Programación Concurrente
Programación Concurrente Multihilos - Java
Parte III

… Continuamos con los mecanismos para la sincronización por competencia

Semáforos Binarios para Exclusión Mutua

Otra posibilidad para lograr la exclusión mutua sobre una sección crítica de un recurso compartido son los semáforos.

Los semáforos son construcciones de control de concurrencia. Son usados para controlar el acceso a un recurso compartido o llevar a cabo una cierta acción al mismo tiempo. Un semáforo gestiona un conjunto de permisos virtuales, cuyo número inicial es dado al crear el semáforo.

Para centrarnos en exclusión mutua vamos a considerar un **semáforo BINARIO**, es decir un semáforo que gestiona solo 1 permiso, con el cual dará acceso a la sección crítica a un hilo. Como sólo gestiona un permiso, asegura la exclusión mutua, ya que mientras un hilo tenga ese permiso, ningun otro hilo podrá adquirirlo.

Las operaciones básicas para trabajar con semáforos son: – adquirir / ***acquire*** (Java) – liberar / ***release*** (Java). Las actividades pueden adquirir un permiso (siempre que haya disponible) para trabajar sobre una sección crítica y liberarlo cuando terminan. Si ningun permiso esta libre entonces la operacion de adquirir (adquire) se bloquea hasta que un permiso este libre, sea interrumpida o expire su tiempo. La operación de liberar un permiso (release) lo retorna al semáforo. Un **semáforo binario**, se utiliza como un **mutex** (exclusión mutua).
La clase **Semaphore de Java** también ofrece la operación ***tryAcquire*** que permite que el hilo interesado en adquirir un permiso pueda verificar si hay uno disponible, y asi evitar bloquearse. Esto es de utilidad cuando el hilo puede seguir con otra tarea mientras espera que se libere algun permiso.

Es una diferencia fundamental con “synchronized”, dado que en el caso de métodos y bloques sincronizados el hilo que intenta ejecutar un método sincronizado o un bloque sincronizado no tiene forma de evitar el bloqueo cuando el lock ya está en poder de otro hilo. En el caso del semáforo, si utiliza el *tryacquire* para solicitar el permiso y no lo consigue, no se bloquea y puede seguir con la ejecución de una parte de la tarea que no requiera de exclusión mutua.

La figura 2 siguiente muestra gráficamente como funciona la exclusión mutua con semáforos. Se puede observar que intervienen 3 hilos:

- Hilo0, es el hilo que crea e inicializa el semáforo ***sem*** con 1 permiso.
- Hilo1 e Hilo2 son los hilos que utilizan el semáforo para tener acceso a algun

recurso.

- Hilo1 e Hilo2 comienzan su ejecución.
- En el tiempo t_1 Hilo2 solicita el permiso del semáforo ***sem*** y lo obtiene, por lo que el semáforo queda con 0 permisos.
- En el tiempo t_2 Hilo1 solicita el permiso del semáforo ***sem*****,** pero no lo obtiene porque no hay permiso disponible, luego Hilo1 se bloquea hasta que el permiso vuelva a estar disponible.
- En el tiempo t_3 Hilo2 termina de trabajar sobre el recurso compartido en exclusion mutua y devuelve el permiso del semaforo sem. Ahora ***sem*** tiene un permiso disponible.
- En el tiempo t_4 Hilo1 se desbloquea y obtiene el permiso de ***sem***, por lo que ***sem*** vuelve a quedar sin permiso.

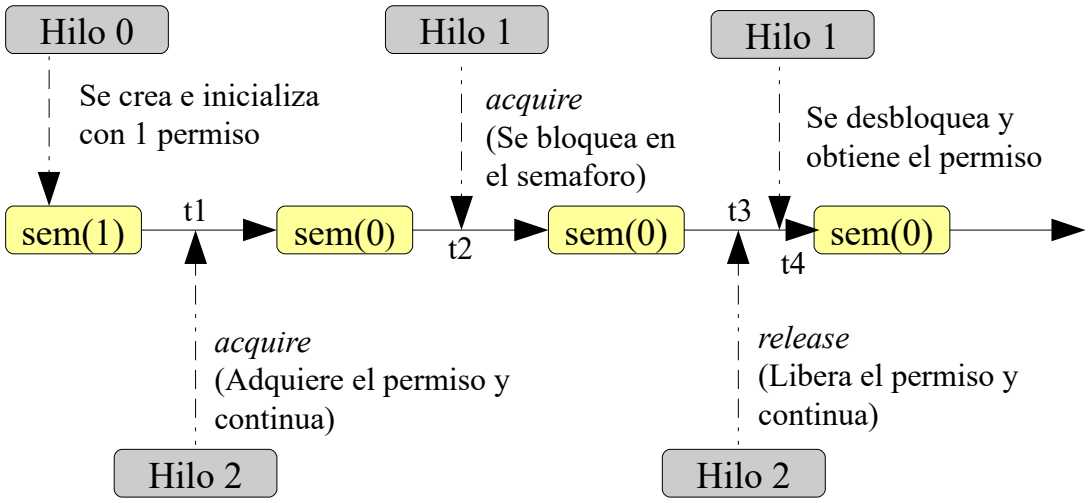
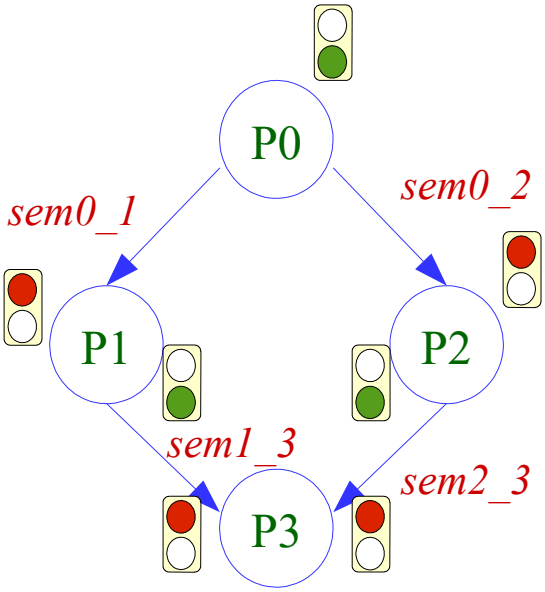


Figura 2: funcionamiento de un semáforo

---

* Los semáforos binarios también se pueden utilizan para establecer un ***orden en la ejecución de los hilos***, es decir un hilo se bloquea en un semáforo esperando que se le de el paso, y otro hilo, que debe ejercutarse con anterioridad será el que le de ese permiso. Los vamos a llamar **"semáforos de paso".**



Utilizamos un digrafo para expresar la precedencia entre los hilos

P0 se ejecute antes que P1 (sem0_1), P0 es quien da el permiso para que P1 pueda proceder
P0 se ejecute antes que P2 (sem0_2), P0 es quien da el permiso para que P2 pueda proceder
P1 se ejecute antes que P3 (sem1_3)
P2 se ejecute antes que P3 (sem1_4)
P3 debe esperar tanto a P1 como a P2, para poder proceder con su ejecución.

```
Proceso P0
        actuar  P0
        liberar sem0_1    //da permiso a P1 para proceder
        liberar sem0_2    //da permiso a P2 para proceder
fin P0
```

```
Proceso P1
        adquirir sem0_1  //espera el permiso de P0 para proceder
        actuar P1
        liberar sem1_3   //da permiso a P3 para proceder
fin P1
```

---

* Un semáforo binario también puede utilizarse cuando es necesaria una sincronización tipo "rendezvous" entre 2 hilos.

¿Qué es una sincronización tipo "rendezvous" ?
Es una situación en la cual dos hilos necesitan encontrarse en un punto en el tiempo para poder comunicarse y continuar. Se da una interacción entre ellos. El primer hilo espera que "llegue" el

segundo hilo para continuar y el segundo hilo espera que "llegue" el primer hilo para continuar.

Una aplicación de rendezvous sería por ejemplo al modelar e implementar la situación siguiente:
Considere un pequeño restaurant en el que hay un cocinero y un mozo. El cocinero debe esperar a que el mozo le entregue un pedido solicitado por un cliente y luego el mozo debe esperar a que el cocinero le entregue la comida para llevar al cliente. Aqui se da una interacción entre el mozo y el cocinero. El mozo atiende un pedido por vez. La interacción entre el cocinero y el mozo requiere de la utilización de 2 semáforos:

- 1 semaforo ´"pedido" para la comunicación entre el mozo y el cocinero cuando se entrega el pedido del cliente. El cocinero espera que el mozo le avise, es decir el cocinero espera un permiso del semáforo y el mozo es el encargado de liberar ese permiso (cuando tenga un pedido para solicitar)
- 1 semáforo "comidaLista" para la comunicación entre el cocinero y el mozo cuando la comida esta lista para servirla. El mozo espera que el cocinero le avise, es decir el mozo espera un permiso del semáforo y el cocinero es el encargado de liberar ese permiso (cuando termina de preparar la comida)

Si bien en muchas ocasiones se encontrará que este tipo de sincronización es controlado directamente por los hilos, en la materia elegimos dar la responsabilidad de la sincronización al recurso compartido, y por ello consideraremos un objeto "comida" para resolver este problema.

En pseudocódigo:

```
hilo Mozo
método generar pedido ()
// metodo propio que simula la recepcion de un pedido

método run ()
    mientras  true
        generar pedido ()
        pedir comida ()
        esperar comida  ()
```

```
hilo Cocinero
método preparar comida()
// metodo propio que simula el tiempo que ocupa el cocinero
en preparar la comida para atender el pedido recibido

método run ()
   mientras true
        esperar pedido ()
        preparar comida ()   // método propio del cocinero
        entregar pedido  ()
```

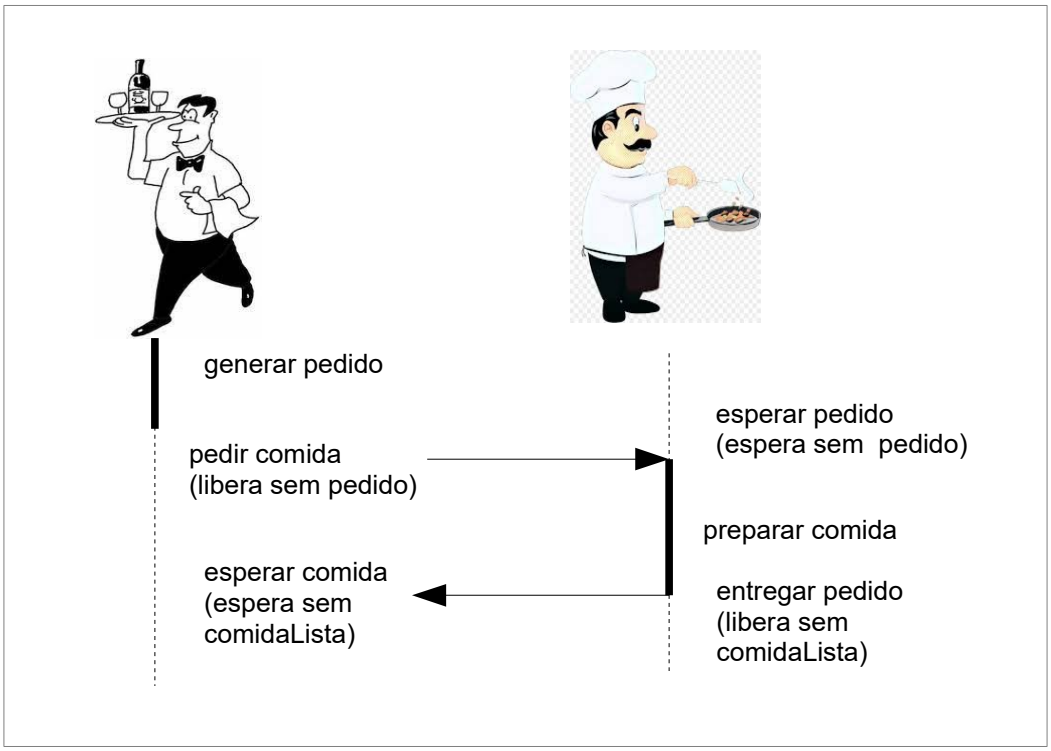
```
comida  // tiene los 2 semaforos indicados
        sem pedido   (0)
        sem comidaLista (0)

        pedir comida ()
                liberar pedido

        esperar comida ()
                adquirir  comidaLista

        esperar pedido ()
                adquirir pedido

        entregar pedido ()
                liberar comidaLista
```



Otra posibilidad para lograr la exclusión mutua: Cerrojos
Interfaz lock en Java

Un cerrojo es una llave de acceso a una sección crítica. A diferencia de los locks o llaves "**implícitas**" disponibles a través del uso del método básico de sincronización (por utilización de ***synchronized*** en Java), **un cerrojo es un lock "explícito"**, es decir es una llave de acceso a la sección crítica que debe ser adquirida explícitamente y al finalizar la sección crítica debe ser liberada explícitamente.

Para trabajar con cerrojos, Java provee la interfaz LOCK y su implementación REENTRANTLOCK.

… Continuará …

Bibliografia

- **Concurrent Programming in JAVA: design principles and patterns** - Doug Lea
- **Thinking in JAVA** - Bruce Eckel - President, MindView Inc. - 2008
- **Java Concurrency in Practice** - Brian Goetz - et all - Addison Wesley 2006.
- **Documentacion Oracle**: http://docs.oracle.com/javase/tutorial/essential/concurrency/ y el paquete java.util.concurrent.
- **Orientación a Objetos con Java y UML** - Carlos Fontella - nueva libreria nl 2004