<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html>

#### acquire

* public void acquire()

throws [InterruptedException](https://docs.oracle.com/javase/7/docs/api/java/lang/InterruptedException.html)

Acquires a permit from this semaphore, blocking until one is available, or the thread is [interrupted](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#interrupt()).

Acquires a permit, if one is available and returns immediately, 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()](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html#release()) method for this semaphore and the current thread is next to be assigned a permit; or
  + Some other thread [interrupts](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#interrupt()) the current thread.

If the current thread:

* + has its interrupted status set on entry to this method; or
  + is [interrupted](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#interrupt()) while waiting for a permit,

then [InterruptedException](https://docs.oracle.com/javase/7/docs/api/java/lang/InterruptedException.html) is thrown and the current thread's interrupted status is cleared.

**Throws:**

[InterruptedException](https://docs.oracle.com/javase/7/docs/api/java/lang/InterruptedException.html) - if the current thread is interrupted

* **tryAcquire**

public boolean tryAcquire()

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, with the value true, reducing the number of available permits by one.

If no permit is available then this method will return immediately with the value false.

Even when this semaphore has been set to use a fair ordering policy, a call to tryAcquire() *will* immediately acquire a permit if one is available, whether or not other threads are currently waiting. This "barging" behavior can be useful in certain circumstances, even though it breaks fairness. If you want to honor the fairness setting, then use [tryAcquire(0, TimeUnit.SECONDS)](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html#tryAcquire(long,%20java.util.concurrent.TimeUnit))which is almost equivalent (it also detects interruption).

**Returns:**

true if a permit was acquired and false otherwise

<https://stackoverflow.com/questions/36992758/java-semaphore-release-without-acquire>

# [**Java - Semaphore release without acquire**](https://stackoverflow.com/questions/36992758/java-semaphore-release-without-acquire)

Q:

I have threads which are given random number (1 to n) and are instructed to print them in sorted order. I used semaphore such that I acquire the number of permits = random number and release one permit more than what was acquired.

acquired = random number; released = 1+random number

Initial permit count for semaphore is 1. So thread with random number 1 should get permit and then 2 and so on.

This is supported as per the documentation given below

There is no requirement that a thread that releases a permit must have acquired that permit by calling acquire().

The problem is my program gets stuck after 1 for n>2.

My program is given below:

import java.util.concurrent.Semaphore;

public class MultiThreading {

public static void main(String[] args) {

Semaphore sem = new Semaphore(1,false);

for(int i=5;i>=1;i--)

new MyThread(i, sem);

}

}

class MyThread implements Runnable {

int var;Semaphore sem;

public MyThread(int a, Semaphore s) {

var =a;sem=s;

new Thread(this).start();

}

@Override

public void run() {

System.out.println("Acquiring lock -- "+var);

try {

sem.acquire(var);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(var);

System.out.println("Releasing lock -- "+var);

sem.release(var+1);

}

}

Output is :

Acquiring lock -- 4  
Acquiring lock -- 5  
Acquiring lock -- 3  
Acquiring lock -- 2  
Acquiring lock -- 1  
1  
Releasing lock -- 1

While If I modify my code with tryAcquire, it runs perfectly well. Below is new run implementation

@Override

public void run() {

boolean acquired = false;

while(!acquired) {

acquired = sem.tryAcquire(var);

}

System.out.println(var);

sem.release(var+1);

}

Can someone please explain the semaphore's permit acquire mechanism when mulitple threads are waiting with different permit request??

A:

It's a clever strategy, but you're misunderstanding how Sempahore hands out permits. If you run your code enough times you'll actually see it reach step two:

Acquiring lock -- 5

Acquiring lock -- 1

1

Releasing lock -- 1

Acquiring lock -- 3

Acquiring lock -- 2

2

Acquiring lock -- 4

Releasing lock -- 2

If you keep on re-running it enough times you'd actually see it successfully finish. This happens because of how Semaphore hands out permits. You're assuming Semaphore will try to accommodate an acquire() call as soon as it has enough permits to do so. If we look carefully at the documentation for [Semaphore.aquire(int)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html#acquire-int-) we'll see that is not the case (emphasis mine):

If insufficient permits are available then the current thread becomes disabled for thread scheduling purposes and lies dormant until ... some other thread invokes one of the release methods for this semaphore, **the current thread is next to be assigned permits** and the number of available permits satisfies this request.

In other words Semaphore keeps a queue of pending acquire request and, upon each call to .release(), *only checks the head of the queue*. In particular if you enable fair queuing (set the second constructor argument to true) you'll see even step one doesn't occur, because step 5 is (usually) the first in the queue and even new acquire() calls that could be fulfilled will be queued up behind the other pending calls.

In short this means you cannot rely on .acquire() to return as soon as possible, as your code assumes.

By using .tryAcquire() in a loop instead you avoid making any blocking calls (and therefore put a lot more load on your Semaphore) and as soon as the necessary number of permits becomes available a tryAcquire() call will successfully obtain them. This works but is wasteful.

Picture a wait-list at a restaurant. Using .aquire() is like putting your name on the list and waiting to be called. It may not be perfectly efficient, but they'll get to you in a (reasonably) fair amount of time. Imagine instead if everyone just shouted at the host "Do you have a table for *n* yet?" as often as they could - that's your tryAquire() loop. It may still work out (as it does in your example) but it's certainly not the right way to go about it.

So what should you do instead? There's a number of possibly useful tools in [java.util.concurrent](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html), and which is best somewhat depends on what exactly you're trying to do. Seeing as you're effectively having each thread start the next one I might use a BlockingQueue as the synchronization aid, pushing the next step into the queue each time. Each thread would then poll the queue, and if it's not the activated thread's turn replace the value and wait again.

Here's an example:

public class MultiThreading {

public static void main(String[] args) throws Exception{

// Use fair queuing to prevent an out-of-order task

// from jumping to the head of the line again

// try setting this to false - you'll see far more re-queuing calls

BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(1, true);

for (int i = 5; i >= 1; i--) {

Thread.sleep(100); // not necessary, just helps demonstrate the queuing behavior

new MyThread(i, queue).start();

}

queue.add(1); // work starts now

}

static class MyThread extends Thread {

int var;

BlockingQueue<Integer> queue;

public MyThread(int var, BlockingQueue<Integer> queue) {

this.var = var;

this.queue = queue;

}

@Override

public void run() {

System.out.println("Task " + var + " is now pending...");

try {

while (true) {

int task = queue.take();

if (task != var) {

System.out.println(

"Task " + var + " got task " + task + " instead - re-queuing");

queue.add(task);

} else {

break;

}

}

} catch (InterruptedException e) {

// If a thread is interrupted, re-mark the thread interrupted and terminate

Thread.currentThread().interrupt();

return;

}

System.out.println("Finished task " + var);

System.out.println("Registering task " + (var + 1) + " to run next");

queue.add(var + 1);

}

}

}

This prints the following and terminates successfully:

Task 5 is now pending...

Task 4 is now pending...

Task 3 is now pending...

Task 2 is now pending...

Task 1 is now pending...

Task 5 got task 1 instead - re-queuing

Task 4 got task 1 instead - re-queuing

Task 3 got task 1 instead - re-queuing

Task 2 got task 1 instead - re-queuing

Finished task 1

Registering task 2 to run next

Task 5 got task 2 instead - re-queuing

Task 4 got task 2 instead - re-queuing

Task 3 got task 2 instead - re-queuing

Finished task 2

Registering task 3 to run next

Task 5 got task 3 instead - re-queuing

Task 4 got task 3 instead - re-queuing

Finished task 3

Registering task 4 to run next

Task 5 got task 4 instead - re-queuing

Finished task 4

Registering task 5 to run next

Finished task 5

Registering task 6 to run next

<https://stackoverflow.com/questions/12104978/need-to-semaphore-relase-if-semaphore-acquire-gets-interruptedexception>

# [**Need to semaphore.relase() if semaphore.acquire() gets InterruptedException?**](https://stackoverflow.com/questions/12104978/need-to-semaphore-relase-if-semaphore-acquire-gets-interruptedexception)

[Ask Question](https://stackoverflow.com/questions/ask)

Q:

From the Java java.util.concurrent.Semaphore docs it wasn't quite clear to me what happens if semaphore.acquire() blocks the thread and later gets interrupted by an InterruptedException. Has the semaphore value been decreased and so is there a need to release the semaphore?

Currently I am using code like this:

try {

// use semaphore to limit number of parallel threads

semaphore.acquire();

doMyWork();

}

finally {

semaphore.release();

}

Or should I rather not call release() when an InterruptedException occurs during acquire() ?

A1:

call release() when an InterruptedException occurs during acquire() ?

You should not. If .acquire() is interrupted, the semaphore is not acquired, so likely should not release it.

Your code should be

// use semaphore to limit number of parallel threads

semaphore.acquire();

try {

doMyWork();

}

finally {

semaphore.release();

}

A2:

nos's accepted answer is partially correct, except semaphore.acquire() also throws InterruptedException. So, to be 100% correct, the code would look like:

try {

semaphore.acquire();

try {

doMyWork();

} catch (InterruptedException e) {

// do something, if you wish

} finally {

semaphore.release();

}

} catch (InterruptedException e) {

// do something, if you wish

}