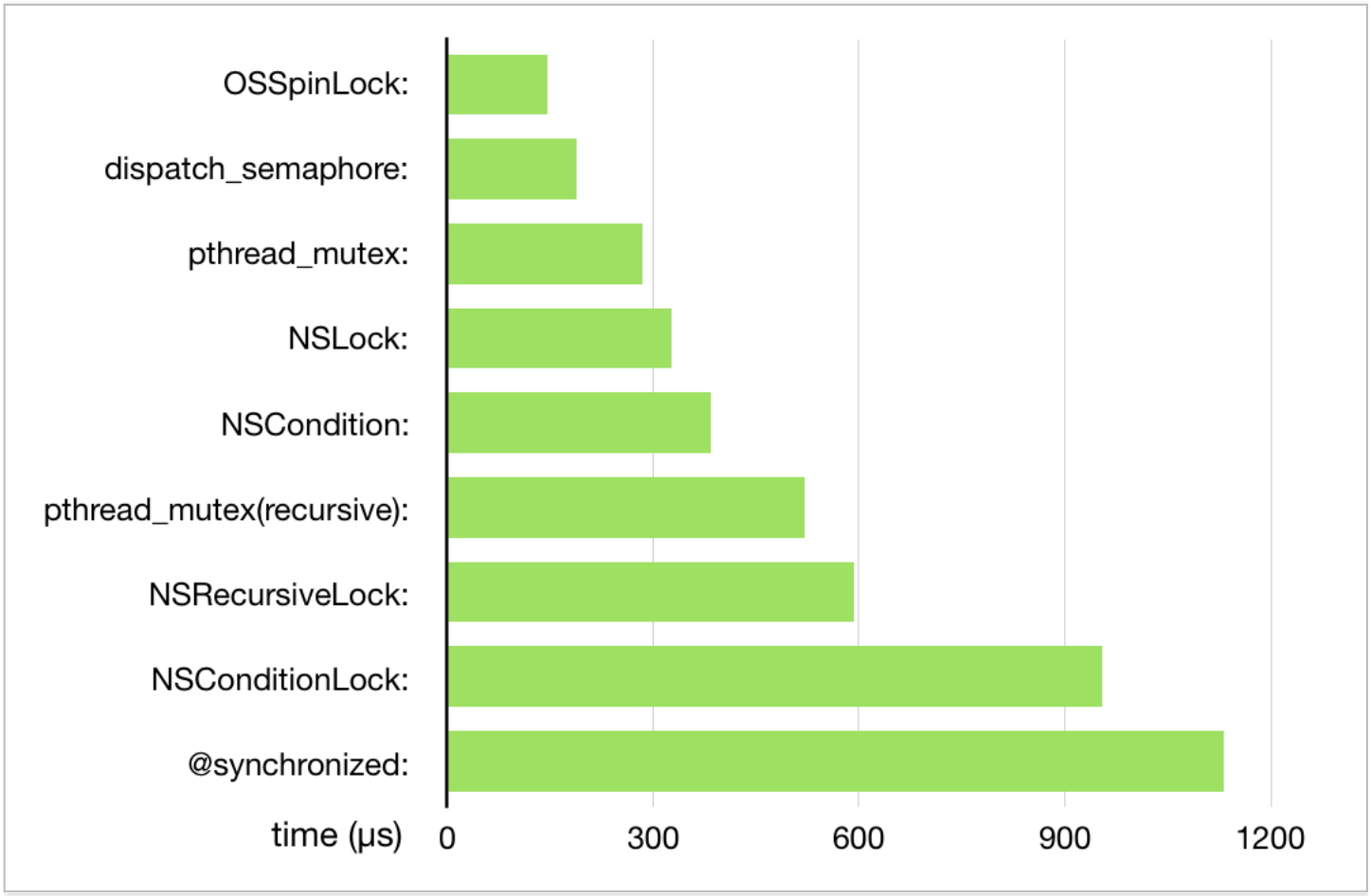
深入理解 iOS 开发中的锁 - KT 的 iOS 开发小站 - CSDN 博客

摘要

本文的目的不是介绍 iOS 中各种锁如何使用，一方面笔者没有大量的实战经验，另一方面这样的文章相当多，比如 iOS 中保证线程安全的几种方式与性能对比 、iOS 常见知识点（三）： Lock 。本文也不会详细介绍锁的具体实现原理，这会涉及到太多相关知识，笔者不敢误人子弟。

本文要做的就是简单的分析 iOS 开发中常见的几种锁如何实现，以及优缺点是什么，为什么会有性能上的差距，最终会简单的介绍锁的底层实现原理。水平有限，如果不慎有误，欢迎交流指正。同时建议读者在阅读本文以前，对 OC 中各种锁的使用方法先有大概的认识。

在 ibireme 的 不再安全的 OSSpinLock 一文中，有一张图片简单的比较了各种锁的加解锁性能:



本文会按照从上至下 (速度由快至慢) 的顺序分析每个锁的实现原理。需要说明的是，加解锁速度不表示锁的效率，只表示加解锁操作在执行时的复杂程度，下文会通过具体的例子来解释。

OSSpinLock

上述文章中已经介绍了 OSSpinLock 不再安全，主要原因发生在低优先级线程拿到锁时，高优先级线程进入忙等 (busy-wait) 状态，消耗大量 CPU 时间，从而导致低优先级线程拿不到 CPU 时间，也就无法完成任务并释放锁。这种问题被称为优先级反转。

为什么忙等会导致低优先级线程拿不到时间片？这还得从操作系统的线程调度说起。

现代操作系统在管理普通线程时，通常采用时间片轮转算法 (Round Robin，简称 RR)。每个线程会被分配一段时间片 (quantum)，通常在 10-100 毫秒左右。当线程用完属于自己的时间片以后，就会被操作系统挂起，放入等待队列中，直到下一次被分配时间片。

自旋锁的实现原理

自旋锁的目的是为了确保临界区只有一个线程可以访问，它的使用可以用下面这段伪代码来描述:

```
do {
    Acquire Lock
        Critical section  // 临界区
    Release Lock
        Reminder section // 不需要锁保护的代码
}
```

在 Acquire Lock 这一步，我们申请加锁，目的是为了保护临界区 (Critical Section) 中的代码不会被多个线程执行。

自旋锁的实现思路很简单，理论上来说只要定义一个全局变量，用来表示锁的可用情况即可，伪代码如下:

```
bool lock = false; // 一开始没有锁上，任何线程都可以申请锁
do {
    while(lock); // 如果 lock 为 true 就一直死循环，相当于申请锁
    lock = true; // 挂上锁，这样别的线程就无法获得锁
        Critical section  // 临界区
    lock = false; // 相当于释放锁，这样别的线程可以进入临界区
        Reminder section // 不需要锁保护的代码
}
```

注释写得很清楚，就不再逐行分析了。可惜这段代码存在一个问题: 如果一开始有多个线程同时执行 while 循环，他们都不会在这里卡住，而是继续执行，这样就无法保证锁的可靠性了。解决思路也很简单，只要确保申请锁的过程是原子操作即可。

原子操作

狭义上的原子操作表示一条不可打断的操作，也就是说线程在执行操作过程中，不会被操作系统挂起，而是一定会执行完。在单处理器环境下，一条汇编指令显然是原子操作，因为中断也要通过指令来实现。

然而在多处理器的情况下，能够被多个处理器同时执行的操作任然算不上原子操作。因此，真正的原子操作必须由硬件提供支持，比如 x86 平台上如果在指令前面加上 “LOCK” 前缀，对应的机器码在执行时会把总线锁住，使得其他 CPU 不能再执行相同操作，从而从硬件层面确保了操作的原子性。

这些非常底层的概念无需完全掌握，我们只要知道上述申请锁的过程，可以用一个原子性操作 `test_and_set` 来完成，它用伪代码可以这样表示:

```
bool test_and_set (bool *target) {
    bool rv = *target;
    *target = TRUE;
    return rv;
}
```

这段代码的作用是把 target 的值设置为 1，并返回原来的值。当然，在具体实现时，它通过一个原子性的指令来完成。

自旋锁的总结

至此，自旋锁的实现原理就很清楚了:

```
bool lock = false; // 一开始没有锁上，任何线程都可以申请锁
do {
    while(test_and_set(&lock); // test_and_set 是一个原子操作
        Critical section  // 临界区
    lock = false; // 相当于释放锁，这样别的线程可以进入临界区
        Reminder section // 不需要锁保护的代码
}
```

如果临界区的执行时间过长，使用自旋锁不是个好主意。之前我们介绍过时间片轮转算法，线程在多种情况下会退出自己的时间片。其中一种是用完了时间片的时间，被操作系统强制抢占。除此以外，当线程进行 I/O 操作，或进入睡眠状态时，都会主动让出时间片。显然在 while 循环中，线程处于忙等状态，白白浪费 CPU 时间，最终因为超时被操作系统抢占时间片。如果临界区执行时间较长，比如是文件读写，这种忙等是毫无必要的。

信号量

之前我在 介绍 GCD 底层实现的文章 中简单描述了信号量 `dispatch_semaphore_t` 的实现原理，它最终会调用到 `sem_wait` 方法，这个方法在 glibc 中被实现如下:

```
int sem_wait (sem_t *sem) {
  int *futex = (int *) sem;
  if (atomic_decrement_if_positive (futex) > 0)
    return 0;
  int err = lll_futex_wait (futex, 0);
```

```
        return -1;
    )
```

首先会把信号量的值减一，并判断是否大于零。如果大于零，说明不用等待，所以立刻返回。具体的等待操作在 `lll_futex_wait` 函数中实现，lll 是 low level lock 的简称。这个函数通过汇编代码实现，调用到 `SYS_futex` 这个系统调用，使线程进入睡眠状态，主动让出时间片，这个函数在互斥锁的实现中，也有可能被用到。

主动让出时间片并不总是代表效率高。让出时间片会导致操作系统切换到另一个线程，这种上下文切换通常需要 10 微秒左右，而且至少需要两次切换。如果等待时间很短，比如只有几个微秒，忙等就比线程睡眠更高效。

可以看到，自旋锁和信号量的实现都非常简单，这也是两者的加解锁耗时分别排在第一和第二的原因。再次强调，加解锁耗时不能准确反应出锁的效率 (比如时间片切换就无法发生)，它只能从一定程度上衡量锁的实现复杂程度。

pthread_mutex

pthread 表示 POSIX thread，定义了一组跨平台的线程相关的 API，pthread_mutex 表示互斥锁。互斥锁的实现原理与信号量非常相似，不是使用忙等，而是阻塞线程并睡眠，需要进行上下文切换。

互斥锁的常见用法如下:

```
pthread_mutexattr_t attr;
pthread_mutexattr_init(&attr);
pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_NORMAL);  // 定义锁的属性

pthread_mutex_t mutex;
pthread_mutex_init(&mutex, &attr) // 创建锁

pthread_mutex_lock(&mutex); // 申请锁
    // 临界区
pthread_mutex_unlock(&mutex); // 释放锁
```

对于 pthread_mutex 来说，它的用法和之前没有太大的改变，比较重要的是锁的类型，可以有 `PTHREAD_MUTEX_NORMAL` 、 `PTHREAD_MUTEX_ERRORCHECK` 、 `PTHREAD_MUTEX_RECURSIVE` 等等，具体的特性就不做解释了，网上有很多相关资料。

一般情况下，一个线程只能申请一次锁，也只能在获得锁的情况下才能释放锁，多次申请锁或释放未获得的锁都会导致崩溃。假设在已经获得锁的情况下再次申请锁，线程会因为等待锁的释放而进入睡眠状态，因此就不可能再释放锁，从而导致死锁。

然而这种情况经常会发生，比如某个函数申请了锁，在临界区内又递归调用了自己。幸运的是 `pthread_mutex` 支持递归锁，也就是允许一个线程递归的申请锁，只要把 attr 的类型改成 `PTHREAD_MUTEX_RECURSIVE` 即可。

互斥锁的实现

互斥锁在申请锁时，调用了 `pthread_mutex_lock` 方法，它在不同的系统上实现各有不同，有时候它的内部是使用信号量来实现，即使不用信号量，也会调用到 `lll_futex_wait` 函数，从而导致线程休眠。

上文说到如果临界区很短，忙等的效率也许更高，所以在有些版本的实现中，会首先尝试一定次数 (比如 1000 次) 的 test_and_test，这样可以在错误使用互斥锁时提高性能。

另外，由于 `pthread_mutex` 有多种类型，可以支持递归锁等，因此在申请加锁时，需要对锁的类型加以判断，这也就是为什么它和信号量的实现类似，但效率略低的原因。

NSLock

NSLock 是 Objective-C 以对象的形式暴露给开发者的一种锁，它的实现非常简单，通过宏，定义了 `lock` 方法:

```
#define MLOCK \
- (void) lock\
{\
  int err = pthread_mutex_lock(&_mutex);\
  // 错误处理 ……
}
```

`NSLock` 只是在内部封装了一个 `pthread_mutex` ，属性为 `PTHREAD_MUTEX_ERRORCHECK` ，它会损失一定性能换来错误提示。

这里使用宏定义的原因是，OC 内部还有其他几种锁，他们的 lock 方法都是一模一样，仅仅是内部 `pthread_mutex` 互斥锁的类型不同。通过宏定义，可以简化方法的定义。

`NSLock` 比 `pthread_mutex` 略慢的原因在于它需要经过方法调用，同时由于缓存的存在，多次方法调用不会对性能产生太大的影响。

NSCondition

`NSCondition` 的底层是通过条件变量 (condition variable) `pthread_cond_t` 来实现的。条件变量有点像信号量，提供了线程阻塞与信号机制，因此可以用来阻塞某个线程，并等待某个数据就绪，随后唤

醒线程，比如常见的生产者 - 消费者模式。

如何使用条件变量

很多介绍 `pthread_cond_t` 的文章都会提到，它需要与互斥锁配合使用:

```
void consumer () { // 消费者
    pthread_mutex_lock(&mutex);
    while (data == NULL) {
        pthread_cond_wait(&condition_variable_signal, &mutex); // 等待数据
    }
    // --- 有新的数据，以下代码负责处理 ↓↓↓↓↓↓
    // temp = data;
    // --- 有新的数据，以上代码负责处理 ↑↑↑↑↑↑
    pthread_mutex_unlock(&mutex);
}

void producer () {
    pthread_mutex_lock(&mutex);
    // 生产数据
    pthread_cond_signal(&condition_variable_signal); // 发出信号给消费者，告诉他们有了新的
    pthread_mutex_unlock(&mutex);
}
```

自然我们会有疑问:“如果不用互斥锁，只用条件变量会有什么问题呢？”。问题在于， `temp = data;` 这段代码不是线程安全的，也许在你把 data 读出来以前，已经有别的线程修改了数据。因此我们需要保证消费者拿到的数据是线程安全的。

wait 方法除了会被 signal 方法唤醒，有时还会被虚假唤醒，所以需要这里 while 循环中的判断来做二次确认。

为什么要使用条件变量

介绍条件变量的文章非常多，但大多都对一个一个基本问题避而不谈:“为什么要用条件变量？它仅仅是控制了线程的执行顺序，用信号量或者互斥锁能不能模拟出类似效果？”

网上的相关资料比较少，我简单说一下个人看法。信号量可以一定程度上替代 condition，但是互斥锁不行。在以上给出的生产者 - 消费者模式的代码中， `pthread_cond_wait` 方法的本质是锁的转移，消费者放弃锁，然后生产者获得锁，同理， `pthread_cond_signal` 则是一个锁从生产者到消费者转移的过程。

如果使用互斥锁，我们需要把代码改成这样:

```
void consumer () { // 消费者
    pthread_mutex_lock(&mutex);
    while (data == NULL) {
        pthread_mutex_unlock(&mutex);
        pthread_mutex_lock(&another_lock)  // 相当于 wait 另一个互斥锁
        pthread_mutex_lock(&mutex);
    }
    pthread_mutex_unlock(&mutex);
}
```

这样做存在的问题在于，在等待 another_lock 之前， 生产者有可能先执行代码， 从而释放了 another_lock。也就是说，我们无法保证释放锁和等待另一个锁这两个操作是原子性的，也就无法保证 “先等待、后释放 another_lock” 这个顺序。

用信号量则不存在这个问题，因为信号量的等待和唤醒并不需要满足先后顺序，信号量只表示有多少个资源可用，因此不存在上述问题。然而与 `pthread_cond_wait` 保证的原子性锁转移相比，使用信号量似乎存在一定风险 (暂时没有查到非原子性操作有何不妥)。

不过，使用 condition 有一个好处，我们可以调用 `pthread_cond_broadcast` 方法通知所有等待中的消费者，这是使用信号量无法实现的。

NSCondition 的做法

`NSCondition` 其实是封装了一个互斥锁和条件变量， 它把前者的 `lock` 方法和后者的 `wait/signal` 统一在 `NSCondition` 对象中，暴露给使用者:

```
- (void) signal {
  pthread_cond_signal(&_condition);
}

// 其实这个函数是通过宏来定义的，展开后就是这样
- (void) lock {
  int err = pthread_mutex_lock(&_mutex);
}
```

它的加解锁过程与 `NSLock` 几乎一致，理论上来说耗时也应该一样 (实际测试也是如此)。在图中显示它耗时略长，我猜测有可能是测试者在每次加解锁的前后还附带了变量的初始化和销毁操作。

NSRecursiveLock

上文已经说过，递归锁也是通过 `pthread_mutex_lock` 函数来实现，在函数内部会判断锁的类型，如果显示是递归锁，就允许递归调用，仅仅将一个计数器加一，锁的释放过程也是同理。

`NSRecursiveLock` 与 `NSLock` 的区别在于内部封装的 `pthread_mutex_t` 对象的类型不同，前者的类型为 `PTHREAD_MUTEX_RECURSIVE` 。

NSConditionLock

`NSConditionLock` 借助 `NSCondition` 来实现，它的本质就是一个生产者 - 消费者模型。“条件被满足” 可以理解为生产者提供了新的内容。 `NSConditionLock` 的内部持有一个 `NSCondition` 对象，以及 `_condition_value` 属性，在初始化时就会对这个属性进行赋值:

```
// 简化版代码
- (id) initWithCondition: (NSInteger)value {
    if (nil != (self = [super init])) {
        _condition = [NSCondition new]
        _condition_value = value;
    }
    return self;
}
```

它的 `lockWhenCondition` 方法其实就是消费者方法:

```
- (void) lockWhenCondition: (NSInteger)value {
    [_condition lock];
    while (value != _condition_value) {
        [_condition wait];
    }
}
```

对应的 `unlockWhenCondition` 方法则是生产者，使用了 `broadcast` 方法通知了所有的消费者:

```
- (void) unlockWithCondition: (NSInteger)value {
    _condition_value = value;
    [_condition broadcast];
    [_condition unlock];
}
```

@synchronized

这其实是一个 OC 层面的锁， 主要是通过牺牲性能换来语法上的简洁与可读。

我们知道 @synchronized 后面需要紧跟一个 OC 对象，它实际上是把这个对象当做锁来使用。这是通过一个哈希表来实现的，OC 在底层使用了一个互斥锁的数组 (你可以理解为锁池)，通过对对象去哈希值来得到对应的互斥锁。

具体的实现原理可以参考这篇文章: 关于 @synchronized，这儿比你想知道的还要多

参考资料

1. pthread_mutex_lock
2. ThreadSafety
3. Difference between binary semaphore and mutex
4. 关于 @synchronized，这儿比你想知道的还要多
5. pthread_mutex_lock.c 源码
6. [Pthread] Linux 中的线程同步机制 (二)–In Glibc
7. pthread 的各种同步机制
8. pthread_cond_wait
9. Conditional Variable vs Semaphore

全文完