LAB1实验报告

姓名：魏伯繁 学号：2011395

首先我在前面按照助教的要求只介绍我在做实验的时候遇到的bug以及解决方法（文风相对“朴素”），同时我也会在后面简单的概述一下我对各个实验的理解（不是回答实验手册上的问题）

实验一遇到的问题及解决方法：

问题1：

在实验一遇到的最大的问题就是不知道这个实验是干嘛的，也不知道想让我干吗。于是我尝试从实验指导书的细节入手去尝试，他说静态分析代码，可是我都不知道要读哪块的代码，然后在实验的最后告诉了我怎么调试makefile，于是我尝试着点开了makefile，然后发现那个文件居然是可以打开的，然后成功在里面通过注释找到了需要的代码（create ucore.img）

问题二：代码看不懂

根本不知道这个makefile文件的编写规则，也不知道每一句代码要干吗，于是充分利用了互联网的优势进行搜索，逐步对makefile文件的编写方式有了一个模糊的印象，有些时候也确实可以通过函数名猜出来这个函数想干嘛（毕竟没让我写代码，能读通就好了）

比如我能明确知道代码在干吗的部分：

```
CC      := clang
```

比如这个语句，我就知道他在声明变量，用这个变量的时候加一个$就能用了，要是想加点东西可以用 += 在这个变量后面再加点东西

```
CFLAGS  += $(addprefix -I,$(INCLUDE))
```

再比如这个，我为数不多能看的差不多懂的代码，addprefix就是加前缀，给谁加前缀？就给INCLUDE变量下的变量内容加前缀，加啥？就加I

下面是我真看不懂的，但我能猜出大概来的：

比如一看这foreach，再看这个bootfiles就大概能猜到这个bootfiles文件大概就是存储了好多文件，然后要依次对他们做事情，做什么呢？看后面有一个call，那应该是就是要调用什么函数了。再往后看一下这个complie，立刻联想到王刚老师教的编译原理，再往后看这个Os，心头一紧，这不是优化选项么，预备实验1做的啊，于是心里就有谱了，应该就是就是要对前头说的的那些文件进行编译。具体的编译参数肯定是由f、CC、CFLAGS定义的

```
$(foreach f,$(bootfiles),$(call cc_compile,$(f),$(CC),$(CFLAGS) -Os -nostdinc))
```

再往回一找，果然差不多，看看这个CFLAGS变量，一查，果然是编译参数，这一句makefile语句就猜完了

```
CFLAGS  := -march=i686 -fno-builtin -fno-PIC -Wall -ggdb -m32 -gstabs -nostdinc
$(DEFS)
```

问题三：在写磁盘扇区规范那道题的时候我自己的答案和网上的“参考答案”不太一样

其他那两个磁盘大小和结尾什么的我都认可，但我看到了这个sign.c里边还有这个代码：

大体就是说，这个stat结构体存储了文件的基本信息，然后我通过一个命令行参数argv[1]把他读进来读给st，然后在这比较了一下st和源文件大小不一致就报错，那我想这不是也是一个条件么，这个为啥不算啊？

但后来思考了一下，这个环节和题目要求的“硬盘主引导扇区”的要求好像没啥关系，这就是为了保证可靠性判断了一下，所以不能把他算作答案。

```
int size = fread(buf, 1, st.st_size, ifp);
    if (size != st.st_size) {
        fprintf(stderr, "read '%s' error, size is %d.\n", argv[1], size);
        return -1;
    }
```

实验二：

问题一：这是我们在网上参考资料上找的的代码，作用是找到CPU上电后的第一条指令，然后break就是要在哪里打断点，但是我们始终不能调试CPU上电后的第一条指令

```
file bin/kernel
target remote :1234
break kern_init
continue
```

最后通过查阅资料把对应的代码修改成了：就可以正确的去到相应的位置，但问题是我们还是看不到相应的代码，只能看到一堆零

```
set architecture i8086
target remote :1234
layout split
```

最后，通过小组间的互相交流和与助教沟通才明白：

pc配合cs得到的地址才是真正的指令在内存的地址，所以真正想找到指令地址，*必须先手动通过* pc=cs*16+ip计算出指令在内存中的地址，然后再用x /8i 地址来找到对应的指令！！！另外，有一个现象，当cs的值变为0x8的时候（也就是完成实模式向保护模式切换的过程中），eip此时已经是20位地址，不再使用pc=cs*16+ip计算指令地址，而是直接使用eip内的数据作为指令地址。

问题二：bootasm和bootmain的关系

一开始真的不理解这两个的关系，突然就是地址有一个比较大的跳转，然后看了代码发现bootasm可以调用bootmain 突然有豁然开朗的感觉

```
file obj/bootblock.o
target remote:1234
break bootmain
continue
```

实验三：

问题一：为什么需要seta20.1和seta20.2来完成对于output port2的更新？直接赋值不就可以了么？

通过阅读实验指导手册可以得知：真正的更新操作没有那么简单，seta20.1是往端口0x64写数据0xd1，告诉CPU我要往8042芯片的P2端口写数据；seta20.2是往端口0x60写数据0xdf，从而将8042芯片的P2端口设置为1. 两段代码都需要先读0x64端口的第2位，确保输入缓冲区为空后再进行后续写操作。

所以说，当准备向8042的输入缓冲区里写数据时，可能里面还有其它数据没有处理，所以，我们要首先禁止键盘操作，同时等待数据缓冲区中没有数据以后，才能真正地去操作8042打开或者关闭 A20 Gate。

问题二：看完整个指导手册的描述有点疑惑就是选择子（也就是段寄存器）和段描述符的联系是什么，为什么对偏移这个东西会有两个描述？

通过上网查阅资料以及再次认真阅读材料我们发现，段寄存器其实存的不再是段的起始地址了，段的起始地址已经存在段描述符里了，而段描述符则存在了全局描述符表里，所以段寄存器中存储的实际上是一个索引，利用该索引可以从全局描述符表中取出我需要的段描述符，从而得到段的基地址。

实验四：

问题：一开始做这个实验很容易被搞糊涂，因为这个实验问的我觉得有点问题

就是我觉得实验指导手册这两个问题很容易把实验割裂开，让我认为bootmain是先读磁盘，再读elf，其实不是的，真正的逻辑是bootmain先从第一个磁盘扇区开始连读8个扇区，把他们读到从0x10000开始的内存地址，然后呢，在宏定义段，0x10000这块地址已经被强制类型转换为了struct elfhdr *，也就是说，要把这一部分的内容当做elf文件来看待，然后根据elf文件记录的各个段的信息写入内存。

实验五：

问题：为什么read_ebp是内联函数而read_eip不是内联函数？

是否设置成内联的区别就是是否会产生函数调用如果设置成了内联是不会产生函数调用的。当函数调用产生时必然会导致栈结构的变化。所以说我ebp必须设置成内联函数，eip可以不设置为内联函数ebp+1也就是只想ret语句，就是函数调用的下一条语句，所以不需要内联

实验六：

最主要的问题就是不知道中断具体的流程是什么，然后感觉实验指导手册写的很混乱。

首先得知道，我CPU从总线上只能拿到索引，我要通过这些异常的索引去找真正的处理异常的地址。而vector表中其实已经存在了我们所谓的处理地址，那么初始化的时候就需要用这些值去初始化IDT，这样我们的cpu在检测到中断后才能第一时间左移3位到IDT表中去查逻辑地址，根据CS段和EIP offset定位真正的入口地址，并判断是否发生了特权级切换等操作然后找到vector的地址进行异常处理指令。

还有一个问题其实就是对vector不太理解，一开始怎么看也看不懂，截止到写这块的时候算是明白一点了

```
.text
.globl __alltraps
.globl vector0
vector0:
 pushl $0
 pushl $0
 jmp __alltraps
```

拿上面这个代码为例，就是说每一个vector先push两个东西，我觉得是用来标记一下这个是中断，然后说一下中断的标记符号就好，最重要的就是这个jmp _alltraps，跳转到这个位置去执行对应的trap对应的操作，这个函数在trapentry.s里面定义过了，大概的意思就是先保存现场环境也就是各个寄存器的值然后再call trap调用真正的trap处理函数，而这个trap函数就是在trap.c中的函数，然后trap函数再调用trapdispatch

challenge1：

说来离谱，但我对于challenge1不太理解的地方就是究竟怎么进行

然后看了很多遍实验手册，也查找了很多资料，找到了一个网上总结的还不错的博客，自己根据ucore修改后的总结如下：

通过中断切换特权级
当中断（硬件中断、软中断（INT指令）、异常）产生时，会保存
SS|ESP|EFLAGS|CS|EIP|ErrorCode，SS和ESP 表示中断时堆栈信息，CS和EIP 表示中断处理完成后返回的地址，EFLAGS 表示中断时程序标志位。也就是说：中断可以发生在任何一个特权级别下，但是不同的特权级处理器使用的栈不同，如果涉及到特权级的变

化，需要对SS和ESP寄存器进行压栈

特权级切换原理：
1）高特权级（level0）到低特权级（level3）：直接将CS的CPL从0改成3、再将SS、ESP、EIP赋值为level3 的地址、再通过执行IRET指令完成切换。

2）低特权级到高特权级：在level3中使用软中断切换到level0,与此同时cpu会将level3中断的现场信息压入level0的栈,在中断返回前将栈中的CS的CPL从3改成0，且将其地址指向level0 要执行的下一个服务地址，去掉SS、ESP，通过执行IRET指令完成切换。

然后为了完成实验还有一个很重要的东西就是TSS：

TSS: 用于存储不同特权级堆栈信息的地址，当程序**从一个特权级切换到另外一个特权级时，则通过TSS获取堆栈地址。**
TSS 位于内存中，GDT 中保存着指向TSS 的单元，在使用时从GDT中寻找。
TSS 由操作系统在初始化时填充，填充过程如下：
a) Allocate TSS memory
b) Init TSS
c) Fill TSS descriptor in GDT
d) Set TSS selector (设置TSS Register)

实验理解：

实验一：

我的理解就是告诉了我们ucore.img的生成过程，首先，生成ucore需要未make前kern和boot里面所有的.S和.C文件，对他们进行编译，编译生成俩文件bin/kernel和bin/block.out，当然了，为了生成规范的引导扇区还需要编译sign.c。最后先用sign规范化bootblock，然后为ucore.img分配5G的内存空间，并将bootblock复制到ucore.img的第一个block，紧接着将kernel复制到ucore.img第二个block开始的位置。

过程还可以，但是那些查找并拿出来所有的.C/.S文件以及生成新的bin/目录然后把新生成的东西的路径前面加上bin/这个处理路径过程的代码真的很艰涩，其实它并不是主干，但是确浪费了我大量的时间去看。

然后正规扇区的格式相对简单，就是读C代码。

那么综上所示（其实这是我做完实验三才明白的）：BIOS做的工作就是第一段做的工作，所谓的bootloader和bootblock可以理解为是一样的，bootloader就是由bootasm和bootmain组成的，只不过bootloader和bootblock一个是二进制代码一个是我们能看懂的反汇编代码，当然，初始化的时候总不能把汇编代码搞到磁盘上吧所以肯定要弄二进制文件。

其次，BIOS所做的就是先初始化设备和终端历程集并且把磁盘扇区的第一部分加载到了内存的7C00位置然后让bootasm开始进行实模式向保护模式的切换。然后在bootasm里面调用了bootmain

实验二：

通过阅读实验报告，我们知道：BIOS做完计算机硬件自检和初始化 后，会选择一个启动设备（例如软盘、硬盘、光盘等），并且读取该设备的第一扇区(即主引 导扇区或启动扇区)到内存一个特定的内存地址0x7c00处，然后CPU控制权会转移到那个地址继续 执行。至此BIOS的初始化工作做完了，进一步的工作交给了ucore的bootloader，而bootloader就是我们在实验二中对比的（bootmain和bootasm）而让我们调试的这个阶段就是BIOS结束后bootloader的工作

实验三：

这一部分的实验旨在让我们理解bootloader的执行过程，大体上包括实验要求的三部分

开启A20：通过两次交互64h和60h接口，传递相应参数，暂停键盘操作在等待缓冲区没有未进行的工作后进行设置

初始化全局描述符表：每一个段描述符是8字节也就是64位，那么全局描述符表又24个字节，也就是有3个段，分别是空段、数据段和代码段

切换为保护模式：将cr0寄存器的PE位置也就是最低位置为1开启保护模式

后续操作：启动完保护模式后需要通过一个长跳转指令来更新cs寄存器的基址。因为进入保护模式后，cpu切换置32位处理模式，以后的寻址便会基于我们刚刚设置的GDT表来进行。CPU会读取我们CS寄存器的值作为段选择子来按照上述方法寻址，然而此时CS寄存器的值为0，我们若不重新设置CS寄存器则会定位到空段描述符（GDT表的第一项），寻址时会导致错误。使用ljmp指令设CS寄存器的值，使其作为段选择子定位到我们刚刚设的GDT表中代码段的描述符。PROT_MODE_CSEG定义为0x8，即将0x8写入CS寄存器，并跳转到protcseg代码段。

我们可以注意到代码段最前面定义了PROT_MODE_CSGE和PROT_MODE_DSEG，分别被定义为0x8h和0x10h，这两个分别是代码段和数据段的选择子。

学习：地址转换规则（这一部分没啥用，是我把重要的地方截取出来放在实验报告里，避免第二天看第一天的实验报告看不懂）

[1] 分段地址转换：CPU把逻辑地址（由段选择子selector和段偏移offset组成）中的段选择子 的内容作为段描述符表的索引，找到表中对应的段描述符，然后把段描述符中保存的段基址 加上段偏移值，形成线性地址（Linear Address）。如果不启动分页存储管理机制，则线性地 址等于物理地址。

[2] 分页地址转换，这一步中把线性地址转换为物理地址。

练习四：

那么，硬盘的第一个扇区bootblock已经被载入内存了，后面的就是以elf文件格式存在的kernel，他保存了操作系统各个段的信息，我们需要根据kernel提供的信息加载各个段的信息到内存中。

所以说bootmain就包含了两部分：第一部分就是从硬盘里加载kern的内容写入内存。第二部分就是通过kern的内容（从内存0x10000开始读kern）逐步通过program header恢复相应段结构。

练习五：

复习一下大二的软件安全和汇编程序设计的知识。

最后想那个输出的时候确实是bootasm掉bootmain压了一次栈，然后变成了0x7c00-4-4=0x6bf8，然后bootmain调用kern_init，但是因为操作系统需要一直工作，所以OS没有返回，那么倒数第二个ebp也就永远不会pop，那么这个ebp指向的就是原来的bootmain的ebp指向的位置也就是0x6bf8，最后的参数就是一直往高地址去找就可以了，就是于是bootmain的高地址去找，本来应该有参数的，但因为bootmain的调用不需要参数所以也就找对应地址的16进制数据就可以了

实验六：

全面充分的介绍了一下中断的处理过程，我的理解是：检查中断是否发生----》取中断索引---》特权级转换------》保护现场----》执行中断处理指令-----》特权级转换-----》恢复现场----》errorcode处理-----》继续执行源程序

challenge1：

- 若CPU在内核态执行时进行中断，特权级并无变化，直接在内核栈进行中断处理，不涉及栈的变化，trapframe的SS、ESP不会由cpu自动压入、也不会弹出，不会使用。
- 若中断涉及特权级的变换，中断的执行也会进行栈的切换。在保护模式下，若CPU在用户态执行时产生了中断，由于中断处理程序是内核态，即CPL从3变为0，特权级进行了提升。内核栈的地址会被初始化在TSS的SS0、ESP0中，当从用户态切换到内核态时，CPU从TSS中得到SS0、ESP0，切换到内核栈，并会在内核栈压入用户栈的SS、ESP。
- 当处于内核态的中断处理程序执行完毕后，恢复现场，执行IRET指令中断返回，从内核栈中弹出用户程序中断点的cs时，特权级会从高特权级变为低特权级，CPL从0变为3，也即从内核态切换回用户态，同时会弹出开始保存在内核栈的用户栈SS、ESP，也即完成了从内核栈到用户栈的转换。