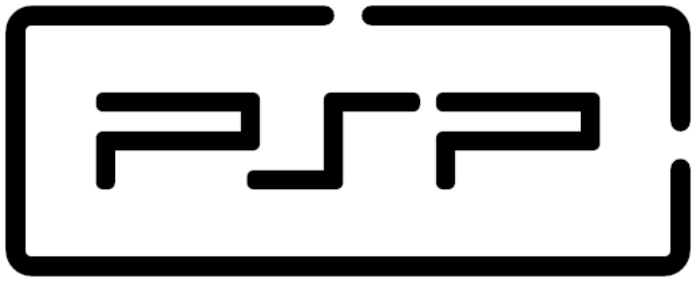

Process and Service Programming

3.4 Alternative synchronization techniques

3.4 Alternative synchronization techniques

3.4.1. Semaphores

There are many other ways to synchronize threads, one of the low-level ones ar `semaphores`. A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. What the counter is counting are permits that allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.

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

Semaphores control access to `critical sections` where shared resources or variables are handled in a special way. Depending on thi initial value of the semaphore, a number of concurrent threads can access simultaneously to a shared resource.

Semaphores can be manages with two methods and their initial value `permits`:

- release(): When thread no longer needs access to a shared resource, it releases the permit, incrementing the semaphore count. By default the semaphore counter `permits` is incremented by 1, though it can get a value and increment the count in that value.
- acquire(): If a thread needs to access a shared resource or critical section, then it must get control over the semaphore. If semaphore count > 0, the thread acquires a permit, decrementing the semaphore's count. Else, the thread is blocked until a permit can be acquired. Other value than 1 can be used to get the semaphore, having `permits` to be bigger than that value in order to get semaphore's control
- permits: The value of a counting semaphore at any point indicates the maximum number of processes that can enter the critical section at the exact same time. Each thread asks for a permit. if value is bigger than 0 that means free resources are available, so the thread will enter the semaphore and reduce the permit count When the semaphore's permit count reaches to 0 that means no more shared resources are available and threads will be locked waiting for another thread to perform a release action on the semaphore.

> **Mutex**
>
> Binary semaphore: A binary semaphore only takes only 0 and 1 as values and is used to implement mutual exclusion as well as synchronize concurrent processes.
>
> The work similar to synchronized, providing **mut**ual **ex**clusion.

Let's take a look at this example

```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 {
          // up to 20 producers can enter at the same time
          productor.acquire();
          // But only 1 (consumer/producer) at a time can update
          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 {
        // Producers allow (notify) consumers to access
        consumidor.release();
      }

    }

    public void consumir(String nombreConsumidor) {
      System.out.println(nombreConsumidor + " intentando retirar un producto");
      try {
          // A producer must be run first, before any consumer
          consumidor.acquire();
          // But only 1 (consumer/producer) at a time can update
          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 {
        // Consumers allow (notify) producers to add more products
        productor.release();

      }
    }

}
```

3.4.2. High level synchronization techniques

The `java.util.concurrent` package provides tools for creating concurrent applications. There are some `thread-safe` classes que to use Collections and basic data types without worrying about concurrent access.

Using these classes in our code we can reduce out apps complexity.

Concurrent Queues

The **BlockingQueue** interface defines a `FIFO` queue that locks threads trying to get elementos from an empty queue until there will be elements in the queue. it can set a maximum number of elements in the queue so that thread are blocked if they try to add elements over that number, having to wait until elements are extracted form the queue.

Classes LinkedBlockingQueue, ArrayBlockingQueue, SynchronousQueue, PriorityBlockingQueue and DelayQueue implement interface BlockingQueue.

Concurrent Collections

Besides Queues, this package supplies Collection implementations designed for use in multithreaded contexts: ConcurrentHashMap, ConcurrentSkipListMap, ConcurrentSkipListSet, CopyOnWriteArrayList, and CopyOnWriteArraySet. When many threads are expected to access a given collection, a ConcurrentHashMap is normally preferable to a synchronized HashMap, and a ConcurrentSkipListMap is normally preferable to a synchronized TreeMap. A CopyOnWriteArrayList is preferable to a synchronized ArrayList when the expected number of reads and traversals greatly outnumber the number of updates to a list.

ConcurrentMap is a subinterface of `java.util.Map` con with atomic operations to add / replace existing key,value pairs or to add non existing key,value pairs. ConcurrentHashMap is the thread-safe version for HashMap.

Atomic variables

Package `java.util.concurrent.atomic` contains a small toolkit of classes that support lock-free thread-safe programming on single variables. Instances of Atomic classes maintain values that are accessed and updated using methods otherwise available for fields using associated atomic VarHandle operations.

Instances of classes AtomicBoolean, AtomicInteger, AtomicLong, and AtomicReference each provide access and updates to a single variable of the corresponding type. Each class also provides appropriate utility methods for that type. For example, classes AtomicLong and AtomicInteger provide atomic increment methods.

3.4.3 Executors, Callables & Future

Executors is an interface to manage thread pools. Thread pools manage a pool of worker threads. The thread pools contain a work queue which holds tasks waiting to get executed.

A thread pool can be described as a collection of Runnable/Callable objects (work queue) and a connection of running threads.

These threads are constantly running and are checking the work query for new work. If there is new work to be done they execute this Runnable/Callable.

Here you can check an illustrative example on how to use Executors

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

We have used a Runnable object to define the tasks that are executed inside a thread. While defining tasks using Runnable is very convenient, it is limited by the fact that the tasks can not return a result.

What if you want to return a result from your tasks?

Well, Java provides a `Callable` interface to define tasks that return a result. A Callable is similar to Runnable except that it can return a result and throw a checked exception.

Callable interface has a single method call() which is meant to contain the code that is executed by a thread.

`Future` interface has methods to obtain the result generated by a Callable object and manage its state. It represents the result of an asynchronous computation.

The result can only be retrieved using method get() when the computation has completed, blocking if necessary until it is ready.