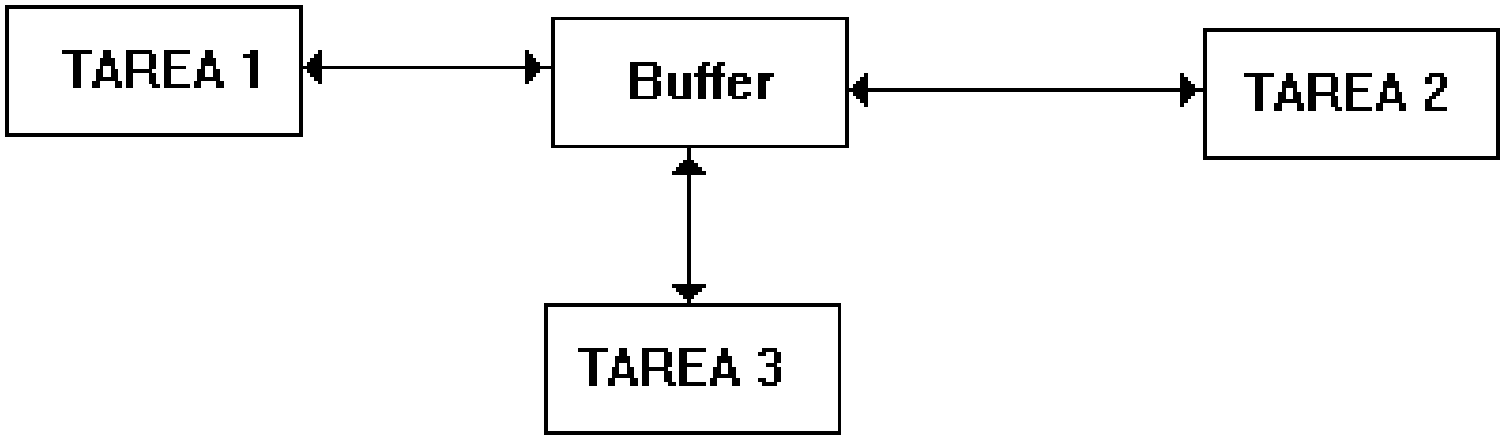
Concurrencia

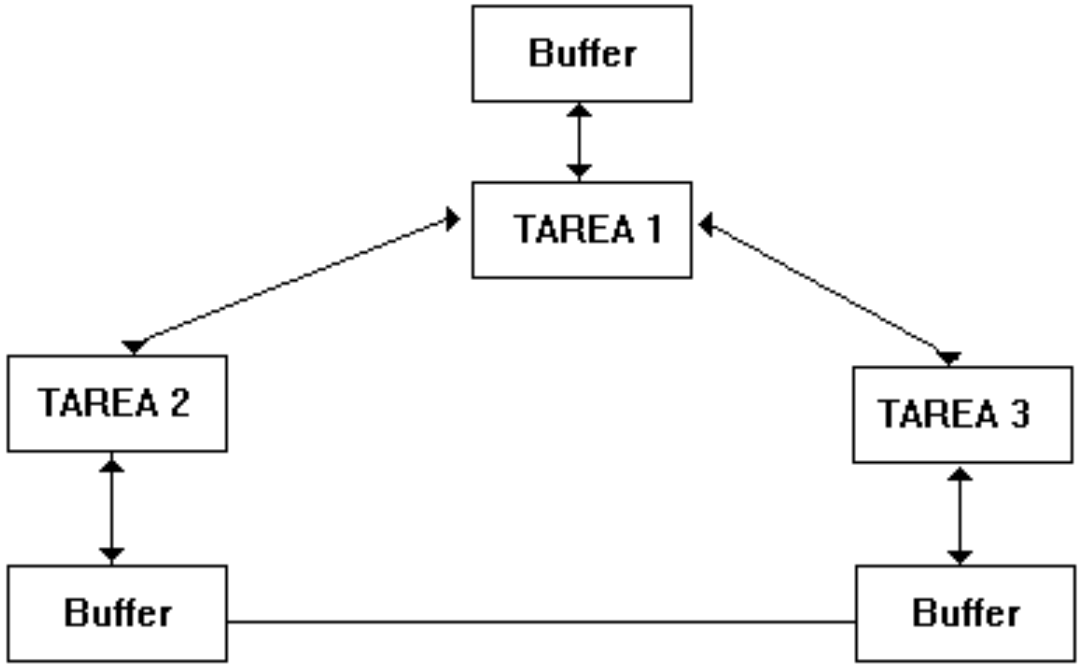
Procesos y Programas

Establecer Comunicación entre procesos (memoria común)



- Comunicación a través de un área común de memoria.

Establecer Comunicación entre procesos (mensajes)



- Comunicación mediante el intercambio de mensajes.

Razones para la Concurrencia

```
Programa
                    N = 40;
                    var
                    a: Vector (1..N) Integer;
                    k: Integer;
  procedure Ordenar (inferior, superior);
                    i, j, temp: Integer;
                    begin
                    for i := inferior to (superior - 1) do
                              begin
                              for j := (i + 1) to superior do
                                        begin
                                        if a(j) < a(i) then
                                                  begin
                                                  temp := a(j);
                                                  a(j) := a(i);
                                                  a(i) := temp
                                                  end;
                                        end;
                              end;
                    end;4
  Begin (* programa principal *)
                    for k:= 1 to n do Read(a,(k));
                    Ordenar (1,n);
                    for k:= 1 to n do Write (a(k));
  end. (* programa principal *)
```

Razones para la Concurrencia

- El orden del algoritmo utilizado es
- (N-1) + (N-2) + ... + 1 = (N + N + ... + N) - (1 + ... + (N- 1)) =

  = (N^2 - N) / 2 = (N (N - 1)) / 2
- que es aproximadamente igual a
  - **O (N^2 / 2)**.

Razones para la Concurrencia (Dividir es mejor ...)

- Si reemplazamos Ordenar (1,N) por

  Ordenar (1, N div 2);

  Ordenar (N div 2 + 1, N);

  Combinar (1, N div2, N div 2 + 1, N);

- Obtenemos que el orden es ahora

  $2 * (N / 2)^2 / 2 + N = 2 * (N^2) / 8 + N$

- O sea $O((N^2 / 4) + N)$

Razones para la Concurrencia (Dividir es mejor ...)

- Pero que si lo consideramos en paralelo resulta O ((N^2 / 8) + N)
- Comparando esta situaciones resulta

		Sin paralelismo	Con Paralelismo
n	$N^2/2$	$(n^2/4)+n$	$(n^2/8)+n$
40	800	440	240
100	5000	2.600	1350
1000	500.000	251.000	126.000

Razones para la Concurrencia (TR)

- Lo anterior se necesita para tareas críticas como TR
- Tiempo Real necesita:
- Adminsitración de Procesador compartible de manera activa
- Soporte para tareas concurrentes bajo esquema de prioridades
- Soporte de comunicación y sincronización entre tareas para cooperación

Razones para la Concurrencia (TR)

- Tiempo Real (Ej.: TM = Tasa Monotónica Creciente)
- Ci = peor uso de procesador Ti = Período
- Cumplir metas significa que cada proceso se ejecute en su Ti
- Una forma poco refinada de saber si se cumplen las metas de todos los procesos es por medio de

$$\Sigma \; Ci / Ti <= 1$$

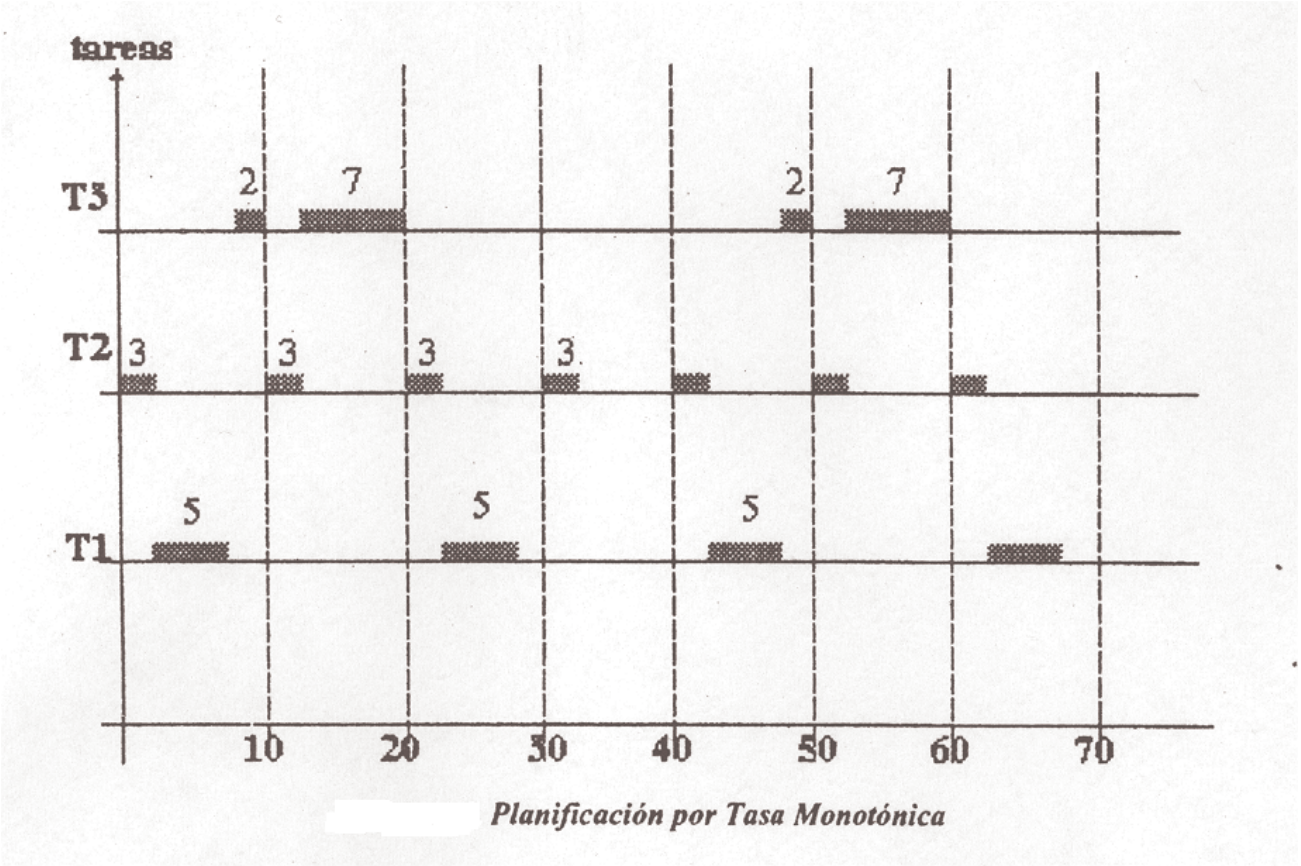
Razones para la Concurrencia (TR)

- Supogamos

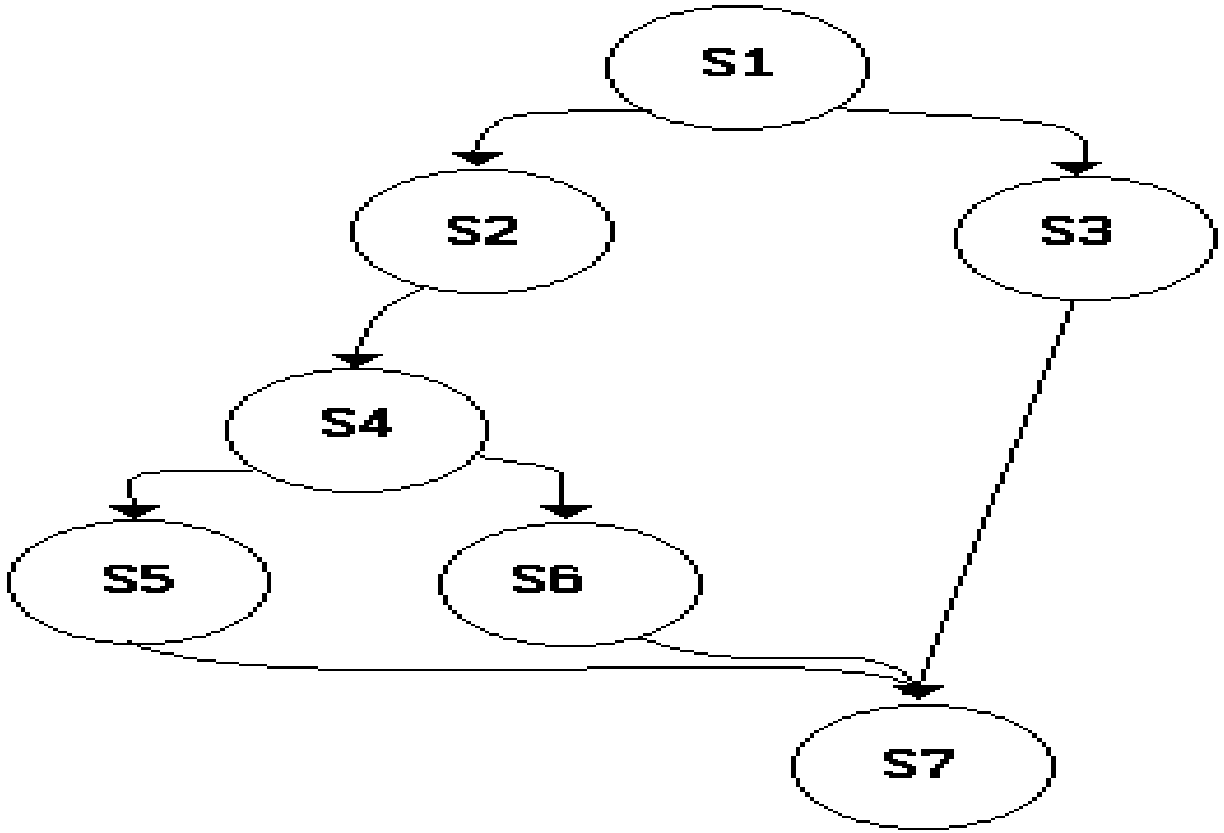
i	1	2	3
Ci	5	3	9
Ti	20	10	40

Σ Ci / Ti = 5/20 + 3/10 + 9/40 = 0.775 <= 1

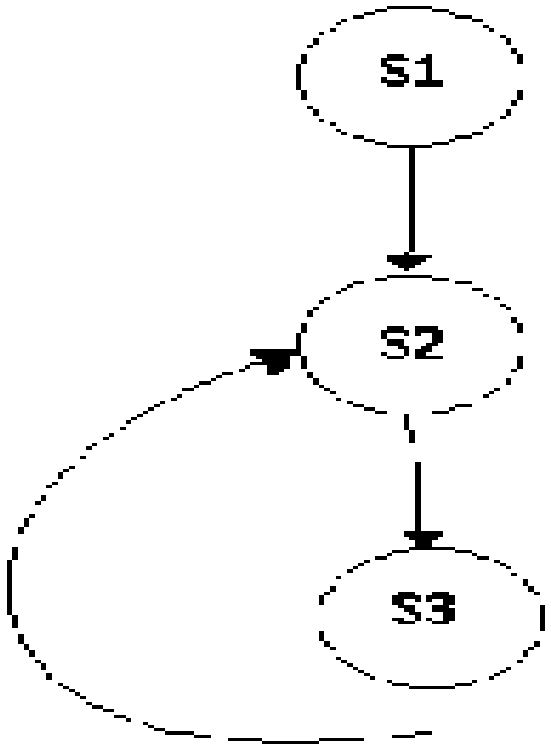
Razones para la Concurrencia (TR)



Grafos de Precedencia



Grafos de Precedencia sin ciclos



Condiciones de Berstein

- **Dados**
- **Conjunto lectura**

  R (Si) = (a1, a2, ... , am)
- **Conjunto escritura**

  W (Si) = (b1, b2, ... , bn)

Condiciones de Berstein

- Si se cumple
  1. $R(Si) \cap W(Sj) = (\emptyset)$.
  2. $W(Si) \cap R(Sj) = (\emptyset)$.
  3. $W(Si) \cap W(Sj) = (\emptyset)$.
- Entonces i j **pueden** ser concurrentes
- Grafos ➔ dependencias transitivas

Dependencias Transitivas

- Dada esta secuencia de intrucciones
  - a = 1
  - b = a + 1
  - c = a + 1
  - d = b + c
  - e = d + 1
- De acuerdo a Bernstein a, b, c, pueden ser concurrentes con e ¿?

Dependencias Transitivas

- Aplicamos Bernstein y vemos las dependencias
- Matriz de Dependencias (Diagonal Superior)

	1	2	3	4	5
1		1	1	0	0
2			0	1	0
3				1	0
4					1
5					

(1) Matriz A

Dependencias Transitivas

- Aplicamos
- **Σ** A ^ i desde i = 1 hasta n-1, siendo n el nro de nodos (instrucciones)
- Que determina la cantidad de caminos por los que puede alcanzar un nodo

Dependencias Transitivas

0 1 1 0 0		0 1 1 0 0		0 0 0 2 0
0 0 0 1 0		0 0 0 1 0		0 0 0 0 1
0 0 0 1 0	x	0 0 0 1 0	=	0 0 0 0 1
0 0 0 0 1		0 0 0 0 1		0 0 0 0 0
0 0 0 0 0		0 0 0 0 0		0 0 0 0 0
(1)				(2)

- (2) alacanzables longitud 2

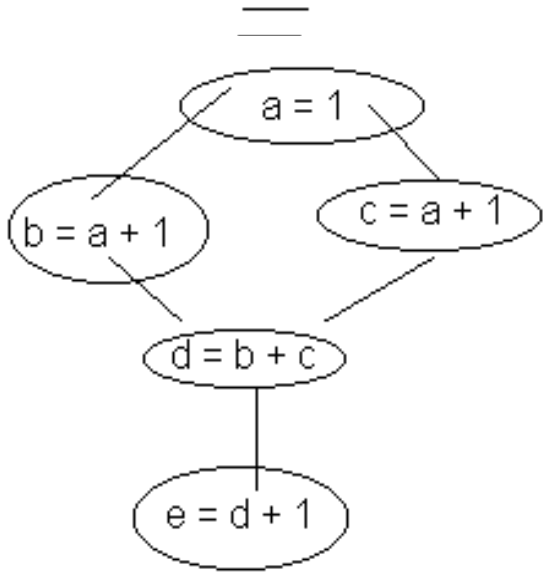
Dependencias Transitivas

- 0 0 0 2 0 0 1 1 0 0 0 0 0 0 2
- 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0
- 0 0 0 0 1 x 0 0 0 1 0 = 0 0 0 0 0
- 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
- 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
- (2) (3) alc long 3
- (opt hasta A^k = 0 con 1 <= K <= n-1

Dependencias Transitivas

- Si sumamos (1) + (2) + (3) obtenemos
- Y observamos (1) y (4)

	1	2	3	4	5	
1	0	1	1	2	2	
2	0	0	0	1	1	
3	0	0	0	1	1	(4)
4	0	0	0	0	1	
5	0	0	0	0	0	



Concurrencia dependiendo del nivel k

do i = 1, n		do i = 1,n
S1: a(i+1) = f(i)		S2: f(i+1) = a (i)
S2: f(i+1) = a(i)	=	S1: a(i+1) = f (i)
enddo		enddo

- Concurrentes S1 y S2 mientras se mantengan en el nivel k

Crear Concurrencia (threads)

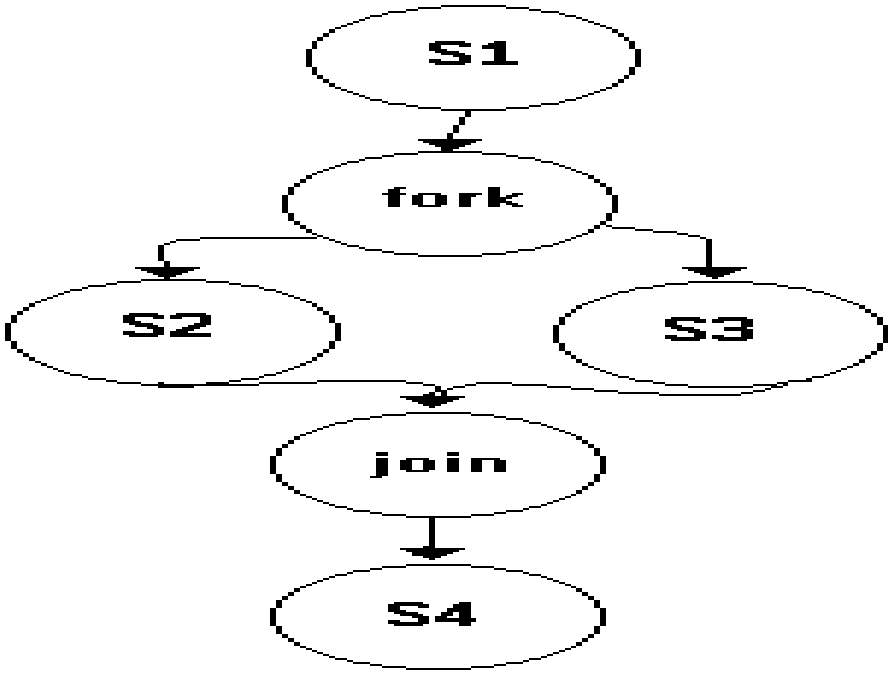
```
main (){
    int a; thread_t cualid; int resu; int satus;int ruti();
    for (a=0,a<1500;a++){
    thr_create(NULL,NULL,ruti,NULL,THR_NEW_LPW,&cualid;}
    }

    ruti() {
       int resu; int b;
       b++;
       thr_exit(&resu);
    }
```

Crear Concurrencia (procesos)

```
main (){
    int a;  int ruti();
    for (a=0,a<1500;a++){
        if (fork()==0) {ruti(); exit(0);}
        }
    }
    ruti() {
        int  b=0;
        b++;
    }
```

Primitivas fork join

- La instrucción fork crea concurrencia y la instrucción join recombina concurrencia en una única secuencia.



Primitivas fork join

```
        - - -
        S1;
        fork L1;
        S2;
        go to L2
L1:     S3;
L2:     join;
        S4;
```

Primitivas fork join

- introducimos la variable count, que toma un valor en el fork y que dentro del join opera como:

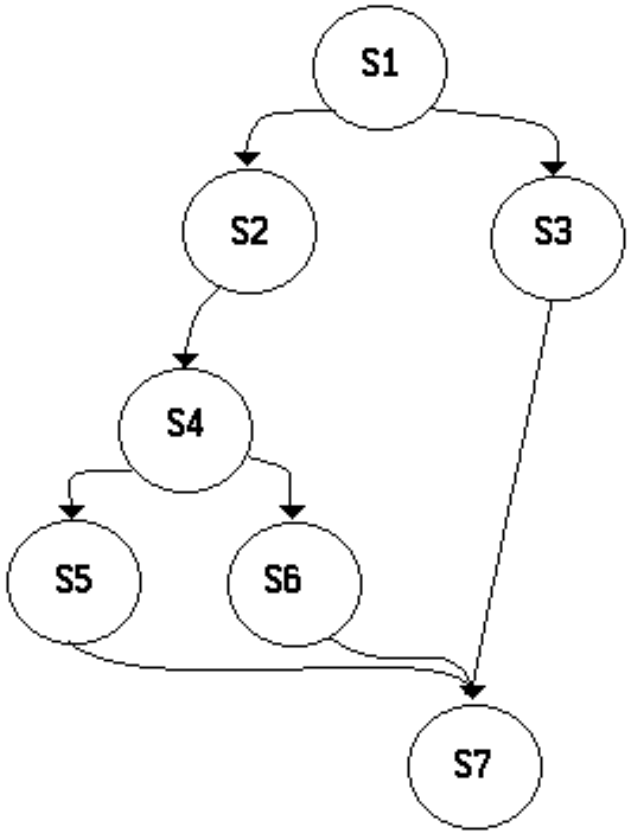
  count = count - 1;

  if count not = 0 then *Espera*;

Primitivas fork join

```
        S1;
        fork L1, count = 2;
        S2;
        go to L2;
L1:     S3;
L2:     join count;
        S4;
        - - - - -
```

Primitivas fork join



Primitivas fork join

- Tiene como solución

```
        S1
        fork L1, count1 = 2
        S2
        S4
        fork L2, count2 = 2
        S5
        goto L4
L2      S6
L4      join, count2
        goto L3
L1      S3
L3      join, count1
        S7
```

Ejemplo de copia de un archivo secuencial f en otro g

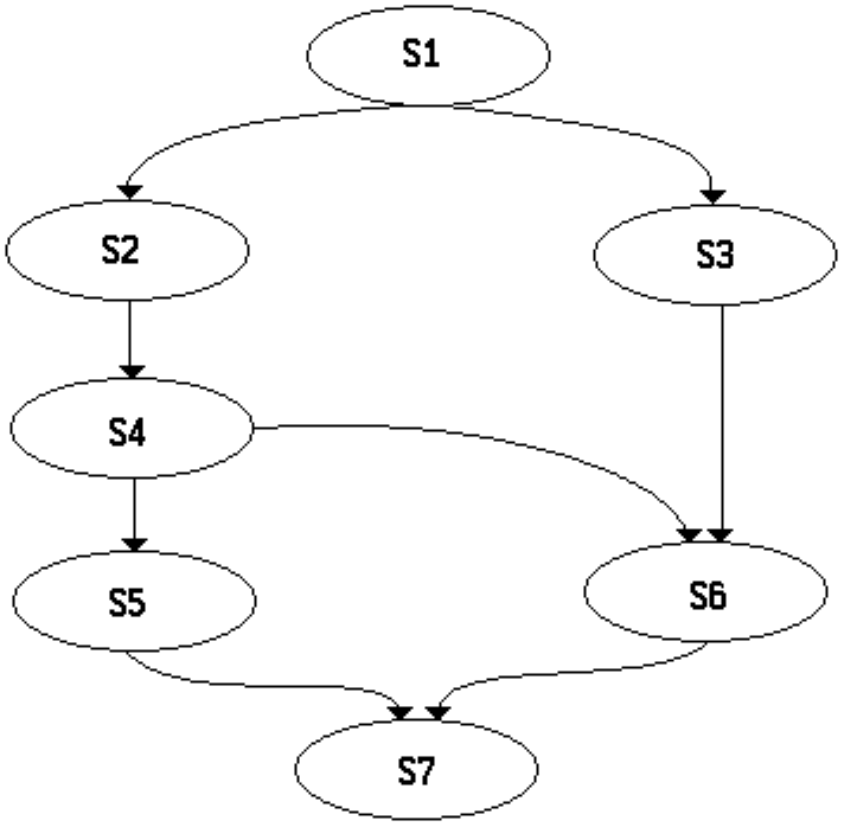
```
begin
        Read (f, r);
        while not eof(f) do
                begin
                 fork L1,count := 2;
                s := r;
                Write (g, s);
                go to L2;
        L1:     Read (f, r);
        L2:     join count;
                end;
        Write (g, r);
end.
```

Primitivas parbegin parend o cobegin coend

- Fork y join no estructurado
- parbegin parend estructurado (vemos el grafo anterior)

```
S1
Parbegin
        S3
        begin
            S2
            S4
            parbegin
                S5
                S6
            parend
        end
parend
S7
```

¿Se puede representar todos los grafos?



fork join

```
        S1;
        fork L1, cuenta1 = 2;
        S2;
        S4;
        fork L2, cuenta2 = 2;
        S5;
        go to L3;
L1:     S3;
L2:     join cuenta1;
        S6;
L3:     join cuenta2;
        S7;
```

parbegin parend

- **S1**
- **parbegin**
- **S3**
- **begin**

  S2 S4

  parbegin

  S5 S6 → ? → S6 depende de S3

parbegin parend (alternativa)

- **S1**
- **parbegin**
- **S3**
- **begin**

S2 S4

end

Parend

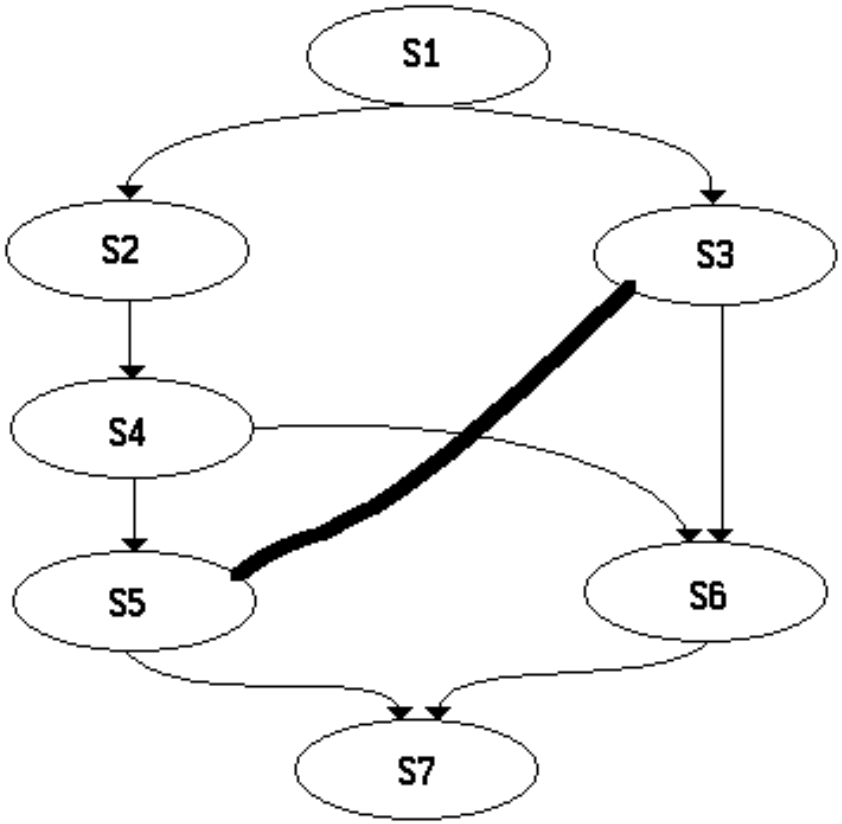
Parbegin

S5 S6

parend

S7

parbegin parend (alternativa) ¿es el mismo grafo?



parbegin parend

- Se generó una dependencia que en el grafo original no existe
- Restringe el paralelismo
- Pero es un grafo equivalente
- No todos los grafos se pueden representar con parbegin parend
- Sólo se pueden representar los que tengan subgrafos disjuntos concurrentes

Problemas críticos de la concurrencia

Premisas de Dijkstra

- Dijkstra, planteó las siguientes cuatro (+ 1) premisas:
- *1)- No deben hacerse suposiciones sobre las instrucciones de máquina ni la cantidad de procesadores. Sin embargo, se supone que las instrucciones de máquina (Load, Store, Test) son ejecutadas atómicamente, o sea que si son ejecutadas simultáneamente el resultado es equivalente a su ejecución secuencial en un orden desconocido.*
- *2)- No deben hacerse suposiciones sobre la velocidad de los procesos.*
- *3)- Cuando un proceso no está en su región crítica no debe impedir que los demás ingresen a su región crítica.*
- *4)- La decisión de qué procesos entran a una parte crítica no puede ser pospuesta indefinidamente.*
- Los puntos 3) y 4) evitan bloqueos mutuos.
- *5)- Se limita la cantidad de veces que un proceso ingresa a la sección crítica*

Algoritmo 1

- Sem inicializada en 0 ó 1 ó i

  While Sem not = i Do Skip

  SECCION CRITICA

  Sem = J

  SECCION No-crítica

- Esta solución asegura un proceso a la vez en la zona crítica pero no respeta la premisa 3), pues si Sem = 0 y el proceso P1 quiere hacer uso no puede aunque P0 no quiera usarlo.
- Una tarea ejecuta en cadencia con otra (10 a 100 ?). Si no se ejecuta Sem = J las demás no entran.

Algoritmo 2

Para remediar lo anterior reemplazamos la variable Sem por un vector donde cada elemento puede ser V ó F y está inicializado en F.

- **While Vector(j) Do Skip**
- **Vector(i) = V**
- **SECCION CRITICA**
- **Vector(i) = F**
- **Sección no-crítica**

Pero este algoritmo no asegura que un solo proceso esté en la zona crítica como puede verse si se sucede la siguiente secuencia:

- T0 P0 encuentra Vector(1) = F
- T1 P1 encuentra Vector(0) = F
- T2 P1 pone Vector(1) = V
- T3 P0 pone Vector(0) = V
- La secuencia anterior puede ocurrir con varios procesadores o con uno solo y cuando el P0 sea interrumpido por un cualquier evento, por ejemplo, un reloj de intervalos.

Algoritmo 3

- El problema anterior es que Pi toma una decisión sobre el estado de Pj antes que Pj tenga la oportunidad de cambiarlo; tratamos de corregir este problema haciendo que Pi ponga su indicador en V indicando que "solo" quiere "entrar" en la sección crítica.
- Vector(i) = V
- While Vector(j) Do Skip
- SECCION CRITICA
- Vector(i) = F
- Sección no-crítica
- Pero aquí la premisa 4) no se cumple, pues si se sucede:
- T0 P0 coloca Vector(0) = V
- T1 P1 coloca Vector(1) = V ambos procesos quedan en loop en la instrucción While (espera indefinida).

Algoritmo 4

La falla anterior se debió a no conocer el preciso estado en que se encontraban los otros procesos. Veamos ahora esta propuesta:

```
Vector(i) = V
While Vector(j)
        Do Begin
           Vector(i) = F
            While Vector(j) Do Skip     (*)
            Vector(i) = V
        End
           SECCION CRITICA
           Vector(i) = F
```

- Esta solución vuelve a no cumplir con la premisa 4), pues si ambos procesos ejecutan a la misma velocidad quedan en loop (Ver While (*)).

Algoritmo 5

Una solución correcta debida al matemático alemán T. Dekker.

Usa Vector(0,1) y Turno(0,1)

Inicialmente Vector(0) = Vector(1) = F y Turno = 0 ó 1.

```
Vector(i) = V
        While Vector(j) Do
                If Turno = j
                Then    Begin
                        Vector(i) = F
                        While Turno = j Do Skip         (*)
                        Vector(i) = V
                        End
                End do
        SECCION CRITICA
        Turno = j
        Vector(i) = F
```

Algoritmo 5

- Con esto se puede verificar que :
- a) existe la mutua exclusión ya que Pi modifica Vector(i) y solo controla Vector(j), y
- b) el bloqueo no puede ocurrir porque Turno es modificado al final del proceso y habilita al otro.
- En realidad Vector controla si se puede entrar y Turno cuándo se puede entrar. Si Turno es igual a 0 se le puede dar prioridad a P0.
- Este algoritmo consiste en una combinación de la primera y la cuarta solución.
- De la primera solución extrae la idea del pasaje explícito de control para entrar en la región crítica; y de la cuarta el hecho de que cada tarea posee su propia llave de acceso a la región crítica.

Algoritmo 6

El anterior algoritmo contempla el problema para 2 procesos, veamos la solución de Dijkstra (1965) para n procesos.

- Usa Vector(0,.....,n-1) que puede valer 0, 1 o 2; 0 indica ocioso, 1 indica quiere entrar y 2 indica dentro.
-

Inicialmente Vector(i) = 0 para todo i y Turno = 0 ó 1 ó .... ó n-1.

```
(1)   Repeat(
      Vector(i) = 1;
      While Turno not = i Do
                        If Vector(Turno) = 0 then turno = i;
      Vector(i) = 2;
      j = 0;
      While (j < n) and (j = i ó Vector(j) not = 2) Do j=j+1;
(1)   Until j > or = n;
      SECCION CRITICA
      Vector(i) = 0;
```

Algoritmo 6

- a) La mutua exclusión está dada porque solo Pi entra en su sección crítica si Vector(j) es distinto de 2 para todo j distinto de i y dado que solo Pi puede poner Vector(i) = 2 y solo Pi inspecciona Vector(j) mientras Vector(i) = 2.
- b) El bloqueo mutuo no puede ocurrir porque:
- - Pi ejecuta Vector(i) not = 0
- - Vector(i) = 2 no implica Turno = i. Varios procesos pueden preguntar por el estado de Vector(Turno) simultáneamente y encuentran Vector(Turno) = 0. De todas maneras, cuando Pi pone Turno = i, ningún otro proceso Pk que no haya requerido ya Vector(Turno) estará habilitado para poner Turno=k, o sea no pueden poner Vector(k) = 2.
- Supongamos {P1,.....,Pm} tengan Vector(i) = 2 y Turno = k ($1 \leq k \leq m$). Todos estos procesos salen del segundo While con j < n y retornan al principio de la instrucción Repeat, poniendo Vector(i) = 1, entonces todos los procesos (excepto Pk) entrarán en loop en el primer While, luego Pk encuentra Vector(j) not = 2 para todo i distinto de j y entra en su sección crítica.
- Este algoritmo cumple con todo lo pedido, pero puede suceder que un proceso quede a la espera mientras otros entran y salen a gusto, para evitar esto se pone una quinta premisa.
- *5)- Debe existir un límite al número de veces que otros procesos están habilitados a entrar a secciones críticas después que un proceso haya hecho su pedido.*

Eisenberg y McGuire

- El primer algoritmo que cumplió las premisas fue el de Knuth (1966) (esperando 2^n turnos), luego De Bruijin (1967) lo redujo a n^2 turnos y finalmente Eisenberg y McGuire (1972) desarrollaron uno que redujo la espera de turnos a n-1, y este es el algoritmo:

Eisenberg y McGuire

- **Repeat**

  Flag(i) = Intento **(quiere entrar)**

  j = turno

  while j not = i do

  if flag(j) not = Ocioso then j = turno

  else j = (j+1) Mod n

  flag(i) = En-SC **(en sección crítica)**

  j = 0

  while (j < n) and (j=i or flag(j) not = En-SC) do j = j+1;

  Until (j ≥ n) and (Turno = i or flag(Turno) = Ocioso);

  SECCION CRITICA

  j = Mod n(Turno + 1)

  While (j not = Turno) and (flag(j) = Ocioso) do j = (j+1) Mod n;

  Turno = j

  flag(i) = Ocioso

Panadería de Lamport

- Si Pi y Pj recibieron el mismo número y i < j se atiende primero a Pi (esta característica lo hace determinístico).
- Usaremos la siguiente notación.
- (a,b) < (c,d)

  si a < c

  si a = c y b < d

Panadería de Lamport

```
Tomar.array(0,.....,n-1) boolean;          inicio F
Número.array(0,.....,n-1) integer;         inicio 0
Para Pi
    Tomar(i) = V                  (avisa que está eligiendo)
    Número(i) = max (Número(0),...,Número(n-1)) + 1;
                                  (Aquí 2 procesos pueden tomar el mismo número)
    Tomar(i) = F
    For j = 0 to n-1              (Aquí comienza la secuencia  de control)
        Begin
        While Tomar(j) do Skip;
                                  (evita entrar en uno que está eligiendo número mientras es V)
        While Número(j) not = 0 and (Número(j),j) < (Número(i),i) do Skip;
                                  (se asegura ser el menor de todos)
        end
    SECCION CRITICA
    Número(i) = 0;                (salida)
    Sección no-crítica
```

SOLUCION HARDWARE PARA LA EXCLUSION MUTUA

- **Función Test_and_Set**(x)

  begin

  Test_and_Set = x

  x = v

  end

- Si la máquina cuenta con la instrucción Test_and_Set, la exclusión mutua puede ser implementada de la siguiente manera declarando previamente la variable booleana **lock** inicializada en FALSE.

  repeat

  while Test_and_Set (lock) do skip;

  < SECCION CRITICA >;

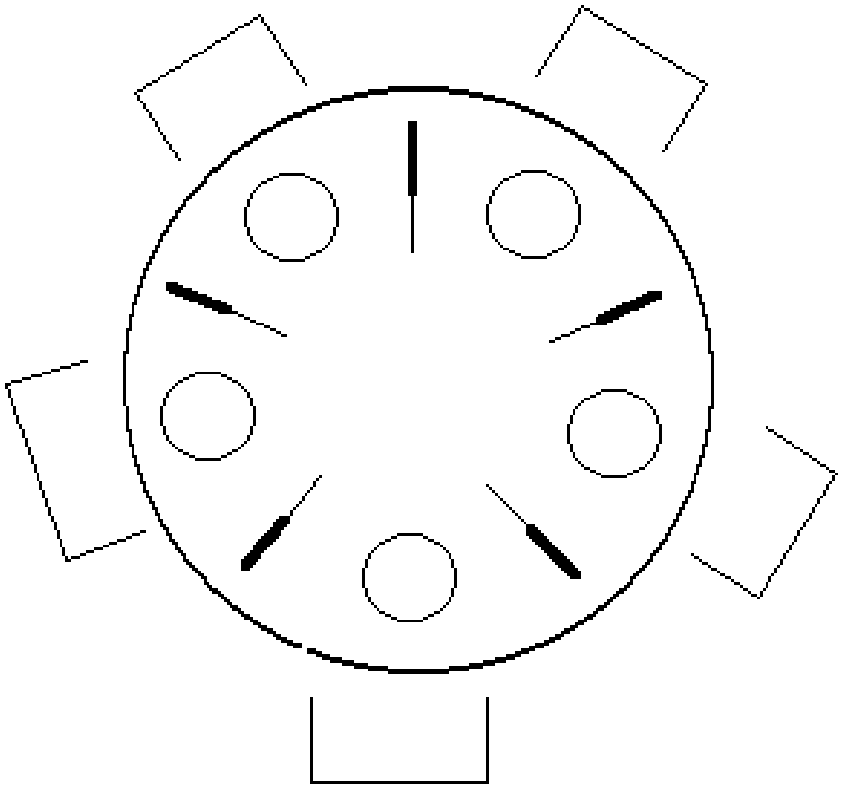
  lock := FALSE;

  < SECCION NO CRITICA >

Semáforos

- Gracias a lo anterior se pueden implementar semáforos y operadores P y V
- Pero ... Sirven para todo ???

Filósofos chinos que cenan



Filósofos chinos que cenan (deadlock ???)

Do forever

P (palito (i));

P (palito (mod 5 (i+1)));

< Comer >;

V (palito(i));

V (palito (mod 5 (i+1)));

< Pensar >

Lectores Escritores

- Escritores: mutuamente excluyentes
- Lectores y Escritores: mutuamente excluyentes
  - 1er. Problema: Ningún lector espera a menos que haya un escritor
  - 2do. Problema: Si un escritor espera acceder ningún lector "puede comenzar a leer"
- (soluciones para ambos problemas pueden generar esperas infinitas)
  - Para 1er. Problema: Los escritores pueden quedar bloqueados infinitamente
  - Para 2do. Problema: Los lectores pueden quedar bloqueados infinitamente

Lectores Escritores (1er. Problema)

Var: exclu, escr : semáforo (valor inicial 1)

Var: cant_lec : entero (valor inicial 0)

- Escritor

```
P(escr);
ESCRIBE;
V(escr);
```

Lectores Escritores (1er. Problema)

- Lector

```
P(exclu);
   Cant_lec ++;
   If cant_lec = 1 then p(escr);
V(exclu);
LEE;
P(exclu);
   Cant_lec --;
   If cant_lec = 0 then v(escr);
V(exclu)
```

- Si bien los escritores pueden esperar infinitamente no ocurre ningún otro problema pues los semáforos planifican sus colas de espera FIFO.

Lectores Escritores (2 do. Problema)

Var: exclu_es, exclu_lec, lec, escr : semáforo (init = 1)

Var: cant_es, cant_lec, flag : enteros (init = 0)

- Escritor

```
P(exclu_es);
        Cant_es ++;
        If cant_es = 1 then p(lec);
V(exclu_es);
P(escr);
        ESCRIBIR;
V(escr);
P(exclu_es);
        Cant_es --;
        If cant_es = 0 then v(lec);
V(exclu_es);
```

Lectores Escritores (2 do. Problema)

- Lector

```
P(exclu_lec);
        Cabt_lec ++;
        If cant_es not = 0 then
        Do
                P(lec); Flag:=1: If cant_lec = 1 then p(escr);
        End;
V(exclu_lec);
LEER;
P(exclu_lec);
        Cant_lec--;
        If  cant_lec = 1 then
        Do
                Flag:=0; V(lec);
        End;
V(exclu_lec);
```

Lectores Escritores (2 do. Problema)

- Problemas varios:
- Cant-es es una variable compartida, bien si estamos en threads y la declaramos global, mal si estamos en procesos, a menos que la usemos con shm (obliga otra implementación, la de shm). **O sea se necesita una nueva estructura**.
- Escritor NO controla que haya lectores leyendo cuando comienza a escribir (pero de todas maneras no es lo que se pide)
- Lo único que asegura es que si hay un escritor NO entren nuevos lectores (que si es lo pedido).

Regiones Críticas

- Una variable v de tipo T y que será compartida, se declara como :
- var v : shared T ;
- y cuando se la utilice
- region v do S;

Regiones Críticas

- O sea si el conjunto S utiliza la variable v lo podrá hacer únicamente de la forma anterior y de tal manera que si dos procesos la quieren utilizar en forma concurrente
- region v do S1;
- region v do S2;

Regiones Críticas

- Una implementación posible de esto cuando el compilador se encuentra con

  var v : shared T

- es que genere un semáforo x = 1 y cuando se encuentra con

  region v do S;

- genere

  P(x)

  S;

  V(x)

Regiones Críticas

- También se puede tener un anidamiento de regiones críticas:

  var x,y : shared T

  Parbegin

  Q1 : region x do region y do S1;

  Q2 : region y do region x do S2;

  parend

- En este caso nótese que hemos construido un deadlock:

  T0 Q1 P(x-x) (léase x de x)

  T1 Q2 P(y-x)

  T2 Q2 P(x-x) luego espera

  T3 Q1 P(y-x) luego espera

- Luego el compilador debería detectar estas situaciones o sino generar un orden. por ejemplo si y está en x e (y< x), reordenarlo.

Regiones Críticas Condicionales

- region V When B Do S;
  - donde B es una expresión booleana, que es evaluada, y si es verdad entonces se ejecuta S.
  - Si B es falsa el proceso libera la exclusión mutua y espera hasta que B sea verdadera y no exista otro proceso ejecutando en la región asociada a v.

Regiones Críticas Condicionales

- Veamos un ejemplo recreando el caso del productor/consumidor.
- El buffer usado está encapsulado de la siguiente forma:

```
var buff : shared record
b : array [ 0.....n ]
count, in, out, enteros
end
```

- El proceso Productor inserta un nuevo elemento PEPE:

```
region buff when count < n
        do begin
        b(in) = PEPE;
        in = (in+1) mod n
        count = count + 1;
        end
```

Regiones Críticas Condicionales

- El proceso Consumidor retira en SPEPE:

```
region buff when count > 0
        do begin
                SPEPE = b(out);
                out = (out+1) mod n;
                count = count - 1;
        end;
```

Regiones Críticas Condicionales

- El compilador debe trabajar de esta forma cuando se encuentra con una declaración. Por cada variable x compartida debe tener las siguientes variables asociadas:

  var x-sem, x-wait : semáforos;

  x-count, x-temp : enteros;

- Donde :

  x-sem = controla la exclusividad en la región crítica.

  x-wait = es el semáforo por el que hay que esperar cuando no se cumple la condición.

  x-count = cuenta la cantidad de procesos esperando por x-wait.

  x-temp = cuenta las veces que un proceso testeó la condición (booleana)

- Los valores iniciales son :

  x-sem = 1 x-wait = 0 x-count = 0 x-temp = 0

Regiones Críticas Condicionales

- **El compilador frente a la region x when B do S podría implementarlo de la siguiente manera :**

```
P(x-sem)
If not B Then
begin
   x-count = x-count + 1;
   V(x-sem);                          libera exclusión
   P(x-wait);                         se pone en espera
   while not B          se despierta pero no se cumple aún condición
      do  begin
            x-temp = x-temp + 1;
            if x-temp < x-count  (el sólo?)
                        then V(x-wait)
                        else V(x-sem);
            P(x-wait);
          end;
   x-count = x-count - 1;
end;
S;
if x-count > 0  then
            begin  x-temp = 0  V(x-wait) end;
else V(x-sem);
```

Monitores

- La idea básica de este mecanismo es la de agrupar, en un mismo módulo, las regiones críticas relacionadas con un determinado recurso, haciendo que toda región crítica se encuentre asociada a un procedimiento del monitor, el que podrá ser llamado por las tareas.
- La propiedad más importante que caracteriza a los monitores es que la ejecución de un procedimiento del monitor por una tarea excluye la ejecución de cualquier otro procedimiento del mismo monitor por otra tarea

Monitores

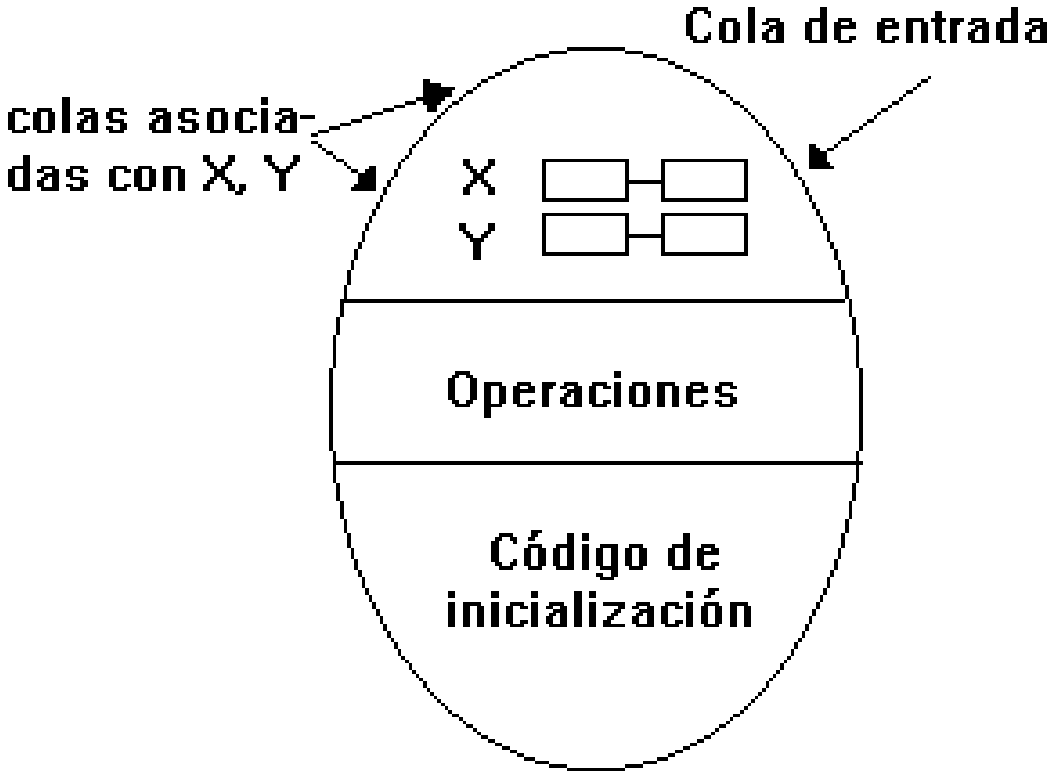
- Un monitor se escribe como:
  - un conjunto de declaración de variables
  - un conjunto de procedimientos
  - un cuerpo de comandos que son ejecutados inmediatamente después de la inicialización del programa que contiene al monitor.

Monitores

- La sintaxis de un monitor es la siguiente:

Monitor <nombre_del_monitor>;
    var < declaración de variables del monitor >
procedure <nombre_del_procedimiento1>;
        begin
        - - - - -
        end;
procedure <nombre_del_procedimiento2>;
        begin
        - - - - -
        end;
    begin
    < bloque de inicialización de variables del monitor >
    end.

Monitores


Cola de entrada
colas asocia-
das con X, Y
X
Y
Operaciones
Código de
inicialización

Monitores

- Pasivo (sólo se activa si es invocado uno de sus procedimientos
- La comunicación con el exterior es sólo por sus parámetros
- Usa primitivas wait y signal (signal no modifica estados si no hay esperas): Utilizan una variable de “condition” que es un método y una estructura

Monitor sencillo

- Ejemplo: Simulación de semáforos **programa Exclusión-Mutua**;
  monitor Simulación de semáforo;
  ocupado : boolean;
  no-ocupado : condición;
  procedure P (v);
  begin
  if ocupado then Wait (no-ocupado);
  ocupado = V;
  end;
  procedure V;
  begin
  ocupado = F;
  Signal (no-ocupado);
  end;
  begin (* Cuerpo del monitor *)
  ocupado = F;
  end; (* fin monitor *)

Monitor sencillo

```
tarea T1;
        begin
        P;
        Región Crítica;
        V;
        end;
tarea T2;
        begin
        P;
        Región Crítica;
        V;
        end;
Programa-Principal
        Begin;
                cobegin T1, T2; coend
        end.
```

Productor Consumidor

- **programa Productor_Consumidor**;
  MAX = ...;
  monitor M;
  buffer : Array (0..MAX-1);
  in, out, n; enteros;
  buff_lleno, buff_vacío: condición;
  procedure Almacenar (v);
  begin
  if n = MAX then Wait (buff_vacío);
  buffer (in) = v;
  in = (in + 1) mod MAX;
  n = n + 1;
  Signal (buff_lleno)
  end;

Productor Consumidor

```
procedure Retirar (v);
        begin
        if n = 0  then Wait (buff_lleno);
        v = buffer (out);
        out = (out + 1) mod MAX;
        n = n - 1;
        Signal (buff_vacio)
        end;
    begin (* Cuerpo del monitor *)
        in, out, n = 0;
    end;  (*     fin monitor    *)
```

Productor Consumidor

```
procedure Productor;
        begin
        v = "dato producido"
        Almacenar (v)
        end;
procedure Consumidor;
        begin
        Retirar (v);
        Hacer algo con v
        end;
begin Programa-Principal
        Begin;
                cobegin
                Productor;
                Consumidor
                coend
        end.
```

Lectores Escritores

```
Type lect_escr : monitor
Var cant_escr, cant_lect : enteros
Var no_escr, nadie : condition

Procedure lector
Do begin
    If cant_escr not = 0 then wait (no_escr);
    Cant_lect ++;
    LEER;
    Cant_lect --;
    If (cant_lect := 0 & cant_escr not = 0 ) then signal (nadie);
End
```

Lectores Escritores

```
Procedure escritor
Do begin
    Cant_escr ++;
    If (cant_lect not = 0 or cant_escr > 1) then wait
    (nadie);
    ESCRIBIR;
    Cant_escr --;
    If cant_escr not = 0       then signal (nadie);
                               Else signal (no_escr);
End
```

Lectores Escritores

```
Begin
    Cant_lect := 0;
    Cant_escr := 0;
End
```

- Parece que es correcto, pero si se piensa en las cualidades del monitor se verá que sólo se permite la lectura exclusiva y no concurrente entre los lectores, entonces hay que armar al monitor de otra manera

Lectores Escritores (varios lectores)

```
Type lect_escr : monitor
Var cant_escr, cant_lect, lect_espera : enteros
Var no_escr, nadie : condition

Procedure pedirlectura
Do begin
   Lect_espera ++;
   If cant_escr not = 0 then wait (no_escr);
   Cant_lect ++;
End
```

Lectores Escritores (varios lectores)

```
Procedure finlectura
    Cant_lect --;
    Lect_espera --;
    If (cant_lect := 0 & cant_escr not = 0 ) then signal (nadie);
    Else
        Do 1, 1, lect-espera
            Signal (no_escr);
        End;
End
```

Lectores Escritores (varios lectores)

```
Procedure escritor
Do begin
    Cant_escr ++;
    If (cant_lect not = 0 or cant_escr > 1) then wait (nadie);
    ESCRIBIR;
    Cant_escr --;
    If cant_escr not = 0 then signal (nadie);
          Else signal (no_escr);
End
```

Lectores Escritores (varios lectores)

- Begin
  Cant_lect := 0;
  Cant_escr := 0;
  Lect_espera :-0;
- End

- Desde programa, para hacer una lectura ahora se hace:
  p.pedirlectura;
  LEER;
  p.finlectura;

- Ahora se permiten lecturas concurrentes.

Monitor Filósofos

- type Filósofos = Monitor;
- var estado : array [0,...,4] of (Pensar, Hambre, Comer);
- var ocio : array [0,...,4] of condición;

  (para quedar en espera si tiene hambre pero no puede comer)

Monitor Filósofos

```
Procedure Tomar-Palitos (i : 0,...,4 )
        Begin
                estado(i) = Hambre;
                test (i);
                if estado(i) not = Comer then ocio(i).wait;
        end;
Procedure Dejar-Palitos ( i : 0,...,4 );
        Begin
                estado(i) = Pensar;
                test ( (i-1) mod 5 );
                test ( (i+1) mod 5 );
        end
```

Monitor Filósofos

```
Procedure test ( k : 0,...,4 )
    Begin
        if estado ( (k-1) mod 5 ) not = Comer and
                estado (k) = Hambre and
                estado ( (k+1) mod 5 ) not = Comer        (*)
        then    begin
                        estado (k) = Comer;
                        ocio(k).signal;
                end;
    end
Begin
    for i = 0 to 4
    do estado(i) = Pensar;
end
```

Monitor Filósofos

- La forma de invocarlo sería:
- x.Tomar-Palitos;

  .....

  Comer

  .....
- x.Dejar-Palitos;

Implementación variables condition

- P(x)
  Procedimiento
- V(x)

- Es muy grosero, además no permite "concurrencia"

Implementación monitor y variables condition

iniciales x=1, próximo = 0, cuenta = 0

(y-sem = 0, y-cuenta=0 asociados a la variable y).

(1) P(x)

Procedimiento

(2) wait (y)

(7) signal (y)

(12) If cuenta > 0 (cuenta los que están en espera)

(13) then V(próximo) (adentro)

else V(x) (afuera)

wait

(3) y-cuenta = y-cuenta + 1

(4) if cuenta > 0

then V(próximo)

(5) else V(x);

(6) P(y-sem) ➔ (sigo 10)

y-cuenta = y-cuenta + 1

signal

```
(8) if y-cuenta > 0 then
            begin
(9)             cuenta = cuenta + 1;
(10)            V(y-sem);           (sale del wait por (6))
(11)            P(próximo);         (13) lo libera
(14)            cuenta = cuenta - 1;
            end
```