与互斥锁不同，条件变量是用来等待而不是用来上锁的。条件变量用来自动阻塞一个线程，直到某特殊情况发生为止。通常条件变量和互斥锁同时使用。条件变 量分为两部分：条件和变量。条件本身是由互斥量保护的。线程在改变条件状态前先要锁住互斥量。条件变量使我们可以睡眠等待某种条件出现。条件变量是利用线 程间共享的全局变量进行同步的一种机制，主要包括两个动作：一个线程等待“条件变量的条件成立”而挂起;另一个线程使“条件成立”(给出条件成立信号)。 条件的检测是在互斥锁的保护下进行的。如果一个条件为假，一个线程自动阻塞，并释放等待状态改变的互斥锁。如果另一个线程改变了条件，它发信号给关联的条 件变量，唤醒一个或多个等待它的线程，重新获得互斥锁，重新评价条件。如果两线程共享可读写的内存，条件变量可以被用来实现这两线程间的线程同步。

  互斥锁一个明显的缺点是它只有两种状态，锁定和非锁定。而条件变量通过允许线程阻塞和等待另一个线程发送信号的方法弥补了互斥锁的不足，它常和互斥锁 一起使用。使用时，条件变量被用来阻塞一个线程，当条件不满足时，线程往往解开相应的互斥锁并等待条件发生变化。一旦其它的某个线程改变了条件变量，它将 通知相应的条件变量唤醒一个或多个正被此条件变量阻塞的线程。这些线程将重新锁定互斥锁并重新测试条件是否满足。一般来说，条件变量被用来进行线程间的同 步。条件变量只是起阻塞和唤醒线程的作用，具体的判断条件还需用户给出。线程被唤醒后，它将重新检查判断条件是否满足，如果还不满足，一般说来线程应该仍 阻塞在这里，被等待被下一次唤醒。这个过程一般用while语句实现。

  条件变量用pthread\_cond\_t结构体来表示。

  (1)pthread\_cond\_init：初始化一个条件变量，当第二个参数属性为空指针时，函数创建的是一个缺省的条件变量，否则条件变量的属性将由第 二个参数的属性值来决定。不能由多个线程同时初始化一个条件变量。当需要重新初始化或释放一个条件变量时，应用程序必须保证这个条件变量未被使用。

  (2)pthread\_cond\_wait：阻塞在条件变量上，函数将解锁第二个参数指向的互斥锁，并使当前线程阻塞在第一个参数指向的条件变量上。被阻塞 的线程可以被pthread\_cond\_signal、pthread\_cond\_broadcast函数唤醒，也可能在被信号中断后被唤醒。

  一般一个条件表达式都是在一个互斥锁的保护下被检查。当条件表达式未被满足时，线程将仍然阻塞在这个条件变量上。当另一个线程改变了条件的值并向条件 变量发出信号时，等待在这个条件变量上的一个线程或所有线程被唤醒，接着都试图再次占有相应的互斥锁。阻塞在条件变量上的线程被唤醒以后，直到 pthread\_cond\_wait函数返回之前，条件的值都有可能发生变化。所以函数返回以后，在锁定相应的互斥锁之前，必须重新测试条件值。最好的测 试方法是循环调用pthread\_cond\_wait函数，并把满足条件的表达式置为循环的终止条件。阻塞在同一个条件变量上的不同线程被释放的次序是不 一定的。

   pthread\_cond\_wait函数是退出点，如果在调用这个函数时，已有一个挂起的退出请求，且线程允许退出，这个线程将被终止并开始执行善后处理函数，而这时和条件变量相关的互斥锁仍将处在锁定状态。

  pthread\_cond\_signal：解除在条件变量上的阻塞。此函数被用来释放被阻塞在指定条件变量上的一个线程。一般在互斥锁的保护下使用相 应的条件变量，否则对条件变量的解锁有可能发生在锁定条件变量之前，从而造成死锁。唤醒阻塞在条件变量上的所有线程的顺序由调度策略决定。

  pthread\_cond\_timewait：阻塞直到指定时间。函数到了一定的时间，即使条件未发生也会解除阻塞。这个时间由第三个参数指定。

  pthread\_cond\_broadcast：释放阻塞的所有线程。函数唤醒所有被

   pthread\_cond\_wait函数阻塞在某个条件变量上的 线程。当没有线程阻塞在这个条件变量上时，此函数无效。此函数唤醒所有阻塞在某个条件变量上的线程，这些线程被唤醒后将再次竞争相应的互斥锁。

  pthread\_cond\_destroy：释放条件变量。条件变量占用的空间未被释放。pthread\_cond\_wait和pthread\_cond\_timewait一定要在mutex的锁定区域内使用;而 pthread\_cond\_signal和pthread\_cond\_broadcoast无需考虑调用线程是否是mutex的拥有者，可以在lock与 unlock以外的区域调用。

  一个特定条件只能有一个互斥对象，而且条件变量应该表示互斥数据“内部”的一种特殊的条件更改。一个互斥对象可以有许多条件变量，但每个条件变量只能有一个互斥对象

等待条件有两种方式：无条件等待pthread\_cond\_wait()和计时等待pthread\_cond\_timedwait()，其中计时等待方式如果在给定时刻前条件没有满足，则返回ETIMEOUT，结束等待，其中abstime以与time()系统调用相同意义的绝对时间形式出现，0表示格林尼治时间1970年1月1日0时0分0秒。     
  
 无论哪种等待方式，都必须和一个互斥锁配合，以防止多个线程同时请求pthread\_cond\_wait()（或pthread\_cond\_timedwait()，下同）的竞争条件（Race   Condition）。mutex互斥锁必须是普通锁（PTHREAD\_MUTEX\_TIMED\_NP）或者适应锁（PTHREAD\_MUTEX\_ADAPTIVE\_NP），**且在调用pthread\_cond\_wait()前必须由本线程加锁（pthread\_mutex\_lock()），而在更新条件等待队列以前，mutex保持锁定状态，并在线程挂起进入等待前解锁。在条件满足从而离开pthread\_cond\_wait()之前，mutex将被重新加锁，以与进入pthread\_cond\_wait()前的加锁动作对应。**   **执行pthread\_cond\_wait()时自动解锁互斥量(如同执行了 pthread\_unlock\_mutex)，并等待条件变量触发。这时线程挂起，不占用 CPU 时间，直到条件变量被触发。**

**因此，全过程可以描述为：**

**（1）pthread\_mutex\_lock()上锁，**

**（2）pthread\_cond\_wait()等待，等待过程分解为为：解锁--条件满足--加锁**

**（3）pthread\_mutex\_unlock()解锁。**

了解 pthread\_cond\_wait() 的作用非常重要 -- 它是 POSIX 线程信号发送系统的核心，也是最难以理解的部分。  
首先，让我们考虑以下情况：线程为查看已链接列表而锁定了互斥对象，然而该列表恰巧是空的。这一特定线程什么也干不了 -- 其设计意图是从列表中除去节点，但是现在却没有节点。因此，它只能：  
锁定互斥对象时，线程将调用 pthread\_cond\_wait(&mycond,&mymutex)。 pthread\_cond\_wait() 调用相当复杂，因此我们每次只执行它的一个操作。  
 pthread\_cond\_wait() 所做的**第一件事**就是同时对互斥对象解锁（于是其它线程可以修改已链接列表），并等待条件 mycond 发生（这样当 pthread\_cond\_wait() 接收到另一个线程的“信号”时，它将苏醒）。现在互斥对象已被解锁，其它线程可以访问和修改已链接列表，可能还会添加项。 【**要求解锁并阻塞是一个原子操作**】  
 此时，pthread\_cond\_wait() 调用还未返回。对互斥对象解锁会立即发生，但等待条件 mycond 通常是一个阻塞操作，这意味着线程将睡眠，在它苏醒之前不会消耗 CPU 周期。这正是我们期待发生的情况。线程将**一直睡眠，直到特定条件发生**，在这期间不会发生任何浪费 CPU 时间的繁忙查询。从线程的角度来看，它只是在等待 pthread\_cond\_wait() 调用返回。  
 现在继续说明，假设另一个线程（称作“2 号线程”）锁定了 mymutex 并对已链接列表添加了一项。在对互斥对象解锁之后，2 号线程会立即调用函数 pthread\_cond\_broadcast(&mycond)。此操作之后，2 号线程将使所有等待 mycond 条件变量的线程立即苏醒。这意味着第一个线程（仍处于 pthread\_cond\_wait() 调用中）现在将**苏醒**。  
  
 现在，看一下第一个线程发生了什么。您可能会认为在 2 号线程调用 pthread\_cond\_broadcast(&mymutex) 之后，1 号线程的 pthread\_cond\_wait() 会立即返回。不是那样！实际上，pthread\_cond\_wait() 将执行最后一个操作：**重新锁定 mymutex**。一旦 pthread\_cond\_wait() 锁定了互斥对象，那么它将返回并允许 1 号线程继续执行。那时，它可以马上检查列表，查看它所感兴趣的更改。

In Thread1:

pthread\_mutex\_lock(&m\_mutex);     
pthread\_cond\_wait(&m\_cond,&m\_mutex);     
pthread\_mutex\_unlock(&m\_mutex);

In Thread2:

pthread\_mutex\_lock(&m\_mutex);     
pthread\_cond\_signal(&m\_cond);     
pthread\_mutex\_unlock(&m\_mutex);

为什么要与pthread\_mutex 一起使用呢？ 这是为了应对 线程1在调用pthread\_cond\_wait()但线程1还没有进入wait cond的状态的时候，此时线程2调用了 cond\_singal 的情况。 如果不用mutex锁的话，这个cond\_singal就丢失了。加了锁的情况是，线程2必须等到 mutex 被释放（也就是 pthread\_cod\_wait() 释放锁并进入wait\_cond状态 ，此时线程2上锁） 的时候才能调用cond\_singal.

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，但是各有有缺点。