Decisiones

Este ejercicio está en las diapositivas prácticas de sincronización entre procesos del segundo cuatrimestre de 2024. Lo elegí porque me parece simple y completo. El problema es fácil de entender. Los alumnos aprenden la diferencia entre *mutex* y *semaforo*, además de aprender el patrón/estrategia del "molinete". Me parece un buen ejercicio para introducir el modelado de problemas de sincronización. Es fundamental que los alumnos entiendan que, en este contexto, usamos el mismo código de proceso para modelar los diferentes casos o escenarios.

Enunciado

Un grupo de N estudiantes se dispone a hacer un TP de su materia favorita. Cada estudiante conoce a la perfección cómo implementarTp() y cómo experimentar(). Curiosamente, cada etapa puede ser llevada acabo de manera independiente por cada uno, así que decidieron dividirse el trabajo. 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.

Presentación

Primero, diría lo siguiente:

> Vamos a interpretar el enunciado por partes para establecer qué es lo que tenemos y qué problema queremos resolver.
>
> Que existan N estudiantes y que cada etapa deba ser llevada a cabo de manera independiente nos indica que hay que modelar a cada estudiante como un *proceso*. Recordemos que un *proceso*, en el contexto de esta unidad, es simplemente una instancia en ejecución del mismo programa. Es decir, **el código que veremos a continuación es igual para todos los procesos que queramos llamar**, en este caso, los estudiantes.
> Esto nos indica que hasta que todos los estudiantes no terminen de `implementarTP()`, nadie puede `experimentar()`. Necesitamos que, cuando algún estudiante termine de `implementarTP()`, espere a que el resto de compañeros también hayan terminado.

Es clave establecer el objetivo y dejar claro que tenemos que usar el mismo código para los posibles casos. (En el contexto de esta unidad, todos los *threads* comparten código).

Herramientas disponibles

Luego, establecería qué herramientas de sincronización hemos visto en la teoría y en la práctica. Le preguntaría a los estudiantes:

> ¿Cuál es la diferencia entre un *mutex* y un *semáforo*? ¿Cuál es su propósito?

Aunque alguien responda correctamente, me gustaría dar las siguientes definiciones para que quede claro a todos:

> Por un lado, los *mutex* son una herramienta de *aislamiento*.
> De esta manera, solo *un* proceso puede acceder a esa zona crítica. Esto es muy útil cuando dos o más procesos comparten el mismo código y modifican la misma variable. Sin el mutex, la variable puede sufrir incoherencias porque el sistema operativo es impredecible.
>
> Por otro lado, los *semaforos* son una herramienta de *sincronización*.
> De esta manera, diferentes procesos pueden organizarse para lograr sus objetivos. Es una forma de tener procesos que ejecutan el mismo código y se mandan señales para avisarse cuándo pueden avanzar o cuándo deben esperar.

Una vez explicada la diferencia, paso a la resolución del ejercicio.

Resolución

> Para este problema, podemos pedir a cada proceso estudiante que, cuando termine de `implementarTP()`, espere al resto de estudiantes usando un semáforo.
>
> Existen dos casos:
>
> 1. El estudiante que terminó de `implementarTP()` antes que el resto y tiene que esperar a que los demás terminen para poder `experimentar().
> 2. El estudiante que terminó último de `implementarTP()` y le tiene que avisar al resto para que *TODOS* puedan empezar a `experimentar()`.

> Pero recordemos que el código es igual para todos los procesos. ¿Cómo podemos modelar ambos casos?

Esto es clave repetirlo muchas veces; tiene que quedar muy claro que modelamos un mismo código para diferentes casos.

> Vamos a encararlo de la siguiente manera: vamos a modelar un semáforo que empieza bloqueado (*con luz roja*) y lo vamos a llamar *Barrera*.
>
> Si un estudiante termina y la cantidad de estudiantes que terminaron de `implementarTP()` es menor a N (o sea, no terminaron todos), entonces el estudiante actual se quedará esperando en el semáforo. Se queda *atrapado* en la barrera.
> Si un estudiante termina y la cantidad de estudiantes que terminaron de `implementarTP()` es exactamente N, significa que es el último estudiante en terminar. Por lo que le daremos luz verde a otro estudiante para que avance, y nos quedaremos esperando como los demás estudiantes.
> Este estudiante que avanza le da luz verde a otro estudiante. Este otro estudiante le da luz verde a otro, y así sucesivamente hasta que todos cruzan la barrera y pasan a la etapa de experimentacion. Es como un efecto dominó.
>
> Cada estudiante que estuvo alguna vez en *luz roja* avanzará a *luz verde* y podrá comenzar su fase de `experimentar()`.

Me gusta la abstracción de luz roja - luz verde para que los alumnos puedan imaginarse a cada proceso como un *auto* esperando el semaforo. De esta manera es un concepto más familiar y fácil de comprender.

Implementación 1

```
Semaphore barrera = new Semaphore(0)    // El semaforo empieza en luz roja (0 estudiantes
pueden pasar)
int terminaronImplementar = 0

ProcesoEstudiante() {
    implementarTP()

    terminaronImplementar += 1

    if (terminaronImplementar == N) {
        barrera.signal()    // Somos el último estudiante, damos luz verde a alguien que
está esperando
    }

    barrera.wait()    // Esperamos a que alguien nos dé luz verde
    barrera.signal()   // Damos luz verde a algún estudiante
    experimentar()
}
```

Acá preguntaría si está todo bien y si con esto el problema está resuelto. Parece que terminamos.

Ojo con las variables compartidas

> En la *implementación 1* parece que todo funciona de maravilla, pero hay un grave error: el código `ProcesoEstudiante()` va a ser ejecutado por múltiples procesos de forma concurrente y la variable `terminaronImplementar` es compartida por todos los procesos, pero no está protegida.

> Supongamos que tenemos N = 3 estudiantes.
>
> - El proceso del estudiante 1 está esperando en `barrera.wait()`.
> - El proceso del estudiante 2 lee `terminaronImplementar: 1` y la va a modificar para que sea `terminaronImplementar: 2`, pero es interrumpido por un *context switch* de threads del sistema operativo. Ahora se ejecuta el estudiante 3, que acaba de terminar de `implementarTP()`.
> - El proceso del estudiante 3 lee `terminaronImplementar: 1` (porque el proceso del estudiante 2 no llegó a modificarla). El proceso del estudiante 3 la modifica a `terminaronImplementar: 2` y se queda esperando en `barrera.wait()`.
> - El proceso del estudiante 2 obtiene tiempo de ejecución nuevamente y escribe `terminaronImplementar: 2`. El proceso del estudiante 2 no se dio cuenta de que el proceso del estudiante 3 modificó la variable.
> - Los 3 procesos se quedan esperando a que alguien ejecute `barrera.signal()` sin saber que eso jamás sucederá, porque hubo una *race condition*.

En texto es complicado de explicar, pero la idea se puede mostrar con un diagrama que represente la ejecución concurrente de los procesos y sus variables compartidas.

Luego preguntaría cómo resolver el tema de esta *race condition*. Si alguien responde, mejor.

> ¿Cómo lo resolvemos? Sencillo: un *mutex* para proteger la variable.

Implementación 2

```
Semaphore barrera = new Semaphore(0)    // El semáforo empieza en luz roja (0 estudiantes
pueden pasar)
Mutex mtx = new Mutex()
int terminaronImplementar = 0

ProcesoEstudiante() {
    implementarTP()

    mtx.lock()
    terminaronImplementar += 1

    if (terminaronImplementar == N) {
        barrera.signal()    // Somos el último estudiante, damos luz verde a alguien que
está esperando
    }
    mtx.unlock()

    barrera.wait()    // Esperamos a que alguien nos dé luz verde
    barrera.signal()   // Damos luz verde a algún estudiante
    experimentar()
}
```