逻辑教育
Logic education

大师班第22天

和谐学习，不急不躁

LG_Cooci

逻辑教育
Logic education

调度组

最直接的作用: 控制任务执行顺序

dispatch_group_create	创建组
dispatch_group_async	进组任务
dispatch_group_notify	进组任务执行完毕通知
dispatch_group_wait	进组任务执行等待时间
dispatch_group_enter	进组
dispatch_group_leave	出组

注意搭配使用

逻辑教育
Logic education

信号量dispatch_semaphore_t

dispatch_semaphore_create	创建信号量
dispatch_semaphore_wait	信号量等待
dispatch_semaphore_signal	信号量释放

同步->当锁, 控制GCD最大并发数

逻辑教育
Logic education

Dispatch_Source

* 其 CPU 负荷非常小，尽量不占用资源

* 联结的优势

在任一线程上调用它的的一个函数 dispatch_source_merge_data 后，会执行 Dispatch Source 事先定义好的句柄（可以把句柄简单理解为一个 block ）

这个过程叫 Custom event ,用户事件。是 dispatch source 支持处理的一种事件

句柄是一种指向指针的指针 它指向的就是一个类或者结构，它和系统有很密切的关系 HINSTANCE（实例句柄），HBITMAP（位图句柄），HDC（设备表述句柄），HICON（图标句柄）等。这当中还有一个通用的句柄，就是HANDLE

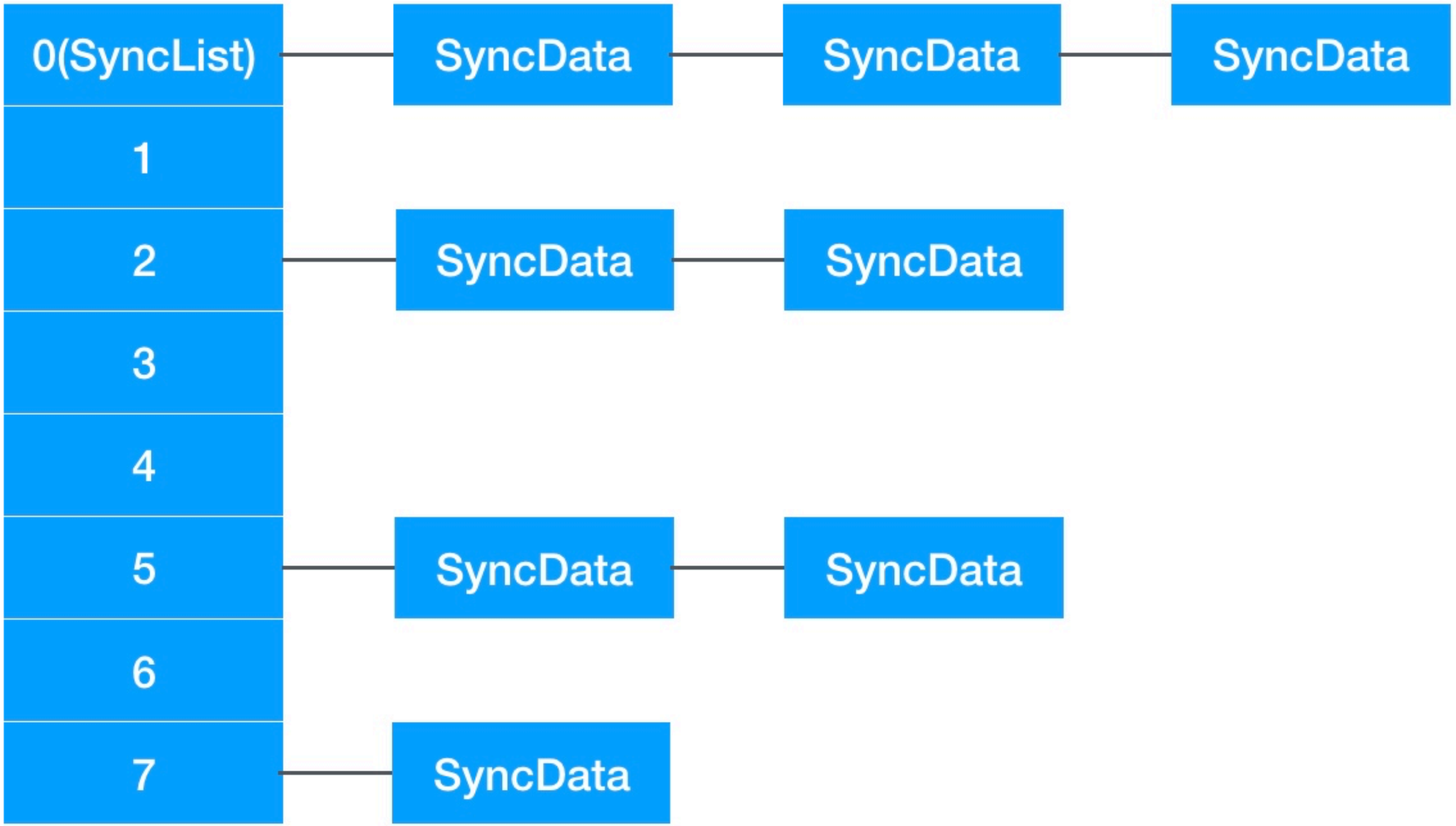
逻辑教育
Logic education

Dispatch_Source

dispatch_source_create	创建源
dispatch_source_set_event_handler	设置源事件回调
dispatch_source_merge_data	源事件设置数据
dispatch_source_get_data	获取源事件数据
dispatch_resume	继续
dispatch_suspend	挂起

逻辑教育

SyncList分析



逻辑教育
Logic education

TLS 线程相关解释

线程局部存储（Thread Local Storage,TLS）：是操作系统为线程单独提供的私有空间，通常只有有限的容量。Linux系统下通常通过pthread库中的

pthread_key_create()、

pthread_getspecific()、

pthread_setspecific()、

pthread_key_delete()

互斥锁　https://baike.baidu.com/item/互斥锁/841823?fr=aladdin

在Posix Thread中定义有一套专门用于线程同步的mutex函数

mutex，用于保证在任何时刻，都只能有一个线程访问该对象。
当获取锁操作失败时，线程会进入睡眠，等待锁释放时被唤醒

1. 创建和销毁

A:POSIX定义了一个宏PTHREAD_MUTEX_INITIALIZER来静态初始化互斥锁
B:int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)
C:pthread_mutex_destroy ()用于注销一个互斥锁

2. 互斥锁属性

3. 锁操作

int pthread_mutex_lock(pthread_mutex_t *mutex)
int pthread_mutex_unlock(pthread_mutex_t *mutex)
int pthread_mutex_trylock(pthread_mutex_t *mutex)
pthread_mutex_trylock()语义与pthread_mutex_lock()类似，不同的是在锁已经被占据时返回EBUSY而不是挂起等待。

互斥锁，分为递归锁和非递归锁。

逻辑教育
Logic education

NSCondition

NSCondition 的对象实际上作为一个锁和一个线程检查器：锁主要为了当检测条件时保护数据源，执行条件引发的任务；线程检查器主要是根据条件决定是否继续运行线程，即线程是否被阻塞

1：[condition lock];//一般用于多线程同时访问、修改同一个数据源，保证在同一时间内数据源只被访问、修改一次，其他线程的命令需要在lock 外等待，只到unlock ，才可访问

2：[condition unlock];//与lock 同时使用

3：[condition wait];//让当前线程处于等待状态

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

NSConditionLock

1.1 NSConditionLock 是锁，一旦一个线程获得锁，其他线程一定等待

1.2 [xxxx lock]; 表示 xxx 期待获得锁，如果没有其他线程获得锁（不需要判断内部的condition) 那它能执行此行以下代码，如果已经有其他线程获得锁（可能是条件锁，或者无条件锁），则等待，直至其他线程解锁

1.3 [xxx lockWhenCondition:A条件]; 表示如果没有其他线程获得该锁，但是该锁内部的condition不等于A条件，它依然不能获得锁，仍然等待。如果内部的condition等于A条件，并且没有其他线程获得该锁，则进入代码区，同时设置它获得该锁，其他任何线程都将等待它代码的完成，直至它解锁。

1.4 [xxx unlockWithCondition:A条件]; 表示释放锁，同时把内部的condition设置为A条件

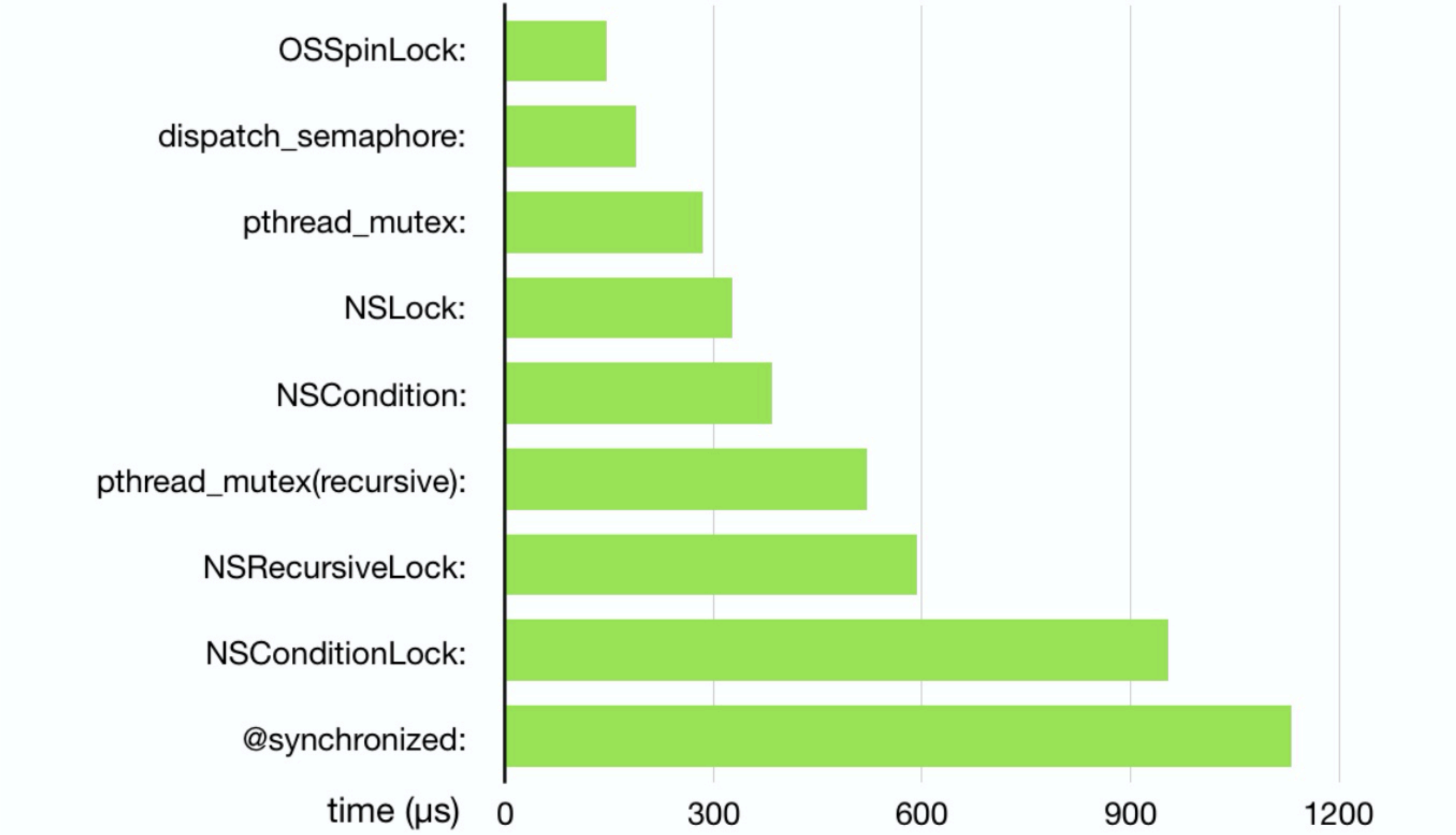
1.5 return = [xxx lockWhenCondition:A条件 beforeDate:A时间]; 表示如果被锁定（没获得锁），并超过该时间则不再阻塞线程。但是注意：返回的值是NO,它没有改变锁的状态，这个函数的目的在于可以实现两种状态下的处理

1.6 所谓的condition就是整数，内部通过整数比较条件

NSConditionLock总结

- 线程 1 调用`[NSConditionLock lockWhenCondition:]`，此时此刻因为不满足当前条件，所以会进入 waiting 状态，当前进入到 waiting 时，会释放当前的互斥锁。
- 此时当前的线程 3 调用`[NSConditionLock lock:]`，本质上是调用`[NSConditionLock lockBeforeDate:]`，这里不需要比对条件值，所以线程 3 会打印
- 接下来线程 2 执行`[NSConditionLock lockWhenCondition:]`，因为满足条件值，所以线程 2 会打印，打印完成后会调用`[NSConditionLock unlockWithCondition:]`,这个时候讲 value 设置为 1，并发送 boradcast, 此时线程 1 接收到当前的信号，唤醒执行并打印。
- 自此当前打印为 线程 3->线程 2 -> 线程 1。
- `[NSConditionLock lockWhenCondition:]`: 这里会根据传入的 condition 值和 Value 值进行对比，如果不相等，这里就会阻塞，进入线程池，否则的话就继续代码执行
- `[NSConditionLock unlockWithCondition:]`:这里会先更改当前的 value 值，然后进行广播，唤醒当前的线程。

锁的性能数据



逻辑教育
Logic education

锁的归类

自旋锁：线程反复检查锁变量是否可用。由于线程在这一过程中保持执行，因此是一种忙等待。一旦获取了自旋锁，线程会一直保持该锁，直至显式释放自旋锁。 自旋锁避免了进程上下文的调度开销，因此对于线程只会阻塞很短时间的场合是有效的。

互斥锁：是一种用于多线程编程中，防止两条线程同时对同一公共资源（比如全局变量）进行读写的机制。该目的通过将代码切片成一个一个的临界区而达成

这里属于互斥锁的有：

- NSLock
- pthread_mutex
- @synchronized

逻辑教育
Logic education

锁的归类

条件锁：就是条件变量，当进程的某些资源要求不满足时就进入休眠，也就是锁住了。当资源被分配到了，条件锁打开，进程继续运行

- NSCondition
- NSConditionLock

递归锁：就是同一个线程可以加锁N次而不会引发死锁

- NSRecursiveLock
- pthread_mutex(recursive)

信号量（semaphore）：是一种更高级的同步机制，互斥锁可以说是semaphore在仅取值0/1时的特例。信号量可以有更多的取值空间，用来实现更加复杂的同步，而不单单是线程间互斥。

- dispatch_semaphore

其实基本的锁就包括了三类 自旋锁 互斥锁 读写锁，
其他的比如条件锁，递归锁，信号量都是上层的封装和实现！

逻辑教育
Logic education

读写锁

读写锁实际是一种特殊的自旋锁，它把对共享资源的访问者划分成读者和写者，读者只对共享资源进行读访问，写者则需要对共享资源进行写操作。这种锁相对于自旋锁而言，能提高并发性，因为在多处理器系统中，它允许同时有多个读者来访问共享资源，最大可能的读者数为实际的逻辑CPU数。写者是排他性的，一个读写锁同时只能有一个写者或多个读者（与CPU数相关），但不能同时既有读者又有写者。在读写锁保持期间也是抢占失效的。
如果读写锁当前没有读者，也没有写者，那么写者可以立刻获得读写锁，否则它必须自旋在那里，直到没有任何写者或读者。如果读写锁没有写者，那么读者可以立即获得该读写锁，否则读者必须自旋在那里，直到写者释放该读写锁。

一次只有一个线程可以占有写模式的读写锁, 但是可以有多个线程同时占有读模式的读写锁. 正是因为这个特性,
当读写锁是写加锁状态时, 在这个锁被解锁之前, 所有试图对这个锁加锁的线程都会被阻塞.
当读写锁在读加锁状态时, 所有试图以读模式对它进行加锁的线程都可以得到访问权, 但是如果线程希望以写模式对此锁进行加锁, 它必须直到所有的线程释放锁.
通常, 当读写锁处于读模式锁住状态时, 如果有另外线程试图以写模式加锁, 读写锁通常会阻塞随后的读模式锁请求, 这样可以避免读模式锁长期占用, 而等待的写模式锁请求长期阻塞.
读写锁适合于对数据结构的读次数比写次数多得多的情况. 因为, 读模式锁定时可以共享, 以写模式锁住时意味着独占, 所以读写锁又叫共享-独占锁.

读写锁

读写锁适合于对数据结构的读次数比写次数多得多的情况. 因为, 读模式锁定时可以共享, 以写模式锁住时意味着独占, 所以读写锁又叫共享-独占锁.

```
#include <pthread.h>
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock)
```

成功则返回0, 出错则返回错误编号.

同互斥量以上, 在释放读写锁占用的内存之前, 需要先通过pthread_rwlock_destroy对读写锁进行清理工作, 释放由init分配的资源.

```
#include <pthread.h>
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

成功则返回0, 出错则返回错误编号.

这3个函数分别实现获取读锁, 获取写锁和释放锁的操作. 获取锁的两个函数是阻塞操作, 同样, 非阻塞的函数为:

```
#include <pthread.h>
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

成功则返回0, 出错则返回错误编号.

非阻塞的获取锁操作, 如果可以获取则返回0, 否则返回错误的EBUSY.

逻辑教育
Logic education

Hello Cooci

我就是我，颜色不一样的烟火