Ejercicios Semáforos 🚦

TEORÍA:

1. **Definición y Operaciones Fundamentales:**
   Un semáforo es una variable especial utilizada como mecanismo de sincronización para controlar el acceso de múltiples procesos a un recurso común en un entorno de programación concurrente. Funciona como un contador que señala la disponibilidad de un recurso.

Las dos operaciones atómicas fundamentales son:

- `semWait(s)` (o P): Decrementa el contador del semáforo. Si el resultado es negativo, el proceso que la ejecutó se bloquea y se pone en una cola de espera. Si es mayor o igual a cero, el proceso continúa.
- `semSignal(s)` (o V): Incrementa el contador del semáforo. Si había procesos esperando en la cola (es decir, si el contador era negativo antes de incrementar), uno de ellos es despertado y puesto en la lista de listos para ejecutarse.

2. **Semáforos Binarios vs. Contadores:**
   Diferencia principal: Un semáforo binario (o mutex) solo puede tomar los valores 0 y 1, usándose típicamente para garantizar la exclusión mutua (acceso a un solo recurso). Un semáforo contador puede tomar cualquier valor entero no negativo y se usa para controlar el acceso a un recurso con múltiples instancias (p. ej., n° impresoras) o para sincronizar eventos complejos.

   Cuándo usar cada uno:

- **Binario**: Cuando solo un proceso a la vez puede acceder a una sección crítica. Ejemplo: Proteger el acceso a una impresora única.
- **Contador**: Cuando se necesita contar recursos. Ejemplo: Gestionar un estacionamiento con 100 plazas disponibles.

3. **Inicialización de Semáforos:**
   Si hay 5 instancias de un recurso, el semáforo `S` debe inicializarse en 5. Esto permite que los primeros 5 procesos que hagan `semWait(S)` accedan al recurso sin bloquearse. Si un semáforo se inicializa en 0, el primer proceso que intente ejecutar `semWait` sobre él se bloqueará inmediatamente, ya que el contador pasará a -1. El proceso deberá esperar hasta que otro proceso ejecute `semSignal` sobre ese semáforo.

4. **Problema del Productor-Consumidor:**
   El propósito de cada semáforo es:

- `mutex`: Es un semáforo binario que garantiza la exclusión mutua para el acceso al buffer. Evita que el productor y el consumidor modifiquen el buffer al mismo tiempo.

- `Elementos`: Es un semáforo contador que representa el número de ítems disponibles en el buffer. El consumidor hace `wait` sobre él, bloqueándose si el buffer está vacío.
- `huecos`: Es un semáforo contador que representa el número de espacios vacíos en el buffer. El productor hace `wait` sobre él, bloqueándose si el buffer está lleno.

5. **Riesgo de Deadlock con Semáforos:**
   Sí, existe un claro riesgo de interbloqueo (deadlock) en esta situación.
   - Secuencia de Deadlock:
   1. P1 ejecuta `semWait(S1)` y adquiere el semáforo S1 (S1=0).
   2. El planificador cambia a P2.
   3. P2 ejecuta `semWait(S2)` y adquiere el semáforo S2 (S2=0).
   4. P2 intenta ejecutar `semWait(S1)` pero se bloquea, ya que S1 está en posesión de P1.
   5. El planificador vuelve a P1.
   6. P1 intenta ejecutar `semWait(S2)` pero se bloquea, ya que S2 está en posesión de P2. Ambos procesos quedan esperando indefinidamente por un recurso que posee el otro.

6. **Uso Incorrecto de Semáforos:**

   Olvidar `semSignal(mutex)`: Si un proceso adquiere el mutex con `semWait` y olvida liberarlo con `semSignal`, el semáforo quedará permanentemente en 0. Cualquier otro proceso que intente adquirir el mutex se bloqueará para siempre, inutilizando esa sección crítica.

   Invertir el orden (`semSignal` antes de `semWait`): Esto anula la exclusión mutua. Un proceso incrementaría el semáforo (por ejemplo, de 1 a 2) y luego lo decrementaría (de 2 a 1). El valor del semáforo nunca sería 0 para bloquear a otros procesos. Además, si varios procesos ejecutan `semSignal` antes de `semWait`, el valor del semáforo se "inflaría", permitiendo que múltiples procesos entren a la sección crítica simultáneamente.

7. **Sincronización de Tareas Específicas:**
   Para asegurar la secuencia A -> B -> C, se necesitan dos semáforos:

- `sem_t sem_A; // Inicializado en 0`
- `sem_t sem_B; // Inicializado en 0`

Proceso A	Proceso B	Proceso C

`// ..código de A..`
 	
`sem_post(&sem_A);`
	`sem_wait(&sem_A);`
	
	`// ..código de B..`
	
	`sem_post(&sem_B);`
		`sem_wait(&sem_B);`
		
		`// ..código de C..`

8. **Espera Activa vs. Bloqueo:**

Los semáforos evitan la espera activa al delegar la gestión de la espera al sistema operativo. Cuando un proceso ejecuta `semWait` y debe esperar, en lugar de gastar tiempo de CPU en un bucle (`while(condicion)`), el S.O lo cambia de estado a **"Bloqueado"** y lo saca de la cola de listos. No consumirá CPU hasta que otro proceso ejecute `semSignal` y el sistema operativo lo despierte. La ventaja principal es una **mayor eficiencia** del uso del procesador.

9. **Interpretación del Valor del Contador:**
   Si un semáforo contador S tiene un valor de 3, significa que hay 3 instancias del recurso disponibles. Los próximos 3 procesos que ejecuten `semWait(S)` podrán continuar sin bloquearse.

   Si `S.contador >= 0`, el valor indica el número de procesos que pueden ejecutar `semWait` sin bloquearse.

10. **Semáforos para Implementar Barreras:**
    Una idea para implementar una barrera para N procesos es usar un contador protegido por un mutex y un se2máforo que actúe como "puerta":
    - **Variables:** Un contador `llegadas` (inicializado en 0), un `mutex` (inicializado en 1) y un semáforo `puerta` (inicializado en 0).
    - **Lógica:** Cada proceso, al llegar a la barrera, adquiere el `mutex`, incrementa `llegadas` y libera el `mutex`.
    - El último proceso en llegar (cuando `llegadas == N`) ejecuta `sem_post(&puerta)` N veces (o una vez si se usa un patrón de paso en cascada) para liberar a todos los demás que están esperando.
    - Todos los procesos (incluido el último) ejecutan `sem_wait(&puerta)` para esperar en la barrera.

PRÁCTICA:

1. Considera tres procesos P1, P2, P3 y tres semáforos `Rojo`, `Verde`, `Azul`, todos inicializados en 1.

PROCESO 1	PROCESO 2	PROCESO 3
sem_wait(Rojo);	sem_wait(verde)	sem_wait(azul)
sem_wait(verde)	sem_wait(azul)	sem_wait(Rojo);
//Sección crítica	//Sección crítica	//Sección crítica
sem_post(verde)	sem_post(azul)	sem_post(rojo)
sem_post(rojo)	sem_post(verde)	sem_post(azul)

Sí, existe un **riesgo de interbloqueo**. La situación es idéntica a la del ejercicio teórico 5, pero con tres procesos.

- **Secuencia de Deadlock**:
    1. P1 ejecuta `sem_wait(Rojo)` y lo adquiere (Rojo=0).
    2. P2 ejecuta `sem_wait(Verde)` y lo adquiere (Verde=0).
    3. P3 ejecuta `sem_wait(Azul)` y lo adquiere (Azul=0).
    4. P1 intenta `sem_wait(Verde)` y se bloquea (espera a P2).
    5. P2 intenta `sem_wait(Azul)` y se bloquea (espera a P3).
    6. P3 intenta `sem_wait(Rojo)` y se bloquea (espera a P1). Se forma una espera circular (P1->P2->P3->P1).

2. Sí, es posible un **deadlock** entre P1, P2 y P3. La lógica es la misma que la del ejercicio anterior. P4, aunque necesita S1, no participa en el ciclo de espera, pero sí puede sufrir **inanición** si S1 es retenido indefinidamente por P1 o P3 debido al deadlock.

3. Suponga que hay 3 semáforos Binarios A, B y C, de los cuales A y B inicializan en 1 y C en 0.

Indicar una trayectoria para que: Todos los procesos terminen y por otro lado que haya un interbloqueo. (Diferenciar las 2 soluciones)

Proceso A	Proceso B	Proceso C
semWait(C)	semWait(A)	semWait(B)
…	…	…
semPost(A)	SemPost(B)	semPost(C)
…	…	…
Sem wait(C)	SemWai(A)	semWait(B)

Trayectoria para que todos terminen:

1. **Proceso B**: `semWait(A)` (A=0). B continúa.
2. **Proceso C**: `semWait(B)` (B=0). C continúa.
3. **Proceso B**: `semPost(B)` (B=1). B termina su primera parte.
4. **Proceso C**: `semPost(C)` (C=1). C termina su primera parte.
5. **Proceso A**: `semWait(C)` (C=0). A continúa.
6. **Proceso A**: `semPost(A)` (A=1). A termina.

Trayectoria para un interbloqueo:

1. **Proceso B**: `semWait(A)` (A=0).
2. **Proceso C**: `semWait(B)` (B=0).
3. **Proceso A**: `semWait(C)` (A se bloquea porque C=0).
4. **Proceso B**: Intenta su segundo `semWait(A)` y se bloquea (A=0).
5. **Proceso C**: Intenta su segundo `semWait(B)` y se bloquea (B=0). Todos los procesos están bloqueados esperando un semáforo que posee otro proceso que también está bloqueado (o esperando una señal que nunca llegará porque quien la envía está bloqueado).

4. Implementar la sincronización de los procesos A y B de tal manera que, siendo `x` una variable global:

int x;	
PROCESO A	PROCESO B

x= 10;
printf(“A1: %d\n”, x);
x = x * 2;
printf (“A2: %d\n”, x);
	x = x + 5;
	printf(“B: %d\n”, x);

- **a. Salida: B: 15, A1: 10, A2: 20** (B completo, luego A completo): Se usa un semáforo `sem_B_termino` inicializado en 0.
  - **Proceso A**: `sem_wait(&sem_B_termino);` al inicio.
  - **Proceso B**: `sem_post(&sem_B_termino);` al final.

- **b. Salida: A1: 10, B: 15, A2: 20** (Ejecución intercalada): Se usan dos semáforos `sem_A_listo` y `sem_B_listo`, ambos en 0.
  - **Proceso A**: Después de `printf("A1")` ejecuta `sem_post(&sem_A_listo)`. Antes de `x = x * 2` ejecuta `sem_wait(&sem_B_listo)`.
  - **Proceso B**: `sem_wait(&sem_A_listo)` al inicio y `sem_post(&sem_B_listo)` al final.

5. Semáforos `S1` y `S2` inicializados en 0, semáforo `S3` en 1:

Proceso 1	Proceso 2	Proceso 3
Wait(S1);	Wai(S2);	Wai(S3);
…	…	…
Post(S2);	Post(S1);	Post(S1);
…	…	…
…	…	Post(S2);

Planificación que permite terminar:

1. **P3** es el único que puede empezar. `Wait(S3)` (S3=0).
2. **P3**: `Post(S1)` (S1=1).
3. **P3**: `Post(S2)` (S2=1). P3 termina.
4. Ahora P1 y P2 pueden ejecutarse. P1: `Wait(S1)` (S1=0), `Post(S2)` (S2=2). P1 termina.
5. P2: `Wait(S2)` (S2=1), `Post(S1)` (S1=1). P2 termina. Todos terminan.

¿Es posible un interbloqueo? No, en este sistema **no es posible un interbloqueo**. El flujo está determinado por P3, que actúa como un "Arranque" para los otros dos procesos sin depender de ellos, rompiendo cualquier posibilidad de espera circular.

6. Considera un sistema con dos procesos, P1 y P2, que deben acceder a una sección crítica. La exclusión mutua se intenta implementar con un semáforo `mutex` inicializado en 1. Sin embargo, P2 tiene un error en su código.

P1	P2 (incorrecto)
semWait(mutex)	// codigo
//Seccion Critica	semPost(mutex)
semPost(mutex)	//Seccion Critica?

El error causa dos problemas principales:

1. **Error de Exclusión Mutua**: P2 nunca espera, por lo que puede entrar a la sección crítica mientras P1 está dentro, causando una condición de carrera.
2. **Aumento del Semáforo**: P2 incrementa el valor de `mutex` sin haberlo decrementado. Si el valor de `mutex` sube por encima de 1, permitirá que múltiples instancias de P1 entren a la sección crítica simultáneamente.

7. Para lograr la alternancia `PING, PONG, PING..`, se usan dos semáforos:

- `sem_t A`(inicializado en 1)
- `sem_t B`(inicializado en 0)

PROCESO A	PROCESO B

for (int i=0; i<5; i++) {
sem_wait(&A);
printf("PING\n");
sem_post(&B);
}
	for (int i=0; i<5; i++) {
	sem_wait(&B);
	printf("PONG\n");
	sem_post(&A);
	}

//Si queremos que se repita usamos por ejemplo un bucle for como el que puse.

8. Semáforo `mutex` inicializado en 1, semáforos `S1` y `S2` inicializados en 0.

PROCESO 1	PROCESO 2	PROCESO 3
`sem_wait(mutex);`	`sem_wait(S1);`	`sem_wait(S2);`
`sem_post(S1);`	`sem_post(mutex);`	`sem_post(mutex);`

El sistema tiene un error de diseño que causa un **bloqueo permanente**.

- **Secuencia**: P1 es el único que puede empezar (`sem_wait(mutex)`).
  - **P1**: Ejecuta y hace `sem_post(S1)`, permitiendo que P2 continúe.
  - **P2**: Ejecuta y hace `sem_post(mutex)`, permitiendo que P1 (si estuviera en un bucle) o P3 pudieran intentar tomarlo.

Problema: P3 espera en `sem_wait(S2)`, pero ningún proceso en el código ejecuta `sem_post(S2)`. Por lo tanto, **P3 se bloqueará para siempre** y nunca podrá ejecutarse.