Nombre: Santos Eduardo Gallegos Cevilla

Problema de concurrencia – El problema del salón de fiestas

Introducción

No resulta fácil entender los problemas que trae la concurrencia a primera vista, como lo es acceder a un recurso al mismo tiempo por dos procesos. Para ello se trata de explicar mediante ejemplos en algunos libros, pero su resolución a veces no queda muy clara o es demasiada técnica, y muchas veces no incluye una implementación total del problema en algún lenguaje de programación.
En el siguiente documento se resolverá paso a paso un problema de concurrencia obtenido del libro "*The Little Book of Semaphores*" por *Allen B. Downey*, el ejercicio pertenece a la sección *problemas remotamente no clásicos.*
Sólo para quedar claros con los términos usados en la resolución del problema se presenta un par de conceptos antes de ir con el problema.

Marco teórico

Semáforos

Un semáforo es una estructura de datos usada para la solución de problemas de sincronización. Fueron inventados por Edsger Dijkstra [1].
Un semáforo es como un número entero, con 3 diferencias:

1. Cuando se crea un semáforo, este debe ser inicializado a un valor, luego de esto el valor del semáforo no puede ser leído, las únicas operaciones válidas son incremente y decremento (por uno).
2. Si al decrementar el semáforo este resulta negativo, el thread que decrementó el semáforo se bloquea, puede continuar únicamente si otro thread incrementa el semáforo.
3. Cuando un thread incrementa el semáforo, si hay otros esperando, sólo uno se desbloquea.

Deadlock

Cuando un proceso se bloquea y no existe manera que otro proceso lo desbloquee, o el proceso encargado de desbloquearlo también está bloqueado, se produce un deadlock [1].

Patrones básicos de sincronización

A continuación de explicará brevemente algunos patrones que son relevantes para la solución del problema.

1. Señalización

La forma más simple de usar un semáforo, es usada para indicar el orden en que debe ejecutarse una sección de código [1].

En la imagen (Illustration 1) tenemos dos threads, se desea que primero se ejecute la sentencia a1, luego la b1. Para ello creamos un semáforo inicializado en 0. Si el thread B llega primero, el semáforo decrementa y queda negativo, por lo tanto el thread queda bloqueado. Luego el thread A se ejecuta e incrementa el semáforo, dando la señal de que el thread B pueda seguir. Si el thread A se ejecuta primero, entonces este incrementa el semáforo en 1, y nada ocurre. Luego se ejecuta el semáforo 2 y decrementa el semáforo, ejecuta la sentencia b1 y termina.

2. Cita

Es una generalización del patrón señalización. Se desea que un proceso de un thread se ejecute antes que el del otro thread [1].
En la imagen (Illustration 2) deseamos que se a1 se ejecute antes que b2, y b1 antes de a2. Para ello usamos dos semáforos inicializados en cero. El proceso siguiente se igual que el explicado en la señalización.

3. Mutex
El uso de semáforos como mutex garantiza que un proceso acceda a una variable compartida a la vez. Es como asignarle un turno a cada uno, conforme el proceso termina le cede el turno a otro [1]. En la imagen (Illustration 3) se tiene dos threads que acceden a la misma zona crítica, se desea asegurar que entre uno a la vez. Para ello se crea un semáforo inicializado en uno. El primer thread en ejecutarse llega al semáforo y lo decrementa (quedando en 0) y pasa al área crítica. Si el otro proceso intenta entrar decrementa el semáforo y dejándolo en un valor negativo, con lo cual queda bloqueado. Cuando el primer thread termina, vuelve a incrementar el semáforo dando paso al otro thread. Y así se ha asegurado que acceda un thread a la vez.

Problema
En una universidad hay un decano a cargo de los estudiantes, los estudiantes organizan una fiesta en un salón.
Las siguientes condiciones de sincronización se aplican a los estudiantes y al decano:

- 1. Cualquier número de estudiantes pueden estar en un cuarto al mismo tiempo.
- 2. El decano puede entrar al cuarto sólo si no hay estudiantes en el cuarto (para llevar a cabo una búsqueda) o si hay más de 50 estudiantes en el cuarto (para interrumpir la fiesta).
- 3. Mientras el decano está en el cuarto, ningún estudiante más puede entrar, pero si pueden retirarse.
- 4. El decano no puede irse del cuarto hasta que todos los estudiantes se hayan ido.
- 5. Hay un solo decano.

El objetivo del problema es escribir un código de sincronización para los estudiantes y el decano, de tal forma que cumpla con todas las condiciones.

Solución
La solución del ejercicio se dividirá en dos partes, la solución para el decano y la solución para los estudiantes, adjunto se encuentra la implementación de la solución en python.

Solución para los estudiantes
Las acciones que pueden realizar los estudiantes son:

1. Entrar a la sala
2. Realizar una fiesta en la sala
3. Salir de la sala.

Se consideran los siguientes casos para los estudiantes:

1. Si el decano está esperando y el estudiante que ingresa es el número 50, este debe informar al decano para que interrumpa la fiesta.
2. Si el decano está en el cuarto, y un estudiante quiere entrar, este debe esperar a que el decano salga.
3. Si el decano está en el cuarto, el último estudiante en salir debe informar al decano.

Una temprana solución luciría como en la Illustration 5.

Solución para el decano
Primero debemos definir las acciones que puede realizar el decano:

1. Entrar a la sala
2. Interrumpir la fiesta
3. Iniciar una búsqueda en la sala
4. Esperar fuera de la sala
5. Salir de la sala

Ahora consideramos que para el decano siempre se tienen 3 casos:

1. Si hay estudiantes en el cuarto, pero hay menos de 50, el decano debe esperar fuera.
2. Si hay más de 50 estudiantes, el decano debe entrar al cuarto e interrumpir la fiesta.
3. Si no hay estudiantes en el cuarto, este inicia la búsqueda y se va.

Una temprana solución para el decano luciría como en la Illustration 6.

Uniendo las soluciones.

Si nos fijamos en las soluciones, existen partes del código donde queremos que sólo las haga un estudiante o sólo el decano, y una vez termine este le pase el relevo a otro estudiante o al decano según se desee. Para esto debemos usar un semáforo del tipo *mutex*.

Para informar al decano si el cuarto no hay estudiantes o están 50 dentro, y bloquear la entrada a los demás estudiante, usaremos dos semáforos con el patrón cita. Así un estudiante no puede entrar hasta que el decano haya salido, y el decano no puede entrar hasta recibir la señal.

Implementación en python

Python nos provee a través del módulo *threading* la clase *Semaphore* con los métodos *acquire()* y *realease()* que son equivalentes a incrementar y decrementar el semáforo.

Se ha creado dos clases, decano y estudiante con las respectivas acciones (métodos) descritas en las soluciones.

Para iniciar el programa, se envía 100 estudiantes al cuarto y cuando entra el estudiante número 45, el decano entra en acción.

Para simular que el estudiante no sale inmediatamente cuando entra, se pone en espera durante 0.2 segundos al estudiante cuando realiza la fiesta.

El semáforo *mutuo* es el *mutex* del que se habló en la sección anterior; el semáforo *lleno_vacio* servirá para indicar el decano si el cuarto ya tiene más de 50 estudiantes o está vacío; y el semáforo *entrada* nos sirve para bloquear o ceder el paso a los estudiantes que deseen entrar al cuarto.

El código fuente se encuentra adjunto con el nombre de *room_party.py* y está basado en la solución descrita por el libro. Además se ha hecho uso de un sólo semáforo para informar si el cuarto está lleno o vacío, y un par de cambios para hacer más clara la implementación.

Referencias

[1] Allen B. Downey, *The Little Book of Semaphores*. .

Anexos

Thread A

```
1  statement a1
2  sem.signal()
```

Thread B

```
1  sem.wait()
2  statement b1
```

Illustration 1: Ejemplo de señalización

Thread A

```
1  statement a1
2  aArrived.signal()
3  bArrived.wait()
4  statement a2
```

Thread B

```
1  statement b1
2  bArrived.signal()
3  aArrived.wait()
4  statement b2
```

Illustration 2: Ejemplo de cita

Thread A

```
mutex.wait()
    # critical section
    count = count + 1
mutex.signal()
```

Thread B

```
mutex.wait()
    # critical section
    count = count + 1
mutex.signal()
```

Illustration 3: Ejemplo de mutex

```
Si el número estudiantes está entre 0 y 50 (no incluidos)
    El decano espera

Si el número de estudiantes es mayor o igual a 50
    El decano entra a interrumpir la fiesta
Sino (no hay estudiantes en el cuarto)
    El decano inicia la busqueda

El decano se retira
```

```
Si el estudiante fue el último en irse, informa al decano
```

Illustration 5: Pseudocodigo solución de estudiantes

```
Si el número estudiantes está entre 0 y 50 (no incluidos)
    El decano espera

Si el número de estudiantes es mayor o igual a 50
    El decano entra a interrumpir la fiesta
Sino (no hay estudiantes en el cuarto)
    El decano inicia la busqueda

El decano se retira
```

Illustration 6: Pseudocódigo de la solución del decano