进程运行轨迹的跟踪与统计-实验报告

2021113117 王宇轩

1. 修改main.c，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()` 。在 `fork` 进程1前，我们使其加载文件系统，并进行对应文件描述符的关联，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。关于文件的属性：O_CREAT——如果文件不存在则创建；O_WRONLY——只写模式；O_TRUNC如果文件存在，并且以只写/读写方式打开，则清空文件全部内容。0666表示所有用户都拥有读取和写入的权限，没有执行权限。

```c
	buffer_init(buffer_memory_end);
	hd_init();
	floppy_init();
	sti();
	move_to_user_mode();

	/* load file system */
	setup((void *) &drive_info);
	(void) open("/dev/tty0",O_RDWR,0);
	(void) dup(0);
	(void) dup(0);
	/* the log file, with the desired attributes */
	(void) open("/var/process.log", O_CREAT|O_TRUNC|O_WRONLY, 0666);

	if (!fork()) {		/* we count on this going ok */
		init();
	}
/*
 *   NOTE!!   For any other task 'pause()' would mean we have to get a
 * signal to awaken, but task0 is the sole exception (see 'schedule()')
 * as task 0 gets activated at every idle moment (when no other tasks
 * can run). For task0 'pause()' just means we go check if some other
 * task can run, and if not we return here.
 */
```

2. 写log文件

在内核状态下，write()功能失效。直接将实验提示中给出的 `fprintk()` 复制进 `kernel/printk.c` ，同时补充对应的 `# include` 宏即可。

3. 在各进程状态切换处添加文件写入代码

进程状态的切换主要涉及到 `kernel` 目录下的三个文件： `fork.c` 、 `sched.c` 和 `exit.c` ，接下来对其进行逐一的分析。

3.1 fork.c

这里涉及到了两种进程状态改变：**新建（N）与新建→就绪（J）**。真正实现进程创建的函数是 `copy_process()`，其中大致流程为：获得一个task_struct结构体空间，之后对其结构体内的各个属性进行赋值，其中语句 `p->start_time = jiffies;` 设置start_time为jiffies，而之后的这段代码是在用寄存器值构建进程的TSS；后面的 `p->state = TASK_RUNNING;` 这一句则是将进程设置为就绪态，在这一句前后进行log的写入：

```
                current->executable->i_count++;
        set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY,&(p->tss));
        set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,&(p->ldt));

        fprintk(3, "%ld\t%c\t%ld\n", p->pid, 'N', jiffies);     /* log:N */
        p->state = TASK_RUNNING;        /* do this last, just in case */
        fprintk(3, "%ld\t%c\t%ld\n", p->pid, 'J', jiffies);     /* log:J */

        return last_pid;
}

```

3.2 sched.c

sched.c中涉及多种进程状态切换。

首先，`schedule()` 函数涉及到了就绪与运行态之间的相互转换，因此有以下两处修改：

```
                                (*p)->signal |= (1<<(SIGALRM-1));
                                (*p)->alarm = 0;
                        }
                if (((*p)->signal & ~(_BLOCKABLE & (*p)->blocked)) &&
                (*p)->state==TASK_INTERRUPTIBLE){
                        (*p)->state=TASK_RUNNING;

                        fprintk(3, "%ld\t%c\t%ld\n", (*p)->pid, 'J', jiffies);  /* log:J */
                }
        }

/* this is the scheduler proper: */

        while (1) {
                c = -1;
                next = 0;
```

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

        /* log:We have to see if the next process is just the current one */
        if (current->pid != task[next]->pid) {
                if (current->state == TASK_RUNNING) {   /* log:current one changed */
                        fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'J', jiffies);       /* log:J */
                }
                /* next one */
                fprintk(3, "%ld\t%c\t%ld\n", task[next]->pid, 'R', jiffies);    /* log:R */
        }

        switch_to(next);
}

int sys_pause(void)
{
```

运行到睡眠依靠的是 `sleep_on()` 和 `interruptible_sleep_on()`，并且这两个函数也涉及到了队列中进程的唤醒（睡眠→就绪），所作修改如下，其中 `interruptible_sleep_on()` 只有在被唤醒进程

与阻塞队列队首进程恰好相同时，才可以将该进程变为就绪态。

```c
void sleep_on(struct task_struct **p)
{
	struct task_struct *tmp;

	if (!p)
		return;
	if (current == &(init_task.task))
		panic("task[0] trying to sleep");
	tmp = *p;
	*p = current;
	current->state = TASK_UNINTERRUPTIBLE;

	fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies);	/* log:W */

	schedule();
	if (tmp){
		tmp->state=0;
		fprintk(3, "%ld\t%c\t%ld\n", tmp->pid, 'J', jiffies);	/* log:J, the first one in the waiting queue wakes up */
	}
}
```

```c
void interruptible_sleep_on(struct task_struct **p)
{
	struct task_struct *tmp;

	if (!p)
		return;
	if (current == &(init_task.task))
		panic("task[0] trying to sleep");
	tmp=*p;
	*p=current;
repeat:	current->state = TASK_INTERRUPTIBLE;

	fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies);	/* log:W */

	schedule();
	if (*p && *p != current) {
		(**p).state=0;

		fprintk(3, "%ld\t%c\t%ld\n", (*p)->pid, 'J', jiffies);	/* log:J, the first one in the waiting queue wakes up */

		goto repeat;
	}
	*p=NULL;
	if (tmp){
		tmp->state=0;
		fprintk(3, "%ld\t%c\t%ld\n", tmp->pid, 'J', jiffies);	/* log:J, the first one in the waiting queue wakes up */
	}
}
```

`wake_up()` 函数将进程唤醒：

```c
void wake_up(struct task_struct **p)
{
	if (p && *p) {
		(**p).state=0;
		fprintk(3, "%ld\t%c\t%ld\n", (*p)->pid, 'J', jiffies);	/* log:J */
		*p=NULL;
	}
}
```

此外，进程主动睡觉也会涉及系统调用 `sys_pause()` 和 `sys_waitpid()` （位于exit.c）。有一个特殊情况：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，这种情况不需要记录，因此对 `sys_pause()` 的修改如下：

```c

int sys_pause(void)
{
	current->state = TASK_INTERRUPTIBLE;

	if (current->pid != 0) {	/* log:don't log this */
		fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies);	/* log:W */
	}

	schedule();
	return 0;
}

```

3.3 exit.c

exit.c主要涉及进程的退出，及父进程等待子进程退出。首先接上文，对于`sys_waitpid()`：

```c
		}
	}
	if (flag) {
		if (options & WNOHANG)
			return 0;
		current->state=TASK_INTERRUPTIBLE;
		fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies);	/* log:W */
		schedule();
		if (!(current->signal &= ~(1<<(SIGCHLD-1))))
			goto repeat;
		else
			return -EINTR;
	}
```

而退出即涉及到函数`do_exit()`，修改如下。

```c
	if (last_task_used_math == current)
		last_task_used_math = NULL;
	if (current->leader)
		kill_session();
	current->state = TASK_ZOMBIE;
	current->exit_code = code;
	fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'E', jiffies);	/* log:E */
	tell_father(current->father);
	schedule();
	return (-1);	/* just to suppress warnings */
}

```

至此，对Linux-0.11代码的修改已完成。

4. 编写process.c

根据模板编写如下。

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

#define HZ  100
```

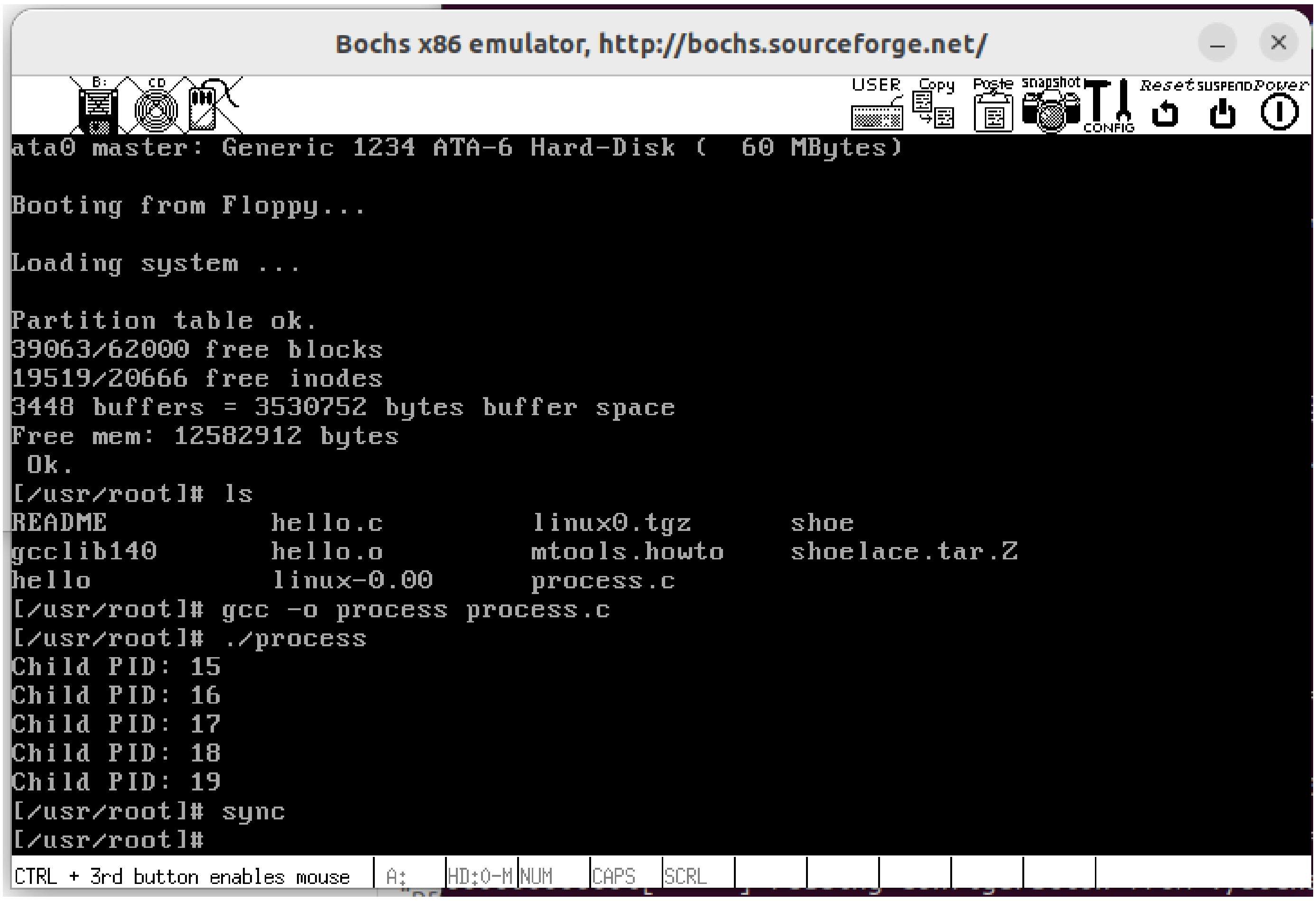
```c
void cpuio_bound(int last, int cpu_time, int io_time);

int main(int argc, char * argv[])
{
    int num_processes = 5;
    pid_t processes[num_processes]; /* create 5 processes */
    int i;
    for (i = 0; i < num_processes; i++) {
        processes[i] = fork();

        if (processes[i] < 0) {
        printf("Failed to create child process %d!\n", i + 1);
        exit(-1);
        } else if (processes[i] == 0) {
            cpuio_bound(10, 2 * i, (10 - 2 * i));
            exit(0);
        } else {
            ;   /* father */
        }
    }

    for (i = 0; i < num_processes; i++) {
        printf("Child PID: %d\n", processes[i]);
    }
    wait(&i);   /* father awaits */

    return 0;
}

/*
 * 此函数按照参数占用CPU和I/O时间
 * last: 函数实际占用CPU和I/O的总时间，不含在就绪队列中的时间，>=0是必须的
 * cpu_time: 一次连续占用CPU的时间，>=0是必须的
 * io_time: 一次I/O消耗的时间，>=0是必须的
 * 如果last > cpu_time + io_time，则往复多次占用CPU和I/O
 * 所有时间的单位为秒
 */
void cpuio_bound(int last, int cpu_time, int io_time)
{
    struct tms start_time, current_time;
    clock_t utime, stime;
    int sleep_time;

    while (last > 0)
    {
        /* CPU Burst */
        times(&start_time);
        /* 其实只有t.tms_utime才是真正的CPU时间。但我们是在模拟一个
         * 只在用户状态运行的CPU大户，就像“for(;;);”。所以把t.tms_stime
         * 加上很合理。*/
        do
        {
```

```
            times(&current_time);
            utime = current_time.tms_utime - start_time.tms_utime;
            stime = current_time.tms_stime - start_time.tms_stime;
        } while ( ( (utime + stime) / HZ )  < cpu_time );
        last -= cpu_time;

        if (last <= 0 )
            break;

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

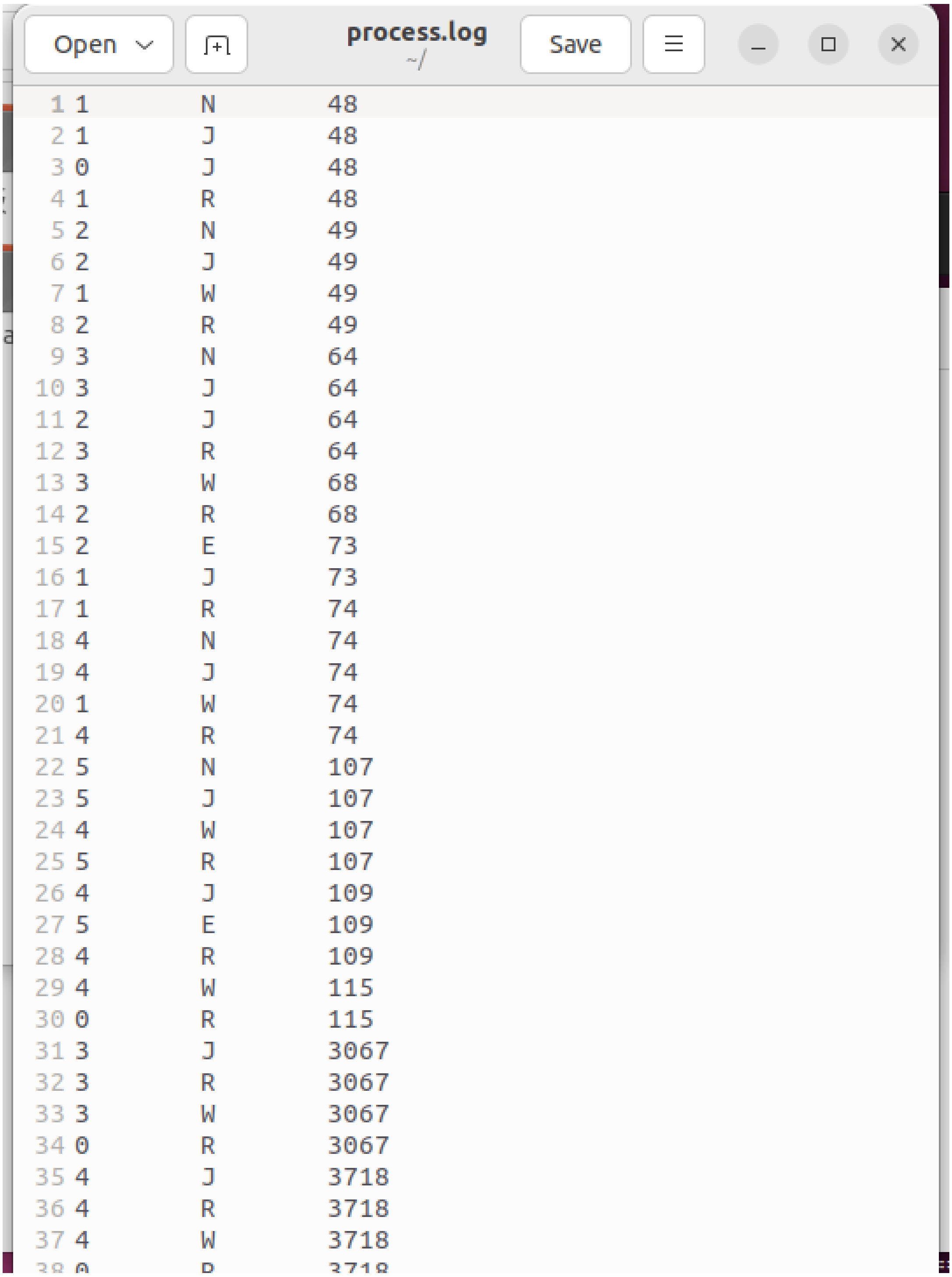
之后，启动根文件系统镜像，将process.c复制到`usr/root/`目录下。

5. 运行

在Linux-0.11中编译、运行process.c，打印显示结果如下。



在Ubuntu中查看process.log：




```
1	N	48
1	J	48
0	J	48
1	R	48
2	N	49
2	J	49
1	W	49
2	R	49
3	N	64
3	J	64
2	J	64
3	R	64
3	W	68
2	R	68
2	E	73
1	J	73
1	R	74
4	N	74
4	J	74
1	W	74
4	R	74
5	N	107
5	J	107
4	W	107
5	R	107
4	J	109
5	E	109
4	R	109
4	W	115
0	R	115
3	J	3067
3	R	3067
3	W	3067
0	R	3067
4	J	3718
4	R	3718
4	W	3718
0	R	3718
```

用stat_log.py统计进程15-19的输出结果（此处，需要根据系统中所安装的python解释器版本相应修改stat_log.py中的解释器目录）：

```
jxnout@jx-Ubuntu: ~/hit-oslab/common/files
正在解压 libpython2-stdlib:amd64 (2.7.18-3) ...
正在设置 python2-minimal (2.7.18-3) ...
正在选中未选择的软件包 python2。
(正在读取数据库 ... 系统当前共安装有 213863 个文件和目录。)
准备解压 .../python2_2.7.18-3_amd64.deb  ...
正在解压 python2 (2.7.18-3) ...
正在设置 libpython2-stdlib:amd64 (2.7.18-3) ...
正在设置 python2 (2.7.18-3) ...
正在处理用于 man-db (2.10.2-1) 的触发器 ...
jxnout@jx-Ubuntu:~/hit-oslab/common/files$ ls /usr/bin/python*
/usr/bin/python2    /usr/bin/python3     /usr/bin/python3-futurize
/usr/bin/python2.7  /usr/bin/python3.10  /usr/bin/python3-pasteurize
jxnout@jx-Ubuntu:~/hit-oslab/common/files$ ./stat_log.py ./process.log 15 16 17 18 19
(Unit: tick)
Process   Turnaround   Waiting   CPU Burst   I/O Burst
     15         1118        63           0        1055
     16         1688       633         200         855
     17         2043      1017         400         626
     18         2229      1221         600         408
     19         2208      1206         800         202
Average:     1857.20    828.00
Throughout: 0.22/s
jxnout@jx-Ubuntu:~/hit-oslab/common/files$
```

6. 修改时间片

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;
```

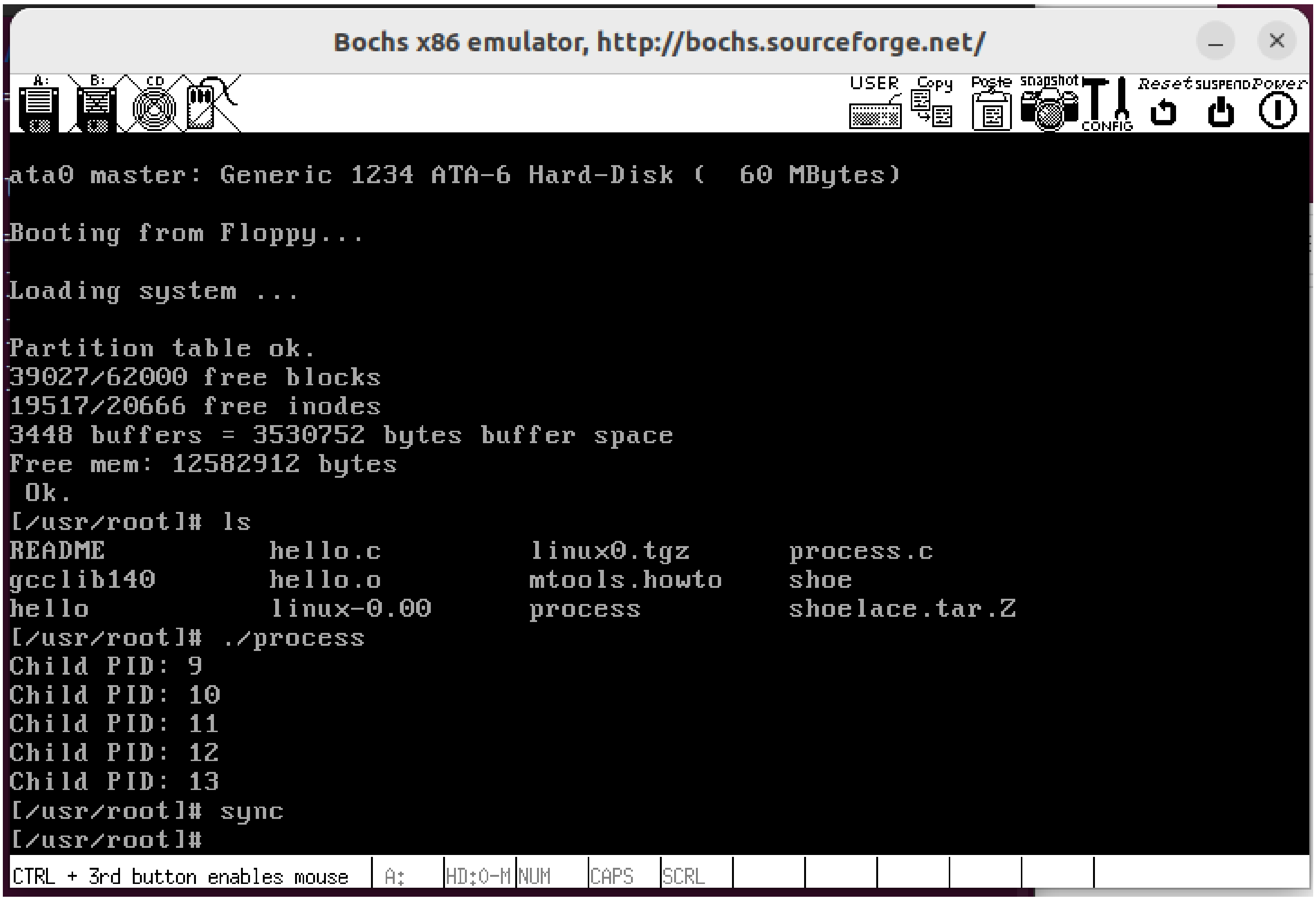
当就绪进程的 `counter` 为0时，不会被调度（ `schedule` 要选取 `counter` 最大的，大于0的进程），而当所有的就绪态进程的 `counter` 都变成0时，会执行

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

即新的 `counter` 值也等于 `priority` ，即初始时间片的大小。因此，要修改时间片，就需要修改 `include/linux/sched.h` 中的宏定义 `INIT_TASK` 。将其改为：

```
#define INIT_TASK \
  { 0,15,5, //分别对应state;counter;和priority;
```

运行结果：



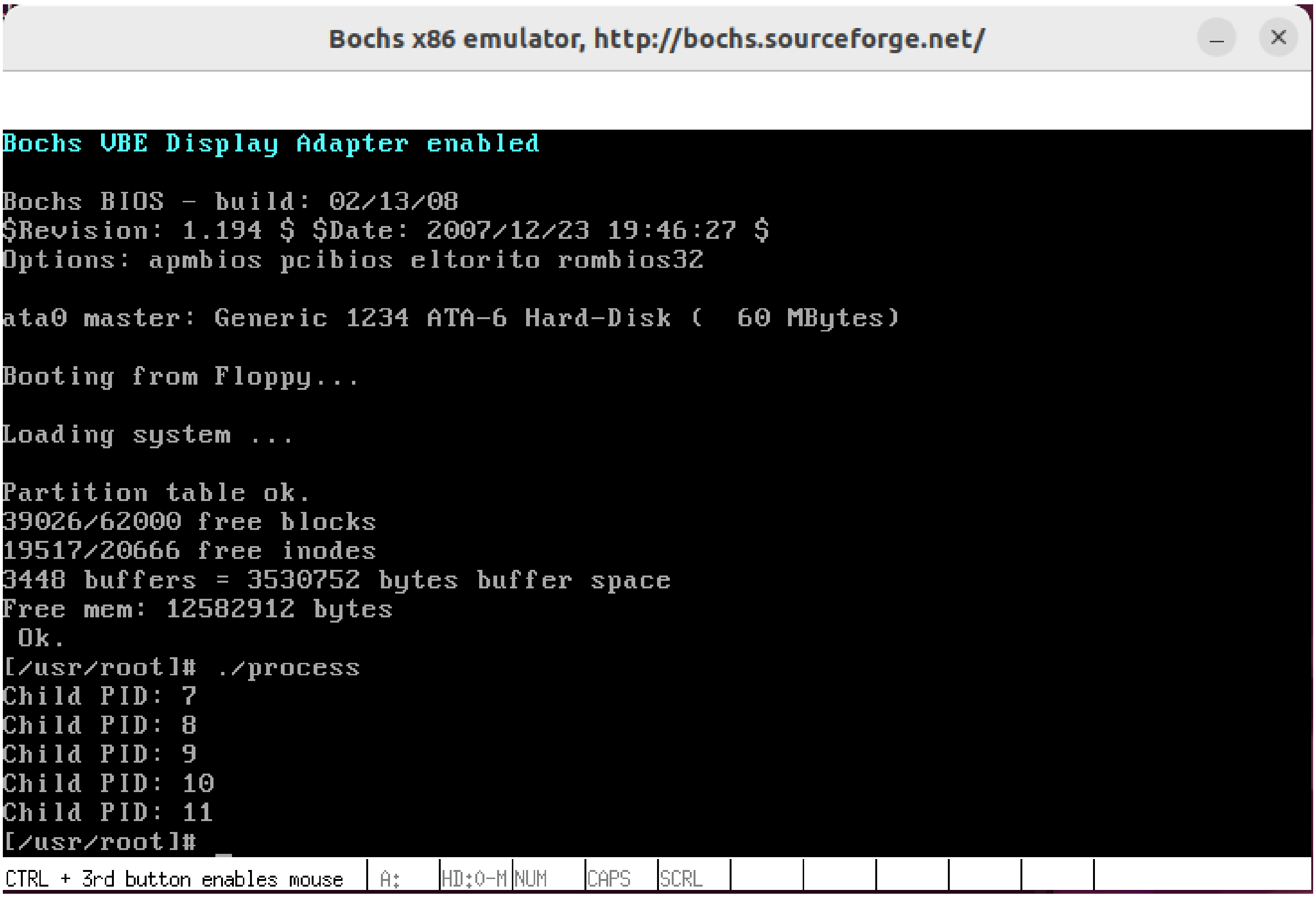
```
1	N	48
1	J	48
0	J	48
1	R	48
2	N	49
2	J	49
1	W	49
2	R	49
3	N	64
3	J	64
2	E	68
1	J	68
1	R	68
4	N	69
4	J	69
1	W	69
3	R	69
3	J	75
4	R	75
4	J	80
3	R	80
3	W	80
4	R	80
5	N	107
5	J	107
4	W	108
5	R	108
4	J	110
5	E	110
4	R	110
4	W	116
0	R	116
4	J	1123
4	R	1123
4	W	1124
```

```
jxnout@jx-Ubuntu: ~/hit-oslab/common/files
jxnout@jx-Ubuntu:~/hit-oslab/common/files$ ./stat_log.py ./process.log 9 10 11 12 13
(Unit: tick)
Process    Turnaround    Waiting    CPU Burst    I/O Burst
      9          1073         23            0         1050
     10          1657        617          200          840
     11          1942       1016          400          525
     12          1916       1211          600          105
     13          2005       1205          800            0
Average:      1718.60     814.40
Throughout: 0.25/s
jxnout@jx-Ubuntu:~/hit-oslab/common/files$
```

改为：

```c
#define INIT_TASK \
  { 0,15,50, //分别对应state;counter;和priority;
```

结果如下：



```
1       N       48
1       J       48
0       J       48
1       R       48
2       N       49
2       J       49
1       W       49
2       R       49
3       N       64
3       J       64
2       E       68
1       J       68
3       R       68
3       W       74
1       R       74
4       N       74
4       J       74
1       W       75
4       R       75
5       N       107
5       J       107
4       W       107
5       R       107
4       J       109
```

jxnout@jx-Ubuntu: ~/hit-oslab/common/files

```
jxnout@jx-Ubuntu:~/hit-oslab/common/files$ ./stat_log.py ./process.log 7 8 9 10
11
(Unit: tick)
Process    Turnaround    Waiting    CPU Burst    I/O Burst
      7          1704        204            0         1500
      8          2060        754          200         1106
      9          2211       1103          400          708
     10          2310       1257          600          453
     11          2209       1207          800          202
Average:      2098.80     905.00
Throughout: 0.22/s
```

可见，时间片很小时，大量的时间浪费在进程切换上，增加了平均周转时间和平均等待时间。时间片适宜的时，平均周转时间和平均等待时间是最低的。时间片足够大的时候，就变成了先进先出的调度算法，虽然不会浪费时间在进程切换上，但是又可能引起对短的交互请求的响应变差，平均周转时间和平均等待时间可能又会变大。

7. 问题回答

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？

多进程编程比单进程编程需要考虑的事情多了很多，需要考虑进程间的调度、进程间互不干扰、将性能最大化等问题。

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？

在**6. 修改时间篇**中均已作出回答。