**Semaphore in Java-2024**

* **A semaphore is a way to limit the number of tasks/threads that can simultaneously operate on a shared (protected) resource**. It maintains a counter and counter is decremented each time a thread completes the task.
* **If the semaphore’s count is greater than zero, then the thread acquires a permit**, which causes the semaphore’s count to be decremented.
* **One interesting property of Semaphores in Java is that release doesn’t have to be called by the same thread as acquire.** **This is a useful property that we don’t have with normal mutexes in Java.**
* **Another trick is to increase the number of permits at runtime.**

**There are two type of semaphore**. **Java's semaphore is a counting semaphore**

1. Counting semaphore (Values varies from 0 to N)

2. Binary semaphore (Values either 0  or  1)

**Use Cases for Semaphore**

* There is only one Bank ATM machine which can be used by only one person to draw money.
* In airport there only two wash rooms which can be used by two persons at a time.
* In a library, there are 10 identical study rooms. When a student has finished using a room, the student must return to the counter and indicate that one room has become free. The clerk at the front desk does not keep track of which room is occupied, only the number of free rooms available. **In this scenario the front desk represents a semaphore**, **the rooms are the resources**, and the **students represent processes/threads.**

**Uses**

What are some possible uses for counting semaphores? The following come to mind:

* **JDBC connection pooling / limiting**
* **Network connection throttling**
* **Throttling CPU or memory intensive tasks**

**Mutex(Mutual Exclusion)**

* It is basically mutual exclusion. In other words, only one thread is allowed to acquire the resource at a time. When one thread acquires the resource, no other thread is allowed to acquire the resource. **It acts as a lock.**
* **No two processes/threads can acquire the critical section at a time.**
* Only one thread owns the mutex at a time, thus a mutex with a unique name is created when a program starts.
* **In probability theory, two events are said to be mutually exclusive if they cannot occur at the same time or simultaneously.**
* **A Mutex is a mutually exclusive flag. It acts as a gate keeper to a section of code allowing one thread in and blocking access to all others**.
* **MUTEX is a kind of lock which locks/allows one thread at a time. If another thread wants to lock it, the thread simply gets blocked.**

**Example:**

* I use a rubber chicken or microphone which I keep in my desk for just such occasions. The person holding the chicken/microphone is the only person who is allowed to talk. If you don't hold the chicken you cannot speak.

**Difference between Mutex and Semaphore**

* **Mutex: Provides mutual exclusion, one thread at a time in the critical section.**
* **Semaphore: Allows multiple threads to access the critical section simultaneously.**

**Hence A semaphore can be a Mutex but a Mutex can never be semaphore.**

**How do you implement Mutex in Java**

1. **Using Intrinsic Lock (synchronized block)**
2. **Using ReentrantLock**
3. **Using Counting Semaphore**
4. **Using Google’s Guava Monitor.**

**The code is given below.**

**public class** TestMutex {  
  
 **private int** currentValue = 0;  
 **private Lock lock = new ReentrantLock(true);**  
 **private Semaphore semaphore = new Semaphore(1, true);**  
 **private Monitor mutex = new Monitor(); 🡸 Google Guava library**  
 **public int** getSequenceValue() {  
 currentValue = currentValue + 1;  
 **return** currentValue;  
 }  
  
 // **Mutex using ReentrantLock**  
 **public int** getNextSequence1() {  
 **synchronized** (**this**) {  
 **return** getSequenceValue();  
 }  
 }  
  
 // **Mutex using ReentrantLock**  
 **public int** getNextSequence2() {  
 lock.lock();  
 **try** {  
 **return** getSequenceValue();  
 } **finally** {  
 lock.unlock();  
 }  
 }  
  
 // **Mutex using Semaphore**  
 **public int** getNextSequence3() {  
 **int** val = 0;  
 **try** {  
 semaphore.acquire();  
 **return** getSequenceValue();  
 } **catch** (InterruptedException ie) {  
 ie.printStackTrace();  
 } **finally** {  
 semaphore.release();  
 }  
 **return** val;  
 }  
  
 // **Mutex using Google Guava Mutex**  
 **public int** getNextSequence4() {  
 **try** {  
 mutex.enter();  
 **return** getSequenceValue();  
 } **finally** {  
 mutex.leave();  
 }  
 }

**public void** m1() {  
 **int** seqValue = getNextSequence1();  
 String name = Thread.*currentThread*().getName();  
 System.***out***.println(name + " got Next Seq Value: " + seqValue);  
 }  
  
 **public void** m2() {  
 **int** seqValue = getNextSequence2();  
 String name = Thread.*currentThread*().getName();  
 System.***out***.println(name + " got Next Seq Value: " + seqValue);  
 }  
  
 **public void** m3() {  
 **int** seqValue = getNextSequence3();  
 String name = Thread.*currentThread*().getName();  
 System.***out***.println(name + " got Next Seq Value: " + seqValue);  
 }  
  
 **public void** m4() {  
 **int** seqValue = getNextSequence4();  
 String name = Thread.*currentThread*().getName();  
 System.***out***.println(name + " got Next Seq Value: " + seqValue);  
 }  
  
 **public void** check() {  
 **for** (**int** i = 0; i < 5; i++) {  
// new Thread(() -> m1()).start();  
// new Thread(() -> m2()).start();  
// new Thread(() -> m3()).start();  
 **new** Thread(() -> m4()).start();  
 }  
 }  
  
 **public static void** main(String[] args) {  
 **new** TestMutex().check();  
 }  
}

**Basic Examples on Semaphore**

**Problem: I want an ATM machine to be used only by maximum 3 users at a time.**

**public class** BankATM {  
 **private** Semaphore semaphore = **new** Semaphore(2);  
  
 **public int** withdrawAmount(String name, **int** amount) {  
 **int** withdrawalAmt = 0;  
 Thread.*currentThread*().setName(name);  
 **try** {  
 **semaphore.acquire();** System.***out***.println(Thread.*currentThread*().getName() + " trying to withdraw " + amount);  
 TimeUnit.***SECONDS***.sleep(5);  
 System.***out***.println(Thread.*currentThread*().getName() + " entering ATM PIN");  
 System.***out***.println("Amount of Rs "+amount+" is getting dispense for "+Thread.*currentThread*().getName());  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 } **finally** {  
 **semaphore.release();** }  
 **return** withdrawalAmt;  
 }  
}

**public class** TestBankATM {  
  
 **public void** drawAmount(BankATM bankATM, String name, **int** amount) {  
 **int** value = bankATM.withdrawAmount(name, amount);  
 System.***out***.println(Thread.*currentThread*().getName() + " got cash of Rs " + amount);  
 }  
  
 **public void** check() {  
 BankATM bankATM = **new** BankATM();  
  
 **new** Thread(() -> drawAmount(bankATM, "John", 2000)).start();  
 **new** Thread(() -> drawAmount(bankATM, "Raveena", 2000)).start();  
 **new** Thread(() -> drawAmount(bankATM, "Swati", 2000)).start();  
 }  
  
 **public static void** main(String[] args) {  
 **new** TestBankATM().check();  
 }  
}

**OUTPUT**

John trying to withdraw 2000

Raveena trying to withdraw 2000

Raveena entering ATM PIN

John entering ATM PIN

Amount of Rs 2000 is getting dispense for John

Amount of Rs 2000 is getting dispense for Raveena

Swati trying to withdraw 2000

Raveena got cash of Rs 2000

John got cash of Rs 2000

Swati entering ATM PIN

Amount of Rs 2000 is getting dispense for Swati

Swati got cash of Rs 2000

**Let us consider a real time example where we will use Semaphore. In an Airport there is only one rest room and that rest room contains only two lavatories or toilets. There are n number of passengers who want to use the rest room. At a any point of time two passengers can use the rest room. If one passenger leaves the room, next one will occupy the room. In this situation, semaphore plays the significant role as it allows the number of permits ie 2.**

Let us see the code below.

**public class** TestAirportWashRoom {  
  
 **public void** useWashRoom(Semaphore semaphore, String name) {  
 **try** {  
 **semaphore.acquire();** System.***out***.println(name + " occupied the rest room ...");  
 TimeUnit.***SECONDS***.sleep(2);  
  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 } **finally** {  
 **semaphore.release();** }  
 System.***out***.println(name + " left the rest room ...");  
 }

**public void** check() {  
 Semaphore sema = **new** Semaphore(2);  
 **for** (**int** i = 1; i < 6; i++) {  
 String name = "P-"+i;  
 **new** Thread( () -> useWashRoom(sema, name)).start();  
 }  
 }  
  
 **public static void** main(String[] args) {  
 **new** TestAirportWashRoom().check();  
 }  
}

OUTPUT

P-2 occupied the rest room ...

P-1 occupied the rest room ...

P-3 occupied the rest room ...

P-1 left the rest room ...

P-5 occupied the rest room ...

P-2 left the rest room ...

P-3 left the rest room ...

P-4 occupied the rest room ...

P-5 left the rest room ...

P-4 left the rest room ...

**The constructor of semaphore optionally accepts a fairness parameter. When set false, this class makes no guarantees about the order in which threads acquire permits**. **When fairness is set true, the semaphore guarantees that threads invoking any of the acquire methods are selected to obtain permits in the order in which their invocation of those methods was processed (first-in-first-out; FIFO).**