<https://stackoverflow.com/questions/17683575/binary-semaphore-vs-a-reentrantlock>

# [Binary Semaphore vs a ReentrantLock](https://stackoverflow.com/questions/17683575/binary-semaphore-vs-a-reentrantlock)

Q:

I've been trying to understand Reentrant locks and Semaphores ( the nesting of Reentrant locks vs release/unlock mechanism ).

It seems that having a Semaphore requires you to write a more thoroughly tested application because the release() method does not check if the thread releasing the permit is actually holding it. When I tested my test code, I found out that this may subsequently increase the number of permits beyond the initial limit. On the other hand, if a thread is not holding a reentrant lock when it invokes the unlock method, we get an IllegalMonitorException.

So would it be right to say that there is no real reason ever to have a binary semaphore as everything that a binary semaphore can do can also be done by a ReentrantLock. If we use binary semaphores we would have to check the entire method call stack to see if a permit was acquired before ( also was it released too if there is a possibility of a subsequent acquire - which might block if a release does not proceed it and so on ). Also since reentrant locks also provide one lock per object, isn't it always a better idea to prefer a reentrant lock to a binary semaphore?

I have checked a post here that talks about difference between a binary semaphore and a mutex but is there a thing like a mutex in Java?

Thanks, Chan.

P.S - I had posted this question in another forum ( <http://www.coderanch.com/t/615796/threads/java/reason-prefer-binary-Semaphore-Reentrant> ) and I haven't received a response yet. I thought I'd post it here as well to see what I can get.

A:

there is no real reason ever to have a binary semaphore as everything that a binary semaphore can do can also be done by a ReentrantLock

If all you need is reentrant mutual exclusion, then yes, there is no reason to use a binary semaphore over a ReentrantLock. If for any reason you need non-ownership-release semantics then obviously semaphore is your only choice.

Also since reentrant locks also provide one lock per object, isn't it always a better idea to prefer a reentrant lock to a binary semaphore?

It depends on the need. Like previously explained, if you need a simple mutex, then don't choose a semaphore. If more than one thread (but a limited number) can enter a critical section you can do this through either thread-confinement or a semaphore.

I have checked a post here that talks about difference between a binary semaphore and a mutex but is there a thing like a mutex in Java?

ReentrantLock and synchronized are examples of mutexes in Java.

Best example:

**The Dining Philosophers**

[**https://leetcode.com/problems/the-dining-philosophers/**](https://leetcode.com/problems/the-dining-philosophers/)

**Pure Semaphore solution:**

class DiningPhilosophers {

private Semaphore[] forks = new Semaphore[5];

private Semaphore semaphore = new Semaphore(4);

public DiningPhilosophers() {

for(int i = 0; i < 5; i++) {

forks[i] = new Semaphore(1);

}

}

public void pickFork(int id, Runnable pick) {

try {

forks[id].acquire();

} catch (InterruptedException e) {

e.printStackTrace();

}

pick.run();

}

public void putFork(int id, Runnable put) {

put.run();

forks[id].release();

}

// call the run() method of any runnable to execute its code

public void wantsToEat(int philosopher,

Runnable pickLeftFork,

Runnable pickRightFork,

Runnable eat,

Runnable putLeftFork,

Runnable putRightFork) throws InterruptedException {

int leftFork = philosopher;

int rightFork = (philosopher + 4) % 5;

semaphore.acquire();

pickFork(leftFork, pickLeftFork);

pickFork(rightFork, pickRightFork);

eat.run();

putFork(leftFork, putLeftFork);

putFork(rightFork, putRightFork);

semaphore.release();

}

}

**Reentrant Lock & Semaphore solution:**

|  |
| --- |
| class DiningPhilosophers { |
|  | private Lock forks[] = new Lock[5]; |
|  | private Semaphore semaphore = new Semaphore(4); |
|  |  |
|  | public DiningPhilosophers() { |
|  | for(int i = 0; i < 5; i++) { |
|  | forks[i] = new ReentrantLock(); |
|  | } |
|  | } |
|  |  |
|  | public void pickFork(int id, Runnable pick) { |
|  | forks[id].lock(); |
|  | pick.run(); |
|  | } |
|  |  |
|  | public void putFork(int id, Runnable put) { |
|  | put.run(); |
|  | forks[id].unlock(); |
|  | } |
|  |  |
|  | // call the run() method of any runnable to execute its code |
|  | public void wantsToEat(int philosopher, |
|  | Runnable pickLeftFork, |
|  | Runnable pickRightFork, |
|  | Runnable eat, |
|  | Runnable putLeftFork, |
|  | Runnable putRightFork) throws InterruptedException { |
|  | int leftFork = philosopher; |
|  | int rightFork = (philosopher + 4) % 5; |
|  |  |
|  | semaphore.acquire(); |
|  |  |
|  | pickFork(leftFork, pickLeftFork); |
|  | pickFork(rightFork, pickRightFork); |
|  | eat.run(); |
|  | putFork(rightFork, putRightFork); |
|  | putFork(leftFork, putLeftFork); |
|  |  |
|  | semaphore.release(); |
|  | } |
|  | } |