**sleep\_on()函数分析**

2017年03月20日 09:57:42 [乘风life](https://me.csdn.net/r21nn) 阅读数：528

 void sleep\_on(struct task\_struct \*\*p)  
 {  
     struct task\_struct \*tmp;  
     if (!p)   //若指针无效，则退出  
         return;  
     if (current == &(init\_task.task)) //若当前任务是任务0则死机  
         panic("task[0] trying to sleep");  
     tmp = \*p; //让tmp指向已经在等待队列上的任务  
     \*p = current; //将睡眠队列头的等待指针指向当前任务  
     current->state = TASK\_UNINTERRUPTIBLE;  //将当前任务置为不可中断的等待状态  
     schedule();   //重新调度  
// 只有当这个等待任务被唤醒时，调度程序才又返回到这里，则表示进程已被明确地唤醒。  
     if (tmp)  

         tmp->state=0;  // 若在其前还存在等待的任务，则也将其置为就绪状态（唤醒）。

 }

在几个进程为等待同一资源而多次调用该函数时，程序就隐式地构筑出一个等待队列。

在插入等待队列后，sleep\_on()函数就会调用schedule()函数去执行别的进程。当进程被唤醒而重新  
执行时就会执行后续的语句，把比它早进入等待队列的一个进程唤醒。

例如，当task1，task2，task3因为资源暂时不可用而先后调用sleep\_on()函数时，且task1为第一个因资源

不可用而调用sleep\_on()的进程，则在这三个进程中的sleep\_on()函数内tmp和p的执行如下

task1：sleep\_on(&p)

{

tmp=NULL;  因为第一次调用，所以之前的等待队列为空

\*p=current=task1;

schedule();

if (tmp) task1被唤醒，schedule()返回，此时tmp==null,下面代码不执行。

tmp->state=0;

}

task2：sleep\_on(&p)

{

tmp=task1;

\*p=current=task2;

schedule();

if (tmp) task2被唤醒，schedule()返回，执行下面代码，唤醒task1;

tmp->state=0;

}

task3：sleep\_on(&p)

{

tmp=task2;

\*p=current=task3;

schedule();

if (tmp)   调用wake\_up()时首先唤醒task3，schedule()返回，执行下面代码，唤醒task2.

tmp->state=0;

}

当该资源可用时，系统调用wake\_up(),此函数定义如下

void wake\_up(struct task\_struct \*\*p)  
 {  
  if (p && \*p) {  
 (\*\*p).state=0; // 置为就绪（可运行）状态。  
 \*p=NULL;  
}  
 }

可知调用wake\_up(&p)时，此时的p指向最后一次调用sleep\_on()的task，即为task3,所以task3被唤醒，task3被唤醒，task3中的schedule()返回

执行后续代码，唤醒task2，于是task2中的schedule()返回，执行后续代码，唤醒task1，于是task1中的schedule()返回，task1中的tmp==NULL，

不执行后续代码，到此，因等待统一资源的三个进程都被唤醒；

关于schedule()函数的分析可见http://blog.csdn.net/r21nn/article/details/63694793