边城浪子L33 2016-07-02 15:11 Linux 计算机专业课

实验代码

“操作系统原理与实践”实验报告

进程运行轨迹的跟踪与统计

1 内核代码修改

1.1对fork.c的修改：

```
int copy_process int long long long long long
        long long long
        long long long
        long long long long long
{
    struct task_struct *p;
    int i;
    struct file *f;

    p = ( struct task_struct *) get_free_page();
    if (!p)
        return - EAGAIN;
    task[nr] = p;
    *p = *current; /* NOTE! this doesn't copy the supervisor stack */
    p->state = TASK_UNINTERRUPTIBLE;
    p->pid = last_pid;

        ......

set_tss_desc(gdt+(nr<< 1)+FIRST_TSS_ENTRY,&(p ->tss));
    set_ldt_desc(gdt+(nr<< 1)+FIRST_LDT_ENTRY,&(p ->ldt));

        /****************/
        fprintk( 3, "%ld\t%c\t%ld\n" , last_pid, 'N' , jiffies); // 添加新建标记
        fprintk( 3, "%ld\t%c\t%ld\n" , last_pid, 'J' , jiffies); // 进入就绪标记
        /****************/

    p->state = TASK_RUNNING; /* do this last, just in case */
    return last_pid;
}
```

1.2 对sched.c的修改 1.2.1 void schedule(void)

```
void schedule(void)
{
    int i,next,c;
    struct task_struct ** p;

        ......

    while (1) {
        c = - 1;
        ......
        for (p = &LAST_TASK ; p > &FIRST_TASK ; -- p)
            if (*p)
                (*p) ->counter = ((*p) ->counter >> 1) +
                        (*p) ->priority;
    }

    /********添加就绪或者是运行标记，若当前进程和下一个进程不相同则说明时间片用完，处于就绪状态，下一
进程准备运行*******/
    if (task[next] ->pid != current ->pid)
    {
        if (current ->state == TASK_RUNNING)
        {
            fprintk( 3, "%ld\t%c\t%ld\n" , current ->pid, 'J' , jiffies);
        }
        fprintk( 3, "%ld\t%c\t%ld\n" , task[next] ->pid, 'R' , jiffies);
    }
    /*******修改完成********/

    switch_to(next);
}
```

1.2.2 int sys_pause(void)

```
int sys_pause(void)
{

    /*****进入阻塞标记****/
    if (current ->state !=TASK_INTERRUPTIBLE)
        fprintk( 3, "%ld\t%c\t%ld\n" , current ->pid, 'W' , jiffies);
    /*****完成添加****/

    current ->state = TASK_INTERRUPTIBLE;
    schedule();
    return 0;
}
```

1.2.3 void sleep_on(struct task_struct **p)

```
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;

    /*******阻塞标记*******/
    if (current ->state != TASK_UNINTERRUPTIBLE)
        fprintk( 3, "%ld\t%c\t%ld\n" , current ->pid, 'W' , jiffies);
    /********添加完成******/

    current ->state = TASK_UNINTERRUPTIBLE;
    schedule();
    if (tmp)
    {
                /******/
        if (tmp ->state != TASK_RUNNING)
// 若唤醒但未能执行，则处于就绪状态
            fprintk( 3, "%ld\t%c\t%ld\n" , current ->pid, 'J' , jiffies); /
完成添加
               /******/

        tmp ->state= 0;
    }
}
```

1.2.4 void interruptible_sleep_on(struct task_struct **p)

```
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;
repeat:

    /********/
    if (current ->state = TASK_INTERRUPTIBLE)
        fprintk( 3, "%ld\t%c\t%ld\n" , current ->pid, 'W' , jiffies);
    /********/

    current ->state = TASK_INTERRUPTIBLE;
    schedule();
    if (*p && *p != current) {

        /*****添加就绪标记***/
        if ((*p) ->state != 0)
            fprintk( 3, "%ld\t%c\t%ld\n" , (**p).pid, 'J' , jiffies);
        /*****完成***/

        (**p).state= 0;
        goto repeat;
    }
    *p=NULL;
    if (tmp)
    {

        /****添加就绪标记****/
        if (tmp ->state != 0)
            fprintk( 3, "%ld\t%c\t%ld\n" , tmp ->pid, 'J' , jiffies);
        /*****完成***/

        tmp ->state =0;
    }
}
```

1.2.5 void wake_up(struct task_struct **p)

```
void wake_up struct
{
    if (p && *p) {

        /*****添加标记***/
        if ((**p).state!= 0)
            fprintk( 3, "%ld\t%c\t%ld\n" , (**p).pid, 'J' , jiffies);
        /***完成**/

        (**p).state= 0;
        *p= NULL;
    }
}
```

1.3 修改exit.c 1.3.1 int do_exit(long code)

```
int do_exit(long code)
{
    int i;
    free_page_tables(get_base(current ->ldt[ 1]),get_limit( 0x0f ));
    free_page_tables(get_base(current ->ldt[ 2]),get_limit( 0x17 ));

        ......

    if (current ->leader)
        kill_session();
    current ->state = TASK_ZOMBIE;

    /****添加退出标记****/
    fprintk( 3, "%ld\t%c\t%ld\n" , current ->pid, 'E' , jiffies);
    /****完成****/

    current ->exit_code = code;
    tell_father(current ->father);
    schedule();
    return (- 1); /* just to suppress warnings */
}
```

1.3.2 int sys_waitpid(pid_t pid,unsigned long *stat_addr, int options)

```
int sys_waitpid pid_t unsigned long int
{
    int flag, code;
    struct task_struct ** p;

    verify_area(stat_addr, 4);
repeat:
    flag= 0;

        ......

        if (options & WNOHANG)
            return 0;
        current ->state=TASK_INTERRUPTIBLE;

        /****添加阻塞标记****/
        fprintk( 3, "%ld\t%c\t%ld\n" , current ->pid, 'W' , jiffies);
        /****完成****/

        schedule();
        if (!(current ->signal &= ~( 1<<(SIGCHLD - 1))))
            goto repeat;
        else
            return - EINTR;
    }
    return - ECHILD;
}
```

1.4 修改printk.c 直接使用老师提供的写文件程序

```
#include "linux/sched.h"
#include "sys/stat.h"

static char logbuf[ 1024 ];
int fprintk int const char
{
    va_list args;
    int count;
    struct file * file;
    struct m_inode * inode;

    va_start(args, fmt);
    count= vsprintf (logbuf, fmt, args);
    va_end(args);

    if (fd < 3)
    {
        __asm__( "push %%fs\n\t"
            "push %%ds\n\t"
            "pop %%fs\n\t"
            "pushl %0\n\t"
            "pushl $logbuf\n\t"
            "pushl %1\n\t"
            "call sys_write\n\t"
            "addl $8,%%esp\n\t"
            "popl %0\n\t"
            "pop %%fs"
            :: "r" (count), "r" (fd): "ax" , "cx" , "dx" );
    }
    else
    {
        if (!(file=task[ 0] ->filp[fd]))
            return 0;
        inode=file ->f_inode;

        __asm__( "push %%fs\n\t"
            "push %%ds\n\t"
            "pop %%fs\n\t"
            "pushl %0\n\t"
            "pushl $logbuf\n\t"
            "pushl %1\n\t"
            "pushl %2\n\t"
            "call file_write\n\t"
            "addl $12,%%esp\n\t"
            "popl %0\n\t"
            "pop %%fs"
            :: "r" (count), "r" (file), "r" (inode): "ax" , "cx" , "dx" );
    }
    return count;
}
```

2 对初始化程序main.c的修改

```
void main void /* This really IS void, no error here. */
{
        ......
    floppy_init();
    sti();
    move_to_user_mode();

    /****将这段代码提前至此****/
    setup(( void *) &drive_info);
    ( void ) open( "/dev/tty0" ,O_RDWR,0);
    ( void ) dup( 0);
    ( void ) dup( 0);
    ( void ) open( "/var/process.log" , O_CREAT|O_TRUNC|O_WRONLY, 0666 );
    /****完成****/

    if (!fork()) { /* we count on this going ok */
        init();
    }

   ......

void init void
{
    int pid,i;
/*注释掉这段代码*/
    //setup((void *) &drive_info);
    //(void) open("/dev/tty0",O_RDWR,0);
    //(void) dup(0);
    //(void) dup(0);
/*完成注释*/
    printf ( "%d buffers = %d bytes buffer space\n\r" ,NR_BUFFERS,
        NR_BUFFERS*BLOCK_SIZE);
    printf ( "Free mem: %d bytes\n\r" ,memory_end - main_memory_start);
    ......
    _exit( 0); /* NOTE! _exit, not exit() */
}
```

3 给定任务 修改/home/teacher 下的proccess.c给出测试的任务

```
 void cpuio_bound( int last , int cpu_time, int io_time);
int main( int argc, char * argv[])
{
    if (! fork ())
        cpuio_bound( 10, 0, 1);
    if (! fork ())
        cpuio_bound( 10, 1, 0);
    if (! fork ())
        cpuio_bound( 10, 1, 1);
    if (! fork ())
        cpuio_bound( 10, 5, 0);
    if (! fork ())
        cpuio_bound( 10, 0, 5);
    if (! fork ())
        cpuio_bound( 10, 4, 4);
    if (! fork ())
        cpuio_bound( 5, 0, 1);
    if (! fork ())
        cpuio_bound( 5, 1, 0);
    if (! fork ())
        cpuio_bound( 5, 1, 1);
    return 0;
}
```

4 修改时间片 打开sched.h头文件，修改

```
#define HZ 100
```

把100改为200(或者是50,随意)

5 进行实验 挂载虚拟机硬盘把process.c拷贝到~/oslab/hdc/usr/root/，运行虚拟机编译process，然后执行，得到日志process.log，将其拷贝到ubuntu下。修改时间片，编译内核，同样得到日志２。 日志１

```
1   N   48
1   J   48
0   J   48
1   R   48
2   N   49
2   J   49
1   W   49
2   R   49
3   N   64
3   J   64
2   J   64
3   R   64
3   W   68
2   R   68
2   E   73
1   R   73
4   N   74
4   J   74
1   W   74
4   R   74
5   N   106
5   J   107
4   W   107
5   R   107
4   J   109
5   E   109
4   R   109
4   W   115
0   R   115
0   W   115
4   J   291
4   R   291
4   W   291
0   R   291
4   J   302
4   R   302
4   W   302
0   R   302
4   J   319
4   R   319
4   W   320
0   R   320
4   J   330
4   R   330
4   W   330
0   R   330
.....
```

日志２

```
1   N   95
1   J   95
0   J   95
1   R   95
2   N   96
2   J   96
1   W   96
2   R   96
3   N   126
3   J   126
2   J   127
3   R   127
3   W   139
2   R   139
2   E   146
1   R   146
4   N   147
4   J   147
1   W   147
4   R   147
5   N   212
5   J   212
4   W   213
5   R   213
4   J   217
5   E   218
4   R   218
4   W   229
0   R   229
0   W   229
4   J   547
4   R   547
4   W   547
0   R   547
4   J   567
4   R   567
4   W   567
0   R   568
4   J   596
4   R   596
4   W   596
0   R   596
4   J   614
4   R   614
4   W   614
0   R   614
.....
```

使用老师提供的统计程序stat_log.py进行分析 日志１的结果（时间片１００ＨＺ）：

```
(Unit:    tick)
Process    Turnaround    Waiting    CPU Burst    I/O Burst
      0           551         67            8          176
      1          2359          0            2         2357
      2            24          4           20            0
      3             4          0            4            0
      4          2185         52           52         2081
      5             3          0            2            0
      6             8          0            8            0
      7          1157         96            5         1056
      8          1848       1653          195            0
      9          1833       1531          192          110
     10          1817       1622          195            0
     11          1104         48            3         1053
     12          1786       1591          195            0
     13           572         32            1          539
     14          1754       1559          195            0
     15          1675       1378          195          102
     16             3          0            3            0
     17           973        883           90            0
     18          1089        989          100            0
     19           670        625           45            0
     20           576         32            1          543
               ......
     26           680         49            0          631
     27           440        395           45            0
     28           558         33            1          524
         ......
     35            16          1           15            0
Average:       765.14     448.31
Throughout:    1.44/s
```

日志２的结果（时间片２００ＨＺ）：

```
(Unit:    tick)
Process    Turnaround    Waiting    CPU Burst    I/O Burst
      0          1024        134           55          318
      1          2268          0            3         2265
      2            50         12           38            0
      3            13          1           12            0
      4          2265         81          100         2084
      5             6          1            5            0
      6           197        181           16            0
      7          1136        101            0         1035
      8          1052        872          180            0
      9          1035        673          156          206
     10          1128        933          195            0
     11          1097         51            0         1046
     12          1110        915          195            0
     13          1078         33            3         1042
     14          1074        879          195            0
     15          1057        681          171          205
     16             6          0            6            0
     17             0          0            0            0
     18            17          2           15            0
     19             2          2            0            0
Average:       780.75     277.60
Throughout:    0.84/s
```

6 总结 第一，系统中进程0、1以及shell进程是占用系统最多的进程，也是系统中必须存在的进程，也可以得出实际上系统大部分时间都在等待中。第二，在修改了时间片后，进程占用时间也在修改，几乎也是一倍。要以看出一个好的调度算法对系统是很重要的。

后记： 在本次实验中可以看到，记录文件并未记录到进程0的诞生，这是因为进程0不是通过fork产生的，而是由系统手动生成的，具体情况此次实验并未深入研究。