

实验报告——实验六

姓名：任文頔

学号：14322181

院系：数据科学与计算机专业

专业、年级：14级计算机科学与技术

指导教师：凌应标

【实验题目】

二状态的进程模型

【实验目的】

- 1、学习进程模型的知识
- 2、初步学习进程表的创建
- 3、学习时间片轮转法实现进程调度
- 4、了解进程交替执行原理，重点学会实现save和restart过程

【实验要求】

保留原型原有特征的基础上，设计满足下列要求的新原型操作系统：

1. 在c程序中定义进程表，进程数量为4或更多。
2. 修改内核，可通过用户命令选择加载1~4个用户程序运行，采用时间片轮转调度进程运行，用户程序的输出各占1/4屏幕区域，信息输出有动感，以便观察程序是否在执行。
3. 在原型中保证原有的系统调用服务可用。再编写1个用户程序，展示你的所有系统调用服务还能工作。
4. 建议：以MINIX的save()和restart()为参考，进行修改，并为你的原型中使用

【实验方案】

1 硬件或虚拟机配置方法

系统环境：Linux Ubuntu 14.04

虚拟机配置方法：在Linux下VMware Player是收费软件，因此选择免费的VirtualBox软件。

配置方法：操作系统选择其他，选择从软盘启动，添加自己的软盘，虚拟机名称为14322181renwendi

2 软件工具与作用

- (1) 汇编语言编译器NASM：针对Intel x86架构的汇编与反汇编程序
- (2) C语言编译器GCC：由 GNU 开发的编程语言编译器。
- (3) 编辑器Vim：功能强大、高度可定制的文本编辑器。
- (4) 虚拟机软件Bochs：自己编写的操作系统的测试环境
- (5) 软盘创建工具:dd 软盘写入工具:Linux 自带挂载命令

3 方案的思想

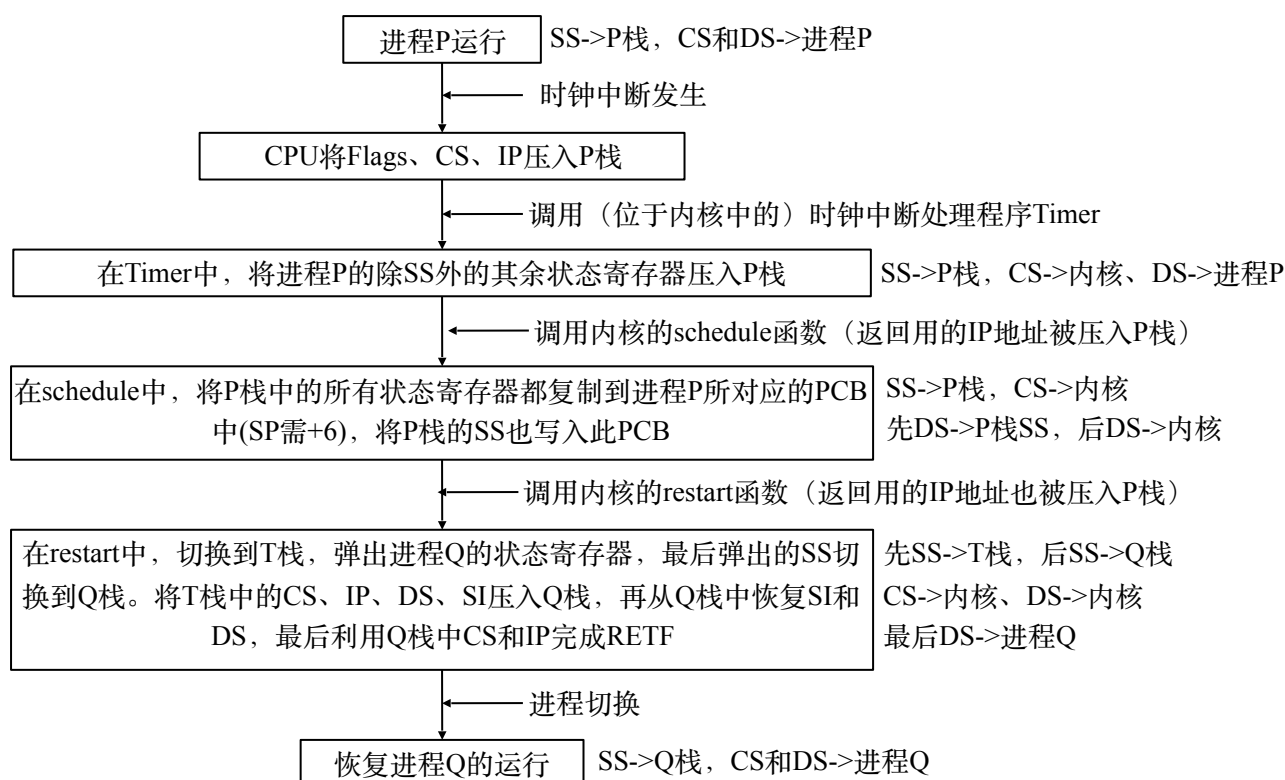
在实验五的原型基础上，保留实验五的所有功能，在内核中新添加一个procecss.c的文件来实现进程表的创建和进程调度，在内核中的其余文件中也要做出相应的修改。此外，编写5个进程，其中前四个是在实验五的基础上对屏幕1/4处显示字符的代码进行修改，使其可以变化颜色和字符，第五个进程是监听用户的键盘输入，如果有按键则退出。最后保留了两个之前实验的用户程序，一个是简单的显示字符，另一个是同时显示4个中断的系统调用，并保留了按键后“ouch”的显示。

4 相关知识原理

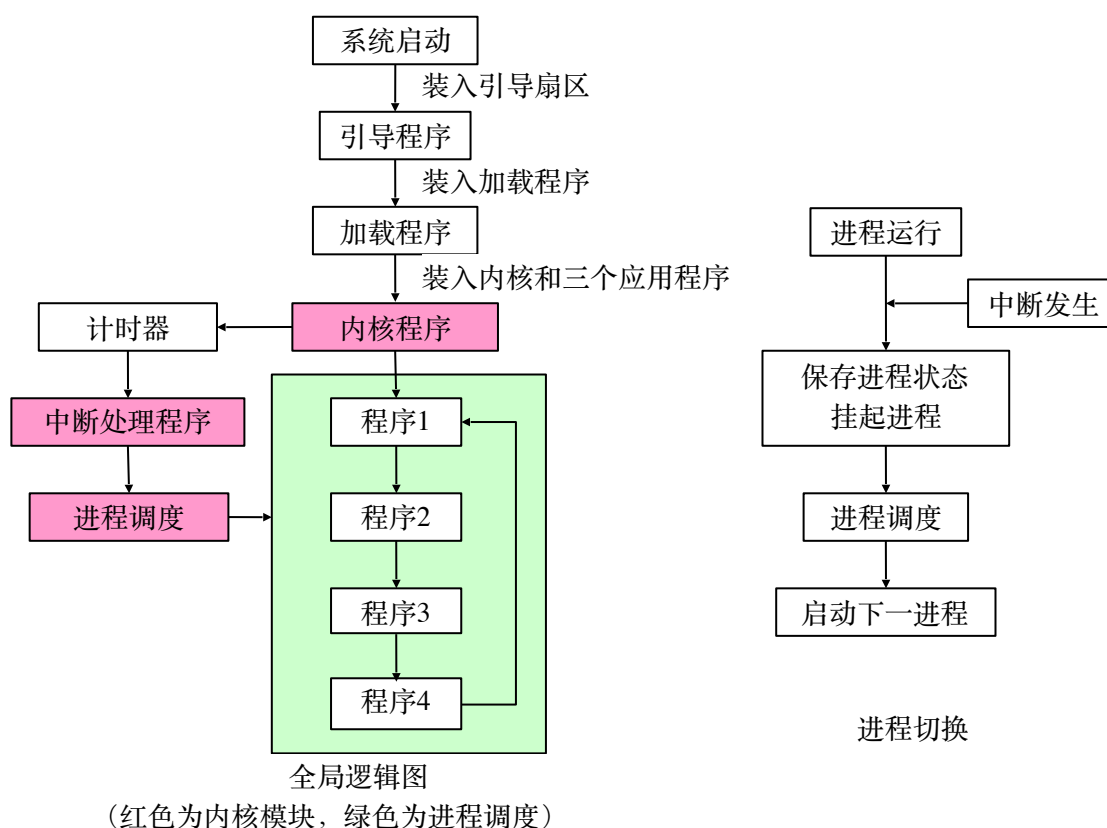
(1) 进程调度与栈堆切换

因为恢复上下文寄存器，和栈切换，以及进程切换的准备操作，是交错进行的，所以程序的逻辑有些复杂，我们先尽可能多地恢复寄存器的值，再进行栈切换。但是将还需要使用到的DS先不恢复，而是在栈切换完成后，先压入新进程栈中，等以后切换进程前再弹出。IP和CS也从PCB中取出后压入新进程的栈中。在这些操作中还需使用一个寄存器，我们用的是SI，也需要先从PCB中取出压入新进程栈中，等最后一刻才恢复。

进程调度与堆栈切换的逻辑框图：



5 程序流程



6 程序关键模块

(1) 引导程序boot.asm

与实验五稍有区别在于，调用13h中断的02h将内核从特定扇区载入内存并调到相应内存地址执行时，内核占用扇区数进行了改变。实验五是在同一个柱面的18个扇区上，内存足够储存所有的内核程序和用户程序。而本次实验的内核程序已经几乎要占满一个柱面了，因此将除了引导程序占用的一个扇区之外的所有17个扇区均分配给内核。

```
28 LoadnEx:
29   xor ax,ax
30   mov es,ax
31   mov bx, OffSetOfUserPrg
32   mov ax, 0235h
33   mov dl,0
34   mov dh,0
35   mov ch,0
36   mov cl,2
37   int 13H
38   jmp OffSetOfUserPrg
```

在vim中用vim -v disk.img指令打开软盘，再用%!xxd指令转换成十六进制机器码显示软盘映像文件如下图，可以看到内核已经占用到了第16个扇区。

```
2. vim -v disk.img (vim)
480 0001df0: 1c00 0000 bc01 0000 dbf4 ffff e600 0000 .....
481 0001e00: 0042 0e08 8502 430d 0502 dfc5 0c04 0400 .B....C.....
482 0001e10: 1400 0000 0000 0000 017a 5200 017c 0801 .....zR..l..
483 0001e20: 1b0c 0404 8801 0000 1c00 0000 1c00 0000 .....
484 0001e30: 89f5 ffff 0a00 0000 0042 0e08 8502 430d .....B....C.
485 0001e40: 0543 c50c 0404 0000 1c00 0000 3c00 0000 .C.....<...
486 0001e50: 73f5 ffff 0a00 0000 0042 0e08 8502 430d s.....B....C.
487 0001e60: 0543 c50c 0404 0000 1c00 0000 5c00 0000 .C.....\...
488 0001e70: 5df5 ffff 2c00 0000 0042 0e08 8502 430d ]...B....C.
489 0001e80: 0565 c50c 0404 0000 1c00 0000 7c00 0000 .e.....l...
490 0001e90: 69f5 ffff 7b01 0000 0042 0e08 8502 430d i...{...B....C.
491 0001ea0: 0503 7401 c50c 0404 1c00 0000 9c00 0000 ..t.....
492 0001eb0: c4f6 ffff 4b01 0000 0042 0e08 8502 430d ....K....B....C.
493 0001ec0: 0503 4401 c50c 0404 1c00 0000 bc00 0000 ..D.....
494 0001ed0: eff7 ffff ab01 0000 0042 0e08 8502 430d .....B....C.
495 0001ee0: 0503 a401 c50c 0404 1800 0000 dc00 0000 .....
496 0001ef0: 7af9 ffff 7b01 0000 0042 0e08 8502 430d z...{...B....C.
497 0001f00: 0500 0000 1c00 0000 f800 0000 d9fa ffff .....
498 0001f10: ea00 0000 0042 0e08 8502 430d 0502 e3c5 .....B....C....
499 0001f20: 0c04 0400 0000 0000 0000 0000 0000 0000 .....
500 0001f30: 0000 0000 0000 0000 0000 0000 0000 0000 .....
501 0001f40: 0000 0000 0000 0000 0000 0000 0000 0000 .....
502 0001f50: 0000 0000 0000 0000 0000 0000 0000 0000 .....
503 0001f60: 0000 0000 0000 0000 0000 0000 0000 0000 .....
504 0001f70: 0000 0000 0000 0000 0000 0000 0000 0000 .....
505 0001f80: 0000 0000 0000 0000 0000 0000 0000 0000 .....
506 0001f90: 0000 0000 0000 0000 0000 0000 0000 0000 .....
507 0001fa0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
disk.img [+]
```

(2) 内核c语言部分os.c

主函数代码:

```
28 void main(){
129  //-----init-----
130  screen_init();
131  __asm__("pop %si");
132  interrupt_init();
133  __asm__("pop %si");
134  print_welcome_msg();
135  __asm__("pop %si");
136  print_message();
137  __asm__("pop %si");
138  print_flag();
139  __asm__("pop %si");
140  ///-----init end-----
```

其余部分均与实验五功能相同，主要就是实现命令行的控制，区别在于此次实验需要在栈堆之间进行切换，因此保持栈中无冗余数据是必要的。由于每次调用汇编函数c都会向栈里面压两个字，但ret返回的时候却只从栈里面弹出一个字，所以每次调用汇编函数都要手动使用内联汇编pop出来一个字。

(3) 内核汇编语言部分os.asm

设置时钟中断

```
21  mov ax,0x1c
22  mov [ interrupt_num], ax
23  mov ax, timer_interrupt_process
24  mov [ interrupt_vector_offset],ax
25  call insert_interrupt_vector
```

```
59 timer_interrupt_process:
60  push ax
61  mov ax,0
62  mov ds,ax
63  mov byte al,[ isProcessRun]
64  mov ah,0
65  cmp al,ah
66  je print_corner
67  pop ax
68  jmp schedule
```

时钟中断处理程序，通过判断 isProcessRun 是否为0来决定该执行进程调度程序，还是执行右下角循环显示的字母ABC时钟中断程序（print_corner）

通过PTR每秒发出18.2次的信号来从8592芯片的RT0 引脚发出终端号int 08h

来触发的用户时钟软中断 int 1ch, 四个进程均分时间片 $T = 1s/18.2 = 0.05494S = 54.94ms$ 。

(4) 本实验核心要求：进程调度的实现（结合process.c和oslib.asm来实现）

A. 在c语言中描述PCB

```
1 __asm__(".code16gcc");
2
3 struct Tss{
4   short int SS;
5   short int GS;
6   short int FS;
7   short int ES;
8   short int DS;
9   short int CS;
10  short int DI;
11  short int SI;
12  short int BP;
13  short int SP;
```

```

14  short int DX;
15  short int CX;
16  short int BX;
17  short int AX;
18  int IP;
19  short int Flags;
20 };
21
22 struct pcb{
23     struct Tss tss;
24     int process_status; // 0 is ready , 1 is runing
25     int process_id;
26 };

```

PCB的初始化:

```

112 void init_pcb( int i, short int current_process_SEG){
113     //tss
114     PCB_queue[ i].tss.DS = 0;
115     PCB_queue[ i].tss.ES = current_process_SEG;
116     PCB_queue[ i].tss.FS = 0;
117     PCB_queue[ i].tss.CS = 0;
118     PCB_queue[ i].tss.SS = 0;
119     PCB_queue[ i].tss.GS = 0;
120     PCB_queue[ i].tss.DI = 0;
121     PCB_queue[ i].tss.SI = 0;
122     PCB_queue[ i].tss.SP = current_process_SEG-4;
123     PCB_queue[ i].tss.BP = 0;
124     PCB_queue[ i].tss.AX = 0;
125     PCB_queue[ i].tss.BX = 0;
126     PCB_queue[ i].tss.CX = 0;
127     PCB_queue[ i].tss.DX = 0;
128     PCB_queue[ i].tss.IP = current_process_SEG;    //cs;ip
129     PCB_queue[ i].tss.Flags = 512;
130     PCB_queue[ i].process_id = i;
131     PCB_queue[ i].process_status = READY;
132 }

```

B. 定义常量

```

48 #define process_num_MAX 5
49 #define process_SEG 0
50 #define READY 0
51 #define RUNNING 1

```

C. 定义全局变量

用于保存通用寄存器，然后再由这些全局变量保存到进程控制模块队列的pcb结构体中。

```

57 struct pcb PCB_queue[ process_num_MAX +1 ];
58
59
60 short int w_is_r;    //which process is running

```

```

61 short int nw_is_r; //which next process will run
62 short int _cs,_flags;
63 int _ip;
64 short int _ax,_bx,_cx,_dx,_es,_ds,_sp,_bp,_si,_di,_fs,_gs,_ss;

```

D. 进程交替执行的实现

换老进程下来的时候首先保存的是通用寄存器到这个进程的上下文 TSS。具体实现就是先把 ax、bx等通用寄存器保存到c语言中定义的全局变量中，然后再由全局变量保存到进程控制模块队列的pcb结构体中。最后保存的是 IP、CS、Flags、SP。前三个直接从当前栈中直接pop出来即可，因为cpu在发生中断的时候已经自动把这三个push到栈中了。最后再保存sp、栈指针寄存器。

接下来根据 di寄存器的值来判断是否该结束调度程序回到主函数控制模块，如果不返回则继续换上新的进程。首先还原的是 sp、ip、cs、flags，后三个直接 push到栈里面就可以了，与pop指令一样，iret的时候会自动取出来并跳转到 cs:ip位置执行。最后跳转到 schedule_end中执行iret结束本次调度。

```

135 void schedule(){
136     saveall_reg();           //note: not inclue sp
137     __asm__("pop %cx");
138     __asm__("pop %eax");
139
140     nw_is_r = w_is_r + 1;
141     if( nw_is_r > process_num_MAX){
142         nw_is_r = RUNNING;
143     }
144
145
146     saveToqueue();
147
148
149     //-----set ip cs flag-----
150     __asm__("pop %ax");
151     __asm__("pop %bx");
152     __asm__("pop %cx");
153
154     saveall_reg_seg();       //include sp
155     __asm__("pop %cx");
156
157     if( _di == 0x1234){
158         isProcessRun = 0;           //shut down process
159         nw_is_r = READY;
160         backto_os();
161     }else{
162         isProcessRun = 1;
163     }
164
165
166     PCB_queue[ w_is_r ].tss.SP = _sp;

```

```

167 PCB_queue[w_is_r].tss.IP = _ip;
168 PCB_queue[w_is_r].tss.CS = _cs;
169 PCB_queue[w_is_r].tss.Flags = _flags;
170
171 //-----end-----
172 _ip = PCB_queue[nw_is_r].tss.IP;
173 _cs = PCB_queue[nw_is_r].tss.CS;
174 _flags = PCB_queue[nw_is_r].tss.Flags;
175 _sp = PCB_queue[nw_is_r].tss.SP;
176
177 restore_reg_seg();
178 __asm__("pop %cx");
179
180
181 queueTodata();
182
183 w_is_r++;
184 if( w_is_r > process_num_MAX){
185     w_is_r = RUNNING;
186 }
187
188 restore_reg_seg();
189 __asm__("pop %di");
190
191 __asm__("jmp schedule_end");
192 while(1);
193 }

```

辅助函数：

```

66 inline void backto_os(){
67     init_flag_position();
68     __asm__("pop %si");
69     screen_init();
70     __asm__("pop %si");
71     print_welcome_msg();
72     __asm__("pop %si");
73     print_message();
74     __asm__("pop %si");
75     print_flag();
76     __asm__("pop %si");
77 }

```

注：返回操作系统主控制模块，仍然要保持栈中无冗余数据。

```

79 inline void saveToqueue(){
80     PCB_queue[w_is_r].tss.ES = _es;
81     PCB_queue[w_is_r].tss.DS = _ds;
82     PCB_queue[w_is_r].tss.GS = _gs;
83     PCB_queue[w_is_r].tss.FS = _fs;
84     PCB_queue[w_is_r].tss.SS = _ss;
85
86     PCB_queue[w_is_r].tss.AX = _ax;
87     PCB_queue[w_is_r].tss.BX = _bx;
88     PCB_queue[w_is_r].tss.CX = _cx;

```



```

89  PCB_queue[ w_is_r].tss.DX = _dx;
90  PCB_queue[ w_is_r].tss.SI = _si;
91  PCB_queue[ w_is_r].tss.DI = _di;
92  PCB_queue[ w_is_r].tss.BP = _bp;
93 }

95 inline void queueTodata(){
96  _es = PCB_queue[ nw_is_r].tss.ES;
97  _ds = PCB_queue[ nw_is_r].tss.DS;
98  _gs = PCB_queue[ nw_is_r].tss.GS;
99  _fs = PCB_queue[ nw_is_r].tss.FS;
100  _ss = PCB_queue[ nw_is_r].tss.SS;
101
102  _ax = PCB_queue[ nw_is_r].tss.AX;
103  _bx = PCB_queue[ nw_is_r].tss.BX;
104  _cx = PCB_queue[ nw_is_r].tss.CX;
105  _dx = PCB_queue[ nw_is_r].tss.DX;
106  _si = PCB_queue[ nw_is_r].tss.SI;
107  _di = PCB_queue[ nw_is_r].tss.DI;
108  _bp = PCB_queue[ nw_is_r].tss.BP;
109 }

```

注：全局变量实际上就相当于寄存器和pcb之间的一个缓存，保存的时候将全局变量的数据保存到pcb中，还原的时候倒过来赋值。

```

70 schedule_end: (os.asm中实现，是汇编代码)
71  push ax
72  mov al, 20h
73  out 20h,al
74  out 0A0h,al
75  pop ax
76  sti
77  iret

```

E. save过程 (oslib.asm中实现)

```

218 extern _cs,_flags,_ip
219 saveall_reg_seg:
220  mov [ _ip],ax
221  mov [ _cs],bx
222  mov [ _flags],cx
223  pop si
224  pop di
225  mov [ _sp],sp
226  push di
227  push si
228  ret

248 extern _ax,_bx,_cx,_dx,_es,_ds,_sp,_bp,_si,_di,_fs,_gs,_ss
249 saveall_reg:
250  mov [ _es],es
251  mov [ _ds],ds
252  mov [ _gs],gs
253  mov [ _fs],fs

```

```

254  mov [_ss],ss
255
256  mov [_ax],ax
257  mov [_bx],bx
258  mov [_cx],cx
259  mov [_dx],dx
260  mov [_di],di
261  mov [_si],si
262  mov [_bp],bp
263 ret

```

保存寄存器数据到全局变量和切换进程栈。为了及时保护中断现场，必须在中断处理函数的最开始处，立即保存被中断程序的所有上下文寄存器中的当前值。

F. restart过程 (oslib.asm中实现)

```

230 restart_reg_seg:
231  mov ax,[_ip]
232  mov bx,[_cs]
233  mov cx,[_flags]
234
235  pop si      ;ret
236  pop di
237
238  mov sp,[_sp]
239
240  push cx     ;flags
241  push bx     ;cs
242  push ax     ;ip
243
244  push di
245  push si
246 ret

```

```

265 restart_reg:
266
267  mov bp,[_bp]
268
269  mov es,[_es]
270  mov ds,[_ds]
271  mov gs,[_gs]
272  mov fs,[_fs]
273  mov ss,[_ss]
274
275  mov ax,[_ax]
276  mov bx,[_bx]
277  mov cx,[_cx]
278  mov dx,[_dx]
279  mov di,[_di]
280  mov si,[_si]
281 ret

```

还原 ip、cs、flags 寄存器数据和切换进程栈。

使用 IRET 指令，在用户进程的栈中保存 IP、CS 和 FLAGS，但必须将 IP、CS 和 FLAGS 放回用户进程栈中。

G. 将进程加载到内存中执行

```
196 void Process(){
197     int current_process_SEG = process_SEG;
198     int i;
199     for( i = start_process_num; i <= process_num; i++){
200         current_process_SEG = i*0x0800;
201         init_pcb( i, current_process_SEG);
202     }
203
204     load_user( 1, 0x0800);
205     __asm__(" pop %cx");
206     load_user( 2, 0x1000);
207     __asm__(" pop %cx");
208     load_user( 3, 0x1800);
209     __asm__(" pop %cx");
210     load_user( 4, 0x2000);
211     __asm__(" pop %cx");
212     load_user( 5, 0x2800);    //wait key
213     __asm__(" pop %cx");
214     // 4000- sub stack
215
216     w_is_r = READY;
217     isProcessRun=1; // enter user process mode
218 }
```

(5) 5个用户进程

前四个分别在屏幕1/4处循环画框显示不同的字符，并不断变化颜色。此外，在画框中间显示姓名。下面以其中一个为例：

process1.asm:

```
1 org 0x1000
2 sti
3
4 mov ax,0xb800
5 mov es,ax
6
7 mov al,'A'
8 mov dl,00010011B
9 again:
10  mov bx,180D    ;stat
11  mov cx,220D    ;stop
12
13 loop_int33_h0:
14  mov byte [es:bx],al
15  inc bx
```

```

16  mov byte [es:bx],dl;font color
17  inc bx
18  inc dl
19  call delay
20  cmp bx,cx
21  jle loop_int33_h0
22
23  ;h1 direction
24  mov bx,220D    ;stat
25  mov cx,1820D   ;stop
26
27 loop_int33_h1:
28  mov byte [es:bx],al
29  inc bx
30  mov byte [es:bx],dl;font color
31  inc dl
32  add bx, 159D
33  call delay
34  cmp bx,cx
35  jle loop_int33_h1
36
37  ;h2 direction
38  mov bx,1820D   ;strat
39  mov cx,1780D   ;end
40
41 loop_int33_h2:
42  mov byte [es:bx],al
43  dec bx
44  mov byte [es:bx],dl
45  inc dl
46  dec bx
47  call delay
48  cmp bx,cx
49  ja loop_int33_h2
50
51  ;h3 direction
52  mov bx,1780D   ;start
53  mov cx,180D    ;end
54
55 loop_int33_h3:
56  mov byte[es:bx],al
57  inc bx
58  mov byte[es:bx],dl
59  inc dl
60  sub bx,161D
61  call delay
62  cmp bx,cx
63  ja loop_int33_h3
64
65
66 mov byte [es:998],'R'
67 call delay
68 call delay

```

```

69 mov byte [es:1000], 'E'
70 call delay
71 call delay
72 mov byte [es:1002], 'N'
73 call delay
74 call delay
75 call delay
76 call delay
77
79 inc al
80 cmp al, 'Z'
81 jle again
82 mov al, 'A'
83 jmp again
84
86 jmp $
91
92 delay:
93 push dx
94 push cx
95 mov si, 8000D
96 mov dx, 00
97 timer2:
98 mov cx, 00
99 timer:
100 inc cx
101 cmp cx, si
102 jne timer
103 inc dx
104 cmp dx, si
105 jne timer2
106 pop cx
107 pop dx
108 ret
109
111 times 512-($-$$) db 0 ;填充剩余扇区0

```

最后一个进程监听用户按键，与其他四个进程拥有同样长的时间片,也就是说在进程调度动画执行的任何时候都可以随便按一个键返回主函数控制模块。

process5.asm:

```

1 org 0x5000
2 sti
3
4 mov ah, 0
5 int 16h
6
7
8 mov di, 0x1234
9
10 jmp $

```

(6) 2个用户程序 (use1.asm, use2.asm)

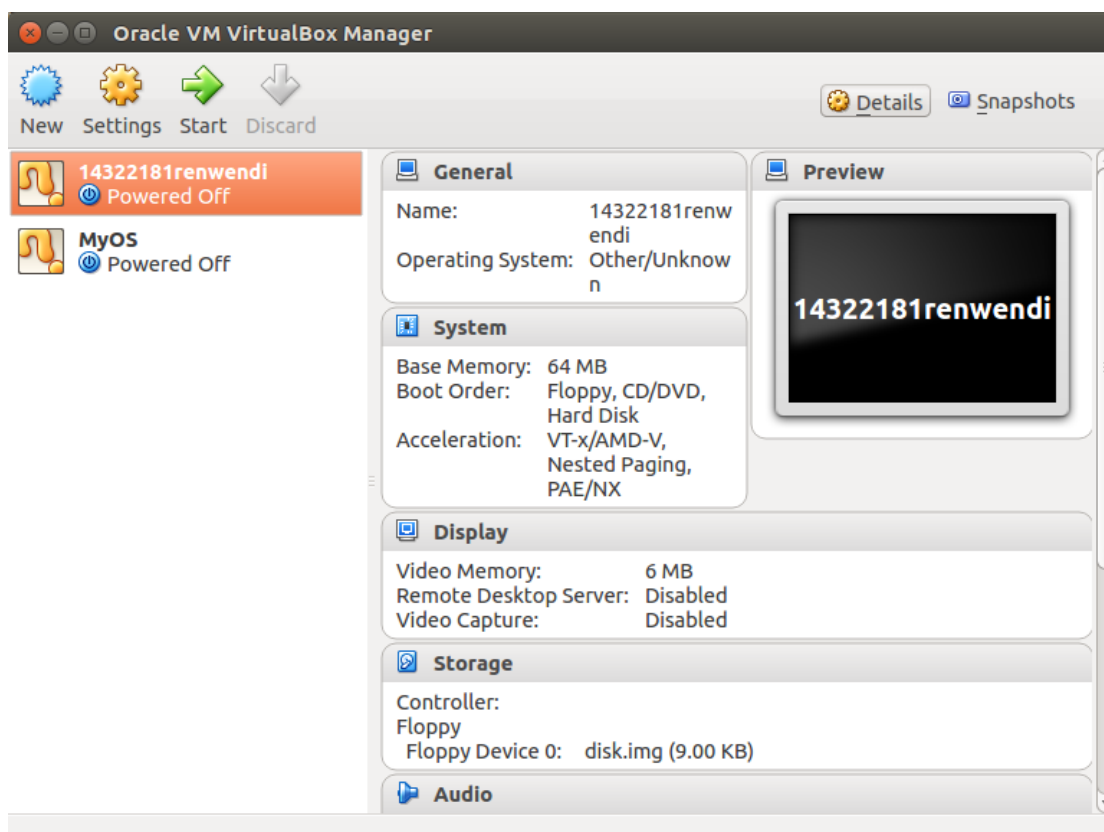
在原型中保证原有的系统调用服务可用。通过run 12依然可实现批处理，敲击键盘时显示OUCH, 敲击4次后返回。use2.asm展示中断系统调用服务还能工作。

use2.asm :

```
1 org 0x1000
2
3
4 int 33h
5 int 34h
6 int 35h
7 int 36h
8
9
10 ;LISTEN_EXIT----
11 listen:
12  mov ch,'A'
13  cmp ch,cl
14  jne listen
15
16 ret
17
18 times 512-($-$$) db 0 ;填充剩余扇区0
```

【实验过程】

1 主要工具安装使用过程及截图结果



2 程序过程中的操作步骤

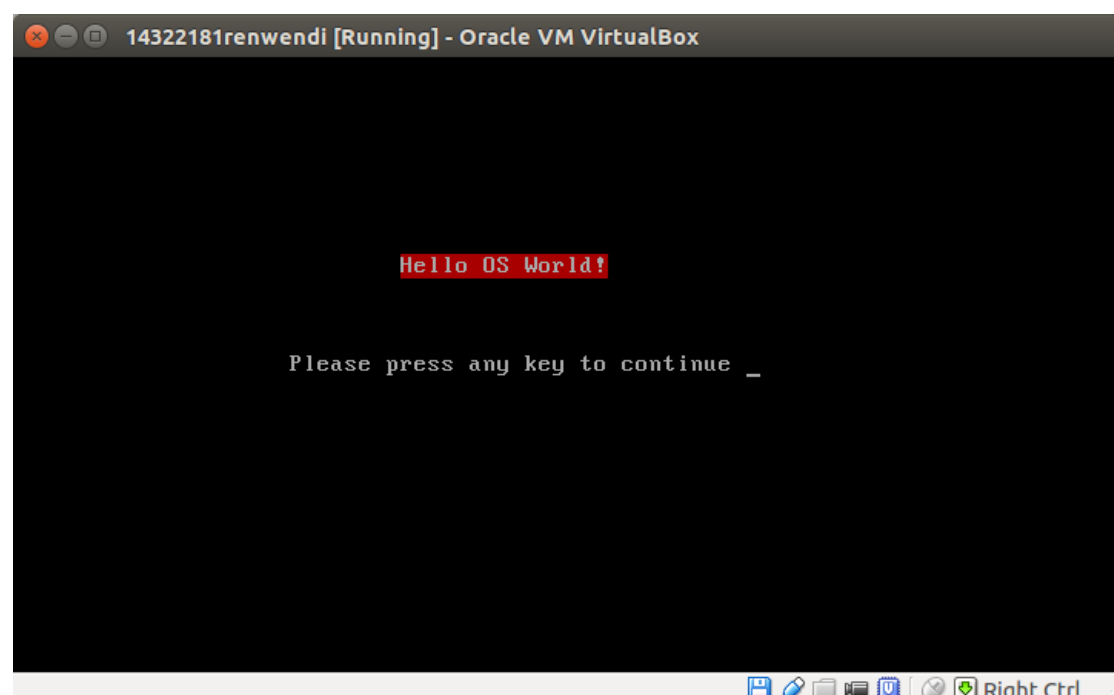
用vim和gedit编写代码。

使用make指令执行Makefile，创建软盘、写入引导程序、编译链接文件。

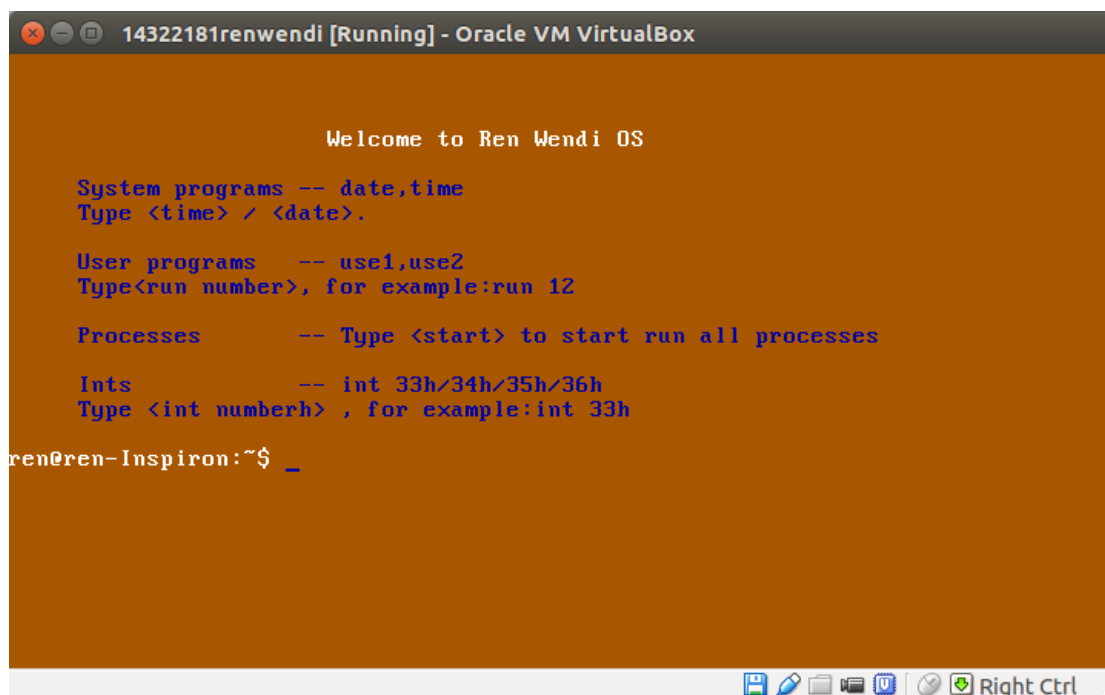
```
nasm -f elf kernel/oslib.asm -o oslib.o
ld -m elf_i386 -Ttext 0x7e00 --oformat binary os.o osc.o oslib.o osclib.o proces
s.o -o os.bin -e main
dd if=/dev/zero of=disk.img count=2880 bs=512
2880+0 records in
2880+0 records out
1474560 bytes (1.5 MB) copied, 0.0319146 s, 46.2 MB/s
dd if=boot.bin of=disk.img
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.000526813 s, 972 kB/s
dd if=os.bin of=disk.img seek=1
22+1 records in
22+1 records out
11476 bytes (11 kB) copied, 0.000486866 s, 23.6 MB/s
dd if=process1.com of=disk.img bs=512 seek=36
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.000298768 s, 1.7 MB/s
dd if=process2.com of=disk.img bs=512 seek=72
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.000203255 s, 2.5 MB/s
dd if=process3.com of=disk.img bs=512 seek=108
1+0 records in
1+0 records out
```

3 输出说明

(1) 引导程序欢迎界面



(2) 系统主菜单



The screenshot shows a terminal window titled "14322181renwendi [Running] - Oracle VM VirtualBox". The terminal displays the "Welcome to Ren Wendi OS" message and a menu of system and user programs. The prompt is "ren@ren-Inspiron:~\$".

```
Welcome to Ren Wendi OS

System programs -- date,time
Type <time> / <date>.

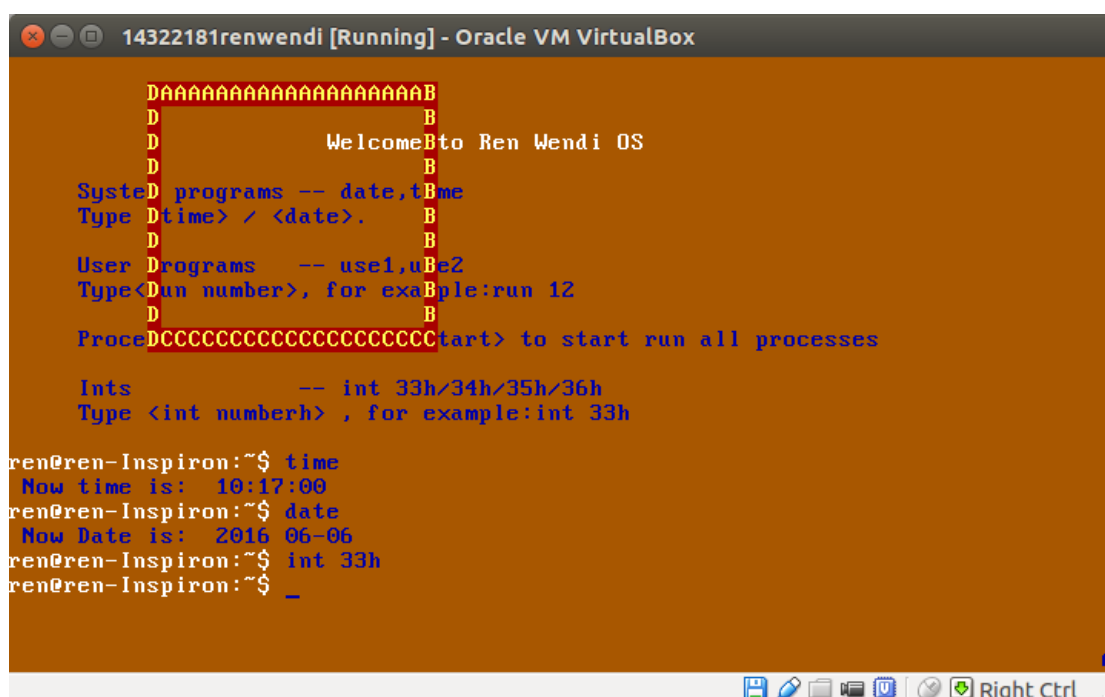
User programs  -- use1,use2
Type<run number>, for example:run 12

Processes      -- Type <start> to start run all processes

Ints           -- int 33h/34h/35h/36h
Type <int numberh> , for example:int 33h

ren@ren-Inspiron:~$ _
```

(3) 展示旧功能（time，date，int以及右下角的字符变化）



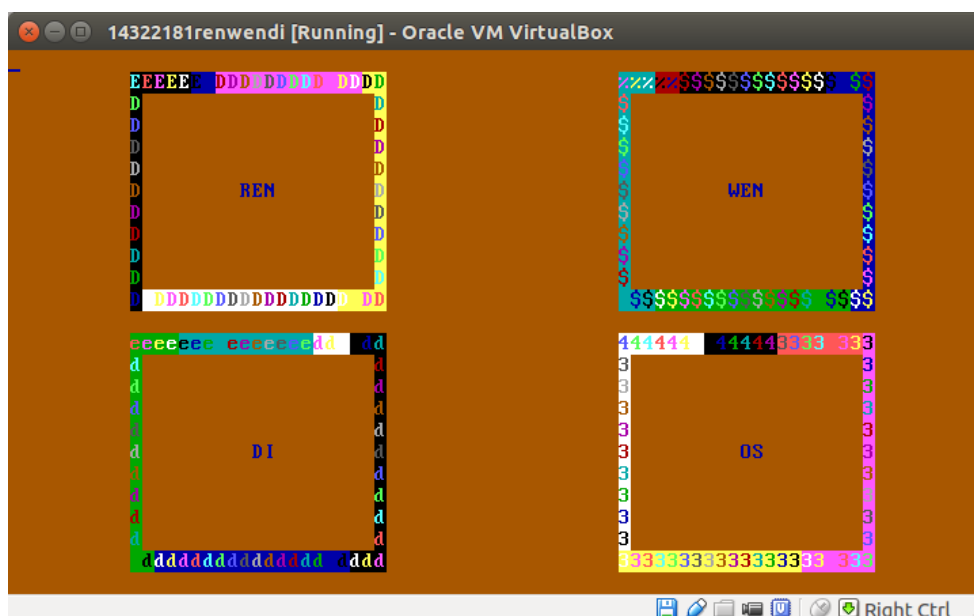
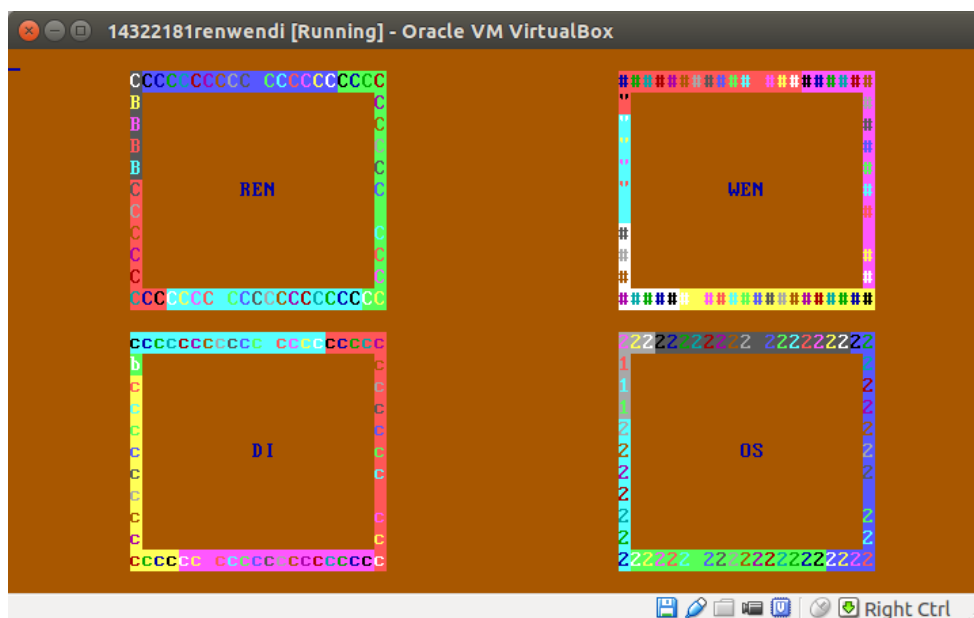
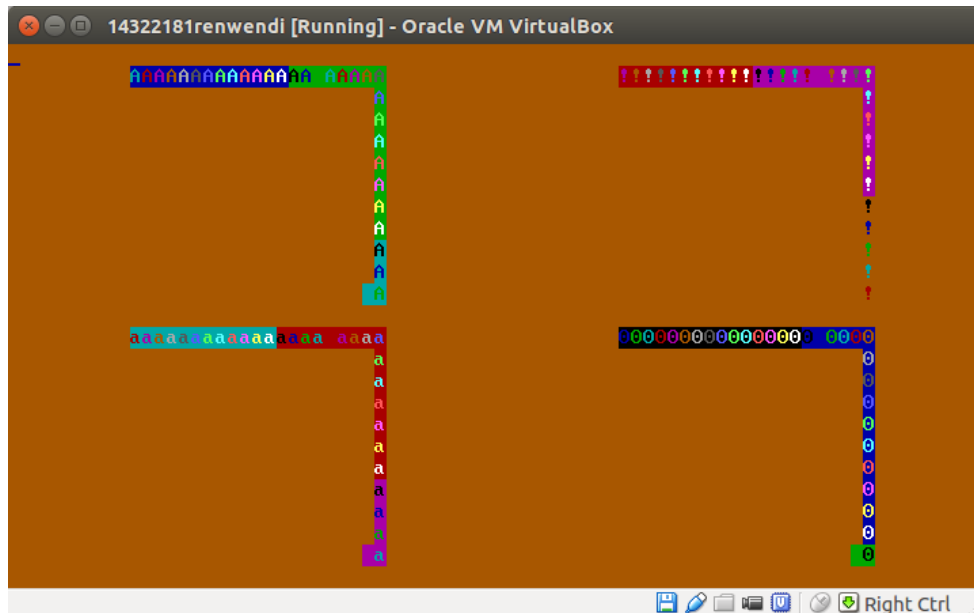
The screenshot shows the same terminal window as before, but with the execution of the 'time', 'date', and 'int' commands. The output shows the current time and date, and the execution of the 'int' command. The prompt is "ren@ren-Inspiron:~\$".

```
ren@ren-Inspiron:~$ time
Now time is: 10:17:00
ren@ren-Inspiron:~$ date
Now Date is: 2016 06-06
ren@ren-Inspiron:~$ int 33h
ren@ren-Inspiron:~$ _
```

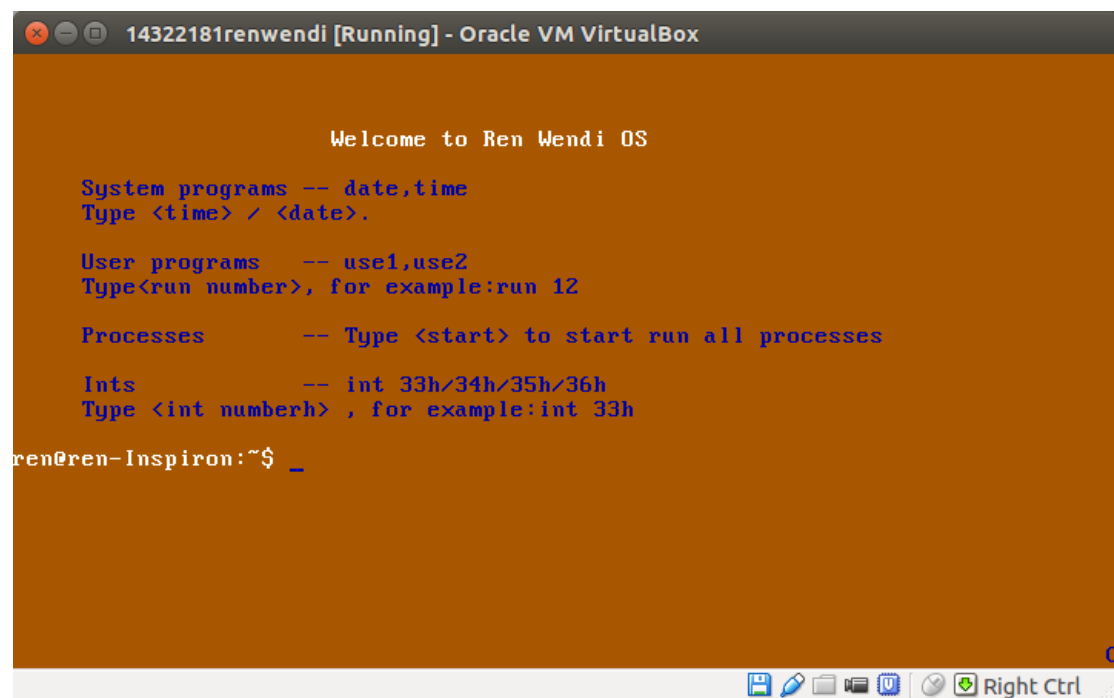
(4) 本实验实现的二态进程模型

输入start触发

动态显示字符（依次为循环大写字母、符号、小写字母、数字）



(5) 任意按键后返回



```
14322181renwendi [Running] - Oracle VM VirtualBox

Welcome to Ren Wendi OS

System programs -- date,time
Type <time> / <date>.

User programs -- use1,use2
Type<run number>, for example:run 12

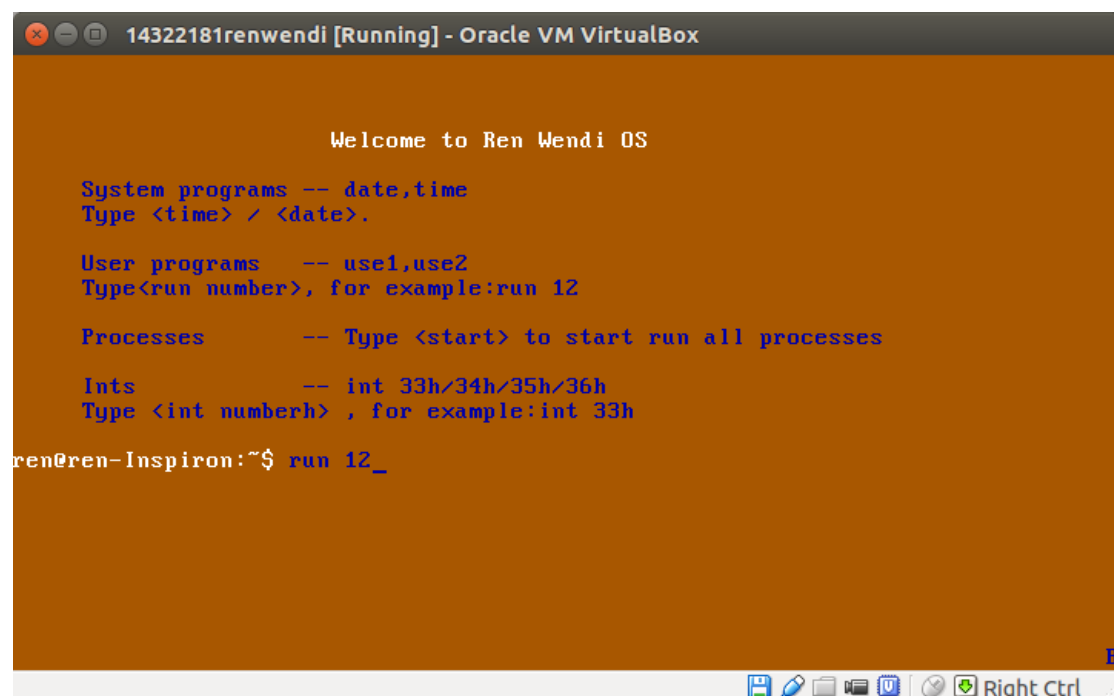
Processes -- Type <start> to start run all processes

Ints -- int 33h/34h/35h/36h
Type <int numberh> , for example:int 33h

ren@ren-Inspiron:~$ _
```

(6) 运行原有用户程序

输入run 12



```
14322181renwendi [Running] - Oracle VM VirtualBox

Welcome to Ren Wendi OS

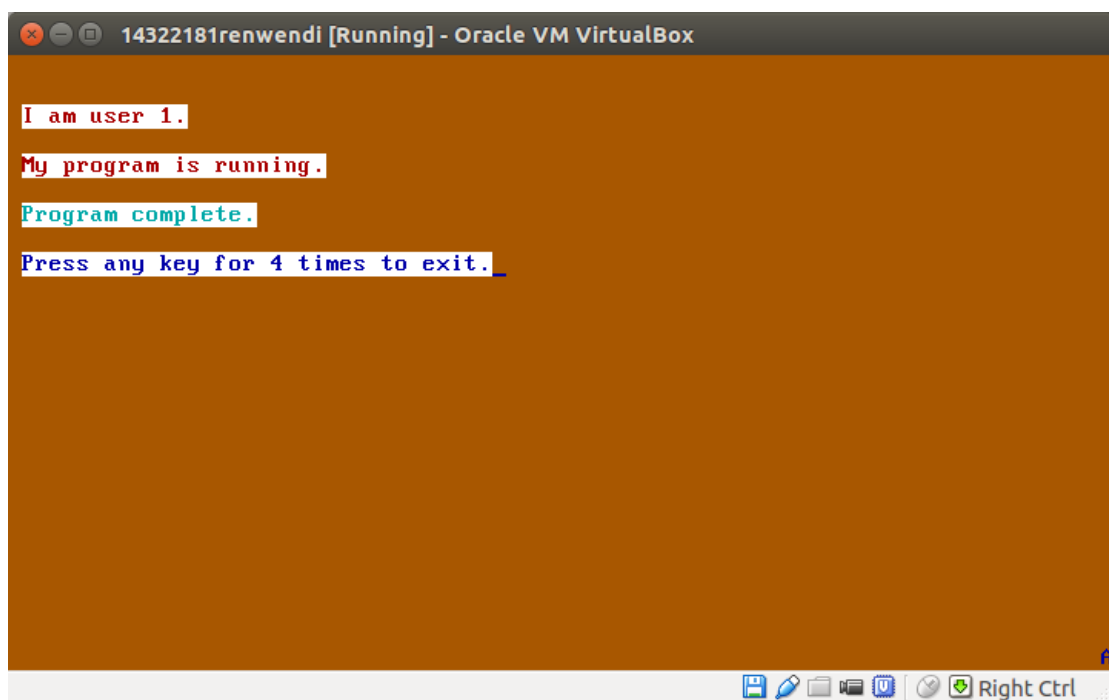
System programs -- date,time
Type <time> / <date>.

User programs -- use1,use2
Type<run number>, for example:run 12

Processes -- Type <start> to start run all processes

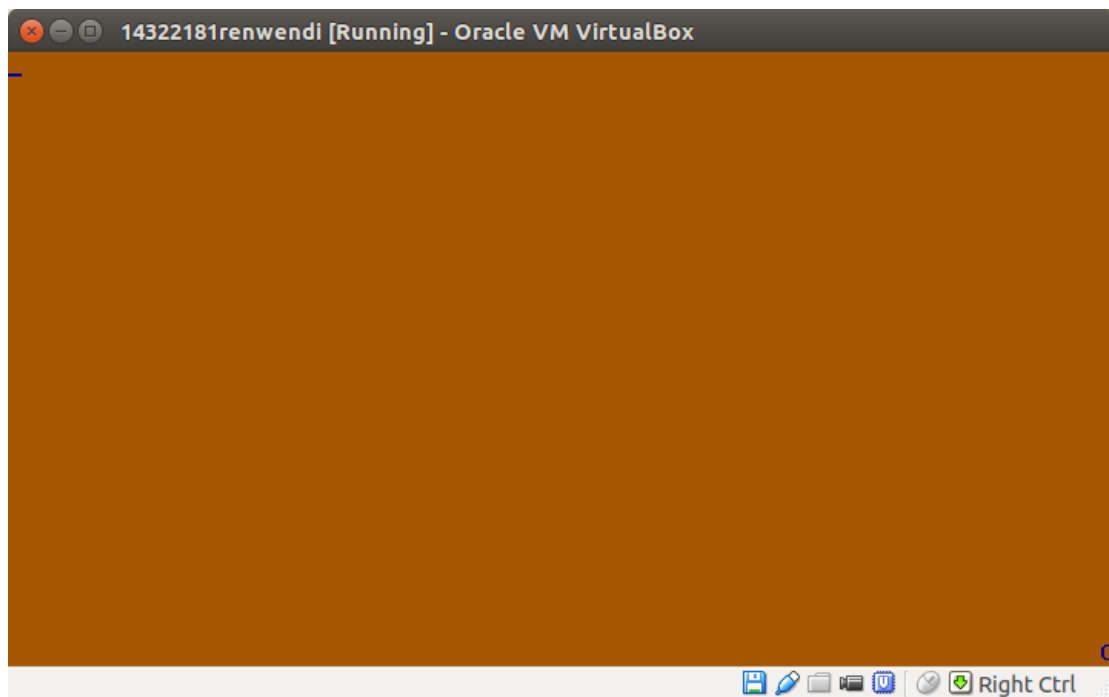
Ints -- int 33h/34h/35h/36h
Type <int numberh> , for example:int 33h

ren@ren-Inspiron:~$ run 12_
```



4 遇到的问题及解决情况

- (1) start无法触发二态进程模型，清屏后无显示
- 时钟中断右下角字符变化显示正常

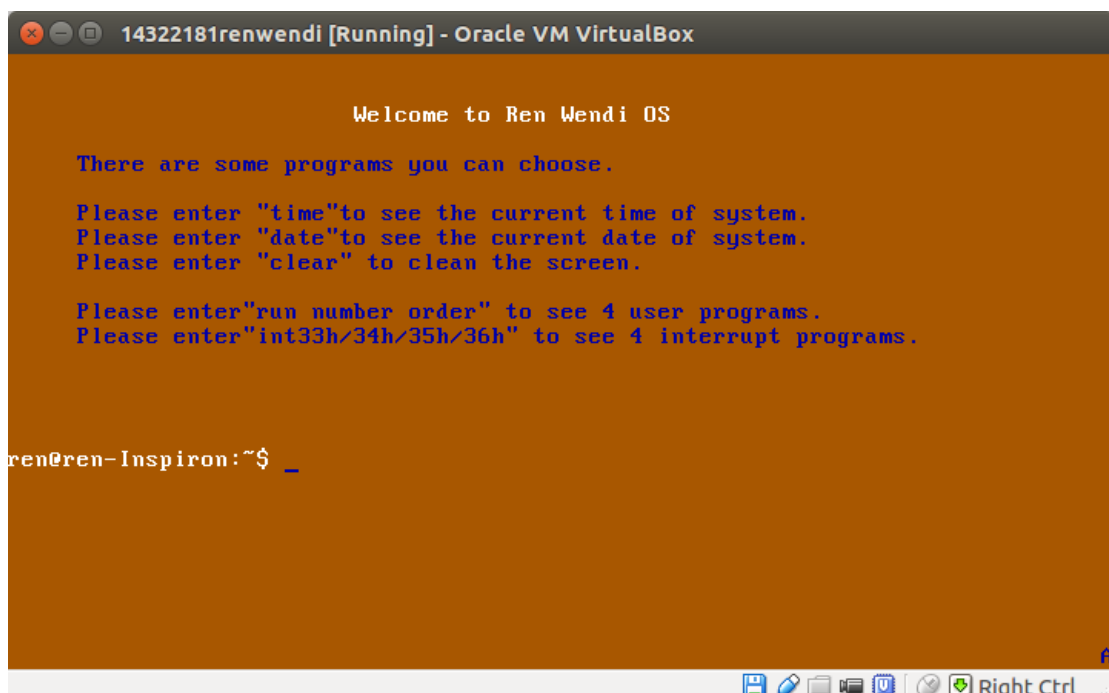


按道理来说在触发二进程模型之后右下角的字符变化中断程序就回停止，但是图中仍在运行，说明应该是没换到新的进程中。

查看代码发现虽然编写了timer_interrupt_proces来选择是执行哪一个时钟中断，但是没有改变初始化的时钟中断，因此就在一直执行原来的那个时钟中断。

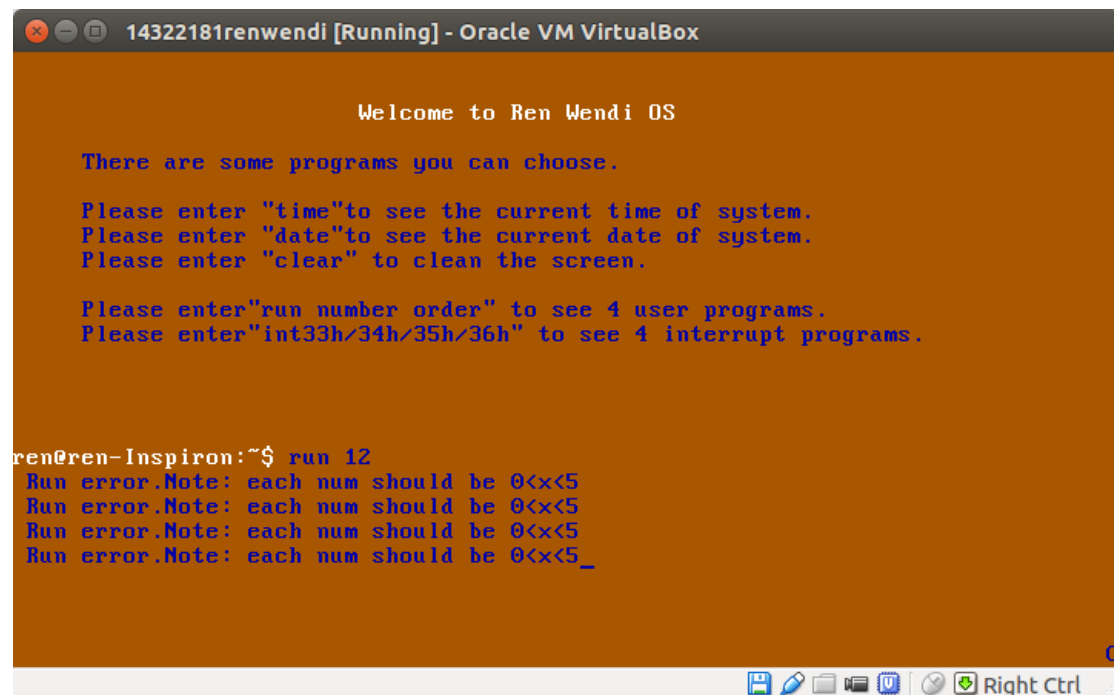
```
mov ax,0x1c
mov [ interrupt_num], ax
mov ax, print_corner
mov [ interrupt_vector_offset],ax
call insert_interrupt_vector
```

(2) 系统卡死，无法输入任何字符，右下角字符也不再改变



这个问题比较严重，最后调试了挺多的问题才改好，其中包括栈的切换问题、时钟中断的清零等，最后具体是因为哪一步出了问题发生也不太确定，最后总算是改好了。

(3) 原有的用户程序use1、2无法运行



```
14322181renwendi [Running] - Oracle VM VirtualBox

Welcome to Ren Wendi OS

There are some programs you can choose.

Please enter "time" to see the current time of system.
Please enter "date" to see the current date of system.
Please enter "clear" to clean the screen.

Please enter "run number order" to see 4 user programs.
Please enter "int33h/34h/35h/36h" to see 4 interrupt programs.

ren@ren-Inspiron:~$ run 12
Run error.Note: each num should be 0<x<5
Run error.Note: each num should be 0<x<5
Run error.Note: each num should be 0<x<5
Run error.Note: each num should be 0<x<5_
```

这个报错是之前有4个用户程序时写的，所以是0-5。

问题应该是由没有修改用户程序的存储位置，因为之前是在同一个柱面中的18个扇区存储的内核和用户程序，而本次的内存安排是：

软盘第1个柱面的第一个扇区存储操作系统引导程序

软盘第1个柱面剩下所有扇区2~36扇区存储操作系统内核

软盘第2, 3, 4, 5, 6柱面分别存储五个进程代码

软盘第7, 8柱面分别存储两个用户程序的程序代码

修改之后正确运行用户程序的代码为：

```
char Usr_num = '3';

inline void run( char *str){
    str += 4;

    while( *str != '\0'){
        if('0'<*str && *str< Usr_num){

            load_user( 5 + *str-'0', 0x1000);
            __asm__(" pop %ax");
```

```

        run_user();
        __asm__(" pop %ax");

    }else{
        run_error();
        return;
    }
    str++;
}
init_flag_position();
screen_init();
print_welcome_msg();
print_message();
print_flag();
}

```

【实验总结】

1、本次实验不足之处

本次实验由于没有使用fat12文件系统，因此在内核文件大小过大时一时间不知道该怎么存储文件了，如果有文件系统就会方便很多，不需要自己再去安排文件的存储扇区位置。

此外，进程调度的过程代码有一些冗余，还可以进一步进行优化。

由于代码量越来越大，因此感觉到回头查代码找bug很容易找不到代码的位置，所以可以考虑将内核文件分开到不同的几个文件中来实现，比如将一部分代码转移到c语言的头文件中，或者将控制模块的监听输入和内核分开，也就是类似终端terminal的那一部分可以单独分开，这样可以使主内核文件 os.c代码量大大减少。而这也是Linux提倡的微内核，以后可以尝试重新架构内核，不仅仅是单纯在之前的基础上进行添加，也可以同时进行优化。

2、实验心得

本次实验需要较多的理论知识的学习，虽然在理论课上已经学习过进程调度切换的相关知识，但是如果真正自己实现，则必须要完全理解，因此本次实验之前也花了一些时间来多次阅读理论知识。

栈堆的切换比较容易出错也比较麻烦，要多加小心，调试起来也比较麻烦，所以在编写代码的时候不能太随意，不能太指望之后的调试过程，非常的费时间，基本上就是半个小时的代码需要三个小时来调试。起初内核一直卡死的时候还是挺着急的，一点一点解决问题的过程中需要足够的耐心和细心。所以在第一次写代码的时候还是要尽量一次到位减少因为粗心造成的错误，这种习惯性的错误和小错误往往难以发现，耗时很久。

【参考文献】

于渊 《Orange's——一个操作系统的实现》

