

# 武汉大学国家网络安全学院 教学实验报告



课程名称	操作系统设计与实践	实验日期	2024 年 10 月 25 日
实验名称	内核雏形	实验周次	6
姓名	学号	专业	班级
黄东威	2022302181148	信息安全	5 班
王浚杰	2022302181143	网络空间安全	5 班
程序	2022302181131	信息安全	4 班
周业营	2022302181145	信息安全	5 班

# Contents

<b>1 实验目的及实验内容</b>	<b>4</b>
<b>2 实验环境</b>	<b>5</b>
<b>3 实验步骤</b>	<b>5</b>
3.1 添加汇编和 C 语言的互相调用 . . . . .	5
3.2 分析 ELF 文件格式 . . . . .	7
3.3 使用 Loader 加载 ELF 文件 . . . . .	9
3.3.1 加载内核到内存 . . . . .	9
3.3.2 跳入保护模式 . . . . .	11
3.4 分析代码结构，研究如何加载并扩展内核，对比真正内核源码的代码组织情况 . . . . .	13
3.4.1 分析内核代码结构 . . . . .	13
3.4.2 对比真正内核源码的代码组织情况 . . . . .	14
3.5 设计题：修改启动代码，在引导过程中在屏幕上画出一个你喜欢的 ASCII 图案，并将第三章的内存管理功能代码、你自己设计的中断代码集成到你的 kernel 文件目录管理中，并建立 makefile 文件，编译成内核，并引导 . . . . .	15
3.5.1 在屏幕上画出一个你喜欢的 ASCII 图案 . . . . .	15
3.5.2 内存管理功能代码集成 . . . . .	17
3.5.3 中断代码集成 . . . . .	24
<b>4 实验过程分析</b>	<b>25</b>
4.1 64 位开发平台不兼容 32 位汇编 . . . . .	25
4.2 栈保护错误 . . . . .	26
4.3 Makefile 中 make all 指令未集成镜像挂载指令 . . . . .	27
4.4 disp_str 函数错误 . . . . .	27
4.5 键盘中断问题 . . . . .	28
4.6 内存管理问题 . . . . .	28
<b>5 实验结果总结</b>	<b>30</b>
5.1 汇编和 C 内定义的函数，相互间调用的方法是怎样的？ . . . . .	30
5.1.1 从 C 调用汇编函数 . . . . .	30
5.1.2 从汇编调用 C 函数 . . . . .	30
5.1.3 C 调用约定（C Calling Convention） . . . . .	31
5.1.4 以示例代码为例分析 C 和汇编相互调用关系 . . . . .	31
5.2 描述 ELF 文件格式以及作用，和大家学习的 PE 相比，结构上有什么相同和差异？ . . . . .	31
5.2.1 ELF 文件格式 . . . . .	31
5.2.2 ELF 文件格式作用 . . . . .	34
5.2.3 ELF 文件格式和 PE 文件格式的相同与差异 . . . . .	34
5.3 如何从 Loader 引导 ELF 的原理 . . . . .	35
5.4 对照书中例程代码，这个内核扩展了哪些功能，这些功能流程是怎样的，他们都是在哪些源文件的代码中进行描述的？这些功能彼此有相互关联吗，给出说明？ . . . . .	35
5.5 书中代码内存的布局是怎样的？在这里有哪些是特权代码，哪些是非特权代码，在处理器控制权切换时，权限变化情况如何？ . . . . .	36
5.6 下载一个真正的内核源文件，分析一下是怎么管理组织源码文件的。 . . . . .	38
5.7 完成设计题并能演示。 . . . . .	38

5.8 附加题：当你阅读到 chapter5/f/start.c 时，请留意是如何对 gdt 进行修改的？	39
5.9 实验改进意见 . . . . .	40
<b>6 各人实验贡献与体会（每人各自撰写）</b>	<b>41</b>
<b>7 教师评语</b>	<b>42</b>

# 1 实验目的及实验内容

**实验目标：**

1. 如何生成一个内核，能引导该内核
2. 对应章节：5.1-5.5

**实验内容：**

1. 汇编和 C 的互相调用方法
  - (a) 在例程基础上，在汇编与 C 程序中各添加一个简单带参数的函数调用，让两种语言撰写的程序实现混合调用，功能可自定义。
2. ELF 文件格式
  - (a) 依照书上方法，分析你修改的这个可执行文件
3. 使用 Loader 加载 ELF 文件
  - (a) 阅读书中给出的代码结构，研究如何加载并扩展内核，对比研究一下真正内核源码的代码组织情况
4. 设计题：修改启动代码，在引导过程中在屏幕上画出一个你喜欢的 ASCII 图案，并将第三章的内存管理功能代码、你自己设计的中断代码集成到你的 kernel 文件目录管理中，并建立 makefile 文件，编译成内核，并引导

**需要回答的问题：**

1. 汇编和 C 内定义的函数，相互间调用的方法是怎样的？
2. 描述 ELF 文件格式以及作用，和大家学习的 PE 相比，结构上有什么相同和差异？
3. 如何从 Loader 引导 ELF 的原理？
4. 对照书中例程代码，这个内核扩展了哪些功能，这些功能流程是怎样的，他们都是在哪些源文件的代码中进行描述的？这些功能彼此有相互关联吗，给出说明？
5. 书中代码内存的布局是怎样的？在这里有哪些是特权代码，哪些是非特权代码，在处理器控制权切换时，权限变化情况如何？
6. 下载一个真正的内核源文件，分析一下是怎么在管理组织源码文件的。
7. 完成设计题并能演示。

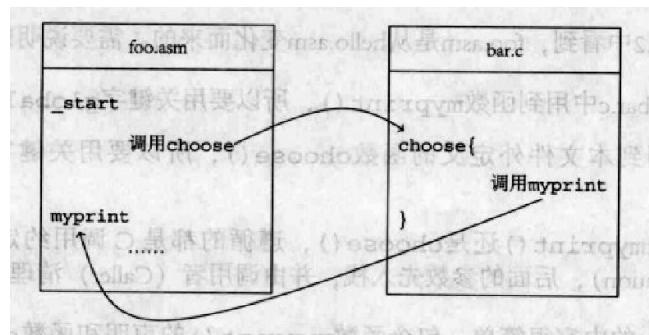
## 2 实验环境

WSL 2  
Ubuntu 20.04  
nasm 2.14.02  
VSCode  
Bochs 2.7

## 3 实验步骤

### 3.1 添加汇编和 C 语言的互相调用

源代码文件包含 foo.asm 和 bar.c，原来的互相调用情况如下图所示：



添加一个 isprime 函数在 bar.c 中用于判断传入的 num1st 是否为素数，同时调用 foo.asm 中的 myprint 函数用于展示计算结果

```
1 void myprint(char *msg, int len);  
2  
3 int choose(int a, int b)  
4 {  
5     if (a >= b)  
6     {  
7         myprint("the 1st one\n", 13);  
8     }  
9     else  
10    {  
11        myprint("the 2nd one\n", 13);  
12    }  
13  
14    return 0;  
15}  
16  
17 void isprime(int a) {  
18     if(a == 2){  
19         myprint("isprime\n",9);  
20     }
```

```

20         return;
21     }else{
22         for(int i = 2; i < a; i++){
23             if(a % i == 0){
24                 myprint("not prime\n",11);
25                 return;
26             }
27         }
28         myprint("isprime\n",9);
29     }
30 }
```

在 foo.asm 中导入外部函数 isprime

```

1 extern choose ; int choose(int a, int b);
2 extern isprime ; void fb(int a);
3 [section .data] ; 数据在此
4 num1st    dd 114
5 num2nd    dd 4
6
7 [section .text] ; 代码在此
8
9 global _start ; 我们必须导出 _start 这个入口，以便让链接器识别。
10 global myprint ; 导出这个函数为了让 bar.c 使用
11 _start:
12     push    dword [num2nd] ; `.
13     push    dword [num1st] ; /
14     ;call   choose       ; / choose(num1st, num2nd);
15     call    isprime
16     add esp, 8           ; /
17
18     mov ebx, 0
19     mov eax, 1           ; sys_exit
20     int 0x80            ; 系统调用
21
22 ; void myprint(char* msg, int len)
23 myprint:
24     mov edx, [esp + 8]   ; len
25     mov ecx, [esp + 4]   ; msg
26     mov ebx, 1
27     mov eax, 4           ; sys_write
28     int 0x80            ; 系统调用
29     ret
```

开始时将 num1st 设置为 114，预期为不是素数

```

● wisdomgo@DESKTOP-KITILU9 ~/c/o/o/c/i (master)> cd ../b
● wisdomgo@DESKTOP-KITILU9 ~/c/o/o/c/b (master)> ls
Makefile bar.c foo.asm foobar
● wisdomgo@DESKTOP-KITILU9 ~/c/o/o/c/b (master)> ./foobar
not prime
● wisdomgo@DESKTOP-KITILU9 ~/c/o/o/c/b (master)>
```

符合预期，我们重新将 num1st 设置为 5，预期为是素数，经过修改和重新 make 运行后结果如下：

```

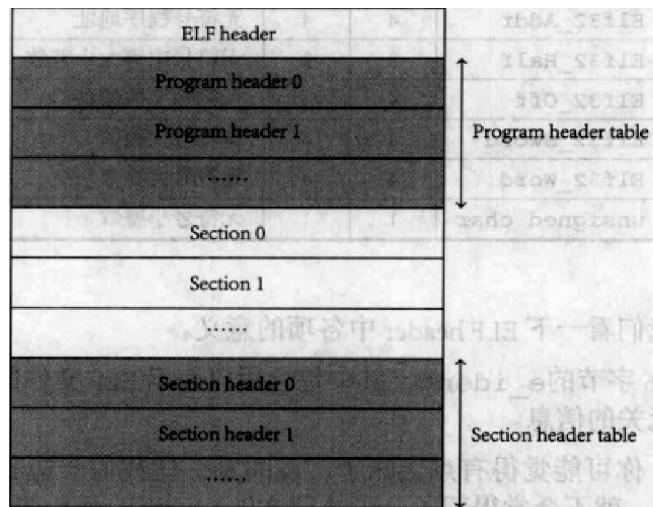
● wisdomg@DESKTOP-KITILU9 ~/c/o/o/c/b (master)> ls
Makefile bar.c foo.asm foobar
● wisdomg@DESKTOP-KITILU9 ~/c/o/o/c/b (master)> make
nasm -f elf -o foo.o foo.asm
gcc -c -m32 -o bar.o bar.c
ld -s -m elf_i386 -o foobar foo.o bar.o
● wisdomg@DESKTOP-KITILU9 ~/c/o/o/c/b (master)> ./foobar
isprime
● wisdomg@DESKTOP-KITILU9 ~/c/o/o/c/b (master)>

```

符合预期

## 3.2 分析 ELF 文件格式

ELF 文件格式如下图所示



ELF 文件由 4 部分组成，分别是 ELF 头，程序表头，节和节头表，使用 010editor 打开可执行文件 foobar 进行分析：

先针对 ELF 文件格式定义新的类型说明：

名称	大小	对齐	用途
Elf32_Addr	4	4	无符号程序地址
Elf32_Half	2	2	无符号中等大小整数
Elf32_Off	4	4	无符号文件偏移
Elf32_Sword	4	4	有符号大整数
Elf32_Word	4	4	无符号大整数
unsigned char	1	1	无符号小整数

在此基础上，ELF header 的格式如下代码所示：

```

1 typedef struct
2 {
3     unsigned char e_ident[16];
4     Elf32_Half e_type;
5     Elf32_Half e_machine;
6     Elf32_Word e_version;
7     Elf32_Addr e_entry;
8     Elf32_Off e_phoff;

```

```

9     Elf32_Off e_shoff;
10    Elf32_Word e_flags;
11    Elf32_Half e_ehsize;
12    Elf32_Half e_phentsize;
13    Elf32_Half e_phnum;
14    Elf32_Half e_shentsize;
15    Elf32_Half e_shnum;
16    Elf32_Half e_shstrndx;
17 } Elf32_Ehdr;

```

通过 010editor，我们可以查看 ELF header 各项的值及含义：着重分析集中几个项的含义：

	Offset	Size	Type	Description
e_ident	0h	14h	struct	The main elf header basically tell.
e_type	0h	10h	struct e_iden...	Object file type
e_machine	0h	10h	enum e_type...	Object file machine
e_version	EV_CURRENT (1)	4h	enum e_ver...	Object file version
e_entry	0x8048000	4h	Elf32_Addr	Entry point virtual address
e_phoff	52h	4h	Elf32_Off	Program header table file offset
e_shoff	12412	20h	Elf32_Off	Section header table file offset
e_flags	0	4h	Elf32_Word	Processor-specific flags
e_ehsize	32h	2h	Elf32_Half	Program header table entry size
e_phentsize	32	2h	Elf32_Half	Program header table entry count
e_phnum	5	2h	Elf32_Half	Program header table entry count
e_shentsize	40h	2h	Elf32_Half	Section header table entry size
e_shnum	8	2h	Elf32_Half	Section header table entry count
e_shstrndx	7	2h	Elf32_Half	Section header string table index

- e\_type 标识该文件的类型，这里 foobar 的值为 0x2，表示是一个可执行文件
- e\_entry 表示程序的入口地址 foobar 的入口地址为 0x80480A0
- e\_phoff 表示 Program header table 在文件中的偏移量，这里的值为 0x34
- e\_shoff 表示 Section header table 在文件中的偏移量，这里的值为 0x1C0

Programheader 的格式如下面代码所示：

```

1  typedef struct {
2      Elf32_Word p_type;
3      Elf32_Off p_offset;
4      Elf32_Addr p_vaddr;
5      Elf32_Addr p_paddr;
6      Elf32_Word p_filesz;
7      Elf32_Word p_memsz;
8      Elf32_Word p_flags;
9      Elf32_Word p_align;
10 } Elf32_Phdr;

```

	Offset	Size	Type	Description
program_header_table	34h	4h	struct	Program headers - describes th...
program_header_element[0]	34h	4h	struct progra...	
p_type	(R_,) C loadable Se_	4h	enum p_type...	Segment type
p_offset	0h	4h	Elf32_Off	Segment file offset
p_vaddr	0x80480000	4h	Elf32_Addr	Segment physical address
p_paddr	0x80480000	4h	Elf32_Addr	Segment physical address
p_filesz	212	4h	Elf32_Word	Segment size in file
p_memsz	212	4h	Elf32_Word	Segment size in memory
p_flags	PF_R (4)	4h	enum p_flag...	Segment flags
p_align	4096	5h	Elf32_Word	Segment alignment
program_header_element[1]	54h	20h	struct progra...	
program_header_element[2]	74h	20h	struct progra...	

其中各项的含义如下：

- p\_type type: 当前 Program header 所描述的段的类型。
- p\_offset offset: 段的第一个字节在文件中的偏移。
- p\_vaddr vaddr: 段的第一个字节在内存中的虚拟地址。
- p\_paddr paddr: 在物理地址定位相关的系统中，此项是为物理地址保留。

- p\_filesz filesz: 段在文件中的长度。
- p\_memsz memsz: 段在内存中的长度。
- p\_flags flags: 与段相关的标志。
- p\_align align: 根据此项值来确定段在文件以及内存中如何对齐。

书上给出了 foobar 的三个 Program header 的各个项的取值

表 5.2 Program header			
名称	Program header 0	Program header 1	Program header 2
p_type	0x1	0x1	0x6474E551
p_offset	0x0	0x134	0
p_vaddr	0x8048000	0x8049134	0
p_paddr	0x8048000	0x8049134	0
p_filesz	0x131	0x8	0
p_memsz	0x131	0x8	0
p_flags	0x5	0x6	0x7
p_align	0x1000	0x1000	0x4

### 3.3 使用 Loader 加载 ELF 文件

Loader 需要做两项工作

- 加载内核到内存
- 跳入保护模式

#### 3.3.1 加载内核到内存

使用 Loader 按照寻找文件、定位文件以及读入内存的步骤把内核放入到内存中

- 寻找文件，首先在 A 盘的根目录寻找 KERNEL.BIN

```

1      ; 下面在 A 盘的根目录寻找 KERNEL.BIN
2      mov word [wSectorNo], SectorNoOfRootDirectory
3      xor ah, ah ; `.
4      xor dl, dl ; / 软驱复位
5      int 13h ; /
6 LABEL_SEARCH_IN_ROOT_DIR_BEGIN:
7      cmp word [wRootDirSizeForLoop], 0 ; `.
8      jz LABEL_NO_KERNELBIN ; / 判断根目录区是不是已经读完,
9      dec word [wRootDirSizeForLoop] ; / 读完表示没有找到 KERNEL.BIN
10     mov ax, BaseOfKernelFile
11     mov es, ax ; es <- BaseOfKernelFile
12     mov bx, OffsetOfKernelFile ; bx <- OffsetOfKernelFile
13     mov ax, [wSectorNo] ; ax <- Root Directory 中的某 Sector 号
14     mov cl, 1
15     call ReadSector
16
17     mov si, KernelFileName ; ds:si -> "KERNEL BIN"
18     mov di, OffsetOfKernelFile
19     cld

```

```

20     mov dx, 10h
21 LABEL_SEARCH_FOR_KERNELBIN:
22     cmp dx, 0
23     jz LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR; / 循环次数控制，如果已经
24     读完
25     dec dx ; / 了一个 Sector, 就跳到下一个
      mov cx, 11

```

- 定位文件

```

1 LABEL_FILENAME_FOUND: ; 找到 KERNEL.BIN 后便来到这里继续
2     mov ax, RootDirSectors
3     and di, OFFF0h ; di -> 当前条目的开始
4
5     push eax
6     mov eax, [es : di + 01Ch] ; `.
7     mov dword [dwKernelSize], eax ; / 保存 KERNEL.BIN 文件大小
8     pop eax
9
10    add di, 01Ah ; di -> 首 Sector
11    mov cx, word [es:di]
12    push cx ; 保存此 Sector 在 FAT 中的序号
13    add cx, ax
14    add cx, DeltaSectorNo ; cl <- LOADER.BIN 的起始扇区号 (0-based)
15    mov ax, BaseOfKernelFile
16    mov es, ax ; es <- BaseOfKernelFile
17    mov bx, OffsetOfKernelFile ; bx <- OffsetOfKernelFile
18    mov ax, cx ; ax <- Sector 号

```

- 使用 InitKernel 函数按照 ELF 文件格式解析二进制文件，并将内核对应分区放到内存的对应分区。

```

; InitKernel: 将 KERNEL.BIN 的内容经过整理对齐后放到新的位置
;
InitKernel: ; 遍历每一个 Program Header, 根据 Program Header 中的信息来确定把什么放进内存, 放到什么位置, 以及放多少
    xor esi, esi
    mov cx, word [BaseOfKernelFilePhyAddr + 2Ch]; | ecx <- pELFHdr->e_phnum
    movzx ecx, cx ; |
    mov esi, [BaseOfKernelFilePhyAddr + 1Ch]; | ; esi <- pELFHdr->e_phoff
    add esi, BaseOfKernelFilePhyAddr ; esi <- OffsetOfKernelFile + pELFHdr->e_phoff
    .Begin:
        mov eax, [esi + 0]
        cmp eax, 0 ; PT_NULL
        jz .NoAction
        push dword [esi + 010h]; size |
        mov eax, [esi + 04h]; |
        add eax, BaseOfKernelFilePhyAddr ; | ; :memcpy( (void*)pPHdr->p_vaddr,
        push eax ; src | uchCode + pPHdr->_offset,
        push dword [esi + 08h]; dst | pPHdr->p_filesz;
        call MemCopy ; |
        add esp, 12 ; |
        .NoAction:
            add esi, 020h ; esi += pELFHdr->e_phentsize
            dec ecx
            jnz .Begin

```

- 加载内核，书上使用了一个 kernel.asm 作为内核

```

1 [section .text] ; 代码在此
2
3 global _start ; 导出 _start
4
5 _start: ; 跳到这里来的时候, 我们假设 gs 指向显存

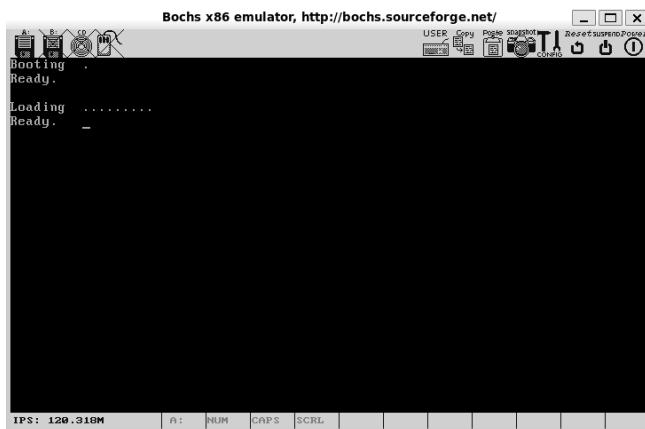
```

```

6   mov ah, 0Fh           ; 0000: 黑底      1111: 白字
7   mov al, 'K'
8   mov [gs:((80 * 1 + 39) * 2)], ax    ; 屏幕第 1 行，第 39 列。
9   jmp $

```

代码实现了打印了 Ready，但此时还未进入保护模式，所以没有打印'K' 字符



### 3.3.2 跳入保护模式

在 Loader 由自己加载后，段地址被确定为了 BaseOfLoader，所以在 Loader 中的物理地址可以用下面的公式来表示：

标号（变量）的物理地址 = BaseOfLoader \* 10h + 标号（变量）的偏移  
将 BaseOfLoader 定义在文件 load.inc 中

```

1 BaseOfLoader      equ 09000h ; LOADER.BIN 被加载到的位置 ----- 段地址
2 OffsetOfLoader   equ 0100h ; LOADER.BIN 被加载到的位置 ----- 偏移地址
3
4 BaseOfLoaderPhyAddr equ BaseOfLoader*10h ; LOADER.BIN 被加载到的位置 -----
   物理地址
5
6 BaseOfKernelFile  equ 08000h ; KERNEL.BIN 被加载到的位置 ----- 段地址
7 OffsetOfKernelFile equ 0h ; KERNEL.BIN 被加载到的位置 ----- 偏移地址

```

由实模式跳入保护模式后，打印一个' P' 字符

```

1 [SECTION .s32]
2
3 ALIGN 32
4
5 [BITS 32]
6
7 LABEL_PM_START:
8     mov ax, SelectorVideo
9     mov gs, ax
10
11    mov ax, SelectorFlatRW
12    mov ds, ax
13    mov es, ax
14    mov fs, ax

```

```

15    mov ss, ax
16    mov esp, TopOfStack
17
18    push    szMemChkTitle
19    call    DispStr
20    add esp, 4
21
22    call    DispMemInfo
23    call    SetupPaging
24
25    mov ah, 0Fh           ; 0000: 黑底      1111: 白字
26    mov al, 'P'
27    mov [gs:((80 * 0 + 39) * 2)], ax   ; 屏幕第 0 行, 第 39 列。
28    jmp $

```

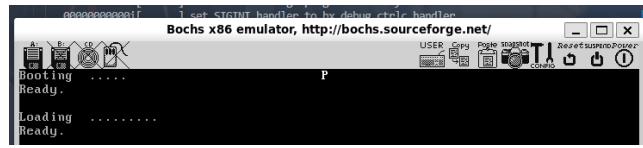
接下来的就是从实模式跳入保护模式的代码:

```

1 LABEL_FILE_LOADED:
2
3     call KillMotor          ; 关闭软驱马达
4
5     mov dh, 1               ; "Ready."
6     call DispStrRealMode    ; 显示字符串
7
8     ; 下面准备跳入保护模式
9
10    ; 加载 GDTR
11    lgdt [GdtPtr]
12
13    ; 关中断
14    cli
15
16    ; 打开地址线 A20
17    in al, 92h
18    or al, 00000010b
19    out 92h, al
20
21    ; 准备切换到保护模式
22    mov eax, cr0
23    or eax, 1
24    mov cr0, eax
25
26    ; 真正进入保护模式
27    jmp dword SelectorFlatC:(BaseOfLoaderPhyAddr+LABEL_PM_START)
28
29    jmp $

```

结果如图所示:



## 3.4 分析代码结构，研究如何加载并扩展内核，对比真正内核源码的代码组织情况

### 3.4.1 分析内核代码结构

该内核的代码结构如图所示：

```
wisdomg@DESKTOP-KITILU9 ~/c/o/o/c/i (master)> tree
.
├── Makefile
├── Makefile.1
├── a.img
└── bochsrc
    ├── boot
    │   ├── boot.asm
    │   ├── boot.bin
    │   └── include
    │       ├── fat12hdr.inc
    │       ├── load.inc
    │       └── pm.inc
    ├── loader.asm
    └── loader.bin
    └── include
        ├── const.h
        ├── global.h
        ├── protect.h
        ├── proto.h
        ├── string.h
        └── type.h
    └── kernel
        ├── global.c
        ├── global.o
        ├── i8259.c
        ├── i8259.o
        ├── kernel.asm
        ├── kernel.o
        ├── protect.c
        ├── protect.o
        ├── start.c
        └── start.o
    └── kernel.bin
    └── lib
        ├── klib.c
        ├── klib.o
        ├── kliba.asm
        ├── kliba.o
        ├── string.asm
        └── string.o
5 directories, 34 files
wisdomg@DESKTOP-KITILU9 ~/c/o/o/c/i (master)>
```

其中各个部分及其作用如下：

- Makefile 构建文件，包含编译和链接整个项目所需的规则和命令。
- Makefile.1 备用的 Makefile
- a.img 磁盘镜像文件，用于模拟软盘。
- bochsrc Bochs 的配置文件。
- boot 该文件夹包含与操作系统的引导程序相关的代码。引导程序负责将内核加载到内存中，完成初步的系统初始化，然后将控制权移交给内核。
- include 包含操作系统内核部分的头文件。

- **kernel** 该文件夹包含操作系统内核的代码。
- **kerenel.bin** 内核的可执行二进制文件，由内核代码（汇编和 C 文件）编译、链接后生成。
- **lib** 该文件夹包含操作系统的库代码，提供了基本的功能支持，类似于操作系统内核的标准库。

### 3.4.2 对比真正内核源码的代码组织情况

我们从 github 上查看了真正的 linux kernel 的项目结构：

- **arch/**: 架构特定代码。包含对不同 CPU 架构（如 x86、ARM、MIPS 等）的支持，每个子目录对应一种架构。各架构文件夹内有特定于该架构的启动代码、汇编指令、系统调用实现、内存管理等。
- **block/**: 块设备层代码。提供块设备（如硬盘、SSD 等）通用的基础设施和接口，包括块层缓存、I/O 调度器（如 CFQ、Deadline 等），以及块设备请求的处理逻辑。
- **certs/**: 系统签名和加密密钥相关代码，通常用于内核模块签名和验证。
- **crypto/**: 内核的加密 API。包含加密算法（如 AES、SHA）和加密框架，供文件系统、网络和其他模块使用。
- **Documentation/**: 内核文档。提供详细的内核子系统、配置选项、API 说明等文档，帮助开发人员理解和使用内核代码。
- **drivers/**: 设备驱动代码。支持各种硬件设备的驱动，包括：
  - **net/**: 网络设备驱动。
  - **gpu/**: 图形处理器驱动。
  - **usb/**: USB 设备驱动。
  - **char/** 和 **block/**: 字符设备和块设备驱动。
  - 还有许多其他子目录（如 **input**、**media**、**sound** 等），对应不同类型的设备。
- **fs/**: 文件系统代码。实现各种文件系统支持，包括：
  - **本地文件系统**: 如 EXT4、XFS、Btrfs。
  - **网络文件系统**: 如 NFS、CIFS。
  - **伪文件系统**: 如 **proc**（进程文件系统）、**sysfs**（系统文件系统）。
- **include/**: 头文件。包含全局的内核头文件，如内核 API、数据结构定义、常量和宏。头文件通常分为：
  - **include/linux/**: 大部分 Linux 内核子系统的头文件。
  - **include/uapi/**: 用户空间 API (uapi) 头文件，供用户态应用程序使用。
  - **include/asm/**: 汇编代码，包含架构相关的汇编头文件。
- **init/**: 内核初始化代码。主要包括内核启动和初始化的代码，负责在启动阶段完成内核初始化并启动第一个用户空间进程（如 init）。

- **ipc/**: 进程间通信 (IPC) 代码。包括 System V IPC 机制 (信号量、共享内存和消息队列) 以及其他内核 IPC 实现。
- **kernel/**: 内核核心代码。实现核心调度器、内存管理、定时器、系统调用处理等基础功能，包括：
  - **进程管理**: 如 `sched/` 调度器和 `pid.c` (进程 ID 管理)。
  - **系统调用**: 如 `sys.c`, 负责系统调用的分发和处理。
  - **中断管理**: 如 `irq/`, 负责内核中断处理机制。
- **lib/**: 内核通用库代码。提供内核常用的库函数和算法实现，如哈希、字符串操作、压缩算法等。
- **mm/**: 内存管理代码。实现内存分配、分页管理、虚拟内存和缓存等，涉及内核的核心内存管理机制，如 `slab` 分配器、`kmalloc`、页表管理等。
- **net/**: 网络协议栈代码。实现 TCP/IP 协议栈和其他网络协议 (如 UDP、ICMP、IPv6 等)，还包含网络防火墙 (如 `netfilter`)、网络命名空间和网络调度。
- **scripts/**: 构建和调试脚本。包含用于构建内核的脚本、配置工具以及代码生成工具。
- **security/**: 内核安全模块。实现 SELinux、AppArmor、Integrity Measurement Architecture (IMA) 等安全框架，用于加强系统安全。
- **sound/**: 音频子系统代码。支持 ALSA (高级 Linux 声音架构) 和其他音频驱动。
- **tools/**: 开发工具。包含内核调试、测试和分析工具，例如 `perf` 性能分析工具、`selftests` 测试套件。
- **usr/**: 用户空间支持代码。包含一些支持用户空间的工具和初始化镜像，例如用于构建 initramfs 的代码。
- **virt/**: 虚拟化支持代码。包含对 KVM (内核虚拟机) 的支持，实现虚拟化功能，允许内核作为 Hypervisor 来运行虚拟机。

## 3.5 设计题：修改启动代码，在引导过程中在屏幕上画出一个你喜欢的 ASCII 图案，并将第三章的内存管理功能代码、你自己设计的中断代码集成到你的 kernel 文件目录管理中，并建立 makefile 文件，编译成内核，并引导

### 3.5.1 在屏幕上画出一个你喜欢的 ASCII 图案

设计一个自定义的 ASCCALL 图案最直观的思路就是在/kernel/stact.c 文件中定义一个 void `printlogo()` 函数，在函数中定义一个 ASCCALL 字符串，再调用 `dis_str()` 函数将其输出出来。

具体操作是：在 `printlogo()` 函数中定义并赋值一个 `const u8 *wisdomgo` 的静态变量，在这里将其声明为静态变量是因为这个字符串是不可修改的常量，因此定义 `const u8 *wisdomgo` 会更安全，这样做可以防止在程序中意外地修改它。对于 `u8` 这个数据结构，在/lib/type.h 文件中已经定义好了，`typedef unsigned char u8`，即 `u8` 表示一个 8 位无符号字符，`u8 *wisdomgo` 则是一个指向 8 位无符号字符的指针。

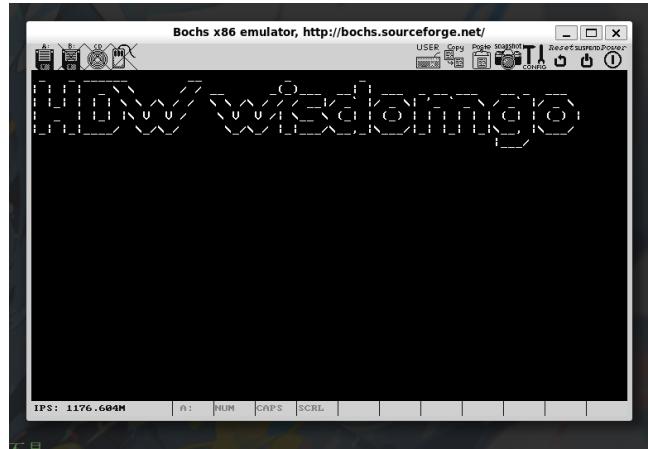
这样就可以将一个 ASCII 艺术字符串分配给 `wisdomgo` 指针。这段静态的字符串在编译时直接保存在程序的只读数据段中，`wisdomgo` 指针会指向这个字符串的第一个字符地址，方便在代码中引用和打印。所以再调用 `dis_str()` 函数即可将 ASCCALL 字符串输出出来。

除了定义 printLogo() 函数之外，因为我们需要在 kernel/start.c 中用 global 导出 printLogo 函数，在 kernel/kernel.asm 中用 extern 声明 printBootLogo 函数。

一开始的输出如下图所示，因为我们没有修改其他模块的代码，在显存上仍旧会显示在启动和内存分析部分输出的信息，这样的效果并不美观。



因此，我们回到`/boot/boot.asm`和`loader.asm`文件，将启动和内存分析部分输出的信息删去，再次运行代码。



可以看到这样显存就干净简洁了许多。

### 3.5.2 内存管理功能代码集成

我们使用第三章中实现的内存管理代码来完成此项任务。

按照 linux-kernel 项目代码的管理模式, 我们创建 mm 文件夹, 并且将 alloc\_page 和 free\_page 代码放置在 pageMange.asm 中。需要在代码中声明标号以及添加所需要的数据结构 pageMange.asm:

```

1 extern disp_str
2 extern disp_pos
3 extern printlogo
4 global alloc_pages
5 global free_pages
6
7 %include "boot/include/gdt.inc"
8 %include "boot/include/selectors.inc"
9 %include "boot/include/load.inc"
10
11 SucessMessage:      db  "Successfully alloc ^-^", 0 ; 进入保护模式后显示
   此字符串
12 OffsetSucessMessage equ SucessMessage - $$

13
14 PageFaultMessage: db "Page Fault T_T", 0
15 OffsetPageFaultMessage equ PageFaultMessage - $$

16 ; =====
17 ; 函数: alloc_pages
18 ; 功能: 为某个程序分配多个物理页
19 ; 输入: eax = 请求的页数
20 ; 输出: 分配成功返回页表基地址, 失败时 eax = 0
21 ; =====
22 alloc_pages:
23     push esi
24     push ebx
25     push ecx
26     push edi           ; 保存 edi 寄存器, 用于存储找到的页的索引
27
28     mov cx, SelectorFlatRW ; 选择扁平段选择子, 覆盖整个 32位地址空间

```

```

29      mov es, cx           ; 将选择子加载到 es 寄存器中
30
31      mov ecx, 0            ; 页表索引
32      mov ebx, BitmapBase   ; 位图基址
33      mov esi, 0             ; 已分配的物理页计数
34      lea edi, [es:FoundPages] ; 将找到的页号保存到 FoundPages 数组中
35
36 find_free_pages:
37      cmp ecx, MapLen       ; 检查是否到达位图末尾
38      jz alloc_fail          ; 没找到足够的空闲页
39
40
41      bt [es:ebx], ecx       ; 测试位图中第 ecx 位
42      jc next_page          ; 如果该页被占用，跳转到 next_page 页
43
44      ; 找到一个空闲页，记录页号
45      mov [es:edi], ecx       ; 保存找到的页号到 FoundPages 数组中
46      add edi, 4              ; 下一个页号位置
47      add esi, 1              ; 更新已分配页计数
48      cmp esi, eax           ; 检查是否分配了请求的页数 (eax = 请求的页数)
49      je alloc_success        ; 如果分配了足够的页数，跳到成功
50
51 next_page:
52      inc ecx                ; 继续检查下一个页
53      jmp find_free_pages     ; 回到循环开始
54
55 alloc_fail:
56      ; 分配失败，将之前分配的页释放
57      mov esi, 0              ; 清空已分配计数
58      lea edi, [es:FoundPages] ; 恢复页号保存指针
59      jmp alloc_done          ; 没有已分配页，直接退出
60
61      mov esi, PageFaultMessage ; 源数据偏移
62      mov edi, (80 * 12 + 0) * 2 ; 目的数据偏移。屏幕第 10 行，第 0 列。
63      call PrintString
64
65 alloc_success:
66      ; 成功分配了请求的页，标记它们为已分配
67      lea edi, [es:FoundPages] ; 恢复页号保存指针
68      mov esi, eax             ; 请求的页数
69      mov edx, PageTblBase     ; 页表基址
70      ;mov    esi, OffsetSucessMessage ; 源数据偏移
71      mov esi, SuccessMessage
72      mov edi, (80 * 10 + 0) * 2 ; 目的数据偏移。屏幕第 10 行，第 0 列。
73      call PrintString
74      ;call printlogo
75 set_allocated_pages:
76
77      mov ecx, [es:edi]         ; 读取找到的页号
78      bts dword [es:ebx], ecx     ; 将页号对应的位标记为分配

```

```

79
80
81     shl ecx, 12          ; 每个页表条目指向物理页地址，左移12位生成
82         物理地址
83     or ecx, 1           ; 设置页表条目最低有效位 (P位，表示该页有
84         效)
85     mov [es:edx], ecx   ; 写入页表条目
86
87     add edx, 4          ; 递增页表基址，指向下一个页表条目
88     add edi, 4          ; 下一个页号
89     dec esi              ; 更新已分配页数
90     jnz set_allocated_pages ; 如果还没有完成，继续
91
92
93     alloc_done:
94         pop edi
95         pop ecx
96         pop ebx
97         pop esi
98         ret
99
100
101
102     ; =====
103     ; 函数: free_pages
104     ; 功能: 释放程序使用的物理页
105     ; 输入:
106     ;       页表的起始地址。
107     ;       要释放的页数。
108     ; 输出: 无
109     ; =====
110
111     free_pages:
112         push esi
113         push ebx
114         push ecx
115         push edi          ; 保存寄存器
116
117         mov cx, SelectorFlatRW ; 选择扁平段选择子，覆盖整个32位地址空间
118         mov es, cx           ; 将选择子加载到 es 寄存器中
119
120         mov edi, ebx          ; 需要释放的页数保存到 edi
121         mov ebx, BitmapBase    ; 位图基址
122
123         ; 开始释放页
124     release_loop:
125
126         mov esi, [es:eax]      ; 读取页表条目 (PTE)
127         and dword [es:eax], 0    ; 清除页表条目

```

```

128
129     shr esi, 12          ; 提取页号 (物理地址的高20位)
130     btr dword [es:ebx], esi ; 清除位图中的对应位
131
132     ; 处理下一个页
133     add eax, 4           ; 下一个页表条目 (PTE 占 4 字节)
134     dec edi              ; 更新剩余的页数
135     jnz release_loop      ; 如果还有页需要释放，继续循环
136
137     pop edi
138     pop ecx
139     pop ebx
140     pop esi
141     ret
142
143 PrintString:
144     pusha
145     mov ax, SelectorVideo ; 选择显存段选择子
146     mov gs, ax             ; 将选择子加载到 gs 寄存器
147
148     mov ah, 0Ch             ; 设置字符属性，红字黑底
149
150     cld                   ; 清除方向标志位，设置为递增模式
151 .print_loop:
152     lodsb                 ; 从 [esi] 加载字符到 al
153     test al, al            ; 检查是否到字符串末尾
154     jz .done_print          ; 如果 al 为 0 (字符串结束)，跳转结束显示
155     mov [gs:edi], ax        ; 将字符和属性写入显存
156     add edi, 2              ; 每个字符占 2 字节，移动到下一个字符位置
157     jmp .print_loop         ; 继续显示下一个字符
158
159 .done_print:
160     popa
161     ret                   ; 返回调用处

```

为了将诸如选择子，GDT，描述符等头文件定义的符号更好的在汇编代码中调用，我们更改了项目代码结构：

```

1 Makefile
2 Makefile.1
3 a.img
4 bochssrc
5 boot
6     boot.asm
7     boot.bin
8     include
9         fat12hdr.inc
10        gdt.inc
11        load.inc
12        pm.inc
13        selectors.inc
14        loader.asm

```

```

15         loader.bin
16 include
17     const.h
18     global.h
19     protect.h
20     proto.h
21     string.h
22     type.h
23 kernel
24     global.c
25     global.o
26     i8259.c
27     i8259.o
28     kernel.asm
29     kernel.o
30     protect.c
31     protect.o
32     start.c
33     start.o
34 kernel.bin
35 lib
36     klib.c
37     klib.o
38     kliba.asm
39     kliba.o
40     string.asm
41     string.o
42 mm
43     pageManage.asm
44     pageManage.o

```

我们将 loader.asm 中的 GDT 部分和选择子部分设置部分单独拆分为了两个头文件: gdt.inc 和 selectors.inc 、 gdt.inc:

```

1 ; gdt.inc
2 ; GDT 表项描述符定义
3 %include "boot/include/pm.inc" ; 使用相对路径引入 pm.inc
4
5 LABEL_GDT:           Descriptor      0,
6                 0, 0          ; 空描述符
7                 ffffffh, DA_CR | DA_32 | DA_LIMIT_4K ; 0 ~ 4G
8 LABEL_DESC_FLAT_C:   Descriptor      0,
9                 ffffffh, DA_DRW | DA_32 | DA_LIMIT_4K ; 0 ~ 4G
10                Descriptor      0,
11                ffffffh, DA_DRW | DA_DPL3        ; 显存首地址
12
13 ; GDT 长度及指针
14 GdtLen    equ    $ - LABEL_GDT
15 GdtPtr    dw     GdtLen - 1          ; 段界限
16             dd     BaseOfLoaderPhyAddr + LABEL_GDT ; 基地址

```

selectors.inc:

```

1 ; selectors.inc
2 global SelectorFlatC
3 global SelectorFlatRW
4 global SelectorVideo
5
6 ; GDT 选择子
7 SelectorFlatC      equ LABEL_DESC_FLAT_C - LABEL_GDT
8 SelectorFlatRW     equ LABEL_DESC_FLAT_RW - LABEL_GDT
9 SelectorVideo       equ LABEL_DESC_VIDEO - LABEL_GDT + SA_RPL3

```

修改后的 loader.asm 需要去除 GDT 和 selectors，同时通过头文件导入进来

```

1 ; 引入 GDT 和选择子的定义
2 %include "gdt.inc"
3 %include "selectors.inc"
4
5 ; 下面是 FAT12 磁盘的头，之所以包含它是因为下面用到了磁盘的一些信息
6 %include "fat12hdr.inc"
7 %include "load.inc"
8 %include "pm.inc"

```

当然，mm/pageMange.asm 中不仅涉及到了 GDT 和 selctors 的调用，还需要调用 load.inc 中涉及到的段地址设置部分

load.inc:

```

1 BaseOfLoader          equ 09000h ; LOADER.BIN 被加载到的位置 ---- 段地址
2 OffsetOfLoader        equ 0100h ; LOADER.BIN 被加载到的位置 ---- 偏移地址
3
4 BaseOfLoaderPhyAddr  equ BaseOfLoader * 10h ; LOADER.BIN 被加载到的位置
      ---- 物理地址 (= BaseOfLoader * 10h)
5
6
7 BaseOfKernelFile      equ 08000h ; KERNEL.BIN 被加载到的位置 ---- 段地址
8 OffsetOfKernelFile    equ 0h ; KERNEL.BIN 被加载到的位置 ---- 偏移地址
9
10 BaseOfKernelFilePhyAddr equ BaseOfKernelFile * 10h
11 KernelEntryPointPhyAddr equ 030400h ; 注意：1、必须与 MAKEFILE 中参数 -
      Ttext 的值相等！
12 ;           2、这是个地址而非仅仅是个偏移，如果 -Ttext 的
      值为 0x400400，则它的值也应该是 0x400400。
13
14 PageDirBase          equ 200000h ; 页目录开始地址：          2M
15 PageTblBase          equ 201000h ; 页表开始地址：          2M + 4K
16
17 BitmapBase           equ 200000h ; 位图基址（每位对应一个物理页）
18 MapLen                equ 256      ; 最多 256 个物理页

```

在 kernel.asm 中加入测试函数，用来验证分页是否成功

```

1 extern alloc_pages
2 extern free_pages
3
4 TestAllocAndFree:

```

```

5    xchg bx,bx
6    mov eax,1
7    call alloc_pages
8    mov eax,ebx
9    mov ebx,1
10   xchg bx,bx
11   call free_pages
12
13   ret

```

在 start.c 中加入有关 TestAllocAndFree 函数的调用：

```

1 PUBLIC void cstart()
2 {
3     printlogo();
4     /* 将 LOADER 中的 GDT 复制到新的 GDT 中 */
5
6     memcpy(&gdt, /* New GDT */
7            ((void*)(((u32*)&gdt_ptr[2]))), /* Base of Old GDT */
8            *((u16*)&gdt_ptr[0]) + 1 /* Limit of Old GDT */
9            );
10    /* gdt_ptr[6] 共 6 个字节：0~15:Limit 16~47:Base。用作 sgdt/lgdt 的
11       参数。 */
12    u16* p_gdt_limit = (u16*)&gdt_ptr[0];
13    u32* p_gdt_base = (u32*)&gdt_ptr[2];
14    *p_gdt_limit = GDT_SIZE * sizeof(DESCRIPTOR) - 1;
15    *p_gdt_base = (u32)&gdt;
16
17    /* idt_ptr[6] 共 6 个字节：0~15:Limit 16~47:Base。用作 sidt/lidt 的
18       参数。 */
19    u16* p_idt_limit = (u16*)&idt_ptr[0];
20    u32* p_idt_base = (u32*)&idt_ptr[2];
21    *p_idt_limit = IDT_SIZE * sizeof(GATE) - 1;
22    *p_idt_base = (u32)&idt;
23    init_prot();
24    TestAllocAndFree();
25    // disp_str("-----\"cstart\" ends-----\n");
}

```

现在，我们需要修改 makefile，使得其能够链接 pageManage.asm 和新引入的头文件以下是要修改的 makefile 部分：

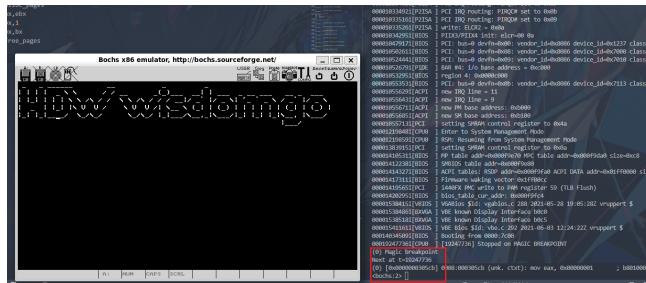
```

1 OBJS          = kernel/kernel.o kernel/start.o kernel/i8259.o kernel/global.o
2               .o kernel/protect.o lib/klib.o lib/kliba.o lib/string.o mm/pageManage.o
3
4 boot/loader.bin : boot/loader.asm boot/include/load.inc \
5               boot/include/fat12hdr.inc boot/include/pm.inc boot/include/
6               gdt.inc \
7               boot/include/selectors.inc
8 $(ASM) $(ASMBFLAGS) -o $@ $<
9
10 mm/pageManage.o : mm/pageManage.asm

```

```
9 $(ASM) $(ASMKFLAGS) -o $@ $<
```

万事俱备，观察分页情况



可以看到在第一个 magic breakpoint 时没有打印出相关信息  
继续运行代码



成功的展示了分页后的打印结果，再次运行代码，应该查看到清除分页



可以看到此时，页表被清除，内存管理功能代码的引入十分成功

### 3.5.3 中断代码集成

我们针对我们显示的 ASCCALL 字符串设计了一个键盘中断，中断效果为将字母 ‘O’ 中的 ‘o’ 的前景颜色加一，也就是每触发一次键盘中断就会改变这个‘o’的颜色。

来到/kernel/kernel.asm 文件中，找到键盘中断 hwint01，将原来触发的显示中断码的代码注释掉，改用自己的中断处理程序。

```
1 hwint01: ; Interrupt routine for irq 1 (keyboard)
2     ; hwint_master      1
3     mov al, [gs:((80 * 2 + 48) * 2 + 1)] ; 读取属性字节到 AL
```

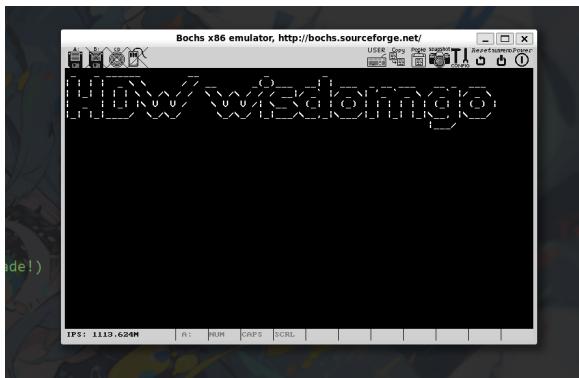
```

4      and al, 0xF0          ; 保留高 4 位 (背景色) , 清
5      零低 4 位 (前景色)
6      mov ah, al             ; 将背景色部分保存在 AH 中
7      mov al, [gs:((80 * 2 + 48) * 2 + 1)] ; 再次读取完整的属性字节
8      and al, 0x0F           ; 清零高 4 位 (背景色) , 保
9      留低 4 位 (前景色)
10     add al, 1              ; 增加前景色值
11     and al, 0x0F           ; 确保前景色在 0 到 15 之间
12     循环
13     or al, ah              ; 将原来的背景色 (高 4 位)
14     合并回来
15
16     mov byte [gs:((80 * 2 + 48) * 2 + 1)], al
17     mov byte [gs:((80 * 3 + 47) * 2 + 1)], al
18     mov byte [gs:((80 * 3 + 49) * 2 + 1)], al
19     mov byte [gs:((80 * 3 + 48) * 2 + 1)], al
      in al, 0x60

      mov al, 20h
      out 20h, al
      iretd

```

这样运行之后，每触发一次键盘中断就可以看到颜色转换的效果。



(a) 初始效果



(b) 触发中断之后的效果

## 4 实验过程分析

(详细记录实验过程中发生的故障和问题，进行故障分析，说明故障排除的过程及方法。根据具体实验，记录、整理相应的数据表格等)

### 4.1 64 位开发平台不兼容 32 位汇编

问题描述：

```

| ~~~~~
gcc -I include/ -c -fno-builtin -fno-stack-protector -o lib/klib.o lib/klib.c
nasm -I include/ -f elf -o lib/kliba.o lib/kliba.asm
nasm -I include/ -f elf -o lib/string.o lib/string.asm
ld -s -Ttext 0x30400 -o kernel.bin kernel/kernel.o kernel/start.o kernel/i8259.o kernel/global.o kernel/protect.o lib/klib.o lib/kliba.o lib/string.o
ld: i386 architecture of input file `kernel/kernel.o' is incompatible with i386:x86-64 output
ld: i386 architecture of input file `lib/kliba.o' is incompatible with i386:x86-64 output
ld: i386 architecture of input file `lib/string.o' is incompatible with i386:x86-64 output
make: *** [Makefile:69: kernel.bin] Error 1
elephant@Zhouyeying ~0/0/c/c/i [2]> 

```

报错原因：开发平台为 64 位 Ubuntu，而书中是以 32 位汇编进行内核开发，所以上面 32 位的汇编程序编译后得到的目标文件不能与 64 位的 Ubuntu 默认的 64 位二进制输出兼容。

解决方案：在链接命令中添加参数：-m elf\_i386，此时仍然会报错，因为 ld 链接命令得到的二进制文件为 32 位后，64 位 Ubuntu 的 gcc 默认情况下编译 start.c 得到的 64 位 start.o，无法与 32 位的二进制文件输出格式兼容，所以也要对 gcc 命令添加参数：gcc -m32 -c -fno-builtin -o start.o start.c，将两者集成到 Makefile 中：

```
1  CFLAGS      = -m32 -fno-stack-protector -I include/ -c -fno-builtin  
2  LDFLAGS     = -m elf_i386 -s -Ttext $(ENTRYPOINT)
```

## 4.2 栈保护错误

问题描述：编译报错‘protect.c:(.text+0x4f9):undefined reference to ‘stack\_chk\_fail\_local’; lib/klib.o:in function disp\_int’；

```
wisdomg@DESKTOP-KITILU9 ~ / c/o/o/c/i (master) > make all  
rm -f kernel/kernel.o kernel/start.o kernel/i8259.o kernel/global.o kernel/protect.o lib/klib.o lib/kliba.o lib/string.o mm/pageManage.o boot/boot.bin boot/loader.bin kernel.bin  
nasm -I boot/include/ -o boot/boot.bin boot/boot.asm  
nasm -I boot/include/ -o boot/loader.bin boot/loader.asm  
boot/include/pm.inc:324: warning: redefining multi-line macro `Descriptor' [-w+other]  
boot/include/pm.inc:338: warning: redefining multi-line macro `Gate' [-w+other]  
nasm -I include/ -f elf32 -o kernel/kernel.o kernel/kernel.asm  
gcc -I include/ -c -fno-builtin -m32 -o kernel/start.o kernel/start.c  
kernel/start.c: In function `cstart':  
kernel/start.c:52:5: warning: implicit declaration of function `TestAllocAndFree' [-Wimplicit-function-declaration]  
52 |     TestAllocAndFree();  
|     ^~~~~~  
gcc -I include/ -c -fno-builtin -m32 -o kernel/i8259.o kernel/i8259.c  
kernel/i8259.c: In function `spurious_irq':  
kernel/i8259.c:57:9: warning: implicit declaration of function `disp_int' [-Wimplicit-function-declaration]  
57 |     disp_int(irq);  
|     ^~~~~~  
gcc -I include/ -c -fno-builtin -m32 -o kernel/global.o kernel/global.c  
gcc -I include/ -c -fno-builtin -m32 -o kernel/protect.o kernel/protect.c  
kernel/protect.c: In function `exception_handler':  
kernel/protect.c:217:9: warning: implicit declaration of function `disp_int' [-Wimplicit-function-declaration]  
217 |     disp_int(eflags);  
|     ^~~~~~  
gcc -I include/ -c -fno-builtin -m32 -o lib/klib.o lib/klib.c  
nasm -I include/ -f elf32 -o lib/kliba.o lib/kliba.asm  
nasm -I include/ -f elf32 -o lib/string.o lib/string.asm  
nasm -I include/ -f elf32 -o mm/pageManage.o mm/pageManage.asm  
ld -s -m elf_i386 -Ttext 0x30400 -o kernel.bin kernel/kernel.o kernel/start.o kernel/i8259.o kernel/global.o kernel/protect.o lib/klib.o lib/kliba.o lib/string.o mm/pageManage.o  
ld: kernel/protect.o: in function exception_handler:  
protect.c:(.text+0x4f9): undefined reference to `__stack_chk_fail_local'  
ld: lib/klib.o: in function `disp_int':  
klib.c:(.text+0xfe): undefined reference to `__stack_chk_fail_local'  
make: *** [Makefile:68: kernel.bin] Error 1
```

报错原因：编译时启用了栈保护（Stack Protector），但缺少支持此功能的库或实现。栈保护机制用于防止栈溢出攻击，一般会在编译时自动加入检查代码，尤其是在 GCC 中，当使用 -fstack-protector 或 -fstack-protector-strong 编译选项时会自动启用此功能。

解决方案：需要在‘Makefile’中的‘\$(CFLAGS)’后面加上‘-fno-stack-protector’，即不需要栈保护。

```
CFLAGS      = -m32 -fno-stack-protector -I include/ -c -fno-builtin
```

## 4.3 Makefile 中 make all 指令未集成镜像挂载指令

问题描述：一开始我们尝试修改 /start.c 文件中 dis\_str 的输出信息时发现无法将我们的修改更新到显存中，输出一直都是—cstartbegins—。

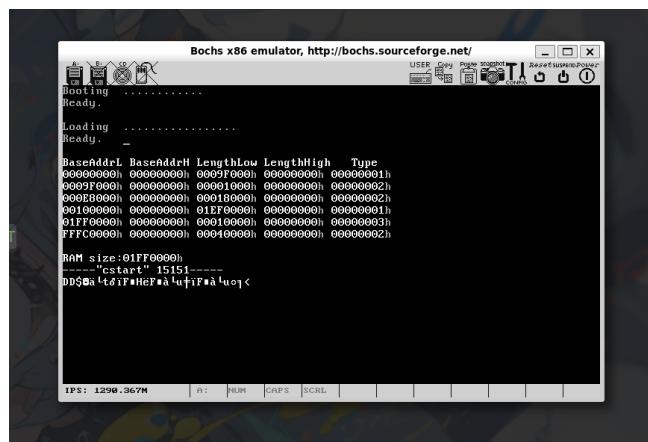
出错原因：我们使用的是 make all 指令将原有编译生成的文件删除并重新编译生成二进制文件，但我们去 Makefile 中仔细观察之后发现，make all 指令并没有集成重新生成镜像和挂载镜像的 buildimg 或 image 指令。

解决方案：在 make all 指令中加入 image 指令内容。

```
1      all : realclean everything image
```

## 4.4 disp\_str 函数错误

问题描述：在第二次调用 dip\_str 函数时会输出乱码。



出错原因：目前新版本的编译器，在支持位置无关重定位时候，使用了 bx 寄存器，而 cstart 调用了 disp\_str，该函数内部对 bx 寄存器又有访问，但是函数被调用时没有做 bx 的现场保护，因此导致错乱。

解决方案：在 disp\_str 函数中对 bx 寄存器进行保护。

```
1      .1:
2          lodsb
3          test    al, al
4          jz     .2
5          cmp    al, 0Ah           ; 是回车吗?
6          jnz    .3
7          push   bx             ; 保护 bx
8          push   eax
9          mov    eax, edi
10         mov    bl, 160
11         div    bl
12         and   eax, OFFh
13         inc    eax
14         mov    bl, 160
15         mul    bl
16         mov    edi, eax
17         pop    eax
18         pop    bx             ; 保护 bx
```

## 4.5 键盘中断问题

问题描述：我们想实现的键盘中断是每触发一次键盘中断就将显存中的某几个块的前景颜色 +1，也就是会看到字符的颜色改变的效果，但在我们自己实验的过程中发现最多每 8 次键盘输入就会改变一次背景颜色。

出错原因：在显存中每个块的颜色由 8 位二进制进行控制，其中高 4 位控制背景颜色，低 4 位控制前景颜色，正常来说我们最多在触发 16 次键盘中断（前景颜色 +16 造成低 4 位数溢出到第 5 位）之后就会修改一次背景颜色。结合汇编知识，我们发现每一次键盘输出分为按下键盘和键盘弹起两个过程，按下的过程中键盘控制器会生成一个按下扫描码，弹起过程中键盘控制器会生成对应的释放扫描码，每个过程都会触发一次键盘中断，所以每 8 次键盘输入就会变动一次背景颜色。

解决方案：保存高 4 位背景颜色信息，只修改低 4 位前景颜色控制信息。

```

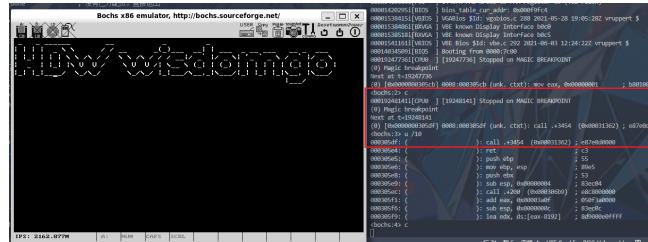
1 ALIGN 16
2 hwint01:           ; Interrupt routine for irq 1 (keyboard)
3     ; hwint_master    1
4     mov al, [gs:((80 * 2 + 48) * 2 + 1)]      ; 读取属性字节到 AL
5     and al, 0xF0                                ; 保留高 4 位 (背景色) ,
6     清零低 4 位 (前景色)
7     mov ah, al                                ; 将背景色部分保存在 AH
8     中
9     mov al, [gs:((80 * 2 + 48) * 2 + 1)]      ; 再次读取完整的属性字节
10    and al, 0x0F                               ; 清零高 4 位 (背景色) ,
11    保留低 4 位 (前景色)
12    add al, 1                                 ; 增加前景色值
13    and al, 0x0F                               ; 确保前景色在 0 到 15 之
14    间循环
15    or al, ah                                ; 将原来的背景色 (高 4
16    位) 合并回来
17    mov byte [gs:((80 * 2 + 48) * 2 + 1)], al
18    mov byte [gs:((80 * 3 + 47) * 2 + 1)], al
19    mov byte [gs:((80 * 3 + 49) * 2 + 1)], al
20    mov byte [gs:((80 * 3 + 48) * 2 + 1)], al
     in al, 0x60

     mov al, 20h
     out 20h, al
     iretd

```

## 4.6 内存管理问题

问题描述：在完成设计题内存管理时，我们沿用了第三章的代码，但这导致了严重的内存寻址问题。运行代码时不能正确查看到分页后的结果。使用 bochs 调试，确信已经完成分页，但是不能查看到成功分页后应该打印的字符串：



可以看到，在调试过程中 u /10 后出现了 call alloc\_page 正常情况下继续运行 (continue) 后应该看到打印的字符串，然而程序出现了莫名的错误，程序卡在了原地不能正常继续，也没有打印字符。

出错原因：

```

1 SucessMessage:      db "Successfully alloc ^-^", 0 ; 进入保护模式后显示
2   此字符串
3 OffsetSucessMessage equ SucessMessage - $$

4 PageFaultMessage: db "Page Fault T_T", 0
5 OffsetPageFaultMessage equ PageFaultMessage - $$
```

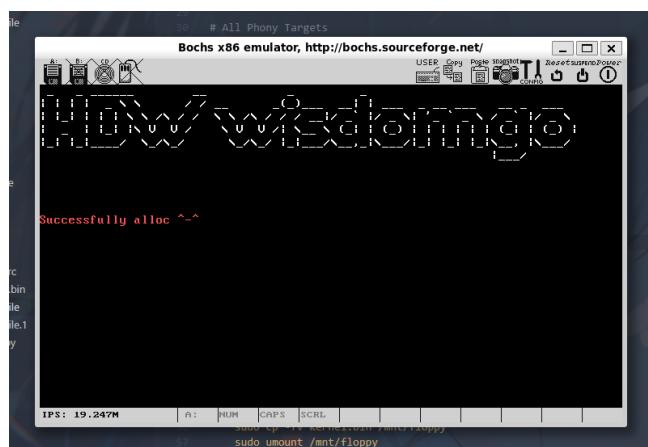
不难看到，我们这里直接使用了第三章代码中有关打印信息的设置，然而，我们的代码会从 kernel.asm 中跳入进来。不得不注意到 \$\$ 表示该段的起始地址。从 kernel.asm 中跳入过来，不能确定在哪一个段，这时减掉的段起始地址后就不知道对应的什么地址，显然也就不是打印字符串的地址。由于第五章是我们将 orangs 操作系统开始搭建为一个项目代码的开始，所以这样的问题，我们是第一次遇到。

解决方案：我们不需要使用段的偏移地址，而是直接将变量赋值给寄存器，这样就不会再产生错误。

```

1 alloc_success:
2   ; 成功分配了请求的页，标记它们为已分配
3   lea edi, [es:FoundPages]      ; 恢复页号保存指针
4   mov esi, eax                ; 请求的页数
5   mov edx, PageTblBase        ; 页表基址
6   ;mov    esi, OffsetSucessMessage ; 源数据偏移
7   mov esi, SucessMessage
8   mov edi, (80 * 10 + 0) * 2 ; 目的数据偏移。屏幕第 10 行，第 0 列。
9   call PrintString
```

再次运行代码，能够看到打印的字符串



## 5 实验结果总结

(对实验结果进行分析，完成思考题目，并提出实验的改进意见)

### 5.1 汇编和 C 内定义的函数，相互间调用的方法是怎样的？

#### 5.1.1 从 C 调用汇编函数

要想从 C 中调用汇编函数，需要分别在 C 和汇编程序中进行声明，并在汇编中实现。

1. 在 C 中，先用 extern 声明汇编函数，以便 C 编译器知道它的存在。

```
1     extern void asm_function(int arg1, int arg2);
```

2. 在汇编中，实现这个函数并用 global 声明。

```
1     global asm_function          ; 声明为全局，C 可以调用它
2
3     asm_function:
4     ; arg1 在 [esp + 4]
5     ; arg2 在 [esp + 8]
6
7     mov eax, [esp + 4]           ; 加载第一个参数
8     mov ebx, [esp + 8]           ; 加载第二个参数
9
10    add eax, ebx               ; eax = arg1 + arg2
11
12    ret                        ; 返回到调用点
```

#### 5.1.2 从汇编调用 C 函数

而要想从汇编中调用 C 的函数，只需要在汇编程序中进行声明，在 C 中实现。

1. 在 C 中定义函数。

```
1     int c_function(int x, int y) {
2         return x + y; // 简单地返回 x + y
3     }
```

2. 在汇编中声明并调用 C 函数。

```
1     extern c_function
2     push 5                      ; 参数 y
3     push 10                     ; 参数 x
4     call c_function              ; 调用 C 函数
5     add esp, 8                  ; 清理栈 (cdecl 调用约定)
```

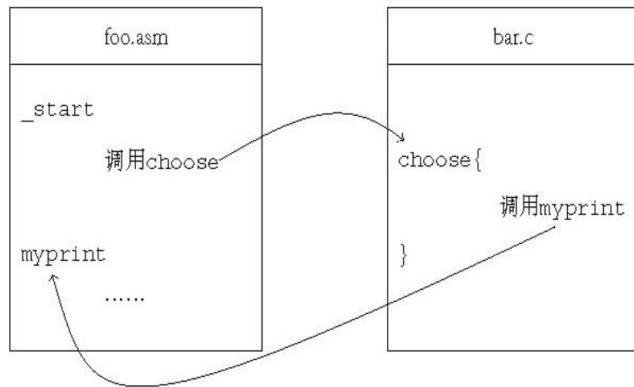
### 5.1.3 C 调用约定 (C Calling Convention)

注意到，不管是 myprint( ) 还是 choose( )，都需要遵循一个相同的调用协定。常用的 C 调用协定有 cdecl 和 stdcall 两种，在 orang os 教材中说明了‘在这里两个程序的代码遵循的都是 C 调用约定 (C Calling Convention)，后面的参数先入栈，并由调用者 (Caller) 清理堆栈。’也就是使用 cdecl 协议。

cdecl 调用约定的特点：

- 参数传递顺序是从右到左将参数压入栈。
- 栈清理责任在调用者，即调用者负责清理栈上的参数。
- 如果有返回值则保存在寄存器 eax 中。
- 编译器可能会对函数名进行下划线修饰。例如，函数 func 可能会被修饰为 \_func。

### 5.1.4 以示例代码为例分析 C 和汇编相互调用关系



在示例代码中的调用关系是：在 foo.asm 中的 \_start 函数跨语言调用 C 语言中的 choose 函数，而 choose 函数又调用 foo.asm 中的 myprint 函数。

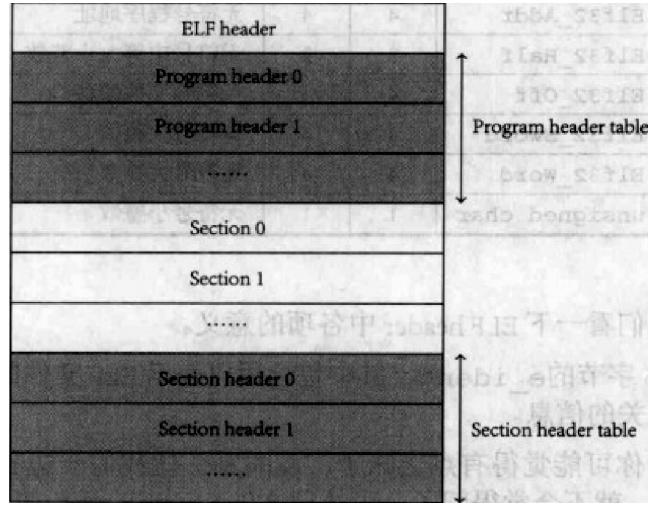
在示例代码中的声明关系是：在 foo.asm 中用 extern 声明 choose，以及在 bar.c 中用 extern 声明 myprint。

## 5.2 描述 ELF 文件格式以及作用，和大家学习的 PE 相比，结构上有什么相同和差异？

### 5.2.1 ELF 文件格式

ELF 文件格式如下图所示 ELF 文件由 4 部分组成，分别是 ELF 头，程序表头，节和节头表，使用 010editor 打开可执行文件 foobar 进行分析：

先针对 ELF 文件格式定义新的类型说明：



名称	大小	对齐	用途
Elf32_Addr	4	4	无符号程序地址
Elf32_Half	2	2	无符号中等大小整数
Elf32_Off	4	4	无符号文件偏移
Elf32_Sword	4	4	有符号大整数
Elf32_Word	4	4	无符号大整数
unsigned char	1	1	无符号小整数

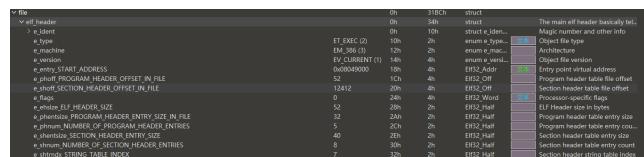
在此基础上，ELF header 的格式如下代码所示：

```

1 typedef struct
2 {
3     unsigned char e_ident[16];
4     Elf32_Half e_type;
5     Elf32_Half e_machine;
6     Elf32_Word e_version;
7     Elf32_Addr e_entry;
8     Elf32_Off e_phoff;
9     Elf32_Off e_shoff;
10    Elf32_Word e_flags;
11    Elf32_Half e_ehsize;
12    Elf32_Half e_phentsize;
13    Elf32_Half e_phnum;
14    Elf32_Half e_shentsize;
15    Elf32_Half e_shnum;
16    Elf32_Half e_shstrndx;
17 } Elf32_Ehdr;

```

通过 010editor，我们可以查看 ELF header 各项的值及含义：着重分析集中几个项的含义：



- e\_type 标识该文件的类型，这里 foobar 的值为 0x2，表示是一个可执行文件
- e\_entry 表示程序的入口地址 foobar 的入口地址为 0x80480A0
- e\_phoff 表示 Program header table 在文件中的偏移量，这里的值为 0x34
- e\_shoff 表示 Section header table 在文件中的偏移量，这里的值为 0x1C0

Programheader 的格式如下面代码所示：

```

1   typedef struct {
2     Elf32_Word p_type;
3     Elf32_Off  p_offset;
4     Elf32_Addr p_vaddr;
5     Elf32_Addr p_paddr;
6     Elf32_Word p_filesz;
7     Elf32_Word p_memsz;
8     Elf32_Word p_flags;
9     Elf32_Word p_align
10 } Elf32_Phdr;

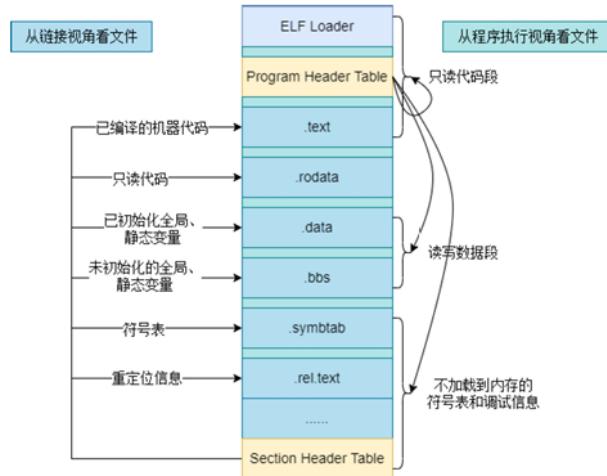
```

▼ program_header_table									
▼ program_header_table_element[0]									
p_type		34h	Afh	struct progra...	文本	Program headers - describes th...			
p_offset FROM_FILE_BEGIN		34h	Afh	Elf32_Off	文本	Segment file offset			
p_vaddr FROM_VIRT_ADDRESS		38h	4h	Elf32_Addr	文本	Segment virtual address			
p_paddr PHYSICAL_ADDRESS		3Bh	4h	Elf32_Addr	文本	Segment physical address			
p_filesz SEGMENT_FILE_LENGTH		3Ch	4h	Elf32_Word	文本	Segment size in file			
p_memsz SEGMENT_LOAD_LENGTH		40h	4h	Elf32_Word	文本	Segment size in memory			
p_flags		212	4h	Elf32_Word	文本	Segment flags			
p_align		40h	4h	Elf32_Word	文本	Segment alignment			
p_data[2]		4016	50h	Elf32_Word	文本	Segment data			
▼ program_header_table_element[1]		LL	0h	Elf32_Word	文本				
▼ program_header_table_element[2]		[R, X] Loadable Se...	54h	Elf32_Word	文本				
▼ program_header_table_element[3]		[R, X] Loadable Se...	74h	Elf32_Word	文本				
▼ program_header_table_element[4]				struct progra...	文本				

其中各项的含义如下：

- p\_type type: 当前 Program header 所描述的段的类型。
- p\_offset offset: 段的第一个字节在文件中的偏移。
- p\_vaddr vaddr: 段的第一个字节在内存中的虚拟地址。
- p\_paddr paddr: 在物理地址定位相关的系统中，此项是为物理地址保留。
- p\_filesz filesz: 段在文件中的长度。
- p\_memsz memsz: 段在内存中的长度。
- p\_flags flags: 与段相关的标志。
- p\_align align: 根据此项值来确定段在文件以及内存中如何对齐。

### 5.2.2 ELF 文件格式作用



ELF (Executable and Linkable Format) 文件是一种通用的文件格式，广泛用于 Unix 和 Linux 系统。文件的格式在程序的连接（编译和链接阶段）和执行（加载和运行阶段）中都扮演着重要的角色。

在编译和链接的过程中，ELF 文件为汇编器和链接器提供了清晰的结构。对于汇编器和链接器来说，ELF 文件是由 Section header table 描述的多个 section (节) 组成的，每个 section 存储特定类型的数据或代码，比如.text 存储代码，.data 存储初始化数据，.bss 存储未初始化数据等。Section header table 描述了每个 section 的位置、大小、类型等信息，从而让链接器可以根据这些信息将多个目标文件链接成一个可执行文件。通过管理这些 sections，ELF 格式帮助构建和管理程序的各个组成部分，使链接器能够将各个编译单元有效地组合为一个完整的程序。

在程序运行时，加载器（Loader）会加载 ELF 文件。对于加载器来说，ELF 文件是由 Program header table 描述的多个 segment (段) 组成的。Program header table 描述了每个 segment 的内存映射方式，例如代码段、数据段的内存地址、大小、权限等。每个 segment 是程序运行时实际加载到内存中的一块数据或代码。加载器根据 Program header table 的信息将段映射到合适的内存地址，为程序的执行做准备。通过这种结构，ELF 文件可以确保程序的各个部分在内存中的位置和权限设置正确，使得操作系统可以有效地加载并运行程序。

### 5.2.3 ELF 文件格式和 PE 文件格式的相同与差异

ELF 和 PE 文件格式是两种常见的可执行文件格式，分别广泛用于 Linux/Unix 系统和 Windows 系统中。

它们的相同点有：

- 它们都包含了程序的代码段、数据段以及符号信息，并支持静态和动态链接，都使用类似的文件结构来描述文件内容。
- 它们的结构由 Header 和多个段组成，用于区分代码、数据和符号表等内容。
- 它们都有一个入口点（Entry Point），指向程序开始执行的位置。

主要差异包括：

特性	ELF 文件格式	PE 文件格式
主要使用平台	Linux/Unix 系统	Windows 系统
头部结构	ELF Header, Program Header, Section Header	DOS Header, PE Header, Optional Header
加载机制	由 Program Header Table 控制加载	Windows Loader, 使用 IAT/EAT 表
符号信息	.dynsym, .dynstr, .plt, .got	Import Table, Export Table
节 (Section) 定义	丰富的节类型, 如.text, .data, .bss	较少的节类型, 如.text, .data, .rsrc
调试信息	DWARF 格式	PDB 文件
安全性	无内置签名支持, 需外部工具	支持数字签名

### 5.3 如何从 Loader 引导 ELF 的原理

Loader 需要做两项工作

- 加载内核到内存: 使用 Loader 按照寻找文件、定位文件以及读入内存的步骤把内核放入到内存中
- 跳入保护模式

详情请见 3.3 使用 Loader 加载 ELF 部分。

### 5.4 对照书中例程代码，这个内核扩展了哪些功能，这些功能流程是怎样的，他们都是在哪些源文件的代码中进行描述的？这些功能彼此有相互关联吗，给出说明？

对照书中第四章的最简易的内核 loader 实现的在屏幕上显示一个 L 的功能，第五章中的内核扩展了软件中断、硬件中断和异常处理程序，并且调用了 cstart.c 文件。

中断程序统一定义在 /protect.c 文件中，文件中初始化了保护模式下的中断向量表 (IDT)，设置中断门描述符，并设置各类异常的处理函数。

```

1 void      divide_error();
2 void      single_step_exception();
3 void      nmi();
4 void      breakpoint_exception();
5 ...
6 void      hwint00();
7 void      hwint01();
8 void      hwint02();
9 void      hwint03();
10 void     hwint04();
11 ...
12 PUBLIC void init_prot()
13 {
14     init_8259A();
15
16     // 全部初始化成中断门(没有陷阱门)
17     init_idt_desc(INT_VECTOR_DIVIDE,      DA_386IGate,
18                   divide_error,      PRIVILEGE_KRNL);

```

```

19     ...
20 }
21 PUBLIC void exception_handler(int vec_no,int err_code,int eip,int cs,int
22   eflags)
23 {
24     ...
}

```

在 /kernel.asm 文件中实现了各种硬件中断的处理程序，并在异常处理程序处调用了 /protect.c 文件中的异常处理函数。

```

1 ALIGN    16
2 hwint02:           ; Interrupt routine for irq 2 (cascade!)
3     hwint_master    2
4
5 ALIGN    16
6 hwint03:           ; Interrupt routine for irq 3 (second serial)
7     hwint_master    3
8     ...
9 divide_error:
10    push    0xFFFFFFFF ; no err code
11    push    0           ; vector_no = 0
12    jmp    exception
13 single_step_exception:
14    push    0xFFFFFFFF ; no err code
15    push    1           ; vector_no = 1
16    jmp    exception
17 nmi:
18    push    0xFFFFFFFF ; no err code
19    push    2           ; vector_no = 2
20    jmp    exception

```

在 /i8259.c 文件中初始化并管理 8259A 可编程中断控制器 (PIC)，从而为系统的中断处理做好硬件设置。使用 init\_8259A() 发送初始化控制字 (ICW) 来配置主从 8259A 的工作模式和中断向量表的起始位置。

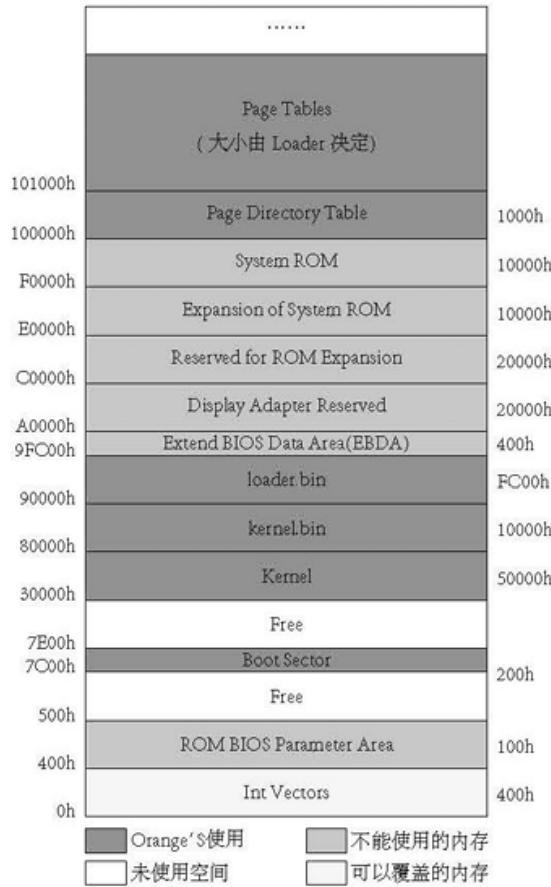
```

1 /* Master 8259, ICW1. */
2 out_byte(INT_M_CTL, 0x11);
3
4 /* Slave 8259, ICW1. */
5 out_byte(INT_S_CTL, 0x11);

```

## 5.5 书中代码内存的布局是怎样的？在这里有哪些是特权代码，哪些是非特权代码，在处理器控制权切换时，权限变化情况如何？

书中代码内存的布局：



在 boot 中加载 loader 代码到内存和在 loader 中加载内核到内存的代码都是在实模式下运行的，没有进入保护模式，没有特权级代码这一说。在 loader 中将内核加载到内存中之后才跳入保护模式，跳入保护模式之后才有特权级代码这一说法。跳入保护模式的代码位于 /loader.asm 代码中的 169 行。

```

1 ; 下面准备跳入保护模式 -----
2
3 ; 加载 GDTR
4     lgdt    [GdtPtr]
5
6 ; 关中断
7     cli
8
9 ; 打开地址线 A20
10    in     al, 92h
11    or     al, 00000010b
12    out    92h, al
13
14 ; 准备切换到保护模式
15     mov    eax, cr0
16     or     eax, 1
17     mov    cr0, eax
18
19 ; 真正进入保护模式
20     jmp    dword SelectorFlatC:(BaseOfLoaderPhyAddr+LABEL_PM_START)

```

这里是跳转到选择子 SelectorFlatC 处运行，所以我们查看选择子的相关定义。

```
1 SelectorFlatC      equ LABEL_DESC_FLAT_C - LABEL_GDT
2 ...
3 LABEL_DESC_FLAT_C:    Descriptor          0,           0
4     ffffffh, DA_CR | DA_32 | DA_LIMIT_4K ; 0 ~ 4G
5 ...
6 DA_32      EQU 4000h ; 32 位段
7 DA_CR      EQU 9Ah ; 存在的可执行可读代码段属性值
8 DA_LIMIT_4K EQU 8000h ; 段界限粒度为 4K 字节
```

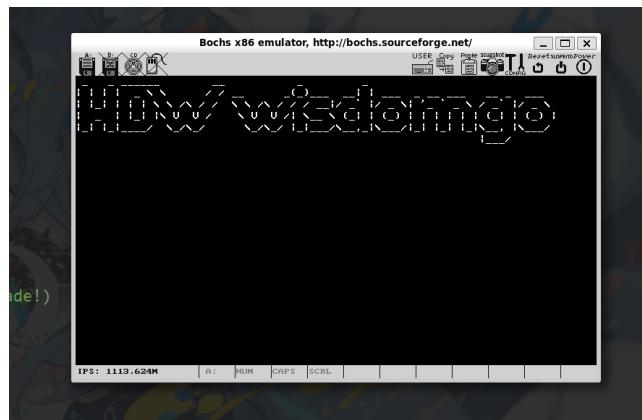
可以看到，选择子对应的描述符 LABEL\_DESC\_FLAT\_C 所定义的 DA\_CR 中规定了 DA\_CR EQU 9Ah, 9A 的二进制形式为 10011010, P 位（存在位）为 1, 表示段存在。S 位为 1, 表示这是代码或数据段，而非系统段。DPL 位（第 5 和第 6 位）为 00, 表示此段的特权级为 0 (Ring 0)。E 位（执行位）为 1, 表示这是代码段。R 位（可读位）为 1, 表示代码段可读。所以跳转到保护模式之后的特权级为 0 (Ring 0)。后续运行的将控制权交给内核，扩展内核、配置中断操作等代码均属于特权级代码，都是在内核态 (Ring 0) 中进行。

## 5.6 下载一个真正的内核源文件，分析一下是怎么管理组织源码文件的。

内核源文件的对比详情请见 3.4.2 对比真正内核源码的代码组织情况。

## 5.7 完成设计题并能演示。

1. 显示一个喜欢的 ASCII 图案。



2. 集成内存管理功能。

分页前



分页后



分页清除后



3. 集成中断程序。



## 5.8 附加题：当你阅读到 chapter5/f/start.c 时，请留意是如何对 gdt 进行修改的？

先看代码实现：

```
1 PUBLIC void cstart()
2 {
3     printlogo();
4     /* 将 LOADER 中的 GDT 复制到新的 GDT 中 */
5     memcpy(&gdt,
```

```

6     (void*)(*((u32*)(&gdt_ptr[2]))), /* Base of Old GDT */
7     *((u16*)(&gdt_ptr[0])) + 1 /* Limit of Old GDT */
8   );
9   /* gdt_ptr[6] 共 6 个字节: 0~15:Limit 16~47:Base。用作 sgdt/lgdt 的
10  参数。*/
11  u16* p_gdt_limit = (u16*)(&gdt_ptr[0]);
12  u32* p_gdt_base = (u32*)(&gdt_ptr[2]);
13  *p_gdt_limit = GDT_SIZE * sizeof(DESCRIPTOR) - 1;
14  *p_gdt_base = (u32)&gdt;
15
16  /* idt_ptr[6] 共 6 个字节: 0~15:Limit 16~47:Base。用作 sidt/lidt 的
17  参数。*/
18  u16* p_idt_limit = (u16*)(&idt_ptr[0]);
19  u32* p_idt_base = (u32*)(&idt_ptr[2]);
20  *p_idt_limit = IDT_SIZE * sizeof(GATE) - 1;
21  *p_idt_base = (u32)&idt;
22  init_prot();
23
24 // disp_str("-----\"cstart\" ends-----\n");
}

```

cstart 函数的作用是将引导加载程序中的 GDT（全局描述符表）复制到内核的 GDT 空间，并将内核 GDT 的基址和界限重新加载到 gdt\_ptr 中，同时初始化 IDT（中断描述符表）的基址和界限。这是保护模式初始化中的关键步骤。

使用 memcpy 函数将 Loader 中的 GDT 复制到内核 GDT 中。string.h 文件中有 memcpy 函数的定义，memcpy 函数有三个形参，分别是（目标地址，源地址，长度）。

```
1 PUBLIC void* memcpy(void* p_dst, void* p_src, int size);
```

- 用 &gdt 取址到在内核内存空间定义的 GDT 数组作为目标地址，其中 gdt 为 global.h 文件中定义的全局变量。
- 用 (void\*)(\*((u32\*)(&gdt\_ptr[2]))) 取到源地址，指向 Loader 中的旧 GDT 的基址，gdt\_str 在 kernel.asm 中也声明了这是一个全局变量。
- 用 \*((u16\*)(&gdt\_ptr[0])) + 1 计算复制的字节数。gdt\_ptr[0:1] 存储了旧 GDT 的 limit，而需要 +1 是因为 limit 从 0 开始计数。

更新内核的 GDT 指针。先将 p\_gdt\_limit 和 p\_gdt\_base 指向 gdt\_ptr 的界限和基址字段。用 \*p\_gdt\_limit 设置新的 GDT 界限，大小为 GDT\_SIZE \* sizeof(DESCRIPTOR) - 1，其中 GDT\_SIZE 是描述符数量，sizeof(DESCRIPTOR) 是每个描述符的大小，要减一是因为这个 limit 是从 0 开始计数。再用 \*p\_gdt\_base 设置新的 GDT 基址，即内核 GDT 的起始地址 &gdt。

更新 IDT 指针与更新 GDT 指针同理，不过中断描述符的描述符类型是 GATE。

最后使用 init\_prot 函数初始化保护模式，通过 lgdt 和 lidt 指令加载新的 GDT 和 IDT。

## 5.9 实验改进意见

1. 希望能够提供更详细的实验指导：在每个实验步骤中，提供更详细的指导和说明，比如预期结果等，这样能帮助我们更好地完成实验。

2. 希望能够提供常见错误类型及纠错指南，帮助同学们在遇到问题的时候能够自己解决问题。
3. 希望老师能讲解汇编代码的一些关键部分，同学们自己阅读汇编码容易忽略一些代码细节问题。

## 6 各人实验贡献与体会（每人各自撰写）

### 1. 黄东威：

承担任务：独立完成实验，完成实验内容 C 语言和汇编语言的互相调用、内存管理、打印 ASCII 码的代码撰写，根据代码编写过程中遇到的问题补充部分实验故障，补充部分思考题。

个人体会：通过本次操作系统实验，我深刻体会到了操作系统设计的严谨性和技术的挑战。特别是在实现内存管理和页分配功能的过程中，我对操作系统如何有效地管理和分配物理内存有了更深入的理解。在实现页表管理时，我需要细致地规划数据结构，并确保位图与页表的正确映射。这让我意识到内存管理不仅仅是简单的分配和释放操作，而是需要在效率和稳定性之间找到平衡。实验过程中，我遇到了不少困难，比如在页分配逻辑和分页机制的实现上出现了很多细小但致命的错误。每当遇到问题时，我需要通过调试和查阅资料来寻找解决方案，这不仅让我掌握了操作系统的根本概念，还锻炼了我的问题分析和自学能力。实验的每一步都充满挑战，但也带给了我极大的成就感。

总体而言，这次实验极大地加深了我对操作系统底层设计的理解，并增强了我的调试能力和解决复杂问题的信心。我希望在后续的实践中继续深入学习操作系统设计，不断提升自己的技能。

### 2. 周业营：

承担任务：独立完成实验，回答思考题 1-7、附加题，撰写设计题、思考题、实验过程分析、附加题的实验报告，帮助队友调试代码问题，提出和解决实验中遇到的问题。

个人体会：这次实验的过程中我才真正接触到了一个 kernel 的设计和撰写，在之前的 OS 理论课中我做的实验仅是在已有的 Linux 内核中添加内核模块，但这次相较于深入接触内核模块设计和撰写的全过程，我才逐渐了解使用 boot 驱动引导和装载 loader 程序，再由 loader 程序寻找和装载内核到内存中，并将系统权限给内核的全过程，这个过程环环相扣，逻辑缜密，让我感受到操作系统的精巧。在不断重复遇到问题和解决问题的过程中，也在不断锻炼我的思考能力。操作系统具有非常精巧的结构，每部分代码环环相扣，严丝合缝，任何一点小错误都可能引发让人摸不着头脑的错误，甚至不知道产生错误的地方，难以调试。在学习汇编语言和 C 语言程序的相互调用的过程中，也再次让我感悟到无论何种编程语言最终都将归一到二进制机器码的形式，所以无论是用什么语言去编写程序，总能将它们糅合在一起，实现嵌入式代码开发。

总的来说，这次实验的任务量不小，但机遇蕴含于挑战之中，完整地完成这次实验内容也增加了我对操作系统内核的理解，为未来学习 OS 打下良好的基础。

### 3. 程序：

承担责任：独立完成实验，完成实验内容前三题的撰写，补充部分实验故障，补充部分思考题。

个人体会：本次实验涉及的书本内容很多，任务量较大。这周实验主要基于之前学过的中断机制、分页机制、保护模式部分的内容，改为用 C 语言和汇编语言交叉的方式书写代码，从 0 开始不断拓展内核，一方面总结了之前学过的内容，另一方面为接下来实现进程提供控制权切换等方面的支持。此外，为了避免在终端输入过多命令行，实验还介绍了 Makefile 方面的知识，帮助我们更好的组织文件结构和快速编译。通过这周实验，我对之前学过的内容有了较快速的回顾，为接下来的实验打下基础。

#### 4. 王浚杰：

承担责任：独立完成实验，补充实验步骤中 C 语言和汇编语言的互相调用、ELF 文件格式、使用 Loader 加载 ELF 文件的部分，并在打印 ASCII 码部分完成内存管理代码的撰写。

个人体会：在上周的实验中，我们已经实现了如何通过引导扇区装载 Loader。本周实验进一步深入，通过 Loader 装载 kernel 到内存中，将控制权转交给 kernel，让我更清楚了通过 loader 引导 ELF 文件的步骤。在三周前，我们实现了内存管理的分页和释放页功能，在本次实验设计题中得到了应用，整个过程让我体会到系统设计的严谨性，让我意识到操作系统开发需要良好的代码结构和模块化设计，从引导扇区、Loader 到内核，每个阶段都需要明确的职责划分和接口设计，以确保系统能够稳定运行。同时书中整理目录章节也正强调良好的项目结构和清晰的文档，我们需要提高代码的可读性和可维护性，为后续的功能扩展和调试打下坚实的基础。

## 7 教师评语

(实验报告的考评：依据实验内容完整度、实验步骤清晰度、实验结果与分析正确性、实验心得与思考的全面性、实验报告文档的规范性等五个维度综合考评)

分数	评语
85-100	<ul style="list-style-type: none"><li>实验内容完整或者有超出课程实验大纲的内容；</li><li>实验步骤详尽，能够体现完整的实验过程；</li><li>实验结果正确且实验数据分析得当；</li><li>实验心得与思考全面并且有自己的独立思考；</li><li>实验报告文档规范、排版整齐。</li></ul>

75-84	<ul style="list-style-type: none"> <li>• 实验内容较为完整；</li> <li>• 实验步骤较为详尽，能够体现实验过程；</li> <li>• 实验结果正确且实验数据分析较为得当；</li> <li>• 实验心得与思考全面；</li> <li>• 实验报告文档规范、排版较为整齐。</li> </ul>
60-74	<ul style="list-style-type: none"> <li>• 实验内容有缺失；</li> <li>• 实验步骤不够详尽，不能够体现完整的实验过程；</li> <li>• 实验结果部分正确；</li> <li>• 实验心得与思考无或者不够深入；</li> <li>• 实验报告文档规范性有待增强。</li> </ul>
60 以下	<ul style="list-style-type: none"> <li>• 实验内容严重缺失、实验态度不够端正；</li> <li>• 实验步骤不够详尽，不能够体现完整的实验过程；</li> <li>• 实验结果部分正确；</li> <li>• 实验心得与思考无或者不够深入；</li> <li>• 实验报告文档规范性有待增强。</li> </ul>

## 教师评分（请填写好姓名、学号）

姓名	学号	分数
黄东威	2022302181148	
王浚杰	2022302181143	
程序	2022302181131	
周业营	2022302181145	

教师签名：

年      月      日