白夜追凶，揭开iOS锁的秘密

2017-11-29 杨恒占 手机京东技术团队

前言

1. 时间片轮转调度算法 这是目前操作系统中大量使用的线程管理方式，大致就是操作系统会给每个线程分配一段时间片（通常100ms左右）这些线程都被放在一个队列中，CPU只需要维护这个队列，当队首的线程时间片耗尽就会被强制放到队尾等待，然后提取下一个队首线程执行

2. 原子操作“原子”：一般指最小粒度，不可分割；原子操作也就是不可分割，不可中断的操作

3. 临界区 每个进程中访问临界资源的那段代码称为临界区（Critical Section）（临界资源是一次仅允许一个进程使用的共享资源）。每次只准许一个进程进入临界区，进入后不允许其他进程进入。不论是硬件临界资源，还是软件临界资源，多个进程必须互斥地对它进行访问

4. 忙等 试图进入临界区的线程，占着CPU而不释放的状态

5. 互斥锁 如果一个线程无法获取互斥量，该线程会被直接挂起，不再消耗CPU时间，当其他线程释放互斥量后，操作系统会激活被挂起的线程。互斥锁会使得线程阻塞，阻塞的过程又分两个阶段，第一阶段是会先空转，可以理解成跑一个 while 循环，不断地去申请加锁，在空转一定时间之后，线程会进入 waiting 状态，此时线程就不占用CPU资源了，等锁可用的时候，这个线程会立即被唤醒

6. 自旋锁 如果一个线程需要获取自旋锁，该锁已经被其他线程占用，该线程不会被挂起，而是不断消耗CPU时间，一直试图获取自旋锁

NSLock

`NSLock` 是 Objective-C 以对象的形式暴露给开发者的一种锁, 它的组成结构是

```
 @protocol NSLocking

- (void)lock;
- (void)unlock;

@end

@interface NSLock : NSObject <NSLocking> {
@private
    void *_priv;
}
```

```
- (BOOL)tryLock;
- (BOOL)lockBeforeDate:(NSDate *)limit;

@property (nullable, copy) NSString *name NS_AVAILABLE(10_5, 2_0);

@end
```

NSLock的内部实现就是通过宏来定义的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

举个🌰

```
NSLock *lock = [[NSLock alloc] init];
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        //[lock lock];
        [lock lockBeforeDate:[NSDate date]];
            NSLog(@"需要线程同步的操作1 开始");
            sleep(2);
            NSLog(@"需要线程同步的操作1 结束");
        [lock unlock];

    });

    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        sleep(1);
        if ([lock tryLock]) {//尝试获取锁，如果获取不到返回NO，不会阻塞该线程
            NSLog(@"锁可用的操作");
            [lock unlock];
        }else{
            NSLog(@"锁不可用的操作");
        }

        NSDate *date = [[NSDate alloc] initWithTimeIntervalSinceNow:3];
        if ([lock lockBeforeDate:date]) {//尝试在未来的3s内获取锁，并阻塞该线程，如果3s内获取不
            NSLog(@"没有超时，获得锁");
            [lock unlock];
        }else{
            NSLog(@"超时，没有获得锁");
```

```
        }

    });
```

```
20:45:08.864 SafeMultiThread[35911:575795] 需要线程同步的操作1 开始
20:45:09.869 SafeMultiThread[35911:575781] 锁不可用的操作
20:45:10.869 SafeMultiThread[35911:575795] 需要线程同步的操作1 结束
20:45:10.870 SafeMultiThread[35911:575781] 没有超时，获得锁
```

NSRecursiveLock 递归锁

NSRecursiveLock实际上定义的是一个递归锁，他和 NSLock 的区别在于，NSRecursiveLock 可以在一个线程中重复加锁（反正单线程内任务是按顺序执行的，不会出现资源竞争问题）， NSRecursiveLock 会记录上锁和解锁的次数，当二者平衡的时候，才会释放锁，其它线程才可以上锁成功。所以这个锁可以被同一线程多次请求，而不会引起死锁。这主要是用在循环或递归操作中, 其组成结构是

```
@interface NSRecursiveLock : NSObject <NSLocking> {
@private
    void *_priv;
}

- (BOOL)tryLock;
- (BOOL)lockBeforeDate:(NSDate *)limit;

@property (nullable, copy) NSString *name NS_AVAILABLE(10_5, 2_0);

@end
```

举个🌰

```
NSRecursiveLock *lock = [[NSRecursiveLock alloc] init];
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{

        static void (^RecursiveBlock)(int);
        RecursiveBlock = ^(int value) {
            [lock lock];
            if (value > 0) {
                NSLog(@"value:%d", value);
                RecursiveBlock(value - 1);
            }
            [lock unlock];
        };
        RecursiveBlock(2);
    });
```

```
14:43:12.327 ThreadLockControlDemo[1878:145003] value:2
14:43:12.327 ThreadLockControlDemo[1878:145003] value:1
```

前文我们已经介绍过，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。 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 条件锁

NSConditionLock的组成结构是

```
@interface NSConditionLock : NSObject <NSLocking> {
@private
    void *_priv;
}

- (instancetype)initWithCondition:(NSInteger)condition NS_DESIGNATED_INITIALIZER;

@property (readonly) NSInteger condition;
- (void)lockWhenCondition:(NSInteger)condition;
- (BOOL)tryLock;
- (BOOL)tryLockWhenCondition:(NSInteger)condition;
- (void)unlockWithCondition:(NSInteger)condition;
- (BOOL)lockBeforeDate:(NSDate *)limit;
- (BOOL)lockWhenCondition:(NSInteger)condition beforeDate:(NSDate *)limit;

@property (nullable, copy) NSString *name NS_AVAILABLE(10_5, 2_0);

@end
```

NSConditionLock 和 NSLock 类似，都遵循 NSLocking 协议，方法都类似，只是多了一个 Condition 属性，以及每个操作都多了一个关于 Condition 属性的方法。例如 tryLock，tryLockWhenCondition:， NSConditionLock 可以称为条件锁，只有 Condition 参数与初始化时候的 Condition 相等， lock 才能正确进行加锁操作。而 unlockWithCondition: 并不是当 Condition 符合条件时才解锁，而是解锁之后，修改 Condition 的值

举个🌰

```
NSMutableArray *products = [NSMutableArray array];
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        while (1) {
            [lock lockWhenCondition:0];
            [products addObject:[[NSObject alloc] init]];
            NSLog(@"produce a product,总量:%zi",products.count);
            [lock unlockWithCondition:1];
            sleep(1);
        }

    });

    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        while (1) {
```

```
            NSLog(@"wait for product");
            [lock lockWhenCondition:1];
            [products removeObjectAtIndex:0];
            NSLog(@"custome a product");
            [lock unlockWithCondition:0];
        }

    });
```

```
20:31:58.699 SafeMultiThread[31282:521698] wait for product
20:31:58.699 SafeMultiThread[31282:521708] produce a product,总量:1
20:31:58.700 SafeMultiThread[31282:521698] custome a product
20:31:58.700 SafeMultiThread[31282:521698] wait for product
20:31:59.705 SafeMultiThread[31282:521708] produce a product,总量:1
20:31:59.706 SafeMultiThread[31282:521698] wait for product
20:32:00.707 SafeMultiThread[31282:521708] produce a product,总量:1
20:32:00.708 SafeMultiThread[31282:521698] custome a product
```

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。其组成结构是

```
@interface NSCondition : NSObject <NSLocking> {
@private
    void *_priv;
}

- (void)wait;
- (BOOL)waitUntilDate:(NSDate *)limit;
- (void)signal;
- (void)broadcast;

@property (nullable, copy) NSString *name NS_AVAILABLE(10_5, 2_0);

@end
```

> 需要注意的是pthread_cond_t要与互斥锁一起使用，为了保证数据的安全性

```
void consumer () { // 消费者
    pthread_mutex_lock(&mutex);
    while (data == NULL) {
        pthread_cond_wait(&condition_variable_signal, &mutex); // 等待数据
    }
    // temp = data;
    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，告诉他们有了新的数据
    pthread_mutex_unlock(&mutex);
}
```

总结一下使用条件变量的原因：

> 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 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);
}
```

`[condition wait];`让当前线程处于等待状态

`[condition signal];`CPU发信号告诉线程不用在等待，可以继续执行

需要注意的是： `NSCondition` 的对象实际上作为一个锁和一个线程检查器，锁上之后其它线程也能上锁，而之后可以根据条件决定是否继续运行线程，即线程是否要进入 waiting 状态，经测试， `NSCondition` 并不会像上文的那些锁一样，先轮询，而是直接进入 waiting 状态，当其它线程中的该锁执行 `signal` 或者 `broadcast` 方法时，线程被唤醒，继续运行之后的方法。

`NSCondition` 信号只能发送，不能保存（如果没有线程在等待，则发送的信号会失效）

使用模型是：

> 1.锁定条件对象。
>
> 2.测试是否可以安全的履行接下来的任务。
>
> 3.如果布尔值是假的，调用条件对象的 `wait` 或 `waitUntilDate:` 方法来阻塞线程。 在从这些方法返回，则转到步骤 2 重新测试你的布尔值。 （继续等待信号和重新测试，直到可以安全的履行接下来的任务。 `waitUntilDate:` 方法有个等待时间限制，指定的时间到了，则放回 NO，继续运行接下来的任务）
>
> 4.如果布尔值为真，执行接下来的任务。
>
> 5.当任务完成后，解锁条件对象

举个🌰

```
NSCondition *condition = [[NSCondition alloc] init];

    NSMutableArray *products = [NSMutableArray array];

    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        while (1) {
            [condition lock];
            if ([products count] == 0) {
                NSLog(@"wait for product");
                [condition wait];
            }
            [products removeObjectAtIndex:0];
```

```
            NSLog(@"custome a product");
            [condition unlock];
        }

    });

    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        while (1) {
            [condition lock];
            [products addObject:[[NSObject alloc] init]];
            NSLog(@"produce a product,总量:%zi",products.count);
            [condition signal];
            [condition unlock];
            sleep(1);
        }

    });
```

```
20:21:25.295 SafeMultiThread[31256:513991] wait for product
20:21:25.296 SafeMultiThread[31256:513994] produce a product,总量:1
20:21:25.296 SafeMultiThread[31256:513991] custome a product
20:21:25.297 SafeMultiThread[31256:513991] wait for product
20:21:26.302 SafeMultiThread[31256:513994] produce a product,总量:1
20:21:26.302 SafeMultiThread[31256:513991] custome a product
20:21:26.302 SafeMultiThread[31256:513991] wait for product
20:21:27.307 SafeMultiThread[31256:513994] produce a product,总量:1
20:21:27.308 SafeMultiThread[31256:513991] custome a product
```

@synchronized

`@synchronized`指令是在Objective-C代码中快速创建互斥锁的一种便捷方式。`@synchronized`指令执行其他任何互斥锁都会执行的操作 - 它可以防止不同线程同时获取同一个锁。 但是，在这种情况下，您不必直接创建互斥锁或锁定对象。 相反，您只需使用任何Objective-C对象作为锁定标记。

传递给 `@synchronized`指令的对象是用于区分受保护块的唯一标识符。如果在两个不同的线程中执行上述方法，则在每个线程上为anObj参数传递一个不同的对象，每个线程都会锁定并继续处理而不被另一个线程阻塞。但是，如果在两种情况下传递相同的对象，则其中一个线程将首先获取锁，另一个会阻塞，直到第一个线程完成临界区。

作为预防措施， `@synchronized`块隐式地向受保护的代码添加一个异常处理程序。如果引发异常，该处理程序会自动释放互斥锁。这意味着为了使用 `@synchronized`指令，还必须在代码中启用Objective-C异常处理。如果您不想由隐式异常处理程序引起额外开销，则应考虑使用锁类。

`@synchronized` 其内部实现大概是这样的

```
@try {
```

```
    objc_sync_enter(obj);
    // do work
} @finally {
    objc_sync_exit(obj);
}
```

当你调用 `objc_sync_enter(obj)` 时，它用 obj 内存地址的哈希值查找合适的 `SyncData`，然后将其上锁。当你调用 `objc_sync_exit(obj)` 时，它查找合适的 `SyncData` 并将其解锁。其中 `SyncData`的内部实现是：

```
typedef struct SyncData {
    id object;
    recursive_mutex_t mutex;
    struct SyncData* nextData;
    int threadCount;
} SyncData;
typedef struct SyncList {
    SyncData *data;
    spinlock_t lock;
} SyncList;
// Use multiple parallel lists to decrease contention among unrelated objects.
#define COUNT 16
#define HASH(obj) ((((uintptr_t)(obj)) >> 5) & (COUNT - 1))
#define LOCK_FOR_OBJ(obj) sDataLists[HASH(obj)].lock
#define LIST_FOR_OBJ(obj) sDataLists[HASH(obj)].data
static SyncList sDataLists[COUNT];
```

`object` 就是我们给 `@synchronized` 传入的那个对象, object 为该锁的唯一标识，只有当标识相同时，才满足互斥。内部 object 被释放或被设为 nil，从我做的测试的结果来看，的确没有问题，但如果 object 一开始就是 nil，则失去了锁的功能。不过虽然 nil 不行，但 `@synchronized([NSNull null])` 是完全可以的

`mutex` 它就是那个跟 object 关联在一起的锁

`nextData` 每个 SyncData 也包含一个指向另一个 SyncData 对象的指针, 所以你可以把每个 SyncData 结构体看做是链表中的一个元素

`threadCount` 这个 SyncData 对象中的锁会被一些线程使用或等待，threadCount 就是此时这些线程的数量。它很有用处，因为 SyncData 结构体会被缓存， `threadCount==0` 就暗示了这个 SyncData 实例可以被复用。

`data` 当做是链表中的节点。每个 SyncList 结构体都有个指向 SyncData 节点链表头部的指针，也有一个用于防止多个线程对此列表做并发修改的锁。

sDataLists 的声明 - 一个 SyncList 结构体数组，大小为16。通过定义的一个哈希算法将传入对象映射到数组上的一个下标。值得注意的是这个哈希算法设计的很巧妙，是将对象指针在内存的地址转化为无符号整型并右移五位，再跟 0xF 做按位与运算，这样结果不会超出数组大小。

LOCK_FOR_OBJ(obj) 和 LIST_FOR_OBJ(obj) 这俩宏就更好理解了，先是哈希出对象的数组下标，然后取出数组对应元素的 lock 或 data。一切都是这么顺理成章哈。

举个🌰

```
NSObject *obj = [[NSObject alloc] init];

    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        @synchronized(obj) {
            NSLog(@"需要线程同步的操作1 开始");
            sleep(3);
            NSLog(@"需要线程同步的操作1 结束");
        }
    });

    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        sleep(1);
        @synchronized(obj) {
            NSLog(@"需要线程同步的操作2");
        }
    });
```

```
20:48:35.747 SafeMultiThread[35945:580107] 需要线程同步的操作1 开始
20:48:38.748 SafeMultiThread[35945:580107] 需要线程同步的操作1 结束
20:48:38.749 SafeMultiThread[35945:580118] 需要线程同步的操作2
```

dispatch_semaphore

dispatch_semaphore 是 GCD 用来同步的一种方式，与他相关的只有三个函数，一个是创建信号量，一个是等待信号，一个是发送信号。他们的代码是：

```
dispatch_semaphore_create(long value);

dispatch_semaphore_wait(dispatch_semaphore_t dsema, dispatch_time_t timeout);

dispatch_semaphore_signal(dispatch_semaphore_t dsema);
```

dispatch_semaphore 和 NSCondition 类似，都是一种基于信号的同步方式，dispatch_semaphore 能保存发送的信号。 dispatch_semaphore 的核心是 dispatch_semaphore_t 类型的信号量。

dispatch_semaphore_t 的实现原理会调用最终会调用到 sem_post 和 sem_wait 方法， 它的实现方法是：

```
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;
)
```

atomic_decrement_if_positive()的语义就是如果传入参数是正数就将其原子性的减一并立即返回。如果信号量为

如果传入参数不是正数，即意味着有竞争，调用lll_futex_wait(futex,0)

> 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

sem_post 函数的作用是给信号量的值加上一个“1”，它是一个“原子操作"即同时对同一个信号量做加“1”操作的两个线程是不会冲突的；而同时对同一个文件进行读、加和写操作的两个程序就有可能会引起冲突。信号量的值永远会正确地加一个“2”－－因为有两个线程试图改变它。

sem_wait 函数也是一个原子操作，它的作用是从信号量的值减去一个“1”，但它永远会先等待该信号量为一个非零值才开始做减法。也就是说，如果你对一个值为2的信号量调用 sem_wait()，线程将会继续执行，信号量的值将减到1。如果对一个值为0的信号量调用 sem_wait()，这个函数就会地等待直到有其它线程增加了这个值使它不再是0为止。如果有两个线程都在 sem_wait() 中等待同一个信号量变成非零值，那么当它被第三个线程增加一个“1”时，等待线程中只有一个能够对信号量做减法并继续执行，另一个还将处于等待状态。

dispatch_semaphore_create(1) 方法可以创建一个 dispatch_semaphore_t 类型的信号量，设定信号量的初始值为 1。注意，这里的传入的参数必须大于或等于 0，否则 dispatch_semaphore_create 会返回 NULL。而且只有 dispatch_semaphore_create 为 1 的时候才能当做锁来用。如果 dispatch_semaphore 的信号量初始值为 x ，则可以有 x 个线程同时访问被保护的临界区

`dispatch_semaphore_wait (signal, overTime);` 方法会判断 signal 的信号值是否大于 0。大于 0 不会阻塞线程，消耗掉一个信号，执行后续任务。如果信号值为 0，该线程会和 `NSCondition` 一样直接进入 waiting 状态，等待其他线程发送信号唤醒线程去执行后续任务，或者当 overTime 时限到了，也会执行后续任务。

`dispatch_semaphore_signal(signal);` 发送信号，如果没有等待的线程接受信号，则使 signal 信号值加一（做到对信号的保存）

在没有等待情况出现时，它的性能比 `pthread_mutex` 还要高，但一旦有等待情况出现时，性能就会下降许多。相对于 `OSSpinLock` 来说，它的优势在于等待时不会消耗 CPU 资源。

举个🌰

```
dispatch_semaphore_t signal = dispatch_semaphore_create(1);
    dispatch_time_t overTime = dispatch_time(DISPATCH_TIME_NOW, 3 * NSEC_PER_SEC);

    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        dispatch_semaphore_wait(signal, overTime);
            NSLog(@"需要线程同步的操作1 开始");
            sleep(2);
            NSLog(@"需要线程同步的操作1 结束");
        dispatch_semaphore_signal(signal);
    });

    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        sleep(1);
        dispatch_semaphore_wait(signal, overTime);
            NSLog(@"需要线程同步的操作2");
        dispatch_semaphore_signal(signal);
    });
```

```
20:47:52.324 SafeMultiThread[35945:579032] 需要线程同步的操作1 开始
20:47:55.325 SafeMultiThread[35945:579032] 需要线程同步的操作1 结束
20:47:55.326 SafeMultiThread[35945:579033] 需要线程同步的操作2
```

如果把超时时间设置为<2s的时候，执行的结果就是：

```
18:53:24.049 SafeMultiThread[30834:434334] 需要线程同步的操作1 开始
18:53:25.554 SafeMultiThread[30834:434332] 需要线程同步的操作2
18:53:26.054 SafeMultiThread[30834:434334] 需要线程同步的操作1 结束
```

pthread_mutex （又称POSIX互斥锁）

`pthread_mutex` 是 C 语言下多线程加互斥锁的方式，你结构如下：

```
int pthread_mutex_init(pthread_mutex_t * __restrict, const pthread_mutexattr_t * __rest

int pthread_mutex_lock(pthread_mutex_t *);
```

```
int pthread_mutex_trylock(pthread_mutex_t *);

int pthread_mutex_unlock(pthread_mutex_t *);

int pthread_mutex_destroy(pthread_mutex_t *);

int pthread_mutex_setprioceiling(pthread_mutex_t * __restrict, int,
  int * __restrict);

int pthread_mutex_getprioceiling(const pthread_mutex_t * __restrict,
  int * __restrict);
```

互斥锁有2种初始化方式:

第一种是静态方式加锁： `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER ;` 全局变量或者static变量

第二种是动态方式加锁： `pthread_mutex_init ( pthread_mutex_t , const pthread_mutexattr_t ) pthread_mutexattr_t *` 为属性

`pthread_mutex_init` 这是初始化一个锁， `pthread_mutex_t` 为互斥锁的类型，传 NULL 为默认类型，一共有 4 类型

```
PTHREAD_MUTEX_NORMAL 缺省类型，也就是普通锁。当一个线程加锁以后，其余请求锁的线程将形成一个等待队列，
PTHREAD_MUTEX_ERRORCHECK 检错锁，如果同一个线程请求同一个锁，则返回 EDEADLK，否则与普通锁类型动作相
PTHREAD_MUTEX_RECURSIVE 递归锁，允许同一个线程对同一个锁成功获得多次，并通过多次 unlock 解锁。
PTHREAD_MUTEX_DEFAULT 适应锁，动作最简单的锁类型，仅等待解锁后重新竞争，没有等待队列。
```

`pthread_mutexattr_t` 可以用来设置锁的类型，比如递归锁

```
pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE)
```

`pthread_mutex_trylock` 方法， `pthread_mutex_trylock` 和 `tryLock` 的区别在于，`tryLock` 返回的是 YES 和 NO， `pthread_mutex_trylock` 加锁成功返回的是 0，失败返回的是错误提示码。

```
pthread_mutex_init(pthread_mutex_t mutex,const pthread_mutexattr_t attr);
初始化锁变量mutex。attr为锁属性，NULL值为默认属性。
pthread_mutex_lock(pthread_mutex_t mutex);加锁
pthread_mutex_tylock(*pthread_mutex_t *mutex);加锁，但是与2不一样的是当锁已经在使用的时候，返回
pthread_mutex_unlock(pthread_mutex_t *mutex);释放锁
pthread_mutex_destroy(pthread_mutex_t* mutex);使用完后释放，常用于递归锁的时候
pthread_mutexattr_setpshared 设置互斥锁范围语法
pthread_mutexattr_getpshared 获取互斥锁范围语法
```

举个🌰

```
__block pthread_mutex_t theLock;
//pthread_mutex_init(&theLock, NULL);

pthread_mutexattr_t attr;
pthread_mutexattr_init(&attr);
pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
pthread_mutex_init(&lock, &attr);
pthread_mutexattr_destroy(&attr);

dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{

    static void (^RecursiveMethod)(int);

    RecursiveMethod = ^(int value) {

        pthread_mutex_lock(&theLock);
        if (value > 0) {

            NSLog(@"value = %d", value);
            sleep(1);
            RecursiveMethod(value - 1);
        }
        pthread_mutex_unlock(&theLock);
    };

    RecursiveMethod(5);
});
```

OSSpinLock （已被苹果弃用）

OSSpinLock 常用的结构形式

```
typedef int32_t OSSpinLock;

bool    OSSpinLockTry( volatile OSSpinLock *__lock );

void    OSSpinLockLock( volatile OSSpinLock *__lock );

void    OSSpinLockUnlock( volatile OSSpinLock *__lock );
```

OSSpinLock 是一种自旋锁 和 NSLock 不同的是 NSLock 请求加锁失败的话，会先轮询，但一秒过后便会使线程进入 waiting 状态，等待唤醒。而 OSSpinLock 会一直轮询，等待时会消耗大量 CPU 资源，不适用于较长时间的任务。

弃用的原因：如果一个低优先级的线程获得锁并访问共享资源，这时一个高优先级的线程也尝试获得这个锁，它会处于 spinlock 的忙等状态从而占用大量 CPU。此时低优先级线程无法与高优先级线程争夺 CPU 时间，从而导致任务迟迟完不成、无法释放 lock。这并不只是理论上的问题，libobjc 已经遇到了很多次这个问题了，于是苹果的工程师停用了 OSSpinLock

自旋锁适合短时间的操作，加锁性能最快，但不能使用不同优先级。

举个 

```
 __block OSSpinLock theLock = OS_SPINLOCK_INIT;
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    OSSpinLockLock(&theLock);
    NSLog(@"需要线程同步的操作1 开始");
    sleep(3);
    NSLog(@"需要线程同步的操作1 结束");
    OSSpinLockUnlock(&theLock);

});

dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    OSSpinLockLock(&theLock);
    sleep(1);
    NSLog(@"需要线程同步的操作2");
    OSSpinLockUnlock(&theLock);

});
```

总结一下

> 根据锁的实现机制还有加解锁的速度，我们可以总结一下：如果应用在递归场景的时候，我们有2种选择，一个是 NSRecursiveLock，另一个是 pthread_mutex（recursive）。追求高性能的话，优先选择 pthread_mutex，苹果已对 pthread_mutex 进行过多次优化。
>
> 现在回到普通场景：不需要考虑性能的前提下，建议使用 @synchronized 进行加锁，这种锁不需要使用者考虑加锁和解锁的问题，已经在内部实现过了，但是他是加锁，解锁速率最慢的。如果是要封装第三方库的话，建议使用 dispatch_semaphore 或者 pthread_mutex 来进行加锁和解锁的操作，其中YYkit框架在大篇幅的使用 dispatch_semaphore_t 进行加锁和解锁。
>
> 在没有等待情况出现时，它的性能比 pthread_mutex 还要高，但一旦有等待情况出现时，性能就会下降许多。多资源分配，线程步调通知也不适合使用互斥锁进行分配。

参考文章

iOS中保证线程安全的几种方式与性能对比 [http://www.jianshu.com/p/938d68ed832c]

不再安全的 OSSpinLock [https://blog.ibireme.com/2016/01/16/spinlock_is_unsafe_in_ios/]

关于 @synchronized，这儿比你想知道的还要多 [http://yulingtianxia.com/blog/2015/11/01/More-than-you-want-to-know-about-synchronized/]

深入理解iOS开发中的锁 [https://bestswifter.com/ios-lock/#]