

# Lab3 进程切换

201300032 肖思远

## 实验环境

Ubuntu 21.04 + gcc 10.3.0

## 实验目的

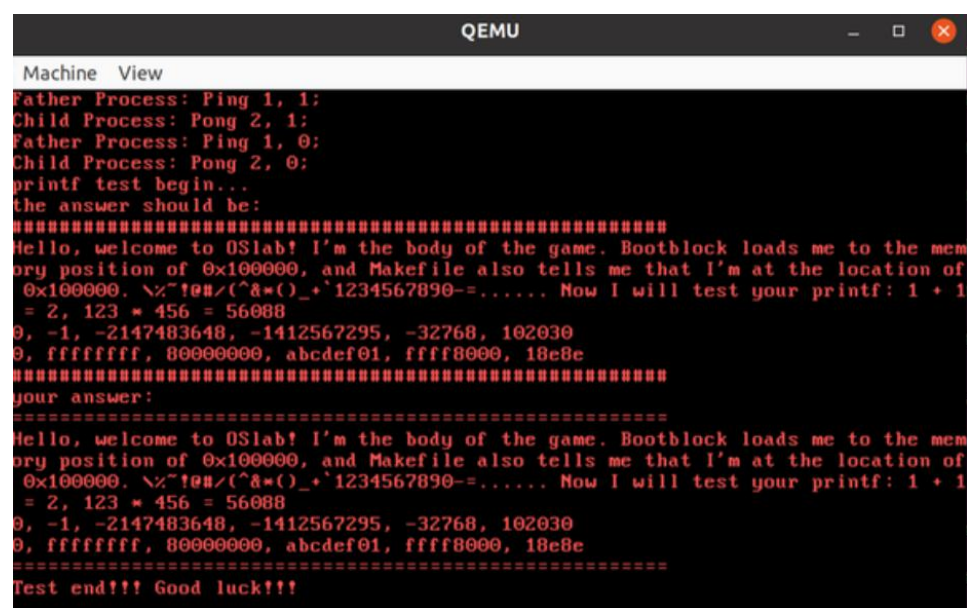
1. 填补 fork, sleep 以及 exit 函数缺失的代码
2. 完成时钟中断处理
3. 完成系统调用库函数

## 实验进度

我完成了全部内容，包括两个选做（但是选做一有些意外情况）

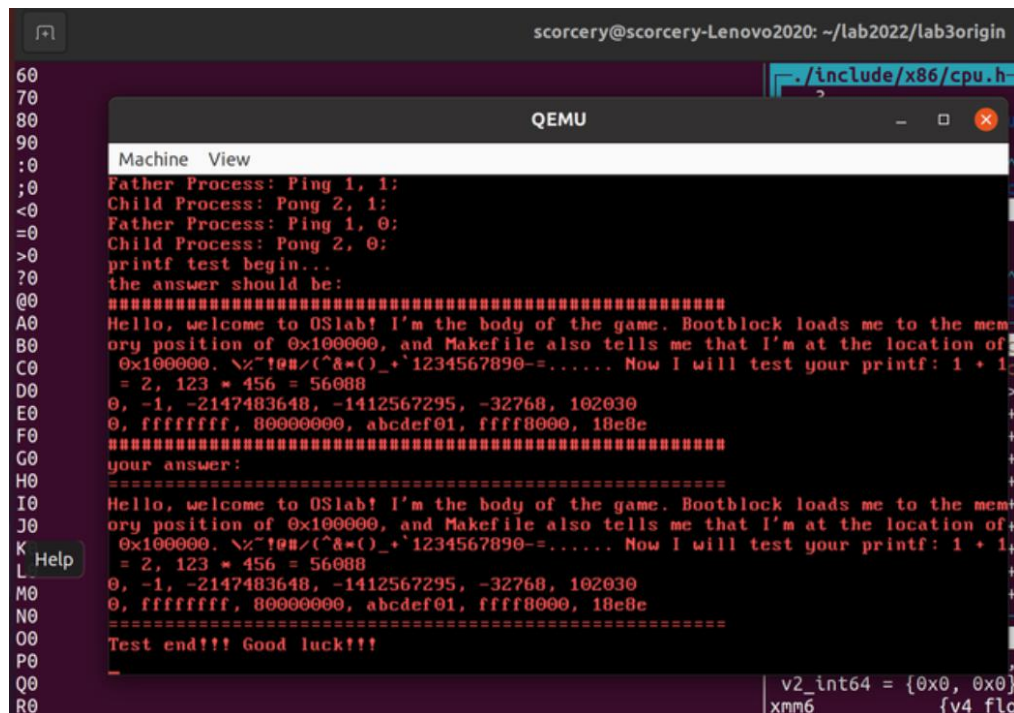
## 实验结果

### 必做部分



```
Machine View
Father Process: Ping 1, 1;
Child Process: Pong 2, 1;
Father Process: Ping 1, 0;
Child Process: Pong 2, 0;
printf test begin...
the answer should be:
Hello, welcome to OSlab! I'm the body of the game. Bootblock loads me to the mem
ory position of 0x100000, and Makefile also tells me that I'm at the location of
0x100000. \x\t@#/(^a*(*)_+'1234567890-=..... Now I will test your printf: 1 + 1
= 2, 123 * 456 = 56088
0, -1, -2147483648, -1412567295, -32768, 102030
0, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
your answer:
Hello, welcome to OSlab! I'm the body of the game. Bootblock loads me to the mem
ory position of 0x100000, and Makefile also tells me that I'm at the location of
0x100000. \x\t@#/(^a*(*)_+'1234567890-=..... Now I will test your printf: 1 + 1
= 2, 123 * 456 = 56088
0, -1, -2147483648, -1412567295, -32768, 102030
0, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
Test end!!! Good luck!!!
```

## 选做一



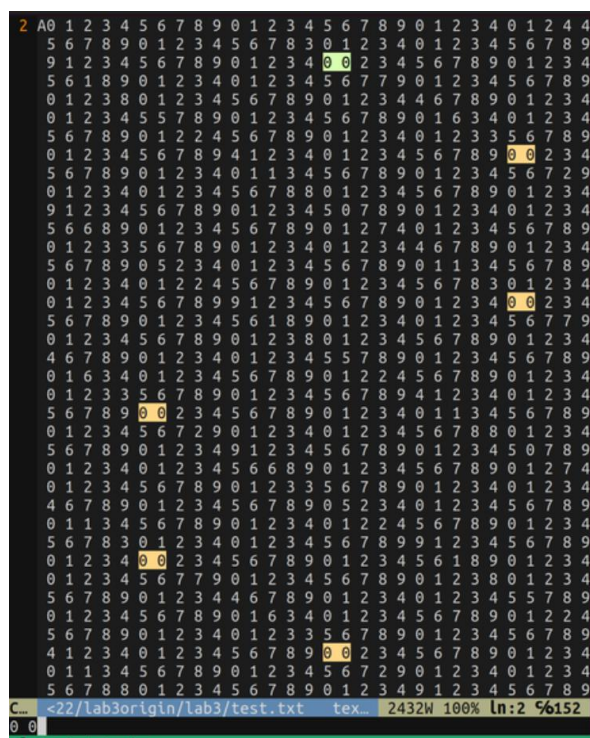
```
scorcery@scorcery-Lenovo2020: ~/lab2022/lab3origin
60
70
80
90
:0
;0
<0
=0
>0
?0
@0
A0
B0
C0
D0
E0
F0
G0
H0
I0
J0
K
L
M0
N0
O0
P0
Q0
R0

Machine View
Father Process: Ping 1, 1:
Child Process: Pong 2, 1:
Father Process: Ping 1, 0:
Child Process: Pong 2, 0:
printf test begin...
the answer should be:
#####
Hello, welcome to OSlab! I'm the body of the game. Bootblock loads me to the mem-
ory position of 0x100000, and Makefile also tells me that I'm at the location of
0x100000. \x!00/(\^&*(*)_+'1234567890-=..... Now I will test your printf: 1 + 1
= 2, 123 * 456 = 56088
0, -1, -2147483648, -1412567295, -32768, 102030
0, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
#####
your answer:
=====
Hello, welcome to OSlab! I'm the body of the game. Bootblock loads me to the mem-
ory position of 0x100000, and Makefile also tells me that I'm at the location of
0x100000. \x!00/(\^&*(*)_+'1234567890-=..... Now I will test your printf: 1 + 1
= 2, 123 * 456 = 56088
0, -1, -2147483648, -1412567295, -32768, 102030
0, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
=====
Test end!!! Good luck!!!

v2_int64 = {0x0, 0x0}
xmm6      {v4_flo
```

注：左边串口数字部分（第二位）表示进程号，第一位是为了确认一直在调度刷新的标示，无实际含义

## 选做二



```
2 A0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0
5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 0 1 2 3 4 5 6 7 8 9
9 1 2 3 4 5 6 7 8 9 0 1 2 3 4 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4
5 6 1 8 9 0 1 2 3 4 0 1 2 3 4 5 6 7 7 9 0 1 2 3 4 5 6 7 8 9
0 1 2 3 8 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 4 6 7 8 9 0 1 2 3 4
0 1 2 3 4 5 5 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 6 3 4 0 1 2 3 4
5 6 7 8 9 0 1 2 2 4 5 6 7 8 9 0 1 2 3 4 0 1 2 3 3 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 4 1 2 3 4 0 1 2 3 4 5 6 7 8 9 0 2 3 4
5 6 7 8 9 0 1 2 3 4 0 1 1 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 2 9
0 1 2 3 4 0 1 2 3 4 5 6 7 8 8 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4
9 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 0 7 8 9 0 1 2 3 4 0 1 2 3 4
5 6 6 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 7 4 0 1 2 3 4 5 6 7 8 9
0 1 2 3 3 5 6 7 8 9 0 1 2 3 4 0 1 2 3 4 4 6 7 8 9 0 1 2 3 4
5 6 7 8 9 0 5 2 3 4 0 1 2 3 4 5 6 7 8 9 0 1 1 3 4 5 6 7 8 9
0 1 2 3 4 0 1 2 2 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 3 0 1 2 3 4
0 1 2 3 4 5 6 7 8 9 9 1 2 3 4 5 6 7 8 9 0 1 2 3 4 0 2 3 4
5 6 7 8 9 0 1 2 3 4 5 6 1 8 9 0 1 2 3 4 0 1 2 3 4 5 6 7 7 9
0 1 2 3 4 5 6 7 8 9 0 1 2 3 8 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4
4 6 7 8 9 0 1 2 3 4 0 1 2 3 4 5 5 7 8 9 0 1 2 3 4 5 6 7 8 9
0 1 6 3 4 0 1 2 3 4 5 6 7 8 9 0 1 2 2 4 5 6 7 8 9 0 1 2 3 4
0 1 2 3 3 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 4 1 2 3 4 0 1 2 3 4
5 6 7 8 9 0 0 0 2 3 4 5 6 7 8 9 0 1 2 3 4 0 1 1 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 2 9 0 1 2 3 4 0 1 2 3 4 5 6 7 8 8 0 1 2 3 4
5 6 7 8 9 0 1 2 3 4 9 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 0 7 8 9
0 1 2 3 4 0 1 2 3 4 5 6 6 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 7 4
0 1 2 3 4 5 6 7 8 9 0 1 2 3 3 5 6 7 8 9 0 1 2 3 4 0 1 2 3 4
4 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 5 2 3 4 0 1 2 3 4 5 6 7 8 9
0 1 1 3 4 5 6 7 8 9 0 1 2 3 4 0 1 2 2 4 5 6 7 8 9 0 1 2 3 4
5 6 7 8 3 0 1 2 3 4 0 1 2 3 4 5 6 7 8 9 9 1 2 3 4 5 6 7 8 9
0 1 2 3 4 0 0 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 1 8 9 0 1 2 3 4
0 1 2 3 4 5 6 7 7 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 8 0 1 2 3 4
5 6 7 8 9 0 1 2 3 4 4 6 7 8 9 0 1 2 3 4 0 1 2 3 4 5 5 7 8 9
0 1 2 3 4 5 6 7 8 9 0 1 6 3 4 0 1 2 3 4 5 6 7 8 9 0 1 2 2 4
5 6 7 8 9 0 1 2 3 4 0 1 2 3 3 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
4 1 2 3 4 0 1 2 3 4 5 6 7 8 9 0 0 2 3 4 5 6 7 8 9 0 1 2 3 4
0 1 1 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 2 9 0 1 2 3 4 0 1 2 3 4
5 6 7 8 8 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 9 1 2 3 4 5 6 7 8 9
C: <22/lab3origin/lab3/test.txt tex... 2432W 100% ln:2 %6152
0 0
```

注：  
这里是在 `syscallprintf` 中进行一些串口输出，在运行时用流重定向到文件 `test.txt` 中。这个结果稍后会分析。

## 关键代码实现方法

- 填补 fork, sleep 以及 exit 函数缺失的代码

```
233 pid_t fork() {
234     /*TODO:call syscall*/
235     return syscall(SYS_FORK,0,0,0,0);
236 }
237
238 int exec(uint32_t sec_start, uint32_t sec_num){
239     /*TODO:call syscall*/
240     return syscall(SYS_EXEC,sec_start,sec_num,0,0,0);
241 }
242
243 int sleep(uint32_t time) {
244     /*TODO:call syscall*/
245     return syscall(SYS_SLEEP,time,0,0,0,0);
246 }
247
248 int exit() {
249     /*TODO:call syscall*/
250     return syscall(SYS_EXIT,0,0,0,0,0);
251 }
```

- 完成时钟中断处理

```
60
61 void timerHandle(struct StackFrame *sf){
62     /*static int checkyy = 0;*/
63     //TODO check
64     for(int i = 0; i < MAX_PCB_NUM;i++) {
65         if (pcb[i].state == STATE_BLOCKED && (--pcb[i].sleepTime) == 0) {
66             pcb[i].state = STATE_RUNNABLE;
67         }
68     }
69
70     if(pcb[current].state != STATE_RUNNING || ++pcb[current].timeCount >= MAX_TIME_COUNT) {
71         //find a new process to run
72         int next = current;
73         int i = 1;
74         for(; i < MAX_PCB_NUM ;i++) {
75             int pro_i = (current + i) % MAX_PCB_NUM;
76             if(pcb[pro_i].state == STATE_RUNNABLE) {
77                 next = pro_i;
78                 break;
79             }
80         }
81         if(i == MAX_PCB_NUM) next = 0;
82
83
84
85         /*if(next != current) {*/
86             pcb[next].state = STATE_RUNNING;
87             if(pcb[current].state == STATE_RUNNING) {
88                 pcb[current].state = STATE_RUNNABLE;
89             }
90             pcb[current].timeCount = 0;
91             current = next;
92
93             /*checkyy ++;*/
94             /*checkyy %= 50;*/
95             /*putChar('0'+checkyy);putChar('0'+ current);putChar('\n');*/
96
97
98             uint32_t tmpStackTop = pcb[current].stackTop;
99             pcb[current].stackTop = pcb[current].prevStackTop;
100             tss.esp0 = (uint32_t)&(pcb[current].stackTop);
101             asm volatile("movl %0, %%esp:::n"(tmpStackTop));
102             asm volatile("popl %gs");
103             asm volatile("popl %fs");
104             asm volatile("popl %es");
105             asm volatile("popl %ds");
106             asm volatile("popal");
107             asm volatile("addl $8, %esp");
108             asm volatile("iret");
109         }
110     }
```

```

109
110
111
112             /*)*/
113
114     }
115
116             /*checkyy ++;*/
117             /*checkyy %= 50;*/
118             /*putChar('0'+checkyy);putChar('0' + current);putChar('\n');*/
119 }

```

## • 完成系统调用库函数

### syscallFork

```

206 void syscallFork(struct StackFrame *sf){
207
208     //TODO: checked 查找空闲pcb, 如果没有就返回-1
209     int i=0;
210     for(; i < MAX_PCB_NUM; i++) {
211         if(pcb[i].state == STATE_DEAD) {
212             break;
213         }
214     }
215
216     if( i == MAX_PCB_NUM) {
217         pcb[current].regs.eax = -1;
218         return; //直接返回
219     }
220
221     //TODO checked 拷贝地址空间
222     memcpy((void*)((i+1)*0x100000), (void*)((current+1)*0x100000), 0x100000);
223
224     /*enableInterrupt();*/
225     /*for (int j = 0; j < 0x100000; j++) {*/
226         /*(uint8_t*)(j + (i + 1) * 0x100000) = *(uint8_t*)(j + (current + 1) * 0x100000);*/
227         /*if(!(j%0x100))asm volatile("int $0x20"); //XXX Testing irqTimer during syscall*/
228     /*}*/
229     /*disableInterrupt();*/
230
231     //将父进程的返回值设为子进程pid
232     pcb[current].regs.eax = i;
233
234     // 拷贝pcb, 这部分代码给出了, 请注意理解
235     memcpy(&pcb[i], &pcb[current], sizeof(ProcessTable));

```

### syscallSleep

```

275 void syscallSleep(struct StackFrame *sf){
276     //TODO: checked 实现它
277     uint32_t sleeptime = sf->ecx;
278
279     if((int)sleeptime < 0 ) return;
280
281     pcb[current].state = STATE_BLOCKED;
282     pcb[current].sleepTime = sleeptime;
283     asm volatile("int $0x20");
284     return;
285 }
286 }

```



## syscallExit

```
288 void syscallExit(struct StackFrame *sf){
289     //TODO checked 先设置成dead, 然后用int 0x20进入调度
290     pcb[current].state = STATE_DEAD;
291     asm volatile("int $0x20");
292     return;
293 }
294 }
```

## syscallExec

```
254 void syscallExec(struct StackFrame *sf) {
255     // TODO 完成exec
256     // hint: 用loadelf, 已经封装好了
257     uint32_t entry = 0;
258     uint32_t secstart = sf->ecx;
259     uint32_t secnum = sf->edx;
260     int ret = loadelf(secstart, secnum, (current+1)*0x100000, &entry);
261     if(ret == 0) {
262         sf->eip = entry;
263     }
264     else {
265         sf->eax = -1;
266     }
267 }
268
269 return;
```

- 选做一：即按讲义添加代码
- 选做二：测试程序为

```
35 int uEntry(void) {
36     fork();
37     while(1) printf("\n\n\n\n\n\n\n");
38 }
39 }
```

## 实验思考题

1. linux 下进程的创建及运行有两个命令 fork 和 exec，请说明他们的区别？

区别：程序是个状态机，操作系统管理着状态机的集合。linux 中 fork 通过复制一个进程来创建一个新进程，就是把一个状态机的所有状态复制一遍，产生一个一模一样的进程。linux 中 execve 有三个参数：const char\* pathname, char \*const argv[], char \*const envp[]，功能是将 pathname 索引的程序运行起来。状态机有个初始状态，然后会按照一条条指令往下跑，execve 是将状态机重置到初始状态。

下面来分析下这次实验中 `syscallFork` 和 `syscallExec` :  
`syscallFork` 主要做的事情是:

```
//TODO checked拷贝地址空间  
memcpy((void*)((i+1)*0x100000),(void*)((current+1)*0x100000),0x100000);
```

```
// 拷贝pcb, 这部分代码给出了, 请注意理解  
memcpy(&pcb[i],&pcb[current],sizeof(ProcessTable));
```

 (下面代码较长, 省略)

产生一个完全地拷贝这个状态机的状态的新状态机

`syscallExec` 主要做的事情是:

```
uint32_t entry = 0;  
uint32_t secstart = sf->ecx;  
uint32_t secnum = sf->edx;  
int ret = loadelf(secstart,secnum,(current+1)*0x100000,&entry);  
if(ret == 0) {  
    sf->eip = entry;  
}  
  
else {  
    sf->eax = -1;  
}
```

加载一个 elf, 然后将 `eip` 指向 elf 的第一条指令, 就是将状态机的状态设置为初始的状态。

## 2. 请在实验报告中简要说明你对 `fork/exec/wait/exit` 函数的分析,并分析 `fork/exec/wait/exit` 在实现中是如何影响进程的执行状态的?

**fork 函数:** 拷贝当前进程来生成新进程。具体行为: 找到空闲 `pcb`, 将父进程的状态 (内存, 寄存器, 栈帧等) 复制给子进程, 为子进程父进程设置返回值等等。

分析进程的执行状态影响: 对进程执行状态有初始化, 比如将状态设置为就绪态。

**exec 函数:** 将状态机的状态 (是程序状态, 不是指进程执行状态) 置为初始状态。具体行为: 加载 elf, 将进程 `eip` 指向此 elf 第一条指令。

分析进程的执行状态影响: 我认为没有太大影响

**wait 函数** (应该是指这里的 `sleep` 函数): 将此进程执行状态设为 `blocked`, 设置睡眠时间, 陷入时间中断来调度进程。

分析进程的执行状态影响: 改变进程状态为 `block`, 让之后调度需要一段时间才会再调到这个进程

**exit 函数:** 将此进程执行状态设为 `dead`, 陷入时间中断来调度进程。

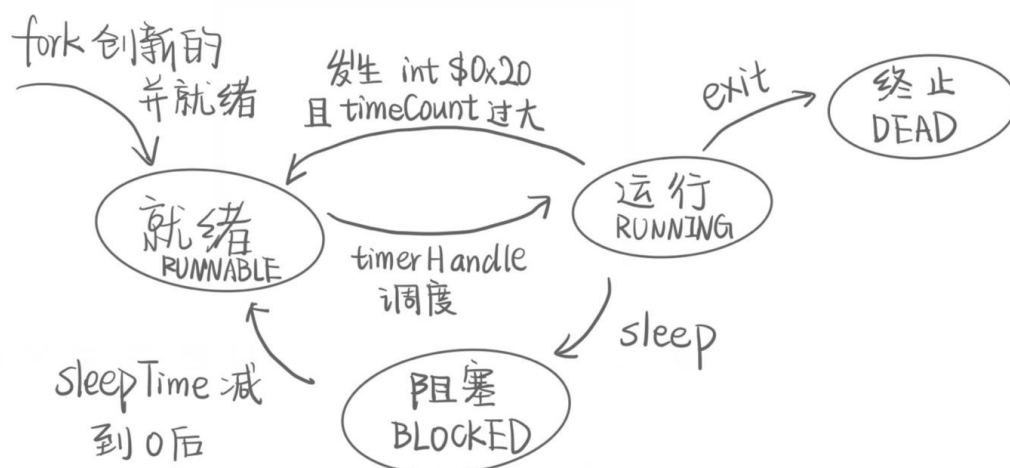
分析进程的执行状态影响: 改变进程状态为 `dead`, 之后再也不会调到这个进程

3. 请在实验报告中描述当创建一个用户态进程并加载了应用程序后，CPU 是如何让这个应用程序最终在用户态执行起来的。即这个用户态进程被 OS 选择占用 CPU 执行（RUNNING 态）到具体执行应用程序第一条指令的整个经过。

首先创建一个 fork 创建一个和用户态进程一致的新进程，用 `excve` 加载应用程序，将状态机变成初始状态，此时此进程 `eip` 指向应用程序第一条指令。

当中断 `int $20` 来的时候，跳转到 `ieqTimer`, `irqsyscall`, `irqHandle`, `irqHandle` 函数。`irqHandle` 中有一个非常有趣的是：用 `tmpStackTop` 存 `pcb[current].stackTop` (相当于把这个栈指针存到了系统栈上，方便之后如果没有 `iret` 直接跳用户态时恢复当前状态)。将 `pcb[current].prevStackTop` 中的值也赋值为 `pcb[current].stackTop`, `pcb[current].stackTop=sf`，然后跳转到 `timerHandle` 进行处理。在这个时候，由简单的推理与归纳可以得知，所有的 `pcb[i].stackTop` 都是一个 `StackFrame` 的指针，`StackFrame` 的指针中包含所有寄存器信息，由设计的特殊性根据 `pid` 我们也可以找到内存信息。也就是说操作系统掌握了各个进程状态机的所有状态。在 `timerHandle` 中，进行进程调度。如果选择了该用户程序，即准备将 `current` 选为该用户程序，操作系统接下来要做的就是将这个状态机放到 `cpu` 上去跑。实现上，就是将存在 `stackTop` 指向的（状态机状态）寄存器等内容放到各个寄存器中，然后 `iret` 直接跳回用户程序执行（不从 `irqHandle` 那些返回了）。接下来，将执行这个用户态程序的第一条指令。

4. 请给出一个用户态进程的执行状态生命周期图（包执行状态，执行状态之间的变换关系，以及产生变换的事件或函数调用）



## 实验结果分析

### 选做一

能跑但是跑的很慢。将 j 输出频率放到 1/0x100 在大约 2 分钟后开始输出结果。通过串口输出观察到确实一直在调度不同进程。我认为这是性能原因（可能是调度选择机制，调度到用户的比较少）。

```
enableInterrupt();
for (int j = 0; j < 0x100000; j++) {
    *(uint8_t *) (j + (i + 1) * 0x100000) = *(uint8_t *) (j + (current + 1) * 0x100000);
    if (!(j % 0x100)) asm volatile("int $0x20"); //XXX Testing irq
    //Timer during syscall
}
disableInterrupt();
```

（这个等两分钟是可以跑出来的）

### 选做二

用户测试代码为：

```
35 int uEntry(void) {
36     fork();
37     while(1) printf("\n\n\n\n\n\n\n");
38
39 }
```

同时修改了一些 syscallPrint 中的内容：

```
for (i = 0; i < size; i++) {
    asm volatile("movb %%es:(%1), %0":"=r"(character):"r"(str + i));
    if (character == '\n') {
        int tmp_dis = displayRow % 10;
        asm volatile("int $0x20");
        putchar('0' + tmp_dis); putchar(' ');

        displayRow++;
        displayCol = 0;
        if (displayRow == 25) {
            displayRow = 0;
        }
    }
}
```

主要修改：1. 添加 putchar 输出，输出结果是 displayRow % 10 的数值。2. 在获得 tmp\_dis 后模拟中断到来，陷入时钟中断。3. 当 displayRow 到达 25 的时候，改为 0 而不是 24。

接下来，用流重定向将串口输出内容重定向到 txt 文件中。从理论上说获得的会



是 01234567890123456789012345 这样循环。但是用 vim 搜索会得到：连续输出两个 0，没有 1 直接 234 这类情况。两个进程在对 displayRow 的访问有竞争，产生了一些两边访问到的没有同步（一致）这样的问题。

```

2 A0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 0 1 2 3 4
5 6 7 8 9 0 1 2 3 4 5 6 7 8 3 0 1 2 3 4 0 1 2 3 4 5 6 7 8 9
9 1 2 3 4 5 6 7 8 9 0 1 2 3 4 0 0 2 3 4 5 6 7 8 9 0 1 2 3 4
5 6 1 8 9 0 1 2 3 4 0 1 2 3 4 5 6 7 7 9 0 1 2 3 4 5 6 7 8 9
0 1 2 3 8 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 4 6 7 8 9 0 1 2 3 4
0 1 2 3 4 5 5 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 6 3 4 0 1 2 3 4
5 6 7 8 9 0 1 2 2 4 5 6 7 8 9 0 1 2 3 4 0 1 2 3 3 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9 4 1 2 3 4 0 1 2 3 4 5 6 7 8 9 0 0 2 3 4
5 6 7 8 9 0 1 2 3 4 0 1 1 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 2 9
0 1 2 3 4 0 1 2 3 4 5 6 7 8 8 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4
9 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 0 7 8 9 0 1 2 3 4 0 1 2 3 4
5 6 6 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 7 4 0 1 2 3 4 5 6 7 8 9
0 1 2 3 3 5 6 7 8 9 0 1 2 3 4 0 1 2 3 4 4 6 7 8 9 0 1 2 3 4
5 6 7 8 9 0 5 2 3 4 0 1 2 3 4 5 6 7 8 9 0 1 1 3 4 5 6 7 8 9
0 1 2 3 4 0 1 2 2 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 3 0 1 2 3 4
0 1 2 3 4 5 6 7 8 9 9 1 2 3 4 5 6 7 8 9 0 1 2 3 4 0 2 3 4
5 6 7 8 9 0 1 2 3 4 5 6 1 8 9 0 1 2 3 4 0 1 2 3 4 5 6 7 7 9
0 1 2 3 4 5 6 7 8 9 0 1 2 3 8 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4
4 6 7 8 9 0 1 2 3 4 0 1 2 3 4 5 5 7 8 9 0 1 2 3 4 5 6 7 8 9
0 1 6 3 4 0 1 2 3 4 5 6 7 8 9 0 1 2 2 4 5 6 7 8 9 0 1 2 3 4
0 1 2 3 3 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 4 1 2 3 4 0 1 2 3 4
5 6 7 8 9 0 0 2 3 4 5 6 7 8 9 0 1 2 3 4 0 1 1 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 2 9 0 1 2 3 4 0 1 2 3 4 5 6 7 8 8 0 1 2 3 4
5 6 7 8 9 0 1 2 3 4 9 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 0 7 8 9
0 1 2 3 4 0 1 2 3 4 5 6 6 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 7 4
0 1 2 3 4 5 6 7 8 9 0 1 2 3 3 5 6 7 8 9 0 1 2 3 4 0 1 2 3 4
4 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 5 2 3 4 0 1 2 3 4 5 6 7 8 9
0 1 1 3 4 5 6 7 8 9 0 1 2 3 4 0 1 2 2 4 5 6 7 8 9 0 1 2 3 4
5 6 7 8 3 0 1 2 3 4 0 1 2 3 4 5 6 7 8 9 9 1 2 3 4 5 6 7 8 9
0 1 2 3 4 0 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 1 8 9 0 1 2 3 4

```

## 实验思考

### 1. 一开始 bootloader 加载不了 kernel 的 elf 的 bug

问题出在将操作系统拷贝到内存后，认为读取开始的内容是 elfHeader，以 elfHeader 结构读取 elf 指针指向（即 0x100000 处）地址。用 gdb 查看，发现：似乎这里是一段代码而不是 elfheader

```

boot.c
B+ 31      for (i = 0; i < 200; i++) {
32          readSect((void*)(elf + i*512), 1
33      }
34
35      kMainEntry = (void*)(void)((struct ELF
>36      phoff = ((struct ELFHeader *)elf)->phoff
37      phnum = ((struct ELFHeader *)elf)->phnum
38
39      for(i = 0; i < phnum; i++){
40          thisph = ((struct ProgramHeader

```

```
remote Thread 1.1 In: bootMain L36 PC: 0x7d71
Undefined info command: "elf". Try "help info".
(gdb) x/10i elf
0x100000 <initTimer>:    jg     0x100047 <initSerial+47>
0x100002 <initTimer+2>:  dec     %esp
0x100003 <initTimer+3>:  inc     %esi
0x100004 <initTimer+4>:  add     %eax,%ecx
0x100006 <initTimer+6>:  add     %eax,%eax
0x100008 <initTimer+8>:  add     %al,%eax
0x10000a <initTimer+10>: add     %al,%eax
0x10000c <initTimer+12>: add     %al,%eax
0x10000e <initTimer+14>: add     %al,%eax
0x100010 <initTimer+16>: add     (%eax),%al
(gdb)
```

所以，之后继续执行的结果就是：会跳到一段全 0 的区域

```
(gdb) x/10i kMainEntry
0x100f73 <kEntry>:    add     %al,%eax
0x100f75 <kEntry+2>:  add     %al,%eax
0x100f77 <kEntry+4>:  add     %al,%eax
0x100f79 <kEntry+6>:  add     %al,%eax
0x100f7b <kEntry+8>:  add     %al,%eax
0x100f7d <kEntry+10>: add     %al,%eax
0x100f7f <kEntry+12>: add     %al,%eax
0x100f81 <kEntry+14>: add     %al,%eax
0x100f83 <kEntry+16>: add     %al,%eax
0x100f85 <kEntry+18>: add     %al,%eax
(gdb)
```

我怀疑是不是这个 elfheader 根本就不在最开始。这个问题最终的解决方案是群里史浩宇同学提出的在 LDFlags 中增加选项，这次读取 elfheader 是正常的。这对我有一些启示：1. 防御性编程的重要性：如果有读取的时候进行一个魔数检查，或许这个问题就不会这么莫名其妙了 2. 怎么样去读手册？在试图解决问题的时候，我觉得或许调节编译选项可以有用，但是我根本找不到，看不懂...这个问题值得我一直思考。

## 2. 关于嵌套中断：

在实验中，嵌套中断究竟是怎么进行的呢？为什么代码说 prevStackTop 能支持嵌套中断呢？首先，我在思考中断什么时候是打开的。关于 int 指令：阅读 i386 手册发现会 int 自动关中断。关于 iret 指令：原文说是"EFLAGS <- SS:[eSP + 8](Sets VM in interrupted routine)"。我没有太看懂，感觉是开中断的意思。看到这里似乎对 prevStackTop 的存在有些解释：为什么是一个 prevStackTop 呢？为什么没有两个,三个呢？我觉得是因为即使在 syscallFork 中打开了中断，在 OS 的 syscallFork 中再引起中断的肯定不是 syscallFork,而在其他中断里，中断会自动关中断且在这个过程中不会打开，所有中断最多两层，有一个 prev 就够了。这样就可以运行最多两层的嵌套中断。