|  |  |  |  |
| --- | --- | --- | --- |
| **武汉大学国家网络安全学院教学实验报告** | | | |
| 课程名称 | 操作系统及安全设计 | 实验日期 | 2019/11/22 |
| 实验名称 | 内核雏形 | 实验序号 | 六 |
| 1. 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） | | | |
| **实验目的：**  如何生成一个内核，能引导该内核，并进行扩展  对应章节：5.1——5.5  **实验内容：**  1. 汇编和C的互相调用方法  2. ELF文件格式  3. 使用Loader加载ELF文件  4. 如何加载并扩展内核  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  **完成本次实验要思考的问题：**  1.汇编和C的调用方法是怎样的？  2.描述ELF文件格式以及作用  3.如何从Loader引导ELF的原理  4.一个内核要能基本使用应该扩展哪些功能，怎  么扩展  5.怎么管理内核文件目录？ | | | |
| 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；具体的实验步骤） | | | |
| **实验环境：**  VMwareWorkstationPro 15.5.0  Ubuntu 12.04.5 desktop i386 32位  bochs 2.6.9  **关键技术：**  1. 汇编和C的互相调用方法  2. ELF文件格式  3. 使用Loader加载ELF文件  4. 加载并扩展内核  **实验步骤：**   1. **Linux下汇编写helloword**   **代码 a.asm**    Nasm -f elf 指定输出文件格式为ELF  Ld -s ld为链接。s意为strip，链接选项，去除输出文件中符号表等内容，使生成的可执行代码减少。  程序中定义了两个节（Section），一个放数据，一个放代码。在代码中值得注意的一点是，入口点默认的是“\_start”，我们不但要定义它，而且要通过global这个关键字将它导出，这样链接程序才能找到它。代码本身，只是两个系统调用，类似于windows的API，用来显示字符串并退出。     1. **汇编和C同步使用（互相调用）**   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。 这样就包含汇编和C的互相调用。      关键在于在汇编中使用global就可以让C调用汇编。使用extern可以调用C中的代码。  关键字：  • extern：引入外部变量、函数的声明  • global：导出到全局作用域  1. 由于在bar.c中用到函数myprint( )，所以要用关键字global将其导出。  2. 由于用到本文件外定义的函数choose( )，所以要用关键字extern声明。  3. 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。  注意代码有点问题。  链接时应该是ld -s foo.o bar.o -o foobar     1. **ELF文件格式**   **ELF文件结构**    ELF文件由4部分组成，分别是ELF头（ELFheader）、程序头表（Program headertable）、节（Sections）和节头表（Section headertable）。实际上，一个文件中不一定包含全部这些内容，而且它们的位置也未必如图所示这样安排，只有ELF头的位置是固定的，其余各部分的位置、大小等信息由ELF头中的各项值来决定。  ELF header的格式如代码所示。其中各类型的说明见表。    由于ELF文件力求支持从8位到32位不同架构的处理器，所以才定义了表中这些数据类型，从而让文件格式与机器无关。  ELF header中最开头是16字节的e\_ident，其中包含用以表示ELF文件的字符，以及其他一些与机器无关的信息。  以刚才生成的foobar为例。  使用linux的xxd查看16进制格式。xxd的作用就是将一个文件以十六进制的形式显示出来，具体选项如下。可以用xxd --help查看    可执行文件foobar的开头如下所示：    开头的4字节是固定不变的，第1个字节值为0x7F，紧跟着就是ELF三个字符，这4字节表明这个文件是个ELF文件。  然后从0x 10开始才是ident。  第一行为magic。7f 、45、4c、46分别对应ascii码的Del(删除)、字母E、字母L、字母F。这四个字节被称为ELF文件的魔数，操作系统在加载可执行文件时会确认魔数是否正确，如果不正确则拒绝加载。 第五个字节标识ELF文件是32位（01）还是64位（02）的。第六个字节标识该ELF文件字节序是小端（01）还是大端（02）的。第七个字节指示ELF文件的版本号，一般是01。后九个字节ELF标准未做定义。一般为00.    以foobar为例说明ELFheader中各项的含义：  从地址0x10开始：  e \_type它标识的是该文件的类型，可能的取值在这里就不一一列出了。文件foobar的e\_type是2，表明它是一个可执行文件（ExecutableFile）。  e \_machi ne 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 \_versi on这个成员确定文件的版本。  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0x1C0。  e \_flags对IA32而言，此项为0。  e \_ehsiz e ELFheader大小（以字节计数）。这里值为0x34。  e \_phent siz e 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 \_shent siz e 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6个。  e \_shstr ndx包含节名称的字符串表是第几个节（从零开始数）。这里值为5，表示第5个节包含节名称。  Program header table在文件中的偏移量（e\_phoff）为0x34，而ELF header大小（e\_ehsize）也是0x34，可见ELFheader后面紧接着就是Program headertable。  Program header数据结构。    实际上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。  foobar程序头表中共有三项（e\_phnum=3），偏移分别是0x34～0x53、0x54～0x73和0x74～0x93。  其中各项的意义如下：  p \_type当前Program header所描述的段的类型。  p \_offse t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。  Program header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。如果我们想把一个文件加载进内存的话，需要的正是这些信息。  在foobar中共有三个Programheader，其取值如表所示。根据这些信息，我们很容易知道foobar在加载进内存之后的情形，如图。     1. **loader到内核**   根据之前的实验。Loader的工作为：加载内核到内存，跳入保护模式。   * 1. **用loader加载ELF**   加载内核到内存这一步和引导扇区的工作非常相似，只是处理内核时我们需要根据Program header table中的值把内核中相应的段放到正确的位置。  我们可以首先像引导扇区处理Loader那样把内核放入内存，只要内核进入了内存，如何处理它便是一件容易的事情了，我们可以在保护模式下挪动它的位置。  依旧是寻找文件、定位文件以及读入内存，实际上，单就把内核读入内存这一部分，除了文件名和读入的内存地址变了，其余其实都是一样的。之所以没有把它写成一个函数分别在boot.asm和loader.asm中调用，是因为函数在调用时堆栈操作会占用更多的空间，在引导扇区中，每一个字节都是珍贵的。  不过，一些常量的定义却可以在boot.asm和loader.asm之间共享。我们不妨把与FAT12文件有关的内容写进一个单独的文件（文件名为fat12hdr.inc），在两个文件的开头相应的位置分别包含进去。  节选 fat12hdr.inc    修改boot.asm头部使其包含fat12hdr.inc    修改loader.asm，使其包含fat12hdr.inc，并加载内核到内存  节选 loader.asm    和第四章中boot.asm差不多，其中用到的函数如DispStr、ReadSector以及 GetFATEntry和boot.asm中是完全一样的。代码用到的一个新函数是KillMotor，用来关闭软驱马达，不然软驱的灯会一直亮着。  加载内核的代码写好了，可如今我们还没有内核，现在运行的话，将会出现图所示的情况，“No KERNEL”字样会被显示出来。  修改bochsrc  生成a.img    编译loader.asm，boot.asm    写入boot.bin到a.img  写入loader.bin    运行 。提示nokernel。因为此时还没有内核。（见loader.asm，loader在寻找kernel.bin）    我们写一个最简单的，文件名为kernel.asm，我们今后的内核就在它的基础上进行扩充，代码实现的功能照例是显示一个字符。显示字符时涉及内存操作，所以用到GDT，我们假设在Loader中段寄存器gs已经指向显存的开始。 现在“内核”也已经有了，我们来编译它并将其写入软盘映像：      运行我们看到，Loading后面出现一个圆点，说明Loader读了一个扇区。不过，由于目前我们除了把内核加载到内存之外没有做其他任何工作，所以除了能看到“Ready.”字样之外，并没有其他现象出现。     * 1. **跳入保护模式**   **修改loader**  **添加GDT以及对应的选择子。**  我们只定义三个描述符，分别是一个0～4GB的可执行段、一个0～4GB的可读写段和一个指向显存开始地址的段    在第3章我们学习保护模式时，大部分描述符的段基址都是运行时计算后填入相应位置的，因为那时我们的程序是由BIOS或者DOS加载的，我们不知道段地址，于是也就不知道程序运行时在内存中的位置。如今，Loader是由我们自己加载的，段地址已经被确定为BaseOfLoader，所以在Loader中出现的标号（变量）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**  这样一来，BaseOfLoader就同时在boot.asm和loader.asm两个文件中使用，我们也把它以及相应的声明放在同一个文件load.inc中。 我们定义了一个宏BaseOfLoaderPhyAddr用以代替BaseOfLoader×10h，它在代码5.10中被用到一次，**用来计算GDT的基址。**  **\*10h的原因在之前学过。因为保护模式寻址为32位，\*10h相当于左移16位，作为基址。**    **进入保护模式后功能：**进入之后只是打印一个字符    **进入保护模式的代码：**    **继续修改loader**  **初始化寄存器值**。以及TopOfStack定义。**有1KB的堆栈。**等到我们进入内核时，可以重新设置堆栈    **打开分页机制。**  打开之前还是应该先知道可使用内存的情况。在第3章代码pmtest7.asm和pmtest8.asm中不但获得了内存信息，而且把它打印了出来，这里，我们也添加打印内存信息的函数。这里用到的DispInt、DispStr、DispReturn等函数直接从第3章的代码中拿过来用，用单独的文件lib.inc保存这些代码，直接把文件复制过来，将其包含。注意，一定要在32位代码段中包含它。不过，这时DispStr被重复定义了，因为我们本来已经有一个DispStr了，现在我们把原来的DispStr改成DispStrRealMode，这样就不会冲突。    添加启动分页的代码。这段代码从第3章复制而来，稍做修改便可使用。    页目录和页表的定义。字符串和变量的定义      添加调用显示内存信息和启动分页的函数。除了有调用DispMemInfo和SetupPaging的两句代码，还显示了内存信息的一个表头。    测试。我们在第3章中见过的类似功能，现在它已经成为我们操作系统的一部分。     * 1. **重新放置内核**   **复制内核到内存中正确位置。**  根据内核的Program header table的信息进行类似下面这个C语言语句的内存复制  **memcpy(p\_vaddr, BaseOfLoaderPhyAddr + p\_offset, p\_filesz);**  复制可能不止一次，如果Program header有n个，复制就进行n次。  每一个Program header都描述一个段。语句中的p\_offset为段在文件中的偏移，p\_filesz为段在文件中的长度，p\_vaddr为段在内存中的虚拟地址。  由ld生成的可执行文件中p\_vaddr的值总是一个类似于0x8048XXX的值，至少我们的例子  中是一个这样的值（见上文）。可是我们启动分页机制时地址都是对等映射的，内存地址0x8048XXX已经处在128MB内存以外（128MB的十六进制表示是0x8000000），如果计算机的内存小于128MB的话，这个地址显然已经超出了内存大小。即便计算机有足够大的内存，显然，我们也不能让编译器来决定内核加载到什么地方。  我们得让它受控制，解决它有两个办法，一是通过修改页表让0x8048XXX映射到较低的地址，另一种方法就是通过修改ld的选项让它生成的可执行代码中p\_vaddr的值变小。  显然，第二种方法更加简单易行，下面我们就把编译链接时的命令行改为：  **▹ nasm -f elf -o kernel.o kernel.asm**  **▹ ld -s -Ttext 0x30400 -o kernel.bin kernel.o**  程序的入口地址就变成0x30400了，ELF header等信息会位于0x30400之前。此时的ELF header和Program header table的情况如表和表所示。    根据表5.3和表5.4我们知道，我们应该这样放置内核：  **memcpy(30000h, 90000h + 0, 40Dh);**  也就是说，我们应该把文件从开头开始40Dh字节的内容放到内存30000h处。由于程序的入口在30400h处，所以从这里就可以看出，实际上代码只有0Dh+1个字节。我们来看一下Kernel.bin的内容：从中可以看出，从400h到40Dh是仅有的代码，看一下代码5.9，就明白了，0xEBFE正是代码最后的“jmp $”。    将Kernel.bin根据ELF文件信息转移到正确的位置。找出每个Program header，根据其信息进行内存复制。    为什么入口地址是0x30400而不是其他？它的确不是个随便指定的数字，甚至于，在前面章节中我们存放Loader.bin和Kernel.bin的位置也不是随便指定的数字，让我们看一下内核被加载完之后内存的使用情况，就明白了。图5.8是一个内存使用分布图示。    我们才往里放了两个文件吗，但是已经很复杂。虽然我们往里存放的内容不多，但它并不单纯。比如我们一直以来用做显示的以0xB8000为开始的内存，显然就不能被OS用在常规用途；再比如0x400～0x4FF这段内存，里面存放了许多参数，为了保证在用得着它们的时候它们还在，不覆盖它为妙。  回头看看图3.33和表3.7就明白了，通过中断15h得到的内存信息已经明确地告诉我们，09FC00h～09FFFFh这段内存不能被用做常规使用。即便0h～09FBFFh可以被使用，仍然应该把BIOS参数区保护起来以备后用，所以，我们真正可以使用的内存是0500h～09FBFFh这一段。  那么，为什么指定的入口地址0x30400离0x500还那么远呢？之所以这么做是为了调试方便。因为大多数的DOS都不占用0x30000以上的内存地址，把内核加载到这里，即便在DOS下调试也不会覆盖掉DOS内存。  现在，0x90000开始的63KB留给了Loader.bin，0x80000开始的64KB留给了Kernel.bin，0x30000开始的320KB留给整理后的内核，而页目录和页表被放置在了1MB以上的内存空间。  我们为Loader.bin留了63KB的空间，差一点不到64KB。一方面因为它本质上是个.COM文件，另一方面我们在写boot.asm时把文件加载在了同一个段中，文件再大也是不允许的，而且，一个Loader也不会有那么大，所以，63KB应该是足够了。  加载文件Kernel.bin到内存时使用的方法跟加载Loader.bin是一样的，也是放在一个段中，所以它也不能超过64KB。  暂时来讲，我们的内核还没有那么大，所以作为权宜之计，未尝不可，到时候再对代码进行小的修改并不是一件困难的事情。  现在内存各部分的使用情况已经很明了了。Orange'S放置的位置使得内存看上去用得比较紧凑，虽然引导扇区（Boot Sector）把剩余内存空间分割成了两块，但实际上引导扇区在完成它的使命之后就已经没有用了，所以它本身也可以当成空闲内存来使用。  当然，我们目前可能还用不到那些空闲的内存。也可以将Orange'S的各个部分放在不同的位置，只要不和图中所示的不能使用的内存冲突就可以了，这不是一件困难的事情，修改几个宏定义就可以了。   * 1. **向内核移交控制权**   **向内核跳转**  **在loader中完成**  KernelEntryPointPhyAddr定义在头文件load.inc中，其值为0x30400。它必须跟我们的ld的参数-Ttext指定的值是一致的。将来如果我们想将内核放在另外的位置（比如1MB以上的内存），只需改动这两个地方就可以了。    **调试。**  我们看到，第二行中央出现字符“K”，这表明我们的内核在执行了。Loader的使命圆满结束，操作系统内核开始运行了。    **在内核获得控制权之时各个寄存器的情况**  在内核中我们需要这些信息。  如图5.10所示，cs、ds、es、fs、ss表示的段统统指向内存地址0h，gs表示的段则指向显存，这是我们在进入保护模式之后设置的（参见代码5.14）。同时，esp、GDT等内容也在Loader中，下面对内核进行扩充时，我们会将它们都挪到内核中，以便于控制。     1. **扩充内核**    1. **切换堆栈和GDT**   前面提到，esp、GDT等内容目前还在Loader中，为了方便控制，得把它们放进内核中。注意，我们现在可以用C语言了，只要能用C，我们就避免用汇编，这将是我们今后的原则之一。  **修改kernel**    StackTop定义在.bss段中，堆栈大小为2KB。操作GDT时用到的gdt\_ptr和cstart分别是一个全局变量和全局函数，它们定义在start.c中    函数cstart( )首先把位于Loader中的原GDT全部复制给新的GDT，然后把gdt\_ptr中的内容换成新的GDT的基地址和界限。复制GDT使用的是函数memcpy，这个函数我们已用过多次了（比如在loader.asm中，当时叫做MemCpy），这次把它的函数体放在string.asm中。  函数cstart( )中除了用到的memcpy定义在其他文件之外，还用到了一些新定义的类型、结构体和宏，可以在type.h、const.h以及protect.h中找到。  宏PUBLIC定义在const.h中，同时定义的还有PRIVATE，它们用来区分全局的和局部的符号。 GDT\_SIZE也定义在cosnt.h中。    u8、u16、u32等类型定义在type.h中，分别代表8位、16位和32位的数据类型。定义它们可以让我们的代码增加可读性，一眼看过去就知道类型的长度，在操作gdt\_ptr这样的数据时一目了然。    Descriptor用来表示描述符，它类似于pm.inc中定义的宏Descriptor。Protect.h    不把定义放在同一个文件中是为了使程序结构更好。另外，把显示字符“K”的代码去掉了。同时，loader.asm中显示字符“P”的代码也被删除了。我们当时显示它们的目的仅仅是看代码是否执行到了那里，现在我们知道代码运行良好，它们的使命也就结束了。  **编译链接**  ▹ nasm -f elf -o kernel.o kernel.asm  ▹ nasm -f elf -o string.o string.asm  ▹ gcc -c -o start.o start.c  ▹ ld -s -Ttext 0x30400 -o kernel.bin kernel.o string.o start.o  **运行。**  我们没有添加任何打印字符或字符串的代码，还删去了“P”和“K”，什么也看不到。    **扩充：**  把在第3章中写过的代码复制过来，把它放到新的文件kliba.asm中    像memcpy一样，简单地声明一下，在C语言代码中就可以方便地使用DispStr了（在这里我们把它改名为disp\_str）。马上修改cstart( )，添加打印字符串的代码。注意，由于变量disp\_pos开始被初始化成零，所以如果直接打印字符的话，字符会出现在屏幕左上角，于是代码中disp\_str的参数字符串使用了许多个回车（\n），以便让字符串越过已经打印的信息。    **编译**    在编译start.c的时候，如果不加参数-fno-builtin，可能会得到一个警告：  **start.c:12: warning: conflicting types for builtin function 'memcpy'**  因为memcpy被编译器默认为是“builtin function”。 （函数重名）  **运行**     * 1. **整理文件夹 （tree结构）**   boot.asm和loader.asm放在单独的目录/boot中，它们所需要的头文件也放在里面；  klib.asm和string.asm放在/lib中，作为库的形象出现；  kernel.asm和start.c放在/kernel里面。     * 1. **MakeFile**   **一个简单的makefile**  放在目录/boot下，可以用来编译 boot.bin和loader.bin。    以字符#开头的行是注释。=用来定义变量，这里，ASM和ASMFLAGS就是两个变量，要注意的是，使用它们的时候要用$(ASM)和$(ASMFLAGS)，而不是它们的原型。  Makefile的最重要的语法：  **target : prerequisites**  **command**  代表两层意思：  **1. 要想得到target，需要执行命令command。**  **2. target 依赖prerequisites，当prerequisites中至少有一个文件比target文件新时，command才被执行。**  比如这个Makefile的最后两行，翻译出来就是：  1. 要想得到loader.bin，需要执行“$(ASM) $(ASMFLAGS) -o $@ $<”。  2. loader.bin依赖于以下文件：  loader.asm  include/load.inc  include/pm.inc  include/fat12hdr.inc  当它们中至少有一个比loader.bin新时，command被执行。  $(ASM) $(ASMFLAGS) -o $@ $< $@和$<意义如下：  $@代表target；  $<代表prerequisites 的第一个名字。  联系前面我们说过的$(ASM)和$(ASMFLAGS)，这个命令行便等价于：  nasm -o loader.bin loader.asm  不但boot.bin和loader.bin两个文件后面有冒号，everything、clean和all后面也有冒号，它们3个并不是3个文件，仅仅是动作名称而已。如果运行“make clean”，将会执行“rm -f $(TARGET)”，也即“rm -f boot.bin loader.bin”。  all后面跟着的是clean和everything，这表明如果执行“make all”，clean和everything所表示的动作将分别被执行。  make all执行的结果：  ▹ **make all**  rm -f boot.bin loader.bin  nasm -I include/ -o boot.bin boot.asm  nasm -I include/ -o loader.bin loader.asm  **关键字.PHONY**，表示它后面的名字并不是文件，而仅仅是一种行为的标号。  我们刚才已经运行过make all了，其实直接输入make也是可以的，这时make程序会从第一个名字所代表的动作开始执行。在本例中，第一个标号是everything，所以make和make everything是一样的。下面的过程明白地表示了这一点：    由于make会自动比较目标和源文件的新旧程度，所以如果运行一个make之后立即运行另一个的话，make程序不会做任何事，因为所有的文件都是新的，不需要生成什么。  我们已经看到，第二次运行make时出现  make: Nothing to be done for 'everything'.  这样就使得我们每一次make时不必把每个源文件都编译一遍（如果一个大型程序有很多源文件的话）。  第一个Makefile写成了，我们只需稍微改造和扩充，它就可以用于编译和链接整个操作系统工程。  **把这个Makefile挪到/boot的父目录中，然后稍做修改**    代码并没有大的改变，主要是把其中的文件统统加上了路径“boot/”。再运行make：  ▹ **make all -f Makefile.boot**    注意这里使用参数“-f”，指定使用Makefile.boot，而不是默认的Makefile、makefile或GNUmakefile  **在Makefile.boot的基础上扩展Makefile**    因为目录层次的原因，我们把GCC的选项也增加了对头文件目录的指定“-I include”。  通过make disasm我们可以反汇编内核到一个文件。甚至于，通过make buildimg或者make image，我们可以直接把引导扇区、loader.bin和kernel.bin写入虚拟软盘。  输入make image，执行情况如下：    **测试**  来到start.c，在cstart( )的结束处添加一行程序，如果我们运行时看到效果改变，就说明make运行正确。    再make  运行。可见cstartme。说明makefile成功     * 1. **添加中断**   作为一个操作系统，进程毫无疑问是最基本也最重要的东西，于是我们的下一个重大目标应该是实现一个进程。再进一步，我们应该逐渐拥有多个进程。从进程本身的角度来看，它只不过是一段执行中的代码，这样看起来它跟我们已经实现的代码没有本质的区别。可  是，如果从操作系统角度来看，进程必须是可控制的，这就涉及到进程和操作系统之间执行的转换。因为CPU只有一个，同一时刻要么是客户进程在运行，要么是操作系统在运行。我们清楚，如果实现进程，需要一种控制权转换机制，这种机制便是中断。  中断我们并不陌生，在第3章中，我们已经看到了时钟中断发生的效果。我们一边复习一边把中断处理添加到我们的OS中。 要做的工作有两项：设置8259A和建立IDT。  **写一个函数设置8259A（见代码5.35）。**    初始化8259A的函数命名为init\_8259A，它本质上跟第3章中的代码3.34是一样的，只是由汇编代码换成C代码，而且相应的端口被定义成宏。宏定义请见代码5.36和代码5.37。    函数init\_8259A中只用到一个函数，用来写端口的out\_byte，它的函数体位于kliba.asm中。其中，不但有out\_byte，用于对端口进行写操作，还有in\_byte，用来对端口进行读操作。由于端口操作可能需要时间，所以两个函数中都加了空操作以便有微小的延迟。    这两个函数的原型放在了include/proto.h中，这是一个新建立的头文件，用来存放函数声明。可以看到start.c中函数disp\_str的声明也被挪到了里面。    memcpy，我们把它也放进一个新建立的头文件，取名为string.h。由于新增加了头文件，在相应的.c文件中不能忘了包含它们。  最后一件重要的事情就是修改Makefile。不但要添加新的目标kernel/i8259.o，而且由于头文件的变化，kernel/start.o的依赖关系也稍有变化（见代码5.40）。    当确定依赖关系的时候，你可能觉得有点麻烦，尤其是当头文件越来越多。GCC提供了一个参数“-M”，可以自动生成依赖关系。下面是“gcc -M”的典型用法：  ▹ **gcc -M kernel/start.c -I include**  start.o: kernel/start.c include/type.h include/const.h include/ protect.h \  include/proto.h include/string.h  直接把输出复制到Makefile中就可以了。  现在我们已经可以make一下了。虽然目前还没有完成任何实质性的工作，但是make一下，测试一下自己的工作有没有错误还是可以的。通过之后运行我们的操作系统并不会有什么新鲜效果出现，我们甚至还没有添加调用init\_8259A的代码。    **初始化IDT。**  首先修改start.c（见代码5.41）。    代码跟先前初始化GDT的部分基本上是一样的，只是所有的GDT字眼变成了IDT。不过你会发现，原来位于start.c开头的gdt[ ] 和gdt\_ptr[ ]的声明不在了，取而代之的是对头文件global.h的包含。gdt[ ]、gdt\_ptr[ ]以及新增加的变量idt[ ]和idt\_ptr[ ]  都放在了这个新建的头文件中。之所以把全局变量声明都放在其中是为了代码的美感和可读性（见代码5.42）。    EXTERN定义在const.h中（代码5.44），通常情况下它被定义成extern。但是在global.h中你会发现，如果宏GLOBAL\_VARIABLES\_HERE被定义的话，EXTERN将会被定义成空值。这样做的意图联系global.c（见代码5.43）你就全明白了。你会发现，通过宏GLOBAL\_VARIABLES\_HERE的使用，在让所有变量只出现一次（在global.h中）的同时，预编译结束后，global.c和其他.c文件中的结果不同。在global.c中，变量前面没有extern关键字，而在其他文件中，变量前将会有extern关键字。    可以看到，IDT\_SIZE的定义也在const.h中。另外，GATE的定义在protect.h中（代码5.45）。    start.c修改完之后，我们在kernel.asm中添加两句，导入idt\_ptr这个符号（代码5.46第17行）并加载IDT（代码5.46第97行）。  现在，加载IDT的代码已经写完了。不过，现在IDT内还没有任何内容，要抓紧添加。  **添加中断异常**  我们曾经在第3章的表3.8中给出了处理器可以处理的中断和异常列表，现在把这些中断和异常的处理程序统统添加上。虽然它们总数有十几个，但我们却可以用相似的方法来处理它们（见代码5.46）。  异常发生时堆栈的变化情况。中断或异常发生时eflags、cs、eip已经被压栈，如果有错误码的话，错误码也已经被压栈。所以我们对异常处理的总体思想是，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。  **函数exception\_handler( )的原型是这样的：**  **void exception\_handler(int vec\_no, int err\_code, int eip, int cs, inteflags);**  由于C调用约定是调用者恢复堆栈，所以不用担心exception\_handler会破坏堆栈中的eip、cs以及eflags。  **节选**      在代码5.46的最后，栈顶被调整为指向eip，堆栈中从顶向下依次是：eip、cs、eflags。这样做有利于提醒我们以后修改时注意，用iretd返回前的样子应该是这样的。  **函数exception\_handler（见代码5.47）**  它的实现实际上也很简单，首先把屏幕的前5行通过打印空格的方式清空，然后把堆栈中的参数打印出来。    新建立了一个文件protect.c用来放置exception\_handler。需要提醒的是，每新建一个源文件，我们都要考虑在Makefile做出相应改变。  为了突出显示，exception\_handler中打印字符串不再使用disp\_str而使用了函数disp\_color\_str( )，它和disp\_str( )基本上是一样的，区别在于增加了一个设置颜色的参数，见代码5.48。 位于lib/kliba.asm    另外，为了显示整数，我们新编写了函数disp\_int( )，它被定义在新建的文件klib.c中，见代码5.49。    disp\_int很简单，用itoa( )将整数转换成字符串后显示出来。itoa( )也定义在klib.c中，不过它和C库函数itoa( )比起来要简单得多，目的只是把一个32位的数值用十六进制的方式显示出来，既不支持其他进制的转换，也不考虑有符号数等情况。  **设置IDT**  现在我们已经有了异常处理函数，该是设置IDT的时候了。我们把设置IDT的代码放进函数init\_prot( )中（见代码5.51），它也位于protect.c中。  protect.c通篇几乎只调用一个函数，就是init\_idt\_desc( )（代码5.50），它用来初始化一个门描述符。其中用到的函数指针类型是这样定义的（位于type.h）：  **typedef void (\*int\_handler) ( );**    所有的异常处理程序都必须与此声明完全一致（见代码5.51）。      在init\_prot( )中，所有描述符都被初始化成中断门。函数中用到了若干宏，其中INT\_VECTOR\_开头的宏表示中断向量， DA\_386IGate表示中断门，在定义protect.h中定义，PRIVILEGE\_KRNL和PRIVILEGE\_USER定义在const.h中。另外，调用init\_8259A( )的语句也放在了这个函数中。  至此设置IDT的代码总算添加得差不多了，我们现在来调用init\_prot( )（代码5.52）。    对Makefile进行相应的修改之后，我们就可以先make一下了，通过之后运行，会发现什么效果也没有。我们添加了异常处理程序，但是没有异常发生，所以没有效果。我们就制造一个异常来试试看。Intel为我们准备了一个指令叫做ud2，能够产生一个#UD异常，我们就在kernel.asm中添加一条ud2指令（代码5.53）。      再make，然后运行，可以看到图5.14所示的效果了，异常的助记符、名字以及eflags、cs、eip的值都被打印了出来。这是个没有错误码的异常，我们再来产生一个有错误码的异常，把ud2这行指令修改成jmp 0x30:0。运 行，你会发现错误码也显示出来了，如图5.15所示。    上面只是初始化8259A和设置IDT这两项任务。现在我们已经有了异常处理机制，今后，即便出了错，我们也能方便地知道错误出在什么地方以及错误的类型。  不过，8259A虽然已经设置完成，但是我们还没有真正开始使用它。  我们知道，两片级联的8259A可以挂接15个不同的外部设备，我们也理应有15个中断处理程序。为简单起见，我们写两个带参数的宏，用它们作为中断处理程序。代码5.54就是8259A的中断例程。    在这里，所有的中断都会触发一个函数spurious\_irq( )，这个函数的定义如代码5.55所示。    spurious\_irq( )其实什么也不做，仅仅是把IRQ号打印出来而已。下面我们就来设置IDT（代码5.56）。      现在make并运行，不会有什么效果，因为我们不但没有通过任何方式设置IF位，而且在init\_8259A( )中把所有中断都屏蔽掉了。  对i8259.c处做代码5.57这样的修改：    在这里，我们向主8259A相应端口写入了0xFD，由于0xFD对应的二进制是11111101，于是键盘中断被打开，而其他中断仍然处于屏蔽状态。最后，在kernel.asm中添加sti指令设置IF位（代码5.58）：    make，运行，开始没有什么特殊的现象，但当我们敲击键盘的任意键时，字符串“spurious\_irq: 0x1”就出现了，这表明当前的IRQ号为1，正是对应的键盘中断，如图所示。     1. **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. **引导过程画ascii图案**   修改start.c，在启动时在屏幕用\*画出 Z。  调用disp\_str()和disp\_color\_str() （/lib/kliba.asm）     * 1. **集成自己设计的中断代码**   集成时钟中断和键盘中断。时钟中断为在0行70列字符自增，键盘中断为在0行72列显示键盘输入的字符。   1. 打开8259A的键盘和时钟中断   修改/kernel/i8259.c  主8259A写入0xFC,11111100.     1. 修改kernel.asm，即修改时钟和键盘中断对应的处理程序   在kernel.asm添加死循环  设置死循环原因是有一个问题：程序马上会继续执行，可能没等第一个中断发生程序已经执行完并退出了。    （3）添加时钟中断：  先在屏幕0行70列显示 字符 ‘I’    然后修改hwint00，即8259A 0端口，时钟中断对应的中断程序。  这样每次时钟中断，0行70列的字符自增1    （4）添加键盘中断：  修改hwint01，即8259A 1端口，键盘中断对应的中断程序。每次按下和松开键盘按键，触发键盘中断，显示扫描码和断码。    运行效果： | | | |
| 1. 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） | | | |
| **遇到的问题及解决：**   1. **做5.2节C与汇编调用时ld链接出错**     原因是代码有问题。应该改为ld -s foo.o bar.o -o foobar   1. **不清楚怎么读ELF文件头**   参看<https://blog.csdn.net/king_cpp_py/article/details/80334086>  结合readelf   1. **Make chapter/i/lib/Klib.c报错**     解决：参考<https://blog.csdn.net/xiaominthere/article/details/18084865>  修改makefile  在Makefile中的$(CFLAGS)后面加上-fno-stack-protector，即不需要栈保护  lib/klib.o:lib/klib.c         $(CC) $(CFLAGS) -fno-stack-protector -o $@ $<  之后make问题解决     1. **Make all 与make image 区分**   对于h，i应该使用make image   1. **h中ud2这行指令修改成jmp 0x40:0显示的错误与书中不一致**   因为我的内存与书中的不一样，修改为jmp 0x30:0即可。  **实验结果记录：**   1. **汇编写hello world**      1. **汇编与C互相调用**        1. **ELF文件格式**   可执行文件foobar的开头如下所示：      Program header数据结构。     1. **Loader到内核**    1. **用loader加载ELF**   nokernel。此时还没有内核。    最简单的kernel.asm编译并将其写入软盘映像：      运行，Loading后面出现一个圆点，说明Loader读了一个扇区。     * 1. **跳入保护模式**      * 1. **重新放置内核**     将Kernel.bin根据ELF文件信息转移到正确的位置。找出每个Program header，根据其信息进行内存复制。   * 1. **向内核移交控制权**      1. **扩充内核**    1. **切换堆栈和GDT**      * 1. **整理文件夹**      * 1. **Makefile**          * 1. **添加中断**        1. **设计题**    1. **引导过程画ascii图案**      * 1. **集成自己设计的中断代码** | | | |