


Tema 2. Sincronización en sistemas de memoria compartida


Rafael Jesús Segura Sánchez
Grado en Ingeniería en Informática, 2º Curso




Objetivos Generales

- ▶ Describir mecanismos de sincronización para sistemas basados en memoria compartida.
 - ▶ Implementar métodos de exclusión mutua basados en espera ocupada.
 - ▶ Seleccionar el método de sincronización apropiado atendiendo al tipo de problema.
- 

Contenidos

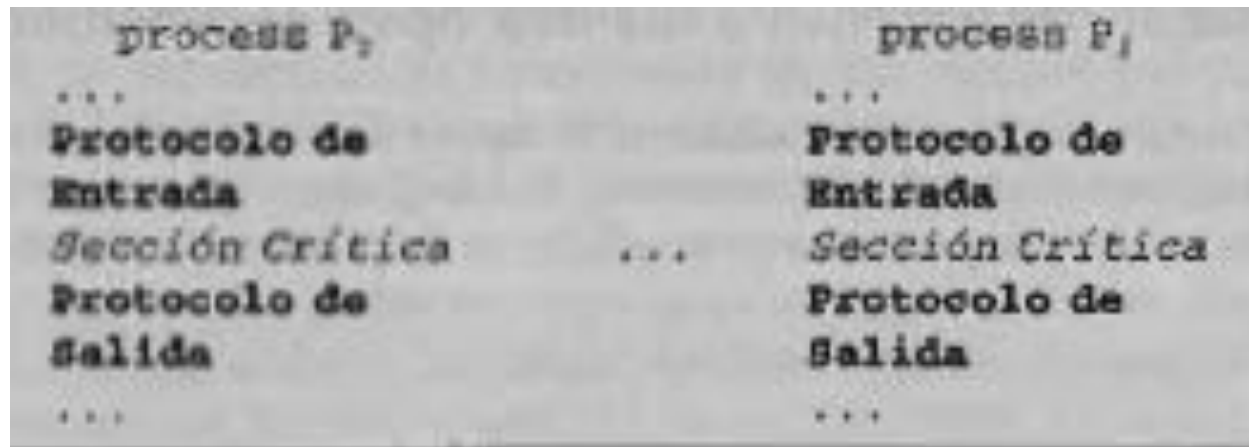
1. Mecanismos de sincronización y comunicación a bajo nivel
 2. Algoritmos básicos de exclusión mutua en sistemas con memoria compartida.
 3. Semáforos
 4. Regiones Críticas condicionales
 5. Monitores
- 

Exclusión Mutua

- ▶ Utilización por varios procesos de un recurso no compartible.
 - ▶ Un proceso que está accediendo a un recurso no compartible se dice que se encuentra en su sección crítica
 - Cuando un proceso está en su sección crítica, el resto de procesos (o al menos aquellos que tengan acceso al mismo recurso no compartible) no deben encontrarse en sus secciones críticas, y si quieren acceder a ellas deben de quedarse a la espera de que la sección crítica quede libre.
 - Cuando un proceso termina de ejecutar su sección crítica, entonces se debe permitir a otro proceso que se encontraba esperando, la posibilidad de entrar a su sección crítica.
- 

Exclusión mutua

- ▶ Condiciones de los protocolos de E/S:
 - Exclusión mutua.
 - Limitación en la espera de cada proceso para acceder a la sección crítica.
 - Progreso en la ejecución:



Condiciones de sincronización

- ▶ la propiedad requerida de que un proceso no realice un evento hasta que otro proceso haya emprendido una acción determinada.

Mecanismos de sincronización

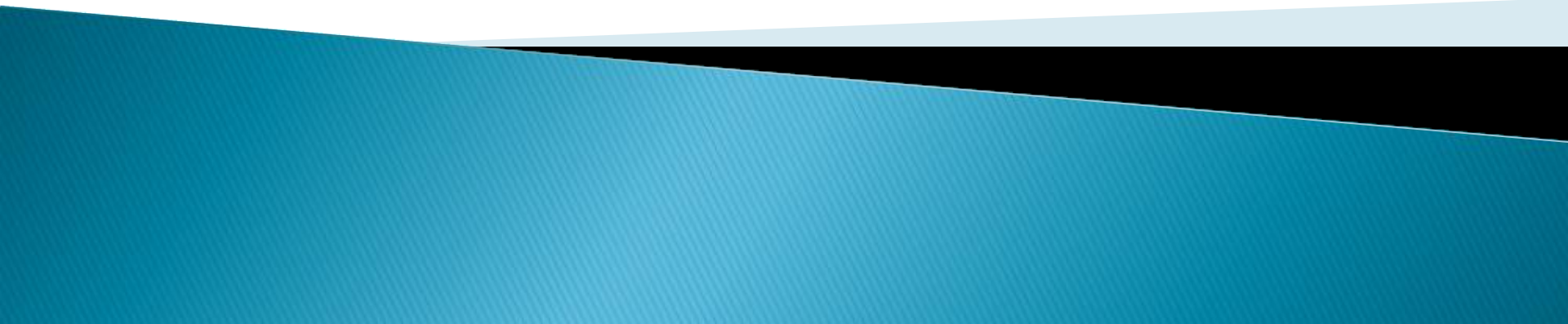
- ▶ Los mecanismos para implementar los tipos de sincronización son los siguientes:
 - Soluciones hardware:
 - Inhibición de las interrupciones
 - Instrucciones específicas
 - Uso variables compartidas:
 - Espera ocupada (busy waiting).
 - Semáforos.
 - Regiones críticas.
 - Regiones críticas condicionales.
 - Monitores.
 - Mecanismos basados en paso de mensajes:
 - Operaciones de paso de mensajes send/receive.
 - Llamadas a procedimientos remotos. (RPC)
 - Invocaciones remotas (RMI)

Inhibición de interrupciones

- ▶ Cada proceso deshabilita todas las interrupciones antes de entrar en la sección crítica y las habilita de nuevo una vez que salga de ella.
- ▶ Problemas:
 - no es correcto que los procesos de usuario tengan el poder de deshabilitar las interrupciones.
 - No válido si la computadora tiene dos o más CPUs.
- ▶ Solución:
 - Garantizar la indivisibilidad de la secciones críticas no compartiendo la CPU
 - El proceso que ha pedido una operación de E/S no libera el procesador, y lleva a cabo hasta el final la operación de E/S, y en ese momento continúa su ejecución.
 - es inadmisibles, ya que los rendimientos se degradan rápidamente

| process P ₁ | | process P ₂ |
|---------------------------|-----|---------------------------|
| ... | | ... |
| deshabilitar | | deshabilitar |
| interrupciones; | | interrupciones; |
| Sección Crítica | ... | Sección Crítica |
| habilitar interrupciones; | | habilitar interrupciones; |
| ... | | ... |

Espera Ocupada



Espera ocupada

- ▶ un proceso espera mediante la comprobación continua del valor de una variable manteniendo ocupada la CPU del sistema
- ▶ Tipos de soluciones
 - soluciones software:
 - las únicas operaciones atómicas que se consideran son las instrucciones de bajo nivel para leer y almacenar (load-store) de/en direcciones de memoria.
 - Soluciones hardware:
 - se utilizan instrucciones especializadas que llevan a cabo una serie de acciones como leer y escribir, intercambiar dos posiciones de memorias, ...
- ▶ para indicar una condición, un proceso inicializa el valor de una variable compartida;
- ▶ para esperar una condición, un proceso chequea reiteradamente el valor de dicha variable hasta que ésta toma el valor deseado.
- ▶ Para implementar la exclusión mutua usando espera ocupada las sentencias que señalan y esperan condiciones se combinan para la construcción de protocolos

Espera ocupada

```
process Pi  
  Repeat  
    Protocolo de Entrada  
      Sección Crítica  
    Protocolo de salida  
      Resto;  
  Forever
```

► Soluciones software:

- Algoritmos incorrectos
- Soluciones para dos procesos
 - Algoritmo de Dekker
 - Algoritmo de Peterson
 - Algoritmo incorrecto de Hyman
- Soluciones para *n procesos*
 - Algoritmo de Eisenberg McGuire
 - Algoritmo de Lamport

Espera ocupada

- ▶ Objetivos de los algoritmos:
 - Garantizar exclusión mútua
 - Evitar espera infinita
 - Garantizar progreso en la ejecución

Algoritmos incorrectos o ineficientes

► Primer intento

```
process  $P_0$ 
  repeat
    //protocolo de entrada
    while  $v = \text{scocupada}$  do;
     $v := \text{scocupada}$ ;
    //ejecuta la sección crítica
    Sección Crítica $_0$ ;
    //protocolo de salida
     $v = \text{sclibre}$ ;
    Resto $_0$ 
  forever
```

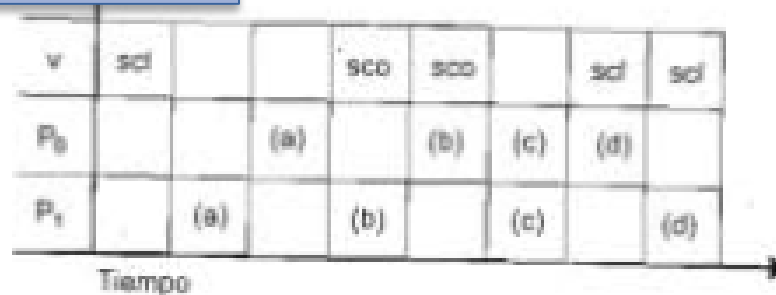
```
process  $P_1$ 
  repeat
    //protocolo de entrada
    while  $v = \text{scocupada}$  do;
     $v := \text{scocupada}$ ;
    //ejecuta la sección crítica
    Sección Crítica $_1$ ;
    //protocolo de salida
     $v = \text{sclibre}$ ;
    Resto $_1$ 
  forever
```

Algoritmos incorrectos o ineficientes

► Primer intento

```
process P0
  repeat
    //protocolo de entrada
    while v=scocupada do;
    v =scocupada;
    //ejecuta la sección crítica
    Sección Crítica0;
    //protocolo de salida
    v =sclibre;
    Resto0
  forever
```

```
process P1
  repeat
    //protocolo de entrada
    while v=scocupada do;
    v =scocupada;
    //ejecuta la sección crítica
    Sección Crítica1;
    //protocolo de salida
    v =sclibre;
    Resto1
  forever
```



Algoritmos incorrectos o ineficientes

▶ Segundo intento

```
process P0
  repeat
    while turno=1 do;
      Sección Crítica0;
    turno=1;
    Resto0
  forever
```

```
process P1
  repeat
    while turno=0 do;
      Sección Crítica1;
    turno=0;
    Resto1
  forever
```

Algoritmos incorrectos o ineficientes

► Tercer intento (falta de exclusión mutua)

```
process P0
  repeat
    while C1=enSC do;
    C0=enSC;
    Sección Crítica0;
    C0 =restoproceso;
    Resto0
  forever
```

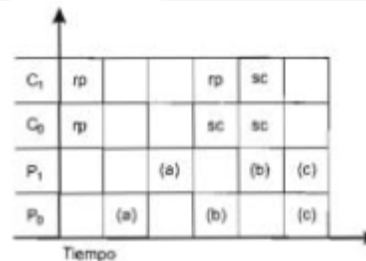
```
process P1
  repeat
    while C0=enSC do;
    C1=enSC;
    Sección Crítica1;
    C1 =restoproceso;
    Resto1
  forever
```


Algoritmos incorrectos o ineficientes

► Tercer intento (falta de exclusión mutua)

```
process P0
  repeat
    while C1=enSC do;
    C0=enSC;
    Sección Crítica0;
    C0:=restoproceso;
    Resto0
  forever
```

```
process P1
  repeat
    while C0=enSC do;
    C1=enSC;
    Sección Crítica1;
    C1:=restoproceso;
    Resto1
  forever
```



Algoritmos incorrectos o ineficientes

► Cuarto intento (espera infinita)

```
process P0
  repeat
    C0=quiereentrar;
    while C1=quiereentrar do;
    Sección Crítica0;
    C0=restoproceso;
    Resto0
  forever
```

```
process P1
  repeat
    C1=quiereentrar;
    while C0=quiereentrar do;
    Sección Crítica1;
    C1=restoproceso;
    Resto1
  forever
```

Algoritmos incorrectos o ineficientes

► Quinto intento

```
process P0
  repeat
    C0=quiereentrar;
    while C1=quiereentrar do
      C0=restoproceso;
      //hacer algo
      ....
      C0=quiereentrar;
    Sección Crítica0:
    C0=restoproceso;
    Resto0
  forever
```

```
process P1
  repeat
    C1=quiereentrar;
    while C0=quiereentrar do
      C1=restoproceso;
      //hacer algo
      ....
      C1=quiereentrar;
    Sección Crítica1:
    C1=restoproceso;
    Resto1
  forever
```

Algoritmo de Dekker

```
process P0
  Repeat
    C0=quiereentrar;
    while C1 =quiereentrar do
      if turno=1 then
        C0=restoproceso;
        while turno=1 do;
          C0=quiereentrar
        Sección Crítica0;
        turno =1;
        C0 =restoproceso;
        Resto0;
    forever
```

```
process P1
  Repeat
    C1=quiereentrar;
    while C0 =quiereentrar do
      if turno=0 then
        C1=restoproceso;
        while turno=0 do;
          C1=quiereentrar
        Sección Crítica1;
        turno =0;
        C1 =restoproceso;
        Resto1;
    forever
```

Algoritmo de Dekker

- ▶ Asegura la exclusión mutua porque si P1 está en la sección crítica $C1 = \text{quiere entrar}$ con lo que P1 no ha podido pasar del while más externo.
- ▶ Cumple la condición de progreso en la ejecución ya que si P1 quiere entrar y P0 no, entonces $C1 = \text{restop proceso}$, y por lo tanto P1 puede entrar.
- ▶ Satisface la limitación en la espera ya que cuando un proceso sale de la sección crítica indica que no quiere entrar y cede el turno al otro.

Algoritmo de Dekker

- ▶ Supongamos que P0 está en su sección crítica
 - cuando se ejecutó “While C1 ...”, C1 valía restoproceso, y en ese momento P1 tenía que estar en Resto1.
 - Si P1 estaba en Resto1 al llegar a “While C0...” se quedaría en el bucle, y si estaba en el bucle “while turno=”, sólo podría salir de él cuando turno=1, lo que ocurrirá cuando P0 salga de la sección crítica.

```
▶ process P0
  ◦ Repeat
    • C0=quiereentrar;
    • while C1 =quiereentrar do
      • if turno=1 then
        • C0=restoproceso;
        • while turno=1 do;
        • C0=quiereentrar
    • Sección Crítica0;
    • turno =1;
    • C0 =restoproceso;
    • Resto0;
  ◦ forever
```

```
▶ process P1
  ◦ Repeat
    • C1=quiereentrar;
    • while C0 =quiereentrar do
      • if turno=0 then
        • C1=restoproceso;
        • while turno=0 do;
        • C1=quiereentrar
    • Sección Crítica1;
    • turno =0;
    • C1 =restoproceso;
    • Resto1;
  ◦ forever
```

Algoritmo de Dekker

- ▶ **Condición de progreso**
 - Si P0 quiere entrar a su sección crítica y está bloqueado, sólo puede estar en While C1 o While turno
- ▶ **limitación en la espera:**
 - Si los dos procesos quieren entrar a la sección crítica, al llegar a la instrucción 3, el que no tiene el turno pasa a la instrucción 4 y pone su indicador a restoproceso, con lo que el que tiene el turno deja de esperar en el segundo bucle.

```
process P0
  Repeat
    C0=quiereentrar;
    while C1 =quiereentrar do
      if turno=1 then
        C0=restoproceso;
        while turno=1 do;
          C0=quiereentrar
    Sección Crítica0;
    turno =1;
    C0 =restoproceso;
    Resto0;
  forever
```

```
process P1
  Repeat
    C1=quiereentrar;
    while C0 =quiereentrar do
      if turno=0 then
        C1=restoproceso;
        while turno=0 do;
          C1=quiereentrar
    Sección Crítica1;
    turno =0;
    C1 =restoproceso;
    Resto1;
  forever
```

Algoritmo de Dekker

- ▶ orientado a entornos centralizados ya que hay una variable que es accedida y modificada por dos procesos (turno).
- ▶ No adecuado para sistemas distribuidos

Algoritmo de Hyman (incorrecto)

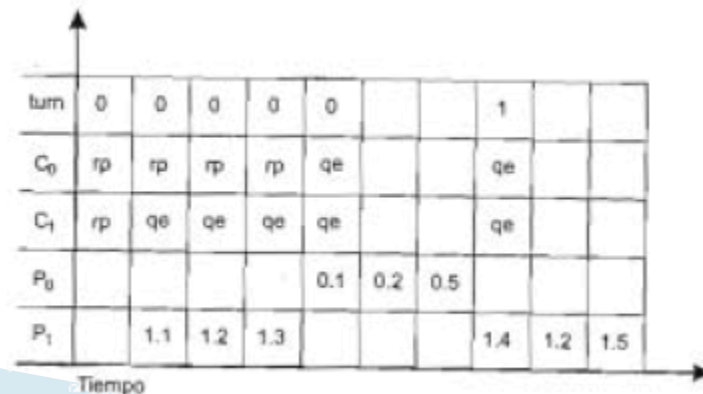
```

process P0
Repeat
    C0=quiereentrar;
    while turno ≠ 0 do
        while C1=quiereentrar do;
            turno=0;
    end;
    Sección Critica0;
    C0=restoproceso;
    Resto0;
forever
    
```

```

process P1
Repeat
    C1=quiereentrar;
    while turno ≠ 1 do
        while C0=quiereentrar do;
            turno=1;
    end;
    Sección Critica1;
    C1=restoproceso;
    Resto1;
forever
    
```

- ▶ C0=restoproceso;
- ▶ C1=restoproceso
- ▶ turno =0 ó 1



Algoritmo de Peterson (1981)

```
process P0
repeat
    C0=quiereentrar;
    turno=1;
    while (C1==quiereentrar) and
        (turno=1) do;
    Sección Crítica0;
    C0=restoproceso;
    Resto0
forever
```

```
process P1
repeat
    C1=quiereentrar;
    turno=0;
    while (C0==quiereentrar) and
        (turno=0) do;
    SecciónCrítica1;
    C1=restoproceso ;
    Resto0
forever
```

- ▶ C0=restoproceso;
- ▶ C1=restoproceso
- ▶ turno =0 ó 1

Algoritmos para n procesos

- ▶ Generalizaciones del algoritmo de Dekker
 - algoritmo de Dijkstra [Dijkstra, 1965].
 - algoritmo de Knuth [Knuth, 1966],
 - algoritmo de Bruijn [de Bruijn, 1967],
 - algoritmo de Eisenberg– McGuire [Eisenberg and McGuire, 1972].

Algoritmo de Dijkstra (1965)

Variables Compartidas

VAR flag : array[0..n-1] of (restoproceso, quiereentrar, enSC); // inicializar a restoproceso

turno : 0..n-1;

Process P_i

var j:0..n-1;

repeat

repeat

 flag[i] = quiereentrar;

while (turno \neq i)

 if flag[turno] = restoproceso then turno = i;

 flag[i] = enSC;

 j = 0;

while (j < n) and (j = i or flag[j] \neq enSC) j=j+1;

until j = n;

 // Ahora se cumple: (flag[j] \neq enSC, $\forall j \neq i$) y flag[i]=enSC

seccion critica_i;

 flag[i] = restoproceso;

forever

Algoritmo de Eisenberg-McGuire

```
process Pi
  repeat
    repeat
      indicador[i]: =quiereentrar;
      j = indice; // (1)
      while (j ≠ i)
        if (indicador [j] ≠ restoproceso)
          then j=indice // (2)
        else j=(j+1)mod n;
        indicador[j] =enSC; // (3)
      j=0;
      While (j<n) and (j=i) or (indicador[j] ≠ enSC)) j=j+1;
    until ( (j≥n) and ((indice=i) or (indicador[indice]=restoproceso))); // (4)
    indice=i;
    Sección Críticai;
    j=(indice+1)mod n;
    while (indicador[j]=restoproceso) j =(j+1)mod n;
    indice=j; // (5)
    indicador[i]: =restoproceso;
    Resto).
  forever
```

Inicialización


- ▶ Indicador[i]=restoproceso
- ▶ Indice= 0..n-1

Algoritmo de Lamport

```
process Pi (i:entero)
  repeat
    C[i] =cognum;
    numero[i]=1+max(numero[0], ... ,numero[n-1]);
    C[i]=nocognum;
    for j=0 to n-1 do
      while (C[j]=cognum) do;
      while ((numero[j]!=0) and ((numero[i],i)>(numero[j],j)) do;
    sección Críticai;
    numero[i]=0;
    Resto!i;
  forever
```

```
Process main()
Var i: entero
Begin
  for i=0 to n-1 do begin
    numero[i]=0; C[i]=nocognum;
  end;
  cobegin
    for i=0 to n -1 do Pi(i)
  coend;
End.
```

Mecanismos de sincronización y comunicación a bajo nivel

- ▶ Algunos procesadores ofrecen unas instrucciones que llevan a cabo varias acciones de una forma indivisible verificar y modificar el contenido de una palabra, intercambiar el contenido de dos palabras, etc.
 - ▶ Hay que decir que en cualquier procesador el acceso a una posición de memoria excluye cualquier otro acceso a la misma posición.
 - ▶ La característica principal de estas instrucciones es que se ejecutan indivisiblemente. Estas instrucciones especiales pueden utilizarse para resolver el problema de la exclusión mutua usando espera ocupada.
- 

Instrucción EXCHANGE (CAS)

- ▶ ***exchange(r, m):***
 - intercambia los contenidos de las direcciones de memoria r y m de forma atómica.
- ▶ **Cumple:**
 - exclusión mutua
 - progreso en la ejecución
- ▶ **No cumple:**
 - limitación en la espera,

```
process Pi
  ri=0;
  repeat
    repeat exchange (ri,m) until ri=1;
    Sección Críticai;
    exchange(ri,m);
    Restoi;
  forever
```

Inicialización
 $m=1$;

Instrucción incremento/decremento

- ▶ **subc(r,m)**: decrementa en 1 el contenido de m y copia el resultado en r de forma atómica.
- ▶ **addc(r,m)**: incrementa en 1 el contenido de m y copia el resultado en r de forma atómica.

```

process Pi
  repeat
    repeat subc (ri,m) until ri=0;
    Sección Críticai;
    m=1;
    Restoi
  forever

```

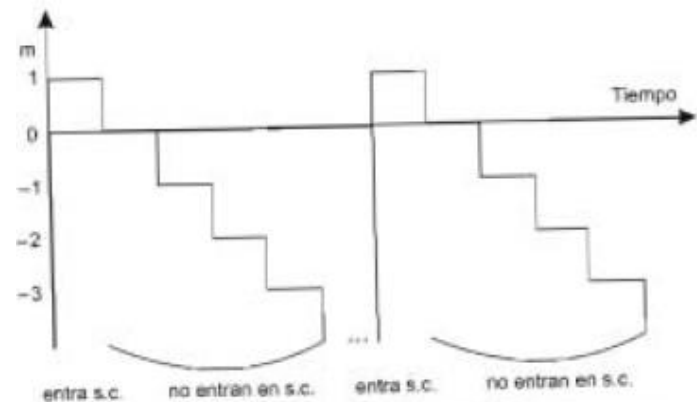
Inicialización

```

process Pi
  repeat
    repeat addc (ri,m) until ri=0;
    Sección Críticai;
    m=-1;
    Restoi
  forever

```

Inicialización



Instrucción testset

- ▶ Testset(m)
 - Comprueba el valor de la variable m.
 - Si el valor es 0, lo cambia por 1 y devuelve el resultado de verdad.
 - En otro caso no cambia el valor de m y devuelve falso.

```
process Pi
  repeat
    repeat until testset(m);
    Sección Críticai;
    m=0;
    Restoi
  forever
```

Inicialización

m=0;

Instrucción testset

```
process Pi
  repeat
    esperando[i]=true;
    llave=false;
    repeat
      llave=testset(m);
    until (not esperando[i]) or (llave);
    esperando[i] =false;
    Sección Críticai ;
    J =(i+1)mod n;
    while (j ≠i) and (not esperando[j]) do
      j=(j+1) mod n
    if (j=i)
      then m=0;
      else esperando[j]=falso
    restoi;
  forever
```

Semáforos

Semáforos

- ▶ Es un TDA:
 - Valores numéricos ≥ 0
 - De más bajo nivel que otras soluciones
- ▶ Tipos:
 - Semáforos genéricos (cualquier valor ≥ 0)
 - Semáforos binarios (0 ó 1)

Semáforos

► Operaciones:

- wait (s).
- signal (s).
- initial (s,v) .

```
Wait (s)
  if  $s > 0$ 
  then  $s = s - 1$ 
  else bloquear proceso;
```

```
Signal (s)
  if (hay procesos bloqueados)
  then desbloquear un proceso
  else  $s = s + 1$ ;
```

```
Initial (s,v) //  $v > 0$ 
   $s = v$ ;
  Inicializar lista procesos bloqueados
```

Semáforos

- ▶ Exclusión mutua

```
process  $P_i$   
begin  
    wait(s);  
    Sección crítica $_i$ ;  
    signal (s);  
end;
```

Semáforos

► Condición de sincronización

```
process P0
  Begin
    A;
    B;
  End;
```

```
process P1
  Begin
    C;
    D;
  End;
```

- supongamos que D no puede ejecutarse hasta que se ejecute A

```
process P0
  Begin
    A;
    signal(s)
    B;
  End;
```

```
process P1
  Begin
    C;
    wait(s)
    D;
  End;
```


Semáforos

- supongamos que D no puede ejecutarse hasta que se ejecute A

```
process P0
  Begin
    A;
    signal(s)
    B;
  End;
```

```
process P1
  Begin
    C;
    wait(s)
    D;
  End;
```

- Supongamos que además B tiene que ejecutarse después de C;

```
process P0
  Begin
    A;
    signal(s);
    wait (t)
    B;
  End;
```

```
process P1
  Begin
    C;
    signal(t)
    wait(s);
    D;
  End;
```

Semáforos

- OJO: INTERBLOQUEO

```
process P0
  Begin
    A;
    wait (t);
    signal(s);
    B;
  End;
```

```
process P1
  Begin
    C;
    wait(s);
    signal(t)
    D;
  End;
```

Semáforos

► Tres procesos

```
process P0
  Begin
    A;
    B;
  End;
```

```
process P1
  Begin
    C;
    D;
  End;
```

```
process P2
  Begin
    E;
    F;
  End;
```

- queremos que P1 sólo pueda pasar a ejecutar D si P2 ha ejecutado E, ó P0 ha ejecutado A.

```
process P0
  Begin
    A;
    signal(s);
    B;
  End;
```

```
process P1
  Begin
    C;
    wait(s);
    D;
  End;
```

```
process P2
  Begin
    E;
    signal(s);
    F;
  End;
```

Semáforos

► Tres procesos

```
process P0
  Begin
    A;
    B;
  End;
```

```
process P1
  Begin
    C;
    D;
  End;
```

```
process P2
  Begin
    E;
    F;
  End;
```

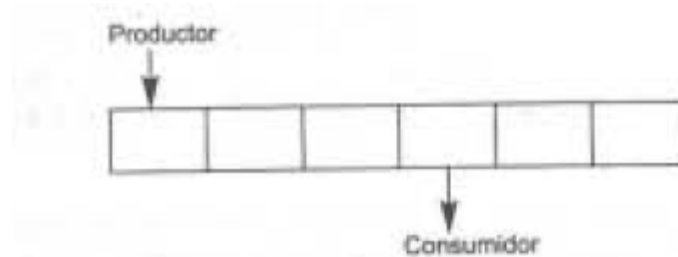
- queremos que P1 sólo pueda pasar a ejecutar D si P2 ha ejecutado E, Y P0 ha ejecutado A.

```
process P0
  Begin
    A;
    signal(s);
    B;
  End;
```

```
process P1
  Begin
    C;
    wait(s);
    wait(t);
    D;
  End;
```

```
process P2
  Begin
    E;
    signal(t);
    F;
  End;
```

productor/consumidor



```
process productor;  
begin  
  repeat  
    producir elemento;  
    protocolo de entrada;  
    insertar elemento en el buffer;  
    protocolo de salida;  
  forever  
end;
```

```
process consumidor;  
begin  
  repeat  
    protocolo de entrada;  
    extraer elemento del buffer;  
    protocolo de salida;  
    consumir elemento  
  forever  
end;
```

producer/consumidor

```
process productor;  
begin  
  repeat  
    producir ítem;  
    wait(vacios);  
    wait(mutex) ;  
    buffer[frente]=item;  
    frente=(frente+1) mod N;  
    signal(mutex);  
    signal(llenos);  
  forever  
end;
```

```
process consumidor;  
begin  
  repeat  
    wait(llenos) ;  
    wait(mutex);  
    item =buffer[cola];  
    cola =(cola+1) mod N;  
    signal(mutex);  
    signal(vacios);  
    consumir ítem;  
  forever  
end;
```

lectores y escritores

```
cobegin
  for i:=1 to nLectores do lector [i]
  for j:=1 to nEscritores do escritor [j];
coend;
```

```
process type lector;
begin
  protocolo de entrada;
  Leer del recurso; //consultar el recurso
  protocolo de salida;
end;
```

```
process type escritor ;
begin
  protocolo de entrada;
  escribir en el recurso; //modificar el recurso
  protocolo de salida;
end;
```

lectores y escritores (Prioridad en la lectura)

```
process type lector;  
begin  
    wait(mutex);  
    nl=nl+1;  
    // Se impide que entre un escritor a escribir  
    if (nl=1) then wait(wrt);  
    signal(mutex);  
    ...  
    Leer del recurso ;  
    ...  
    wait(mutex);  
    nl=nl-1;  
    //El último lector intenta desbloquear a algún  
    escritor  
    if (nl=0) then signal(wrt);  
    signal(mutex);  
end;
```

```
nl=0; // N° de lectores leyendo  
mutex=1; // acceso exc. Mutua  
wrt=1; // sincronización lect/escr.
```

```
process type escritor;  
begin  
    // La escritura se hace en exclusión mutua  
    wait(wrt);  
    Escribir en el recurso;  
    signal(wrt);  
end;
```


lectores y escritores (Prioridad en la escritura)

```
nl=0; // N° de lectores leyendo  
nee=0; /N° escritores esperando  
escribiendo=false; //Indica si hay escritores escribiendo  
mutex=1; //acceso exc. Mutua
```

```
process type lector;  
begin  
    wait(mutex);  
    // Mientras existan escritores en espera o algún  
    // escritor esté escribiendo esperar  
    while (escribiendo or nee>0) do  
    begin  
        signal(mutex);  
        wait(mutex);  
    end;  
    nl=nl+1;  
    signal(mutex);  
    ...  
    Leer del recurso;  
    ...  
    wait(mutex);  
    nl=nl-1;  
    signal(mutex);  
End;
```

```
process type escritor;  
begin  
    wait(mutex);  
    // Mientras algún escritor esté accediendo al  
    // recurso o existan lectores leyendo, esperar  
    nee=nee+1;  
    while (escribiendo or nl>0) do  
    begin  
        signal(mutex);  
        wait(mutex);  
    end;  
    escribiendo=true;  
    nee=nee-1;  
    signal(mutex);  
    Escribir en el recurso;  
    wait(mutex);  
    escribiendo=false;  
    signal(mutex);  
end;
```

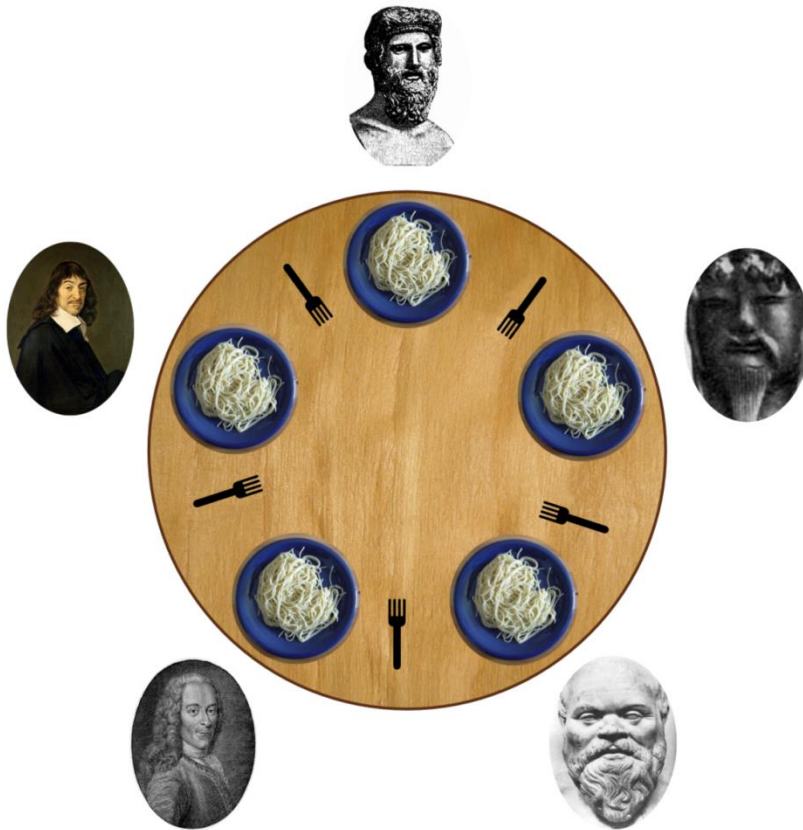
lectores y escritores (Prioridad en la escritura)

```
nl=0; // N° de lectores leyendo
nle:=0; //N° lectores esperando
nee=0; //N° escritores esperando
escribiendo=false; //Indica si hay escritores escribiendo
mutex=1; //acceso exc. Mutua
Escritor=0; //semáforo para bloquear escritores
Lector=0; // semáforo para bloquear lectores
```

```
process type lector;
begin
    wait (mutex) ;
    // Si se está escribiendo o existen escritores en
    // espera el lector debe ser bloqueado
    if (escribiendo or nee>0) then
    begin
        nle=nle+1;
        signal (mutex) ;
        wait (lector);
        nle=nle-1;
    end;
    nl=nl+1;
    if (nle>0) then // Desbloqueo encadenado
        signal (lector)
    else signal (mutex) ;
    Leer del recurso;
    wait (mutex) ;
    nl=nl-1;
    // Desbloquear un escritor si es posible
    if (nl=0 and nee>0) then signal (escritor)
    else signal (mutex);
end;
```

```
process type escritor;
begin
    wait (mutex) ;
    // Si se está escribiendo o existen lectores el
    // escritor debe ser bloqueado .
    if (nl>0 or escribiendo) then
    begin
        nee=nee+1;
        signal (mutex) ;
        wait (escritor);
        nee=nee-1;
    end;
    escribiendo=true;
    signal (mutex) ;
    ...
    Escribir en el recurso;
    ...
    wait (mutex) ;
    ne=ne-1;
    //Desbloquear un escritor que esté en espera
    //sino desbloquear a un lector en espera.
    if (nee>0) then signal (escritor)
    else
        if (nle>0) then signal (lector)
        else signal (mutex)
    end;
end;
```

Filósofos



```
var
semaphore palillo[5]={1,1,1,1,1};

process type filosofo (i:integer);
begin
    repeat
        piensa;
        wait (palillo[i]);
        wait (palillo[(i+1)mod 5]);
        come;
        signal (palillo[i]);
        signal (palillo[(i+1)mod 5]);
    forever
end;
```

Filósofos

```
Var
semaphore sitio= 4;
process type filosofo (i:integer);
begin
    repeat
        piensa;
        wait (sitio);
        wait (palillo[i]);
        wait (palillo[(i+1)mod 5]);
        come;
        signal (palillo[i]);
        wait (palillo[(i+1)mod 5]);
        signal (sitio);
    forever
end;
```

Filósofos

```
Var
Boolean libres[5]={true,true,true,true,true};
semaphore mutex=1;

process type filosofo (i:integer);
begin
    repeat
        piensa;
        wait (mutex) ;
        while not (libres[i] and libres[(i+1) mod 5]) do
            begin
                signal (mutex) ;
                wait (mutex);
            end;
        1 libres[i]=false; libres[(i+1) mod 5]=false;
        signal (mutex);
        come;
        wait (mutex);
        libres[i]=true; libres[(i+1) mod 5]=true;
        signal (mutex)
    forever
end;
```

Filósofos

```
var  
semaphore palillo[5]={1,1,1,1,1};
```

```
process type filosofo_par (i:integer);  
begin  
  repeat  
    piensa;  
    wait (palillo[(i+1)mod 5]);  
    wait (palillo[i]);  
    come;  
    signal (palillo[i]);  
    signal(palillo[(i+1)mod 5]);  
  forever  
end;
```

```
process type filosofo _impar (i:integer);  
begin  
  repeat  
    piensa;  
    wait (palillo[i]);  
    wait (palillo[(i+1)mod 5]);  
    come;  
    signal (palillo[i]);  
    Signal (palillo[(i+1)mod 5]);  
  forever  
end;
```

Implementación de semáforos

► Usando variables enteras

```
type semaphore: struct{  
    valor:integer;  
    l: lista de procesos;  
}
```

```
procedure initial (var s: semaphore; v: integer) ;  
begin  
    s.valor=v;  
    inicializar(s.l) ;  
end;
```

```
procedure wait (var s : semaphore) ;  
begin  
    s.valor=s.valor-1;  
    if s.valor<0 then  
        begin  
            añadir proceso a s.l;  
            Bloquear proceso;  
        end;  
    end;  
end;
```

```
procedure signal (var s: semaphore );  
begin  
    s.valor=s.valor+1;  
    if s.valor<=0 then  
        begin  
            eliminar proceso de s.l;  
            desbloquear proceso;  
        end;  
    end;  
end;
```

Implementación de semáforos


► Usando variables enteras

protocolo de entrada con espera ocupada;
Sección Crítica (código de usuario);
protocolo de salida;



protocolo de entrada con espera ocupada;
Sección Crítica (código wait) ;
protocolo de salida;
Sección Crítica (código usuario);
protocolo de entrada con espera ocupada;
Sección crítica (código signal);
protocolo de salida;

Inconvenientes de los semáforos

- ▶ Se descarga al programador:
 - La redacción correcta (wait signal)
 - La inclusión en la sección crítica de todos los recursos compartidos
 - ▶ No se puede restringir el tipo de operaciones realizadas sobre los recursos.
 - ▶ Tanto la exclusión mutua como la condición de sincronización se implementan usando el mismo par de primitivas.
 - difícil el identificar el propósito de un wait o signal.
 - ▶ el código de sincronización está repartido entre los distintos procesos, con lo que cualquier modificación implica la revisión de todos los procesos.
- 

Merienda

```
process type niño;  
begin  
    wait (mutex) ;  
    if galletas=0 then  
        begin  
            signal (platovacio);  
            wait (platolleno);  
        end;  
        galletas=galletas- 1;  
        signal (mutex) ;  
    end;
```

```
process mama;  
begin  
    repeat  
        wait (platovacio) ;  
        galletas:=N;  
        signal (platolleno) ;  
    forever  
end;
```

```
process merienda;  
Shared Var  
    galletas : integer;  
    mutex,platovacio,platovacio:semaphore;  
  
var  
    niños :array[1.. n] of niño;  
    i:integer;  
  
Begin  
    galletas=N;  
    initial (mutex,1) ;  
    initial (espera,0);  
    initial (mama, 0) ;  
    cobegin  
        for i:=1 to n do niños[i] ;  
        mama;  
    coend;  
end;
```

Monitores

Definición

- ▶ Es un TDA
 - Encapsular la representación de un recurso compartido y sus operaciones de acceso/manejo.
 - Contiene:
 - EEDD para representar el recurso
 - Operaciones de acceso al recurso
 - Se garantiza que la ejecución de los métodos de acceso al monitor se realiza en exclusión mutua.
 - Dos procesos activos no pueden estar dentro del monitor (en el mismo o en distintos procedimientos).

Definición

- ▶ Un programa concurrente contiene dos clases de procesos:
 - procesos activos:
 - utilizan los monitores
 - procesos pasivos:
 - implementan los monitores
 - las variables compartidas están en el interior del monitor
 - Ventajas:
 - los procesos activos no tienen que preocuparse de cómo está implementado el recurso compartido,
 - el programador del monitor no se debe preocupar en dónde y cómo se va a utilizar el monitor,
 - Desarrollar de forma casi independiente los distintos procesos.
 - Mejorar la modularidad.


Elementos

- ▶ Variables permanentes:
 - Definen el estado del recurso compartido
 - Solo son accesibles desde el interior del monitor
- ▶ Código de inicialización:
 - Inicializa las variables permanentes
- ▶ Procedimientos internos
 - Manipular las variables permanentes
- ▶ Procedimientos exportados
 - Métodos accesibles por los procesos activos
- ▶ Variables de condición
 - Gestionar las condiciones de sincronización
 - Funcionan como colas con procesos bloqueados
 - Llevan asociados métodos de bloqueo/desbloqueo y consulta

Invocación externa

- ▶ `nombre_monitor.nombre_procedimiento(argumentos);`

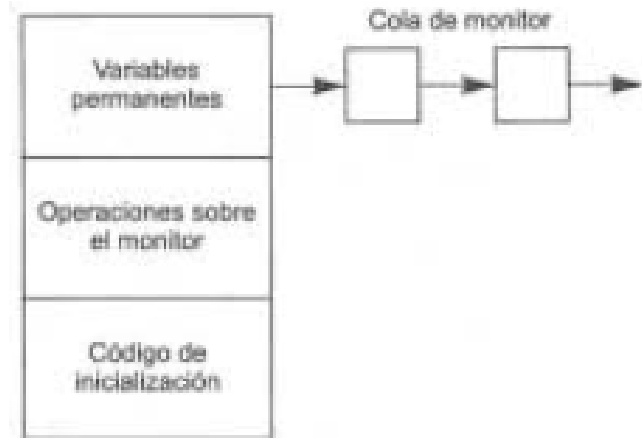
Control de la exclusión mutua

- ▶ Los monitores son una construcción del lenguaje de programación
 - El compilador genera código adicional para garantizar que si un proceso está en un procedimiento del monitor, ningún otro proceso puede acceder.
 - Se suelen implementar mediante semáforos binarios
- 

Control de la exclusión mutua

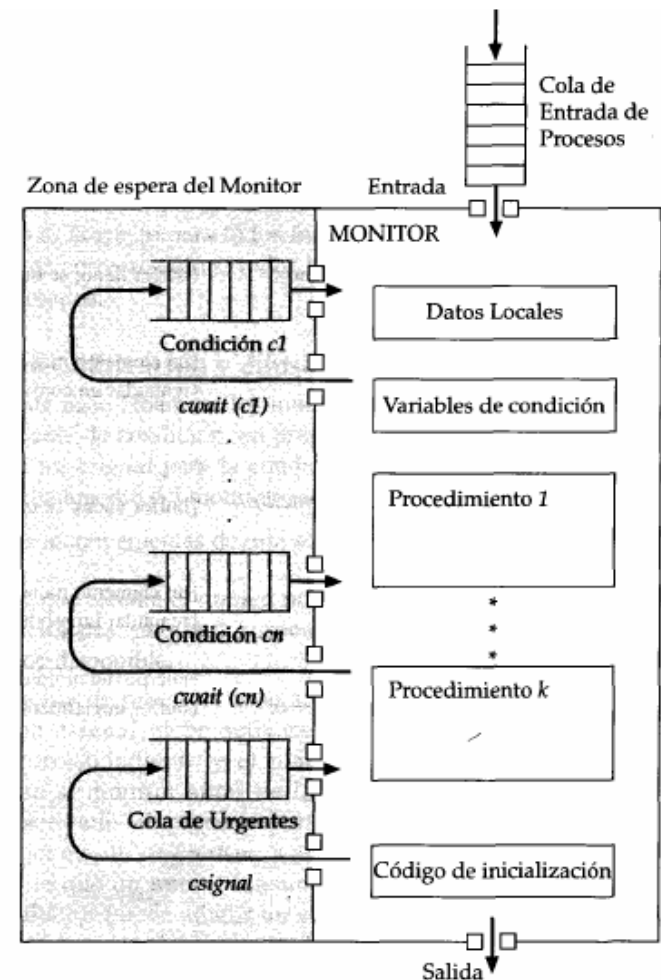
► Cola del monitor

- Cuando un proceso activo está ejecutando uno de los procedimientos del monitor y otro proceso activo intenta ejecutar otro de los procedimientos, el código de acceso al monitor bloquea el proceso que realiza la llamada y lo inserta en la cola del monitor.
- Cuando un proceso activo abandona el monitor, el monitor selecciona el proceso que está al frente de la cola del monitor y lo desbloquea.
 - Si la cola del monitor está vacía, el monitor quedará libre y el primer proceso activo que llame a uno de sus procedimientos entra en el monitor.



Condiciones de sincronización

- ▶ Variables condición
 - Cada variable tiene asociada una cola
 - Los monitores bloquean/desbloquean procesos en dichas variables de condición
 - Operaciones:
 - Empty
 - Delay
 - Resume



Condición de sincronización

- ▶ Operaciones sobre variables de condición

```
empty (c)  
    return (c cola.size()==0);
```

Condición de sincronización

► Operaciones sobre variables de condición

delay (c)

liberar la exclusión mutua del monitor

bloquear el proceso que realiza la llamada al final de la cola;

empty (c)

return (c.cola.size()==0);

Condición de sincronización

► Operaciones sobre variables de condición

delay (c)

liberar la exclusión mutua del monitor

bloquear el proceso que realiza la llamada al final de la cola;

resume (c)

if (hay procesos bloqueados) then

liberar exclusión mutua del monitor

desbloquear un proceso

Ceder monitor a proceso desbloqueado

empty (c)

return (c.colasize()==0);

Semántica de la operación resume

▶ Resume & continue (RC)

- el proceso desbloqueador continúa su ejecución hasta que sale del monitor (o bien se bloquea en una variable condición, o bien termina la ejecución del procedimiento);
- en ese momento el proceso desbloqueado es seleccionado y continúa su ejecución con la instrucción siguiente al delay.
- Problemas:
 - no podemos garantizar que siga siendo verdadera la condición que generó el delay

```
while not B do delay(C);
```

Semántica de la operación resume

- ▶ Retorno forzado (resume & exit)
 - la operación resume implica la finalización del procedimiento
 - el proceso desbloqueador termina la ejecución del procedimiento cediendo el monitor al proceso desbloqueado.
 - el proceso desbloqueado no tiene que volver a comprobar si la condición por la que se bloqueó ha dejado de ser cierta.
 - Concurrent Pascal

Semántica de la operación resume

- ▶ Desbloquear y esperar (resume & wait)
 - el proceso desbloqueador se bloquea en la cola del monitor y libera el monitor para que el proceso desbloqueado pueda ejecutarse.
 - tendrá que volver a competir por volver a adquirir la exclusión mutua del monitor con el resto de procesos.
 - el estado del monitor que hizo posible la ejecución del resume no ha cambiado.
 - coloca al proceso desbloqueador al mismo nivel que el resto de procesos que compite por adquirir la exclusión mutua del monitor.
 - no es muy justa
 - el proceso desbloqueador debería tener cierta preferencia sobre los procesos bloqueados en la cola del monitor.
 - Modula-2

Semántica de la operación resume

- ▶ Desbloquear y espera urgente (DU) (resume & urgent wait)
 - además de la cola del monitor, se asocia la cola de cortesía.
 - el proceso desbloqueador activa el proceso desbloqueado
 - se bloquea en la cola de cortesía teniendo los procesos bloqueados en esta cola preferencia sobre los procesos de la cola del monitor.

Ventajas e inconvenientes

- ▶ Facilidad de uso,
 - la política DR es la más compleja,
 - reescribir los códigos de los procedimientos y particionarlos de forma artificial,
 - el resto de soluciones representa un menor coste a la hora de programar el monitor, aunque siempre tendremos que tener en cuenta rodear la operación delay en un bucle while si trabajamos con monitores que utilizan la semántica (OC).
- ▶ Eficiencia:
 - *delay* y *resume* implican un cambio de contexto en la CPU
 - DE y DU resultan ineficientes si la operación resume se coloca al final de los procedimientos,
 - el proceso desbloqueador se bloquea cuando sólo queda pendiente la instrucción de salida del procedimiento lo que obliga a dos cambios de contexto para ejecutar dicha instrucción.
 - DC también resulta ineficiente (delay dentro de bucles while).

Trabajando con monitores

- ▶ Identificar recurso compartido
- ▶ Definir EEDD para manejar recurso compartido
 - variables privadas
 - Métodos privados de manejo de la EEDD
 - Métodos exportables de acceso al recurso
- ▶ Identificar procesos que van a usar el recurso
 - Definir variables de condición
 - Establecer mecanismos de sincronización

Semáforos mediante monitores

```
monitor semaforo_binario;
var
    sem: boolean;           //Indica si el semáforo está ocupado
    csem: condition;        // para bloquear los procesos en el wait
export wait, signal;       // métodos accesibles desde el exterior

procedure wait;
begin
    if sem then delay(csem);
    sem=true;
end;

procedure signal ;
begin
    if not empty(csem) then resume (csem);
    else sem=false;
end;

begin
    sem=false;
end;
```

Productor/consumidor

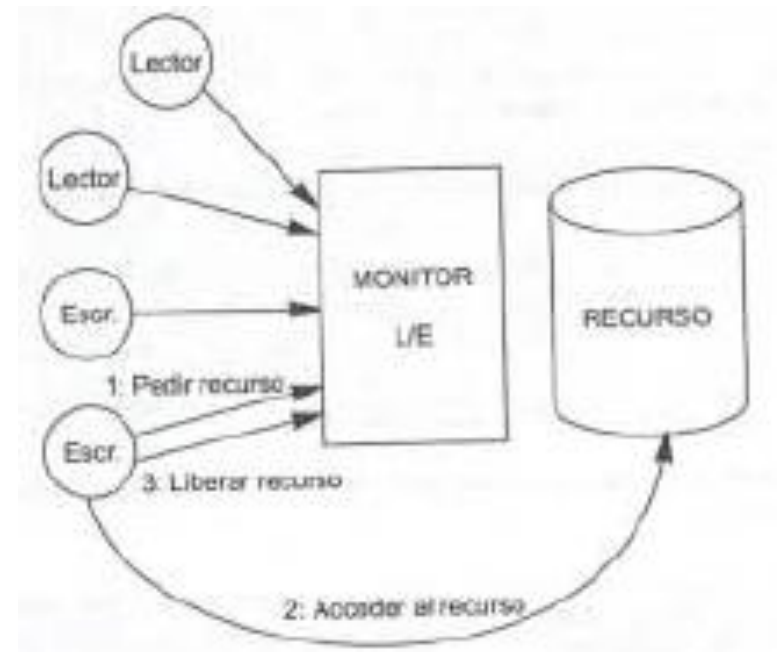
```
monitor buffer;  
const  
    N= //Tamaño del buffer;  
Var  
    lleno: condition; // bloquear al productor cuando el buffer esté lleno  
    vacio: condition; // bloquear al consumidor cuando el buffer esté vacío  
    recurso: tContenedor; //Habría que definir este TDA  
export  
    insertar, extraer;  
  
procedure insertar(elemento:item)  
begin  
    if recurso.size()=N then delay (lleno); //bloquearse si no hay espacio.  
    recurso.insert(elemento);  
    resume (vacio); //desbloquear al consumidor.  
end;  
  
function extraer: item;  
begin  
    if recurso.size()=0 then delay (vacio); //bloquearse si no hay elementos en el buffer  
    extraer=recurso.extract();  
    resume (lleno); //desbloquear a un productor.  
end;  
begin  
    recurso=new tContenedor(); // Llamada al constructor del buffer  
end;
```

Lectores/escritores

- ▶ tipos de procesos activos:
 - Lectores
 - Escritores
- ▶ El recurso compartido es el almacenamiento sobre el que los procesos activos operan
 - Operaciones exportables:
 - Leer
 - escribir
- ▶ Vamos a resolverlo a dos niveles:
 - Controlar el acceso al recurso
 - Controlar la lectura/escritura

Lectores/escritores

- ▶ Acceso al recurso:
 - operaciones exportables
 - abrir_lectura
 - cerrar_lectura,
 - abrir_escritura,
 - cerrar_escritura.



lectores/escritores

```
....  
    control_acceso: monitor_L_E;  
    almacen: contenedor; // Almacén compartido  
  
process lector;  
begin  
    control_acceso.abrir_lectura ();  
    almacen.read ();  
    control_acceso.cerrar_lectura();  
end;  
  
process escritor;  
Begin  
    control_acceso.abrir_escritura();  
    almacen.write(element);  
    control_acceso.cerrar_escritura();  
end;
```

```
Monitor acceso_L_E; // Prioridad en la lectura  
var  
    nl:integer; // N° de lectores leyendo  
    escribiendo:boolean; // ¿Hay escritores escribiendo?  
    lector:condition; // Cola en la que esperarán los lectores  
    escritor:condition; // Cola en la que esperarán los escritores  
export  
    abrir_lectura, cerrar_lectura, abrir_escritura, cerrar_escritura;
```


lectores / escritores (Prioridad en lectura)

```
procedure abrir_lectura;  
begin  
    if (escribiendo) then delay(lector) // bloquearse si un proceso escritor está accediendo.  
    nl=nl+1;  
    resume(lector); //desbloquear en cadena al resto de lectores.  
end;  
  
procedure cerrar_lectura;  
begin  
    nl=nl-1;  
    if (nl=0) then resume(escriptor); // desbloquear a un escritor si no quedan lectores  
  
end;  
  
procedure abrir_escritura;  
begin  
    if (nl <> 0) or escribiendo then delay(escriptor); // bloquearse si el recurso está ocupado.  
    escribiendo:=true  
end;  
  
procedure cerrar_escritura;  
begin  
    escribiendo=false;  
    if empty(lector) then resume(escriptor) // desbloquear a un escritor si no hay lectores esperando.  
    else resume(lector); //sino desbloquear a un lector.  
end;
```

lectores / escritores (Prioridad en la escritura)

```
procedure abrir_lectura;  
begin  
    if (escribiendo) or not empty(escritor) then delay(lector) // bloquearse si un escritor  
                                                                // está accediendo o bloqueado.  
    nl=nl+1;  
    resume(lector); //desbloquear al resto de lectores.  
end;  
  
procedure cerrar_lectura;  
begin  
    nl=nl-1;  
    if (nl=0) then resume(escritor) //desbloquear a un escritor si no quedan lectores  
end;  
  
procedure abrir_escritura;  
begin  
    if (nl<>0) or escribiendo then delay(escritor); //bloquearse si el recurso está ocupado.  
    escribiendo=true  
end;  
  
procedure cerrar_escritura;  
begin  
    escribiendo:=false;  
    if not empty(lector) then resume(lector) // si hay lectores bloqueados despertar al primero.  
    else resume(escritor) //sino desbloquear a un escritor.  
end;
```

Comida de los filósofos

- ▶ Acceso al recurso:

- Palillos
- operaciones exportables
 - Coger_palillos, Soltar_palillo

- ▶ Usaremos:

Un array para los estados de los filósofos (thinking, eating, hungry)

Un array de variables condición en la que se bloqueará cada filósofo.

```
monitor comida_filosofos;  
const N=5; // N° de filósofos  
var  
    estado: array [0.. N-1] of (thinking, eating, hungry) ;  
    dormir: array [0..N-1] of condition;  
    i:integer; // Variable privada  
export coger_palillos , soltar_palillos;  
begin  
    for i=0 to N-1 do estado[i]=pensando;  
end.
```

Filósofos

```
procedure coger_palillos(i:integer);
begin
    estado[i] =hungry; // Deseo comer
    comprueba(i) ; // ¿Puedo comer?
    if estado [i] <>eating then delay (dormir[i]);
end;

procedure soltar_palillos (i:integer);
begin
    estado[i] =thinking;
    //Deja de comer e intenta desbloquear a los filósofos que tiene a ambos lados.
    comprueba((i-1)+N) mod N); //Sumamos N para evitar negativos
    comprueba((i+1) mod N)
end;

procedure comprueba (k: integer);
begin
    If (estado[k]) == hungry) then
        //Comprueba si los filósofos de los lados están comiendo
        If (estado[(k-1)+N) mod N]<>eating) and (estado[(k+1) mod N] <>eating) then
            estado [k]=eating;
            resume (dormir[k]);
end;
```

Implementación de monitores mediante semáforos

- ▶ Para garantizar la exclusión mutua en el acceso al monitor, la ejecución de los procedimientos usará un semáforo `m_mutex` (inicializado a 1);

```
wait (m_mutex);  
procedimiento();  
signal(m_mutex);
```

Implementación de monitores mediante semáforos

- ▶ Implementar cola de cortesía
 - Mediante un semáforo `m_next`, inicializado a 0

```
wait (m_mutex) ;  
Cuerpo del procedimiento;  
//Si hay procesos bloqueados en la cola de cortesía, desbloquearlos  
if m_next_count>0 then signal(m_next);  
else signal (m_mutex) ;
```

Implementación de monitores mediante semáforos

- ▶ Implementar de variables condición
 - La cola FIFO asociada a cada variable la representaremos mediante un semáforo m_x
 - Contador de procesos en cola en cada variable

```
Delay (m_x)
begin
    m_x_count=m_x_count+1;
    if m_next_count<>0 then signal (m_next)
    else signal (m_mutex) ;
    wait (m_x_sem) ;
    m_x_count=m_x_count-1;
end;
```

Implementación de monitores mediante semáforos

- ▶ Implementar de variables condición
 - La cola FIFO asociada a cada variable la representaremos mediante un semáforo m_x
 - Contador de procesos en cola en cada variable

```
resume(m_x)
begin
  if m_x_count <> 0 then begin
    m_next_count = m_next_count + 1;
    Signal(m_x_sem);
    wait(m_next);
    m_next_count = m_next_count - 1;
  end;
end;
```


Implementación de monitores mediante semáforos

- ▶ Implementar de variables condición
 - La cola FIFO asociada a cada variable la representaremos mediante un semáforo m_x
 - Contador de procesos en cola en cada variable

```
empty(m_x)
begin
    return (mx_count==0);
end;
```