Since: **1.5**

Module java.base

Package java.util.concurrent

Class Semaphore

Java.util.concurrent.Semaphore

public class Semaphore extends Object implements Serializable

---

Java.util.concurrent.Semaphore

Conceptually, a semaphore maintains a set of permits.

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.

In general, to use a semaphore, the thread that wants access to the shared resource tries to acquire a permit.

If the semaphore's count is greater than zero, then the thread acquires a permit, which causes the semaphore's count to be decremented. Otherwise, the thread will be blocked until a permit can be acquired.

When the thread no longer needs access to the shared resource, it releases the permit, which causes the semaphore's count to be incremented.

If there is another thread waiting for a permit, then that thread will acquire a permit at that time. Java's Semaphore class implements this mechanism.

Semaphore has the two constructors: -

1. **Semaphore (int permits):** - Here, **permits** specifies the initial permit count. Thus, **permits** specifies the number of threads that can access a shared

resource at any one time. If **permit** is one, then only one thread can access the resource at any one time.

2. **Semaphore (int permits, boolean fair):** -Creates a Semaphore with the given number of permits and the given fairness setting. By default, waiting threads are granted a permit in an **undefined order**. By setting **fair** to **true**, you can ensure that waiting threads are **granted a permit in the order in which they requested access.**

Methods of Semaphore: -

1. **void acquire ()** Acquires a permit from this semaphore, blocking until one is available, or the thread is interrupted.
2. **void acquire (int permits)** Acquires the given number of permits from this semaphore, blocking until all are available, or the thread is interrupted.
3. **void acquireUninterruptibly()** Acquires a permit from this semaphore, blocking until one is available.
4. **void acquireUninterruptibly (int permits)** Acquires the given number of permits from this semaphore, blocking until all are available.
5. **int availablePermits()** Returns the current number of permits available in this semaphore.
6. **int drainPermits()** Acquires and returns all permits that are immediately available, or if negative permits are available, releases them.
7. **protected Collection<Thread> getQueuedThreads()** Returns a collection containing threads that may be waiting to acquire.
8. **int getQueueLength()** Returns an estimate of the number of threads waiting to acquire.
9. **boolean hasQueuedThreads()** Queries whether any threads are waiting to acquire.
10. **boolean isFair()** Returns true if this semaphore has fairness set true.
11. **protected void reducePermits (int reduction)** Shrinks the number of available permits by the indicated reduction.
12. **void release ()** Releases a permit, returning it to the semaphore.
13. void release (int permits) Releases the given number of permits, returning them to the semaphore.
14. **String toString()** Returns a string identifying this semaphore, as well as its state.

15. **boolean tryAcquire()** Acquires a permit from this semaphore, only if one is available at the time of invocation.
16. **boolean tryAcquire (int permits)** Acquires the given number of permits from this semaphore, only if all are available at the time of invocation.
17. **boolean tryAcquire (int permits, long timeout, TimeUnit unit)** Acquires the given number of permits from this semaphore, if all become available within the given waiting time and the current thread has not been interrupted.
18. **boolean tryAcquire (long timeout, TimeUnit unit)** Acquires a permit from this semaphore, if one becomes available within the given waiting time and the current thread has not been interrupted.

What is the difference between acquire () and acquire (int permits)?

Answer: -

acquire ()	**acquire (int permits)**
It requires one permit.	It requires number of permits which are passed to methods parameters.

What is the difference between acquire () and **acquireUninterruptibly()?**

Answer: -

acquire ()	**acquireUninterruptibly()**
Acquire () is interruptible. That means if a thread A is calling acquire () on a semaphore, and thread B interrupts threads A by calling interrupt (), then an **InterruptedException** will be thrown on thread A.	**acquireUninterruptibly**() is not interruptible. That means if a thread A is calling acquireUninterruptibly() on a semaphore, and thread B interrupts threads A by calling interrupt (), then no **InterruptedException** will be thrown on thread A, just that thread A will have its interrupted status set after acquireUninterruptibly() returns.

What is the difference between acquire () and **tryAcquire ()?**

Answer: -

acquire ()	**tryAcquire ()**
Acquires a permit from this semaphore, blocking until one is available, or the thread is interrupted.	Acquires a permit from this semaphore, only if one is available at the time of invocation.
Acquires a permit, if one is available and returns immediately, reducing the number of available permits by one.	Acquires a permit, if one is available and returns immediately, with the value true, reducing the number of available permits by one.
If no permit is available then the current thread becomes disabled for thread scheduling purposes and lies dormant until one of two things happens: • Some other thread invokes the release () method for this semaphore and the current thread is next to be assigned a permit; or • Some other thread interrupts the current thread.	If no permit is available then this method will return immediately with the value false.

Give real life example of Semaphore uses?

Answer: - Suppose there is an ATM house where 3 numbers of ATM machines are available, so at a time only 3 users can do the transactions simultaneously. Think that more than 20 Persons trying to access the ATM machines at a same time, if there is no any crowd management team available then might be hotchpotch situation came. Here we can take help of Semaphore to manage the crowd.

Here Semaphore guard the gate of ATM house and will allow only 3 persons to enter the ATM house, i.e., it only gives permits to 3 persons only and rest have to wait and seek permit from him, as one person completed transaction and release the ATM machine, then Semaphore will give that ATM machine permission to next person, similar rest people take their turn.

Some More!!!

Using Semaphore as Lock

Java allows us to use a semaphore as a lock. It means, it locks the access to the resource. Any thread that wants to access the locked resource, must call the **acquire ()** method before accessing the resource to acquire the lock. The thread must release the lock by calling the **release ()** method, after the completion of the task. Remember that set the upper bound to 1.

Characteristics of Semaphore

There are the following characteristics of a semaphore:

- It provides synchronization among the threads.
- It decreases the level of synchronization. Hence, provides a low-level synchronization mechanism.
- The semaphore does not contain a negative value. It holds a value that may either greater than zero or equal to zero.
- We can implement semaphore using the test operation and interrupts, and we use the file descriptors for executing it.

Types of Semaphores

There are four types of semaphores, which are as follows:

- Counting Semaphores
- Bounded Semaphores
- Timed Semaphores
- Binary Semaphores

Counting Semaphores

The Counting Semaphores are used to resolve the situation in which more than one process wants to execute in the critical section, simultaneously. Hence to overcome this problem we use counting semaphore.

Bounded Semaphores

We can set the upper bound limit using the **bounded semaphores**. It is used in place of the counting semaphores because the counting semaphores do not contain any upper bound value. The upper bound value denotes **how many signals it can store.**

Timed Semaphores

The **timed semaphores** allow a thread to run for a specified period of time. After a particular time, the timer resets and releases all other permits.

Binary Semaphores

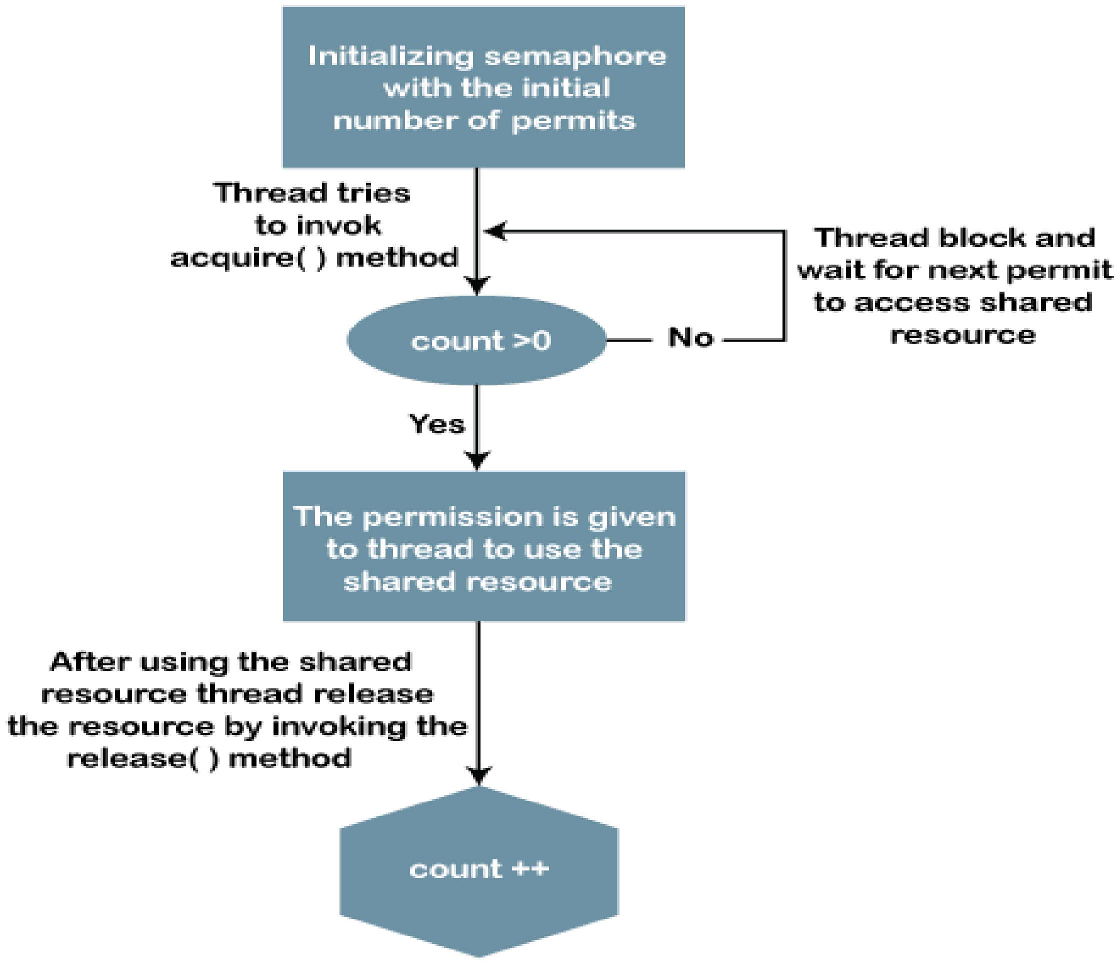
The Binary Semaphores are the same as counting semaphores. But remember that it accepts only binary values either 0 or 1. Its implementation is easy in comparison to other semaphores. If the value is 1, the signal operation is success, fails otherwise.

Working of Semaphore

Semaphore controls over the shared resource through a counter variable. The counter is a non-negative value. It contains a value either greater than 0 or equal to 0.

- If **counter > 0**, the thread gets permission to access the shared resource and the counter value is **decremented** by 1.
- Else, the thread will be blocked until a permit can be acquired.
- When the execution of the thread is completed then there is no need for the resource and the thread releases it. After releasing the resource, the counter value **incremented** by 1.
- If another thread is waiting for acquiring a resource, the thread will acquire a permit at that time.
- If **counter = 0**, the thread does not get permission to access the shared resource.

Let's understand the working of semaphore with the help of a flow chart.


Initializing semaphore with the initial number of permits
Thread tries to invok acquire( ) method
count >0
No
Thread block and wait for next permit to access shared resource
Yes
The permission is given to thread to use the shared resource
After using the shared resource thread release the resource by invoking the release( ) method
count ++

Working of Semaphore in Java

Scenario where Semaphore can be used:

1. To implement a better Database connection pool which will block if no more connection is available instead of failing and handover Connection as soon as it's available.
2. To put a bound on collection classes. by using semaphore, you can implement bounded collection whose bound is specified by counting semaphore.
3. To guard a critical section against entry by more than N threads at a time.
4. To send signals between two threads.