

Lab5 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。

2.1. 内容一

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；

2.2. 内容二

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。

2.3. 内容三

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。

2.4. 内容四

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

3.1. 编写样本程序

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;

3.1.1. 样本程序

样本程序是一个在 `unix/Linux` 系统上运行的生成各种进程的程序, 可以用来检验有关 `Log` 文件的修改是否正确, 还是数据统计工作的基础。操作系统对这些进程的调度情况都记录到 `process.log` 中。修改系统的调度算法或调度参数, 再运行样本程序后, 得到新的 `Log` 文件。通过比较修改前后的 `Log` 文件, 可以比较调度算法的优缺点。

3.1.2. `process.c` 模板

`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。

- 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。

3.1.3. `process.c` 完整代码

通过 `process.c` 模板和模板修改参考规则, `process` 完整代码及其具体分析如下所示。

```
#include <stdio.h>
#include <sys/times.h>
#include <time.h>
#include <unistd.h>

#define HZ 100

void cpuio_bound(int last, int cpu_time, int io_time);

int main(int argc, char* argv[]) {
    pid_t n_proc[10]; /* 数组存放10个子进程的 PID */
    int i; /* 声明一个整数 i 用于循环计数 */
    for (i = 0; i < 10; i++) { /* 循环10次, 每次创建一个子进程 */
        n_proc[i] = fork(); /* 调用 fork 函数创建子进程, fork 会返回两种值 */
        /* 在父进程中返回子进程的 PID */
        /* 在子进程中返回0 */

        /* 子进程 */
        if (n_proc[i] == 0) {
            cpuio_bound(10, i, 10 - i); /* 每个子进程都占用10s */
            return 0; /* 执行完cpuio_bound 以后, 结束该子进程 */
        }
        /* fork 失败 */
        else if (n_proc[i] < 0) {
            printf("Failed to fork child process %d!\n", i + 1);
            return -1; /* 结束子进程 */
        }
        /* 父进程继续fork */
    }
    /* 打印所有子进程PID */
    for (i = 0; i < 10; i++) printf("Child PID: %d\n", n_proc[i]);
}
```

```

/* 等待所有子进程完成 */
wait(&i);          /* Linux 0.11 上 gcc要求必须有一个参数, gcc3.4+则不需要 */
return 0;          /* 结束父进程 */
}
/*
* 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) {
    /* 定义两个 tms 结构体, 用于保存进程的时间统计, 包括用户时间和系统时间 */
    struct tms start_time, current_time;
    clock_t utime, stime;          /* 保存用户时间和系统时间的变量 */
    int sleep_time;                /* 用于记录IO操作的时间 */
    /* 循环, 直到last小于等于0, 表示函数已经执行完所有的CPU和IO操作 */
    while (last > 0) {
        /* CPU Burst */
        times(&start_time);        /* 获取当前进程的CPU时间, 存储到start_time中 */
        /* 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。
        */
        /* 循环用于持续占用CPU, 直到占用的CPU时间达到cpu_time秒
        * times 返回的 utime 和 stime 分别表示用户态和内核态的时间
        */
        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;          /* 每次占用CPU时间后减少last, 表示剩余时间 */

        if (last <= 0) break;      /* 如果last小于等于0, 退出循环 */
        /* IO Burst */
        /* 用sleep(1)模拟1秒钟的I/O操作 */
        sleep_time = 0;
        while (sleep_time < io_time) {
            /* 模拟I/O操作, 使用 sleep(1) 每次让进程暂停1秒, 直到总共暂停的时间达到
            io_time 秒 */
            sleep(1);
            sleep_time++;
        }
        last -= sleep_time;        /* IO Burst */
    }
}

```

3.1.4. 运行程序测试

1. 编译 Linux-0.11 源码

切换到 `Linux-0.11` 目录下执行以下命令, 生成 `Image` 镜像文件。

```
make clean
make Image
```

```
make[1]: Leaving directory '/home/guojun/oslab/linux-0.11/kernel/bl
k_drv'
make[1]: Entering directory '/home/guojun/oslab/linux-0.11/kernel/c
hr_drv'
sync
make[1]: Leaving directory '/home/guojun/oslab/linux-0.11/kernel/ch
r_drv'
make[1]: Entering directory '/home/guojun/oslab/linux-0.11/kernel/m
ath'
make[1]: Leaving directory '/home/guojun/oslab/linux-0.11/kernel/ma
th'
make[1]: Entering directory '/home/guojun/oslab/linux-0.11/lib'
make[1]: Leaving directory '/home/guojun/oslab/linux-0.11/lib'
1+0 records in
1+0 records out
512 bytes copied, 0.000119198 s, 4.3 MB/s
0+1 records in
0+1 records out
311 bytes copied, 9.186e-05 s, 3.4 MB/s
309+1 records in
309+1 records out
158465 bytes (158 kB, 155 KiB) copied, 0.000849637 s, 187 MB/s
2+0 records in
2+0 records out
2 bytes copied, 3.4117e-05 s, 58.6 kB/s
```

图3-1-1 编译Linux-0.11源码

2. 挂载文件系统镜像

在 `oslab` 目录下，运行命令切换到挂载的文件系统 `hdc` 目录，即 `Linux-0.11` 使用的文件系统。

```
sudo ./mount-hdc
```

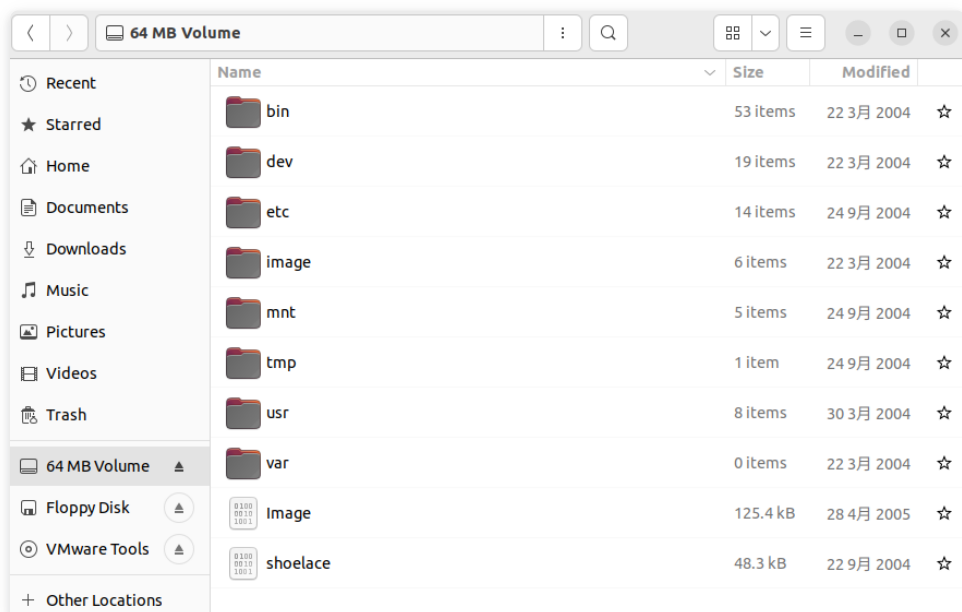


图3-1-2 linux-0.11文件系统

3. 复制 `process.c` 到 `~/oslab/hdc/usr/root/` 目录中

```
cp process.c ~/oslab/hdc/usr/root/
```

4. 在 `~/oslab` 目录下运行以下命令，在终端输入 `c` 后进入 `Bochs` 界面。

```
./dbg-bochsgui246
```

```
guojun@guojunos:~/oslab$ ./dbg-bochsgui246
=====
Bochs x86 Emulator 2.4.6
Build from CVS snapshot, on February 22, 2011
Compiled at Oct 31 2023, 13:21:11
=====
00000000000i[      ] LTDL_LIBRARY_PATH is set to '/usr/local/bochs2
46/lib/bochs/plugins'
00000000000i[      ] BXSHARE is set to '/usr/local/bochs246/share/b
ochs'
00000000000i[      ] reading configuration from ./bochs/linux-0.11-
gui246.bxrc
00000000000i[      ] lt_dlhandle is 0x58db4adc4fc0
00000000000i[PLGIN] loaded plugin libbx_x.so
00000000000i[      ] installing x module as the Bochs GUI
00000000000i[      ] using log file ./bochsout.txt
Next at t=0
(0) [0x00000000ffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b
; ea5be000f0
<bochs:1> c
```

图3-1-3 终端输入

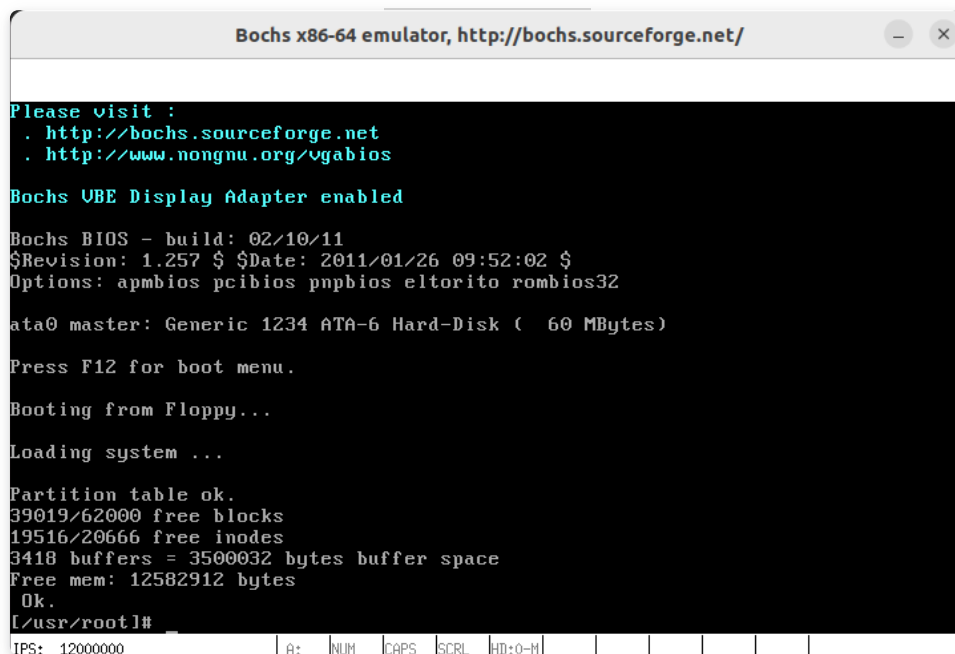


图3-1-4 进入Bochs界面

5. 在 /usr/root/ 下对 process.c 进行编译

```
gcc -o process process.c
```

6. 运行 process

```
./process
```

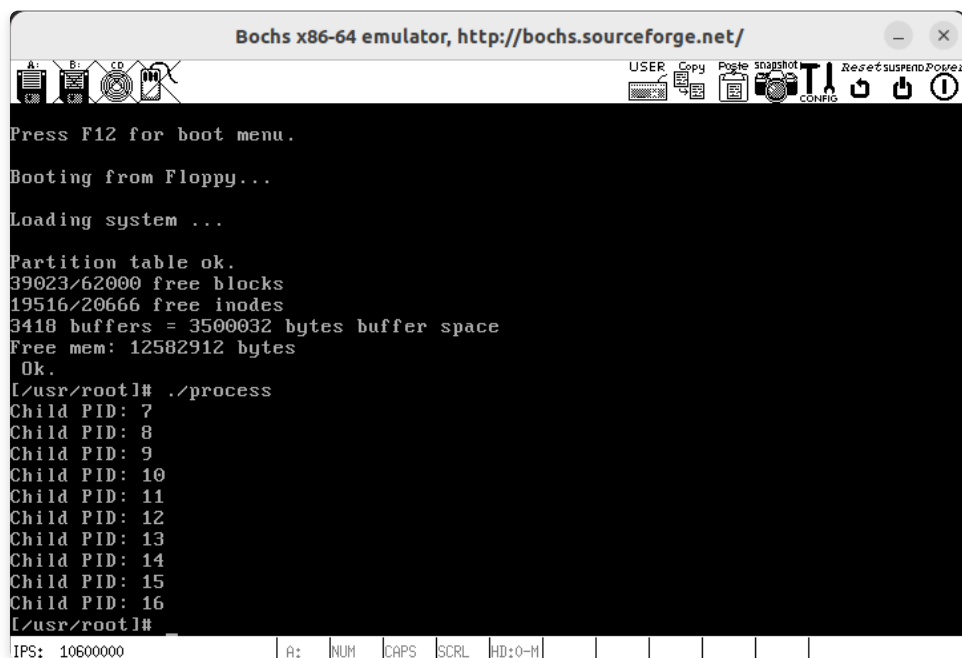


图3-1-5 运行样本程序

7. 卸载文件系统

在 `oslab` 目录下执行以下命令，进行文件系统的卸载。

```
sudo umount hdc
```

3.2. 在 Linux 0.11 上实现进程运行轨迹的跟踪

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。

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。

3.2.1. 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：

```

void init(void)
{
    // ...
    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。

```

// ...
move_to_user_mode();
/*
    添加到这里！不能再靠前的
    还要加上打开 log 文件的代码
*/
if (!fork()) {           /* we count on this going ok */
    init();
}
// ...

```

上述工作是在进程 1 中完成的（因为是进程 1 调用的 `init()`），为了能尽早访问 `Log` 文件，需要在进程 0 中就完成这部分工作。所以可以把这段代码从 `init()` 中移动到 `main()` 中：

```

// ...
move_to_user_mode();
/*****添加开始*****/
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
(void) open("/var/process.log",O_CREAT|O_TRUNC|O_WRONLY,0666); // 打开log文件
/*****添加结束*****/
if (!fork()) {           /* we count on this going ok */
    init();
}
// ...

```

```

176 void init(void)
177 {
178     int pid,i;
179     printf("%d buffers = %d bytes buffer space\n\r",NR_BUFFERS,
180           NR_BUFFERS*BLOCK_SIZE);
181     printf("Free mem: %d bytes\n\r",memory_end-main_memory_start);
182     if (!(pid=fork())) {
183         close(0);
184         if (open("/etc/rc",O_RDONLY,0))
185             _exit(1);
186         execve("/bin/sh",argv_rc,envp_rc);
187         _exit(2);
188     }
189 }

```

图3-2-1 从init()函数中移除代码


```

126 #ifdef RAMDISK
127     main_memory_start += rd_init(main_memory_start, RAMDISK*1024);
128 #endif
129     mem_init(main_memory_start, memory_end);
130     trap_init();
131     blk_dev_init();
132     chr_dev_init();
133     tty_init();
134     time_init();
135     sched_init();
136     buffer_init(buffer_memory_end);
137     hd_init();
138     floppy_init();
139     sti();
140     move_to_user_mode();
141     setup((void *) &drive_info);
142     (void) open("/dev/tty0", O_RDWR, 0);
143     (void) dup(0);
144     (void) dup(0);
145     (void) open("/var/process.log", O_CREAT|O_TRUNC|O_WRONLY, 0666);
146     if (!fork()) { /* we count on this going ok */
147         init();
148     }
149 /*
150 * NOTE!! For any other task 'pause()' would mean we have to get a
151 * signal to awaken, but task0 is the sole exception (see 'schedule()')
152 * as task 0 gets activated at every idle moment (when no other tasks
153 * can run). For task0 'pause()' just means we go check if some other
154 * task can run, and if not we return here.
155 */
156     for(;;) pause();
157 }

```

图3-2-2 从main()函数添加代码

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 1 中建立了。根据 `fork()` 的原理，进程 1 之后的进程会继承这些文件描述符，所以就不必再 `open()` 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 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。

3.2.2. 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 */

```

```

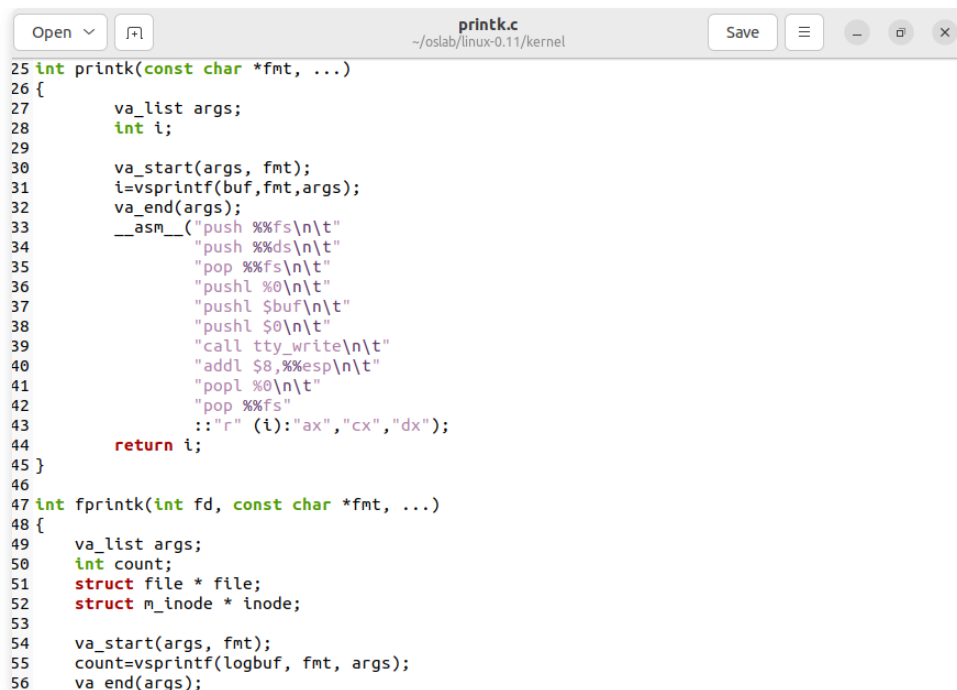
{
    __asm__(
        "push %%fs\n\t"
        "push %%ds\n\t"
        "pop %%fs\n\t"
        "pushl %0\n\t"
        "pushl $logbuf\n\t" /* 注意对于Windows环境来说，是_logbuf,下同 */
        "pushl %1\n\t"
        "call sys_write\n\t" /* 注意对于Windows环境来说，是_sys_write,下同 */
        "addl $8,%%esp\n\t"
        "popl %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;

        // 修改为如下：
        // task[1]->filp is not ready or f_inode->i_dev is not ready
        /* 从进程1的文件描述符表中得到文件句柄 */
        if (!(file=task[1]->filp[fd]) || !task[1]->filp[fd]->f_inode->i_dev) {
            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;
}

```

可以新建一个 `fprintk.c` 进行定义，并且修改 `Makefile`，将 `fprintk.c` 和其它 Linux 代码编译链接到一起。也可以鉴于该函数和 `printk` 的功能近相似，所以直接将以上源码放到 `kernel/printk.c` 中，不用修改 `Makefile`。



```
25 int printk(const char *fmt, ...)
26 {
27     va_list args;
28     int i;
29
30     va_start(args, fmt);
31     i=vsprintf(buf,fmt,args);
32     va_end(args);
33     __asm__ ("push %%fs\n\t"
34             "push %%ds\n\t"
35             "pop %%fs\n\t"
36             "pushl %0\n\t"
37             "pushl $buf\n\t"
38             "pushl $0\n\t"
39             "call tty_write\n\t"
40             "addl $8,%%esp\n\t"
41             "popl %0\n\t"
42             "pop %%fs"
43             ::"r" (i):"ax","cx","dx");
44     return i;
45 }
46
47 int fprintk(int fd, const char *fmt, ...)
48 {
49     va_list args;
50     int count;
51     struct file * file;
52     struct m_inode * inode;
53
54     va_start(args, fmt);
55     count=vsprintf(logbuf, fmt, args);
56     va_end(args);
```

图3-2-3 将fprintk添加到printk.c中

因为和 `printk` 的功能近似，建议将此函数放入到 `kernel/printk.c` 中。`fprintk()` 的使用方式与 C 标准库函数 `fprintf()` 相似，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
```

3.2.3. 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多少个 10ms。

3.2.4. 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。下面有两个例子。

第一个例子，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
}
```

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：

1. 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。
2. `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，`switch_to(next)` 也会被调用。这种情况下相当于当前进程的状态没变。
3. 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。

状态	对应函数
新建-N	<code>copy_process()</code>
就绪-J	<code>copy_process()</code> 、 <code>schedule()</code> 、 <code>sleep_on()</code> 、 <code>interruptible_sleep_on()</code> 、 <code>wake_up()</code>
运行-R	<code>schedule()</code>
等待-W	<code>sys_pause()</code> 、 <code>sleep_on()</code> 、 <code>interruptible_sleep_on()</code> 、 <code>sys_waitpid()</code>
退出-E	<code>do_exit()</code>

1. 修改 `fork.c` 中的 `copy_process` 函数

在 97 行添加如下代码，新建进程。

```
fprintk(3, "%d\tN\t%d\n", p->pid, jiffies);
```

```

95     p->cutime = p->cstime = 0;
96     p->start_time = jiffies;
97     fprintk(3, "%d\tN\t%d\n", p->pid, jiffies);
98     p->tss.back_link = 0;
99     p->tss.esp0 = PAGE_SIZE + (long) p;
100    p->tss.ss0 = 0x10;

```

图3-2-4 在97行添加代码

在 138 行添加如下代码，进程就绪。

```
fprintk(3, "%d\tJ\t%d\n", p->pid, jiffies);
```

```

133     if (current->executable)
134         current->executable->i_count++;
135     set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY,&(p->tss));
136     set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,&(p->ldt));
137     p->state = TASK_RUNNING; /* do this last, just in case */
138     fprintk(3, "%d\tJ\t%d\n", p->pid, jiffies);
139     return last_pid;
140 }

```

图3-2-5 在138行添加代码

2. 修改 `sched.c` 中的 `schedule` 函数

在 121 行添加如下代码，进程就绪。

```
fprintk(3, "%d\tJ\t%d\n", (*p)->pid, jiffies);
```

```
111     for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
112         if (*p) {
113             if ((*p)->alarm && (*p)->alarm < jiffies) {
114                 (*p)->signal |= (1<<(SIGALRM-1));
115                 (*p)->alarm = 0;
116             }
117             if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
118                 (*p)->state==TASK_INTERRUPTIBLE)
119             {
120                 (*p)->state=TASK_RUNNING;
121                 fprintk(3, "%d\tJ\t%d\n", (*p)->pid, jiffies);
122             }
123         }
124     }
```

图3-2-6 在121行添加代码

在 145~150 行添加如下代码，切换到相同的进程不输出，之后时间片到时程序=>就绪。

```
if(current->pid != task[next] ->pid)
{
    if(current->state == TASK_RUNNING)
        fprintk(3, "%d\tJ\t%d\n", current->pid, jiffies);
    fprintk(3, "%d\tR\t%d\n", task[next]->pid, jiffies);
}
```

```
145     if(current->pid != task[next] ->pid)
146     {
147         if(current->state == TASK_RUNNING)
148             fprintk(3, "%d\tJ\t%d\n", current->pid, jiffies);
149         fprintk(3, "%d\tR\t%d\n", task[next]->pid, jiffies);
150     }
151     switch_to(next);
152 }
```

图3-2-7 在145~150行添加代码

3. 修改 sched.c 中的 sys_pause 函数

在 157~158 行添加如下代码，运行=>可中断睡眠。

```
if(current->pid != 0)
    fprintk(3, "%d\tW\t%d\n", current->pid, jiffies);
```

```
154 int sys_pause(void)
155 {
156     current->state = TASK_INTERRUPTIBLE;
157     if(current->pid != 0)
158         fprintk(3, "%d\tW\t%d\n", current->pid, jiffies);
159     schedule();
160     return 0;
161 }
```

图3-2-8 在157~158行添加代码

4. 修改 sched.c 中的 sleep_on 函数

在 174 行添加如下代码，当前进程=>不可中断睡眠；在 179 原等待队列第一个进程=>唤醒（就绪）。

```
fprintk(3, "%d\tW\t%d\n", current->pid, jiffies);
fprintk(3, "%d\tJ\t%d\n", tmp->pid, jiffies);
```

```

163 void sleep_on(struct task_struct **p)
164 {
165     struct task_struct *tmp;
166
167     if (!p)
168         return;
169     if (current == &(init_task.task))
170         panic("task[0] trying to sleep");
171     tmp = *p;
172     *p = current;
173     current->state = TASK_UNINTERRUPTIBLE;
174     fprintfk(3, "%d\tW\t%d\n", current->pid, jiffies);
175     schedule();
176     if (tmp)
177     {
178         tmp->state=0;
179         fprintfk(3, "%d\tJ\t%d\n", tmp->pid, jiffies);
180     }
181 }

```

图3-2-9 在174、179行添加代码

5. 修改 sched.c 中的 interruptible_sleep_on 函数

在 194 行添加如下代码，唤醒队列中间进程，过程中使用 wait；在 198 行添加如下代码，当前进程=>可中断睡眠；在 205 行添加如下代码，原等待队列第一个进程=>唤醒（就绪）。

```

fprintfk(3, "%d\tW\t%d\n", current->pid, jiffies);
fprintfk(3, "%d\tJ\t%d\n", (*p)->pid, jiffies);
fprintfk(3, "%d\tJ\t%d\n", tmp->pid, jiffies);

```

```

183 void interruptible_sleep_on(struct task_struct **p)
184 {
185     struct task_struct *tmp;
186
187     if (!p)
188         return;
189     if (current == &(init_task.task))
190         panic("task[0] trying to sleep");
191     tmp=*p;
192     *p=current;
193 repeat: current->state = TASK_INTERRUPTIBLE;
194     fprintfk(3, "%d\tW\t%d\n", current->pid, jiffies);
195     schedule();
196     if (*p && *p != current) {
197         (**p).state=0;
198         fprintfk(3, "%d\tJ\t%d\n", (*p)->pid, jiffies);
199         goto repeat;
200     }
201     *p=NULL;
202     if (tmp)
203     {
204         tmp->state=0;
205         fprintfk(3, "%d\tJ\t%d\n", tmp->pid, jiffies);
206     }
207 }

```

图3-2-10 在194、198、205行添加代码

6. 修改 sched.c 中的 wake_up 函数

在 213 行添加如下代码，唤醒最后进入等待序列的进程。

```

fprintfk(3, "%d\tJ\t%d\n", (*p)->pid, jiffies);

```

```

209 void wake_up(struct task_struct **p)
210 {
211     if (p && *p) {
212         (**p).state=0;
213         fprintfk(3, "%d\tJ\t%d\n", (*p)->pid, jiffies);
214         *p=NULL;
215     }
216 }

```

图3-2-11 在213行添加代码

7. 修改 exit.c 中的 do_exit 函数

在 130 行添加如下代码，退出第一个进程。

```
fprintk(3, "%d\\tE\\t%d\\n", current->pid, jiffies);
```

```
123         if (current->leader && current->tty >= 0)
124             tty_table[current->tty].pgrp = 0;
125         if (last_task_used_math == current)
126             last_task_used_math = NULL;
127         if (current->leader)
128             kill_session();
129         current->state = TASK_ZOMBIE;
130         fprintk(3, "%d\\tE\\t%d\\n", current->pid, jiffies);
131         current->exit_code = code;
132         tell_father(current->father);
133         schedule();
134         return (-1);    /* just to suppress warnings */
135     }
```

图3-2-12 在130行添加代码

8. 修改 exit.c 中的 sys_waitpid 函数

在 188 行添加如下代码，当前进程=>等待。

```
fprintk(3, "%d\\tw\\t%d\\n", current->pid, jiffies);
```

```
184         if (flag) {
185             if (options & WNOHANG)
186                 return 0;
187             current->state=TASK_INTERRUPTIBLE;
188             fprintk(3, "%d\\tw\\t%d\\n", current->pid, jiffies);
189             schedule();
190             if (!(current->signal &= ~(1<<(SIGCHLD-1))))
191                 goto repeat;
192             else
193                 return -EINTR;
194         }
195         return -ECHILD;
196     }
```

图3-2-13 在188行添加代码

3.2.5. 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. 一定要仔细阅读关于 Ubuntu 和 Linux 0.11 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
.....

```

3.2.6. 运行程序测试

1. 编译 Linux-0.11 源码

切换到 `Linux-0.11` 目录下执行以下命令，生成 `Image` 镜像文件。

```

make clean
make Image

```

```

make[1]: Leaving directory '/home/guojun/oslab/linux-0.11/kernel/bl
k_drv'
make[1]: Entering directory '/home/guojun/oslab/linux-0.11/kernel/c
hr_drv'
sync
make[1]: Leaving directory '/home/guojun/oslab/linux-0.11/kernel/ch
r_drv'
make[1]: Entering directory '/home/guojun/oslab/linux-0.11/kernel/m
ath'
make[1]: Leaving directory '/home/guojun/oslab/linux-0.11/kernel/ma
th'
make[1]: Entering directory '/home/guojun/oslab/linux-0.11/lib'
make[1]: Leaving directory '/home/guojun/oslab/linux-0.11/lib'
1+0 records in
1+0 records out
512 bytes copied, 0.000119198 s, 4.3 MB/s
0+1 records in
0+1 records out
311 bytes copied, 9.186e-05 s, 3.4 MB/s
309+1 records in
309+1 records out
158465 bytes (158 kB, 155 KiB) copied, 0.000849637 s, 187 MB/s
2+0 records in
2+0 records out
2 bytes copied, 3.4117e-05 s, 58.6 kB/s

```

图3-2-14 编译Linux-0.11源码

2. 挂载文件系统镜像

在 `oslab` 目录下，运行命令切换到挂载的文件系统 `hdc` 目录，即 `Linux-0.11` 使用的文件系统。

```
sudo ./mount-hdc
```

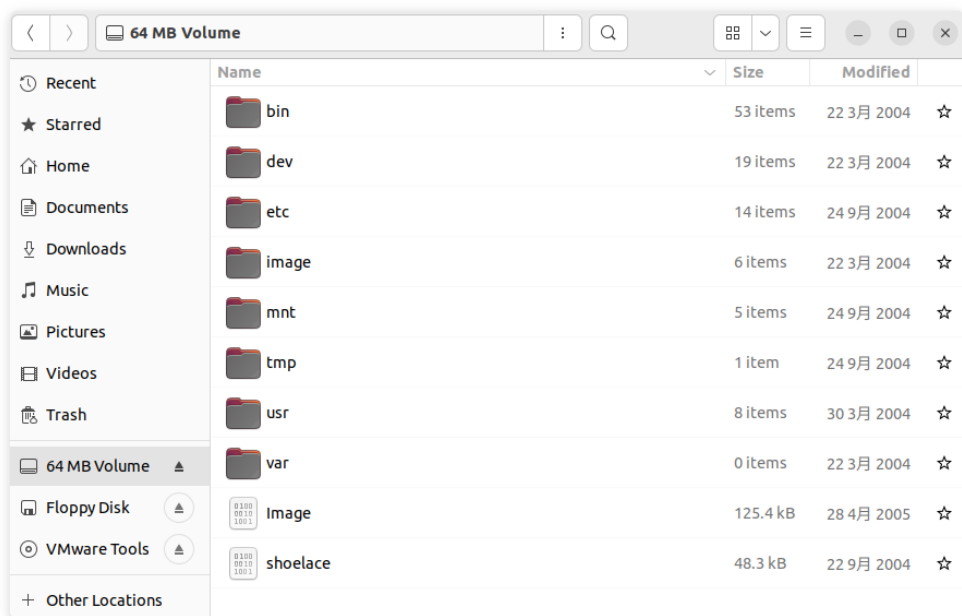


图3-2-15 linux-0.11文件系统

3. 在 `~/oslab` 目录下运行以下命令，在终端输入 `c` 后进入 `Bochs` 界面。

```
./dbg-bochsgui246
```

```

guojun@guojunos:~/oslab$ ./dbg-bochsgui246
=====
=====
Bochs x86 Emulator 2.4.6
Build from CVS snapshot, on February 22, 2011
Compiled at Oct 31 2023, 13:21:11
=====
=====
00000000000i[      ] LTDL_LIBRARY_PATH is set to '/usr/local/bochs2
46/lib/bochs/plugins'
00000000000i[      ] BXSHARE is set to '/usr/local/bochs246/share/b
ochs'
00000000000i[      ] reading configuration from ./bochs/linux-0.11-
gui246.bxrc
00000000000i[      ] lt_dlhandle is 0x58db4adc4fc0
00000000000i[PLGIN] loaded plugin libbx_x.so
00000000000i[      ] installing x module as the Bochs GUI
00000000000i[      ] using log file ./bochsout.txt
Next at t=0
(0) [0x00000000ffffffff] f000:fff0 (unk. ctxt): jmp far f000:e05b
; ea5be00f0
<bochs:1> c

```

图3-2-16 终端输入

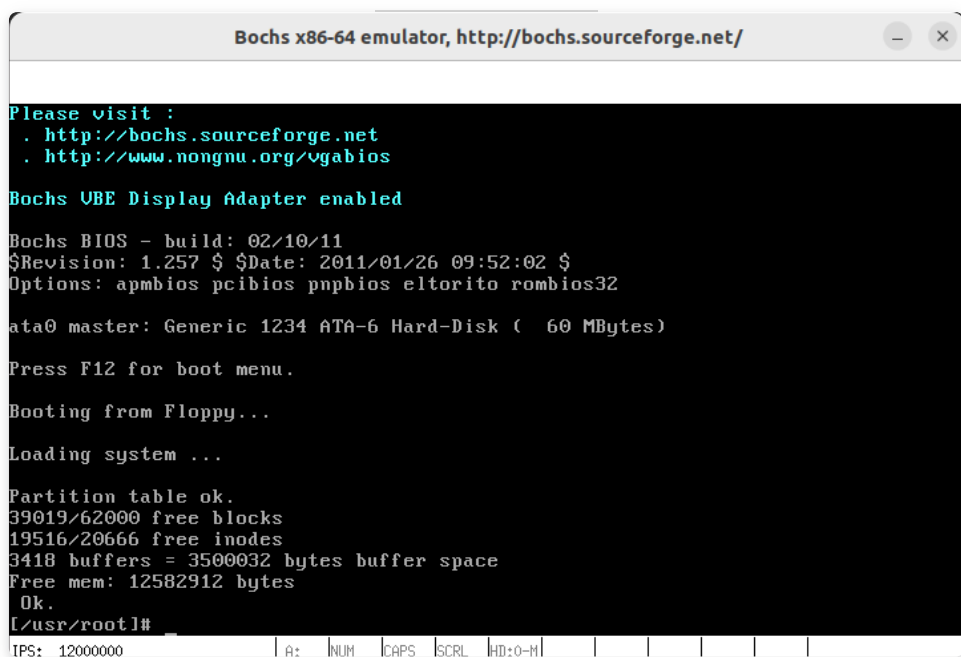


图3-2-17 进入Bochs界面

4. 编译 process.c，运行 process，之后输入 sync 进行刷新 cache，确保文件写入了磁盘。

```

gcc -o process process.c
./process
sync

```

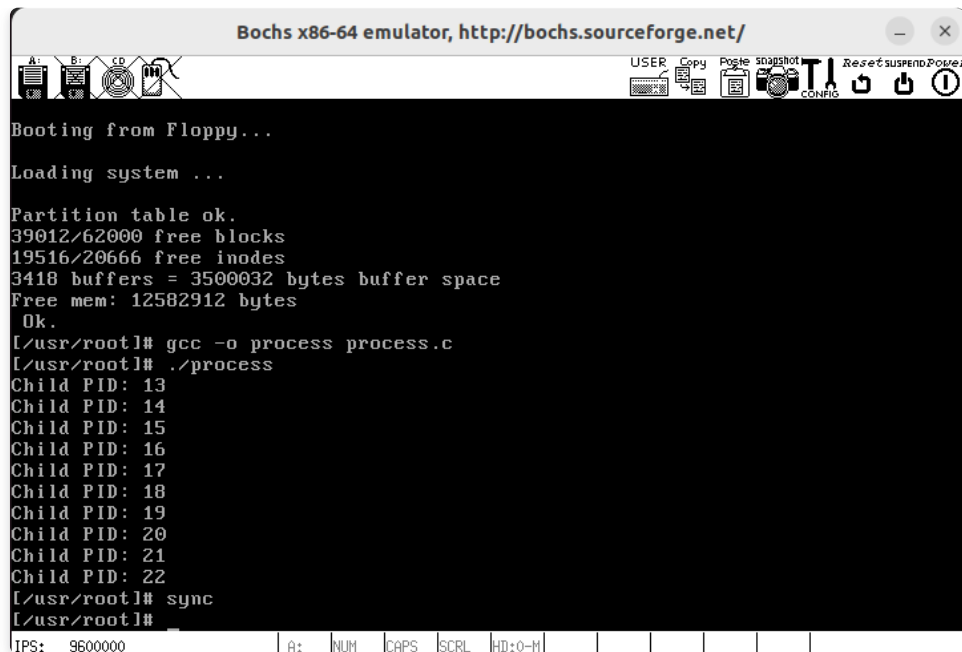


图3-2-18 运行样本程序

5. 在运行 process 程序之后，会在 /var 中发现生成一个 process.log 文件。现在将这个 process.log 文件拷贝到 Ubuntu 系统上，在 ~/oslab 目录下输入：

```
cp ./hdc/var/process.log ~/Documents/code/process-15.log
```

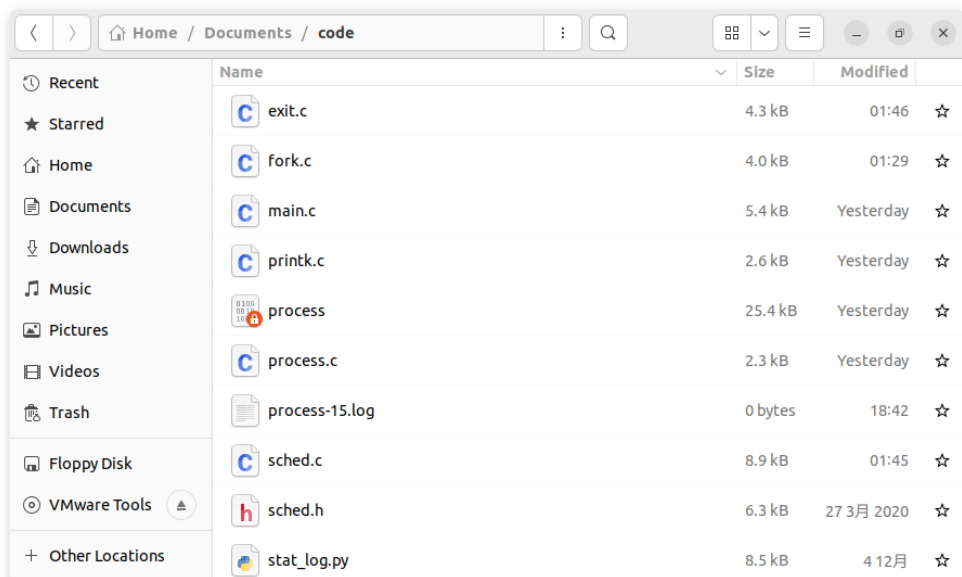


图3-2-19 拷贝process.log

process-15.log		
~/Documents/code		
1	1	N 45
2	1	J 46
3	0	J 46
4	1	R 46
5	2	N 46
6	2	J 47
7	1	W 47
8	2	R 47
9	3	N 64
10	3	J 65
11	2	E 70
12	1	J 70
13	1	R 70
14	4	N 70
15	4	J 71
16	1	W 71
17	3	R 71
18	3	W 77
19	4	R 78
20	5	N 114
21	5	J 114
22	4	W 115
23	5	R 115
24	4	J 117
25	5	E 118

图3-2-20 process.log内容

6. 卸载文件系统

在 `oslab` 目录下执行以下命令，进行文件系统的卸载。

```
sudo umount hdc
```

3.3. 分析并统计 log 文件

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。

3.3.1. 数据统计 sta_log.py 文件

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）。

```
#!/usr/bin/python
import sys
import copy

P_NULL = 0
P_NEW = 1
P_READY = 2
P_RUNNING = 4
P_WAITING = 8
P_EXIT = 16

S_STATE = 0
S_TIME = 1

HZ = 100

graph_title = r"""
-----< COOL GRAPHIC OF SCHEDULER >-----

[Symbol]    [Meaning]
```

```

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

```

```

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

```

```
usage = ""
```

```
Usage:
```

```
%s /path/to/process.log [PID1] [PID2] ... [-x PID1 [PID2] ... ] [-m] [-g]
```

```
Example:
```

```
# Include process 6, 7, 8 and 9 in statistics only. (Unit: tick)
```

```
%s /path/to/process.log 6 7 8 9
```

```
# Exclude process 0 and 1 from statistics. (Unit: tick)
```

```
%s /path/to/process.log -x 0 1
```

```
# Include process 6 and 7 only and print a COOL "graphic"! (Unit: millisecond)
```

```
%s /path/to/process.log 6 7 -m -g
```

```
# Include all processes and print a COOL "graphic"! (Unit: tick)
```

```
%s /path/to/process.log -g
```

```
"""
```

```
class MyError(Exception):
```

```
    pass
```

```
class DuplicateNew(MyError):
```

```
    def __init__(self, pid):
```

```
        args = "More than one 'N' for process %d." % pid
```

```
        MyError.__init__(self, args)
```

```
class UnknownState(MyError):
```

```
    def __init__(self, state):
```

```
        args = "Unknown state '%s' found." % state
```

```
        MyError.__init__(self, args)
```

```
class BadTime(MyError):
```

```
    def __init__(self, time):
```

```
        args = "The time '%d' is bad. It should >= previous line's time." % time
```

```
        MyError.__init__(self, args)
```

```
class TaskHasExited(MyError):
```

```
    def __init__(self, state):
```

```
        args = "The process has exited. Why it enter '%s' state again?" % state
```

```
        MyError.__init__(self, args)
```

```
class BadFormat(MyError):
```

```
    def __init__(self):
```

```

        args = "Bad log format"
        MyError.__init__(self, args)

class RepeatState(MyError):
    def __init__(self, pid):
        args = "Previous state of process %d is identical with this line." %
(pid)
        MyError.__init__(self, args)

class SameLine(MyError):
    def __init__(self):
        args = "It is a clone of previous line."
        MyError.__init__(self, args)

class NoNew(MyError):
    def __init__(self, pid, state):
        args = "The first state of process %d is '%s'. Why not 'N'?" % (pid,
state)
        MyError.__init__(self, args)

class statistics:
    def __init__(self, pool, include, exclude):
        if include:
            self.pool = process_pool()
            for process in pool:
                if process.getpid() in include:
                    self.pool.add(process)
        else:
            self.pool = copy.copy(pool)

        if exclude:
            for pid in exclude:
                if self.pool.get_process(pid):
                    self.pool.remove(pid)

    def list_pid(self):
        l = []
        for process in self.pool:
            l.append(process.getpid())
        return l

    def average_turnaround(self):
        if len(self.pool) == 0:
            return 0
        sum = 0
        for process in self.pool:
            sum += process.turnaround_time()
        return float(sum) / len(self.pool)

    def average_waiting(self):
        if len(self.pool) == 0:
            return 0
        sum = 0
        for process in self.pool:
            sum += process.waiting_time()
        return float(sum) / len(self.pool)

```



```

def begin_time(self):
    begin = 0xEFFFFFFF
    for p in self.pool:
        if p.begin_time() < begin:
            begin = p.begin_time()
    return begin

def end_time(self):
    end = 0
    for p in self.pool:
        if p.end_time() > end:
            end = p.end_time()
    return end

def throughput(self):
    return len(self.pool) * HZ / float(self.end_time() - self.begin_time())

def print_graphic(self):
    begin = self.begin_time()
    end = self.end_time()

    print(graph_title)

    for i in range(begin, end+1):
        line = "%5d " % i
        for p in self.pool:
            state = p.get_state(i)
            if state & P_NEW:
                line += "-"
            elif state == P_READY or state == P_READY | P_WAITING:
                line += "|"
            elif state == P_RUNNING:
                line += "#"
            elif state == P_WAITING:
                line += ":"
            elif state & P_EXIT:
                line += "-"
            elif state == P_NULL:
                line += " "
            elif state & P_RUNNING:
                line += "+"
            else:
                assert False
            if p.get_state(i-1) != state and state != P_NULL:
                line += "%-3d" % p.getpid()
            else:
                line += "   "
        print(line)

class process_pool:
    def __init__(self):
        self.list = []

    def get_process(self, pid):
        for process in self.list:

```

```

        if process.getpid() == pid:
            return process
    return None

def remove(self, pid):
    for process in self.list:
        if process.getpid() == pid:
            self.list.remove(process)

def new(self, pid, time):
    p = self.get_process(pid)
    if p:
        if pid != 0:
            raise DuplicateNew(pid)
        else:
            p.states=[(P_NEW, time)]
    else:
        p = process(pid, time)
        self.list.append(p)
    return p

def add(self, p):
    self.list.append(p)

def __len__(self):
    return len(self.list)

def __iter__(self):
    return iter(self.list)

class process:
    def __init__(self, pid, time):
        self.pid = pid
        self.states = [(P_NEW, time)]

    def getpid(self):
        return self.pid

    def change_state(self, state, time):
        last_state, last_time = self.states[-1]
        if state == P_NEW:
            raise DuplicateNew(pid)
        if time < last_time:
            raise BadTime(time)
        if last_state == P_EXIT:
            raise TaskHasExited(state)
        if last_state == state and self.pid != 0: # task 0 can have duplicate
state
            raise RepeatState(self.pid)

        self.states.append((state, time))

    def get_state(self, time):
        rval = P_NULL
        combo = P_NULL
        if self.begin_time() <= time <= self.end_time():

```

```

        for state, s_time in self.states:
            if s_time < time:
                rval = state
            elif s_time == time:
                combo |= state
            else:
                break
        if combo:
            rval = combo
    return rval

def turnaround_time(self):
    return self.states[-1][S_TIME] - self.states[0][S_TIME]

def waiting_time(self):
    return self.state_last_time(P_READY)

def cpu_time(self):
    return self.state_last_time(P_RUNNING)

def io_time(self):
    return self.state_last_time(P_WAITING)

def state_last_time(self, state):
    time = 0
    state_begin = 0
    for s,t in self.states:
        if s == state:
            state_begin = t
        elif state_begin != 0:
            assert state_begin <= t
            time += t - state_begin
            state_begin = 0
    return time

def begin_time(self):
    return self.states[0][S_TIME]

def end_time(self):
    return self.states[-1][S_TIME]

# Enter point
if len(sys.argv) < 2:
    print(usage.replace("%s", sys.argv[0]))
    sys.exit(0)

# parse arguments
include = []
exclude = []
unit_ms = False
graphic = False
ex_mark = False

try:
    for arg in sys.argv[2:]:

```

```

    if arg == '-m':
        unit_ms = True
        continue
    if arg == '-g':
        graphic = True
        continue
    if not ex_mark:
        if arg == '-x':
            ex_mark = True
        else:
            include.append(int(arg))
    else:
        exclude.append(int(arg))
except ValueError:
    print("Bad argument '%s'" % arg)
    sys.exit(-1)

# parse log file and construct processes
processes = process_pool()

f = open(sys.argv[1], "r")

# Patch process 0's New & Run state
processes.new(0, 40).change_state(P_RUNNING, 40)

try:
    prev_time = 0
    prev_line = ""
    for lineno, line in enumerate(f):

        if line == prev_line:
            raise SameLine
        prev_line = line

        fields = line.split("\t")
        if len(fields) != 3:
            raise BadFormat

        pid = int(fields[0])
        s = fields[1].upper()

        time = int(fields[2])
        if time < prev_time:
            raise BadTime(time)
        prev_time = time

        p = processes.get_process(pid)

        state = P_NULL
        if s == 'N':
            processes.new(pid, time)
        elif s == 'J':
            state = P_READY
        elif s == 'R':
            state = P_RUNNING
        elif s == 'W':

```

```

        state = P_WAITING
    elif s == 'E':
        state = P_EXIT
    else:
        raise UnknownState(s)
    if state != P_NULL:
        if not p:
            raise NoNew(pid, s)
        p.change_state(state, time)
except MyError as err:
    print("Error at line %d: %s" % (lineno+1, err))
    sys.exit(0)

# Stats
stats = statistics(processes, include, exclude)
att = stats.average_turnaround()
awt = stats.average_waiting()
if unit_ms:
    unit = "ms"
    att *= 1000/HZ
    awt *= 1000/HZ
else:
    unit = "tick"
print("(Unit: %s)" % unit)
print("Process   Turnaround   Waiting   CPU Burst   I/O Burst")
for pid in stats.list_pid():
    p = processes.get_process(pid)
    tt = p.turnaround_time()
    wt = p.waiting_time()
    cpu = p.cpu_time()
    io = p.io_time()

    if unit_ms:
        print("%7d   %10d   %7d   %9d   %9d" % (pid, tt*1000/HZ, wt*1000/HZ,
cpu*1000/HZ, io*1000/HZ))
    else:
        print("%7d   %10d   %7d   %9d   %9d" % (pid, tt, wt, cpu, io))
print("Average:   %10.2f   %7.2f" % (att, awt))
print("Throughout: %.2f/s" % (stats.throughput()))

if graphic:
    stats.print_graphic()

```

3.3.2. 运行程序测试

1. `python` 是一种跨平台的脚本语言。在 `ubuntu` 下这样安装：

```
sudo apt-get install python2
```

2. 然后只要给 `stat_log.py` 加上执行权限，就可以直接运行它。

```
chmod +x stat_log.py
```

3. 运行统计程序，分析 Log 文件。

stat_log.py 相关命令及参数说明如下所示：

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 `python2 stat_log.py process.log 13 14 15 16 17 18 19 20 21 22 -g >`

`result.txt`（只统计 PID 为 13、14、15、16、17、18、19、20、21 和 22 的进程），并将输出的结果保存到 `result.txt` 文件中。

```
python2 stat_log.py process-15.log 13 14 15 16 17 18 19 20 21 22 -g > result-15.txt
```

```
(Unit: tick)
Process  Turnaround  Waiting  CPU Burst  I/O Burst
13      1206        146      0         1060
14      1907        850     100        957
15      2101       1586     200        315
16      2115       1859     255         0
17      2099       1843     255         0
18      2188       1917     270         0
19      2172       1901     270         0
20      2156       1885     270         0
21      2140       1869     270         0
22      2124       1853     270         0
Average: 2020.80  1570.90
Throughout: 0.46/s

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

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

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

3896 -13
3897 |13 -14
3898 | |14 -15
3899 | | |15 -16
3900 | | | |16 -17
3901 | | | | |17 -18
3902 | | | | |18 -19
3903 | | | | |19 -20
3904 | | | | |20 -21
3905 | | | | |21 -22
3906 | | | | |22
3907 | | | | |#22
3908 | | | | |#
```

图3-3-1 统计结果

3.4. 修改时间片

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。

3.4.1. `schedule()` 函数分析

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。

3.4.2. 进程 `counter` 的初始化

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;
```

3.4.3. 当进程的时间片用完时，被重新赋成何值

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。

3.4.4. 运行程序测试

1. 修改时间片

在 `sched.h` 中的修改宏 `INIT_TASK` 中的第三个值。此处修改 `priority` 为 1、10 和 50。

```
115 #define INIT_TASK \
116 /* state etc */ { 0,15,15, \
117 /* signals */ 0,{},{},0, \
118 /* ec,brk... */ 0,0,0,0,0,0, \
119 /* pid etc.. */ 0,-1,0,0,0, \
120 /* uid etc */ 0,0,0,0,0,0, \
121 /* alarm */ 0,0,0,0,0,0, \
122 /* math */ 0, \
123 /* fs info */ -1,0022,NULL,NULL,NULL,0, \
124 /* filp */ {NULL,}, \
125 { \
126 {0,0}, \
127 /* ldt */ {0x9f,0xc0fa00}, \
128 {0x9f,0xc0f200}, \
129 }, \
130 /*tss*/ {0,PAGE_SIZE+(long)&init_task,0x10,0,0,0,0,(long)&pg_dir,\
131 0,0,0,0,0,0,0, \
132 0,0,0x17,0x17,0x17,0x17,0x17,0x17, \
133 _LDT(0),0x80000000, \
134 {} \
135 }, \
136 }
```

图3-4-1 统计结果

2. 重新编译并统计

此环节和第二小节的运行程序的流程是相同的，在修改时间片之后重新编译，打开 `Bochs` 重新编译运行样本程序 `process.c`，得到新的 `process.log`，接着运行统计脚本 `stat_log.py` 分析新的 Log 文件。

3. 运行结果

- 时间片为 1


```

(Unit: tick)
Process  Turnaround  Waiting  CPU Burst  I/O Burst
13      1076      64      1      1010
14      1834      824      100     909
15      1909      1506     200     202
16      1986      1744     241      0
17      1980      1739     240      0
18      1973      1733     239      0
19      1965      1726     238      0
20      1956      1718     237      0
21      1946      1709     236      0
22      1935      1699     235      0
Average: 1856.00 1446.20
Throughout: 0.50/s

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

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

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

4747 -13
4748 +13
4749 :13 -14
4750 :  +14
4751 :  #14
4752 : |14 -15
4753 : |  +15
4754 : |  #15
4755 : #14 |15
4756 : |14 | -16

```

图3-4-2 时间片为1的统计结果

- 时间片为 10

```

(Unit: tick)
Process  Turnaround  Waiting  CPU Burst  I/O Burst
13      1171      77      0      1093
14      1941      869      100     971
15      2091      1568     200     322
16      2130      1869     260      0
17      2119      1858     260      0
18      2108      1847     260      0
19      2097      1836     260      0
20      2086      1825     260      0
21      1935      1684     250      0
22      1994      1733     260      0
Average: 1967.20 1516.60
Throughout: 0.47/s

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

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

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

3938 -13
3939 |13 -14
3940 |  |14 -15
3941 |  |  |15 -16
3942 |  |  |  |16 -17
3943 |  |  |  |  |17 -18
3944 |  |  |  |  |  |18 -19
3945 |  |  |  |  |  |  |19 -20
3946 |  |  |  |  |  |  |  | +20
3947 |  |  |  |  |  |  |  | #20
3948 |  |  |  |  |  |  |  | #

```

图3-4-3 时间片为10的统计结果

- 时间片为 15

```

(Unit: tick)
Process  Turnaround  Waiting  CPU Burst  I/O Burst
13      1206        146      0          1060
14      1907        850      100         957
15      2101       1586      200         315
16      2115       1859      255          0
17      2099       1843      255          0
18      2188       1917      270          0
19      2172       1901      270          0
20      2156       1885      270          0
21      2140       1869      270          0
22      2124       1853      270          0
Average: 2020.80  1570.90
Throughout: 0.46/s

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

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

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

3896 -13
3897 |13 -14
3898 |  |14 -15
3899 |  |  |15 -16
3900 |  |  |  |16 -17
3901 |  |  |  |  |17 -18
3902 |  |  |  |  |  |18 -19
3903 |  |  |  |  |  |  |19 -20
3904 |  |  |  |  |  |  |  |20 -21
3905 |  |  |  |  |  |  |  |  |21 -22
3906 |  |  |  |  |  |  |  |  |  |22
3907 |  |  |  |  |  |  |  |  |  | #22
3908 |  |  |  |  |  |  |  |  |  | #

```

图3-4-4 时间片为15的统计结果

- 时间片为 50

```

(Unit: tick)
Process  Turnaround  Waiting  CPU Burst  I/O Burst
13      1916        461      0          1454
14      2622       1217      100         1304
15      3174       1867      200         1107
16      3535       2426      300          809
17      3534       2873      400          261
18      3533       3032      500          0
19      3482       2981      500          0
20      3431       2930      500          0
21      3380       2879      500          0
22      3329       2828      500          0
Average: 3193.60  2349.40
Throughout: 0.28/s

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

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

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

5584 -13
5585 |13 -14
5586 |  |14
5587 |  |  |15
5588 |  |  |  |15 -16
5589 |  |  |  |  |16 -17
5590 |  |  |  |  |  |17 -18
5591 |  |  |  |  |  |  |18 -19
5592 |  |  |  |  |  |  |  |19 -20
5593 |  |  |  |  |  |  |  |  |20 -21
5594 |  |  |  |  |  |  |  |  |  |21 -22
5595 |  |  |  |  |  |  |  |  |  |22
5596 |  |  |  |  |  |  |  |  |  | #22
5597 |  |  |  |  |  |  |  |  |  | #

```

图3-4-5 时间片为50的统计结果

4. 结果分析

在一定的范围内，平均等待时间，平均完成时间的变化随着时间片的增大而减小。这是因为在时间片小的情况下，CPU 将时间耗费在调度切换上，所以平均等待时间增加。

对时间片为 1 的情况仔细分析，可以发现：父进程没有一次性创建所有子进程，另外的是时间片中断后创建的。后续的进程新建的时间晚很多，这就拉低了平均数。

```
4747 -13
4748 +13
4749 :13 -14
4750 : +14
4751 : #14
4752 : |14 -15
4753 : | +15
4754 : | #15
4755 : #14 |15
4756 : |14 | -16
4757 : | | +16
4758 : | | #16
4759 : | #15 |16
4760 : #14 |15 |
4761 : |14 | | -17
4762 : | | | +17
4763 : | | | #17
4764 : | | #16 |17
4765 : | #15 |16 |
4766 : #14 |15 |
4767 : |14 | | -18
4768 : | | | +18
4769 : | | | #18
4770 : | | | #17 |18
4771 : | | #16 |17 |
4772 : | #15 |16 |
4773 : #14 |15 |
4774 : |14 | | -19
4775 : | | | +19
4776 : | | | #19
4777 : | | | #18 |19
4778 : | | #17 |18 |
4779 : | #16 |17 |
4780 : #15 |16 |
4781 : #14 |15 |
4782 : |14 | | -20
```

图3-4-6 时间片为1分析

对时间片为 10 的情况仔细分析，可以发现：父进程没有一次性创建 21 子进程，另外的是时间片中断后创建的。后续的进程新建的时间晚很多，这就拉低了平均数。

```
3938 -13
3939 |13 -14
3940 | |14 -15
3941 | | |15 -16
3942 | | | |16 -17
3943 | | | | |17 -18
3944 | | | | |18 -19
3945 | | | | |19 -20
3946 | | | | | +20
3947 | | | | | #20
3948 | | | | | #
3949 | | | | | #
3950 | | | | | #
3951 | | | | | #
3952 | | | | | #
3953 | | | | | #
3954 | | | | | #
3955 | | | | | #
3956 | | | | | #19 |20
```

图3-4-7 时间片为10分析

```

4086 : |14 | | | | | | | -21
4087 : | | | | | | | | |21 -22
4088 : | | | | | | | | | |22
4089 : | | | | | | | | | |#22
4090 : | | | | | | | | | |#
4091 : | | | | | | | | | |#
4092 : | | | | | | | | | |#
4093 : | | | | | | | | | |#
4094 : | | | | | | | | | |#
4095 : | | | | | | | | | |#
4096 : | | | | | | | | | |#
4097 : | | | | | | | | | |#
4098 : | | | | | | | | | |#
4099 : | | | | | | | | | |#21 |22
4100 : | | | | | | | | | |#
4101 : | | | | | | | | | |#
4102 : | | | | | | | | | |#

```

图3-4-8 时间片为10分析

对时间片为 15 的情况仔细分析，可以发现它是一次性fork出的。

```

3896 -13
3897 |13 -14
3898 | |14 -15
3899 | | |15 -16
3900 | | | |16 -17
3901 | | | | |17 -18
3902 | | | | | |18 -19
3903 | | | | | | |19 -20
3904 | | | | | | | |20 -21
3905 | | | | | | | | |21 -22
3906 | | | | | | | | | |22
3907 | | | | | | | | | |#22
3908 | | | | | | | | | |#

```

图3-4-9 时间片为15分析

对时间片为 50 的情况仔细分析，可以发现它也是一次性fork出的。

```

5584 -13
5585 |13 -14
5586 | |14
5587 | | | -15
5588 | | | |15 -16
5589 | | | | |16 -17
5590 | | | | | |17 -18
5591 | | | | | | |18 -19
5592 | | | | | | | |19 -20
5593 | | | | | | | | |20 -21
5594 | | | | | | | | | |21 -22
5595 | | | | | | | | | | |22
5596 | | | | | | | | | | |#22
5597 | | | | | | | | | | |#

```

图3-4-10 时间片为50分析

相较于 50 的时间片，15 明显更优；若是考虑上时间片为 10 和 1 得拉到平均值效果，15 的时间片其实也是更优的选择。

四、实验报告

- 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？
1. 单进程编程进程从上到下顺序进行；多进程编程可以通过并发执行，即多个进程之间交替执行，如某一个进程正在 I/O 输入输出而不占用 CPU 时，可以让 CPU 去执行另外一个进程。
 2. 单进程编程 CPU 利用率低，因为在等待 I/O 时，CPU 是空闲的；多进程编程的 CPU 利用率高，因为当某一进程等待 I/O 时，CPU 会去执行另一个进程。
 3. 单进程的数据是同步的，因为单进程只有一个进程，改变数据会影响该进程；多进程的数据是异步的，因为子进程数据是父进程数据在内存另一个位置的拷贝，改变其中一个进程的数据不会影响到另一个进程。

- 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？
1. 如何修改时间片：对于时间片 `counter`，由于时间片的初始化操作为 `p->counter = p->priority`，只与优先级 `priority` 有关，所以只需要修改宏 `INIT_TASK` 中的 `priority`。
 2. 一定范围内，平均等待时间，平均完成时间的变化随着时间片的增大而减小。这是因为在时间片小的情况下，`CPU` 将时间耗费在调度切换上，所以平均等待时间增加。超过这个的范围，这些参数将不再有明显的变化，因为 `RR` 轮转调度就变成了 `FCFS` 先来先服务。随着时间片的修改，吞吐量始终没有明显的变化，因为在单位时间内，系统所能完成的进程数量不会变化。

五、总结

通过本次实验，在 `Linux-0.11` 下通过进行进程运行轨迹的跟踪与统计，我掌握 `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。在实验过程中，我还不断学习操作系统 `linux0.11` 源码并了解其中的作用和含义。通过本次实验还锻炼了我解决问题的能力。