[线程同步:条件变量的使用细节分析](http://blog.chinaunix.net/uid-24410388-id-3758616.html) 2013-06-06 19:47:24

分类： LINUX

原文地址：[线程同步:条件变量的使用细节分析](http://blog.chinaunix.net/uid-28852942-id-3757186.html) 作者：[ifndef](http://blog.chinaunix.net/uid/28852942.html)

如同互斥量和读写锁一样，条件变量也需要初始化和回收  
#include  
int pthread\_cond\_init(pthread\_cond\_t \*restrict cond,  
pthread\_condattr\_t \*restrict attr);  
int pthread\_cond\_destroy(pthread\_cond\_t \*cond);  
  
  
互斥量和读写锁解决了多线程访问共享变量产生的竞争问题，那么条件变量的作用何在呢。  
  
  
**条件变量的作用在于他给多个线程提供了一个汇合的场所**。什么意思呢？  
举个最简单的例子，比如运动会赛跑中，所有选手都会等到发令枪响后才会跑，吧选手比作  
其他的子线程。发令员比作主线程。 那么就是说，所有的子线程现在都在等待主线程给予  
一个可以运行的信号（发令枪响）。这就是这些子线程的汇合点。如果主线程没给信号，那么子线程就会阻塞下去。  
  
大概明白了　条件变量的作用，现在我们来考虑　第一个使用细节上的问题  
  
考虑一个情况：b c d 三个线程都期望在一个条件变量等待主线程发送信号，如果此时条件测试为假，那么三个线程下一步应该是阻塞休眠。  
**但是在判断条件不正确和休眠这之间有个时间窗口**，假如在ｂｃｄ三个线程检查条件为假后，ｃｐｕ切换到另一个线程Ａ，  
在线程Ａ中却使条件变为真了。那么当ｃｐｕ切换回ｂｃｄ线程中时线程还是会休眠。**也就是说在线程检查条件变量和进入休眠等待**  
**条件改变这两个操作之间存在一个时间窗口。这里存在着竞争。**  
  
  
**我们知道互斥量是可以用来解决上面的竞争问题的，所以条件变量本身　是由互斥量来保护的**。  
**既然判断和睡眠是由互斥量来保护从而成为一个原子操作，那么其他改变条件的线程就应该以一致的方式修改条件**  
**也就是说其他线程在改变条件状态前也必须首先锁住互斥量**。（如果修改操作不是用互斥量来保护的话，那么判断和休眠使用互斥量来保护也就没有意义。因为其他线程还是可以在两个操作的空隙中改变条件。但是如果修改操作也使用互斥量。因为判断和休眠之前先加锁了。那么修改操作就只能等到判断失败和休眠两个操作完成才能进行而不会在这之间修改）  
  
下面是提供的接口：  
int pthread\_cond\_wait(pthread\_cond\_t \*restrict cond,  
pthread\_mutex\_t \*restrict mutex);  
int pthread\_cond\_timedwait(pthread\_cond\_t \*restrict cond,  
pthread\_mutex\_t \*restricr mutex,  
const struct timespec \*restrict timeout);  
  
使用pthread\_cond\_wait等待条件变为真，传递给pthread\_cond\_wait的互斥量对条件变量进行保护，调用者把锁住的互斥量传递给函数。  
**（互斥量在传递给函数之前已经调用pthread\_mutex\_lock锁住**）   
**函数把调用线程放到等待条件的线程列表上，然后对互斥量解锁。这样使得判断和休眠成了原子操作**。也就关闭了他们之间的  
时间窗口。  
**当pthread\_cond\_wait返回时，会重新获取互斥量**（互斥量再次被锁住）。  
   
 pthread\_cond\_timedwait与pthread\_cond\_wait的区别在于它指定了休眠的时间，如果时间到了，但是条件还是没有出现，那么pthread\_wait\_timedwait也将  
 重新获取互斥量。然后返回 错误ETIMEDOUT  
 **需要注意的一点是。pthread\_cond\_timedwait的参数timeout不是相对值，而是绝对值。比如你想最多休眠三分钟，那么timeout不是3分钟**  
**而是当前时间加上3分钟。**  
  
 有两个函数可以用来通知线程条件已满足。pthread\_cond\_signal函数将唤醒等待该条件的某个线程。  
 pthread\_cond\_broadcast函数将唤醒等待该条件的所有线程。  
   
 int pthread\_cond\_signal(pthread\_cond\_t \*cond);  
 int pthread\_cond\_broadcast(pthread\_cond\_t \*cond);  
   
 现在我们来看一个具体的例子。  
   
 在下面这个程序中两个子线程在一个条件变量cond上等待条件 i 等于一亿成立。主线程中对 i 做自增操作，当i增加到一亿的时候。条件成立 那么主线程 向条件变量发送信号。那么两个子线程就会从休眠中醒来从而继续运行。  
   
 pthread\_mutex\_t mutex;  
pthread\_cond\_t  cond;  
unsigned long i=0;  
  
  
void \*th1(void \*arg){  
        pthread\_mutex\_lock(&mutex);            //条件变量是由互斥量来保护的  
       
        pthread\_cond\_wait(&cond,&mutex);  
       
        pthread\_mutex\_unlock(&mutex);  
        printf("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\n");  
        pthread\_exit((void \*)0);  
}  
void \*th2(void \*arg){  
        pthread\_mutex\_lock(&mutex);  
       
        pthread\_cond\_wait(&cond,&mutex);  
       
        pthread\_mutex\_unlock(&mutex);  
        printf("bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb\n");  
        pthread\_exit((void \*)0);  
}  
int main(void){  
        pthread\_mutex\_init(&mutex,NULL);  
        pthread\_cond\_init(&cond,NULL);  
  
  
        pthread\_t t1,t2;  
        pthread\_create(&t1,NULL,th1,(void \*)0);  
        pthread\_create(&t2,NULL,th2,(void \*)0);  
  
  
        while(1){         
                pthread\_mutex\_lock(&mutex);          // i为两个子线程等待的条件，就像上面说的修改它也应该先锁住互斥量  
                i++;  
                pthread\_mutex\_unlock(&mutex);  
                if(i==100000000){  
                        pthread\_cond\_broadcast(&cond);  
                        break;  
                }  
        }  
        pthread\_join(t1,NULL);  
        pthread\_join(t2,NULL);  
  
  
        pthread\_cond\_destroy(&cond);  
        pthread\_mutex\_destroy(&mutex);  
        exit(0);  
}  
 程序运行后停顿几秒输出：  
 bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
也就是主线程中不停的 对i 进行自增操作。当i等于一亿的时候，那么条件满足。主线程向条件变量发送信号。两个在条件变量杀上等待  
的子线程收到信号后便开始运行。  
  
  
但是 这个程序是存在问题的。这是我们要说的 第二个关于条件变量上的细节。  
  
**在两个子线程中 我们只是简单的使用了  pthread\_cond\_wait(&cond,&mutex); 在条件变量上休眠等待主线程发送信号过来。**  
**但是在pthread\_cond\_wait调用等待的时候，线程是释放锁的**。（当他返回时才会再次获得锁）。  
那么就存在一个问题  
假想一下。当主线程发送信号过来后。在子线程 在pthread\_cond\_wait上等待发现信号发过来了，那么子线程将醒来并运行（**注意这个时候pthread\_cond\_wait还未返回，那么锁是释放的，因为pthread\_cond\_wait在等待是会释放锁，返回时才会重新获得锁**），那么如果这时候另一个线程改变了 i（对i进行了增减操作。）  
那么此时i 不在是 一亿。但是切换到子线程时他并不知情，他会仍旧认为条件是满足的。**也就是说 我们不应该仅仅依靠pthread\_cond\_wait的返回**  
就认为条件满足。  
所以 上面的程序 中 子线程中的 pthread\_cond\_wait（&cond,&mutex） 应该改为：  
while(i！=100000000){  
pthread\_cond\_wait(&cond,&mutex);  
}  
这样即使 在子线程中 pthread\_cond\_wait返回前还未获得锁的这段空隙有其他线程改变了 i 使条件不在成立。那么当pthread\_cond\_wait返回时他仍旧能发现 i 条件不成立。就会继续调用pthread\_cond\_wait再条件变量上等待。  
  
  
最后再来看个上面的一个问题：  
  
  
在给  在条件变量上等待的线程  发送信号的线程中有下面两个步骤；  
a：  
（1）对互斥量加锁(pthread\_mutex\_lock)  
（2）改变互斥量保护的条件。（对应上面的例子就是在主线程中的 i++ 操作）  
（3）向等待条件的线程发送信号（pthread\_cond\_broadcast）  
（4）对互斥量解锁(pthread\_mutex\_unlock)  
b:  
（1）对互斥量加锁(pthread\_mutex\_lock)  
（2）改变互斥量保护的条件。（对应上面的例子就是在主线程中的 i++ 操作）  
（3）对互斥量解锁(pthread\_mutex\_unlock)  
  （4）向等待条件的线程发送信号（pthread\_cond\_broadcast）  
    
这两种步骤其实都是可以的 但是都存在一些不足。  
在 a 步骤中。 也就是主线程在发送条件成立信号在解锁前。（上面给的例子是在解锁后，在b中会说明）  
那么也就是主线程发送信号后还是持有锁的,**当子线程收到信号后会结束休眠**  
**但是前面说过pthread\_cond\_wait返回时会再次获得锁，但是主线程还并未释放**  
**锁，所以会造成子线程收到信号开始运行并立即阻塞**。  
  
在b步骤中。  主线程在释放锁后才发送信号。我们上面的例子就是这么做的。但是这也存在一个问题  
但释放锁后，另一个线程很可能会在发送信号之前获得锁并修改 变量i 导致条件再次不成立  
但是会到主线程中他却并不知情，导致仍会发送信号给子线程。子线程认为条件满足  
从休眠中醒来开始运行，但此时条件是不满足的。  
所以在上面的例子中我们将  
pthread\_cond\_wait（&cond,&mutex） 改为：  
while(i！=100000000){  
pthread\_cond\_wait(&cond,&mutex);  
}  
让子线程醒来后再次判断条件是否成立。这样就可以避免了上面的问题。  
  
  
总结一下： 条**件变量的要点在于 他提供了一个让多个线程汇合的点。但是条件变量本身是需要**  
**互斥量来进行保护的。**  
**我们不能仅仅根据pthread\_cond\_wait返回就认为条件满足了。而需再次判断条件是否正确**