“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

实验过程：

一.首先我们先修改 process.c 文件，它在 /home/teacher 目录下。这里因为截不了全部的代码，所以就截取了一部分，不过都是重复的代码。

代码：

```
int main(int argc, char * argv[])
{
        int pid1, pid2, pid3;
        int i = 0;
        /*create child process error*/
        if((pid1 = fork()) < 0)
        {
                printf("fork error");
        }
        /*child process*/
        else if(pid1 == 0)
        {
                //the primary task is IO
                cpuio_bound(10, 0, 4);
        }
        /*father process*/
        else
        {
                printf("new process:%d\n", pid1);
                /*father process waits for it's child process*/
                waitpid(pid1, NULL, 0);
        }
        /*create child process error*/
        if((pid2 = fork()) < 0)
        {
                printf("fork error");
        }
        /*child process*/
        else if(pid2 == 0)
        {
                /*the primary task is CPU*/
                cpuio_bound(10, 4, 0);
        }
        else
        {
                printf("new process:%d\n", pid2);
```

运行结果：

```
shiyanlou@8579e821f842:/home/teacher$ sudo gcc -o process process.c
shiyanlou@8579e821f842:/home/teacher$ ls
hit-oslab-linux-20110823.tar  process    stat_log.py  testlab2.sh
oslab                         process.c  testlab2.c
shiyanlou@8579e821f842:/home/teacher$ ./process
new process:27901
new process:28876
new process:30574
new process:31956
^C
shiyanlou@8579e821f842:/home/teacher$ vim process.c
shiyanlou@8579e821f842:/home/teacher$ sudo gcc -o process process.c
shiyanlou@8579e821f842:/home/teacher$ ./process
new process:4235
new process:5398
^[[A^[[B^[[A^[[B^[[A^[[B^[[A^C
shiyanlou@8579e821f842:/home/teacher$ vim process.c
shiyanlou@8579e821f842:/home/teacher$ pwd
/home/teacher
shiyanlou@8579e821f842:/home/teacher$ vim process.c
shiyanlou@8579e821f842:/home/teacher$ ./process
new process:29772
new process:30973
new process:692
new process:2580
new process:4458
new process:6669
new process:8490
shiyanlou@8579e821f842:/home/teacher$
```

二.接着我们来修改 /home/shiyanlou/oslab/oslab/linux-0.11/init 目录下的 main.c ，目的是在linux-0.11启动后，尽早记录Log文件。

代码：

```
#endif
        mem_init(main_memory_start,memory_end);
        trap_init();
        blk_dev_init();
        chr_dev_init();
        tty_init();
        time_init();
        sched_init();
        buffer_init(buffer_memory_end);
        hd_init();
        floppy_init();
        sti();
        move_to_user_mode();

/**********this is beginning**********/
        setup((void*)&drive_info);//load file system
        (void)open("/dev/tty0", O_RDWR, 0);//establish the association of file descriptor 0 and /dev/tty0
        (void)dup(0);//establish the asssociation of file descriptor 1 and /dev/tty0
        (void)dup(0);
        (void)open("/var/process.log", O_CREAT|O_TRUNC|O_WRONLY, 0666);
/**********this is end      **********/

        if (!fork()) {          /* we count on this going ok */
                init();
        }
/*
 *   NOTE!!   For any other task 'pause()' would mean we have to get a
 * signal to awaken, but task0 is the sole exception (see 'schedule()')
 * as task 0 gets activated at every idle moment (when no other tasks
 * can run). For task0 'pause()' just means we go check if some other
 * task can run, and if not we return here.
 */
        for(;;) pause();
}
```

添加的这段代码前4行原本就存在于 void init(void) 函数中，我们做的不过就是把这4行代码提前了而已，所以要把 init() 函数中对应的那4行代码注释掉。

```
        write(1,printbuf,i=vsprintf(printbuf, fmt, args));
        va_end(args);
        return i;
}

static char * argv_rc[] = { "/bin/sh", NULL };
static char * envp_rc[] = { "HOME=/", NULL };

static char * argv[] = { "-/bin/sh",NULL };
static char * envp[] = { "HOME=/usr/root", NULL };

void init(void)
{
        int pid,i;

/*      setup((void *) &drive_info);
        (void) open("/dev/tty0",O_RDWR,0);
        (void) dup(0);
        (void) dup(0);
*/
        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);
        if (!(pid=fork())) {
                close(0);
                if (open("/etc/rc",O_RDONLY,0))
                        _exit(1);
                execve("/bin/sh",argv_rc,envp_rc);
                _exit(2);
        }
        if (pid>0)
                while (pid != wait(&i))
                        /* nothing */;
        while (1) {
                if ((pid=fork())<0) {
                        printf("Fork failed in init\r\n");
```

三.我们已经完成了测试程序 process.c 以及开机打开log文件这两个功能。但是由于进程之间状态的变化都是在内核中进行的，所以普通的写入文件的函数不能使用，这段代码已经给出，只需要我们手动添上。修改的 printk.c 文件在 /home/shiyanlou/oslab/oslab/linux-0.11/kernel 目录下。

四.现在已经可以写入Log文件里，接下来，只需要在每一次进程改变状态的时候把对应的信息写入Log文件中就行了。

(1).修改 /home/shiyanlou/oslab/oslab/linux-0.11/kernel 目录下的 sched.c 文件里的 schedule() 函数。这个函数用于进程的就绪态到运行态之间的转换。所以我们应该在它改变状态的时候调用 fprintk() 函数就行了。

代码：

```
 * tasks can run. It can not be killed, and it cannot sleep. The 'state'
 * information in task[0] is never used.
 */
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;
                                }
                        if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
                        (*p)->state==TASK_INTERRUPTIBLE)
                        {
                                /*this is the code we added*/
                                if((*p)->state != TASK_RUNNING)
                                        fprintk(3, "%ld\t%c\t%ld\n", (*p)->pid, 'J', jiffies);
                                (*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;
-- 插入 --
```

```
                }

/* 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;
        }
        if(task[next]->pid != current->pid)
        {
                /* this is the code we added */
                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);
        }
        switch_to(next);

}

int sys_pause(void)
{
-- 插入 --
```

(2).在当前目录下，我们还需要修改 sys_pause(void) 函数，它就在 schedule() 函数的下面。这个函数的作用是让进程主动睡眠。

代码：

```
                for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
                        if (*p)
                                (*p)->counter = ((*p)->counter >> 1) +
                                                (*p)->priority;
        }
        if(task[next]->pid != current->pid)
        {
                /* this is the code we added */
                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);
        }
        switch_to(next);

}

int sys_pause(void)
{
        /* this is the code we added */
        if(current->state != TASK_INTERRUPTIBLE)
                fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies);
        current->state = TASK_INTERRUPTIBLE;
        schedule();
        return 0;
}

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");
-- 插入 --
```

(3).在当前目录下，修改 sleep_on(struct task_struct **p) 函数。阅读该函数代码可以得知，这个函数的作用是用来让进程睡眠以及唤醒队列中的上一个进程。

代码：

```
        if(current->state != TASK_INTERRUPTIBLE)
                fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies);
        current->state = TASK_INTERRUPTIBLE;
        schedule();
        return 0;
}

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;
        /* this is the code we added */
        fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies);
        schedule();
        if (tmp)
        {
        /* this is the code we added */
                if(tmp->state != TASK_RUNNING)
                {
                        fprintk(3, "%ld\t%c\t%ld%n", tmp->pid, 'J', jiffies);
                }
                tmp->state=0;

        }
}

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

(4).在当前目录下，修改 interruptible_sleep_on(struct task_struct **p) 。这个函数的作用和 sleep_on 函数的作用不同之处是是当队列中有被唤醒的进程以及队首的进程被唤醒时，有两个唤醒的进程如何处理。处理的方法是把队列中被唤醒的进程再次睡眠。

代码：

```
}

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:
        /* this is the code we added */
        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) {
                /* this is the code we added */
                if((*p)->state != TASK_RUNNING)
                {
                        fprintk(3, "%ld\t%c\t%ld\n", (*p)->pid, 'J', jiffies);
                }
                (**p).state=0;
                goto repeat;
        }
        *p=NULL;
        if (tmp)
                tmp->state=0;
}

void wake_up(struct task_struct **p)
{
        if (p && *p) {
                if((*p)->state != TASK_RUNNING)
-- 插入 --
```

(5).在当前目录下，修改 wake_up 函数，这个函数的作用是主动唤醒进程。

代码：

```
                if((*p)->state != TASK_RUNNING)
                {
                        fprintk(3, "%ld\t%c\t%ld\n", (*p)->pid, 'J', jiffies);
                }
                (**p).state=0;
                goto repeat;
        }
        *p=NULL;
        if (tmp)
                tmp->state=0;
}

void wake_up(struct task_struct **p)
{
        if (p && *p) {
        /* this is the code we added */
                if((*p)->state != TASK_RUNNING)
                        fprintk(3, "%ld\t%c\t%ld\n", (*p)->pid, 'J', jiffies);
                (**p).state=0;
                *p=NULL;
        }
}

/*
 * OK, here are some floppy things that shouldn't be in the kernel
 * proper. They are here because the floppy needs a timer, and this
 * was the easiest way of doing it.
 */
static struct task_struct * wait_motor[4] = {NULL,NULL,NULL,NULL};
static int  mon_timer[4]={0,0,0,0};
static int moff_timer[4]={0,0,0,0};
unsigned char current_DOR = 0x0C;

int ticks_to_floppy_on(unsigned int nr)
{
        extern unsigned char selected;
-- 插入 --
```

(6).修改 /home/shiyanlou/oslab/oslab/linux-0.11/kernel 目录下的 fork.c 文件里的 copy_process 函数。这个函数顾名思义，是把父进程的必要的信息拷贝给子进程。 代码：

```
        if (copy_mem(nr,p)) {
                task[nr] = NULL;
                free_page((long) p);
                return -EAGAIN;
        }
        for (i=0; i<NR_OPEN;i++)
                if ((f=p->filp[i]))
                        f->f_count++;
        if (current->pwd)
                current->pwd->i_count++;
        if (current->root)
                current->root->i_count++;
        if (current->executable)
                current->executable->i_count++;
        set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY,&(p->tss));
        set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,&(p->ldt));
        /* this is the code we added */
        fprintk(3, "%ld\t%c\t%ld\n", last_pid, 'N', jiffies);
        fprintk(3, "%ld\t%c\t%ld\n", last_pid, 'J', jiffies);
        p->state = TASK_RUNNING;        /* do this last, just in case */
        return last_pid;
}

int find_empty_process(void)
{
        int i;

        repeat:
                if ((++last_pid)<0) last_pid=1;
                for(i=0 ; i<NR_TASKS ; i++)
                        if (task[i] && task[i]->pid == last_pid) goto repeat;
        for(i=1 ; i<NR_TASKS ; i++)
                if (!task[i])
                        return i;
        return -EAGAIN;
}
-- 插入 --
```

(7).修改 /home/shiyanlou/oslab/oslab/linux-0.11/kernel 目录下的 exit.c 文件里的 do_exit 函数。该函数关闭该进程占用的资源。

代码：

```
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));
        for (i=0 ; i<NR_TASKS ; i++)
                if (task[i] && task[i]->father == current->pid) {
                        task[i]->father = 1;
                        if (task[i]->state == TASK_ZOMBIE)
                                /* assumption task[1] is always init */
                                (void) send_sig(SIGCHLD, task[1], 1);
                }
        for (i=0 ; i<NR_OPEN ; i++)
                if (current->filp[i])
                        sys_close(i);
        iput(current->pwd);
        current->pwd=NULL;
        iput(current->root);
        current->root=NULL;
        iput(current->executable);
        current->executable=NULL;
        if (current->leader && current->tty >= 0)
                tty_table[current->tty].pgrp = 0;
        if (last_task_used_math == current)
                last_task_used_math = NULL;
        if (current->leader)
                kill_session();
        current->state = TASK_ZOMBIE;
        /* this is the code we added */
        fprintk(3, "%ld\t%c\t%ld\n", current->state, 'E', jiffies);
        current->exit_code = code;
        tell_father(current->father);
        schedule();
        return (-1);    /* just to suppress warnings */
}

-- 插入 --
```

(8).在当前目录下，修改 sys_waitpid 函数。该函数作用是等待子进程结束。

代码：

```
                }
                switch ((*p)->state) {
                        case TASK_STOPPED:
                                if (!(options & WUNTRACED))
                                        continue;
                                put_fs_long(0x7f,stat_addr);
                                return (*p)->pid;
                        case TASK_ZOMBIE:
                                current->cutime += (*p)->utime;
                                current->cstime += (*p)->stime;
                                flag = (*p)->pid;
                                code = (*p)->exit_code;
                                release(*p);
                                put_fs_long(code,stat_addr);
                                return flag;
                        default:
                                flag=1;
                                continue;
                }
        }
        if (flag) {
                if (options & WNOHANG)
                        return 0;
                current->state=TASK_INTERRUPTIBLE;
                /* this is the code we added */
                fprintk(3, "%ld\t%c\t%ld\n", current->state, 'W', jiffies);
                schedule();
                if (!(current->signal &= ~(1<<(SIGCHLD-1))))
                        goto repeat;
                else
                        return -EINTR;
        }
        return -ECHILD;
}

-- 插入 --
```

五.把 process.c 拷贝到虚拟机上，然后编译了运行虚拟机，并查看 process.log 日志文件。

```
[/usr/root]# more /var/process.log
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
3       E       73
1       J       73
1       R       73
4       N       74
4       J       74
1       W       74
4       R       74
5       N       106
5       J       106
--More--(2%)
```

```
shiyanlou@8579e821f842:~/oslab/oslab/linux-0.11/kernel$ cd ..
shiyanlou@8579e821f842:~/oslab/oslab/linux-0.11$ ls
boot  fs  Image  include  init  kernel  lib  Makefile  mm  System.map  tags  tools
shiyanlou@8579e821f842:~/oslab/oslab/linux-0.11$ cd ..
shiyanlou@8579e821f842:~/oslab/oslab$ ./run
========================================================================
                       Bochs x86 Emulator 2.3.7
               Build from CVS snapshot, on June 3, 2008
========================================================================
00000000000i[     ] reading configuration from ./bochs/bochsrc.bxrc
00000000000i[     ] installing x module as the Bochs GUI
00000000000i[     ] using log file ./bochsout.txt
```

六.把 process.log 文件拷贝到ubuntu中，然后在 /home/teacher 目录下，执行 stat_log.py 脚本程序，执行之前如果没有执行权限，则键入 sudo chmod +x stat_log.py 添加权限。

```
/home/teacher
shiyanlou@8579e821f842:/home/teacher$ ./stat_log.py ../shiyanlou/oslab/oslab/process.log 0 1 2 3 4 5
 -g
(Unit: tick)
Process   Turnaround   Waiting   CPU Burst   I/O Burst
      0            8         0           8           0
      1            1         0           1           0
      2           15         0          15           0
      3            0         0           0           0
Average:        6.00      0.00
Throughput: 16.67/s

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

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

-----===< !!!!!!!!!!!!!!!!!!!!!!!!! >===-----

   40 -0
   41 #0
   42 #
   43 #
   44 #
   45 #
   46 #
   47 #
   48 |0  -1
   49     :1  -2
```

七.修改时间片我们只用修改 /home/shiyanlou/oslab/oslab/linux-0.11/include/linux 目录下的 sched.h 文件里的全局常量HZ，默认值是100。可以把HZ改大一倍或者改小。

```
#ifndef _SCHED_H
#define _SCHED_H

#define NR_TASKS 64
#define HZ 50

#define FIRST_TASK task[0]
#define LAST_TASK task[NR_TASKS-1]

#include <linux/head.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <signal.h>

#if (NR_OPEN > 32)
#error "Currently the close-on-exec-flags are in one word, max 32 files/proc"
#endif

#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define TASK_ZOMBIE             3
#define TASK_STOPPED            4

#ifndef NULL
#define NULL ((void *) 0)
#endif

extern int copy_page_tables(unsigned long from, unsigned long to, long size);
extern int free_page_tables(unsigned long from, unsigned long size);

extern void sched_init(void);
extern void schedule(void);
extern void trap_init(void);
#ifndef PANIC
volatile void panic(const char * str);
```

再执行 process.c 并记录下 process.log

```
1       N       24
1       J       24
0       J       24
1       R       24
2       N       24
2       J       24
1       W       24
2       R       24
3       N       32
3       J       32
3       E       34
1       J       34
3       R       34
"process.log" 13L, 91C
```

实验结果：

问题一：

单进程编程导致CPU利用率低，在相同的任务量下，如果涉及到文件IO等操作，效率会非常低。多进程编程可以有效的利用CPU。

问题二：

修改了 sched.h 里面宏定义的 HZ ，不修改和修改之后的Log文件得出的结果表明进程占用的时间少了一半。这是因为原本是每10ms产生一次时钟中断，HZ改成50之后，变成了每5ms产生一次时钟中断。