本文档用来介绍实现本工程所需的所有技术。

目录：

1.工程文件介绍

2.运行代码流程

3.一些重要的结构

4.调试技术

5.代码细节

1.工程文件介绍：

* 所有文件：

代码部分：boot\_ext2.asm loader\_ext2.asm kernel.asm

头文件 : load.inc pm.inc

脚本程序：boot.run load.run kernel.run

配置文件：bochsrc

软盘 ：a.img

* 文件内容：

代码boot,loader,kernel是我们操作系统的代码。

头文件load.inc和pm.inc中我们定义了一些常量和宏

脚本程序boot.run，load.run和kernel.run辅助我们编译和向软盘中写入文件

配置文件bochsrc是bochs虚拟机的配置文件

软盘a.img是我们的启动盘

2.运行代码流程

2.1.执行./boot.run，具体代码如下：

nasm boot\_ext2.asm -o boot\_ext2.bin

sleep 1

dd if=boot\_ext2.bin of=a.img bs=512 count=1 conv=notrunc

他负责把boot\_ext2.asm汇编成二进制文件，然后将其写入a.img的第一个扇区中。

这样，a.img就可以作为启动盘了。

2.2.执行./kernel.run

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

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

他负责把kernel.asm汇编成elf格式的文件，然后将其链接成入口地址为0x30400

的elf文件。（这个具体过程未深究，只用知道kernel.bin是elf格式的文件）

2.3.执行./load.run，具体代码如下：

nasm loader\_ext2.asm -o loader.bin

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

sudo cp loader.bin /mnt/floppy/ -v

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

sleep 1

sudo umount /mnt/floppy/

他负责把loader\_ext2.asm汇编成二进制文件loader.bin。再将a.img挂载到/mnt/floop目录下（这样才能将文件复制进软盘a.img）。随后便将loader.bin和kernel.bin复制进/mnt/floop中，即a.img中。最后将a.img卸载下来。(注意，此处是将loader拷贝到a.img的文件系统内，而2.1中的dd命令是直接将boot写入a.img的第一个扇区中)

2.4.执行bochs命令，运行bochs虚拟机。

3.一些重要的结构

* 在叙述代码功能，并详细解释之前我们还需要了解一些基本知识。首先我们必须知道ext2文件系统中的inode和目录项的结构（此处只叙述我们需要知道的成员，与本实验无关的暂不考虑）。
* inode结构

struct inode{

\_u32 unknow ;unknow代表我也不清楚此成员的含义，由于不需要

\_u32 size; ;所以我们也就不深究

\_u32 Creat\_T

\_u32 Modif\_T

\_u32 Access\_T

\_u32 Del\_T

\_u16 unknow

\_u16 link\_count

\_u32 Block\_count ;记载了block×2数

\_u32 unknow

\_u32 unknow

\_u32 Block[15] ;15个索引项，包括12个一级索引，2个二级，1个三级

\_u32 Generation

\_u32 unkown[6] ;代表有6个未知项，不是数组的意思

}

这是存放在ext2磁盘上的inode结构，位置也是一一对应。也就是说索引项block数组

是从偏移量为40的地方开始，block\_count距头部偏移为28。

我们需要的成员是：Block\_count和Block[15]。

最后我们可以看出一个inode占128字节。

* 目录项的结构就比较简单了

struct Entry\_Dir{

\_u32 inode ;inode节点号

\_u16 rec\_len ;本目录项的长度

\_u8 name\_len ;目录项名的长度

\_u8 file\_type ;文件类型，常规文件或目录或链接等

char name[] ;长度不固定。

}

从中可以看出目录项的长度是不固定的，所以需要两个字节来存放本目录项的长度

信息rec\_len。

* 最后我们需要清楚ext2文件系统的格式

图 3.1 ext2文件系统

如图3.1，这是一个1.44M大小的ext2文件系统结构，其InodeTable从第10个block开始，每个block1024字节，也就是占两个扇区，物理偏移为2800h。BlockTable同理。下面我们用一个ext2文件系统的真实情况来讲解一下ext2文件系统存储文件的方式。

图 3.2 ext2文件系统实例

如图3.2，通过根目录中的内容我们可以查找到loader.bin和kernel.bin的目录项，通过目录项结构我们可以知道，loader.bin的inode号为18号，目录项长度为18个字节，目录名长度为10个字节，文件类型为1号，也就是普通文件，文件名为loader.bin。同理可得到kernel.bin的结构。然后查找InodeTable，得到第18个inode节点的信息，与struct inode结合我们可以得到loader.bin有占2个扇区，block号为96（96是从boot的第0个开始算起），进而可以寻址到loader.bin的内容。同理可得到kernel的内容。

有了这些知识我们就可以加载存放在ext2类型的软盘中的文件loader.bin和

kernel.bin了。

4.调试技术

我们可以用的调试技术很简单，有两种方式，一种是通过bochs，一种是通过debugfs

* bochs基本调试命令：

b 0x7c00 打断点

blist 显示所有断点信息

info b 同上

del n 删除第n个断点

c 执行到断点处

n 单步执行，不跳入函数

s N 单步执行，跳入函数。N可选，表示执行N条指令

x /12xcb addr 按字符显示地址addr处的12个字节内存信息，

u /100 显示100行反汇编代码

* dubugfs调试：

将ext2类型的a.img挂载上后，使用df查看挂载点，再使用debugfs+挂载点 （如：

Sudo debugfs /dev/loop0）打开此文件系统。

Ls 显示文件及其inode节点信息。

stat <12> 显示第12个inode节点信息

mi <12> 同上

* 最后一个可能会有帮助的命令是：

xxd -u -a -g l -c 16 -s +0x2800 -l 512 a.img

这个命令可以用来查看镜像内容

++ 查看文件系统信息.将文件系统挂载上之后，用df命令查看挂载点，用dumpe2fs查看ext2文件系统信息。

5.代码细节

最后我们来看看代码实现。在开始之前，我们先整体看一下代码流程，再进入函数仔细分析代码。（下文所说的boot就代表boot\_ext2.asm或其二进制文件。loader，kernel同理）

代码流程：

* boot负责把loader加载到内存9000h:100h处，然后jmp到loader处执行loader的代码。
* loader将kernel加载到8000h:0000处。然后进入保护模式，在保护模式下设置段寄存器，启动分页机制（call SetupPaging），装载8000h:0000处ELF格式的kernel（call InitKernel）（由于kernel是elf格式的，所以不能直接jmp到8000h:0000处执行，要按照elf格式将其装载到正确地址处，再jmp过去才能正确执行我们这里将其装载到物理内存的30400h处）。 好了现在我们已经把内核加载到了正确位置，我们可以将控制权交给kernle了：

jmp SelectorCodeKernel:KernelEntryPointPhyAddr

这个jmp跳到了3G+30400处。为什么执行kernel代码要跳到这里呢？我们明明只把它装载到30400h了，为什么加上3G也能正确运行呢？这就有劳我们loader.bin中的分页机制了。看了SetupPaging自会理解。

* 最后kernel仅输出一个K字符便进入死循环。

代码细节：

* boot\_ext2.asm代码详解：

boot的代码逻辑很简单，主要分为两部分：第一部分加载根目录文件，并在其中找到loader目录项。找到目录项后就是第二步，加载其文件内容。找loader的过程逻辑上可分为三部：

1.将根目录的block读入内存

2.比较目录项：先比较NameLen是否为10,如果是，继续比较文件名，如果否，继续下一个目录项的比较

3.如果比较了32个目录下都没有loader.bin，则输出:No Loader

在第二步中如果比较文件名后确定是loader.bin，则跳到LABEL\_FILENAME\_FOUND,执行加载文件内容。加载过程也分为三步：

1.计算loader.bin的inode的地址

2.将inode所在扇区读入8f00:0000处内存

3.循环读入block数组，即loader.bin的内容。

加载完loader.bin后便跳入loader的代码中执行loader程序。至此boot结束

* loader\_ext2.asm详解：

loader的过程上文已经叙述，现在来详细说明。

1.首先是加载kernel。此过程与boot加载loader过程基本一样，再次不再叙述。

2.然后就可以进入保护模式了。进入的过程比较简单，此处略过。

3.进入保护模式后由于寻址方式不同，所以要从新设置段寄存器。

4.启动分页机制。我们把页目录表放在1M--（1M+4k-1）处，页表紧随其后。我们需要把0-1G和3G-4G的线性地址空间映射到0-1G的物理内存（我们使用的内存就是1G大小）。由于一个页目录表便可对应1G的空间，所以我们使前256个页目录项和后256个页目录项都指向代表前1G物理内存的页表，这样便实现了我们上面的目标。读者可以对照代码自己分析一遍。

5.装载ELF格式的kernel。按照ELF的ProgramHeaderTable来加载各个段的内容，

ProgramHeader会指定某段应该加载到什么位置的内存。有关这部分详细信息可以

参考《一个操作系统的实现》第5.3节和5.4.3节。

6.通过jmp我们便可以跳入kernel了。看到这里读者应该能够理解loader最后jmp

到3G+30400h处为何能正确执行了，因为我们线性地址空间的0-1G和3G-4G对应的

是相同的物理空间，也就是这两部分的线性地址会转化成相同的物理地址。