … … …
until false;
```

Problema: sólo se utilizan n-1 posiciones del buffer

Introducción

- **Ejemplo:** Problema de un productor consumidor con buffer limitado
- Segunda solución: Memoria compartida

Datos compartidos

```
var n;
type item = … ;
var buffer : array[0 … n-1] of item;
in, out : 0 … n-1;
counter : 0 … n;

int := 0;
out := 0;
counter := 0;
```

Introducción

- **Ejemplo:** Problema de un productor consumidor con buffer limitado

```
                Proceso productor
repeat
        …   …   …
        produce un item en var nextp
        …   …   …
        while counter =  n do no-op;
        buffer[in] := nextp;
        in := in+1 mod n;
        counter := counter + 1;
until false;
```


Introducción

- **Ejemplo:** Problema de un productor consumidor con buffer limitado

```
                Proceso consumidor
repeat
        while counter = 0 do no-op;
        nextc := buffer[out];
        out := out+1 mod n;
        counter := counter - 1;
        …   …   …
        consume un item en var nextc
        …   …   …
until false;
```


Introducción

- Si productor y consumidor se ejecutan concurrentemente, ¿la solución es válida?

NO

```
counter := counter + 1;          counter := counter - 1;
    reg1 := counter;     P(T1)       reg1 := counter;     C(T1)
    reg1 := reg1 + 1;    P(T2)       reg1 := reg1 - 1;    C(T2)
    counter := reg1;     P(T3)       counter := reg1;     C(T3)
```

```
Posible ejecución:   Posible ejecución:   Posible ejecución:
    P(T1)                P(T1)                P(T1)
    P(T2)                P(T2)                P(T2)
    P(T3)                C(T1)                C(T1)
    C(T1)                C(T2)                C(T2)
    C(T2)                P(T3)                C(T3)
    C(T3)                C(T3)                P(T3)
```


El problema de la sección crítica

- Considerar un sistema formado por n procesos: $\{P_0, P_1, \ldots \ldots, P_{n-1}\}$ que usan una serie de datos compartidos
- **Def.-** Se denomina ***sección crítica*** al trozo de código de cada proceso en el que se accede a los datos compartidos
- **Característica del sistema**:
  - Cuando un proceso está ejecutando el código de su sección crítica, ningún otro proceso puede ejecutar el código de su sección crítica

El problema de la sección crítica

- **Hipótesis generales**:
  - Cada proceso se ejecuta a velocidad no nula
  - No se hace ninguna hipótesis sobre la velocidad relativa de los n procesos
  - Las instrucciones de lenguaje máquina se ejecutan atómicamente, cada una de forma indivisible

Problema de la sección crítica
≅
Garantizar la exclusión mutua entre los n procesos durante la ejecución de su sección crítica

El problema de la sección crítica

- **Solución:**
  - Diseñar un “protocolo”, que los procesos puedan usar, para permitir la cooperación entre ellos
  - El “protocolo” debe asegurar que cuando un proceso esté ejecutando su sección crítica, ningún otro proceso tenga permiso para ejecutar su sección crítica

El problema de la sección crítica

- Estructura del proceso P_i

```
repeat

        entrada a la sección crítica
                Sección crítica
        salida de la sección crítica

        Trabajo propio

until false;
```


El problema de la sección crítica

- Condiciones que debe satisfacer una solución al problema de la sección crítica:
  - Exclusión mutua
  - Progreso
  - Espera limitada

El problema de la sección crítica

- **Exclusión mutua**
  - Si P_i está ejecutando su sección crítica, ningún otro proceso puede estar ejecutando su sección crítica
- **Progreso**
  - Si ningún proceso está ejecutando su sección crítica y hay procesos que desean entrar en su sección crítica, la selección del proceso que podrá entrar en ella NO será pospuesta indefinidamente
- **Espera limitada**
  - Cuando un proceso realiza una solicitud para entrar en su sección crítica, el número de veces que otros procesos pueden entrar en sus secciones críticas antes de que sea atendida la solicitud del primer proceso está acotado

El problema de la sección crítica

- **Algoritmo 1 (Dijkstra 1965)**

  Solución para dos procesos

Sean los procesos P_0 y P_1 .
Notación: P_i, P_j son procesos tal que j = i - 1

Datos compartidos

var turno: (0, 1);

turno = 0;

/* Inicialmente turno = 0 */
/* turno = i $\Rightarrow$ P_i puede entrar en su sección crítica */

El problema de la sección crítica

- **Proceso P_i**

repeat

while turno $\neq$ i do no-op;
Sección crítica
turno := j;

Trabajo propio

until false;

El problema de la sección crítica

- Análisis de propiedades:
  - Exclusión mutua: **SI** (Por reducción al absurdo)

Supongamos que los dos procesos se encuentran ejecutando sus secciones críticas

Esto significa que la variable turno al ejecutar la sección de entrada, en el proceso P_i tiene el valor i = 0 $\wedge$ 1

Esto no es posible ya que turno sólo puede tener un valor y su valor sólo se cambia cuando un proceso sale de su sección crítica

El problema de la sección crítica

- Análisis de propiedades:
  - Progreso y espera limitada: **NO** (Contra-ejemplo)

    Supongamos que el proceso P_0 entra en su sección crítica

    Al salir turno = 1, permitiendo que P_1 pueda entrar en su sección crítica

    Supongamos que el proceso P_1 no desea entrar nunca en su sección crítica

    Si P_0 desea volver a entrar en su sección crítica, nunca lo conseguirá y no podrá acabar su trabajo

El problema de la sección crítica

- **Algoritmo 2 (Dijkstra 1965)**

  Solución para dos procesos

Sean los procesos P_0 y P_1
Notación: P_i, P_j son procesos tal que j = i - 1

Datos compartidos

```
var flag: array[0, 1] of boolean;

flag[0] := false;
flag[1] := false;

/* Inicialmente flag[0] = flag[1] = false;     */
/* flag[i] := true ⇒ Pi  puede entrar en su sección crítica */
```


El problema de la sección crítica

- **Proceso P_i**

```
repeat

        flag[i] := true;
        while flag[j] do no-op;
                Sección crítica
         flag[i] := false;

        Trabajo propio

until false;
```


El problema de la sección crítica

- Análisis de propiedades:
  - Exclusión mutua: **SI** (Por reducción al absurdo)

    Supongamos que los dos procesos se encuentran ejecutando sus secciones críticas

    Esto significa que flag[0] = false, flag[1] = false

    Esto no es posible ya que si flag[i] = false para que el proceso P_j se encuentre en su sección crítica, el proceso P_i no ha podido pasar por su sección de entrada y por lo tanto no se encuentra en su sección crítica

El problema de la sección crítica

- Análisis de propiedades:
  - Progreso y espera limitada: **NO** (Contra-ejemplo)

    Supongamos que el proceso P_0 y P_1 ejecutan la sentencia de asignación del código de entrada a su sección crítica:

    flag[0] = true

    flag[1] = true

    A partir de este momento ambos procesos se quedan permanentemente ejecutando el bucle de la sección de entrada sin que ninguno pueda alcanzar la sección crítica

    (Inter-bloqueo mutuo)

El problema de la sección crítica

- **Algoritmo 3 (Peterson 1981)**

  Solución para dos procesos

Sean los procesos P_0 y P_1

Notación: P_i, P_j son procesos tal que j = i - 1

Datos compartidos

```
var turno: (0, 1);
var flag: array[0, 1] of boolean;

turno := 0;
flag[0] := false;
flag[1] := false;
```

/* (turno = i and flag[i] := true) $\Rightarrow$ P_i puede entrar en su sección crítica */

El problema de la sección crítica

- **Proceso P_i**

```
repeat

        flag[i] := true;
        turno := j;
        while (flag[j] and turno = j) do no-op;
                Sección crítica
         flag[i] := false;

        Trabajo propio

until false;
```


El problema de la sección crítica

- Análisis de propiedades:
  - P_i entra en su sección crítica sólo si flag[j] = false $\vee$ turno = i
  - Exclusión mutua: **SI** (Por reducción al absurdo)

    Supongamos que los dos procesos se encuentran ejecutando sus secciones críticas

    La condición que se debe verificar para que los dos procesos se encuentren en la sección crítica es:

    (flag[1] = false $\vee$ turno = 0) and (flag[0] = false $\vee$ turno = 1)

    Demostrando que esta condición no puede ser cierta, llegaremos a una contradicción

El problema de la sección crítica

- Análisis de propiedades:
  - Exclusión mutua: **SI** (Por reducción al absurdo)
    - turno = 0 and turno = 1

      Una variable sólo puede tener un valor.

      Si ambos procesos tratan de entrar simultáneamente en su sección crítica, de las dos asignaciones que se realizan a esta variable una será la que se realiza en último lugar, impidiendo acceder a la sección crítica al proceso que la realizó y permitiendo acceder al otro proceso

El problema de la sección crítica

- Análisis de propiedades:
  - Exclusión mutua: **SI** (Por reducción al absurdo)
    - flag[0] = false and flag[1] = false

      Esto significa que ningún proceso ha podido pasar por su sección de entrada y por lo tanto no se encuentra en la sección crítica

    - flag[i] = false and turno = i

      Si flag[i] = false, el proceso P_i no ha pasado por su sección de entrada y por lo tanto no se encuentra en su sección crítica

El problema de la sección crítica

- Análisis de propiedades:
  - Progreso: **SI**

    Si P_i desea acceder a su sección crítica y está ejecutando el bucle "while", la condición lógica que le impide abandonarlo es:
    (flag[j] = true and turno = j)

    - flag[j] = true ⇒

      P_j está en su sección de entrada o en su sección crítica

    - turno = j ⇒

      Si P_j está en su sección de entrada antes de ejecutar la sentencia de asignación turno = i; al ejecutarla permitirá que P_i entre en su sección crítica

      Si P_j está en su sección crítica, al abandonarla P_i entrará en su sección crítica ya que flag[j] = false

El problema de la sección crítica

- Análisis de propiedades:
  - Espera limitada: **SI**

    De acuerdo con la demostración de la propiedad anterior, un proceso esperará, como máximo, para entra en su sección crítica una ejecución de la sección crítica de otro proceso

El problema de la sección crítica

- **Algoritmo de la panadería:** "The bakery algorithm"
  Solución para n procesos
  - Bases del algoritmo:
    - Todo proceso que desea entrar en la sección crítica recibe un número
    - El proceso que posee el menor número cuando la sección crítica está libre será el que accederá a la misma

Relación de orden:

$P_i \Rightarrow$ (numero[i], PID)

(a, b) < (c, d) $\Leftrightarrow$ (a < c) or ((a = c) and (b < d))

max(a_0 , … … a_{n-1}) es un número k tal que k $\geq a_i$ $\forall$ i = 0, … …, n-1

El problema de la sección crítica

- **Algoritmo de la panadería:** "The bakery algorithm"
  Solución para n procesos
  - Bases del algoritmo:
    - Todo proceso que desea entrar en la sección crítica recibe un número
    - El proceso que posee el menor número cuando la sección crítica está libre será el que accederá a la misma
    - El esquema de generación de números garantiza que estos son generados en orden no decreciente
      - Ejemplo: 1, 2, 3, 3, 3, 4, 5, 6, 6, 6, 7, …
    - Si dos procesos P_i y P_j reciben el mismo número, si i < j entonces P_i ejecutará su sección crítica, en caso contrario P_j ejecutará su sección crítica

El problema de la sección crítica

- **Algoritmo de la panadería**
  Solución para n procesos

Datos compartidos

var elección: array[0, … …, n-1] of boolean;

var numero: array[0, … …, n-1] of integer;

$\forall$ i
elección[i] := false;
numero[i] := 0;

El problema de la sección crítica

- **Proceso P_i**

```
repeat
    eleccion[i] := true;
    numero[i] := max(numero[0], … …, numero[n-1]) + 1;
    eleccion[i] := false;
    for j := 0 to n-1 do
    begin
       while eleccion[j] do no-op;
       while (numero[j] ≠ 0 and (numero[j], j) < (numero[i], i)) do no-op;
    end;
      Sección crítica
    numero[i] := 0;

    Trabajo propio

until false;
```

Sincronización hardware

- Las características del hardware pueden facilitar la tarea de programación y mejorar la eficiencia del sistema
- El hardware puede simplificar la programación de la sincronización entre procesos
- **"Primer idea"**
  - El problema de la sección crítica se puede resolver de una forma fácil si se deshabilitan las interrupciones mientras se accede a una variable compartida

Sincronización hardware

- **Inconvenientes:**
  - Esta solución no siempre es aplicable
    - Ejemplo: Sistemas multiprocesador
  - La desactivación de interrupciones en un sistema multiprocesador requiere mucho tiempo
    - Se debe esperar a que los mensajes de desactivación y activación de las interrupciones llegue a todos los procesadores
    - Esta transferencia de mensajes retarda el ingreso en cada sección crítica
    - La eficiencia del sistema disminuye
  - **Nota**: ¿Que ocurre si el reloj del sistema se mantiene actualizado mediante interrupciones?

Sincronización hardware

- **"Segunda idea"**
  - Añadir al procesador instrucciones que se ejecuten de forma atómica y permitan:
    - Hacer un test y modificar el contenido de una palabra

      Test-and-Set (TAS)
    - Intercambiar los contenidos de dos palabras

      Swap

Sincronización hardware

- **Instrucción Test-and-Set (TAS)**

```
function Test-and-Set (var target: boolean) : boolean
    begin
        Test-and-Set := target;
        target := true;
    end;
```

Sincronización hardware

- **Solución 1 :** Problema de la sección crítica

  Implementación de la exclusión mutua con TAS

Variables compartidas

```
var lock: boolean;

lock := false;
```


Sincronización hardware

- **Proceso P_i**

```
repeat

        while  Test-and-Set (lock)  do no-op;

        Sección crítica

        lock := false;

        Trabajo propio

until false;
```

Problema: Esta solución no satisface la propiedad de espera limitada

Sincronización hardware

- **Instrucción Swap**

```
function Swap (var a, b: boolean)
    var temp: boolean;

    begin
        temp := a;
        a := b;
        b := temp;
    end;
```


Sincronización hardware

- **Solución 2 :** Problema de la sección crítica

  Implementación de la exclusión mutua con Swap

Variables compartidas

```
var lock: boolean;

lock := false;
```

Sincronización hardware

- **Proceso P_i**

```
var key : boolen;

repeat
        key := true;
        repeat  Swap (lock, key)
        until key = false;

        Sección crítica

        lock := false;

        Trabajo propio
until false;
```

Problema: Esta solución no satisface la propiedad de espera limitada

Sincronización hardware

- **Solución 3 :** Problema de la sección crítica

  Satisface las propiedades de exclusión mutua, progreso y espera limitada

Variables compartidas

```
var lock: boolean;
var waiting: array[0, … …, n-1] of boolean;

lock := false;

waiting[i] := false;        /*      ∀ i = 0, … …, n-1       */
```


Sincronización hardware

- **Proceso P_i**

```
var j: 0, … …, n-1;
var key : boolen;

repeat
        waiting[i] := true;
        key = true;
        while  (waiting[i] and key)  do
            key := Test-and-Set (lock);

        waiting[i] := false;

          Sección crítica
```


Sincronización hardware

- **Proceso P_i**

```
        j := i+1 mod n;
        while  ( (j ≠ i) and (not waiting[j]) )  do
            j := j+1 mod n;

        if  (j = i)  then lock := false;
        else waiting[j] := false;

        Trabajo propio

until false;
```

Semáforos

- **Problemas de las soluciones presentadas:**
  - Presentan “espera activa”
  - Son difíciles de generalizar
- **Solución:**
  - Utilizar semáforos
  - **Def.-** Un semáforo es una herramienta de sincronización que evita una “espera activa” para acceder a la sección crítica
  - Un semáforo es una variable entera con un valor inicial
  - Sólo se puede acceder a ella mediante dos operaciones atómicas:

    wait ( ); **signal ( )**;

Semáforos

Semáforos: Son variables

Nombre: S, Mutex /* Identificador */

Tipo: Entero

Valor de inicialización: “depende del problema”

Dos operaciones:

wait (S)

```
while (S ≤ 0) do no-op      /* espera activa */
    S := S - 1;
```

signal (S)

```
S := S + 1;
```


Semáforos

- **Ejemplo 1 :** Problema de la sección crítica

Variables compartidas

semáforo: mutex;

Inicialización

mutex := 1;

Semáforos

- **Proceso P_i**

```
repeat

        wait (mutex);

                Sección crítica

        signal(mutex);

        Trabajo propio

until false;
```

Semáforos

- **Ejemplo 2 :** Ordenación de actividades concurrentes mediante sincronización de procesos

P_1 ejecuta S_1
P_2 ejecuta S_2

S_2 se debe ejecutar después de S_1

Variables compartidas

semáforo: sincronizacion;

Inicialización

sincronizacion := 0;

Semáforos

- **Proceso P_1**

```
begin
    S1;
    signal(sincronizacion);
end;
```

- **Proceso P_2**

```
begin
    wait(sincronizacion);
    S2;
end;
```


Semáforos

- **Problemas de las soluciones presentadas:**
  - Todas las soluciones planteadas que aseguran la exclusión mutua requieren “esperas activas”
  - La implementación de semáforo presentada también requiere una “espera activa”
- **Solución:**
  - Definir un semáforo como un registro (estructura de datos) que además de contener un entero contenga una lista de procesos
  - Notar que las operaciones wait( ) y signal( ) deberán ser modificadas

Semáforos

- **Implementación:**
  - Modificación de la función **wait( )**
    - Bloquea el proceso y lo sitúa en una cola de espera asociada al semáforo
  - Modificación de la función **signal( )**
    - Extrae un proceso de la cola de espera asociada al semáforo y los despierta

Semáforos

- **Implementación:**

Semáforo:

type semaforo = record

valor : integer;

L : lista de procesos;

end;

Semáforos

- **Implementación:**

wait (S)

S.valor := S.valor - 1;

if S.valor < 0 then

Añadir proceso a S.L;

Block

end;

Semáforos

- **Implementación:**

signal (S)

S.valor := S.valor + 1;

if S.valor ≤ 0 then

Extraer proceso P de S.L;

Wakeup(P);

end;

Semáforos

- **Notas:**
  - El semáforo puede tener valores negativos
    - Si S < 0 ⇒ |S| = Número de procesos esperando en wait sobre S
  - La cola de procesos que esperan para realizar wait sobre S se puede implementar con un lista de PCB's

Semáforos

- **Dos decisiones a tomar:**
  - ¿Qué política seguiremos en la selección de un proceso de los que están en la cola de espera?
    - Una estrategia FIFO nos garantiza la “espera limitada”
  - ¿Cómo resolver el problema de atomicidad de las operaciones wait( ) y signal( )?
    - Añadir instrucciones especiales hardware de sincronización para garantizar el acceso en exclusión mutua al semáforo

Semáforos

- **¿Cómo resolver el problema de atomicidad de las operaciones wait( ) y signal( )?**

wait(S):

```
while  Test-and-Set (lock)  do no-op;
S.valor := S.valor - 1;

if  S.valor  <  0 then
    Añadir proceso a S.L;
    lock := false;
    Block;
end;

else lock := false;
```


Semáforos

- **¿Cómo resolver el problema de atomicidad de las operaciones wait( ) y signal( )?**

signal(S):

```
while  Test-and-Set (lock)  do no-op;
S.valor := S.valor + 1;

if  S.valor  ≤  0 then
    Extraer proceso P de S.L;
    lock := false;
    Wakeup(P);
end;

else lock := false;
```


Semáforos

- **Tipos de semáforos:**
  - Semáforos contador:
    - Variable entera que toma valores en un intervalo no restringido
  - Semáforos binarios:
    - Variable entera que toma valores en el intervalo [0,1]

Semáforos

- **Ejemplo:** Realización de un semáforo contador S con semáforos binarios

```
var     S1: semáforo-binario;
        S2: semáforo-binario;
        C: integer;

/*      Inicialización de variables     */
  S1 := 1;
  S2 := 0;
  C := valor inicial de S;
```


Semáforos

- **Ejemplo:** Realización de un semáforo contador S con semáforos binarios

```
wait(S):
        wait(S1);
        C := C - 1;

        if C < 0 then
            begin
                signal(S1);
                wait(S2);
            end;

        else signal(S1);
```


Semáforos

- **Ejemplo:** Realización de un semáforo contador S con un semáforo binario

```
signal(S):
        wait(S1);
        C := C + 1;

        if C ≤ 0 then signal(S2);
        signal(S1);
```


Problemas clásicos de sincronización

- Son problemas que aparecen con mucha frecuencia en problemas de control de la concurrencia
- **Los Problemas clásicos de sincronización:**
  - El Problema del Buffer Limitado
  - Lectores y Escritores
  - Filósofos Comensales
  - **Nota:** Estos problemas sirven para probar cualquier esquema de sincronización que se proponga

Problemas clásicos de sincronización

- **El Problema del Buffer Limitado:** Enunciado
  - N procesos productores
  - M procesos consumidores
  - Los datos son producidos por los procesos productores y depositados en un buffer compartido de capacidad k
  - Los datos son consumidos por los procesos consumidores que los extraen del buffer compartido
  - El acceso al buffer compartido se realiza en excusión mutua

Problemas clásicos de sincronización

- **El Problema del Buffer Limitado:** Implementación

Datos compartidos

```
var     objetos: semáforo_contador;
        huecos: semáforo_contador;
        mutex: semáforo_binario;

/*      Inicializaciones        */

objetos := 0;
huecos := k;
mutex := 1;
```


Problemas clásicos de sincronización

- **El Problema del Buffer Limitado:** Implementación

Productor P_i

```
repeat
        Producir dato en nextp;

        wait(huecos);
        wait(mutex);

        Añadir nextp al buffer;

        signal(mutex);
        signal(objetos);

until false;
```


Problemas clásicos de sincronización

- **El Problema del Buffer Limitado:** Implementación

Consumidor C_i

```
repeat
        wait(objetos);
        wait(mutex);

        Extraer de buffer en nextc;

        signal(mutex);
        signal(huecos);

        Consumir dato en nextc:

until false;
```

Problemas clásicos de sincronización

- **El Problema del Buffer Limitado:** Propiedades a verificar
  - Nunca se desborda la capacidad del buffer
  - Se puede utilizar toda la capacidad del buffer
  - El acceso al buffer se realiza en exclusión mutua
  - Se cumple la propiedad de espera limitada
  - No se produce inanición

Problemas clásicos de sincronización

- **El Problema de los Lectores y Escritores:** Enunciado
  - Tenemos un repertorio de datos (ej. un fichero) que deben ser compartidos entre varios procesos concurrentes
  - Existen dos clases de procesos:
    - **Lectores**: Sólo leen datos del repertorio
    - **Escritores**: Actualizan los datos del repertorio
  - Un “escritor” debe acceder al repertorio de datos en exclusión mutua con cualquier otro proceso
  - Si dos o más “lectores” acceden al repertorio de datos simultáneamente no pasará nada

Problemas clásicos de sincronización

- **El Problema de los Lectores y Escritores:** Variantes
  - 1º Problema de Lectores y Escritores
    - Ningún “lector” esperará para acceder al repertorio de datos a menos que un “escritor” ya haya obtenido permiso para usarlo
    - Ningún “lector” deberá esperar a que otros “lectores” terminen cuando un “escritor” esté esperando
  - 2º Problema de los Lectores y Escritores
    - Cuando un “escritor” esté preparado para acceder al repertorio de datos, realizará su escritura lo antes posible
    - Si un “escritor” está esperando acceder al repertorio de datos, ningún “lector” nuevo puede comenzar a leer

Problemas clásicos de sincronización

- **El Problema de los Lectores y Escritores:** Notas
  - Cualquiera de las variantes presentadas puede dar lugar a inanición:
    - Variante 1º, los “escritores” podrían sufrir inanición
    - Variante 2º, los “lectores” podrían sufrir inanición

Problemas clásicos de sincronización

- **El Problema de los Lectores y Escritores:** Implementación 1ª solución

Datos compartidos

```
var     mutex: semáforo_binario;
        wrt: semáforo_binario;
        readcount: integer;

/*      Inicializaciones        */

mutex := 1;
wrt := 1;
readcount := 0;
```


Problemas clásicos de sincronización

- **El Problema de los Lectores y Escritores:** Implementación 1ª solución

Escritor E_i

```
repeat

        wait(wrt);

        Escritura en repertorio de datos;

        signal(wrt);

until false;
```


Problemas clásicos de sincronización

- **El Problema de los Lectores y Escritores:** Implementación 1ª solución

Lector L_i

```
repeat
        wait(mutex);
        readcount := readcount + 1;
        if readcount = 1 then wait(wrt);
        signal(mutex);

        Lectura en repertorio de datos;

        wait(mutex);
        readcount := readcount – 1;
        if readcount = 0 then signal (wrt);
        signal(mutex);
until false;
```


Problemas clásicos de sincronización

- **El Problema de los Lectores y Escritores:** Implementación 2ª solución

Datos compartidos

```
var     mutex: semáforo_binario;
        lectores: semáforo_contador;

/*      Inicializaciones        */

mutex := 1;
lectores := N_lectores;
```

Problemas clásicos de sincronización

- **El Problema de los Lectores y Escritores:** Implementación 2ª solución

Escritor E_i

```
repeat
        wait(mutex)
        for j := 1 to N_lectores do
            wait(lectores);

        Escritura en repertorio de datos;

        for j := 1 to N_lectores do
            signal(lectores);
        signal(mutex);
until false;
```


Problemas clásicos de sincronización

- **El Problema de los Lectores y Escritores:** Implementación 2ª solución

Lector L_i

```
repeat
        wait(mutex);
        wait(lectores);
        signal(mutex);

        Lectura en repertorio de datos;

        signal(lectores);
until false;
```


Problemas clásicos de sincronización

- **El Problema de los Lectores y Escritores:** Propiedades a verificar
  - Un proceso escritor accede siempre en exclusión mutua al repertorio de datos
  - Se cumple la propiedad de espera limitada
  - No se produce inanición

Problemas clásicos de sincronización

- **El Problema de los Filósofos Comensales:** Enunciado
  - Imaginemos cinco filósofos que es pasan la vida pensando y comiendo
  - Los filósofos comparten una mesa circular rodeada de cinco sillas, cada una de las cuales pertenece a un filósofo
  - En el centro de la mesa hay un “platón” de arroz y la mesa está puesta con cinco palillos chinos individuales
  - Cuando un filósofo piensa, no interactúa con sus colegas
  - De vez en cuando, un filósofo siente hambre y trata de coger los dos palillos chinos que le quedan más cerca (el de su derecha y el de su izquierda)

Problemas clásicos de sincronización

- **El Problema de los Filósofos Comensales:** Enunciado
  - Un filósofo sólo puede coger un palillo a la vez; obviamente no puede coger un palillo que ya está en manos de otro filósofo
  - Cuando un filósofo hambriento tiene sus dos palillos al mismo tiempo, come sin soltar sus palillos
  - Cuando termina de comer, deja sobre la mesa ambos palillos y comienza a pensar otra vez
  - **Nota**: Es una representación sencilla de la necesidad de asignar varios recursos entre varios procesos sin que haya bloqueos mutuos ni inanición

Problemas clásicos de sincronización

- **El Problema de los Filósofos Comensales:** Implementación
  - Un filósofo trata de coger un palillo ejecutando una operación **wait** sobre un semáforo y suelta un palillo ejecutando una operación **signal** sobre un semáforo

Datos compartidos

```
var     tenedor: array[0 ... 4] of semáforo_binario;

/*      Inicializaciones            */

tenedor[i] := 1;        /*      ∀ i = 0, ..., 4     */
```


Problemas clásicos de sincronización

- **El Problema de los Filósofos Comensales:** Implementación

Filósofo F_i

```
repeat
        Pensando;
        wait(tenedor[i]);
        wait(tenedor[i+1 mod 5]);
        Comiendo;
        signal(tenedor[i]);
        signal(tenedor[i+1 mod 5]);
until false;
```


Problemas clásicos de sincronización

- **El Problema de los Filósofos Comensales:** Problemas
  - La solución presentada puede crear un bloqueo mutuo:
    - Supongamos que los cinco filósofos tienen hambre simultáneamente y cada uno toma su palillo izquierdo
    - En este momento todos los elementos palillo tendrán el valor 0
    - Cuando un filósofo intente tomar un palillo sufrirá un bloqueo infinito

Problemas clásicos de sincronización

- **El Problema de los Filósofos Comensales:** Soluciones
  - Permitir que cuanto más cuatro filósofos se sienten simultáneamente en la mesa
  - Permitir que sólo un filósofo tome sus palillos si ambos están disponibles. Los deberá tomar dentro de una sección crítica
  - Usar una solución asimétrica. Un filósofo impar toma primero su palillo izquierdo y luego toma el derecho. Un filósofo par toma primero su palillo derecho y luego toma el izquierdo

Problemas clásicos de sincronización

- **El Problema de los Filósofos Comensales:** Nota
  - Una solución libre de bloqueos mutuos no elimina necesariamente la posibilidad de inanición
  - Cualquier solución satisfactoria al problema de los filósofos comensales debería evitar la posibilidad de que uno de los filósofos muera de hambre

Regiones críticas

- **Introducción:**
  - Los semáforos son mecanismos de sincronización de bajo nivel
  - Los semáforos requieren gran destreza en la programación
  - El uso incorrecto de los semáforos puede generar errores difíciles de detectar ya que sólo ocurren en algunas secuencias de ejecución específicas

Regiones críticas

- **Ejemplo 1:**

P_0

```
repeat
    signal(S);
    Sección crítica
    wait(S);
    Trabajo propio
until false;
```

P_1

```
repeat
    wait(S);
    Sección crítica
    signal(S);
    Trabajo propio
until false;
```

Regiones críticas

- **Ejemplo 2:**

P_0

```
repeat
    wait(S);
    Sección crítica
    wait(S);
    Trabajo propio
until false;
```

P_1

```
repeat
    wait(S);
    Sección crítica
    signal(S);
    Trabajo propio
until false;
```


Regiones críticas

- **Introducción:**
  - **Def.-** Una ***región crítica*** es una construcción de alto nivel para la sincronización de procesos
  - Requiere la declaración de una variable compartida de tipo T como:

    var v: shared T
  - Sólo se puede acceder a la variable **v** mediante la instrucción "region"

    region v when B do S
  - **Nota**: La expresión B es una expresión "booleana" que controla el acceso a la región crítica

Regiones críticas

- **Semántica:**
  - Mientras se ejecuta S, ningún otro proceso puede acceder a la variable **v**
  - Cuando un proceso trata de ejecutar la instrucción "region", se evalúa primero la expresión booleana B:
    - Si B tiene valor true, se ejecuta el enunciado de S
    - Si B tiene valor false, el proceso se suspende hasta que B tome el valor true y no haya ningún otro proceso ejecutando la instrucción S

Regiones críticas

- **El Problema del Buffer Limitado:** Implementación

Datos compartidos

```
var     buffer: shared record;
            pool: array[0, ...,n] of item;
            count, in, out: integer;
        end;
```

Regiones críticas

- **El Problema del Buffer Limitado:** Implementación

Productor P_i

```
region buffer when count  <  n
do begin
        pool[ìn] : = nextp;
        in := in + 1 mod n;
        count := count + 1;
end;
```


Regiones críticas

- **El Problema del Buffer Limitado:** Implementación

Consumidor C_i

```
region buffer when count  >  0
do begin
        nextc := pool[out];
        out := out + 1 mod n;
        count := count - 1;
end;
```


Regiones críticas

- **Implementación de:**

  region x when B do S

  - **Variables asociadas con x:**

```
var     mutex, delay1, delay2: semáforo;
        count1, count2: integer;

/* Inicialización de variables */
        mutex := 1;
        delay1 := 0;
        delay2 := 0;
        count1 := 0;
        count2 := 0;
```


Regiones críticas

- **Notas:**
  - El acceso en exclusión mutua a la sección crítica se logra con el semáforo mutex
  - Si un proceso no puede entrar en la sección crítica porque B es false, espera en el semáforo delay1. El proceso antes de evaluar otra vez B, espera en el semáforo delay2
  - El número de procesos esperando en los semáforos delay1 y delay2 se memorizan en count1 y count2 respectivamente
  - Se asume una estrategia FIFO en las colas asociadas a los semáforos

Regiones críticas

```
wait(mutex);

while not B
      do begin
         count1 := count1 + 1;
         if count2  >  0  then signal(delay2);
         else signal(mutex);

         wait(delay1);
         count1 := count1 – 1;
         count2 := count2 + 1;
         if count1  >  0 then signal(delay1);
         else signal(delay2);

         wait(delay2);
         count2 := count2 – 1;
      end;
```


Regiones críticas

```
S;

if count1  >  0 then signal(delay1);
else  if count2  >  0 then signal(delay2);
      else signal(mutex);
```


Regiones críticas

- **Notas:**
  - La implementación que hemos visto, requiere volver a evaluar la expresión B para todos los procesos que esperan cada vez que un proceso sale de la sección crítica
  - El gasto para volver a evaluar la expresión B podría hacer el código ineficiente

Monitores

- **Def.-** Un ***monitor*** es una construcción de alto nivel para la sincronización que permite compartir de forma segura un tipo abstracto de dato entre procesos concurrentes
- La representación de un tipo monitor consiste en:
  - Declaraciones de variables: sus valores definen el estado del monitor
  - Declaraciones de los cuerpos de los procedimientos o funciones que implementan las operaciones del monitor

Monitores

- **Implementación**

```
type  monitor_name = monitor

   . . . Declaración de variables

   procedure entry P1( . . . );
   begin
      . . . .
   end;

   . . . . . .

   procedure entry Pk( . . . );
   begin
      . . . .
   end;

   begin
         . . .  Código de inicialización
   end.
```


Monitores

- **Características**
  - La representación de un tipo monitor no puede ser usada directamente por varios procesos
  - Un procedimiento definido dentro de un monitor, sólo puede acceder a las variables declaradas localmente en el monitor y a los parámetros formales
  - Las variables locales de un monitor sólo pueden ser accedidas por los procedimientos locales
  - La construcción de un monitor garantiza que sólo un proceso puede estar activo en el monitor
    - El programador no necesita programar esta restricción de sincronización

Monitores

- **Características**
  - Para permitir que un proceso pueda esperar dentro de un monitor, se añaden variables condición:

```
/*      Declaración          */
var     x, y: condition;
```

  - Las variables condición sólo se pueden usar con las operaciones wait y signal

Monitores

- **Operaciones con las variables condición:**
  - Operación x.wait
    - El proceso que invoca esta operación se suspende hasta que otro proceso invoca la operación x.signal
  - Operación x.signal
    - Despierta un proceso suspendido. Si no hay ningún proceso suspendido, la operación signal no tiene ningún efecto

Monitores

- **El Problema de las variables condición:**
  - P invoca una operación x.signal
  - Q está suspendido en la cola de la condición x

    Si Q se despierta
    Entonces
    Si P no es suspendido, P y Q están activos en el monitor a la vez
  - Notar que conceptualmente los dos procesos pueden continuar la ejecución

Monitores

- **El Problema de las variables condición:**
  - Posibilidades:
    - 1.- P puede hacer una de estas dos cosas
      - Espera hasta que Q deje el monitor
      - Espera en otra variable condición
    - 2.- Q puede hacer una de estas dos cosas
      - Espera hasta que P deje el monitor
      - Espera en otra variable condición
    - Notar que es razonable que Q deje el monitor ya que P esta en él. Pero puede ocurrir que la condición para que Q despierte ya no se verifique cuando P termine

Monitores

- **Ejemplo:** El Problema de los Filósofos Comensales
  - Necesitaremos distinguir entre los tres estados en los que un filósofo puede estar: “pensando”, “comiendo” y “hambriento”
  - Introducimos la siguiente estructura de datos:

    var estado: array[0, …, 4] of {pensando, hambriento, comiendo};
  - El filósofo i puede asignar el valor comiendo a la variable estado[i] si:

    estado[i+4 mod 5] ≠ comiendo and estado[i+1 mod 5] ≠ comiendo
  - Declaramos la variable condición C.

    var C: array[0, …, 4] of condition;
  - El filósofo i se puede retardar si no puede obtener los palillos cuando tiene hambre

Monitores

- **Ejemplo:** El Problema de los Filósofos Comensales
  - **Implementación**.

```
type filosofos = monitor;
    var estado:array[0, … ,4] of {pensar, hambre, comer};
    var C: array[0, …, 4] of condition;

procedure entry coger (i : 0, …, 4);
    begin
        estado[i] := hambre;
        test(i);
        if estado[i] ≠ comer then C[i].wait;
    end;
```

Monitores

- **Ejemplo:** El Problema de los Filósofos Comensales
  - **Implementación**.

```
procedure entry dejar (j : 0, …, 4);
    begin
        estado[j] := pensar;
        test(j+4 mod 5);
        test(j+1 mod 5);
    end;
```


Monitores

- **Ejemplo:** El Problema de los Filósofos Comensales
  - **Implementación**.

```
procedure entry test (k : 0, …, 4);
    begin
        if ((estado[k+4 mod 5] ≠ comer)
                and (estado[k] = hambre)
                and (estado[k+1 mod 5] ≠ comer)) then
            begin
                estado[k]  := comer;
                C[k].signal;
            end;
    end;
```


Monitores

- **Ejemplo:** El Problema de los Filósofos Comensales
  - **Implementación**.

```
begin
    for i = 0 to 4 do
        estado[i] := pensar;
end;
```


Monitores

- **Ejemplo:** El Problema de los Filósofos Comensales
  - La distribución de los palillos se controla a través del monitor “filosofos”
  - Es necesario crear una instancia del monitor “filosofos”
  - Cada filósofo antes de comenzar a comer, invoca la operación coger. Si la operación tiene éxito, el filósofo puede comer. Si no es así el filósofo queda suspendido
  - Después de comer, el filósofo invoca la operación dejar y puede comenzar a pensar

Monitores

- **Ejemplo:** El Problema de los Filósofos Comensales
  - El filósofo i debe invocar las operaciones coger y dejar en la secuencia siguiente:

```
... ... ...
filosofos.coger(i);
    Código operación comer
filosofos.dejar(i);
```

  - Notar que esta solución asegura que dos filósofos adyacentes no comen simultáneamente. Además está libre de bloqueos

Monitores

- **Implementación de un monitor con semáforos:**

  Semáforos necesarios

  - **mutex**: semáforo binario
    - Controla el acceso al monitor en exclusión mutua
  - **sig**: semáforo binario
    - Un proceso que ha realizado una operación signal espera en este semáforo hasta que el proceso que se ha despertado deja el monitor o se suspende en otra condición
  - **sig_cont**: variable entera
    - Cuenta el número de procesos suspendidos en el semáforo sig

Monitores

- **Implementación de un monitor con semáforos:**

  Variables compartidas

```
var     mutex: semaforo_binario;
        sig: semaforo_binario;
        sig_cont: integer;

/*      Inicializaciones */

        mutex = 1;
        sig = 0;
        sig_cont = 0;
```


Monitores

- **Implementación de un monitor con semáforos:**

  Cuerpo asociado a un procedimiento externo F

```
wait(mutex);

... ... ... ...

Cuerpo procedimiento F

... ... ... ...

if sig_cont > 0 then signal(sig);
else signal(mutex);
```

Monitores

- **Implementación de un monitor con semáforos:**

  Implementación de las variables condición

  Variables asociadas a la condición x

```
var     x_sem: semaforo_binario;
        x_cont: integer;

/*      Inicializaciones */

        x_sem = 0;
        x_cont = 0;
```


Monitores

- **Implementación de un monitor con semáforos:**

  Operación x.wait

```
x_cont := x_cont + 1;

if sig_cont  >  0 then signal(sig);
else signal(mutex);

wait(x_sem);
x_cont := x_cont – 1;
```


Monitores

- **Implementación de un monitor con semáforos:**

  Operación x.signal

```
if x_cont  >  0 then
    begin
        sig_cont := sig_cont + 1;
        signal(x_sem);
        wait(sig);
        sig_cont := sig_cont - 1;
    end;
```


Monitores

- **Cuerpo del procedimiento F_1**

```
procedure entry coger (i : 0, …, 4);
    begin
        estado[i] := hambre;
        test(i);
        if estado[i] ≠ comer then C[i].wait;
    end;
```

Operación x.wait

```
x_cont := x_cont + 1;

    if sig_cont  >  0 then signal(sig);
    else signal(mutex);

    wait(x_sem);
    x_cont := x_cont – 1;
```

Monitores

- **Cuerpo del procedimiento F_2**

```
procedure entry test (k : 0, …, 4);
    begin
        if ((estado[k+4 mod 5] ≠ comer)
                and (estado[k] = hambre)
                and (estado[k+1 mod 5] ≠ comer)) then
            begin
                estado[k]  := comer;
                C[k].signal;
            end;
    end;
```

Operación x.signal

```
if x_cont  >  0 then
    begin
      sig_cont := sig_cont + 1;
      signal(x_sem);
      wait(sig);
      sig_cont := sig_cont - 1;
    end;
```

Monitores

- **Implementación de un monitor con semáforos:**

¿Cómo ordenar la reanudación de los procesos dentro de un monitor?

- Suponer que varios procesos están suspendidos en la condición x y algún proceso ejecuta una operación x.signal
- ¿Cómo determinar cual de los procesos suspendidos se debe reanudar ahora?
  - **Solución 1**: FCFS
  - **Solución 2**: Utilizando la construcción wait condicional: x.wait(c)

Monitores

- **Implementación de un monitor con semáforos:**

Solución 2:

- Utilización de la construcción wait condicional: x.wait(c)
  - c es una expresión entera que se evalúa cuando se ejecuta la operación wait
  - El valor de c es un “número de prioridad” que se almacena con el nombre del proceso en el momento de realizar la suspensión
  - Cuando se ejecuta x.signal, el proceso con menor “número de prioridad” asociado será el que despierte
- Notar que “número de prioridad” ≅ turno de FCFS

Monitores

- **Ejemplo:** Monitor para asignar solo un recurso mediante la utilización de un wait condicional
  - **Implementación**.

```
type asignar_recurso = monitor;
    var ocupado: boolean;
    x: condition

procedure entry adquirir(tiempo: integer);
    begin
        if ocupado then x.wait(tiempo);
        ocupado := true;
    end;
```

Monitores

- **Ejemplo:** Monitor para asignar solo un recurso mediante la utilización de un wait condicional
  - **Implementación**.

```
procedure entry liberar(tiempo: integer);
    begin
        ocupado := false;
        x.signal;
    end;
```


Monitores

- **Ejemplo:** Monitor para asignar solo un recurso mediante la utilización de un wait condicional
  - Un proceso que necesite acceder al recurso, deberá ejecutar la siguiente secuencia de instrucciones:

```
R.adquirir(t);
    … … …
    acceder al recurso R;
    … … …
R.liberar;
```


Monitores

- **Ejemplo:** Monitor para asignar solo un recurso mediante la utilización de un wait condicional
  - **Problema:**
    - El monitor no puede garantizar la secuencia de instrucciones anterior, es responsabilidad del programador:
      - Un proceso podría acceder al recurso sin obtener permiso de acceso al recurso
      - Un proceso podría no liberar el recurso una vez que se le ha asignado
      - Un proceso podría liberar un recurso que nunca solicitó
      - Un proceso podría solicitar el mismo recurso varias veces sin liberarlo