第5节 进程跟踪实验

这个实验包含几个小部分，我们只做里面前2个，后面分析日志文件我们就不做了。

实验的第一个部分就在ubuntu下完成即可，也就是process就在ubuntu下运行，不在linux-0.11里面运行了。第二个部分是修改linux-0.11里面进程状态变化的几个函数，在进程状态变化的时候向日志文件打印信息，因此第二个部分修改完核心函数后，只要运行linux-0.11，后面就自动向日志记录信息了，然后关闭挂载hdc就能查看日志了。完成后的实验打包成oslab2.tar

实验的第一个部分

首先会给你一个现成的写好的process.c作为参考，你只要复制到oslab目录下即可。关于process.c里面有两个函数，下面先讲解这两个函数，讲完后再告诉你这个实验要你干啥。

process.c中的cpuio_bound()

```
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;
    }
```

```
}
```

这个程序里面的cpuio_bound()函数是一个进程，这个进程干的事情是模拟cpu干活和io干活，具体来说，给定三个参数last,cpu_time,io_time，last是这个进程总共运行持续的时间，一旦到达这个时间，这个进程就结束。然后cpu_time,io_time分别指的是这个进程一会使用cpu运行一段时间，一会使用io休眠一段时间，当然，这里只是从逻辑上模拟，实际都是通过cpu完成的。

举个例子，调用cpuio_bound(20，3，2)，这个进程是这样运行的：

cpu3秒->io2秒->cpu3秒->io2秒->cpu3秒->io2秒->cpu3秒->io2秒->do_exit

把cpu运行3秒通过一个循环实现，io运行2秒通过sleep实现。

然后cpu运行3秒这个数据计算，实际是通过进程PCB数据结构里面的utime和stime数值算出来的，具体是这个原理：

系统时钟中断每隔10ms(设定HZ=100的情形下)发送一个中断信号调用时钟中断time_interupt，然后这个中断调用一些列函数完成一些事情，其中就包括使得current->utime或current->stime也就是当前进程用户态或内核态运行的滴答数+1，这样就完成了当前运行进程使用cpu的内核态和用户态的运行时间实时更新，注意，你可以假装执行这个时钟中断是瞬间完成的，不耗费时间，然后这样你就当作这个进程运行过程中，内部两个统计运行时间的变量在实时进行更新。

然后你应该懂了，怎么使用循环让这个进程占用3秒cpu时间了吧，就是不停循环，不停拿到进程的时间，看时间到底够不够3秒，不够3秒继续循环，够3秒就结束循环，就是这么一个逻辑。

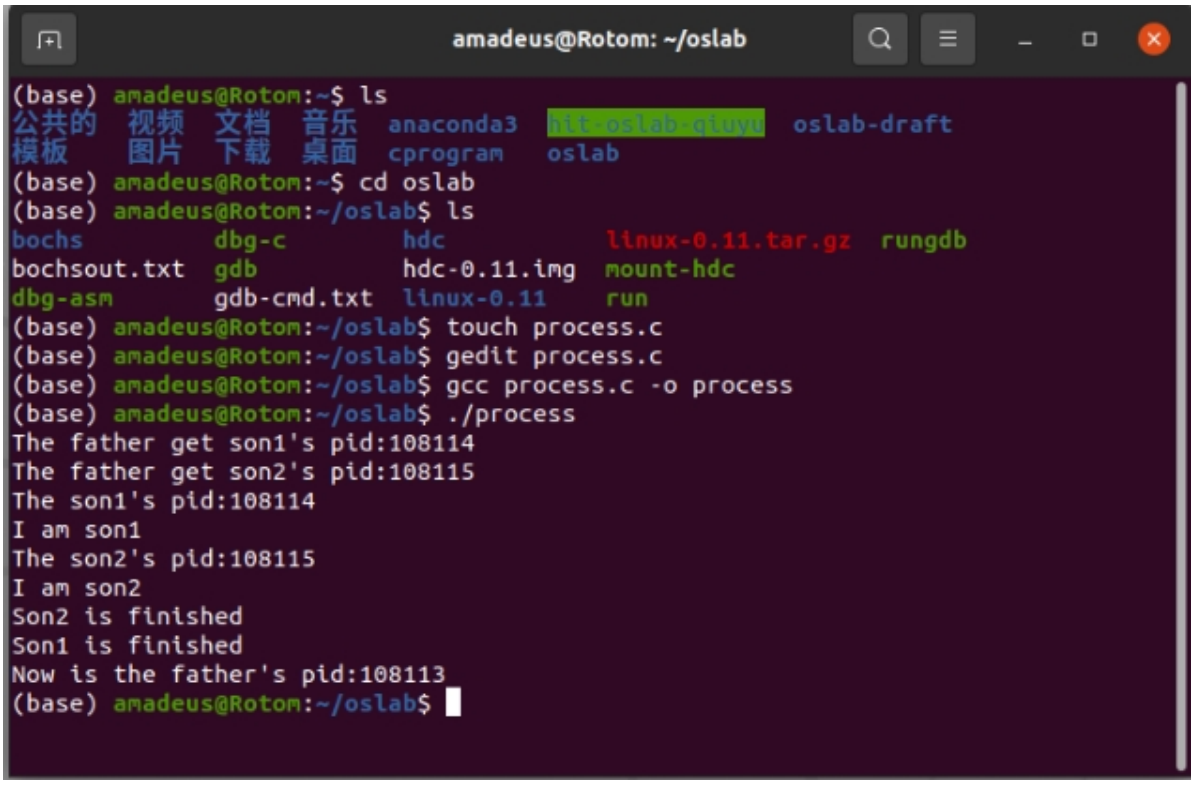
至于io运行2秒，就是简单sleep（2）就行了，有兴趣自己去查看它是如何实现的，这里你就当作它也是一个死循环，直到时间够2秒才退出循环。

process.c中的main

```
int main(int argc, char * argv[])
{
    pid_t father,son1,son2,son3,tmp1,tmp2,tmp3;
    tmp1=fork();
    if(tmp1==0)         /* son1 */
    {
        son1=getpid();
        printf("The son1's pid:%d\n",son1);
        printf("I am son1\n");
        cpuio_bound(10, 3, 2);
        printf("Son1 is finished\n");
    }
    else if(tmp1>0)
    {
        son1=tmp1;
        tmp2=fork();
        if(tmp2==0)     /* son2 */
        {
            son2=getpid();
            printf("The son2's pid:%d\n",son2);
            printf("I am son2\n");
            cpuio_bound(5, 1, 2);
            printf("Son2 is finished\n");
```

```
        }
        else if(tmp2>0)     /* father */
        {
            son2=tmp2;
            father=getpid();
            printf("The father get son1's pid:%d\n",tmp1);
            printf("The father get son2's pid:%d\n",tmp2);
            wait((int *)NULL);
            wait((int *)NULL);
            printf("Now is the father's pid:%d\n",father);
        }
        else
            printf("Creat son2 failed\n");
    }
    else
        printf("Creat son1 failed\n");
    return 0;
}
```

这个main函数做的事情就是跟踪fork出来的进程和父进程main的运行轨迹，在这个程序里面标记了多个输出，从运行结果就能知道所有进程运行轨迹。想要完全理解这个程序的运行结果，就需要你彻底搞懂fork源代码以及进程调度（什么时候发生进程切换）。注意，我们这个程序就不弄到linux-0.11中运行了，直接在我们的ubuntu上运行。当然，你也可以在linux-0.11中试试看，应该也能运行。运行结果如下：



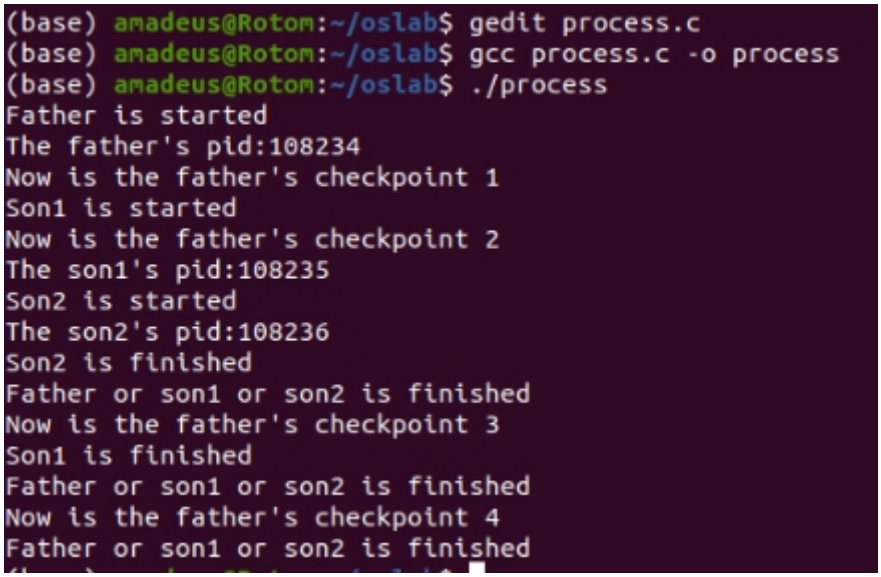
改进后的main

为了更方便观察运行到的位置，我们对main稍作修改

```
int main(int argc, char * argv[])
{
    pid_t father,son1,son2,son3,tmp1,tmp2,tmp3;
```

```
    father=getpid();
    printf("Father is started\n");
    printf("The father's pid:%d\n",father);

    tmp1=fork();
    if(tmp1==0)         /* son1 */
    {
        son1=getpid();
        printf("Son1 is started\n");
        printf("The son1's pid:%d\n",son1);
        cpuio_bound(10, 3, 2);
        printf("Son1 is finished\n");
    }
    else if(tmp1>0)
    {
        printf("Now is the father's checkpoint 1 \n");
        son1=tmp1;
        tmp2=fork();
        if(tmp2==0)     /* son2 */
        {
            son2=getpid();
            printf("Son2 is started\n");
            printf("The son2's pid:%d\n",son2);
            cpuio_bound(5, 1, 2);
            printf("Son2 is finished\n");
        }
        else if(tmp2>0)     /* father */
        {
            son2=tmp2;
            printf("Now is the father's checkpoint 2 \n");
            wait((int *)NULL);
            printf("Now is the father's checkpoint 3 \n");
            wait((int *)NULL);
            printf("Now is the father's checkpoint 4 \n");
        }
        else
            printf("Creat son2 failed\n");
    }
    else
        printf("Creat son1 failed\n");
    printf("Father or son1 or son2 is finished \n");
    return 0;
}
```

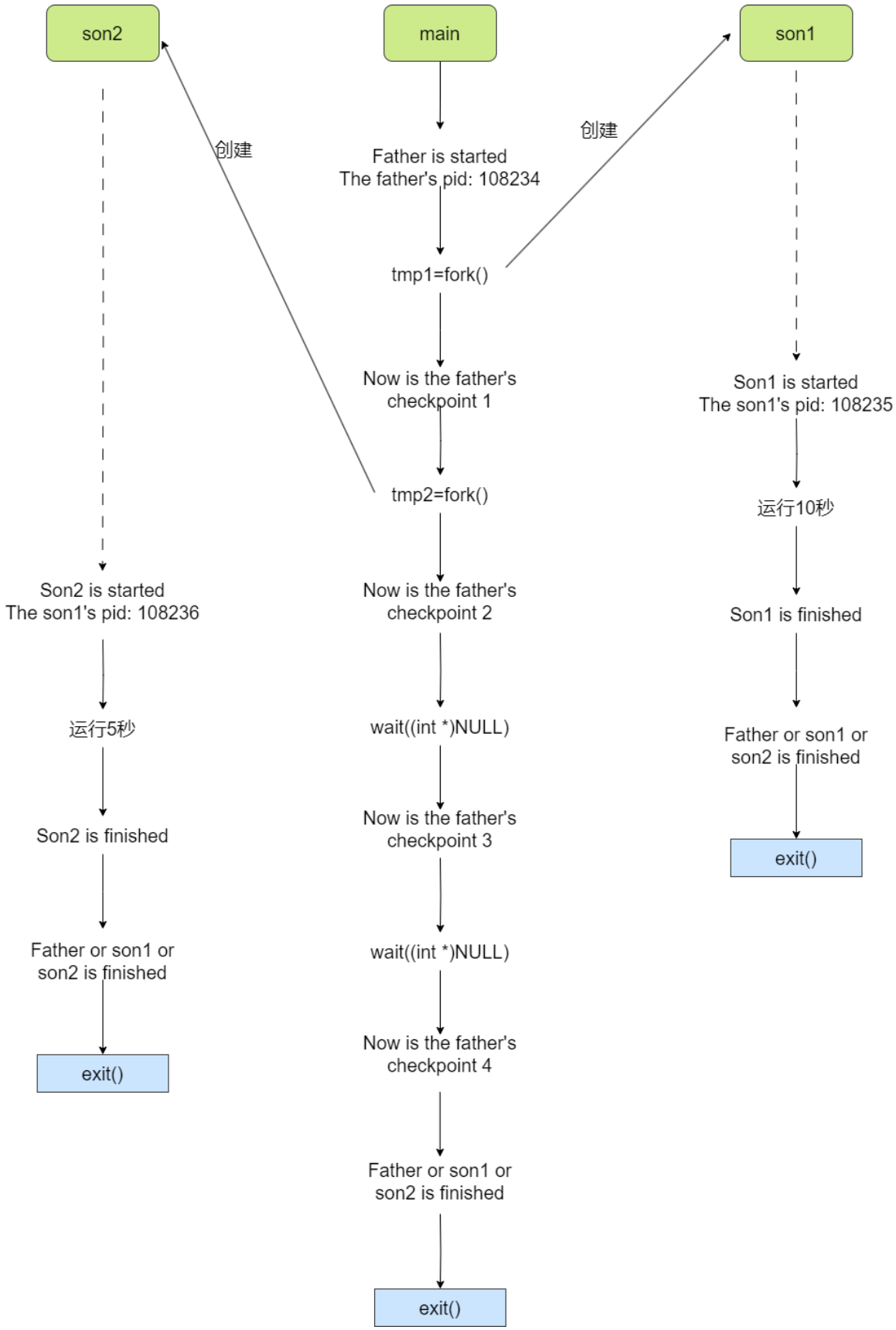


改进后的process分析

运行过程如下图，主要有几个关键的点，首先son1和son2在main创建他俩之前是不可能运行的，创建之后才有可能会被时钟中断引发的调度程序运行（ubuntu系统里面的fork里面可能也会有调度程序）。第一个fork创建出son1后，调度进程会让main和son1之前来回切换，具体请查看上面运行结果，son1 is started后，并不是一直在son1中运行到son1 is finished，而是会被调度的main中进行，并创建了son2，此时调度程序就会来回在这3个进程中来回切换，但是，main运行到wait会停下来进入休眠，等待某个进程结束再被唤醒重新参与调度。

另外注意，Father or son1 or son2 is finished这句话不在任何一个分支中，是所有进程都会执行的代码片段，因此前面分支里面的son1 is finished和son2 is finished不准确。

process.c程序分析



有兴趣的可以继续修改上面的程序，可以在cpuio_bound()进程里面的几个地方添加记录点，这样在进程运行的时候就能知道到底是一次性运行完，还是中途切换到其他进程中运行。

实验的第二个部分

参考链接：

(浓缩+精华)哈工大-操作系统-MOOC-李治军教授-实验3-进程运行轨迹的跟踪与统计_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-CSDN博客

操作系统实验（3：进程运行轨迹的跟踪与统计）-CSDN博客

哈工大操作系统实验(三)进程管理 - 简书 (jianshu.com)

前面第一部分还只是在用户程序层面上记录进程的活动，实验的第二个部分要求修改内核函数，来记录系统启动后每个进程（除了0号进程）的活动轨迹，也就是每个进程的状态，具体实验原本的要求如下：

- 基本任务是在内核中维护一个日志文件*/var/process.log*，把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

  /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
    - N 进程新建
    - J 进入就绪态
    - R 进入运行态
    - W 进入阻塞态
    - E 退出
  - 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
  ......
  ```

实验分析

首先想要理解这个实验你必须理解进程的全生命周期过程中状态的变化

1. 【创建，就绪】先讲最简单的，一个进程首先由父进程fork创建并进入就绪态，这两个状态变化都是在fork里面的copy_process里面切换的，因此我们需要在kernel/fork.c文件里面的copy_process函数中添加两行代码，把这个进程的创建的信息（N）和变成就绪状态的信息（J）打印出来，也就是pid号，状态信息，当前时刻滴答数。
2. 【运行】就绪态的进程会被系统时钟中断进行调度，或者在其他某个地方被调度，一旦进行调度，进程就进入运行态，因此我们需要在调度函数schdule中添加代码，打印进程变成运行态的信息。
3. 【挂起（睡眠，阻塞）】和进程挂起相关的函数主要有这么4个：pause（调用sys_pause），sleep_on，interruptible_sleep_on，wait(调用sys_waitpid)
4. 【唤醒】进程挂起后，需要重新唤醒进入就绪态，涉及到的函数有：wake_up，schdule（根据信号唤醒进程的代码片段）
5. 【退出（僵死）】exit（调用do_exit）

注意，这里的实验本身不难，就是在上面涉及的函数中改变进程状态的后面添加输出进程状态信息的代码而已，但是仅仅这样实际上是学不到很多东西的，真正需要弄懂的是每个函数在哪个地方使用，什么时候进程被调度，什么时候进程被阻塞，什么时候进程变成僵尸，什么时候进程PCB被回收，正确的做法是把这些函数的源代码的核心部分在做什么事情给搞懂。如果每个进程都不涉及IO，都不会进入阻塞休眠状态，那么所有进程的运行简单靠着时钟中断进行调度，直到运行结束调用do_exit回收资源变成僵尸，然后发送给父进程信号，由父进程（shell）调用wait给子进程收尸（回收PCB）。整个运行过程相对来说不是很复杂，但是如果涉及到了IO，而IO涉及到的函数通常比较复杂（函数调用一层套一层，比较难以摸清楚每个函数干什么事情，以及IO是如何在这么多函数配合下完成的），这样就会导致当这个进程进行IO的时候，进程经历了什么状态变化，具体发生的过程细节是什么样的，你就很难搞明白。因此如果到这里还没搞懂IO的全过程，那就不用理会，就是简单修改睡眠唤醒相关函数即可，不用再去思考整个事情发生的细节过程。但是前面从0号进程，init进程，shell进程，再到其他进程的创建，运行，调度，结束，这些事情必须要非常详细的搞清楚怎么回事。

另外，实验要求我们把输出的信息打印到log日志中，因为打印到log日志，我们需要在0号进程中把log文件与文件描述符3建立关联，然后后面由于每个进程fork会复制父进程的文件描述符，因此所有子进程都继承了这个文件描述符3，通过向文件描述符3写入我们要打印的信息，就能正确在log日志文件中写入。

实验细节

1. **提前打开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()`。

原先文件描述符关联是在进程1中的init函数中完成的，现在拷贝到进程1创建之前的进程0当中。

原始的init和main如下：

```
void init(void)
{
    // ...前置代码片段...
    setup((void *) &drive_info);

    (void) open("/dev/tty0",O_RDWR,0);  // 用读写访问方式打开设备“/dev/tty0”，
                                        // 这里对应终端控制台。
                                        // 返回的句柄号0 -- stdin 标准输入设备。
```

```
    (void) dup(0);      // 复制句柄，产生句柄1 号-- stdout 标准输出设备。
    (void) dup(0);      // 复制句柄，产生句柄2 号-- stderr 标准出错输出设备。
    printf("%d buffers = %d bytes buffer space\n\r",NR_BUFFERS, \
        NR_BUFFERS*BLOCK_SIZE); // 打印缓冲区块数和总字节数，每块1024 字节。
    printf("Free mem: %d bytes\n\r",memory_end-main_memory_start);//空闲内存字节
数。
    // ...后置代码片段...

}

void main(void)
{
    // ...前置代码片段...
    move_to_user_mode();    // 移到用户模式
    // 就是要在这个地方提前打开文件，关联文件描述符
    if (!fork()) {      // 创建进程1，在进程1的条件下执行init
        init();
    }
    for(;;) pause();
}
```

修改后的init和main如下：

```
void init(void)
{
    // ...前置代码片段...

    // ...后置代码片段...

}

void main(void)
{
    // ...前置代码片段...
    move_to_user_mode();    // 移到用户模式
    setup((void *) &drive_info);

    (void) open("/dev/tty0",O_RDWR,0);  // 用读写访问方式打开设备“/dev/tty0”，
                                        // 这里对应终端控制台。
                                        // 返回的句柄号0 -- stdin 标准输入设备。
    (void) dup(0);      // 复制句柄，产生句柄1 号-- stdout 标准输出设备。
    (void) dup(0);      // 复制句柄，产生句柄2 号-- stderr 标准出错输出设备。
    printf("%d buffers = %d bytes buffer space\n\r",NR_BUFFERS, \
        NR_BUFFERS*BLOCK_SIZE); // 打印缓冲区块数和总字节数，每块1024 字节。
    printf("Free mem: %d bytes\n\r",memory_end-main_memory_start);//空闲内存字节
数。
    if (!fork()) {      // 创建进程1，在进程1的条件下执行init
        init();
    }
    for(;;) pause();
}
```

这样修改后，从进程1开始，就会继承这些文件描述符，然后就支持标准输入标准输出和标准错误了。

但是我们需要在自己的/var/process.log文件中保存信息，因此需要分配文件描述符3给我们自己的文件。

再次修改后的main文件如下：

```
void main(void)
{
    // ...前置代码片段...
    move_to_user_mode();    // 移到用户模式
    setup((void *) &drive_info);

    (void) open("/dev/tty0",O_RDWR,0);  // 用读写访问方式打开设备“/dev/tty0”，
                                        // 这里对应终端控制台。
                                        // 返回的句柄号0 -- stdin 标准输入设备。
    (void) dup(0);      // 复制句柄，产生句柄1 号-- stdout 标准输出设备。
    (void) dup(0);      // 复制句柄，产生句柄2 号-- stderr 标准出错输出设备。
    printf("%d buffers = %d bytes buffer space\n\r",NR_BUFFERS, \
        NR_BUFFERS*BLOCK_SIZE); // 打印缓冲区块数和总字节数，每块1024 字节。
    printf("Free mem: %d bytes\n\r",memory_end-main_memory_start);//空闲内存字节
数。
    (void) open("/var/process.log",O_CREAT|O_TRUNC|O_WRONLY,0666);
    if (!fork()) {      // 创建进程1，在进程1的条件下执行init
        init();
    }
    for(;;) pause();
}
```

文件描述符3我们确实准备好了，但是在内核态我们使用哪个函数向这个文件写入数据呢，一般来说这个任务由sys_write来完成的，并且为了格式化输出，这里哈工大课程里面给我们提供了fprintk函数，结合了sys_write和printk，我们直接拷贝到kernel/printk.c中（原来的内容清空）即可。

```
/*
 *  linux/kernel/printk.c
 *
 *  (C) 1991  Linus Torvalds
 */

/*
 * When in kernel-mode, we cannot use printf, as fs is liable to
 * point to 'interesting' things. Make a printf with fs-saving, and
 * all is well.
 */
#include <stdarg.h>
#include <stddef.h>
#include <linux/sched.h>
#include <sys/stat.h>
#include <linux/kernel.h>

static char buf[1024];

extern int vsprintf(char * buf, const char * fmt, va_list args);

int printk(const char *fmt, ...)
{
```

```
    va_list args;
    int i;

    va_start(args, fmt);
    i=vsprintf(buf,fmt,args);
    va_end(args);
    __asm__("push %%fs\n\t"
        "push %%ds\n\t"
        "pop %%fs\n\t"
        "pushl %0\n\t"
        "pushl $buf\n\t"
        "pushl $0\n\t"
        "call tty_write\n\t"
        "addl $8,%%esp\n\t"
        "popl %0\n\t"
        "pop %%fs"
        ::"r" (i):"ax","cx","dx");
    return i;
}

/*
*  write by sunner
*/
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;
}
```

`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);
```

我们后面要做的事情就是在各个内核函数中把`fprintk(3, "%ld\t%c\t%ld\n", current->pid, '状态', jiffies);`放在合适的地方来跟踪打印进程信息。

由于单个字母表示状态不是很直观，这里我们重新修改几个状态的表示

- Created 进程新建
- Ready 进入就绪态
- Runnning 进入运行态
- Blocked 进入阻塞态
- Exit 退出

```
fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'Created', jiffies);
fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'Ready', jiffies);
fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'Runnning', jiffies);
fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'Blocked', jiffies);
fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'Exit', jiffies);
```

2. **修改fork文件**

```
_sys_fork:
call _find_empty_process # 调用find_empty_process()(kernel/fork.c,135)。
testl %eax,%eax
js 1f push %gs
pushl %esi
pushl %edi
pushl %ebp
pushl %eax
call _copy_process       # 调用C 函数copy_process()(kernel/fork.c,68)。
addl $20,%esp            # 丢弃这里所有压栈内容。
1: ret
```

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

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“Created（新建）”和“Ready（就绪）”。

```
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;
//  这里输出状态信息
    fprintk(3, "%ld\t%c\t%ld\n", p->pid, 'Created', 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;
//  这里输出状态信息
    fprintk(3, "%ld\t%c\t%ld\n", p->pid, 'Ready', jiffies);

    return last_pid;
}
```

3. **修改调度算法**

调度函数schedule在`kernel/sched.c`中定义的，这个函数就是遍历task数组（除了0号进程）,然后查找所有状态为就绪态的进程，从里面选个counter最大的作为next进程，然后切换过去。当然，如果next还是当前进程，在切换switch_to(next)中会省略切换过程，直接什么也不做。如果所有就绪态的进程时间片counter都用完了，会重新给counter赋值，然后回到循环开始重新查找next。如果1-63号进程槽都为空，则会直接切换到0号进程。

```
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){
                (*p)->state=TASK_RUNNING;

            }
        }

while (1) {
    c = -1; next = 0; i = NR_TASKS; p = &task[NR_TASKS];

// 找到 counter 值最大的就绪态进程
    while (--i) {
        if (!*--p)    continue;
        if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
            c = (*p)->counter, next = i;
    }

// 如果有 counter 值大于 0 的就绪态进程，则退出
    if (c) break;

// 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
    for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
          if (*p) (*p)->counter = ((*p)->counter >> 1) + (*p)->priority;
}

// 切换到 next 进程
switch_to(next);
```

修改后的代码：

```
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){
                (*p)->state=TASK_RUNNING;
                // 唤醒父进程，这里暂时不用理会
                fprintk(3, "%ld\t%c\t%ld\n", (*p)->pid, 'Ready', jiffies);
            }
        }

while (1) {
    c = -1; next = 0; i = NR_TASKS; p = &task[NR_TASKS];

// 找到 counter 值最大的就绪态进程
```

```
    while (--i) {
        if (!*--p)    continue;
        if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
            c = (*p)->counter, next = i;
    }

// 如果有 counter 值大于 0 的就绪态进程，则退出
    if (c) break;

// 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
    for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
          if (*p) (*p)->counter = ((*p)->counter >> 1) + (*p)->priority;
}
//添加的代码
if(current != task[next]){
    if(current->state == TASK_RUNNING){
        fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'Ready', jiffies);
    }
    fprintk(3, "%ld\t%c\t%ld\n", task[next]->pid, 'Running', jiffies);
}
// 切换到 next 进程
switch_to(next);
```

主要就是在`switch_to(next)`前面添加了下面这部分代码：

```
if(current != task[next]){
    if(current->state == TASK_RUNNING){
        fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'Ready', jiffies);
    }
    fprintk(3, "%ld\t%c\t%ld\n", task[next]->pid, 'Running', jiffies);
}
```

只有`current != task[next]`，也就是选中的next进程不是当前进程的时候，才会去记录信息。

然后就是需要记录两个信息，一个是讲当前进程的信息记录为就绪态，因为马上当前进程就要被切走了，另一个是next进程记录为运行态。

4. **修改进程阻塞和唤醒相关的代码**

- sleep_on() 和 interruptible_sleep_on()

  这两个函数在`kernel/sched.c`文件中，修改后的代码如下：

```
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;
//  添加代码记录信息
```

```
    fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'Blocked', jiffies);
// 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;
        //  添加代码记录信息
        fprintk(3, "%ld\t%c\t%ld\n", tmp->pid, 'Ready', jiffies); }
}
```

```
void interruptible_sleep_on(struct task_struct **p)
{
    struct task_struct *tmp;
       …
    tmp=*p;
    *p=current;
repeat:    current->state = TASK_INTERRUPTIBLE;
           //  添加代码记录信息
            /*0号进程是守护进程，cpu空闲的时候一直在waiting，输出它的话是不会通过脚本检查的
哦*/
            if(current->pid != 0)
            fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'Blocked', jiffies);
    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;
        //  打印状态信息
        fprintk(3, "%ld\t%c\t%ld\n", (*p)->pid, 'Ready', jiffies);
        goto repeat;
    }
    *p=NULL;
//作用和 sleep_on 函数中的一样
    if (tmp){
        tmp->state=0;
        //  添加代码记录信息
        fprintk(3, "%ld\t%c\t%ld\n", tmp->pid, 'Ready', jiffies); }
}
```

- sys_pause()

也在 `kernel/sched.c` 文件中，这个代码也很简单，就是强制使得进程进入TASK_INTERRUPTIBLE阻塞状态，然后调度其他进程运行，如果调度过程中，这个进程收到信号，并在schdule中唤醒，当再次被调度的时候，将运行到return 0直接结束sys_pasue，然后执行暂停之后的代码。这就是这个函数的原理。

修改后的代码如下：

```
int sys_pause(void)
{
    current->state = TASK_INTERRUPTIBLE;
    //  添加代码记录信息
    if(current->pid != 0)
        fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'Blocked', jiffies);
    schedule();
    return 0;
}
```

- sys_waitpid()

在 `kernel/exit.c` 文件中，修改后的代码如下：

```
int sys_waitpid(pid_t pid,unsigned long * stat_addr, int options)
{
    int flag, code;
    struct task_struct ** p;

    verify_area(stat_addr,4);
repeat:
    flag=0;
    for(p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
        // 前面这几个if就是排除一些进程，来找到当前进程的子进程
        if (!*p || *p == current)
            continue;
        if ((*p)->father != current->pid)
            continue;
        if (pid>0) {
            if ((*p)->pid != pid)
                continue;
        } else if (!pid) {
            if ((*p)->pgrp != current->pgrp)
                continue;
        } else if (pid != -1) {
            if ((*p)->pgrp != -pid)
                continue;
        }
        // 检查子进程的状态，如果是僵尸，直接收尸返回尸体pid
        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;
        }
    }
    // 处理default情形，也就是子进程还没变成僵尸，父进程根据options是否为WNOHANG决定是否挂起
    // 如果options==WNOHANG，父进程不会阻塞等待，直接返回继续执行后面的代码，对于shell来说就是
    // 回到循环开始获取用户输入的命令
    // 如果options!=WNOHANG，父进程会被挂起，直到子进程tell_father(current->father);然后
    // schdule重新唤醒父进程，然后父进程后面再次被调度的时候，重新goto repeat
    if (flag) {
        if (options & WNOHANG)
            return 0;
        current->state=TASK_INTERRUPTIBLE;
        //  添加代码记录父进程阻塞信息
        // 0号进程是守护进程，cpu空闲的时候一直在waiting，输出它的话不能通过脚本检查
        if(current->pid != 0)
            fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'Blocked', jiffies);
        schedule();
        if (!(current->signal &= ~(1<<(SIGCHLD-1))))
            goto repeat;
        else
            return -EINTR;
    }
    return -ECHILD;
}
```

- wake_up()

在 `kernel/sched.c` 文件中，修改后的代码如下：

```
void wake_up(struct task_struct **p)
{
    if (p && *p) {
        if((**p).state != TASK_RUNNING){
            (**p).state=TASK_RUNNING;
            //  添加代码记录
            fprintk(3, "%ld\t%c\t%ld\n", (*p)->pid, 'Ready', jiffies);
        }
    }
}
```

5. **修改进程退出代码**

在 `kernel/exit.c` 文件中，修改后的代码如下：

```
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;
    // 添加代码记录退出信息
    fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'Exit', jiffies);
    current->exit_code = code;
    tell_father(current->father);
    schedule();
    return (-1);    /* just to suppress warnings */
}

int sys_exit(int error_code)
{
    return do_exit((error_code&0xff)<<8);
}
```

实验结果

注意：上面所有fprintk都写错了，需要把%c改为%s，单引号改成双引号，否则会输出不正确。

通过上面修改后，从main中开始执行第一个fork开始，也就是创建出1号进程开始，就会不停向日志文件输出信息，背后也会创建杂七杂八的进程，不过最终是停留在shell进程等待用户输入，然后我们就可以输入sync确保将文件写入磁盘。就可以关闭bochs虚拟机了，然后在ubuntu下挂载hdc，找到我们的日志文件用gedit打开它，就能看到内容了，我的运行结果如下（只截取前100行，需要打开相应的代码，结合后面的注释才能看懂，注释是我自己参考其他人的写的，不一定正确）：

```
// 前5行，主要从move_to_usermode运行到pause
1   Created 48  // 进程0使用fork创建进程1
1   Ready   48  // 还是在fork里面，新建后立马改变状态进入就绪态
1   Ready   48  // 不清楚为啥又出现一次
0   Ready   48  // 进程0执行到pause（虽然被挂起，但是由于是进程0，不输出它的Blocked状态）
                // 然后在sys_pause中调用schedule，这个里面自动输出当前进程为Ready，
                // 被调度的下一个next进程为Running，也就是下一行
                // 实际这里显示进程0为Ready不正确，不过被schdule输出，知道就行
1   Running 48
```

```
// 从这里开始执行init
2   Created 48  // 使用fork创建进程2，随即进入Ready（下一行）
2   Ready   49
1   Blocked 49  // 然后执行到wait，被阻塞
2   Ready   49  // 不清楚为啥又出现一次
1   Ready   49  // sys_waitpid阻塞后，会调用schdule，里面让当前1号变成Ready,next2号变成
Running
2   Running 49

// 从这里开始运行shell
3   Created 63  // 2号进程就是shell进程，会fork出3号进程
3   Ready   64
3   Ready   64
2   Ready   64
1   Ready   64
2   Ready   64
3   Running 64
3   Blocked 68
3   Ready   68
2   Ready   68
1   Ready   68
2   Running 68
2   Exit    74
3   Ready   74
2   Ready   74
1   Ready   74
1   Running 74
3   Ready   74
1   Ready   74
4   Created 74
4   Ready   74
1   Blocked 74
3   Ready   74
4   Ready   74
1   Ready   74
4   Running 74
3   Ready   89
4   Ready   89
1   Ready   89
3   Ready   104
4   Ready   104
1   Ready   104
5   Created 107
5   Ready   107
4   Blocked 107
5   Ready   107
3   Ready   107
4   Ready   108
1   Ready   108
5   Running 108
4   Ready   109
5   Exit    110
5   Ready   110
3   Ready   110
4   Ready   110
```

```
1   Ready   110
4   Running 110
0   Ready   110
3   Ready   110
4   Ready   110
1   Ready   110
4   Blocked 116
3   Ready   116
4   Ready   116
1   Ready   116
0   Running 116
3   Ready   116
4   Ready   116
1   Ready   116
3   Ready   116
4   Ready   116
1   Ready   116
3   Ready   116
4   Ready   116
1   Ready   116
3   Ready   116
4   Ready   116
1   Ready   116
3   Ready   116
4   Ready   116
1   Ready   116
3   Ready   116
4   Ready   116
1   Ready   116
3   Ready   116
4   Ready   117
1   Ready   117
3   Ready   117
4   Ready   117
1   Ready   117
3   Ready   117
4   Ready   117
1   Ready   117
3   Ready   117
4   Ready   117
1   Ready   117
3   Ready   117
4   Ready   117
1   Ready   117
```