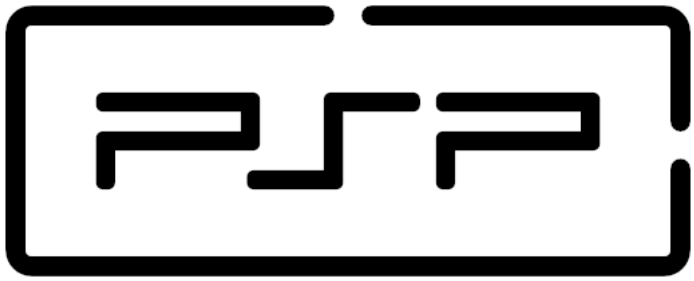

Programación de Servicios y Procesos

3.4 Mecanismos alternativos de sincronización

3.4 Mecanismos alternativos de sincronización

3.4.1. Semáforos

Otro posible mecanismo para sincronizar hilos son los `semáforos`. Un semáforo es un mecanismo para permitir, o restringir, el acceso a recursos compartidos en un entorno de multiprocesamiento, con varios hilos ejecutándose de forma concurrente.

> Especificación de **java.util.concurrent.Semaphore (https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/Semaphore.html)**

Los semáforos se emplean para permitir el acceso a diferentes partes de programas (llamados secciones críticas) donde se manipulan variables o recursos que deben ser accedidos de forma especial. Según el valor con que son inicializados se permiten a más o menos procesos utilizar el recurso de forma simultánea.

El funcionamiento de los semáforos se basa en el uso de dos métodos, así como en el valor inicial `permits` con el que se crea el semáforo:

- release(): Ejecutado por un hilo para liberar el semáforo cuando el hilo ha terminado de ejecutar la sección crítica. Por defecto se incrementa la variable `permits` en 1, aunque puede recibir un valor e incrementarla en esa cantidad.
- acquire(): Ejecutado por un hilo para acceder al semáforo. Para que un hilo pueda tomar el control del semáforo y no quedarse bloqueado, la variable `permits` debe tener un valor mayor que cero. También puede recibir un valor, por lo que `permits` tendrá que ser mayor que dicho valor.
- permits: Se inicializa a la cantidad de recursos existentes o hilos que queramos que puedan acceder simultáneamente. Así, cada proceso, al ir solicitando un recurso, verificará que el valor del semáforo sea mayor de 0; si es así es que existen recursos libres, seguidamente acaparará el recurso y restará el valor del semáforo. Cuando el semáforo alcance el valor 0, significará que todos los recursos están siendo utilizados, y los procesos que quieran solicitar un recurso deberán esperar a que el semáforo sea positivo (algún hilo haga un release).

> **Mutex**
>
> Un tipo simple de semáforo es el binario, que puede tomar solamente los valores 0 y 1.
>
> Se inicializan en 1 y son usados cuando sólo un proceso puede acceder a un recurso a la vez. Se les suele llamar mutex.
>
> Tienen un funcionamiento similar a synchronized, funcionando en exclusión mutua (**mut**ual **ex**clusion).

Veamos un ejemplo en el que varios Productores y Consumidores acceden de forma simultánea a un objeto compartido

```java
public class Almacen {

  private final int MAX_LIMITE = 20;
  private int producto = 0;
  private Semaphore productor = new Semaphore(MAX_LIMITE);
  private Semaphore consumidor = new Semaphore(0);
```

```
    private Semaphore mutex = new Semaphore(1);

    public void producir(String nombreProductor) {
      System.out.println(nombreProductor + " intentando almacenar un producto");
      try {
          // En el ejemplo, hasta 20 productores pueden acceder a la vez
          productor.acquire();
          // Sin embargo, sólo 1 (consumidor/productor) a la vez podrá actualizar
          mutex.acquire();

          producto++;
          System.out.println(nombreProductor + " almacena un producto. "
              + "Almacén con " + producto + (producto > 1 ? " productos." : " producto."));
          mutex.release();

          Thread.sleep(500);

      } catch (InterruptedException ex) {
        Logger.getLogger(Almacen.class.getName()).log(Level.SEVERE, null, ex);
      } finally {
        // El productor permite que un consumidor pueda acceder
        consumidor.release();
      }

    }

    public void consumir(String nombreConsumidor) {
      System.out.println(nombreConsumidor + " intentando retirar un producto");
      try {
          // En el ejemplo siempre tiene que llegar un consumidor antes que un productor
          consumidor.acquire();
          // Sin embargo, sólo 1 (consumidor/productor) a la vez podrá actualizar
          mutex.acquire();

          producto--;
          System.out.println(nombreConsumidor + " retira un producto. "
              + "Almacén con " + producto + (producto > 1 ? " productos." : " producto."));
          mutex.release();

          Thread.sleep(500);
      } catch (InterruptedException ex) {
        Logger.getLogger(Almacen.class.getName()).log(Level.SEVERE, null, ex);
      } finally {
        // El consumidor avisa para que un productor pueda volver a dejar productos.
        productor.release();

      }
    }

}
```

3.4.2. Mecanismos de alto nivel

Java, en su paquete `java.util.concurrent` proporciona varias clases `thread-safe` que nos permiten acceder a los elementos de colecciones y tipos de datos sin preocuparnos de la concurrencia.

Es un paquete muy amplio que contiene multitud de clases que podemos utilizar en nuestros desarrollos multihilo para simplificar la complejidad de los mismos.

Colas concurrentes

La interfaz **BlockingQueue** define una cola `FIFO` que bloquea hilos que intentan extraer un elemento si la cola está vacía, hasta que vuelva a haber elementos. También permite establecer un número máximo de elementos, de marea que se bloquean los procesos cuando intentan añadir por encima de ese límite, a la espera que se extraigan.

Las clases LinkedBlockingQueue, ArrayBlockingQueue, SynchronousQueue, PriorityBlockingQueue y DelayQueue implementan la interfaz BlockingQueue.

Colecciones concurrentes

El uso de colecciones simultáneas es una forma recomendada de crear estructuras de datos compatibles con procesos. Dichas colecciones incluyen ConcurrentHashMap, ConcurrentSkipListMap, ConcurrentSkipListSet, CopyOnWriteArraylist y CopyOnWriteArraySet.

ConcurrentMap es una subinterfaz de `java.util.Map` con operaciones atómicas para eliminar o reemplazar pares clave/valor existentes o añadir pares clave/valor no existentes. ConcurrentHashMap es la versión thread-safe análoga a HashMap.

Variables atómicas

El paquete `java.util.concurrent.atomic` incluye clases que proporcionan acciones atómicas sobre tipos de datos básicos. Tenemos AtomicBoolean, AtomicInteger, AtomicDouble, ... y proporcionan métodos para recuperar su valor, incrementar, decrementar, etc.

3.4.3 Executors, Callables y Future

Existen muchas aproximaciones y librerías que permiten el uso y gestión de hilos desde un programa. Una de las que nos ofrece Java como parte del JDK es la interfaz Executors.

`Executors` nos va a permitir definir un pool de threads (un conjunto de hilos) que se encargarán de ejecutar las tareas, pero con un límite en cuanto al número de hilos creados y gestionando la JVM la cola de hilos que serán ejecutados en ese pool.

Se sale del ámbito de este módulo estudiar y analizar el funcionamiento de Executors y todas sus posibilidades. Aquí os dejo un enlace a un artículo que lo explica con un ejemplo muy ilustrativo.

Executors: Ejemplo supermercado (https://jarroba.com/multitarea-e-hilos-en-java-con-ejemplos-ii-runnable-executors/)

`Callable` viene a poner solución a uno de los problemas que tenemos con la interfaz Runnable, la posibilidad de devolver un valor desde este método.

Si se necesita que un proceso devuelva datos al finalizar, se debe crear una clase que implemente la interfaz Callable y defina un método `call()` que desempeñe la misma función que run() en Runnable. En este caso se tendrán que crear los procesos de forma diferente; la clase Thread no acepta un objeto Callable como argumento. Por contra, la clase Executors ofrece diversos métodos estáticos que crean un proceso a partir de su clase Callable.

`Future` es una interfaz que implementa el objeto que devuelve el resultado de la ejecución de un Callable. Se puede seguir ejecutando una aplicación hasta que necesite obtener el resultado del hilo *Callable*, momento en el que se invoca el método get() en la instancia Future. Si el resultado ya está disponible se recoge y en caso contrario se bloqueará en la llamada hasta que su método call() devuelva el resultado.