Sistema Operativos Practica 3

Ejercicio 1

A continuación se muestran dos códigos de procesos que son ejecutados concurrentemente. La variable X es compartida y se inicializa en 0.

a) X global inicia en 0

```
void A()
    X = X + 1;
    printf("%d", X);

void B()
    X = X + 1;
```

La salida no es unica. Si A termina sin que corra B, imprime 1. Si B hace el incremento antes del print de A, imprime 2.

b) Las variables X e Y son compartidas y se inicializan en 0.

```
void A()
    for (; X < 4; X++) {
    Y = 0;
    printf("%d", X);
    Y = 1;
    }

void B()
    while (X < 4) {
        if (Y == 1)
        printf("a");
    }
```

La salida no es unica, lo que si siempre empieza imprimiendo 0 porque sino Y=0 asi que B no imprime nada. Despues como se puede ejecutar en cualquier orden, siempre que y=1 B puede imprimir a. El output general seria:

0 a . . . a 1 a . . . a 2 a . . . a 3 a. . . a

donde a . . . a puede ser 0 a'es hasta infinitas.

Ejercicio 2

Se tiene un sistema con cuatro procesos accediendo a una variable compartida x y un mutex, el siguiente código lo ejecutan los cuatro procesos. Del valor de la variable dependen ciertas decisiones que toma cada proceso. Se debe asegurar que cada vez que un proceso lee de la variable compartida, previamente solicita el mutex y luego lo libera. ¿Estos procesos cumplan con lo planteado? ¿Pueden ser víctimas de race condition?

```
x = 0; // Variable compartida
mutex(1); // Mutex compartido
while (1) {
    mutex.wait();
    y = x; // Lectura de x
    mutex.signal();
    if (y <= 5) {
        x++;
    } else {
        x--;
    }
}
```

No cumple lo planteado, cuando hacemos x++ tambien estamos leyendo la varible. Primero la lee, despues la aumenta y por ultimo la guarda en memoria.

Puede haber race condition: Si un proceso hace x++ y antes de guardar en memoria lo desalojan, si vuelve a ejecturse despues de varias escrituras en x va a guardar un valor viejo de x a la memoria.

La solucion seria meter todo enla zona critica del mutex.

Ejercicio 3

La operación wait() sobre semáforos suele utilizar una cola para almacenar los pedidos que se encuentran en espera. Si en lugar de una cola utilizara una pila (LIFO), determinar si habría inanición o funcionaría correctamente.

Propongo un programa para analizar starvation:

```
mutex(1) //variable compartida de todos los procesos

void programm(){
    while(1){
        mutex.wait();
        //procesamiento
        mutex.signal();
    }
}
```

Supongamos que tenemos 3 procesos P1, P2, P3 que corren programm y entran en orden al wait inciialmente. P1 va a empezar a ejecutar y despues el resto va a correr sus wait metiendose en la LIFO, donde va a estar primero P3. Si P1 llega a hacer su wait antes que lo desalojen (es muy probable es hacer el jmp del final del loop al wait) se va a meter primero en la cola y despues P3 va a empezar a ejecutar. Lo mismo pasa con P3 y asi se pueden estar pasando el mutex entre P1 y P3. Asi que si, es una posibilidad (bastante alta) que haya starvation, la unica posibilidad de que se ejecute P2 es que el scheduler desaloje a alguno antes de que llegue a hacer su wait. Este problema no lo tenemos si usamos una cola FIFO

Ejercicio 4

Demostrar que, en caso de que las operaciones de semáforos wait() y signal() no se ejecuten atómicamente, entonces se viola la propiedad de exclusión mutua, es decir que un recurso no puede estar asignado a más de un proceso. **Pista**: Revise el funcionamiento interno del wait() y del signal() mostrados en clase, el cual no se haría de forma atómica, y luego piense en una traza que muestre lo propuesto.

```
wait(s){
    while(s<=0) dormir;
    s--;
}

signal(s){
    s++;
    if(alguien espera por s) despertar a alguno;
}
```

Imaginemos para un semaforo,S inicialmente en 0 la siguiente secuencia:

1. Un P1 envia un signal, llega a terminar el s++ y lo interrumpe el scheduler.
2. Un P2 retoma su ejecucion y llega a un wait(), en el wait no entra al while porque s=1. P2 decrementa s y se pone a ejecutar el recurso del semaforo.
3. En ese instante P1 se despierta y ejecuta el if que le quedo pendiente, despertando a un P3 que esperaba por el semaforo.

Asi el recurso del semaforo esta asignado a P2 y P3

Ejercicio 5

Se tienen n procesos: P1, P2, ... , Pn que ejecutan el siguiente código. Se espera que todos los procesos terminen de ejecutar la función preparado() antes de que alguno de ellos llame a la función critica(). ¿Por qué la siguiente solución permite inanición? Modificar el código para arreglarlo.

```
preparado()

mutex.wait()
count = count + 1
mutex.signal()

if (count == n)
    barrera.signal()

barrera.wait()

critica()
```

Hay dos problemas, el primero es que no sabemos cuantas veces se va a ejecutar el cuerpo del if. Es posible que el n-esimo proceso haga que count = n y otros procesos hayan sido desalojados arriba del if. Cuando esos retomen su ejecucion van a entrar al cuerpo del if. Este no es el comportamineto esperado, queremos un unico signal que "abra" la barrera. (se veia que iba a ser un problema porque se accede a un recurso compartido afuera del mutex).

El segundo problema es que una vez que un proceso pase la barrera esta va a quedar cerrada y ningun otro la va a poder pasar. Para solucionarlo hay que agregar un signal despues del wait, asi se genera el efecto cascada.

La barrera empieza en 0 y el mutex en 1.

```
preparado()

mutex.wait()
count = count + 1
if (count == n)
    barrera.signal()
mutex.signal()

barrera.wait()
barrera.signal()

critica()
```

Ejercicio 6

Cambie su solución del ejercicio anterior con una solución basada solamente en las herramientas atómicas vista en las clases, que se implementan a nivel de hardware, y responda las siguientes preguntas:

a. ¿Cuál de sus dos soluciones genera un código más legible?

b. ¿Cuál de ellas es más eficiente? ¿Por qué?

c. ¿Qué soporte requiere cada una de ellas del SO y del HW?

```
//varibale compartida
atomic<int> count;

preparado()
count = count.getAndInc();
while(count < n);
critica();
```

a)

Esta opcion es mucho mas legible, no me estoy ocupando explicitamente del manejo de semaforos.

b) cual es mas eficiente

La anterior pareceria ser mas eficiente, en esta solucion a mayor sea el n y dependiendo de la complejidad de la funcion preparado(), crece mucho el busy waiting. Vamos a gastar mucho procesador en ese while. En cambio usando semaforos los procesos esperan a que los despierten.

c)

El uso de variables atomicas requiere del hardware instrucciones atomicas para que no se puedan interrumpir.

Los semaforos requieren su implementacion del SO para poder llevar el registro y despertar a los que estan esperando y del hardware las instrucicones atomicas como wait y signal necesarias para implementarlos.

Ejercicio 7

Se tienen N procesos, P0,P1, . . . , PN-1 (donde N es un parámetro). Se requiere sincronizarlos de manera que la secuencia de ejecución sea Pi ,Pi+1, . . . , PN-1,P0, . . . , Pi-1 (donde i es otro parámetro). Escriba el código que deben ejecutar cada uno de los procesos para cumplir con la sincronización requerida utilizando semáforos (no olvidar los valores iniciales).

```
//codigo compartido i viene dado n tambien
semaforos[n]
for(j=0;j<n; j++){
    semaforos[j] = sem(0);
    //poner a correr el proceso j asumo que le puedo pasar su i
    correr(proceso,j);
}

semaforos[i].signal() //para que arranque el iesimo

proceso(int i){
    semaforos[i].wait();
    //hace lo que tenga que hacer...
    if(i==n){ //si es el ultimo que le mande al primero
        semaforos[0].signal();
    }
    else{
        semaforos[i+1].signal();
    }
}
```

Ejercicio 8

Considere cada uno de los siguientes enunciados, para cada caso, escriba el código que permita la ejecución de los procesos según la forma de sincronización planteada utilizando semáforos (no se olvide de los valores iniciales). Debe argumentar porqué cada solución evita la inanición:

1

Se tienen tres procesos (A, B y C). Se desea que el orden en que se ejecutan sea el orden alfabético, es decir que las secuencias normales deben ser: ABC, ABC, ABC, ...

```
//variables compartidas
semA = sem(1);
semB = sem(0);
semC = sem(0);

A(){
    while(1){
        semA.wait();
        tarea_A();
        semB.signal();
    }
}
B(){
    while(1){
        semB.wait();
        tarea_B();
        semC.signal();
    }

}
C(){
    while(1){
        semC.wait();
        tarea_C();
        semA.signal();
    }

}
```

Se ve que no hay inanicion, arranca desde el 1 porque el resto esta en 0. Despues se van abriendo los semaforos en orden.

2

Idem anterior, pero se desea que la secuencia normal sea: BBCA, BBCA, BBCA, ...

```
//variables compartidas
semA = sem(0);
semB = sem(2);
semC = sem(0);

A(){
    while(1){
        semA.wait();
        tarea_A();
        semB.signal();
        semB.signal();
    }
}
B(){
    int contador = 0;
    while(1){
        semB.wait();
        tarea_B();
        contador++;
        if(contador == 2){
            contador = 0;
            semC.signal();
        }
    }

}
C(){
    while(1){
        semC.wait();
        tarea_C();
        semA.signal();
    }

}
```

Por lo mismo que antes no tengo inanicion.

3

Se tienen un productor (A) y dos consumidores (B y C) que actúan no determinísticamente. La información provista por el productor debe ser retirada siempre 2 veces, es decir que las secuencias normales son: ABB, ABC, ACB o ACC. Nota: ¡Ojo con la exclusión mutua!

```
// compartidas
int contador = 0;
sp = sem(1);
sc = sem(0);
mutex = sem(1);

A(){
    while(1){
        sp.wait();
        producir();
        sc.signal();
        sc.signal();
    }
}

B(){
    while(1){
        sc.wait();
        consumir(); // si lo unico que hace es leer no hace falta que este en el mutex
        mutex.wait();
        contador++;
        if(contador==2){
            contador = 0;
            sp.signal();
        }
        mutex.signal();
    }
}

C(){
    while(1){
        sc.wait();
        consumir(); // si lo unico que hace es leer no hace falta que este en el mutex
        mutex.wait();
        contador++;
        if(contador==2){
            contador = 0;
            sp.signal();
        }
        mutex.signal();
    }
}
```

4

Se tienen un productor (A) y dos consumidores (B y C). Cuando C retira la información, la retira dos veces. Los receptores actúan en forma alternada. Secuencia normal: ABB, AC, ABB, AC, ABB, AC...

```
//compartidas
SP = sem(1);
SB = sem(0);
SC = sem(0);
i = 0;

A(){
    while(1){
        SP.wait();

        producir();

        if(i==0){ //le toca a B
            SB.signal();
            SB.signal();
        } else{ //va C
            SC.signal();
        }
    }
}

B(){
    int contador = 0;
    while(1){
        SB.wait();
        consumir();

        contador++;

        if(contador == 2){
            contador = 0;
            i = 1;
            SP.signal();
        }
    }
}

C(){
    SC.wait()
    consumir();
    i = 0;
```

```
    SP.signal();
}
```

Ejercicio 9

Suponer que se tienen N procesos Pi, cada uno de los cuales ejecuta un conjunto de sentencias ai y bi. ¿Cómo se pueden sincronizar estos procesos de manera tal que los bi se ejecuten después de que se hayan ejecutado todos los ai?

Uso la idea de "barrera" que vimos en clase y en el ejercicio 5 de la guia.

```
//variables compartidas (todos conocen n tambien)
int contador = 0;
mutex = sem(1);
barrera = sem(0);

void proceso(){
    ai(); //ejecutar conjunto de sentencias...

    mutex.wait(); //mutex para evitar race condition (el contador lo usa todo P)

    contador++;
    if(contador == n){
        barrera.signal();//ya terminaron los n, abre barrera
    }

    mutex.signal();

    barrera.wait();
    barrera.signal(); //efecto cascada

    bi(); // ahora ejecuta el  resto
}
```

Ejercicio 10

Se tienen los siguientes dos procesos, foo y bar, que son ejecutados concurrentemente. Además comparten los semáforos S y R, ambos inicializados en 1, y una variable global x, inicializada en 0.

```
void foo(){
    do{
        semWait(S);
        semWait(R);
        x++;
        semSignal(S);
        semSignal(R);
    } while (1);
}

void bar(){
    do {
        semWait(R);
        semWait(S);
        x--;
        semSignal(S);
        semSignal(R);
    } while (1);
}
```

a) ¿Puede alguna ejecución de estos procesos terminar en deadlock? En caso afirmativo, describir una traza de ejecución.

Si puede.

1. Empieza foo y ejecuta wait(S), el semaforo s pasa a 0.
2. Se desaloja a foo y se empieza a ejecutar bar.
3. Se ejecuta wait(r), el semaforo r pasa a 0.

Ahora foo se va a quedar esperando a r y bar a s, como ambos son 0 tenemos deadlock.

b) ¿Puede alguna ejecución de estos procesos generar inanición para alguno de los procesos? En caso afirmativo, describir una traza.

Si.

Por ejemplo con un scheduler sin deselojo el proceso foo nunca necesitaria dormirse, asi que nunca devolveria el scheduler (ya que el mismo se prende los semaforos que necesita para seguir). Foo tomaria control de la CPU lo que llevaria a la starvation de bar.

Ejercicio 11

Se quiere simular la comunicación mediante pipes entre dos procesos usando las syscalls read() y write(), pero usando memoria compartida (sin usar file descriptors). Se puede pensar al pipe como si fuese un buffer de tamaño N, donde en cada posición se le puede escribir un cierto mensaje. El read() debe ser bloqueante en caso que no haya ningún mensaje y si el buffer está lleno, el write() también debe ser bloqueante. No puede haber condiciones de carrera y se puede suponer que el buffer tiene los siguientes métodos: pop() (saca el mensaje y lo desencola), push() (agrega un mensaje al buffer).

```
//compartidas
buffer[n] //dado con push y pop
Sread = sem(0); //semaforo que indica cuantos elementos hay para leer
Swrite = sem(n); //semaforo que indica cuanto espacio queda en el buffer
mutex = sem(1);

write(m){
    Swrite.wait() //si no hay lugar para escribir es bloqueante

    mutex.wait(); //para evitar el race condition, no se puede escribir y leer a la vez
    buffer.push(m);
    mutex.signal();

    Sread.signal(); // indica que hay un elemento para leer en el buffer
}

read(){
    Sread.wait(); // Si no hay nada que leer es bloqueante

    mutex.wait();//para evitar el race condition, no se puede escribir y leer a la vez
    res = buffer.pop();
    mutex.signal();

    Swrite.signal(); //le aviso que hay un lugar mas para escribir
    return res
}
```