**Java Semaphore**

**semaphore**  is used to control access to a shared resource that uses a counter variable.

A [Semaphore](https://www.javatpoint.com/os-semaphore-introduction) is used to limit the number of threads that want to access a shared resource. In other words, it is a non-negative variable that is shared among the threads known as a **counter**. It sets the limit of the threads. A mechanism in which a thread is waiting on a **semaphore** can be signaled by other threads.

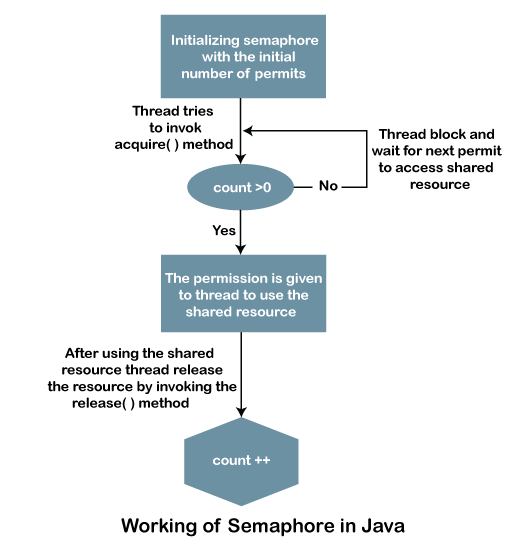
* If **counter > 0**, access to shared resources is provided.
* If **counter = 0**, access to shared resources is denied.

semaphore grants permission to threads to share a resource.

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.



## **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 semaphore](https://www.javatpoint.com/os-counting-semaphore)s are used to resolve the situation in which more than one process wants to execute in the critical section, simultaneously.

**public** **class** CountingSemaphoreExample

{

**private** **int** counter = 0;

**public** **synchronized** **void** take()

{

**this**.counter++;

**this**.notify();

}

**public** **synchronized** **void** release() **throws** InterruptedException

{

**while**(**this**.counter == 0)

wait();

**this**.counter--;

}

}

### **Bounded Semaphores**

We can set the upper bound limit using the **bounded semaphores**.

**public** **class** BoundedSemaphoresExample

{

**private** **int** counter = 0;

**private** **int** bound = 0;

**public** BoundedSemaphoreexample(**int** upperBound)

{

**this**.bound = upperBound;

}

**public** **void** **synchronized** take() **throws** InterruptedException

{

**while**(**this**.counter == bound)

wait();

**this**.counter++;

**this**.notify();

}

**public** **void** **synchronized** release() **throws** InterruptedException

{

**while**(**this**.counter == 0)

wait();

**this**.counter--;

}

}

**Semaphore in Java:**

Java provides a **Semaphore** class to implement the semaphore mechanism. It belongs to the **java.util.concurrent** package. It implements the **Serializable** interface.

The Semaphore class provides the following **two constructors:**

* Semaphore(int permits)
* Semaphore(int permits, boolean fair)

### **Semaphore(int permits)**

It creates a Semaphore and parses the number of permits (initial number of permits available) as an argument. It specifies the number of threads that can share a resource at a time. The value of permits may be negative. In such a case, a release must occur before any acquires will be granted.

**Syntax:**

1. **public** Semaphore(**int** permits)

### **Semaphore(int permits, boolean fair)**

It creates a Semaphore with the given number of permits and the given fairness settings.

1. **public** Semaphore(**int** permits, **boolean** fair)

It parses two parameters:

* **permits:** The value of permits may be **negative**. In such a case, the release must occur before any acquires will be granted.
* **fair:** If we set the value to **true**, the semaphore guarantees FIFO to the threads in the order they are requested, **false** By default, all the threads that are waiting for the resource grants permit in an **undefined** order.

### **Methods of the Semaphore Class**

The class provides the following methods:

**acquire() Method:** The method acquire the permits from the semaphore, blocking until one is available, or the thread is interrupted. It reduces the number of available permits by 1.

If there is no permit is available for the current thread, the thread becomes disabled for the thread scheduling purposes. The current thread goes in the inactive state until one of two things happens:

* If the other thread invokes the **release()** method to release the resource then the current thread gets permits.
* If the other thread interrupts the current thread.

It throws **InterruptedException** if the current thread is interrupted. The method does not return any value.

**Syntax:**

1. **public** **void** acquire() **throws** InterruptedException

**release() Method:** It releases a permit and returns it to the semaphore. It increments the number of available permits by 1. If a thread tries to acquire a permit, the semaphore grants permission to acquire the resource that was just released by other threads. Further, that thread is considered for thread scheduling purposes.

**Syntax:**

1. **public** **void** release()

**availablePermits() Method:** The method returns the number of permits available in semaphore for granting the resource. Usually, it is used for debugging and testing purposes.

**Syntax:**

1. **public** **int** availablePermits()

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.

**import** java.util.concurrent.\*;

//a shared resource or class.

**class** Shared

{

**static** **int** count = 0;

}

**class** SemaphoreExample **extends** Thread

{

Semaphore semaphore;    //object of the Semaphore class

String threadName;   //variable for storing the thread name

**public** SemaphoreExample (Semaphore semaphore, String threadName)

{

**super**(threadName);

**this**.semaphore = semaphore;

**this**.threadName = threadName;

}

@Override

**public** **void** run()

{

// run by thread A

**if**(**this**.getName().equals("A"))

{

System.out.println("Starting thread " + threadName);

**try**

{

System.out.println(threadName + " is waiting for a permit.");

semaphore.acquire();   //acquiring the lock

System.out.println(threadName + " gets a permit.");

**for**(**int** i=0; i < 5; i++)

{

Shared.count++;

System.out.println(threadName + ": " + Shared.count);

Thread.sleep(1000);

}

}

**catch** (InterruptedException e)

{

e.printStackTrace();

}

//release the permit

System.out.println("Thread " +threadName + " releases the permit.");

semaphore.release();

}

// run by thread B

**else**

{

System.out.println("Starting thread " + threadName);

**try**

{

System.out.println("Thread " +threadName + " is waiting for a permit.");

semaphore.acquire();   //acquiring the lock

System.out.println("Thread " +threadName + " gets a permit.");

**for**(**int** i=0; i < 5; i++)

{

Shared.count--;   //decrements the count value

System.out.println(threadName + ": " + Shared.count);

Thread.sleep(1000);

}

}

**catch** (InterruptedException e)

{

e.printStackTrace();

}

//release the permit

System.out.println("Thread " +threadName + " releases the permit.");

//release the resource

semaphore.release();

}

}

}

**public** **class** SemaphoreAsLock

{

**public** **static** **void** main(String args[]) **throws** InterruptedException

{

//creating the constructor of the Semaphore with initial permit 1

Semaphore semaphore = **new** Semaphore(1);

SemaphoreExample dt1 = **new** SemaphoreExample (semaphore, "A");

SemaphoreExample dt2 = **new** SemaphoreExample (semaphore, "B");

dt1.start();

dt2.start();

dt1.join();

dt2.join();

System.out.println("count: " + Shared.count);

}

}

Output:

Starting thread B

Starting thread A

A is waiting for a permit.

A gets a permit.

A: 1

A: 2

A: 3

A: 4

A: 5

Thread A releases the permit.

Thread B gets a permit.

B: 4

B: 3

B: 2

B: 1

B: 0

Thread B releases the permit.

count: 0

# Producer-Consumer solution using Semaphores

|  |
| --- |
| package PCSemaphore;  import java.util.concurrent.Semaphore;  class Q  {  int item;  // semCon initialized with 0 permits to ensure put() executes first  static Semaphore semCon = new Semaphore(0);  static Semaphore semProd = new Semaphore(1);  void get()  {  try  {  // Before consumer can consume an item, it must acquire a permit from semCon  semCon.acquire();  }  catch (InterruptedException e)  {  System.out.println("InterruptedException caught");  }    // consumer consuming an item  System.out.println("Consumer consumed item : " + item);    // After consumer consumes the item, it releases semProd to notify producer  semProd.release();  }    void put(int item)  {  try  {  // Before producer can produce an item, it must acquire a permit from semProd  semProd.acquire();  }  catch (InterruptedException e)  {  System.out.println("InterruptedException caught");  }    // producer producing an item  this.item = item;    System.out.println("Producer produced item : " + item);    // After producer produces the item, it releases semCon to notify consumer  semCon.release();  }  }    // Producer class  class Producer implements Runnable  {      Q q;      Producer(Q q)      {          this.q = q;          new Thread(this, "Producer").start();      }        public void run()      {          for (int i = 0; i < 5; i++)              // producer put items              q.put(i);      }  }    // Consumer class  class Consumer implements Runnable  {      Q q;      Consumer(Q q)      {          this.q = q;          new Thread(this, "Consumer").start();      }        public void run()      {          for (int i = 0; i < 5; i++)              // consumer get items              q.get();      }  }    class PC {      public static void main(String args[])      {          // creating buffer queue          Q q = new Q();            // starting consumer thread          new Consumer(q);            // starting producer thread          new Producer(q);      }  } |

Output:

Producer produced item : 0

Consumer consumed item : 0

Producer produced item : 1

Consumer consumed item : 1

Producer produced item : 2

Consumer consumed item : 2

Producer produced item : 3

Consumer consumed item : 3

Producer produced item : 4

Consumer consumed item : 4

**The sequencing of put() and get() calls is handled by two semaphores:** **semProd** and **semCon**.

* Before put( ) can produce an item, it must acquire a permit from semProd. After it has produce the item, it releases semCon.
* Before get( ) can consume an item, it must acquire a permit from semCon. After it consumes the item, it releases semProd.
* This “give and take” mechanism ensures that each call to put( ) must be followed by a call to get( ).
* Also notice that semCon is initialized with no available permits. This ensures that put( ) executes first. The ability to set the initial synchronization state is one of the more powerful aspects of a semaphore.