进程运行轨迹的跟踪与统计

难度系数：★★★☆☆

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。

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……

本次实验包括如下内容：

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
……
```

实验报告

完成实验后，在实验报告中回答如下问题：

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？
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？

评分标准

- process.c，50%
- 日志文件建立成功，5%
- 能向日志文件输出信息，5%
- 5种状态都能输出，10%（每种2%）
- 调度算法修改，10%
- 实验报告，20%

实验提示

process.c的编写涉及到fork()和wait()系统调用，请自行查阅相关文献。0.11内核修改涉及到init/main.c、kernel/fork.c和kernel/sched.c，开始实验前如果能详细阅读《注释》一书的相关部分，会大有裨益。

编写样本程序

所谓样本程序，就是一个生成各种进程的程序。我们的对0.11的修改把系统对它们的调度情况都记录到log文件中。在修改调度算法或调度参数后再运行完全一样的样本程序，可以检验调度算法的优劣。理论上，此程序可以在任何Unix/Linux上运行，所以建议在Ubuntu上调试通过后，再拷贝到0.11下运行。

process.c是样本程序的模板。它主要实现了一个函数：

```
/*
 * 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，直到总运行时间超过last为止
 * 所有时间的单位为秒
 */
cpuio_bound(int last, int cpu_time, int io_time);
```

比如一个进程如果要占用10秒的CPU时间，它可以调用：

```
cpuio_bound(10, 1, 0);  // 只要cpu_time>0，io_time=0，效果相同
```

以I/O为主要任务：

```
cpuio_bound(10, 0, 1);  // 只要cpu_time=0，io_time>0，效果相同
```

CPU和I/O各1秒钟轮回：

```
cpuio_bound(10, 1, 1);
```

较多的I/O，较少的CPU：

```
cpuio_bound(10, 1, 9);  // I/O时间是CPU时间的9倍
```

修改此模板，用fork()建立若干个同时运行的子进程，父进程等待所有子进程退出后才退出，每个子进程按照你的意愿做不同或相同的cpuio_bound()，从而完成一个个性化的样本程序。它可以用来检验有关log文件的修改是否正确，同时还是数据统计工作的基础。

wait()系统调用可以让父进程等待子进程的退出。

小技巧：

1. 在Ubuntu下，top命令可以监视即时的进程状态。在top中，按u，再输入你的用户名，可以限定只显示以你的身份运行的进程，更方便观察。按h可得到帮助。
2. 在Ubuntu下，ps命令可以显示当时各个进程的状态。“ps aux”会显示所有进程；“ps aux | grep xxxx”将只显示名为xxxx的进程。更详细的用法请问man。
3. 在Linux 0.11下，按F1可以即时显示当前所有进程的状态。

log文件

操作系统启动后先要打开/var/process.log，然后在每个进程发生状态切换的时候向log文件内写入一条记录，其过程和用户态的应用程序没什么两样。然而，因为内核状态的存在，使过程中的很多细节变得完全不一样。

打开log文件

为了能尽早开始记录，应当在内核启动时就打开log文件。内核的入口是init/main.c中的main()（Windows环境下是start()），其中一段代码是：

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

这段代码在进程0中运行，先切换到用户模式，然后全系统第一次调用fork()建立进程1。进程1调用init()。在init()中：

```
……
setup((void *) &drive_info);            //加载文件系统
(void) open("/dev/tty0",O_RDWR,0);      //打开/dev/tty0，建立文件描述符0和/dev/tty0的关联
(void) dup(0);                          //让文件描述符1也和/dev/tty0关联
(void) dup(0);                          //让文件描述符2也和/dev/tty0关联
……
```

这段代码建立了文件描述符0、1和2，它们分别就是stdin、stdout和stderr。这三者的值是系统标准（Windows也是如此），不可改变。可以把log文件的描述符关联到3。文件系统初始化，描述符0、1和2关联之后，才能打开log文件，开始记录进程的运行轨迹。为了能尽早访问log文件，我们要让上述工作在进程0中就完成。所以把这一段代码从init()**移动到**main()中，放在move_to_user_mode()之后（不能再靠前了），同时加上打开log文件的代码。修改后的main()如下：

```
……
move_to_user_mode();

/***************添加开始***************/
setup((void *) &drive_info);
(void) open("/dev/tty0",O_RDWR,0);      //建立文件描述符0和/dev/tty0的关联
(void) dup(0);          //文件描述符1也和/dev/tty0关联
(void) dup(0);          //文件描述符2也和/dev/tty0关联
(void) open("/var/process.log",O_CREAT|O_TRUNC|O_WRONLY,0666);
/***************添加结束***************/

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

打开log文件的参数的含义是建立只写文件，如果文件已存在则清空已有内容。文件的权限是所有人可读可写。

这样，文件描述符0、1、2和3就在进程0中建立了。根据fork()的原理，进程1会继承这些文件描述符，所以init()中就不必再open()它们。此后所有新建的进程都是进程1的子孙，也会继承它们。但实际上，init()的后续代码和/bin/sh都会重新初始化它们。所以只有进程0和进程1的文件描述符肯定关联着log文件，这一点在接下来的写log中很重要。

写log文件

log文件将被用来记录进程的状态转移轨迹。所有的状态转移都是在内核进行的。在内核状态下，write()功能失效，其原理等同于《系统调用》实验中不能在内核状态调用printf()，只能调用printk()。编写可在内核调用的write()的难度较大，所以这里直接给出源码。它主要参考了printk()和sys_write()而写成的：

```
#include <linux/sched.h>
#include <sys/stat.h>

static char logbuf[1024];
int fprintk(int fd, const char *fmt, ...)
{
        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)     /* 如果输出到stdout或stderr，直接调用sys_write即可 */
        {
                __asm__("push %%fs\n\t"
                        "push %%ds\n\t"
                        "pop %%fs\n\t"
                        "pushl %0\n\t"
                        "pushl $logbuf\n\t" /* 注意对于Windows环境来说，是_logbuf,下同 */
                        "pushl %1\n\t"
                        "call sys_write\n\t" /* 注意对于Windows环境来说，是_sys_write,下同 */
                        "addl $8,%%esp\n\t"
                        "popl %0\n\t"
                        "pop %%fs"
                        ::"r" (count),"r" (fd):"ax","cx","dx");
        }
        else    /* 假定>=3的描述符都与文件关联。事实上，还存在很多其它情况，这里并没有考虑。*/
        {
                if (!(file=task[0]->filp[fd]))  /* 从进程0的文件描述符表中得到文件句柄 */
                        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;
}
```

因为和printk的功能近似，建议将此函数放入到kernel/printk.c中。fprintk()的使用方式类同与C标准库函数fprintf()，唯一的区别是第一个参数是文件描述符，而不是文件指针。例如：

```
fprintk(1, "The ID of running process is %ld", current->pid); //向stdout打印正在运行的进程的ID
fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'R', jiffies); //向log文件输出
```

跟踪进程运行轨迹

jiffies，滴答

jiffies在kernel/sched.c文件中定义为一个全局变量：

```
long volatile jiffies=0;
```

它记录了从开机到当前时间的时钟中断发生次数。在kernel/sched.c文件中的sched_init()函数中，时钟中断处理函数被设置为：

```
set_intr_gate(0x20,&timer_interrupt);
```

而在kernel/system_call.s文件中将timer_interrupt定义为：

```
timer_interrupt:
        ……
        incl jiffies     #增加jiffies计数值
        ……
```

这说明**jiffies表示从开机时到现在发生的时钟中断次数**，这个数也被称为“滴答数”。

另外，在kernel/sched.c中的sched_init()中有下面的代码：

```
outb_p(0x36, 0x43); //设置8253模式
outb_p(LATCH&0xff, 0x40);
outb_p(LATCH>>8, 0x40);
```

这三条语句用来设置每次时钟中断的间隔，即为LATCH，而LATCH是定义在文件kernel/sched.c中的一个宏：

```
#define LATCH  (1193180/HZ)
#define HZ 100  //在include/linux/sched.h中
```

再加上PC机8253定时芯片的输入时钟频率为1.193180MHz，即1193180/每秒，LATCH=1193180/100，时钟每跳11931.8下产生一次时钟中断，即每1/100秒（10ms）产生一次时钟中断，所以**jiffies实际上记录了从开机以来共经过了多少个10ms**。

寻找状态切换点

必须找到所有发生进程状态切换的代码点，并在这些点添加适当的代码，来输出进程状态变化的情况到log文件中。此处要面对的情况比较复杂，需要对kernel下的fork.c、sched.c有通盘的了解，而exit.c也会涉及到。我们给出两个例子描述这个工作该如何做，其他情况实验者可仿照完成。

第一个例子是看看如何记录一个进程生命期的开始，当然这个事件就是进程的创建函数fork()，由《系统调用》实验可知，fork()功能在内核中实现为sys_fork()，该“函数”在文件kernel/system_call.s中实现为：

```
sys_fork:
        call find_empty_process
        ……
        push %gs   //传递一些参数
        pushl %esi
        pushl %edi
        pushl %ebp
        pushl %eax
        call copy_process  //调用copy_process实现进程创建
        addl $20,%esp
```

所以真正实现进程创建的函数是copy_process()，它在kernel/fork.c中定义为：

```
int copy_process(int nr,……)
{
        struct task_struct *p;
        ……
        p = (struct task_struct *) get_free_page();  //获得一个task_struct结构体空间
        ……
        p->pid = last_pid;
        ……
        p->start_time = jiffies;        //设置start_time为jiffies
       ……
        p->state = TASK_RUNNING;        //设置进程状态为就绪。所有就绪进程的状态都是
                                        //TASK_RUNNING(0)，被全局变量current指向的
                                        //是正在运行的进程。
        return last_pid;
```

```
}
```

因此要完成进程运行轨迹的记录就要在copy_process()中添加输出语句。这里要输出两种状态，分别是“N（新建）”和“J（就绪）”。

第二个例子是记录进入睡眠态的时间。sleep_on()和interruptible_sleep_on()让当前进程进入睡眠状态，这两个函数在kernel/sched.c文件中定义如下：

```
void sleep_on(struct task_struct **p)
{
        struct task_struct *tmp;
        ……
        tmp = *p;
        *p = current;  //仔细阅读，实际上是将current插入“等待队列”头部，tmp是原来的头部
        current->state = TASK_UNINTERRUPTIBLE; //切换到睡眠态
        schedule();  //让出CPU
        if (tmp)
                tmp->state=0;   //唤醒队列中的上一个（tmp）睡眠进程。0换作TASK_RUNNING更好
                                //在记录进程被唤醒时一定要考虑到这种情况，实验者一定要注意!!!
}
/* TASK_UNINTERRUPTIBLE和TASK_INTERRUPTIBLE的区别在于不可中断的睡眠
 * 只能由wake_up()显式唤醒，再由上面的 schedule()语句后的
 *
 *   if (tmp) tmp->state=0;
 *
 * 依次唤醒，所以不可中断的睡眠进程一定是按严格从“队列”（一个依靠
 * 放在进程内核栈中的指针变量tmp维护的队列）的首部进行唤醒。而对于可
 * 中断的进程，除了用wake_up唤醒以外，也可以用信号（给进程发送一个信
 * 号，实际上就是将进程PCB中维护的一个向量的某一位置位，进程需要在合
 * 适的时候处理这一位。感兴趣的实验者可以阅读有关代码）来唤醒，如在
 * schedule()中：
 *
 *  for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
 *      if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
 *          (*p)->state==TASK_INTERRUPTIBLE)
 *              (*p)->state=TASK_RUNNING;//唤醒
 *
 * 就是当进程是可中断睡眠时，如果遇到一些信号就将其唤醒。这样的唤醒会
 * 出现一个问题，那就是可能会唤醒等待队列中间的某个进程，此时这个链就
 * 需要进行适当调整。interruptible_sleep_on和sleep_on函数的主要区别就
 * 在这里。
 */
void interruptible_sleep_on(struct task_struct **p)
{
        struct task_struct *tmp;
       …
        tmp=*p;
        *p=current;
repeat: current->state = TASK_INTERRUPTIBLE;
        schedule();
        if (*p && *p != current) { //如果队列头进程和刚唤醒的进程current不是一个，说明从队列中间唤醒了一个进程，需要处理
                (**p).state=0;   //将队列头唤醒，并通过goto repeat让自己再去睡眠
                goto repeat;
        }
        *p=NULL;
        if (tmp)
                tmp->state=0;  //作用和sleep_on函数中的一样
}
```

相信实验者已经找到合适的地方插入记录进程从运行到睡眠的语句了。

总的来说，Linux 0.11支持四种进程状态的转移：就绪到运行、运行到就绪、运行到睡眠和睡眠到就绪，此外还有新建和退出两种情况。其中就绪与运行间的状态转移是通过schedule()（它亦是调度算法所在）完成的；运行到睡眠依靠的是sleep_on()和interruptible_sleep_on()，还有进程主动睡觉的系统调用sys_pause()和sys_waitpid()；睡眠到就绪的转移依靠的是wake_up()。所以只要在这些函数的适当位置插入适当的处理语句就能完成进程运行轨迹的全面跟踪了。

为了让生成的log文件更精准，以下几点请注意：

- **进程退出的最后一步是通知父进程自己的退出，目的是唤醒正在等待此事件的父进程。从时序上来说，应该是子进程先退出，父进程才醒来。**
- **schedule()找到的next进程是接下来要运行的进程（注意，一定要分析清楚next是什么）。如果next恰好是当前正处于运行态的进程，swith_to(next)也会被调用。这种情况下相当于当前进程的状态没变。**
- **系统无事可做的时候，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，因为它是唯一一个在CPU上运行的进程，只不过运行的效果是等待。**

管理log文件

日志文件的管理与代码编写无关，有几个要点要注意：

1. 每次关闭bochs前都要执行一下“sync”命令，它会刷新cache，确保文件确实写入了磁盘。
2. 在0.11下，可以用“ls -l /var”或“ll /var”查看process.log是否建立，及它的属性和长度。
3. 一定要实践《实验环境的搭建与使用》一章中关于文件交换的部分。最终肯定要把process.log文件拷贝到主机环境下处理。
4. 在0.11下，可以用“vi /var/process.log”或“more /var/process.log”查看整个log文件。不过，还是拷贝到Ubuntu下看，会更舒服。

5. 在0.11下，可以用“tail -n NUM /var/process.log”查看log文件的最后NUM行。

一种可能的情况下，得到的process.log文件的前几行是：

```
1       N       48      //进程1新建（init()）。此前是进程0建立和运行，但为什么没出现在log文件里？
1       J       49      //新建后进入就绪队列
0       J       49      //进程0从运行->就绪，让出CPU
1       R       49      //进程1运行
2       N       49      //进程1建立进程2。2会运行/etc/rc脚本，然后退出
2       J       49
1       W       49      //进程1开始等待（等待进程2退出）
2       R       49      //进程2运行
3       N       64      //进程2建立进程3。3是/bin/sh建立的运行脚本的子进程
3       J       64
2       E       68      //进程2不等进程3退出，就先走一步了
1       J       68      //进程1此前在等待进程2退出，被阻塞。进程2退出后，重新进入就绪队列
1       R       68
4       N       69      //进程1建立进程4，即shell
4       J       69
1       W       69      //进程1等待shell退出（除非执行exit命令，否则shell不会退出）
3       R       69      //进程3开始运行
3       W       75
4       R       75
5       N       107     //进程5是shell建立的不知道做什么的进程
5       J       108
4       W       108
5       R       108
4       J       110
5       E       111     //进程5很快退出
4       R       111
4       W       116     //shell等待用户输入命令。
0       R       116     //因为无事可做，所以进程0重出江湖
4       J       239     //用户输入命令了，唤醒了shell
4       R       239
4       W       240
0       R       240
……
```

数据统计

为展示实验结果，需要编写一个数据统计程序，它从log文件读入原始数据，然后计算平均周转时间、平均等待时间和吞吐率。任何语言都可以编写这样的程序，实验者可自行设计。我们用python语言编写了一个——stat_log.py（这是python源程序，可以用任意文本编辑器打开）。

python是一种跨平台的脚本语言，号称“可执行的伪代码”，非常强大，非常好用，也非常有用，建议闲着的时候学习一下。其解释器免费且开源，Ubuntu下这样安装：

```
sudo apt-get install python
```

然后只要给stat_log.py加上执行权限（chmod +x stat_log.py）就可以直接运行它。

Windows用户需要先到http://www.python.org下载、安装python解释器，才能运行stat_log.py。

此程序必须在命令行下加参数执行，直接运行会打印使用说明。

```
Usage:
  ./stat_log.py /path/to/process.log [PID1] [PID2] ... [-x PID1 [PID2] ... ] [-m] [-g]
Example:
  # Include process 6, 7, 8 and 9 in statistics only. (Unit: tick)
  ./stat_log.py /path/to/process.log 6 7 8 9
  # Exclude process 0 and 1 from statistics. (Unit: tick)
  ./stat_log.py /path/to/process.log -x 0 1
  # Include process 6 and 7 only. (Unit: millisecond)
  ./stat_log.py /path/to/process.log 6 7 -m
  # Include all processes and print a COOL "graphic"! (Unit: tick)
  ./stat_log.py /path/to/process.log -g
```

运行“./stat_log.py process.log 0 1 2 3 4 5 -g”（只统计PID为0、1、2、3、4和5的进程）的输出示例：

```
(Unit: tick)
Process   Turnaround   Waiting   CPU Burst   I/O Burst
     0           75        67           8           0
     1         2518         0           1        2517
     2           25         4          21           0
     3         3003         0           4        2999
     4         5317         6          51        5260
     5            3         0           3           0
Average:     1823.50     12.83
Throughout: 0.11/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
  50 |   :   #2
  51 |   :   #
  52 |   :   #
  53 |   :   #
  54 |   :   #
  55 |   :   #
  56 |   :   #
  57 |   :   #
  58 |   :   #
  59 |   :   #
  60 |   :   #
  61 |   :   #
  62 |   :   #
  63 |   :   #
  64 |   :   |2  -3
  65 |   :   |   #3
  66 |   :   |   #
  67 |   :   |   #
.....
```

小技巧：如果命令行程序输出过多，可以用“command *arguments* | more”的方式运行，结果会一屏一屏地显示。“more”在Linux和Windows下都有。Linux下还有一个“less”，和“more”类似，但功能更强，可以上下翻页、搜索。

修改时间片

下面是0.11的调度函数schedule，在文件kernel/sched.c中定义为：

```
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;
        }               //找到counter值最大的就绪态进程
        if (c) break;    //如果有counter值大于0的就绪态进程，则退出
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
          if (*p) (*p)->counter = ((*p)->counter >> 1) + (*p)->priority;
                        //如果没有，所有进程的counter值除以2衰减后再和priority值相加，产生新的时间片
}
switch_to(next);  //切换到next进程
```

分析代码可知，0.11的调度算法是选取counter值最大的就绪进程进行调度。其中运行态进程（即current）的counter数值会随着时钟中断而不断减1（时钟中断10ms一次），所以是一种比较典型的时间片轮转调度算法。另外，由上面的程序可以看出，当没有counter值大于0的就绪进程时，要对所有的进程做“(*p)->counter = ((*p)->counter >> 1) + (*p)->priority”。其效果是对所有的进程（包括阻塞态进程）都进行counter的衰减，并再累加priority值。这样，对正被阻塞的进程来说，一个进程在阻塞队列中停留的时间越长，其优先级越大，被分配的时间片也就会越大。所以总的来说，Linux 0.11的进程调度是一种综合考虑进程优先级并能动态反馈调整时间片的轮转调度算法。

此处要求实验者对现有的调度算法进行时间片大小的修改，并进行实验验证。

为完成此工作，我们需要知道两件事情：

1. 进程counter是如何初始化的？
2. 当进程的时间片用完时，被重新赋成何值？

首先回答第一个问题，显然这个值是在fork()中设定的。Linux 0.11的fork()会调用copy_process()来完成从父进程信息拷贝（所以才称其为fork），看看copy_process()的实现（也在kernel/fork.c文件中），会发现其中有下面两条语句：

```
*p = *current;                  //用来复制父进程的PCB数据信息，包括priority和counter
p->counter = p->priority;       //初始化counter
```

因为父进程的counter数值已发生变化，而priority不会，所以上面的第二句代码将p->counter设置成p->priority。每个进程的priority都是继承自父亲进程的，除非它自己改变优先级。查找所有的代码，只有一个地方修改过priority，那就是nice系统调用：

```
int sys_nice(long increment)
{
        if (current->priority-increment>0)
                current->priority -= increment;
        return 0;
```

```
}
```

本实验假定没有人调用过nice系统调用，时间片的初值就是进程0的priority，即宏INIT_TASK中定义的：

```
#define INIT_TASK \
        { 0,15,15, //分别对应state;counter;和priority;
```

接下来回答第二个问题，当就绪进程的counter为0时，不会被调度（schedule要选取counter最大的，大于0的进程），而当所有的就绪态进程的counter都变成0时，会执行下面的语句：

```
(*p)->counter = ((*p)->counter >> 1) + (*p)->priority;
```

显然算出的新的counter值也等于priority，即初始时间片的大小。

提示就到这里。如何修改时间片，自己思考、尝试吧，👉。