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
2.5 Panadero 9
2.5.1 VERSION SIMPLIFICADA: para dos procesos 9
2.5.2 VERSION N PROCESOS 10

3 Monitores **10**

4 Problemas tipicos con monitores **10**

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         Int_Ptr := Int_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.

```
B: array[0..N] of Integer
In_Ptr, Out_Ptr : Integer := 0
Elements: Semaphore := 0
Spaces: Semaphore := N

task body Producer is
begin
    loop
        wait(Spaces)
        produce(I)
        B[In_Ptr] := I
        In_Ptr := (In_Ptr + 1) mod N
        signal(Elements)
    end loop
end

task body Consumer is
begin
    loop
        wait(Elements)
        I = B[Out_Ptr]
        Out_Ptr = (Out_Ptr + 1) mod N
        signal(Spaces)
        consume(I)
    end loop
end
```

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

2.3 Lectores escritores

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

La solución es asimetrica. El escritor y el productor ejecutan códigos distintos antes de entrar en la seccion critica.

Las restricciones de sincronizacion son:

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

El patron de exclución se llama Categorical Mutual Exclution. Un hilo en la sección critica no necesita excluir otro hilos pero en la presencia de una categoria en la seccion critica excluye otras categorias.

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(roomEmpy)
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.

```
mutex: semaforo := 1  -- Protege el contador.
reader: Integer := 0  -- Cuenta cuantos lectores hay en el cuarto

procedure LockIfFirst(s: Semaphore, count: Integer)
begin
    wait(mutex)
    if (count = 1) then wait(s) end if  -- Si es el primero toma el semaforo
    signal(mutex)
end

procedure UnlockIfLast(s: Semaphore, count: Integer)
    wait(mutex)
    if (count = 0) then signal(s) end if --Si es el ultimo libera el semaforo
    signal(mutex)
begin
end
-- Entonces se simplifica el Lector de esta manera:
reader: Integer := 0  -- Cuenta cuantos lectores hay en el cuarto
mutex: semaforo := 1  -- Protege el contador.
roomEmpty: semaforo := 1 -- Vale 1 sin no hay escritores y ni lector en la SC

task body Reader
begin
    LockIfFirst(roomEmpty, reader)
    seccion_critica
    UnlockIfLast(roomEmpty, reader)
end
```

PROBLEMA: Es posible que el writer tenga stavation. Si un escritor llega mientras hay escritores en la sección critica, este puede esperar en la cola por siempre mientras que los lector continuan llegando. Mientras llegen 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).

```
roomEmpty: Semaphore := 1
readers:Integer := 0 -- Cuenta cuantos lectores estan en la sala
turnstile: Semaphore := 1 -- es un molineta para los reader y un mutex para los writers

task body Writer
begin
    wait(turnstile)   -- Si lo toma el escrito, fuerza a los lectores que se encolen en el turnline
        wait(roomEmpty) -- Si hay lectores en la sala, el escritor se bloquea aca!  lo que significa que esta bloqueado el molineta
        # seccion_critica
    signal(turnstile) -- libera el molineta y desbloquea los lectores encolados o otros writes
    signal(roomEmpy)
end

task body Reader
begin
    wait(turnstile)
    signal(turnstile)

    LockIfFirst(roomEmpty, reader)
    # seccion_critica
    UnlockIfLast(roomEmpty, reader) -- Cuando el ultimo reader deje la sala, se garantiza que al menos un escritor este en la sala
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.

```
writers: Integer := 0 -- Nuevo contador
readers: Integer := 0
noWrites: semaforo := 1
noReader: semaforo := 1  -- Nuevo semaforo

task body Writer
begin
    LockIfFirst(noReader, writers)

        wait(noWrites)
        # seccion_critica -- Garantiza que no hay otro escritor y lector al tomar noReaders y noWriters
                          -- Muchos escritores pueden pasar
        signal(noWrites)

    UnlockIfLast(noReader, writers)
end

task body Reader
begin
    wait(noReader)
        LockIfFirst(noWrites, readers)
    signal(noReader)

    # seccion_critica -- Si el lector esta en la seccion critica, mantiene tomado noWriter para encolar a los escritores

    UnlockIfLast(noWrites, readers) -- Si termina el ultimo escritor, se desbloquea noWrites para que pasen mas escritores.
end
```

2.4 Filosofos comenzales

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

Sincronización:

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

Restricciones:

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

2.4.1 SOLUCIÓN 1

```
Fork: array(0...4) of Semaphore := (other => 1)
task body Filosofo is
begin
    loop
        think
        wait(Fork[i])   -- Toma el tenedor de la izquirda
        wait(Fork[i+1 mod 5]) -- Toma el tenedor de la Derecha
        eat
        signal(Fork[i])
        signal(Fork[i+1 mod 5])
    end loop
end
```

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

2.4.2 SOLUCIÓN 2

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

Se asumio 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.

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

De esta manera nos evitamos entrar en deadlock o starvation

2.5 Panadero

Es un algoritmo de exclución mutua para N procesos.

Un proceso que desea entrar en la seccion critica, se requiere que tome un numero de ticket el cual sera el valor mas grande de todos lo que hayan tomado un ticket previamente. Situacion similar a la de los clientes entran a una panaderia y toman un ticket para ser atendidos.

2.5.1 VERSION SIMPLIFICADA: para dos procesos

```
np:integer := 0
nq:integer := 0

task body P is
begin
    loop
        seccion no-critica
        np := nq + 1
        await nq = 0 or np <= nq
        seccion critica
        np := 0
    end loop
end
task body Q is
begin
    loop
        seccion no-critica
        nq := np + 1
        await np = 0 or nq < np
        seccion critica
        nq := 0
    end loop
end
```

2.5.2 VERSION N PROCESOS

```

number: integer Array := [0,0,...,0]
task body P is
begin
    loop
        seccion no-critica
        number[i] := 1 + max(number)    -- cada proceso toma el valor maximo
        for all other process j
            await (number[j] = 0) or number[i] << number[j]
        seccion critica
        number[i] := 0
    end loop
end
```

Conclución, no es práctico por dos razones.

1. El número de ticket no tiene limite si algún proceso siempre esta en la zona critica.
2. Cada proceso tiene que consultar el número de ticket del resto de los procesos.

3 Monitores

4 Problemas tipicos con monitores