**自旋锁和互斥锁的区别**

**POSIX threads**(简称Pthreads)是在多核平台上进行并行编程的一套API。线程同步是并行编程中非常重要的通讯手段，其中最典型的应用就是用Pthreads提供的锁机制(lock)来对多个线程之间的共享临界区(Critical Section)进行保护(另一种常用的同步机制是barrier [屏障;障碍物;障碍;阻力;关卡;分界线;隔阂])。

**Pthreads提供了多种锁机制：**

* Mutex(**互斥量**)：pthread\_mutex\_t mtx
* Spin lock(**自旋锁**): pthread\_spin\_t spin
* Condition Variable(**条件变量**): pthread\_cond\_t cond
* Read/Write lock(**读写锁**)：pthread\_rwlock\_t rwlock
* Semaphore(**信号量**): sem\_t sem

1. **Pthreads提供的Mutex锁操作相关的API主要有：**

* pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);
* pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);
* pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

1. **Pthreads提供的Spin Lock锁操作相关的API主要有：**

* pthread\_spin\_lock(pthread\_spinlock\_t \*lock);
* pthread\_spin\_trylock(pthread\_spinlock\_t \*lock);
* pthread\_spin\_unlock(pthread\_spinlock\_t \*lock);

1. **Pthreads提供的Condition Variable(条件变量)操作相关的API主要有：**

* pthread\_cond\_init(pthread\_cond\_t\* cond, const pthread\_condattr\_t\* attr);
* pthread\_cond\_wait(pthread\_cond\_t\* cond, pthread\_mutex\_t\* mtx);
* pthread\_cond\_signal(pthread\_cond\_t\* cond);
* pthread\_cond\_destroy(pthread\_cond\_t\* cond);

1. **Pthreads提供的Read/Write Lock锁操作相关的API主要有：**

* **#include <pthread.h>**
* **pthread\_rwlock\_t rwlock;**

// 定义读写锁变量

* **pthread\_rwlock\_init(pthread\_rwlock\_t\* rwlock, const pthread\_rwlockattr\_t\* attr);**

// 读写锁初始化。

// 读写锁属性, 传NULL为默认属性。

* pthread\_rwlock\_rdlock(pthread\_rwlock\_t\* rwlock);

// 加读锁

* pthread\_rwlock\_tryrdlock(pthread\_rwlock\_t\* rwlock);

//

* pthread\_rwlock\_wrlock(pthread\_rwlock\_t\* rwlock);

//

* pthread\_rwlock\_trywrlock(pthread\_rwlock\_t\* rwlock);

//

* pthread\_rwlock\_unlock(pthread\_rwlock\_t\* rwlock);

//

* pthread\_rwlock\_destroy(pthread\_rwlock\_t\* rwlock);

//

1. **Semaphore信号量操作相关的API主要有：**

* **#include <semaphore.h>**
* **sem\_t sem;**

// 定义信号量

* **int sem\_init(sem\_t\* sem, int pshared, unsigned int value);**

// 初始化信号量

// sem: 信号量变量

// pshared: 0表示线程同步, 1表示进程同步

// value: 表示最多有几个线程/进程操作共享数据

// 成功返回0, 失败返回-1, 并设置errno值

* **sem\_wait(sem\_t\* sem);**

// 调用该函数一次相当于sem--, sem == 0时, 引起阻塞。

// 成功返回0, 失败返回-1, 并设置errno值。

* **sem\_post(sem\_t\* sem);**

// 调用该函数一次相当于sem++

// 成功返回0, 失败返回-1, 并设置errno值。

* **sem\_trywait(sem\_t\* sem);**

// 尝试加锁, 失败直接返回不阻塞。

// 成功返回0, 失败返回-1, 并设置errno值。

* **sem\_getvalue(sem\_t\* sem, int\* sval);**

// 把sem指向的信号量当前值放置在sval指向的整数上。

// 成功返回0, 失败返回-1, 并设置errno值。

// **注:** 信号量可能在返回值已经被修改。

* **sem\_destroy(sem\_t\* sem);**

// 销毁信号量

// 成功返回0, 失败返回-1, 并设置errno值。

从实现原理上来讲，**Mutex（互斥锁）属于sleep-waiting类型的锁**。例如在一个双核的机器上有两个线程（线程A和线程B）,它们分别运行在Core0和Core1上。假设线程A想要通过pthread\_mutex\_lock操作去得到一个临界区的锁，而此时这个锁正被线程B所持有，那么线程A就会被阻塞，Core0会在此时进行上下文切换(Context Switch)将线程A置于等待队列中，此时Core0就可以运行其它的任务而不必进行忙等待。而**Spin lock（自旋锁）**则不然，它属于**busy-waiting类型的锁**，如果线程A是使用pthread\_spin\_lock操作去请求锁，那么线程A就会一直在Core0上进行忙等待并不停的进行锁请求，直到得到这个锁为止。

**自旋锁（Spin lock）**

自旋锁与互斥锁有点类似，只是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，**“自旋锁”的作用是为了解决某项资源的互斥使用。因为自旋锁不会引起调用者睡眠，所以自旋锁的效率远高于互斥锁。**

自旋锁的不足之处：

**自旋锁一直占用着CPU，他在未获得锁的情况下，一直运行（自旋），所以占用着CPU，如果不能在很短的时间内获得锁，这无疑会使CPU效率降低。**

在用自旋锁时有可能造成死锁，当递归调用时有可能造成死锁，调用有些其他函数也可能造成死锁，如 copy\_to\_user()、copy\_from\_user()、kmalloc()等。

因此我们要慎重使用自旋锁，自旋锁只有在内核可抢占式或SMP的情况下才真正需要，在单CPU且不可抢占式的内核下，自旋锁的操作为空操作。自旋锁适用于锁使用者保持锁时间比较短的情况下。

**自旋锁-原理**

跟[互斥锁](https://baike.so.com/doc/6150631-6363825.html)一样，一个执行单元要想访问被自旋锁保护的**共享资源**，必须**先得到锁，在访问完共享资源后，必须释放锁**。如果在获取自旋锁时，没有任何执行单元保持该锁，那么将立即得到锁;如果在获取自旋锁时锁已经有保持者，那么获取锁操作将自旋在那里，直到该自旋锁的保持者释放了锁。由此我们可以看出，自旋锁是一种比较低级的保护数据结构或代码片段的原始方式，这种锁可能存在两个问题:

1、[死锁](https://baike.so.com/doc/414521-439063.html)。**试图递归地获得自旋锁必然会引起死锁**:递归程序的持有实例在第二个实例循环，以试图获得相同自旋锁时，不会释放此自旋锁。

在递归程序中使用自旋锁应遵守下列策略:

递归程序决不能在持有自旋锁时调用它自己，也决不能在[递归调用](https://baike.so.com/doc/6115126-6328268.html)时试图获得相同的自旋锁。此外如果一个进程已经将资源锁定，那么，即使其它申请这个资源的进程不停地疯狂"自旋",也无法获得资源，从而进入死循环。

2、过多占用cpu资源。如果不加限制，由于申请者一直在循环等待，因此自旋锁在锁定的时候,如果不成功,不会睡眠,会持续的尝试,单cpu的时候自旋锁会让其它process动不了. 因此，一般自旋锁实现会有一个参数限定最多持续尝试次数. 超出后, 自旋锁放弃当前time slice. 等下一次机会

由此可见，**自旋锁**比较**适用于**锁使用者**保持锁时间比较短**的情况。正是由于自旋锁使用者一般保持锁时间非常短，因此选择自旋而不是睡眠是非常必要的，自旋锁的效率远高于[互斥锁](https://baike.so.com/doc/6150631-6363825.html)。[信号量](https://baike.so.com/doc/6799211-7016022.html)和读写信号量适合于保持时间较长的情况，它们会导致调用者睡眠，因此只能在[进程上下文](https://baike.so.com/doc/2315046-2448796.html)使用，而自旋锁适合于保持时间非常短的情况，它可以在任何上下文使用。如果被保护的共享资源只在进程上下文访问，使用信号量保护该共享资源非常合适，如果对共享资源的访问时间非常短，自旋锁也可以。但是如果被保护的共享资源需要在中断上下文访问(包括底半部即[中断处理](https://baike.so.com/doc/4973549-5196235.html)句柄和顶半部即[软中断](https://baike.so.com/doc/7715429-7989524.html))，就必须使用自旋锁。**自旋锁保持期间是抢占失效的，而信号量和读写信号量保持期间是可以被抢占的**。自旋锁只有在[内核](https://baike.so.com/doc/665564-704577.html)可抢占或[SMP](https://baike.so.com/doc/4522250-4732301.html)(多处理器)的情况下才真正需要，在单CPU且不可抢占的内核下，自旋锁的所有操作都是空操作，

互斥锁(mutexlock)：

最常使用于线程同步的锁；标记用来保证在任一时刻，只能有一个线程访问该对象，**同一线程多次加锁操作会造成死锁**；临界区和互斥量都可用来实现此锁，通常情况下锁操作失败会将该线程睡眠等待锁释放时被唤醒

自旋锁(spinlock)：

同样用来标记只能有一个线程访问该对象，在同一线程多次加锁操作会造成死锁；使用硬件提供的swap指令或test\_and\_set指令实现；同互斥锁不同的是在锁操作需要等待的时候并不是睡眠等待唤醒，而是循环检测保持者已经释放了锁，这样做的好处是节省了线程从睡眠状态到唤醒之间内核会产生的消耗，在加锁时间短暂的环境下这点会提高很大效率

读写锁(rwlock)：

高级别锁，区分读和写，符合条件时允许多个线程访问对象。处于读锁操作时可以允许其他线程和本线程的读锁， 但不允许写锁， 处于写锁时则任何锁操作都会睡眠等待；常见的操作系统会在写锁等待时屏蔽后续的读锁操作以防写锁被无限孤立而等待，在操作系统不支持情况下可以用引用计数加写优先等待来用互斥锁实现。 读写锁适用于大量读少量写的环境，但由于其特殊的逻辑使得其效率相对普通的互斥锁和自旋锁要慢一个数量级；值得注意的一点是按POSIX标准 在线程申请读锁并未释放前本线程申请写锁是成功的，但运行后的逻辑结果是无法预测