开发操作系统漫游

目录

环境准备

linux操作系统

linux操作系统对于CS是绕不过去的一环,非常推荐大家使用,甚至是作为主力系统使用！
如果不会安装linux系统,那正好是一个动手学习的机会.在使用linux的过程中,或许会让你留下痛苦的回忆,但它们都会化为财富.
以下的示例全部使用ubuntu16.04为例.

gnu系列工具

ps:以下工具不需要专门安装,使用ubuntu自带的版本

gcc(**GNU Compiler Collection**) 是gnu开发的编程语言编译器,现在可以处理多种编程语言.我们需要它来完成对C语言以及部分汇编的编译工作.
ubuntu16.04是自带版本为5.4.0
使用以下命令查看版本：

```
gcc --version
```

nm命令可以显示符号文件的符号,可以对一个gcc编译出来的可执行文件(.out/.exe)或者可重定向文件(.o)使用看一下.如果要研究链接的过程nm会起到很大的作用.

gas是gnu系列的汇编器,我们主要使用nasm所以一般不用它,但是涉及到gcc和nasm联合编译时,可能会研究c的代码行为时,需要汇编出来研究一下.为了结合nasm,我们手动指定格式为intel,即加上`-masm=intel`.

ld是链接命令,建议认真了解一下,也就是把多个可重定向文件打包在一起,可以打包成可执行文件,也可以打包成链接库.

objcopy命令把一种目标文件中的内容复制到另一种类型的目标文件中.我们主要在减小文件大小时使用-S参数去掉一些东西.

readelf可以查看elf格式的文件内容

qemubochs

qemu是一个开源仿真器,用于模拟一台计算机,可以用来运行我们编写的操作系统.
使用以下命令进行安装：

```
sudo apt-get install qemu
```

bochs也是一个开源模拟器,可以模拟各种硬件的配置,不过需要一份配置文件.不过它的界面与调试比qemu上都很好用.
使用以下命令安装

```
sudo apt-get install bochs vgabios bochs-x bochsbios bochs-doc xorg-dev bochs-sdl bximage
```

bximage可以生成一个.img文件,生成的是一个全0的空文件.可以制定大小,格式(硬盘/软盘). 我们后面会用到

nasm

nasm为一种汇编器,用来编译汇编代码.常用的参数为`-f 格式` 比如bin,elf,优化等级 (-O0,-O1等)
gcc自带的gas对初学者可能更难一些.

前置知识

浅谈

操作系统可能是CS里最具有挑战性的一门课程了,在这门课程真的写一个操作系统还是非常困难的,但总是写一些东西总还是可以的.写这个东西一是为了在总结中进步,二是以后说不定也可以给别人做参考.
本次环境的选用在一定程度上参考了清华学堂在线课程的配置,也作出了一些变化

- 开发环境为：qemu+make+gcc+nasm+ubuntu16.04.
- 面向intel-i386开发,汇编使用intel格式

要完成一个操作系统,需要关注的知识点太多了.鉴于篇幅,个人的精力与能力,很多知识都只能简单的点一下,有些我也不懂,也是在一边学习一边写.

linux下常用的命令

我们对操作系统的控制的方式有两种,一种是常用的GUI式的（图形化界面）,另外一种是CLI（命令行）.一般我们将输入命令的地方称作shell,Terminal,终端,命令提示符.
无论是在windows下,或者linux,乃至OS X都是有命令行的,不过在具体的命令下有些差别的.这里只介绍最常用/必须要用到的命令.

```
ls      展示当前目录下的文件与目录
ls -a   展示所有文件（包括隐藏文件/文件夹)
ls -l   展示更详细的信息
```

```
cp  复制文件命令
mv  移动文件命令
rm  删除文件命令
cat 输出文件内容到终端
```

```
cd    切换目录命令
locate
find  查找文件命令
```

```
dd   用指定大小的块拷贝一个文件,并在拷贝的同时进行指定的转换.
```

```
mount 分区挂载命令
umount  取消挂载命令
```

具体的参数与使用方式,可以自行探索/在网上搜索

初学者可能要遇到的几个问题：

linux的权限问题
linux下的路径与windows是有区别的
linux安装软件的方式（在ubuntu下试用apt-get）

gcc的编译过程中发生了什么

gcc的编译过程

使用以下代码作为示例.命名为hello.c

```
//"hello.c"
#include<stdio.h>
int main(){
    printf("hello world"); //注释
    return 0;
}
```

打开终端,将位置切换至hello.c所在的文件夹.
使用以下命令

```
gcc hello.c
```

在这个文件夹下会生成a.out文件
终端内执行

```
./a.out
```

即可看到hello world的输出.

a.out为没有指定输出结果文件名时候的默认名,使用-o 可以指定文件名.以下命令可以看到生成了hello.out文件

```
gcc hello.c -o hello.out
```

---

gcc的编译过程整体上分为 预处理,编译,汇编,链接.

我们在调用gcc的时候,实际上是一步步完成,然后再删除了中间结果.

预处理的过程是除去注释以及展开宏.
头文件的引用就是在展开宏中完成的.

以下两个命令都会将预处理的结果输出到终端上.

```
gcc  -E hello.c
cpp hello.c
```

使用-o 参数指定输出文件名

```
gcc -E hello.c -o hello.i
```

编译的过程是将预处理好的文件编译为汇编文件.
使用以下命令会生成一个hello.s的文件

```
gcc -S hello.i
```

gcc默认会使用AT&T格式的汇编,可以通过添加参数-masm=intel来指定为intel格式

汇编的过程是将汇编转化为二进制文件,不过此时的二进制文件还没有办法运行.详细参考后面多个文件连编的内容.
执行以下命令,会生成一个hello.o文件.又叫做可重定位文件.

```
gcc -c hello.s
as -o hello.o
```

链接之后就会生成可执行文件

```
gcc hello.o
```

使用-save-temps 参数可以保留中间的各种结果

```
gcc -save-temps helloworld.c
```

多个文件联合编译

有以下两个文件,分别命名为main.c与t1.c

```
//main.c
#include<stdio.h>
int main(){
    int a=3,b=4;
    printf("%d",add(a,b));
    return 0;
}
```

```
//t1.c
int add(int a,int b){
    return a+b;
}
```

使用以下两种命令编译的结果是相同的,执行./main.out会输出7

```
gcc main.c t1.c -o main.out
```

```
gcc -c main.c
gcc -c t1.c
gcc main.o t1.o -o main.out
```

第一种实际上是gcc帮我们做好了各种工作,第二个更能体现实际过程.多个源代码文件联合编译的过程实际上是各自生成他们的可重定位文件(.o文件) ,然后再将他们结合在一起.结合在一起的过程称作链接.我们调用的一些库函数(如printf)实际上也是有生成好的可重定位文件.

这方面还可以作更深入的了解,如有兴趣可以围绕下面学习.

elf头与符号表
链接的具体过程
静态链接与动态链接

关于汇编

汇编已经非常底层,毕竟对于我们来说,又不会自己造硬件.所以汇编已经是我们写的最底层的代码了.
关于汇编,随便说几个名字,或许你就会感觉很乱.
arm64,i386,arm,intel,amd,AT&T,x86,x86_64,IA-64,MIPS,RISC;
当然上述的东西有些对代码的完成并不影响,但对这些基本概念还是要有认识的.

1. CPU依靠指令来计算和控制系统,每款CPU在设计时就规定了一系列与其硬件电路相配合的指令系统.指令可以说就是机器语言.
2. 指令集就是处理器支持的指令的集合.常见的有arm指令集,x86指令集.它们也都有各自的版本.
3. 指令集架构(ISA)就像是特定处理器的设计图纸,规定了该类处理器支持哪些机器指令（指令集）、寄存器有哪些状态以及输入输出模型.它决定了处理器的实现.
   常见的有CISC,RISC.一种是复杂指令集架构,一种是精简指令集架构.
4. 处理器可能支持多种指令集.
5. 微架构又称为微体系结构/微处理器体系结构.是具体的对一种指令集架构的实现方式.比如X86架构,amd64,IA64,Core(酷睿)等
6. 汇编语言是对机器语言的一种助记符.应用程序在切换硬件平台时,如果有汇编语言就需要重写.因为不同的硬件平台的支持的指令集是不同的.同一系列的指令集基本都是向前兼容的,升级后就也就是加入一些新的指令.
7. 汇编的书写格式分为intel格式与AT&T格式,gcc默认支持后者.这两种的差别可以查阅资料.

我们在写汇编代码的时候只需要关注希望运行平台支持的指令集就好了.
我们使用的就是intel汇编格式+面向i386处理器使用的指令集.

make与Makefile

想一想gcc编译c语言代码的过程,针对小型项目,手动逐步编译倒也没有问题。但整个工程的架构逐渐变得复杂,数不清的源文件,又放在不同的目录中。就不能每次手动编译了。所以需要一个自动化编译的东西,这个东西就是make。对makefile做一个简单的讲述。

在项目的根目录下建立一个名为Makefile的文件写好相关内容,然后你只需要在终端中输入make命令,即可完成。

make的核心分为两个部分,即依赖关系与命令。

简单地说,就是会建立目标文件到若干依赖文件的依赖,当出现以下情况时,执行制定的命令。

1. 目标文件不存在
2. 依赖文件的更新时间晚于目标文件
3. 目标文件没有依赖文件

下面的是例子：

```
main.out: main.cpp
    g++ main.cpp -o main.out
```

ps：命令有一个缩进,表示从属于上面的依赖关系。
如上的一个makefile文件,简单的说就是,当main.out不存在或者main.cpp比main.out新时执行下面的命令,重新生成了main.out。
怎么理解“新”呢：当你编译main.cpp 后,生成了main.out,main.out就比cpp新；然后你修改了main.cpp,main.cpp就比main.out新,新就证明被修改过。

makefile的第一个依赖关系的目标文件是整个makefile的目标文件,用上面写过的代码为例子

```
//main.c
#include<stdio.h>
int add(int x,int y);
int main(){
    int a=3,b=4;
    printf("%d",add(a,b));
    return 0;
}
```

```
//t1.c
int add(int x,int y){
    return x+y;
}
```

他的makefile可以写成（假设最终结果为main.out

```
main.out: main.o t1.o
    gcc main.o add.o -o main.out
main.o: main.c
    gcc -c main.c
t1.o: t1.c
    gcc -c t1.c
```

可以看出来,依赖可以是个树形结构,当一个文件被更新后,会逐层的更新相关的文件。第一个依赖关系是整个文件的目标文件。整个文件的目标文件的意思是执行make命令时的目标文件,也可以手动指定目标文件,比如： make main.o。不过这一点大多数时候是用来做以下情况的。

```
clean:
    -rm -f  main.o t1.o main.out
```

ps：以上这段为上面的补充;减号的作用是发生错误时Makefile继续。
这样就可以执行make clean 来删除生成的文件了。

当然不仅仅是clean了,还有更多的用途比如写一个debug,自动执行debug的命令。

makefile还支持变量的使用来简化书写。并且可以在执行make的时候,手动指定变量的值。

如

```
target:= main.o t1.o
main.out: main.o t1.o
    gcc $(target) -o main.out
main.o: main.c
    gcc -c main.c
t1.o: t1.c
    gcc -c t1.c

.PHONY:clean
clean:
    -rm -f  $(target) main.out
```

makefile支持一隐含依赖规则。篇幅太长,不适合在这里说明。

伪目标的作用可以去了解一下,也不做说明。
当你看到如 .PHONY:clean 这样的东西要明白这是伪目标

makefile中还支持函数,我认为是make中比较难的一部分。

关于make,更多的规则可以自己在网上学习。
更多可以参考《跟我一起写makefile》一文,里面讲述的很全面。

调试技能

我们主要使用gdb与bochs自带的调试功能.

gdb

gdb是一个调试工具,比起很多ide提供的调试功能有些简陋。但是它麻雀虽小五脏俱全让人使用之后爱不释手。

事实上,我们只用其中的一小部分功能,就足够了。

```
//"hello.c"
#include<stdio.h>
int main(){
    printf("hello world"); //注释
    return 0;
}
```

加入-g参数编译,使用gdb开始调试

```
gcc -g hello.c -o hello.out
gdb -q hello.out
```

gdb 的-q参数意思是不输出Copyright,不然会在进入的时候有一大段Copyright,-x可以读取一个文件写好的命令

命令	作用	示例
r	Run的缩写,如果此前没有下过断点,则执行完整个程序,否则暂停在第一个断点处	
c	Continue的缩写,继续执行到下一个断点或程序结束。	
b	设置断点。可以使用行号,函数名称,地址等方式指定断点位置	
d	指定编号的某个断点或所有断点	
s,n,si,ni	单步执行,s进入函数内部,n不进入；si,ni汇编层次	

命令	作用	示例
p	print的缩写,显示指定变量的值。	
display	设置一些自动显示的变量,当程序停住时,或是在你单步跟踪时,这些变量会自动显示	
l	list的缩写查看源代码	
quit	退出	
help [命令名称]	帮助命令	
bt	打开函数调用堆栈	

具体使用不详细介绍了,说一些重要的点

按下ctrl+x+a可以打开一个关键窗口,可以打开一个代码的显示窗口
调试的核心就是断点+单步执行+查看
print于和display非常有用,可以指定格式输出

查看汇编的时候,我经常使用的3条:指定汇编文件格式为intel/查看汇编代码/查看寄存器

```
set disassembly-flavor intel
layout asm
layout regs
```

下面说一说如何用gdb调试我们的操作系统,这个要配合qemu一起使用

使用qemu-system-i386启动img文件,加上两个参数

```
-S 一启动就会暂停
-s 监听tcp:1234端口等待Ggdb的连接
```

然后在打开gdb,使用`target remote :1234`来连接就可以. 参考各个pro里各自的情况,有些我也还没弄明白.

bochs的调试技能

暂略(还没总结出来)

磁盘基础知识介绍

base

整个硬盘上一般有很多的盘片组成，
每个盘片如同切西瓜一样被“切”成一块一块的扇面，同时沿着半径的方向被划分成了很多同心圆，就是传说中的磁道，每条磁道被扇面切成很多的扇形区域叫做扇区（扇区是从磁盘读出和写入信息的最小单位，通常大小为512字节），不同盘片上的同半径磁道组成了柱面，这些都是磁盘物理上的概念，知道便可。有了这些概念，我帮便可以计算磁盘的容量：

磁头数 × 磁道(柱面)数 × 每道扇区数 × 每扇区字节数

磁头（head）数：每个盘片一般有上下两面，分别对应1个磁头，共2个磁头；
磁道（track）数：磁道是从盘片外圈往内圈编号0磁道，1磁道...，靠近主轴的同心圆用于停靠磁头，不存储数据；
柱面（cylinder）数：同磁道数量；
扇区（sector）数：每个磁道都别切分成很多扇形区域，每道的扇区数量相同；
圆盘（platter）数：就是盘片的数量。

数据定位

CHS(3D)
扇区（sector），柱面（cylinder），磁头（head），对早期的磁盘（上图所示）非常有效，知道用哪个磁头，读取哪个柱面上的第几扇区就OK了。
CHS模式支持的硬盘容量有限，用8bit来存储磁头地址，用10bit来存储柱面地址，用6bit来存储扇区地址，而一个扇区共有512Byte，这样使用CHS寻址一块硬盘最大容量为256 * 1024 * 63 * 512B = 8064 MB(1MB = 1048576B)（若按1MB=1000000B来算就是8.4GB）

LBA(Logical Block Addressing)
但现在很多硬盘采用同密度盘片，意味着内外磁道上的扇区数量不同，扇区数量增加，容量增加，3D很难定位寻址.
在LBA地址中，地址不再表示实际硬盘的实际物理地址（柱面、磁头和扇区）。LBA编址方式将CHS这种三维寻址方式转变为一维的线性寻址，它把硬盘所有的物理扇区的C/H/S编号通过一定

的规则转变为一线性的编号，系统效率得到大大提高，避免了烦琐的磁头/柱面/扇区的寻址方式。在访问硬盘时，由硬盘控制器再将这种逻辑地址转换为实际硬盘的物理地址。

LBA下的编号，扇区编号是从0开始。

逻辑扇区号LBA的公式：
LBA(逻辑扇区号)=磁头数 × 每磁道扇区数 × 当前所在柱面号 + 每磁道扇区数 × 当前所在磁头号 + 当前所在扇区号 − 1

例如：CHS=0/0/1，则根据公式LBA=255 × 63 × 0 + 63 × 0 + 1 − 1= 0
也就是说物理0柱面0磁头1扇区，是逻辑0扇区。

从机器启动到操作系统的启动

整个的启动过程大致可以分为以下几个阶段：

1. 上电后,机器进行初始化
2. 启动bios,硬件自检后,选择启动设备
3. 读取主引导记录,然后加载bootloader
4. bootloader加载内核

我们所需要关注的是后两个阶段.

i386的实模式与保护模式

i386的cpu支持三种运行模式,分别为16位的实模式,32位的保护模式,虚拟8086模式(V86模式).

最后一种模式已经完全过时了,不用讲述,主要要说明的就是前两者.

8086具有16位的寄存器长度,我们称它为16位的处理器。它利用20位的地址总线+分段的方式,可以访问1MB的内存.但是只能分段进行,段的长度最大只能是64kb.8086只有一种工作模式.

从80286开始引入了保护模式的概念,虽然80286是16位处理器+24位地址总线.但在保护模式下,段寄存器中保存的不再是段地址,而是段选择子,真正的段地址位于段寄存器的描述符高速缓存中,是24位的。因此可以访问全部16MB内存。段选择子会在后面再提.

80386 处理器的寄存器是 32 位的,而且拥有 32 根地址线,可以访问 2^32=44294967296字节的内存,4294967296/1024/1024=4GB.

另外保护模式下,应用程序对内存的访问完全没有限制,这放在今天是肯定不行的.现在可以说是多用户多任务的时代,必须对用户的程序加以隔离.而保护模式的保护就是指这一点.

在CPU刚加电时,自动处于实模式下.需要专门设置才能运行在保护模式下。这个需要汇编好搞.

保护模式的好处:

- 确保应用程序无法对操作系统进行破坏
- 所有的地址总线都可以寻址,物理寻址空间大
- 支持内存分页机制,提供了对虚拟内存的良好支持。
- 保护模式下80386支持多任务,还支持优先级机制,不同的程序可以运行在不同的特权级上

具体保护模式的种种,相关的内容会在后面再谈.

ps:80x86的模式区分是为了向前兼容的历史遗留问题,而arm处理器就没有实模式.

bios与分区

bios的全称是,基本输入输出系统,Basic Input/Output System
在机器上电后,第一件事就是读取bios.

1. 首先是硬件自检,检查机器是否能启动
2. 自检完成后,会把控制权交给下个阶段的启动程序.
   这个启动程序又在哪?经常装系统的同学就不陌生了,想一想用u盘/cd重装系统时,要修改启动项的顺序/选择单次的启动设备.
3. bios会读取选择设备的第一个扇区,即前512字节,这512字节叫做`主引导记录`(MBR).
   先检查第511个字节是否是0x55,第512个字节是否是0xAA.如果是则认为这个设备是一个可用于启动的设备,否则选择下一个设备.这最后两个字节是约定好的.在写代码的时候要关注端序的问题,intelx86使用的是小端序.
   大小端序是怎么回事,我就不用说了吧.
4. 这512字节要分为3个部分:
   - 第1-446字节：调用操作系统的机器码。
   - 第447-510字节：分区表（Partition table）
   - 第511-512字节：主引导记录签名（0x55和0xAA）
     分区表里面记录了分区的情况,总共只能有4个主分区,每个主分区花16字节信息.

实际上主引导记录的划分早就不是这么简单了,更多的可以关注wikiMBR

5. 接下来的过程就有很多种方案了,我们使用的是最自主的方案,使用bootloader程序来加载内核.
   bootloader程序是我们自己写的一段程序,如何读取这段程序会在下一部分介绍.
   关于其他方案,有兴趣的可以自行搜索了解.关键词 `扩展分区与逻辑分区` , `卷引导记录` 等.
   可以参考阮一峰老师的博客计算机是如何启动的？

总结一下:机器上电读取bios->bios控制进行硬件自检->对选择的启动项进行检查(511,512字节的内容)判断是否是一个合适的主引导记录->将这个MBR读取到内存中,然后加载bootloader,交出控制权给bootloader程序.

下一部分将说明如何交出控制权.

与bios相对应的还有一种方案为uefi模式,它是与bios相对应的.因为兼容问题,原来的bios模式就比变成了bios的传统模式(legacy),uefi为新模式.

这有点像保护模式出现前只有实模式,保护模式出现后实模式才开始叫做实模式,实际上它与原来的方案是彻底不同的方案.

mbr与bootloader的工作

我们使用mbr来加载bootloader,再用bootloader来加载内核的原因是mbr可以用的太小了,实际上只有446个字节可以使用.我们可以在mbr里跑一些程序,可太复杂的就放不下了.在前几个pro里这两个文件便合为一体.

mbr加载bootloader

现在来说一个约定就是0x7c00这个地址.
bios在通过MBR的验证后,会把这份MBR的代码读取到0x7c00处,然后jmp过去.
所以我们的代码也要围绕这个地址做出相应的调整.
具体怎么调整当然是代码中见.

关于0x7c00这个地址,其实也是有历史原因的,可以参考阮一峰老师的博客为什么主引导记录的内存地址是0x7C00？

mbr是如何做加载工作的,这个和bootloader加载内核的方式基本一样,所以在后面的篇章说明.

bootloader的任务

bootloader的主要工作是进入保护模式+加载内核.
实模式只有很小的寻址能力+没有做出隔离,所以保护模式是必要的.然后便是加载内核了.

保护模式

现在要来谈一谈如何进入保护模式了.在介绍GDT的过程中会涉及到一些保护模式各种机制.

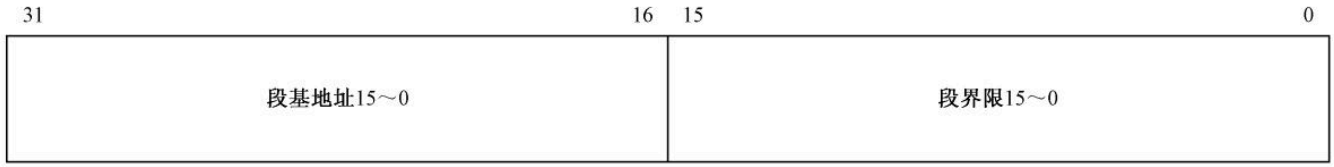
进入保护模式要做的事情大概有: 完成全局描述符表,打开A20线,刷新流水线(jmp到32位代码区),重新指定ds等寄存器里的值.

全局描述符表(GDT)

首先是寻址方式,保护模式中的寻址方式同样支持`段地址:偏移地址`的形式,但是段地址上发生了变化.原来的段寄存器变成段选择子.段选择子是一张表的索引,这张表叫做全局描述符表.表里面的每一项叫做一个描述符,描述符里详细定义了段的各种信息,比如起始地址,界限,特权级等.保护模式的种种都是从GDT开始的.

具体一个段选择子的格式如下(i386中):

上面为高32位,下面是低32位.



一个描述符总共64位,也就是8个字节,用16进制表示起来是16位.
其中段基地址和段界限有些散乱的原因是80286就已经支持保护模式了,而它是16位寄存器+24位地址总线的结构.后续的cpu为了为了向前兼容所以成了这个样子.

关于部分位具体介绍如下:(我现在也不是全部明白,有些暂时用不到).大部分内容来自于从实模式到保护模式一书.

段基地址总共占据32位,表示一个段开始的位置,32位可以表示的内存大小为4GB.(2**32/1024/

1024/1024==4)
段界限实际上就是长度,总共是20位的长度.单位与高32位中第23位的G有关系.

G是粒度位
如果G位是0时,段界限以字节为单位,此时,段的扩展范围是从1字节到1兆字节(1B~1MB);
如果该位是1,段界限是以4KB为单位的。段的扩展范围是从4KB到4GB。

S位用于指定描述符的类型。
为0,表示是一个系统段;
为1,表示是一个代码段或者数据段或者堆栈段.

DPL 表示描述符的特权级(Descriptor Privilege Level,DPL)
。这两位用于指定段的特权级。共有 4 种处理器支持的特权级别,分别是 0、1、2、3,其中 0 是最高特权级别,3 是最低特权级别。

刚进入保护模式时执行的代码具有最高特权级 0 (可以看成是从处理器那里继承来的),这些代码通常都是操作系统代码,因此它的特权级别最高。每当操作系统加载一个用户程序时,它通常都会指定一个稍低的特权级,比如 3 特权级。不同特权级别的程序是互相隔离的,其互访是严格限制的,而且有些处理器指令(特权指令)只能由 0 特权级的程序来执行,为的就是安全。

P是段存在位(Segment Present)。P 位用于指示描述符所对应的段是否存在。一般来说,描述符所指示的段都位于内存中。但是,当内存空间紧张时,有可能只是建立了描述符,而没有内存空间,也有可能后面被换了出去.

D/B 位是“默认的操作数大小”(Default Operation Size)或者“默认的堆栈指针大小”(Default Stack Pointer Size).设立该标志位,主要是为了能够在32位处理器上兼容运行 16 位保护模式的程序。0表示16位,1表示32位.

L 位是 64 位代码段标志(64-bit Code Segment),保留此位给 64 位处理器使用。

AVL 是软件可以使用的位(Available),通常由操作系统来用,处理器并不使用它。

TYPE 字段共 4 位,用于指示描述符的子类型,或者说是类别。如表 11-1 所示,对于数据段来说,这 4 位分别是 X、E、W、A 位;而对于代码段来说,这 4 位则分别是 X、C、R、A 位.

x位是可执行位,数据段为0,代码段位1.
E位指示段的扩展方向。E=0 是向上扩展的,也就是向高地址方向扩展的,是普通的数据段;E=1 是向下扩展的,也就是向低地址方向扩展的,通常是堆栈段。W 位指示段的读写属性,或者说段是否

可写,W=0 的段是不允许写入的,否则会引发处理器异常中断;W=1的段是可以正常写入的。

C 位指示段是否为特权级依从,C=0 表示非依从的代码段,
这样的代码段可以从与它特权级相同的代码段调用,或者通过门调用;C=1 表示允许从低特权级的程序转移到该段执行。
R 位指示代码段是否允许读出。代码段总是可以执行的,但是,为了防止程序被破坏,它是不能写入的。至于是否有读出的可能,由R位指定。R=0 表示不能读出,如果企图去读一个 R=0 的代码段,会引发处理器异常中断;如果 R=1,则代码段是可以读出的,即可以把这个段的内容当成 ROM 一样使用。R属性并非用来限制处理器,而是用来限制程序和指令的行为。

数据段和代码段的 A 位是已访问(Accessed)位,用于指示它所指向的段最近是否被访问过。
在描述符创建的时候,应该清零。之后,每当该段被访问时,处理器自动将该位置“1”。对该位的清零是由软件(操作系统)负责的,通过定期监视该位的状态,就可以统计出该段的使用频率。当内存空间紧张时,可以把不经常使用的段退避到硬盘上,从而实现虚拟内存管理。

参考如下:

表 4-3 代码段和数据段描述符类型

类型（TYPE）字段					描述符类型	说明
十进制	位 11	位 10	位 9	位 8		
		E	W	A		
0	0	0	0	0	数据	只读
1	0	0	0	1	数据	只读，已访问
2	0	0	1	0	数据	可读/写
3	0	0	1	1	数据	可读/写，已访问
4	0	1	0	0	数据	向下扩展，只读
5	0	1	0	1	数据	向下扩展，只读，已访问
6	0	1	1	0	数据	向下扩展，可读/写
7	0	1	1	1	数据	向下扩展，可读/写，已访问
		C	R	A		
8	1	0	0	0	代码	仅执行
9	1	0	0	1	代码	仅执行，已访问
10	1	0	1	0	代码	执行/可读
11	1	0	1	1	代码	执行/可读，已访问
12	1	1	0	0	代码	一致性段，仅执行
13	1	1	0	1	代码	一致性段，仅执行，已访问
14	1	1	1	0	代码	一致性段，执行/可读
15	1	1	1	1	代码	一致性段，执行/可读，已访问

然后我们需要完成建立gdt的工作,有一个约定是第一个描述符为空描述符.这个是为什么,我还没有搞清楚.

然后我们需要尝试建立一些描述符.

比如我们尝试建立下面这个段

```
线性基地址为 0x000B8000;
段界限为 0x07FFF;
粒度为字节(G=0),即该段的长度为 64KB。
属于存储器的段(S=1);
这是一个 32 位的段(D=1);
该段目前位于内存中(P=1);
段的特权级为 0(DPL=00);
这是一个可读可写、向上扩展的数据段(TYPE=0010)
```

这个段我们之后会用到,因为这里在内存中是控制文字模式限制的位置.

所以他的低32位应该是0x80007fff,高32位是0x0040920B.

实际上大多数位我们短期都不需要调整.

GDTR是专门为gdt服务的寄存器,指向gdt保存的位置.
LGDT指令将gdt的入口地址装入此寄存器

实际上后面有很多东西的设计思路都是类似与gdt,就是有一个寄存器来保存表的入口,然后在相应的地方保存表.比如分页机制,中断机制.

打开A20总线:
a20总线的打开,又是因为历史原因.8086是16位寄存器+20位地址总线.在16位程序时代,部分程序员利用会0xFFFFF+1的溢出变为了0x00000.在后面的时代总线数超过了20,为了兼容这些程序.故意做了一个设置就是必须打开a20总线,才能使用更多的总线数.方法是把0x92端口的1位设置位1.

```
    in al,0x92
    or al,00000010B
    out 0x92,al
```

CR0寄存器:
CR0 是 32 位的寄存器,包含了一系列用于控制处理器操作模式和运行状态的标志位。他的0位就是保护模式,实模式选择位.1代表保护模式,0代表实模式.设置一下就好.

```
mov eax,cr0
or eax,1
mov cr0,eax
```

cs等寄存器的刷新
因为进入实模式后,内存访问便是段选择子:偏移地址的形式.所以便要修改cs里面的值.

相关的代码内容查看pro2的相关文档,pro2完成了一个从实模式到保护模式的切换,并且直接控制显存输出的项目.
ps:在进入实模式之后bios的中断已经不能使用了!

内核的加载

在读取内核与bootloader的过程中,又与硬盘打交道,相应的代码与这些也是相关的.

这部分参考<<ORANGES_一个操作系统的实现>>一书,选用了软盘1.44MB+fat12文件系统来进行.建立软盘需要bximage的配合.

还有一个是bios 13h的中断可以用读取磁盘

fat12

FAT12是DOS时代就开始使用的文件系统（File System),直到2009年仍然在软盘上使用。

引导扇区是整个软盘的第0个扇区,我们用一个数据结构来描述BPB(BIOS Parameter Block).然后是两个完全相同的fat表,称呼为fat1,fat2.然后是根目录区和数据区.如图:

BS开头的不属于BPB数据结构

名称	偏移	长度	内容	软盘参考值
BS_jmpBoot	0	3		jmp LABEL_STA nop
BS_OEMName	3	8	厂商名	'ForrestY'
BPB_BytsPerSec	11	2	每扇区字节数	0x200（即十进制
BPB_SecPerClus	13	1	每簇扇区数	0x01
BPB_RsvdSecCnt	14	2	Boot记录占用多少扇区	0x01
BPB_NumFATs	16	1	共有多少FAT表	0x02
BPB_RootEntCnt	17	2	根目录文件数最大值	0xE0 （224）
BPB_TotSec16	19	2	扇区总数	0xB40（2880）
BPB_Media	21	1	介质描述符	0xF0
BPB_FATSz16	22	2	每FAT扇区数	0x09
BPB_SecPerTrk	24	2	每磁道扇区数	0x12
BPB_NumHeads	26	2	磁头数	0x02
BPB_HiddSec	28	4	隐藏扇区数	0
BPB_TotSec32	32	4	如果BPB_TotSec16是0,由这个值记录扇区数	0xB40（2880）
BS_DrvNum	36	1	中断13的驱动器号	0
BS_Reserved1	37	1	未使用	0
BS_BootSig	38	1	扩展引导标记	0x29
BS_VolD	39	4	卷序列号	0
BS_VolLab	43	11	卷标	'OrangeS0.02'
BS_FileSysType	54	8	文件系统类型	'FAT12'

名称	偏移	长度	内容	软盘参考值
引导代码	62	448	引导代码、数据及其他填充字符等	
结束标志	510	2		0xAA55

根目录区

根目录区开始的扇区是18,条目最大为BPB_RootEntCnt.根目录区大小不固定.然后根目录区的每个条目占据32字节,格式如下.

名称	偏移	长度	描述
DIR_Name	0	0xB	文件名8字节,扩展名3字节
DIR_Attr	0xB	1	文件属性
保留	0xC	10	
DIR_WrtTime	0x16	2	最后修改时间
DIR_WrtDate	0x18	2	最后修改日期
DIR_FstClus	0x1A	2	此条目对应的开始簇号(数据区&fat表)
DIR_FileSize	0x1C	4	文件大小

这样就可以看到每个文件的名称,属性,大小,磁盘中的位置.

1. 在fat12中,一个簇只包含一个扇区,所以 簇==扇区
2. 数据区的第一个簇号是2,而不是0/1
3. 根目录区开始区号==19
4. 数据区开始区号==根目录区+根目录区大小
5. 根目录区大小==(BPB_RootEntCnt*32)+(BPB_BytesPerSec-1)/BPB_BytesPerSec.

关于不在目录下的情况,没有研究,后面再研究.

fat表

每个fat表项占据12位,包含一个字节和另一个字节的一半.
fat项代表的是文件的下一个簇号,如果大于0xff8,就代表是最后一个项

它包含有两份fat表,这是出于系统冗余考虑,尽管它很少使用,即使是磁盘修复工具也很少使用它。
它是分区信息的映射表,指示簇是如何存储的。
每个FAT项占12bit,这也所被成为FAT12的原因,FAT项的值代表的是文件下一个簇号,但如果值大于等于0xFF8,则表示当前簇已经是本文件的最后一个簇。如果值是0xFF7,表示它是一个坏簇。
一个FAT项可能跨越两个扇区,编码实现的过程要考虑在内。

int 13h中断

1. AH=00h DL=驱动器号 DH=类型(00软盘) 复位磁盘
2. AL＝扇区数|CH＝柱面|CL＝扇区|DH＝磁头|DL＝驱动器类型,00H7FH：软盘;80H0FFH：硬盘|ES:BX＝缓冲区的地址 从磁盘读取数据到es:bx指向的缓冲区

读取的过程

实际上读取就是遍历根目录区所有扇区,然后从里面找到文件名对应上的条目,找到后加载(jmp).
文件名比如说是loader.bin,kernel.bin这样子这一部分代码在pro4中.

关于fat12参考百度百科FAT12

C语言与汇编的相互调用

这里我们使用的是nasm与gcc的联合编译.主要过程便是nasm编译为elf格式的可重定位文件,gcc编译为elf格式的可重定位文件,然后ld链接为bin.

GCC与nasm的联合编译

这个部分实际上需要说明的就是相应的一些参数
gcc参数:

参数	作用
-m32	-m32 生成32位机器的汇编代码
-c	生成obj文件
-fno-builtin	不使用C语言的内建函数
-fno-stack-protector	禁用栈保护措施

参数	作用
-ggdb	-ggdb 产生 GDB 所需的调试信息
-g	产生调试信息(不限于gdb)
-O0	不做任何优化

nasm参数:

参数	作用
-f bin	指定格式为bin
-f elf	指定格式为elf
-O0	不做任何优化

ld参数:

参数	作用
-m elf_i386	指定所需要的格式
-nostdlib	不连接系统标准启动文件和标准库文件,只把指定的文件传递给链接器
-e start(例子)	指定程序的入口符号为start
-Ttext 0x7C00(例)	链接时将初始地址重定向为0x7c00,默认为0

elf格式

```
elf文件分4部分,elf头,程序头表(Program header table),节(Sections),节头表(Section header table).其
elf头的结构如下:
typedef struct{
    unsigned char e_ident[EI_NIDENT];
    Elf32_Half e_type;
    Elf32_Half e_machine;
    Elf32_Word e_version;
    Elf32_Addr e_entry;
    Elf32_Off e_phoff;
    Elf32_Off e_shoff;
    Elf32_Word e_flags;
    Elf32_Half e_ehsize;
    Elf32_Half e_phentsize;
    Elf32_Half e_phnum;
    Elf32_Half e_shentsize;
    Elf32_Half e_shnum;
    Elf32_Half e_shstrndx;
}Elf32_Ehdr;
```

elf格式为了支持8~64位的处理器,所以有上面的数据类型.
其中这里的数据类型定义如下:

名称	大小	对齐	用途
Elf32_Addr	4	4	无符号程序地址
Elf32_Half	2	2	无符号中等整数
Elf32_Off	4	4	无符号文件偏移
Elf32_Sword	4	4	有符号大整数
Elf32_Word	4	4	无符号大整数
unsigned char	1	1	无符号小整数

现在详细地看这个elf头. 开头的4个字节是elf格式的签名0x7f,ELF
后面的

名字	含义
e_type	文件类型,比如可执行文件为2
e_machine	体系结构(比如i386)
e_version	文件版本
e_entry	程序入口地址
e_phoff	Program header table的偏移量,单位字节
e_shoff	Section header table的偏移量
e_flags	不知道
e_ehsize	elf头的大小
e_phentsize	Program header table每一条目的大小
e_phnum	Program header table的条目数
e_shentsize	Section header table的每一条目的大小
e_shnum	Section header table的条目数
e_shstrndx	包含节名称的字符串节是第几个节

elf头之后就是Program header table,他的数据结构如下

```
typedef struct {
    Elf32_Word    p_type;
    Elf32_Off     p_offset;
    Elf32_Addr    p_vaddr;
    Elf32_Addr    p_paddr;
    Elf32_Word    p_filesz;
    Elf32_Word    p_memsz;
    Elf32_Word    p_flags;
    Elf32_Word    p_align;
} Elf32_Phdr;
```

名字	含义
p_type	段的类型
p_offset	该段第一个字节的偏移。
p_vaddr	该段的第一个字节的虚拟地址。
p_paddr	为物理地址保留
p_filesz	段在文件中的长度。
p_memsz	段在内存中的大小
p_flags	与段相关的标志。
p_align	与对齐相关

关于elf格式与链接实际上对我们在代码的编写上可能没有直接的影响,但在写相关的代码中在反复的研究过程中是经常要用到的

函数的调用

汇编语言中函数调用使用call指令调用函数。而对应的ret执行将从栈顶pop出一个地址后,跳转到此指令执行。
c语言间的函数调用,只要对调用的函数有相应的实现,然后链接时链接了就好了.

而c语言和汇编的函数调用,实际上有些多余的工作要做.

1. 汇编中是有global来标注被引用的函数
2. 汇编中通过extern来作为未定义的声明。

参数的传递

实际上,参数的传递很依赖与平台!!!
我在参数的传递上尚且有一些问题!

1. 参数的传递通过栈来进行,后面参数先入栈,由调用者清理堆栈."从右向左,一次压栈"
2. 函数的返回值将存入寄存器%eax中。如果结果为64位,则其高32位存入寄存器%edx中

分页机制&中断设置&保护机制

分页机制

在386中,页是固定4096字节(4KB)的一块内存.
保护模式针对分段机制实际已经提供了保护机制,而引入分页机制的原因实际上是为了实现虚拟存储来灵活的管理内存.

逻辑地址--(分段机制)-->线性地址--(分页机制)-->物理地址
在没有打开分页机的时候,线性地址等于物理地址,而打开后就多了一层转换.

这个转换的过程要使用到两级页表

第一级是页目录(PDE),1024项每项4字节==4KB,每项对应第二季的一个页表
第二级是页表(PTE), 1024项每项4字节==4KB,每项对应一个物理页

然后页目录保存在cr3寄存器中
cr0的最高位决定分页机制是否打开.

这个与gdt是非常类似的.

PDE与PTE的结构

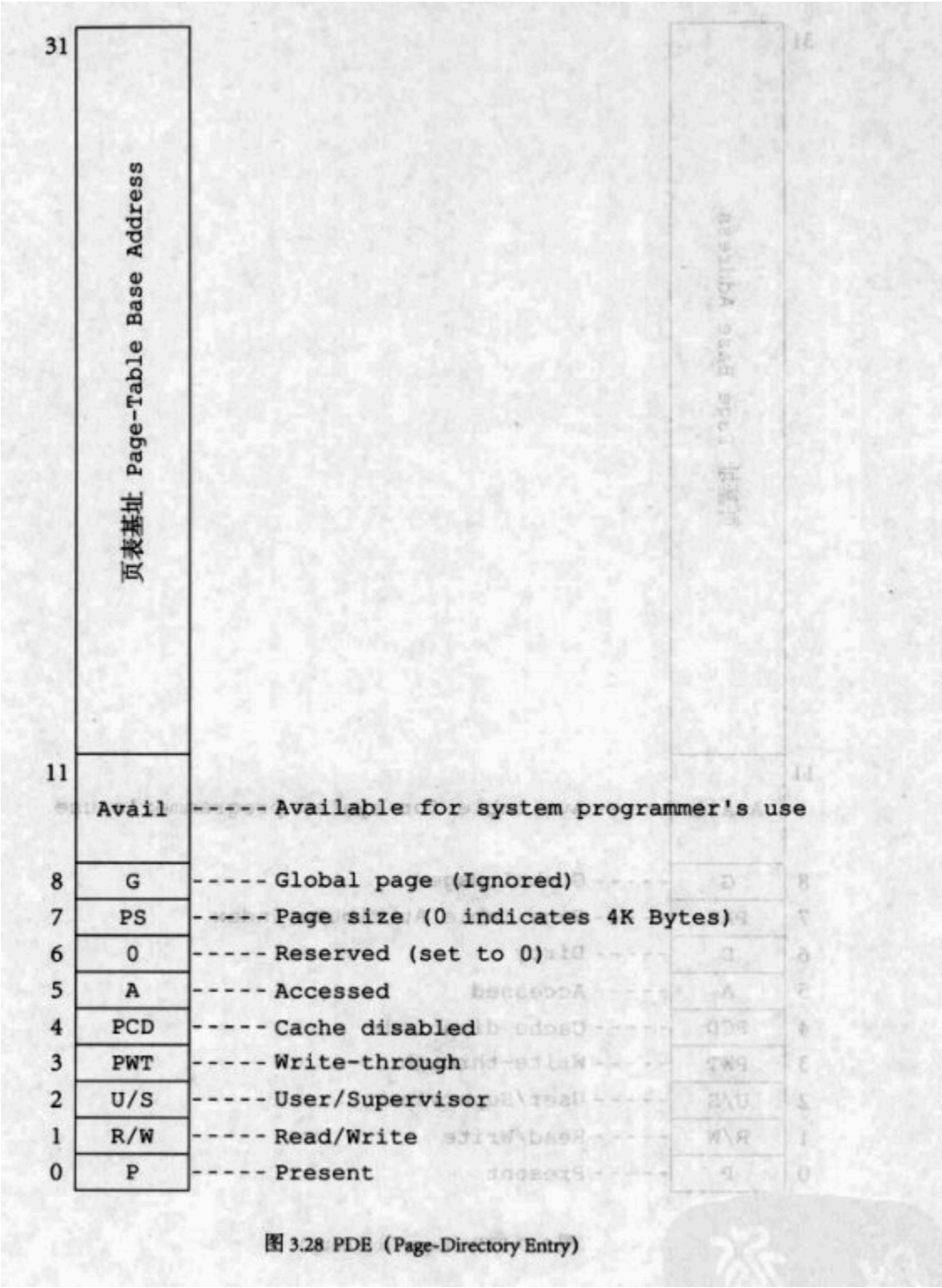


图 3.28 PDE (Page-Directory Entry)

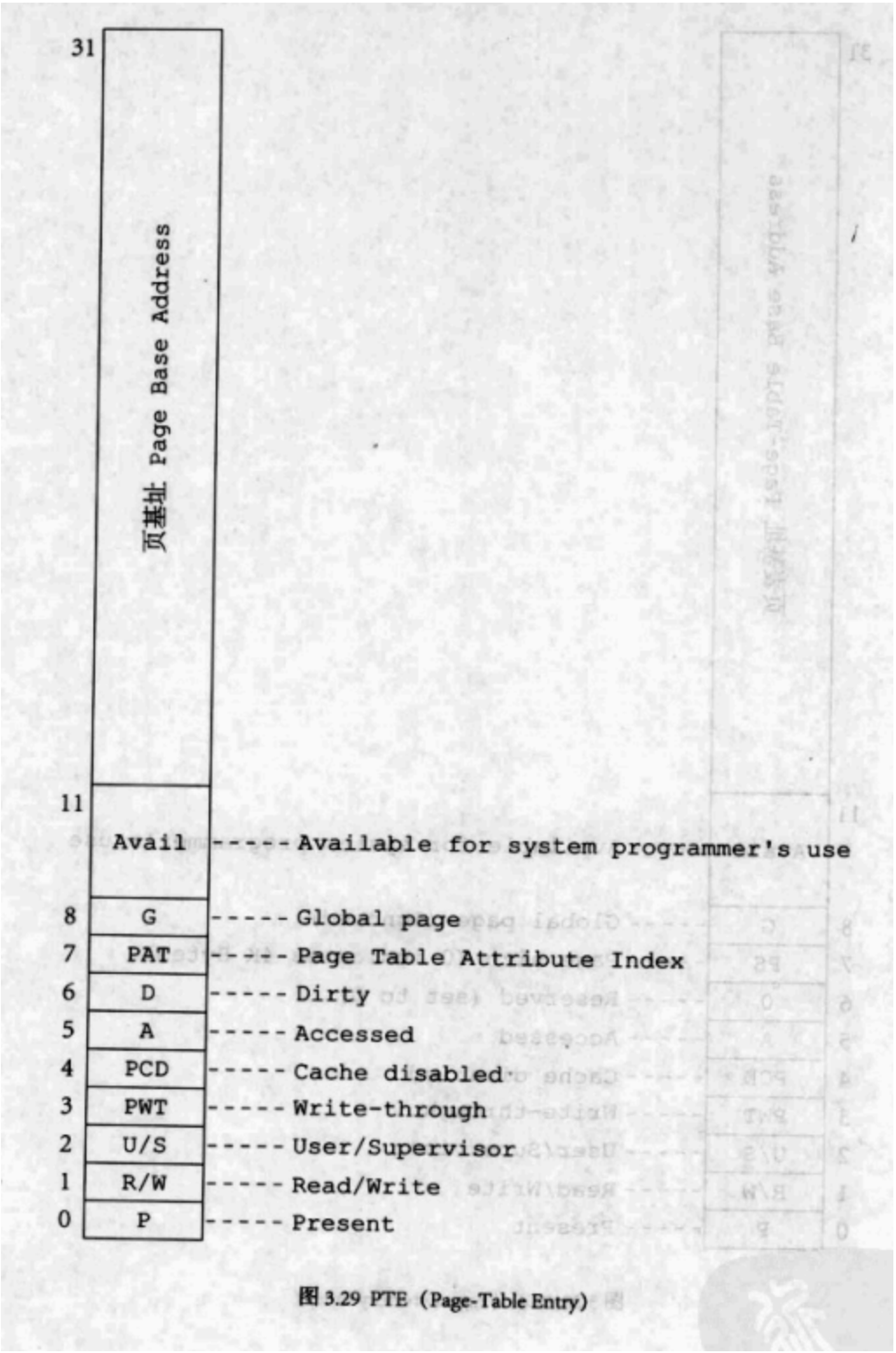


图 3.29 PTE (Page-Table Entry)

P 存在位,P=1表示在内存中

R/W表示读写权限

U/S与特权级相关

PWT与缓冲策略相关

PCD用于控制单个页或者页表的缓冲
A表示是否可以被访问
D表示是否可以被写入
PS决定页的大小
G表示全局页

cr3的结构

cr3又叫做PDBR(Page-Directory Base Register),它的高20位是页目录的地址的高20位,页目录的地址的低12位是0=>PDE是4KB对其的.

获取内存大小来初始化PDE

暂无

中断设置

说起中断,我们应该已经使用了bios的10h,13h,15h中断.而都是在实模式下调用的,原因是进入保护模式后,bios的中断已经不能使用了.需要我们自己来完成.

IDT(中断描述符表),实际上和gdt等是类似的.一个专用的寄存器指向IDT,在这个表中有若干描述符.对于中断我们实际上需要的是一个处理程序.在描述符中会给出这个相应代码的执行位置.基本上就是 中断向量-处理程序

中断分为外中断与内中断,内中断是我们软件造成的,比如我们手动执行的int 10h,这个和一个函数简直没有差别.而外中断来自于外设.
与中断经常提起的还有异常,不过这二者没有本质上的差别.

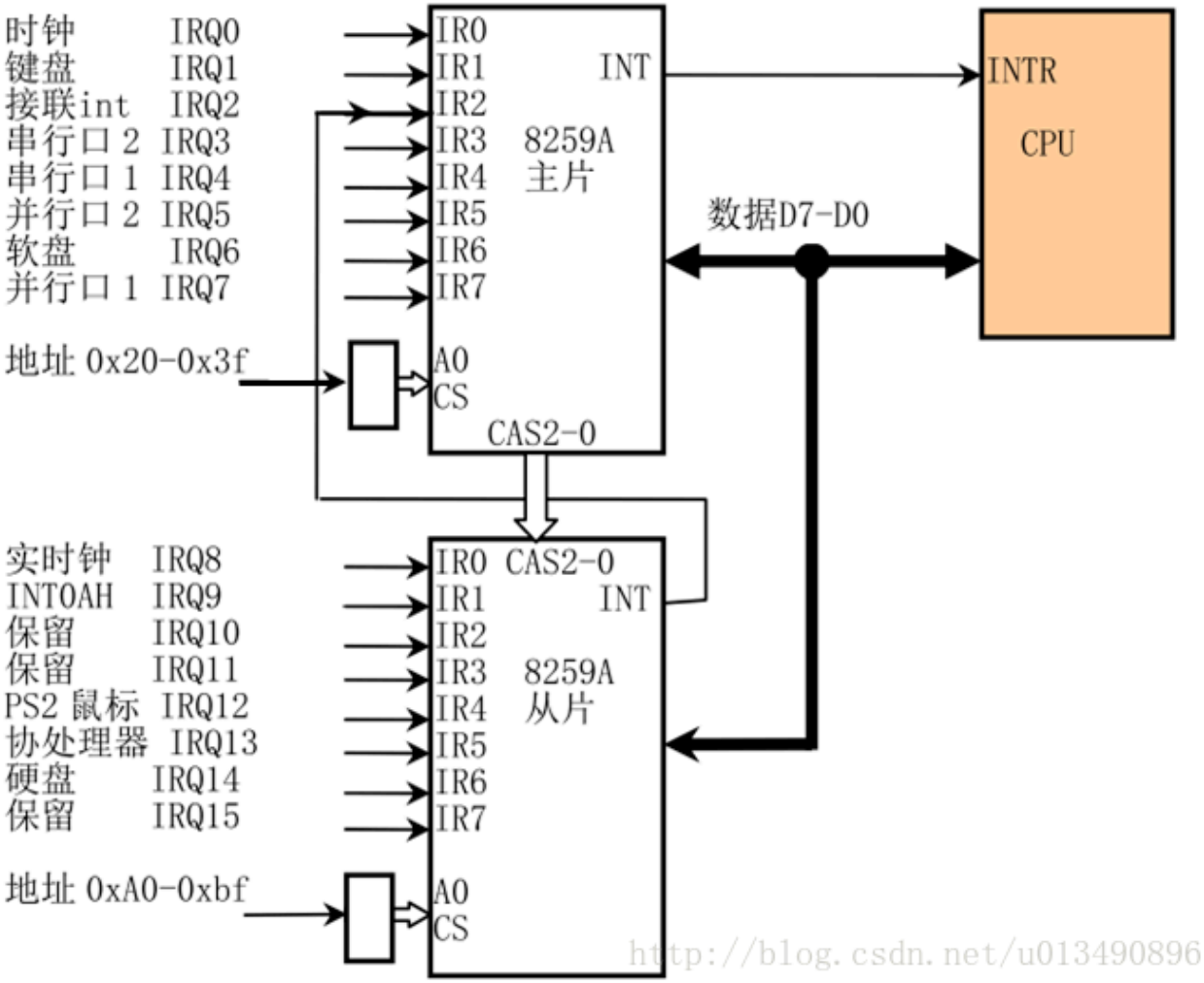
8259a芯片

外中断的情况比内中断要复杂一点,因为涉及到了硬件.

要提到的硬件就是8259A芯片,80x86系列的cpu用的就是它.而且是两片,一主一从.每个8259a有8根中断信号线,总共可以挂在15个不同的外设.通过对8259a的来完成这些设备的中断请求到中断向量的联系.

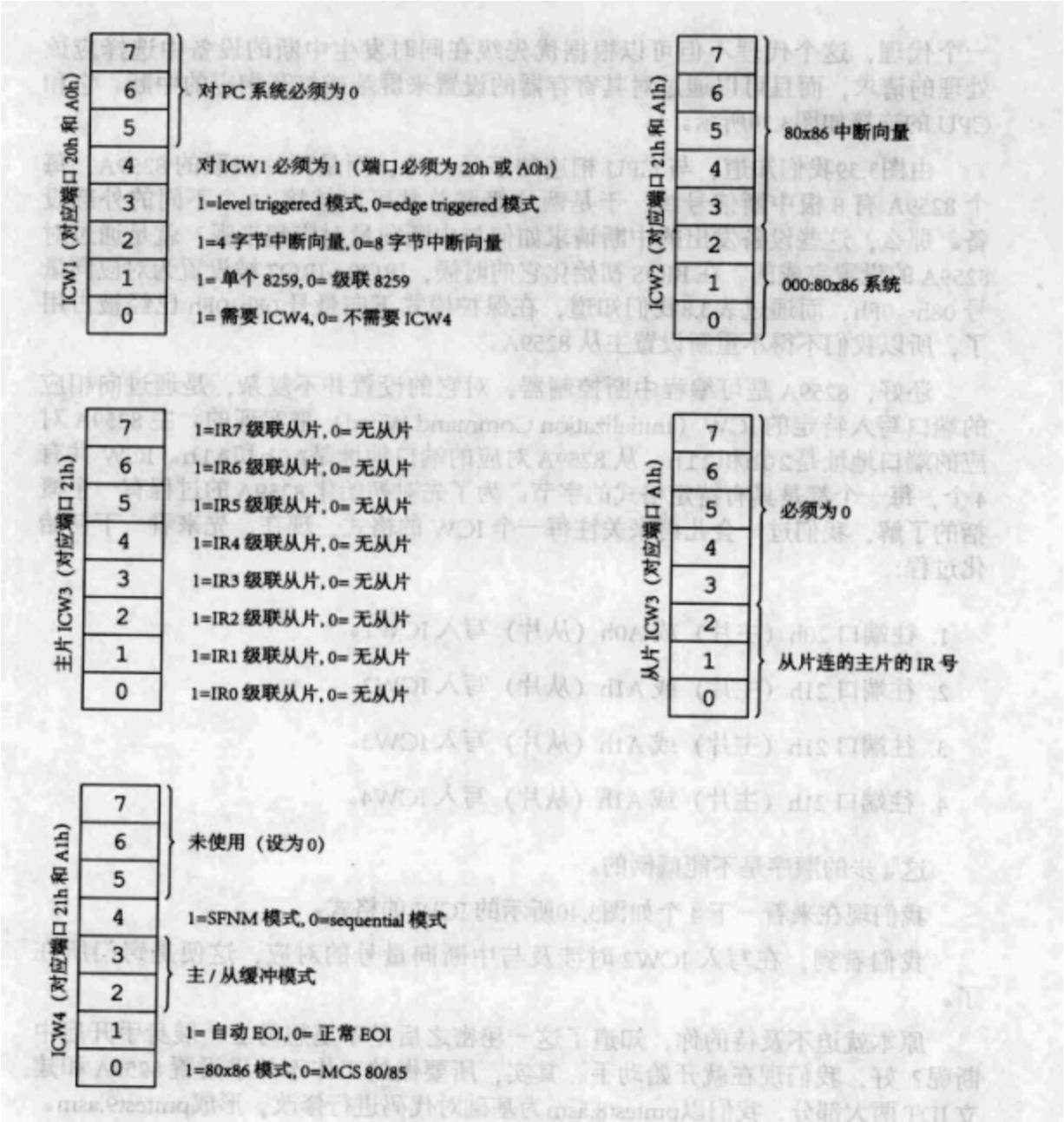
关于8259a芯片的详细的内容可以参考中断技术

8259a芯片如下:



初始化四个中断命令字(ICW)(Initialization Command Word寄存器来完成8259a的初始化：

1. 主芯片端口20h与21h.从芯片端口A0h与A1h
2. 向两个芯片分别写入4个中断命令字(顺序不能倒)
3. 与中断向量号对应的部分在ICW2中



然后通过三个写入ocw(Operation Control Word)来操作8259a,可以完成屏蔽/打开中断,优先权控制及中断结束.

这一部分实在是太硬件了,需要相应的资料,参考了一些代码

建立IDT

lidt是类似lgdt的命令,可以用来设置中断的寄存器.

idt的一个表项占据8个字节
他的结构如下:

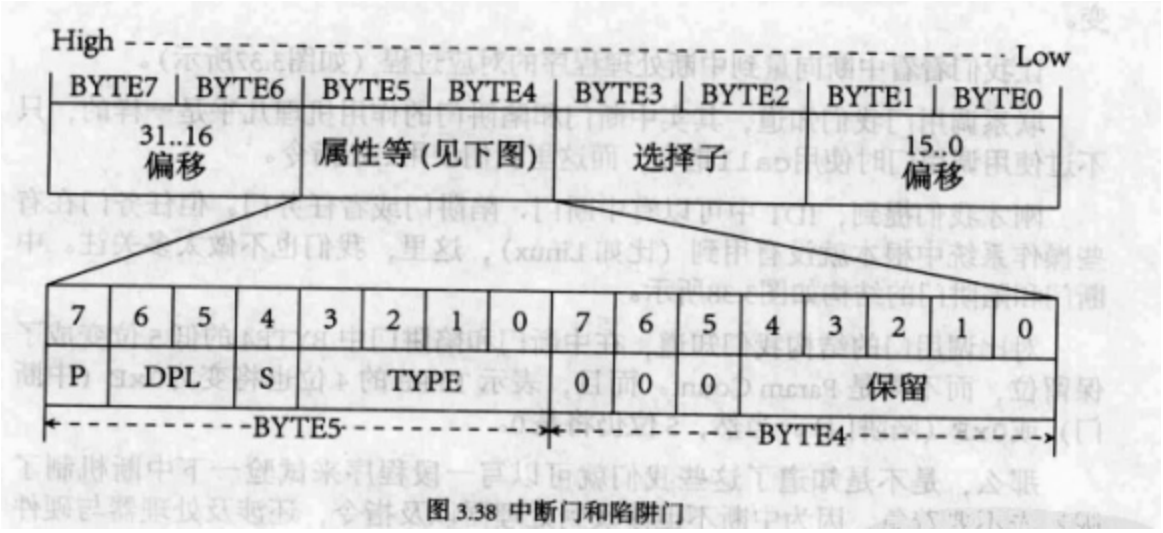


图 3.38 中断门和陷阱门

我们在汇编中使用如下的宏来快速完成一个表项

```
%macro Gate 4
        dw        (%2 & 0FFFFh)                                 ; 偏移 1(2 字节)
        dw        %1                                                                ; 选择子(2 字节)
        dw        (%3 & 1Fh) | ((%4 << 8) & 0FF00h);属性(2字节)
        dw        ((%2 >> 16) & 0FFFFh)                ; 偏移2(2字节)
%endmacro
```

```
; 系统段描述符类型值说明
DA_LDT                EQU          82h         ; 局部描述符表段类型值
DA_TaskGate        EQU          85h         ; 任务门类型值
DA_386TSS        EQU          89h         ; 可用 386 任务状态段类型值
DA_386CGate        EQU          8Ch         ; 386 调用门类型值
DA_386IGate        EQU          8Eh         ; 386 中断门类型值
DA_386TGate        EQU          8Fh         ; 386 陷阱门类型值
```

使用上比如 `Gate        SelectorCode32, SpuriousHandler, 0, DA_386IGate` 便是一个SpuriousHandler为中断处理函数,段选择子是SelectorCode32的中断处理程序,对应的中断号按照声明顺序来.

其他

在I/O操作后,我们需要调用延迟函数来等待操作完成.

参考

```
io_delay:
        nop
        nop
        nop
        nop
        ret
```

可以写入OCW1来屏蔽外设的中断

保护机制

暂无

总结

操作系统可能是大学最难的一门课程,在尝试完成这个操作系统的过程中非常的坎坷.

1. 合适的资料太少
   涉及到操作系统的资料实际上并不少,但是真的教你如何完成一个操作系统的资料实际上并不多.linux如今已经达到了数千万行,而且早些年的代码因为编译环境的问题,甚至不能编译.而且对于初学者很多资料也很容易显得晦涩.
2. 知识点太多
   完成一个操作系统所需要的知识点太多了!从硬件机制到平台特性到工具的使用.
   当尝试阅读一个完整的操作系统的代码时,面对前辈们积累了许多年的成果,即使代码很优秀,你想要吃下来,也就像从一件毛衣上找到它最初的线头.知识点之间经常互相关联,难以分清主次.
   我看过好几份不同的操作系统书籍,他们在开发环境&工具选用&实现方式上都有些差别,而我缺少其中的一环就感觉无从下手.
   比如我一开始选用的虚拟软件是qemu是因为参考的项目ucroe.我在最初是不准备使用bochs的,但是当我在参考OrangeS一个操作系统的实现一书的时候,我几乎没有办法继续.
   比如从实模式到保护模式一书的开发完全使用汇编,我在其中完全的不到汇编如何与c结合的经验,这与我想要开发的结果是完全不同的.
   比如orangeS一书中使用了软盘+fat12,如果我只想用硬盘呢,后面的代码在参照时就会出很多的问题.
   各个资料都用自己的方式完成了答案,如果你的方案与他们有一环不同而你又是个初

学者,你就很容易陷入困难.那就更要求学习者的能力了.

3. 平台依赖性太强
   在写操作系统时,你必须一直关注你的操作系统的土壤,也就是你的平台是什么.我选用了i386的原因是参考的数目里多选用了这个版本.里面有很多的trick都是与平台密切相关的,还有很多的历史原因.要想把所有的地方搞清楚很难.
4. 编程难度大
   在理解操作系统的理论时,对于细节不是需要知道的那么清楚.
   而到了实际上手的时候,就会发现很有可能一个细节理解的错误导致整个的错误.很有可能思路

虽然即使能够完成最终的项目,也只是个玩具.但是做这样的一次尝试更多的是一种好奇,一种求知欲的支撑.

本来计划是完成一个小型可用的os,但是实际上远远低估了项目的难度.
而且曾经多次陷入停工状态.即使到现在完成的进度里,里面还是有很多我不懂的/不是完全懂的地方.而这些地方我都只能暂时记下来,慢慢调试.
目前整个项目完成了进入保护模式,汇编&C混编,loader&kernel的加载,分页机制,中断机制等任务.
接下来准备对已经完成的部分做一次完善总结后,再开始后面的内容比如进程.
这份报告的完成度并不好,因为不知道如何与project里的代码相结合起来.纯粹的知识点很难讲清楚

虽然完成的很困难,但是我在其中是有巨大收获的,而且当你回头看的时候你才发现好像没有那么的困难.