实验六 多进程操作系统 实验报告

数据科学与计算机学院 2017 级计算机科学与技术 17341146 王程钥

1 实验题目

多进程操作系统

2 实验目的

在内核实现多进程操作系统框架
通过时间片轮转实现多个用户程序轮转调度
搭建好多进程操作系统框架，为后续拓展实验打好基础

3 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 个。

(2)内核一次性加载 4 个用户程序运行时，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，展示系统调用服务还能工作。

4 实验方案

4.1 实验环境

编程环境：Dosbox+TCC+TASM+Tlink,NASM
16 进制编辑器：Hex Editor
虚拟机：WMware Workstation
虚拟机环境

- 操作系统 MS-DOS
- 内存 1MB
- 硬盘 102MB
- 处理器 1 核心

编译命令

```
nasm -f bin guide.asm -o guide.com
tasm myos.asm myos.obj
tcc -mt -c –omain.obj main.c
tlink /3 /t myos.obj main.obj,test.com,
```

4.2 搭建多进程框架，实现多用户程序并发执行

根据实验要求，需要在原有操作系统的基础上，在原型操作系统中引入进程的概念，搭建一个多进程框架，支持多用户程序并发执行。我为用户程序和内核分别建立了进程，并通过时钟中断，以每 1/18.2 秒一次的频率调用一次时钟中断更换执行的进程，以此实现多用户程序并发的功能。

4.3 设计进程状态

对于每个进程，我设置了阻塞，就绪，死亡，运行四个状态，并在内核提供了开始运行(run)，结束运行(stop)，暂停(pause)三种操作。进程状态根据用户输入命令和程序运行情况进行变化。

4.4 程序存储设计与地址存放

以下为存储扇区位置。

功能	程序名	存储扇区
引导程序	guide.com	1
文件夹	file.txt	2
内核	test.com	3-36
用户程序 1	stone1.com	37
用户程序 2	Stone2.com	38
用户程序 3	stone3.com	39
用户程序 4	stone4.com	40
脚本程序 1	shell1.bat	41
脚本程序 2	shell2.bat	42

以下为内存地址。

功能	地址
引导程序	7c00h
内核	a100h
用户程序	Cs=1000h+40h*k ip=100h

上表中 x 表示创建进程的数量，第一次为 1。

5 实验过程

5.1 相关声明

本次实验以实验五实现的增加了系统调用的原型操作系统为基础，加入了多进程框架，实现了多用户程序并发执行。

为了在用户程序执行时能够在键盘输入进行操作，我删除了实验四中实现的 9 号键盘中断显示“OUCH!”的功能。我保留了 8 号时钟中断中原有的显示“风火轮”和显示系统时间的功能，并在 8 号中断中增加了进程切换的相关功能。

因为出现了一些内存冲突的问题，我干脆将系统中断部分，即修改后的 int21h 中断移回了内核，放在内核中的汇编部分，即 myos.asm 程序中。

我修改了内核的命令，增加了 pause 命令和 stop 命令，分别表示进程暂停和进程结束。我增加了 show 命令，用于显示当前所有非死亡进程的状态。我删除了控制 9 号中断修改的 ouch 命令。

我增加了一个头文件 pcb.h，用于存放进程控制块轮换的相关代码和 PCB 类的定义。

5.2 进程控制块的设计

进程控制块包含寄存器镜像，进程状态和进程名字。进程名字为用户程序的程序名。在本次实验中每个用户程序只对应一个进程。进程状态的设计见 5.4 节。

寄存器镜像包含如下的 14 个需要保存的寄存器：

主寄存器：ax,bx,cx,dx,si,di

段寄存器：cs,ds,es,ss

偏移寄存器：ip,sp,bp

标志寄存器：flags

以下为进程控制块的设计。

```
typedef struct Register_Image{
  int AX,BX,CX,DX,SI,DI,BP;
  int ES,DS,SS,SP,IP,CS,FLAGS;
}RegImg;

typedef struct PCB{
  RegImg registers;
  int state;
  char name[BUFLEN];
}PCB;
```

5.3 进程创建与结束

5.3.1 思路

每执行一次 run 命令运行用户程序时，内核会为该用户程序创建一个新的进程。该进程的段地址为 0x1000+0x40*x，x 表示运行用户程序的次数。主寄存器的初值均为 0，标志寄存器 flags 的初值为 512。

我希望用户程序执行结束后能够返回内核的指定位置，执行相关操作自动将对应进程的进程状态设置为死亡状态。之前我是使用 call 指令完成这一操作。现在如果使用 call 指令的话内核会跳到用户程序，无法实现并发。因此，在创建新进程时，我需要模拟 call 指令，先将用户程序需要返回的指定位置的 cs 和 ip 压入该进程的栈中。

5.3.2 实验情况

在之前的用户程序中，进程运行结束后会执行 ret 指令返回内核，即会将栈中的 cs 和 ip 取出来，并跳转回到相应位置。我进行了相关操作，装入虚拟机运行时却发现子进程运行结束后无法跳回我希望它跳回的地方。

我查阅了一下资料，发现 ret 这条伪指令只会 pop ip，不会 pop cs。由于之前的实验执行的不是段间跳，所以这样做并不会出事。要执行段间的回跳，需要将 ret 指令改为 retf 指令。retf 指令对应的汇编指令为 pop ip，pop cs，刚好与我的要求相对应。

我把 ret 改成了 retf，装入虚拟机再进行测试，结果发现还是无法返回正确的位置。我尝试输出的 pcb 中所有变量的值，并将设置的段地址和偏移量取出进行比对。结果发现两着不一样。

我突然想起来实验四时遇到的一个问题。当时我打算存下 8 号中断和 9 号中断默认的 cs 和 ip 的值，结果因为段地址寄存器 ds 改变了导致存下来的值无法写入全局变量 cs8/cs9 和 ip8/ip9 中。当时我通过增加两个临时变量解决了这个问题。

本次实验中，CS 和 IP 这两个变量也是全局变量。因此，我尝试开两个临时变量 tmp_cs 和 tmp_ip，使用这两个变量进行复制。再装入虚拟机中测试，结果用户程序结束后成功跳转到了设定的偏移地址。

以下为相关代码。

```
int init(char *str,int segment,int offset)
{
  int i;
  for(i=0;i<Pro_Size;i++)
    if(Pro_List[i].state==DIE)break;
  strcpy(Pro_List[i].name,str);
  Pro_List[i].state=DIE;
  Pro_List[i].registers.AX=0;
  Pro_List[i].registers.BX=0;
  Pro_List[i].registers.CX=0;
  Pro_List[i].registers.DX=0;
  Pro_List[i].registers.SI=0;
  Pro_List[i].registers.DI=0;
  Pro_List[i].registers.BP=0;
  Pro_List[i].registers.FLAGS=512;
  Pro_List[i].registers.CS=segment;
  Pro_List[i].registers.DS=segment;
  Pro_List[i].registers.ES=segment;
  Pro_List[i].registers.IP=offset;
  Pro_List[i].registers.SS=segment;
  Pro_List[i].registers.SP=offset-4;
  return i;
}
```

初始化 PCB 并返回进程 id

```
int CS=0x1000,base=0x040,IP;
void run(char *str)
{
  int i,cur_pos,pos,id,len=strlen(str);
  int tmp_cs,tmp_ip;
  char sec[BUFLEN],buf[BUFLEN];
  strcpy(buf,str);
  for(i=4;i<len;i=cur_pos+1)
  {
    cur_pos=get_name(i,buf,sec);
    if(cur_pos<0)break;
    id=Find_Pro(sec);
    if(id>=0)
    {
      if(Is_Pause(id))Make_Alive(id);
      else puts("this program is already running");
      continue;
    }
    CS+=base;IP=OffSetOfUserPrg;
    pos=find(sec,"com");
```

```
    if(!pos)continue;
    id=init(sec,CS,IP);
    fake_call(CS,IP);
    Make_Alive(id);
    load_com(pos,CS,IP);
    have_run=1;
  }
}
```

run 指令，创建新进程

```
void fake_call(int CS,int IP)
{
  int tmp_cs=CS,tmp_ip=IP;
  asm mov cx,ds                    /*要返回到内核！*/
  asm push ds
  asm mov ax,tmp_cs
  asm mov ds,ax
  asm mov bx,tmp_ip
  asm sub bx,2
  asm mov [bx],cx             /*返回段地址进栈*/
  asm sub bx,2
  asm mov ax,offset com_ret
  asm mov [bx],ax             /*返回偏移量进栈*/
  asm pop ds
}
```

模拟 call

```
void fake_ret()
{
  asm com_ret:
  asm mov ax,0a00h
  asm mov ds,ax
  asm mov es,ax
  Make_Die(Cur_Pro);
  asm jmp $
}
```

模拟 ret

```
void print_all()
{
  printf("ax: ");putint(ax_save);puts("");
  printf("bx: ");putint(bx_save);puts("");
  printf("cx: ");putint(cx_save);puts("");
  printf("dx: ");putint(dx_save);puts("");
  printf("cs: ");putint(cs_save);puts("");
```

```
  printf("ds: ");putint(ds_save);puts("");
  printf("es: ");putint(es_save);puts("");
  printf("ip: ");putint(ip_save);puts("");
  printf("si: ");putint(si_save);puts("");
  printf("di: ");putint(di_save);puts("");
  printf("bp: ");putint(bp_save);puts("");
  printf("flags: ");putint(flags_save);puts("");
  printf("Cur_Pro: ");putint(Cur_Pro);puts("");
}
```

调试代码，输出所有寄存器的值

5.4 进程状态的设计

进程状态及其对应标志值如下表:

进程状态	标志值
死亡(die)	0
阻塞(block)	1
就绪(ready)	2
运行(run)	3

进程状态根据以下方式变化。新建的进程自动进入就绪状态，暂停的进程进入阻塞状态，重新执行后则恢复就绪状态。当用户程序执行结束后，或执行 stop 命令结束一个用户程序后，将这个进程变为死亡状态，使其在时钟中断时不再执行。

每次时钟中断发生时，运行 schedule 函数修改进程状态并选择接下来要执行的进程。首先修改正在运行的进程的状态，若它已经死亡则将其改为死亡态，否则激情其改为就绪状态。之后，根据进程编号选择一个就绪状态的进程，将其设置为运行状态并执行该进程。

进程状态改变的相关代码如下。

```
#define DIE   0
#define BLOCK 1
#define READY 2
#define RUN   3
//defination
void Make_Alive(int id){Pro_List[id].state=READY;}
void Make_Die(int id){Pro_List[id].state=DIE;}
void Make_Pause(int id){Pro_List[id].state=BLOCK;}
int Is_Pause(int id){return Pro_List[id].state==BLOCK;}
//check and change
void Schedule()
{
  Save_Pro(&Pro_List[Cur_Pro]);
  if(Pro_List[Cur_Pro].state!=DIE)
    Pro_List[Cur_Pro].state=READY;
  while(1)
  {
```

```
    Cur_Pro=(Cur_Pro+1)%Pro_Size;
    if(Pro_List[Cur_Pro].state==READY)break;
  }
  Restore_Pro(&Pro_List[Cur_Pro]);
  Pro_List[Cur_Pro].state=RUN;
}
```

5.5 进程切换

进程切换的时候，需要取出当前 14 个寄存器的值，将其存入 PCB，并将即将执行的新进程中 PCB 存储的寄存器镜像中寄存器的值存入寄存器中。这里需要涉及到两个操作，save 操作和 restore 操作。

5.5.1 save 操作

我的 save 操作分为两部分，第一部分将寄存器的值存到 ax_save,bx_save,cx_save 等全局变量中，第二部分将上述全局变量的值存入相应进程的 PCB 中。

第一部分的实现思路如下，我首先将 ds 变成内核的 cs，这样才能读写上述全局变量。接下来，我要将栈中的 flags,cs,ip 按照入栈的顺序依次退栈并存入相应变量。因为这三个寄存器不能直接访问，所以我先将 ax 存入相应变量 ax_save，随后将 ax 作为出栈的临时寄存器，先用 ax 保存出栈变量，再将其存入 flags,cs,ip 中。最后，我使用 mov 指令将剩余的寄存器的值存入相应全局变量中。

相关代码如下：

```
Save:
  push ds                 ; stack: *\flags\cs\ip\ds(user)
  push cs                 ; stack: *\flags\cs\ip\ds(user)\cs(kernal)
  pop ds                  ; stack: *\flags\cs\ip\ds(user)
                          ; ds=cs(kernal)

  mov _ax_save,ax         ; save ax
  pop ax                  ; Stack: *\flags\cs\ip
  mov _ds_save,ax         ; save ds
  pop ax                  ; stack: *\flags\cs
  mov _ip_save,ax         ; save ip
  pop ax                  ; stack: *\flags
  mov _cs_save,ax         ; save cs
  pop ax                  ; stack: *
  mov _flags_save,ax      ; save ax
  mov _bx_save,bx         ; save bx
  mov _cx_save,cx         ; save cs
  mov _dx_save,dx         ; save dx
  mov _si_save,si         ; save si
  mov _di_save,di         ; save di
  mov _bp_save,bp         ; save bp
  mov _es_save,es         ; save es
  mov _ss_save,ss         ; save ss
```

```
  mov _sp_save,sp         ; save sp
```

第二部分比较简单，直接对结构体变量赋值即可。相关代码如下：

```
void Save_Pro(PCB *Pro)
{
  Pro->registers.AX=ax_save;
  Pro->registers.BX=bx_save;
  Pro->registers.CX=cx_save;
  Pro->registers.DX=dx_save;
  Pro->registers.SI=si_save;
  Pro->registers.DI=di_save;
  Pro->registers.BP=bp_save;
  Pro->registers.ES=es_save;
  Pro->registers.DS=ds_save;
  Pro->registers.SS=ss_save;
  Pro->registers.SP=sp_save;
  Pro->registers.IP=ip_save;
  Pro->registers.CS=cs_save;
  Pro->registers.FLAGS=flags_save;
}
```

5.5.2 restore 操作

restore 操作与 save 操作基本是对称的关系。在确定好下一段时间要执行的进程后，我先将其 PCB 中寄存器映象中的值赋值到 ax_save 等全局变量中，后回到汇编部分将上述全局变量的值存入相应寄存器中。

第一部分直接赋值即可，相关代码如下。

```
void Restore_Pro(PCB *Pro)
{
  ax_save=Pro->registers.AX;
  bx_save=Pro->registers.BX;
  cx_save=Pro->registers.CX;
  dx_save=Pro->registers.DX;
  si_save=Pro->registers.SI;
  di_save=Pro->registers.DI;
  bp_save=Pro->registers.BP;
  es_save=Pro->registers.ES;
  ds_save=Pro->registers.DS;
  ss_save=Pro->registers.SS;
  sp_save=Pro->registers.SP;
  ip_save=Pro->registers.IP;
  cs_save=Pro->registers.CS;
  flags_save=Pro->registers.FLAGS;
}
```

第二部分的实现思路如下：先将 bx,cx,dx,si 等寄存器相应全局变量的值存入寄存器，后将 flags,cs,ip 三个寄存器依次进栈。和 save 时相同，我使用 ax 作为临时的辅助寄存

器，先将这三个寄存器对应的值存入 ax，在将 ax 弹入栈中。最后，将 ax_save 的值存入 ax。相关代码如下。

```
Restore:
  mov bx,_bx_save         ; restore bx
  mov cx,_cx_save         ; restore cs
  mov dx,_dx_save         ; restore dx
  mov si,_si_save         ; restore si
  mov di,_di_save         ; restore di
  mov bp,_bp_save         ; restore bp
  mov es,_es_save         ; restore es
  mov ss,_ss_save         ; restore ss
  mov sp,_sp_save         ; restore sp
  mov ax,_flags_save      ; save flags
  push ax                 ; stack *\flags
  mov ax,_cs_save         ; save cs
  push ax                 ; stack *\flags\cs
  mov ax,_ip_save         ; save ip
  push ax                 ; stack *\flags\cs\ip
  mov ax,_ax_save         ; save ax
  mov ds,_ds_save         ; save ds
```

5.6 内核进程初始化与时钟中断操作

在装入时钟中断之前，先为内核分配一个进程，使得用户程序在运行时内核可以同步操作，执行更多的用户程序。内核进程放在进程队列的第 0 个位置，设置为就绪状态，将进程名赋为“kernal”，并初始化当前进程变量 Cur_Pro=0。

同时，我保留了实验四中实现的在屏幕右下角显示系统时间和风火轮。当输入 time 命令后，显示或消失这两项输出。我将这两项输出也移植到了内核中，在 clock.h 中编写了相关函数，使用 C 内嵌汇编的方法对代码进行了重构。以下为相关代码。

```
initialize()
{
  int i;
  for(i=0;i<Pro_Size;i++)Clear_PCB(&Pro_List[i]);
  Cur_Pro=0;
  Make_Alive(Cur_Pro);
  strcpy(Pro_List[Cur_Pro].name,"kernal");
}
```

时钟中断初始化

```
time_main()
{
  Schedule();
  if(TIME)Show_Int8();
}
```

时钟中断 main 函数

```
void Show_Wheel()
```

```
{
  Time_wait--;
  if(Time_wait)return;
  Time_wait=4;
  if(now==0)putchar_no_format('|');
  if(now==1)putchar_no_format('/');
  if(now==2)putchar_no_format('-');
  if(now==3)putchar_no_format(92);
  now=(now+1)%4;
}
//显示风火轮

void bcd_to_ascii(unsigned char tmp)
{
  unsigned char A,B;
  asm push ax
  asm mov ah,tmp
  asm mov al,tmp
  asm and al,0x0f
  asm add al,0x30
  asm shr ah,4
  asm and ah,0x0f
  asm add ah,0x30
  asm mov A,ah
  asm mov B,al
  asm pop ax
  putchar_no_format(A);
  putchar_no_format(B);
}
//BCD 码转 ASCII 码

void Show_Time()
{
  char Yh,Yl,M,D,H,S;

  asm mov ah,04h
  asm int 1ah

  asm mov Yh,ch
  asm mov Yl,cl
  asm mov M,dh
  asm mov D,dl
  bcd_to_ascii(Yh);
  bcd_to_ascii(Yl);
```

```
  putchar_no_format('/');
  bcd_to_ascii(M);
  putchar_no_format('/');
  bcd_to_ascii(D);
  putchar_no_format(' ');

  asm mov ah,02h
  asm int 1ah

  asm mov H,ch
  asm mov M,cl
  asm mov S,dh
  bcd_to_ascii(H);
  putchar_no_format(':');
  bcd_to_ascii(M);
  putchar_no_format(':');
  bcd_to_ascii(S);
}
//显示系统时间

void Show_Int8()
{
  int X=x,Y=y;
  move(24,59);
  Show_Time();
  putchar_no_format(' ');
  Show_Wheel();
  move(X,Y);
}
```

显示系统时间与风火轮

5.7 显示进程表

当输入 show 命令时，会显示当前正在执行或处于暂停状态的所有进程的相关信息，包括进程编号和进程名字。和 ls 命令类似，枚举进程队列中每个进程，按照相关格式输出即可。以下为相关代码。

```
void Show_Pro()
{
  int i;
  puts("============================");
  print_len("id",7);putchar('|');
  print_len("name",10);putchar('|');
  print_len("state",10);puts("");
  for(i=0;i<Pro_Size;i++)
  {
```

```
    if(Pro_List[i].state==DIE)continue;
    puts("----------------------------");
    putint_len(i,7);putchar('|');
    print_len(Pro_List[i].name,10);putchar('|');
    if(Pro_List[i].state==BLOCK)print_len("pause",10);
    else print_len("run",10);puts("");
  }
  puts("============================");
}
```

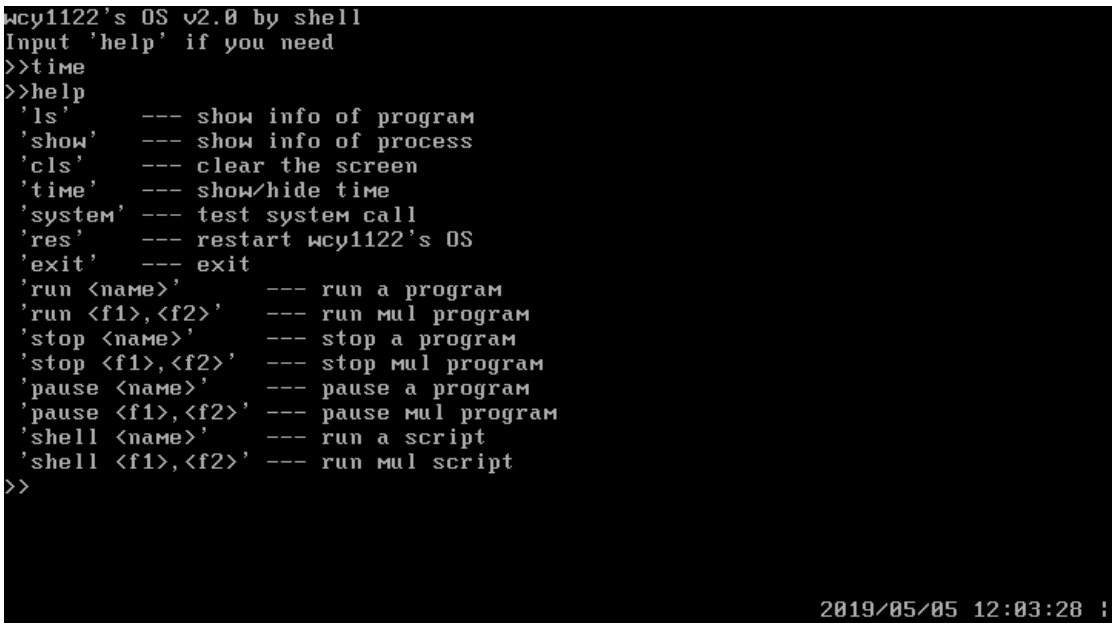
6 操作系统使用指南

6.1 命令解释

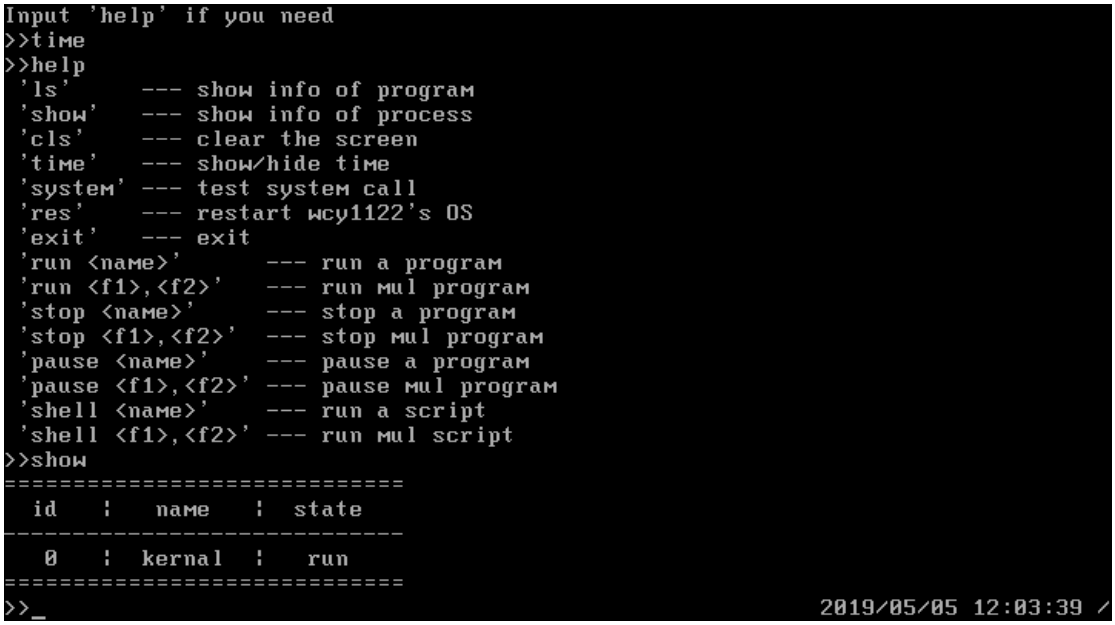
ls	显示用户程序表
show	显示进程表
cls	界面清屏
time	显示/隐藏系统时间
system	测试系统调用
res	重启操作系统
exit	离开操作系统
run <name1>,<name2>	开始执行/恢复执行多个用户程序
stop <name1>,<name2>	结束执行多个用户程序
pause <name1>,<name2>	暂停执行多个用户程序
shell <name1>,<name2>	执行脚本程序
上下左右与退格键都有相关功能，具体功能与 windows 中的 cmd 或 linux 中的 terminal 类似（如删除，移动光标，使用历史命令）。你可以放心地使用这些快捷键。	

6.2 实验结果

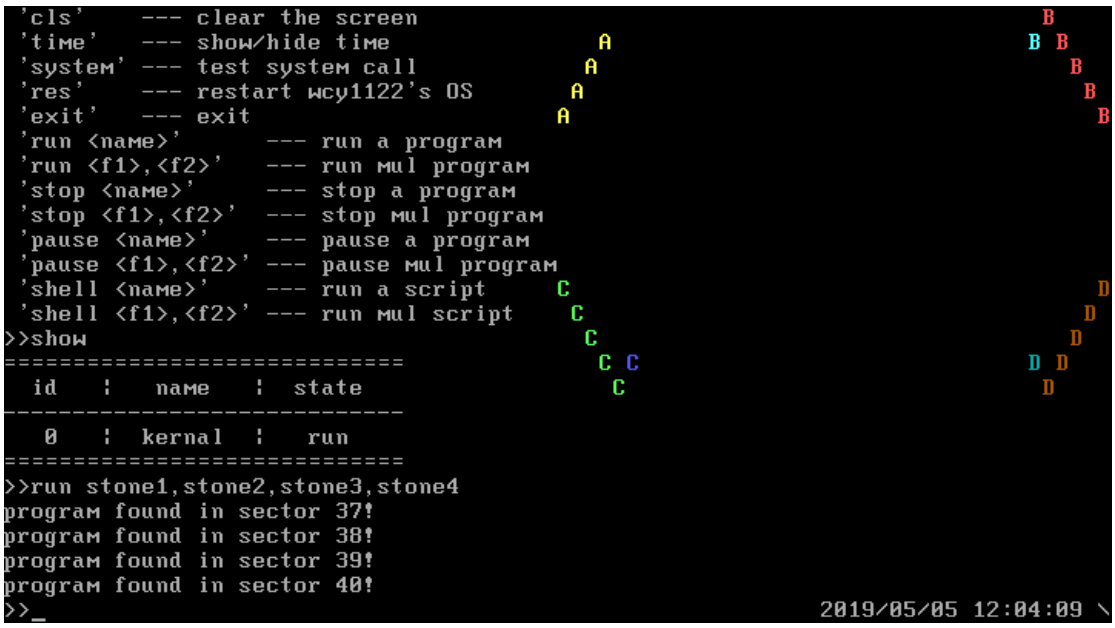
进入操作系统



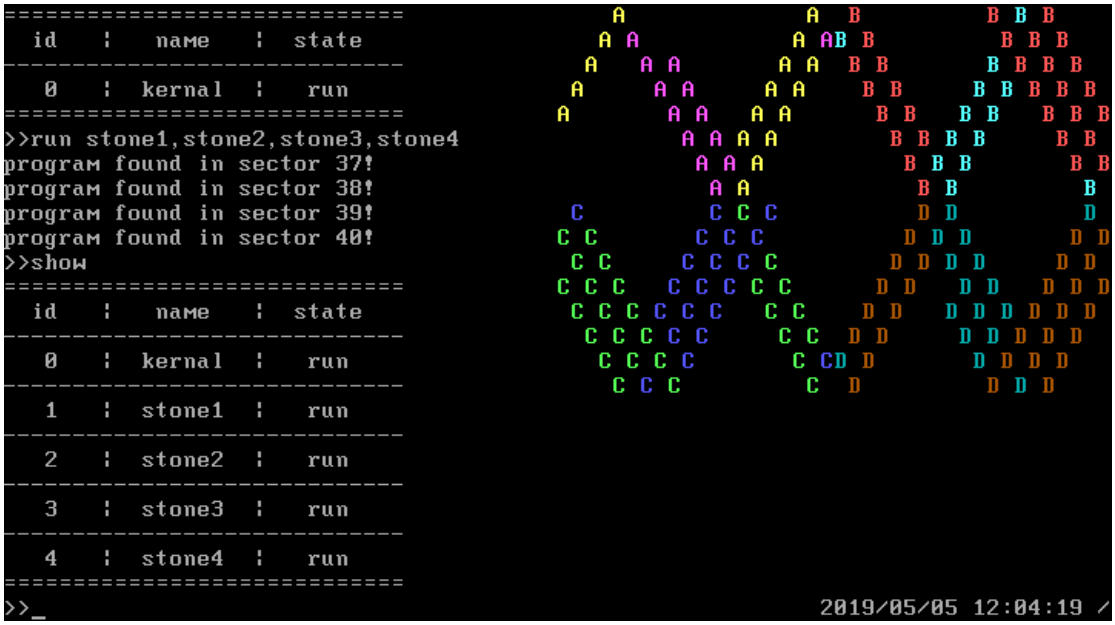
输入 time 命令显示时间，输入 help 命令显示可执行命令集



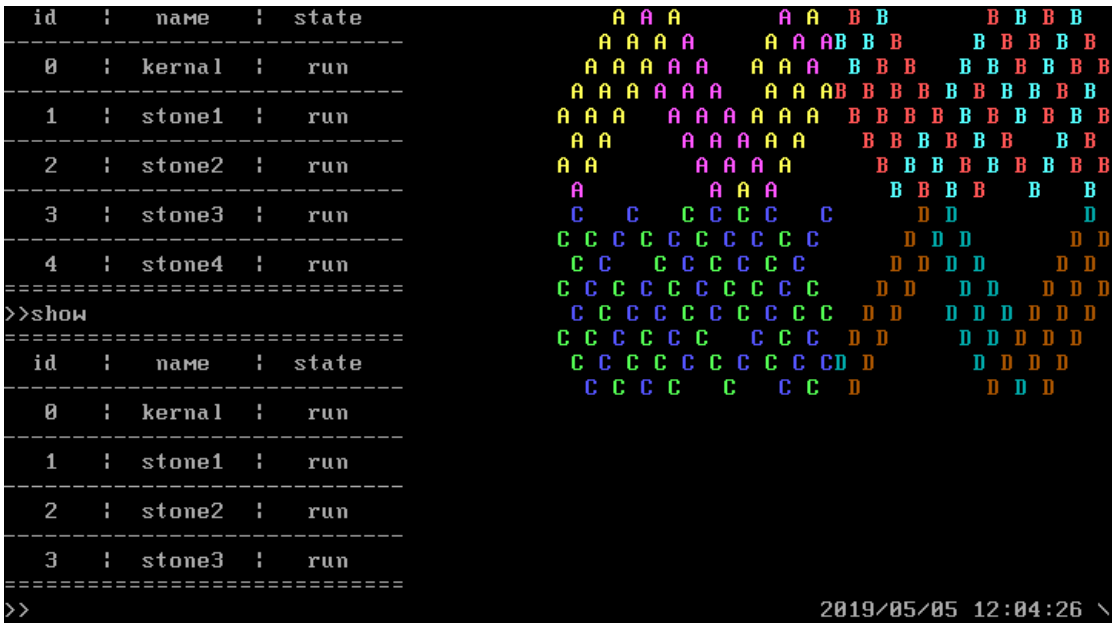
输入 show 命令，当前只有 kernal 进程在执行



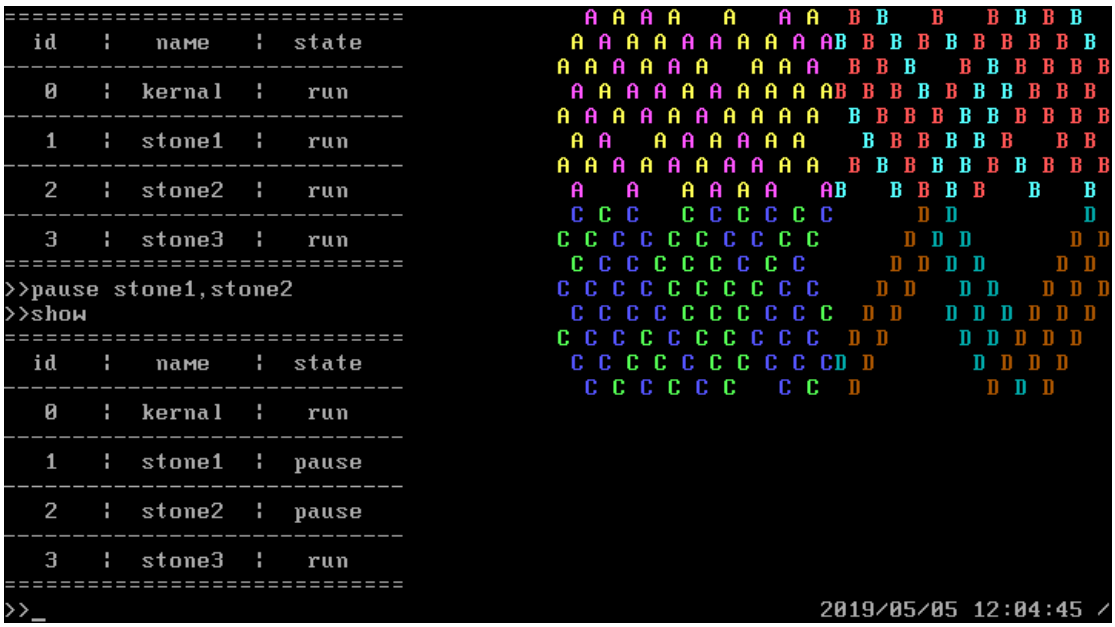
执行 run 命令，同时执行四个用户程序



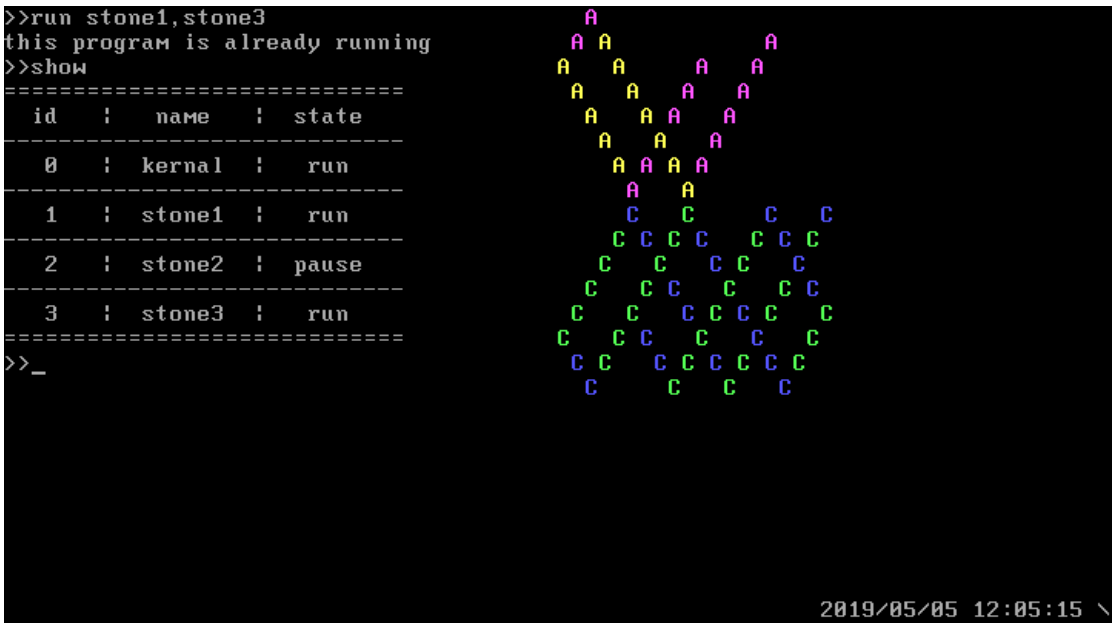
执行 show 命令，五个进程（包括内核）都在执行



stone4 的执行次数比其他三个少，执行结束后不再出现在进程表中。

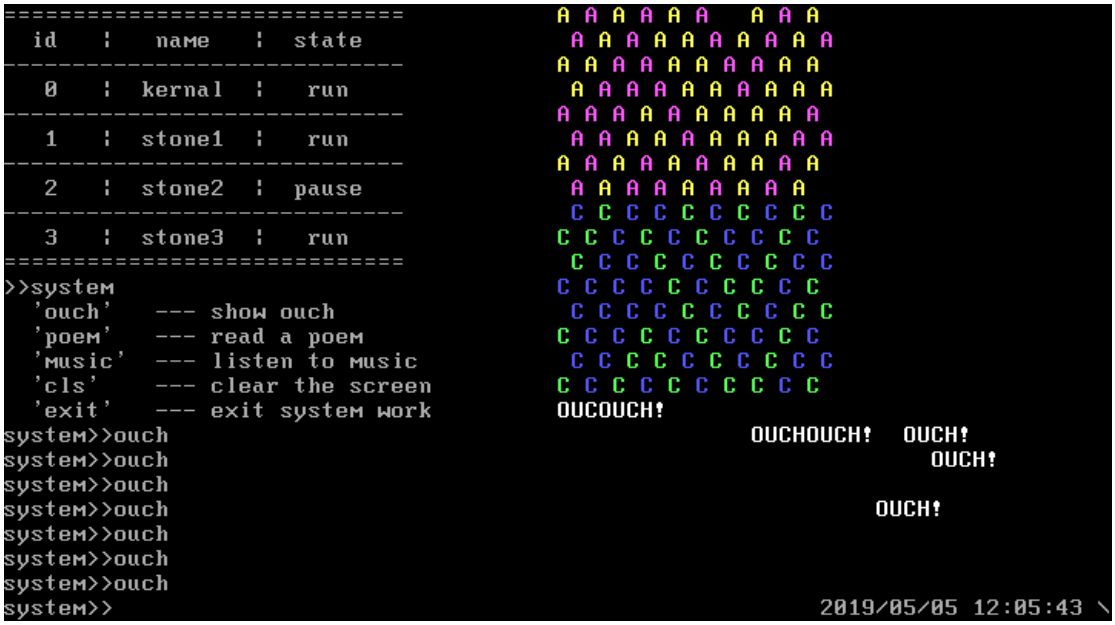


暂停 stone1,stone2 对应进程

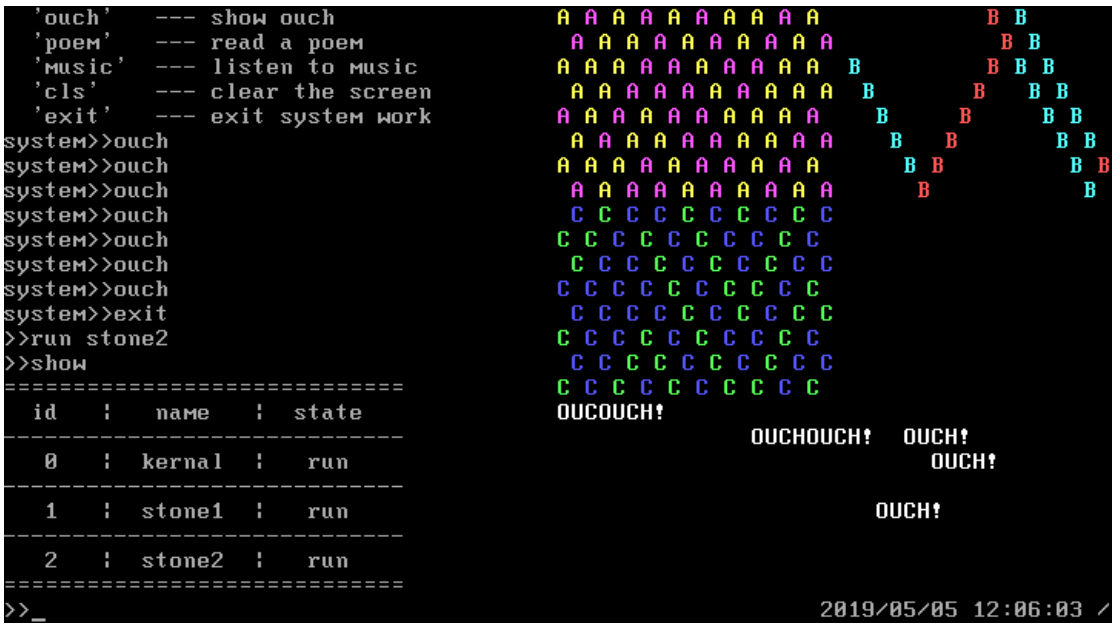


清屏，并运行 stone1，stone3。

stone1 重新开始运行，stone3 已在运行。

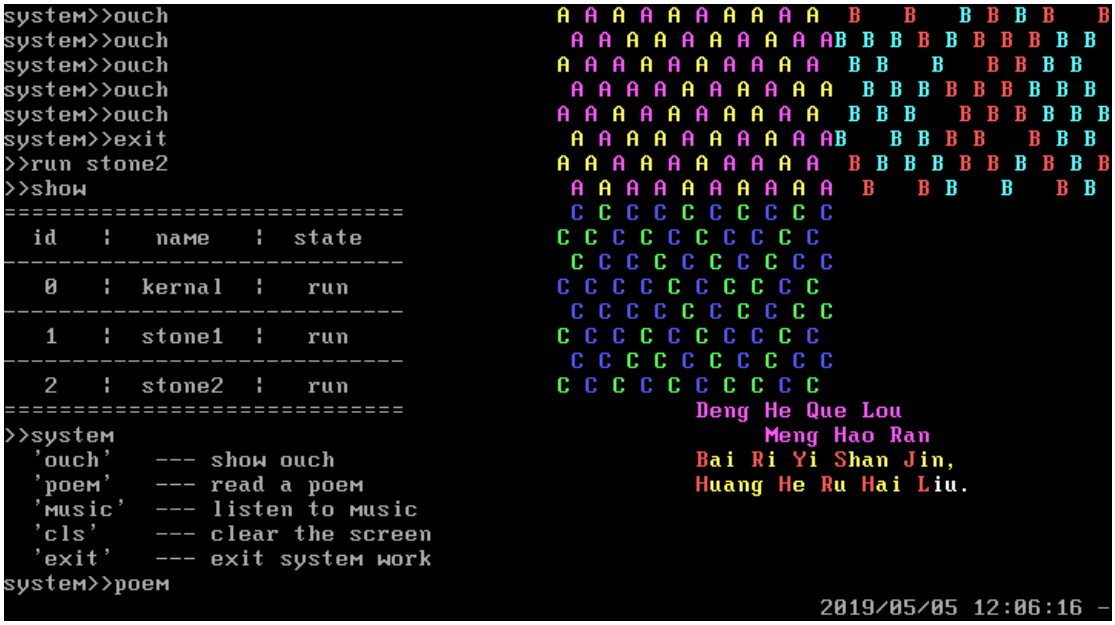


进入系统调用测试页面，输出多次“OUCH!”

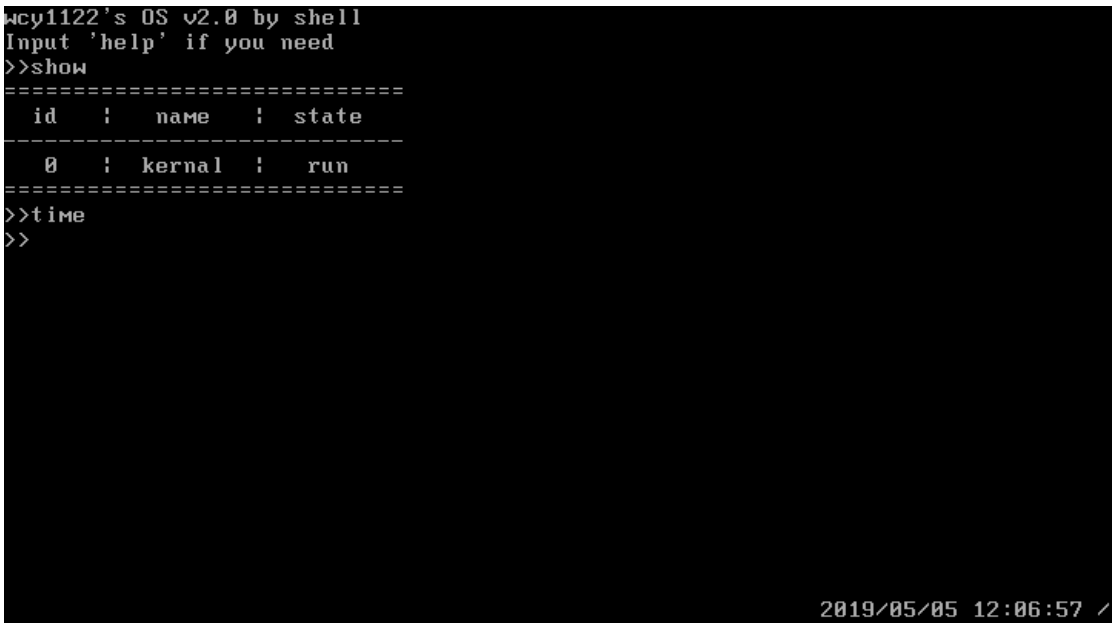


运行 stone2，查看进程表。

此时 stone3 已运行结束，stone1 和 stone2 仍在运行

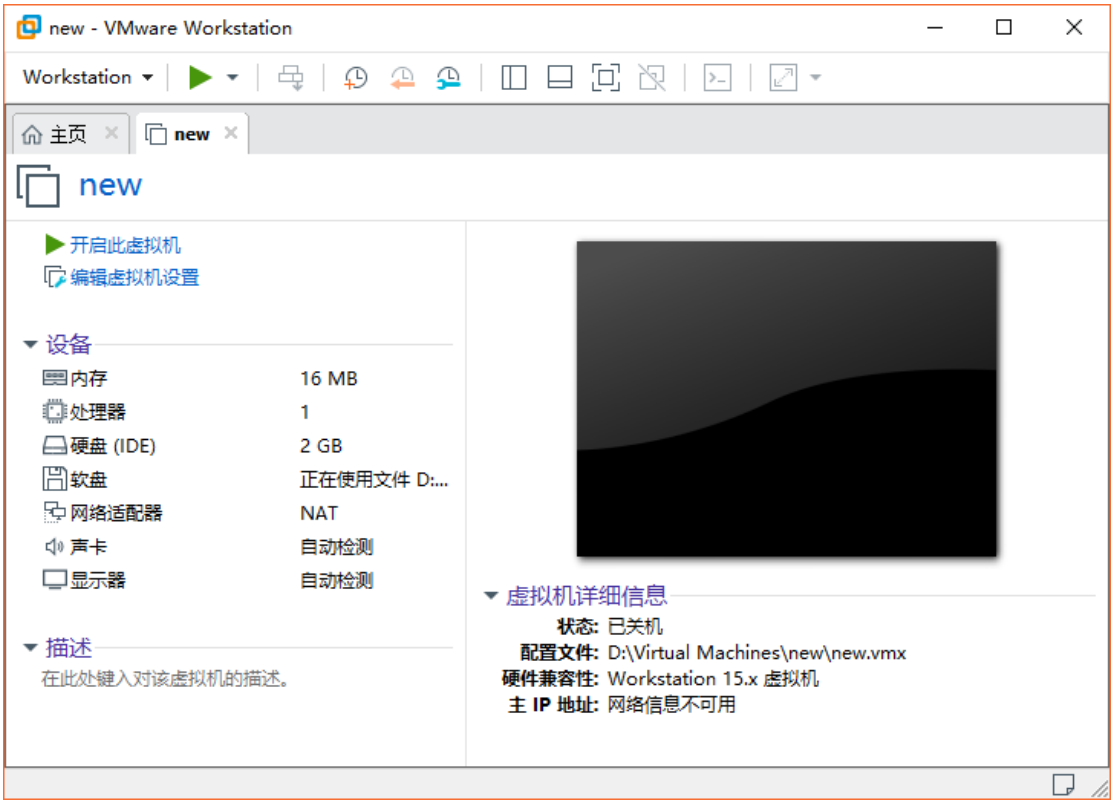


再进入系统调用，测试 poem 指令，成功显示一首诗。由于系统调用放在内核中，没有专门开一个进程，所以此时内核无法进行操作。



清屏，显示进程表和时间。用户程序均已执行完毕，不出现在进程表上。

准备退出操作系统



成功退出操作系统

7 实验总结

本次实验是本学期操作系统实验中继实验三之后的有一个重要实验。这个实验要求搭建好原型操作系统的多进程框架，在不影响原有原型操作系统的情况下，实现多个用户程序同步运行的功能。这个实验有一定难度，不过深入理解相关原理后其实也不算太难。

一开始我因为 ret 指令和 retf 指令的问题，卡了很长的一段时间，后来去网上查阅了相关资料，并结合之前几次实验所遇到的问题，才成功解决了用户程序返回内核的问题。这主要是我太想当然了，把 ret 指令和 retf 指令搞混了才导致的结果。做实验的时候不能太想当然，要想清楚原理再开始实验。

不过总体上来说本次实验还是比较顺利的，我成功实现了一个简单的多进程原型操作系统，自己实现了 save，restore 等操作。通过时钟中断，我实现了进程的执行，暂停，恢复，停止等功能，实现了一个可以算是五状态模型的简单进程模型（新建态直接进入就绪队列）。大体上达到了实验开始前对本次实验的预期。

这个实验是基础实验中的最后一个实验，也为后序的拓展实验打下了基础。在后序实验中，我会尝试将操作系统理论课中学到的相关知识应用到实验中，利用信号量等工具实现一些和同步，互斥相关的操作。我还会实现原型操作系统中自己的文件系统。如果技术成熟，我还会在操作系统的界面和用户体验上进行一些设计，加入鼠标，加入滚轮。

参考文献

[1] BIOS 中断大全 https://www.cnblogs.com/coderCaoyu/p/3638713.html

[2] ret，retf 指令 https://blog.csdn.net/longintchar/article/details/50989444