目录

[武汉大学国家网络安全学院教学实验报告 2](#_Toc181002565)

[1 实验目的及实验内容 2](#_Toc181002566)

[1.1 实验目标 2](#_Toc181002567)

[1.2 实验内容 2](#_Toc181002568)

[2 实验环境及实验步骤 2](#_Toc181002569)

[2.1实验环境 2](#_Toc181002570)

[vmware+ubuntu16.04-32位 2](#_Toc181002571)

[2.2实验步骤 2](#_Toc181002572)

[2.2.1 汇编和C内定义的函数，相互间调用的方法是怎样的？ 2](#_Toc181002575)

[2.2.2 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？ 2](#_Toc181002576)

[2.2.3 如何从Loader引导ELF的原理？ 3](#_Toc181002577)

[2.2.4 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？ 3](#_Toc181002578)

[2.2.5 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？ 3](#_Toc181002579)

[2.2.6 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。 3](#_Toc181002580)

[2.2.7 完成设计题并能演示。 3](#_Toc181002581)

[2.3实验内容 3](#_Toc181002582)

[2.3.1 汇编语言和c语言的互相调用方法 3](#_Toc181002584)

[2.3.2 描述ELF文件格式以及作用，与PE相比结构上有什么相同和差异，并分析foobar文件. 4](#_Toc181002585)

[2.3.3 从Loader引导ELF的原理 10](#_Toc181002586)

[2.3.4 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？ 12](#_Toc181002587)

[2.3.5 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？ 15](#_Toc181002588)

[2.3.6 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) 15](#_Toc181002589)

[2.3.7 完成设计题并能演示。 15](#_Toc181002590)

[3 各人实验贡献与体会（每人各自撰写） 18](#_Toc181002591)

[3.1 王亚鹏 18](#_Toc181002592)

[3.2 杨依磊 18](#_Toc181002593)

[3.3 杜泓波 19](#_Toc181002594)

[3.4 侯名扬 21](#_Toc181002595)

[4 教师评语 21](#_Toc181002596)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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 | | 2024.10.21 |
| 实验名称 | 内核雏形 | | 实验周次 | | 7 |
| 姓名 | 学号 | | 专业 | | 班级 |
| 王亚鹏 | 2022302181161 | | 信息安全 | | 5班 |
| 杨依磊 | 2022302181159 | | 信息安全 | | 5班 |
| 杜泓波 | 2022302181162 | | 信息安全 | | 5班 |
| 侯名扬 | 2022302181165 | | 信息安全 | | 5班 |
| 1. 实验目的及实验内容   （要求掌握的知识；实验内容；原理分析）   * 1. 实验目标   生成内核，并引导该内核。   * 1. 实验内容  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。   1. ELF文件格式   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。  （4）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。  （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。 | | | | | |
| 1. 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；具体实验步骤） | | | | | |
| 2.1实验环境  vmware+ubuntu16.04-32位  2.2实验步骤   * 2. 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。   2.3实验内容   * 1. 1. 汇编语言和c语言的互相调用方法   汇编语言中利用关键字extern引入外部变量、函数的声明，global：导出到全局关键域中；而c语言就可以引用汇编代码中导出的函数。    当然，注意到global调用了\_start这个入口让链接器识别，如代码osfs05/b/foo.asm部分代码所示，调用了bar.c中的choose函数也将myprint函数导出供bar.c引用。  实现了函数的互相调用，仍需要通过链接器生成可执行文件，打开osfs05/b/Makefile,查看具体的编译步骤。下图中的三行命令行解释为：   1. ld -s -o foobar foo.o bar.o 2. nasm -f elf -o foo.o foo.o foo.asm 3. gcc -c -o bar.o bar.c   为foo.asm添加swap函数实现交换参数：   1. swap: 2. mov eax,[esp+4] 3. mov ebx,[esp+8] 4. mov ecx,[eax] 5. mov edx,[ebx] 6. mov [eax],edx 7. mov [ebx],ecx 8. ret   为bar.c添加repeat函数实现两次调用choose函数：   1. int repeat(int a,int b) 2. { 3. swap(&a,&b); 4. choose(a,b); 5. swap(&a,&b); 6. choose(a,b); 7. return 0; 8. }   这样我们就可以得到三行输出，如下图所示。     * + 1. 描述ELF文件格式以及作用，与PE相比结构上有什么相同和差异，并分析foobar文件.        1. 通过foobar文件描述ELF文件格式及相应的作用。   ELF文件由ELF header，Program headertable,Sections(etc)，Sections headertable四部分组成  ELF header的格式如下：   1. #define EI\_NIDENT 16 2. typedef struct 3. { 4. unsigned char e\_ident [EI\_NIDENT]; 5. Elf32\_Half    e\_type;  //表示文件类型 6. Elf32\_Half    e\_machine;  //体系结构 7. Elf32\_word    e\_version;  //文件版本 8. Elf32\_Addr    e\_entry;  //程序入口地址 9. Elf32\_Off     e\_phoff;  //Program header table在文件中的偏移量（byte） 10. Elf32\_Off     e\_shoff;  //Section header table在文件中的偏移量（byte） 11. Elf32\_word    e\_flags;  //对IA32而言，这项为0 12. Elf32\_Haif    e\_ehsize; //ELF header大小 13. Elf32\_Haif    e\_phentsize;  //Program header table中每一条目的大小 14. Elf32\_Haif    e\_phnum;  //Program header table中有多少条目 15. Elf32\_Haif    e\_shentsize;  //Section header table中每一条目的大小 16. Elf32\_Haif    e\_shnum;  //Section header table中有多少条目 17. Elf32\_Haif    e\_shstrndx;  //包含节名称中的字符串表是第几节（from zero） 18. }Elf32\_Ehdr;   ELF定义上述格式是为了支持从8位到32位不同架构的处理器，让文件类型与机器无关  我们使用xxd检查foobar文件   1. xxd -u -a -g 1 -c 16 -l 80 foobar   开头四字节是固定不变的，7Fh后紧跟着的就是ELF三个字符，表明这是ELF文件。  0x10-0x11:0002h是e\_type，表明是可执行文件  0x12-0x13:0003h是e\_machine，表明需要运行的体系架构是Intel80386  0x14-0x17:00000001h是e\_version  0x18-0x1B:080480A0h是e\_entry，即foobar的入口地址。  0x1C-0x1F:00000034h是e\_phoff。  0x20-0x23:00000270h是e\_shoff。  0x24-0x27:00000000h是e\_flags，对于IA32而言此项是0。  0x28-0x29:0034h是e\_ehsize，ELF header大小为34B。  0x2A-0x2S:0020h是e\_phentsize，Program header table中每一条目Programheader的大小。  0x2C-0x2D:0003h是e\_phnum，Program header table中有三个条目。  0x2E-0x2F:0028h是e\_shentsize, Section header table中每一条目Sectionheader的大小。  0x30-0x31:0007h是e\_shnum，Section header table中有7个条目。  0x32-0x33:0006h是e\_shstrndx，表示第7个节包含节名称（from 0）  发现e\_phoff和e\_ehsize均为0034h，表明ELF header后紧跟着Program header  Program header数据结如下：   1. typedef struct 2. { 3. Elf32\_Word          p\_type; //当前所描述的段类型 4. Elf32\_Off           p\_offset; //段的第一个字节在文件中的偏移 5. Elf32\_Addr          p\_vaddr; //段的第一个字节的虚拟地址 6. Elf32\_Addr          p\_paddr; //为物理地址保留 7. Elf32\_Word          p\_filesz; //段在文件中的长度 8. Elf32\_Word          p\_memsz; //段在内存中的长度 9. Elf32\_Word          p\_flags; //与段相关的标志 10. Elf32\_Word          p\_align; //确定段在文件以及内存中如何对齐 11. }Elf32\_Phdr;   Program header描述的是系统准备程序运行所需的一个段segment或其他信息。  上述e\_phnum=0003h, e\_phentsize=0020h,则三个段范围分别是  0x34-0x53,0x54-0x73,0x74-0x93  使用xxd查询0x34开始的60h个字节：  1. xxd -u -a -g 1 -c 16 -s 0x34 -l 0x60 foobar  则foobar中三个Program header取值如下:  Program header 0:  0x34-0x37:00000001h,p\_type  0x38-0x3B:00000000h,p\_offset  0x3C-0x3F:08048000h,p\_vaddr  0x40-0x43:08048000h,p\_paddr  0x44-0x47:00000200h,p\_filesz  0x48-0x4B:00000200h,p\_memsz  0x4C-0x4F:00000005h,p\_flags  0x50-0x53:00001000h,p\_align  Program header 1:  0x54-0x57:00000001h,p\_type  0x58-0x5B:00000200h,p\_offset  0x5C-0x5F:08049200h,p\_vaddr  0x60-0x63:08049200h,p\_paddr  0x64-0x67:00000008h,p\_filesz  0x68-0x6B:00000008h,p\_memsz  0x6C-0x6F:00000006h,p\_flags  0x70-0x73:00001000h,p\_align  Program header 2:  0x34-0x37:6474E551h,p\_type  0x38-0x3B:00000000h,p\_offset  0x3C-0x3F:00000000h,p\_vaddr  0x40-0x43:00000000h,p\_paddr  0x44-0x47:00000000h,p\_filesz  0x48-0x4B:00000000h,p\_memsz  0x4C-0x4F:00000007h,p\_flags  0x50-0x53:00000010h,p\_align  根据上述信息可以知道foobar加载进内存之后的情形：    e\_shentsize=0028h，e\_shnum=0006h，表明有7个Section，且e\_shoff=00000270h，则Section 1的偏移为00000270h。  ELF Section Header Table是一个节头数组，位于节之后的文件末尾，每一个节头都描述了其所对应的的节的信息，编译器，链接器，装载器都是通过节头表来定位和访问各个节的属性。  使用readelf来查看节头表：   1. readlef -S foobar     上图给出了各节的节名，节类型，节的虚拟地址，在文件中的偏移，节大小，读写权限等。而Section Header Table中的各个ELF\_shdr的数据结构如下   1. typedef struct 2. { 3. Elf32\_Word    sh\_name; //节名是一个字符串，保存在.shstrtab的字符串表中，值是其节名在该表上的偏移值 4. Elf32\_Word    sh\_type; //节类型 5. Elf32\_Word    sh\_flags; //节标志位 6. Elf32\_Addr    sh\_addr; //节的虚拟地址 7. Elf32\_Off     sh\_offset; // 节在文件中的偏移 8. Elf32\_Word    sh\_size; //节大小 9. Elf32\_Word    sh\_link; //节链接信息 10. Elf32\_Word    sh\_info; //节链接信息 11. Elf32\_Word    sh\_addralign; //节地址对齐方式 12. Elf32\_Word    sh\_entsize; //节项大小 13. } Elf32\_Shdr;   具体的值的含义引自[ELF 格式详解\_elf文件格式-CSDN博客](https://blog.csdn.net/kunkliu/article/details/129648744?ops_request_misc=%257B%2522request%255Fid%2522%253A%25220C2A9D01-2EC6-4066-A6A4-0AE2709A762F%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=0C2A9D01-2EC6-4066-A6A4-0AE2709A762F&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-1-129648744-null-null.142%5ev100%5epc_search_result_base1&utm_term=ELF%E6%96%87%E4%BB%B6%E6%A0%BC%E5%BC%8F&spm=1018.2226.3001.4187)。这里不过多展开赘述。  对于foobar文件，第一个节是空节。  第二个节.text是保存了程序代码指令的代码节，存在于text段中。    第三个节.rodata保存了只读数据，存在于text段中。    第四个节.eh\_frame用于跨函数调用的异常处理和栈展开，由编译器自动生成，包含用于异常处理和栈展开的指令和数据。    第五个节.data保存了初始化的全局变量等数据，存在于data段中。可以看到.data节中存放00000003,00000004两个全局变量。    第六个节.comment存放编译器版本信息。    第七个节.shstrtab是节区头部表名字的字符串表。     * + - 1. ELF文件与PE文件结构上的异同。   一、相同点  1、文件头结构：ELF Header 和 PE Header 都位于文件的起始位置，包含文件类型、目标架构、入口点地址等基本信息。它们都起到了指示程序如何加载和执行的作用。  2、段和节的概念：ELF 文件分为段（Segment）和节（Section）。段用于内存映射和加载，而节用于逻辑组织（如代码、数据等）。PE 文件也具有\*\*段（Section）\*\*的概念，PE 文件中每个段既可以用于内存映射，也用于逻辑组织。PE 文件没有显式的段与节的分离概念。  3、程序代码和数据的分离：两种格式都包含分离的代码段、数据段、只读数据段等，分别存放可执行代码、可读写的数据以及只读的常量。  4、重定位表和导入表：两者都使用重定位表（Relocation Table）和导入表（Import Table）。重定位表用于在加载时调整符号地址，导入表则列出程序所需的外部函数或库。  5、符号表：两者都可以包含符号表（Symbol Table），用于存储变量和函数的名称及其地址，通常用于调试和链接。  二、不同点  1、文件头差异：ELF Header 具有固定的大小（52字节或64字节，依赖于32位还是64位系统）。其中包含版本信息、目标操作系统、目标架构等。PE Header 比较复杂，除了主头部（DOS Header）以外，还包含多个表头（COFF Header, Optional Header）。PE 文件的起始是一个 DOS 头（用于显示“该程序不能在 DOS 模式下运行”消息），然后偏移一段距离才是实际的 PE 头。  2、段与节的管理：ELF 文件严格区分段（Program Headers）和节（Section Headers），段用于程序执行时的内存映射，节用于符号管理和调试信息。段是系统加载器关心的部分，节是链接器和调试器关心的部分。PE 文件不区分段与节，它们都被称为“Section”，既用于内存映射，也用于存储调试信息。PE 文件的段被直接用于程序加载。  3、文件布局：ELF 文件通常在文件头后紧接着是程序头表（Program Header Table），用于描述段的内存映射信息；然后是各个段和节。PE 文件的文件头（PE Header）在文件的开头位置之后，紧跟着的是各个区段（Section）和相关的表格（如导入表、导出表、重定位表等）。  4、符号解析和动态链接机制：ELF 文件的符号解析和动态链接使用动态段（Dynamic Segment），包括动态符号表（.dynsym）和动态重定位表（.rel.dyn）等。PE 文件使用的是\*\*导入表（Import Table）和导出表（Export Table）\*\*来处理动态链接。  5、段的名称和作用：ELF 文件的段有固定的名称和作用（如 .text 段存放代码，.data 段存放可读写数据，.rodata 段存放只读数据）。PE 文件的段名称和作用相对灵活，常见的段有 .text（代码），.data（初始化数据），.rsrc（资源数据），但也可以自定义段名称。   * + 1. 从Loader引导ELF的原理   Loader分别需要加载内核到内存，跳入保护模式。   * + - 1. 加载内核到内存   类似于引导扇区，处理内核时要根据Program header table中的值把内核中的段放到正确的位置。 Loader经过寻找文件，定位文件，读入内存三步将内核读入内存。与实验5一样，使用boot.asm,loader.asm调用（不写成一个函数是为了节省引导扇区的空间），为此将一些常量的定义写进fat12hdr.inc。     * + - 1. 跳入保护模式   GDT以及对应的选择子只定义了三个描述符，分别是一个4GB的可执行段，一个4GB的可读写段，一个指向显存开始地址的段。Loader的段地址不是BIOS或DOS而是编译加载的，因此可确定为BaseOfLoader。则标号变量的物理地址可以确定为：   1. PhysicalAddr=BaseOfLoader\*10h+OffsetOfLoader   同**2.3.3.1**，也将上述声明放在宏定义loader.inc中。  在此基础上，osfs05/d/loader.asm还增加了获取并打印内存信息模块（**实验2**中pmtest7中的模块）和启动分页模块，运行结果如下：     * + - 1. 根据Program header table中的值把内核中的段放到正确的位置。   原理类似于以下C语言代码，有多少个Program header就执行多少次。   1. memcpy（p\_vaddr,BaseOfLoaderPhyAddr+p\_offset,p\_filesz）;   要注意的是由ld生成的可执行文件中p\_vaddr总是一个类似于0x8048XXX的值，这对于启动分页机制对等映射来说已经超出128MB，并且也不能决定内核的装载位置，我们通过修改ld的选项使其生成的可执行代码中p\_vaddr变小，这样将程序的入口地址修改为了0x30400。   1. nasm -f elf -o kernel.o kernel.asm 2. ld -s -Ttext 0x30400 -o kernel.bin kernel.o   我们使用xxd来查看kernel.bin的ELF header和rogram header：      与**2.3.2**的做法相同  18h-1Bh：e\_entry=00030400h，程序入口为30400h。  3Ch-3Fh：p\_vaddr=00030000h，段的第一个字节的虚拟内存地址。  44h-47h：p\_filesz=0000040Dh，我们应该把文件从开头开始的40Dh个字节放到内存30000h处。很容易就能得出代码只有0Dh+1个字节。\*代表省略的0,400h-40Dh是代码，0xFBFE正好是代码最后的jmp $。  在osfs05/e/loader.asm实现将kernel.bin根据ELF转移到正确的位置。  至于将程序入口放在30400h是因为内存的分布情况决定。9FC00h-9FFFFh不被用作常规使用，用户可以使用的内存是0500h-09FBFFh。并且大多数DOS不会占用30000h以上的内存，可以避免覆盖掉DOS内存。   * + - 1. 向内核交出控制权   osfs05/e/loader.asm中的KernelEntryPointPhyAddr即程序入口点，在load.inc中定义，与ld的参数-Ttext一致，事实上我们只需要修改这两个值就可以改变内核装载的位置。运行结果如下：     * + 1. 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？   使用tree查看文件布局：    **2.3.4.1内存块的复制-string.asm**  对memcpy（）进行编译  **2.3.4.2切换堆栈和GDT-kernel.asm**  切换GDT用到的cstart函数在osfs05/f/start.c中，通过cstart()改变全局变量gdt\_ptr使其指向新的GDT——cstart()首先把位于loader的原GDT全部复制给新的GDT然后把gdt\_ptr的内容换成新的GDT的基地址和界限。  **2.3.4.3打印字符串-kliba.asm，显示整数-klib.c**  **2.3.4.4添加中断处理**  通过i8259.c设置8259A，通过protect.c实现中断异常处理。我们对中断异常的主题思想是：如果有错误码，把向量号压栈，然后执行exception\_headler()，若没有错误码，就压栈0XFFFFFFFF,在把向量号压栈并随后执行 exception\_headler():   1. divide\_error: 2. push  0XFFFFFFFF 3. push  0 4. jmp   exception 5. ... 6. exception: 7. call  exception\_headler 8. add   esp 4\*2 9. hlt   而设置IDT则是通过调用init\_idt\_desc，将其初始化为门描述符，在init\_prot()中,所有的描述符都被初始化为中断门。  现在，我们分别使用ud2产生#UD异常，使用jmp 0X40:0产生错误码    最后，所有的中断都会触发kernal.asm中的spurious\_irq()，实现将IRQ号打印出来。当前IRQ号为1，对应的正是键盘中断。    **2.3.4.5功能流程**  在boot文件夹中的boot.asm，loader.asm实现了引导ELF加载到内存并进入保护模式，打印内存信息；fat12hdr.inc是对fat12的宏定义；pm.inc是对描述符，选择子和门的宏定义；load.inc是引导ELF加载到内存时的一些baseaddr，offset等地址的宏定义。  include文件夹中，const.h是对PUBLIC，PRIVATE，描述符个数，权限，和中断端口的宏定义，protect.h是对存储段描述符，门描述符和中断向量的宏定义，type.h是对数据类型的宏定义，proto.h中是in\_byte,out\_byte（对端口进行读写操作），disp\_str的函数声明，global.h中是全局变量gdt[],gdt\_ptr[],idt[]和idt\_ptr[]。  kernel文件夹中的在**2.3.4.2**中介绍，lib文件夹中的在**2.3.4.1**和**2.3.4.3**中介绍   * + 1. 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）。  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。   * + 1. 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)     2. 完成设计题并能演示。   **2.3.7.1启动显示的ASCII图案**  设计自定义的ASCII图案，在start.c中编写函数实现功能，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。  **2.3.7.2中断函数集成**  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）   1. csinit： 2. sti 3. deadloop：S 4. hlt 5. 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。   1. hwint01： 2. inc      byte 3. in        al,0X60 4. mov    al,20h 5. out      20h,al 6. iretd   测试程序，每次按键盘，发现颜色变化，中断程序集成成功。  **2.3.7.3内存集成管理**  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。  在kernel/kernel.asm中添加测试函数并调用，被测试函数使用extern声明。   1. TestAlloAndFree: 2. xchg       bx,bx 3. mov       eax,4 4. call         alloc\_pages 5. xchg       bx,bx 6. mov        eax,ebx 7. mov        ebx,4 8. call          free\_pages 9. xchg        bx,bx 10. 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。   1. mm/pageManage.o : mm/pageManage.asm $(ASM) $(ASMKFLAGS) -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：  查看alloc\_pages前地址映射关系：  查看alloc\_pages前地址映射关系： | | | | | |
| 1. 各人实验贡献与体会（每人各自撰写） | | | | | |
| 实际上每个人都各自独立地完成了实验。   * 1. 王亚鹏   1. 实验收获和体会  不仅学习到了汇编语言与 C 语言的相互调用方法，还深入了解了 ELF 文件格式的结构与作用。这次实验让我更全面地认识了操作系统的加载过程，尤其是 Loader 加载 ELF 文件的原理。通过分析 ELF 与 PE 文件的结构异同，我对文件格式与操作系统的关系有了更深的理解  2. 实验中的挑战与问题解决  在实现函数的调用和内存管理时，遇到了链接器配置的错误。通过查阅相关资料和调试代码，最终通过修改 Makefile 文件中的配置解决了这些问题  3. 深入理解的关键知识点  汇编与 C 语言的互相调用：实验中通过 extern 和 global 关键字来实现两个语言之间的互相调用，这一过程让我明白了一年前我的一个疑惑，那就是为什么一个C语言项目有多个文件，最后只会生成一个程序，并且函数还能够在不同的文件中定义，最后相互借用  ELF 文件格式：软件安全课上已经详细学习过，在这里不加赘述   * 1. 杨依磊   通过构建并扩展一个简易的操作系统内核，我理解了汇编语言与C语言之间的互相调用、ELF文件格式的结构和作用、ELF加载到内存的流程，以及内核中断机制的实现。实验内容充实，涉及操作系统底层知识和实际编程实现，增强了对操作系统内核启动和管理流程的理解。以下是一些主要的收获：汇编与C语言的调用：通过编写简单的汇编函数并在C代码中调用，掌握了使用extern和global等关键字实现跨语言调用的方法 ELF文件格式：分析了ELF文件的头部结构、段和节的划分、加载原理等。通过与PE文件的对比，理解了ELF在Linux系统上的内存映射特点，也学习了如何使用工具解析并查看ELF文件的结构。内核加载和中断管理：使用Loader将内核加载到内存后，切换到保护模式并启动中断处理机制。完成了键盘中断的集成，实现了简单的键盘按键响应功能。这一过程帮助我理解了处理器特权级别的切换、GDT 的配置和IDT的初始化方法。Makefile的使用：在编写内核代码时，需要合理组织和管理源文件。通过Makefile定义编译流程，不仅节省了编译时间，还确保了不同模块之间的依赖关系。设计任务：在内核启动阶段增加ASCII图案显示，并完成了内存管理和中断处理的集成，增强了内核功能。这些设计任务提升了对操作系统设计灵活性的理解，并体会到如何在现有代码基础上进行扩展。实验中遇到的问题函数声明冲突：在2.3.5实验步骤中，遇到了disp\_int未声明的警告。通过在相应文件前添加该函数的声明解决了这一问题，但后续发现disp\_str和disp\_color\_str在使用EBX寄存器时未正常保护和恢复，导致调用错误。最终在Makefile的CFLAGS中添加-fno-stack-protector选项，才成功避免了栈保护相关的问题。ASCII图案显示位置：在设计启动时显示自定义ASCII图案的功能时，最初未能找到合适的调用位置。经过多次尝试，最终确定在新GDT设置完成后调用该函数，成功实现了预期效果。   * 1. 杜泓波   本次实验主要涉及汇编与C语言的混合编程、ELF文件格式的分析、Loader加载ELF文件、内核的扩展与结构分析、以及启动代码的修改和内存管理、中断集成等多个方面。通过这次实验，我对底层系统编程有了更加深入的理解，同时也加深了对操作系统内核启动流程、内存管理、中断处理等核心概念的掌握。  1. 汇编与C语言的互相调用  在实验中，我首先学习并实现了汇编与C语言的互相调用。通过汇编和C语言互相调用的实验，我对两种语言的调用约定（如栈的管理、参数传递、函数返回值等）有了更清晰的认识。汇编语言的灵活性和C语言的高效开发能力相结合，使我们在编写底层系统时既可以利用汇编的精细控制，又可以借助C语言的高层抽象和库支持，提升开发效率。  具体来说，我在C程序中定义了一个简单的函数 ，并在汇编程序中调用该函数；同时，我也在汇编中编写了一个函数 ，通过C程序调用，实现了简单的参数传递与返回值的获取。这一部分的工作让我更好地理解了混合语言编程在系统开发中的应用。int add(int a, int b)asm\_add  2. ELF文件格式分析  ELF（Executable and Linkable Format）文件是Linux系统中常见的可执行文件格式。在分析修改后的可执行文件时，我使用了和工具，重点查看了文件头、程序头、节头表等信息。通过对比修改前后的文件结构，我理解了C函数和汇编函数如何被组织到同一个ELF文件中，并深入理解了ELF文件的段（section）和符号表（symbol table）的作用。readelfobjdump  ELF格式的分析让我认识到，这种格式不仅仅是存储机器码的一个容器，还包含了丰富的元数据，帮助操作系统在程序加载时找到代码段、数据段、符号表等关键信息。这部分的学习为后续Loader加载ELF文件的工作打下了基础。  3. Loader加载ELF文件  在实现Loader加载ELF文件时，我首先复习了如何从磁盘读取文件，并解析ELF头，找到程序的入口点，并将各个段加载到内存中。通过这一过程，我对操作系统如何加载用户程序有了进一步的理解。  Loader的实现不仅仅是简单地将文件读入内存，还涉及到地址转换、段的权限设置等工作。通过这一过程，我加深了对操作系统加载程序机制的理解，特别是虚拟内存的概念。实际上，Loader的工作与现代操作系统中的内存管理息息相关。  4. 内核扩展与源码分析  本次实验还研究了如何扩展内核，并对比了自己实现的简化版内核与真实操作系统内核代码的组织结构。通过阅读真实内核源码（如Linux内核），我发现内核代码的结构非常复杂，但同时也非常模块化和组织化。内核的各个模块（如进程管理、内存管理、设备驱动等）之间通过清晰的接口进行通信。  在扩展内核的过程中，我添加了内存管理和中断处理机制。内存管理部分的工作主要是建立页表，并实现简单的动态内存分配；中断处理部分则通过设置中断向量表和中断处理函数，实现在内核中捕获并处理硬件中断和软件中断。  5. 启动代码修改与内核集成  为了增强内核的功能，我修改了引导代码，在启动过程中在屏幕上画出了一个简单的ASCII图案。通过这部分实验，我对操作系统的引导流程和VGA图形模式有了更深入的理解。  随后，我将第三章的内存管理功能代码和我自己设计的中断代码集成到内核文件目录中，并编写了Makefile文件，以方便自动化编译和链接内核。通过这一部分的工作，我更加熟悉了内核的模块化设计，并理解了Makefile的作用——它通过定义依赖关系和编译规则，简化了复杂项目的编译过程。   * 1. 侯名扬   顺利完成了除2.3.6的所有实验步骤，在进行实验过程中充分认识到了汇编语言和C语言的互相调用，分析了ELF文件格式，比较其与PE文件的异同，明白ELF文件装载内存的具体原理，最后实现扩展内核，并添加中断等功能流程。深入理解了Makefile的使用，当然，我在实验过程遇到了如下问题：  **3.4.1**在进行2.3.5实验步骤时，出现了如下问题：  前者warning通过在相应文件前添加disp\_int的函数声明，后者起先我在kliba.asm的disp\_str，disp\_color\_str均做了ebx的保护和恢复，但不起作用，并且在查询后得知在Makefile的CFLAGS中添加-fno-stack-protector解决该问题。  **3.4.2** kernel/kernel.asm中 extern声明 printBootLogo函数，同时在合适的位置调用。  这个位置挑选了若干次，最后确定在使用新GDT之后调用该函数 | | | | | |
| 1. 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）   |  |  | | --- | --- | | 85-100 | * 实验内容完整或者有超出课程实验大纲的内容； * 实验步骤详尽，能够体现完整的实验过程； * 实验结果正确且实验数据分析得当； * 实验心得与思考全面并且有自己的独立思考； * 实验报告文档规范、排版整齐。 | | 75-84 | * 实验内容较为完整； * 实验步骤较为详尽，能够体现实验过程； * 实验结果正确且实验数据分析较为得当； * 实验心得与思考全面； * 实验报告文档规范、排版较为整齐。 | | 60-74 | * 实验内容有缺失； * 实验步骤不够详尽，不能够体现完整的实验过程； * 实验结果部分正确； * 实验心得与思考无或者不够深入； * 实验报告文档规范性有待增强。 | | 60以下 | * 实验内容严重缺失、实验态度不够端正 * 实验步骤不够详尽，不能够体现完整的实验过程； * 实验结果部分正确； * 实验心得与思考无或者不够深入； * 实验报告文档规范性有待增强。 | | | | | | |
|  | | | | | |
| **教师评分（请填写好姓名、学号）** | | | | | |
| 姓名 | | 学号 | | 分数 | |
| 王亚鹏 | | 2022302181161 | |  | |
| 杨依磊 | | 2022302181159 | |  | |
| 杜泓波 | | 2022302181162 | |  | |
| 侯名扬 | | 2022302181165 | |  | |
| 教师签名：  年 月 日 | | | | | |