进程运行轨迹的跟踪与统计

一、实验目的

（1）掌握 Linux 下的多进程编程技术；

（2）通过对进程运行轨迹的跟踪来形象化进程的概念；

（3）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……

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 `Linux0.11` 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、实验原理

建议实验之前先学习 Bilibili 哈工大李治军老师的相关课程视频 L8 CPU 管理的直观想法和 L9 多进程图像。

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。

1、编写多进程程序。

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);
```

下面是 4 个使用的例子：

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

修改模板 process.c，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。

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 下编译运行。

小技巧：

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。

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。

在 Linux 0.11 下，按 F1 可以即时显示当前所有进程的状态。

2、log 文件

（1）打开 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。

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()，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：

```
// ……
//加载文件系统
setup((void *) &drive_info);

// 打开/dev/tty0，建立文件描述符 0 和/dev/tty0 的关联
```

```
(void) open("/dev/tty0",O_RDWR,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
(void) dup(0);

// ……
```

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，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);

// 建立文件描述符 0 和/dev/tty0 的关联
(void) open("/dev/tty0",O_RDWR,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) dup(0);

(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。

（2）写 log 文件

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);/* 如果输出到 stdout 或 stderr，直接调用 sys_write 即可 */
    if (fd < 3)
    {
        __asm__("push %%fs\n\t"
            "push %%ds\n\t"
            "pop %%fs\n\t"
            "pushl %0\n\t"
        /* 注意对于 Windows 环境来说，是_logbuf,下同 */
            "pushl $logbuf\n\t"
            "pushl %1\n\t"
        /* 注意对于 Windows 环境来说，是_sys_write,下同 */
            "call sys_write\n\t"
            "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。*/
    {
    /* 从进程0的文件描述符表中得到文件句柄 */
        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;
}
```

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：

```
// 向 stdout 打印正在运行的进程的 ID

fprintk(1, "The ID of running process is %ld", current->pid);
// 向 log 文件输出跟踪进程运行轨迹
fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'R', jiffies);
```

3、滴答数（jiffies）

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 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:!    ……! 增加 jiffies 计数值
    incl jiffies!    ……
```

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：

```
// 设置 8253 模式
outb_p(0x36, 0x43);
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：

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

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。

4、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。

例子 1：记录一个进程生命期的开始

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
!    ……
! 传递一些参数
    push %gs
    pushl %esi
    pushl %edi
    pushl %ebp
    pushl %eax
! 调用 copy_process 实现进程创建
    call copy_process
```

```
    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;//    ……// 获得一个 task_struct 结构体空间
    p = (struct task_struct *) get_free_page();//    ……
    p->pid = last_pid;//    ……// 设置 start_time 为 jiffies
    p->start_time = jiffies;
    fprintk(3,"%d\t%c\t%d\n",p->pid,'N',jiffies);
    //       ……
    /* 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。*/
    p->state = TASK_RUNNING;
    fprintk(3,"%d\t%c\t%d\n",p->pid,'J',jiffies);
    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）”。

例子 2：记录进入睡眠态的时间

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;// 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
    *p = current;// 切换到睡眠态
    current->state = TASK_UNINTERRUPTIBLE;// 让出 CPU
    schedule();// 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!!!
    if (tmp)
        tmp->state=0;
}

/* 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();// 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
    if (*p && *p != current) {
// 将队列头唤醒，并通过 goto repeat 让自己再去睡眠
        (**p).state=0;
        goto repeat;
    }
    *p=NULL;//作用和 sleep_on 函数中的一样
    if (tmp)
        tmp->state=0;
}
```

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。

5、修改好相关文件后，编译运行内核，在 0.11 中编译运行 process.c 后，关闭 boches，将生成的 process.log 文件拷贝到 ubuntu 下查看

6、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：

- 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。
- 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。
- 一定要实践实验一（内核编译运行）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。
- 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。
- 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
……
```

7、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：

```
# 在实验楼的环境中已经安装了 python，可以不必进行此操作
$ sudo apt-get install python
```

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

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。

在 ubuntu 中运行$ ./stat_log.py process.log 0 1 2 3 4 5 （则只统计 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
```

8、修改时间片

为完成此工作，我们需要知道两件事情：

- 进程 counter 是如何初始化的
- 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：

```
// 用来复制父进程的 PCB 数据信息，包括 priority 和 counter
*p = *current;
// 初始化 counter
p->counter = p->priority;// 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（在 include/linux/sched.h）：

```
#define INIT_TASK \
    { 0,15,15,// 上述三个值分别对应 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。

四、实验思考

（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？