哈尔滨工业大学计算机科学与技术学院
2016年秋季学期《操作系统》

Lab3：进程运行轨迹的跟踪与统计

姓名	学号	联系方式
匡盟盟	1143220116	kuangmeng@msn.com
樊晨霄	15S008199	18513534698

目　录

一、实验目的

- 掌握Linux下的多进程编程技术；
- 通过对进程运行轨迹的跟踪来形象化进程的概念；
- 在进程运行轨迹跟踪的基础上进行相应的数据统计，从而能对进程调度算法进行实际的量化评价，更进一步加深对调度和调度算法的理解，获得能在实际操作系统上对调度算法进行实验数据对比的直接经验。

二、实验内容

进程从创建（Linux下调用fork()）到结束的整个过程就是进程的生命期，进程在其生命期中的运行轨迹实际上就表现为进程状态的多次切换，如进程创建以后会成为就绪态；当该进程被调度以后会切换到运行态；在运行的过程中如果启动了一个文件读写操作，操作系统会将该进程切换到阻塞态（等待态）从而让出CPU；当文件读写完毕以后，操作系统会在将其切换成就绪态，等待进程调度算法来调度该进程执行……

实验基本内容

1. 基于模板“process.c”编写多进程的样本程序，实现如下功能：
    1. 所有子进程都并行运行，每个子进程的实际运行时间一般不超过30秒；
    2. 父进程向标准输出打印所有子进程的id，并在所有子进程都退出后才退出；
2. 在Linux 0.11上实现进程运行轨迹的跟踪。基本任务是在内核中维护一个日志文件/var/process.log，把从操作系统启动到系统关机过程中所有进程的运行轨迹都记录在这一log文件中。
3. 在修改过的0.11上运行样本程序，通过分析log文件，统计该程序建立的所有进程的等待时间、完成时间（周转时间）和运行时间，然后计算平均等待时间，平均完成时间和吞吐量。可以自己编写统计程序，也可以使用python脚本程序－－ stat_log.py －－进行统计。
4. 修改0.11进程调度的时间片，然后再运行同样的样本程序，统计同样的时间数据，和原有的情况对比，体会不同时间片带来的差异。

/var/process.log文件的格式必须为：

```
pid     X       time
```

其中：

- pid是进程的ID；
- X可以是N,J,R,W和E中的任意一个，分别表示进程新建(N)、进入就绪态(J)、进入运行态(R)、进入阻塞态(W)和退出(E)；
- time表示X发生的时间。这个时间不是物理时间，而是系统的滴答时间(tick)；
- 三个字段之间用制表符分隔。

例如：

```
12      N       1056
12      J       1057
4       W       1057
12      R       1057
13      N       1058
13      J       1059
14      N       1059
14      J       1060
15      N       1060
15      J       1061
12      W       1061
15      R       1061
```

15 J 1076
14 R 1076
14 E 1076
……

三、实验过程

1. 编写process.c程序，先放在oslab3目录下，用到了系统调用方法fork()，创建了十个子进程，包括IO密集型和CPU密集型进程，通过cpuio_bound()方法实现：

```
void cpuio_bound(int last, int cpu_time, int io_time)
{
    struct tms start_time, current_time;
    clock_t utime, stime;
    int sleep_time;

    while (last > 0)
    {
        /* CPU Burst */
        times(&start_time);
        do
        {
            times(&current_time);
            utime = current_time.tms_utime - start_time.tms_utime;
            stime = current_time.tms_stime - start_time.tms_stime;
        } while ( ( (utime + stime) / HZ )  < cpu_time );
        last -= cpu_time;

        if (last <= 0 )
            break;

        /* IO Burst */
        /* 用sleep(1)模拟1秒钟的I/O操作 */
        sleep_time=0;
        while (sleep_time < io_time)
        {
            sleep(1);
            sleep_time++;
        }
        last -= sleep_time;
    }
}
```

2. 在linux-0.11/init/目录下，修改main.c文件，添加创建日志文件/var/process.log语句，将log文件关联到文件描述符3，0、1、2分别是stdin、stdout、stderr，添加如下：

```
setup((void *) &drive_info);  /*加载系统文件*/
(void) open("/dev/tty0",O_RDWR,0);     /*建立文件描述符0和dev/tty0相关联*/
(void) dup(0);/*同上*/
(void) dup(0);/*同上*/
(void) open("/var/process.log",O_CREAT|O_TRUNC|O_WRONLY,0666);
```

3. 向linux-0.11/kernel目录下的printk.c文件中添加日志打印功能，在源文件中添加：

```
if (fd < 3)
{
    __asm__("push %%fs\n\t"
        "push %%es\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","cx");
}
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;
```

4. 在linux-0.11/kernel目录下的fork.c文件的对应位置添加打印输出（打印状态信息到log中），主要修改如下：

```
p->start_time = jiffies;    /*设置start_time为jiffies*/

fprintk(3,"%d\tN\t%d\n",p->pid,jiffies);  /*新建进程，pid N    jiffies,在3上面的打印log */

p->tss.back_link = 0;
```

5. 在linux-0.11/kernel目录下的sched.c文件中找到正确位置，添加如下代码（功能同上）：

```
        if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
        (*p)->state==TASK_INTERRUPTIBLE)
            {
                (*p)->state=TASK_RUNNING;/*唤醒*/
                /*可中断睡眠唤醒到就蓄状态*/
                /*睡眠->就绪*/
                fprintk(3, "%d\tJ\t%d\n", (*p)->pid, jiffies);
            }
              if (*p)
                  (*p)->counter = ((*p)->counter >> 1) +
                          (*p)->priority;
    }

    if(current->pid!=task[next]->pid)
    {
        if(current->state == TASK_RUNNING) /*时间片到，转为就绪状态*/
        {
            /*运行—>就绪*/
            fprintk(3, "%d\tJ\t%d\n", current->pid, jiffies);
        }
        /*就绪->运行*/
        fprintk(3, "%d\tR\t%d\n", task[next]->pid, jiffies);
    }
    switch_to(next);/*切换到next进程*/
```

```
int sys_pause(void)
{
    current->state = TASK_INTERRUPTIBLE;
    /*运行->可中断睡眠*/
    if(current->pid != 0)
    {
        fprintk(3,"%d\tW\t%d\n",current->pid,jiffies);
    }
    schedule();
    return 0;
}
```

```
*p = current;      /*将current插入等待队列的头部*/
current->state = TASK_UNINTERRUPTIBLE;    /*切换状态到不可中断睡眠状态*/
/*运行->不可中断睡眠*/
fprintk(3,"%d\tW\t%d\n",current->pid,jiffies);
schedule();          /*睡眠状态让出CPU*/
if (tmp)
{
    tmp->state=0; /*唤醒队列中的上一个睡眠进程。这儿的0可以换为TASK_RUNNING*/
    /*原等待队列中。第一个进程->唤醒就绪*/
    fprintk(3,"%d\tJ\t%d\n",tmp->pid,jiffies);
}
```

```
*p=current;
repeat: current->state = TASK_INTERRUPTIBLE; /*切换到可中断睡眠状态*/
/*唤醒队列的申请成功。通过goto去实现了唤醒  队列中的头进程，进行等待*/
fprintk(3,"%d\tW\t%d\n",current->pid,jiffies);
schedule();
if (*p && *p != current) {  /*如果队列头指针的进程和现场量的不是不是同一个进程，说明在队列的申请唤醒来一个进程，需要处理一下*/
    (**p).state=0;    /*将队列头唤醒，然后通过goto让自己睡眠*/
    /*当前进程进行 可中断睡眠*/
    fprintk(3,"%d\tJ\t%d\n",(*p)->pid,jiffies);
    goto repeat;
}
```

```
*p=NULL;
if (tmp)
{
    /*原等待队列  第一个进程 唤醒到就绪状态*/
    fprintk(3,"%d\tJ\t%d\n",tmp->pid,jiffies);
    tmp->state=0;
}
```

```
void wake_up(struct task_struct **p)
{
    if (p && *p) {
        (**p).state=0;
        /*唤醒到就绪状态*/
        fprintk(3,"%d\tJ\t%d\n",(*p)->pid,jiffies);
        *p=NULL;
    }
}
```

6. 在linux-0.11/kernel目录下的exit.c文件中，找到正确的状态转换点，添加如下代码（功能同上），主要代码如下：

```
                fprintk(3,"%d\tW\t%d\n",current->pid,jiffies);
                schedule();
                if (!(current->signal &= ~(1<<(SIGCHLD-1))))
                    goto repeat;
                else
                    return -EINTR;
            }
            return -ECHILD;
        current->cstime += (*p)->stime;
        flag = (*p)->pid;
        code = (*p)->exit_code;
        release(*p);
        put_fs_long(code,stat_addr);
        return flag;
```

7. 进入linux-0.11/目录，编译linux-0.11，使用命令“make all”；
8. 运行bochs，编译并运行process.c（需要将process.c放到hdc/usr/root/目录）：

```
[/usr/root]# gcc -o process process.c
[/usr/root]# ./process
Our MICos child pid: 13
Our MICos child pid: 14
Our MICos child pid: 15
Our MICos child pid: 16
Our MICos child pid: 17
Our MICos child pid: 18
Our MICos child pid: 19
Our MICos child pid: 20
Our MICos child pid: 21
Our MICos child pid: 22
```

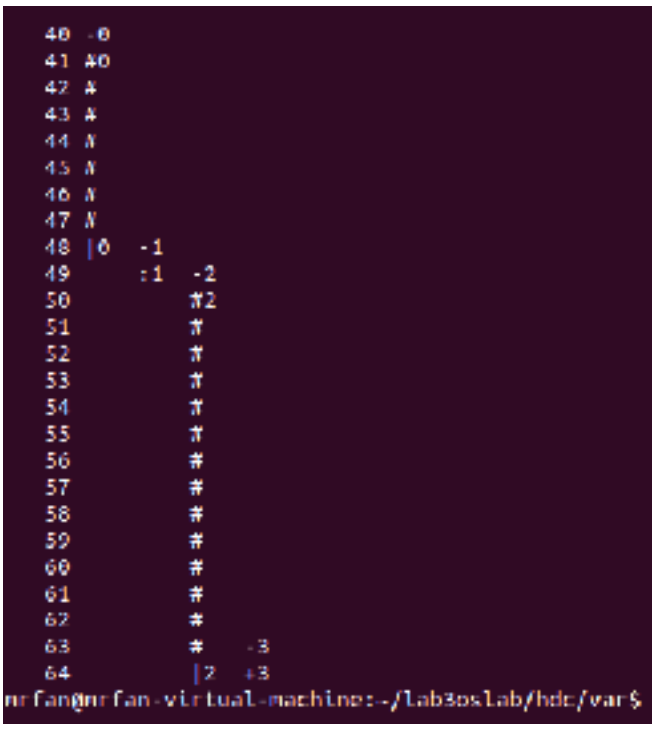
9. 在bochs上运行“ls -l /var”或“ll /var”查看process.log的相关属性：

```
[/usr/root]# ll /var
total 10
-rw-r--r--    1 root     root           799 ???  ??  ????  process.log
-rwx--x--x    1 1000     232           8496 Jan   4  2016  stat_log.py
```

10. 给stat_log.py加上执行权限（chmod+x stat_log.py），运行“./stat_log.py process.log 0 1 2 3……N -g”（统计PID为0——N的进程，为了方便我们只统计了0 1 2 3）：

```
mrfan@mrfan-virtual-machine:~/lab3oslab/hdc/var$ ./stat_log.py process.log 0 1 2 3 -g
(Unit: tick)
Process    Turnaround   Waiting    CPU Burst   I/O Burst
      0             8         0           8           8
      1             1         0           1           8
      2            15         0          15           8
      3             1         0           0           8
Average:         6.25      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
-----===< !!!!!!!!!!!!!!!!!!!!!!!! >===-----
```



11. 修改linux-0.11/include/linux目录下的sched.h文件中通过宏INIT_TASK定义的时间片，同时修改priority值，即可实现对不同时间片大小的调整：

至此本实验完毕！

四、回答问题

1. 结合自己的体会，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：对于小规模的程序，单进程编程所写的程序是顺序执行的，彼此之间有严格的执行逻辑，编程思路清晰，不易出错，而且执行效率比较高，速度比较快，而且可以通过同进程的多个线程之间进行相互通讯和交换数据，线程间的关系比较紧密也便于管理，启动一个线程的开销要比启动进程的开销要小的多，且切换耗费小；在没有其他程序的干扰下，数据是同步的，但是其缺点也比较明显，线程间相互的影响比较大，某个现成的崩溃可能直接引起其他线程无法正常工作。

对于比较复杂、规模较大的程序任务，则多进程的优势非常明显，多进程的独立性强，有各自独立的代码段和数据段，相互影响较小，适合于独立工作。多进程编程所写的程序是同时执行的，虽然共享文件等，但是由于多个进程之间执行顺序无法得知，故而要考虑进程之间的关系和影响，尤其是数据异步，程序员要做好进程之间同步，通信，互斥等。相比较而言，多进程编程比单进程编程复杂得多，但用途广泛得多。

总的来说，二者最大区别是，多进程需要使用多个独立的代码段、数据段和堆栈段，创建和切换的开销较大，而且多个进程间的通信较单进程编程要更为复杂。但它比单进程下多线程工作要更

稳定，程序更健壮。而且更有利于同时执行多个独立的任务。而单进程编程更轻便、处理更简单方便。

2. 你是如何修改时间片的？仅针对样本程序建立的进程，在修改时间片前后，log文件的统计结果（不包括Graphic）都是什么样？结合你的修改分析一下为什么会这样变化，或者为什么没变化？

答：通过修改INIT_TASK中的priority的值来修改时间片的大小。也即修改进程0的优先权，因为所有进程其实都是进程1的子进程，而进程1是任务0的子进程，因此所有进程的优先权都是继承自任务0的，而他们在创建时时间片就是由优先权赋值的。在include/linux/sched.h中修改#define INIT_TASK宏定义中counter和priority数值。原始时间片为15，修改了两次时间片，分别为10和20。结果如下：

时间片10

(Unit: tick)

Process	Turnaround	Waiting	CPU Burst	I/O Burst
7	2298	97	0	2200
8	2319	1687	200	432
9	2368	2098	270	0
10	2358	2087	270	0
11	2347	2076	270	0
12	2336	2066	270	0
13	2326	2055	270	0
14	2315	2044	270	0
15	2304	2034	270	0
16	2292	2021	270	0
Average:	2326.30	1826.50		

Throughout: 0.42/s

时间片15

(Unit: tick)

Process	Turnaround	Waiting	CPU Burst	I/O Burst
7	2247	142	0	2105
8	2202	1686	200	315
9	2246	1991	255	0
10	2230	1975	255	0
11	2215	1959	255	0
12	2199	1944	255	0
13	2183	1928	255	0
14	2168	1912	255	0
15	2152	1897	255	0
16	2137	1881	255	0
Average:	2197.90	1731.50		

Throughout: 0.45/s

时间片20

(Unit: tick)

Process	Turnaround	Waiting	CPU Burst	I/O Burst
7	2587	187	0	2400
8	2567	1766	200	600
9	2608	2308	300	0
10	2585	2285	300	0
11	2565	2264	300	0
12	2544	2244	300	0
13	2523	2223	300	0
14	2503	2202	300	0
15	2482	2182	300	0

```
        16      2461    2161      300       0
Average:    2542.50  1982.20
Throughout: 0.38/s
```

时间片变小，进程因时间片到时产生的进程调度次数变多，该进程等待时间越长。

然而随着时间片增大，进程因中断或者睡眠进入的进程调度次数也增多，等待时间随之变长。故而需要设置合理的时间片，既不能过大，也不能过小。

在修改时间片前，由于样本程序中创建的子进程一次连续占用cpu的时间比时间片要大，因此样本程序中子进程还未执行完，就被调度为就绪状态了。表现为占用cpu的子进程在R状态后15个滴答后就转为J状态了，而把时间片修改到30个滴答后，虽然子进程仍然是未执行完就被调度了，但一次运行的滴答明显改变了。而对于样本程序中不占用cpu只占用I/O的进程则没变化，因为他们一旦执行，就进入睡眠等待状态，等待I/O处理。

实验心得

其实做完实验我还是不能保证我对OS Lab这个平台有很好的全面的了解，但是对一些基本操作及其快捷键我算是大致掌握了，通过这个平台我也是认识到了“没有做不到的，只有想不到的”，我觉得创建这个平台的人们真的是很了不起，这个平台让我们便动手便了解了平时我们看不到的操作系统的相关知识。要做好实验，得按照实验教程上面的内容一步步落实，要边做变领悟相关原理及运行结果的出现的原因，这样我们才能在试验中学到更多、掌握更多。其次，也遇到问题我们自然是要先自己思考，通过不同的尝试来解决，之后不能解决的我们要多向老师同学请教，通过互相交流得来的知识也是会让我们难忘的。

附录

process.c

```
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include <sys/times.h>
#include <sys/types.h>
/*基本数据类型*/
#define HZ      100
void cpuio_bound(int last, int cpu_time, int io_time);
int main(int argc, char * argv[])
{
        pid_t proc_num[10]; /*创建10个子进程*/
        int i;
           for(i = 0; i < 10 ; i++){
                    proc_num[i] = fork();

                if(proc_num[i] == 0) {
                            cpuio_bound(10, i, 10-i);
                    } else if (proc_num[i] < 0){
                            printf("Failed in process %d\n", i+1);
                            return -1;
                    }
           }
           for(i = 0; i < 10; i ++){
                    printf("child pid: %d\n", proc_num[i]);
           }

           wait(&i);/*父进程等到所有的子进程完成然后返回*/
```

```
        return 0;
}

/*
 * 此函数按照参数占用CPU和I/O时间
 * last: 函数实际占用CPU和I/O的总时间，不含在就绪队列中的时间，>=0是必须的
 * cpu_time: 一次连续占用CPU的时间，>=0是必须的
 * io_time: 一次I/O消耗的时间，>=0是必须的
 * 如果last > cpu_time + io_time，则往复多次占用CPU和I/O
 * 所有时间的单位为秒
 */
void cpuio_bound(int last, int cpu_time, int io_time){
        struct tms start_time, current_time;
        clock_t utime, stime;
        int sleep_time;
        while (last > 0)
        {
                /* CPU Burst */
                times(&start_time);
                /* 其实只有t.tms_utime才是真正的CPU时间。但我们是在模拟一个
                 * 只在用户状态运行的CPU大户，就像“for(;;);”。所以把t.tms_stime
                 * 加上很合理。*/
                do{
                        times(&current_time);
                        utime = current_time.tms_utime - start_time.tms_utime;
                        stime = current_time.tms_stime - start_time.tms_stime;
                } while ( ( (utime + stime) / HZ )  < cpu_time );
                last -= cpu_time;
                if (last <= 0 )
                        break;
                /* IO Burst */
                /* 用sleep(1)模拟1秒钟的I/O操作 */
                sleep_time=0;
                while (sleep_time < io_time)
                {
                        sleep(1);
                        sleep_time++;
                }
                last -= sleep_time;
        }
}
```