|  |  |  |  |
| --- | --- | --- | --- |
| 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 | 2025.10.29 |
| 实验名称 | LAB6 | 实验周次 | 第8周 |
| 姓名 | 学号 | 专业 | 班级 |
| 林水利 |  | 网安 | 试验班 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| 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） | | | |
| • 如何生成一个内核，能引导该内核  • 对应章节：5.1——5.5  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  – 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、实验步骤、实验结果   （本次实验所使用的器件、仪器设备等的情况；具体实验步骤；实验结果） | | | |
| **实验环境：**  Bochs  **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。  汇编调用c的方法：在c中实现一个函数，然后在汇编中extern这个函数就可以call过去了  c调用汇编的方法：在汇编中global一个函数，然后在C中声明这个函数，就可以在C中使用了。  他们的调用约定是\_\_cdecl，参数从右往左先后压栈。由调用者维护这个栈。以及调用者来保存eax,ecx,edx。被调用者需要保存ebx,esi,edi,ebp,esp。返回值为eax。  自定义函数：实现了汇编在调用choose之后，调用C中的change，C中的change调用汇编中的add1st的功能。并重新打印。    图 1 asm代码    图 2 bar.c的change定义  我们预测一下执行结果应该是先输出the 2st one（因为b大），然后调用change更改了值，具体来说调用了asm中的add1st，然后之后就a大了所以输出的1st。    图 3 输出结果  2. ELF文件格式  – 依照书上方法，分析你修改的这个可执行文件  1.首四字节 即0x7f 'ELF'表明这是个ELF文件    图 4 ELF文件  2. ELF结构分析，为了方便，我们使用二进制编辑器。    图 5 elf header  010editor中使用elf模板解析可得头相关字段。  e\_type为exec，machine为80386，版本为1，入口地址0x80480A0，program\_header在文件中的偏移52d，section\_header在文件中的偏移为468d。elf头大小为52d.  下面的是program header条目的大小，一条目为0x20，以及后面是program header条目数，验证如图    图 6 program header条目  然后是section header条目大小，section header的条目数  最后一个根据书上的解释，这个是节名称字符串表作为一个节，这个节的节头所在的位置。即，我们索引下标为6的节头    图 7 按照section索引节  转到节对应位置    图 8 节对应位置  可以发现这里存了各个节的名称，而每个节头存的是偏移地址。  如第element[5]，name字段0x29，就是相对于1A1+0x29=1CA对应的是这个节的名字的偏移    图 9 存储名字偏移    图 10 节名表  以上为elf\_header解析。  program\_header：    图 11 program header解析  p\_type描述段的类型  p\_offset为该段的偏移  vaddr,paddr分别为虚拟地址和物理地址  filesz是文件中段长度  memsz为内存中段的长度  p\_flags是段相关标志，可以发现这个段是读和执行  p\_align是对齐标志。  可见这个元素指向了这样一个段：    图 12 Loadable Segment  这里包含的是程序头和程序运行要使用的内容比如代码和字符串常量（在后续节表分析中可以发现这里存了除数据段和后两项之外的节数据）。后面8字节是下一个元素指向的段，结合后面的调试过程，认为可能是数据段，也就是num1和num2。    图 13 最后一个program table element  最后一个元素是指向栈段，可以发现该元素位置都为0，似乎有点问题。实际上栈是由操作系统进行管理的，这里只是对栈的权限进行了描述。  不过这些table似乎都是描述给程序加载器，描述这些段应该加载到内存的什么位置。  接下来是初步分析一下sections，以解答一下上面的问题。    图 14 section table  节表的位置都不在以上加载进内存中的位置，或许程序执行时用不上这些东西。  继续观察发现.comment指向如图    图 15 comment段的指向  .shstrtab指向下面的位置。那么整个程序的指向都明了了。根据program file header的分析，认为这里之后的位置都不会被内存加载进去。elf的节表主要用于调试器和链接器，loader不会对这里有什么关注（本句话总结自GPT）；  ELF和PE这两个结构作用相同，都是对程序进行了描述，包括定义他们的装载地址，执行权限，大小等，且都以节表来索引节（其中elf不使用节表加载）。但是elf头更加精简，PE结构的头有太多不用的字段。而且PE头使用节表来加载内存，而elf头使用program header对内存进行映射，还存在不使用的节。他们都在相应的header中对这些区域的权限进行了限制。  3. 使用Loader加载ELF文件    图 16 boot加载loader以及loader加载elf  如图，运行书中的代码即可。  先前已经实现了从引导扇区加载loader到内存并跳转过去的功能，我们只要把这个功能的内存地址和文件名改一下就可以在loader中加载elf。  代码原理跟上一次lab一样，这里就不解释了。    图 17 kernel文件加载到内存  注意到kernel被加载到内存中的位置如图。    图 18 kernel.bin 在内存中的表现形式  验证可知loader把整个kernel.bin复制到内存该段，此时显然是不能执行的。  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  第三步已经描述了如何将内核原封不动的复制到内存中。  现在在重定位内核之前，我们先按照书上所述，用Loader开启保护模式    图 19 准备切换到保护模式  这里跳过去的是相对于相对于0的loader偏移+PM相对于loader的偏移那就是pm相对于0的偏移。    图 20 PM\_START代码段  这里做了初始化段寄存器和栈，栈定义在最末尾。    图 21 栈定义  然后打印这个图在实模式下获取到的内存布局    图 22 打印物理可用内存结构  最后启用分页。  运行一下，我们得到如下图    图 23 运行结果  与之前在DOS中的区别是我们在这里实现了从BIOS到引导程序到loader到保护模式的实现，之前是DOS到保护模式。  内核重定位：  在这里需要解析program header的格式并把对应的区域映射到对应的地址中。    图 24 program header第0项  也就是说把p\_offset 对应p\_filesz的部分复制到p\_vaddr处，这里vaddr过大，因此需要链接器args -Ttext进行重定位。    图 25 编译与链接重定位入口地址  如书上所述。而且代码还给了memcpy的实现，代码给的太多了。    图 26 内核结构加载函数  1Ch是program header的偏移，2CH是program header 条目的数量。  esi在.Begin前就是内存中program header起始相对于0的位置。（因为偏移加上了kerneladdr）  push [esi + 010h]是p\_filesz字段，该字段大小为4。+04是p\_offset字段，取出之后加了kerneladdr，然后入栈。+08 是p\_vaddr字段，这个不用加，因为就是目的地址。然后入栈。  最开始比较的是第一个字节是否为0，为0则退出循环。以及.NoAction还有一个作用是跳到下一个条目。    图 27 pgt的结尾字段  可以发现program header结束有一些0，可以作为结束循环的条件。    图 28 内核内存地址以及入口    图 29 跳转到kernel入口  最后我们跳转过去就可以了。    图 30 内核加载运行结果  发现P，保护模式开启，K，内核已经运行。加载内核就到这里。  以下为到现在的内存结构：    图 31 内存结构[[1]](#footnote-1)  内存布局：0x80000h开始的是kernel文件，loader文件在0x90000h。kernel的运行在0x30000  扩展内核。注意只有kernel.bin是elf格式的，loader.bin是nasm直接编译成机器码所以不用重定位。    图 32 寄存器相关指向  进入内核时，寄存器情况如图，可以发现esp还是loader区域的，除了gs，其他段寄存器存的选择子指向的都还是loader区域的GDT。  1.切换堆栈和GDT  由于堆栈和GDT都还在loader中，那我们要切换到kernel中。    图 33 为什么要为\_start    图 34 编译结果  如果\_start改成\_start1就会发生这种事情，所以这是ld默认的程序入口点（在不指定ENTRY的情况下）。    图 35 初始化栈指针到内核的内存区域  mov esp, StackTop干了把esp赋StackTop的工作，注意此时的StackTop应该为虚拟地址，也就是说不是偏移。所以一条命令就简单重定位了栈指针。  接下来是把gdtr的内容保存到C里面定义的变量中    图 36 进行gdt替换所需的指向gdt的指针  GDTR结构如图    图 37 GDTR结构[[2]](#footnote-2)  然后用cstart拷贝GDT，如图    图 38 cstart对loader中的GDT进行拷贝并替换  认为从gdt\_ptr[2]的地址是一个u32指针，以u32的格式去读取gdt\_ptr[5..2]，然后外面一个\*解引用出gdt\_ptr[5..2]的内容作为老的GDT的地址，这个地址作为一个指针传入memcpy。  注意在这里，跟我的好友讨论之后，可以用union的方式更优雅的解决，避免了指针的复杂运算，详细请看rust版本代码。  limit只需要读出值，所以外层没有类型转换。+1才是大小，不然是最大地址。  memcpy的位置    图 39 string.asm  老GDT的内容拷贝到了gdt变量中。    图 40 段描述符结构体  这个描述符跟asm中的结构对应。    图 41 汇编中的段描述符宏  用C很好的一点是不用看这种复杂的位运算了。    图 42 进行gdt\_ptr的替换，用于后续lgdt  这两行就是把&gdt\_ptr[0]当做一个16位的指针，意思是指向一个gdt\_ptr[1..0]为两字节的内容。  下面同理。进行修改（意思是把低2位当成p\_gdt\_limit的修改单元，高4位为p\_gdt\_base的修改单元，所以用指针会很方便），填充回gdt\_ptr即可。  然后回到kernel进行lgdt。    图 43 选择子的结构    图 44 KERNEL\_CS选择子  注意到CS选择子为新GDT的下标为1的项。    图 45 jmp过去  那么就jmp过去，类似于之前实验中，进入保护模式的史诗级jmp，只是这里不是进入保护模式，而是使用新的GDT的jmp。    图 46 EFLAGS结构  将eflags清零。    图 47 kliba.asm  disp\_str辅助函数在kliba中。    图 48 与c联合实现的kernel  成功运行。  以下为给内核添加中断：    图 49 init\_8259A  跟书上描述的一样，与asm对应    图 50 IO操作的封装  kliba实现端口操作的底层指令。参数不是u16吗，为什么此处是这样？因为高位用不上，所以这里不会错误。据说\_\_cdecl调用约定的压栈单位都是4B。因为时间关系这里就不实验了。  我只需要在这个汇编中global这些函数，然后在C的header中声明这些函数那我就可以在C中使用。  gcc -M的用法    图 51 GCC -M  直接复制到Makefile中。    图 52 初始化IDT    图 53 init\_idt\_desc  这里就是根据描述符的结构进行填充，不过可能要注意这里的对齐问题，特别是结构体。    图 54 中断处理函数的声明，实现在asm中    图 55 kernel.asm中对中断处理程序外部接口  他们的实现在kernel的asm中    图 56 exception，这里调用c实现的异常处理函数  异常处理程序    图 57 C的异常处理函数，对asm触发异常之后进行处理    图 58 各种中断或异常发生的堆栈变化情况  handler的声明合理，根据这个栈和汇编实现来看的话，没有错误码就push一个fff，有错误码就push向量号。从而能够统一到一个函数中处理。    图 59 方便输出的辅助函数    图 60 global.h  注意到global.h的实现    图 61 global.c  只有global.c才声明了变量，其他地方的include都是extern过来的。保证了只有一次的声明。    图 62 异常触发运行图  运行发现如图。  正常触发了异常。    图 63 异常触发指令 ud2  接下来是接入中断。    图 64 中断的外部接口  定义了一个处理宏，在汇编代码段收到中断之后跳到C的spurious\_irq函数中处理    图 65 spurious\_irq  irq打印中断号    图 66 将中断初始化到IDT中    图 67 在这里设置中断mask    图 68 设置IFlag  测试：  注意make要在Cflag加入-fno-stack-protector  不然会提示：    图 69 错误提示  gcc可能自动插入栈检查    图 70 加入-fno-stack-protector  再make就行了。因为exception handler定义了一堆字符串，都是存在栈上的。而gcc no builtin导致栈保护相关机制不能执行。    图 71 产生键盘中断  发现按下键盘后产生中断。  5.分析真正的内核源码：  以linux仓库，commit hash = 566771afc7a81e343da9939f0bd848d3622e2501 为例。  在arch/x86/boot/header.S中找到了\_start    图 72 linux的\_start  由于linux的boot是基于grub的，因此我们阅读的源码缺失了boot和bootloader的部分。可以发现他的跳转指令直接以机器码形式写了。这里是elf的setup了。    图 73 elf的header  一大堆宏定义是确定是32位还是64位的，因为elf32跟64的格式有点不同。    图 74 setup的开始  这里是entrytext，设置各种段寄存器和堆栈，有栈回绕则设置sp为0xfffc。esp高2字节为0。最后push ds和下一段代码的ip。所以这里为什么cs会等于ds+0x20.似乎网上没有人提出这个问题。（TODO）setup用于加载linux内核。    图 75 进入到C  这里到main里面，    图 76 main.c  main就是各种初始化，跟我们的代码差不多。    图 77 进入保护模式    图 78 jmp到code32的开始    图 79 hdr结构的一部分  在这里，start0x100000，应该是压缩内核bzImage的地址    图 80 linux boot时的内存布局  在这里，我们也可以发现，Kernel setup 的cs比Protected-mode kernel 多了0x20，所以之前那个0x20可能是老协议的遗留问题。  bootloader会加载两部分文件，一个是setup program，还有一个是压缩内核。[[3]](#footnote-3)  我们发现linux使用linker script 对内核内存布局进行考虑    图 81 linker script  在x86这个文件夹下，linux的布局跟我们的布局其实应该差不多，boot里面包含加载内核的代码。  在之后可能还要进入长模式，以及解压缩bzImage。  在compressed的Header32/64.S中    图 82 解压bzimage    图 83 的c实现  这里是解压缩代码，解压缩完之后就到/x86/kernel里面了。发现这里的kernel大部分跟硬件与初始化相关，因此认为是最初的kernel。分析就到这里吧。  我们总结一下，linux首先从/boot/head.S里面处理了各种东西，然后进到.c里面打开保护模式，然后转移到/boot/compressed里面去解压bzImage，/boot/compressed跳转到解压后的/kernel对应的内核，然后执行内核。这之后的就是系统上层的工作了。  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.画图案。    图 84 图案  对已有的DispStrRealMode稍加修改：    图 85 修改loader的显示    图 86 演示  2.添加内存管理功能代码  由于上一次实验直接使用汇编编写了内存管理代码，这里沿用上次的工作。只需要在基础上修改调用方式即可。    图 87 修改后的mm  新增两个获取bitmap状态以及初始化bitmap的函数    图 88 mm.h  测试函数与上次类似    图 89 测试mm  初始时，initbitmap，因此bitmap4个u32全0。  设va=0x401000  0x401000代表这个va初始的映射pa  0x401007代表unmap之后获得的va对应页表项  0xF 代表unmap之后对应va的页表项不存在  后面是allocate了3个frame  bitmap显示7正确。  0x300000 0x301000 0x302000是分配的pa们  把0x301000跟va map 显示出是这个  然后再unmap是0xF  再free掉0x301000，看一下bitmap正确。  最后再allocate 1 frame，测试bitmap重用。  3. 添加上次尚未来得及fix的鼠标中断。  这里鼠标中断使用的是osdev[[4]](#footnote-4)的代码，复用即可。注意mask要开键盘的、IRQ2的、IRQ12的。  演示视频[[5]](#footnote-5)中实现的是显示当前鼠标位移，注意看二进制的最高位，1的话就是负数。比如向右就是x=1，向左x=-2.上下同理。  **勘误：**  **1.**    图 90 disp\_pos  之前使用的是C语言，对寄存器的利用可能没那么强，使用rust的话，甚至会用edi传参，因此这里需要加上对edi的存储。    图 91 exception触发（rust版本）    图 92键盘中断（rust版本）  鼠标中断的掩码应该MASTER 0xF9 SLAVE 0xEF 也就是键盘跟鼠标联动的。  **思考题解答**  **1.汇编和C内定义的函数，相互间调用的方法是怎样的？**  详细见实验内容1. 简述为：在汇编中global 相应函数，在C语言中声明相关函数即可调用。在C中实现相关函数，并在asm中extern这个函数即可调用。  **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？**  请看实验内容2. 简述为：ELF格式是程序生成的一种外壳，用于描述程序的结构，对加载器来说是很有用的。操作系统要运行一个程序就是从ELF对这个程序的结构描述这里对内存进行布局以及加载各种变量的值。如已初始化的值。  **3.如何从Loader引导ELF的原理？**  LOADER加载KERNEL.BIN到内存中，此时还只是一个文件。然后读取program header table对将要放入内存中的东西进行加载，加载到内存中某处之后，读取elf\_header中的entry，然后跳转过去。  **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？**  1.扩展了GDT和IDT。这些数据结构都在include中定义的。流程就是将loader的GDT复制到kernel的GDT中并将gdtr指向它。而IDT是新定义的。在这里它用protect.c/init\_prot中先初始化8259A这个中断控制器，然后初始化IDT为各种中断门。其中中断门的处理函数又写在kernel.asm中。触发中断之后，kernel.asm的中断处理函数又跳到C中的处理函数来。其中exception的处理在protect.c中，用于处理异常。而硬件中断的handler实现在i8259.c中。实际上后续应该分别实现各种处理函数。目前的都是默认处理函数。这些功能的关联似乎差不多就这些。  在rust中只需要在某个文件中extern "C" {函数头}，然后pub他们，或者不pub甚至进行一次安全封装到pub函数。在其他地方调用它们即可。  而asm调用rust需要rust函数前声明#[unsafe(no\_mangle)]以防止编译器对函数的重命名。然后asm中extern这些函数即可调用。注意rust中需要extern "C"来启用\_\_cdecl调用约定。此时也需要格外注意汇编对应该由被调函数保存的寄存器。因为rust的优化会利用到各种寄存器。没有保存会UB。  **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？**  内存布局见实验步骤3的书上的图。从boot到loader进入保护模式之前，都是实模式下的，实模式下没有特权非特权的概念。而进入保护模式之后，从PM\_START到KernelEntry，再到\_start到cstart到csinit都是特权代码（ring 0），从GDT我们可以看到只有VIDEO的描述符的DPL=3。由LAB2实验报告“数据段总是非一致性的，只能由高特权级和相同特权级的代码访问”我们知道这是合理的，且推断出kernel代码段都是ring0。注意到特殊的一点：在init\_prot中init\_idt\_desc的参数有些权限是KERN,有些是USER。    如图，CALL GATE是当前代码段的权限高于或等于GATE DPL，而小于或等于目标代码段。则在中断发生的时候会进行特权检查。但是由于当前代码段和中断处理代码段都是ring0的所以并没有发生特权转换。也就是说，目前的代码都是kernel内的。那么如果是用户程序在ring3下运行，并发生了例如overflow这样的异常，那么它进入之后就会进入到内核的handler，内核的handler的特权高（ring 0）。  **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。**  见实验步骤5.除了各种必须的文件在内核中之外，各种组件似乎都是分文件夹存放的。  **7.完成设计题并能演示。**  见代码仓库. | | | |
| 1. 实验分析与总结   （对实验过程、结果进行分析总结，提出实验的改进意见） | | | |
| 分析总结：  1.本次实验学习了三级引导的过程，即从BIOS到BOOT到LOADER的引导过程。  2.本次实验学习了ELF的文件格式，并与PE文件格式进行了较浅对比。  3.本次实验初步探索了kernel的写法，并实现了一个简单的带异常和中断机制的kernel。  4.本次实验实现了对之前实验的mm以及鼠标中断的移植。并绘制了一个图案。  5.本次实验初探linux代码和目录结构。  改进意见：  无 | | | |
| 1. 个人实验贡献与体会（每人各自撰写） | | | |
| 1.林水利：完成所有任务  分析总结：了解了C的\_\_cdecl的调用约定，导入外部的东西使用extern，对外公开使用global。rust使用no\_mangle来关闭编译器对函数名的重命名。学习ELF格式，主要了解了entry字段和program header字段。学习了loader加载elf到文件，再以elf的格式约定加载到内存的过程。了解了书中的代码结构以及linux的代码结构和目录管理。从设计题的过程中强化了一些工程能力和调试能力。由于高级程序设计语言都有生成elf的特性，不妨尝试使用rust编写内核。详细见博文1[[6]](#footnote-6)和博文2[[7]](#footnote-7)。 | | | |

1. 《Orange's 一个操作系统的实现》P141. [↑](#footnote-ref-1)
2. 《IA-32架构软件开发人员手册\_卷3：系统编程指南》图2-4. [↑](#footnote-ref-2)
3. https://www.0xkato.xyz/linux-boot [↑](#footnote-ref-3)
4. https://forum.osdev.org/viewtopic.php?t=10247 [↑](#footnote-ref-4)
5. https://www.bilibili.com/video/BV1hx1FBvEdf/ [↑](#footnote-ref-5)
6. https://zhuanlan.zhihu.com/p/1968439333670322661. [↑](#footnote-ref-6)
7. https://zhuanlan.zhihu.com/p/1968440372133863773 [↑](#footnote-ref-7)