# 《操作系统实验》Lab3 实验报告

09300240004 计算机科学与技术朱恬骅

## 实验目的

- 了解 Linux 的信号机制；
- 学习有关进程调度的内容。

## 实验要求

- 修改 Linux 0.11 内核代码，使得单个进程能记录自己被调度的次数；
- 使用信号传递延长测试程序的运行时间，增加被调度的次数；
- 编写一个建立多进程的测试程序，使各进程竞争 CPU 资源，最后输出被调度的次数。

## 实验原理

1. 多进程。Linux 0.11 使用分时技术调度系统中的进程，使它们虽然在某一时刻只有一个在占用 CPU 资源，却因为切换频繁而给人造成同时运行的印象。
2. 进程控制块（PCB）。内核通过一个进程任务表 task 对进程进行管理，同时可以控制 64 个进程（但是，pid 可以大于这个数目）。每个进程的运行信息保存在 task_struct 这个结构体中。所以可以通过修改这一结构体来增加保存进程调度次数的信息，以及模拟 sleep 的信号闹钟。
3. 计时。时钟中断处理程序通过 jiffies 变量来累积子系统启动以经过的时钟周期。每一次将 jiffies 自增 1，然后调用 do_timer()进行处理。如果某个定时器间到，则调用该处理函数然后对当前进程运行处理，对时间片减 1。如果进程时间片值递减后还大于 0，退出 do_timer 继续运行当前进程，否则根据被中断程序的级别来确定处理方法。若为用户态，调 schedule 切换到其它进程去运行；若为内核态， 运行；若为内核态， 运行；若为内核态， do_timer 立即退出。进程在内核态序中运行时是不可抢占的，但处于用户态程序中运行时则是可以被抢占的。

## 实验内容

1. 修改 PCB 信息，在文件/include/linux/sched.h 中：

```
struct task_struct {
/* these are hardcoded - don't touch */
        long state;       /* -1 unrunnable, 0 runnable, >0 stopped */
        long counter;
        long priority;
        long signal;
```

```
        struct sigaction sigaction[32];
        long blocked;  /* bitmap of masked signals */
/* various fields */
        int exit_code;
        unsigned long start_code,end_code,end_data,brk,start_stack;
        long pid,father,pgrp,session,leader;
        unsigned short uid,euid,suid;
        unsigned short gid,egid,sgid;
        long alarm;
        long utime,stime,cutime,cstime,start_time;
        unsigned short used_math;
/* file system info */
        int tty;            /* -1 if no tty, so it must be signed */
        unsigned short umask;
        struct m_inode * pwd;
        struct m_inode * root;
        struct m_inode * executable;
        unsigned long close_on_exec;
        struct file * filp[NR_OPEN];
/* ldt for this task 0 - zero 1 - cs 2 - ds&ss */
        struct desc_struct ldt[3];
/* tss for this task */
        struct tss_struct tss;
/* LAB4: added for counting sched time and for alarms */
        int schedCount;
        int schedAlarm;
};
```

2. 修改 fork()，增加对 schedCount 和 schedAlarm 的初始化：

```
int copy_process(int nr,long ebp,long edi,long esi,long gs,long none,
            long ebx,long ecx,long edx,
            long fs,long es,long ds,
            long eip,long cs,long eflags,long esp,long ss)
{
        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->schedCount = 0; /* LAB4 */
```

```
    p->schedAlarm = 0; /* LAB4 */
    p->state = TASK_UNINTERRUPTIBLE;
    p->pid = last_pid;
……
```

3. 修改 schedule()函数，增加对 schedCount 的记数和 schedAlarm 的处理：

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

/* check alarm, wake up any interruptible tasks that have got a signal */

    for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
        if (*p) {
            if ((*p)->alarm && (*p)->alarm < jiffies) {
                    (*p)->signal |= (1<<(SIGALRM-1));
                    (*p)->alarm = 0;
                }
            /* LAB4 [ */
            if ((*p)->schedAlarm && (*p)->schedAlarm < jiffies) {
                    (*p)->signal |= (1<<(SIGALRM-1));
                    (*p)->schedAlarm = 0;
                }
            /* ] LAB4 */
            if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
            (*p)->state==TASK_INTERRUPTIBLE)
                (*p)->state=TASK_RUNNING;
        }

/* this is the scheduler proper: */

    while (1) {
        c = -1;
        next = 0;
        i = NR_TASKS;
        p = &task[NR_TASKS];
        while (--i) {
            if (!*--p)
                continue;
            if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                c = (*p)->counter, next = i;
        }
        if (c) break;
```

```
            for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
                if (*p)
                    (*p)->counter = ((*p)->counter >> 1) +
                            (*p)->priority;
        }
        (task[next])->schedCount++;
        switch_to(next);
}
```

4. 修改系统调用相关的文件 system_calls.s、sys.h 和 unistd.h，增加 getSchedCount 和 schedAlarm 这两个系统调用：

在 include/linux/sys.h 中：

```
extern int sys_getSchedCount();
extern int sys_schedAlarm();

fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm,
sys_fstat, sys_pause, sys_utime, sys_stty, sys_gtty, sys_access,
sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid,
sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys,
sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
sys_uname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
sys_getpgrp, sys_setsid, sys_sigaction, sys_sgetmask, sys_ssetmask,
sys_setreuid, sys_setregid, sys_getSchedCount, sys_schedAlarm }
```

在 include/unistd.h 中：

```
#define __NR_getSchedCount72
#define __NR_schedAlarm     73
……
int getSchedCount();
int schedAlarm(int ms);
```

在 kernel/sys.c 中：

```
    int sys_schedAlarm(int ms) {
        int old = current->schedAlarm;
        if (old) old = (old - jiffies) / HZ;
        current->schedAlarm = (ms > 0) ? (jiffies + ms * HZ / 1000) : 0;
        return old;
    }

    int sys_getSchedCount() {
```

```
        return current->schedCount;
}
```

在 kernel/system_call.s 中，修改 nr_system_calls 定义

```
nr_system_calls = 74
```

5. 修改/usr/include/unistd.h 使之与编译时的源码保持一致，然后键入测试程序：

```
#define __LIBRARY__
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

_syscall0(int, getSchedCount);
_syscall1(int, schedAlarm, int, ms);

void h() {}

int main()
{
   int x;
   fork(); fork(); fork();
   for(x=0; x<1000; ++x) {
      signal(SIGALRM, h);
      schedAlarm(20); pause();
   }
   x=getSchedCount();
   printf("pid = %d, sched count = %d\n", getpid(), x);
   signal(SIGALRM, h);
   schedAlarm(1000);
   pause();
}
```

6. 屏幕输出：

```
[/usr/root]# gcc -o hello hello.c; ./hello
pid = 61, sched count = 1001
pid = 64, sched count = 1001
pid = 65, sched count = 1001
pid = 63, sched count = 1001
pid = 68, sched count = 1001
pid = 67, sched count = 1001
pid = 66, sched count = 1001
pid = 62, sched count = 1001
[/usr/root]#
```

这就完成了实验。

# 实验中遇到的问题和解决方法

1. 启动停留在 Loading system…不能继续，这是因为没有把 gcc 版本调低的缘故，使用 gcc 4.1 重新 make 之后能够成功引导系统。
2. 其它程序访问 PCB 时是按照原先 PCB 的偏移量来访问其中成员的，所以需要将新添加的两个 int 类型数据放在最后，以防止其它程序调用 PCB 信息时产生错误。
3. 测试程序每次运行的结果中，输出的条目个数不一致，这是因为子进程未执行到输出的时候父进程已经结束，导致子进程退出所致。所以在父进程的最后用 schedAlarm(1000); 暂停，以等候子进程结束。
4. 每次 Rest 虚拟机之后发现上次运行所修改的信息保存不完整或丢失，因为退出时未执行 exit，缓冲区中的内容就没有写入磁盘。

# 实验收获

1. 了解 Linux 的信号机制；学习有关进程调度的内容。
2. 发现了上次实验中被忽视的几个问题，包括：①在 Ubuntu 中修改源码之后编译，使 bochs 用外面编译好的 Image 文件引导；②退出时执行 exit 命令是必须的，否则会造成文件系统异常。

2011 年 10 月 18 日