# Why use a ReentrantLock if one can use synchronized(this)?

I'm trying to understand what makes the lock in concurrency so important if one can use `synchronized (this)`. In the dummy code below, I can do either:

1. synchronized the entire method or synchronize the vulnerable area (synchronized(this){...})
2. OR lock the vulnerable code area with a ReentrantLock .

Code:

```
private final ReentrantLock lock =  new ReentrantLock();
    private static List<Integer> ints;

    public Integer getResult(String name) {
        .
        .
        .
        lock.lock();
        try{
            if (ints.size()==3) {
                ints=null;
                return -9;
            }

            for (int x=0; x<ints.size(); x++) {
                System.out.println("["+name+"] "+x+"/"+ints.size()+". values
>>>>"+ints.get(x));
            }

        }finally {
            lock.unlock();
        }
        return random;
}
```

java multithreading concurrency synchronize reentrantlock

edited Nov 7 '13 at 15:46

asked Aug 6 '12 at 2:08

adhg
**1,295** 2 17 39

## 3 Answers

I believe what a better comparison is between ReentrantLock and `synchronized`. ReadWriteLock serves a different purpose and is thus not directly analogous to `synchronized`. In that code, there appears to be no reason to be using a `ReadWriteLock`, but it may just be lack of shown code.

A `ReentrantLock` is *unstructured*, unlike `synchronized` constructs -- i.e. you don't need to use a block structure for locking and can even hold a lock across methods. An example:

```
private ReentrantLock lock;

public void foo() {
  ...
  lock.lock();
  ...
}

public void bar() {
  ...
  lock.unlock();
  ...
}
```

Such flow is impossible to represent via a single monitor in a `synchronized` construct.

Aside from that, `ReentrantLock` supports lock polling and interruptible lock waits that support time-out. `ReentrantLock` also has support for configurable *fairness* policy, allowing more flexible thread scheduling.

> The constructor for this class accepts an optional *fairness* parameter. When set `true`, under contention, locks favor granting access to the longest-waiting thread. Otherwise this lock does not guarantee any particular access order. Programs using fair locks accessed by many threads may display lower overall throughput (i.e., are slower; often much slower) than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation. Note however, that fairness of locks does not guarantee fairness of thread scheduling. Thus, one of many threads using a fair lock may obtain it multiple times in succession while other active threads are not progressing and not currently holding the lock. Also note that the untimed `tryLock` method does not honor the fairness setting. It will succeed if the lock is available even if other threads are waiting.

`ReentrantLock` **may** also be **more scalable**, performing much better under higher contention. You can read more about this here.

This claim has been contested, however; see the following comment:

> In the reentrant lock test, a new lock is created each time, thus there is no exclusive locking and the resulting data is invalid. Also, the IBM link offers no source code for the underlying benchmark so its impossible to characterize whether the test was even conducted correctly.

When should you use `ReentrantLock`s? According to that developerWorks article...

> The answer is pretty simple -- use it when you actually need something it provides that `synchronized` doesn't, like timed lock waits, interruptible lock waits, non-block-structured locks, multiple condition variables, or lock polling. `ReentrantLock` also has scalability benefits, and you should use it if you actually have a situation that exhibits high contention, but remember that the vast majority of `synchronized` blocks hardly ever exhibit any contention, let alone high contention. I would advise developing with synchronization until synchronization has proven to be inadequate, rather than simply assuming "the performance will be better" if you use `ReentrantLock`. Remember, these are advanced tools for advanced users. (And truly advanced users tend to prefer the simplest tools they can find until they're convinced the simple tools are inadequate.) As always, make it right first, and then worry about whether or not you have to make it faster.

edited Jul 24 '14 at 12:45
Michael Kohne
**8,484** 1 25 53

answered Aug 6 '12 at 2:29
oldrinb
**12.2k** 2 19 32

4 The 'known to be more scalable' link to lycog.com should be removed. In the reentrant lock test, a new lock is created each time, thus there is no exclusive locking and the resulting data is invalid. Also, the IBM link offers no source code for the underlying benchmark so its impossible to characterize whether the test was even conducted correctly. Personally, I'd just remove the whole line about scalability, as the entire claim is essentially unsupported. – Dev Sep 30 '13 at 20:41

2 I modified the post in light of your response. – oldrinb Sep 30 '13 at 22:19

We can achieve timed locking using wait() also, so can't we apply ReentrantLock functionality in it? – akash746 Feb 27 '14 at 6:09

2 Can't. Upvote. Enough. Great information here. – Justin Johnson Apr 4 '14 at 20:02

1 If the performance is a high concern for you, don't forget to look for a way where you need NO synchronization at all. – mcoolive Nov 25 '14 at 14:48

ReentrantReadWriteLock is a specialized lock whereas synchornized(this) is a general purpose lock. They are similar but not quite the same.

You are right in that you could use synchronized(this) instead of ReentrantReadWriteLock but the opposite is not always true.

If you'd like to better understand what makes ReentrantReadWriteLock special look up some information about producer-consumer thread synchronization.

In general you can remember that whole-method synchronization and general purpose synchronization (using the synchronized keyword) can be used in most applications without thinking *too much* about the semantics of the synchronization but if you need to squeeze performance out of your code you may need to explore other more fine-grained, or special-purpose synchronization mechanisms.

By the way, using synchronized(this) - and in general locking using a public class instance - can be problematic because it opens up your code to potential dead-locks because somebody else not knowingly might try to lock against your object somewhere else in the program.

edited Aug 6 '12 at 2:19

answered Aug 6 '12 at 2:10

Mike Dinescu

**24.8k** 2 47 79

---

A reentrant lock will allow the lock holder to enter blocks of code even after it has already obtained the lock by entering other blocks of code. A non-reentrant lock would have the lock holder block on itself as it would have to release the lock it obtained from another block of code to reobtain that same lock to enter the nested lock requiring block of code

```
public synchronized void functionOne() {

// do something

functionTwo();

// do something else

// redundant, but permitted...
synchronized(this) {
// do more stuff
}
}

public synchronized void functionTwo() {
// do even more stuff!
}
```

## Extended capabilities of reentrant lock include :-

1) The ability to have more than one condition variable per monitor. Monitors that use the synchronized keyword can only have one. This means reentrant locks support more than one wait()/notify() queue. 2) The ability to make the lock "fair". "[fair] locks favor granting access to the longest-waiting thread. Otherwise this lock does not guarantee any particular access order." Synchronized blocks are unfair. 3) The ability to check if the lock is being held. 4) The ability to get the list of threads waiting on the lock.

## Disadvantages of reentrant locks are :-

Need to add import statement. Need to wrap lock acquisitions in a try/finally block. This makes it more ugly than the synchronized keyword. The synchronized keyword can be put in method definitions which avoids the need for a block which reduces nesting.

## When to use :-

1. ReentrantLock might be more apt to use if you need to implement a thread that traverses a linked list, locking the next node and then unlocking the current node.
2. Synchronized keyword is apt in situation such as lock coarsening, provides adaptive spinning,biased locking and the potential for lock elision via escape analysis. Those optimizations aren't currently implemented for ReentrantLock.

For more information visit: http://www.ibm.com/developerworks/java/library/j-jtp10264/

answered Feb 27 '14 at 7:07

akash746

**159** 1 9

---