TRABAJO PRACTICO 4

EJERCICIO 4.1

PROCESOS COOPERATIVOS

Pueden acceder directamente a espacios de memoria compartida o intercambiar información a través del envío de mensajes entre ellos. El acceso concurrente a datos compartidos, como lo es una sección de memoria compartida, puede llevar a inconsistencias. Es por esto que necesitamos una forma de sincronizar este acceso para evitar este problema.

CONDICION DE CARRERA

Situación en la que el resultado de una ejecución de procesos que acceden y modifican los mismos datos concurrentemente depende del orden de esos accesos. Para evitar esto, tenemos que sincronizar los accesos para asegurar que solo un proceso pueda modificar los datos compartidos a la vez.

EJERCICIO 4.2

En el caso de depósito y retiro, se puede producir un problema de sincronización si se intentan realizar las dos acciones en simultaneo.
El usuario A quiere hacer un depósito. El B un retiro. El usuario A comienza la transacción y lee su saldo que es 1000. En ese momento pierde su turno de ejecución (y su saldo queda como 1000) y el usuario B inicia el retiro: lee el saldo que es 1000, retira 200 y almacena el nuevo saldo que es 800 y termina. El turno de ejecución regresa al usuario A el cual hace su depósito de 100, quedando saldo = saldo + 100 = 1000 + 100 = 1100. Como se ve, el retiro se perdió y eso le encanta al usuario A y B, pero al banquero no le convino esta transacción. El error pudo ser al revés, quedando el saldo final en 800.
Una forma de evitar este problema, sería que el sistema bloquee cualquier transacción mientras se está procesando otra, de esta forma, el usuario B no podría hacer la consulta del saldo hasta que la transacción del usuario A haya terminado.

EJERCICIO 4.3

EJERCICIO 4.4

Para que una solución a este problema sea válida, se deben satisfacer las siguientes condiciones:

- **Exclusión Mutua:** si un proceso P está en su sección crítica, ningún otro proceso puede estarlo.
- **Progreso:** si hay proceso que quieren ingresar a la sección crítica, y no hay ningún otro proceso en la sección crítica; entonces la selección del próximo proceso a ingresar en dicha sección no puede ser pospuesta indefinidamente.
- **Espera limitada:** debe existir un límite de veces en las que un proceso puede acceder a su sección crítica después que otro proceso que requirió entrar en su sección crítica no pudo.

EJERCICIO 4.5

EJERCICIO 4.6

Busy Waiting refiere a la situación en que mientras un proceso está en su sección crítica, el resto debe ciclar continuamente (spinlocks) hasta que dicho proceso salga de su sección crítica y alguno de estos pueda entrar.
…

EJERCICIO 4.7

El uso de spinlocks en un sistema de un solo procesador es un desperdicio de CPU que otros procesos podrían usar de una mejor forma. Sin embargo, son útiles cuando la espera es corta y debemos tener en cuenta que no requiere ningún cambio de contexto. Son más utilizados en sistemas multiprocesador, donde un thread cicla en un procesador y otro thread hace su trabajo sobre la sección crítica en otro procesador.

EJERCICIO 4.8

EJERCICIO 4.9

- A B C D E
- A B D C E
- A B D E C
- A D B C E
- A D B E C
- A D E B C
- …

EJERCICIO 4.10

Se puede modelar mediante semáforos una solución al problema de la sección crítica.

EJERCICIO 4.11

La verificación y modificación del valor, así como la posibilidad de irse a dormir (bloquearse) se realiza en conjunto, como una sola e indivisible acción atómica. El sistema operativo garantiza que al iniciar una operación con un semáforo, ningún otro proceso puede tener acceso al semáforo hasta que la operación termine o se bloquee. Esta atomicidad es absolutamente esencial para resolver los problemas de sincronización y evitar condiciones de competencia.
Suponiendo que la operación wait no se ejecutara atómicamente y esté construida por 3 instrucciones, y suponiendo además que la verificación se hace en la primer instrucciones (C > 0) y el bloqueo se hace en la última (C = C + 1) podría darse el caso que si existen dos procesos en espera (wait), se le conceda el permiso a ambos.

EJERCICIO 4.12

Los semáforos permiten gestionar N cantidad de recursos mediante la inicialización (init) con N.
De esta forma, un proceso que intente utilizar el recurso, solo tendrá acceso si el semáforo es mayor que 0. Si accede al recurso, se decrementa el valor del semáforo. Al liberar el recurso, incrementa el valor del semáforo, permitiendo que otro proceso acceda al recurso.

EJERCICIO 4.13

A)

P1	P2
Print(i1) S2.signal() S1.wait() Print(i2)	Print(j1) S1.signal() S2.wait() Print(j2)

B)

No

C)

P1	P2	P3

S1 = 1
While true
  Wait(S1)
  Do..
  Signal(S3)
End
	While true
	  Wait(S2)
	  Do..
	  Signal(S1)
	End
		While true
		  Wait(S3)
		  Do..
		  Signal(S2)
		End

D)

P1	P2	P3	P4
S1 = 1	While true	While true	While true

While true Wait(S1) Do.. Signal(S2) Signal(S3) End	Wait(S2) Do.. Signal(S4) End	Wait(S3) Do.. Signal(S4) End	Wait(S4) Wait(S4) Do.. Signal(S1) End

E)

```
P1..P4
While(true)
        Whait(mutex)
        Do..
        Signal(mutex)
End
```

EJERCICIO 4.14

```
Init (B,5)
Init (P,3)
Init (M,10)

While(true)
        Wait(B)
                Do Sentadillas
        Signal(B)
        Wait(P)
                Do Pull over
        Signal(P)
        Wait(M)
                Do Estocadas
        Signal(M)
End
```

EJERCICIO 4.15

```
Init (B,5)
Init (P,3)
Init (T,1)

While(true)
        Wait(B)
                Do Sentadillas
        Signal(B)
        Wait(P)
                Do Pull over
        Signal(P)
        Wait(M)
                Do Estocadas
        Signal(M)
        Wait(T)
                Do Dejar Toalla
        Signal(T)
End
```

EJERCICIO 4.16

A)

Mutex se inicializa en 1, para que en un comienzo esté liberada la zona crítica

B)

Empty se inicializa en 1 para que el proceso productor pueda acceder a la zona crítica y producir un producto

C)

Full se inicializa en 0 para que el proceso consumidor no intente consumir hasta que no se haya producido un producto

D)

Si empty = 0, entrará primero el proceso consumidor. Si empty = 1, entrarán los dos “al mismo tiempo”

E)

Full = N
Empty = 0
El tercer proceso de productor no se alcanza a ejecutar

EJERCICIO 4.17

A)

Variables compartidas
    Semáforos: mutex, wrt
    Entero: readcount
Init Mutex = 1
Init wrt = 1
Readcount = 0

Proceso Escritor:
    Mientras(true)
        Wait(wrt)
            Do Write
        Signal(wrt)

Proceso Lector:
    Mientras(true)
        Wait(mutex)
            Readcount ++
            Si readcount = 1 wait(wrt)
        Signal(mutex)
        Do Read
        Wait(mutex)
            Readcount --
            Si readcount = 0 signal(wrt)
        Signal(mutex)

B)

Variables compartidas
    Semáforos: rsem, wsem,x,y,z
    Entero: readcount,writercount
Init rsem,wsem,x,y,z = 1
Readcount = 0

```
Writercount = 0
Proceso Escritor:
    Mientras(true)
        Wait(y)
            Writercount ++
            If writecount == 1 wait(rsem)
        Signal(y)
        Wait(wsem)
            Do Write
        Signal(wsem)
        Wait(y)
            Writercount –
            If writercount = 0 signal(rsem)
        Signal(y)

Proceso Lector:
    Mientras(true)
        Wait(z)
            Wait(rsem)
                Wait(x)
                    Readcount ++
                    Si readcount = 1 wait(wsem)
                Signal(x)
            Signal(rsem)
        Signal(z)
        Do Read
        Wait(x)
            Readcount --
            Si readcount = 0 signal(wsem)
        Signal(x)
```

C)

En ambos casos, uno de los procesos debe esperar que el recurso sea liberado por el otro proceso, provocando una espera indefinida. Un tercer problema podría resolver esto otorgando el recurso por orden de solicitud.

EJERCICIO 4.18

Se utilizará la funcionalidad de monitor para resolver este problema

```
Type dining-philosophers = monitor
    Var state: array[0..4] of (thinking, hungry, eating)
    Var self: array[0..4] of condition
    Procedure entry pickup(i:0..4)
        State[i] := hungry
        Test(i)
        If state[i] != eating then self[i].wait

    Procedure entry putdown(i:0..4)
        State[i] := thinking
        Test(i+4 mod 5)
        Test(i+1 mod 5)

    Procedure test (k: 0..4)
        If (state[k+4 mod 5] != eating and state[k] = hungry and state[k+1 mod 5] != eating
            State[k] := eating
            Self[k].signal
```

Cada proceso de filósofo (i) quedara ciclando indefinidamente en el siguiente proceso
Db.pickup(i)
    Do comer..
Db.putdown(i)

B)

Este problema demuestra la incapacidad de resolver ciertos tipos de problemas de sincronización utilizando simples semáforos evitando que ocurran deadlocks.

EJERCICIO 4.19

EJERCICIO 4.20

En la utilización de semáforos pueden existir errores de temporización debido a su incorrecto uso y a que son difíciles de detectas. De aquí nace la idea de monitores, que proporcionan un mecanismo alternativo al de los semáforos.
Desventajas de utilizar semáforos:

- El uso y la función de las variables no es explicito
- Están basados en variables globales -> no es modular
- Las operaciones sobre las variables dispersas y no protegidas
- Acceso no estructurado ni encapsulación -> Funte de errores