

```
void sleep_on(struct task_struct **p)
{
    struct task_struct *tmp;

    if (!p)
        return;
    if (current == &(init_task.task))
        panic("task[0] trying to sleep");
    tmp = *p;
    *p = current;
    current->state = TASK_UNINTERRUPTIBLE;
    /*添加内容*/
    fprintf(3, "%d\t%c\t%d\n", current->pid, 'W', jiffies); //向log文件输出进入阻塞态的时间
    /*添加内容*/
    schedule();
}
```

```
void interruptible_sleep_on(struct task_struct **p)
{
    struct task_struct *tmp;

    if (!p)
        return;
    if (current == &(init_task.task))
        panic("task[0] trying to sleep");
    tmp = *p;
    *p = current; //把current放在等待队列表头

repeat: current->state = TASK_INTERRUPTIBLE;
/*添加内容*/
    if (*p==current) //为什么要这个if条件 看下面的解释
        fprintf(3, "%d\t%c\t%d\n", current->pid, 'W', jiffies); //向log文件输出进入阻塞态的时间
/*添加内容*/
    schedule();
}
```

```

    }
}
if (flag) {
    if (options & WNOHANG)
        return 0;
    current->state=TASK_INTERRUPTIBLE;
/*添加内容*/
    fprintf(3, "%d\t%c\t%d\n", current->pid, 'W', jiffies); //向log文件输出进入阻塞态的时间
/*添加内容*/
    schedule();
    if (!(current->signal &= ~(1<<(SIGCHLD-1))))
}
```

(2)sys_pause()中:要处理0进程的主动睡眠,详见截图

```
int sys_pause(void)
{
    /**
     * 当系统无事可做时 进程0会不停地调用sys_pause()...
     * 以激活调度算法 此时它的状态可以是等待态...等待有其它可运行的进程.
     * 也可以叫运行态. 因为它是唯一的一个在CPU上运行的进程 只不过运行的效果是等待.
     * 也就是说 此时进程0是处于运行态的, 虽然它调用了sys_pause.
     * 100行的对schedule()的注释也说进程0永不sleep
     */
    current->state = TASK_INTERRUPTIBLE;
/*添加内容*/
    if (current->pid!=0)
        fprintf(3, "%d\t%c\t%d\n", current->pid, 'W', jiffies);
/*添加内容*/
    schedule();
    return 0;
}
```

3.从"睡眠态"到"就绪态"

(1)sleep_on()和wake_up()中:对sleep_on()中的tmp的处理不太理解,仍是无脑添加.

```

    if (tmp)
    {
        tmp->state=0; //0就是TASK_RUNNING,这里是唤醒队列中的上一个(tmp)睡眠进程
/*添加内容*/
        fprintf(3, "%d\t%c\t%d\n", tmp->pid, 'J', jiffies); //向log文件输出进入就绪态的时间
/*添加内容*/
    }
}
```

```
void wake_up(struct task_struct **p)
{
    if (p && *p) {
        (**p).state=0;
/*添加内容*/
        fprintf(3, "%d\t%c\t%d\n", (**p).pid, 'J', jiffies); //向log文件输出进入就绪态的时间
/*添加内容*/
        *p=NULL; //根据225行 把*p=NULL这句删掉
    }
}
```

(2)ininterruptible_sleep_on()中:详见截图.

```

tmp = *p;
*p=current; //把current放在等待队列表头

repeat: current->state = TASK_INTERRUPTIBLE;
/*添加内容*/
    if (*p==current) //为什么要这个if条件 看221行的解释
        fprintf(3, "%d\t%c\t%d\n", current->pid, 'W', jiffies); //向log文件输出进入阻塞态的时间
/*添加内容*/
    schedule();
}
if (*p && *p != current) //只有当这个等待任务被唤醒时 程序才会在这里继续执行.
    (**p).state=0; //如果队列表头和刚唤醒的进程current不是同一个 说明当前任务被放入队列表后 又有新的任务被放入队列表
    //这样就把队列表头进程唤醒(优先处理队列表头) 所以goto repeat,让当前进程(current)仍然等待,此时 无需再往log中打印睡眠信息
/*添加内容*/
    fprintf(3, "%d\t%c\t%d\n", (**p).pid, 'J', jiffies); //向log文件输出进入就绪态的时间
/*添加内容*/
    goto repeat;
}
```

(4)在schedule()中用信号唤醒TASK_INTERRUPTIBLE

```

for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
    if (*p) {
        if ((*p)->alarm && (*p)->alarm < jiffies) {
            (*p)->signal |= (1<<(SIGALRM-1));
            (*p)->alarm = 0;
        }
        if (((*p)->signal & ~(BLOCKABLE & (*p)->blocked)) &&
            (*p)->state==TASK_INTERRUPTIBLE)
        {
            (*p)->state=TASK_RUNNING; //TASK_INTERRUPTIBLE可被用信号唤醒
/*添加内容*/
            fprintf(3, "%d\t%c\t%d\n", (*p)->pid, 'J', jiffies); //向log文件输出进入就绪态的时间
/*添加内容*/
        }
    }
}
```

4.从"运行态"到"就绪态"

这个状态变化发生在schedule()中,详见截图.

```

    if (c break;
    for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
    if (*p)
        (*p)->counter = ((*p)->counter >> 1) +
        (*p)->priority;
/*添加内容*/
    if (task(next)->pid != current->pid) //调度算法有可能会把next调度为当前正在运行的进程 相当于当前进程的状态没有变化.
    //这种情况下 下面的switch_to(next)什么都不做(见其源码注释)
    if (current->state==TASK_RUNNING) //如果当前运行进程是被设为睡眠态后再调用的schedule() 就不可以打印下面这句了
        fprintf(3, "%d\t%c\t%d\n", current->pid, 'J', jiffies); //当前运行的进程current要被抢占了,所以向log写入"从运行态到就绪态"的信息
        fprintf(3, "%d\t%c\t%d\n", task(next)->pid, 'R', jiffies); //task是一个队列 其中存放了所有就绪态的进程,而next就是这个队列的下标
/*添加内容*/
    switch_to(next); //注意switch_to(next)的作用是把当前任务指针current指向任务号为next的任务,其一旦被执行,就马上切换到next中去运行
}
```

5.从"就绪态"到"运行态"

这个也是发生在schedule()中,注意:switch_to()是一个宏,其展开为一段内嵌汇编,且其一旦被执行,立刻就会切换到next指代的进程中去 所以不要把fprintf()放在switch_to()后面,详见4中的截图.

二.实验报告

1.谈谈从程序设计者的角度看,单进程编程和多进程编程最大的区别是什么?

答:我觉得最大的区别是后者(1)具有并行性,提高的CPU的使用效率;(2)需要调度;(3)进程间的切换更加频繁得多;(4)需要保存各个进程的上下文环境.

2.我按着实验指导书上说的,更改了include/linux/sched,h中的宏INIT_TASK中的counter,priority(它们的原本的值都为15):

```
#define INIT_TASK \
/* state etc. 分别对应state,counter,priority; */ { 0,15,15 \
/* signals */ 0,{},{},0, \
/* ec brk... */ 0,0,0,0,0,0, \
/* pid etc.. */ 0,-1,0,0,0, \
/* uid etc */ 0,0,0,0,0,0, \
/* alarm */ 0,0,0,0,0,0, \
/* math */ 0, \
/* fs info */ -1,0022,NULL,NULL,NULL,0, \
/* filp */ {NULL}, \
}
```

分别把进程0的counter和priority改成了7,30,1500,和原本的值15进行比较

15:

Process	Turnaround	Waiting	CPU Burst	I/O Burst
7	1006	0	500	505
8	1005	0	500	505
9	1006	0	500	505
10	1006	0	500	505
Average:	1005.75	0.00		
Throughout:	0.10/s			

-----< COOL GRAPHIC OF SCHEDULER >-----

[Symbol]	[Meaning]
number	PID or tick
"-"	New or Exit
"#"	Running
" "	Ready
":"	Waiting
/	Running with
"+" -	Ready
	\and/or Waiting

7:

Process	Turnaround	Waiting	CPU Burst	I/O Burst
7	1005	0	500	505
8	1005	0	500	505
9	1006	0	500	505
10	1006	0	500	505
Average:	1005.50	0.00		
Throughout:	0.10/s			

-----< COOL GRAPHIC OF SCHEDULER >-----

[Symbol]	[Meaning]
number	PID or tick
"-"	New or Exit
"#"	Running
" "	Ready
":"	Waiting
/	Running with
"+" -	Ready
	\and/or Waiting