

## 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	2023. 11. 14
实验名称	内核雏形	实验周次	第六周
姓名	学号	专业	班级
聂森	2021302191536	网络空间安全	9 班
程子洋	2021301051114	网络空间安全	9 班
王卓	2021302191791	网络空间安全	9 班
刘琥	2021302121234	网络空间安全	9 班

### 一、实验目的及实验内容

(本次实验所涉及并要求掌握的知识；实验内容；必要的原理分析)

#### 实验内容：

##### (一)

1. 汇编和 C 的互相调用方法在例程基础上，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 文件格式使用 xxd 命令分析 ELF 文件中的 ELF header、Program header，了解各项的作用
3. 使用 Loader 加载 ELF 文件，重新放置内核
4. 扩展内核，切换堆栈和 GDT、整理文件结构、使用 makefile 编译程序、添加中断处理
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

##### (二)

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 加载 ELF，如何确定 ELF 文件加载到内存的位置？
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（选做）。
7. 完成设计题并能演示。

## 二、实验环境及实验步骤

(本次实验所使用的器件、仪器设备等的情况；具体的实验步骤)

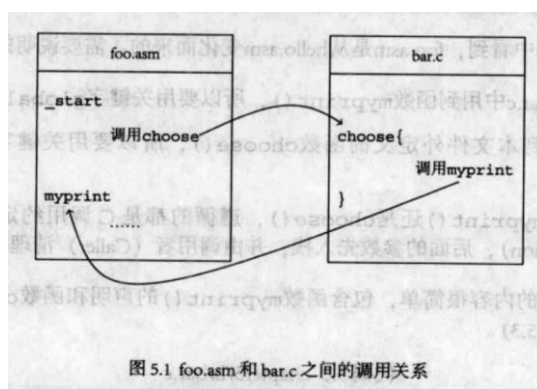
实验环境：

1. 虚拟机： VMware Workstation Pro/VMware Workstation 16 player
2. 操作系统： Ubuntu 16.04
3. 模拟系统软件： Bochs 2.7

实验步骤：

1. 汇编和 C 的互相调用方法在例程基础上，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。

以书本举例为例，源代码包含两个文件：foo.asm 和 bar.c。程序入口\_start 在 foo.asm 中，一开始程序将会调用 bar.c 中的函数 choose(), choose() 将会比较传入的两个参数，根据比较结果的不同打印出不同的字符串。打印字符串的工作是由 foo.asm 中的函数 myprint() 来完成的。整个过程如图所示：



对例程进行略微修改：

```
void myprint(char* msg, int len);
int choose(int a, int b)
{
    if(a > b){
        myprint("the 1st one\n", 13);
    }
    else if(a < b){
        myprint("the 2nd one\n", 13);
    }
    else {
        myprint("2021301051114 czy\n2021302191791 wz\n2021302191536 ns\n2021302121234 lx\n", 80);
    }
    return 0;
}
```

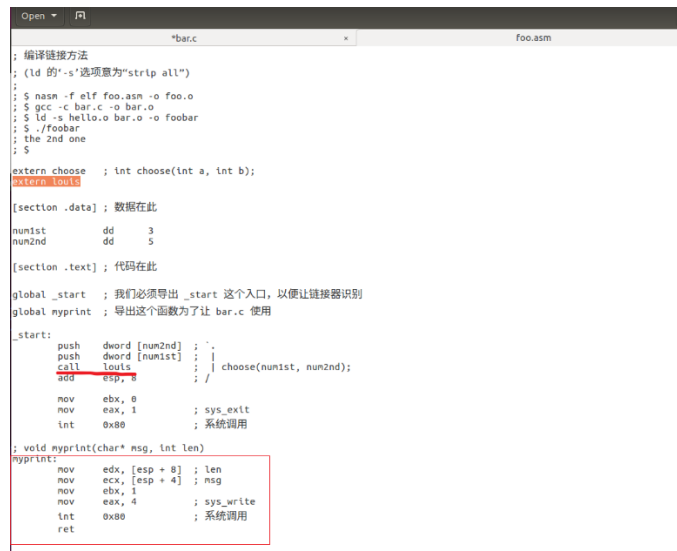
在终端运行：

```
nasm -f elf -o foo.o foo.asm
gcc -c -o bar.o bar.c
ld -s -o foobar foo.o bar.o
./foobar
```

编译链接并执行得到结果：

```
shizi@shizi-virtual-machine:~/Desktop/chapter5/b$ nasm -f elf foo.asm -o foo.o
shizi@shizi-virtual-machine:~/Desktop/chapter5/b$ gcc -c bar.c -o bar.o
shizi@shizi-virtual-machine:~/Desktop/chapter5/b$ ld -s -o foobar foo.o bar.o
shizi@shizi-virtual-machine:~/Desktop/chapter5/b$ ./foobar
2021301051114 czy
2021302191791 wz
2021302191536 ns
2021302121234 lx
```

对汇编代码进行修改，关键字 `extern` 调用 C 语言函数 `louis`，函数 `myprint` 作为用于调用的函数。



```

; 编译链接方法
; (ld 的'-s'选项意为'strip all')
;
; $ nasm -f elf foo.asm -o foo.o
; $ gcc -c bar.c -o bar.o
; $ ld -s -o foobar foo.o bar.o
; the 2nd one
; $
extern choose ; int choose(int a, int b);
extern louis

[section .data] ; 数据在此
num1st      dd 3
num2nd      dd 5

[section .text] ; 代码在此

global _start ; 我们必须导出 _start 这个入口，以便让链接器识别
global myprint ; 导出这个函数为了让 bar.c 使用

_start:
    push dword [num2nd] ; .
    push dword [num1st] ; |
    call louis           ; | choose(num1st, num2nd);
    add esp, 8           ; /
    mov ebx, 0
    mov eax, 1           ; sys_exit
    int 0x80             ; 系统调用

; void myprint(char* msg, int len)
myprint:
    mov edx, [esp + 8] ; len
    mov ecx, [esp + 4] ; msg
    mov ebx, 1
    mov eax, 4           ; sys_write
    int 0x80             ; 系统调用
    ret

```

对 C 语言代码进行修改，调用函数 `myprint`，函数 `louis` 用于汇编调用。

```
shizi@shizi-virtual-machine:~/Desktop/chapter5/b$ nasm -f elf foo.asm -o foo.o
shizi@shizi-virtual-machine:~/Desktop/chapter5/b$ gcc -c bar.c -o bar.o
shizi@shizi-virtual-machine:~/Desktop/chapter5/b$ ld -s -o foobar foo.o bar.o
shizi@shizi-virtual-machine:~/Desktop/chapter5/b$ ./foobar
2021301051114 czy
2021302191791 wz
2021302191536 ns
2021302121234 lx

2021301051114 czy
2021302191791 wz
2021302191536 ns
2021302121234 lx
```

输出了 `b-a` 次的预设结果。

2. ELF 文件格式使用 `xxd` 命令分析 ELF 文件中的 ELF header、Program header，了解各项的作用

1) ELF header

具体的 ELF 文件格式以及作用的描述在后续思考题部分讲述，这里我们直接使用 `xxd` 命令分析编译后生成的 ELF 文件中的 ELF header

```
wz@ubuntu:~/Desktop/chapter5/b$ xxd -u -a -g 1 -c 16 -l 80 foobar
00000000: 7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00 .ELF.....
00000010: 02 00 03 00 01 00 00 00 A0 80 04 08 34 00 00 00 .....4....
00000020: D4 01 00 00 00 00 00 00 34 00 20 00 03 00 28 00 .....4. ...
00000030: 07 00 06 00 01 00 00 00 00 00 00 00 80 04 08 .....
00000040: 00 80 04 08 64 01 00 00 64 01 00 00 05 00 00 00 ....d.....
wz@ubuntu:~/Desktop/chapter5/b$
```

图 x ELF header

开头的 4 字节固定不变，第一个字节值为 0x7F，紧跟着的就是 ELF 三个字符，这 4 字节表明这个文件是一个 ELF 文件，前 16 个字节标识 ELF 文件格式，从 ascii 码中可以看到 ELF 字符。

ELF header 的格式如下面代码所示：

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

下面是 ELF header 中各项的含义：

- **e\_type**：它标识的是该文件的类型，文件 foobar 的 e\_type 是 0x02，表明它是一个可执行文件（ExecutableFile）。
- **e\_machine**：foobar 中此项的值为 3，表明运行该程序需要的体系结构为 Intel80386。
- **e\_version**：它确定文件的版本，foobar 中的版本值是 1。
- **e\_entry**：程序的入口地址。文件 foobar 的入口地址为 0x80480A0。
- **e\_phoff**：Program header table 在文件中的偏移量（以字节计数）。这里的值是 0x34。
- **e\_shoff**：Section header table 在文件中的偏移量（以字节计数）。这里的值是 0x350。
- **e\_flags**：对 IA32 而言，此项为 0。
- **e\_ehsize**：ELFheader 大小（以字节计数）。这里值为 0x34。
- **e\_phentsize**：Program header table 中每一个条目（一个 Programheader）的大小。这里值为 0x20。
- **e\_phnum**：Program header table 中有多少个条目，这里有 3 个。
- **e\_shentsize**：Section header table 中每一个条目（一个 Sectionheader）的大小，这里值为 0x28。
- **e\_shnum**：Section header table 中有多少个条目，这里有 7 个。

- e\_shstrndx 包含节名称的字符串表是第几个节（从零开始数）。这里值为 6，表示第 6 个节包含节名称。

2) Program header

我们看到。Program header table 在文件中的偏移量是 0x34，而 ELF header 大小 (e\_ehsize) 也是 0x34，可见 ELF header 后面紧接着就是 Program header table。我们使用 xxd 来分析 Program header，它描述的是系统准备程序运行所需的一个段 (Segment) 或其他信息。

```
wz@ubuntu:~/Desktop/chapter5/b$ xxd -u -a -g 1 -c 16 -s 0x34 -l 0x60 foobar
00000034: 01 00 00 00 00 00 00 00 80 04 08 00 80 04 08 .....
00000044: 64 01 00 00 64 01 00 00 05 00 00 00 10 00 00 d...d.....
00000054: 01 00 00 00 64 01 00 00 64 91 04 08 64 91 04 08 ....d...d...d...
00000064: 08 00 00 00 08 00 00 00 06 00 00 00 10 00 00 .....
00000074: 51 E5 74 64 00 00 00 00 00 00 00 00 00 00 00 Q.td.....
00000084: 00 00 00 00 00 00 00 00 07 00 00 00 10 00 00 .....
wz@ubuntu:~/Desktop/chapter5/b$
```

程序头表中共有三项 (e\_phnum=3)，偏移分别是 0x34~0x53、0x54~0x73 和 0x74~0x93。

Program header 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  ;
```

其中各项的含义如下：

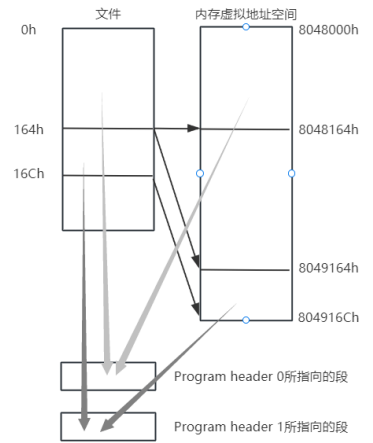
- p\_type: 当前 Program header 所描述的段的类型
- p\_offset: 段的第一个字节在文件中的偏移
- p\_vaddr: 段的第一个字节在内存中的虚拟地址
- p\_paddr: 在物理地址定位相关的系统中，此项是为物理地址保留
- p\_filesz: 段在文件中的长度
- p\_memsz: 段在内存中的长度
- p\_flags: 与段相关的标志
- p\_align: 根据此项值来确定段在文件以及内存中如何对齐

在 foobar 中共有三个 Program header，取值表如下所示：

名称	Program0	Program1	Program2
----	----------	----------	----------

p_type	0x1	0x1	0x6474E551
p_offset	0x0	0x164	0x0
p_vaddr	0x8048000	0x8049164	0x0
p_paddr	0x8048000	0x8049164	0x0
p_filesz	0x164	0x8	0x0
p_memsz	0x164	0x8	0x0
p_flags	0x5	0x6	0x7
p_align	0x1000	0x1000	0x10

根据上述消息，可以大致画出文件内偏移地址到虚拟地址的映射关系。



事实上，这也就是 foobar 在加载进内存之后的情形。

3. 使用 Loader 加载 ELF 文件，重新放置内核

Loader 需要做两项工作：1. 加载内核到内存；2. 跳入保护模式

a) 加载内核到内存

我们希望使用 Loader 加载 ELF 文件，在之后肯定是要加载内核文件的。加载一个文件的步骤依旧是寻找文件、定位文件以及读入内存。我们修改 loader.asm，让它把内核放进内存。

首先，我们在 a 盘的目录下寻找 KERNEL.BIN

```

; 下面在 A 盘的根目录寻找 KERNEL.BIN
mov     word [wSectorNo], SectorNoOfRootDirectory
xor     ah, ah ; `.`
xor     dl, dl ; | 软驱复位
int     13h ; |
LABEL_SEARCH_IN_ROOT_DIR_BEGIN:
cmp     word [wRootDirSizeForLoop], 0 ; `.`
jz      LABEL_NO_KERNELBIN ; | 判断根目录区是不是已经读完,
dec     word [wRootDirSizeForLoop] ; / 读完表示没有找到 KERNEL.BIN
mov     ax, BaseOfKernelFile
mov     es, ax ; es <- BaseOfKernelFile
mov     bx, OffsetOfKernelFile ; bx <- OffsetOfKernelFile
mov     ax, [wSectorNo] ; ax <- Root Directory 中的某 Sector 号
mov     cl, 1
call    ReadSector

mov     si, KernelFileName ; ds:si -> "KERNEL BIN"
mov     di, OffsetOfKernelFile
cld
mov     dx, 10h
LABEL_SEARCH_FOR_KERNELBIN:
cmp     dx, 0 ; `.`
jz      LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR; | 循环次数控制, 如果已经读完
dec     dx ; / 了一个 Sector, 就跳到下一个
mov     cx, 11

```

接着，我们执行以下操作去定位文件

```

LABEL_FILENAME_FOUND: ; 找到 KERNEL.BIN 后便来到这里继续
mov     ax, RootDirSectors
and     di, 0FFF0h ; di -> 当前条目的开始

push    eax
mov     eax, [es : di + 01Ch] ; `.`
mov     dword [dwKernelSize], eax ; / 保存 KERNEL.BIN 文件大小
pop     eax

add     di, 01Ah ; di -> 首 Sector
mov     cx, word [es:di]
push    cx ; 保存此 Sector 在 FAT 中的序号
add     cx, ax
add     cx, DeltaSectorNo ; cl <- KERNEL.BIN 的起始扇区号(0-based)
mov     ax, BaseOfKernelFile
mov     es, ax ; es <- BaseOfKernelFile
mov     bx, OffsetOfKernelFile ; bx <- OffsetOfKernelFile
mov     ax, cx ; ax <- Sector 号

```

可以看到，代码的大致工作和 boot.asm 是类似的。

加载内核的代码大致完成,这里我们使用一个最简单的 kernel.asm 文件作为内核(实际算不上内核)来测试，实现的功能是显示一个字符“K”

```

; 编译链接方法
; $ nasm -f elf kernel.asm -o kernel.o
; $ ld -s kernel.o -o kernel.bin #-s'选项意为"strip all"

[section .text] ; 代码在此

global _start ; 导出 _start

_start: ; 跳到这里来的时候，我们假设 gs 指向显存
mov     ah, 0Fh ; 0000: 黑底 1111: 白字
mov     al, 'K'
mov     [gs:((80 * 1 + 39) * 2)], ax ; 屏幕第 1 行, 第 39 列
jmp     $

```

首先修改 bochsrc，然后执行以下命令进行编译：

```
$ nasm -f elf -o kernel.o kernel.asm
```

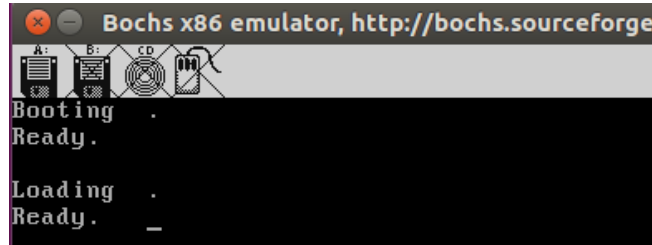
```
$ld -s -o kernel.bin kernel.o
```

```
$sudo mount -o loop a.img /mnt/floppy/

$sudo cp kernel.bin /mnt/floppy/ -v

$sudo umount /mnt/floppy/
```

结果如下所示：



可以看到，Loading 后面出现一个圆点，说明 Loader 读了一个扇区。现在，内核被我们加载进内存了，不过除了能看到“Ready.”字样之外，没有其他现象出现。

#### b. 跳入保护模式

不同于之前在保护模式中学习的：大部分描述符的段基址是运行时计算后填入相应位置，因为我们不知道段地址，也就不知道程序运行时在内存中的位置。现在，我们不需要这样了，因为我们自己加载了 loader，已经确定了段地址为 BaseOfLoader，所以在 Loader 中出现的变量的物理地址可以由以下公式计算：

$$\text{标号物理地址} = \text{BaseOfLoader} * 10h + \text{标号的偏移}$$

这样就导致 BaseOfLoader 同时在 boot.asm 和 loader.asm 中使用，于是我们将 BaseOfLoader 定义在一个文件 load.inc 中

```
BaseOfLoader      equ 09000h ; LOADER.BIN 被加载到的位置 ---- 段地址
OffsetOfLoader    equ 0100h ; LOADER.BIN 被加载到的位置 ---- 偏移地址

BaseOfLoaderPhyAddr equ BaseOfLoader*10h ; LOADER.BIN 被加载到的位置 ---- 物理地址

BaseOfKernelFile  equ 08000h ; KERNEL.BIN 被加载到的位置 ---- 段地址
OffsetOfKernelFile equ 0h ; KERNEL.BIN 被加载到的位置 ---- 偏移地址
```

直接定义了一个宏 BaseOfLoaderPhyAddr 来表示 BaseOfLoader \* 10h

接下来进入保护模式，同之前实验操作，进入之后打印字符“P”



```

; 从此以后的代码在保护模式下执行 -----
; 32 位代码段, 由实模式跳入 -----
[SECTION .s32]

ALIGN 32

[BITS 32]

LABEL_PM_START:
    mov     ax, SelectorVideo
    mov     gs, ax

    mov     ax, SelectorFlatRW
    mov     ds, ax
    mov     es, ax
    mov     fs, ax
    mov     ss, ax
    mov     esp, TopOfStack

    push    szMemChkTitle
    call    DispStr
    add     esp, 4

    call    DispMemInfo
    call    SetupPaging

    mov     ah, 0Fh                ; 0000: 黑底    1111: 白字
    mov     al, 'P'
    mov     [gs:((80 * 0 + 39) * 2)], ax ; 屏幕第 0 行, 第 39 列
    jmp     $

%include    "lib.inc"

```

执行以下命令进行编译运行:

```

$ nasm loader.asm -o loader.bin

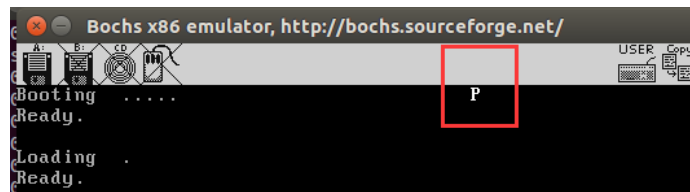
$ sudo mount -o loop a.img /mnt/floppy/

$ sudo cp loader.bin /mnt/floppy/

$ sudo umount /mnt/floppy/

$ sudo bochs -f bochsrc

```



看到字母“P”，说明成功进入保护模式。

接下来我们打开分页机制，同之前实验，我们先使用 15h 中断得到获取内存信息。

```

; 得到内存数
mov     ebx, 0                ; ebx = 后续值, 开始时需为 0
mov     di, _MemChkBuf        ; es:di 指向一个地址范围描述符结构(ARDS)
.MemChkLoop:
    mov     eax, 0E820h        ; eax = 0000E820h
    mov     ecx, 20            ; ecx = 地址范围描述符结构的大小
    mov     edx, 0534D4150h    ; edx = 'SMAP'
    int     15h               ; int 15h
    jc      .MemChkFail
    add     di, 20
    inc     dword [_dwMCRNumber] ; dwMCRNumber = ARDS 的个数
    cmp     ebx, 0
    jne     .MemChkLoop
    jmp     .MemChkOK
.MemChkFail:
    mov     dword [_dwMCRNumber], 0
.MemChkOK:

```

接着显示并打印内存信息。

```

; 显示内存信息 -----
DispMemInfo:
push     esi
push     edi
push     ecx

mov     esi, MemChkBuf
mov     ecx, [dwMCRNumber]; for(int i=0; i<[MCRNumber]; i++) // 每次得到一个ARDS
.loop:
mov     edx, 5 ; for(int j=0; j<5; j++) // 每次得到一个ARDS中的成员
mov     edi, ARDStruct ; // 依次显示: BaseAddrLow, BaseAddrHigh, LengthLow
; LengthHigh, Type
.1:
push     dword [esi]
call     DispInt ; DispInt(MemChkBuf[j*4]); // 显示一个成员
pop     eax
stosd ; ARDStruct[j*4] = MemChkBuf[j*4];
add     esi, 4
dec     edx
cmp     edx, 0
jnz     .1
call     DispReturn ; printf("\n");
cmp     dword [dwType], 1 ; if(Type == AddressRangeMemory)
jne     .2
mov     eax, [dwBaseAddrLow];
add     eax, [dwLengthLow];
cmp     eax, [dwMemSize] ; if(BaseAddrLow + LengthLow > MemSize)
jb     .2
mov     [dwMemSize], eax ; MemSize = BaseAddrLow + LengthLow;
.2:
loop     .loop
call     DispReturn ; printf("\n");
push     szRAMSize
call     DispStr ; printf("RAM size:");
add     esp, 4
push     dword [dwMemSize]
call     DispInt ; DispInt(MemSize);
add     esp, 4

pop     ecx
pop     edi
pop     esi
ret

```

得到内存信息之后，启动分页机制，这个基本类似之前实验，故不放图。

这里运行之后，结果如下所示：

The screenshot shows the Bochs x86 emulator window. The output text is as follows:

```

Bochs x86 emulator, http://bochs.sourceforge.net/
Booting ..... P
Ready.
Loading .
Ready.
BaseAddrL BaseAddrH LengthLow LengthHigh Type
00000000h 00000000h 0009F000h 00000000h 00000001h
0009F000h 00000000h 00001000h 00000000h 00000002h
000E8000h 00000000h 00018000h 00000000h 00000002h
00100000h 00000000h 01FF0000h 00000000h 00000001h
01FF0000h 00000000h 00010000h 00000000h 00000003h
FFFC0000h 00000000h 00040000h 00000000h 00000002h
RAM size:01FF0000h

```

## 2) 重新加载内核

前面编写了一个十分简单的 kernel 函数并加载到了内存当中，现在就需要对其进行整理，并移交控制权。

### a) 整理内核程序

编写的内核程序是一个 ELF 程序，ELF 程序的 program header table 字段有重要含义，整理这个内核程序，实际上是基于 program header table 的信息进行如下面 C 语言语句的内存复制：

```
memcpy(p_vaddr, BaseOfLoaderPhyAddr + p_offset, p_filesz)
```

但是，由于 ld 生成的可执行文件中 p\_vaddr 的值较大，在这里超过我们的内存范围，

所以我们需要修改一下 ld 指令时的参数。编译时使用以下命令：

```
$nasm -f elf -o kernel.o kernel.asm
```

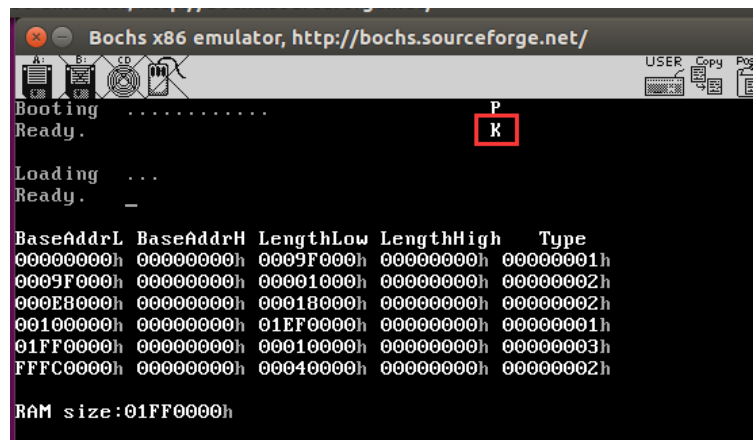
```
$ld -s -Ttext 0x30400 -o kernel.bin kernel.o
```

这样就解决了内存的问题，只需要向内核交出控制权

b. 移交控制权

```
;*****  
jmp     SelectorFlatC:KernelEntryPointPhyAddr ; 正式进入内核 *  
;*****
```

运行结果如下所示：



```
Bochs x86 emulator, http://bochs.sourceforge.net/  
Booting Ready. P  
K  
Loading Ready.  
BaseAddrL BaseAddrH LengthLow LengthHigh Type  
00000000h 00000000h 0009F000h 00000000h 00000001h  
0009F000h 00000000h 00001000h 00000000h 00000002h  
000E8000h 00000000h 00018000h 00000000h 00000002h  
00100000h 00000000h 01EF0000h 00000000h 00000001h  
01FF0000h 00000000h 00010000h 00000000h 00000003h  
FFFC0000h 00000000h 00040000h 00000000h 00000002h  
RAM size:01FF0000h
```

可以看到，第二行中央出现字符“K”，说明我们的内核开始执行了。

4. 扩展内核，切换堆栈和 GDT、整理文件结构、使用 makefile 编译程序、添加中断处理

切换堆栈和 GDT 的汇编代码如下：

```
78 ; 把 esp 从 LOADER 挪到 KERNEL  
79 mov esp, StackTop ; 堆栈在 bss 段中  
80  
81 sgdt [gdt_ptr] ; cstart() 中将会用到 gdt_ptr  
82 call cstart ; 在此函数中改变了 gdt_ptr，让它指向新的 GDT  
83 lgdt [gdt_ptr] ; 使用新的 GDT  
84  
85 ;lidt [idt_ptr]  
86  
87 jmp SELECTOR_KERNEL_CS:cstart  
88 cstart: ; “这个跳转指令强制使用刚刚初始化的结构”——<<OS:D&I 2nd>> P90.  
89  
90 push 0  
91 popfd ; Pop top of stack into EFLAGS  
92  
93 hlt  
94
```

C 语言函数代码如下：函数 cstart() 首先把 Loader 中的原 GDT 复制给新的 GDT，然后把 gdt\_ptr 中内容换成新的 GDT 的基地址和界限。

```
7 #include "type.h"
8 #include "const.h"
9 #include "protect.h"
10
11 PUBLIC void* memcpy(void* pDst, void* pSrc, int isize);
12
13 PUBLIC void disp_str(char * pszInfo);
14
15 PUBLIC u8 gdt_ptr[6]; /* 0~15:Limit 16~47:Base */
16 PUBLIC DESCRIPTOR gdt[GDT_SIZE];
17
18 PUBLIC void cstart()
19 {
20     /*disp_str("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n")
21      "-----"cstart\'' begins-----\n");*/
22
23     /* 将 LOADER 中的 GDT 复制到新的 GDT 中 */
24     memcpy(&gdt, /* New GDT */
25            (void*)((u32*)(&gdt_ptr[2]])), /* Base of Old GDT */
26            ((u16*)(&gdt_ptr[0])) + 1 /* Limit of Old GDT */
27           );
28     /* gdt_ptr[6] 共 6 个字节: 0~15:Limit 16~47:Base. 用作 sgdt/lgdt 的参数。*/
29     u16* p_gdt_limit = (u16*)(&gdt_ptr[0]);
30     u32* p_gdt_base = (u32*)(&gdt_ptr[2]);
31     *p_gdt_limit = GDT_SIZE * sizeof(DESCRIPTOR) - 1;
32     *p_gdt_base = (u32)&gdt;
```

其中用到新定义的类型、结构体和宏可在 `type.h`、`const.h` 以及 `protect.h` 中找到。编译链接：

```
liu@liu-VirtualBox:~/Desktop/lab6$ nasm -f elf -o kernel.o kernel.asm
liu@liu-VirtualBox:~/Desktop/lab6$ nasm -f elf -o string.o string.asm
liu@liu-VirtualBox:~/Desktop/lab6$ gcc -c -o start.o start.c
liu@liu-VirtualBox:~/Desktop/lab6$ ld -s -Ttext 0x30400 -o kernel.bin kernel.o s
tring.o start.o
```

运行，结果无事发生。

添加打印字符的代码:

```

17 ;
18 ; void disp_str(char * info);
19 ;
20 disp_str:
21     push    ebp
22     mov     ebp, esp
23
24     mov     esi, [ebp + 8] ; pszInfo
25     mov     edi, [disp_pos]
26     mov     ah, 0Fh
27 .1:
28     lodsb
29     test    al, al
30     jz      .2
31     cmp     al, 0Ah ; 是回车吗?
32     jnz     .3
33     push    eax
34     mov     eax, edi
35     mov     bl, 160
36     div     bl
37     and     eax, 0FFh
38     inc     eax
39     mov     bl, 160
40     mul     bl
41     mov     edi, eax
42     pop     eax
43     jmp     .1
44 .3:
45     mov     [gs:edi], ax
46     add     edi, 2
47     jmp     .1
48
49 .2:
50     mov     [disp_pos], edi
51
52     pop     ebp
53     ret

```

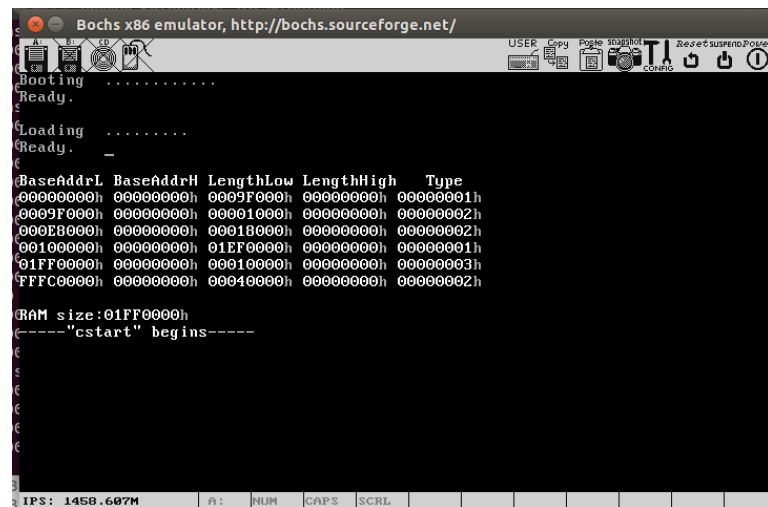
在 start.c 中调用:

[illegible]

重新编译:

```
liu@liu-VirtualBox:~/Desktop/lab6$ nasm -f elf -o kernel.o kernel.asm
liu@liu-VirtualBox:~/Desktop/lab6$ nasm -f elf -o string.o string.asm
liu@liu-VirtualBox:~/Desktop/lab6$ nasm -f elf -o kliba.o kliba.asm
liu@liu-VirtualBox:~/Desktop/lab6$ gcc -c -fno-builtin -o start.o start.c
liu@liu-VirtualBox:~/Desktop/lab6$ ld -s -Ttext 0x30400 -o kernel.bin kernel.o s
tring.o start.o kliba.o
```

运行，结果如下：

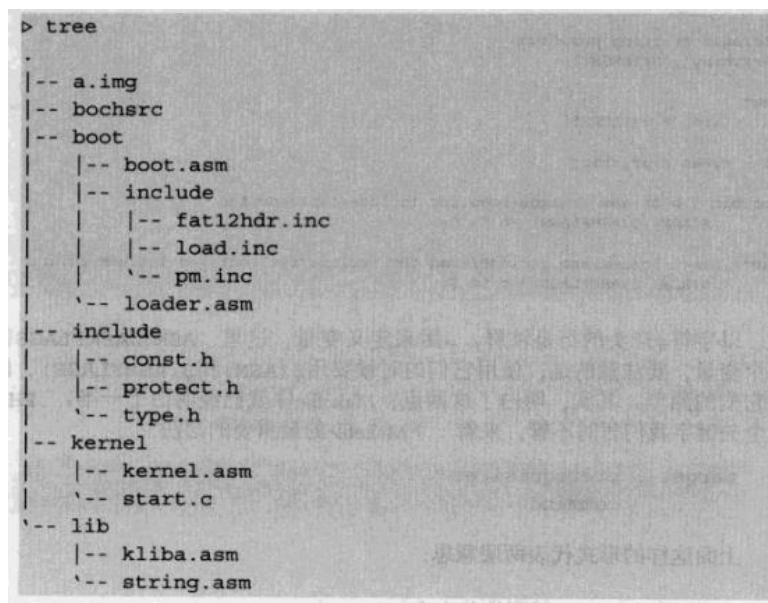


The screenshot shows the Bochs x86 emulator window. The title bar reads "Bochs x86 emulator, http://bochs.sourceforge.net/". The window contains a black area with white text indicating the boot process. The text shows "Booting" and "Loading" stages, both followed by "Ready.". Below this, a table of memory addresses and types is displayed. The table has five columns: BaseAddrL, BaseAddrH, LengthLow, LengthHigh, and Type. The rows show memory addresses from 00000000h to FFFC0000h. The text "RAM size: 01FF0000h" and "cstart" begins is also visible. At the bottom, a status bar shows "IPS: 1458.607M" and various flags like A, NUM, CAPS, SCRL.

```
Bochs x86 emulator, http://bochs.sourceforge.net/
Booting .....
Ready.
Loading .....
Ready.
BaseAddrL BaseAddrH LengthLow LengthHigh Type
00000000h 00000000h 0009F000h 00000000h 00000001h
0009F000h 00000000h 00001000h 00000000h 00000002h
000E8000h 00000000h 00018000h 00000000h 00000002h
00100000h 00000000h 01EF0000h 00000000h 00000001h
01FF0000h 00000000h 00010000h 00000000h 00000003h
FFFC0000h 00000000h 00040000h 00000000h 00000002h
RAM size: 01FF0000h
cstart" begins-----
IPS: 1458.607M  A:  NUM  CAPS  SCRL
```

a) 关于：Makefile

整理文件夹后，目录树如下所示：



The screenshot shows a terminal window with the output of the 'tree' command. The directory structure is as follows:

```
> tree
.
|-- a.img
|-- bochsrc
|-- boot
|   |-- boot.asm
|   |-- include
|   |   |-- fat12hdr.inc
|   |   |-- load.inc
|   |   '-- pm.inc
|   '-- loader.asm
|-- include
|   |-- const.h
|   |-- protect.h
|   '-- type.h
|-- kernel
|   |-- kernel.asm
|   '-- start.c
'-- lib
    |-- kliba.asm
    '-- string.asm
```

写一个 makefile 来编译/boot 文件夹内的 boot.bin 和 loader.bin:

```

1 # Makefile for boot
2
3 # Programs, flags, etc.
4 ASM = nasm
5 ASMFLAGS = -I include/
6
7 # This Program
8 TARGET = boot.bin loader.bin
9
10 # All Phony Targets
11 .PHONY : everything clean all
12
13 # Default starting position
14 everything : $(TARGET)
15
16 clean :
17     rm -f $(TARGET)
18
19 all : clean everything
20
21 boot.bin : boot.asm include/load.inc include/fat12hdr.inc
22     $(ASM) $(ASMFLAGS) -o $@ $<
23
24 loader.bin : loader.asm include/load.inc include/fat12hdr.inc include/pm.inc
25     $(ASM) $(ASMFLAGS) -o $@ $<
26
27

```

“#”开头的是注释；“=”用来定义变量，ASM,ASMFLAGS 就是变量，使用它们的时候需要 \$(ASM),\$(ASMFLAGS) 。

Makefile 的最后两行：

得到 loader.bin 文件，需要执行 “\$(ASM) \$(ASMFLAGS) -o \$@ \$<” 命令。

而 loader.bin 则依赖 boot/loader.asm boot/include/load.inc boot/include/fat12hdr.inc boot/include/pm.inc 文件，其中至少一个文件比 loader.bin 新时，command 被执行。

这条命令中\$@代表 target，\$<代表 prerequisites 的第一个名字。故而这条命令实际上等价于” nasm -o loader.bin loader.asm”。此外，在 makefile 中，除了 boot.bin 和 loader.bin 两个文件后面有冒号，everything、clean 和 all 后面也有冒号，但这三个不是文件，而是动作名称，比如运行 “make clean”，就会执行 “rm -f \$(TARGET)”，也就是 “rm -f boot.bin loader.bin”。

运行相关操作：

```

liu@liu-VirtualBox:~/Desktop/lab6/boot$ make all
rm -f boot.bin loader.bin
nasm -I include/ -o boot.bin boot.asm
nasm -I include/ -o loader.bin loader.asm
liu@liu-VirtualBox:~/Desktop/lab6/boot$ make clean
rm -f boot.bin loader.bin
liu@liu-VirtualBox:~/Desktop/lab6/boot$ make everything
nasm -I include/ -o boot.bin boot.asm
nasm -I include/ -o loader.bin loader.asm
liu@liu-VirtualBox:~/Desktop/lab6/boot$ make clean
rm -f boot.bin loader.bin
liu@liu-VirtualBox:~/Desktop/lab6/boot$ make
nasm -I include/ -o boot.bin boot.asm
nasm -I include/ -o loader.bin loader.asm
liu@liu-VirtualBox:~/Desktop/lab6/boot$ make
make: Nothing to be done for 'everything'.
liu@liu-VirtualBox:~/Desktop/lab6/boot$

```

b) 扩展 Makefile:

现在，我们来扩展 makefile，使得它能够编译链接我们整个项目。代码太长不具体列

输入 `make image` 来把引导扇区、`loader.bin` 和 `kernel.bin` 写入虚拟软盘:

往 start.c 里面添加一行打印代码:

重新 make 一下，启动，结果如下：

### 添加中断处理:

初始化 8259A:

```

9      #include "type.h"
10     #include "const.h"
11     #include "protect.h"
12     #include "proto.h"
13
14
15     /*=====
16     |                                     init_8259A
17     |=====*/
18     PUBLIC void init_8259A()
19     {
20         /* Master 8259, ICW1. */
21         out_byte(INT_M_CTL, 0x11);
22
23         /* Slave 8259, ICW1. */
24         out_byte(INT_S_CTL, 0x11);
25
26         /* Master 8259, ICW2. 设置 '主8259' 的中断入口地址为 0x20. */
27         out_byte(INT_M_CTLMASK, INT_VECTOR_IRQ0);
28
29         /* Slave 8259, ICW2. 设置 '从8259' 的中断入口地址为 0x28 */
30         out_byte(INT_S_CTLMASK, INT_VECTOR_IRQ8);
31
32         /* Master 8259, ICW3. IR2 对应 '从8259'. */
33         out_byte(INT_M_CTLMASK, 0x4);
34
35         /* Slave 8259, ICW3. 对应 '主8259' 的 IR2. */
36         out_byte(INT_S_CTLMASK, 0x2);
37
38         /* Master 8259, ICW4. */
39         out_byte(INT_M_CTLMASK, 0x1);
40
41         /* Slave 8259, ICW4. */
42         out_byte(INT_S_CTLMASK, 0x1);
43
44         /* Master 8259, OCW1. */
45         out_byte(INT_M_CTLMASK, 0xFF);
46
47         /* Slave 8259, OCW1. */
48         out_byte(INT_S_CTLMASK, 0xFF);
49     }
50
51

```

宏定义:

const.h

```

28     /* 8259A interrupt controller ports. */
29     #define INT_M_CTL    0x20 /* I/O port for interrupt controller    <Master> */
30     #define INT_M_CTLMASK 0x21 /* setting bits in this port disables ints <Master> */
31     #define INT_S_CTL    0x28 /* I/O port for second interrupt controller<Slave> */
32     #define INT_S_CTLMASK 0x29 /* setting bits in this port disables ints <Slave> */

```

protect.h

```

95     /* 中断向量 */
96     #define INT_VECTOR_IRQ0    0x20
97     #define INT_VECTOR_IRQ8    0x28

```

函数 `init_8259A` 只用到一个函数，就是用来写端口的 `out_byte`，此外我们还添加了 `in_byte` 用于对端口进行读操作，由于端口操作需要时间，所以两个函数都加了 `nop` 指令来添加延迟:

```

96     ; =====
97     ;                                     void out_byte(u16 port, u8 value);
98     ; =====
99     out_byte:
100     mov edx, [esp + 4]    ; port
101     mov al, [esp + 4 + 4] ; value
102     out dx, al
103     nop ; 一点延迟
104     nop
105     ret
106
107     ; =====
108     ;                                     u8 in_byte(u16 port);
109     ; =====
110     in_byte:
111     mov edx, [esp + 4]    ; port
112     xor eax, eax
113     in al, dx
114     nop ; 一点延迟
115     nop
116     ret
117

```

将函数声明写入头文件 `include/proto.h` 中，同时我们还将 `disp_str` 的声明也移入了头文件中，将 `memcpy` 放入头文件 `string.h` 中。



修改 makefile: 可以使用 gcc 的-M 参数来自动生成依赖关系, 然后再把依赖复制到 makefile 中。

```
liu@liu-VirtualBox:~/Desktop/BookCode/chapter5/h$ gcc -M kernel/start.c -I include
de
start.o: kernel/start.c /usr/include/stdc-predef.h include/type.h \
include/const.h include/protect.h include/proto.h include/string.h \
include/global.h
liu@liu-VirtualBox:~/Desktop/BookCode/chapter5/h$
```

接下来, 初始化 IDT, 修改 start.c:

```
35 /* idt_ptr[6] 共 6 个字节: 0~15:Limit 16~47:Base. 用作 sidt/lidt 的参数. */
36 u16* p_idt_limit = (u16*)(&idt_ptr[0]);
37 u32* p_idt_base = (u32*)(&idt_ptr[2]);
38 *p_idt_limit = IDT_SIZE * sizeof(GATE) - 1;
39 *p_idt_base = (u32)&idt;
```

c) 定义中断和异常:

我们对异常处理的主体思想是, 如果有错误码, 则直接把向量号压栈, 然后执行一个函数 exception\_handler, 如果没有错误码, 则先在栈中压入一个 0xFFFFFFFF, 再把向量号压栈并随后执行 exception\_handler。我们使用 C 语言调用函数时无需担心破坏堆栈中的 eip, cs 和 eflags。代码示例如下:

```
111 ; 中断和异常 -- 异常
112 divide_error:
113     push    0xFFFFFFFF ; no err code
114     push    0          ; vector_no = 0
115     jmp     exception

171 exception:
172     call    exception_handler
173     add     esp, 4*2 ; 让栈顶指向 EIP, 堆栈中从顶向下依次是: EIP、CS、EFLAGS
174     hlt
```

函数 exception\_handler, 即异常处理函数, 其主要思路就是把屏幕前 5 行通过打印空格的方式清空, 然后再把堆栈当中的参数打印出来。

```
115 PUBLIC void exception_handler(int vec_no, int err_code, int eip, int cs, int eflags)
116 {
117     int i;
118     int text_color = 0x74; /* 灰底红字 */
119
120     char * err_msg[] = {"#DE Divide Error",
121                         "#DS RESERVED",
122                         "-- NMJ Interrupt",
123                         "#BP Breakpoint",
124                         "#OF Overflow",
125                         "#BR BOUND Range Exceeded",
126                         "#UD Invalid Opcode (Undefined Opcode)",
127                         "#NM Device Not Available (No Math Coprocessor)",
128                         "#DF Double Fault",
129                         "  Coprocessor Segment Overrun (reserved)",
130                         "#TS Invalid TSS",
131                         "#NP Segment Not Present",
132                         "#SS Stack-Segment Fault",
133                         "#GP General Protection",
134                         "#PF Page Fault",
135                         "-- (Intel reserved. Do not use.)",
136                         "#MF x87 FPU Floating-Point Error (Math Fault)",
137                         "#AC Alignment Check",
138                         "#MC Machine Check",
139                         "#XF SIMD Floating-Point Exception"
140     };
141
142     /* 通过打印空格的方式清空屏幕的前五行, 并把 disp_pos 清零 */
143     disp_pos = 0;
144     for(i=0; i<5; i++){
145         disp_str(" ");
146     }
147     disp_pos = 0;
148
149     disp_color_str("Exception! -> ", text_color);
150     disp_color_str(err_msg[vec_no], text_color);
151     disp_color_str("\n\n", text_color);
152     disp_color_str("EFLAGS:", text_color);
153     disp_int(eflags);
154     disp_color_str("CS:", text_color);
155     disp_int(cs);
156     disp_color_str("EIP:", text_color);
157     disp_int(eip);
158
159     if(err_code != 0xFFFFFFFF){
160         disp_color_str("Error code:", text_color);
161         disp_int(err_code);
162     }
```

d) 设置 IDT:

```

93  /*=====*/
94          init_idt_desc
95  /*=====*/
96  初始化 386 中断门
97  /*=====*/
98  PRIVATE void init_idt_desc(unsigned char vector, u8 desc_type,
99                          int_handler handler, unsigned char privilege)
100  {
101      GATE * p_gate = &idt[vector];
102      u32 base      = (u32)handler;
103      p_gate->offset_low = base & 0xFFFF;
104      p_gate->selector   = SELECTOR_KERNEL_CS;
105      p_gate->dcount     = 0;
106      p_gate->attr       = desc_type | (privilege << 5);
107      p_gate->offset_high = (base >> 16) & 0xFFFF;
108  }

```

调用 `init_prot()`:

```
41     init_prot();
42
43     disp_str("-----\"cstart\" ends-----\n");
```

```
100 csinit: ; 这个跳转指令强制使用刚刚初始化的结构
101
102     ud2
```

```

C:\Program Files\VirtualBox>./Desktop/BochsCode/bochs.exe
Bochs x86 emulator, http://bochs.sourceforge.net/
Exception! --> #UD Invalid Opcode (Undefined Opcode)
EFLAGS:0x10046 CS:0x8 EIP:0x30430

BaseAddrL BaseAddrH LengthLow LengthHigh Type
00000000h 00000000h 0009F000h 00000000h 00000001h
0009F000h 00000000h 00001000h 00000000h 00000002h
000E8000h 00000000h 00018000h 00000000h 00000002h
00100000h 00000000h 01EF0000h 00000000h 00000001h
01FF0000h 00000000h 00010000h 00000000h 00000003h
FFFF0000h 00000000h 00040000h 00000000h 00000002h

RAM size:01FF0000h
-----"cstart" begins-----
-----"cstart" ends-----

```

使用 8259A 进行中断:

现在已经完成了 8259A 的设置, 在 kernel.asm 中定义了所有的中断例程, 所有的中断都会触发一个 spurious\_irq 的函数:

```
54 PUBLIC void spurious_irq(int irq)
55 {
56     disp_str("spurious_irq: ");
57     disp_int(irq);
58     disp_str("\n");
59 }
```

这个函数其实就是把 IRQ 号打印出来。

接下来设置 IDT (显示部分示例):

```
35 void hwint00();
36 void hwint01();
37 void hwint02();
38 void hwint03();
```

```
50 void hwint15();
51
52 /*=====
53      init_prot
54      =====*/
55 PUBLIC void init_prot()
56 {
57     init_8259A();
58
59     // 全部初始化成中断门(没有陷阱门)
60     init_idt_desc(INT_VECTOR_DIVIDE, DA_386IGate,
61                  divide_error, PRIVILEGE_KRNL);
```

现在已经可以运行了, 只不过不会有结果, 因为我们没有通过任何方式来设置 IF 位, 而且在 init\_8259A()里把中断都屏蔽了, 现在修改 i8259.c, 打开键盘中断:

```
44 /* Master 8259, OCW1. */
45 out_byte(INT_M_CTLMASK, 0xFD);
46
47 /* Slave 8259, OCW1. */
48 out_byte(INT_S_CTLMASK, 0xFF);
```

在这里, 我们向主 8259A 相应端口写入了 0xFD, 即打开了键盘中断, 其他中断仍然处于屏蔽状态。然后在 kernel.asm 中添加 sti 指令设置 IF 位:

```
117 csinit:
118     sti
119     hlt
```

make 运行, 并敲击键盘:

make，运行，开始没有什么特殊现象，但当我们敲击键盘任意键时，出现字符串“spurious\_irq: 0x1”，这表明当前的 IRQ 号为 1，正是对应的键盘中断。

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

#### ➤ 启动显示 ASCII 图案

设计自定义的 ASCII 图案，并在 kernel/start.c 中编写函数，显示图案。

```
PUBLIC void printmylogo()
{
    // 使用disp_str函数打印一个有趣的logo
    u8 *cat =
        "  /\_/\  \n"
        "(  o.o  ) \n"
        "> ^ < \n";
    disp_str(cat);
}
```

在 kernel/start.c 中 global 导出 printBootLogo 函数，在 kernel/kernel.asm 中 extern 声明

printBootLogo 函数，同时在合适的位置调用。成功输出既定的图案。

#### ➤ 中断函数集成

为便于测试函数效果，修改 kernel/kernel.asm 代码，使 kernel 进入死循环（不然 hlt 被中断唤醒后会继续执行 hlt 后面的指令，导致出现错误，使得程序运行至非预设位

置)

```
csinit:
    sti
deadloop:S
    hlt
    jmp deadloop
```

修改 hwint01 函数，改成自己需要添加的中断处理代码。

处理键盘中断，递增存储在 gs 段指定位置的字节值，然后从键盘控制器读取一个字节，向主片（PIC）发送一个中断结束的信号，并最终执行中断返回指令。主要功能是实现按键盘切换固定位置颜色的中断功能。

```
ALIGN 16
hwint01: ; Interrupt routine for irq 1 (keyboard)
    inc byte [gs:((80 * 10 + 35) * 2 + 1)]
    in al, 0x60
    mov al, 20h
    out 20h, al
    iretd
;hwint_master _ 1
```

测试程序，每次按键盘，发现颜色变化，中断程序集成成功。

```
BaseAddrL BaseAddrH LengthLow LengthHigh Type
00000000h 00000000h 0009F000h 00000000h 00000001h
0009F000h 00000000h 00001000h 00000000h 00000002h
000E8000h 00000000h 00018000h 00000000h 00000002h
00100000h 00000000h 01EF0000h 00000000h 00000001h
01FF0000h 00000000h 00010000h 00000000h 00000003h
FFFC0000h 00000000h 00040000h 00000000h 00000002h
```

```
RAM size:01FF0000h
-----"cstart" begins-----
-----"cstart" ends-----
  ^ ^
( o.o )
> ^ <
```

```
BaseAddrL BaseAddrH LengthLow LengthHigh Type
00000000h 00000000h 0009F000h 00000000h 00000001h
0009F000h 00000000h 00001000h 00000000h 00000002h
000E8000h 00000000h 00018000h 00000000h 00000002h
00100000h 00000000h 01EF0000h 00000000h 00000001h
01FF0000h 00000000h 00010000h 00000000h 00000003h
FFFC0000h 00000000h 00040000h 00000000h 00000002h
```

```
RAM size:01FF0000h
-----"cstart" begins-----
-----"cstart" ends-----
  ^ ^
( o.o )
> ^ <
```

#### ➤ 内存管理集成

为了便于编译挂载，我们创建 mm 文件夹，将需要添加的内存管理的函数放入该文件夹的 pageManage.asm 文件中。

将第三章中添加的 Alloc\_pages 函数和 Free\_pages 函数拷贝进入 pageManage.asm 中，声明标号以及添加所需要的数据结构。

建立线性地址和物理地址的映射，不在于如何管理线性空间，因此简单的选取可用线性地址返回即可，给定线性地址和页大小（不用考虑是否越界的问题），修改对应页表项和页目录项，取消映射关系。

```

[SECTION .data]
_BitMap: times 32 db 0xff
         times 32 db 0x0
BitMapLen equ $ - $$
_AvaLinearAddress dd 0x8000_0000
global Linear2Physical
global alloc_pages
global free_pages

alloc_a_4k_page:
; save reg
push ds
push es
xor eax, eax

.search:
bts [_BitMap], eax
jnc .find
inc eax
cmp eax, BitMapLen*8
jl .search

; no available physical space
; we should move one page back to disk
; but for simplicity, we hlt
hlt

.find:
shl eax, 12
pop es
pop ds
ret

alloc_pages:
; save registers
push ds
push es
mov ecx, eax ; ecx means the number of page
mov ebx, 4096
mul ebx ; ebx means the size of pages

mov ebx, [es:_AvaLinearAddress] ; ebx means the return value
add [es:_AvaLinearAddress], eax ; update the address of free linear address
push ebx ; save the return value
mov eax, ebx
mov ebx, cr3
and ebx, 0xfffff000
and eax, 0xffc00000
shr eax, 20
add ebx, eax ; ebx means the pde item
mov edx, ebx
mov ebx, [ebx] ; ebx means the corresponding page table item

test ebx, 0x0000_0001
jnz .pde_exist

mov ebx, cr3
mov ebx, [ebx]
and ebx, 0xfffff000
shl eax, 10 ; eax means the size of used pages

```

在 kernel/kernel.asm 中添加测试函数并调用，被测试函数使用 extern 声明。

```

TestAllocAndFree:
; extern alloc_pages, free_pages

xchg bx, bx
mov eax, 4
call alloc_pages

xchg bx, bx
mov eax, ebx
mov ebx, 4
call free_pages
xchg bx, bx

ret

```

修改 Makefile，在 OBJS 中添加 mm/pageManage.o，在文件末尾添加对该文件的汇编命令。

```

mm/pageManage.o : mm/pageManage.asm
$(ASM) $(ASMFLAGS) -o $@ $<

```

在断点处分别查看页表的映射情况，alloc\_pages 和 free\_pages 也正常工作，内存管理程序集成成功。

查看 alloc\_pages 前地址映射关系：

```

<bochs:2> info tab
cr3: 0x000000200000
0x0000000000000000-0x0000000000000000 -> 0x0000000000000000-0x0000000000000000

```

查看 alloc\_pages 后地址映射关系：

```
<bochs:4> info tab
cr3: 0x000000200000
0x0000000000000000-0x0000000001ffffff -> 0x000000000000-0x000001ffffff
0x0000000080000000-0x0000000080003fff -> 0x000000100000-0x000000103fff
```

查看 free\_pages 后地址映射关系:

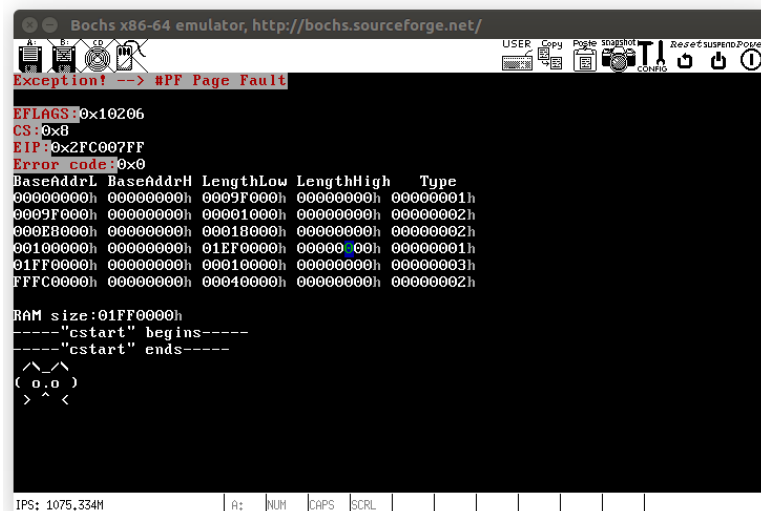
```
<bochs:6> info tab
cr3: 0x000000200000
0x0000000000000000-0x0000000001ffffff -> 0x000000000000-0x000001ffffff
```

### 三、实验过程分析

(实验分工, 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)

1. 未修改 kernel/kernel.asm 代码, 未使 kernel 进入死循环, 导致程序出错:

由于未修改 kernel/kernel.asm 代码, 未使 kernel 进入死循环, 使得 kernel 程序中的 hlt 被中断唤醒后会继续执行 hlt 后面的指令, 自动进入了后续的分页等函数, 而并非按照预先设定好的调用顺序执行代码, 造成错误。



2. 内存管理功能集成时未保存好寄存器内容:

在 free\_pages 函数中, 由于一开始忽略了对 eax 和 ebx 两个寄存器值的保存, 造成了对 linear address 和 page number 的丢失, 使得后续的内存释放过程中出现了页错误, 在及时保存以及恢复寄存器后, 错误解决。

```
free_pages:                ; arg  eax,linear address , ebx  page number
    push ds
    push es
    push ebx                ; save eax and ebx
    push eax
```

#### 四、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)

1. 汇编和 C 内定义的函数，相互间调用的方法是怎样的？

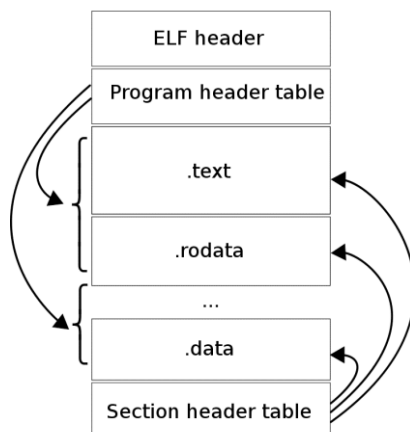
对于在 bar.c 中用到函数 myprint(), 所以要用关键字 global 将其导出。由于用到本文件外定义的函数 choose(), 所以要用关键字 extern 声明。不管是 myprint() 还是 choose(), 遵循的都是 C 调用约定 (C Calling Convention), 后面的参数先入栈, 并由调用者 (Caller) 清理堆栈。

c 语言和汇编互相调用的关键就在于关键字 global 和 extern, 有了这两个关键字就可以在 c 和 asm 之间自由来去。

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?

- 1) ELF 文件格式





可以看出，ELF 文件由 4 个部分组成，分别是 ELF header、Program header、节和 Section header table

- ELF header

ELF header 格式的代码见具体实验步骤 2 部分。ELF 文件力求支持从 8 位到 32 位不同架构的处理器。为了使文件格式与机器无关，定义了下表中的这些数据类型。

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
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	小符号小整数

- Program header

Program header 格式的代码见具体实验步骤 2 部分。

- Section header

Section header 的结构如下所示：

```

1.  typedef struct {
2.      Elf32_Word  sh_name  ;
3.      Elf32_Word  sh_type  ;
4.      Elf32_Word  sh_flags  ;
5.      Elf32_Addr  sh_addr  ;
6.      Elf32_Off   sh_offset ;
7.      Elf32_Word  sh_size  ;
8.      Elf32_Word  sh_link  ;
9.      Elf32_Word  sh_info  ;

```

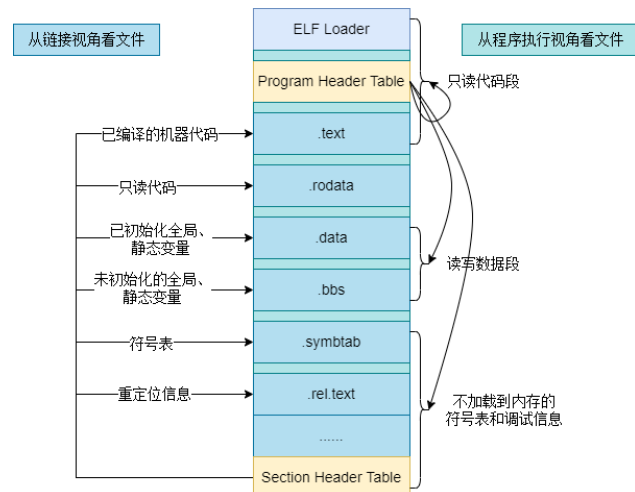
```

10. Elf32_Word  sh_addralign  ;
11. Elf32_Word  sh_entsize  ;
12. }Elf32_Shdr  ;

```

## 2) ELF 的作用

ELF 的作用可以用下图来进行表示：



ELF 文件参与程序的连接（建立一个程序）和程序的执行（运行一个程序）。在汇编器和链接器看来，ELF 文件是由 Section header table 描述的一系列 section 的集合，而执行一个 ELF 文件时，在加载器（Loader）看来它是由 Program header table 描述的一系列 segment 的集合。

## 3) ELF 和 PE 文件的对比

简单来说，ELF 对应于 UNIX 下的文件，而 PE 则是 Windows 的可执行文件，它们都是基于 Unix 的 COFF（Common Object file format）格式的变种，它们都包含了整个文件的基本属性，如文件版本，目标机器型号，程序入口等等。他们都有段表，保存了各种各样段的基本属性，比如段名，段长，文件中的偏移，读写权限。他们都有字符串表，将使用的字符串统一放在那里，然后通过偏移量来引用字符串。他们描述的是相同的信息，只是在数据段的链接方式上有所区别。

然而，二者之间也存在一些差异。例如，ELF 文件可以用于不同的硬件平台和操作系统，而 PE 文件主要使用在 32 位和 64 位的 Windows 操作系统上。此外，ELF 文件中的段可以有多种类型，包括代码段、数据段、只读数据段、调试信息等，而 PE 文件中的段类型可能会有所不同。这些差异主要是由于两种文件格式分别针对不同的操作系统进行了优化。

事实上，Linux 系统中有 wine 命令，它是一个能够在多种“POSIX-compliant”操作

系统上运行 Windows 应用的兼容层。wine 通过将 Linux 不能理解的指令翻译成 Linux 能够理解的指令，来实现 Linux 系统下运行 Windows 程序。

### 3. 如何从 Loader 加载 ELF，如何确定 ELF 文件加载到内存的位置？

加载 ELF (Executable and Linkable Format) 文件到内存是操作系统和程序加载器的核心功能之一。这个过程大致可以分为以下几个步骤：

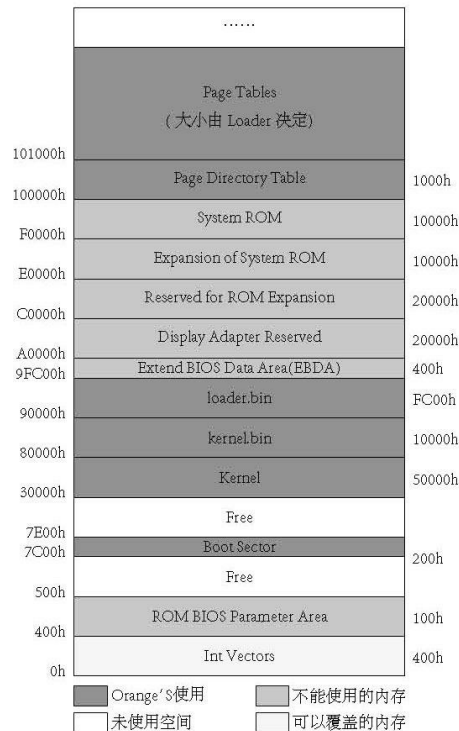
- **打开 ELF 文件：**加载器需要打开 ELF 文件。这通常涉及到读取文件系统中的数据。
- **解析 ELF 头：**ELF 文件的开始部分是 ELF 头，它包含了关于整个文件的元数据，如魔数、机器类型、入口点地址等。解析这些信息以了解如何处理文件的其余部分。
- **读取程序头表：**ELF 头后面是程序头表，它描述了文件中的各个段 (segments)。每个段有特定的类型，指示它如何被加载到内存中。例如，有些段包含可执行代码，有些则包含初始化数据。
- **加载段到内存：**加载器根据程序头表中的描述，将这些段加载到内存的适当位置。通常，加载到内存的地址由程序头表中的 `p_vaddr` (虚拟地址) 字段指定。有的系统会进行地址空间布局随机化 (ASLR)，这可能会改变加载地址。
- **处理重定位：**如果 ELF 文件是可重定位的，加载器需要根据重定位表调整代码和数据段中的某些地址。
- **设置程序入口点：**ELF 头中指定了程序的入口点地址。加载器需要确保当操作系统启动程序时，控制权能传递到这个地址。
- **启动程序：**最后，操作系统调用入口点地址，开始执行程序。

要确定 ELF 文件加载到内存的位置，可以查看程序头表中的 `p_vaddr` 字段，它指定了每个段应该被加载到的虚拟地址。在没有地址空间布局随机化 (ASLR) 的系统中，这通常是确定的地址。可以通过调试工具或在程序内部检查内存映射来获取实际的加载地址。

- ### 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？
- 功能及其流程和分析具体见实验过程。

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？

内存布局如下：



加载内核到内存的代码是非特权代码，此时我们没有进入保护模式。在把内核加载到内存后，该是跳入保护模式的时候了，在此时会切换处理器的特权级别，特权级别通常被设置为 Ring 0，后续执行的代码都是特权代码。常见的特权级别有用户态（非特权模式）和内核态（特权模式）。

```
160 LABEL_FILE_LOADED:
161
162 call KillMotor ; 关闭软驱马达
163
164 mov dh, 1 ; "Ready."
165 call DispStrRealMode ; 显示字符串
166
167 ; 下面准备跳入保护模式
168
169 ; 加载GDTR
170 lgdt [GdtPtr]
171
172 ; 关中断
173 cli
174
175 ; 打开地址线A20
176 in al, 92h
177 or al, 00000010b
178 out 92h, al
179
180 ; 准备切换到保护模式
181 mov eax, cr0
182 or eax, 1
183 mov cr0, eax
184
185 ; 真正进入保护模式
186 jmp dword SelectorFlatC: (BaseOfLoaderPhyAddr+LABEL_PM_START)
```

跳入保护模式后，后续运行的初始化寄存器、堆栈、显示内存、分页、整理内存中的内核并把控制权交给内核，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。

实验改进意见

王卓：

<p>1. 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。</p> <p>2. 介绍实验目的和背景：实验开始之前提供实验的目的和背景，解释一下为什么需要进行该实验以及其与操作系统的关系，这能帮助我们更好理解实验的意义和重要性。</p> <p>程子洋：</p> <p>错误排除指南：提供学生在遇到常见问题时进行自我排除的指南。这可以包括常见错误消息的解释以及如何解决这些问题的步骤。</p> <p>聂森：</p> <p>1. 可以先让同学阅读教材资料后再介绍实验内容和实验相关的知识，可以帮助同学更快更好的了解实验的目标和实验的原理及任务</p> <p>2. 给出具体的实验指导或给出更加丰富的实验参考资料，可以帮助同学更快上手实验并在有问题时找到解决方式</p> <p>刘虢：</p> <p>如果能有详细的操作教学可以帮助学生更快地上手，并且希望老师能讲解汇编代码的一些关键部分，学生自己阅读汇编代码容易忽略一些问题。</p>
---

## 五、各人实验贡献与体会（每人各自撰写）

### 同学：程子洋

此次实验为本人独立完成大部分实验内容，并主要负责实验内容第 5 题(设计题)，思考题第 5 题，第 7 题实验报告的撰写。

实验体会：

这次实验我有很大的收获，总体上来说，本周的实验比较复杂，有很多重难点和需要理解的新知识，比如对于我们网安的学生来说，之前从来没有接触过 ELF 和 PE 文件，就需要花时间去学习；除此之外，本章的内容也很多，有很多值得慢慢分析和理解的地方，首先有很多新的概念要去熟悉；同时，阅读汇编代码、熟悉语句和段落的功能也是较为困难的任务。

通过本次实验，我深入地分析、理解并掌握了以下内容：汇编和 C 的互相调用方法、ELF 文件格式、使用 Loader 加载 ELF 文件、如何加载并扩展内核、学习使用 makefile、代码内存的布局、自己修改启动代码、把很多功能集成到了自己的内核文件中并引导……对其各种相关知识都有了一定程度的理解和自我的掌握；同时，对汇编代码的阅读和分析过程，也对我自己汇编的语法和代码编写的知识和能力带来的极大的锻炼和提升。

本次实验中最大的困难点可能在于最后一个任务：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。虽然需要实现的功能比较简单，可是由于我对汇编不是特别熟练、对本章知识不是完全理解等等原因，还有也是第一次把之前实现的功能和本章知识融合，集成到本章的代码中，编写和调试还是花了很长时间，中间出现了各种奇怪的错误，不过最终还是得到了正确的结果，感觉自己收获颇丰，极大地加深了我对内核和汇编编写的理解，之前学的知识也不仅仅只是停留在纸面上，而是通过真正的编程和实践把知识融会贯通。

### 同学：王卓

在本次实验中，本人独立完成大部分实验内容，并负责实验内容第 2、3 题，思考题第 2 题实验报告的撰写。

这次的实验总体来说，内容庞杂，涉及的新知识很多且较难理解。尤其对

于我网安专业的学生，更是需要花费更多的时间去了解相关基础知识。此外，阅读教材、汇编代码以及熟悉语句和段落的功能对我来说也较为困难。

通过本次实验，我深入地分析、理解并掌握了以下内容：保护模式的跳转(包括 GDT 的设置)、用 Loader 加载 ELF 文件等。我掌握了 C 与汇编的相互调用，这对于后续实现复杂功能是十分必要的。在 Loader 加载 ELF 的调试过程中，对 ELF 的格式有了更加全面的理解。另外，我掌握了 makefile 的使用，能够更加高效地管理文件与工程。感谢老师和助教准备的这次实验，希望自己能从实验中学会更多的知识，掌握更多的技能。

**同学：刘琥**

本次实验为本人独立完成大部分实验内容，并负责实验内容第 1 题，思考题第 1，6 题实验报告的撰写。

本次实验过程中，我接触学习了接触过 ELF 和 PE 文件，掌握了汇编和 C 的互相调用方法，学会了加载和拓展内核和使用 makefile，对于课程相关知识有了更深的感受，在前几次实验中得到是收获活用于本次实验，让我对于操作系统的各类操作更加得心应手，经过这几次的实验我能感受到自己动手能力的大幅提升，对操作系统的知识逐渐转化成动手实操的能力，我从中得到了极大的提升。

**同学：聂森**

本次实验为本人独立完成大部分实验内容，并负责实验内容第 4 题，思考题第 3-4 题实验报告的撰写。

通过本次实验，我深刻体会到了操作系统设计的复杂性和精妙之处。特别是在扩展内核和切换堆栈的过程中，我对操作系统如何管理硬件资源和执行环境有了更加深入的理解。文件结构的整理和 makefile 的使用让我认识到了在大型软件项目中代码组织和自动化构建的重要性。这不仅提高了我的开发效率，也让我对软件工程有了更深的理解。此外，添加中断处理是一个特别有趣的挑战，它不仅让我更加熟悉了中断的概念和处理流程，也让我有机会深入了解硬件和软件之间的交互。在这个过程中，我遇到了许多挑战，比如在切换堆栈和整理文件结构时遇到的各种错误和难题。这些问题让我不得不深入研究教材和参考资料，甚至需要寻求同学和教师的帮助。这个过程虽然艰难，但却极大地提高了我的问题解决能力和自学能力。

总的来说，这次实验不仅加深了我对操作系统核心概念的理解，也锻炼了我的编程技能和解决复杂问题的能力。希望能在实践中取得更大的进步。

**六、教师评语**

**教师评分（请填写好姓名、学号）**

姓名	学号	分数
聂森	2021302191536	

程子洋	2021301051114	
王卓	2021302191791	
刘虢	2021302121234	
教师签名：		
年月日		