

操作系统实验五 进程运行轨迹的跟踪与统计

一、实验目的

- 掌握 Linux 下的多进程编程技术；
- 通过对进程运行轨迹的跟踪来形象化进程的概念；
- 在进程运行轨迹跟踪的基础上进行相应的数据统计，从而能对进程调度算法进行实际的量化评价，更进一步加深对调度和调度算法的理解，获得能在实际操作系统上对调度算法进行实验数据对比的直接经验。

二、实验环境

Vmware 17.5.1, Ubuntu 20.04, Bochs 2.4.5

三、实验内容

- (1) 基于模板 `process.c` 编写多进程的样本程序，实现如下功能：
 - 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；
 - 父进程向标准输出打印所有子进程的 `id`，并在所有子进程都退出后才退出。
- (2) 在 Linux 0.11 上实现进程运行轨迹的跟踪。

基本任务是在内核中维护一个日志文件 `/var/process.log`，把从操作系统启动到系统关机过程中所有进程的运行轨迹都记录在这一 `log` 文件中。

(3) 在修改过的 0.11 上运行样本程序，通过分析 `log` 文件，统计该程序建立的所有进程的等待时间、完成时间（周转时间）和运行时间，然后计算平均等待时间，平均完成时间和吞吐量。可以自己编写统计程序，也可以使用 `python` 脚本程序 `stat_log.py` 进行统计。

(4) 修改 0.11 进程调度的时间片，然后再运行同样的样本程序，统计同样的时间数据，和原有的情况对比，体会不同时间片带来的差异。

四、实验过程

4.1 修改 `process.c` 文件

提供的 `process.c` 主要实现了 `cpuio_bound` 函数，此函数按照参数占用 CPU 和 I/O 时间，在模板基础上进行修改，用 `fork()` 创建 4 个同时运行的子进程，分别以 CPU 为主要任务、以 I/O 为主要任务、CPU 和 I/O 各 1 秒钟轮回，以及较多的 I/O+较少的 CPU。

实现效果：所有子进程都并行运行，每个子进程的实际运行时间不超过 30 秒，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。

```
10 void cputo_bound(int last, int cpu_time, int io_time);
11
12 int main(int argc, char * argv[])
13 {
14     pid_t pid_1, pid_2, pid_3, pid_4;
15     int exit_pid;
16
17     printf("parent pid = [%d]\n", getpid()); /* 获取父进程pid */
18
19     /* 以CPU为主要任务 */
20     pid_1 = fork();
21     if (pid_1 == 0) {
22         printf("[%d] is running now.\n", getpid());
23         cputo_bound(10, 1, 0);
24         exit(0); /* 终止进程, 返回状态码0 */
25     }
26
27     /* 以I/O为主要任务 */
28     pid_2 = fork();
29     if (pid_2 == 0) {
30         printf("[%d] is running now.\n", getpid());
31         cputo_bound(10, 0, 1);
32         exit(0); /* 终止进程, 返回状态码0 */
33     }
34
35     /* CPU和I/O各1秒钟轮回 */
36     pid_3 = fork();
37     if (pid_3 == 0) {
38         printf("[%d] is running now.\n", getpid());
39         cputo_bound(10, 1, 1);
40         exit(0); /* 终止进程, 返回状态码0 */
41     }
42
43     /* 较多的I/O+较少的CPU */
44     pid_4 = fork();
45     if (pid_4 == 0) {
46         printf("[%d] is running now.\n", getpid());
47         cputo_bound(10, 1, 9);
48         exit(0); /* 终止进程, 返回状态码0 */
49     }
50
51     exit_pid = wait(NULL);
52     printf("[%d] have exited.\n", exit_pid);
53
54     exit_pid = wait(NULL);
55     printf("[%d] have exited.\n", exit_pid);
56
57     exit_pid = wait(NULL);
58     printf("[%d] have exited.\n", exit_pid);
59
60     exit_pid = wait(NULL);
61     printf("[%d] have exited.\n", exit_pid);
62
63     printf("the program was executed successfully.\n");
64     return 0;
65 }
```

4.2 修改 Linux 0.11 源代码文件

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。状态转移的函数涉及/kernel 目录下的 fork.c、sched.c、exit.c 文件，找到所有发生进程状态切换的位置，添加适当代码向 log 文件输出进程状态变化的信息，从而跟踪进程运行轨迹。

4.2.1 修改 init/main.c

为了在内核启动时就打开 log 文件，将 init 函数中加载文件系统的代码移至 main 函数中“move_to_user_mode();”后面，这段代码建立文件描述符 0（stdin）、1（stdout）、2（stderr），并和/dev/tty0 关联。为了将 log 文件的描述符关联到 3，在其后添加一行“open("/var/process.log", O_CREAT | O_TRUNC | O_WRONLY, 0666);”，建立只写文件，并设置权限为所有人可读可写。

实现效果：进程 0 运行 move_to_user_mode()切换到用户模式，接着加载文件系统，打开 log 文件，开始记录进程的运行轨迹，然后全系统第一次调用 fork()建立进程 1，进程 1 调用 init()进行初始化，并执行后续任务。

```
main.c
~/OS/linux011/cslab5/linux-0.11/init

104 void main(void) /* This really IS void, no error here. */
105 { /* The startup routine assumes (well, ...) this */
106 /*
107  * Interrupts are still disabled. Do necessary setups, then
108  * enable them
109  */
110     ROOT_DEV = ORIG_ROOT_DEV;
111     drive_info = DRIVE_INFO;
112     memory_end = (1<<20) * (EXT_MEM_K<<10);
113     memory_end &= 0xfffff000;
114     if (memory_end > 16*1024*1024)
115         memory_end = 16*1024*1024;
116     if (memory_end > 12*1024*1024)
117         buffer_memory_end = 4*1024*1024;
118     else if (memory_end > 0*1024*1024)
119         buffer_memory_end = 2*1024*1024;
120     else
121         buffer_memory_end = 1*1024*1024;
122     main_memory_start = buffer_memory_end;
123 #ifdef RAMDISK
124     main_memory_start += rd_init(main_memory_start, RAMDISK*1024);
125 #endif
126     mem_init(main_memory_start, memory_end);
127     trap_init();
128     blk_dev_init();
129     chr_dev_init();
130     tty_init();
131     time_init();
132     sched_init();
133     buffer_init(buffer_memory_end);
134     hd_init();
135     floppy_init();
136     st1();
137     move_to_user_mode();
138     setup(void *) &drive_info; /* 加载文件系统 */
139     (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 */
140     (void) dup(0); /* 让文件描述符1也和/dev/tty0关联 */
141     (void) dup(0); /* 让文件描述符2也和/dev/tty0关联 */
142     open("/var/process.log", O_CREAT | O_TRUNC | O_WRONLY, 0666); /* 让文件描述符3和log文件关联 */
143     if (!fork()) { /* we count on this going ok */
144         t1nt();
145     }
146 /
147 * NOTE!! For any other task 'pause()' would mean we have to get a
148 * signal to awaken, but task0 is the sole exception (see 'schedule()')
149 * as task 0 gets activated at every idle moment (when no other tasks
150 * can run). For task0 'pause()' just means we go check if some other
151 * task can run, and if not we return here.
152 */
153     for(;;) pause();
154 }
155
```

4.2.2 修改 kernel/printk.c

所有的状态转移都是在内核进行的，而 write()只能在用户模式下执行，在内核状态下不可用（类似实验四中的 printf 和 printk 函数），因此需编写 fprintk()调用，使得在内核状态下也可以写 log 文件。

fprintk()源码已在实验指导中给出，将此函数放入 kernel/printk.c。

```
printk.c
~/OS/linux011/cslab5/linux-0.11/kernel

46 int fprintk(int fd, const char *fmt, ...)
47 {
48     va_list args;
49     int count;
50     struct file * file;
51     struct m_inode * inode;
52
53     va_start(args, fmt);
54     count=vsprintf(logbuf, fmt, args);
55     va_end(args);
56
57     if (fd < 3) /* 如果输出到stdout或stderr, 直接调用sys_write即可 */
58     {
59         __asm__ ("push %%fs\n\t"
60                "push %%ds\n\t"
61                "pop %%fs\n\t"
62                "pushl $logbuf\n\t" /* 注意对于windows环境来说, 是_logbuf,下同 */
63                "pushl $1\n\t"
64                "call sys_write\n\t" /* 注意对于windows环境来说, 是_sys_write,下同 */
65                "addl $8,%%esp\n\t"
66                "popl %0\n\t"
67                "pop %%fs"
68                : "r" (count), "r" (fd): "ax", "cx", "dx");
69     }
70     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。 */
71     {
72         if (!file=task[1]->filp[fd] || !task[1]->filp[fd]->f_inode->i_dev) { /* 从进程的的文件描述符表中得到文件句柄 */
73             return 0;
74         }
75         inode=file->f_inode;
76
77         __asm__ ("push %%fs\n\t"
78                "push %%ds\n\t"
79                "pop %%fs\n\t"
80                "pushl $0\n\t"
81                "pushl $logbuf\n\t"
82                "pushl $1\n\t"
83                "pushl $2\n\t"
84                "call file_write\n\t"
85                "addl $12,%%esp\n\t"
86                "popl %0\n\t"
87                "pop %%fs"
88                : "r" (count), "r" (file), "r" (inode): "ax", "cx", "dx");
89     }
90     return count;
91 }
92
```

4.2.3 修改 kernel/fork.c

创建新进程需要通过系统调用 fork()实现，查看其在内核中的具体实现 sys_fork()可知真正实现进程创建的函数是 copy_process(), 在 kernel/fork.c 中。

“p->start_time=jiffies;”表示进程新建完毕，故向 log 文件输出一条新建记录；“p->state = TASK_RUNNING;”设置进程进入就绪态，故向 log 文件输出一条就绪记录。

```
fork.c
~/OS/linux011/oslab5/linux-0.11/kernel

90 p->utime = p->stime = 0;
91 n->curtime = n->stime = 0;
92 p->start_time = jiffies;
93 fprintf(3, "%ld\t%c\t%ld\n", p->pid, 'N', jiffies);
94 p->tss.uack_link = 0;
95 p->tss.esp0 = PAGE_SIZE + (long) p;
96 p->tss.ss0 = 0x10;
97 p->tss.eip = eip;
98 p->tss.eflags = eflags;
99 p->tss.eax = 0;
100 p->tss.ecx = ecx;
101 p->tss.edx = edx;
102 p->tss.ebx = ebx;
103 p->tss.esp = esp;
104 p->tss.ebp = ebp;
105 p->tss.esi = esi;
106 p->tss.edi = edi;
107 p->tss.es = es & 0xffff;
108 p->tss.cs = cs & 0xffff;
109 p->tss.ss = ss & 0xffff;
110 p->tss.ds = ds & 0xffff;
111 p->tss.fs = fs & 0xffff;
112 p->tss.gs = gs & 0xffff;
113 p->tss.ldt = _ldt(nr);
114 p->tss.trace_bitmap = 0x80000000;
115 if (last_task_used_math == current)
116     _asm__("cldts; fnsave %0":"n" (p->tss.i387));
117 if (copy_nem(nr,p) {
118     task[nr] = NULL;
119     free_page((long) p);
120     return -EAGAIN;
121 }
122 for (i=0; i<NR_OPEN;i++)
123     if ((f=p->filp[i]))
124         f->f_count++;
125 if (current->pwd)
126     current->pwd->i_count++;
127 if (current->root)
128     current->root->i_count++;
129 if (current->executable)
130     current->executable->i_count++;
131 set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY,&(p->tss));
132 set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,&(n->ldt));
133 p->state = TASK_RUNNING; /* do this last, just in case */
134 fprintf(3, "%ld\t%c\t%ld\n", p->pid, 'J', jiffies);
135 return last_pid;
136 }
```

4.2.4 修改 kernel/sched.c

(1) 就绪态↔运行态:

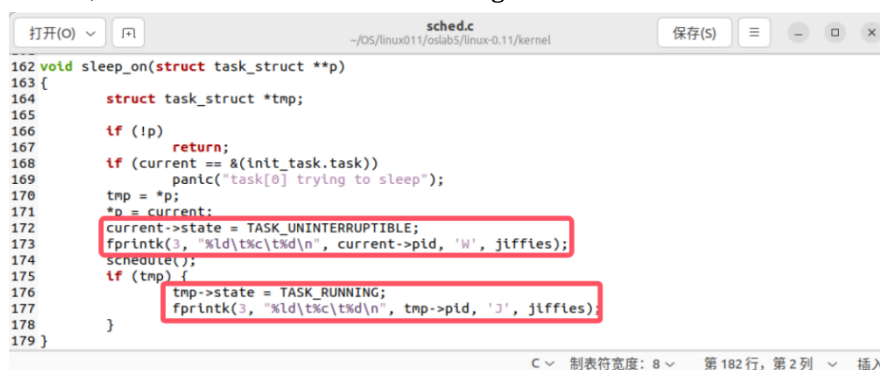
• schedule(): “(p)->state = TASK_RUNNING;” 设置进程进入就绪态, 故向 log 文件输出一条就绪记录; 当前进程时间片到时, 需转换为就绪态, 故向 log 文件输出一条就绪记录, next 进程是接下来要运行的进程, 故向 log 文件输出一条运行记录, 之后通过 “switch_to(next);” 实现当前运行进程的切换。

```
sched.c
~/OS/linux011/oslab5/linux-0.11/kernel

104 void schedule(void)
105 {
106     int i,next,c;
107     struct task_struct ** p;
108
109     /* check alarm, wake up any interruptible tasks that have got a signal */
110
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)) && (*p)->state==TASK_INTERRUPTIBLE) {
118                 (*p)->state=TASK_RUNNING;
119                 fprintf(3, "%ld\t%c\t%ld\n", (*p)->pid, 'J', jiffies);
120             }
121         }
122
123     /* this is the scheduler proper: */
124
125     while (1) {
126         c = -1;
127         next = 0;
128         i = NR_TASKS;
129         p = &task[NR_TASKS];
130         while (--i) {
131             if (!*p)
132                 continue;
133             if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
134                 c = (*p)->counter, next = i;
135             if (c) break;
136         }
137         for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
138             if (*p)
139                 (*p)->counter = ((*p)->counter >> 1) +
140                     (*p)->priority;
141     }
142
143     /* 编号为next的进程转为运行态 */
144     if(task[next]->pid != current->pid) {
145         /* 时间片到时进程转为就绪态 */
146         if(current->state == TASK_RUNNING)
147             fprintf(3, "%ld\t%c\t%ld\n", current->pid, 'J', jiffies);
148         fprintf(3, "%ld\t%c\t%ld\n", task[next]->pid, 'R', jiffies);
149     }
150     switch_to(next);
151 }
```

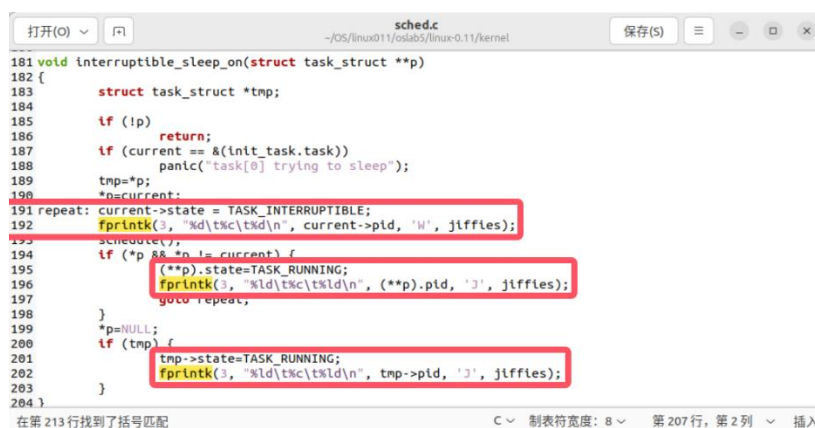
(2) 运行态→睡眠态:

• `sleep_on()`: “`current->state = TASK_UNINTERRUPTIBLE;`” 设置进程进入不可中断睡眠态 (只能由 `wake_up()` 显式唤醒), 故向 `log` 文件输出一条阻塞记录; “`tmp->state = TASK_RUNNING;`” 设置进程进入就绪态, 故向 `log` 文件输出一条就绪记录。



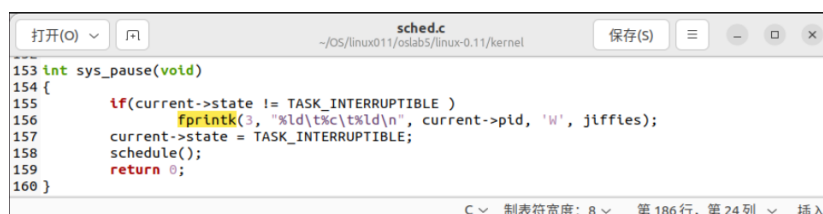
```
162 void sleep_on(struct task_struct **p)
163 {
164     struct task_struct *tmp;
165     if (!p)
166         return;
167     if (current == &(init_task.task))
168         panic("task[0] trying to sleep");
169     tmp = *p;
170     *p = current;
171     current->state = TASK_UNINTERRUPTIBLE;
172     fprintf(3, "%d\t%c\t%d\n", current->pid, 'W', jiffies);
173     schedule();
174     if (tmp) {
175         tmp->state = TASK_RUNNING;
176         fprintf(3, "%d\t%c\t%d\n", tmp->pid, 'J', jiffies);
177     }
178 }
179 }
```

• `interruptible_sleep_on()`: “`current->state = TASK_INTERRUPTIBLE;`” 设置进程进入可中断睡眠态, 故向 `log` 文件输出一条阻塞记录; “`(**p).state = TASK_RUNNING;`” 设置进程进入就绪态, 故向 `log` 文件输出一条就绪记录; “`tmp->state = TASK_RUNNING;`” 设置进程进入就绪态, 故向 `log` 文件输出一条就绪记录。



```
181 void interruptible_sleep_on(struct task_struct **p)
182 {
183     struct task_struct *tmp;
184     if (!p)
185         return;
186     if (current == &(init_task.task))
187         panic("task[0] trying to sleep");
188     tmp = *p;
189     *p = current;
190     repeat: current->state = TASK_INTERRUPTIBLE;
191     fprintf(3, "%d\t%c\t%d\n", current->pid, 'W', jiffies);
192     schedule();
193     if (*p && *p != current) {
194         (**p).state = TASK_RUNNING;
195         fprintf(3, "%d\t%c\t%d\n", (**p).pid, 'J', jiffies);
196         goto repeat;
197     }
198     *p = NULL;
199     if (tmp) {
200         tmp->state = TASK_RUNNING;
201         fprintf(3, "%d\t%c\t%d\n", tmp->pid, 'J', jiffies);
202     }
203 }
204 }
```

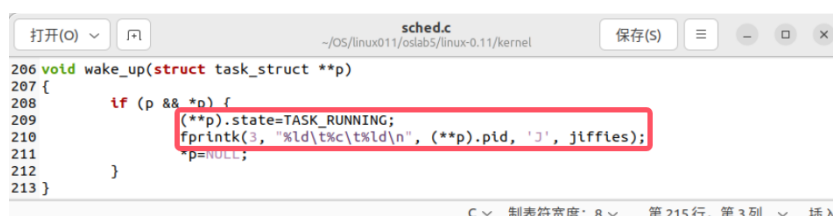
• `sys_pause()`: “`current->state = TASK_INTERRUPTIBLE;`” 设置进程进入可中断睡眠态, 故向 `log` 文件输出一条阻塞记录。



```
153 int sys_pause(void)
154 {
155     if (current->state != TASK_INTERRUPTIBLE)
156         fprintf(3, "%d\t%c\t%d\n", current->pid, 'W', jiffies);
157     current->state = TASK_INTERRUPTIBLE;
158     schedule();
159     return 0;
160 }
```

(3) 睡眠态→就绪态:

• `wake_up()`: “`(**p).state = TASK_RUNNING;`” 设置进程进入就绪态, 故向 `log` 文件输出一条就绪记录。



```
206 void wake_up(struct task_struct **p)
207 {
208     if (p && *p) {
209         (**p).state = TASK_RUNNING;
210         fprintf(3, "%d\t%c\t%d\n", (**p).pid, 'J', jiffies);
211         *p = NULL;
212     }
213 }
```

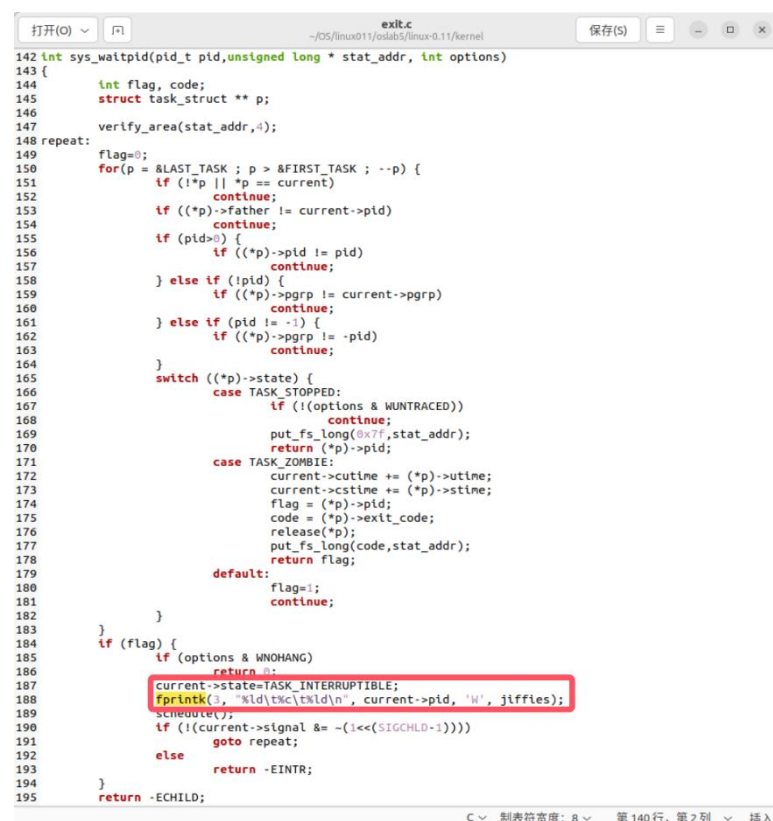
4.2.5 修改 kernel/exit.c

• `do_exit()`: “`current->state = TASK_ZOMBIE;`” 表示将要杀死进程，故向 log 文件输出一条退出记录。



```
102 int do_exit(long code)
103 {
104     int i;
105     free_page_tables(get_base(current->ldt[1]), get_limit(0x0f));
106     free_page_tables(get_base(current->ldt[2]), get_limit(0x17));
107     for (i=0; i<NR_TASKS; i++)
108         if (task[i] && task[i]->father == current->pid) {
109             task[i]->father = 1;
110             if (task[i]->state == TASK_ZOMBIE)
111                 /* assumption task[i] is always init */
112                 (void) send_sig(SIGCHLD, task[i], 1);
113         }
114     for (i=0; i<NR_OPEN; i++)
115         if (current->filp[i])
116             sys_close(i);
117     iput(current->pwd);
118     current->pwd=NULL;
119     iput(current->root);
120     current->root=NULL;
121     iput(current->executable);
122     current->executable=NULL;
123     if (current->leader && current->tty >= 0)
124         tty_table[current->tty].pggrp = 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     fprintf(3, \"%ld\\t%\\t%ld\\n\", current->pid, 'E', jiffies);
131     current->exit_code = code;
132     tell_father(current->father);
133     schedule();
134     return (-1); /* just to suppress warnings */
135 }
```

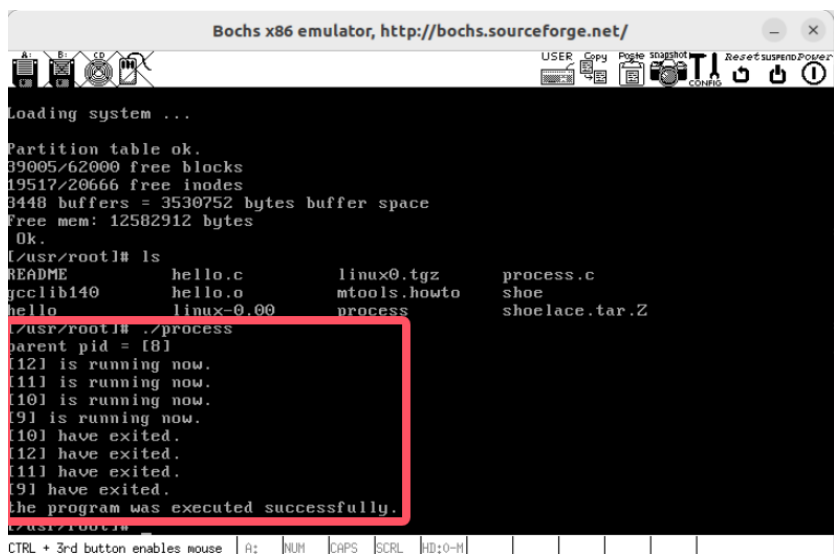
• `sys_waitpid()`: “`current->state=TASK_INTERRUPTIBLE;`” 设置进程进入可中断睡眠态，故向 log 文件输出一条阻塞记录。



```
142 int sys_waitpid(pid_t pid, unsigned long * stat_addr, int options)
143 {
144     int flag, code;
145     struct task_struct ** p;
146     verify_area(stat_addr, 4);
147     repeat:
148     flag=0;
149     for(p = &LAST_TASK; p > &FIRST_TASK; --p) {
150         if (!*p || *p == current)
151             continue;
152         if ((*p)->father != current->pid)
153             continue;
154         if (pid > 0) {
155             if ((*p)->pid != pid)
156                 continue;
157         } else if (!pid) {
158             if ((*p)->pggrp != current->pggrp)
159                 continue;
160         } else if (pid != -1) {
161             if ((*p)->pggrp != -pid)
162                 continue;
163         }
164         switch ((*p)->state) {
165             case TASK_STOPPED:
166                 if (!(options & WUNTRACED))
167                     continue;
168                 put_fs_long(0x7f, stat_addr);
169                 return (*p)->pid;
170             case TASK_ZOMBIE:
171                 current->cutime += (*p)->utime;
172                 current->stime += (*p)->stime;
173                 flag = (*p)->pid;
174                 code = (*p)->exit_code;
175                 release(*p);
176                 put_fs_long(code, stat_addr);
177                 return flag;
178             default:
179                 flag=1;
180                 continue;
181         }
182     }
183     if (flag) {
184         if (options & WNOHANG)
185             return 0;
186         current->state=TASK_INTERRUPTIBLE;
187         fprintf(3, \"%ld\\t%\\t%ld\\n\", current->pid, 'W', jiffies);
188         schedule();
189         if (!(current->signal &= ~(1<<(SIGCHLD-1))))
190             goto repeat;
191         else
192             return -EINTR;
193     }
194     return -ECHILD;
195 }
```

4.3 运行测试文件

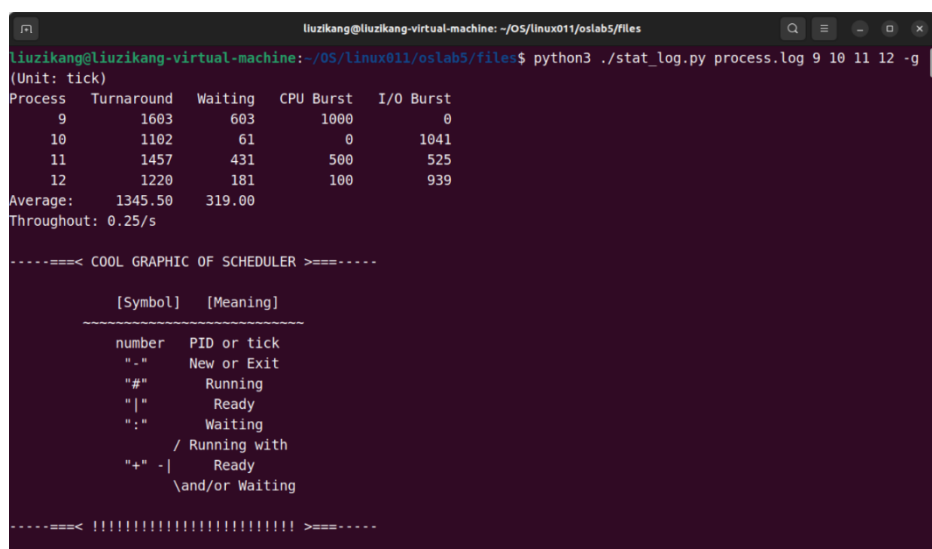
将修改完的 `process.c` 文件复制到 Linux 0.11 的 `/usr/root` 目录下。启动 Bochs，在 Linux 0.11 下编译运行 `process.c`，屏幕输出如下所示。样本程序建立的进程 `pid` 分别为 9、10、11、12。



```
Bochs x86 emulator, http://bochs.sourceforge.net/

Loading system ...
Partition table ok.
89005/62000 free blocks
19517/20666 free inodes
8448 buffers = 3530752 bytes buffer space
Free mem: 12582912 bytes
Ok.
[/usr/root]# ls
README          hello.c          linux0.tgz       process.c
gcclib140       hello.o          mtools.howto    shoe
hello           linux-0.00       process          shoelace.tar.Z
[/usr/root]# ./process
parent pid = [0]
[12] is running now.
[11] is running now.
[10] is running now.
[9] is running now.
[10] have exited.
[12] have exited.
[11] have exited.
[9] have exited.
the program was executed successfully.
[/usr/root]#
```

将生成的/var/process.log 文件复制到虚拟机，使用 stat_log.py 程序统计样本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，并计算平均等待时间、平均完成时间和吞吐量。运行结果如下所示：



```
liuzikang@liuzikang-virtual-machine: ~/OS/linux011/oslab5/files
liuzikang@liuzikang-virtual-machine:~/OS/linux011/oslab5/files$ python3 ./stat_log.py process.log 9 10 11 12 -g
(Unit: tick)
Process  Turnaround  Waiting  CPU Burst  I/O Burst
9        1603         603      1000       0
10       1102         61       0        1041
11       1457         431      500       525
12       1220         181      100       939
Average:  1345.50   319.00
Throughout: 0.25/s

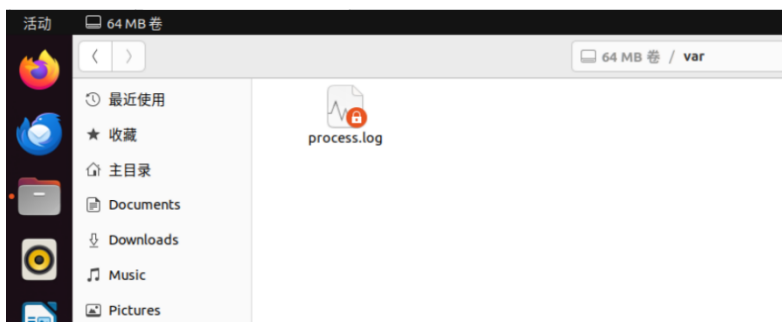
-----< COOL GRAPHIC OF SCHEDULER >-----

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

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

五、实验结果

查看/var 目录，可以看到 process.log 日志文件建立成功。



process.log 内容如下图所示，进程的五种状态（新建 N、就绪 J、运行 R、阻塞 W 和退出 E）均有体现。由于进程 0 运行时才开始加载文件系统，因此进程 0 进入创建和运行状态

的信息并未被写入 log 文件中，而是从进程 1 开始。



1	1	N	48
2	1	J	48
3	0	J	48
4	1	R	48
5	2	N	49
6	2	J	49
7	1	W	49
8	2	R	49
9	3	N	63
10	3	J	64
11	2	J	64
12	3	R	64
13	3	W	68
14	2	R	68
15	2	E	73
16	1	J	73
17	1	R	73
18	4	N	74
19	4	J	74
20	1	W	74
21	4	R	74
22	5	N	106
23	5	J	106
24	4	W	107
25	5	R	107
26	4	J	109
27	5	E	109
28	4	R	109
29	4	W	115
30	0	R	115
31	0	W	115
32	4	J	292

回答问题：

1. 结合自己的体会，谈谈从程序设计者的角度看，单进程编程和多进程编程最大的区别是什么？

单进程编程和多进程编程的最大区别在于进程并发性和资源共享机制。

(1) 进程并发性

- 单进程编程：始终只有一个进程在运行。程序内的各个任务按顺序依次执行，没有并发执行的能力。若需要同时执行多个任务，程序需要通过轮询、回调或协作的方式来模拟“并发”。

- 多进程编程：有多个独立的进程同时运行。操作系统可以调度不同进程在不同的 CPU 核心上并行执行，能够充分利用多核处理器的计算能力，从而实现真正的并行处理。

(2) 资源共享机制

- 单进程编程：同一进程中的不同任务可以共享内存、文件句柄等资源。数据传递通常通过内存共享、局部变量、堆栈等方式完成，因此程序设计也较为简单。

- 多进程编程：每个进程拥有独立的地址空间和资源，因此进程间的数据共享变得更加复杂。通常需要使用进程间通信（IPC）机制，如管道、消息队列、共享内存等来实现数据交换。不同进程的资源（如内存）是隔离的，互不干扰。

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？

进程的 counter 是在 fork() 中设定的，fork() 会调用 copy_process() 拷贝父进程信息。查看 /kernel/fork.c 文件，copy_process() 中 “*p = *current;” 复制父进程的 PCB 数据信息，“p->counter = p->priority;” 初始化子进程的 counter，这里 p->priority 继承自父进程的优先级，其不会像 counter 一样发生改变。

而当所有就绪态进程的 counter 都为 0 时，会执行 “(*p)->counter = ((*p)->counter >> 1) + (*p)->priority;” 语句，算出的新的 counter 值也等于 priority，即初始时间片的大小。

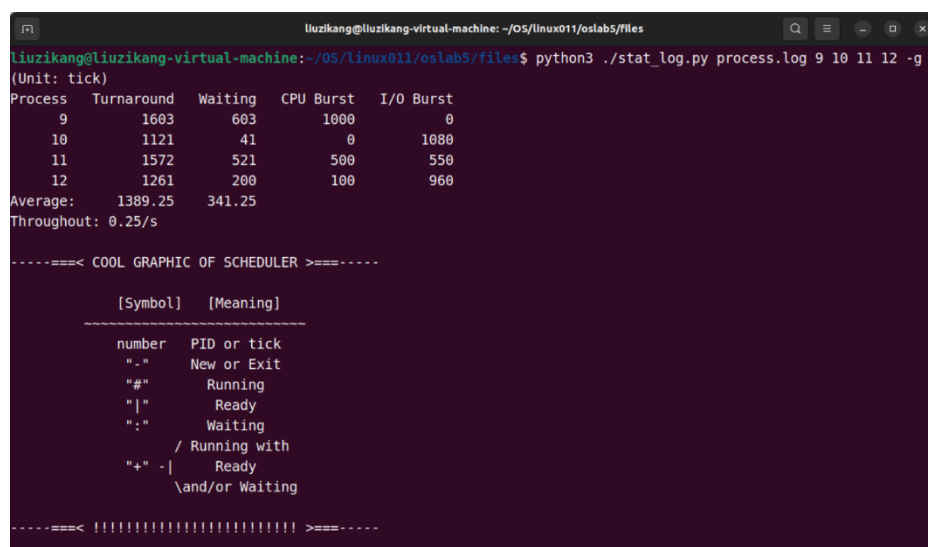
因此每个进程的初始 counter 和更新 counter 都是父进程的 priority，递归后可知时间片的初值即为进程 0 的 priority，该值在 /include/linux/sched.h 中由宏 INIT_TASK 定义，0、15

和 15 分别对应 state、counter 和 priority。



```
109};
110
111/*
112 * INIT_TASK is used to set up the first task table, touch at
113 * your own risk!. Base=0, limit=0x9ffff (=640kB)
114 */
115#define INIT_TASK \
116/* state etc */ { 0,15,15, \
117/* signals */ 0, {}, 0, \
118/* ec, brk... */ 0, 0, 0, 0, 0, \
119/* pid etc.. */ 0, -1, 0, 0, 0, \
120/* uid etc */ 0, 0, 0, 0, 0, \
121/* alarm */ 0, 0, 0, 0, 0, \
122/* math */ 0, \
123/* fs info */ -1, 0, 0, 22, NULL, NULL, NULL, 0, \
124/* filp */ {NULL,}, \
125{ \
126{ \
127/* ldt */ \
128{ \
129}, \
130/*tss*/ {0, PAGE_SIZE+(long)&init_task, 0x10, 0, 0, 0, 0, (long)&pg_dir, \
1310, 0, 0, 0, 0, 0, 0, \
1320, 0, 0x17, 0x17, 0x17, 0x17, 0x17, 0x17, \
133_LDT(0), 0x80000000, \
134{ } \
135}, \
136} \
137
138extern struct task_struct *task[NR_TASKS];
139extern struct task_struct *last_task_used_math;
```

修改此处 priority 的值为 10 或 20，并分别在 Linux 0.11 下编译运行 process.c，统计结果分别如下所示：

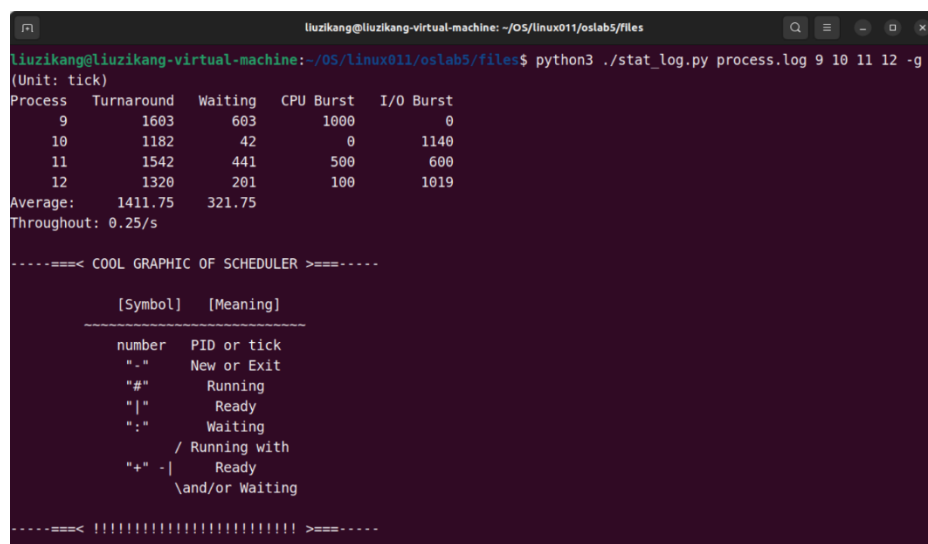


```
liuzikang@liuzikang-virtual-machine: ~/OS/linux011/oslab5/files
liuzikang@liuzikang-virtual-machine:~/OS/linux011/oslab5/files$ python3 ./stat_log.py process.log 9 10 11 12 -g
(Unit: tick)
Process  Turnaround  Waiting  CPU Burst  I/O Burst
    9      1603      603    1000      0
   10      1121       41      0    1080
   11      1572      521     500     550
   12      1261      200     100     960
Average:   1389.25   341.25
Throughout: 0.25/s

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

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

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



```
liuzikang@liuzikang-virtual-machine: ~/OS/linux011/oslab5/files
liuzikang@liuzikang-virtual-machine:~/OS/linux011/oslab5/files$ python3 ./stat_log.py process.log 9 10 11 12 -g
(Unit: tick)
Process  Turnaround  Waiting  CPU Burst  I/O Burst
    9      1603      603    1000      0
   10      1182       42      0    1140
   11      1542      441     500     600
   12      1320      201     100    1019
Average:   1411.75   321.75
Throughout: 0.25/s

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

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

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

对比 4.3 部分修改时间片前的统计结果，发现平均周转时间和等待时间都有所增加。

猜测增大时间片可能导致每个进程在 CPU 上执行的时间变长，因此一个进程从就绪态到完成的时间较长，且在等待队列中的进程等待获取 CPU 资源的时间也相应会增加；而减小时间片可能导致频繁的上下文切换，增加了进程切换的开销，且进程更频繁地进入就绪队列，等待执行次数增加。这些均会导致平均周转时间和等待时间增加。