**[条件变量pthread\_cond\_t怎么用](http://blog.csdn.net/zclongembedded/article/details/7337729)**

2012-03-09 17:287309人阅读[评论](http://blog.csdn.net/zclongembedded/article/details/7337729" \l "comments)(3)[收藏](javascript:void(0);)[举报](http://blog.csdn.net/zclongembedded/article/details/7337729#report)

[thread](http://www.csdn.net/tag/thread)[signal](http://www.csdn.net/tag/signal)[null](http://www.csdn.net/tag/null)[join](http://www.csdn.net/tag/join)[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[unix](http://www.csdn.net/tag/unix)

最近看《UNIX环境高级编程》多线程同步，看到他举例说条件变量pthread\_cond\_t怎么用，愣是没有看懂，只好在网上找了份代码，跑了跑，才弄明白

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/hongmy525/article/details/5194006)

1. #include <pthread.h>
2. #include <stdio.h>
3. #include <stdlib.h>
4. pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;/\*初始化互斥锁\*/
5. pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;/\*初始化条件变量\*/
6. void \*thread1(void \*);
7. void \*thread2(void \*);
8. **int** i=1;
9. **int** main(void)
10. {
11. pthread\_t t\_a;
12. pthread\_t t\_b;
13. pthread\_create(&t\_a,NULL,thread1,(void \*)NULL);/\*创建进程t\_a\*/
14. pthread\_create(&t\_b,NULL,thread2,(void \*)NULL); /\*创建进程t\_b\*/
15. pthread\_join(t\_a, NULL);/\*等待进程t\_a结束\*/
16. pthread\_join(t\_b, NULL);/\*等待进程t\_b结束\*/
17. pthread\_mutex\_destroy(&mutex);
18. pthread\_cond\_destroy(&cond);
19. exit(0);
20. }
21. void \*thread1(void \*junk)
22. {
23. for(i=1;i<=6;i++)
24. {
25. pthread\_mutex\_lock(&mutex);/\*锁住互斥量\*/
26. printf("thread1: lock %d/n", \_\_LINE\_\_);
27. if(i%3==0){
28. printf("thread1:signal 1 %d/n", \_\_LINE\_\_);
29. pthread\_cond\_signal(&cond);/\*条件改变，发送信号，通知t\_b进程\*/
30. printf("thread1:signal 2 %d/n", \_\_LINE\_\_);
31. sleep(1);
32. }
33. pthread\_mutex\_unlock(&mutex);/\*解锁互斥量\*/
34. printf("thread1: unlock %d/n/n", \_\_LINE\_\_);
35. sleep(1);
36. }
37. }
38. void \*thread2(void \*junk)
39. {
40. while(i<6)
41. {
42. pthread\_mutex\_lock(&mutex);
43. printf("thread2: lock %d/n", \_\_LINE\_\_);
44. if(i%3!=0){
45. printf("thread2: wait 1 %d/n", \_\_LINE\_\_);
46. pthread\_cond\_wait(&cond,&mutex);/\*解锁mutex，并等待cond改变\*/
47. printf("thread2: wait 2 %d/n", \_\_LINE\_\_);
48. }
49. pthread\_mutex\_unlock(&mutex);
50. printf("thread2: unlock %d/n/n", \_\_LINE\_\_);
51. sleep(1);
52. }
53. }

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/zclongembedded/article/details/7337729)

1. #include <pthread.h>
2. #include <stdio.h>
3. #include <stdlib.h>
4. pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;/\*初始化互斥锁\*/
5. pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;/\*初始化条件变量\*/
6. void \*thread1(void \*);
7. void \*thread2(void \*);
8. **int** i=1;
9. **int** main(void)
10. {
11. pthread\_t t\_a;
12. pthread\_t t\_b;
13. pthread\_create(&t\_a,NULL,thread1,(void \*)NULL);/\*创建进程t\_a\*/
14. pthread\_create(&t\_b,NULL,thread2,(void \*)NULL); /\*创建进程t\_b\*/
15. pthread\_join(t\_a, NULL);/\*等待进程t\_a结束\*/
16. pthread\_join(t\_b, NULL);/\*等待进程t\_b结束\*/
17. pthread\_mutex\_destroy(&mutex);
18. pthread\_cond\_destroy(&cond);
19. exit(0);
20. }
21. void \*thread1(void \*junk)
22. {
23. for(i=1;i<=6;i++)
24. {
25. pthread\_mutex\_lock(&mutex);/\*锁住互斥量\*/
26. printf("thread1: lock %d/n", \_\_LINE\_\_);
27. if(i%3==0){
28. printf("thread1:signal 1 %d/n", \_\_LINE\_\_);
29. pthread\_cond\_signal(&cond);/\*条件改变，发送信号，通知t\_b进程\*/
30. printf("thread1:signal 2 %d/n", \_\_LINE\_\_);
31. sleep(1);
32. }
33. pthread\_mutex\_unlock(&mutex);/\*解锁互斥量\*/
34. printf("thread1: unlock %d/n/n", \_\_LINE\_\_);
35. sleep(1);
36. }
37. }
38. void \*thread2(void \*junk)
39. {
40. while(i<6)
41. {
42. pthread\_mutex\_lock(&mutex);
43. printf("thread2: lock %d/n", \_\_LINE\_\_);
44. if(i%3!=0){
45. printf("thread2: wait 1 %d/n", \_\_LINE\_\_);
46. pthread\_cond\_wait(&cond,&mutex);/\*解锁mutex，并等待cond改变\*/
47. printf("thread2: wait 2 %d/n", \_\_LINE\_\_);
48. }
49. pthread\_mutex\_unlock(&mutex);
50. printf("thread2: unlock %d/n/n", \_\_LINE\_\_);
51. sleep(1);
52. }
53. }



编译：

[X61@horizon threads]$ gcc thread\_cond.c -lpthread -o tcd

以下是程序运行结果：

[X61@horizon threads]$ ./tcd   
thread1: lock 30  
thread1: unlock 40  
  
thread2: lock 52  
thread2: wait 1 55  
thread1: lock 30  
thread1: unlock 40  
  
thread1: lock 30  
thread1:signal 1 33  
thread1:signal 2 35  
thread1: unlock 40  
  
thread2: wait 2 57  
thread2: unlock 61  
  
thread1: lock 30  
thread1: unlock 40  
  
thread2: lock 52  
thread2: wait 1 55  
thread1: lock 30  
thread1: unlock 40  
  
thread1: lock 30  
thread1:signal 1 33  
thread1:signal 2 35  
thread1: unlock 40  
  
thread2: wait 2 57  
thread2: unlock 61

这里的两个关键函数就在pthread\_cond\_wait和pthread\_cond\_signal函数。

本例中：  
  
线程一先执行，获得mutex锁，打印，然后释放mutex锁，然后阻塞自己1秒。  
 **线程二此时和线程一应该是并发的执行** ，这里是一个要点，为什么说是线程此时是并发的执行，因为此时不做任何干涉的话，是没有办法确定是线程一先获得执行还是线程二先获得执行，到底那个线程先获得执行，取决于操作系统的调度，想刻意的让线程2先执行，可以让线程2一出来，先sleep一秒。  
这里并发执行的情况是，线程一先进入循环，然后获得锁，此时估计线程二执行,阻塞在  
pthread\_mutex\_lock(&mutex);  
这行语句中，直到线程1释放mutex锁  
pthread\_mutex\_unlock(&mutex);/\*解锁互斥量\*/  
然后线程二得已执行，获取metux锁，满足if条件，到**pthread\_cond\_wait** (&cond,&mutex);/\*等待\*/  
这里的线程二阻塞，不仅仅是等待cond变量发生改变，**同时释放mutex锁** ，因为当时看书没有注意，所以这里卡了很久。  
mutex锁释放后，线程1终于获得了mutex锁，得已继续运行，当线程1的if（i%3==0）的条件满足后，通过pthread\_cond\_signal发送信号，告诉等待cond的变量的线程（这个情景中是线程二），cond条件变量已经发生了改变。

**不过此时线程二并没有立即得到运行** ，因为线程二还在等待mutex锁的释放，所以线程一继续往下走，直到线程一释放mutex锁，线程二才能停止等待，打印语句，然后往下走通过pthread\_mutex\_unlock(&mutex)释放mutex锁，进入下一个循环。

[**pthread\_cond\_wait()函数的理解**](http://blog.csdn.net/zclongembedded/article/details/7323816)

2012-03-06 10:49977人阅读[评论](http://blog.csdn.net/zclongembedded/article/details/7323816" \l "comments)(0)[收藏](javascript:void(0);)[举报](http://blog.csdn.net/zclongembedded/article/details/7323816#report)

[signal](http://www.csdn.net/tag/signal)[thread](http://www.csdn.net/tag/thread)[null](http://www.csdn.net/tag/null)[api](http://www.csdn.net/tag/api)[struct](http://www.csdn.net/tag/struct)[unix](http://www.csdn.net/tag/unix)

pthread\_cond\_wait(pthread\_cond\_t \*cond, pthread\_mutex\_t\*mutex)函数

传入的参数mutex用于保护条件，因为我们在调用pthread\_cond\_wait时，如果条件不成立我们就进入阻塞，但是进入阻塞这个期间，如果条件变量改变了的话，那我们就漏掉了这个条件。因为这个线程还没有放到等待队列上，所以调用pthread\_cond\_wait前要先锁互斥量，

即调用pthread\_mutex\_lock(),pthread\_cond\_wait在把线程放进阻塞队列后，自动对mutex进行解锁，使得

其它线程可以获得加锁的权利。这样其它线程才能对临界资源进行访问并在适当的时候唤醒这个阻塞的进程。当pthread\_cond\_wait返回的时候又自动给mutex加锁。  
实际上边代码的加解锁过程如下：  
/\*\*\*\*\*\*\*\*\*\*\*\*pthread\_cond\_wait()的使用方法\*\*\*\*\*\*\*\*\*\*/  
pthread\_mutex\_lock(&qlock); /\*lock\*/  
pthread\_cond\_wait(&qready, &qlock); /\*block-->unlock-->wait() return-->lock\*/  
pthread\_mutex\_unlock(&qlock); /\*unlock\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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。   
  
停止并回顾！   
那个过程非常复杂，因此让我们先来回顾一下。第一个线程首先调用：   
  
pthread\_mutex\_lock(&mymutex);   
  
然后，它检查了列表。没有找到感兴趣的东西，于是它调用：   
  
pthread\_cond\_wait(&mycond, &mymutex);   
  
然后，pthread\_cond\_wait() 调用在返回前执行许多操作：   
  
pthread\_mutex\_unlock(&mymutex);   
它对 mymutex 解锁，然后进入睡眠状态，等待 mycond 以接收 POSIX 线程“信号”。一旦接收到“信号”（加引号是因为我们并不是在讨论传统的 UNIX 信号，而是来自 pthread\_cond\_signal() 或 pthread\_cond\_broadcast() 调用的信号），它就会苏醒。但 pthread\_cond\_wait() 没有立即返回 -- 它还要做一件事：重新锁定 mutex：   
  
pthread\_mutex\_lock(&mymutex);   
  
pthread\_cond\_wait() 知道我们在查找 mymutex “背后”的变化，因此它继续操作，为我们锁定互斥对象，然后才返回。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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。   
  
1． 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   
  
动态方式调用pthread\_cond\_init()函数，API定义如下：   
int pthread\_cond\_init(pthread\_cond\_t \*cond, pthread\_condattr\_t \*cond\_attr)   
  
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()，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。API定义如下：   
int pthread\_cond\_destroy(pthread\_cond\_t \*cond)   
  
2． 等待和激发   
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)

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。   
  
激发条件有两种形式，pthread\_cond\_signal()激活一个等待该条件的线程，存在多个等待线程时按入队顺序激活其中一个；而pthread\_cond\_broadcast()则激活所有等待线程。

现在来看一段典型的应用：看注释即可。

**[cpp]** [view plaincopy](http://blog.csdn.net/hairetz/article/details/4535920)

1. #include <pthread.h>
2. #include <unistd.h>
3. **static** pthread\_mutex\_t mtx = PTHREAD\_MUTEX\_INITIALIZER;
4. **static** pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;
5. **struct** node {
6. **int** n\_number;
7. **struct** node \*n\_next;
8. } \*head = NULL;
9. /\*[thread\_func]\*/
10. **static** **void** cleanup\_handler(**void** \*arg)
11. {
12. printf("Cleanup handler of second thread./n");
13. free(arg);
14. (**void**)pthread\_mutex\_unlock(&mtx);
15. }
16. **static** **void** \*thread\_func(**void** \*arg)
17. {
18. **struct** node \*p = NULL;
19. pthread\_cleanup\_push(cleanup\_handler, p);
20. **while** (1) {
21. pthread\_mutex\_lock(&mtx);//这个mutex主要是用来保证pthread\_cond\_wait的并发性
22. **while** (head == NULL)
23. {
24. //这个while要特别说明一下，单个pthread\_cond\_wait功能很完善，为何这里要有一个while (head == NULL)呢？因为pthread\_cond\_wait里的线程可能会被意外唤醒，如果这个时候head != NULL，则不是我们想要的情况。这个时候，应该让线程继续进入pthread\_cond\_wait
25. pthread\_cond\_wait(&cond, &mtx);
26. // pthread\_cond\_wait会先解除之前的pthread\_mutex\_lock锁定的mtx，然后阻塞在等待对列里休眠，直到再次被唤醒（大多数情况下是等待的条件成立而被唤醒，唤醒后，该进程会先锁定先pthread\_mutex\_lock(&mtx);，再读取资源
27. //用这个流程是比较清楚的/\*block-->unlock-->wait() return-->lock\*/
28. }
29. p = head;
30. head = head->n\_next;
31. printf("Got %d from front of queue/n", p->n\_number);
32. free(p);
33. pthread\_mutex\_unlock(&mtx); //临界区数据操作完毕，释放互斥锁
34. }
35. pthread\_cleanup\_pop(0);
36. **return** 0;
37. }
38. **int** main(**void**)
39. {
40. pthread\_t tid;
41. **int** i;
42. **struct** node \*p;
43. pthread\_create(&tid, NULL, thread\_func, NULL); //子线程会一直等待资源，类似生产者和消费者，但是这里的消费者可以是多个消费者，而不仅仅支持普通的单个消费者，这个模型虽然简单，但是很强大
44. /\*[tx6-main]\*/
45. **for** (i = 0; i < 10; i++) {
46. p = malloc(**sizeof**(**struct** node));
47. p->n\_number = i;
48. pthread\_mutex\_lock(&mtx); //需要操作head这个临界资源，先加锁，
49. p->n\_next = head;
50. head = p;
51. pthread\_cond\_signal(&cond);
52. pthread\_mutex\_unlock(&mtx); //解锁
53. sleep(1);
54. }
55. printf("thread 1 wanna end the line.So cancel thread 2./n");
56. pthread\_cancel(tid); //关于pthread\_cancel，有一点额外的说明，它是从外部终止子线程，子线程会在最近的取消点，退出线程，而在我们的代码里，最近的取消点肯定就是pthread\_cond\_wait()了。关于取消点的信息，有兴趣可以google,这里不多说了
57. pthread\_join(tid, NULL);
58. printf("All done -- exiting/n");
59. **return** 0;
60. }