|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 课程名称 | 操作系统设计与实践 | | | 成 绩 |  | 教师签名 |  |
| 实验名称 | 内核雏形 | | | 实验序号 | 6 | 实验日期 |  |
| 姓 名 | 张子航 | 学 号 | 2021302181026 | | | | 组长 |
| 姓 名 | 辜汝曦 | 学 号 | 2021302141194 | | | | 组员 |
| 姓 名 | 杨馨悦 | 学 号 | 2021302181212 | | | | 组员 |
| 姓 名 | 赵敏 | 学 号 | 2021302181215 | | | | 组员 |

**《操作系统设计与实践》实验报告**

# 实验目的及实验内容

（本次实验所涉及并要求掌握的知识；实验内容等）

## 实验目的

* 如何生成一个内核，能引导该内核

## 参考资料

* 《Orange’S 一个操作系统的实现》第5章

## 实验要求

* 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。
* ELF文件格式
  + 依照书上方法，分析你修改的这个可执行文件
* 依照书上方法，分析你修改的这个可执行文件
* 阅读书中给出的代码（/chapter5/f/），研究如何扩展内核
* 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（虚拟地址转换为物理地址、内存的分配和释放）、你自己设计的键盘中断代码集成到kernel文件目录管理中，并建立makefile文件，编译成内核，并引导

# 实验环境及实验步骤

（列出本次实验所使用的软件、工具；简要概括实验步骤）

## 实验环境

虚拟机工具：VMWare Workstation 16

虚拟机版本：Ubuntu 14.04.6(内存4GB，硬盘40GB，双核处理器)

开发与调试工具：bochs 2.6.8

# 实验过程分析

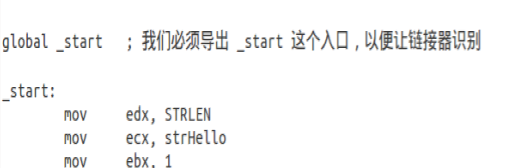
（详细记录实验过程，通过截图展示得到的结果。特别是对于实验中发生的故障和问题，要进行故障分析，说明故障排除的过程及方法。）

## 本次实验内容

### 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。

在汇编代码中，通过使用extern来引入C语言文件中定义的文件



例如：

extern choose：引入C文件中定义的choose函数

于此同时，汇编代码通过global导出函数出口

global \_start 导出函数入口，让链接器识别

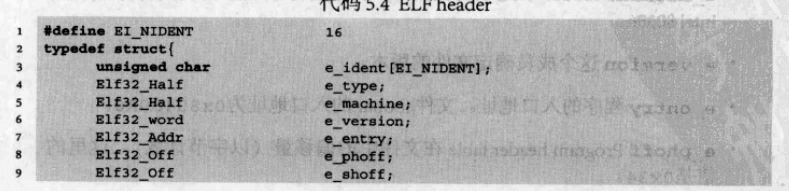
global myprint 导出函数入口，让C文件识别

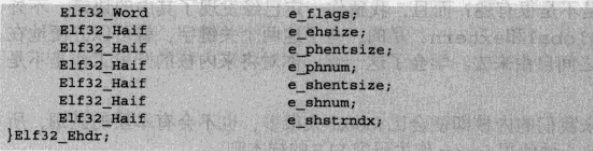


### ELF文件格式

ELF文件由头、程序头表、节、节头表组成，实际上只有头是固定的，剩余部分的位置、大小都由头的各项值来决定

查看刚才的foobar文件：xxd -u -a -g 1 -c 16 -l 80 foobar





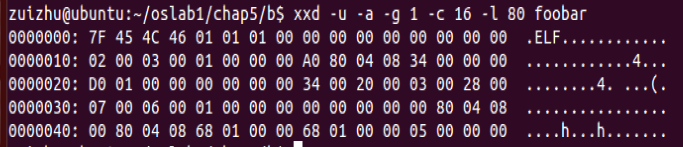


最开头的16字节是e\_ident(下图的第一行)，其中开头的四个字节是固定不变的，代表（.ELF）表明是一个elf文件

下面1行为e\_type，代表文件的类型，（因为其为elf32\_half，所以是两字节02 00小端读法，所以是00 02）2表示是一个可执行文件

然后是e\_machine，为00 03，表明运行该程序需要的体系结构为80386

E\_version为00 00 00 01，用于确定文件的版本



E\_entry为08 04 80 A0，是程序的入口地址

**E\_phoff为00 00 00 34，是program header table在文件中的偏移量（以字节计数）**

E\_shoff为00 00 01 D0，是section header table在文件中的偏移量（以字节计数）

E\_flags为00 00 00 00，对IA32而言，此项是0

E\_ehsise为00 34，为ElF header 的大小

E\_phentsize为00 20，即program header table中每一个条目（一个program header）的大小

E\_phnum为00 03，即program header table中有3个条目

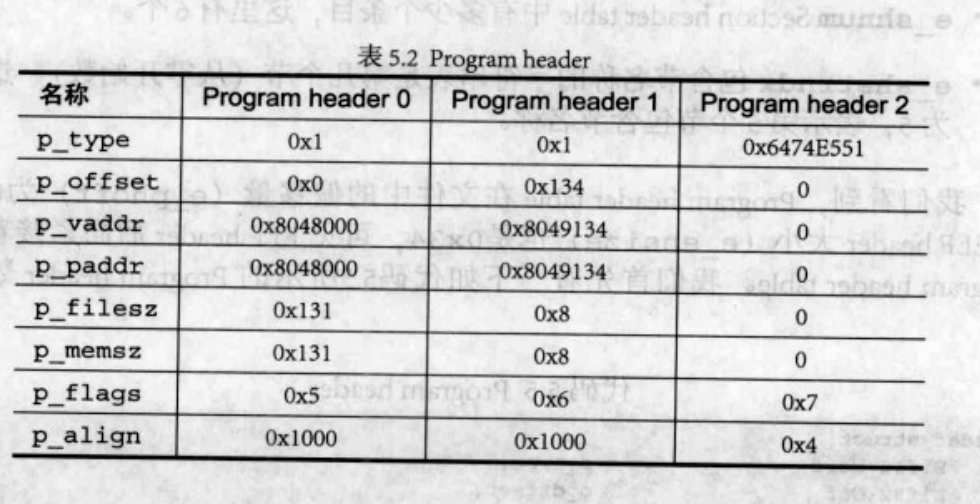
E\_shentsize为00 28，即section header table中每一个条目的大小

S\_shnum为00 07,即section header table中有多少条目

E\_shstrndx为00 06，包含节名称的字符串表是第6个节（从零开始数）

我们知道了program header table在文件中的偏移量为34，所以elf\_header后紧跟着program header table。

Program header（program header table中的一个条目）描述的是一个段在文件中的位置、大小以及它被放进内存后所在的位置和大小。我们前面已经知道E\_phentsize为00 20，即program header table中每一个条目（一个program header）的大小，且E\_phnum为00 03，即program header table中有3个条目，所以从0x34开始，偏移分别是34-53，54，73，74-93。



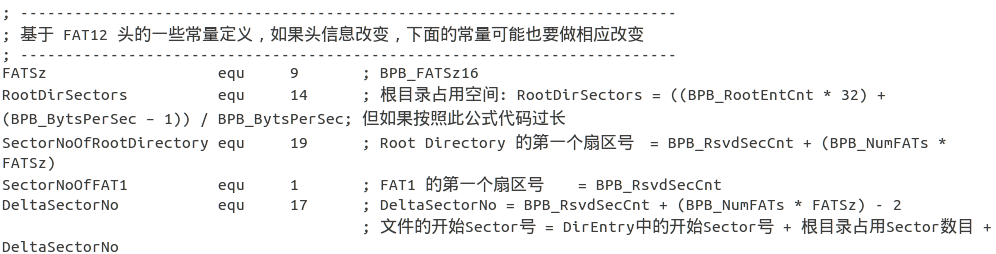
通过各项的参数我们就能知道一个段在文件中的位置、大小以及它被放进内存后所在的位置和大小。

### Program header（program header是program header table中的一个条目）描述的是一个段在文件中的位置、大小以及它被放进内存后所在的位置和大小。我们前面已经知道E\_phentsize为00 20，即program header table中每一个条目（一个program header）的大小，且E\_phnum为00 03，即program header table中有3个条目，所以从0x34开始，偏移分别是34-53，54，73，74-93。使用Loader加载ELF文件

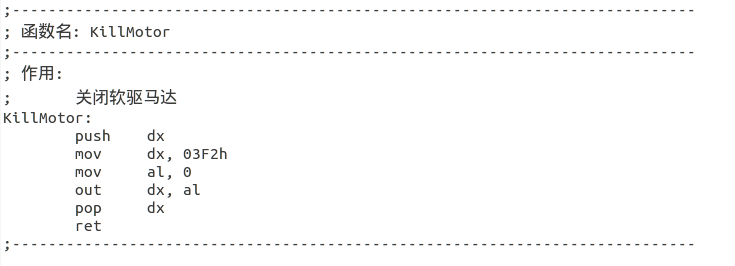
在本实验中，ELF文件就是需要加载的内核。该过程可以分为以下步骤：将内核加载进内存；进入保护模式；在保护模式下重新放置内核；将控制权转交给内核。下面我们对这些内容依次进行分析。

**1，将内核加载进内存**

该步骤与之前引导扇区加载loader等操作的步骤类似，包括寻找文件、加载进内存等。因此，我们可以复用一些这些操作都会用到的常量，并放在fat12hdr.inc中，之后，无论是boot.asm还是loader.asm都可以直接使用%include “fat12hdr.inc”语句来获取到所需要的常量。文件内容大致如下：



loader.asm的代码与前一个实验的对应部分大致相同，比如ReadSector、GetFATEntry等。但是loader.asm新增了一个KillMotor的函数，其用处在于关闭软驱马达。KillMotor的代码如下：



我们使用以下语句实现内核的装入：

**nasm -f elf -o kernel.o kernel.asm**

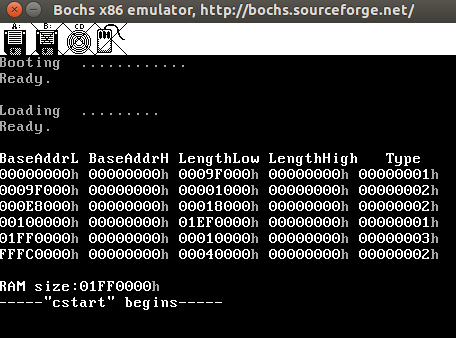
**ld -s -o kernel.bin kernel.o**

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

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

**sudo umount /mnt/floppy**

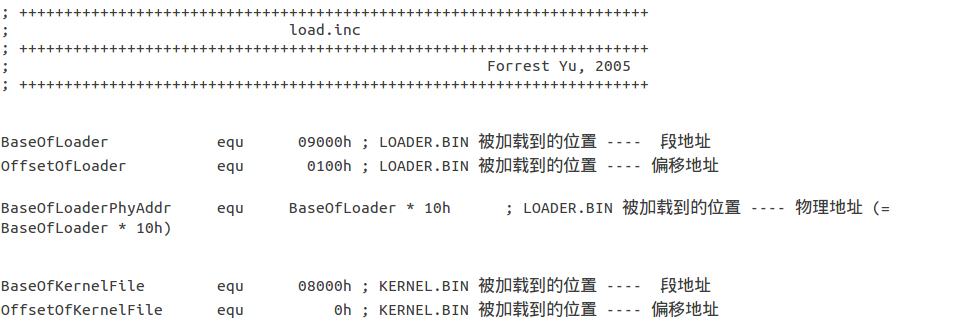
然后使用bochs打开，其效果如下：



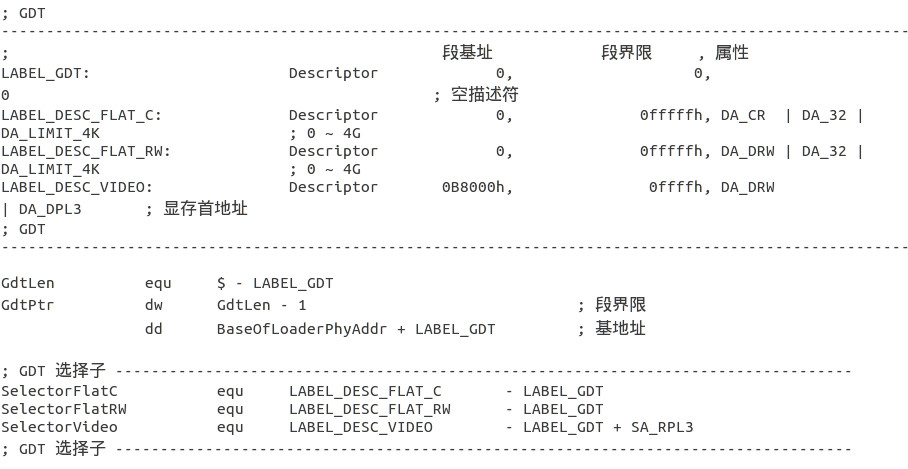
程序在最上面四行的输出表示程序成功将内核加载进内存中。

**2，进入保护模式**

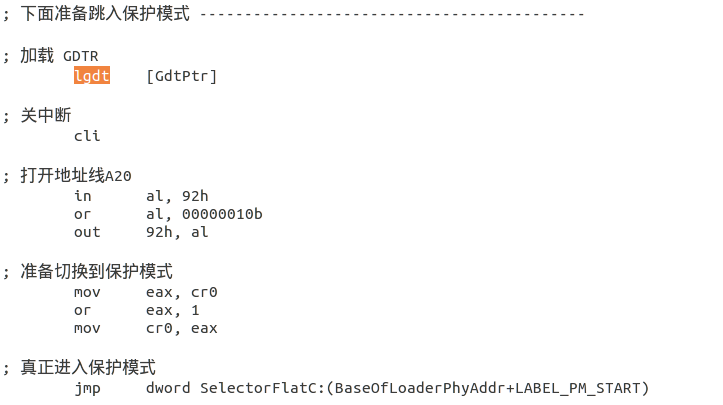
将内核加载进内存之后，我们需要在保护模式下修改内核的位置，也因此，我们需要进入保护模式。那么，我们首先初始化GDT、定义Selector。需要注意的是，我们知道loader在内存中的地址，并可以由此推导出描述符的段基址；因此，我们可以在load.inc文件中存放该常量：



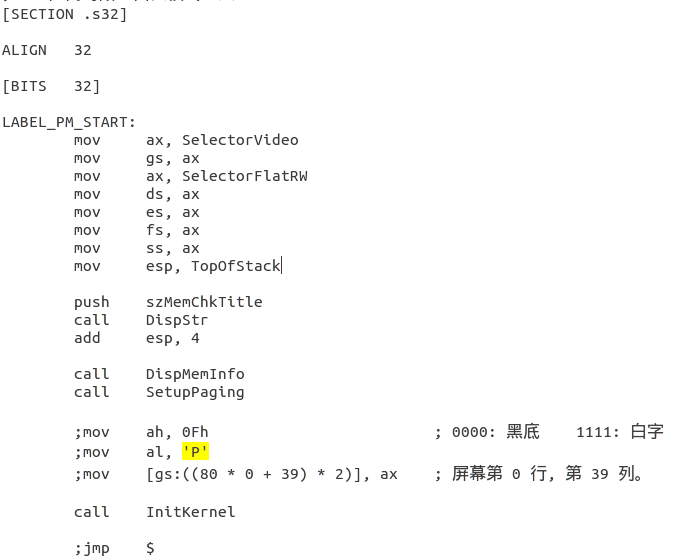
具体的初始化语句如下：



之后我们就可以进入保护模式了。该过程与第三章的流程基本一致：

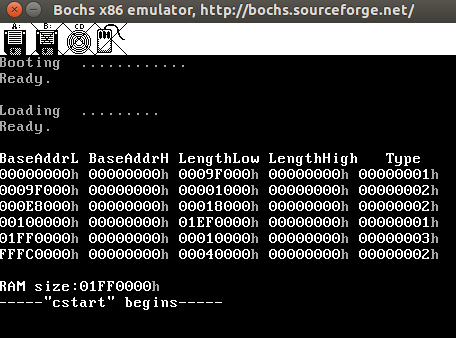


为测试是否成功进入保护模式，程序会在32位代码段中尝试输出一个P，如果终端成功输出了P，那就证明我们成功进入了保护模式。



同时，内核在保护模式下打印出内存信息，并启用分页机制。具体代码则为上图中的call DisMemInfo以及call SetupPaging。

具体内存信息请看下图：



**3，在保护模式下重新放置内核，并将控制权转交给内核**

首先，我们根据内核中的Program header table进行内存复制，复制次数由Program header的个数决定。每一个Program header都描述一个段，p\_filesz表示段在文件中的长度，p\_vaddr表示段在内存中的虚拟地址。内存复制的具体汇编代码如下：



ld命令生成可执行文件的p\_vaddr的值始终是一个类似于0x8048XXXX的值，该值超出128MB的内存大小。一共有两种解决方案，一是通过修改页表将p\_vaddr映射到更低的地址，二是修改ld的可选项降低p\_vaddr的值。第二种方案更加易行，其代码为：

**nasm -f elf -o kernel.o kernel.asm**

**ld -s -Ttext 0x30400 -o kernel.bin kernel.o**

经过上述操作，程序的入口地址就变为0x30400。下面是此时的ELF header以及Program header table的具体信息。

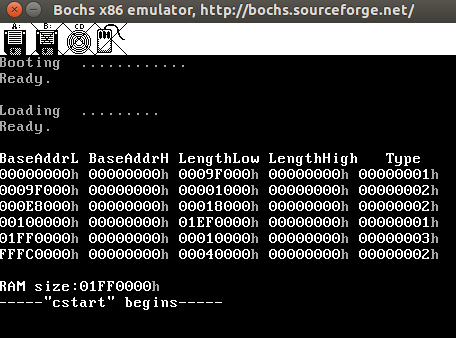


也就是说，文件开头40Dh字节的内容被放到内存30000h处，而由于程序的入口位于30400h，所以代码只有0Dh+1个字节。其中，0xEBFE对应着代码结尾的jmp $。最后内存情况为：

* 0x90000开始的63kb留给Loader.bin；
* 0x80000开始的64kb留给Kernel.bin；
* 0x30000开始的320kb留给整理后的内核；
* 页目录和页表被放置在了1MB以上的内存空间。

**4，将控制权转交给内核**

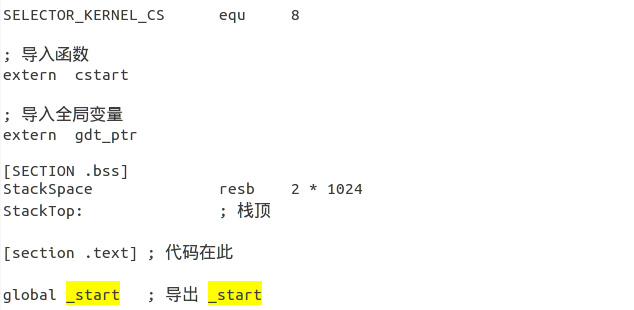
使用jmp SelectorFlatC:KernelEntryPointPhyAddr命令，就可以把控制权交给内核了。

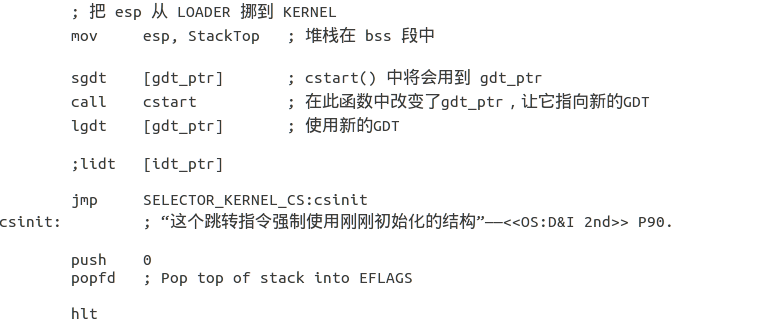


可以注意到，在最后，终端打印出“cstart” begins字样，这将在下一章节，也就是扩展内核中讲到。

### 阅读书中给出的代码（/chapter5/f/），研究如何扩展内核

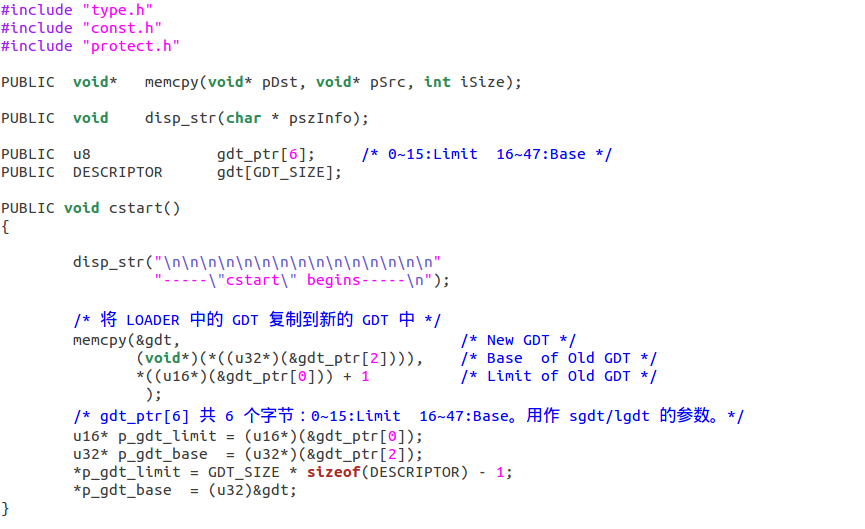
承接上文，当内核获取到控制权之后，esp、GDT等内容还在Loader中。因此，在对内核进行扩充时，需要将它们放到内核中。切换堆栈、更换GDT的具体的代码如下：





其中，StackSpace resb 2\*1024中的resb表示当前位置往后的2\*1024个字节全部为0；sgdt [gdt\_ptr]中的sgdt与之前学习的lgdt用法相反，也就是将GDTR储存到[gdt\_ptr]之中。

gdt\_ptr以及cstart分别表示一个全局变量与全局函数，定义在start.c中。



cstart()函数的具体作用是：使用memcpy()将位于Loader中的GDT全部复制到新的GDT中，并将gdt\_ptr中的内容更新为新GDT的base基地址以及limit界限。注意到，memcpy并不是使用C语言string.h中的库函数，而是string.asm中自己编写的汇编函数。个人猜测时因为内核编程不能使用一些非内核编程时使用的库函数，要么内核可以提供相似的替代函数，比如Linux内核编程时printk对应于printf，要么需要自己编写函数。cstart调用的disp\_str函数同理。

此外，cstart()函数还打印了“cstart” begins字样，证明该程序成功运行。

### 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（虚拟地址转换为物理地址、内存的分配和释放）、你自己设计的键盘中断代码集成到kernel文件目录管理中，并建立makefile文件，编译成内核，并引导（根据chapter5/i进行修改）

**（1）启动显示ASCII图案**

* 在kernel/start.c中编写函数，显示特定图案

**PUBLIC** void **printlogo**()

    {

**disp\_str**(""

    "       iiii     \n"

    "   |:H:a:p:p:y:| \n"

    " \_\_|\_\_\_\_\_\_\_\_\_\_\_|\_\_ \n"

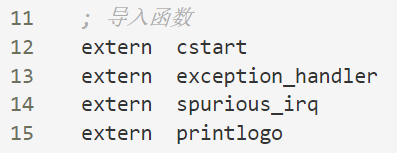
    "|^^^^^^^^^^^^^^^^^| \n"

    "|:B:i:r:t:h:d:a:y:| \n"

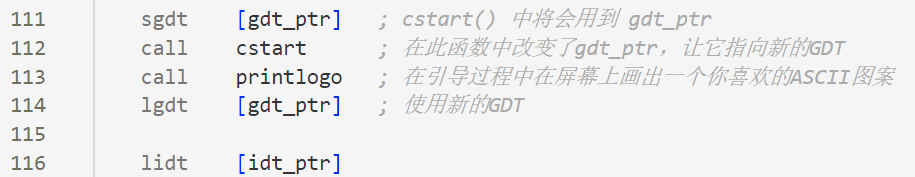
    "~~~~~~~~~~~~~~~~~~~\n");

}

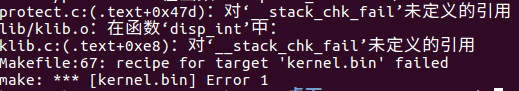
* 在kernel/kernel.asm中extern声明printlogo函数



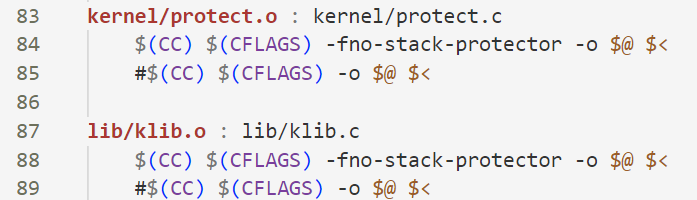
* 同时在kernel.asm中合适的位置调用该函数



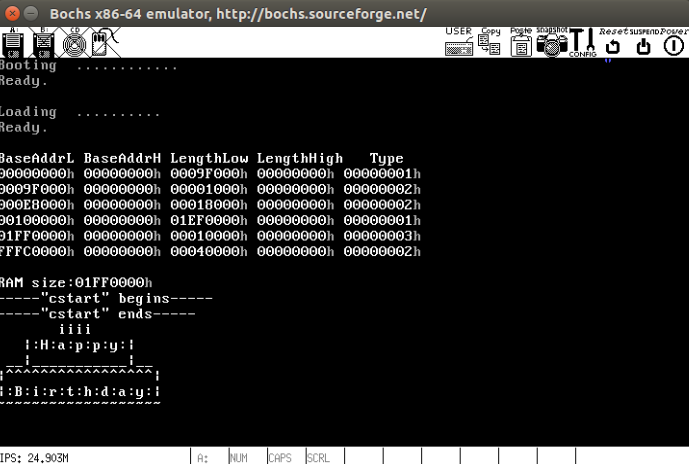
* 修改完程序后进行保存，依次使用make cleanmnake image和make命令。通过make buildimg或make image我们可以直接把引导扇区loader.bin和kernel.bin写入虚拟软盘



* 发现在 pritect.c和klib.c 中的 disp\_int 报错，修改Makefile文件，在 Makefile 中的 $(CFLAGS) 后面加上 -fno-stack-protector ，即不需要栈保护，之后编译链接就可以正常完成了



* 进入bochs，运行后可以看到成功显示图案



**（2）键盘中断函数集成**

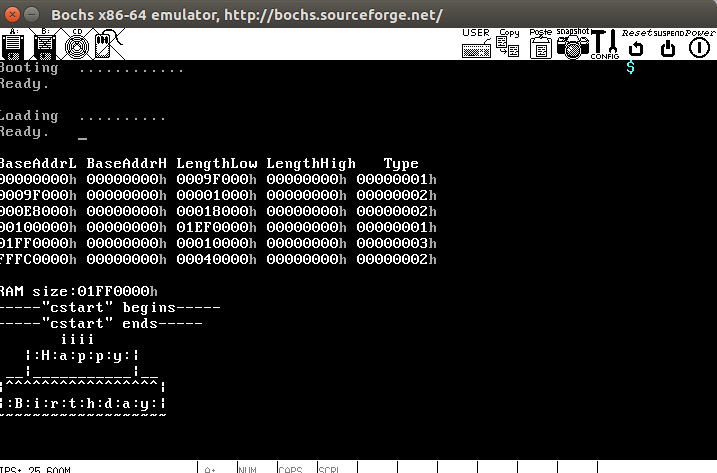
* 修改kernel.asm，使kernel进入死循环（否则hlt被中断唤醒后会继续执行hlt后面的指令，导致出现错误）

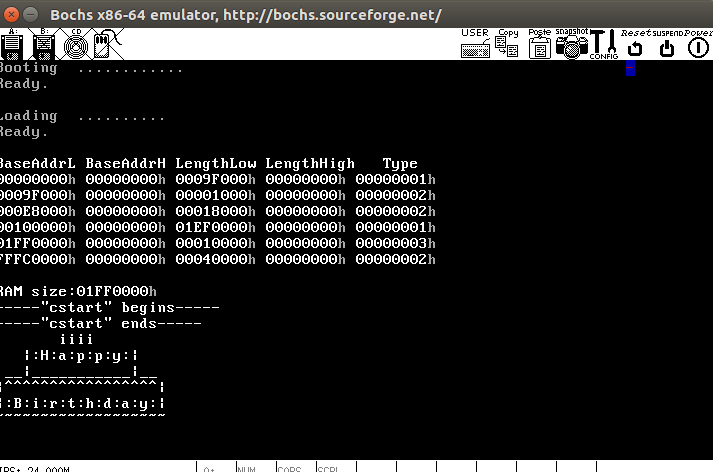
|  |  |
| --- | --- |
| 原始 | IMG_261 |
| 修改 | IMG_262 |

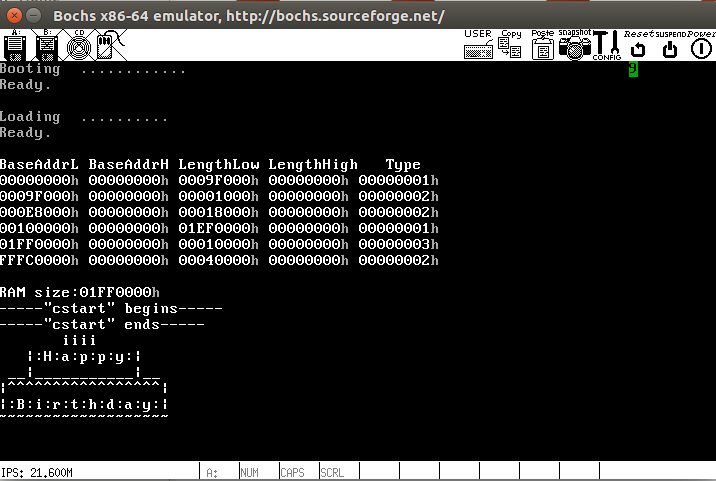
* 在kernel.asm中修改hwint01函数，改成自己的中断处理代码

|  |  |
| --- | --- |
| 原始 | IMG_263 |
| 修改 | IMG_264 |

* 测试程序，每次接收到键盘输入，修改字符的颜色





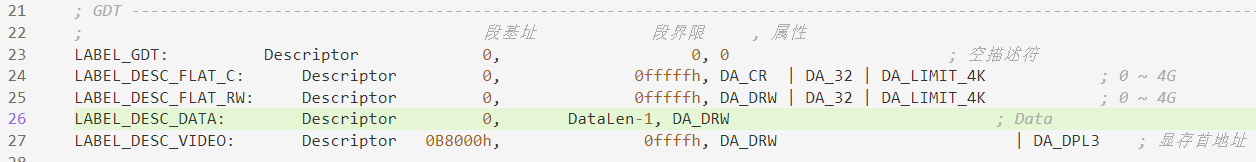


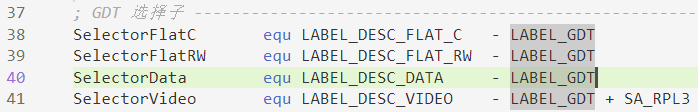
1. **地址映射集成**

* 在load.inc更改

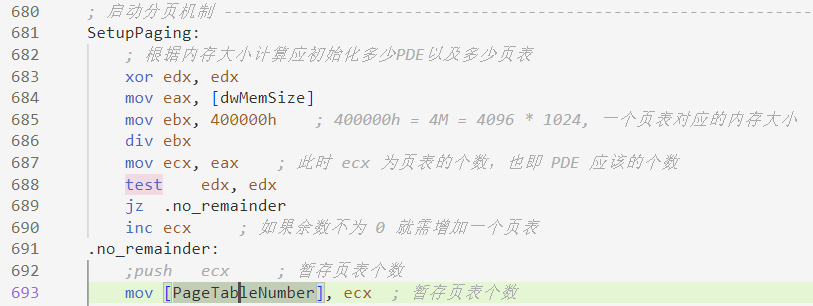


* 在loader.asm更改



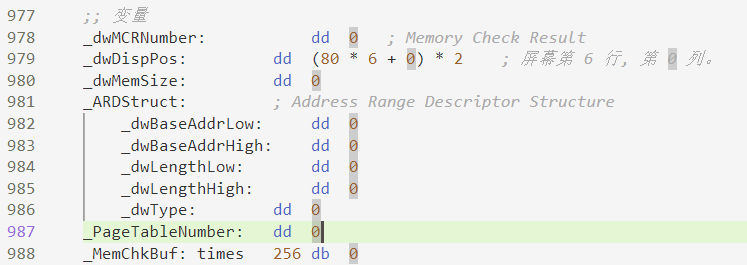


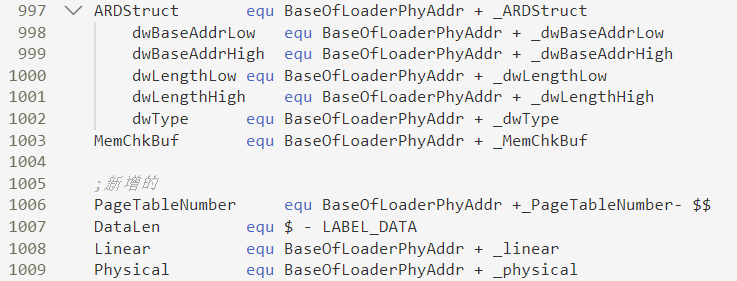












* 添加显示线性地址和物理地址的函数、完成线性地址映射到物理地址的函数

*; 显示线性地址和物理地址-----------------------------------------------------*

TestL2P:

        push ds

        push edx

*; mov  dx, SelectorData*

*; mov  ds,dx*

    push eax

    call DispReturn

    pop  eax

        push eax

        push Linear

        call DispStr

        add  esp,4

        pop  eax

        push eax

        call DispInt

        pop  eax

        call Linear2Physical

    push eax

    call DispReturn

    pop  eax

        push eax

        push Physical

        call DispStr

        add  esp,4

        pop  eax

        push ebx

        call DispInt

        pop  ebx

        pop edx

        pop ds

        ret

*; 线性地址转物理地址函数------------------------------------------------------*

Linear2Physical:

    push ds

    mov bx, SelectorFlatRW

    mov ds,bx

    mov ebx,cr3

    and ebx,0xfffff000

    mov ecx, eax

    shr eax,22

    and eax,0x3ff

    shl eax,2

    mov ebx,[eax+ebx]

    test ebx,0x00000001

    jz   .not\_exist

    and ebx, 0xfffff000

    mov eax, ecx

    shr eax,12

    and eax,0x3ff

    shl eax,2

    mov ebx,[eax+ebx]

    test ebx,0x00000001

    jz .not\_exist

    and ebx,0xfffff000

    and ecx,0xfff

    add ebx,ecx

    mov eax,ebx

    jmp .exit

.not\_exist:

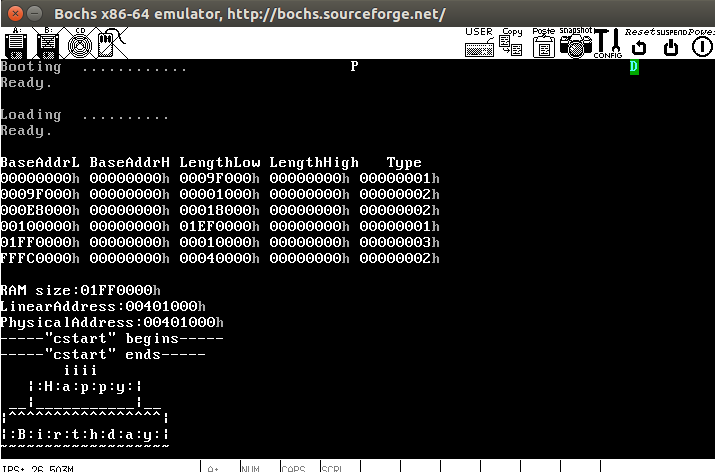
    mov ebx,0xffffffff

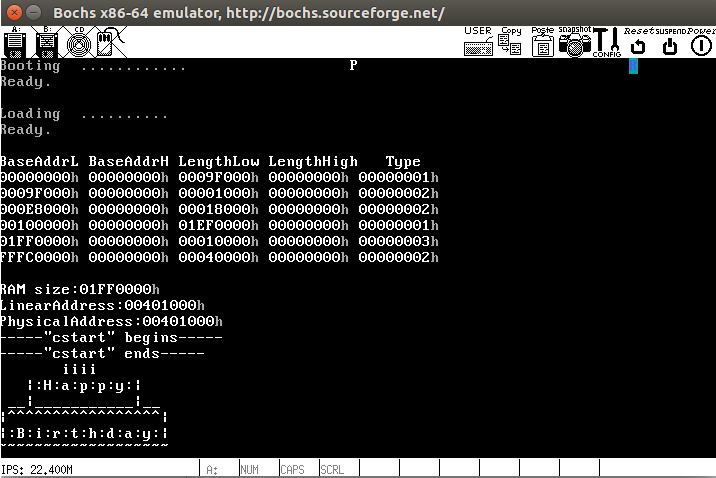
.exit:

    pop ds

    ret

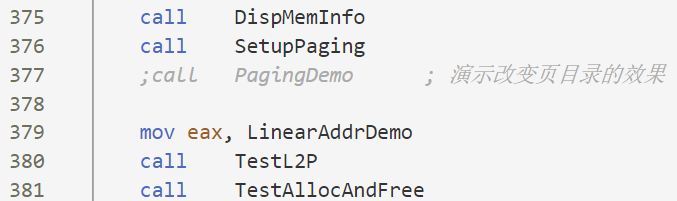
* 地址映射结果如下

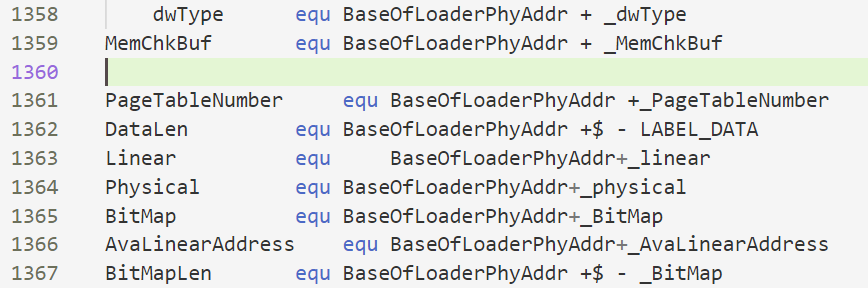
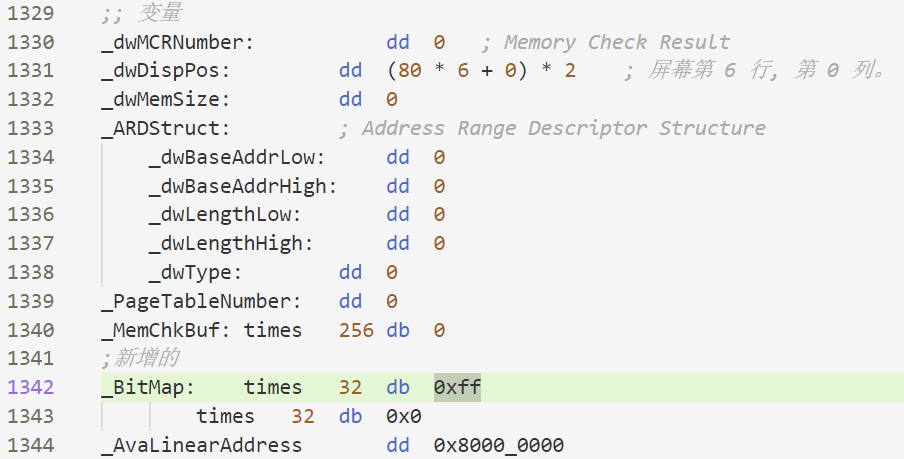




1. **内存管理函数集成**

* 修改loader.asm文件





* 添加函数

alloc\_a\_4k\_page:                        *; arg none*

*; return eax: physical address*

*; physical address begin at 0x00000000*

    push  ds

    push  es

    xor eax, eax

    mov ax, SelectorFlatRW

    mov es, ax

    mov ax, SelectorData

    mov ds, ax

.search:

    bts  [BitMap], eax

    jnc  .find

    inc  eax

    cmp  eax,BitMapLen\*8

    jl   .search

    hlt

.find:

    shl  eax,12

    pop  es

    pop  ds

    ret

* 内存分配函数

alloc\_pages:                               *; arg: eax : page number*

*; return ebx : linear address*

*; save registers*

    push ds

    push es

    mov bx, SelectorFlatRW

    mov ds, bx

    mov bx, SelectorData

    mov es, bx

    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 addresss 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*

    add ebx, eax

    or  ebx, 0x0000\_0007

    mov [edx], ebx

.pde\_exist:

    mov eax,[esp]

    and ebx, 0xfffff000

    and eax, 0x003ff000

    shr eax, 10

    add ebx, eax

.change\_pte:

    call alloc\_a\_4k\_page

    or eax, 0x00000007

    mov [ebx] , eax

    add ebx, 4

    loop  .change\_pte

    pop  ebx

    pop es

    pop ds

    ret

* 内存释放函数

free\_pages:              *; arg  eax,linear address , ebx  page number*

    push ds

    push es

    push ebx         *; save eax and ebx*

    push eax

    mov bx, SelectorFlatRW

    mov ds, bx

    mov bx, SelectorData

    mov es, bx       *; normal init*

*; find the pde and pte*

    mov ebx, cr3

    and ebx, 0xfffff000

    and eax, 0xffc00000

    shr eax, 20      *; 20 = 22 -2*

    add ebx, eax     *; ebx now means the pde item*

    mov edx, [ebx]

    and edx, 0xfffffff8

    mov [ebx], edx   *; set the final 3-bit zero and store back*

    mov ebx, [ebx]   *; now ebx means the first pte item*

    mov eax, [esp]          *; now eax is the liner address*

    add esp, 4

    and ebx,0xfffff000

    and eax,0x003ff000

    shr eax,10

    add ebx, eax       *;  now ebx means the right pte item*

    mov ecx, [esp]          *; here ecx means page number*

    add esp,4

.change\_pte:                 *; set every item's last 3-bit zero*

    mov eax, [ebx]

    and eax, 0xfffffff8

mov edx, eax                *; now eax is the physical address*

    shr edx, 12

    btr [BitMap], edx

    mov [ebx], eax

    add ebx,32

    loop .change\_pte

    pop es

    pop ds

    ret

* 自己编写的测试函数

TestAllocAndFree:

    xchg bx,bx

    mov eax,4

    call alloc\_pages

    xchg bx,bx

    mov eax,ebx

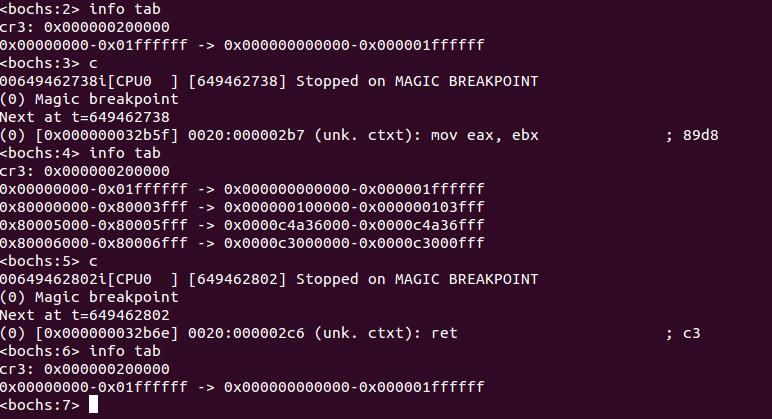
    mov ebx,4

    call free\_pages

    xchg bx,bx

    ret

* 编译成功，调试，实验结果如下



在调用AllocPages之前，存在一条地址映射；调用AllocPages之后，存在四条地址映射；调用FreePages之后，又只剩下了一条地址映射。由此可见，内存管理的alloc函数与free函数编写成功。

## 完成本次实验要回答的问题

### 汇编和C内定义的函数，相互间调用的方法是怎样的？

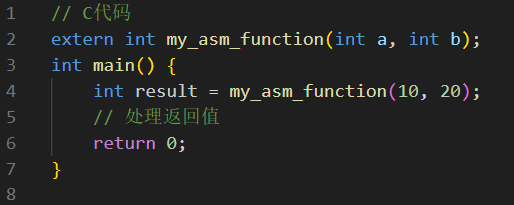
**从C调用汇编函数：**

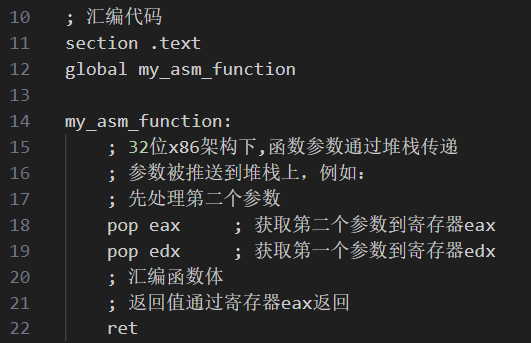
* 函数声明： 在C代码中，需要提供汇编函数的声明（使用extern），包括函数名和参数类型。
* 参数传递： C和汇编之间的参数传递可能涉及将参数放入正确的寄存器或堆栈中。
* 调用方式： 使用汇编指令（例如call）来调用汇编函数。
* 处理返回值： 汇编函数通过寄存器或堆栈返回值，C代码需要相应地获取返回值。

**从汇编调用C函数：**

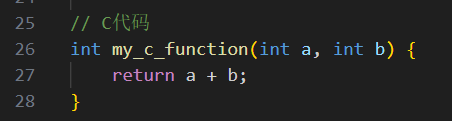
* 函数声明： 汇编代码中需要有C函数的声明（使用extern），以便正确调用该函数。
* 调用方式： 汇编代码需要按照C函数调用的规范将参数准备好，然后使用适当的指令（例如call）来调用C函数。
* 处理返回值： 汇编代码需要处理C函数返回的值，通常通过寄存器或堆栈来获取返回值。

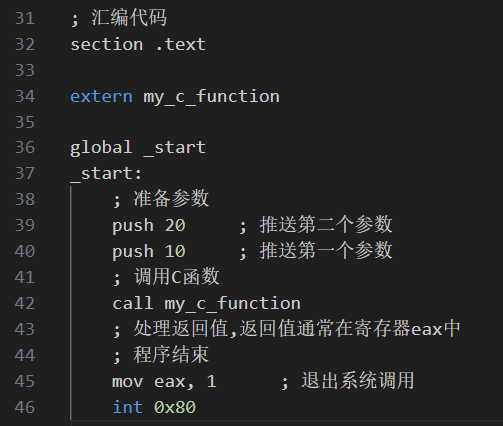
**从C调用汇编函数：**





**从汇编调用C函数：**





**global**：在汇编语言中，global是一个关键字或指令，用于声明全局符号或标签。它的作用是将符号（函数或变量）的作用域扩展到整个程序，使得其他文件或模块可以访问并使用这些全局符号。

在汇编语言中，如果想让一个符号（如函数或变量）在其他文件中可见和可访问，需要使用global指令来声明这个符号。这样，链接器在链接时才能够正确地识别这个符号，并确保不同模块之间能够正确地引用和使用这些全局符号。

举例来说，在汇编语言中，如果你有一个全局函数my\_function，可以使用global指令将其声明为全局符号，使得其他文件能够调用它。

**extern**：extern 是一个关键字，在C/C++语言中用于声明一个变量或函数，该变量或函数实际上是在其他文件中定义的。

在C/C++中，extern 的作用是告诉编译器某个变量或函数的定义并不在当前文件中，而是在其他文件中，并且该变量或函数是全局可见的。这样一来，编译器就知道这个符号是在其他地方定义的，而当前文件中只是在声明其存在。

举例来说，如果你在一个文件中需要使用另一个文件中定义的全局变量或函数，你可以在当前文件中使用 extern 来声明它，以便编译器知道它的存在。

### 描述ELF文件格式以及作用

ELF（Executable and Linkable Format）文件格式是一种通用的可执行文件和目标文件格式，广泛应用于Unix和Linux系统中。它定义了文件的结构和组织方式，包括文件头部、节表、段表等部分。ELF文件格式的作用是提供了一种统一的文件格式标准，使得不同操作系统和处理器架构之间的可执行文件和目标文件可以互相兼容和交互。这种通用性使得开发者可以在不同的平台上编写和运行程序，而无需担心文件格式的兼容性问题。

ELF文件格式还提供了丰富的信息，如符号表、重定位表等，方便链接器和调试器进行符号解析和代码重定位。这些信息使得程序的链接和调试工作更加高效和方便。此外，ELF文件格式还支持动态链接，允许程序在运行时加载共享库，实现了更加灵活的程序运行方式。

总之，ELF文件格式的设计使得程序的开发、编译、链接和调试变得更加简单和灵活，同时也提供了更好的可移植性和互操作性。

### 如何从Loader引导ELF的原理？

（1）Loader加载内核进内存：寻找文件、定位文件、读入内存，和引导扇区加载Loader进内存的方式一样。

Loader在内存中的物理地址是0x90000h，内核在内存中的物理地址是0x80000h

（2）跳转至保护模式

与之前一样，加载GDTR，关中断，打开地址线A20，cr0最后一位置为1，跳转至32位代码开头段

（3）在保护模式下根据ELF中Program header table中的值，找到每一个Program header，调用MemCpy函数，把内核中相应的段复制到正确的位置

（4）将控制权转交给内核：jmp跳转至内核的起始物理地址，进入内核

### 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？

在本节，对扩充功能的分析限于实验中已经提供的代码，对于自定义中断、物理地址与虚拟地址的转换、以及分页回收，请看设计题的部分。

目前，实验提供的内核扩展功能主要有：切换堆栈与GDT、添加分页、添加中断、构建目录树结构与Makefile文件。

* 构建目录树结构：

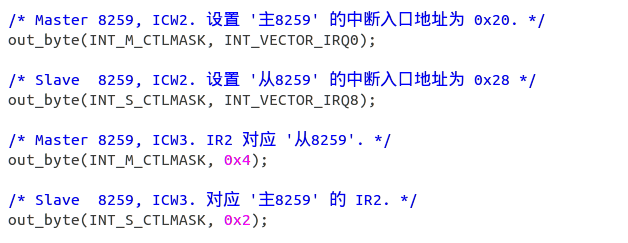
该功能属于内核扩展的基础，它重构了整个项目的架构，因此，需要最先进行说明。如果不构建目录树的话，源文件过多，项目管理较为混乱。

* 功能流程：boot.asm, loader.asm和需要的头文件放在目录/boot中；const.h, protect.h, type.h放在/include中作为头文件；kliba.asm和string.asm放在/lib中作为库；kernel.asm, start.c放在/kernel中。
* 源文件代码：无
* 与其他功能之间的关联：它改变了某些函数调用时的结构。
* 切换堆栈与GDT：

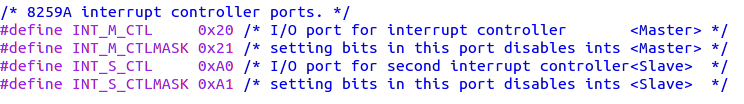
当内核获取到控制权之后，esp、GDT等内容还在Loader中。因此，在对内核进行扩充时，需要将它们放到内核中。

* 功能流程：在start.c中，取得新的GDT，将新GDT的界限与基地址写入原来的gdt\_ptr中，并在kernel.asm中进行更新，具体方式可以看本次实验内容的第4题。
* 源文件代码：/kernel/kernel.asm以及/kernel/start.c
* 与其他功能之间的关联：更换堆栈与GDT后，可以方便其他使用esp以及GDT函数的编写。
* 添加分页：
* 功能流程：使用call SetupPaging语句开启分页。
* 源文件代码：/boot/loader.asm
* 与其他功能之间的关联：分页是内存管理的基础，它与内存中运行的每一个函数都息息相关。
* 添加中断：
* 功能流程：设置8259A、建立IDT。

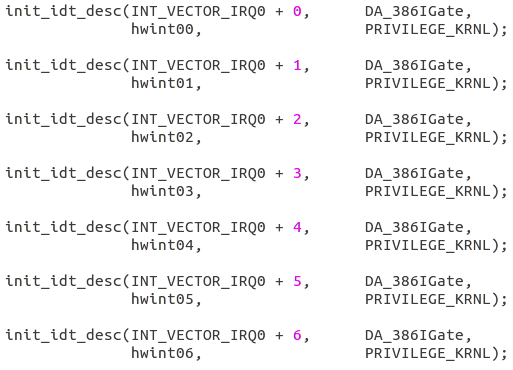
建立8259A的代码流程与之前学习中断时的基本一致，只不过是C语言的形式：



其中，INT\_M\_CTLMASK等常量的定义位于\include\const.h中：



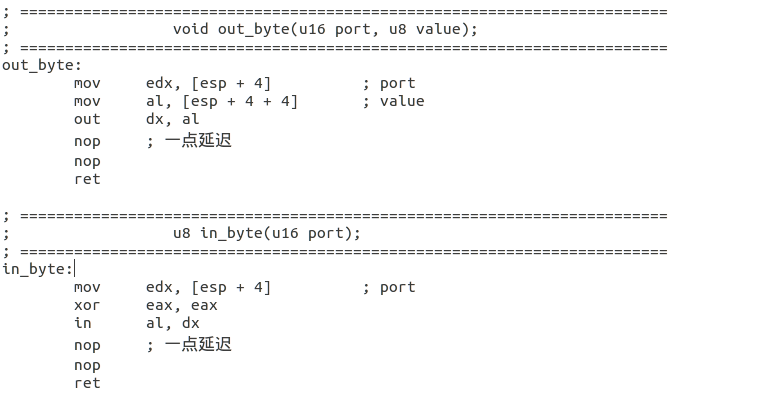
中断向量的定义位于\kernel\protect.c中：



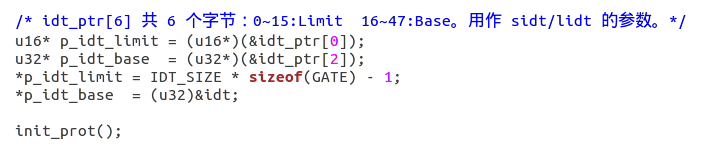
中断处理全局变量的声明在\include\global.h中。



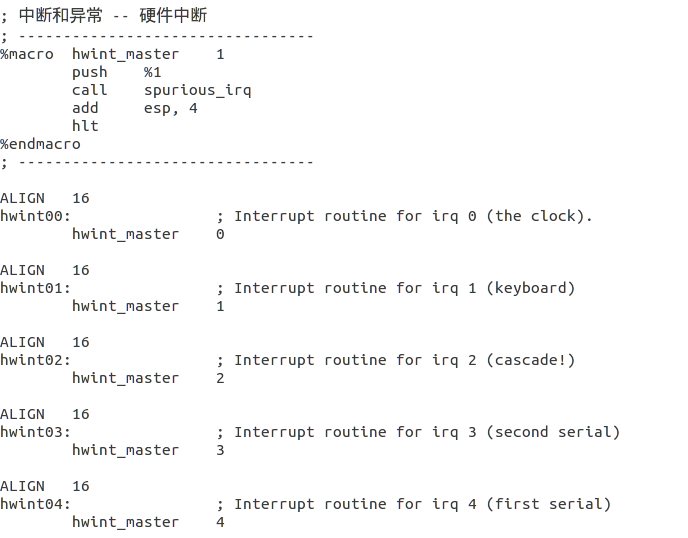
然后是读写端口功能的实现，位于\lib\kliba.asm中。



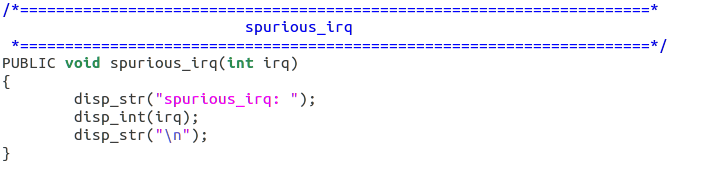
准备工作完成后，我们来到start.c中，初始化IDT：

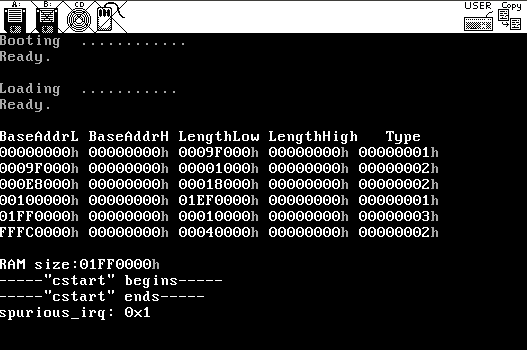


在kernel\kernel.asm中，我们导入idt\_ptr并加载IDT，然后，添加处理器可以处理的中断、异常以及相关处理程序：



正常情况下，中断会触发位于/kernel/i8259.c中的spurious\_irq函数，打印中断号：





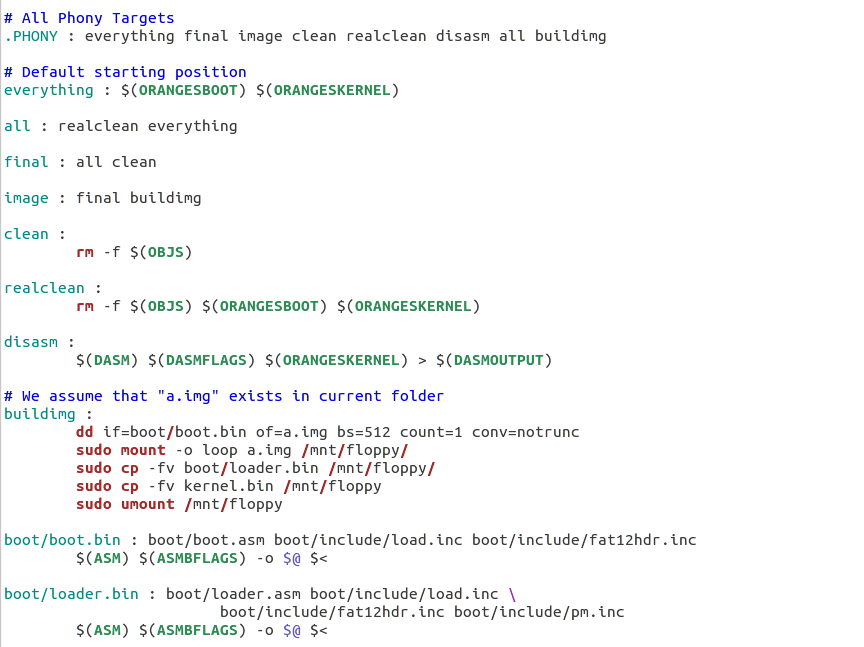
否则，异常处理函数位于kernel\protect.c文件中。其处理方式为清空屏幕的前5行，并将disp\_pos清零；之后，打印EFLAGS、CS、EIP等参数。



* 源文件代码：\kernel\i8259.c、\kernel\protect.c、\lib\kliba.asm、\kernel\kernel.asm以及\kernel\protect.c等
* 与其他功能之间的关联：该功能与目前添加的功能没有关联。不过，之后涉及到进程管理时，中断就显得尤为重要。
* 创建Makefile文件：

随着项目文件增多，每次进行编译链接耗时很大，因此，引入Makefile来加速项目的编译过程。

* 功能流程：在Makefile文件中按照语法写入项目编译全流程的命令，那么之后只需要在命令行输入make，项目就可以自动编译。下面是本项目的Makefile的一个示例：



* 源文件代码：./Makefile
* 与其他功能之间的关联：无

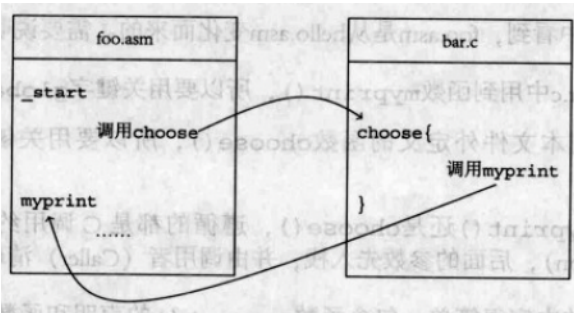
### 完成设计题并能演示。

# 实验结果总结

（对实验结果进行分析。并理论联系实际，思考并列出本实验对应的OS原理的知识点，并说明本实验中的实现部分如何对应和体现了原理中的基本概念和关键知识点。）

## 汇编代码和 C 语言代码的同步使用

Linux 内核主要是用 C 语言编写的，所以可以预见在以后进行内核开发的过程中会经常需要汇编代码和 C 语言代码之间的相互调用。因此在本实验中，我们通过一个简单的例子熟悉了 C 语言和汇编代码的链接和同步使用方式。在这个例子中包含汇编程序对 C 语言程序的调用 choose，也有 C 语言程序对汇编程序的调用 myprint()。整个程序结构如下图所示。



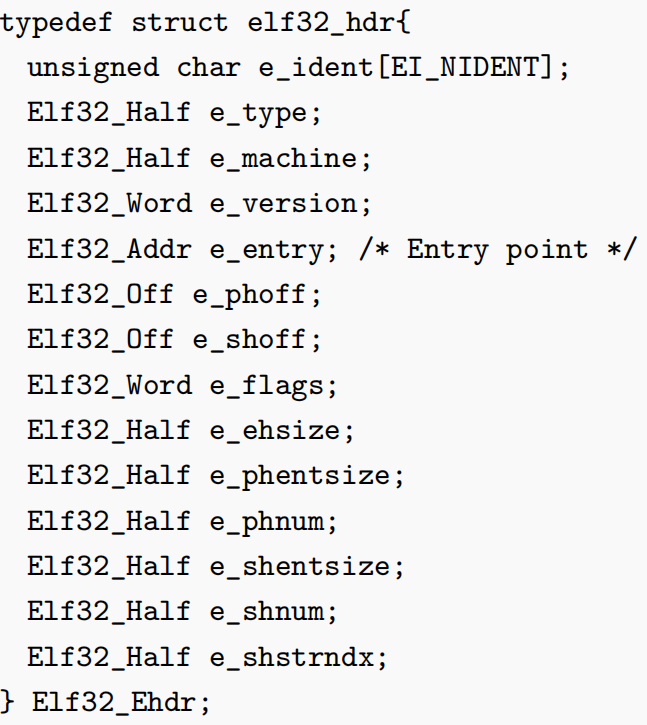
C 程序和汇编程序的同步原理并不复杂。因为 C 语言和汇编语言都会生成同样格式的中间文件——对象文件（.o 文件），里面包括函数符号表、参数表等信息，这些信息主要是提供给链接阶段使用，而链接阶段会将函数调用的符号变成相对地址，在这个过程中遵守一定的约定，就使得 C 语言和汇编语言相互调用成为可能。具体来说就是以下两点：

1. 在汇编语言中引用 C 语言的函数名时要用关键字 extern。在 C 语言中引用汇编语言的函数名时要用关键字 global。

2. 汇编语言调用和被调用时都要遵循 C 语言调用约定进行参数的传递，将各个参数倒序压栈。

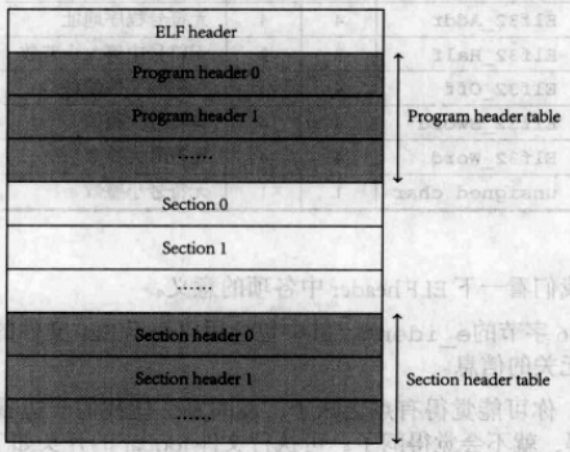
## **ELF 文件结构**

ELF 文件是一种重要的 Linux 可执行文件格式。在内核加载的过程中需要根据内核 ELF 文件的Program Header Table 的信息进行内存复制，所以对 ELF 文件格式进行基本了解是很有必要的。ELF文件分为 ELF Heaer、Program header table、Section 以及 Section header table 几个部分。如下图所示。



ELF Header 包含了一些与机器和程序相关的信息，定义在/include/elf.h 中，包含如下字段，其中比较重要的是 e\_entry，表示程序的入口地址。

Program Header 则描述的是一个段在文件中的位置、大小以及它被放进内存后的所在位置和大小，我们将 ELF 文件加载时，需要这些信息来将他的每个段都加载到内存中。



## 内核的加载机制

在之前的实验中我们已经初步了解了 BIOS——>Loader——>Kernel 的加载流程，本次实验中正式完成了从 Loader 到 Kernel 这一环。Loader 加载内核相关 ELF 文件的过程大致分为下面几个步骤。

1. 按照上一章 boot.asm 加载 loader.asm 的方法，根据文件名到 FAT12 的根目录区查找，将 Kernel.bin加载到 0x08000:0x0000h 处
2. 跳入到保护模式，完成分页。
3. 根据 ELF 文件格式，将 kernel.bin 的段都加载到 program header 中 p\_vaddr 字段指定的内存区域，接着跳转到 elf header 中的 e\_entry 指向的内存处执行。

## Makefile 的使用

makefile 的核心语法为如下：

Target : prerequisites

Command

target 表示目标文件，prerequisites 表示与目标文件的构建相关的依赖文件，command 是一条指令，在检测到存在任意一个依赖文件比目标文件新时执行的用于生成目标文件的命令。可以用 $( variable)表示变量，用等号 = 定义变量，另外还可以通过.PHONY 标号定义动作，动作表示了一组指令，通过make 命令后面加动作名，可以来执行这组动作。Makefile 可以大大减少每次编译时的命令长度，便于我们进行频繁的构建工作。

# 个人分工及心得体会

（每个人分别填写自己在本次实验中的分工，并总结实验的心得体会。）

## 张子航

完成本次实验内容第1题、第2题，本次实验要回答的问题的第3题，并撰写实验报告的对应部分，以及实验目的及实验内容部分。

本次实验，我学习到了汇编和C语言的互相调用的方法，同时将可执行文件生成为ELF格式，了解到了ELF文件的格式和结构。于此同时，本次实验也是对之前所有内容的综合。是对从保护模式、分页管理，到中断机制的知识点的综合考察。此外，我还学习到了如何从Loader引导ELF的步骤，复习了将loader加载入内存、跳转至保护模式的步骤，实现了对过去知识点的巩固，通过查看Elf中的特定参数来加载文件也是对新知识应用的考验。总的来说，本次实验收获很大

## 辜汝曦

完成本次实验内容第3题、第4题，本次实验要回答的问题的第4题，并撰写实验报告的对应部分，以及实验目的及实验内容部分。除此之外，也独立完成了本次实验内容的第1题，也就是实现C与汇编的相互调用。

本次实验算是一个小综合的实验，它结合了之前学习的分页机制、中断处理，并带领我们走向了内核编程的第一步。在本次实验中，我了解了内核编程时应该如何管理我们的代码；同时，在阅读其他的内核代码时，也知道从何入手。另一方面，我也从本次实验中学会了C语言与汇编语言相互调用的机制，也启发了我，当要编写某一个较为复杂的功能时，可以借助C语言的函数，避免编写过于冗长的汇编代码。

## 杨馨悦

完成本次实验内容第5题，本次实验要回答的问题的第2题，并撰写实验报告的对应部分及实验结果总结部分。

本次实验主要是构建一个内核雏形，融合了前几次的实验内容。从这一步开始，我们逐渐使用C语言作为编程语言，学习了C语言和汇编的相互调用，了解了ELF格式文件的相关信息，用Loader加载内核到内存，并把控制权转移给内核等过程。此外，在扩充内核的任务里，我们将此前编写好的虚拟地址转换为物理地址、内存的分配和释放代码，以及键盘中断代码集成到kernel文件目录管理中，并通过Makefile执行。在make出错时，通过查阅资料修改了Makefile文件，关闭了栈保护。通过详细阅读代码，在反复的修改和调试后，我也逐渐了解了内核的工作机制。

## 赵敏

完成本次实验内容第5题，本次实验要回答的问题的第1题，并撰写实验报告的对应部分。

本次实验综合了前面几次实验的内容，尤其是设计题部分是对前面实模式和保护模式、内存加载、中断异常、分页机制的一个综合，在完成本次实验的过程中也回顾了之前的知识，对整体的架构有了一个更宏观的了解，同时也新学到了如何生成一个内核，能引导该内核以及汇编和C的互相调用方法、Makefile文件的内容的意义，以及用树状的方式管理自己自己的文件夹。本次实验内容较为丰富复杂，在集成中断、分页以及地址转换等功能时也花费了不少的功夫，但是也收获了许多。