[linux同步机制之wait\_event和wake\_up](http://blog.csdn.net/liuxd3000/article/details/17913221)

2014-01-06 10:38 4101人阅读 [评论](http://blog.csdn.net/liuxd3000/article/details/17913221#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/liuxd3000/article/details/17913221#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

linux（75） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

目录[(?)[+]](http://blog.csdn.net/liuxd3000/article/details/17913221)

1. 关于 wait\_event\_interruptible() 和 wake\_up()的使用   
     
读一下wait\_event\_interruptible()的源码，不难发现这个函数先将   
当前进程的状态设置成TASK\_INTERRUPTIBLE，然后调用schedule()，   
而schedule()会将位于TASK\_INTERRUPTIBLE状态的当前进程从runqueue   
队列中删除。从runqueue队列中删除的结果是，当前这个进程将不再参   
与调度，除非通过其他函数将这个进程重新放入这个runqueue队列中，   
这就是wake\_up()的作用了。   
    
由于这一段代码位于一个由condition控制的for(;;)循环中，所以当由   
shedule()返回时(当然是被wake\_up之后，通过其他进程的schedule()而   
再次调度本进程)，如果条件condition不满足，本进程将自动再次被设   
置为TASK\_INTERRUPTIBLE状态，接下来执行schedule()的结果是再次被   
从runqueue队列中删除。这时候就需要再次通过wake\_up重新添加到   
runqueue队列中。   
    
如此反复，直到condition为真的时候被wake\_up.   
    
可见，成功地唤醒一个被wait\_event\_interruptible()的进程，需要满足：   
    
   在 1)condition为真的前提下，2) 调用wake\_up()。   
  
所以，如果你仅仅修改condition，那么只是满足其中一个条件，这个时候，   
被wait\_event\_interruptible()起来的进程尚未位于runqueue队列中，因   
此不会被 schedule。这个时候只要wake\_up一下就立刻会重新进入运行调度。   
    
2. 关于wait\_event\_interruptible的返回值   
    
根据 wait\_event\_interruptible 的宏定义知：   
    
   1) 条件condition为真时调用这个函数将直接返回0，而当前进程不会   
      被 wait\_event\_interruptible和从runqueue队列中删除。   
    
   2) 如果要被wait\_event\_interruptible的当前进程有nonblocked pending   
      signals, 那么会直接返回-ERESTARTSYS(i.e. -512)，当前进程不会   
      被wait\_event\_interruptible 和从runqueue队列中删除。   
    
   3) 其他情况下，当前进程会被正常的wait\_event\_interruptible，并从   
      runqueue队列中删除，进入TASK\_INTERRUPTIBLE状态退出运行调度，   
      直到再次被唤醒加入runqueue队列中后而参与调度，将正常返回0。   
    
附1：wait\_event\_interruptible  宏   
    
#define wait\_event\_interruptible(wq, condition)    \   
({                                                 \   
     int \_\_ret = 0;                                  \   
     if (!(condition))                               \   
      \_\_wait\_event\_interruptible(wq, condition, \_\_ret); \   
      \_\_ret;                                         \   
})   
    
注: C语言中{a,b, ..., x}的的值等于最后一项，即x，因此上述   
宏的值是 \_\_ret。   
    
    
附2：wait\_event\_interruptible()和 wake\_up的等效代码   
    
wait\_event\_interruptible(wq, condition) /\*等效没有考虑返回值\*/   
{   
     if (!(condition))   
     {   
         wait\_queue\_t \_ \_wait;   
         init\_waitqueue\_entry(&\_ \_wait, current);   
         add\_wait\_queue(&wq, &\_ \_wait);   
         for (;;)   
         {   
            set\_current\_state(TASK\_INTERRUPTIBLE);   
            if (condition)   
            break;   
            schedule();  /\* implicit call: del\_from\_runqueue(current)\*/   
         }   
         current->state = TASK\_RUNNING;   
         remove\_wait\_queue(&wq, &\_ \_wait);   
      }   
}   
    
    
void wake\_up(wait\_queue\_head\_t \*q)   
{   
      struct list\_head \*tmp;   
      wait\_queue\_t \*curr;   
      list\_for\_each(tmp, &q->task\_list)   
      {   
        curr = list\_entry(tmp, wait\_queue\_t, task\_list);   
        wake\_up\_process(curr->task);   
        /\* implicit call: add\_to\_runqueue(curr->task);\*/   
        if (curr->flags)   
          break;   
      }

}

[Linux中等待队列机制分析](http://blog.csdn.net/lhf_tiger/article/details/7173572" \t "_blank)

分类： [Linux 系列](http://blog.csdn.net/lhf_tiger/article/category/858937)2012-01-03 21:40 210人阅读 [评论](http://blog.csdn.net/lhf_tiger/article/details/7173572#comments)(0) 收藏 [举报](http://blog.csdn.net/lhf_tiger/article/details/7173572#report)

[linux](http://www.csdn.net/tag/linux)[任务](http://www.csdn.net/tag/%e4%bb%bb%e5%8a%a1)[linux内核](http://www.csdn.net/tag/linux%e5%86%85%e6%a0%b8)[struct](http://www.csdn.net/tag/struct)[任务调度](http://www.csdn.net/tag/%e4%bb%bb%e5%8a%a1%e8%b0%83%e5%ba%a6)[list](http://www.csdn.net/tag/list)

目录[(?)[+]](http://blog.csdn.net/lhf_tiger/article/details/7173572)

**什么是等待队列？**

      在软件开发中任务经常由于某种条件没有得到满足而不得不进入睡眠状态，然后等待条件得到满足的时候再继续运行，进入运行状态。这种需求需要等待队列机制的支持。Linux中提供了等待队列的机制，该机制在内核中应用很广泛。

      在Linux内核中使用等待队列的过程很简单，首先定义一个wait\_queue\_head，然后如果一个task想等待某种事件，那么调用wait\_event（等待队列，事件）就可以了。

**Linux中等待队列的实现**

      等待队列应用广泛，但是内核实现却十分简单。其涉及到两个比较重要的数据结构：

1)       \_\_wait\_queue\_head，该结构描述了等待队列的链头，其包含一个链表和一个原子锁，结构定义如下：

struct\_\_wait\_queue\_head {

             spinlock\_tlock;

             struct list\_headtask\_list;

};

2)       \_\_wait\_queue，该结构是对一个等待任务的抽象。每个等待任务都会抽象成一个wait\_queue，并且挂载到wait\_queue\_head上。该结构定义如下：

struct\_\_wait\_queue {

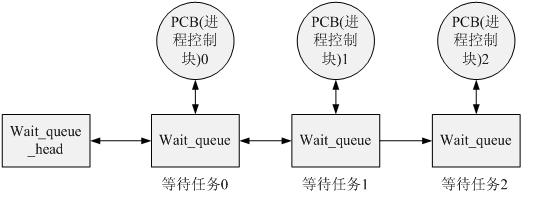
      unsigned intflags;

      void\*private;

      wait\_queue\_func\_tfunc;

      struct list\_headtask\_list;

};

       Linux中等待队列的实现思想如下图所示，当一个任务需要在某个wait\_queue\_head上睡眠时，将自己的进程控制块信息封装到wait\_queue中，然后挂载到wait\_queue的链表中，执行调度睡眠。当某些事件发生后，另一个任务（进程）会唤醒wait\_queue\_head上的某个或者所有任务，唤醒工作也就是将等待队列中的任务设置为可调度的状态，并且从队列中删除。  
  
[](http://space.ednchina.com/Upload/Blog/2008/7/11/61ed3ba8-b428-42d0-adfc-89c427234a75.JPG)

   使用等待队列时首先需要定义一个wait\_queue\_head，这可以通过DECLARE\_WAIT\_QUEUE\_HEAD宏来完成，这是静态定义的方法。该宏会定义一个wait\_queue\_head，并且初始化结构中的锁以及等待队列。当然，动态初始化的方法也很简单，初始化一下锁及队列就可以了。

      一个任务需要等待某一事件的发生时，通常调用wait\_event，该函数会定义一个wait\_queue，描述等待任务，并且用当前的进程描述块初始化wait\_queue，然后将wait\_queue加入到wait\_queue\_head中。函数实现流程说明如下：

1、 用当前的进程描述块（PCB）初始化一个wait\_queue描述的等待任务。

2、 在等待队列锁资源的保护下，将等待任务加入等待队列。

3、 判断等待条件是否满足，如果满足，那么将等待任务从队列中移出，退出函数。

4、 如果条件不满足，那么任务调度，将CPU资源交与其它任务。

5、 当睡眠任务被唤醒之后，需要重复（2）、（3）步骤，如果确认条件满足，退出等待事件函数。

**等待队列编程接口**

|  |  |  |
| --- | --- | --- |
| 序号 | 编程接口 | 使用说明 |
| 1 | wait\_event | 这是一个宏，让当前任务处于等待事件状态。输入参数如下：  @wq：等待队列  @conditions：等待条件 |
| 2 | wait\_event\_timeout | 功能与wait\_event类似，多了一个超时机制。参数中多了一项超时时间。 |
| 3 | wait\_event\_interruptible | 这是一个宏，与前两个宏相比，该宏定义的等待能够被消息唤醒。如果被消息唤醒，那么返回-ERESTARTSYS。输入参数如下：  @wq：等待队列  @condition：等待条件  @rt：返回值 |
| 4 | wait\_event\_interruptible\_timeout | 与（3）相比，多了超时机制 |
| 5 | wake\_up | 唤醒等待队列中的一个任务 |
| 6 | wake\_up\_all | 唤醒等待队列中的所有任务 |