<https://blog.csdn.net/yeyuangen/article/details/37593533>

===============================man pthread\_cond\_wait的解释==========================

LINUX环境下多线程编程肯定会遇到需要条件变量的情况，此时必然要使用pthread\_cond\_wait()函数。但这个函数的执行过程比较难于理解。  
    pthread\_cond\_wait()的工作流程如下（以MAN中的EXAMPLE为例）：  
       Consider two shared variables x and y, protected by the mutex mut, and a condition vari-  
       able cond that is to be signaled whenever x becomes greater than y.

              int x,y;  
              pthread\_mutex\_t mut = PTHREAD\_MUTEX\_INITIALIZER;  
              pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;

       Waiting until x is greater than y is performed as follows:

              pthread\_mutex\_lock(&mut);  
              while (x <= y) {  
                      pthread\_cond\_wait(&cond, &mut);  
              }  
              /\* operate on x and y \*/  
              pthread\_mutex\_unlock(&mut);

       Modifications on x and y that may cause x to become greater than y should signal the con-  
       dition if needed:

              pthread\_mutex\_lock(&mut);  
              /\* modify x and y \*/  
              if (x > y) pthread\_cond\_broadcast(&cond);  
              pthread\_mutex\_unlock(&mut);

     这个例子的意思是，两个线程要修改X和 Y的值，第一个线程当X<=Y时就挂起，直到X>Y时才继续执行（由第二个线程可能会修改X,Y的值，当X>Y时唤醒第一个线程），即 首先初始化一个普通互斥量mut和一个条件变量cond。之后分别在两个线程中分别执行如下函数体：

              pthread\_mutex\_lock(&mut);  
              while (x <= y) {  
                      pthread\_cond\_wait(&cond, &mut);  
              }  
              /\* operate on x and y \*/  
              pthread\_mutex\_unlock(&mut);

和：       pthread\_mutex\_lock(&mut);  
              /\* modify x and y \*/  
              if (x > y) pthread\_cond\_signal(&cond);  
              pthread\_mutex\_unlock(&mut);   
    其实函数的执行过程非常简单，在第一个线程执行到pthread\_cond\_wait(&cond,&mut)时，此时如果X<=Y，则此函数就将mut互斥量解锁 ，再将cond条件变量加锁 ，此时第一个线程挂起 （不占用任何CPU周期）。  
    而在第二个线程中，本来因为mut被第一个线程锁住而阻塞，此时因为mut已经释放，所以可以获得锁mut，并且进行修改X和Y的值，在修改之后，一个IF语句判定是不是X>Y，如果是，则此时pthread\_cond\_signal()函数会唤醒第一个线程 ，并在下一句中释放互斥量mut。然后第一个线程开始从pthread\_cond\_wait()执行，首先要再次**锁mut** ， 如果锁成功，再进行**条件的判断** （至于为什么用WHILE，即在被唤醒之后还要再判断，后面有原因分析），如果满足条件，则**被唤醒** 进行处理，最后释放互斥量mut 。

    至于为什么在被唤醒之后还要再次进行条件判断（即为什么要使用while循环来判断条件），是因为可能有“惊群效应”。有人觉得此处既然是被唤醒的，肯定 是满足条件了，其实不然。如果是多个线程都在等待这个条件，而同时只能有一个线程进行处理，此时就必须要再次条件判断，以使只有一个线程进入临界区处理。 对此，转来一段：

引用下POSIX的RATIONALE：   
  
Condition Wait Semantics   
  
It is important to note that when pthread\_cond\_wait() and pthread\_cond\_timedwait() return without error, the associated predicate may still be false. Similarly, when pthread\_cond\_timedwait() returns with the timeout error, the associated predicate may be true due to an unavoidable race between the expiration of the timeout and the predicate state change.   
  
The application needs to recheck the predicate on any return because it cannot be sure there is another thread waiting on the thread to handle the signal, and if there is not then the signal is lost. The burden is on the application to check the predicate.   
  
Some implementations, particularly on a multi-processor, may sometimes cause multiple threads to wake up when the condition variable is signaled simultaneously on different processors.   
  
In general, whenever a condition wait returns, the thread has to re-evaluate the predicate associated with the condition wait to determine whether it can safely proceed, should wait again, or should declare a timeout. A return from the wait does not imply that the associated predicate is either true or false.   
  
It is thus recommended that a condition wait be enclosed in the equivalent of a "while loop" that checks the predicate.   
  
**从上文可以看出：**   
1，pthread\_cond\_signal在多处理器上可能同时唤醒多个线程，当你只能让一个线程处理某个任务时，其它被唤醒的线程就需要继续 wait,while循环的意义就体现在这里了，而且规范要求pthread\_cond\_signal至少唤醒一个pthread\_cond\_wait上 的线程，其实有些实现为了简单在单处理器上也会唤醒多个线程.   
2，某些应用，如线程池，pthread\_cond\_broadcast唤醒全部线程，但我们通常只需要一部分线程去做执行任务，所以其它的线程需要继续wait.所以强烈推荐此处使用while循环.

       其实说白了很简单，就是pthread\_cond\_signal()也可能唤醒多个线程，而如果你同时只允许一个线程访问的话，就必须要使用while来进行条件判断，以保证临界区内只有一个线程在处理。

pthread\_cond\_wait()  用于阻塞当前线程，等待别的线程使用 pthread\_cond\_signal() 或pthread\_cond\_broadcast来唤醒它 。  pthread\_cond\_wait()   必须与pthread\_mutex 配套使用。pthread\_cond\_wait() 函数一进入wait状态就会自动release mutex。当其他线程通过 pthread\_cond\_signal() 或pthread\_cond\_broadcast ，把该线程唤醒，使 pthread\_cond\_wait()通过（返回）时，该线程又自动获得该mutex 。

        pthread\_cond\_signal 函数的作用是发送一个信号给另外一个正在处于阻塞等待状态的线程,使其脱离阻塞状态,继续执行.如果没有线程处在阻塞等待状态,pthread\_cond\_signal也会成功返回。

        使用pthread\_cond\_signal一般不会有“惊群现象”产生，他最多只给一个线程发信号。假如有多个线程正在阻塞等待着这个条件变量的话，那么是根据各等待线程优先级的高低确定哪个线程接收到信号开始继续执行。如果各线程优先级相同，则根据等待时间的长短来确定哪个线程获得信号。但无论如何一个pthread\_cond\_signal调用最多发信一次。

        但是 pthread\_cond\_signal 在多处理器上可能同时唤醒多个线程，当你只能让一个线程处理某个任务时，其它被唤醒的线程就需要继续 wait，

==============================另一篇很好的文章===========================

[POSIX线程详解](http://www.ibm.com/developerworks/cn/linux/thread/posix_thread2/index.html)

==============================使用效率问题============================

pthread\_cond\_signal函数的作用是发送一个信号给另外一个正在处于阻塞等待状态的线程,使其脱离阻塞状态,继续执行.如果没有线程处在阻塞等待状态,pthread\_cond\_signal也会成功返回。

但使用pthread\_cond\_signal不会有“惊群现象”产生，他最多只给一个线程发信号。假如有多个线程正在阻塞等待着这个条件变量的话，那么是根据各等待线程优先级的高低确定哪个线程接收到信号开始继续执行。如果各线程优先级相同，则根据等待时间的长短来确定哪个线程获得信号。但无论如何一个pthread\_cond\_signal调用最多发信一次。

另外，互斥量的作用一般是用于对某个资源进行互斥性的存取,很多时候是用来保证操作是一个原子性的操作，是不可中断的。

用法：

pthread\_cond\_wait必须放在pthread\_mutex\_lock和pthread\_mutex\_unlock之间，因为他要根据共享变量的状态来决定是否要等待，而为了不永远等待下去所以必须要在lock/unlock队中

共享变量的状态改变必须遵守lock/unlock的规则

pthread\_cond\_signal即可以放在pthread\_mutex\_lock和pthread\_mutex\_unlock之间，也可以放在pthread\_mutex\_lock和pthread\_mutex\_unlock之后，但是各有各缺点。

之间：

pthread\_mutex\_lock

xxxxxxx

pthread\_cond\_signal

pthread\_mutex\_unlock

缺点：在某下线程的实现中，会造成等待线程从内核中唤醒（由于cond\_signal)然后又回到内核空间（因为cond\_wait返回后会有原子加锁的行为），所以一来一回会有性能的问题。

在code review中，我会发现很多人喜欢在pthread\_mutex\_lock()和pthread\_mutex\_unlock(()之间调用 pthread\_cond\_signal或者pthread\_cond\_broadcast函数，从逻辑上来说，这种使用方法是完全正确的。但是在多线程环境中，这种使用方法可能是低效的。posix1标准说，pthread\_cond\_signal与pthread\_cond\_broadcast无需考虑调用线程是否是mutex的拥有者，也就是说，可以在lock与unlock以外的区域调用。如果我们对调用行为不关心，那么请在lock区域之外调用吧。这里举个例子：  
       我们假设系统中有线程1和线程2，他们都想获取mutex后处理共享数据，再释放mutex。请看这种序列：  
       1)线程1获取mutex，在进行数据处理的时候，线程2也想获取mutex，但是此时被线程1所占用，线程2进入休眠，等待mutex被释放。  
       2)线程1做完数据处理后，调用pthread\_cond\_signal（）唤醒等待队列中某个线程，在本例中也就是线程2。线程1在调用pthread\_mutex\_unlock（）前，因为系统调度的原因，线程2获取使用CPU的权利，那么它就想要开始处理数据，但是在开始处理之前，mutex必须被获取，很遗憾，线程1正在使用mutex，所以线程2被迫再次进入休眠。  
       3)然后就是线程1执行pthread\_mutex\_unlock（）后，线程2方能被再次唤醒。  
       从这里看，使用的效率是比较低的，如果再多线程环境中，这种情况频繁发生的话，是一件比较痛苦的事情。

但是在LinuxThreads或者NPTL里面，就不会有这个问题，因为在Linux 线程中，有两个队列，分别是cond\_wait队列和mutex\_lock队列， cond\_signal只是让线程从cond\_wait队列移到mutex\_lock队列，而不用返回到用户空间，不会有性能的损耗。

所以在Linux中推荐使用这种模式。

之后：

pthread\_mutex\_lock

xxxxxxx

pthread\_mutex\_unlock

pthread\_cond\_signal

优点：不会出现之前说的那个潜在的性能损耗，因为在signal之前就已经释放锁了

缺点：如果unlock和signal之前，有个低优先级的线程正在mutex上等待的话，那么这个低优先级的线程就会抢占高优先级的线程（cond\_wait的线程)，而这在上面的放中间的模式下是不会出现的。

所以，在Linux下最好pthread\_cond\_signal放中间，但从编程规则上说，其他两种都可以