Jukebox: Semaphores

Table of contents

1. Background

Initially, Jukebox used `Win16` semaphores, then `OS/2` semaphores, then, when the project was being ported into Java, it turned out that there's no adequate concept. Well, it was silly to bring all the OS-dependent stuff along, so new semaphore model was created - it is strange, but then again, over a decade of production usage prove that it has the right to exist.

2. Introduction

Java core didn't provide semaphore semantics until `JDK 1.5`. To fix this annoying oversight, the following semaphore abstractions were created:

- Basic semaphore abstraction. Provides the `waitFor()` template method, with indefinite and timed options.

  When the `waitFor(timeout)` is not satisfied within a timeout, the exception is thrown. Otherwise, a boolean result corresponding to the semaphore status is returned.

Event semaphore. Provides inter-thread synchronization based on events.

Mutex semaphore. Classic one.

ACT, a.k.a. Asynchronous Completion Token. One-time use, disposable event semaphore.

Semaphore group. Provides multiple-wait feature, very useful in complex systems.

Reader/writer lock. Self-explanatory.

3. Jukebox Semaphores vs. java.util.concurrent

They do not compete, but rather compliment each other. First of all, Jukebox semaphores precede `java.util.concurrent` by about a decade (`java.util.concurrent` appeared in `JDK 1.5`, and Jukebox semaphores in late 1995), second, they were created in times when the only concurrency currency, pun intended, was the `synchronized` keyword and Java itself wasn't that sophisticated, either.

Today, Jukebox semaphores and `java.util.concurrent` coexist happily because they are at somewhat different levels of abstraction, and the entity maps they operate with are not quite compatible. Sometimes, Jukebox semaphores can be rewritten simpler using tools that `java.util.concurrent` privdes, and it's not "if", it's "when" it will happen.

4. Event Semaphore Semantics

Event semaphore is the most complicated animal here. It is not like any other semaphore in any operating system, but the rationale behind the design has survived many projects for over a decade, so there's a good chance it was a right decision to make.

Detailed description, including sequence diagrams, will be provided later (if ever), for now, please see the Javadoc documentation for `net.sf.jukebox.sem.EventSemaphore` class.

Simplistic explanation: Suppose there's a mailbox, a postman and a subscriber.

- The mailbox can contain at most one envelope.
- The subscriber waits until the mailbox is not empty.
- The postman always delivers one envelope, either white or black.
- If the mailbox is not empty, the postman discards the contents of the mailbox, and then puts the new envelope into the mailbox.
- When the subscriber finds the envelope in the mailbox, he removes it, and the cycle repeats.

The mailbox is the semaphore, the postman is the one who calls `trigger()` on the semaphore (the producer), and the subscriber is the one who calls `waitFor()` (the consumer).

Note:

If the subscriber decided to go to the restroom, or decided to start looking for an envelope too late, and the postman came more than once before the subscriber had a chance to check the content of the mailbox, the subscriber has missed at least one envelope.

Note:

If the subscriber came after the postman, he doesn't discard the contents of the mailbox, but instead just takes the envelope and goes on his business.

5. Mutex Semaphore Semantics

A mutex (mutual exlusive) semaphore is not really needed in Java at the first glance because the functionality is apparently provided by the `synchronized` modifier. However, it represents a useful abstraction, and sometimes the clarity should go before performance.

One more consideration, mutex, as well as event semaphore, may be included into the semaphore group - now the usefulness becomes obvious.

And the last consideration - the mutex abstraction is safe in distributed context (many hosts running parts of the application; never mind that it's silly to use mutexes in a distributed

context), whereas `synchronized` modifier is not.

6. Asynchronous Completion Token Semantics

Asynchronous Completion Token (`ACT` hereinafter) is nothing other than one-time use, disposable event semaphore. It is intended to allow waiting on the asynchronous process completion, as well as to pass the result of it back to the waiting entity.

`ACT` can be triggered only once.

Simplistic explanation: suppose you go to the restaurant.

- You make an order, and the waiter gives you a box.
- You go minding your own business.
- At the time the order is ready, your box beeps, you open the small door on its side, and your order drops out of the box into your lap.
- However, if the order could not be completed for some reason, the hand grenade with the pin out drops out of the box instead of your order.

The box is the `ACT`. The beeper is the representation of the `ACT` as a semaphore you can `waitFor()`. The box content is the result that the `ACT` supplier wanted to deliver to you when signalling the completion. The hand grenade is the exception that might have caused the process failure.

The `ACT` is related to the concept of the `Messenger` from the Services component.

7. Semaphore Group Semantics

The semaphore group provides the multiple-wait semantics. There are two kinds of multiple wait:

- Wait for one. The `waitForOne()` method returns as soon as one of the semaphores belonging to the group triggers (variant: triggers with desired value).

  Use Case: multiple threads looking for data, the data from the thread that finished first is taken.
- Wait for all. The `waitForAll()` method returns only after all the semaphores in the group have been triggered (variant: triggered with desired value; variant: returns as soon as there was at least one value which is not desired).

  Use Case: the condition is satisfied only after all the sub-conditions are satisfied.

Note:

`ACT` can be a member of the semaphore group as well. However, it can be triggered only once, that's why it has to be removed from the group. In practice, the semaphore groups are rarely reused, so using ACT is OK.

8. Reader/Writer Lock Semantics

The single writer, multiple reader lock is a pretty trivial thing. The only non-trivial thing about this particular implementation is that it is not anonymous. The entity requesting the lock must get the lock token, and use the lock token to release the lock afterwards, like this:

```
RWLock lock;
Object lockToken = null;

try {

  lockToken = lock.getReadLock();

  doRead();

  lockToken = lock.release(lockToken);
  lockToken = lock.getWriterLock();

  doWrite();

} finally
  lockToken = lock.release(lockToken);
}
```

Note:

The lock token is assigned the value returned by `lock.release()`. It is always null, and it was done so provide the 'syntax sugar' to save an extra line - you always have to assign a valid value to the lock token, lest the lock will refuse to take it. Think about the situation when the assignment to `null` is not done, and an attempt to get a write lock fails with exception.

Warning:

Caution and common sense must be excercised to avoid the resource starvation. Since the writers have inherent priority against readers, it is possible that the readers will be starving if there's too many writers.