Problemas de sincronización

Ejercicio 1

Se tiene un proceso productor `P` que hace `producir()` y dos consumidores `C1, C2` que hacen `consumir1()` y `consumir2()` respectivamente. Se desea sincronizarlos de manera tal que las secuencias de ejecución sean: `producir, producir, consumir1, consumir2, producir, producir, consumir1, consumir2,...`

Solución

```
permisoC2 = sem(0)
permisoC1 = sem(0)
permisoP = sem(1)

P():
  permisoP.wait()
  producir()
  producir()
  permisoC1.signal()

C1():
  permisoC1.wait()
  consumir1()
  permisoC2.signal()

C2():
  permisoC2.wait()
  consumir2()
  permisoP.signal()
```

Ejercicio 2

Se tienen 2 procesos A y B.
El proceso A tiene que ejecutar A1() y luego A2().
B debe ejecutar B1() y después B2().
En cualquier ejecución A1() tiene que ejecutarse antes que B2().
Construya el código con semáforos de manera tal que cualquier ejecución cumpla lo pedido.

Solución

```
permisoB = sem(0)

A():
  A1()
  permisoB.signal()
  A2()

B():
  B1()
  permisoB.wait()
  B2()
```

Ejercicio 3

Usando los procesos A y B de los ejercicios anteriores, ahora quiero que A1 y B1 ejecuten antes de B2 y A2.

Solución

```
permisoA = sem(0)
permisoB = sem(0)

A():
  A1()
  permisoB.signal()
  permisoA.wait()
  A2()

B():
  B1()
  permisoA.signal()
  permisoB.wait()
  B2()
```

Ejercicio 4

Un grupo de N estudiantes se dispone a hacer un tp de su materia favorita. Para ello acordaron dividirse el trabajo y cada quien conoce a la perfección como `implementarTp()` y luego como `experimentar()`. Curiosamente, cada etapa puede ser llevada acabo de manera independiente por cada estudiante. Sin embargo, acordaron que para que alguien pudiera `experimentar()` todos deberían haber terminado de `implementarTp()`.
Se pide diseñar un programa concurrente que utilice procesos y que modele esta situación utilizando semáforos.

Primera solución de la clase

```
variables compartidas:\\
n, mutex = sem(1), barrera = sem(0);

implementarTp()

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

if(n < N)
  barrera.wait()

barrera.signal()
experimentar()
```

Observaciones:
Al utilizar un semáforo como mutex logramos que el incremento de la variable compartida n no tenga pérdida de sumas. Esto también se pudo haber hecho utilizando un entero atómico usando la función `getAndInc()`.
La barrera comienza baja, sin permisos disponibles, pero el valor final es al menos uno. Cada proceso, eventualmente, suma un permiso con el `signal` y, en caso de que $N-1$ de ellos pasen por el `wait`, quedará un permiso pendiente por consumir en la barrera. Pero, de hecho, hay trazas donde podría terminar en N. Sin embargo esto no es algo que nos preocupe demasiado para este caso.
Un ejemplo de traza donde la barrera termina en N es aquella en donde cada proceso ejecuta hasta justo antes del *if*. Es decir, incrementan el valor de n sin que ninguno lea el *if*. Por lo tanto, una vez que todos hayan hecho esto, se verifica que $n == N$ haciendo que ninguno entre al *if* para hacer el `wait`. Finalmente, cada proceso hará únicamente un `signal` y se sumarán, eventualmente, N permisos a la barrera. Esto no rompe la restricción de sincronización ya que el primero en `experimentar()` lo hizo luego de que todos ya hayan pasado por `implementarTp()`.

Segunda solución de la clase

```
implementarTp()

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

if(n == N)
  repeat N: barrera.signal()

barrera.wait()
experimentar()
```

Observaciones:
En este caso hay al menos una traza donde la barrera termina en 0. Sin embargo, también es cierto que hay trazas donde termina en N^2. Una de ellas es la misma que en la solución anterior forzaba a que el valor final sea n.
Se puede utilizar la notación `barrera.signal(N)` de manera equivalente al repeat dentro del *if*. Sin embargo, es importante notar que si bien cada signal se comporta de manera atómica, por implementación de semáforos, el programa admite *interleavings* entre cada envío de las mismas. En cualquier caso, esta situación no compromete el correcto funcionamiento de nuestro programa.

¿Es reusable el patrón?

Supongamos que el grupo se da cuenta que es una pésima idea diagramar el esquema de todo un tp de esta forma tan rígida. Propone entonces un esquema **iterativo** ahora con dos etapas más cortas de implementación y experimentación. El acuerdo es que alguien puede empezar una nueva si todos terminaron la anterior ¿Podemos reutilizar la solución anterior?

Durante la clase vimos que una buena idea podría ser que la lectura del *if* se haga dentro del `mutex`, de forma de poder evitar las trazas en donde varios procesos creen que son los últimos. Este cambio en la primera solución generaría un problema, ya que el primer proceso se va a quedar esperando en la barrera con el `mutex` tomado, evitando que el resto de los procesos pueda progresar.
Por eso, propusimos reescribir la solución 1.1 de la siguiente manera:

```
implementarTp()

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

barrea.wait()
barrera.signal()

experimentar()
```

Como antes, los procesos se van despertando entre ellos luego de desbloquearse de la barrera. Ahora, con el *if* dentro del `mutex`, evitamos el problema de generar demasiados permisos de más. Sin embargo, como siempre se hacen $N + 1$ signals y solo N waits, la barrera siempre termina en 1. Esta garantía es más fuerte que la que teníamos y nos deja en una mejor posición para encarar la solución definitiva.

Solución de la clase

```
variables compartidas:
n, mutex = sem(1), barrera_entrada = sem(0), barrera_salida = sem(1)

while(true):

 implementarTp()

 mutex.wait()
 n = n + 1
 if n == N :
    barrera_salida.wait()
    barrera_entrada.signal()
 mutex.signal()

 barrera_entrada.wait()
 barrera_entrada.signal()

 experimentar()

 mutex.wait()
 n = n - 1
 if n == 0:
    barrera_entrada.wait()
    barrera_salida.signal()
 mutex.signal()

 barrera_salida.wait()
 barrera_salida.signal()
```

Observaciones:

No alcanza hacer solo el `signal` de la barrera correspondiente en cada *if*. Como vimos antes, la barrera siempre termina en 1 y esto, al considerar que los procesos loopean, puede generar situaciones indeseables. De hecho, supongamos que sacamos los `wait` dentro de cada *if* y que todos los procesos se encuentran en la segunda etapa haciendo `experimentar()` concurrentemente. A medida que van terminando se traban en el `wait` de `barrera_salida`. El último en hacerlo entra al *if* y libera un permiso en `barrera_salida`. Inmediatamente podría consumirlo, liberar otro permiso y loopear. Ahora, si bien está en todo su derecho de ejecutar `implementarTp()` ya que todos terminaron el experimentar previo, podría, luego de tomar y liberar el `mutex` para incrementar n, consumir el único permiso de `barrera_entrada` que quedó de la etapa anterior. Esto haría que pueda ejecutar la etapa de experimentar sin que el resto haya hecho `implementarTp`. Viola la restricción de sincronización que buscamos.

Notar que la `barrera_salida` se inicializa en 1, por lo tanto el último proceso en `implementarTp()` consume el permiso sin bloquearse pero baja, astutamente, la barrera para que el resto no pueda adelantarse y salir rápido de `experimentar()`. Además, una vez que todos hayan pasado por la `barrera_salida` el valor vuelve a restituirse a 1 como al principio.

Solución alternativa

```
variables compartidas:
n, mutex = sem(1), barrera_entrada = sem(0), barrera_salida = sem(0)
implementarTp()

mutex.wait()
n = n + 1
if n == N: barrera_entrada.signal(N)
mutex.signal()

barrera_entrada.wait()

experimentar()

mutex.wait()
n = n -1
if n == 0: barrera_salida.signal(N)
mutex.signal()

barrera_salida.wait()
```

Observaciones:

Esta solución alternativa surgió en algún momento pero no quedo escrita. Sería una forma de extender la solución 1.2 pero arreglando el problema de generar permisos de más protegiendo el *if* con el `mutex`. La forma de despertar a los procesos es ligeramente distinta ya que no forzamos la çadena de permisos"que teníamos en las anteriores. Por otro lado, nos aseguramos que, una vez que todos los procesos pasaron por una barrera, la misma queda baja, con cero permisos disponibles.

Ejercicio 5

Modelar con semáforos un micro de larga distancia directo entre Buenos Aires y Mendoza. El micro tiene asientos para N personas y funciona de la siguiente manera:

- Empieza en Buenos Aires.
- Espera a llenarse.
- Viaja hasta Mendoza.
- Estaciona en una terminal, permitiendo que los pasajeros desciendan.
- Repite el procedimiento desde el principio pero desde la otra terminal.

Notar que para subir al micro no importa el orden de llegada. Además el micro permite que pasajeros puedan subir y bajar al mismo tiempo.

Solución de la clase

```
variables compartidas:
permisoSubir  = [sem(0), sem(0)]
asientos = sem(N)
asientosOcupados = sem(0)
permisoBajar = sem(0)

bus(ciudad):
  while true:
    repeat N: permisoSubir[ciudad].signal()
    repeat N: asientosOcupados.wait()
    viajar(ciudad)
    repeat N: permisoBajar.signal()
    ciudad = 1 - ciudad


pasajero(i, ciudad):
  permisoSubir[ciudad].wait()
  asientos.wait()
  subir(i, ciudad)
  asientoOcupado.signal()
  permisoBajar.wait()
  bajar(i, ciudad)
  asiento.signal()
```

Observaciones:

En todo momento no hay mas de N personas en el bus. Cuando el bus llega a una ciudad permite que N pasajeros esperando en esa terminal puedan subir y estos irán subiendo a medida que los que bajan liberen sus asientos (concurrentemente).
El bus solo puede viajar si la cantidad de signals sobre asientosOcupados es N, y esto ocurre si todos lograron conseguir el permiso de asiento (`asiento.wait`). Cuando un pasajero que estaba en el bus baja, libera el recurso (`asiento.signal`) permitiendo que alguno de los que esperaba en la terminal contraria lo consiga. Como el bus da N `signals` sobre `permisoBajar`, todos los pasajeros tienen permitido bajar, eventualmente liberando los N asientos y solo esa cantidad de pasajeros de la terminal contraria tiene permitido subir.
Notar también que nunca se tiene en cuenta el orden de llegada de los pasajeros para permitir que suban (ejercicio copado, pensar como implementar esta variante).

Solución alternativa

```
variables compartidas:
permisoSubir  = [sem(0), sem(0)]
asientos = sem(N)
permisoBajar = sem(0)
mutex = sem(1)
permisoViajar = sem(0)
asientosOcupados = 0

bus(ciudad):
  while true:
    repeat N: permisoSubir[terminal].signal()
    pasajerosListos.wait()
    viajar()
    repeat N: permisoBajar.signal()
    terminal = 1 - terminal


pasajero(i):

  permisoSubir[ciudad].wait()
  asientos.wait()
  subir()
  mutex.wait()
  asientosOcupados++
  if (asientosOcupados == N)
    permisoViajar.signal()
  mutex.signal()
  permisoBajar.wait()
  bajar(i, ciudad)
  asiento.signal()
```

Observaciones:

Esta solución se asemeja a la idea de la solución 1.1 para el ejercicio 4, donde llevamos la cuenta de cuantos procesos pasaron por cierto estado y el último da el primero aviso que permite desbloquear al resto (con la diferencia que acá tenemos que coordinarnos con el bus). De forma similar, la solución 2.1 se asemeja a la idea 1.3.2.
Notar que `asientosOcupados` pudo haber sido un `atomic int` pero eso no nos garantiza que solo un proceso entre al *if*, en ese caso también es necesario el `mutex`.