Semaphore in Java

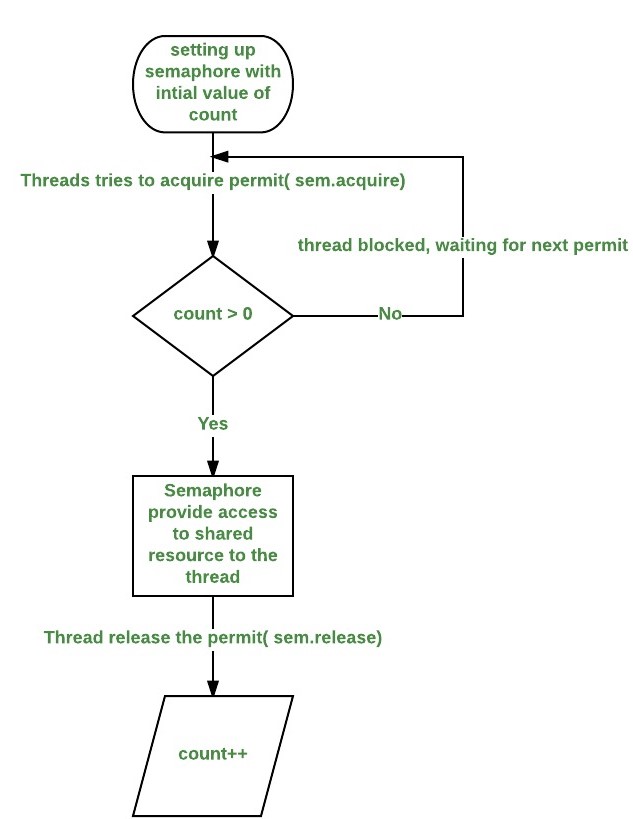
A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. What the counter is counting are permits that allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.

**Working of 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 an 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 provide **Semaphore** class in *java.util.concurrent* package that implements this mechanism, so you don’t have to implement your own semaphores.

**Flow Diagram :**  
[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/d2.jpeg)

**Constructors in Semaphore class :** There are two constructors in Semaphore class.

Semaphore(int num)

Semaphore(int num, boolean how)

Here, *num* specifies the initial permit count. Thus, it specifies the number of threads that can access a shared resource at any one time. If it is one, then only one thread can access the resource at any one time. By default, all waiting threads are granted a permit in an undefined order. By setting *how* to true, you can ensure that waiting threads are granted a permit in the order in which they requested access.

**Using Semaphores as Locks(**[**preventing race condition**](http://practice.geeksforgeeks.org/problems/what-is-race-condition)**)**

We can use a semaphore to lock access to a resource, each thread that wants to use that resource must first call *acquire( )* before accessing the resource to acquire the lock. When the thread is done with the resource, it must call *release( )* to release lock. Here is an example that demonstrate this:

|  |
| --- |
| // java program to demonstrate  // use of semaphores Locks  import java.util.concurrent.\*;    //A shared resource/class.  class Shared  {      static int count = 0;  }    class MyThread extends Thread  {      Semaphore sem;      String threadName;      public MyThread(Semaphore sem, String threadName)      {          super(threadName);          this.sem = sem;          this.threadName = threadName;      }        @Override      public void run() {            // run by thread A          if(this.getName().equals("A"))          {              System.out.println("Starting " + threadName);              try              {                  // First, get a permit.                  System.out.println(threadName + " is waiting for a permit.");                    // acquiring the lock                  sem.acquire();                    System.out.println(threadName + " gets a permit.");                    // Now, accessing the shared resource.                  // other waiting threads will wait, until this                  // thread release the lock                  for(int i=0; i < 5; i++)                  {                      Shared.count++;                      System.out.println(threadName + ": " + Shared.count);                        // Now, allowing a context switch -- if possible.                      // for thread B to execute                      Thread.sleep(10);                  }              } catch (InterruptedException exc) {                      System.out.println(exc);                  }                    // Release the permit.                  System.out.println(threadName + " releases the permit.");                  sem.release();          }            // run by thread B          else          {              System.out.println("Starting " + threadName);              try              {                  // First, get a permit.                  System.out.println(threadName + " is waiting for a permit.");                    // acquiring the lock                  sem.acquire();                    System.out.println(threadName + " gets a permit.");                    // Now, accessing the shared resource.                  // other waiting threads will wait, until this                  // thread release the lock                  for(int i=0; i < 5; i++)                  {                      Shared.count--;                      System.out.println(threadName + ": " + Shared.count);                        // Now, allowing a context switch -- if possible.                      // for thread A to execute                      Thread.sleep(10);                  }              } catch (InterruptedException exc) {                      System.out.println(exc);                  }                  // Release the permit.                  System.out.println(threadName + " releases the permit.");                  sem.release();          }      }  }    // Driver class  public class SemaphoreDemo  {      public static void main(String args[]) throws InterruptedException      {          // creating a Semaphore object          // with number of permits 1          Semaphore sem = new Semaphore(1);            // creating two threads with name A and B          // Note that thread A will increment the count          // and thread B will decrement the count          MyThread mt1 = new MyThread(sem, "A");          MyThread mt2 = new MyThread(sem, "B");            // stating threads A and B          mt1.start();          mt2.start();            // waiting for threads A and B          mt1.join();          mt2.join();            // count will always remain 0 after          // both threads will complete their execution          System.out.println("count: " + Shared.count);      }  } |

Run on IDE

Output:

Starting A

Starting B

B is waiting for a permit.

B gets a permit.

A is waiting for a permit.

B: -1

B: -2

B: -3

B: -4

B: -5

B releases the permit.

A gets a permit.

A: -4

A: -3

A: -2

A: -1

A: 0

A releases the permit.

count: 0

**Note :**The output can be different in different executions of above program, but final value of *count*variable will always remain 0.

**Explanation of above program :**

* The program uses a semaphore to control access to the *count* variable, which is a static variable within the Shared class. *Shared.count* is incremented five times by thread A and decremented five times by thread B.To prevent these two threads from accessing Shared.count at the same time, access is allowed only after a permit is acquired from the controlling semaphore. After access is complete, the permit is released. In this way, only one thread at a time will access Shared.count, as the output shows.
* Notice the call to sleep( ) within run( ) method inside MyThread class. It is used to “prove” that accesses to Shared.count are synchronized by the semaphore. In run( ), the call to sleep( ) causes the invoking thread to pause between each access to Shared.count. This would normally enable the second thread to run. However, because of the semaphore, the second thread must wait until the first has released the permit, which happens only after all accesses by the first thread are complete. Thus, Shared.count is first incremented five times by thread A and then decremented five times by thread B. The increments and decrements are not intermixed at assembly code.
* Without the use of the semaphore, accesses to *Shared.count* by both threads would have occurred simultaneously, and the increments and decrements would be intermixed.To confirm this, try commenting out the calls to *acquire( )* and *release( )*. When you run the program, you will see that access to Shared.count is no longer synchronized, thus you will not always get *count* value 0.

**Next:**[Java.util.concurrent.Semaphore class in Java](http://www.geeksforgeeks.org/java-util-concurrent-semaphore-class-java/)