什么是条件变量

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）。
为了防止竞争，条件变量的使用总是和一个互斥锁结合在一起。
条件变量类型为 pthread_cond_t。

条件变量有什么用

使用条件变量可以以原子方式阻塞线程，直到某个特定条件为真为止。条件变量始终与互斥锁一起使用，**对条件的测试是在互斥锁（互斥）的保护下进行的**。

如果条件为假，线程通常会基于条件变量阻塞，并以原子方式释放等待条件变化的互斥锁。如果另一个线程更改了条件，该线程可能会向相关的条件变量发出信号，从而使一个或多个等待的线程执行以下操作：

唤醒

再次获取互斥锁

重新评估条件

条件变量的用法

创建和注销

条件变量和互斥锁一样，都有静态动态两种创建方式，静态方式使用 `PTHREAD_COND_INITIALIZER` 常量，如下：

```
pthread_cond_t cond=PTHREAD_COND_INITIALIZER;   1
```

动态方式调用 `pthread_cond_init()` 函数，定义如下：

```
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t   *cond_a
ttr);
  //成功返回 0，失败返回错误码.
```

尽管 POSIX 标准中为条件变量定义了属性，但在 LinuxThreads 中没有实现，因此 `cond_attr` 值通常为 NULL，且被忽略。

注销一个条件变量需要调用 `pthread_cond_destroy()` 函数，只有在没有线程在该条件变量上等待的时候才能注销这个条件变量，否则返回 EBUSY。因为 Linux 实现的条件变量没有分配什么资源，所以注销动作只包括检查是否有等待线程。
定义如下：

```
int pthread_cond_destroy(pthread_cond_t  *cond) ;
//成功返回0，失败返回错误码.
```

等待和唤醒

等待

两种等待方式，无条件等待 `pthread_cond_wait()` 和计时等待 `pthread_cond_timedwait()`。接口为：

```
int pthread_cond_wait(pthread_cond_t *cond,pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
const struct timespec   *abstime);
//成功返回 0，失败返回错误码.
```

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()`）的竞争条件（Race Condition）。mutex 互斥锁必须是普通锁或者适应锁，且在调用`pthread_cond_wait()`前必须由本线程加锁，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。

`pthread_cond_wait()`函数的返回并不意味着条件的值一定发生了变化，必须重新检查条件的值。

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。如：

```
pthread_mutex_lock();

while (condition_is_false)

    pthread_cond_wait();

pthread_mutex_unlock();
```

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。

注意：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。

唤醒

唤醒条件有两种形式，`pthread_cond_signal()`唤醒一个等待该条件的线程，存在多个等待线程时按入队顺序唤醒其中一个；而`pthread_cond_broadcast()`则唤醒所有等待线程。

```
int pthread_cond_signal(pthread_cond_t *cptr);
int pthread_cond_broadcast (pthread_cond_t * cptr);
//成功返回0，失败返回错误码.
```

必须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，如果线程的调度策略是 SCHED_OTHER 类型的，系统将根据线程的优先级唤醒线程。

如果没有线程被阻塞在条件变量上，那么调用 `pthread_cond_signal()`将没有作用。

由于 pthread_cond_broadcast 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，所以必须小心使用 `pthread_cond_broadcast` 函数。

唤醒丢失问题

在线程未获得相应的互斥锁时调用 `pthread_cond_signal` 或 `pthread_cond_broadcast` 函数可能会引起唤醒丢失问题。

唤醒丢失往往会在下面的情况下发生：

一个线程调用 pthread_cond_signal 或 pthread_cond_broadcast 函数；

另一个线程正处在测试条件变量和调用 pthread_cond_wait 函数之间；

没有线程正在处在阻塞等待的状态下。

实例

条件变量的使用可以分为两部分：

等待线程

使用 pthread_cond_wait 前要先加锁；
pthread_cond_wait 内部会解锁，然后等待条件变量被其它线程激活；
pthread_cond_wait 被激活后会再自动加锁；

激活线程：

加锁（和等待线程用同一个锁）；
pthread_cond_signal 发送信号；
解锁；

激活线程的上面三个操作在运行时间上都在等待线程的 pthread_cond_wait 函数内部。

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t count_lock;
pthread_cond_t count_nonzero;
unsigned count = 0;

void *decrement_count(void *arg)
{
    pthread_mutex_lock(&count_lock);
    printf("decrement_count get count_lock/n");
    while(count == 0)
    {
        printf("decrement_count count == 0 /n");
        printf("decrement_count before cond_wait /n");
        pthread_cond_wait(&count_nonzero, &count_lock);
        printf("decrement_count after cond_wait /n");
    }

    count = count + 1;
    pthread_mutex_unlock(&count_lock);
}

void *increment_count(void *arg)
{
    pthread_mutex_lock(&count_lock);
    printf("increment_count get count_lock /n");
    if(count == 0)
    {
        printf("increment_count before cond_signal /n");
        pthread_cond_signal(&count_nonzero);
```

```
        printf("increment_count after cond_signal /n");
    }

    count = count + 1;
    pthread_mutex_unlock(&count_lock);
}

int main(void)
{
    pthread_t tid1, tid2;

    pthread_mutex_init(&count_lock, NULL);
    pthread_cond_init(&count_nonzero, NULL);

    pthread_create(&tid1, NULL, decrement_count, NULL);
    sleep(2);
    pthread_create(&tid2, NULL, increment_count, NULL);

    sleep(10);
    pthread_exit(0);

    return 0;
}
```

运行结果：

```
decrement_count get count_lock
decrement_count count == 0
decrement_count before cond_wait
increment_count get count_lock
increment_count before cond_signal
increment_count after cond_signal
decrement_count after cond_wait
```