

实验六：实现时间片轮转的二态进程模型

18340052 何泽

实验六：实现时间片轮转的二态进程模型

18340052 何泽

一、实验目的

二、实验要求

三、实验内容

四、实验方案

1.相关基础原理

2.实验环境与工具版本

3.实验思路

五、实验过程与结果

1. 操作系统功能

2. 引导程序

3.内核与用户程序

5.编译

6.运行

六、创新工作

七、实验总结

一、实验目的

1. 学习多道程序与CPU分时技术
2. 掌握操作系统内核的二态进程模型设计与实现方法
3. 掌握进程表示方法
4. 掌握时间片轮转调度的实现

二、实验要求

1. 了解操作系统内核的二态进程模型
2. 扩展实验五的内核程序，增加一条命令可同时创建多个进程分时运行，增加进程控制块和进程表数据结构。
3. 修改时钟中断处理程序，调用时间片轮转调度算法
4. 设计实现时间片轮转调度算法，每次时钟中断，就切换进程，实现进程轮流运行
5. 修改 `save()` 和 `restart()` 两个汇编过程，利用进程控制块保存当前被中断进程的现场，并从进程控制块恢复下一个进程的现场
6. 编写实验报告，描述实验工作的过程和必要的细节，如截屏或录屏，以证实实验工作的真实性

三、实验内容

1. 修改实验5的内核代码，定义进程控制块PCB类型，包括进程号、程序名、进程内存地址信息、CPU寄存器保存区、进程状态等必要数据项，再定义一个PCB数组，最大进程数为10个
2. 扩展实验五的内核程序，增加一条命令可同时执行多个用户程序，内核加载这些程序，创建多个进程，再实现分时运行
3. 修改时钟中断处理程序，保留无敌风火轮显示，而且增加调用进程调度过程

```
1 Timer:
2     save()
3     call showWingFireWheel() ;无敌风火轮显示
4     call _schedule()        ; 调用进程调度过程
5     jmp restart
```

4. 内核增加进程调度过程：每次调度，将当前进程转入就绪状态，选择下一个进程运行，如此反复轮流运行。

```
1 void schedule(){
2     CurrentProcessNo++;
3     if (CurrentProcessNo=MaxProcessNo)
4         CurrentProcessNo=0;
5 }
```

5. 修改`save()`和`restart()`两个汇编过程，利用进程控制块保存当前被中断进程的现场，并从进程控制块恢复下一个进程的运行
6. 实验5的内核其他功能，如果不必要，可暂时取消服务

四、实验方案

1.相关基础原理

- 进程模型就是实现多道程序和分时系统的一个理想的方案
 - 多个用户程序并发执行
 - 进程模型中，操作系统可以知道有几个用户程序在内存运行，每个用户程序执行的代码和数据放在什么位置，入口位置和当前执行的指令位置，哪个用户程序可执行或不可执行，各个程序运行期间使用的计算机资源情况等等
- 二状态进程模型
 - 执行和等待
 - 目前进程的用户程序都是 **COM** 格式的，是最简单的可执行程序
 - 进程仅涉及一个内存区、CPU、显示屏这几种资源，所以进程模型很简单，只要描述这几个资源
- 初级进程
 - 现在的用户程序都很小，只要简单地将内存划分为多个小区，每个用户程序占用其中一个区，就相当于每个用户拥有独立的内存
 - 根据我们的硬件环境，CPU可访问 **1M** 内存，我们规定 **MYOS** 加载在第一个 **64K** 中，用户程序从第二个 **64K** 内存开始分配，每个进程 **64K**，作为示范，我们实现的 **MYOS** 进程模型只有两个用户程序，大家可以简单地扩展，让 **MYOS** 中容纳更多的进程
 - 对于键盘，我们先放后解决，即规定用户程序没有键盘输入要求，我们将在后继的关于终端的实验中解决
 - 对于显示器，我们可以参考内存划分的方法，将 **25** 行 **80** 列的显示区划分为多个区域，在进程运行后，操作系统的显示信息是很少的我们就将显示区分为 **4** 个区域，用户程序如果要显示信息，规定在其中一个区域显示。当然，理想的解决方案是用户程序分别拥有一个独立的显示器，这个方案会在关于终端的实验中提供
 - 文件资源和其它系统软资源，则会通过扩展进程模型的数据结构来实现，相关内容将安排在文件系统实验和其它一些相关实验中
- 进程表
 - 初级的进程模型可以理解为将一个CPU模拟为多个逻辑独立的CPU，每个进程具有一个独立的逻辑CPU
 - 同一计算机内并发执行多个不同的用户程序，**MYOS** 要保证独立的用户程序之间不会互相干扰。为此，内核中建立一个重要的数据结构：进程表和进程控制块 **PCB**
 - 现在的 **PCB** 它包括进程标识和逻辑CPU模拟

- 逻辑CPU

- 8086CPU 的所有寄存器：AX/BX/CX/DX/BP/SP/DI/SI/CS/DS/ES/SS/IP/FLAG
- 用内存单元模拟
- 逻辑CPU轮流映射到物理CPU，实现多道程序的并发执行

- 进程交替执行

- 在以前的原型操作系统顺序执行用户程序，内存中不会同时有两个用户程序，所以CPU控制权交接问题简单，操作系统加载了一个用户到内存中，然后将控制权交接给用户程序，用户程序执行完再将控制权交接回操作系统，一次性完成用户程序的执行过程
- 采用时钟中断打断执行中的用户程序实现CPU在进程之间交替
- 简单起见，我们让两个用户的程序均匀地推进，就可以在每次时钟中断处理时，将CPU控制权从当前用户程序交接给另一个用户程序

- 内核

- 利用时钟中断实现用户程序轮流执行
- 在系统启动时，将加载两个用户程序A和B，并建立相应的PCB。
- 修改时钟中断服务程序
 - 每次发生时钟中断，中断服务程序就让A换B或B换A。
 - 要知道中断发生时谁在执行，还要把被中断的用户程序的CPU寄存器信息保存到对应的PCB中，以后才能恢复到CPU中保证程序继续正确执行。中断返回时，CPU控制权交给另一个用户程序

- 现场保护：save 过程

- Save 是一个非常关键的过程，保护现场不能有丝毫差错，否则再次运行被中断的进程可能出错
- 涉及到三种不同的栈：应用程序栈、进程表栈、内核栈。其中的进程表栈，只是我们为了保存和恢复进程的上下文寄存器值，而临时设置的一个伪局部栈，不是正常的程序栈
- 在时钟中断发生时，实模式下的CPU会将FLAGS、CS、IP先后压入当前被中断程序（进程）的堆栈中，接着跳转到（位于kernel内）时钟中断处理程序（Timer函数）执行。注意，此时并没有改变堆栈的SS和SP，换句话说，我们内核里的中断处理函数，在刚开始时，使用的是被中断进程的堆栈
- 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。不能先进行栈切换，再来保存寄存

器。因为切换栈所需的若干指令，会破坏寄存器的当前值。这正是我们在中断处理函数的开始处，安排代码保存寄存器的内容

- 我们 `PCB` 中的 16 个寄存器值，内核一个专门的程序 `save`，负责保护被中断的进程的现场，将这些寄存器的值转移至当前进程的 `PCB` 中

- 进程切换: `restart` 过程

- 用内核函数 `restart` 来恢复下一进程原来被中断时的上下文，并切换到下一进程运行。这里面最棘手的问题是 `SS` 的切换。
- 使用标准的中断返回指令 `IRET` 和原进程的栈，可以恢复（出栈）`IP`、`CS` 和 `FLAGS`，并返回到被中断的原进程执行，不需要进行栈切换。
- 如果使用我们的临时（对应于下一进程的）`PCB` 栈，也可以用指令 `iret` 完成进程切换，但是却无法进行栈切换。因为在执行 `iret` 指令之后，执行权已经转到新进程，无法执行栈切换的内核代码；而如果在执行 `iret` 指令之前执行栈切换（设置新进程的 `SS` 和 `SP` 的值），则 `iret` 指令就无法正确执行，因为 `iret` 必须使用 `PCB` 栈才能完成自己的任务。
- 解决办法有三个，一个是所有程序，包括内核和各个应用程序进程，都使用共同的栈。即它们共享一个（大栈段）`SS`，但是可以有各自不同区段的 `SP`，可以做到互不干扰，也能够用 `iret` 进行进程切换。第二种方法，是不使用 `iret` 指令，而是改用 `retf` 指令，但必须自己恢复 `FLAGS` 和 `SS`。第三种方法，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，这也是我们程序所采用的方案

2.实验环境与工具版本

- 平台: `Windows + Ubuntu`

```
heze@ubuntu:~/os/os3$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 20.04 LTS
Release:        20.04
Codename:       focal
heze@ubuntu:~/os/os3$ cat /proc/version
Linux version 5.4.0-29-generic (build@lgw01-amd64-035) (gcc version 9.3.0 (Ubuntu 9.3.0-10ubuntu2)) #33-Ubuntu SMP Wed Apr 29 14:32:27 UTC 2020
```

- c语言编译器: `tcc` ,运行环境: `DosBox 0.74`
- 汇编工具: `nasm` + `tasm`

```
heze@ubuntu:~$ nasm --version
NASM version 2.14.02
```

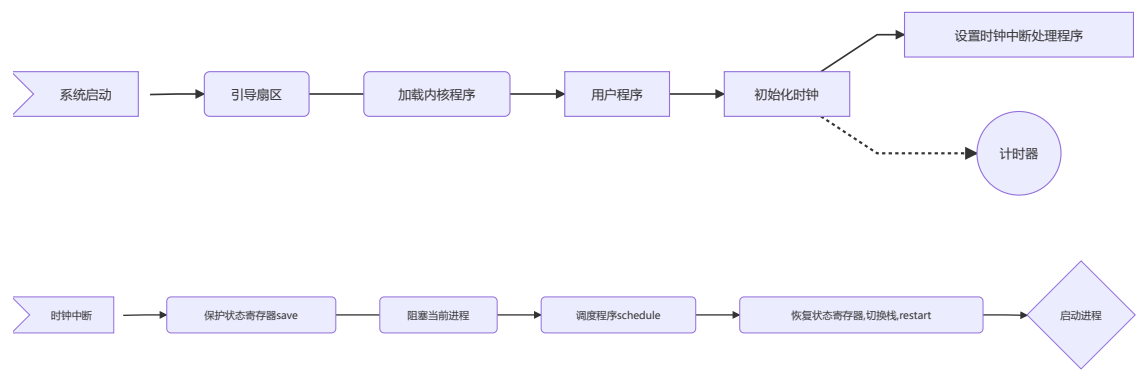
- `Make` :

```
heze@ubuntu:~$ make --version
GNU Make 4.2.1
Built for x86_64-pc-linux-gnu
Copyright (C) 1988-2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

- 虚拟机： VMware Workstation Pro 15

3.实验思路

- 系统框架



五、实验过程与结果

1. 操作系统功能

- 操作系统内核功能

进入系统后，会在右下角显示“无敌风火轮”

输入命令	功能描述与参数解释
name	显示程序的名字
ls	显示用户程序信息
clean	清屏，只留下开头的指引
time	获取当前的时间

- 执行用户程序

用户程序功能为数字在屏幕反弹，第几个程序就是数字几在反弹，可以“串行”执行也可以“并行”执行

输入命令	解释
run + 程序序号	可以执行单个程序，如 run 1，也可以按顺序执行多个，如 run 2341

输入命令	解释
run_plus + 多个程序序号	同时执行多个用户程序，如 <code>run_plus 1234</code>

- 中断服务程序

输入命令	功能
int 33h	在左上显示字符 “This is int 33h”
int 34h	在右上显示字符 “This is int 34h”
int 35h	在左下显示字符 “This is int 35h”
int 36h	在右下显示字符 “This is int 36h”

- 系统调用，`int 21h`，输入 `int 21h+功能号`，如 `int 21h 2`，可进行系统调用，各功能号对应的功能如下：

功能号	功能
0	显示ouch字符
1	将输入的字符串再输出
2	小写转为大写
3	大写转为小写
4	二进制数转十进制数
5	十六进制数转十进制数

2. 引导程序

- 引导程序的作用是加载操作系统内核，同时输出字符，因为和之前的一样，不再详细叙述
- 因为引导成功后直接进入内核，而内核会先清屏后输出字符，所以在实际过程中引导程序的字符并不会被看见，因为太快了，只有在程序出错导致无法正确引导的时候才能看见这串字符

3. 内核与用户程序

我的文件结构如下：

文件名	功能
interr.asm	所有的中断程序、风火轮、 save 与 restart 的实现
lib.asm	一些C程序调用的库函数以及各中断的调用函数
MyOS.asm	主汇编文件
MyPro.h	进程控制块 PCB 的实现
MyOS.c	主C程序

由于大量的代码和上个实验都是相同的，之前也有过详细的叙述，下面就只叙述这次实验改动、新加的部分

- 改进了时钟中断处理程序，并保留风火轮显示，这一部分增加了调用进程调度过程

```

1  Timer:
2      push ax
3      push bx
4      push cx
5      push dx
6      push bp
7      push es
8
9      dec byte ptr es:[count]          ; 递减计数变量
10     jnz End1                        ; >0: 跳转
11     inc byte ptr es:[dir]           ; dir表示风火轮方向
12     cmp byte ptr es:[dir],1
13     jz dir1
14     cmp byte ptr es:[dir],2
15     jz dir2
16     cmp byte ptr es:[dir],3
17     jz dir3
18     jmp show

```

跳转的 **End1** 如下，首先将寄存器 **AL** 置为 **EOI**，再将 **EOI** 发送到主、从 **8529A**，再恢复寄存器信息

```

1  End1:
2      mov al,20h                      ; AL = EOI
3      out 20h,al                      ; 发送EOI到主8529A
4      out 0A0h,al                    ; 发送EOI到从8529A
5
6      pop ax                          ; 恢复寄存器信息
7      mov es,ax
8      pop bp
9      pop dx
10     pop cx
11     pop bx
12     pop ax

```



```

13     iret
14
15     str1 db '/ | \'
16     str2 db '| \ /'
17     str3 db '\ / |'
18     delayT equ 5 ; 计时器延迟计数
19     count db delayT ; 计时器计数变量

```

- 关于进程控制块 **PCB**

- 新进程段基址

```

1 int current_Seg = 0x2000;

```

- 枚举出进程的四种状态

```

1 typedef enum Status{NEW,READY,RUNNING,EXIT}Status;

```

- 定义寄存器的结构体

```

1 typedef struct RegisterImage
2 {
3     int SS;
4     int GS;
5     int FS;
6     int ES;
7     int DS;
8     int DI;
9     int SI;
10    int BP;
11    int SP;
12    int BX;
13    int DX;
14    int CX;
15    int AX;
16    int IP;
17    int CS;
18    int FLAG;
19 } RegisterImage;

```

- **PCB** 结构体，将上面两个结构组合

```

1 typedef struct PCB
2 {
3     RegisterImage regImg;
4     Status ProcessStatus;
5 }PCB;
6
7 PCB PCB_Queue[MAX_SIZE];

```

- 保存当前进程控制块

```

1 void SavePCB(int gs,int fs,int es,int ds,int di,int si,int bp,int sp,
2             int dx,int cx,int bx,int ax,int ss,int ip,int cs,int flag)

```

```

3 {
4     PCB_Queue[CurPCBNum].regImg.AX = ax;
5     PCB_Queue[CurPCBNum].regImg.BX = bx;
6     PCB_Queue[CurPCBNum].regImg.CX = cx;
7     PCB_Queue[CurPCBNum].regImg.DX = dx;
8     PCB_Queue[CurPCBNum].regImg.SP = sp;
9     PCB_Queue[CurPCBNum].regImg.BP = bp;
10    PCB_Queue[CurPCBNum].regImg.SI = si;
11    PCB_Queue[CurPCBNum].regImg.DI = di;
12    PCB_Queue[CurPCBNum].regImg.DS = ds;
13    PCB_Queue[CurPCBNum].regImg.ES = es;
14    PCB_Queue[CurPCBNum].regImg.FS = fs;
15    PCB_Queue[CurPCBNum].regImg.GS = gs;
16    PCB_Queue[CurPCBNum].regImg.SS = ss;
17    PCB_Queue[CurPCBNum].regImg.IP = ip;
18    PCB_Queue[CurPCBNum].regImg.CS = cs;
19    PCB_Queue[CurPCBNum].regImg.FLAG = flag;
20 }

```

■ 进程调度，进行进程轮转

```

1 void Schedule()
2 {
3     /* 当前进程转为就绪态 */
4     PCB_Queue[CurPCBNum].ProcessStatus = READY;
5     /* 切换下一个进程 */
6     CurPCBNum ++;
7     if( CurPCBNum > processNum )
8         CurPCBNum = 1;
9     /* 切换后进程转为运行态*/
10    if( PCB_Queue[CurPCBNum].ProcessStatus != NEW )
11        PCB_Queue[CurPCBNum].ProcessStatus = RUNNING;
12    return;
13 }
14 PCB* Current_Process()
15 {
16     return &PCB_Queue[CurPCBNum];
17 }

```

■ 初始化进程控制块

```

1 void PCBInit(PCB *p, int seg, int offset)
2 {
3     p->ProcessStatus = NEW;
4     p->regImg.GS = 0xb800;
5     p->regImg.ES = seg;
6     p->regImg.DS = seg;
7     p->regImg.FS = seg;
8     p->regImg.SS = seg;
9     p->regImg.DI = 0;
10    p->regImg.SI = 0;
11    p->regImg.BP = 0;
12    p->regImg.SP = offset - 4;
13    p->regImg.BX = 0;
14    p->regImg.AX = 0;
15    p->regImg.CX = 0;

```

```

16     p->regImg.DX = 0;
17     p->regImg.IP = offset;
18     p->regImg.CS = seg;
19     p->regImg.FLAG = 512;
20 }

```

■ 创建新进程

```

1 void createNewPCB()
2 {
3     if(processNum > MAX_SIZE)
4         return;
5     PCBInit( &PCB_Queue[processNum] ,processNum, current_Seg);
6     processNum++;
7     current_Seg += 0x1000;
8 }

```

• save

利用进程控制块保存当前被中断进程的现场，就是将各寄存器的值压栈，并调用前面的 `SavePCB` 和 `schedule` 函数

```

1 Save:
2     inc word ptr[Runtime]
3     cmp word ptr[Runtime],512
4     jnz Save_continue
5     mov word ptr[_CurPCBNum],0
6     mov word ptr[Runtime],0
7     mov word ptr[_processNum],0
8     mov word ptr[_Segment],2000h
9     jmp Pre
10 Save_continue:
11     push ss
12     push ax
13     push bx
14     push cx
15     push dx
16     push sp
17     push bp
18     push si
19     push di
20     push ds
21     push es
22     .386
23     push fs
24     push gs
25     .8086
26
27     mov ax,cs
28     mov ds, ax
29     mov es, ax
30     call near ptr _SavePCB
31     call near ptr _Schedule

```

• restart

从进程控制块恢复下一个进程的现场，将前面保存的各寄存器值存入寄存器再弹栈，都是 `push` 和 `pop` 就不贴代码了

- **21h** 中断这次新加入了二进制转十进制和十六进制转十进制的功能

- 二进制转十进制相对简单一些

```
1 int BIN2DEC(char* word)
2 {
3     int num = 0;
4     while( (*word) != '\0' )
5     {
6         num *= 2;
7         num += (*word) - '0';
8         word++;
9     }
10    return num;
11 }
```

- 十六进制转十进制稍麻烦，因为要判断字母

```
1 int HEX2DEC(char *phex)
2 {
3     int dec = 0;
4     while(*phex != '\0')
5     {
6         int tem = 0;
7         dec = dec*16;
8         if(*phex >= '0' && *phex <= '9')
9             tem = *phex - '0';
10        else if(*phex >= 'A' && *phex <= 'F')
11            tem = *phex - 'A' + 10;
12        else if(*phex >= 'a' && *phex <= 'f')
13            tem = *phex - 'a' + 10;
14        else
15            tem = 0;
16        dec += tem;
17        phex++;
18    }
19    return dec;
20 }
```

- 对于多用户程序同时运行，先判断程序号是否在范围内，再调用前面的函数

```
1 void process()
2 {
3     for(j=9; j < strlen(input); j++)
4     {
5         if(input[j] < '1' || input[j] > '4')
6         {
7             print("\n\rPlease input one number of 1,2,3,4!\n\n\r");
8             return ;
9         }
10    }
```

```

11     for(j=9; j < strlen(input); j++)
12     {
13         if(input[j] == ' ')
14             continue;
15         else if(input[j]>'0'&&input[j]<='5')
16         {
17             k = input[j] - '0' + 2;
18             if( Segment > 0x6000 )
19             {
20                 print("\n\rThere have been 5 Processes !\n\n\r");
21                 break;
22             }
23             PCBInit(&PCB_Queue[CurPCBNum],Segment,0x1400);
24             another_load(Segment,k);
25             Segment += 0x1000;
26             processNum++;
27         }
28     }
29 }

```

用户程序和之前的一样，就不再叙述了

5.编译

- 首先在 **DosBox** 中使用 **TCC**、**TASM** 以及 **TLINK** 编译内核，并生成 **.com** 程序

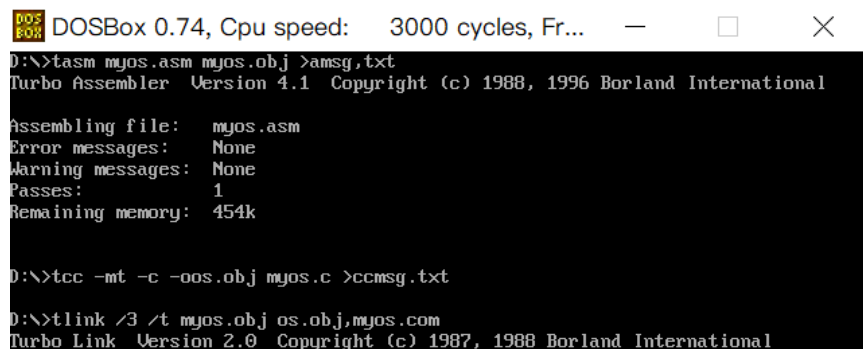
- 启动 **DosBox**，将目录挂载到 **DosBox** 的D盘并进入

```

Z:\>mount D D:\os
Drive D is mounted as local directory D:\os\
Z:\>d:\

```

- 使用 **TCC**、**TASM**、**TLINK**链接



```

DOS
FOR
DOSBox 0.74, Cpu speed: 3000 cycles, Fr...
D:\>tasm myos.asm myos.obj >msg.txt
Turbo Assembler Version 4.1 Copyright (c) 1988, 1996 Borland International

Assembling file:   myos.asm
Error messages:   None
Warning messages: None
Passes:           1
Remaining memory: 454k

D:\>tcc -mt -c -oos.obj myos.c >ccmsg.txt

D:\>tlink /3 /t myos.obj os.obj,myos.com
Turbo Link Version 2.0 Copyright (c) 1987, 1988 Borland International

```

- 剩下的汇编我使用 **NASM** 编译，并在 **Ubuntu** 下使用 **dd** 命令写入软盘

这里我使用 **Make** 自动完成创建空白软盘、**nasm** 编译引导程序、将各个程序写入扇区的工作

下面是我的 **MakeFile**

```

1  BIN = boot.bin prog1.bin prog2.bin prog3.bin prog4.bin
2  IMG = heze.img
3  all: clear ${BIN} ${IMG}
4  clear:
5      rm -f ${BIN} ${IMG}
6  %.bin: %.asm
7      nasm -fbin $< -o $@
8  %.img:
9      /sbin/mkfs.msdos -C $@ 1440
10     dd if=boot.bin of=$@ conv=notrunc
11     dd if=MYOS.COM of=$@ seek=1 conv=notrunc
12     dd if=prog1.bin of=$@ seek=20 conv=notrunc
13     dd if=prog2.bin of=$@ seek=21 conv=notrunc
14     dd if=prog3.bin of=$@ seek=22 conv=notrunc
15     dd if=prog4.bin of=$@ seek=23 conv=notrunc
16 clean:
17     rm *.bin

```

其中，先将以前生成的文件都删除，然后1.44MB软盘是用 `/sbin/mkfs.msdos -C $@ 1440` 这一句完成创建的，`nasm` 命令将所有 `.asm` 文件汇编为 `.bin` 文件，然后将所有程序都写入软盘。

编译：

```

heze@ubuntu:~/os/os6$ make
rm -f boot.bin prog1.bin prog2.bin prog3.bin prog4.bin heze.img
nasm -fbin boot.asm -o boot.bin
nasm -fbin prog1.asm -o prog1.bin
nasm -fbin prog2.asm -o prog2.bin
nasm -fbin prog3.asm -o prog3.bin
nasm -fbin prog4.asm -o prog4.bin
/sbin/mkfs.msdos -C heze.img 1440
mkfs.fat 4.1 (2017-01-24)
dd if=boot.bin of=heze.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.000119486 s, 4.3 MB/s
dd if=MYOS.COM of=heze.img seek=1 conv=notrunc
13+1 records in
13+1 records out
6719 bytes (6.7 kB, 6.6 KiB) copied, 0.000148822 s, 45.1 MB/s
dd if=prog1.bin of=heze.img seek=20 conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.000114259 s, 4.5 MB/s
dd if=prog2.bin of=heze.img seek=21 conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.000113081 s, 4.5 MB/s
dd if=prog3.bin of=heze.img seek=22 conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.000110713 s, 4.6 MB/s
dd if=prog4.bin of=heze.img seek=23 conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.00011185 s, 4.6 MB/s

```

可见所有的命令都自动执行了。

6.运行

- 进入后首界面

```

Welcome to HeZe's operating system.
Just input the name of the instructions.

ls: Show the files's size and name
clean: Clear the screen          time: Get the time
author: Show the author of this operating system
run: Run any number of program  e.g: run 2 or run 2431
run_plus: Run programs at the same time e.g: run_plus 1234
int 33h    int 34h    int 35h    int 36h
int 21h+function_number e.g: int 21h 1
heze-os#

```

- 显示文件信息和时间

```

author: Show the author of this operating system
run: Run any number of program  e.g: run 2 or run 2431
run_plus: Run programs at the same time e.g: run_plus 1234
int 33h    int 34h    int 35h    int 36h
int 21h+function_number e.g: int 21h 1
heze-os#ls
-----
:   name   :  segNum  :   size  :
:   prog1   :    20    :  4058B  :
:   prog2   :    21    :  4069B  :
:   prog3   :    22    :  4076B  :
:   prog4   :    23    :  4063B  :
-----
heze-os#time
The time is: 2020/06/30 20:18:26
heze-os#_

```

- 软中断，这里以 34h 为例：

This is int 34h

- 系统调用，输出输入字符、大小写转换、二进制转十进制

```

heze-os#int 21h 1

input your string:
DDDggg

21h_1 print:
DDDggg
heze-os#int 21h 2

Please input a sentence:
DDDggg
DDDGGG
heze-os#int 21h 3

Please input a sentence:
DDDggg
dddggg
heze-os#int 21h 4

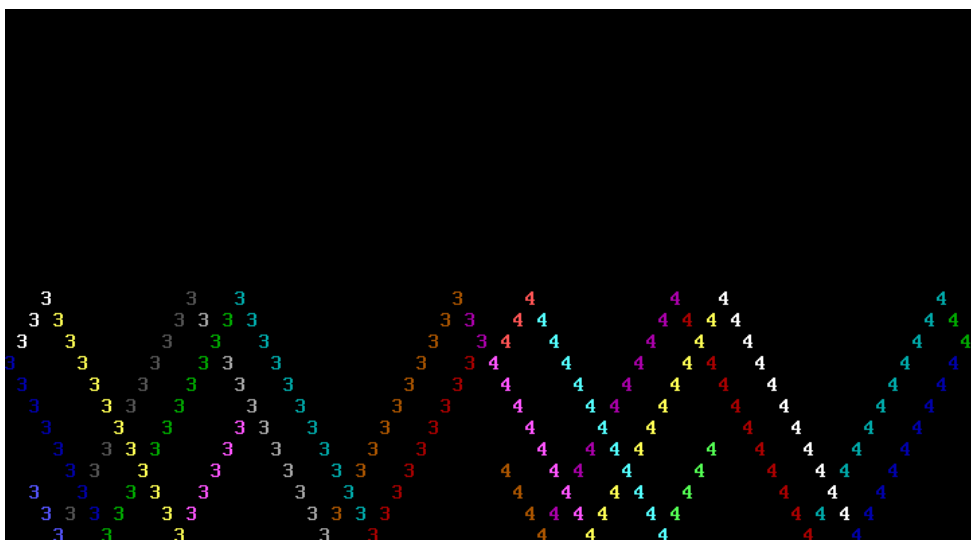
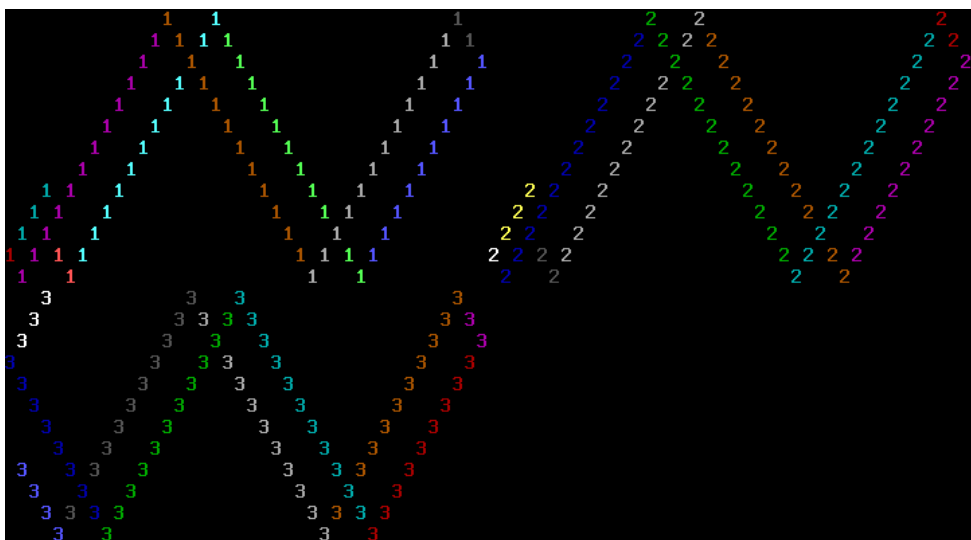
please input your bin number:

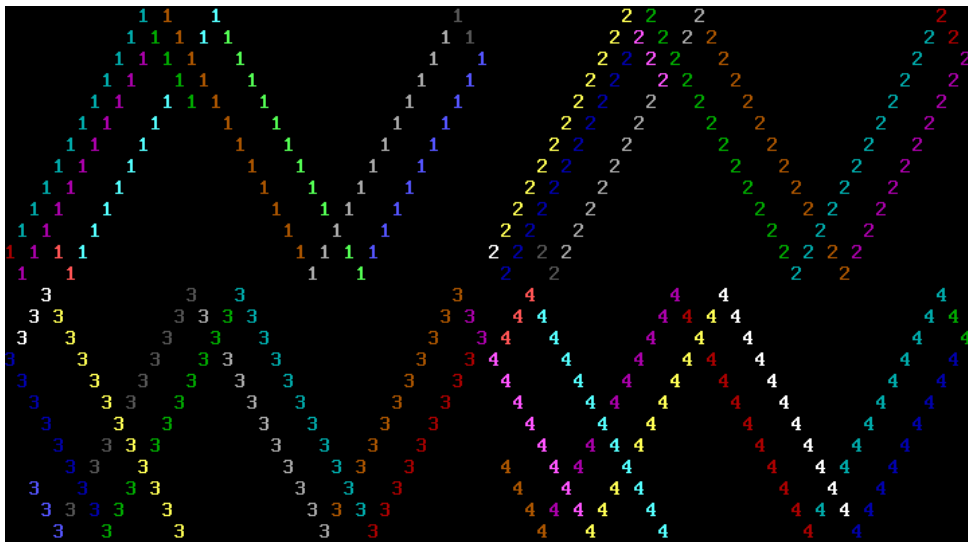
0110

6
heze-os#_

```

- 同时运行用户程序





六、创新工作

1. 使用 `Make`，各种命令自动执行
2. `21h` 系统调用除了大小写转换还新加入了二进制、十六进制转十进制的功能

七、实验总结

这次实验涉及的知识量还是很多的，从进程的模型、进程表这些理论知识，再到如何保护现场，之后如何恢复现场的方法，我刚开始看了很久，也查了很多资料才完全搞懂这里面的知识，但我认为这次试验不像之前可能会有各种各样的小问题，这次只要对相关的理论知识有了充足的了解再去实现就会很容易，虽然可能会比较麻烦，但有了很明确的设计、实现方向再加上之前编程的经验（比如要时刻保护寄存器）之后，只要实现完成了，便基本上没什么问题了，我这次就是这样，虽然花了很多时间学习理论概念，但实现完成后并没有什么大问题，程序运行的也比较好，也没有什么玄学bug。

通过这次试验我的收获也蛮多的，对于进程的模型有了很深入的了解。