**小型多任务操作系统的设计与实现**

**TINY MULTI-TASKING OPERATING SYSTEM: DESIGN AND IMPLEMENTATION**

**专 业：计算机科学与技术**

**姓 名：刘亚飞**

**指 导 教 师：**

**申请学位级别：学 士**

**论文提交日期：2013年6月5日**

**学位授予单位：天津科技大学**

摘 要

操作系统是计算机的灵魂，然而随着操作系统的智能化与民用化，越来越多的底层细节被操作系统无情的屏蔽掉了，如今的程序员也许精通很多种语言，也可以写出很棒的程序，但对计算机到底是怎样工作的可能都不是理解的很透彻，在海量的存储空间和高速的运行速度面前，很多程序放弃底层优化，使得程序运行效率不高。而在几十年前，程序员与电脑用户的差异却非常的小，他们不得不关心内存大小，栈是否溢出，以及如何利用流水线优化自己的程序，而他们发明的各种高效算法和思想，在如今的程序中依然广泛流行。所以对于如今的程序员，了解系统底层依然十分重要，而编写一个小型操作系统正可以加深对计算机的了解。

本文通过汇编语言实现对CPU和外设的读写与控制,实现一个简易的可运行于裸机的微型操作系统。因为要在裸机上运行，所以必然要和硬件打交道，那么传统的高级语言就完全派不上了用场，在低级语言中使用汇编，因为汇编语言是与硬件关系最密切的编程语言了。既然要编写操作系统，那么大名鼎鼎的开源内核Linux就是我们的参照了，但是与Linux不同，本文采用了Intel格式的汇编语法和非AT&T格式语法。在编程环境方面，我们使用cygwin来模拟Linux环境，从而可以使用Linux丰富的命令来使整个制作过程简便。最后对于作品的测试使用了Bochs虚拟机呈现，之所以选用Bochs不仅因为他是一个虚拟机，而且它还可以充当调试器来调试我们的代码。

关键词：操作系统；汇编；Bochs虚拟机；cygwin环境

ABSTRACT

Operating system is the seal of a computer. However, with the os becoming more and more intelligence and easy to use, a lot of details about hardware were ignored. Now a day, programmers can build excellent software by using lots kind of high-level languages, but know little about the low-level. In the time which is full of massive storage spaces and high-speed hardware, a lot of programmers give up to optimization, which makes their code inefficient. However, back to a few years, the different between coder and user is so small, computer users had to care about the size of the memory, whether the stack is overflow, how to optimize their codes by using assembly line. Many typical algorithms were born at that time, and still widely used in today. So it is necessary to get knowledge of low-level. And to get a the knowledge, build a tiny os is an easy way.

In this text, we will build a tiny OS to control our CPU and hardware by using assembly language. Because the OS is work on a bare computer, so we have to deal with hardware, and all kinds of high-level language is none of use at all. In low-level language, the assembly language is the best choice witch is related closely with hardware. Since we want to write the operating system, then the famous open-source Linux kernel is our reference, but with Linux, this paper uses the Intel assembler syntax format not AT & T format syntax. In the programming environment, we use cygwin to simulate the Linux environment, which can use the Linux command easy to make the entire production process. Finally, for the work test using Bochs virtual machine presents, Bochs was selected not only because he is a, but also it can act as a debugger to debug our code.

**Key words:** Operating system; assembly language; Bochs virtual machine; cygwin environment

目 录

[**第一章 操作系统简介 1**](#_Toc359249214)

[第一节 操作系统发展史 1](#_Toc359249215)

[**第二章 编程环境的搭建 4**](#_Toc359249216)

[第一节CYGWIN简介 4](#_Toc359249217)

[第二节BOCHS使用与调试 4](#_Toc359249218)

[第三节 认识MAKEFILE 9](#_Toc359249219)

[**第三章 intel IA-32体系构架 11**](#_Toc359249220)

[第一节 保护模式内存管理 11](#_Toc359249221)

[第二节 中断和异常处理 15](#_Toc359249226)

[第三节 任务管理 16](#_Toc359249227)

[**第四章 具体代码实现 19**](#_Toc359249228)

[第一节 引导部分 19](#_Toc359249229)

[第二节 内核部分 27](#_Toc359249234)

[**结论 39**](#_Toc359249237)

[**参考文献 41**](#_Toc359249238)

[**致 谢 42**](#_Toc359249239)

# 操作系统简介

## 第一节 操作系统发展史

操作系统与计算机相生相伴，计算机的发展史就是操作系共同的发展史。计算机最初是由电子管组成，人们把需要的程序做成有孔的纸带输入计算机，再将结果由打印机打在纸上。这个时期没有操作系统，所有的操作都需要人工手动完成。在整个过程中工作人员需要不断的装纸带，拆纸带，费时费力。

到了第二代计算机的问世，由晶体管组成的计算机大大缩减了计算机的体积，出现了磁盘，通道，终端的部件，这是为了解决人机交互的问题，避免人工操作给计算机硬件资源造成浪费，人们提出了一种解决方法：把作业的输入，编译，运行和输出的流程交给预存在计算机内存中的监控程序进行管理。监控程序为操作人员提供一些指令，可以是操作人员进行一些复杂的控制。但是监控程序过于简单，他不允许操作人员和正在执行的作业交互，也无法防止作业对计算机的修改。

为了防止作业队计算机系统造成的伤害，人们将指令分配了不同的级别，不同程度的操作需要不同级别的指令。例如使用系统资源时需要系统级别特权的指令，这种特权一般是最高的，只有监控程序才有权使用这种指令，而用户只能使用普通指令，如果用户想使用系统资源，必须通过系统调用，向监控程序发出请求，监控系统在通过调用相应的内部程序来完成用户的请求。

随着第三代计算机的问世，集成芯片开始具备多道处理的能力，多道处理中的分时处理机制为家用计算机实现多任务奠定了基础。而大规模集成电路的问世，使得微型计算机的成本急剧下降，个头也变得更小。随着微型计算机的普及，操作系统也出现了百花齐放的态势。

DOS系统

DOS系统的学名叫做磁盘操作系统。这个操作系统诞生于 1979年，当时的微软公司一直在致力于帮助IBM改进了PC的语言设计和应用程序设计，但是他们意识到只有为PC机研发出一套操作系统才能使自己发展，但是当时的微软已经承诺IBM为其提供一个操作系统，而从头开始开发一个操作系统显然已经为时已晚。为了解决这难题，微软公司购买了一个基于8086构架的单任务[操作系统](http://baike.baidu.com/view/880.htm)，名字叫做86-DOS。这个操作系统是一个年龄只有24岁的程序员蒂姆·帕特森用了四个月的时间编写而成的，微软公司在这个操作系统的基础上加以改进，然后给每台IBM电脑安装一个此DOS系统，而微软每台电脑收取200美金。正是这种营销模式，创造了如今的微软帝国。

Unix系统

该操作系统是一个强大的多使用者、多进程操作系统。该系统是于1969年由肯·汤普森、丹尼斯·里奇和道格拉斯·麦克罗伊三位编写的。它支持多种[处理器架构](http://baike.baidu.com/view/2656836.htm),是一个名符其实的[分时操作系统](http://baike.baidu.com/view/880784.htm)。两年后他们推出了该操作系统的第一个版本，所使用的电脑其物理内存只有几十KB。而Unix自己就占用了一半的内存，剩下的空间被用来执行一个叫做星际旅行的多玩家游戏。又过了两年，肯·汤普森 与丹尼斯·里奇觉得用低级语言做各种不同平台的系统实在是太麻烦了，于是他们想到使用高级语言来完成新的版本，而在他们那个年代并没有太多优秀的高级语言，而像这种程序大都是用汇编写的。一开始他们选择了fortran，但是结果却不理想。于是这帮强大的神仙在各种尝试之后研发出了B语言，这就是C语言的前身。随后，肯·汤普森 与丹尼斯·里奇使用这种语言编写出了信一半的内核。从此以后，该操作系统就实现了跨硬件，各种改写和移植都变得非常简单。从此以后，提到该系统就会涉及C语言，二者相辅相成的走向了计算机历史的神坛。

Linux系统

又是一个天才所创作的杰作，他的名字叫做莱纳斯。Linux的出现，他编写这个系统的初衷是想设计一个系统，从而能够取代Minix。Minix是一位叫做安德鲁的大学老师编写的，他的目的是想让学生理解操作系统，于是就编写了一个。这个操作系统几乎适用于当时所有的机型，而且具有UNIX的功能。而在当时UNIX的使用是付费的，为了响应开源的理念，Linux的构想就逐渐成型了。

该操作系统有着无可比拟的高效性。灵活性也是其一大特点。它是一个多任务的操作系统，而且实现了它的前辈操作系统的全部功能。该操作系统积极响应开源世界的号召，在许可下就可以免费获得。当时开源领域的天才们为了对抗各种封闭的商业协议，已经开发出了各种开软软件，从文字编辑到网页浏览器应有尽有，但是就是欠缺一个开源的操作系统，而Linux的出现填补了这一空白。正式归功于这些先行者的努力，我们如今可以使用该系统来实现各种作业，而随着开源的X-WINDOW的出现，如今是用该系统也能实现很好的实现我们常用的可视化操作。

LINUX的广泛传播与巨大的影响力是因为本身的特点：一是这个操作系统是纯粹开源的，每个人都可以无偿的使用和修改它，并且可以加以修改后再次传播，如今流行的红帽，吴本图操作系统都是以这种形式修改完善并传播的。第二点是Linux有着UNIX的全部功能，而随着时间的推移，庞大的用户群体都可以零难度的转向该开源的系统。这两点使得该操作系统在同类产品中迅速地脱颖而出，无数的电脑爱好者从中获得了知识，从此了解和改造内核不再是一件难事，这是多么令人激动的一件事。而普通用户也拥有数量巨大的软件可供选择，而且无需付费。

windows系统

这个操作系统已经无需多言。比尔盖茨要世界上所有的电脑都长同一张脸，他的确做到了。Windows无论哪次更新都会引发无数的埋怨甚至骂声，但他的用户数量却从来没有被动摇过，这个操作系统应该是目前最容易使用的了。这个操作系统的视窗概念绝对是计算机历史的一大重要里程碑，他使得计算机不再是极客们手中的万物，而是飞入了寻常百姓家。因为有着图形界面的缘故，人们使用计算机无需在了解繁杂的命令，只需要使用鼠标，通过简单的点击就可以实现各种操作。这种界面方式提供了巨大的边界给用户，从此把计算机的概念刷新到了一个新的高度。

# 第二章 编程环境的搭建

## 第一节CYGWIN简介

该是一个微软平台下的命令行模拟软件，能够实现Linux shell的全部功能。之所以选用模拟器而非linux系统主要是因为大多数学习者都是从windows环境起步的，而且笔者的windows8系统为了快速启动修改了开机时的一些设置，因此安装双系统比较麻烦。但是随着代码的深入，笔者发现使用linux有着无可比拟的优势，因此采用了折中的办法。

CYGWIN的使用与linux shell毫无区别，下图是使用make编译程序的情形：

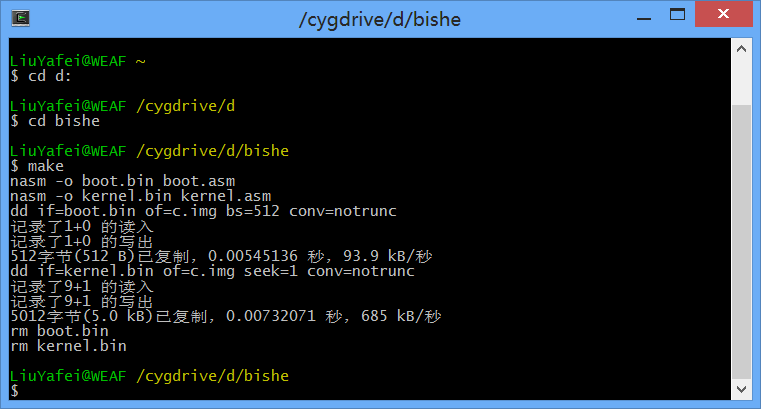


图2-1 cygwin界面

## 第二节BOCHS使用与调试

代码编写完成以后是需要调试的，如今的市面上调试器的种类很多，有集成的也有独立的，但这些调试器都是建立在操作系统的基础上，而我们现在编写的程序就是一个操作系统，因此没有调试器可供我们使用。对于操作系统的调试，就不得不提BOCHS这个软件。BOCHS是一个虚拟机，而与一般的虚拟机不同，BOCHS完全模拟计算机的每一个步骤，而一般的虚拟机萎了效率只是部分模拟计算机步骤，其他很大部分是使用真实的硬件。而完全模拟虽然效率低下，但带来的好处是我们可以在任何时刻观察计算机内指令进行的情况，这不就是调试么？

Bochs的配置非常简单，在配置文件中选择启动方式和镜像的地址就可以了：

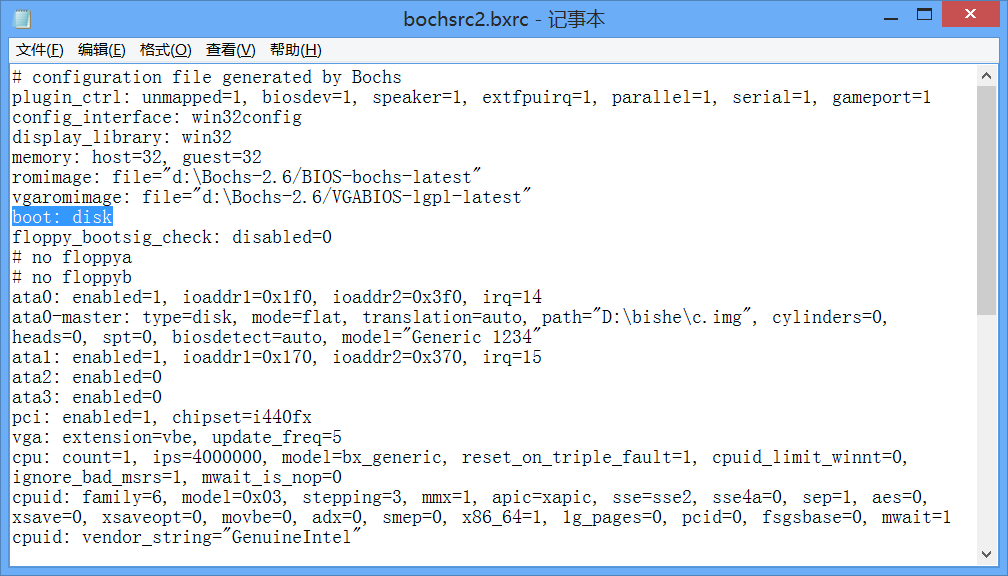


图2-2-1 BOCHS配置图1

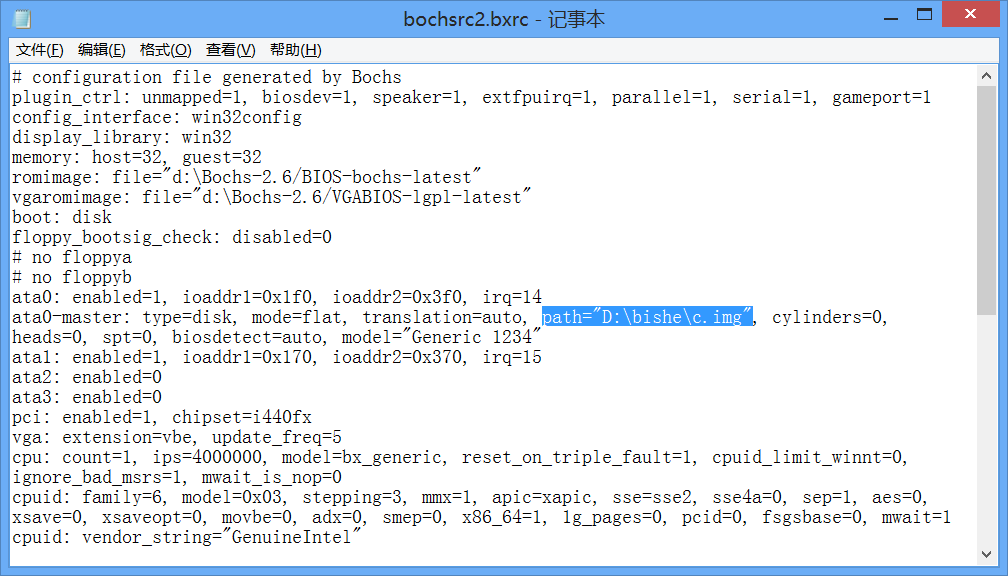


图2-2-2 BOCHS配置图2

下面以我编写的boot引导程序为例讲解BOCHS的各种命令。

我们知道，计算机开机后运行bios自检，然后将0扇区的代码复制到0000:7c00地址处（关于开启动具体细节参见第四章），因此我们将0000:7c00设为断点。Bochs中，设置断点的命令为b [32位物理地址]，跳转至断点的指令c，具体命令如下：

b 0x7c00

c

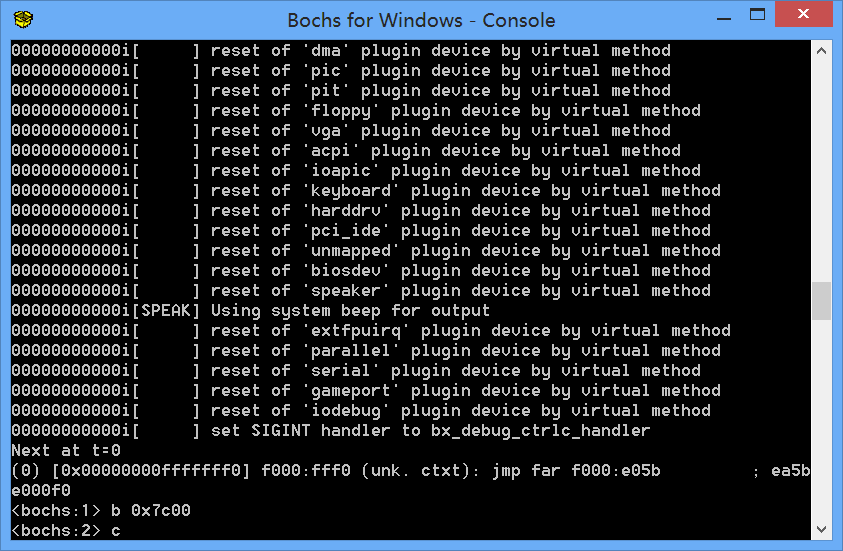


图2-2-3 BOCHS调试图1

输入这两条指令后，系统就会自动执行，直到遇到断点处停下来：

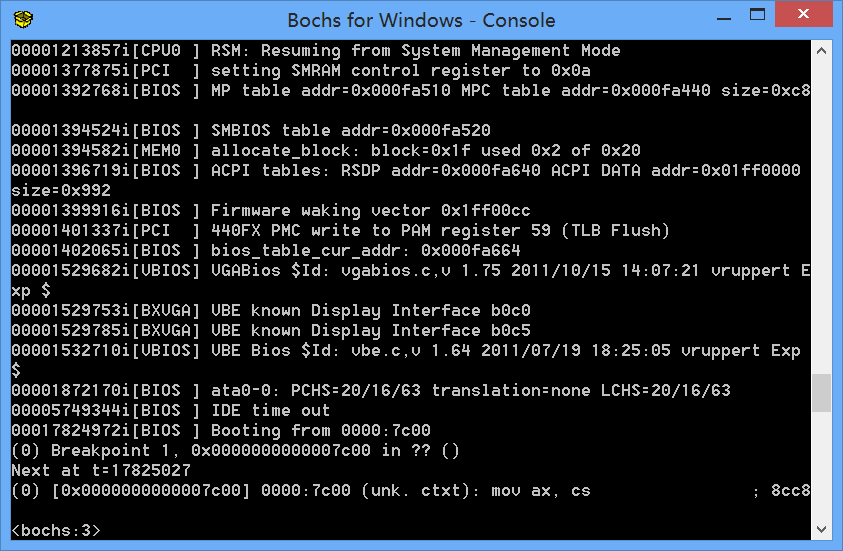


图2-2-4 BOCHS调试图2

从这里开始就是我们编写的引导程序boot.asm了。这时如果想查看代码，可以使用u命令：

u [/行数] [起始地址]

我们要查看的就是当前位置的代码，因此输入：

U /20

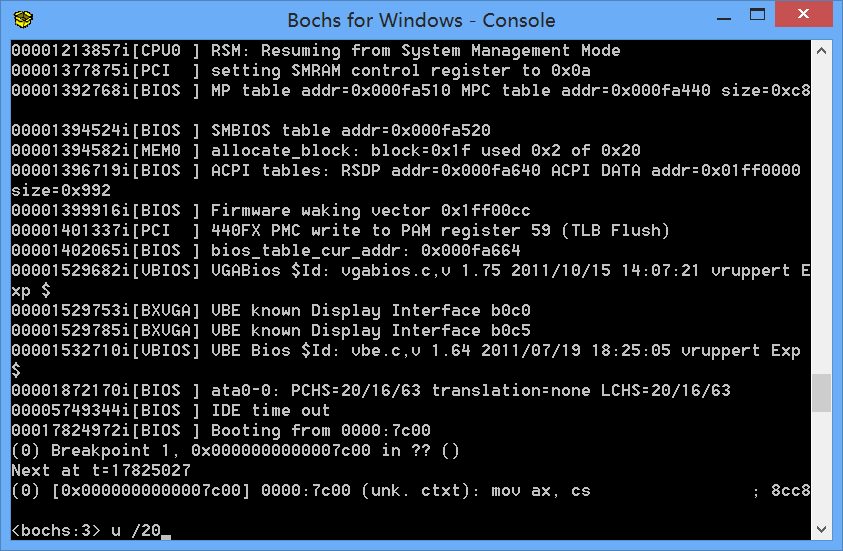


图2-2-5 BOCHS调试图3

结果如下：

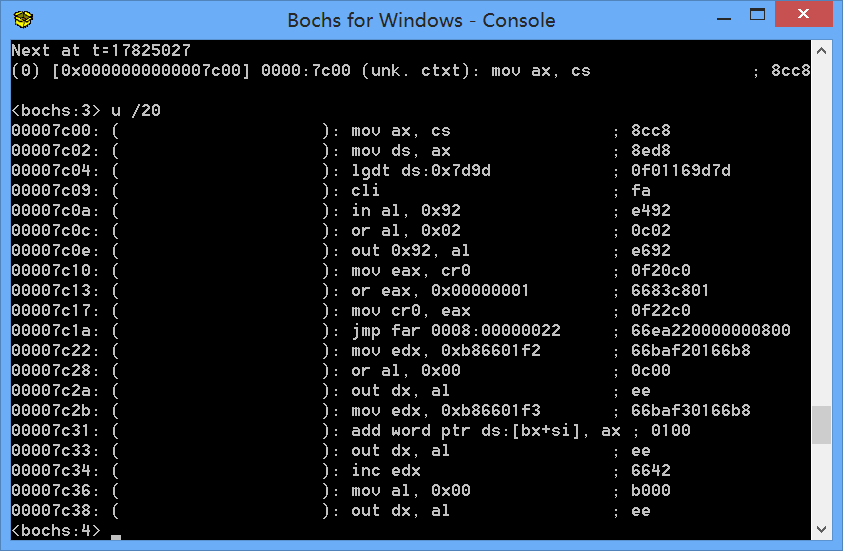


图2-2-6 BOCHS调试图4

此时如果想要单步执行指令，可以使用s命令:

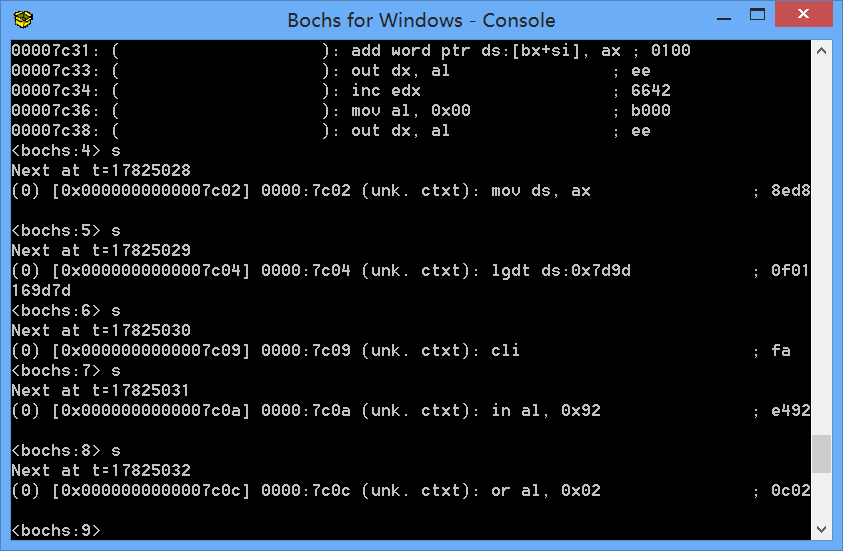


图2-2-7 BOCHS调试图5

不同于s命令，p命令虽然也是执行下一条语句，但是当遇到调用函数之类的情况的时候，p命令会跳过函数过程直接返回，这杨调试就不必跳到函数体内部，节省了时间。

要查看指定内存为之的数据，可以使用xp命令：

Xp [/[数据进制][数据个数]] [起始地址]

命令如下： Xp /20 0x7c00

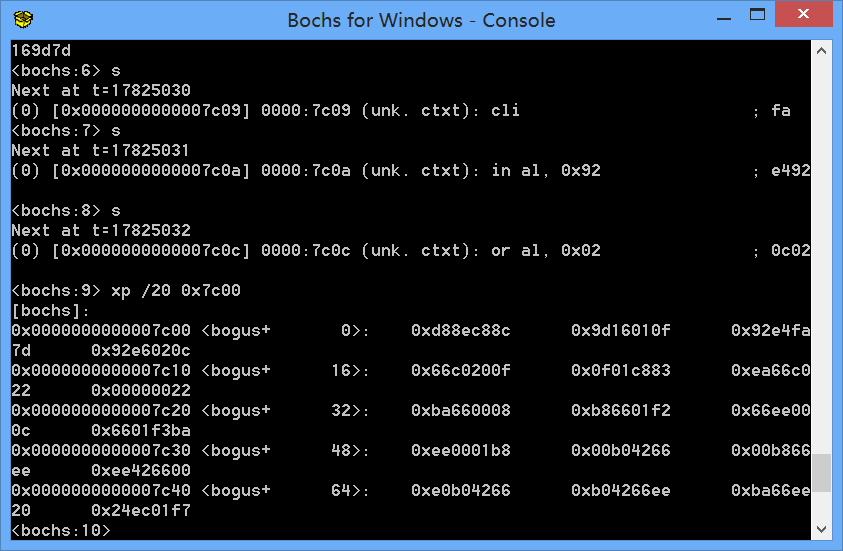


图2-2-8 BOCHS调试图6

在调试的过程中,有时候会出现一些bug的时候，而这些bug调试器并不能很直观的反映给我们，这时候就用到了reg和sreg命令。当我们要查看一些CPU的状态，可以使用这两个命令，前者查看通用寄存器，后者查看和段有关寄存器（比如说tr，gdrt，cs等重要的寄存器。查看寄存器状态对于我们debug是非常重要的。

鉴于我们编写的代码的规模，目前掌握这几个命令就足够了。

## 第三节 认识MAKEFILE

随着代码的深入，你会发现每次修改代码都需要重新编译，重新连接，重新将二进制文件写入镜像文件。没修改一次代码，你都要输入以下命令：

-nasm–o boot.bin boot.asm

-nasm–o init.bin init.asm

dd if=boot.bin of=c.imgbs=512 conv=notrunc

dd if=init.bin of=c.img seek=1 conv=notrunc

每修改一次代码，都要输入这些命令，这是非常繁琐而且很没效率的一件事。而编写一个脚本文件，可以把这些繁琐的步骤交给计算机自己完成。我们之所以选择用linux环境的原因，是因为因为它有这个强大的工具。以下是一个例子：

Makefile的语法如下：

Target : prerequisites

Command

说白了就是：一个目标文件需要一个源文件，以及生成目标文件需要的命令。例如

boot.bin:boot.asm

nasm -o boot.bin boot.asm

当在命令行里输入make boot.bin的时候，就会检查源文件是否是最新的，如果是就执行编译语句，如果不是最新的，就找到源文件的定义，在生成源文件。这就是一个递归的过程，有这个机制我们可以把需要变异的文件设为一个标号的依赖，这样我们只需要编译标号，源文件就会一次的编译，非常方便。以下是我在本例中使用的：

all:compiledd clean

compile:boot.binkernel.bin

dd:

dd if=boot.bin of=c.imgbs=512 conv=notrunc

dd if=kernel.bin of=c.img seek=1 conv=notrunc

clean:

rmboot.bin

rmkernel.bin

boot.bin:boot.asm

nasm -o boot.bin boot.asm

kernel.bin:kernel.asm

nasm -o kernel.bin kernel.asm

当在shell里只输入make时，默认会执行第一个标号，我么这个标号非常有技巧：它依赖于三个新的标号，分别是编译，写入，删除中间文件，这样每次输入make，这三条之领会依次执行，非常效率。

# 第三章 intel IA-32体系构架

## 保护模式内存管理

### 分段机制

由于新的CPU将地址线扩展到32条，因此最大寻址空间也得到了提升，变成了232大小，而以前的寻址方式为：段地址\*16+段偏移，这只能表示1M大小的空间，已经不够用了，所以新一代CPU中将偏移的位数增加为32位，同时段寄存器的含义页发生变化，段寄存器改名为段选择子，它存储的也不再是一个段地址，而是一个段的索引，通过索引在一个叫段描述表的数据结构中找到相应的段。这是的线性地址为：段选择子所映射的段的基地址（基地址为32位）+32位段偏移。如果没有开启分页机制，那所得到的线性地址就为最后的物理地址。

### 段选择子

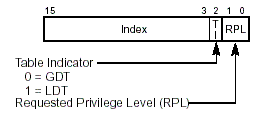


图3-1-1 段选择子

段选择子取代了16位代码下的段偏移,这个数据结构的的作用是通过他的值找到映射的段描述符。因为段描述符是8字节对其的，因此选择子的后三位必然为0，而intel把这3位设置为TI位和RPL位。TI为一比特，用来标识需要检索的段描述符存放在哪里，是全局的还是局部的。当TI=0，则存放在全局描述符表中，TI=1则在局部描述符表中。RPL的值为0,1,2,3，分别代表4个特权级，该标志位用于特权级检查，具体请见第三章保护。

#### 段描述符

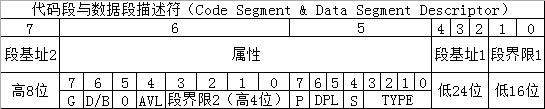


图3-1-2 段描述符

中央处理器采用如图所示的数据结构来定位一个段。这8字节的数据结构存储了一个段的重要信息。每当代码要从一个部分跳转到其他部分，处理器会检查slelctor和描述符特权等级的内容以确保该部分内容能被合法访问。

段描述符之所以属性分配如此不规则，是因为历史原因，当时286的段描述符只有6字节，386为了向上能使用以前的模式，所以不得以将段描述加长，造成如此凌乱的结构。

段描述符的属性众多：

段类型：这个属性表明了这个段的类型，以及他的延伸方向（因为栈的延伸方向与一般的段是相反的），同时这个属性也可以表明该段的类别。这个属性和类型标志属性是息息相关的，因为前者不同关系到后者是一个代码段还是算数据段。

描述符类型标志：上面已经介绍，因此不再赘述。

描述符[特权级](http://baike.baidu.com/view/2188561.htm)：这个属性用来表明该段的所能被访问特权级。[特权级](http://baike.baidu.com/view/2188561.htm)范围从0到3。0级[特权级](http://baike.baidu.com/view/2188561.htm)最高，3级最低。该属性的作用是通过特权级检查用来实现安全访问，阻挡非法的访问。

段存在位：这个属性的作用是表示这个段是不是在内存中。有人可能有疑问，不在内存中还要这个描述符做什么？其实这个属性是为了虚拟存储而存在的。当所要访问的段不存在，就会引发一个异常。我们可以编写程序来处理这个异常，比如说：我们可以把这个段是现存在硬盘中，当访问的时候触发异常，接着调用异常处理程序，把原本在外村中的数据加载到内存中，这样就实现了动态加载。这就是虚拟存储技术的前身，只不过现在都是以页为单位进行调换了。页表项中自然也有这个存在位属性。

D/B：对于不同种类的段，该位所代表的含义也不一样。其实这个位的名字会根据用途而有所不同,当他是一个普通的段,这个位就叫做默认操作大小位。如果这个段是一个栈，那么它所代表的含义就变的与栈属性相关了,这个时候他的名字叫做默认栈[指针](http://baike.baidu.com/view/159417.htm)大小。对与代码段，这个属性比较简单：如果是32为段，这个属性的值就恒为1，如果是16位的，这个属性的值就恒为0。关于在不同情况下该位的不同含义以及具体结构可以参阅一下INTEL开发手册。这本多达3044页的庞然大物是目前最权威的CPU编程手册，一切一切的定义与规范都应该参照这本书。

长度单位标志：这个属性的作用是用来定义一个段的最大偏移值的单位，单位不同，即使描述符中的最大长度的数值相同，这个段的长度页不同。该属性由两种取值：当这个属性为零，说明单位是BYTE。如果这个属性的值为一，那么单位是4KB。这个时候段的真实长度是最大偏移值乘以4KB。

保留位：这个属性是Intel留给系统编写者所使用的属性，编写这个已对这个属性赋予不同的含义以实现程序要求。一般情况下的用语是记录该段的使用次数。

#### 段描述符表

众多的段描述符组成了段描述符表，CPU中的gdtr寄存器用来存放全局段描述表的地址和长度，而ldtr寄存器存放局部描述符表的地址和长度。每当段寄存器接受一个段选择子的时候，CPU都会检查请求的段索引是否越界以及特权级是否符合规范。因为gdtr中limit位是16位，因此一个描述附表最多可以有1000H/8=8192个段描述符。同时注意，limit指的是最大偏移量而不是长度，所以应该是最：大偏移量=长度-1。

### 分页机制

#### 页

为了解决内存空间碎片问题和实现虚拟存储，386引入分页机制，将内存分为若干个大小为4KB的片段，称为页。页在内存中是连续的，也就意味着页是4KB对齐的。当启动分页的时候，线性地址不在等于物理地址，线性地址需要通过存储在cr0寄存器中的页目录相映射而计算出最终的物理地址。

#### 页表

为了对页进行索引，在内存中要存放页表，通过页表可以将线性地址转换为物理地址。一般的分页机制采用二级页表：一个页目录表（PDE）索引内存中的页表，而页表索引页。因此一个线性地址可分为三段：高十位用来查找目录，中间十位为用来查找相应的表，低12为页内偏移地址。因为页是4KB对齐的，因此无论是页表还是页目录中的页表项，低12位必然为0，同样的，intel将这12位定义为属性位。

#### 页目录寄存器cr0

这个寄存器和gdtr寄存器类似，存放的是一个页目录的基地址。在多任务中，每一个任务都可以有一个页目录，这就为虚拟内存提供了硬件支持。每当任务切换的时候，硬件会自动加载新任务的页目录到cr0寄存器。

### 保护

#### 特权级保护

386CPU对段进行严格的特权级保护，防止代码任意访问重要的段。

**DPL：描述符特权（Descriptor Privilege Level）**

这个属性存储在段描述符的权限位，它用于代表这个段的特权级别，也是访问这个段所需要的级别。一个程序可能会有很多的段，在初始化这些段的时候就应该设定好每个段的特权级，而且在以后的过程中这个级别应该是不被更改的。DPL是一个最主要的安全属性，是可以访问与否的标杆。特权级的检查往往是用别的安全属性和这个属性相对照来确定是否具有访问该段的资格。

**RPL：请求特权级RPL(Request Privilege Level)**

这个属性存储在selector的第零位和第一位。这个属性说明的是当前代码如果想访问其他段的时候所需要提供的请求权限。这个属性的值很特殊，因为它是由操作系统的编写者自己确定的，但是使用起来确实和CPL配合使用的。想要访问一个段，要检测请求特权和当前特权，这两个值只要有一个大于要访问段的DPL，那该段就拒绝你的访问。之所以有了当前特权级还要引入这样一个属性，是为了防止有些程序利用高的DPL来非法的访问段，系统会对这种程序很低的请求特权级，这样计划访问搞特权及的段的意图就会被阻止。

**CPL：当前任务特权级（Current Privilege Level）**

该属性代表此时所在运行的程序的级别。这个属性保存在代码段寄存器的第0位和第1位，是相对于代码段寄存器而言的。当一个选择子被加载到它以后，选择子中的请求特权级就变成了正在执行的特权级。所代表的意思就是请求得到了满足，因此级别就是请求级别。

因为只有一个段寄存器，所以当前任务特权级也只有一个。这和请求特权级不同，请求特权级可以有很多个。

当要访问一个段的时候，CPU要进行特权级检查以保证该段不被非法读取,检查特权级的规则为:

数据段:数值上cpl和rpl要小于目标段的dpl,才可以访问该数据段。既高特权级可以访问低特权级的数据段。

非一致代码段：当前特权级必须与目标段的段特权级相同才可以访问。

调用门：cpl与rpl必须高于们选择符的dpl，而又必须小于门缩影数的数据段的dpl。

#### 特权级的转换

特权及发生变换时,相应的堆栈也要发生变化,例如当任务在3特权级跳到0特权级时(比如调用系统函数)，为了系统的安全，CPU不允许使用任务自己的堆栈，而必须使用一个任务在0特权级下专用的堆栈。每个堆栈的地址都记录在任务自己的tss上，当发生特权级转换的时候，cpu和自动寻找对应特权级的栈，然后将原来栈中的内容复制到新的栈里面。并且把原来的栈地址也压入新的栈中。而当特权级变回原来的时候，cpu先出栈以获取冤栈的地址，然后再把新栈的内容复制到原栈中。这些都是由硬件自动实现的，而栈的切换对于用户来说是透明的。应为intel规定有四个特权阶级存在，一个任务除了自己的栈以外，还要额外准备三个栈以供三个特权升级变换使用。

## 中断和异常处理

386沿用了中断响亮这一理念，但是做出了变化，中断用一个叫中断描述符表（idt）的表来映射相应的中断处理程序。比起386以前的中断向量，这个变化有着这样的优势：中断描述符表的地址和长度存放在寄存器中，这样就允许idt在内存的任意位置存放。其次每个中断都映射到相应的段描述符，这样可以实现特权级检查，防止非法调用中断。每个中断映射实质上是一个门，门中存放着中断处理程序所需在的段选择子和偏移：

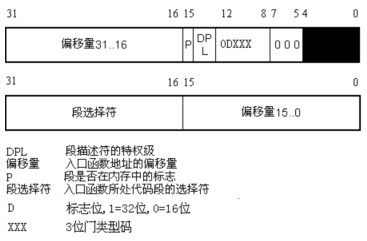


图3-2-1 中断描述符

386中的中断分为异常和中断，中断号在0—31之前的都是异常，其后为中断。

异常有三种类别：

错误：这是一种可以被改正的异常。当发生错误时，系统会将正在执行的语句保存，当错误被处理以后，再重新这行这条语句。

陷阱：与错误相似，也是一种可更正的异常。当时当一场被处理以后，执行的是发生一场语句的下一条语句。

终止:这是一种无法被改正的异常，遇到这种异常说明程序出无法修正的错误，只能将任务终止。

## 第三节 任务管理

386引入任务的概念使得多任务从硬件上得到了支持。在发生任务切换时，硬件会自动保存前一个任务的信息，而恢复新任务的信息。所有信息都保存在一个叫tss的数据结构中：

|  |  |  |
| --- | --- | --- |
| Not used | Back link | |
| PL0 ESP | | |
| Not used | PL0 SS | |
| PL1 ESP | | |
| Not used | PL1 SS | |
| PL1 ESP | | |
| Not used | PL2 SS | |
| CR3 | | |
| EIP | | |
| EFLAGS | | |
| EAX | | |
| ECX | | |
| EDX | | |
| EBX | | |
| ESP | | |
| EBP | | |
| ESI | | |
| EDI | | |
| Not used | ES | |
| Not used | CS | |
| Not used | SS | |
| Not used | DS | |
| Not used | FS | |
| Not used | GS | |
| Not used | LDT | |
| I/O BITMAP | NOT USED | T |

表3-3-1 TSS

NOT USED为Intel的保留位，在编程的时候这些地方我们最好要填0。

这个数据结构中详细的保存了任务的各种信息:

该数据结构中的最高位表示是否有任务链。任务不可以冲入但是可以嵌套，因此有可能是一个任务唤醒了另一个任务，back link表示该任务是否是嵌套的。

PL0 SS表示当任务切换到特权级为0的时候用到的栈，后面两个PL1和PL2同理。

Cr3保存了任务自己的页目录，这个对于虚拟内存有很大帮助。

随后的数据保存的是各种通用寄存器的值。

任务转换的时候CPU的具体步骤是：

从jmp或者call命令的段地址中找出tss的段选择符。

检查特权级来判断是否允许切换任务。（但是异常和非INT命令引发的中断不检查特权只直接切换任务）

检查新的任务的描述符是否存在并且段的长度要大于67H（任务的描述符的大小为104字节，允许用户延长它已加入更多数据）

把旧任务的b位（Busy）值改变，其中如果是jmp引发的任务转换，则b=0，如果是call引发的任务转换则b=1。

如果任务由iret产生，则CPU会把EFLAGS的NT位复位，其他情况NT不变。

把当前任务的状态保存到当前的tss中，然后取出新任务的tss，将新tss的信息加载到寄存器中，既恢复现场。

如果任务是由call，异常或者中断引发，就将新的标志寄存器的NT变为0，其他情况该位不变。

设置新任务的B（Busy）=1。（如果是有iret引发的任务切换则B位不变）

将新的tss加载到tr，加载页目录到CR0。

关于NT位和任务链的相关知识比较复杂而且本文中并未用到，因此就不提及了。

# 第四章 具体代码实现

整个代码实现这样一个过程：首先由BOIS将引导程序加载到物理内存7c00H处，然后引导程序建立段描述符表，并且打开A20。在这些初始化都完成以后，我们要做的是打开位于CRO的protect位开关,最后进入保护模式。在boot阶段就进入保护模式的原因是哦我要把内核加载到内存地址为100000H的高地址，这个地址只有保护模式才能访问。接着调用显示函数在屏幕上显示先关信息，同时调用自己编写的硬盘驱动将硬盘中的内核加载到内存地址100000H处。成功后跳转进内核。

在内核中第一步要重新建立段描述表以方便管理，然后建立中段描述符表实现打印中断，时钟中断等系统中断，并且设置好外部中断芯片8259a。中断建立以后，就开始核心部分：建立多任务。建立多任务以前要将所需要的数据结构都准备好,包括：在全局描述附表（GDT）中建立两个tss段描述符，俩个LDt段描述符，然后建立两个tss，两个LDT并分别与之前建立的描述符映射。还要为每个任务建立两个栈，一个自己使用（3特权级），一个供代码切换到0特权级时使用。最后，还需要两个人物的代码，我这里简单起见设置为一个任务循环打印字母’a’，一个任务循环打印字母’b’。最后要建立堆栈伪造特权级为3的任务现场，以实现从特权级0降如特权级3。

整个程序在运行时的流程是：系统运行一个任务（循环打印），当时钟周期到了以后引发时钟中断，时钟中断触发任务管理程序，有任务管理程序选择该切换到哪个任务运行，最后再把控制权交给新的任务，然后等待下一个时钟周期，如此循环。如今的操作系统实现多任务，就是靠的这个机制。

## 第一节 引导部分

### 一、 制作引导扇区

开机启动后系统进行BOIS自检，当各项外设均能正常工作后，BOIS检查引导扇区（即0道0面1扇区）是否为可用引导扇区，但判定最后两字节为55aaH时，即认为该扇区是可用的引导扇区，然后将此扇区的数据复制到内存地址7cooH处。因为历史原因，我们的引导扇区代码的地址并不是0，可采用两种方法：可设置短地址为07c0H，偏移地址为0，既cs=7c0，ip=0。本文采用第二种方法，使用伪指令org。org伪指令的作用是提示编译器此代码以org以后的数字作为段的起始地址。淫才在我们引导扇区代码的第一句有这行代码

Org 7c00h

引导扇区的作用是进入保护模式，并将内核代码加载到物理地址为1M处，然后跳转至内核代码。

为了最后两字节为55aaH，我们需要知道自己的代码有多长，并在代码结束后填充足够的0，保证最后此文件大小为512b，这样拷贝到0道0面1扇区才能保证此扇区是一个有效的引导扇区。因此我们需要使用电小技巧来填充文件：

times 510-($-$$) db 0

dw aa55H

在nasm语法中$代表他所在的行在代码中的偏移地址，$$代表他所在段的起始地址，$-$$正好计算出了有效代码的长度，然后填充510-($-$$)个0，最后两字节为55aaH。注意因为intel规范是地址从小到大，而数据表示是从大到小，因此使用dw的话是dw aa55H，而是用db的话，代码为：db 55H，0aaH。

### 二、 进入保护模式

想要进入保护模式，首先需要建立段描述符表，在表中，我们使用到四个段：

codeSegment:：指向引导代码，既指向自己，因为进入保护模式后寻址发生变化，因此即使在转换模式前后执行的是同一代码段，也要在保护模式后也要重新跳转到一个段地址才能继续执行，因此需要一个段来指向他自己才能保证代码的连续执行。

kernelSegment：这个段指向物理地址为1M的高地址处，并将内核代码加载到这个段中。因为在加载中数据复制的目标必须是数据段，而代码执行的目标段必须是代码段，所以此段必须为数据段，还必须有一个代码段同样指向内核代码，及两个段指向的是同一个代码，但是一个是数据段，一个是代码段。

kernelSegment2：这个段就是指向内核的代码段，前面已经介绍过，不再赘述。

videoSegment：该段指向物理地址为b8000H处，此地址为显存地址，代码中我们将向此段中写入数据以实现在屏幕上输出字符。

由于段描述符的数据结构很凌乱，因此在这里我用到了一个制作描述符的宏pm.inc：

; 描述符类型

DA\_32 EQU 4000h ; 32 位段

DA\_DPL0 EQU 00h ; DPL = 0

DA\_DPL1 EQU 20h ; DPL = 1

DA\_DPL2 EQU 40h ; DPL = 2

DA\_DPL3 EQU 60h ; DPL = 3

; 存储段描述符类型

DA\_DR EQU 90h ; 存在的只读数据段类型值

DA\_DRW EQU 92h ; 存在的可读写数据段属性值

DA\_DRWA EQU 93h ; 存在的已访问可读写数据段类型值

DA\_C EQU 98h ; 存在的只执行代码段属性值

DA\_CR EQU 9Ah ; 存在的可执行可读代码段属性值

DA\_CCO EQU 9Ch ; 存在的只执行一致代码段属性值

DA\_CCOR EQU 9Eh ; 存在的可执行可读一致代码段属性值

; 系统段描述符类型

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

SA\_RPL0 EQU 0 ; ┓

SA\_RPL1 EQU 1 ; ┣ RPL

SA\_RPL2 EQU 2 ; ┃

SA\_RPL3 EQU 3 ; ┛

SA\_TIG EQU 0 ; ┓TI

SA\_TIL EQU 4 ; ┛

%macro Descriptor 3

dw %2 & 0FFFFh ; 段界限1

dw %1 & 0FFFFh ; 段基址1

db (%1 >> 16) & 0FFh ; 段基址2

dw ((%2 >> 8) & 0F00h) | (%3 & 0F0FFh) ; 属性1 + 段界限2 + 属性2

db (%1 >> 24) & 0FFh ; 段基址3

%endmacro ; 共 8 字节[1]

这个宏看起来复杂，其实非常简单，首先将段描述符的各种属性都宏定义，然后要罪的只不过是把填入的段基址，段偏移，属性这三个参数重新组合到正确的位置罢了。这样，我们一句

Descriptor 0x100000h, 0ffffh, DA\_32 +DA\_C

就完成了创建一个描述符的操作，非常简单。

另外还有一个创建们描述符的宏，与这个宏极其相似，我在这里就不描述了，有兴趣的可以到pm.inc里一探究竟。

段描述符表做好以后，需要将表的地址和大小加载到寄存器gdtr中，加载段描述符表的语法为lgdt m48，因此需要一个48位大小的内存存放表的地址和大小：

gdtPtr dw gdtLen-1 ;表的最大偏移

ddgdt ;表在内存中的地址

注意还是数据在内存中摆放的问题，高地址在后，低地址在前。然后调用命令：

lgdt [gdtPtr]

将段描述符表加载到相应的寄存器中。然后我们打开保护模式的开关：

mov eax,cr0

or eax,1

mov cr0,eax

然而这样还不行，再试模式下最大寻址范围为0FFFFH，即只有20根地址线，而保护模式支持4GB寻址，由32跟地址线，后12跟地址线在实模式中是关闭的，我们要打开它：

In al,92h

or al,00000010b

out 92h,al

关于为什么打开地址线是调用键盘借口可查阅相关资料（这是一个历史问题）。

为了代码能够继续执行，我们要跳转到保护模式下目前代码所在的段，但是实模式的段间跳转和保护模式的段间跳转命令在指令的数据部分位数不同，直接在实模式下段间跳转的话，偏移位数会被截断，这就是一个纠结的问题：要再试模式下使用保护模式的命令，在linux中，linus直接写入了保护模式段间跳转的机器码，而我们用如下方法：

jmpdword 8:next-7c00h

这种语法为nasm独有，可以把偏移强制编译为32位而不会被截断。这句jmp后，系统就完完全全的跳入到保护模式中了。

### 三、 加载内核代码

要加载内核代码，必须与硬盘发生数据交换，最简单的办法是调用BIOS中断将硬盘数据加载到内存中，但是进入保护模式后，代码变为32为格式，因此BIOS无法使用，因此我们不得不自己写驱动来实现从硬盘读取数据。硬盘有6个端口，分别为1f2H到1f7H，其中1f2H存放要读取的扇区数， 1f3H~1f6H存放要读取的起始扇区号， 1f7H为状态端口：

mov dx,1f2h

mov ax,12

outdx,al

mov dx,1f3h

mov ax,1

outdx,al

inc dx

mov al,0

outdx,al

mov ax,0

inc dx

outdx,al

inc dx

mov al,11100000b

outdx,al

因为扇区号是28位，因此1f6H端口的高4位空出，作为标志位，因此mov那一句命令的意思为将最高四位置为1110，代表主盘LBA格式。

之所以用LBA格式是因为简单，传统的CHS格式就是怎是的读取此盘手段，它符合硬盘的事实：读完0面0磁道后是要读取1面0磁道的。但是这样就是的夺取的瞬息不是逻辑连续的，给变成带来一些麻烦，程序员不得不汉族与磁盘面的问题。于是硬盘厂家引入了LBA规范，把实际上不连续的磁道变为逻辑上连续的磁道。我们顺序读取0、1、2磁道，实际上读取的是0面0磁道，1面0磁道，22面0磁道。这样的逻辑连续给了我们很大的方便。在Linus编写硬盘驱动的时候使用的是CHS规范，但是日新月异，我们使用更方便的LBA规范。

接着我们监测1f7H端口是否忙碌，如果不忙碌并且准备就绪则开始读取数据：

inc dx

mov al,20h

outdx,al

mov dx,1f7h

.waits:

inal,dx

and al,88h

cmp al,8h

jne .waits

要读取的数据存放在1f0H端口，因此我们需要循环的读取此端口的内容并把数据写入内存：

movax,kernelSegment-gdt

moves,ax

mov di,0

mov ecx,256\*12

mov dx,1f0h

.readw:

inax,dx

mov [es:di],ax

add di,2

loop .readw

为了实现代码重用的目的，我们将常用的操作最好写为函数，但提到函数就要涉及到传参的问题。为了与现代的规范接轨，现在一般的惯例是采用c语言的传参标准，也就是利用栈来传递参数，其规范为：

第一步：逆序将参数入栈

PUSH PRAM3

PUSH PRAM2

PUSH PRAM1

第二步：实现跳转的时候要压入段地址（如果是段内跳转不用段地址）和eip：

PUSH CS

PUSH EIP

第三步：将ebp的值压栈保存，因为在函数体内部需要用ebp来索引个数据，所以先要把ebp原来的值保存起来。

第四步：把esp的值复制给ebp。

第五步：如果函数内由局部变量，就把局部变量压栈

PUSH PRAM3

PUSH PRAM2

PUSH PRAM1

当需要调用参数的时候，可以用[bp+4]这种方式来索引。当函数返回的时候，要恢复栈的状态。

首先需要将bp的值赋给sp，这样就清除了局部函数（这就是为什么在c语言中函数内部的变量会消失），然后使用ret n命令返回到源代码。ret n的作用为返回的同时清除要入栈中的参数。

### 四、 显示字符

当内核加载完成后，我们要在屏幕上打印一行字符，提示我们进入内核。想要在屏幕上打印字符，需要记录光标位置，此处有两种方案，一是开辟一个变量用来记录光标位置，二是使用显卡自带的光标位置寄存器，本文采用后者。VGA的寄存器中多，在本文中我们只用到3d4H和3d5H两个寄存器，其中3d4H是索引寄存器，将要读取的端口索引号写入这个端口，就可以在3d5H这个端口中读到相应端口的数据。其中光标位置寄存器的索引号为0eH和0dH， 0eH存放光标位置的高八位， 0dH存放低八位：

pushbx

mov dx,3d4h

mov al,0eh

outdx,al

mov dx,3d5h

inal,dx

movah,al

mov dx,3d4h

mov al,0fh

outdx,al

mov dx,3d5h

inal,dx

这样就把光标的位置保存到了寄存器ax中。同理，我们可以修改光标的位置以实现光标后移，换行，清屏等操作。

在vga标准中，屏幕大小80\*25个字符大小，每一个字符的属性由两字节表示。其中高字节存放字符的各种属性，比如说背景，颜色和一些显示效果。后一个字节存放字符的ASCII码。因此如果想要在屏幕上显示字符，首先要获取光标位置，然后将要显示的字符的ASCII码写入到光标位置\*2的内存地址，再将光标数+1就可以了。

## 第二节 内核部分

### 一、 建立任务

内核部分的主要作用是先重建gdt。然后再建立中断表，对8259a芯片重新编程以改变中断号，然后建立两个任务，这两个任务的功能是分别向屏幕打印a和b，以证明系统的确在执行多任务。以为我们要运行的俩个任务都是三特权级的，而正常的操作系统显存都是受保护的，只有0特权级才可以访问，三特权级无法访问显存段，所以我们要建一个门来使低特权级通过门来调用打印字符函数：

mov ebx,80\*8+idt

mov word [ebx+2],codeSegment-gdt

moveax,putchar

mov word [ebx],ax

shr eax,16

mov word [ebx+6],ax

moval,byte [ebx+5]

add al,01100000b

mov byte [ebx+5],al

这段代码的意思是把80号中断设为系统调用中断，低特权级程序需要调用这个中断来实现打印字符。门中的段地址为系统代码段，偏移地址为putchar，即为打印函数的偏移。伺候所用打印字符的调用都要使用int 80命令。

在putchar例程的编写中要注意这样一个地方：为了实现滚屏我们不妨设置一个清屏功能，最简单的思路就是把显存中高8为设置成为零，这样就什么也不现实了，这个思路没错，但是想想在这之后会发生什么呢：我们无论输出什么字符，在屏幕上都不会有所显示。这个小bug的原因是再后来我们虽然将要输出的字母用ASCII码的形式写入到显存的低8位中，但是不要忘了因为清屏例程把属性设置为了黑底黑字，而我们后来的程序只写入ASCII码去不管属性，这样虽然显示器的确显示了我们要输出的字符，但是黑底黑字，说我们看不见，所以以为程序有问题没有输出文字。这个小bug会把人误导到别的错误上，需要注意一下。

要建立两个不同的任务,我决定写两个无限循环，一个不断打出字符a，一个不断打印字符b：

task0:

movcl,'a'

int 80

jmp tss1Segment-gdt:0

jmp task0

task1:

movcl,'b'

int 80

jmp tss0Segment-gdt:0

jmp task1

俩个任务需要两个tss，每个任务需要2个栈，一个自己使用的栈，一个当任务特权级改变为0的时候使用的栈，因此一共需要四个栈，每个任务需要一个ldt选择符，一个ldt，其中我们打算利用堆栈伪造返回现场来实现从特权级0降到特权级3，因此任务0的tss的一些部分可以不写，而任务2则是正常的任务切换，因此任务2的tss一定要仔细填写完整:

dd0,tss1Stack,10h,0,0,0,0,0,task1,200h,0,0,0,0,usrStack1,0,0,0,17h,0fh,17h,17h,17h,017h,ldt1Segment-gdt,8000000h

注意EFLAGS位，这个地方要特别注意，因为再三特权级下EFLAGS寄存器是受保护的，因此填写什么数据取决于你在什么特权级下进行任务切换，如果为3特权级下任务切换，加载tss是要把EFLAGS加载，而如果不小心修改了IF位会引起异常，我在这里就遇到过这个问题，而且差错查了好几天，这个错我非常隐蔽，一定要注意。

当代码跳转发生特权及转换的时候，处理器先将任务本身的栈段选择子和栈指针压栈，然后压入当前正在执行的语句的地址，最后跳转的高特权级继续执行指令。当指令结束后，在依次将栈中的数据出栈一番回到原来的代码。我们就利用这一点实现从特权级0降到特权级3——先堆栈，然后返回（在此之前的当然要把代表这个任务的tss加载到任务寄存器，把代表该任务的中断表加载到ldtr寄存器）：

mov ax,tss0Segment-gdt

ltr ax

mov ax,ldt0Segment-gdt

lldt ax

push ldt0data-ldt0+7

pushsystemStack

push ldt0code-ldt0+7

push task0

retf

自此以后，系统就完全运行在3特权级中了。这种从高特权及降低到低特权级的手法只需要使用一次，以后的任务转换，无论是压栈，还是将各种tss和ldt加载到相应的寄存器都是有CPU自动完成的，不再需要我们操心了。

到了task0代码段，系统执行完中断命令后，在屏幕上打印一个字符，然后执行jmp跳转任务。因为跳转的段选择子定义为的是一个tss选择子，而tss中存放着程序的入口地址，因此偏移地址必须要写，但是写什么都无所谓：tss1Segment-gdt:0

这里有一个疑问：任务就这样跳到了另一个任务，那在跳转代码结束以后的代码岂不是永远得不到执行？我这个问题也纠结了好久，究其原因是对任务切换的不理解。当执行任务切换的时候，CPU会把现场的各种数据保存起来，其中就包括eip。当再次跳转回这个任务的时候，CPU回再把eip拿出来继续执行，因此jmp语句后面的是能够得到执行的。这个问题好多新手都有，这里强调一下。

到此为止，我们就实现了任务切换，任务0打印一个a，然后跳转到任务1，任务1打印一个字符b，再跳到任务0，如此循环。

### 二、 建立中断

首先我们要让计算机能够响应外部中断。8259a的默认中断号是从8开始，但是386以后intel为中断制定了规范：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 向量号 | 助记符 | 说明 | 类型 | 错误号 | 产生源 |
| 0 | #DE | 除出错 | 故障 | 无 | DIV或IDIV指令 |
| 1 | #DB | 调试 | 故障/陷阱 | 无 | 任何代码或数据引用，或是INT 1指令 |
| 2 | -- | NMI中断 | 中断 | 无 | 非屏蔽外部中断 |
| 3 | #BP | 断点 | 陷阱 | 无 | INT 3指令 |
| 4 | #OF | 溢出 | 陷阱 | 无 | INTO指令 |
| 5 | #BR | 边界范围超出 | 故障 | 无 | BOUND指令 |
| 6 | #UD | 无效操作码（未定义操作码） | 故障 | 无 | UD2指令或保留的操作码。（Pentium Pro中加入的新指令） |
| 7 | #NM | 设备不存在（无数学协处理器） | 故障 | 无 | 浮点或WAIT/FWAIT指令 |
| 8 | #DF | 双重错误 | 异常终止 | 有（0） | 任何可产生异常、NMI或INTR的指令 |
| 9 | -- | 协处理器段超越（保留） | 故障 | 无 | 浮点指令（386以后的CPU不产生该异常） |
| 10 | #TS | 无效的任务状态段TSS | 故障 | 有 | 任务交换或访问TSS |
| 11 | #NP | 段不存在 | 故障 | 有 | 加载段寄存器或访问系统段 |
| 12 | #SS | 堆栈段错误 | 故障 | 有 | 堆栈操作和SS寄存器加载 |
| 13 | #GP | 一般保护错误 | 故障 | 有 | 任何内存引用和其他保护检查 |
| 14 | #PF | 页面错误 | 故障 | 有 | 任何内存引用 |
| 15 | -- | （Intel保留，请勿使用） |  | 无 |  |
| 16 | #MF | x87 FPU浮点错误（数学错误） | 故障 | 无 | x87 FPU浮点或WAIT/FWAIT指令 |
| 17 | #AC | 对起检查 | 故障 | 有（0） | 对内存中任何数据的引用 |
| 18 | #MC | 机器检查 | 异常终止 | 无 | 错误码（若有）和产生源与CPU类型有关（奔腾处理器引进） |
| 19 | #XF | SIMD浮点异常 | 故障 | 无 | SSE和SSE2浮点指令（PIII处理器引进） |
| 20-31 | -- | （Intel保留，请勿使用） |  |  |  |
| 32-255 | -- | 用户定义（非保留）中断 | 中断 |  | 外部中断或者INT n指令 |

表4-2-2 Intel中断规范[3]

由此可见从8开始的中断已经被定义了,因此我们需要修改外部中断的中断向量号,一般是把外部中断放到32号开始的位置,由表可知32号开始的中断允许用户自定义。要修改向量号，必须对外部中断芯片进行重新编程，要编程需要先了解该芯片的几8位个寄存器,因为该芯片室友俩个芯片构成,因此需要对两个芯片的端口都进行操作该芯片只有两个端口，但是在不同时间写入不同的数据，意义是不一样的，因此对该芯片端口写入的顺序一定不能乱:

往20H和0a0H写入ICW1

往21H和0a1H写入ICW2

往21H和0a1H写入ICW3

往21H和0a1H写入ICW4

往21H和0a1H写入OCW1

往21H和0a1H写入OCW2

这个顺序不可更改。

其中：

ICW1(对应端口为:主片20H,从片0a0H)

8位寄存器，各位的还以如下：

0：是否需要ICW4。该位为0时不需要，为一时需要

1：0-有从片，1-只有一个芯片

2：0-八字节中断向量 1-四字节中断向量

3:0-边缘触发模式 1-波形触发模式

4:此位必须为1

5，6，7：pc机必须为0

ICW2(对应端口为：主片 21H，从片0a1H)

0,1,2：都为0表示80x86系统

3,4,5,6,7：80x86中断向量

ICW3(主片 21H)

这8为哪一位为1说明在该8259a芯片的那一个端口上连有从片。

ICW3(从片0a1H)

2,1,0：这三位表示与主片那一个端口相连

3,4,5,6,7：必须为0

ICW4(对应端口为：主片 21H，从片0a1H)

0：0-MCS 80/85 1-80x86

1：0-自动EOI 1-正常EOI

2,3：主/从缓冲模式

4:0-sequential模式 1-SFNM模式

5,6,7：未使用，填0

OCW1(对应端口为：主片 21H，从片0a1H)

哪一位为1，说明该片上的那一个中断被屏蔽。

OCW2(对应端口为：主片 21H，从片0a1H)

5：EOI

其他各位我们可以不必要了解。

EOI是一个信号，外部中断在接受这个信号之前是不在响应这个中断的，只有接收到EOI信号以后才会重新响应这个中断。

init8259a:

mov al, 011h

out 020h, al ; 主片, ICW1.

nop

out 0A0h, al ; 从片, ICW1.

nop

mov al, 020h ; IRQ0 对应中断向量 20H

out 021h, al ; 主片, ICW2.

nop

mov al, 028h ; IRQ8 对应中断向量 0x28

out 0A1h, al ; 从片, ICW2.

nop

mov al, 004h ; IR2 对应从片

out 021h, al ; 主片, ICW3.

nop

mov al, 002h ; 对应主片的 IR2

out 0A1h, al ; 从片, ICW3.

nop

mov al, 001h

out 021h, al ; 主片, ICW4.

nop

out 0A1h, al ; 从片, ICW4.

nop

这段代码的作用是将该芯片的外部中断映射到32号中断以后。首先我们要新建一个变量current拉存放当前正在执行的任务号：

current dd 0

以后每次切换任务，都把要执行的任务号填写到这个变量里面。

然后我们先编写一个处理任务切换的函数，然后把它设置为一个系统中断：

timeInterrupt:

movax,dataSegment-gdt

movds,ax

mov eax,1

cmpeax,[current]

je .1

mov [current],eax

jmp tss1Segment-gdt:0

jmp .2

.1:

mov byte [current],0

jmp tss0Segment-gdt:0

.2:

mov al,0x20

out 0x20,al

iret

这个中断的作用是查看当前的任务号，如果是1，那就说明应该调到任务0，反之亦然。当然我们现在只有两个任务，可以这么写。

mov al,0x20

out 0x20,al

这两句的作用是向系统发送中断响应信号，因为我们计划开启时钟中断，每到一个时钟周期就调用timeinterrupt函数来切换进程，所以为了响应外部中断就必须发送这个信号，否则时钟中断只会发生一次，许多刚接触中断的人都会忘记为外部中断的处理程序发送该信号，造成中断死锁。

现在让我们试验一下利用时钟中断来实现任务切换。首先要把时钟中断打开：

mov al, 11111110b ; 仅仅开启定时器中断

out 021h, al ; 主片, OCW1.

nop

mov al, 11111111b ; 屏蔽从片所有中断

out 0A1h, al ; 从片, OCW1.

nop

其次在任务开始前打开外部中断响应：

sti

将任务2的tss中的eflags寄存器修改为20H0，表示低9位置为1，既响应外部中断。

两个任务循环打印字符的速度是很快的，这使得屏幕可能已经好几页a了才看到b，有好几页b了才看到a，影响效果，因此我们要加入循环使打印延时（这里设置的延时高于了时钟中断的延时，这可能会导致在任务切换以后恰巧进入的是循环语句，再循环语句结束之前由切换到原来的任务，使得屏幕上出现两个连续的相同字母这种现象，这种现象无论从思路还是功能上都没有错误，只是在现实的时候显得不如abab这样循环来的好看）。新的任务代码如下：

task0:

movcl,'a'

int 80

mov ecx,100000

.s:

loop .s

jmp task0

task1:

movcl,'b'

int 80

mov ecx,100000

.s:

loop .s

jmp task1

最后，我们需要将所写的代码编程二进制文件，并且将这个二进制文件写入镜像中，这样让bochs加载这个镜像竟可以看到结果了：

dd if=boot.bin of=c.img bs=512 conv=notrunc

dd if=kernel.bin of=c.img seek=1 conv=notrunc

dd命令的格式为 dd if=源文件 of=目的文件 [参数]…

其中bs参数表示写入的字节数，因为引导扇区是512字节，所以bs=512（其实我们在编写代码的时候已经将文件大小控制成512字节了，所以这个参数是可以省略的）。而参数seek表示要拖个目标文件的若干个块以后再写入，因为我们把kernel放在了第二个扇区，自然要跳过一个扇区（跳过的是引导扇区）。有时候dd命令发现写入的目标是一个大小可变的文件（比若说我们这里的img镜像文件）后，会自作聪明的在写入文件完成后把写入之后的部分拦腰截断，但是我们的镜像文件是模拟一个软盘，因此不能让他自作主张拦腰截断，这里就要用到参数conv，当这个参数的值为notrunc的时候，dd命令就不会阶段文件了。了解完这些以后，就可以把一些列命令都写入makefil以后就不用一遍一遍的输入命令了。

镜像做好以后，就要用Bochs架在。在加载之前，首先应该确定它的配置。Bochs由中断参数，Memory选项是选择内存大小，这里选择内存为32M，虽然32M的内存在现在看来是古董了，不过这么大的空间足够这个程序用了。在Disk&Boot选项中，我们设置从硬盘启动，把硬盘的路径设置为我们刚才制作的img格式的路径，注意不要要把从软盘启动这个选项错选了，好多教材都是教的制作软盘再从软盘启动，但是我觉得既然软盘已经被淘汰了，我们也要与时俱进放弃从软盘启动这种方法。一切准备妥当以后，就可以点击start启动我们的程序了，经过一番段在的BOIS自检，我们引导扇区的boot程序打出了字符：This os is made by Liu Yafei。这证明我们的内核已经加载完成了，然后系统开始通过时钟中断来实现任务切换，当屏幕上不停地打印出字母‘a’和字母‘b’的时候，我们就知道我们所编写的能实现任务切换的OS已经成功运行了。

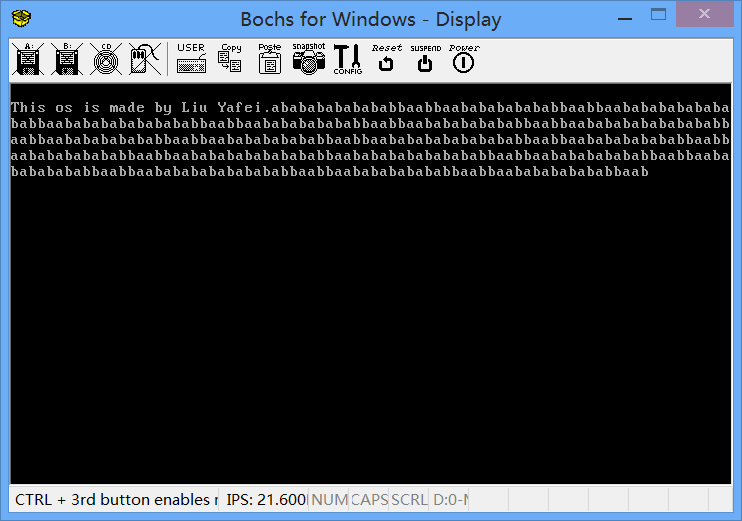


图4-2-1 任务切换效果

# 结 论

Dropbox的常使人说个这样一句话：“编程是获得超能力的最接近的一种方法了。”想想看，在一天电脑前你就是上帝，用这个机器你可做很多神奇的事。当Linus在在论坛上说自己已经做好了一个开源的操作系统的时候，他的心情是怎样的呢？如今的编程环境充斥着效率，人们已经完全不关注底层，认为底层是一个很复杂很神秘的东西，有的人甚至写得一手好代码，几分钟就能做出一个程序，一个网站，但是根本不知道计算机是怎么实现的。如今人们追求平台无关性以追求简洁与效率，但副作用就是我问所做的东西好像都像空中楼阁一样，优化困难，如果想要控制一些硬件操作也非常难，最重要的是已经丝毫没有什么自由度可言了。

我所编写的这个程序非常简陋，甚至称不上是个操作系统，但是麻雀虽小五脏俱全，无论是分段分页，保护异常，任务切换，这都是现代操作系统的基础。对于一个hacker来说，无论是组织硬件交互，读写内存，还有代码的优化，都是一件非常享受的事情，把你的想法写出来，就有一个笨家伙去忠实的执行他。有人说底层神秘底层复杂，而事实恰恰相反，底层是最简单朴素的，他不懂预测不会变通，就是按部就班的执行，执行再执行。

学习汇编要与许多数制打交道，其中最多的就是十进制与十六进制。在编写代码的时候一定要注意，有时候本来意愿是写一个十六进制的数字，但是忘了在数字后面加’h’，这样就变成了十进制导致错误。

想要编写操作系统内核，Linux源代码是必须要看的，但是该操作系统采用的是AT&T语法，而我在这里使用的是Intel语法，这两个语法最纠结的地方就是AT&T语法的原操作数在右面，目的操作数在左边，而Intel则恰恰相反。有时候源代码看多了该自己时间，很容易搞错所以一定要注意。

好多人和我一样是从masm学汇编的，虽然masm和nasm都遵循Intel标准，但是还是有一些不一样，而我是用的编辑器是notepad++，这个编辑器只要是汇编语法都加亮，但是要注意，有些语法是nasm不支持的但是他还是会加亮，所以身边最好备一本规范手册。

有的时候出现bug实在解决不了，一定要借助网络求助，我曾经追一个bug追了3天也查不出来，后来无奈上CSDN和stackoverflow上面求助，刚问出去就得到回答，并且说这些错他们也遇到过，是经验之谈。这要比自己埋头找bug来得实在。

Linux在编写像我们这种需要和底层打交道的程序的时候有着巨大的便利，我在今后的代码编写中一定会改成Linux操作系统。

通过这次学习与实践，我感到操作系统不再神秘，而这个作品也没有结束，我要在今后的日子里继续完善它。

# 参考文献

[1] 王爽. 汇编语言. 北京:清华大学出版社.2003

[2] 于渊.ORANGE’S：一个操作系统的实现.北京：电子工业出版社.2009.

[3] 赵炯.Linux内核完全剖析-基于0.12内核.北京:机械工业出版社.2009

[4] 李忠,王晓波,于洁.x86汇编语言:从实模式到保护模式.北京:电子工业出版社.2013

[5] （美）马兹迪（Mazidi,M. A.）,（美）马兹迪（Mazidi,J. G.）,（美）考西（Causey, C.）著;高升译.x86 PC汇编语言、设计与接口.北京:电子工业出版社.2001.1

[6] [(日)川合秀实](http://search.china-pub.com/s/?key1=(%c8%d5)%b4%a8%ba%cf%d0%e3%ca%b5) ;[周自恒](http://search.china-pub.com/s/?key1=%d6%dc%d7%d4%ba%e3&zyandor=and) [李黎明](http://search.china-pub.com/s/?key1=%c0%ee%c0%e8%c3%f7&zyandor=and) [曾祥江](http://search.china-pub.com/s/?key1=%d4%f8%cf%e9%bd%ad&zyandor=and) [张文旭](http://search.china-pub.com/s/?key1=%d5%c5%ce%c4%d0%f1&zyandor=and)译.30天自制操作系统.北京:[人民邮电出版社](http://www.china-pub.com/love/chubanshe/chubanshe.asp?newid=141).2012

[7] 于渊.自己动手写操作系统.北京:电子工业出版社.2005

[8] 杨季文.80X86汇编语言程序设计教程.北京:清华大学出版社.1998

[9] 李建伟,刘金河.实用操作系统教程.北京:清华大学出版社.2011

[10] 唐朔飞.计算机组成原理.北京:高等教育出版社.2008

[11] 曹加恒.新一代汇编语言程序设计.北京:高等教育出版社.2003

[12] [(美)ANDREW S.TANENBAUM](http://book.douban.com/search/%28%E7%BE%8E%29ANDREW%20S.TANENBAUM) / [ALBERT S.WOODHULL](http://book.douban.com/search/ALBERT%20S.WOODHULL).操作系统设计与实现.北京: 清华大学出版社.2008

[13] [程杰](http://book.douban.com/search/%E7%A8%8B%E6%9D%B0).大话数据结构.北京: 清华大学出版社.2011

[14] [潘嘉杰](http://book.douban.com/search/%E6%BD%98%E5%98%89%E6%9D%B0).易学C++.北京: 人民邮电出版社.2008

[15] [布鲁姆](http://book.douban.com/search/%E5%B8%83%E9%B2%81%E5%A7%86).汇编语言程序设计.北京: 机械工业出版社. 2006

# 致 谢

在本论文的创作过程中得到了许多同学和老师的热心帮助，正是有了他们的无私帮助才能是我的论文和代码的已修改和完善。感谢我的指导老师柳映辉老师，他在我的论文创作过程中多次及时的支出了我的错误和不足，并且凭借着丰富的编程经验瞬间就帮我查出了一个bug。感谢各位评审本论文的专家和老师，感谢你们在百忙之中抽出时间。感谢所有参考文献的作者，感谢你们对我们这些学子的努力付出，我的这个作品也是你们的智慧结晶。

最后由于时间仓促和本人的能力有限，本文中难免会有所纰漏，还希望各位专家不吝赐教，文中难免有不足，疏漏甚至错误之处，恳请给予批评指正，以便在今后的学习中进一步修正和完善，谢谢！