

Contents

1	Semaforos	2
2	Problemas tipicos con Semaforos	2
2.1	Productor consumidor - buffer infinito	2
2.2	Productor consumidor - buffer finito	3
2.3	Lectores escritores	4
2.3.1	SOLUCION 1: Prioridad a Lectores	4
2.3.2	SOLUCION 1a: Usando Lightswich	5
2.3.3	SOLUCION 2: Agregar un molinete (turnstile).	6
2.3.4	SOLUCION 3: Mas prioridad a los escritores	7
2.4	Filosofos comenzales	8
2.4.1	SOLUCIÓN 1	8
2.4.2	SOLUCIÓN 2	9
2.4.3	SOLUCIÓN 3: SOLUCION ASIMETRICA	9
3	Monitores	9
4	Problemas tipicos con monitores	9

1 Semaforos

2 Problemas tipicos con Semaforos

2.1 Productor consumidor - buffer infinito

```
1
2 B: array(0..infinity) of Integer
3 In_Ptr, Out_Ptr: Integer := 0
4 Elements: Semaphore := 0
5
6
7 task body Producer is
8     I: Integer
9 begin
10     loop
11         Produce(I)
12         B[In_Ptr] := I
13         In_Ptr := In_Ptr + 1
14         Signal(Elements)
15     end loop
16 end
17
18 task body Consumer is
19     I: Integer
20 begin
21     loop
22         wait(Elements)
23         I = B[Out_Ptr]
24         Out_Ptr := Out_Ptr + 1
25         Consume(I)
26     end loop
27 end
```

2.2 Productor consumidor - buffer finito

Invariantes:

- No se puede consumir lo que no hay.
- Todos los item producidos son eventualmente consumidos.
- Al espacio de almacenamiento se accede de a uno.
- Se debe respetar el orden de almacenamiento y retiro de los elementos.

El buffer sirve para comunicar dos componentes que colocan y sacan a velocidades parecidas. Hay dos problemas de sincronismo.

1. No se puede consumir si el buffer esta vacio
2. No se puede producir si el buffer esta lleno.

```
1 B: array[0..N] of Integer
2 In_Ptr, Out_Ptr : Integer := 0
3 Elements: Semaphore := 0
4 Spaces: Semaphore := N
5
6 task body Producer is
7 begin
8   loop
9     wait(Spaces)
10    produce(I)
11    B[In_Ptr] := I
12    In_Ptr := (In_Ptr + 1) mod N
13    signal(Elements)
14  end loop
15 end
16
17 task body Consumer is
18 begin
19   loop
20     wait(Elements)
21     I = B[Out_Ptr]
22     Out_Ptr = (Out_Ptr + 1) mod N
23     signal(Spaces)
24     consume(I)
25   end loop
26 end
```

La lectura es destructiva y La soluciones es simetrica, ambos tienen el mismo código.

2.3 Lectores escritores

Para situaciones donde las estructuras de datos, base de datos o sistema de archivos son leídos y escritos por hilos concurrentes.

La solución es asimétrica. El escritor y el productor ejecutan códigos distintos antes de entrar en la sección crítica.

Las restricciones de sincronización son:

- Readers : Es proceso no requiere la exclusión de otro proceso Readers.
- Writers : Proceso que requiere la exclusión de otros procesos Reader y Writers

El patrón de exclusión se llama Categorical Mutual Exclusion. Un hilo en la sección crítica no necesita excluir otros hilos pero en la presencia de una categoría en la sección crítica excluye otras categorías.

2.3.1 SOLUCION 1: Prioridad a Lectores

```
1
2 reader: Integer := 0  -- Cuenta cuantos lectores hay en el cuarto
3 mutex: semaforo := 1  -- Protege el contador.
4 roomEmpty: semaforo := 1 -- Vale 1 sin no hay escritores y ni lector en la SC
5
6
7 task body Writer
8 begin
9     wait(roomEmpty)
10    seccion_critica
11    signal(roomEmpty)
12 end
13
14 task body Reader
15 begin
16
17    wait(mutex)
18    reader := reader + 1
19    if reader = 1 then wait(roomEmpty)
20    signal(mutex)
21
22    seccion_critica
23
24    wait(mutex)
25    reader := reader - 1
26    if reader = 0 then signal(roomEmpty) -- Si es el ultimo reader, señalo room empty
27    signal(mutex)
28 end
```

2.3.2 SOLUCION 1a: Usando Lightswich

Se utiliza un patron llamado Lightswich. El primero que entra prende la luz, el último que sale lo apaga. Entonces se arma un pre y pos protocolo para simplificar el reader.

```
1 mutex: semaforo := 1 -- Protege el contador.
2 reader: Integer := 0 -- Cuenta cuantos lectores hay en el cuarto
3
4 procedure LockIfFirst(s: Semaphore, count: Integer)
5 begin
6     wait(mutex)
7     if (count = 1) then wait(s) end if -- Si es el primero toma el semaforo
8     signal(mutex)
9 end
10
11 procedure UnlockIfLast(s: Semaphore, count: Integer)
12 wait(mutex)
13 if (count = 0) then signal(s) end if --Si es el ultimo libera el semaforo
14 signal(mutex)
15 begin
16 end
17 -- Entonces se simplifica el Lector de esta manera:
18 reader: Integer := 0 -- Cuenta cuantos lectores hay en el cuarto
19 mutex: semaforo := 1 -- Protege el contador.
20 roomEmpty: semaforo := 1 -- Vale 1 sin no hay escritores y ni lector en la SC
21
22 task body Reader
23 begin
24     LockIfFirst(roomEmpty, reader)
25     seccion_critica
26     UnlockIfLast(roomEmpty, reader)
27 end
```

PROBLEMA: Es posible que el writer tenga stavation. Si un escritor llega mientras hay escritores en la sección crítica, este puede esperar en la cola por siempre mientras que los lector continuan llegando. Mientras llegan lectores ante del último de los lectores, siempre habra un lector en el Room. Hasta que los lectores no llegen a CERO, no pueden dale paso a los escritores.

2.3.3 SOLUCION 2: Agregar un molinete (turnstile).

```
1 roomEmpty: Semaphore := 1
2 readers: Integer := 0 -- Cuenta cuantos lectores estan en la sala
3 turnstile: Semaphore := 1 -- es un molinete para los reader y un mutex para los writers
4
5 task body Writer
6 begin
7     wait(turnstile) -- Si lo toma el escrito, fuerza a los lectores que se encolen en el turnline
8     wait(roomEmpty) -- Si hay lectores en la sala, el escritor se bloquea aca! lo que
9     # seccion_critica
10    signal(turnstile) -- libera el molinete y desbloquea los lectores encolados o otros writes
11    signal(roomEmpty)
12 end
13
14 task body Reader
15 begin
16     wait(turnstile)
17     signal(turnstile)
18
19     LockIfFirst(roomEmpty, reader)
20     # seccion_critica
21     UnlockIfLast(roomEmpty, reader) -- Cuando el ultimo reader deje la sala, se garantiza que al
22     menos un escritor este en la sala
23 end
```

Esta solución garantiza que al menos un escritor proceda. En esta solución no hay prioridad entre escritores y lectores. Hay Justicia!. Pero aun es posible que lectores entran mientras haya mas escritores encolados.

A veces es preferible una solución que da mas prioridad a los escritores! Y esto depende del contexto. Por ejemplo: Si los writes tienen tiempos criticos de updates, es mejor minimizar las lecturas de datos viejos y priorizar la actualización.

2.3.4 SOLUCION 3: Mas prioridad a los escritores

Mientras entran escritores, ningún lector debe entrar hasta que se haya ido el último escritor.

```
1 writers: Integer := 0 -- Nuevo contador
2 readers: Integer := 0
3 noWrites: semaforo := 1
4 noReader: semaforo := 1 -- Nuevo semaforo
5
6 task body Writer
7 begin
8     LockIfFirst(noReader, writers)
9
10    wait(noWrites)
11    # seccion_critica -- Garantiza que no hay otro escritor y lector al tomar noReaders y
    noWriters
12    -- Muchos escritores pueden pasar
13    signal(noWrites)
14
15    UnlockIfLast(noReader, writers)
16 end
17
18 task body Reader
19 begin
20    wait(noReader)
21    LockIfFirst(noWrites, readers)
22    signal(noReader)
23
24    # seccion_critica -- Si el lector esta en la seccion critica, mantiene tomado noWriter para
    encolar a los escritores
25
26    UnlockIfLast(noWrites, readers) -- Si termina el ultimo escritor, se desbloquea noWrites
    para que pasen mas escritores.
27 end
```

2.4 Filósofos comenzales

- Acciones Pensar y comer
- 1 Mesa con 5 platos, 5 tenedores

Sincronización:

- Cada filósofo puede tomar un tenedor de la izquierda y de la derecha, pero solo uno a la vez.
- Un filósofo solo puede comer si tiene dos tenedores.

Restricciones:

- Un filósofo solo puede comer si tiene dos tenedores.
- Dos filósofos no pueden tener el mismo tenedor simultaneamente
- No hay deadlock
- No hay individual stavation
- Comportamiento eficiente bajo ausencia de contención.

2.4.1 SOLUCIÓN 1

```
1 Fork: array(0...4) of Semaphore := (other => 1)
2 task body Filosofo is
3 begin
4   loop
5     think
6     wait(Fork[i])  -- Toma el tenedor de la izquierda
7     wait(Fork[i+1 mod 5]) -- Toma el tenedor de la Derecha
8     eat
9     signal(Fork[i])
10    signal(Fork[i+1 mod 5])
11  end loop
12 end
```

Presenta deadlock porque los 5 filosofos pueden tomar el tener de la izq al mismo tiempo.

2.4.2 SOLUCIÓN 2

```
1 Fork: array(0...4) of Semaphore := (other => 1)
2 Room: Semaphore := 4
3 task body Filosofo is
4 begin
5   loop
6     think
7     wait(room) -- Limitamos la cantidad de filosofos en un cuarto para 4. Evitamos DEADLOCK al
      no tomar los 5 al mismo tiempo.
8     wait(Fork[i])
9     wait(Fork[i+1 mod 5])
10    eat
11    signal(Fork[i])
12    signal(Fork[i+1 mod 5])
13    signal(room)
14  end loop
15 end
```

Se asumió que el semaforo Room es de tipo Cola-Bloqueante, entonces cada filosofo entrar al cuarto eventualmente. Con esto nos aseguramos que haya justicia. El resto de los semaforos, pueden ser block-set.

2.4.3 SOLUCIÓN 3: SOLUCION ASIMETRICA

Los primeros 4 filosofos ejecutan la solución 1, pero el 5to filosofo (cualquier), toma primero el de la derecha y luego el de la izquierda.

```
1 task body filosofo is
2 begin
3   think
4   wait(Fork[0]) -- Tenedor de la derecha. Con I = 4 -> 4 + 1 mod 5 = 0
5   wait(Fork[4]) -- Tenedor de la izquierda
6   eat
7   signal(Fork[4])
8   signal(Fork[0])
9 end
```

De esta manera nos evitamos entrar en deadlock o starvation

3 Monitores

4 Problemas tipicos con monitores