**Semaphore in Java**

A semaphore controls access to a shared resource through the use of a counter.

**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.

**Semaphore vs. Mutex**

Mutex acts similarly to a binary semaphore, we can use it to implement mutual exclusion.

In the following example, we’ll use a simple binary semaphore to build a counter:

class CounterUsingMutex {

    private Semaphore mutex;

    private int count;

    CounterUsingMutex() {

        mutex = new Semaphore(1);

        count = 0;

    }

    void increase() throws InterruptedException {

        mutex.acquire();

        this.count = this.count + 1;

        Thread.sleep(1000);

        mutex.release();

    }

    int getCount() {

        return this.count;

    }

    boolean hasQueuedThreads() {

        return mutex.hasQueuedThreads();

    }

}

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

**import** java.util.concurrent.Semaphore;  
**import** java.util.concurrent.TimeUnit;  
**public class BankATM** {  
 **private** Semaphore **semaphore** = **new** Semaphore(3);  
 **private** CashDispenser **cashDispenser** = **new** CashDispenser();  
  
 **public int** withdrawAmount(**int** amount) {  
 **int** withdrawalAmt = 0;  
 **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"**);  
 withdrawalAmt = **cashDispenser**.getDispensedCash(amount);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 } **finally** {  
 **semaphore**.release();  
 }  
 **return** withdrawalAmt;  
 }  
}

**import** java.util.concurrent.TimeUnit;  
**import** java.util.concurrent.locks.Lock;  
**import** java.util.concurrent.locks.ReentrantLock;  
  
**public class CashDispenser** {  
 **private** Lock **lock** = **new** ReentrantLock();  
  
 **public int** getDispensedCash(**int** amount) **throws** InterruptedException {  
 **lock.lock();**  
 System.***out***.println(**"-------- Cash Dispenser --------"**);  
 System.***out***.println(**"Cash is getting dispensed for "** + Thread.*currentThread*().getName());  
 TimeUnit.***SECONDS***.sleep(3);  
 **lock.unlock();**  
 System.***out***.println(**"-------- Cash Dispenser --------"**);  
 **return** amount;  
 }  
}

**public class RunnableCustomer** **implements** Runnable {  
 **private** BankATM **bankATM**;  
 **private int amount**;  
  
 **public** RunnableCustomer(BankATM bankATM, **int** amount) {  
 **this**.**bankATM** = bankATM;  
 **this**.**amount** = amount;  
 }  
  
 @Override  
 **public void** run() {  
 **int** value = **bankATM**.withdrawAmount(**amount**);  
 System.***out***.println(Thread.*currentThread*().getName() + **" got cash of Rs "** + **amount**);  
 }  
}

**public class TestSemaphore** {  
 **public static void** main(String[] args) {  
 BankATM bankATM = **new** BankATM();  
 Thread t1 = **new** Thread(**new** RunnableCustomer(bankATM, 2000), **"John"**);  
 Thread t2 = **new** Thread(**new** RunnableCustomer(bankATM, 12000), **"Rama"**);  
 Thread t3 = **new** Thread(**new** RunnableCustomer(bankATM, 3000), **"Vidya"**);  
 Thread t4 = **new** Thread(**new** RunnableCustomer(bankATM, 500), **"Puja"**);  
 Thread t5 = **new** Thread(**new** RunnableCustomer(bankATM, 7000), **"Krishna"**);  
 Thread t6 = **new** Thread(**new** RunnableCustomer(bankATM, 9000), **"Shiva"**);  
  
 t1.start();  
 t2.start();  
 t3.start();  
 t4.start();  
 t5.start();  
 t6.start();  
 }  
}

OUTPUT

John trying to withdraw 2000

Rama trying to withdraw 12000

Vidya trying to withdraw 3000

🡸 It blocks other threads, so we have given a blank space.

John entering ATM PIN

Rama entering ATM PIN

Vidya entering ATM PIN

-------- Cash Dispenser --------

Cash is getting dispensed for John

-------- Cash Dispenser --------

-------- Cash Dispenser --------

Cash is getting dispensed for Rama

John got cash of Rs 2000

Puja trying to withdraw 500

-------- Cash Dispenser --------

-------- Cash Dispenser --------

Cash is getting dispensed for Vidya

Rama got cash of Rs 12000

Krishna trying to withdraw 7000

Puja entering ATM PIN

-------- Cash Dispenser --------

-------- Cash Dispenser --------

Cash is getting dispensed for Puja

Shiva trying to withdraw 9000

Vidya got cash of Rs 3000

Krishna entering ATM PIN

-------- Cash Dispenser --------

-------- Cash Dispenser --------

Cash is getting dispensed for Krishna

Puja got cash of Rs 500

Shiva entering ATM PIN

-------- Cash Dispenser --------

-------- Cash Dispenser --------

Krishna got cash of Rs 7000

Cash is getting dispensed for Shiva

-------- Cash Dispenser --------

Shiva got cash of Rs 9000

**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 Passenger extends Thread {  
 private Semaphore sema;  
  
 public Passenger(Semaphore sema, String name) {  
 this.sema = sema;  
 setName(name);  
 }  
  
 @Override  
 public void run() {  
 try {  
 **sema.acquire();**  
 System.*out*.println(getName() + " occupied the rest room ...");  
 TimeUnit.*SECONDS*.sleep(4);  
  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 **sema.release();** }  
 System.*out*.println(getName() + " left the rest room ...");  
 }  
}

import java.util.concurrent.Semaphore;  
public class Test1 {  
 public static void main(String[] args) {  
 Semaphore sema = new Semaphore(2);  
 for (int i = 1; i < 6; i++) {  
 Passenger passenger = new Passenger(sema, "P-" + i);  
 passenger.start();  
 }  
 }  
}

OUTPUT

P-1 occupied the rest room ...

P-2 occupied the rest room ...

P-1 left the rest room ...

P-5 occupied the rest room ...

P-2 left the rest room ...

P-4 occupied the rest room ...

P-4 left the rest room ...

P-5 left the rest room ...

P-3 occupied the rest room ...

P-3 left the rest room ...

In this similar line you can think about another situation in a bank, there are two teller counters and there are 5 customers and at a time only two customer can go to the teller counter.

Similarly semaphore can be used as an example in Rest Room in Airport and can be used in ATM machines.

**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).** Note that FIFO ordering necessarily applies to specific internal points of execution within these methods. So, it is possible for one thread to invoke acquire before another, but reach the ordering point after the other, and similarly upon return from the method. Also note that the untimed **tryAcquire() methods do not honor the fairness** setting, but will take any permits that are available.

**Semaphore and Mutex(Mutual Exclusion) and Other Thread Concepts-2021-2022**

**A semaphore is a way to limit the number of tasks that can simultaneously operate on a shared (protected) resource**. A protected resource or critical section of code might include writing to global variables or communicating with external instruments. It is useful for protecting two or more critical sections of code that should not be called concurrently. Before entering a critical section, the thread must acquire a semaphore. This thread must release the semaphore once the critical section is complete. Other threads that want to enter the critical section must wait until the first thread releases the semaphore.

Library analogy

Suppose a library has 10 identical study rooms, intended to be used by one student at a time. To prevent disputes, students must request a room from the front counter if they wish to make use of a study room. When a student has finished using a room, the student must return to the counter and indicate that one room has become free. If no rooms are free, students wait at the counter until someone relinquishes a room. The clerk at the front desk does not keep track of which room is occupied, only the number of free rooms available. When a student requests a room, the clerk decreases this number. When a student releases a room, the clerk increases this number. Once access to a room is granted, the room can be used for as long as desired, and so it is not possible to book rooms ahead of time. **In this scenario the front desk represents a semaphore**, **the rooms are the resources**, and the **students represent processes**. **The value of the semaphore in this scenario is initially 10**. **When a student requests a room he or she is granted access and the value of the semaphore is changed to 9**. After the next student comes, it drops to 8, then 7 and so on. If someone requests a room and the resulting value of the semaphore is negative,[2] they are forced to wait. When multiple people are waiting, they will either wait in a queue, or use Round-robin scheduling and race back to the counter when someone releases a room (depending on the nature of the semaphore).

**Important observations**

**When used for a pool of resources, a semaphore does not keep track of which of the resources are free, only how many there are**.

http://www.javacodegeeks.com/2011/09/java-concurrency-tutorial-semaphores.html

**So what is semaphore? The simplest way to think of a semaphore is to consider it an abstraction that allows n units to be acquired, and offers acquire and release mechanisms.**

**Note: A semaphore can be a Mutex but a Mutex can never be semaphore.**

**Semaphores are often used to restrict the number of threads than can access some (physical or logical) resource.**

A semaphore will either allow or disallow access to the resource, depending on how it is set up.

**A semaphore object is a synchronization object that maintains a count between zero and a specified maximum value. The count is decremented each time a thread completes a wait for the semaphore object and incremented each time a thread releases the semaphore.** When the count reaches zero, no more threads can successfully wait for the semaphore object state to become signaled. The semaphore object is useful in controlling a shared resource that can support a limited number of users. It acts as a gate that limits the number of threads sharing the resource to a specified maximum number.

**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.**

Finally, there are a few useful methods to be familiar with in Java’s Semaphore. The method acquireInterruptibly() will acquire a resource, reattempting if it is interrupted. This means no outside handling of InterruptedException. The method tryAcquire() allows us to limit how long we will wait for a permit – we can either return immediately if there is no permit to obtain, or wait a specified timeout. If you somehow have known deadlocks that you can’t fix easily or track down, you could help prevent locking up processes by using **tryAcquire()** with suitable timeouts.

**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**

**Mutual Exclusion**

**A way of making sure that if one process is using a shared modifiable data, the other processes will be excluded from doing the same thing. No two processes may at the same moment inside their critical sections.**

A mutual exclusion (mutex) is a program object that prevents simultaneous access to a shared resource. This concept is used in concurrent programming with a critical section, a piece of code in which processes or threads access a shared resource. Only one thread owns the mutex at a time, thus a mutex with a unique name is created when a program starts. When a thread holds a resource, it has to lock the mutex from other threads to prevent concurrent access of the resource. Upon releasing the resource, the thread unlocks the mutex. **Mutex comes into the picture when two threads work on the same data at the same time. It acts as a lock and is the most basic synchronization tool**. When a thread tries to acquire a mutex, it gains the mutex if it is available, otherwise the thread is set to sleep condition. Mutual exclusion reduces latency and busy-waits using queuing and context switches. Mutex can be enforced at both the hardware and software levels. Disabling interrupts for the smallest number of instructions is the best way to enforce mutex at the kernel level and prevent the corruption of shared data structures. If multiple processors share the same memory, a flag is set to enable and disable the resource acquisition based on availability. The busy-wait mechanism enforces mutex in the software areas. This is furnished with algorithms such as Dekker's algorithm, the black-white bakery algorithm, Szymanski's algorithm, Peterson's algorithm and Lamport's bakery algorithm.

A way of making sure that if one process is using a shared modifiable data, the other processes will be excluded from doing the same thing. Formally, while one process executes the shared variable, all other processes desiring to do so at the same time moment should be kept waiting; when that process has finished executing the shared variable, one of the processes waiting; while that process has finished executing the shared variable, one of the processes waiting to do so should be allowed to proceed. In this fashion, each process executing the shared data (variables) excludes all others from doing so simultaneously. This is called Mutual Exclusion. Note that mutual exclusion needs to be enforced only when processes access shared modifiable data - when processes are performing operations that do not conflict with one another they should be allowed to proceed concurrently. Mutual Exclusion Conditions If we could arrange matters such that no two processes were ever in their critical sections simultaneously, we could avoid race conditions. We need four conditions to hold to have a good solution for the critical section problem (mutual exclusion). No two processes may at the same moment inside their critical sections. No assumptions are made about relative speeds of processes or number of CPUs. No process should outside its critical section should block other processes. No process should wait arbitrary long to enter its critical section. "mutex" redirects here

**Mutex – Mutually Exclusive**

**In probability theory, two events are said to be mutually exclusive if they cannot occur at the same time or simultaneously.**

https://stackoverflow.com/questions/34524/what-is-a-mutex

**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**. This ensures that the code being controlled will only be hit by a single thread at a time. Just be sure to release the mutex when you are done.When I am having a big heated discussion at work, I use a rubber chicken which I keep in my desk for just such occasions. The person holding the chicken is the only person who is allowed to talk. If you don't hold the chicken you cannot speak. You can only indicate that you want the chicken and wait until you get it before you speak. Once you have finished speaking, you can hand the chicken back to the moderator who will hand it to the next person to speak. This ensures that people do not speak over each other, and also have their own space to talk. **Replace Chicken with Mutex** and **person with thread** and you basically have the concept of a mutex.

The chicken is the *mutex*. People hoilding the mu.. chicken are *competing threads*. The Moderator is the *OS*. When people requests the chicken, they do a lock request. When you call mutex.lock(), your thread stalls in lock() and makes a lock request to the OS. When the OS detects that the mutex was released from a thread, it merely gives it to you, and lock() returns - the mutex is now yours and only yours. Nobody else can steal it, because calling lock() will block him. There is also try\_lock() that will block and return true when mutex is yours and immediately false if mutex is in use. **Instead of Rubber Chicken, you can use Microphone to Speak**.That is a thread must acquire a lock before entering into a critical section (In critical section multi threads share a common variable, updating a table, writing a file and so on), it releases the lock when it leaves critical section.

Consider **single** toilet with a *key*. When someone enters, they take the key and the toilet is *occupied*. If someone else needs to use the toilet, they need to wait in a *queue*. When the person in the toilet is *done*, they pass the key to the next person in queue. Make sense, right?

Convert the *toilet* in the story to a *shared resource*, and the *key* to a *mutex*. Taking the key to the toilet (acquire a lock) permits you to use it. If there is no key (the lock is locked) you have to wait. When the key is returned by the person (*release the lock*) you're free to acquire it now. **MUTEX is a kind of lock which locks one thread at a time. If another thread wants to lock it, the thread simply gets blocked.**

**Mutex**:- 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 until the thread owning the resource releases. All threads waiting for acquiring resource would be blocked.

**Difference between Mutex and Semaphore**

* A mutex object allows multiple process threads to access a single shared resource but only one at a time. On the other hand, **semaphore allows multiple process threads to access the finite instance of the resource until available**. In mutex, the lock can be acquired and released by the same process at a time.
* A mutex is an object but semaphore is an integer variable.
* In mutex, the lock can be acquired and released by the same process at a time. But the value of the semaphore variable can be modified by any process that needs some resource but only one process can change the value at a time.
* **Semaphore** is simply a variable that is non-negative and shared between threads. A semaphore is a signaling mechanism, and a thread that is waiting on a semaphore can be signaled by another thread. It uses two atomic operations, 1)wait, and 2) signal for the process synchronization.

**There are two type of semaphore**

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

2. Binary semaphore (Values either 0  or  1)

***Note*:- Java's semaphore is a counting semaphore.**

At high level, binary semaphore is similar to Mutex. Binary semaphore can act as Mutex. However, there is fundamental difference between Mutex and Semaphore :-  the concept of "**ownership**".

Semaphores have no notion of ownership, this means that any thread can release a semaphore, Whereas a mutex does have the concept of ownership The process that locked the mutex is supposed to unlock it.

Mutex primary use is **to guard shared resources** in multi-threaded environment.

**When do we use semaphore and Mutex** :- The usefulness of Mutex and Semaphore lies in - how many threads are allowed to acquire the resource at once ?

**If answer is one - Mutex is suitable candidate.**  
**Otherwise Semaphore as it allows multiple threads(equal to number of permitted semaphore values) to access shared resources.**

**What is a race condition**

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.

In order to prevent race conditions from occurring, you would typically put a lock around the shared data to ensure only one thread can access the data at a time.