**UCore学习手册**

**东北大学（沈阳）**

**WKN**

目录

[简介： 5](#_Toc68603229)

[基础知识： 7](#_Toc68603230)

[启动过程 8](#_Toc68603231)

[GCC的编译过程 9](#_Toc68603232)

[第一章 12](#_Toc68603233)

[所需了解的知识 12](#_Toc68603234)

[加载BIOS 12](#_Toc68603235)

[读取MBR（磁盘的第一个扇区） 12](#_Toc68603236)

[X86CPU的相关知识 13](#_Toc68603237)

[例题： 18](#_Toc68603238)

[gcc和make 21](#_Toc68603239)

[源代码详解 28](#_Toc68603240)

[craeteISO.c 29](#_Toc68603241)

[bootloaderasm.S 30](#_Toc68603242)

[bootloader.c 35](#_Toc68603243)

[原bootloader.c： 40](#_Toc68603244)

[bootloader.ld 43](#_Toc68603245)

[makefile 43](#_Toc68603246)

[第二章 48](#_Toc68603247)

[ELF 48](#_Toc68603248)

[特权级 49](#_Toc68603249)

[CPL 50](#_Toc68603250)

[DPL 50](#_Toc68603251)

[RPL 50](#_Toc68603252)

[CPL，DPL，RPL举例 50](#_Toc68603253)

[段描述符字段格式 51](#_Toc68603254)

[一致代码段与非一致代码段 54](#_Toc68603255)

[访问一致性代码段的特权级检查规则： 54](#_Toc68603256)

[非一致代码段的特权级检查： 54](#_Toc68603257)

[门描述符： 55](#_Toc68603258)

[调用门： 55](#_Toc68603259)

[结构 55](#_Toc68603260)

[通过调用门访问代码段 56](#_Toc68603261)

[对于call和jmp检查规则 57](#_Toc68603262)

[C语言内存分布 58](#_Toc68603263)

[GCC的段与x86段的关系 59](#_Toc68603264)

[ESP与EBP概念： 60](#_Toc68603265)

[堆栈段和数据段访问的特权级检查规则 60](#_Toc68603266)

[TSS 61](#_Toc68603267)

[TSS字段格式 61](#_Toc68603268)

[堆栈切换 63](#_Toc68603269)

[转移和返回的具体过程 64](#_Toc68603270)

[返回的过程 65](#_Toc68603271)

[实模式/保护模式下的中断 66](#_Toc68603272)

[BIOS中断服务程序： 67](#_Toc68603273)

[BIOS中断服务一览表： 68](#_Toc68603274)

[中断与陷阱描述符字段： 69](#_Toc68603275)

[中断处理流程 70](#_Toc68603276)

[中断处理程序执行结束返回 71](#_Toc68603277)

[中断与陷阱的权限检测： 72](#_Toc68603278)

[I/O端口 72](#_Toc68603279)

[独立编址 73](#_Toc68603280)

[内存映射编址： 73](#_Toc68603281)

[显示 73](#_Toc68603282)

[1MB内存分布 74](#_Toc68603283)

[显示字符 75](#_Toc68603284)

[C函数栈变化 76](#_Toc68603285)

[struct成员在内存的排列 77](#_Toc68603286)

[源代码 78](#_Toc68603287)

[CRTDisplay.c 79](#_Toc68603288)

[trap.c 81](#_Toc68603289)

[Ivector.S 82](#_Toc68603290)

[entrytrap.S 83](#_Toc68603291)

[测试中断 85](#_Toc68603292)

[第三章 内存管理 86](#_Toc68603293)

[常见内存模型 86](#_Toc68603294)

[实模式的内存 86](#_Toc68603295)

[段式内存 86](#_Toc68603296)

[页式内存 87](#_Toc68603297)

[段页式内存 89](#_Toc68603298)

[逻辑地址 89](#_Toc68603299)

[物理地址 90](#_Toc68603300)

[线性地址 90](#_Toc68603301)

[地址小总结 90](#_Toc68603302)

[平坦内存 91](#_Toc68603303)

[页自映射技术 97](#_Toc68603304)

[页表字段 98](#_Toc68603305)

[内存探测 99](#_Toc68603306)

[管理内存 101](#_Toc68603307)

[编译地址变动 101](#_Toc68603308)

[寄存器与数据段寻址的差距 103](#_Toc68603309)

[TLB 103](#_Toc68603310)

[小块内存开辟 104](#_Toc68603311)

[内存回收 106](#_Toc68603312)

[代码解析 106](#_Toc68603313)

[entryInit.S 108](#_Toc68603314)

[memoryInit.c 110](#_Toc68603315)

[第四章 进程 118](#_Toc68603316)

[网络参考博客： 119](#_Toc68603317)

[致谢 120](#_Toc68603318)

[附录 121](#_Toc68603319)

[BIOS中断: 121](#_Toc68603320)

[IO端口地址分配表 140](#_Toc68603321)

简介：

四岁第一次见到计算机，就被PC所吸引，也一直想实现一个OS，但看了看自己的能耐，也就一直搁置了。本科是计算机专业，毕设的时候选择了基于UCore来制作一个教学操作系统，最后做的也马马虎虎，所以研究生又开始把这个坑挖了出来，想做出最简单的操作系统。

本教程是一个菜鸡的操作系统之路，是面向小白的入门。本人水平也不高，只想尽力将自己所学的按大白话解释出来，如有错误，还希望指正。本系统是对UCore的个人注解。在此基础上想做出一个OS。

下面开始正文，先说说个人历程，做了一圈下来，发现最难的不是代码，是对各种工具的使用，这里主要是：编译器（GCC，LD），CMAKE/MAKEFILE，脚本控制。我开始并不在意这些，直到后来才慢慢从windows的集成环境脱离，适应了Linux一切自己动手丰衣足食的状态，我姑且将这些非代码的部分称为外围环境。因为操作系统是要与硬件直接交互的，故要面对硬件的复杂性（本教程中对硬件的支持都是最简单的状态），硬件容忍不了任何错误，故很多工作要强制指定程序的分布状况，这些都与在Windows下由IDE一键生成不同，一个最简答的问题：在windows怎么让代码的地址可以从0x7C00处开始排列，并获知这段代码占用的总长度并在这段代码中使用这个长度？？？我反正没有在windows解决这个问题，在Linux下这些都是依靠GCC组件来实现，在后面会向大家介绍。说到这里能看到，其实代码的底层也没有那么神秘，不过繁琐是真的，所以人们开发了C语言，后来又是JAVA，现在的Python，目标都是一个，屏蔽底层，更快更简洁的实现更多的功能，可能有一天，C语言都变成了像汇编一样古老神秘的语言。

操作系统的概念就不多说，百度百科有，这里说说操作系统为什么出现（水一水字数）：没有操作系统，只有一堆硬件，能用吗？能，至少还有一个BIOS，但能多高效的使用PC，基本就是0了，不对，好像能，CPU只认二进制，所以只要能在内存中放一段二进制代码，再让CPU的IP值指向代码的起始处，这段代码就跑起来了呀，还能显示字符，还能画图像，不过再来一份代码呢？？？？谁先运行，谁控制CPU，最关键的是，这段代码的运行地址和先前一份代码地址有重复地址怎么办？？？？打一架？不知道各位在学C语言的时候有没有这样的想法：指针能获取地址，每个程序都有地址，而我可以让指针指向我想要的地址，那么，我不可以窃取别的程序的数据了吗？我就是下一个黑客了呀，结果就是内存指向错误。后来才知道，操作系统把各个程序都虚化成独立运行了，早就把我这低端思想扼杀了。以上是操作系统最基本的功能，算是对OS概念的简单补充。

接下来说说我们需要干啥：实现一个简单操作系统，从代码角度学习操作系统系统，面向和我一样的小白。

限于我的个人能力，讲解的有不充分之处，可以在网络上搜索困惑之处的一些关键字或者专业名词，通过网络博客学习。

基础知识：

1. 汇编语言，是MASM还是AT&T汇编风格并不重要，重要的是你要能理解CPU的运作原理，这很重要，因为这里没有几行汇编代码，而且我会给处汇编的每一行注释，但不理解CPU的工作过程和原理真的不行。
2. GCC，这是编译器，具体到gcc的每一个命令行参数怎么倒不至于，但至少要能清楚编译过程是啥样的，参考学科是：编译原理（这玩意学好了我觉得学啥语言都快，重要还是思想）。
3. C语言，这自然是主力，可能会问为什么不用C++，这是因为，C++的符号表和组织结构比C复杂，在和硬件层贴合的时候，会难度陡增，事实上，将C语言和汇编衔接的过程不亚于弄清楚C语言的编译流程（后面会一直遇见这个状况，会详细结合代码说明的）。
4. QEMU，一个计算机模拟器，这东西只需会学安装和使用就行了，虽然它能看到CPU各个寄存器的数值，并且支持逐步调试，但，如果用过OLLY那种反汇编软件就发现，这功能基本实用不大，毕竟突然冒出来几千个16进制数你人都傻了，没有耐心看完结果的。
5. Lds脚本，这个负责编译最后一步的重定位功能了，比如我想让代码的地址从0x53处开始，到0x99停，余下的代码地址从0xfff开始，直到结束。你可能觉得我吃撑了这么做，如此管理代码的弟子分布，但操作系统中就得这么干。

说了一些与操作系统代码无关的外围环境，接下来说说操作系统代码长啥样，它，长的和你用C语言写出来的代码没有区别，都是汇编生成的二进制，不过它能一直占据CPU的控制权，只是经常让出控制权，让别的代码也用CPU。理论上，你可以让OS只实现”hello world”，但那意义何在。操作系统其实拥有这么霸道的特权还有硬件给他撑腰，不然，它早就被我上面那个摇篮中的思想击垮了。所以操作系统特殊就特殊在我们愿意让它特权化，给他很多特权，才铸造了它，不然他和hello world代码没有区别。

环境配置：Linux，在其上安装QEMU，GCC，CLion，make，我们可以上路了。

接单介绍一下各个东西：QEMU，一个简易的虚拟机，和VM等价，主要是为了降低我们的开发难度；GCC，编译器，对应的是Windows下的MinGw；CLion，代码编译环境，和VS有些像，用起来舒服一些而已；make：管理工程的，组织各个.cpp .h .lds文件的，这些工作在Windows下都被集成环境所完成，我们不知道，就像操作系统屏蔽了底层硬件一样，现在我们要正视他了，而且他并不简单。

听完上面的东西，可能会有畏难情绪，我也一样，要不是不写没有毕业证，我早就跑去搞GNN了（现在就是搞GNN的），我会尽力介绍清楚这些的。

## 启动过程

先来看看操作系统的启动过程，他是如何从寄人（BIOS）篱下，又崛起成为整个PC的最高拥有者的。

* 当PC电源打开后，80x86结构的CPU自动进入实模式。并从地址0xFFFF0 (FFFF:0) 开始自动执行程序代码，这个地址通常是是ROM-BIOS中的地址。
* PC机的BIOS将执行某些系统检测，并在物理地址0处开始初始化中断向量。 此后，它将可启动设备的第一个扇区(磁盘引导扇区，512字节)读入绝对内存地址 0x7C00 处，并跳转到这个地方。

这个过程中，出现的关键字：

80x86架构：可以粗略理解为就是Intel的CPU的意思，当年IBM为了防止Intel垄断供应CPU，所以Intel也把X86架构（即它的设计思想与CPU架构）提供给了AMD，这也是AMD和IntelCPU非常相似的原因。

实模式：CPU有实模式和保护模式两种工作方式，个人感觉更像是CPU发展历史造成的。实模式出现于早期8088CPU时期。当时由于CPU的性能有限，一共只有20位地址线（所以地址空间只有1MB），以及8个16位的通用寄存器，以及4个16位的段寄存器。所以为了能够通过这些16位的寄存器去构成20位的主存地址，必须采取一种特殊的方式。当某个指令想要访问某个内存地址时，它通常需要用下面的这种格式来表示：(段基址：段偏移量)，也就是在汇编课程中学习的那种CPU工作方式，实模式会在不同的段基址和偏移量的情况下，访问到同一个物理内存地址（想不到吧）。“实”的意思更多是访问的地址就是物理真实的地址，实模式访问的地址空间也受地址总线的限制而只能访问1MB大小。保护模式是扩增后的内存访问模式，CPU对操作系统安全也提供了更多的硬件支持。保护模式会在后面专门介绍。实模式和保护模式可以切换，在后面都会通过代码来展现。

0xFFFF0地址：CPU的IP就会指向这里，这里是ROM-BIOS的地址。为什么不是别的地址，因为BIOS地址在这里，指向别的地址读取到错误的指令。

ROM-BIOS：ROM是只读内存，BIOS（基本输出输入系统）是由厂商刷进ROM中的一个系统，帮你封装一些底层的硬件操作。可以通过汇编的INT指令来执行BIOS中的功能，它比操作系统还底层。

磁盘引导扇区：磁盘中的第一个扇区（扇区，自己查询是什么），用于加载并转让CPU控制权给操作系统。它里面存的是一段代码，大小不可以超过512B，它为操作系统设置一些基本的标志位，以确保操作系统有正常的工作环境。为什么不把第一个扇区直接整合到操作系统中？原因是：BIOS只会读第一个扇区加载到内存中，所以其余的代码都要通过第一个扇区的代码加载进入内存，这样所有扇区的代码都可以被加载，这也就回答了为什么第二个扇区BIOS不管。最重要的是有的电脑会安装双系统。

0x7C00：低于这个值的内存被BIOS占用了。

* 此时BIOS将磁盘第一扇区的代码加载进了内存的0x7C00处，这个代码即：bootloader
* Bootloader知道操作系统存放位置，并将OS加载进来，然后将CPU的控制权交给了OS

Bootloader在UCore是由我们实现的，故知道OS的各种数据。不过bootloader在实模式下运行，访问内存上限只有1MB，所以boot loader还肩负着进入保护模式的功能，然后加载操作系统，并跳转执行OS，不然内存不够用（只有1MB内存，0X7C00以下还被占用了）。

至此操作系统进入内存并接管计算机，之后的路怎么走，就自由发挥了。可以看到这个过程几经周折，在实现代码前，我们有个问题：

怎么让bootloader的编译起始地址为0X7C00，我们以往编程从来不管程序内存地址的排列问题。指定编译地址我们在后面还要多次用到。这需要通过编译器来实现。

Bootloader和OS明显是两个程序，我们怎么在代码层面怎么管理，这需要MAKE来解决。

以上两个问题难很难，我们需要详细了解GCC的流程和MAKE的使用，如果忽略这两个问题，你将看不懂代码和开发的整体框架。

## GCC的编译过程

C语言写代码的时候我们会有一个头文件和.c文件，他们有啥区别吗？答案是没有，在GCC看来都是文本文件而已，你也可以将他们的后缀改成.qqqq都没事，在gcc看来还是文本。这样做的初衷是为了更方便的管理代码和工程而已。GCC先将#include<\*.h>处的命令，根据\*.h所指的文件，将\*.h中的代码全部复制过来，这也是为什么.h会有预编译指令防止你多次添加同一个\*.h文件，不然一个.c文件中会重复n次.h中的同样代码，你也会疯掉的。合并后就只有.c文件了，.h就没用了

然后将这个添加完.h的.c文件各自进行编译，变成汇编代码，什么叫各自？意思就是有多少个.c文件就编译出多少个汇编代码，你可能会问，有的.c文件中会用到别的.c中的函数，怎么办？答案是不管，先放着（这时其实GCC也不知道一个.c文件中提到的另一个.c文件中的函数到底存不存在，也不知道那个函数的格式到底是什么样，这也凸显出了头文件的作用），最后一步会处理这个问题得。从这里能看出来C语言其实就是汇编转换器，所以GCC是可以将汇编和C语言文件混在一起编译成一个程序的。

然后将各个汇编代码编译成独立（独立的含义同上）的二进制文件（也就是目标文件.o），也就是机器码了。随后开始最后一步，重定位和链接。链接的意思，就是上一步中各自编译中涉及使用别的文件中的函数或者变量，现在彼此都开始找这些函数在哪，并且指向它（函数就是一个特殊指针而已），这样，之前各个文件的.c代码都组合起来了，这也就是为什么两个.c文件中有同样的函数，编译报错由ld（GCC的链接器）报出（直到最后一步才能发现两个同样的函数，不知道用哪个），而不是第一步预编译爆出。重定位：各个独立的二进制文件要编成一个二进制文件（也就是最终程序），地址肯定会冲突，毕竟之前的内存地址各自为战，各个二进制文件编译地址都是从0开始的，为了解决冲突，就需要将一些二进制文件的地址重新调整一下，让大家都不冲突，这个过程中自然也就可以指定一些代码的起始地址了。

通过上面的流程，我们看到GCC的编译操作其实可以让我们实现任何形式的代码，当然，细节过程还要上网再搜索。

再提及以下题外话，为什么会有目标文件和库文件这些东西，GCC为什么不一次直接生成一个可执行文件。有一种情况，是我们有时希望别人会用到我们的代码，带却不想公开源代码，就可以使用头文件+库文件的形式，头文件告诉别人我们的代码有哪些函数，各类函数的声明是什么样的，别人通过在自己的代码中添加这个头文件就可以使用我们的函数，但头文件只有声明，没有实际代码，实际代码编译成二进制码存放在库文件中，在最后链接步骤中，别人就可以将我们的执行代码加入他们的代码了，这样在没有泄露源代码的情况下传播了我们的贡献。

MakeFile：在windows下，管理一个工程的所有工作都由IDE帮我们做了，我们总以为是GCC有什么特殊功能，其实不是，是工程编译文件的作用，在VS下也就是.sln（WPF的工程管理文件后缀）或者别的后缀。在linux下，由我们指示GCC来完成，但那样太麻烦了，于是makefile出现了，它足够灵活，细节很到位。

至此，对于开发小的操作系统的简介算是结束了，综上所述，我们需要配置哪些东西：

Linux虚拟机，推荐用Ubuntu

虚拟机中需要安装GCC，Make，QEMU

虚拟机中的开发IDE是CLion

需要学习的知识：简单的汇编，GCC，makefile。

现在可以进行开发了。

第一章

## 所需了解的知识

1.x86CPU的实模式与保护模式，及这两种模式所涉及的寄存器。  
2.AT&T汇编指令格式（这与我们之前汇编课程学习的MASM汇编语法风格有所出入），能够看懂即可。  
3.操作系统的启动过程（理论课程已经有所了解），基于理论课程，  
4.GCC编译器的编译过程，及它的一些指令使用方法。C语言如何生成最终二进制代码过程。  
5.CPU在实模式与保护模式下的寻址方式与过程，着重了解保护模式的寻址方式，段表的数据格式。  
6.段式内存的思想与原理。  
7.ELF文件的格式与MBR引导。

### 加载BIOS

计算机电源加电质检，首先加载基本输入输出系统（Basic Input Output System，BIOS），BIOS中包含硬件CPU、内存、硬盘等相关信息，包含设备启动顺序信息、硬盘信息、内存信息、时钟信息、即插即用（Plug-and-Play，PNP）特性等。BIOS是由硬件厂商写好的，不需要我们管，我们只需关心BIOS的协议是什么，我们遵守它的协议就可以了。在我们的实验中，涉及的BIOS的知识并不多。

### 读取MBR（磁盘的第一个扇区）

读取完BIOS信息，计算机将会查找BIOS所指定的硬盘MBR引导扇区，将其内容复制到0x7c00地址起所在的物理内存中。被复制到物理内存的内容是Boot Loader，然后CPU从0x7c00出开始执行，此时CPU就开始执行由我们编写的BootLoader了。

在BootLoader这里，需要先了解一下CPU的操作模式（现在统共有5中，我们只用到2种）。

## X86CPU的相关知识

第一代x86CPU只有16位的寻址总线，也就是实模式的运行环境，故最大内存地址为：2^16=65536字节（64KB），这在当时或许够用（阿波罗号登月飞船中，宇航员用来计算轨道的小型计算机中的内存其实也是64KB，但是它可以满足航天需求，除了操作流程极其复杂外没有任何缺憾）。但是很快，计算机需要更大的内存，Intel为了保证前向兼容，便添加了CS，DS，ES，SS寄存器，每个寄存器16位。在之前的汇编课程中，我们知道这种寻址过程是：CS中的数值左移4位加上IP的数值，得到了内存中的实际物理地址，或许会有人发现，这样一来，两组不同的CS与IP数值是可以指向同一个地址的，怎么办？确实会有这个现象，但是没办法，避免内存冲突取决于管理人员的技术。现在，我们内存的寻址能力达到了2^（4+16）=1MB的空间。不过到了保护模式下，CS与DS段的作用则不是直接寻址了，而是变为了段描述符的索引值，要注意这种变化。

实模式的理解不算困难，毕竟之前的汇编语言大家都接触过，接下来是保护模式。实模式最大的问题是内存寻址空间不够，只有1MB，并且对操作系统的支持也不如保护模式，在了解完保护模式就能明白为什么它对操作系统的支持不够大。

Intel推出了保护模式，但不能丢弃实模式，因为还有大量的代码是实模式编写的，要保持兼容性，不然自己的产品直接从市场就消失了。那么自然就是在实模式的基础上扩增，通过开启开关位来进入保护模式，这样，以前的代码能运行，想进入保护模式就进入保护模式。

保护模式都是32位的，寄存器都扩增到了32位，实模式有的寄存器它都有，保护模的总体更新如下：

* 内存管理采用段式+页式的方式；
* 最大寻指位2的32次方，即4G，还可以通过PAE模式访问超过4G的部分。
* 有4个安全级别，内存操作时有安全检查；
* 因为有了分页功能，因此有虚拟地址和物理地址的区别；
* 各种寄存器的位宽基本上都是32位的，但是可以兼容同名的16位寄存器；

回顾以下我们在操作系统原理中学的知识，内存的管理方式有：段式内存，页式内存，段页式内存。

段式内存管理：给每一个载入内存中的程序分配一个段，段中记录了段的起始地址，段的长度，访问内存地址时：程序中的地址+段的起始地址得出实际物理地址，这和实模式下的CS（代码），DS（数据），SS（堆栈）对应上了。操作系统记录下哪些内存分配给了哪个段，这样实现了内存的分配与回收。

页式内存管理：把内存都变成一个个固定大小的块（即页），然后通过映射，将一个个物理页与程序中的地址对应起来。这样访问流程如下：

在页式系统中，指令所给出的地址分为两部分：逻辑页号和页内地址。CPU中的内存管理单元(MMU)按逻辑页号通过查进程页表得到物理页框号，将物理页框号与页内地址相加形成物理地址(见图4-3)。上述过程通常由处理器的硬件直接完成（为了加速），不需要软件参与。通常，操作系统只需在进程切换时，把进程页表的首地址装入处理器特定的寄存器中即可。一般来说，页表存储在主存之中。这样处理器每访问一个在内存中的操作数，就要访问两次内存。第一次用来查找页表将操作数的 逻辑地址变换为物理地址；第二次完成真正的读写操作。这样做时间上耗费严重。为缩短查找时间，可以将页表从内存装入CPU内部的关联存储器(例如，快表) 中，实现按内容查找。此时的地址变换过程是：在CPU给出有效地址后，由地址变换机构自动将页号送人快表，并将此页号与快表中的所有页号进行比较，而且这 种比较是同时进行的。若其中有与此相匹配的页号，表示要访问的页的页表项在快表中。于是可直接读出该页所对应的物理页号，这样就无需访问内存中的页表。由 于关联存储器的访问速度比内存的访问速度快得多。

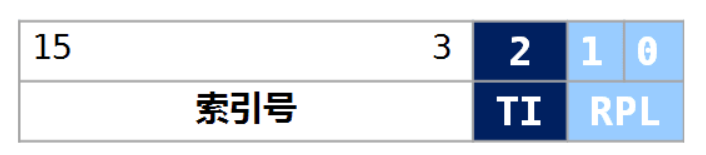
还有一个是段页式，这里不再细介绍了。回到我们的UCore中，当初Intel是希望做出段页式的，不过很快暴露出个问题，页式内存能等价掉段式内存的价值，意思就是：页式内存和段式都是内存映射技术（将物理地址与虚拟地址映射），不过页式的每个段比较小（每个页大小4KB），需要记录内存映射的量比较大，但本质都一样。

编译出的程序都有代码段，数据段，堆栈段等各种段，实模式下就对应了CS，DS，SS等，每个代码通常有4个段，操作系统需要记录各个程序的各个段，问题也出现在了，记录各个段由GTD完成的，但是它的记录空间是有限制的，最多只能记录：8192个段。

GDT：全局描述符表，它中的每一项占64bit，每一项描述了一个段（如C语言的数据段，代码段，堆栈段）的信息，总共可以有2^13=8192个段。每个程序一般都要4个段，8192/4=2048，基本载入不了几个程序，而且还要频繁访问记录段的表格。

接下来讲4个安全级别，内存操作时有安全检查：四个特权级：0，1，2，3。数字越小级别越高，0是操作系统的内核级别，3是一般程序的运行级别，不许低级访问高级的空间，这样从硬件上支持了操作系统的内核，特权级是一个极其复杂的体系，后面会详细介绍，这里了解到这里即可。我们将特权级和段管理结合起来，就能看到CPU的保护模式的含义了：它保护的是各个特权级，使其不会彼此影响。

接下来我们回到Linux如何绕过了段管理的数量上限问题：32位CPU，Intel扩增了所有寄存器的位长，CS和DS等段寄存器的长度变了，但你能操作的长度没有变（扩增的部分不可见，不可操作，CPU自动使用），加之需要支持特权级和操作系统的原因，Intel也改变了CS，DS，ES，SS的含义：



段寄存器的字段格式变为了：段选择子。

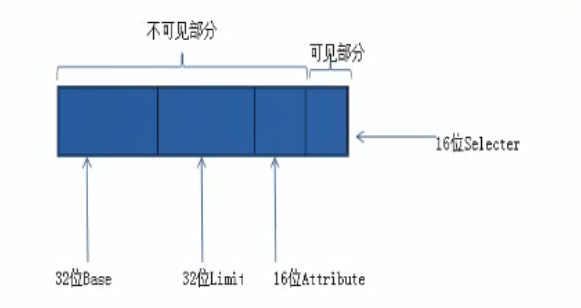
段寄存器的高15~3位的作用是在GDT中做索引：共可以索引2^13=8192个段，PRL字段表示的是特权等级，TI字段的含义如下：

TI=0：查GDT表

TI=1：差LDT表

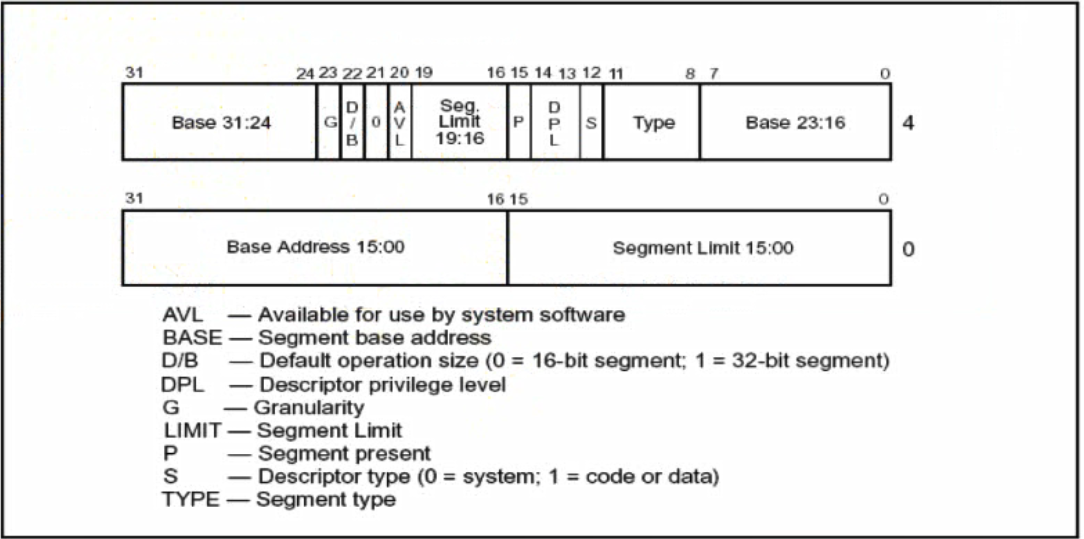
UCore中无需关心LDT，他们的含义大家可以自行百度。

这就是CS，DS，SS，ES，FS，GS在32位下的全部字段如下（大部分不可见，CPU自动操作）：



可见部分的16selector就是我们上面介绍的那些。

接下来介绍操作系统记录所有段的GDT，GDT是一个表格，存放在内存中，由寄存器GDTR存储它在内存中的起始地址。这个表格中一行数据表示一个段，一个段的格式如下：



P=1：段描述符有效

P=0：段描述符无效

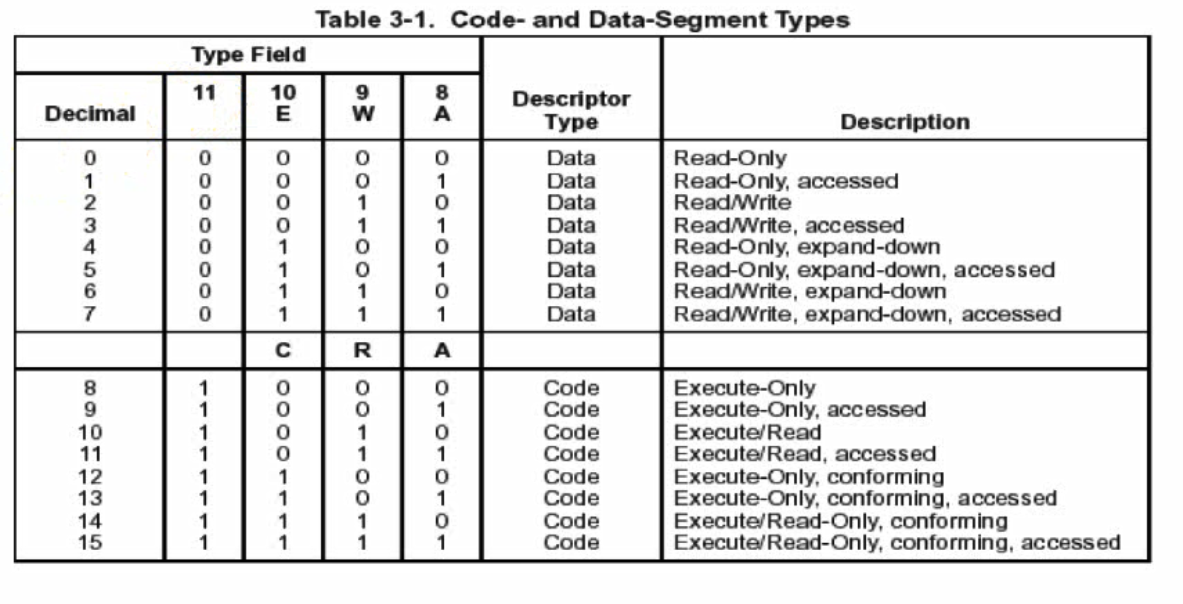
G=0：段寄存器Limit单位为字节，Limit最大值0x000FFFFF

G=1：段寄存器Limit单位为4KB，Limit最大值0xFFFFFFFF

S=1：代码段或数据段描述符

S=0：系统段描述符

**如果S为1：**



数据段：

A:是否被访问过

W:是否可写

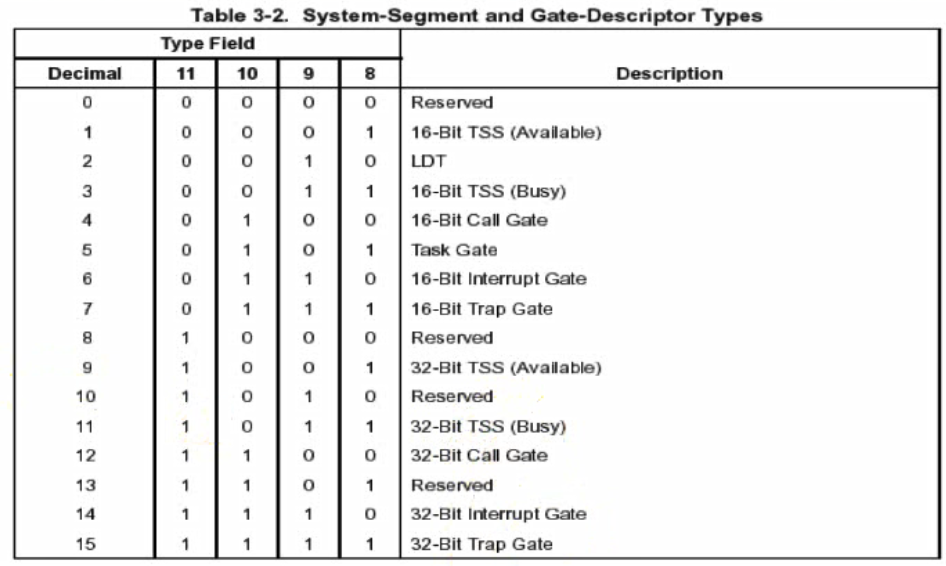
E：是否向下拓展

代码段：

R:是否可读

C：是否为一致代码段

**如果S位为0：**



D/B位：

对CS段：

D=1：采用32位寻址方式

D=0：采用16位寻址方式

对SS段：

D=1：隐式堆栈访问命令使用32位堆栈指针寄存器ESP

D=0：隐式堆栈访问命令使用16位堆栈指针寄存器SP

隐式堆栈访问命令（如push，pop，call）

对向下拓展的数据段：

D=1：段上限4GB

D=0：段上限64KB

DPL：描述符特权级别，可以看作与段选子中的RPL对应。

目前这些字段含义不必关注，列出来只是辅助理解CPU是如何寻址和管理内存的，能看懂Bootloader代码就好。

到目前，这和我们说的Linux如何用页式内存解决GDT段的数量上限有什么关系？事实上，页式内存下，我们只需要不到20个段记录就能管理所有的段，要不是特权级问题，我们连一个段都不需要。

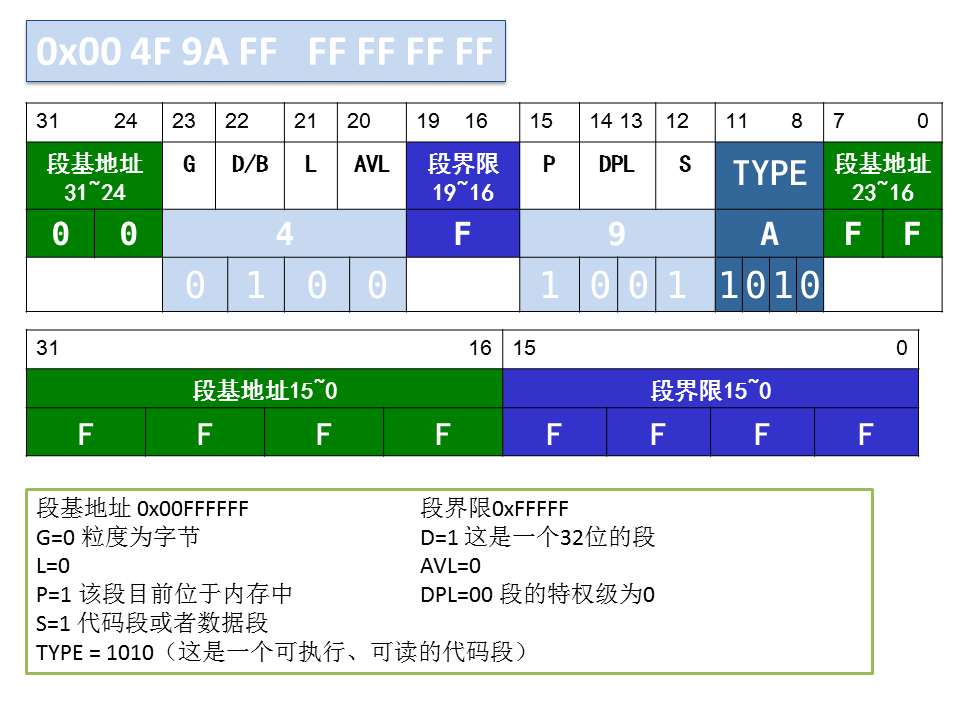
在UCore中，GDT中的段记录其实没有段式内存的本有意义了（记录段的起始地址和长度），而是用来表表达特权级的，在UCore中，有数个0级特权级的数据段，代码段，堆栈段，和数个3级特权级的数据段，代码段，堆栈段，还有IDT中的陷阱段和中断段，这些段记录的起始地址都是0X000，长度都是4GB，也就是所有内存空间，不同的是有的段可读可写，有的不可以。

这些段都是通过CS，DS，SS等段寄存器中的段索引值（也就是段选择子的高15位~3位）来查询GDT，获得当前代码各个段的特权级和访问权限，从而实现约束代码的特权级和访问权限。这样看，GDT没必要记录那么多段，段页式内存也变成了纯页式管理

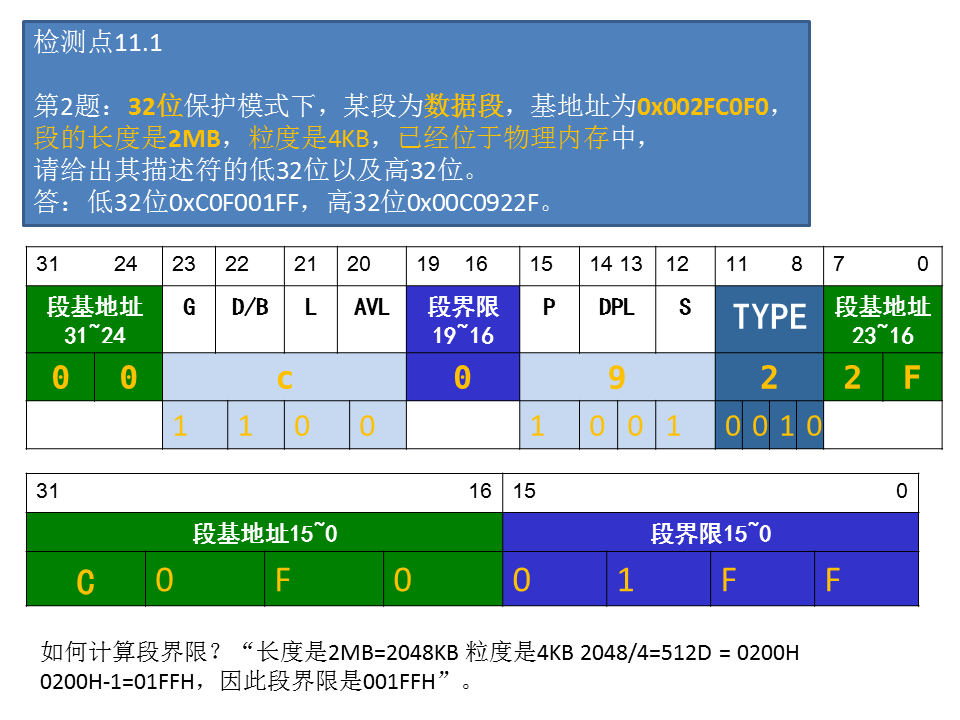
### 例题：

第一题：

某描述符是64位的0x4F9AFFFFFFFFFF，请问，段基地址是多少？段界限是多少？G、D、L、AVL、P、DPL、S和TYPE各是多少？



第二题：



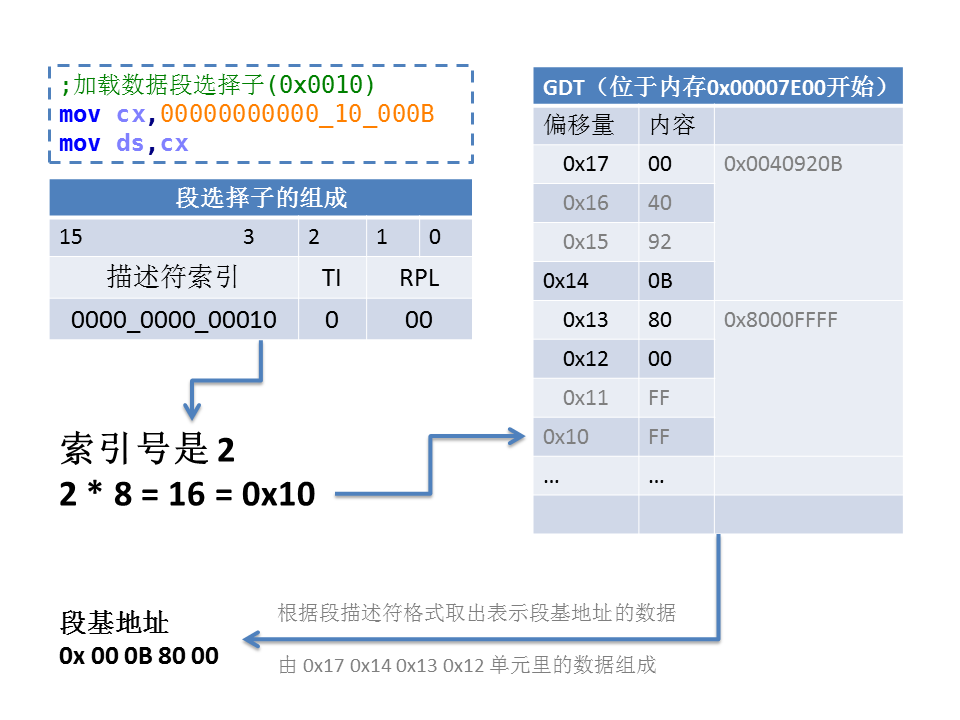
第三题：

 进入32位保护模式，段寄存器 **CS 、DS、 ES、 FS、 GS、 SS** ，它们还是16位的，但是不再存着**所谓的段地址**了，而是成为**段选择器**，里面存着**段选择子**，**段选择子**里面有**描述符索引**，根据这个**索引**去位于内存的**GDT（Global Descriptor Table 全局描述符表）**里找真正的**段基地址（线性地址）**；

 GDT有很多描述符：代码段描述符、数据段描述符、栈段描述符；

 在GDT里，每个描述符，占用**8个字节**单元；

 索引号 乘以 8 得到**要找的描述符**在**GDT表**内的**偏移地址**；



继续说保护模式的32位：这个很好理解，2^32=4GB，访问空间是4GB。PAE：支持访问超过4GB的能力，但是PAE不稳定，支持度也不高，实际意义不大，所以没有太多人用。

最后是保护模式的分页功能，这个在后面会详细介绍。本章都是与硬件进行大量的交互，所以需要掌握大量的硬件知识，但仅仅这些理论不够，细读代码会收获更多知识。

## gcc和make

在读代码前，需要先了解gcc和make，这两个工具的影响极大。先从GCC开始：

如何使用GNU工具编译出一个合格的BootLoader？这听起来简单，但实际并不容易实现。

在Linux系统下，生成一个可执行文件的流程是：gcc 文件名 -o 输出名 就可以看到一个C语言变成了可执行文件，比如最简单的hello world，应该没有人仔细想过，输出hello world这个过程的具体细节，这个过程是依赖操作系统的，即：不同的操作系统在实现printf这个功能的时候是不一样的，假设printf是通过调用系统陷阱或者中断来实现输出字符的，不同的操作系统这个功能的中断号不同。这就牵扯出一个问题，代码会依赖平台，可bootloader是操作系统的引导者，它没可依赖，所以它的任何功能都要自力更生，不依赖任何库，是单纯的二进制代码，单纯的二进制代码？？还有不单纯的？有，每个在Linux或者windows下的可执行文件都会有一个ELF格式的头文件（头文件会给出它需要哪些库，记录自己的运行架构，各个section和调试信息等等，这些都不是二进制可执行代码，而是文本记录信息，在bootloader只会占空间，我们需要扔掉，我们以前用C语言写出的程序都是带着ELF的，这些在Linux下都可以使用各种命令查看），而bootloader不需要这个头文件，但是别忘了，gcc是Linux平台下的，它编译出来的必然要符合Linux平台标准，自然会带有ELF格式头文件。解决上面的这个问题要求要对编译器有了解，并对ELF要有了解。关于ELF大家自行上网百度一下即可，不必太过深入，只需知道，不同的操作系统有着不同的ELF格式。

Make脚本大家自行上网了解以下，我会在makefile脚本中一行行解释为什么这么写makefile脚本。

还有lds脚本，它负责gcc编译流程的最后一步，可以操作代码的内存分布状况。我们的代码不需要太多复杂的lds脚本，所以不再单独介绍，而是仅仅解释一下我们的lds脚本即可。

之前说过GCC编译流程，不过没有依照代码来解释，而是站在工程管理的角度泛泛而谈，现在详细说一遍。GCC生成一个程序需要经过下面4个步骤：

（1） 预处理， C 编译器对各种预处理命令进行处理，包括头文件包含、宏定义的扩展、条件编译的选择等；

（2） 编译，将预处理得到的源代码文件，进行“翻译转换”，产生出机器语言的目标

程序，得到机器语言的汇编文件；

（3） 汇编，将汇编代码翻译成了机器码，但是还不可以运行；

（4） 链接，处理可重定位文件，把各种符号引用和符号定义转换成为可执行文件中

的合适信息，通常是虚拟地址。

这里有两个文件：hello.c和hello.k，没有起名字为.h是为了举例：.h和.c在gcc看来都是文本文件，后缀是什么都无所谓

hello.c代码：

#include<stdio.h>

#include"hello.k"

int main ()

{

display();

return 0;

}

hello.k代码：

#include<stdio.h>

void display()

{

printf("hello world");

}

第一步是预处理：gcc -E hello.c -o hello.i

-E选项告诉GCC只预处理即可，-o是指定生成文件的名字，然后我们得到了hello.i文件，打开看一看（这里代码都是复制stdio.h的，篇幅过长，这里就展示一小小部分）：

# 1 "hello.c"

# 1 "<built-in>"

# 1 "<command-line>"

# 31 "<command-line>"

# 1 "/usr/include/stdc-predef.h" 1 3 4

# 32 "<command-line>" 2

# 1 "hello.c"

# 1 "/usr/include/stdio.h" 1 3 4

# 27 "/usr/include/stdio.h" 3 4

# 1 "/usr/include/x86\_64-linux-gnu/bits/libc-header-start.h" 1 3 4

# 33 "/usr/include/x86\_64-linux-gnu/bits/libc-header-start.h" 3 4

# 1 "/usr/include/features.h" 1 3 4

# 424 "/usr/include/features.h" 3 4

# 1 "/usr/include/x86\_64-linux-gnu/sys/cdefs.h" 1 3 4

# 427 "/usr/include/x86\_64-linux-gnu/sys/cdefs.h" 3 4

# 1 "/usr/include/x86\_64-linux-gnu/bits/wordsize.h" 1 3 4

# 428 "/usr/include/x86\_64-linux-gnu/sys/cdefs.h" 2 3 4

# 1 "/usr/include/x86\_64-linux-gnu/bits/long-double.h" 1 3 4

# 429 "/usr/include/x86\_64-linux-gnu/sys/cdefs.h" 2 3 4

# 425 "/usr/include/features.h" 2 3 4

# 448 "/usr/include/features.h" 3 4

………………

……………… 省略几百行代码

………………

extern int pclose (FILE \*\_\_stream);

extern char \*ctermid (char \*\_\_s) \_\_attribute\_\_ ((\_\_nothrow\_\_ , \_\_leaf\_\_));

# 840 "/usr/include/stdio.h" 3 4

extern void flockfile (FILE \*\_\_stream) \_\_attribute\_\_ ((\_\_nothrow\_\_ , \_\_leaf\_\_));

extern int ftrylockfile (FILE \*\_\_stream) \_\_attribute\_\_ ((\_\_nothrow\_\_ , \_\_leaf\_\_)) ;

extern void funlockfile (FILE \*\_\_stream) \_\_attribute\_\_ ((\_\_nothrow\_\_ , \_\_leaf\_\_));

# 868 "/usr/include/stdio.h" 3 4

# 2 "hello.c" 2

# 1 "hello.k" 1

# 2 "hello.k"

void display()

{

printf("hello world");

}

# 3 "hello.c" 2

int main ()

{

display();

return 0;

}

这里面有很多函数和定义大家应该都用过。瞬间多出这么多代码，基本都是stdio.h中的东西，我们写的.c和.k的源代码在最后短短几行出现了。#include语句就是文本复制，将sdio.h的内容全抄过来了。

现在执行第二步：编译，将C语言代码处理成汇编，C语言实际就是汇编语言翻译器。使用：gcc -S hello.i -o hello.S，将得到hello.i的汇编代码：

.file "hello.c"

.text

.section .rodata

.LC0:

.string "hello world"

.text

.globl display

.type display, @function

display:

.LFB0:

.cfi\_startproc

pushq %rbp

.cfi\_def\_cfa\_offset 16

.cfi\_offset 6, -16

movq %rsp, %rbp

.cfi\_def\_cfa\_register 6

leaq .LC0(%rip), %rdi

movl $0, %eax

call printf@PLT

nop

popq %rbp

.cfi\_def\_cfa 7, 8

ret

.cfi\_endproc

.LFE0:

.size display, .-display

.globl main

.type main, @function

main:

.LFB1:

.cfi\_startproc

pushq %rbp

.cfi\_def\_cfa\_offset 16

.cfi\_offset 6, -16

movq %rsp, %rbp

.cfi\_def\_cfa\_register 6

movl $0, %eax

call display

movl $0, %eax

popq %rbp

.cfi\_def\_cfa 7, 8

ret

.cfi\_endproc

.LFE1:

.size main, .-main

.ident "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"

.section .note.GNU-stack,"",@progbits

这里就看到了全部的汇编代码，相比于之前的预处理，代码量少了好多，因为汇编只调有用的代码编译，多余的函数声明都不用编译。

记下来进行第三步：汇编，使用命令：gcc -c hello.S -o hello.o，现在生成了二进机器码，我们需要使用objdump才能查看.o文件：objdump -D hello.o，objdump指令有很多选项，这里查看了汇编码，还可以自行看看ELF头格式。

hello.o： 文件格式 elf64-x86-64

Disassembly of section .text:

0000000000000000 <display>:

0: 55 push %rbp

1: 48 89 e5 mov %rsp,%rbp

4: 48 8d 3d 00 00 00 00 lea 0x0(%rip),%rdi # b <display+0xb>

b: b8 00 00 00 00 mov $0x0,%eax

10: e8 00 00 00 00 callq 15 <display+0x15>

15: 90 nop

16: 5d pop %rbp

17: c3 retq

0000000000000018 <main>:

18: 55 push %rbp

19: 48 89 e5 mov %rsp,%rbp

1c: b8 00 00 00 00 mov $0x0,%eax

21: e8 00 00 00 00 callq 26 <main+0xe>

26: b8 00 00 00 00 mov $0x0,%eax

2b: 5d pop %rbp

2c: c3 retq

Disassembly of section .rodata:

0000000000000000 <.rodata>:

0: 68 65 6c 6c 6f pushq $0x6f6c6c65

5: 20 77 6f and %dh,0x6f(%rdi)

8: 72 6c jb 76 <main+0x5e>

a: 64 fs

...

Disassembly of section .comment:

0000000000000000 <.comment>:

0: 00 47 43 add %al,0x43(%rdi)

3: 43 3a 20 rex.XB cmp (%r8),%spl

6: 28 55 62 sub %dl,0x62(%rbp)

9: 75 6e jne 79 <main+0x61>

b: 74 75 je 82 <main+0x6a>

d: 20 37 and %dh,(%rdi)

f: 2e 35 2e 30 2d 33 cs xor $0x332d302e,%eax

15: 75 62 jne 79 <main+0x61>

17: 75 6e jne 87 <main+0x6f>

19: 74 75 je 90 <main+0x78>

1b: 31 7e 31 xor %edi,0x31(%rsi)

1e: 38 2e cmp %ch,(%rsi)

20: 30 34 29 xor %dh,(%rcx,%rbp,1)

23: 20 37 and %dh,(%rdi)

25: 2e cs

26: 35 .byte 0x35

27: 2e 30 00 xor %al,%cs:(%rax)

Disassembly of section .eh\_frame:

0000000000000000 <.eh\_frame>:

0: 14 00 adc $0x0,%al

2: 00 00 add %al,(%rax)

4: 00 00 add %al,(%rax)

6: 00 00 add %al,(%rax)

8: 01 7a 52 add %edi,0x52(%rdx)

b: 00 01 add %al,(%rcx)

d: 78 10 js 1f <.eh\_frame+0x1f>

f: 01 1b add %ebx,(%rbx)

11: 0c 07 or $0x7,%al

13: 08 90 01 00 00 1c or %dl,0x1c000001(%rax)

19: 00 00 add %al,(%rax)

1b: 00 1c 00 add %bl,(%rax,%rax,1)

1e: 00 00 add %al,(%rax)

20: 00 00 add %al,(%rax)

22: 00 00 add %al,(%rax)

24: 18 00 sbb %al,(%rax)

26: 00 00 add %al,(%rax)

28: 00 41 0e add %al,0xe(%rcx)

2b: 10 86 02 43 0d 06 adc %al,0x60d4302(%rsi)

31: 53 push %rbx

32: 0c 07 or $0x7,%al

34: 08 00 or %al,(%rax)

36: 00 00 add %al,(%rax)

38: 1c 00 sbb $0x0,%al

3a: 00 00 add %al,(%rax)

3c: 3c 00 cmp $0x0,%al

3e: 00 00 add %al,(%rax)

40: 00 00 add %al,(%rax)

42: 00 00 add %al,(%rax)

44: 15 00 00 00 00 adc $0x0,%eax

49: 41 0e rex.B (bad)

4b: 10 86 02 43 0d 06 adc %al,0x60d4302(%rsi)

51: 50 push %rax

52: 0c 07 or $0x7,%al

54: 08 00 or %al,(%rax)

...

但现在.o文件还不能运行，因为还未进行地址重定位，这里只是单文件，如果是多个.o文件，彼此之间还要进行链接，寻找自己代码中用到了别的.c文件中的函数，并连接起来，为了展示，我多写一个函数，并改一下源代码，添加一个show.c和show.h:

show.c的源代码

#include<stdio.h>

#include”show.h”

void myshow()

{

printf("my show\n");

}

show.h的源代码

#include<stdio.h>

void myshow();

在之前的hello.c中添加代码后：

#include<stdio.h>

#include"hello.k"

#include"show.h"

int main ()

{

myshow();

display();

return 0;

}

现在我们直接gcc -c hello.c show.c，可以看到多出两个文件：hello.o show.o。其实，到这一步，hello.c和hello.o根本不知道show中的myshow()函数的内容到底是什么，只知道myshow()的声明形式而已：它的返回值是void（也就是没有返回值），不需要参数。hello.o中除了记录了一个myshow()函数的形式外，并没有将myshow()函数的代码copy过来，只有在最后一步链接中，才会将show代码挪过来，这也就是C语言有时两个源文件使用对方定义的函数，使用时没有写对函数名，会返回一个ld错误，就是在链接时找不到这个函数导致的，而不是在汇编时就爆出这样的错误。现在还有一个问题：使用objdump查看两个.o文件，能看到他们的地址其实都是从0开始的，换言之，他们两个的地址是冲突的，但在链接步骤中，重定位会解决这些，因为重定位会将他们的地址都理顺。

第四步：链接，使用gcc hello.o show.o -o final，生成最终的结果final，也在这一步，hello.o拿着myshow()函数的声明在show.o中翻出了myshow()的全部代码，并连接了上去，才能执行，链接过程还可以使用ld命令来执行，其实gcc就是调用ld来完成的。

最后输入./final运行这个可执行程序。至此，gcc的编译过程我们全部展示了出来，但在这里面，还有很多GCC参数可选，能生成各种各样我们所需的代码。现在final的文件大小为8kb左右，实际上CPU要用到的机器码只有400B大小都不到，多出来的体积大多数都是各种记录数据，比如ELF，这些都可以通过objdump来查看。

在链接中，还会使用一个ld脚本（ld script）的东西来控制链接，主要是指定重定位中的地址分配问题。GCC默认的ld脚本很复杂：ld -verbose查看默认的lds。UCore用不上那么复杂。

## 源代码详解

在第一章实验中，我们的代码不多，目标是实现一个bootloader。

libs文件夹下是整个开发过程中需要的所有头文件，主要是各种宏定义，屏蔽繁杂的底层操作的，不必过细理解。

构成bootloader的源代码分布在：bootloader.c，bootloader.h，bootloaderasm.S中。

但bootloader还要转换成ISO格式文件，我们通过tool下的createISO.c编译出一个构造ISO格式文件的程序，注意，这个程序不是操作系统的一部分，是开发操作系统的工具程序，它读入一个文件，加上ISO文件格式头后写入磁盘，就完成了自己的使命，所以它的编译最简单。一个被系统认为是符合规范的硬盘主引导扇区具有如下特征：

（1）硬盘主引导扇区共512（200h）字节；

（2）硬盘主引导程序大小不超过510字节，且位于主引导扇区第0-509（0-1FDh）字节，若程序大小不足510字节，剩余空间补零；

（3）引导扇区有有效标志，位于第510-511（1FEh-1FFh）字节处，值为AA55h，即1FEh字节存0x55，1FFh字节存0xAA。

我们说过C语言就是生成汇编源码的，所以bootloader是C和汇编源代码混在一起编写的（bootloader.c和 bootloaderasm.S），他们两个中的函数都是彼此互相调用。

bootloader的起始地址要从0x7C00开始，可以通过gcc的选项命令定义起始地址，但这里我们还是选择用ld的链接脚本（在gcc编译的最后一步使用它）：bootloader.ld来告诉编译器bootloader的地址如何排列。

在OS文件夹下还有一个Makefile文件，这是用来管理我们整个工程的，到时候我们只需在该文件夹下打开shell，输入make指令，make指令默认打开Makefile读取里面的脚本编译所有的代码，生成操作系统。

### craeteISO.c

源代码的解析先从craeteISO.c开始：

#include <stdio.h>

#include <errno.h>

#include <string.h>

#include <sys/stat.h>

int

main(int argc, char \*argv[]) {

struct stat st;

if (argc != 3) { //判断输入参数个数

fprintf(stderr, "Usage: <input filename> <output filename>\n");

return -1;

}

//stat()用来获取指定路径的文件或者文件夹的信息。

if (stat(argv[1], &st) != 0) {

fprintf(stderr, "Error opening file '%s': %s\n", argv[1], strerror(errno));

return -1;

}

printf("'%s' size: %lld bytes\n", argv[1], (long long)st.st\_size);

//bootloader的大小不可以超过512字节，ISO要占用2个字节做标记，512-2=510

if (st.st\_size > 510) {

fprintf(stderr, "%lld >> 510!!\n", (long long)st.st\_size);

return -1;

}

//将bootloader读入

char buf[512];

memset(buf, 0, sizeof(buf));

FILE \*ifp = fopen(argv[1], "rb");

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;

}

fclose(ifp);

//这两个数字是ISO格式的魔数，加上它代表完成ISO格式的改造

buf[510] = 0x55;

buf[511] = 0xAA;

//将botloader的ISO格式文件写回磁盘

FILE \*ofp = fopen(argv[2], "wb+");

size = fwrite(buf, 1, 512, ofp);

if (size != 512) {

fprintf(stderr, "write '%s' error, size is %d.\n", argv[2], size);

return -1;

}

fclose(ofp);

printf("build 512 bytes boot sector: '%s' success!\n", argv[2]);

return 0;

}

上面的源代码不难，通过注释可以看懂。

接下来是bootlaoder的源代码，我们先讲汇编部分，因为汇编代码调用了C语言部分的函数，为什么不都用C语言来写？主要是C语言没有操作汇编的各种控制器的开关指令（可以在C语言中通过内联汇编来实现，但那样有太多不可控性）。汇编部分主要是铺设各种硬件环境，为进入保护模式设置好各种变量，内存管理依然是段式管理，不是页式管理。

### bootloaderasm.S

bootloaderasm.S源代码：

#include "bootloader.h"

# Start the CPU: switch to 32-bit protected mode, jump into C.

# The BIOS loads this code from the first sector of the hard disk into

# memory at physical address 0x7c00 and starts executing in real mode

# with %cs=0 %ip=7c00.

#设置CS，DS的段选择子，这里需要将这些数字结合起前面说过的段选择子的格式结合起来看

.set PROT\_MODE\_CSEG, 0x8 # kernel code segment selector

.set PROT\_MODE\_DSEG, 0x10 # kernel data segment selector

#设置打开保护模式的数值，保护模式的打开要通过设置CR0寄存器的数值

.set CR0\_PE\_ON, 0x1 # protected mode enable flag

# start address should be 0:7c00, in real mode, the beginning address of the running bootloader

.globl start

start:

#表示使用16位模式编译

.code16 # Assemble for 16-bit mode

cli # Disable interrupts

cld # String operations increment

# Set up the important data segment registers (DS, ES, SS).

#设置CS，DS，ES，SS的数值，这里是实模式，还没有到保护模式，是段式内存管理，所以设为0即可

#表示内存起始地址为0，结合实模式来看这里的设置即可

#xorw是汇编语言常用的快速置0的手段

xorw %ax, %ax # Segment number zero

movw %ax, %ds # -> Data Segment

movw %ax, %es # -> Extra Segment

movw %ax, %ss # -> Stack Segment

# Enable A20:

# For backwards compatibility with the earliest PCs, physical

# address line 20 is tied low, so that addresses higher than

# 1MB wrap around to zero by default. This code undoes this.

#开启A20地址线，这样就可以访问地址大于1MB的内存区域了。

#开启A20地址总线（A20 Gate是Intel为了保持前向兼容设计的，

#在80286中，cpu拥有24根地址总线（16MB），如果不开启A20，

#访问超过1MB的地址，高四位会被截断，访问的依然是低1MB内存地址，

#开启之后则可以正常访问），历史遗留下来的问题使得开启A20 Gate的方式很奇怪，

#居然要使用分配给键盘的端口来开启，而非一条专用的汇编指令，

#这是IBM当时为了保持兼容发明的方法。

seta20.1:

inb $0x64, %al # Wait for not busy(8042 input buffer empty).

testb $0x2, %al

jnz seta20.1

movb $0xd1, %al # 0xd1 -> port 0x64

outb %al, $0x64 # 0xd1 means: write data to 8042's P2 port

seta20.2:

inb $0x64, %al # Wait for not busy(8042 input buffer empty).

testb $0x2, %al

jnz seta20.2

movb $0xdf, %al # 0xdf -> port 0x60

outb %al, $0x60 # 0xdf = 11011111, means set P2's A20 bit(the 1 bit) to 1

#开启A20地址线中，inb和outb式用来操作IO端口的命令，

#inb $0x64, %al代表从0x64端口读入数据放入al中，0x64是键盘的端口号

# Switch from real to protected mode, using a bootstrap GDT

# and segment translation that makes virtual addresses

# identical to physical addresses, so that the

# effective memory map does not change during the switch.

#加载GDT：讲gdtdesc的内存地址值加载到GDTR寄存器中

lgdt gdtdesc

#通过将cr0寄存器PE位置1，开启保护模式

movl %cr0, %eax

orl $CR0\_PE\_ON, %eax

movl %eax, %cr0

# Jump to next instruction, but in 32-bit code segment.

# Switches processor into 32-bit mode.

#通过长跳转进入32位保护模式下的代码，自行查询limp指令的作用，

#注意这里的跳转带了一个段选择子的数值

ljmp $PROT\_MODE\_CSEG, $protcseg

#指示编译器以32位编译下列代码

.code32 # Assemble for 32-bit mode

#设置保护模式下的段选择子数值

protcseg:

# Set up the protected-mode data segment registers

movw $PROT\_MODE\_DSEG, %ax # Our data segment selector

movw %ax, %ds # -> DS: Data Segment

movw %ax, %es # -> ES: Extra Segment

movw %ax, %fs # -> FS

movw %ax, %gs # -> GS

movw %ax, %ss # -> SS: Stack Segment

# Set up the stack pointer and call into C. The stack region is from 0--start(0x7c00)

#为进入C语言函数设置EBP和ESP的数值

movl $0x0, %ebp

movl $start, %esp

#跳转进入了C语言编写的bootmain()

call bootmain

# If bootmain returns (it shouldn't), loop.

spin:

jmp spin

# Bootstrap GDT

#强制地址4字节对齐：所有变量的起始地址都要能被4整除

.p2align 2 # force 4 byte alignment

#gdt表在内存中的存储地址

gdt:

#设置一个空段

SEG\_NULLASM # null seg

#设置CS段，起始地址0，长度4GB，权限为可度可执行，GDT中的各个段的数值需要结合前面说过的GDT格式结合起来看，最好将这里的宏展开，逐位对照说明他们的含义

SEG\_ASM(STA\_X|STA\_R, 0x0, 0xffffffff) # code seg for bootloader and kernel

设置DS段，起始地址为0，长度4GB，权限为可写

SEG\_ASM(STA\_W, 0x0, 0xffffffff) # data seg for bootloader and kernel

#存入GDTR的地址数值

gdtdesc:

#GDT的大小

.word 0x17 # sizeof(gdt) - 1

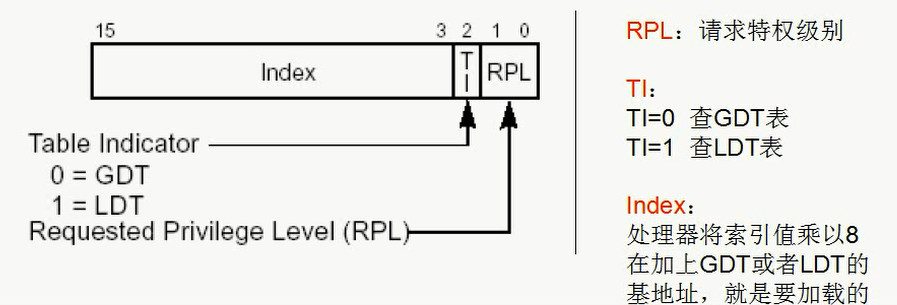
.long gdt # address gdt

以上是汇编代码段的内容，主要构建了GDT表，从BISO接受过来的是实模式，进入了保护模式，最后跳转进入bootmain()函数中。这里需记得，汇编中的标签的含义，还要明白，标签为什么叫伪指令。汇编的标签本质是该处内存地址值。

Linux的gcc本身就是平坦内存模型，编译出来的代码也是如此，它的特点就是代码和数据的地址并未遵照Intel所设想的段式内存管理，Intel所设想的是一个程序：代码一个段，数据一个段，堆栈一个段，各个段的起始地址都为0才对。但是Linux直接三段合一，三个段编译出来的地址彼此继承连续，但Intel的段式又必须要用，Linux就用它来作为一个权限检验，使得代码段只可读可执行，数据段可读可写，其余的功能并未使用，各个段内存地址起始都是0，长度都是4GB，且只有四个段（内核代码段，内核数据段，用户代码段，用户数据段），大家可能会好奇，一个操作系统上运行的程序数量绝对超过一个，只有一个用户段，彼此地址如何做到不冲突。这是通过后面的页式管理来解决这一问题。

·这里着重注意访问地址的计算过程，实模式下，CS中的值+IP中的值得到实际内存地址，所以代码在16位编译模式下，我们设CS的值为0，别忘了，我们指示ld将bootloader代码的起始地址设为0X7C00，而BIOS将Bootloader加载到了内存的0X7C00处，且同时BIOS置IP的值为0X7C00（实际是BIOS跳转到了0X7C00处），所以程序执行：（段起始值）0（CS段起始值）+0X7C00（IP值）=实际访问地址，正好是bootloader的内存物理地址。

这一状态持续到ljmp $PROT\_MODE\_CSEG, $protcseg，我们进入了保护模式。保护模式最大的不同是内存访问方式变了，CS等段寄存器的值是用来索引GDT中的段的，这也是为什么ljmp要带着$PROT\_MODE\_CSEG值跳转，该跳转先将$PROT\_MODE\_CSEG写入CS，即将CS为0X8，对照段选择子格式来看，它的含义是：RPL为0，TI为0，在GDT中索引段描述符，index为1（实际会乘8），再对照着GDT来看：GDT中一共三个段，SEG\_NULLASM，SEG\_ASM(STA\_X|STA\_R, 0x0, 0xffffffff)，SEG\_ASM(STA\_W, 0x0, 0xffffffff)，一个段描述符占8个字节，index\*8后，索引的是第二段，这个段可读可执行，起始地址为0，长度为4GB。CS的值变了，但是段起始地址+代码中的地址（IP中的值），实际访问的还是实模式下的同一个地址，代码依然和之前实模式代码无缝衔接的执行了。



上面给详细展示了实模式切换保护模式的过程，重点注意在执行代码：

movl %cr0, %eax

orl $CR0\_PE\_ON, %eax

movl %eax, %cr0

后，程序就已经是保护模式下执行了，ljmp $PROT\_MODE\_CSEG, $protcseg就是在保护模式下执行，而不是从.code32 protcseg:后才开始保护模式的。所以要提前设置号GDT还GDTR数值。这也是bootlaoder中最有价值的代码。

最后的设置EBP和ESP是因为C语言函数存储访问局部变量和函数实参需要，这里需要很多知识来支撑，目前不多讲，只提一下。ESP：栈指针寄存器(extended stack pointer)，其内存放着一个指针，该指针永远指向系统栈最上面一个栈帧的栈顶。ESP的值会随PUSH与POP指令的使用改变其值，从而永远指向栈顶。EBP：基址指针寄存器(extended base pointer)，其内存放着一个指针，该指针永远指向系统栈最上面一个栈帧的底部。栈帧就是C语言函数用来保存现场和本地局部变量用的。后面再详细介绍，或者自行查询C语言函数编译成汇编后的代码是什么。

### bootloader.c

接下来是bootloader.c的代码：

先将bootloader.c中的代码替换为下面的代码，编译运行一遍：

#include <defs.h>

#include <x86.h>

#include <elf.h>

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* This a dirt simple boot loader, whose sole job is to boot

\* an ELF kernel image from the first IDE hard disk.

\*

\* DISK LAYOUT

\* \* This program(bootasm.S and bootmain.c) is the bootloader.

\* It should be stored in the first sector of the disk.

\*

\* \* The 2nd sector onward holds the kernel image.

\*

\* \* The kernel image must be in ELF format.

\*

\* BOOT UP STEPS

\* \* when the CPU boots it loads the BIOS into memory and executes it

\*

\* \* the BIOS intializes devices, sets of the interrupt routines, and

\* reads the first sector of the boot device(e.g., hard-drive)

\* into memory and jumps to it.

\*

\* \* Assuming this boot loader is stored in the first sector of the

\* hard-drive, this code takes over...

\*

\* \* control starts in bootasm.S -- which sets up protected mode,

\* and a stack so C code then run, then calls bootmain()

\*

\* \* bootmain() in this file takes over, reads in the kernel and jumps to it.

\* \*/

#define SECTSIZE 512

#define ELFHDR ((struct elfhdr \*)0x10000) // scratch space

/\* waitdisk - wait for disk ready \*/

static void

waitdisk(void) {

while ((inb(0x1F7) & 0xC0) != 0x40)

/\* do nothing \*/;

}

/\* readsect - read a single sector at @secno into @dst \*/

static void

readsect(void \*dst, uint32\_t secno) {

// wait for disk to be ready

waitdisk();

outb(0x1F2, 1); // count = 1

outb(0x1F3, secno & 0xFF);

outb(0x1F4, (secno >> 8) & 0xFF);

outb(0x1F5, (secno >> 16) & 0xFF);

outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);

outb(0x1F7, 0x20); // cmd 0x20 - read sectors

// wait for disk to be ready

waitdisk();

// read a sector

insl(0x1F0, dst, SECTSIZE / 4);

}

/\* \*

\* readseg - read @count bytes at @offset from kernel into virtual address @va,

\* might copy more than asked.

\* \*/

static void

readseg(uintptr\_t va, uint32\_t count, uint32\_t offset) {

uintptr\_t end\_va = va + count;

// round down to sector boundary

va -= offset % SECTSIZE;

// translate from bytes to sectors; kernel starts at sector 1

uint32\_t secno = (offset / SECTSIZE) + 1;

// If this is too slow, we could read lots of sectors at a time.

// We'd write more to memory than asked, but it doesn't matter --

// we load in increasing order.

for (; va < end\_va; va += SECTSIZE, secno ++) {

readsect((void \*)va, secno);

}

}

#define COM1 0x3F8

#define CRTPORT 0x3D4

#define LPTPORT 0x378

#define COM\_TX 0 // Out: Transmit buffer (DLAB=0)

#define COM\_LSR 5 // In: Line Status Register

#define COM\_LSR\_TXRDY 20 // Transmit buffer avail

static uint16\_t \*crt = (uint16\_t \*) 0xB8000; // CGA memory 图形显示适配器

/\* stupid I/O delay routine necessitated by historical PC design flaws \*/

static void

delay(void) {

inb(0x84);

inb(0x84);

inb(0x84);

inb(0x84);

}

/\*

通过并口进行输出的过程也很简单：

第一步：执行inb指令读取并口的I/O地址（LPTPORT + 1）的值，如果发现发现读出的值代表并口忙，

则空转一小会再读；

如果发现发现读出的值代表并口空闲，则执行outb指令把字符写到并口的I/O地址（LPTPORT ），

这样就完成了一个字符的并口输出。

\*/

/\* lpt\_putc - copy console output to parallel port \*/

static void

lpt\_putc(int c) {

int i;

for (i = 0; !(inb(LPTPORT + 1) & 0x80) && i < 12800; i ++) {

delay();

}

outb(LPTPORT + 0, c);

outb(LPTPORT + 2, 0x08 | 0x04 | 0x01);

outb(LPTPORT + 2, 0x08);

}

/\*

通过CGA显示控制器进行输出的过程也很简单：首先通过in/out指令获取当前光标位置；

然后根据得到的位置计算出显存的地址，直接通过访存指令写内存来完成字符的输出；

最后通过in/out指令更新当前光标位置。

\*/

/\* cga\_putc - print character to console \*/

static void

cga\_putc(int c) {

int pos;

// cursor position: col + 80\*row.

outb(CRTPORT, 14);

pos = inb(CRTPORT + 1) << 8;

outb(CRTPORT, 15);

pos |= inb(CRTPORT + 1);

if (c == '\n') {

pos += 80 - pos % 80;

}

else {

crt[pos ++] = (c & 0xff) | 0x0700;

}

outb(CRTPORT, 14);

outb(CRTPORT + 1, pos >> 8);

outb(CRTPORT, 15);

outb(CRTPORT + 1, pos);

}

/\*

通过串口进行输出的过程也很简单：第一步：执行inb指令读取串口的I/O地址（COM1 + COM\_LSR）的值，

如果发现发现读出的值代表串口忙，则空转一小会（0x84是什么地址???）；

如果发现发现读出的值代表串口空闲，则执行outb指令把字符写到串口的I/O地址（COM1 + COM\_TX），

这样就完成了一个字符的串口输出。

\*/

/\* serial\_putc - copy console output to serial port \*/

static void

serial\_putc(int c) {

int i;

for (i = 0; !(inb(COM1 + COM\_LSR) & COM\_LSR\_TXRDY) && i < 12800; i ++) {

delay();

}

outb(COM1 + COM\_TX, c);

}

/\* 显示字符的函数接口\*/

/\* 一个cons\_putc函数接口，完成字符的输出\*/

/\* cons\_putc - print a single character to console\*/

static void

cons\_putc(int c) {

lpt\_putc(c);

cga\_putc(c);

serial\_putc(c);

}

/\* 提供了一个cons\_puts函数接口：完成字符串的输出\*/

/\* cons\_puts - print a string to console \*/

static void

cons\_puts(const char \*str) {

int i;

for (i = 0; \*str != '\0'; i ++) {

cons\_putc(\*str ++);

}

}

/\* bootmain - the entry of bootloader \*/

void

bootmain(void) {

cons\_puts("This is a bootloader: Hello world!!");

/\* do nothing \*/

while (1);

}

上面的代码主要是测试bootloader模块是否正常加载进入内存并工作，正常的话会显示一行话：This is a bootloader: Hello world!!。

下面是原来的代码，它开始读入操作系统并运行。不过目前还没有写操作系统呢，所以它执行到后面会出错，不过这里先看懂代码并了解什么是ELF。

### 原bootloader.c：

#include <defs.h>

#include <x86.h>

#include <elf.h>

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* This a dirt simple boot loader, whose sole job is to boot

\* an ELF kernel image from the first IDE hard disk.

\*

\* DISK LAYOUT

\* \* This program(bootasm.S and bootmain.c) is the bootloader.

\* It should be stored in the first sector of the disk.

\*

\* \* The 2nd sector onward holds the kernel image.

\*

\* \* The kernel image must be in ELF format.

\*

\* BOOT UP STEPS

\* \* when the CPU boots it loads the BIOS into memory and executes it

\*

\* \* the BIOS intializes devices, sets of the interrupt routines, and

\* reads the first sector of the boot device(e.g., hard-drive)

\* into memory and jumps to it.

\*

\* \* Assuming this boot loader is stored in the first sector of the

\* hard-drive, this code takes over...

\*

\* \* control starts in bootasm.S -- which sets up protected mode,

\* and a stack so C code then run, then calls bootmain()

\*

\* \* bootmain() in this file takes over, reads in the kernel and jumps to it.

\* \*/

#define SECTSIZE 512

#define ELFHDR ((struct elfhdr \*)0x10000) // scratch space

//等待虚拟磁盘空闲

/\* waitdisk - wait for disk ready \*/

static void

waitdisk(void) {

while ((inb(0x1F7) & 0xC0) != 0x40)

/\* do nothing \*/;

}

//从虚拟磁盘上读取指定的某个块，这里的各个端口号都是控制磁盘工作的

/\* readsect - read a single sector at @secno into @dst \*/

static void

readsect(void \*dst, uint32\_t secno) {

// wait for disk to be ready

waitdisk();

outb(0x1F2, 1); // count = 1

outb(0x1F3, secno & 0xFF);

outb(0x1F4, (secno >> 8) & 0xFF);

outb(0x1F5, (secno >> 16) & 0xFF);

outb(0x1F6, ((secno >> 24) & 0xF) | 0xE0);

outb(0x1F7, 0x20); // cmd 0x20 - read sectors

// wait for disk to be ready

waitdisk();

// read a sector

insl(0x1F0, dst, SECTSIZE / 4);

}

/\* \*

\* readseg - read @count bytes at @offset from kernel into virtual address @va,

\* might copy more than asked.

\* \*/

static void

readseg(uintptr\_t va, uint32\_t count, uint32\_t offset) {

uintptr\_t end\_va = va + count;

// round down to sector boundary

va -= offset % SECTSIZE;

// translate from bytes to sectors; kernel starts at sector 1

uint32\_t secno = (offset / SECTSIZE) + 1;

// If this is too slow, we could read lots of sectors at a time.

// We'd write more to memory than asked, but it doesn't matter --

// we load in increasing order.

for (; va < end\_va; va += SECTSIZE, secno ++) {

readsect((void \*)va, secno);

}

}

/\* bootmain - the entry of bootloader \*/

//汇编代码中的call就是调用这里的bootmain()函数

void

bootmain(void) {

// read the 1st page off disk

//读入操作系统 ELFHDR是一个宏定义，((struct elfhdr \*)0x10000)，表示将从磁盘读入的数据放在0x10000起始处，其实读入的是可执行代码，不过CPU也分不清数据和代码，对它而言都是0101罢了。

//注意ELFHDR前有一个强制类型转换

readseg((uintptr\_t)ELFHDR, SECTSIZE \* 8, 0);

/\*

// is this a valid ELF?

//检查是否是正确的ELF文件

if (ELFHDR->e\_magic != ELF\_MAGIC) {

goto bad;

}

\*/

//开始按照ELF格式文件加载操作系统的各个段

struct proghdr \*ph, \*eph;

// load each program segment (ignores ph flags)

ph = (struct proghdr \*)((uintptr\_t)ELFHDR + ELFHDR->e\_phoff);

eph = ph + ELFHDR->e\_phnum;

for (; ph < eph; ph ++) {

readseg(ph->p\_va & 0xFFFFFF, ph->p\_memsz, ph->p\_offset);

}

// call the entry point from the ELF header

// note: does not return

//根据ELF格式获取内核程序的入口地址，将入口地址强制转换成一个函数指针然后进入这个函数执行，即进入操作系统内核并开始执行，至此，我们离开了bootloader。

((void (\*)(void))(ELFHDR->e\_entry & 0xFFFFFF))();

/\*

bad:

outw(0x8A00, 0x8A00);

outw(0x8A00, 0x8E00);

\*/

/\* do nothing \*/

while (1);

}

第一章的代码结束了，总量不多，但要掌握的相关联知识庞杂，需要大量查询，还对编程技巧有一定的要求。

### bootloader.ld

接下来是ld的脚本，它指导GCC如何编译我们的代码，这里主要给ld指示了代码的起始执行函数和指定内存地址分布：bootloader.ld

OUTPUT\_FORMAT("elf32-i386") 指定输出格式为i386

OUTPUT\_ARCH(i386) 指定输出架构为i386

ENTRY(start) 指定代码的执行起始地址为start，也就是bootloaderasm.S中的start:标签处

SECTIONS {

. = 0x7C00; ld脚本中 “.” 代表当前地址数值，这里指当前地址编译改为0x7C00。

.startup : { 生成一个startup的section，将所有之前.o文件中的.text的section合并在一起

\*是通配符的意思

\*bootloaderasm.o(.text)

}

同上，合并多个.o文件的同名section

.text : { \*(.text) }

同上

.data : { \*(.data .rodata) }

/DISCARD/ : { \*(.eh\_\*) }

}

## makefile

接下来是我们如何编译出一个合格的操作系统，并能被QEMU识别使用。UCore中使用make来管理整个项目，make执行需使用makefile脚本，makefile中都是字符串处理，

在讲述Makefile之前，(<https://seisman.github.io/how-to-write-makefile/index.html> makefile的学习网站,大概一天就能看完)我们先来粗略地看一看Makefile的规则:

**target... : prerequisites ...**

**command**

target就是目标，prerequisites是依赖项，command是依赖项改变后需要执行的shell命令，所以还需对shell命令有一些了解。所谓的依赖项可以是文件或者目标，依赖项是文件的时候，比如源代码，文件被修改后，那么这个依赖项的command就会被执行，当依赖项是目标（出现在这个makefile中的目标）时，其实描述的是依关系，target目标只是用来表示这个依赖项中的文件而已。举个例子：依赖项是helloworld.c，目标随便取个名字：A，command写gcc helloworld.c -o makefile.a，当helloworld中的代码修改后，输入make ，就会执行gcc helloworld.c -o makefile.a这个命令，这里你觉得那目标A呢？没有用到呀，别急，比如多个文件，他们都需要生成各个库文件，如：A.c;B.c:C.c … … X.c;Y.c;Z.c共24个源代码，24个文件有24个目标：A B C…… X Y Z，需要生成24个库文件A.o B.o C.o……Z.o，但是，他们需要有顺序和依赖关系，B需要先生成A后才能生成库，D依赖C才能生成，Y需要在B和D生成后才能生成Y.o，现在我们就需要目标了，它表示了各个目标间的依赖关系，不然乱做一团，满屏的错误报告。例子中的makefile这样写注意：

A:A.c

生成A.o

B:B.c A

生成B.o

C:C.c

生成C.o

D:C D.c

生成D.o

Y:B D

生成Y.o

……………

现在就学习这个项目中简单的makefile：

# .PHONY 指该目标是伪指令

.PHONY : target

# target依赖三个目标

target : create bootloader ISO

#定义一个变量,其实就是字符串,这里是Gcc编译要用到的选项

CFLAGS := -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -Os -fno-stack-protector

CC := gcc

LD := ld

#定义链接器的选项指令 -m elf\_i386:让ld链接输出一个i386的可执行文件 -nostdlib:不用标准库,UCore不需要, -N不清楚 -T使用指定ld脚本

LDFLAGS := -m elf\_i386 -nostdlib -N -T

CommonLib := ./libs

#create Tools

createISO := ./Tools/createISO.c

OutputPathCreateISO := ./bin/CreateISO/createiso

create : $(createISO)

$(CC) $(createISO) -o $(OutputPathCreateISO)

#create bootloader

BootLoaderInPath := ./BootLoader/

#文本处理,挑选出所有的源代码文件

BootLoaderFile := $(shell ls $(BootLoaderInPath))

BootLoaderFile := $(filter %.c %.S,$(BootLoaderFile))

BootLoaderOutPath := ./bin/BootLoader/

BootLoaderOutFile := $(foreach var,$(BootLoaderFile), $(addsuffix .o,$(addprefix $(BootLoaderOutPath),$(var))))

BLldspath := ./BootLoader/bootloader.ld

bianryBL := BL

#生成bootloader，使用ld命令

bootloader : $(foreach var,$(BootLoaderFile), $(addprefix $(BootLoaderInPath),$(var)))

echo $(BootLoaderOutFile) ; \

for i in $(BootLoaderFile); do $(CC) -c $(BootLoaderInPath)$$i $(CFLAGS) -I$(CommonLib) -o $(BootLoaderOutPath)$$i.o; done ; \

$(LD) $(BootLoaderOutFile) $(LDFLAGS) $(BLldspath) -o ./bin/BootLoader/bootloader ; \

objcopy -S -O binary $(BootLoaderOutPath)bootloader $(BootLoaderOutPath)$(bianryBL)

UCoreName :=ucore

ISOName := UCore.img

#生成ISO

ISO :bootloader

$(OutputPathCreateISO) bin/BootLoader/$(bianryBL) bin/$(UCoreName) ; \

dd if=/dev/zero of=bin/$(ISOName) count=10000 ; \

dd if=bin/$(UCoreName) of=./bin/$(ISOName) conv=notrunc ; \

#伪目标

.PHONY : qemu

qemu : #将镜像载入qemu并运行qemu

qemu-system-i386 -parallel stdio -hda bin/$(ISOName) -serial null

.PHONY : clean

clean: #删除一些文件

rm -r ./bin/BootLoader/\* ./bin/CreateISO/\*

在运行这个make时可以使用命令：make V=>文件名（将make执行过程中所有指令都输出到文件中），就能在文件中看到make都执行了哪些命令。make V=输出的执行过程：

#编译createISO为可执行程序

gcc ./Tools/createISO.c -o ./bin/CreateISO/createiso

echo ./bin/BootLoader/bootloaderasm.S.o ./bin/BootLoader/bootloader.c.o ; \ #用于检测显示

#for循环生成bootloader与bootloaderasm的库文件，这里解释以下gcc选项的意义：

#-c只编译到.o文件即可，不生成最终可执行二进制文件，

#-fno-builtin 不使用内建函数

#-Wall 编译后显示所有警告

#-ggdb 生成带gdb调试的相关信息

#-m32生成32位平台代码

#-nostdinc 不要搜索头文件的标准路径(即默认路径),而只搜索-I选项指定的路径和当前路径

#-Os 指定优化等级，Os是最小代码体积

#-fno-stack-protector禁用堆栈保护

#-I指定头文件路径

#-o结果输出文件名

for i in bootloaderasm.S bootloader.c; do

gcc -c ./BootLoader/$i -fno-builtin -Wall -ggdb -m32 -gstabs -nostdinc -Os -fno-stack-protector -I./libs -o ./bin/BootLoader/$i.o; done ; \

#将两个库文件链接成一个库文件，

#-m elf\_i386指定输出架构为i386

#-nostdlib 不链接标准库的库文件，一般用于裸平台链接

#-N

#-T 指定连接脚本

ld ./bin/BootLoader/bootloaderasm.S.o ./bin/BootLoader/bootloader.c.o -m elf\_i386 -nostdlib -N -T ./BootLoader/bootloader.ld -o ./bin/BootLoader/bootloader ; \

#使用objcopy将二进制代码复制到另一个文件中

objcopy -S -O binary ./bin/BootLoader/bootloader ./bin/BootLoader/BL

./bin/BootLoader/bootloaderasm.S.o ./bin/BootLoader/bootloader.c.o

./bin/CreateISO/createiso bin/BootLoader/BL bin/ucore ; \

#使用dd命令生成QEMU使用的镜像文件，该文件大小为10000\*512B

dd if=/dev/zero of=bin/UCore.img count=10000 ; \

#将生成的ISO格式的ucore放入UCore.img

dd if=bin/ucore of=./bin/UCore.img conv=notrunc ; \

bin/BootLoader/BL' size: 369 bytes

build 512 bytes boot sector: 'bin/ucore' success!

这里说一下objcopy是一个复制代码的指令，它可以提取出程序或库文件所有的二进制执行代码，不带编译生成的各种格式头文件（这里也能看到库文件和程序的区别其实不大，感兴趣可以自行查询相关资料）；dd命令是是一个磁盘复制命令，只有使用dd命令才能生成一个QEMU可使用的ISO格式文件，我也不清楚为什么必须这样；最后，你会看到我们只生成了库文件，并将两个库文件链接成了一个库文件，之所以不用gcc直接生成可执行文件后使用objcopy和dd命令的原因是：生成可执行文件后再处理，发现体积会永远大于bootloader所要求的512B(大概是800B，而用链接库那样的方法则是300B左右，差距很大)，具体原因我不清楚。

在makefile中出现了大量的gcc命令，他们都比较复杂，主要有：gcc，ld，objcopy，dd。建议gcc命令参照gcc开发手册，其余命令上网查询了解即可。

在shell中先执行make，然后执行make qemu，就可以看到我们的第一个bootloader运行了起来，这对我们给出的两个bootloader.c的代码操作流程都是这样，只不过第二个只能运行一半。至此，第一章的全部内容全部结束。

第一章代码内容不多，主要是各种工具的使用。万事开头难，我们已经过了最难的一部分，当初仅仅第一章，我翻来覆去的查询资料，一个月才勉勉强强懂了个大概，后来又花了一个月才彻底明白，当初最难的点在于，不知道第一章的架构是什么，管理工程的makefile和gcc的编译过程更是完全不懂，没有一个全局视角和方向，我根本不知道该查哪些资料。所以大家读源代码的时候，需要明白生成一个最终的镜像文件流程是怎么样的，每一步都用到了哪些文件。这才是重要的。

第二章

在第一章中，我们学习了硬件层大量的知识，工程管理，链接脚本，编译命令和重要的bootloader，在之后的章节中主要关注代码，而不是开发环境和编译过程。在第二章中，主要实现第一个由bootloader加载的内核。内核中都有哪些内容？其实写什么程序都可以，可以是输出hello world，但这里，我们选择实现一个内核与用户态的切换。

ELF格式：

承接上一节，最后在bootloader中，我们通过C语言的指针转换为函数跳转进入内核，跳转的入口地址就是这节要写的内核。先来看看那段代码：

//读入操作系统 ELFHDR是一个宏定义，((struct elfhdr \*)0x10000)，表示将从磁盘读入的数据放在0x10000起始处，其实读入的是可执行代码，不过CPU也分不清数据和代码，对它而言都是0101罢了。

//注意ELFHDR前有一个强制类型转换

readseg((uintptr\_t)ELFHDR, SECTSIZE \* 8, 0);

// is this a valid ELF?

//检查是否是正确的ELF文件

if (ELFHDR->e\_magic != ELF\_MAGIC) {

goto bad;

}

//开始按照ELF格式文件加载操作系统的各个段

struct proghdr \*ph, \*eph;

// load each program segment (ignores ph flags)

//读取8个磁盘扇区的内容至ELFHDR处（即1MB处），并判断读取的是否是一个ELF格式的文件。

ph = (struct proghdr \*)((uintptr\_t)ELFHDR + ELFHDR->e\_phoff);

eph = ph + ELFHDR->e\_phnum;

for (; ph < eph; ph ++) {

//开始加载每个段的信息。Ph变量类型是struct proghdr，根据ELF头部信息来获取各个段。

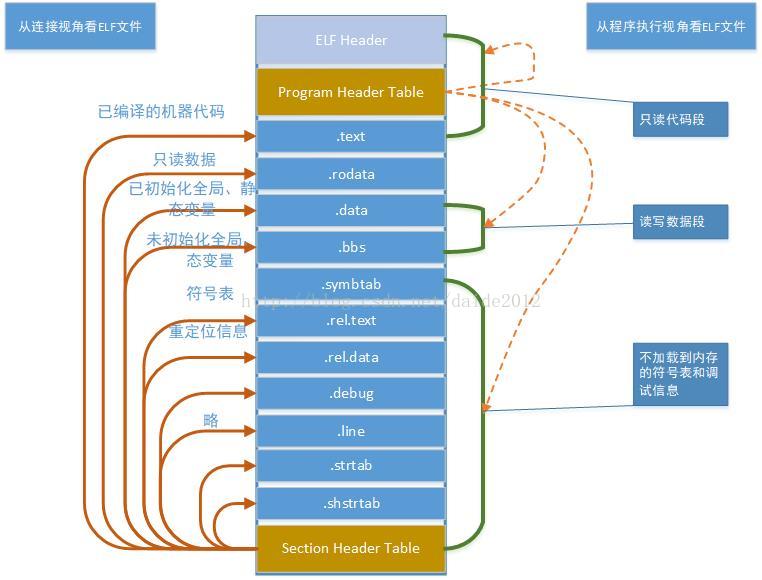
readseg(ph->p\_va & 0xFFFFFF, ph->p\_memsz, ph->p\_offset);

}

## ELF

ELF在计算机科学中，是一种用于二进制文件、可执行文件、目标代码、共享库和核心转储格式文件。是UNIX系统实验室（USL）作为应用程序二进制接口（Application Binary Interface，ABI）而开发和发布的，也是Linux的主要可执行文件格式。1999年，被86open项目选为x86架构上的类Unix操作系统的二进制文件标准格式，用来取代COFF。因其可扩展性与灵活性，也可应用在其它处理器、计算机系统架构的操作系统上。

ELF文件由4部分组成，分别是ELF头（ELF header）、程序头表（Program header table）、节（Section）和节头表（Section header table）。实际上，一个文件中不一定包含全部内容，而且它们的位置也未必如同所示这样安排，只有ELF头的位置是固定的，其余各部分的位置、大小等信息由ELF头中的各项值来决定。

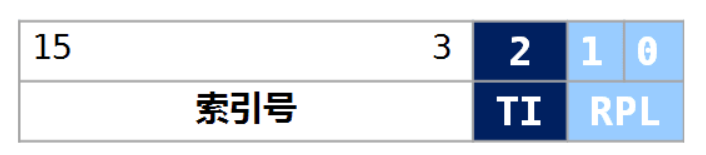


ELF格式

对于ELF我们只需能看懂这一块代码即可。在bootloader中读取了磁盘中后续的8个块数据载入至内存中1MB起始的区域中，然后根据ELF给出的数值通过函数跳转执行内核代码，对于CPU而言，它分不清代码与数据，它也可以将数据按照指令格式执行，不然可以将GDT中的CS段描述符的读写与执行权限修改后看看。

## 特权级

至此，第二章与第一章衔接上了，接下来着重了解第一章提及的保护模式的特权级。接下来所说的段描述符都存储在GDT中。



之前提及的段选择子字段格式

### CPL

CPL：Current Privilege Level，当前特权级

CPL是当前执行的程序或任务的特权级。它被存储在CS的第0位和第1位上（ CS.RPL，这里的RPL不是下文中的RPL请求特权级意思，是指RPL字段）。 通常情况下，CPL代表当前代码的特权级。当程序转移到不同特权级的代码段时，处理器可能改变CPL。通常使用3和0两个值，分别表示用户态和内核态。

### DPL

DPL：Descriptor Privilege Level，描述符特权级

由待访问的数据段或代码段描述符的DPL位域决定。它被存储在GDT中的段描述符或者门描述符的DPL字段中。

### RPL

  RPL：Request Privilege Level，请求特权级

RPL是通过段选择子的第0和第1位表现出来的。RPL是最难理解的一个，不同于CPL与DPL有着清晰且易理解的存在意义，RPL的出现是为了更保险，赋予了当前代码更多的等级。

### CPL，DPL，RPL举例

当前CPU中CS寄存器的值为0x08（二进制：00000000 00001000），表示CS索引GDT中第2条段的信息，则CPL为00。GDT中第二条段描述符的DPL字段值为00，则DPL为0。那么RPL呢？这个例子没有用到RPL。

学习汇编应该见过这种代码风格：

MOV AX , DS:0x3

ljmp $PROT\_MODE\_CSEG, $protcseg

CALL 0X08, A\_FUNCTION

实模式下是用DS值+0x3得到实际内存地址的意思，在保护模式下，使用DS索引GDT中的段，得到该段DPL，起始地址与该段长度，通过特权级检测后（后面讲），起始地址+0x3得到待访问的内存地址，这里，将DS换为一个数字，0x08，0x18等等，都是一样的意思，

MOV AX,0x18:0x3

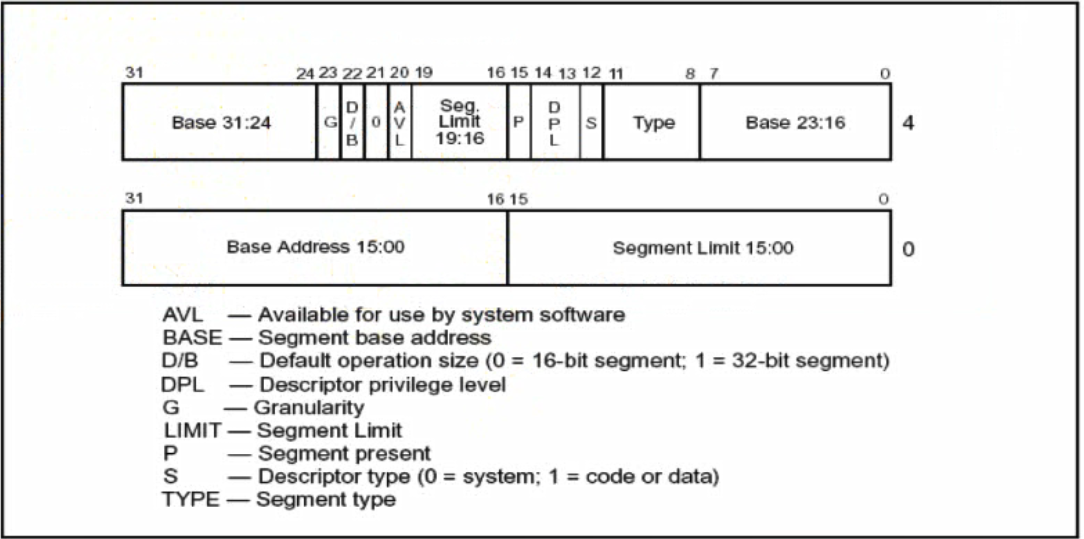
0x18这里做段选择子，上面说过段选择子的字段格式，段选择子的字段格式适用于段寄存器（CS，DS，SS，ES，FS，GS）与段选择子，这里我们第一次见到了段选择子实体，就好比：段选择子的字段格式是类，段选择子是它的对象。段选择子中的RPL字段是真正的RPL，注意区分“RPL字段”与“RPL”。

RPL的值为00，若是0x19，RPL为1，想要访问GDT中的第4条段描述符。

## 段描述符字段格式

CPL，DPL，RPL是基础概念，需要掌握并能轻松指出他们存在的位置。X86保护模式划分特权级是为了保护操作系统正常运行，一般而言，不许越级执行。但用户程序有时需要操作系统帮忙甚至用户等级的程序需要提权成为0级，同时又要防范用户程序恶意利用提权后的特权，x86为此提供了一致/非一致代码段与门来解决。

我们目前已经接触了“代码段或数据段描述符“（即S=1），通过设置 TYPE和S字段，区分代码与数据段。11位的值为1表示代码段，为0表示数据段，8，9，10位根据11位的值有不同的涵义。详细含义见下：



P=1：段描述符有效

P=0：段描述符无效

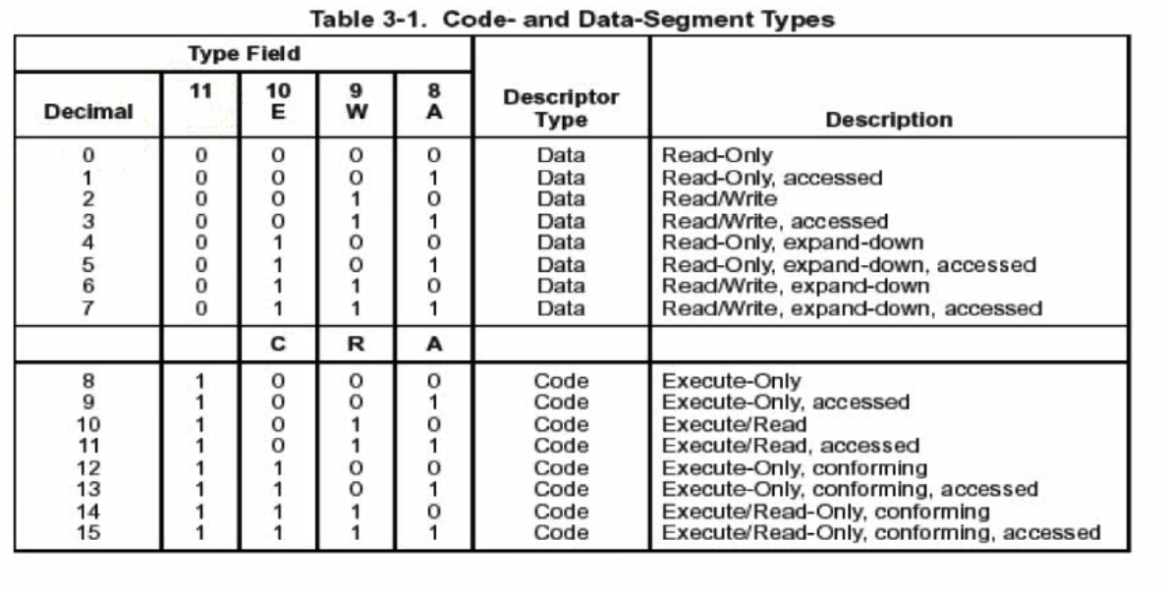
G=0：段寄存器Limit单位为字节，Limit最大值0x000FFFFF

G=1：段寄存器Limit单位为4KB，Limit最大值0xFFFFFFFF

S=1：代码段或数据段描述符

S=0：系统段描述符

**如果S为1：**



数据段：

A:是否被访问过

W:是否可写

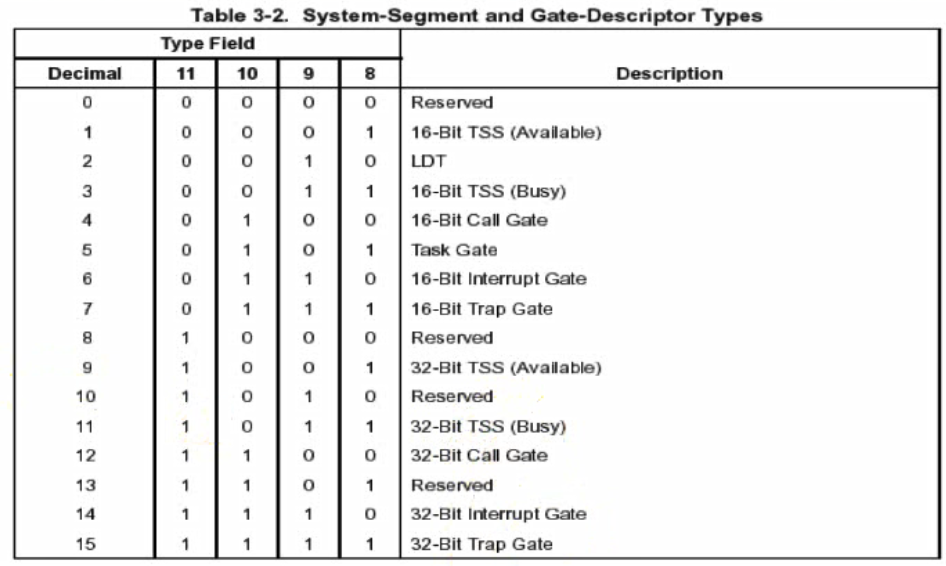
E：是否向下拓展

代码段：

R:是否可读

C：是否为一致代码段

**如果S位为0：**



D/B位：

对CS段：

D=1：采用32位寻址方式

D=0：采用16位寻址方式

对SS段：

D=1：隐式堆栈访问命令使用32位堆栈指针寄存器ESP

D=0：隐式堆栈访问命令使用16位堆栈指针寄存器SP

隐式堆栈访问命令（如push，pop，call）

对向下拓展的数据段：

D=1：段上限4GB

D=0：段上限64KB

通过上面的表格，可以看出，代码（注意，不是数据与门）只有一致代码段与非一致段（Conforming / Nonconforming）之分，且为代码段时，S必须为1，与之对应的S=0时，是门描述符，这样我们可以将段描述符划分如下：

除了数据段（数据段较简单，与代码段是同类，最后介绍即可），其余的段可以根据S字段值分为代码段与门，代码段根据第10位字段C分为一致代码段与非一致代码段，门描述符根据TYPE字段分为：调用门，任务门，中断门，陷阱门。他们的共性都是与特权级有极大关系，都围绕CPL，DPL，RPL的值工作。

## 一致代码段与非一致代码段

什么是一致代码段：简单理解，就是操作系统拿出来被共享的代码段，可以被低特权级的用户直接调用访问的代码。向特权级更高的一致性代码段的控制转移，允许程序以当前特权级继续执行。通常这些共享代码，是"不访问"受保护的资源和某些类型异常处理。对于一致性代码来说，特权级高的程序不允许访问特权级低的数据:即核心态不允许调用用户态的数据；特权级低的程序可以访问到特权级高的数据。但是特权级不会改变：用户态还是用户态。

### 访问一致性代码段的特权级检查规则：

CPL>=DPL，不检查RPL（即：对CALL或者JMP指令提供的段选择子中的RPL不管）

这里的DPL不是指CS寄存器索引得到的段描述符的DPL，而是欲访问的一致代码段的段描述符（存储在GDT）中的DPL。

非一致性代码段：避免低特权级访问而被操作系统保护起来的系统代码。向特权级不相等的非一致代码段转移将导致“一般保护异常”，除非使用任务门或者调用门。非一致性代码段只允许同级间访问，绝对禁止不同级访问:核心态不用用户态，而用户态也不使用核心态。

### 非一致代码段的特权级检查：

CPL==DPL && RPL<= DPL

通过检查后，目标段描述的DPL会赋给CPL，但由于要求CPL==DPL，所以CPL的特权级还是没有变化。这里用一个新的DDPL（Destination DPL，欲访问的段的DPL），即CPL会等于DDPL。

产生一致性代码和非一致性代码的主要原因是：单纯的0-3特权级只能保证高特权级可以访问特权级的东西，而低特权级的段有时候要访问内核数据段，此时就需要一些灵活策略。

不访问保护措施的系统工具和某些异常类型的处理过程需要放在一致性代码段中。需要防止低特权级程序访问的工具要放在非一致代码段中。

每当调用门用于把程序控制转移到一个更高级别的非一致性代码段时，CPU会自动切换到目的代码段特权级的堆栈去（堆栈切换详细流程后面介绍）。

## 门描述符：

门（GATE），从字面意思看不出什么。x86中门的作用如同大坝的船闸（负责将船从大坝水位低的一面送到水位高的一面），只不过这里的抬升的不是船，而是当前代码的特权级CPL，门就是船闸，当前代码跳转（JMP，CALL）时指明段选择子（指出要进入哪个船闸，假设这个大坝有多个船闸，并连接多条河流，各个河流高度不同，这里的高度也就是特权级高低），如果该段选择子索引的是GDT中的一个门，那么当前代码的特权级就提升到该门给出的特权级（通过了船闸，进入了另一面，船的高度得到了抬升），然后就可以执行其他高特权级的代码了。不过x86的船闸不是想进入就能进入的，各个船闸都有进入门槛，门槛就是门描述符中的DPL值，只有大于这个DPL，才能访问这个门。门还是为一致代码段和非一致代码段服务的（一些代码段不允许低级代码段随便执行，通过门可以安全的限制用户代码）。门可以临时给低特权级代码提供一个高特权级的身份，同时它还可以限制用户代码提权后的执行范围，如何限制后面展开讲。

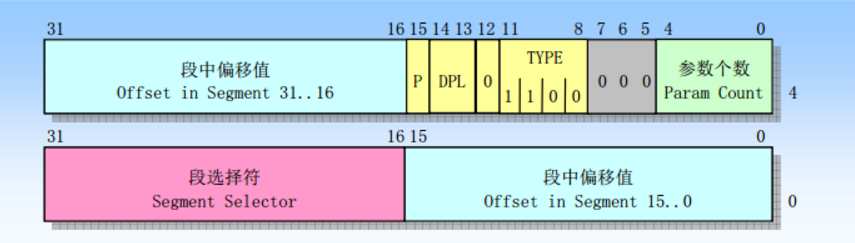
门总共有四种：任务门，调用门，中断门，陷阱门。这里面，任务门Linux没有使用，不再讲解，可自行搜索学习。其余的三种门都很重要。

三种门的格式都大同小异：

## 调用门：

### 结构

　　本质上，它只是一个描述符，一个不同于代码段和数据段的描述符，可以安装在GDT或者LGT中，但是不能安装在IDT（中断描述符表）中。它主要是定义了目标代码对应段的选择子、入口地址的偏移和一些属性等。结构跟代码段数据段描述符大不相同。结构如下图所示：



一个门描述了由一个段选择符（即段选择子）和一个“段中偏移值”所指定的线性地址，程序正是通过这个线性地址进行转移的。TYPE字段指出这是什么门，1100表示调用门；描述符中的P位是有效位，通常是1，当它为0时，调用这样的门会导致处理器产生异常；参数个数字段指明在发生堆栈切换时从调用者堆栈复制到新堆栈中的参数个数；DPL字段指出要访问这门所需的最低特权级；段选择符按照段选择子字段格式解释，其中的高13位用于GDT中索引一个代码段（一致或非一致的），随后就会跳转进入这个代码段，低2位是通过门检查后，代码新的特权级的值；段中偏移量与新的代码段起始地址相加，得到跳转入口，具体如下。

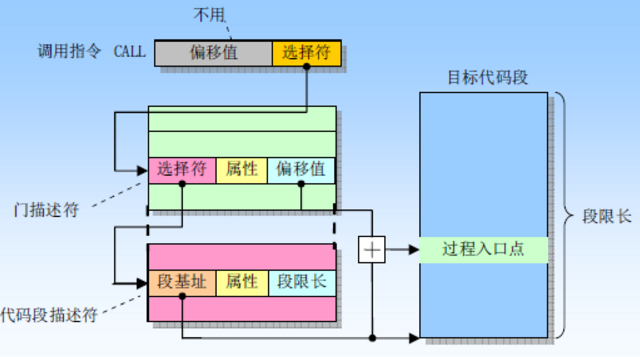
### 通过调用门访问代码段

调用门的访问一般通过call、jmp指令的操作数提供的一个远指针，该指针中的段选择子用于指定调用门，如：

JMP A\_GATE, OFFSET

A\_GATE是一个段选择子，索引GDT中一个门描述符，A\_GATE.RPL值是RPL，OFFSET其实没有用

CPU会使用调用门中的偏移值实现跳转。如下图：



首先通过call或者jmp指令给出的段选择子，在GDT中索引门描述符，通过门权限检查后，使用门描述符给出的选择符与偏移量进行下一步。使用<选择符>给出的特权级在GDT中根据选择子索引<代码段描述符>（到了这一步，可以想成<选择符>将自己的.RPL数值赋给了CS，即将当前代码提权，这也正是门的本有作用：提权，然后进行下一步），然后根据代码段描述符给出的起始地址加门描述符的偏移值得到最终的过程入口点。这个过程中，都没使用call或者jmp给出的offset值，最终的入口点，全由调用门决定。

各步骤检查规则如下：

通过调用门进行程序的转移控制时，CPU会检查以下这几个字段：

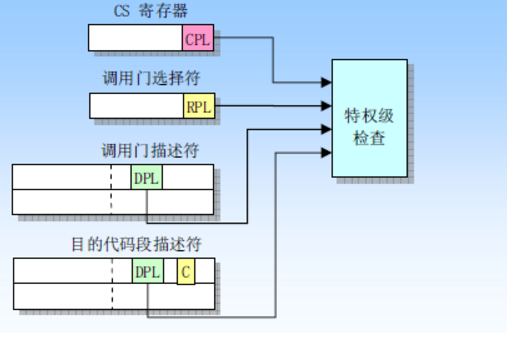
1.当前代码的CPL；

2.调用门描述符中的DPL；

3.调用门描述符中选择符的RPL；

4.目的代码描述符的DPL（目的代码描述符即上图中的代码段描述符）；

5.目标代码段描述符中的一致性标志C（一致与非一致下面会提到）。如下图：



### 对于call和jmp检查规则

　　对call来说，有所不同：当前CPL<=调用门描述符DPL，RPL<=调用门描述符DPL，当前CPL>=目的代码段描述符DPL；

对jmp来说：除了跟call的“当前CPL<=调用门描述符DPL，RPL<=调用门描述符DPL”一样外，如果目的代码段的一致的话，CPL>=目的代码段的DPL，而如果目的代码段是非一致的话，CPL==目的代码段的DPL。

通过门后访问一致代码与非一致代码本质还是回到了之前讲的一致代码中。

从上面的流程中也可以看到，DPL在各个段描述符中的都是充当门槛的作用，防止过低等级的代码访问该段。

另外，只有call指令可以将代码通过调用门转移到特权级更高的非一致性代码之中。对于非一致性代码的成功转移，CPL被目的代码的DPL刷新，会引起堆栈切换；对于一致性代码，不会刷新，也不会切换。堆栈切换随后讲。

调用门的作用是，让一个代码段中的过程被不同特权级的程序访问。通常用于低特权级代码来访问高特权级的代码段。

## C语言内存分布

BSS段:（bss segment）通常是指用来存放程序中 未初始化的 全局变量的一块内存区域。BSS是英文Block Started by Symbol的简称。BSS段属于静态内存分配。

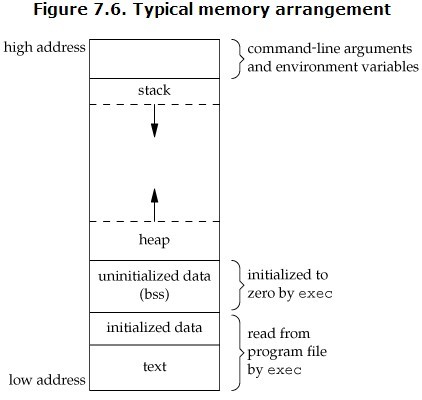
数据段：数据段（data segment）通常是指用来存放程序中 已初始化的 全局变量的一块内存区域。数据段属于静态内存分配。

代码段：代码段（code segment/text segment）通常是指用来存放 程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于 只读, 某些架构也允许代码段为可写，即允许修改程序。在代码段中，也有可能包含一些 只读的常数变量，例如字符串常量等。

堆（heap）：堆是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩张或缩减。当进程调用malloc/free等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）/释放的内存从堆中被剔除（堆被缩减）。每次free时，我们不需要告诉需要free多少空间，但free函数却总能释放对的空间，原因在于malloc()空间为n字节时，分配的实际时n+1字节，前面的那个字节中记录本次分配多少空间，free时先读取前一个字节便可得知释放的空间。

栈(stack)：栈又称堆栈， 存放程序的 局部变量（但不包括static声明的变量， static意味着 在数据段中存放变量）。除此以外，在函数被调用时，栈用来传递参数和返回值。由于栈的先进先出特点，所以栈特别方便用来保存/恢复调用现场。

这里的堆不是数据结构中的堆。代码编译完后，代码段，数据段吗，BSS段的长度都固定下来了，唯有堆和栈段是代码中随意分配的，不具有可预知性，Intel的栈是入栈自减ESP，所以栈被安排到了高地址处，每次函数调用是向低地址分配空间。堆安排到了BSS段结束处，堆每次分配空间是向上分配，用完后通过free函数回收。



栈是记录函数的调用过程，抽象一点，它记录的是程序的处理逻辑；堆则是记录数据。

## GCC的段与x86段的关系

在讲保护模式特权级的变化引起的堆栈切换前，先来了解一下GCC编译器生成的各个段都是哪些（此段与x86段的含义不同）：

栈（stack）：编译器自动分配，翻译的变量。存放函数的参数与局部变量值。操作方式类似于数据结构中的栈。由x86的ESP与EBP寄存器结合起来工作。

堆（heap）：由程序员分配释放。如果程序员没有释放，程序结束时由操作系统释放。它不是数据结构中所说的堆。操作方式类似于链表。C++中就是new与delete开辟的空间。

BSS段：存放未初始化的全局变量与静态变量。

数据段DATA：存放初始化之后的全局变量和静态变量。

代码段TEST：程序代码，为二进制格式。

GCC编译器生成代码的各个段对应X86保护模式的各个段寄存器是：

堆与栈段与SS寄存器对应；数据段和BSS与DS寄存器段对应；代码段TEXT与CS寄存器对应。在此基础上，ESP，EBP寄存器与SS寄存器结合使用。

## ESP与EBP概念：

ESP：栈指针寄存器(extended stack pointer)，其内存放着一个指针，该指针永远指向系统栈最上面一个栈帧的栈顶。ESP的值会随PUSH与POP指令的使用改变其值，从而永远指向栈顶。

EBP：基址指针寄存器(extended base pointer)，其内存放着一个指针，该指针永远指向系统栈最上面一个栈帧的底部。

ESP与EBP存在的意义：为了使用堆栈框架 ，堆栈框架（stack frame）也称活动记录（activation record），它是为传递参数，子例程（即调用函数）的返回值，局部变量和保存的寄存器保留的对堆栈空间。他按以下步骤创建：

1. 如果有传递的参数，则压入堆栈
2. 子例程被调用，子例程的返回地址压入堆栈
3. 子例程开始执行，EBP被压入堆栈
4. EBP被设为ESP的值，从这时开始，EBP就被作为寻址所有子例程参数的基址指针使用了
5. 如果有局部变量，ESP减去一个数值，以便在堆栈上为局部变量保留空间
6. 如果任何寄存器需要保存，则压入堆栈

注意第四步。在第四步，ESP与EBP的使用联系在一起。EBP正好处在子例程局部变量（C语言函数的{ }之内，函数体内的变量）与非局部变量（函数的返回值，型参）的分界线上。传递完参数后，以后需要修改型参的值，都是通过[EBP+OFFSET]来实现。注意不要混淆ESP，EBP与SS寄存器的作用，后面会将C语言与汇编混合使用这个知识，故需要着重掌握。

读取变量数据。关于.bss .rodata等GCC编译生成的段更多信息可以自行搜索了解。

## 堆栈段和数据段访问的特权级检查规则

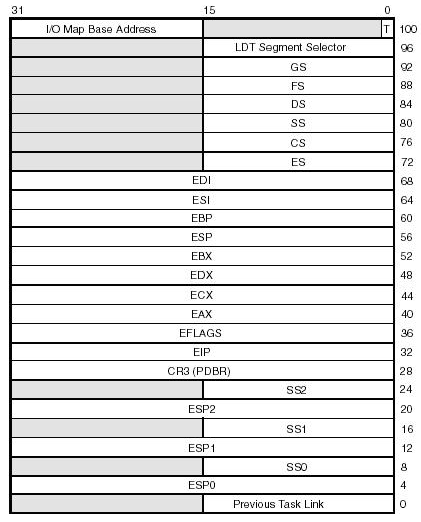
数据段（DS）：CPL<=数据段DPL（DS.RPL） && RPL<=数据段DPL(DS.RPL)。允许高特权级访问低特权级的数据。

堆栈段（SS）：CPL==RPL==DPL。不允许跨特权级访问数据。

堆栈保护的核心是切换堆栈，但代码执行结束后，还需切换回去，所以要记录切换的堆栈信息，这些信息记录在TSS中（TSS还记录了其他各种信息）。

## TSS

### TSS字段格式



TSS在任务切换过程中起着重要作用，通过它实现任务的挂起和恢复。所谓任务切换是指，挂起当前正在执行的任务，恢复或启动另一任务的执行。在任务切换过程中，首先，处理器中各寄存器的当前值被自动保存到TR（任务寄存器）所指定的TSS中；然后，下一任务的TSS的选择子被装入TR；最后，从TR所指定的TSS中取出各寄存器的值送到处理器的各寄存器中。由此可见，通过在TSS中保存任务现场各寄存器状态的完整映象，实现任务的切换。 Linux中并没有按照Intel设想的使用TSS，按照Intel的设计，切换一次进程实在太耗费时间了，仅仅一个切换汇编指令就花费300个CPU时间片。

TSS的基本格式由104字节组成。这104字节的基本格式是不可改变的，但在此之外系统软件还可定义若干附加信息。基本的104字节可分为链接字段区域、内层堆栈指针区域、地址映射寄存器区域、寄存器保存区域和其它字段等五个区域。（下面的内容理解即可，不必深入理解）

(1)寄存器保存区域 寄存器保存区域位于TSS内偏移20H至5FH处，用于保存通用寄存器、段寄存器、指令指针和标志寄存器。当TSS对应的任务正在执行时，保存区域是未定义的；在当前任务被切换出时，这些寄存器的当前值就保存在该区域。当下次切换回原任务时，再从保存区域恢复出这些寄存器的值，从而，使处理器恢复成该任务换出前的状态，最终使任务能够恢复执行。从上图可见，各通用寄存器对应一个32位的双字，指令指针和标志寄存器各对应一个32位的双字；各段寄存器也对应一个32位的双字，段寄存器中的选择子只有16位，安排再双字的低16位，高16位未用，一般应填为0。

(2)内层堆栈指针区域：为了有效地实现保护，同一个任务在不同的特权级下使用不同的堆栈。例如，当从外层特权级3变换到内层特权级0时，任务使用的堆栈也同时从3级变换到0级堆栈；当从内层特权级0变换到外层特权级3时，任务使用的堆栈也同时从0级堆栈变换到3级堆栈。所以，一个任务可能具有四个堆栈，对应四个特权级。四个堆栈需要四个堆栈指针。TSS的内层堆栈指针区域中有三个堆栈指针，它们都是48位的全指针(16位的选择子和32位的偏移)，分别指向0级、1级和2级堆栈的栈顶，依次存放在TSS中偏移为4、12及20开始的位置。当发生向内层转移时，把适当的堆栈指针装入SS及ESP寄存器以变换到内层堆栈，外层堆栈的指针保存在内层堆栈中。没有指向3级堆栈的指针，因为3级是最外层，所以任何一个向内层的转移都不可能转移到3级。但是，当特权级由内层向外层变换时，并不把内层堆栈的指针保存到TSS的内层堆栈指针区域。实际上，处理器从不向该区域进行写入，除非程序设计者认为改变该区域的值。这表明向内层转移时，总是把内层堆栈认为是一个空栈。因此，不允许发生同级内层转移的递归，一旦发生向某级内层的转移，那么返回到外层的正常途径是相匹配的向外层返回。

(3)地址映射寄存器区域：从虚拟地址空间到线性地址空间的映射由GDT和LDT确定，与特定任务相关的部分由LDT确定，而LDT又由LDTR确定。如果采用分页机制，那么由线性地址空间到物理地址空间的映射由包含页目录表起始物理地址的控制寄存器CR3确定。所以，与特定任务相关的虚拟地址空间到物理地址空间的映射由LDTR和CR3确定。显然，随着任务的切换，地址映射关系也要切换。 [Page] TSS的地址映射寄存器区域由位于偏移1CH处的双字字段(CR3)和位于偏移60H处的字字段(LDTR)组成。在任务切换时，处理器自动从要执行任务的TSS中取出这两个字段，分别装入到寄存器CR3和LDTR。这样就改变了虚拟地址空间到物理地址空间的映射。 但是，在任务切换时，处理器并不把换出任务但是的寄存器CR3和LDTR的内容保存到TSS中的地址映射寄存器区域。事实上，处理器也从来不向该区域自动写入。因此，如果程序改变了LDTR或CR3，那么必须把新值人为地保存到TSS中的地址映射寄存器区域相应字段中。可以通过别名技术实现此功能。

(4)链接字段：链接字段安排在TSS内偏移0开始的双字中，其高16位未用。在起链接作用时，地16位保存前一任务的TSS描述符的选择子。如果当前的任务由段间调用指令CALL或中断/异常而激活，那么链接字段保存被挂起任务的 TSS的选择子，并且标志寄存器EFLAGS中的NT位被置1，使链接字段有效。在返回时，由于NT标志位为1，返回指令RET或中断返回指令IRET将使得控制沿链接字段所指恢复到链上的前一个任务。

(5)其它字段：为了实现输入/输出保护，要使用I/O许可位图。任务使用的I/O许可位图也存放在TSS中，作为TSS的扩展部分。在TSS内偏移66H处的字用于存放I/O许可位图在TSS内的偏移(从TSS开头开始计算)。关于I/O许可位图的作用，以后的文章中将会详细介绍。在TSS内偏移64H处的字是为任务提供的特别属性。在80386中，只定义了一种属性，即调试陷阱。该属性是字的最低位，用T表示。该字的其它位置被保留，必须被置为0。在发生任务切换时，如果进入任务的T位为1，那么在任务切换完成之后，新任务的第一条指令执行之前产生调试陷阱。

### 堆栈切换

如果通过调用门把控制转移到了更高特权级的非一致代码段中，那么CPL就会被设置为目标代码段的DPL值，并且会引起堆栈切换。为什么要切换堆栈呢？原因有以下几点：

因为栈段的特权级必须同CPL保持一致；

防止高特权级程序由于栈空间不足而崩溃；

防止低特权级程序通过共享的栈有意或无意地干扰高特权级程序。

为了切换栈，每个任务除了自己的固有栈之外，还必须额外定义一套或多套栈，具体多少个堆栈取决于任务的特权级别。0特权级的任务不需要额外的栈，因为除了从调用高特权级的例程（通常是操作系统例程）返回外，不允许将控制从特权级高的代码段转移到特权级低的代码段——操作系统不会引用可靠性比自己低的代码；1特权级的任务需要额外定义一个DPL为0的栈，以便将控制转移到0特权级时使用；2特权级的任务需要额外定义两个栈，其DPL分别为0和1；3特权级的任务最多额外定义三个栈，其DPL分别为0、1、2。

### 转移和返回的具体过程

首先，通过调用门进行控制转移，可以使用jmp far或者call far指令。指令执行时，段选择子必须指向调用门，32位的偏移量可以是任意值（会被CPU忽略）。其次，必须符合上文第3节提到的检查规则。

当使用call far指令通过调用门转移控制时，如果改变了CPL，则必须切换栈，即从当前任务的固有栈切换到与目标代码段特权级相同的栈上。栈的切换是由处理器固件自动进行的。

当前栈是由SS和ESP的当前内容所指示的。要切换到的新栈的相关信息位于当前任务的TSS中，处理器知道如何找到它。栈切换过程如下：

1. 根据目标代码段的DPL（也就是新的CPL）到当前任务的TSS中读取新栈的选择子和栈指针。在读取栈选择子、栈指针或者栈段描述符的过程中，任何违反段界限的错误都将导致产生一个无效TSS异常。

2. 检查栈段描述符的特权级和类型是否有效，若无效同样产生一个无效TSS异常。

3. 临时保存SS和ESP的当前值，把新栈的选择子和栈指针加载到SS和ESP中。然后把临时保存的SS和ESP的内容压入新栈中。

4. 根据调用门描述符中“参数个数”字段，把旧栈中的所有参数复制到新栈中。如果参数个数为0，则不复制参数。

5. 将当前CS和EIP的内容压入新栈。通过调用门实施的控制转移一定是远转移，所以要压入CS和EIP。

6. 从调用门描述符中把目标代码段的选择子和段内偏移值传送到CS和EIP中，开始执行被调用过程。

相反，如果没有改变特权级别，则不切换栈，继续使用调用者当前的栈，只是在原来的基础上压入当前CS和EIP的内容。

另外，如果通过调用门的控制转移是jmp far指令发起的，结果就是“肉包子打狗——有去无回”，且没有特权级变化，也不需要切换栈。相反，如果是call far指令发起的，则可以使用远返回指令retf把控制返回到调用者。

有几点需要注意：

1. 每个栈必须可读可写，并且具有足够的空间来存放以下信息：

(1)调用过程的SS、ESP、CS和EIP寄存器的内容  
(2)被调用过程的参数和临时变量所需使用的空间  
(3)当隐含调用一个异常或者中断过程时标志寄存器EFLAGS和出错码使用的空间

2. 由于一个过程可以调用其他过程，因此每个栈必须有足够的空间来容纳多帧信息

3. TSS中的中的SSx、ESPx（x=0,1,2）字段是静态的，除非软件进行修改，处理器从来不会改变它们。举例来说，假设操作系统为一个用户任务的TSS填写了ESP0，其值为0x800；当这个任务通过调用门进入0特权级的代码段时，会切换到0特权级堆栈，堆栈指针ESP的初始值就是0x800；返回时，假设ESP变成了0x808，处理器并不会把0x808更新到TSS中的ESP0域；下次再通过调用门进入0特权级代码段时，使用的还是当初设置的静态值0x800。

### 返回的过程

对于相同特权级的返回，CPU从堆栈中弹出EIP和CS；会发生特权级改变的远返回仅允许返回到低特权级程序中，即返回到的代码段的DPL在数值上要大于CPL。返回的全部过程如下：

1. 检测被调用者栈中CS寄存器的RPL字段值，以确定在返回时特权级是否发生改变。
2. 弹出并使用被调用过程栈上的值加载EIP和CS寄存器。在此过程中会对代码段描述符和代码段选择子的RPL进行特权级与类型检查。
3. 如果远返回指令是带参数的，则将参数和ESP寄存器的当前值相加，以跳过被调用者栈中的参数部分，最后的结果是ESP寄存器指向调用者SS和ESP的压栈值。注意，retf指令的参数必须等于调用门中所有参数的总字节数之和。
4. 如果返回时需要改变特权级，则从栈中将ESP和SS弹出，并把值代入寄存器ESP和SS，切换到调用者的栈。
5. 如果远返回指令是带参数的，则将参数和ESP寄存器的当前值相加，以跳过调用者栈中的参数部分，最后的结果是调用者的栈恢复平衡。
6. 如果返回时需要改变特权级，则检查DS,ES,FS和GS的内容，如果段选择子指向数据段或者非一致代码段且段描述符的DPL在数值上小于返回后的新CPL，那么就把数值0传送到该段寄存器。

## 实模式/保护模式下的中断

首先介绍IDT（中断描述符表，与GDT地位等价），它由IDTR指出内存中的起始地址，在保护模式下，中断向量表中存储的是和调用门类似的门描述符，通过门描述符获得中断处理程序的入口地址。X86体系下最多有256个中断（中断号：0x00-0xff）。不论谁触发了中断，都会有一个中断号：n，中断号n乘8，加上IDT起始地址，得到中断处理描述符，根据中断描述符中的处理地址跳转到处理函数。

实模式下的中断入口叫中断向量表，保护模式下入口是IDT，我们不再讨论实模式下的中断具体形式（可自行上网搜索），二者的基本都相同。低32个（0~31）中断号是Intel自保留的，以上是我们可以随意使用的，linux习惯使用0x80号做系统调用，另外，外设中断的处理不受特权级限制，这很好理解，你正在使用一个用户级的软件，它需要联网，网卡触发中断让该程序去接受数据，总不能等到切换回内核态再处理，那样该用户级的软件网络延迟会很大，尤其当你打联机游戏时。

我们习惯用中断笼统的称呼所有需要当前程序停下来去短暂处理的事件。这里的中断指的是：中断和异常。

狭义的中断还可划分为：硬中断和软中断。

硬中断，由8259A（这个熬死了8086，奔腾的老家伙，我看还能熬死酷睿，堪称Intel最成功的芯片）或者I/O APIC（和8259A都是管理外设的）发出中断信号，请求CPU来处理，如：定时器，时钟发除的。他两都通过数根引线与CPU相连，硬中断的发生时间完全是随机的，由外设说了算。

软中断：int n触发（n就是中断号），比如开机获取内存信息就是通过int 0x15指令来获得，这是BIOS中断，后面我们还要用到他呢。它是软件模拟硬中断的一种方式。

异常：和狭义中断最大的不同是，狭义中断都是外部产生的（CPU外部），而且不是很严重，但是异常基本都是CPU产生的，并且部分异常是无力回天的，会造成整个操作系统崩溃（解决方法就是写好代码，别让他们出现）。异常按照严重性分为三种：

1. 故障（faults）：故障通常都能纠正，比如内存缺页故障
2. 陷阱（trap）：由CPU现行指令引起
3. 终止（aborts）：最严重。如硬件错误，描述符数据不一致或无效。该类异常无法准确报告错误指令位置，发生时，程序或者任务都不可能重新启动

中断和异常发生时，CPU会挂起当前进程或者任务，然后执行中断处理程序，而被打断的程序不失连续性，除非遇到终止类型的异常。一些异常和中断，CPU还会压入一个错误代码以帮助处理程序诊断。

中断和陷阱的主要区别：

1. 陷阱通常由处理机正在执行的现行指令引起，而中断则是由与现行指令无关的中断源引起的。
2. 陷阱处理程序提供的服务为当前进程所用，而中断处理程序提供的服务则不是为了当前进程的。
3. CPU 在执行完一条指令之后，下一条指令开始之前响应中断，而在一条指令执行中也可以响应陷阱。
4. 在有的系统中，陷入处理程序被规定在各自的进程上下文中执行，而中断处理程序则在系统上下文中执行。

软中断与硬中断的比较：

相同点：其中断源发中断请求或软中断信号后， CPU 或接收进程在适当的时机自动进行中断处理或完成软中断信号所对应的功能。

不同点：接收软中断信号的进程不一定正好在接收时占有处理机，而相应的处理必须等到该接收进程得到处理机之后才能进行。

当然，中断还由可屏蔽与不可屏蔽之分，那些都是后话了。

### BIOS中断服务程序：

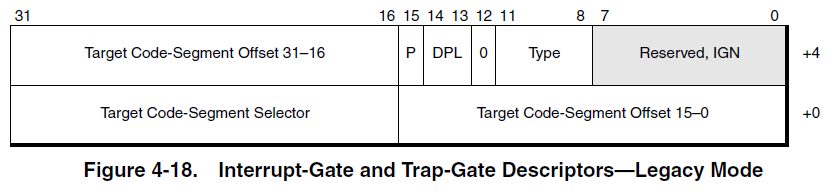
BIOS中断服务程序：封装底层操控外设的代码，他们存放在0-1KB内存处，只可以在实模式下使用，使用形式int n形式。很多人看到保护模式的中断号分布后很容易困惑，保护模式下的中断号都是Intel CPU的，BIOS的中断服务却既有CPU的，也有集成操作外设的，同一个中断号居然前后功能不一样？注意：BIOS只在实模式下工作，进入保护模式后就消失了，他们两个并不冲突，也就是说进入保护模式后还需按照保护模式的中断号再写一遍中断处理程序，我们的代码没有在实模式逗留，快速进入了保护模式。

### BIOS中断服务一览表：

|  |  |  |  |
| --- | --- | --- | --- |
| 类型号 | 中断处理程序功能 | 类型号 | 中断处理程序功能 |
| 0 | 除数为0中断 | 1 | 单步中断 |
| 2 | NMI中断 | 3 | 断点中断 |
| 4 | 溢出中断 | 5 | 打印屏幕 |
| 6 | 保留 | 7 | 保留 |
| 8 | 定时中断（ 中断） | 9 | 键盘中断（ 中断） |
| A | 保留（ 中断） | B | 串行口COM2（ 中断） |
| C | 串行口COM1（ 中断） | D | 硬盘中断（ 中断） |
| E | 软盘中断（ 中断） | F | 打印机中断（ 中断） |
| 10 | CRT显示驱动程序 | 11 | 设备检测 |
| 12 | 存储器容量检测 | 13 | 磁盘I/O驱动程序 |
| 14 | RS-232 I/O驱动程序 | 15 | 磁带机I/O处理 |
| 16 | 键盘I/O驱动程序 | 17 | 打印机驱动程序 |
| 18 | ROM BASIC | 19 | 系统引导 |
| 1A | BIOS日期、时钟中断 | 1B | BIOS提供的Ctrl\_Break处理 |
| 1C | 定时器时标（54.9ms） | 1D | CRT初始化参数表 |
| 1E | 磁盘参数 | 1F | 图形字符集 |
| 20 | 程序结束 | 21 | DOS系统功能调用 |
| 22 | 结束地址 | 23 | DOS的Ctrl\_Break处理 |
| 24 | 严重错误处理 | 25 | 磁盘顺序读 |
| 26 | 磁盘顺序写 | 27 | 程序结束且驻留（TSR） |
| 28～2E | 为DOS保留 | 2F | DOS内部使用 |
| 30～3F | DOS保留 | 40～7F | 未用 |
| 80～85 | BASIC保留 | 86～F0 | BASIC使用 |
| F0～FF | 未用 | / | / |

BIOS的中断服务是开机BIOS执行加载到内存中的低0x00~0x3FF中的，也就是实模式下的中断向量表。

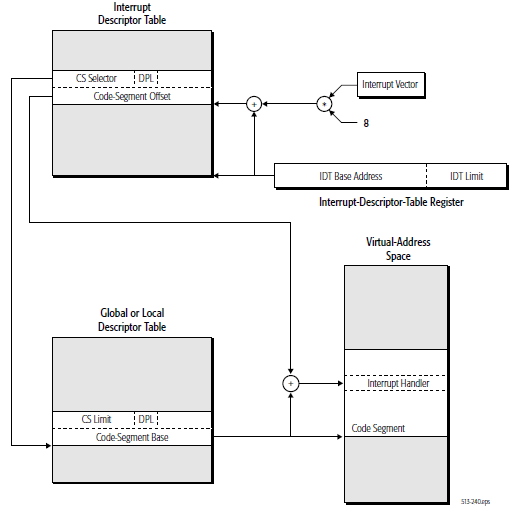
### 中断与陷阱描述符字段：



两者字段格式相同，通过type来区分。trap gate 与 interrupt gate 不同的一点是：**使用 trap gate 的，processor 进入 interrupt handler 前并不改变 eflags.IF 标志**，这意味着在 interrupt handler 里将允许可屏蔽中断的响应。

type：1110是中断门；1111是陷阱门

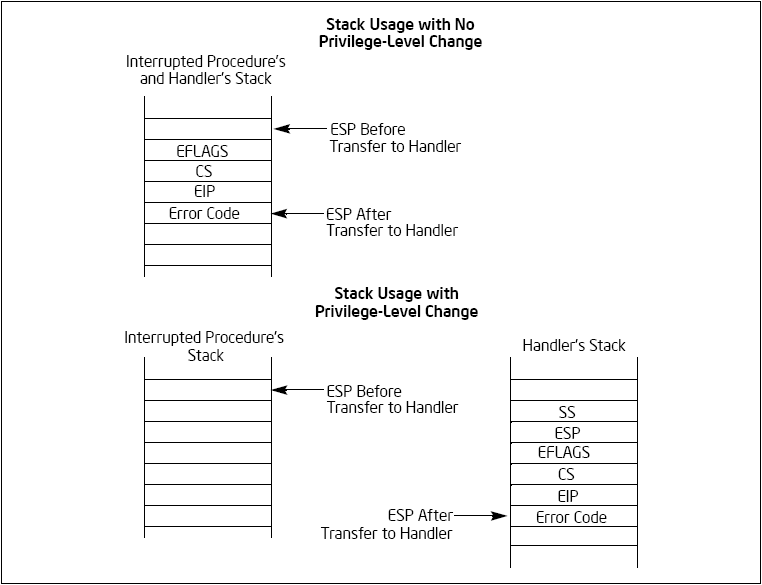
### 中断处理流程



在发生中断时，processor 在 IDTR.base 里可以获取 IDT 的地址，根据中断号在 IDT 表里读取 descriptor，descriptor 的职责是给出 interrupt handler 的入口地址，processor 会检查中断号（vector）是否超越 IDT 的 limit 值。

32 位模式下的过程是：

1. 从 IDTR.base 得到 IDT 表地址
2. 从 **IDTR.base + vector \* 8**（每个 descriptor 为 8 bytes）处读取 8 bytes 宽的 escriptor
3. 对 descriptor 进行分析检查，包括：
   * descriptor 类型的检查
   * IDT limit 的检查
   * 访问权限的检查
4. 从 gate descriptor 里读取 selector
5. 判断 code segment descriptor 是存放在 **GDT** 表还是 **LDT** 表
6. 使用 selector 从 descriptor table 读取 code segment descriptor，当然这里也要经过对 code segment descriptor 的检查
   * descriptor 类型检查
   * GDT limit 检查
   * 访问权限的检查
7. CPU根据特权级的判断设定即将运行的中断服务程序要使用的栈的地址。CPU会根据CPL和中断服务程序段描述符的DPL信息确认是否发生了特权级的转换，比如当前程序正运行在用户态，而中断程序是运行在内核态的，则意味着发生了特权级的转换，这时CPU会从当前程序的TSS信息（该信息在内存中的首地址存在TR寄存器中）里取得该程序的内核栈地址，即包括ss和esp的值，并立即将系统当前使用的栈切换成新的栈。这个栈就是即将运行的中断服务程序要使用的栈。紧接着就将当前程序使用的ss,esp压到新栈中保存起来。
8. 保护当前现场：CPU开始利用栈保护被暂停执行的程序的现场：依次压入当前程序使用的eflags，cs，eip，errorCode（如果是有错误码的异常）信息



保护现场前后栈的变化

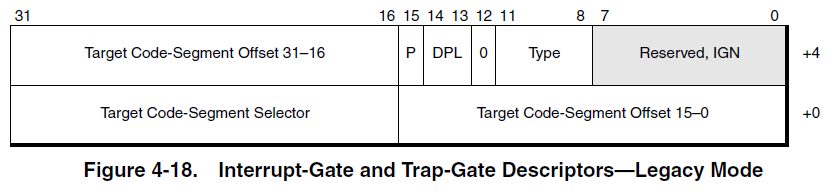
1. 从 code segment descriptor 中读取 code segment base 值
2. 从 gate descriptor 里读取 interrupt handler 的 offset 值
3. 得取 interrupt handler 的入口地址：**base + offset**，转入执行 interrupt handler

### 中断处理程序执行结束返回

在每个中断服务程序的最后，必须有中断完成返回先前程序的指令，这就是iret（或iretd）。程序执行这条返回指令时，会从栈里弹出先前保存的被暂停程序的现场信息，即eflags,cs,eip重新开始执行。如果存在特权级转换还会弹出ss和esp，这样也意味着栈也被切换回原先使用的栈了。

这里有个地方需要注意：如果此次处理的是带有错误码(errorCode)的异常，CPU在恢复先前程序的现场时，并不会弹出errorCode，也就是说CPU似乎忘记了曾经压过一个errorCode入栈，因此要求相关的中断服务程序在调用iret返回之前需要主动弹出errorCode。

### 中断与陷阱的权限检测：



**CPL**：当前 processor 所处的权限级别

**DPLg**：代表 **DPL of gate**，也就是 IDT 中 gate descriptor 所要求的访问权限级别

**DPLs**：代表 **DPL of code segment**，也就是 interrupt handler （在GDT中，不是用户代码段就是内核代码段）的目标 code segment 所要求的访问权限级别

规则：

CPL<= DPLg

CPL>= DPLs

在设置门描述符时，还需注意中断或陷阱描述的target code-segment selector字段的权限要高于等于DPLs。

中断学习博客：

<https://www.cnblogs.com/t1mes/articles/7905336.html>

## I/O端口

目前我们的工作只接触CPU和内存，外设还没有接触到。但现在要接触他们了，所以先学IO端口。

何为端口（port）？就是一些寄存器，不过他们没有呆在CPU中（AX，BX这些呆在CPU中），而是呆在外设中，每个外设可以有多个IO端口，不过一般各个设备都会遵守协议，只占用划分给自己的端口号范围，比如划分给键盘的端口号区间就是：0x060-0x06f。CPU通过端口给外设发数据，操控外设执行各种命令，本质都是传递存储数值。

端口编址方式有独立编址和内存映射编址。Intel现在就是独立编址。

### 独立编址

独立编址，使用专用的操作端口指令来操作端口，比如我们已经用过的inb和outb，好处就是它的地址编排不和内存地址冲突，内存是内存，外设是外设，但缺点也很明显，要用专门的命令，汇编要多学一种指令。

### 内存映射编址：

外设的端口还在外设中，但是底层硬件通过地址映射技术，将端口映射到了内存地址中，这样在地址总线数相同的状况下，内存映射技术势必挤压了内存的可用空间，但好处就是mov指令变得很通用，访问内存与外设都很普适。

## 显示

虽然操作系统比较简单，但也要能输出字符，既是操作系统的要求，也为了后面方便调试，如果没有显示系统，那我们连系统是否在运行都不知道，总不能接根飞线看CPU访问内存的地址吧？

计算机开机后，BIOS将计算机的显示初始化为了VGA文本模式或者MONO文本模式，与文本模式对立的自然是图形模式。文本模式只可以显示ASCII中的那些字符，图形模式可以操作逐个像素，这里我们使用文本模式就可以了，学完文本模式再说像素模式。

VGA和MONO模式都是IBM早期的产物，现在的图形显示技术早就不用了，但在BIOS中还是保留下来作为前向兼容在使用，越简单越可靠。

显卡，全称：图形显示适配器，他像CPU有内存一样，显卡也有显存，想要显示的字符都要存入到外设显卡的显存中相应的位置，显卡才能知道你想在屏幕的哪个地方显示上面字符，操作现存就是访问外设的寄存器（可以这么说），这个自然很繁琐，不过毕竟是IBM自家的产品，伟大的IBM工程师将这两个模式的显存通过地址映射技术将他们映射到了内存中的0XA0000~0XBFFFF中，这样就可以像往常访问内存一样读写显存了，记住，显存依然在显卡中，这是地址映射技术实现的。

这里有必要说一下开机后的内存分布了，你会对显示有更深入的了解。

### 1MB内存分布

|  |  |  |  |
| --- | --- | --- | --- |
| **起始** | **结束** | **大小** | **用途** |
| FFFF0 | FFFFF | 16B | BIOS入口地址，此地址也属于BIOS代码，同样属于顶部的640KB字节。只是为了强调其入口地址才单独贴出来。此处16字节的内容是跳转指令jmp f000：e05b |
| F0000 | FFFEF | 64KB-16B | 系统BIOS范围是F0000～FFFFF共640KB，为说明入口地址，将最上面的16字节从此处去掉了，所以此处终止地址是0XFFFEF |
| C8000 | EFFFF | 160KB | 映射硬件适配器的ROM或内存映射式I/O |
| C0000 | C7FFF | 32KB | 显示适配器BIOS |
| B8000 | BFFFF | 32KB | 用于文本模式显示适配器 |
| B0000 | B7FFF | 32KB | 用于黑白显示适配器 |
| A0000 | AFFFF | 64KB | 用于彩色显示适配器 |
| 9FC00 | 9FFFF | 1KB | EBDA（Extended BIOS Data Area）扩展BIOS数据区 |
| 7E00 | 9FBFF | 622080B约608KB | 可用区域 |
| 7C00 | 7DFF | 512B | MBR被BIOS加载到此处，共512字节 |
| 500 | 7BFF | 30464B约30KB | 可用区域 |
| 400 | 4FF | 256B | BIOS Data Area（BIOS数据区） |
| 000 | 3FF | 1KB | Interrupt Vector Table（中断向量表） |

实模式下，虽然CPU能访问1MB地址空间，但它其实只能访问640KB大的内存条，剩下的都是通过地址映射技术得到的空间。0x000~0x9FFFF是DRAM（俗称内存条）空间，这640KB的0X000~X7DFF都被占用了（开机启动后BIOS加载占用了），只有0x003FF~0X400之间有一截可用，再说高0XC000以上的地址，都是留给BIOS的，其中0XF0000~0XFFFFF是系统BIOS映射过来的，0XC0000~0XF0000是留给其他设备BIOS的，这样一来0XA0000~0XBFFFF就空出来了，这段就被显示地址占用了，因为显示确实太重要了，这段显示也是兼容了各个版本的显存，其中VGA和MONO也在其中。

### 显示字符

文本模式下显示一个字符，屏幕上显示一个字符在显存中占两个字节，高字节用来说明颜色（字体与背景），低字节存储要显示字符的ASCII，文本模式下屏幕共可以显示25行\*80个字符，总计2000个字符，共占用4000字节的显存，但是只把字符扔进显存中你会发现，显示器无动于衷，不是说显存中的数值发生改变，显卡立马更新屏幕，如果讲过早期的打屁股电视机，那种射电管的，它一个赫兹只能更新一个像素点，逐行逐列扫描更新全部。这里也一样，它一次只更新一个位置，更新的位置由光标指出，比如光标的值当前为0x03，那就更新屏幕上第一行第4个字符处的显示，即使你修改的是显存中的第5个字符，屏幕上第五个字符不会更新。

文本模式下的光标如何获取？这里以VGA模式来举例，MONO也一样，只不过它端口号不一样。由两个端口共同指出光标的位置，你自然会想到，一个端口表示列，一个端口表示行，但它不是这么设计的。

先说说显存中的那2000个字符的位置如何与屏幕对应的，显示器左上角的第一个位置对应VGA的0XB8000+0，第二行的第一个对应0xB8000+80\*1，第三行对应0xB8000+80\*2，一次类推，屏幕第n行第一个字符对应0XB8000+80\*(n-1)处的内存，那么光标两个端口（各是8位）读取到的值该如何解读？低8位端口的值和高8位端口值按位拼接成一个16位数值：crt，然后0XB8000+crt\*2，就是光标目前要在屏幕更新的字符，乘2是因为一个显示一个字符要占两个字节。可不是一个表示行，一个表示列，那样就成了0XB800+A\*80+B，（A表示行端口提供的值，B表示列端口提供的值）。

好了，具体来说说，显存映射在内存中是0XB8000-0XB8F9F，这里有一个东西，叫索引寄存器，出现原因是有的外设之前分配的端口数量在技术更新后不够用了，那就用索引端口来扩增，一个8位的索引寄存器就能扩充256个端口。比如要访问某个外设的17号端口（这个17号端口是某个外设内部编码的，不与17号端口冲突），那先把17号存入这个外设的索引寄存器“INDEX”中，CPU只认为这个INDEX寄存器这是该外设的端口，但外设内部不是这么看待这个寄存器的。随后通过out 17，INDEX指令，在INDEX中写入了17，这时这个外设是清楚写入17的功能是什么（你当然也知道，不然你不会这么干），随后通过其他端口读取外设返回的数值，或者外设执行某个任务，索引寄存器就像灰白你中的间接寻址一样。

如何获取光标？首先往VGA的索引寄存器0X3D4写入0xE这个值，在通过索引寄存器0X3D5读取读取到了光标低8位值，高8位通过0XF读写，也就是：先往VGA的索引寄存器0X3D4写入0xF这个值，通过索引寄存器0X3D5读取读取到了光标高8位值，好了。

现在光标的读写知道了，显存操作也知道了，就可以显示字符了，MONO模式是0XB0000是显存起始地址，0X3B4是它的索引端口。

再需要处理的就是：换行，退格和字符显示满25行，屏幕整体往上挪动一行，这些实现都不难，修改显存数值然后逐个改变光标让其更新即可。

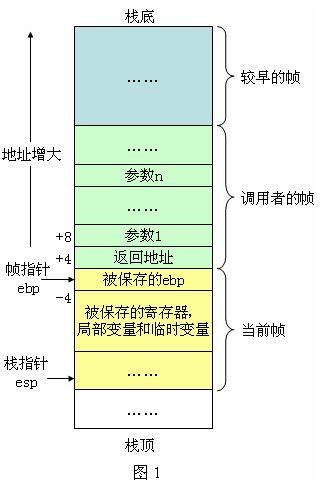
庆幸，保护模式的VGA显示和实模式下一样，万岁~！

## C函数栈变化

Intel的栈由两个寄存器控制，EBP与ESP。Intel的栈增长是地址递减的，也就是栈底在高地址处，栈顶在低地址，（栈顶是栈出入的一端，即ESP指向的内存位置），Intel的push指令是ESP减小，pop指令ESP增加，EBP和ESP的介绍之前讲过。

C函数调用时**栈帧（stack frame）**的变化，当调用某个函数时，它需要保存当前状态，并将实参也压入栈中，栈中预留返回值的空间，然后保存当前EBP，更新EBP为ESP（也就等价于开辟了一个新栈，只不过和old栈帧紧挨着），最后栈中分配被调用函数用到的局部变量（ESP=ESP+被调用函数局部变量所需空间值）。这个过程需要静下心慢慢看，并要彻底掌握。

栈此时内存分布如下（注意Intel的栈底栈顶！）：



调用者的浅绿色栈帧中，省略号表示的是调用者自己的局部变量，参数n到参数1表示的就是要传给被调用函数的实参，C语言的实参传递是从右往左的！！！你也可以在GCC中设置指令使其从左往右传递实参；返回地址就是函数的返回值存放处，比如sum=add(1,3)，这时sum的地址就会被存在这里；随后是保存当前EBP，此时EBP还指向调用者栈帧的首个地址（浅绿色的最上面），压入栈后，将ESP值赋给EBP，后面的事情就是被调用函数该处理的了。如果黄色函数内还调用函数，那么黄色和浅绿栈帧表达的意思都一样。

整个过程中，注意一个是调用函数的现场保护；一个是EBP的变化，会有一条movw %ESP,%EBP汇编指令修改EBP；ESP的值没有被强制修改过，ESP的值都是通过push和pop命令改变。

函数返回时是原路返回。

## struct成员在内存的排列

简单说一下，”struct test{int a,int b};”，这个strcut有两个成员，a和b，他们是局部变量时，a在靠近栈顶，b在靠近栈底那一遍，访问时，ESP+0就是b的值，ESP+2（Intel的栈是压栈ESP减，出栈ESP加）就是a的值，如果a前面还有一个成员c，那就是ESP+6，一个int占两个字节。

struct会涉及字节对齐问题，比如一个int，一个char，你以为占3个字节，但实际是4个字节，他会让每个变量的起始地址偶数对齐（地址取余2要等于0），这个概念要有，后面的内存管理各种地址对齐，这样做是为了加速读取，你也可以在gcc中关掉这个选项。

struct的成员变量声明时还可以这样:

struct test {

int a : 6;

int b : 2;

int c : 15;

char d :1;

}

这里的 “变量 : n”意思是，这个变量占用n个位，这时a可写入的最大值是2^6-1，b的最大值3。有特殊状况，struct还是爱整字节对齐，如果这里把b改为 b : 3，a已经占了6个位，一个字节8个位，8-6=2，紧跟其后的b却要3个位，一个字节容不下，只能空2个字节，然后给b分配3个位。这种状况，:0的意思则是跳过残缺的位，直接从下一个整字节位置开始排列。

## 源代码

这章主要扩增的源代码是driver/CRTDisplay.c和trap/trap.c；trap/entrytrap.S；trap/Ivector.S；

CRTDisplay.c负责显示，其余三个负责中断。显示的代码通过代码注释可以看懂，重点讲解中断代码。 中断最重要的IDT实在trap.c中生成，初始化IDT时需要处理函数的入口地址（即IDT描述符中32位的offset字段）和DPL，中断只允许内核来处理，故DPL是0，陷阱的DPL是3，设置目标代码段选择子为：内核代码段。现在IDT初始化完成，通过lidt汇编送入IDTR中。

IDT中所需的处理函数入口地址由Ivector.S提供，该文件源码非常有规律，注意\_\_vecotrs这个.globl标签即可，这里如果你理解了标签的本质（标签就是当前地址值），这里的代码就没有难点，这里每个vector标签后面的代码其实就是处理函数，但是为了集中管理中断响应，这里的每个函数都是压入中断码，然后通过jmp跳转到\_\_alltraps处，\_\_alltraps在entrytrap.S中，该汇编代码存储必要数据，准备进入真正的处理函数，这里压入许多寄存器值，然后call trap函数，trap我们为了方便写在了trap.c中，中断处理后又逐级返回。很复杂，但却环环相扣很严谨。entrytrap.S中的代码需要结合C函数调用时栈帧变化和结构体成员变量在内存中如何排列来看，这两个知识点都说过。

### CRTDisplay.c

#include <CRTDisplay.h>

//CRT’：阴极射线显示管

#define MONO\_BASE 0x3B4 //MONO的索引端口

#define MONO\_BUF 0xB0000 //MONO的显存映射起始地址

#define CGA\_BASE 0x3D4 //CGA的索引端口

#define CGA\_BUF 0xB8000 //CGA的显存映射起始地址

#define CRT\_ROWS 25 //屏幕显示25行

#define CRT\_COLS 80 //一行显示80个字符

#define CRT\_SIZE (CRT\_ROWS \* CRT\_COLS)

static uint16\_t \*crt\_buf;

static uint16\_t crt\_pos; //光标值

static uint16\_t addr\_6845;

void cga\_init(void) {

//VGA显示模式的缓冲地址

volatile uint16\_t \*DISPLAY\_BUF = (uint16\_t \*)(CGA\_BUF);

uint16\_t recordDISPLAY\_BUF = \*DISPLAY\_BUF;

//0xA55A是一个测试字符，检测能否写入DISPLAY\_BUF，如二次读取一致，则当前VGA模式，否则为MONO模式，并修改相关IO端口

\*DISPLAY\_BUF = (uint16\_t) 0xA55A;

if (\*DISPLAY\_BUF != 0xA55A) {

DISPLAY\_BUF = (uint16\_t\*)(MONO\_BUF );

addr\_6845 = MONO\_BASE;

} else {

\*DISPLAY\_BUF = recordDISPLAY\_BUF;

addr\_6845 = CGA\_BASE;

}

//获取光标位置 通过索引寄存器及 0xe 0xf端口来读写

uint32\_t pos;

outb(addr\_6845, 14);

pos = inb(addr\_6845 + 1) << 8;

outb(addr\_6845, 15);

pos |= inb(addr\_6845 + 1);

crt\_buf = (uint16\_t\*) DISPLAY\_BUF;

crt\_pos = pos;

}

/\* cga\_putc - print character to console \*/

void cga\_putc(int c) {

//字符是一个字节，字符显示模式下，一个字符要两个字节，一个字节存储字体颜色及背景颜色，一个字节存储ascii

// 设置字体为白色

if (!(c & ~0xFF)) {

c |= 0x0700;

}

switch (c & 0xff) {

case '\b': //显示一个退格

if (crt\_pos > 0) {

crt\_pos --; //直接crt\_pos自减即可，理解有困惑再仔细学习

crt\_buf[crt\_pos] = (c & ~0xff) | ' ';

}

break;

case '\n': //屏幕换行

crt\_pos += CRT\_COLS;

case '\r': //屏幕回车

crt\_pos -= (crt\_pos % CRT\_COLS);

break;

default: //显示一个字符

crt\_buf[crt\_pos ++] = c; // write the character

break;

}

if(crt\_pos>=80\*25)

{ //超过屏幕能显示字符的上限，显示整体往上挪一行

int \* crt\_mov=crt\_buf+CRT\_COLS;

//显存字符挪动

for (int i=0;i<(CRT\_ROWS-1)\*CRT\_COLS;i++)

{

crt\_buf[i]=crt\_mov[i];

//还差更新显示，自己完成

}

crt\_pos-=CRT\_COLS;

}

//通过索引寄存器刷新屏幕，更新当前光标处的字符

outb(addr\_6845, 14);

outb(addr\_6845 + 1, crt\_pos >> 8);

outb(addr\_6845, 15);

outb(addr\_6845 + 1, crt\_pos);

}

### trap.c

中断的流程比较复杂，先说一下各个源文件是如何组织的。

#include <trap.h>

#include <defs.h>

#include <x86.h>

#include <CRTDisplay.h>

//IDT表，共256个中断

struct trapgate IDT[256]={0};

//IDTR，存储IDT其实地址IDT界限

static struct pseudodesc idt\_pd = {

sizeof(IDT)-1, (unsigned long int)IDT

};

//！！！32位下，一个指针变量的类型就是unsigned long int

//\_\_vectors是从Ivector.S中引入的

extern unsigned long int \_\_vectors[];

//中断初始化函数

void idtinit(void) {

//for循环中的每一次都会在IDT中初始化一个中断门描述符，它的处理入口地址来自Ivector.S。

//DPL都设置为了0级，只有内核才可以调用

//P设置为1，表示在内存中，不然还出触发一个异常

//目标代码段选择子设为内核代码段，因为要进入内核处理

//Type字段为0XE，表示是中断，0XF表示是陷阱

for (int i=0;i<sizeof(IDT)/sizeof (struct trapgate);i++)

{

IDT[i].CodeSegmentOffset\_0\_15=\_\_vectors[i]&0xffff;

IDT[i].CodeSegmentOffset\_16\_31=(uint32\_t)(\_\_vectors[i]) >> 16;

IDT[i].DPL=0;

IDT[i].TargetCodeSegmentSelector=GD\_KTEXT;

IDT[i].Type=0xE; //中断

IDT[i].P=1;

IDT[i].Reserved=0;

}

//按照惯例，0x80是系统调用，这里我们也保留下来，系统调用供用户程序调用，故DPL为3，但目标段选择子依然是内核

IDT[0x80].CodeSegmentOffset\_0\_15=\_\_vectors[0x80]&0xffff;

IDT[0x80].CodeSegmentOffset\_16\_31=(uint32\_t)(\_\_vectors[0x80]) >> 16;

IDT[0x80].DPL=3; //DPL为3，用户程序可以使用

IDT[0x80].TargetCodeSegmentSelector=GD\_KTEXT;

IDT[0x80].Type=0xF; //陷阱

IDT[0x80].P=1;

IDT[0x80].Reserved=0;

//asm内联汇编，加载IDT表到IDTR中初始化到此完成

asm volatile ("lidt (%0)" : : "r" (&idt\_pd) : "memory");

}

//这里是真正的处理函数

void trap(struct trapframe \*tf)

{

if (tf->tf\_trapno==0x32)

{

cga\_putc('t');

cga\_putc('h');

cga\_putc('e');

cga\_putc(' ');

cga\_putc('0');

cga\_putc('x');

cga\_putc('3');

cga\_putc('2');

cga\_putc('i');

cga\_putc('n');

cga\_putc('t');

cga\_putc('e');

cga\_putc('r');

cga\_putc('!');

}

}

### Ivector.S

Ivector的代码都是重复的，这里只选部分讲解

.text

#globl指明是全局变量，这里是引入这个变量而不是声明

.globl \_\_alltraps

.globl vector0

#vector0是个标签，是个指针

vector0:

#压入了中断号，随后跳转进入entrytrap.S的\_\_alltraps处

pushl $0

pushl $0

jmp \_\_alltraps

.globl vector1

vector1:

pushl $0

pushl $1

jmp \_\_alltraps

.globl vector2

…………

# vector table

.data

.globl \_\_vectors

\_\_vectors:

#vector0来自上面，在这里是地址的含义，long表明这个指针长度为32位

.long vector0

.long vector1

……………

### entrytrap.S

#include <defsForASM.h>

# vectors.S sends all traps here.

.text

#此处是生成全局变量\_\_alltraps

.globl \_\_alltraps

\_\_alltraps:

# push registers to build a trap frame

# therefore make the stack look like a struct trapframe

#保护现场，压入各寄存器

pushl %ds

pushl %es

pushl %fs

pushl %gs

#pushal是以4字节长度压入所有寄存器，详细上网搜索

pushal

# load GD\_KDATA into %ds and %es to set up data segments for kernel

#调整数据段为内核数据段

movl $GD\_KDATA, %eax

movw %ax, %ds

movw %ax, %es

# push %esp to pass a pointer to the trapframe as an argument to trap()

#压入ESP是为了符合C函数的规则，前面讲过，现在这个函数可以传参数了，而这个参数其实就是struct trapframe类型的指针，需要掌握struct成员变量在内存中的排列。这里是压入栈中的各个数据的字段格式正好和struct trapframe的成员变量分布是一致的，所以就等于为trap函数传入了一个struct trapframe \* tf的实参，很难理解，不懂多看看前面的知识，c函数栈帧的变化和struct成员变量的排列

pushl %esp

# call trap(tf), where tf=%esp

call trap

# pop the pushed stack pointer

popl %esp

# return falls through to trapret...

.globl \_\_trapret

\_\_trapret:

# restore registers from stack

popal

# restore %ds, %es, %fs and %gs

popl %gs

popl %fs

popl %es

popl %ds

# get rid of the trap number and error code

#之前还压入了中断号和错误码，每个4字节，这两个CPU不会自己弹出，需要手动解决，他们都是当初压入栈中，只需esp+8即可（就是弹出栈了）

addl $0x8, %esp

iret

.globl forkrets

forkrets:

# set stack to this new process's trapframe

movl 4(%esp), %esp

jmp \_\_trapret

### 测试中断

最后又回到了trap.c的trap函数中了，现在测试是否可以执行，在init.c中使用int $0x32来测试，同时在trap.c中写好处理函数看看能否正常处理。

init.c中的中断触发代码：

asm volatile ("int $0x32");

trap.c中的处理函数，根据传入的tf中的中断码区分不同的中断，这里自然是0x32：

trap.c中的trap函数

void trap(struct trapframe \*tf)

{

if (tf->tf\_trapno==0x32)

{

cga\_putc('t');

cga\_putc('h');

cga\_putc('e');

cga\_putc(' ');

cga\_putc('0');

cga\_putc('x');

cga\_putc('3');

cga\_putc('2');

cga\_putc('i');

cga\_putc('n');

cga\_putc('t');

cga\_putc('e');

cga\_putc('r');

cga\_putc('!');

}

}

如果中断正确的话，屏幕上会显示”the 0x32inter!”，make后make qemu，看看结果：

---我已经测试过了，虚拟机截图又不方便，结果正确，自己执行一遍吧。

现在你掌握了中断的全部过程，中断的处理函数你也会了，其他的中断处理都是复制一遍罢了，陷阱也没啥难的，唯一需要你自己学的是屏蔽中断响应，和其他外设管理读写，键盘的操作我就不再展示了。

最难的一章结束了，一致代码段也没有代码，其余几个门自己需要可以写，任务门基本不使用，TSS后面再使用。下一章更难。

第三章 内存管理

内存管理是一个重要的部分，CPU直接与内存交互存储数据，这章的内容已经极少用到汇编代码，基本以C语言为主，有许多计算需要掌握，其中不乏一些很难理解的内容。

## 常见内存模型

### 实模式的内存

实模式下的内存我们已经见过了，再之前的汇编课程中也都实践过，基本格式是：段寄存器左移4位+IP寄存器值得到实际内存值，它无法做到代码载入内存后再进行内存映射，只能通过CS，DS，SS等动态调整来区分各个进程的地址空间。我们代码中快速的跳过了实模式的内存，所以不必再关注。

### 段式内存

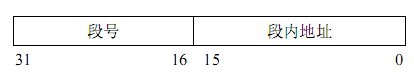
这里的段式存储管理是理论概念，Intel在进入保护模式不开启页式内存就是段式内存管理。

在段式存储管理 中，将程序的地址空间划分为若干个段(segment)，这样每个进程有一个二维的地址空间。在段式存储管理系统中，操作系统为每个段分配一个连续的分区，而进程中的各个段可以不连续地存放在内存的不同分区中。程序加载时，操作系统为 所有段分配其所需内存，这些段不必连续，物理内存的管理采用动态分区的管理方法。在为某个段分配物理内存时，可以采用首先适配法、下次适配法、最佳适配法 等方法。在回收某个段所占用的空间时，要注意将收回的空间与其相邻的空间合并。段式存储管理也需要硬件支持，实现逻辑地址到物理地址的映射。程序通过分段 划分为多个模块，如代码段、数据段、共享段。这样做的优点是：可以分别编写和编译源程序的一个文件，并且可以针对不同类型的段采取不同的保护，也可以按段为单位来进行共享。总的来说，段式存储管理的优点是：没有内碎片，外碎片可以通过内存紧缩来消除；便于实现内存共享。缺点式不够灵活，某个段特别大的时候，无法一次装入，段式内存就无能为力了，但是页式内存则可以躲避这个问题。我们代码中快速跳过了段式内存，进入了平坦内存模型。

为了实现段式管理，操作系统需要如下的数据结构来实现进程的地址空间到物理内存空间的映射，并跟踪物理内存的使用情况，以便在装入新的段的时候，合理地分配内存空间。

进程段表：描述组成进程地址空间的各段，可以是指向系统段表中表项的索引。每段有段基址(baseaddress)，即段内地址。

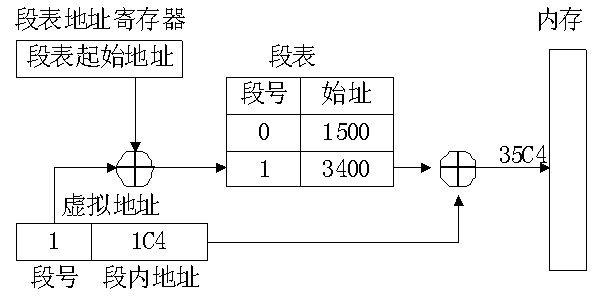
在系统中为每个进程建立一张段映射表，如图：



系统段表：系统所有占用段（已经分配的段）。

空闲段表：内存中所有空闲段，可以结合到系统段表中。

段式内存寻址过程：

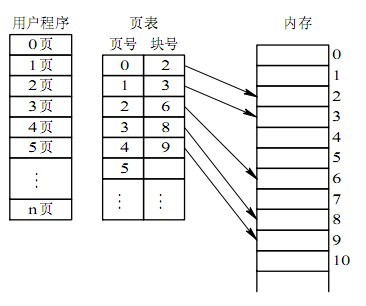


### 页式内存

 在页式系统中进程建立时，操作系统为进程中所有的页分配页框。当进程撤销时收回所有分配给它的页框。在程序的运行期间，如果允许进程动态地申请空间，操作系统还要为进程申请的空间分配物理页框。操作系统为了完成这些功能，必须记录系统内存中实际的页框使用情况。操作系统还要在进程切换时，正确地切换两个不同的进程地址空间到物理内存空间的映射。这就要求操作系统要记录每个进程页表的相关信息。为了完成上述的功能，—个页式系统中，一般要采用如下的数据结构。

进程页表：完成逻辑页号(本进程的地址空间)到物理页面号(实际内存空间，也叫块号)的映射。

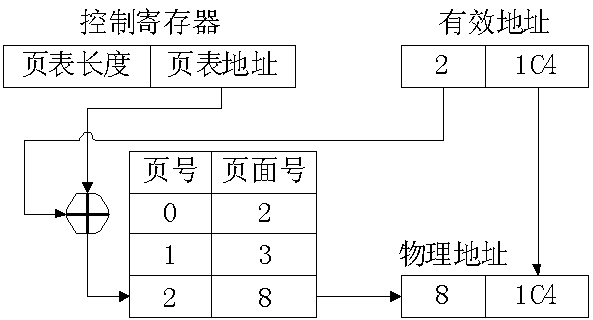
每个进程有一个页表，描述该进程占用的物理页面及逻辑排列顺序，如图：



在页式系统中，指令所给出的地址分为两部分：逻辑页号和页内地址。

原理：CPU中的内存管理单元(MMU)按逻辑页号通过查进程页表得到物理页框号，将物理页框号与页内地址相加形成物理地址。

逻辑页号，页内偏移地址－>查进程页表，得物理页号－>物理地址：



页式管理的地址变换

上述过程通常由处理器的硬件直接完成，不需要软件参与。通常，操作系统只需在进程切换时，把进程页表的首地址装入处理器特定的寄存器中即可。一般来说，页表存储在主存之中。这样处理器每访问一个在内存中的操作数，就要访问两次内存：

第一次用来查找页表将操作数的 逻辑地址变换为物理地址；

第二次完成真正的读写操作。

这样做时间上耗费严重。为缩短查找时间，可以将页表从内存装入CPU内部的关联存储器(例如，快表) 中，实现按内容查找。此时的地址变换过程是：在CPU给出有效地址后，由地址变换机构自动将页号送人快表，并将此页号与快表中的所有页号进行比较，而且这 种比较是同时进行的。若其中有与此相匹配的页号，表示要访问的页的页表项在快表中。于是可直接读出该页所对应的物理页号，这样就无需访问内存中的页表。由于关联存储器的访问速度比内存的访问速度快得多。

页式内存最大的好处在于，它能同时载入多个文件，尽管这些文件的内存总和已经远远大于内存条大小，但CPU依然可以正常工作。原因在于，程序的执行在一段时间内都是集中在一个地方，其余地方的代码暂且不读取，那就不载入，用时通过缺页警告载入，于是就有了这样的画面。

### 段页式内存

分页系统能有效地提高内存的利用率，而分段系统能反映程序的逻辑结构，将分页与分段结合起来，就形成了段页式管理方式。在段页式管理系统中，用户程序的逻辑地址空间首先被划分成若干个逻辑分段，每段都有自己的段号，然后再将每段分成若干个大小相等的页。

逻辑地址映射成物理地址的过程：段表寄存器中存放段表长度和段表起始地址，指令给出的地址包括段号、段内页号和页内地址，用段号去索引段表，段表中存放段号、页表长度、页表起始地址，索引之前，先比较段号与段表长度，若段号大于等于段表长度，则发生越界中断，否则，将段表起始地址与段号相加可以得到该段在段表中的位置，进而得到页表起始地址，将页表起始地址与段内页号相加可以得到对应的物理块地址，将物理块地址和页内地址相加即可得到物理地址。

优点：程序是以段为单位分割的，每个段内是连续的，但段间可以不连续；没有外碎片，能减少存储空间的浪费。

缺点：有内部碎片；由于管理软件的增加，复杂性和开销也随之增加。

## 逻辑地址

是在有地址变换功能的计算机中,访内指令给出的地址 (操作数) 叫逻辑地址,也叫相对地址，也就是是机器语言指令中，用来指定一个操作数或是一条指令的地址。要经过寻址方式的计算或变换才得到内存储器中的实际有效地址即物理地址。一个逻辑地址由两部份组成，段标识符: 段内偏移量。段标识符是由一个16位长的字段组成，称为段选择符。

## 物理地址

CPU地址总线传来的地址，由硬件电路控制（现在这些硬件是可编程的了）其具体含义。物理地址中很大一部分是留给内存条中的内存的，但也常被映射到其他存储器上（如显存、BIOS等）。在没有使用虚拟存储器的机器上，虚拟地址被直接送到内存总线上，使具有相同地址的物理存储器被读写；而在使用了虚拟存储器的情况下，虚拟地址不是被直接送到内存地址总线上，而是送到存储器管理单元MMU，把虚拟地址映射为物理地址。

## 线性地址

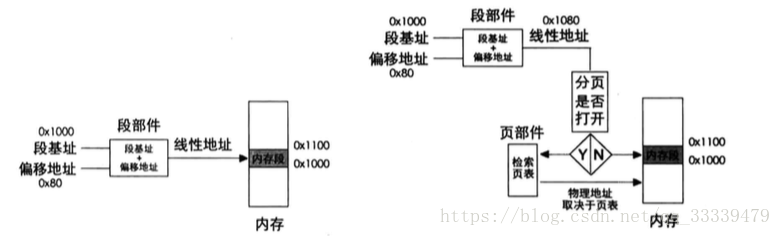
也叫**虚拟地址**(virtual address)是逻辑地址到物理地址变换之间的中间层。在分段部件中逻辑地址是段中的偏移地址，然后加上基地址就是线性地址。是一个32位无符号整数，可以用来表示高达4GB的地址，也就是，高达4294967296个内存单元。线性地址通常用十六进制数字表示，值得范围从0x00000000到0xfffffff）程序代码会产生逻辑地址，通过逻辑地址变换就可以生成一个线性地址。如果启用了分页机制，那么线性地址可以再经过变换以产生一个物理地址。如果没有启用分页机制，那么线性地址直接就是物理地址。

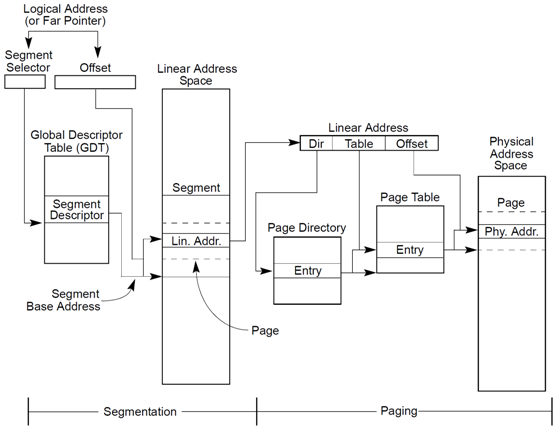
## 地址小总结

GCC生成的代码地址就是逻辑地址，比如声明一个全局变量varx，那么这个全局变量的地址是固定的嘛？在objet中是一个相对代码起始处的偏移量。在最终的可执行文件不是，取决于链接器设定的起始地址，以代码为例，设定起始地址0xC0000000，那么varx地址必大于0XC0000000，它是依赖于代码的。我们在代码中能操作的都是逻辑地址，比如int \* test = 0xfff，当往(\*test)中赋值时，不一定写入的就是0xfff这个物理地址，是经过线性转换和映射后的某个物理地址。

在实模式下，逻辑地址加上段起始值，就得到了线性地址，这也是地址映射，之所以叫线性地址，大概是因为，线性地址=逻辑地址\*1+段起始值，这个地址映射函数不就是线性函数嘛？保护模式下，通过权限检测后相加该段起始值，得到线性地址，就是物理地址，实模式下没有段描述符，直接用段寄存表述段起始的意义。

保护模式下线性地址值等于物理地址值为什么不直接称呼其为物理地址？因为还有页寻址，通过段起始值相加后是线性地址，用这个线性地址通过页表索引，得到物理地址。注意开启页寻址后，一个逻辑地址如何变换得到物理地址的过程。





在代码中，需要注意区分这三个概念，代码中出现的变量，通过取地址符“&”得到的都是逻辑地址，GDTR和IDTR中存储的都是线性地址，CR3存储的则是物理地址，页表中存储的也是物理地址。在启动和内核初始化内存阶段中，需要时刻明确这三者概念，并要时刻通过逻辑地址就知道其对应的物理地址。

## 平坦内存

平坦内存模型才是我们要掌握的，你可以把它想象成段页式，毕竟它用到了段寄存器（CS，DS，SS），也开启了页式内存管理，不过它的段起始值有些不同，所有段起始地址都是0，并且段长都是最大内存值4GB（32位下），也不能说它丢失了段的功能，它只是给段增加了表达特权级的能力（GDT，段选择子…），削弱了段的基址地位：让所有段的起始地址都为0，但它段的能力依然在(这次代码中就会展现出奇淫技巧来利用它的段起始地址功能)，只是各个操作系统开发商发现置为0更精简。剥离掉编译器生成的段的概念（.BSS;.Text……），你把段式内存中的每个段变得特别小（比如4KB），段式内存就已经和页式内存没有什么本质区别了。先介绍几个概念

#### page frame

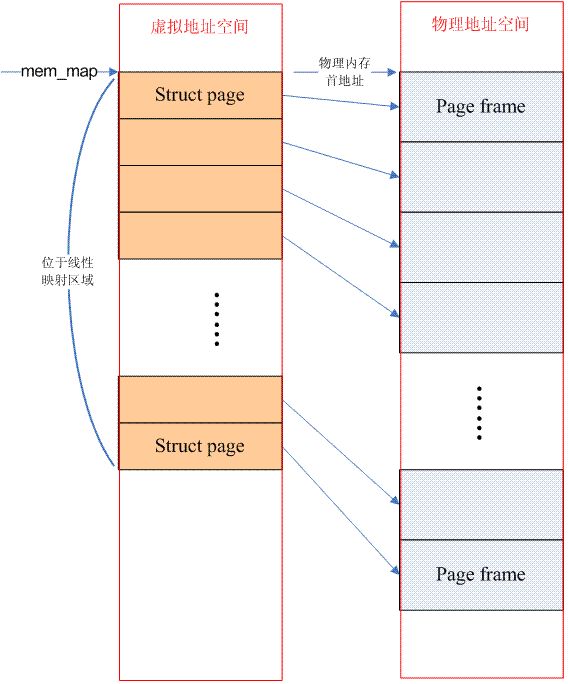
操作系统最重要的作用之一就是管理计算机系统中的各种资源，做为最重要的资源：内存，我们必须管理起来。在linux操作系统中，物理内存是按照page size（一个大小4KB带）来管理的，具体page size是多少是和硬件以及linux系统配置相关的，4kb大小的页是最经典的设定。因此，对于物理内存，我们将其分成一个个按page size排列的page，每一个物理内存中的page size的内存区域我们称之page frame（就像之前C函数栈帧概念一样）。我们针对每一个物理的page frame建立一个struct page的数据结构来跟踪每一个物理页面的使用情况：是用于内核的正文段？还是用于进程的页表？是用于各种file cache还是处于free状态

#### PFN

是page frame number的缩写，所谓page frame，就是针对物理内存而言的，把物理内存分成一个个的page size的区域，并且给每一个page 编号，这个号码就是PFN。假设物理内存从0地址开始，那么PFN等于0的那个页帧就是0地址（物理地址）开始的那个page。假设物理内存从x地址开始，那么第一个页帧号码就是（x>>12），12是因为低12位不用于寻址，而起标记作用。

#### FLAT memory model

如果从系统中任意一个processor的角度来看，当它访问物理内存的时候，物理地址空间是一个连续的，没有空洞的地址空间，那么这种计算机系统的内存模型就是Flat memory。这种内存模型下，物理内存的管理比较简单，每一个物理页帧都会有一个page数据结构来抽象，因此系统中存在一个struct page的数组（mem\_map），每一个数组条目指向一个实际的物理页帧（page frame）。在flat memory的情况下，PFN（page frame number）和mem\_map数组index的关系是线性的（有一个固定偏移，如果内存对应的物理地址等于0，那么PFN就是数组index）。因此从PFN到对应的page数据结构是非常容易的，反之亦然。

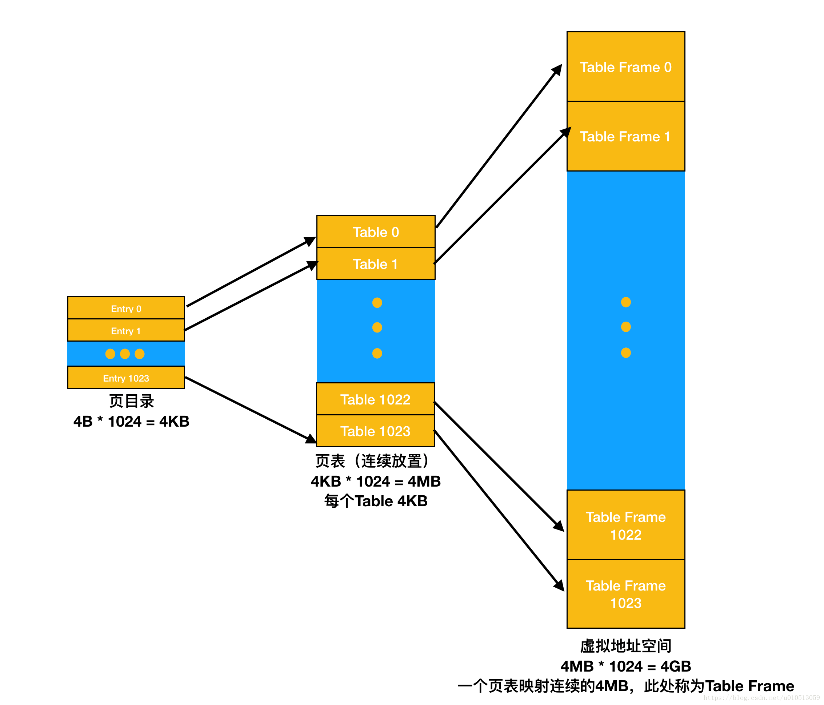


#### MMU

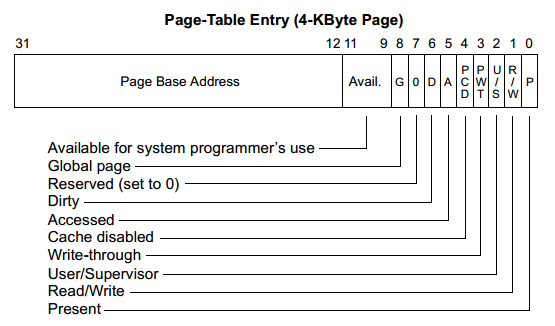
MMU是Memory Management Unit的缩写，中文名是内存管理单元，有时称作分页内存管理单元（英语：paged memory management unit，缩写为PMMU）。它是一种负责处理中央处理器（CPU）的内存访问请求的计算机硬件。它的功能包括虚拟地址到物理地址的转换（即虚拟内存管理）、内存保护、中央处理器高速缓存的控制

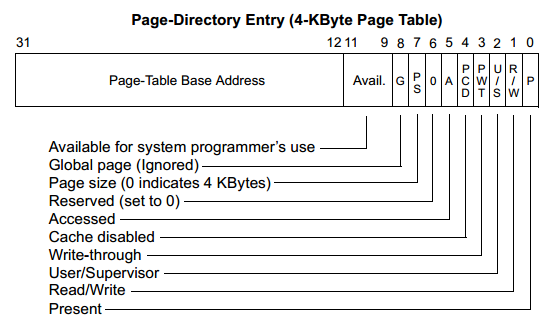
好啦，上面就是我们用到的内存模型的概念介绍，本质就是页式内存管理，一些概念有些晦涩，下面讲解一下。

给每一个进程都构造一个页式地址映射，访问空间范围都是0~4GB，这样各个进程的地址空间都不会冲突。段只用于权限检测。实际会有两级页表，总的目录叫页目录，页目录中的每项记录索引一个页表。



页式内存管理：一个内存页大小设为4KB，将内存按页划，4GB空间将会划分出1M（2^20）个页。这些页是实际的页，目前存储的东西有我们的内核，低1MB处的内容，同时为了管理这些页，一些内存空间存储每一个页的相关信息：内存是否内存驻留，是否为脏块，被分配与否等，每个页的记录信息占4字节，管理用的空间共占用4MB。页表是4KB对齐的（即每个页帧的起始地址都是可以被4KB整除的，第一个页帧的起始地址自然为0X0，不可以为0X1），只需记录高20位，低12位用于控制字段。下图为页表与页目录项字段：





【P】：存在位。为1表示页表或者页位于内存中。否则，表示不在内存中，必须先予以创建或者从磁盘调入内存后方可使用。

【R/W】：读写标志。为1表示页面可以被读写，为0表示只读。当处理器运行在0、1、2特权级时，此位不起作用。页目录中的这个位对其所映射的所有页面起作用。

【U/S】：用户/超级用户标志。为1时，允许所有特权级别的程序访问；为0时，仅允许特权级为0、1、2的程序访问。页目录中的这个位对其所映射的所有页面起作用。

【PWT】：Page级的Write-Through标志位。为1时使用Write-Through的Cache类型；为0时使用Write-Back的Cache类型。当CR0.CD=1时（Cache被Disable掉），此标志被忽略。对于我们的实验，此位清零。

【PCD】：Page级的Cache Disable标志位。为1时，物理页面是不能被Cache的；为0时允许Cache。当CR0.CD=1时，此标志被忽略。

【A】：访问位。该位由处理器固件设置，用来指示此表项所指向的页是否已被访问（读或写），一旦置位，处理器从不清这个标志位。这个位可以被操作系统用来监视页的使用频率。

【D】：脏位。该位由处理器固件设置，用来指示此表项所指向的页是否写过数据。

【PS】：Page Size位。为0时，页的大小是4KB；为1时，页的大小是4MB（for normal 32-bit addressing ）或者2MB（if extended physical addressing is enabled).

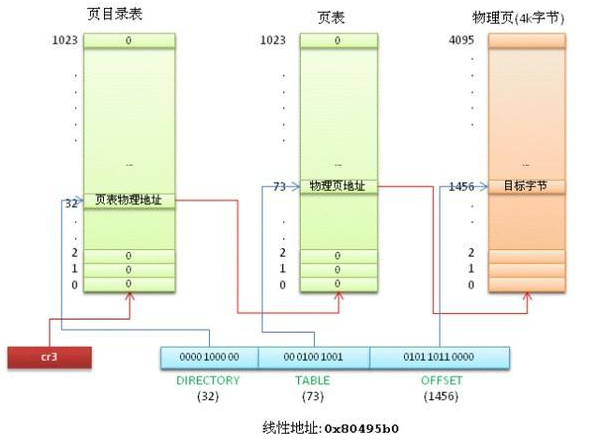
【G】：全局位。如果页是全局的，那么它将在高速缓存中一直保存。当CR4.PGE=1时，可以设置此位为1，指示Page是全局Page，在CR3被更新时，TLB内的全局Page不会被刷新。

【AVL】：被处理器忽略，软件可以使用。

最低位的P，表示页面表或者相应的页面是否在内存中，如果P为0，则试图存取该页面时，CPU会触发缺页异常“Page Fault”，这个异常时你要处理的，自己查找这个异常号，后面要在trap.c中完成。

32位下，CPU每次访问的地址总线是32个根，那么每个地址就是32位。每个页大小为4KB，页内寻址就需要12位（4KB=2^12 byte），32位的地址剩下了20位，记录一个页帧需要32位（上图的字段格式），共有1M（2^20）个页，则需要4 byte \* 2^20=4MB的空间（寻址4GB空间，需要32位，故为4\*2^20），但并非所有程序都会一次占用4GB内存，有的程序就只有1MB，也需要4MB管理内存的空间，这样太耗费内存了，将其拆分为二级页表，逻辑地址的高10位用于在页目录表中做偏移量寻址，2^10\*4=4KB页刚好是一个页大小，逻辑地址的中10位在二级页表中寻址，这样会多浪费4KB的空间，但却更加灵活，在实际使用中更节省内存，向刚才1MB的内存，只需要12KB就可以解决。下图展示了两级页式管理的寻址过程，在页目录表和页表中，存储的都是物理地址，页目录表的物理地址会被存储到CR3，在开启页模式后，CPU的MMU根据CR3中的数值来进行地址定位。实际寻址时，还会先使用CS或DS在GDT中获取段的相关信息，所有的段基址都是0，段长4GB。

下图是两级页式内存寻址过程：



根据示意图来理解，二级页表没有太多难度。注意：Intel页模式下（两级页表），页表和页目录中存储的都是物理地址，不是逻辑地址。若存储的是逻辑地址，那么变成了一个逻辑地址通过页表得到一个逻辑地址，永远得不到物理地址，进入了死循环。

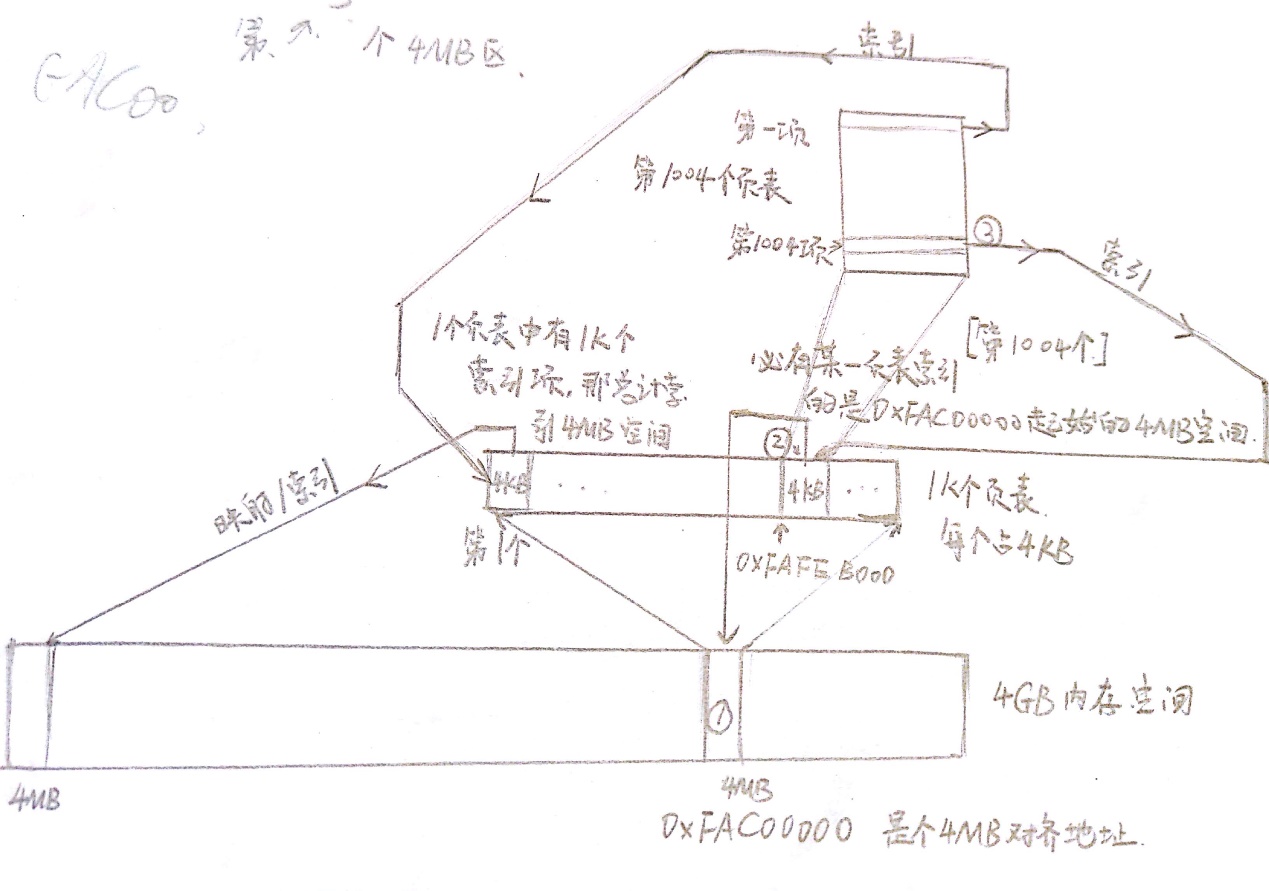
### 页自映射技术

前面介绍完两级页表技术，Intel的MMU要求必须是二级页表的，不可以只有一个页表，即便如此，你也可以只用一个页表就能替换二级页表，这就是页自映射技术，它的推到过程极其复杂，按部就班论述。不过我先使用一点小聪明证明页自映射是对的，然后再按部就班推导。

二级页表中，内存为4GB时，总共有多少个页表？答案是4GB/4KB=1K，这1K就是上面说的页表，为了索引这1K个页表，还需一个大小4KB的页目录来索引，那总共就是1K+1=1025个（计算机的1K是2^10=1024）。不过4GB内存总共只需要1K个页表就可以全部索引，但会丢失灵活性，看到了没？这等价于说二级页表中有一个页表是冗余记录，那么1K个页表能实现处二级页表的功能，不过要求一次要连续划分一个4MB空间来记录这1K个页表。换言之，页子映射技术适用于占用内存特别大的进程，这样就可以节省4KB内存。

以记录1K页表起始地址为0XFAC00000为例，这个地址是4MB对齐的，1个页表中记录1K项索引，一项索引索引4KB大小，那么一个页表总共索引4MB，所以地址必须要4MB对齐。1K个页表的起始地址是4MB对齐，那么现在必有某一个页表映射的是0XFAC00000起始的4MB空间，现在已经自己映射自己了，只不过具体的地址值还没有找到，不要着急。

0XFAC00000按4MB划分，它前面有：0XFAC00000/4\*1024\*1024=1003个4MB块，那0XFAC00000处开始的4MB自然是被1K（1024）个页表中的第1004个页表索引，那这个页表的起始物理地址是多少？一个页表占4KB空间，即：



0XFAC00000+1003\*4KB=0XFAC00000+0X3EB000=0xFAFE B000

1处的存储了1K个页表，2处是1里面的一部分，2的起始地址已经算出来了：0xFAFE B000，而且2处的页表索引的空间就是1处的4MB空间，现在看看2处页表的内部，即3处所画的，这个页表的第一项索引的就是1K页表中的第一个4KB页表空间，第1004项索引的是2处的4KB空间，最后一项索引的是1K页表中最后一个4KB空间，换言之，2处的这个页表正好索引了全部的页表，那它就是页目录表，也是页表，即，2处的这个页表能自己映射自己，而它的起始地址就是页目录表的地址，推理完毕。

通过上面的这个图能理解，页自映射地址非常依赖于空间顺序排列，不可拆分，一次就需要开辟4MB空间，所以能使用这个技术的也就是内核，或者某个特别大的进程。

现在你应该可以计算别的页自映射技术地址了，比如0XFAC00000-4MB起存储1K个页表，那么自映射技术的地址是多少？

### 页表字段



P字段：present标志，vP=1表示有效；P=0表示无效。在页转换过程中，如果说涉及的页目录或页表的表项无效，则会导致一个异常。如果P=0，那么除表示该表项无效外，其余位可供程序自由使用（余下31个位），操作系统可以使用这些位来保存已存储在磁盘上的页面的序号，这对页置换很有用，P=0时的异常处理代码自己完成。

R/W--位1是读/写（Read/Write）标志。如果等于1，表示页面可以被读、写或执行。如果为0，表示页面只读或可执行。当处理器运行在超级用户特权级（级别0、1或2）时，则R/W位不起作用。页目录项中的R/W位对其所映射的所有页面起作用。

U/S--位2是用户/超级用户（User/Supervisor）标志。如果为1，那么运行在任何特权级上的程序都可以访问该页面。如果为0，那么页面只能被运行在超级用户特权级（0、1或2）上的程序访问。页目录项中的U/S位对其所映射的所有页面起作用。

A--位5是已访问（Accessed）标志。当处理器访问页表项映射的页面时，页表表项的这个标志就会被置为1。当处理器访问页目录表项映射的任何页面时，页目录表项的这个标志就会被置为1。处理器只负责设置该标志，操作系统可通过定期地复位该标志来统计页面的使用情况。

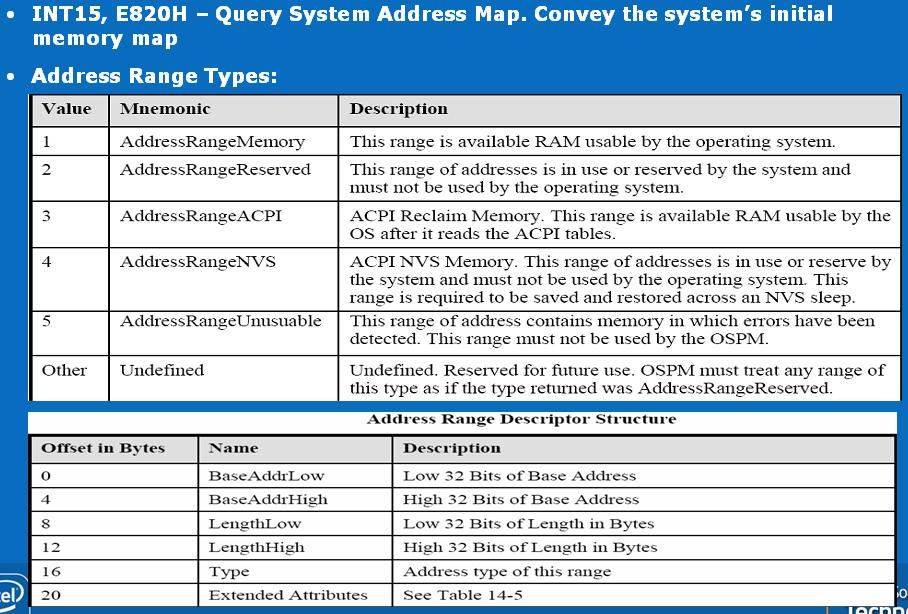
D--位6是页面已被修改（Dirty）标志。当处理器对一个页面执行写操作时，就会设置对应页表表项的D标志。处理器并不会修改页目录项中的D标志。

AVL--该字段保留专供程序使用。处理器不会修改这几位，以后的升级处理器也不会。

## 内存探测

如何获知这个计算机的内存有多大？可以自己写，这里利用实模式下的BIOS自带的程序，通过int 0x15的三个子功能：e820h、e801h、88h，获得内存信息。我们使用e820h，它的功能最强大，得到的信息最全面。系统内存分成很多内存块，这些内存块加起来构成系统的总内存，他们类型不同，有的操作系统系统可以使用，有的不可以使用。每次调用总段智慧得到一块内存的描述，需要多次调用才能得到所有内存块的描述。

e820将物理内存探测的结果以**地址范围描述符**的格式放在内存中。地址范围描述符共计20字节。



| **字节范围** | **描述** |
| --- | --- |
| 0~7 | 内存块基地址 |
| 8~15 | 这块内存的大小 |
| 16~20 | 这块内存的类型 |

在调用0x15中断前，需要将参数放置在寄存器中：

EAX：子功能号，这些输入0xE820

EDX：534D4150h(ascii字符”SMAP”)，签名，约定填”SMAP”

EBX：每调用一次int $0x15，ebx会加1。当ebx为0表示所有内存块检测完毕。

ECX：存放地址范围描述符的内存大小，至少要设置为20。

ES:DI：告诉BIOS要把地址描述符写到这个地址。

一次执行后的返回值：

CF标志位：若中断执行失败，则置为1。

EAX：值是534D4150h(“SMAP”)

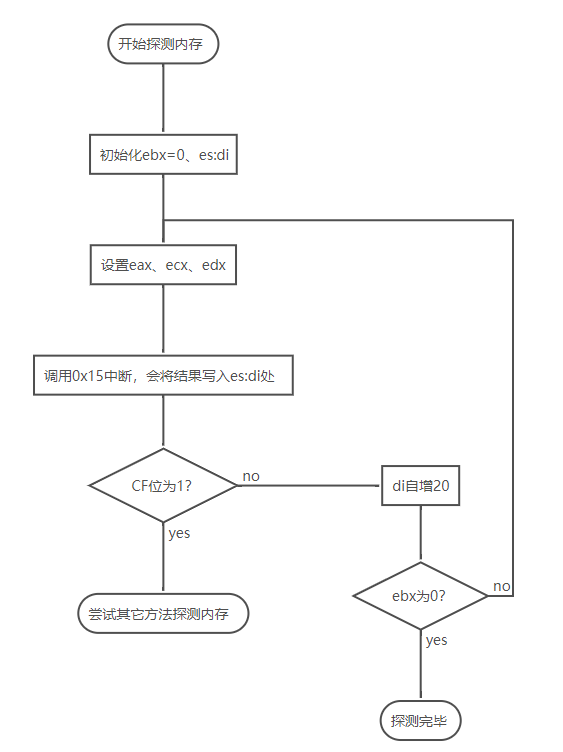
ES:DI：中断不改变该值，值与参数传入的值一致

EBX：下一个中断描述符的计数值（见后面的案例）

ECX：返回BIOS写到cs:di处的地址描述符的大小（应该就是20吧？）

AH：若发生错误，表示错误码

代码流程如下：



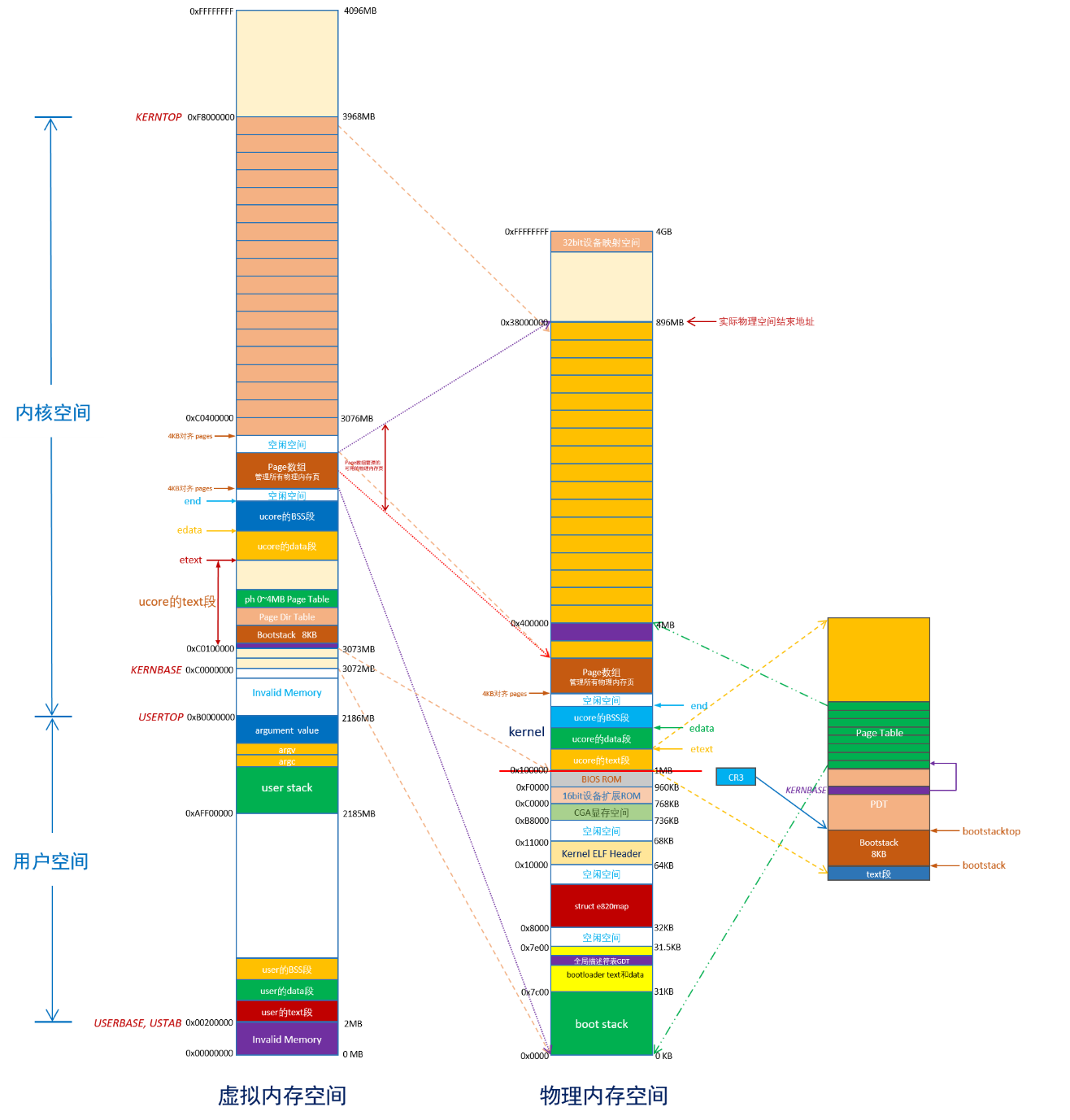
## 管理内存

现在我们知道了如何探测内存，页式寻址内存，我们还差管理这些页：哪些页帧已经分配出去，哪些没有，某个进程结束后还要回收页帧，这些都需要记录，由于数量比较大（1M个），各种管理策略也不相同，最成功的当属伙伴算法，这里不再介绍，大家自行上网学习，不过不推荐初学者掌握并实现，能理解它的思想就可以了。有时我们只需要1byte字节，这些内存如何分配，总不能分配一个4KB吧，对于这种小空间分配，就涉及SLOB算法了。上面说的这些都是内存分配器该实现的，代码中只实现最简单的。

## 编译地址变动

进入页式内存后，内存分布变化比较大，内核逻辑地址变为了高地址（3GB左右），现在就涉及编译器生成的地址，加载地址和映射地址之间的问题了。内核加载地址依然是1MB（物理地址），这无法变化，因为只能在实模式下完成加载，由于需要将内核地址安排到3GB处，编译器生成的地址排列就是3GB起始的，这对局部变量影响不大，但对全局函数和变量影响比较大，为了能在实模式下也正常访问，则在实非页式内存需要将段的起始值暂且设置为-3GB，这样可以正常运行，进入页式内存后，将段起始值又变为0。内核的加载地址是1MB，进入页模式后要变为3GB，难道要将内核复制到3GB处？不需要，别忘了页式内存管理的核心是映射，通过映射，将3GB起始空间访问映射到物理0MB起始处即可。

可能会有疑惑为什么要将内核地址移动至3GB处？原因很简单，一般编译器都是从0处开始排列地址的，操作系统不能侵占这些地址，不然会加大编译器的复杂度，只能将内核搬迁至内存高端处。



UCore的内存分布示意图

我们的操作系统是借鉴UCore的，这里简要讲一下UCore的内存分布，在物理内存空间中，低1MB基本大多是BIOS相关的，只有bootloader是我们写的启动引导，1MB之后是我们的内核代码。注意KERNBASE是0XC0100000！编译起始地址是0XC0000000！

Page数组是管理全部物理页的，它与物理页的地址是顺序对应的，将一个页分配出去，填写页表后，同时要在page数组中记录，不然如何得知某个物理页是否可以被分配？

在Page以上的虚拟空间是内核用来存储管理用户进程的，比如用户进程的显示缓存，TSS，页表等等。我们的操作系统将其设置为128MB。

最右边的是内核的数据区（没有低1MB的的数据），在这里就有我们在entryinit.S中填写的映射了物理0~4MB空间的页表。CR3存储页目录表起始物理地址。其他页表需要我们在内存初始化中扩充。

## 寄存器与数据段寻址的差距

注意，GDTR与IDTR中的.base字段访问内存与gcc数据段访问内存不同，前者.base字段存储的是线性地址，不使用DS寄存器访问GDTR，不用加DS索引段的起始值，不是逻辑地址。后者访问内存就是我们之前学习的，逻辑地址加上DS索引段的起始地址最后通过页映射（如果开启了页模式）。GDTR中某个描述符的起始地址数值可为负。

以上的内容描述起来不多，但实现起来并不简单。

## TLB

TLB：转译后备缓冲区，在CPU缓存了页目录和页表，如果没有TLB，那么每次访问某个地址，CPU需要先依靠CR3的物理地址得到页目录表，通过页目录表访问内存得到页表，根据页表访问内存，需要访问3次内存，如果TLB中缓存了，MMU只需要一次就可以。

CPU中还有一个缓存，即cache，一般CPU的cache都是指内存访问缓存，即产品参数中的L1，L2，L3级缓存，缓存的出现都是为了加快内存读写速度，这些特性与TLB都很像，但最大的区别是，cache的刷新同步是透明的，我们无需关心，但TLB不一样。

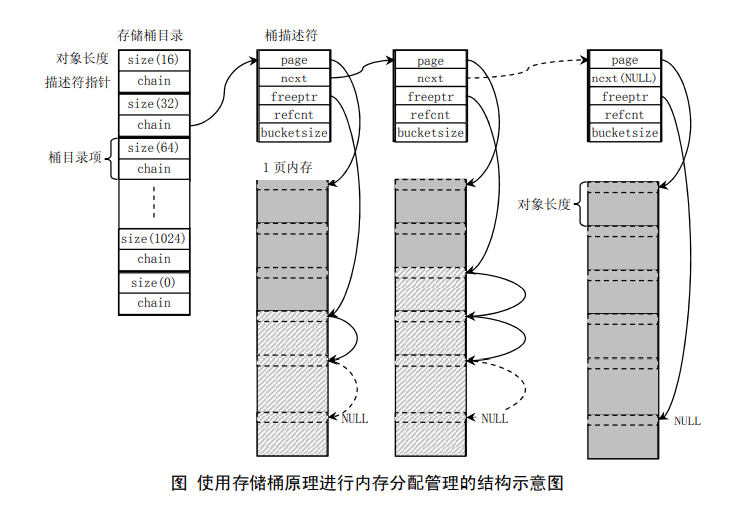
TLB何时更新：

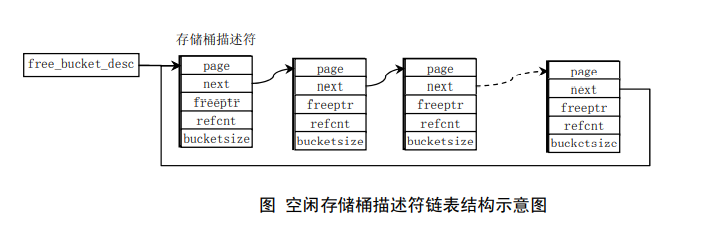
1. 执行加载CR3寄存器的汇编指令，CPU会将所有TLB缓存都失效，等价于刷新了所有。
2. 执行invlpg m汇编指令，该指令指明线性地址m在TLB中的记录失效。
3. 未命中，即：某个线性地址再TLB中查询不到记录，TLB才会从内存中读取三次访问内存，同时更新TLB该条记录。

上面三个情况，只有第二个有点难以理解，针对第二个情况举个例：比如我们第一次访问0Xffc00000这个线性地址，TLB更新了（第三种情况），但是现在，我们将0Xffc00000对应的页目录表或者页表所映射的物理内存帧替换掉，即：0Xffc00000对应的是页目录表中的1023项，根据1023项给出的物理地址可以得到页表，页表的0项存储的是0Xffc00000映射的物理页帧R，现在将R换成物理页帧W，再次访问0Xffc00000这个内存的数据，你得到的是R中的第1个字节的数据，而不是W的第1个字节的数据！！！注意这个区别，代码中会遇到这种情况。

## 小块内存开辟

小块内存的管理使用最简单的桶存储（bucket），这也是Linux0.11的实现，比 如对于请求内存块的长度在 32 字节或 32 字节以下但大于 16 字节时，就使用存储桶目录第二项对应的存储桶描述符链表分配内存块。其基本结构示意图见下图所示。该函数目前一次所能分配的最大内存长度是一个内存页面，即 4096 字节。





基本步骤如下：

1. 首先搜索目录，寻找适合请求内存块大小的目录项对应的描述符链表。当目录项的对象字节长 度大于请求的字节长度，就算找到了相应的目录项。如果搜索完整个目录都没有找到合适的目 录项，则说明用户请求的内存块太大。
2. 在目录项对应的描述符链表中查找具有空闲空间的描述符。如果某个描述符的空闲内存指针 freeptr 不为 NULL，则表示找到了相应的描述符。如果没有找到具有空闲空间的描述符，那么 我们就需要新建一个描述符。新建描述符的过程如下：
   1. 如果空闲描述符链表头指针还是 NULL 的话，说明是第一次调用 malloc()函数，此时需要 init\_bucket\_desc()来创建空闲描述符链表。
   2. 然后从空闲描述符链表头处取一个描述符，初始化该描述符，令其对象引用计数为 0，对象大小等于对应目录项指定对象的长度值，并申请一内存页面，让描述符的页面指针page指向该内存页，描述符的空闲内存指针 freeptr 也指向页开始位置。
   3. 对该内存页面根据本目录项所用对象长度进行页面初始化，建立所有对象的一个链表。也 即每个对象的头部都存放一个指向下一个对象的指针，最后一个对象的开始处存放一个 NULL 指针值。
   4. 然后将该描述符插入到对应目录项的描述符链表开始处。
3. 将该描述符的空闲内存指针 freeptr 复制为返回给用户的内存指针，然后调整该 freeptr 指向 描述符对应内存页面中下一个空闲对象位置，并使该描述符引用计数值增 1。

这个思想和磁盘管理也是一致的。

## 内存回收

怎么来，怎么回去。

## 代码解析

内存管理的代码较为繁琐，难度不比之前的简单，加之遇到了许多新的概念，本章代码变动比较大，主要变动是内核地址全部迁移至0XC0000000以上，编译起始地址不再是1MB，但bootloader的编译起始地址依然是从0X7C00开始，这无法修改，思考以下为什么？在bootloader中我们加载了GDT，起始地址是0，但现在内核的逻辑地址是0XC0000000开始，跳转进入内核后，逻辑地址+段起始地址得到物理地址，可bootloader将内核加载到了1MB起始处，这必然会访问错误，为此，我们只能在内核的第一行就让其加载CR3（CR3指明CPU页目录的起始地址，CR3存储的是物理地址，不同于GDTR是线性地址！）使其指向页目录表起始，并通过CR0开启页机制，所以需要提前就填写好页目录表与页表内容，由于我们的编译起始地址是0XC010000，自然将逻辑地址0XC0000000~0XC0000000+4MB映射到0~4MB物理地址，为什么映射4MB大小，仅仅是因为方便，页目录表中一项最多映射4MB，内核大小远小于4MB，这足够我们使用。不过这里没有这么简单，bootloader 中，GDT物理地址是在0X7C00~1MB之间，GDTR.base字段存储值同它的物理地址，那么问题来了，开启页机制后，GDTR中的地址变成了线性地址，GDTR中的数值还要访问页表获得物理地址，但现在我们映射的是0XC0000000~0XC0000000+4MB到0~4MB物理地址，那GDTR自然是访问出错的，谁也不知道用一个0X7C00~1MB间的GDTR地址在页表中索引最终得到一个什么值（野值，有极小概率能蒙对正确的值），可如果先设置GDTR的值为0XC0000000以上的范围（即0XC0000000+GDT物理存在地址），再开启页机制，问题就更大了，逻辑地址变换线性地址时就已经错了，自然无法使用一个错误的线性地址正确加载CR3的物理地址，这种临界交替的局面是非常棘手的，解决方案就是，让逻辑地址0XC0000000~0XC0000000+4MB与逻辑地址0X0~4MB都映射同样的物理地址：0~4MB，后开启页机制，这样寻址过程都是正确的，相当于我们在逻辑地址空间中为GDTR做了两手准备（页机制前和页机制后）。

现在内核算平稳进入了页机制，随后我们自行利用映射了的4MB空间构建页表，映射更多的物理空间，内核可用的内存就变得更多了。这里物理地址0~4MB被映射了两次！需要掌握为什么要映射两次。

非页机制下，只需要设置好GDT，整个地址空间就都可以正常运行，映射规则比较简单，页机制下的映射则需要挨个填写页表，同时，为了管理所有的页，还需要struct Page结构来记录每个物理页的使用状况，struct Page每个占36字节，以内存512MB为例，512\*1024\*1024/4/1024=131072个页，这131072个struct Page要组成一个数组，这个数组没有声明，依靠的是C语言如何生成数组的原理，仅struct Page数组就要占用：131072\*36=4718592byte=4.5MB，加上内核与BIOS各种显存地址，总共达5.6MB，但只映射了4MB空间，超出部分自然是访问出错，内存崩溃。所以进入内存初始化后，第一件事是开疆拓土，填写页表，要能装载下整个struct Page数组。

使用struct page数组来管理物理内存，记录哪些物理页被分配，哪些没有被分配。同时还需要管理虚拟地址，哪些虚拟地址被分配出去，哪些没有使用，这不难想，以最简单的C语言为例，使用malloc函数开辟一块内存时，返回值是一个指针，这个指针指向的是逻辑地址，不是物理地址，自然不可以将某个已经分配出去的逻辑地址二次分配给某个变量，那会导致内存混乱。

内存管理中另一个难题是，如何操作页表，理论中只教会你页表如何工作，填写怎样的数值可以让页表映射某个物理页帧，理论可没有教你如何读写页表，页目录表还好读写，至少内核的页目录表的逻辑地址我们知道并且可以直接读写，那新建的页表呢？可能会说，通过CR3得到页目录表，通过页目录表就可以得到页表的地址了，注意，页目录表中可是物理地址，不是逻辑地址，准确的说，没有仔细思考并且做出大量尝试，连开辟页表的实现不了，更不要想着读写一个新建立的页表了。

在内存管理中，一直都存在一个类似相对论的难点：你无法跳出自己的空间来了解自己的空间。也就是你无法开启上帝视角读写页表。

Linux中内核内存（896MB大小）采用了直接映射，即，将0XC0000000 ~ 0XC0000000+896MB逻辑地址线性一对一映射到物理地址的0 ~ 896MB上，这里的线性是开启页机制的，不是关闭页机制使用GDT的那个线性映射。这样做，直接将内核的虚拟地址管理直接甩给了物理内存页帧，只要物理页帧的引用值不为0，那么该物理内存就被使用，对应的逻辑地址（物理地址加上KERNBASE即对应的逻辑地址）自然也无法使用。

通过上面，读写页表一个方法是：始终让新建立的页表处在可访问范围内，说的通俗一点，新建立的页表你要得到它的逻辑地址，并且这个逻辑地址还要已经映射到这个页表上，很绕吧，如果没有熟练掌握线性地址，逻辑地址，物理地址，页机制，建议彻底掌握再来继续学习。

另一个方法，选定一个逻辑地址，比如：4GB-4MB=0Xffc00000，这个地址始终映射到需要读写的页表上，通过这个地址读写页表，记住，每次让该地址映射到某个新的物理页帧后，一定要使用invlpg指令，不然它会使用上一次TLB缓存读写页表，读写的页表本不是本次而是之前的，使用invplg命令强制让CPU使以前TLB缓存的失效，从内存中刷新。第二种方法本质让0xffc00000这个地址充当指针，具体实现如下：

在链接脚本中保留一个4KB空间P（该空间物理地址也要4KB对齐），P是一个页表。映射0xffc00000逻辑地址，高10位对应页目录表的1023项，所以将P的物理地址填入页目录表的1023项，P是处在内核低4MB空间的，所以P中的数据可以直接使用逻辑地址读写，当我们想要操作某个物理页X时，将X的物理地址填入P的0项，现在0xffc00000其实映射到了物理页X，对0Xffc00000读写就是读写X，当要读写物理页W时，将W的地址写入P的0项处，使用invplg刷新0xffc00000的TLB缓存，再次使用0xffc00000逻辑地址读写就能读写W物理页。

这种方法其实是将代码中采用第二种方法，这样会损失4MB的逻辑地址空间和4KB物理内存，其实只要你技术够好，只会损失1B的逻辑第和4B的物理空间，这却决于你对链接脚本和三种地址关系的驾驭能力，代码没有对此做优化。

在本章中，迫于时间问题，只实现了最简单的内存开辟：每次开辟4KB整数倍的内存空间，SLOB桶分配并没实现。至此，所有的底层硬件操作基本结束，完成了最艰难的历程。

### entryInit.S

#include<MMU.h>

#include<MMLayout.h>

#define REALLOC(x) (x - KERNBASE)

.text

.globl kern\_entry

kern\_entry:

# load pa of boot pgdir

#REALLOC将逻辑地址换算为物理地址

movl $REALLOC(\_\_boot\_pgdir), %eax

#加载页目录物理地址至CR3

movl %eax, %cr3

# enable paging

#读取CR0寄存器值，通过“或”操作将页机制开启，并写回CR0

movl %cr0, %eax

orl $(CR0\_PE | CR0\_PG | CR0\_AM | CR0\_WP | CR0\_NE | CR0\_TS | CR0\_EM | CR0\_MP), %eax

andl $~(CR0\_TS | CR0\_EM), %eax

movl %eax, %cr0

# update eip

# now, eip = 0x1.....

leal next, %eax

# set eip = KERNBASE + 0x1.....

#通过间接跳转更新EIP

jmp \*%eax

next:

# unmap va 0 ~ 4M, it's temporary mapping

xorl %eax, %eax

movl %eax, \_\_boot\_pgdir

# set ebp, esp

movl $0x0, %ebp

# the kernel stack region is from bootstack -- bootstacktop,

# the kernel stack size is KSTACKSIZE (8KB)defined in memlayout.h

movl $bootstacktop, %esp

# now kernel stack is ready , call the first C function

call kernel\_init

# should never get here

spin:

jmp spin

.data

#4KB地址对齐编译

.align PGSIZE

#内核栈空间

.globl bootstack

bootstack:

#告知编译器空出8KB\*2空间

.space KSTACKSIZE \*2

# 声明bootstacktop标签为全局标签，可被其他文件引入使用

.globl bootstacktop

bootstacktop:

# kernel builtin pgdir

# an initial page directory (Page Directory Table, PDT)

# These page directory table and page table can be reused!

.section .data.pgdir

#4KB地址对齐

.align PGSIZE

#声明页目录表，并为全局标签

.globl \_\_boot\_pgdir

\_\_boot\_pgdir:

# map va 0 ~ 4M to pa 0 ~ 4M (temporary)

.long REALLOC(\_\_boot\_pt1) + (PTE\_P | PTE\_U | PTE\_W)

#空出(KERNBASE >> PGSHIFT >> 10 << 2) - (. - \_\_boot\_pgdir)大小空间

.space (KERNBASE >> PGSHIFT >> 10 << 2) - (. - \_\_boot\_pgdir) # pad to PDE of KERNBASE

# map va KERNBASE + (0 ~ 36M) to pa 0 ~ 36M

.long REALLOC(\_\_boot\_pt1) + (PTE\_P | PTE\_U | PTE\_W)

.space PGSIZE - (. - \_\_boot\_pgdir) # pad to PGSIZE

#设置循环，填写页表，映射0~4MB空间

.set i, 0\*1024

.globl \_\_boot\_pt1

\_\_boot\_pt1:

.rept 1024

.long i \* PGSIZE + (PTE\_P | PTE\_W)

.set i, i + 1

.endr

### memoryInit.c

#include <defs.h>

#include <MMLayout.h>

#include <MMU.h>

#include <x86.h>

#include <stdio.h>

#include <memoryInit.h>

//GCC编译后的内核结束地址，从kernel.ld脚本中引入

extern char end[];

//从entryInit.S中引入

extern int \_\_boot\_pt1[];

//内核加内核初始化数据占用空间的结束地址，通过计算可以得到

char \* kernnelpageend;

//struct Page数组的起始地址

struct Page \* pageRecord=(struct Page \*)0x0;

//物理内存页数量

int sumpagenum=0;

//内核页目录表起始地址，从entryInit.S中引入

extern char \_\_boot\_pgdir[];

//从kernel.ld中引入，与0xffc00000逻辑地址结合使用

extern char pagepionter[];

//内存探测得到的最大/最小可用空间

int maxMemory=0;

int minMemory=0;

int num4KB=0;

//刷新TLB对@la逻辑地址的记录，@pgdir是页目录起始地址

void

tlb\_invalidate(pde\_t \*pgdir, uintptr\_t la) {

if (rcr3() == pgdir-KERNBASE) {

invlpg((void \*)la);

}

}

//重新加载的GDT表

static struct segdesc gdt[] = {

SEG\_NULL,

[SEG\_KTEXT] = SEG(STA\_X | STA\_R, 0x0, 0xFFFFFFFF, DPL\_KERNEL),

[SEG\_KDATA] = SEG(STA\_W, 0x0, 0xFFFFFFFF, DPL\_KERNEL),

[SEG\_UTEXT] = SEG(STA\_X | STA\_R, 0x0, 0xFFFFFFFF, DPL\_USER),

[SEG\_UDATA] = SEG(STA\_W, 0x0, 0xFFFFFFFF, DPL\_USER),

[SEG\_TSS] = SEG\_NULL,

};

static struct pseudodesc gdt\_pd = {

sizeof(gdt) - 1, (uintptr\_t)gdt

};

//内存初始化

void memory\_init()

{

//一个struct Page 36字节

//将读写页表的逻辑地址0xffc00000填入页目录表，并设置权限

((int \*)\_\_boot\_pgdir)[1023]=((((uint32\_t)pagepionter)-KERNBASE) & 0xfffff000) + (PTE\_P | PTE\_U | PTE\_W);

getMaxMemory();

mapPageForKernel();

pageRecordInit();

/\*test the kmalloc4KB function

char \* tmp=0;

tmp=kmalloc4KB(5000);

tmp[5000\*1024\*4-1]=4;

\*/

}

//重新加载GDT

void reloadGDT()

{

asm volatile ("lgdt (%0)" :: "r" (&gdt\_pd));

asm volatile ("movw %%ax, %%gs" :: "a" (USER\_DS));

asm volatile ("movw %%ax, %%fs" :: "a" (USER\_DS));

asm volatile ("movw %%ax, %%es" :: "a" (KERNEL\_DS));

asm volatile ("movw %%ax, %%ds" :: "a" (KERNEL\_DS));

asm volatile ("movw %%ax, %%ss" :: "a" (KERNEL\_DS));

// reload cs

asm volatile ("ljmp %0, $1f\n 1:\n" :: "i" (KERNEL\_CS));

}

//根据实模式下int 0x15内存探测功能得到可用内存最大值

static void getMaxMemory()

{

// memory\_record指向int 15h返回起始地址，目前已开启页机制，故需加上KERNBASE得到逻辑地址

struct e820map \* memory\_record=(struct e820map \*)(0x8000+KERNBASE);

for (int i=0;i<memory\_record->nr\_map;i++)

{

uint32\_t seg\_start = memory\_record->map[i].addr;

uint32\_t seg\_end = memory\_record->map[i].addr + memory\_record->map[i].size;

if (seg\_end > maxMemory)

maxMemory = seg\_end;

if (seg\_start < minMemory)

minMemory = seg\_start;

}

}

//为内核映射内存，主要是记录物理内存的struct page数组需要开辟更多内存，但此时kamlloc函数工作依赖struct page数组，故需通过计算为struct page数组映射内存

static void mapPageForKernel() {

//内核页目录逻辑地址

uint32\_t \*kernelbootpgdir = (uint32\_t \*) \_\_boot\_pgdir+KERNBASE;

//内存总共有多少个4KB物理页

sumpagenum = maxMemory / (4 \* 1024);

uint32\_t neededmemory = sumpagenum \* sizeof(struct Page);//struct page数组所需空间

if(0!=neededmemory & 0xfff) //4KB地址对齐

neededmemory=((neededmemory>>12)+1)<<12;

//如果内核加struct page数组超过4MB，还需要考虑新建页表所耗费的空间

neededmemory+= neededmemory/1024+1024\*4; //记录page数组，页表映射需要的空间

//内核实际结束地址（逻辑地址）

kernnelpageend=end + neededmemory;

if(neededmemory<(4\*1024\*1024- (uint32\_t) end + KERNBASE ))//初始化提供的4MB空间足够,多余的映射消除

{

if(0 != kernnelpageend & 0xfff)

kernnelpageend= (((uint32\_t)kernnelpageend>>12)+1)<<12;

//置0即为消除映射

for (int i=(((uint32\_t)kernnelpageend-KERNBASE)>>12)+1;i<1024;i++)

((int \*)\_\_boot\_pt1)[i]=0;

//更新内核结束地址

pageRecord=(struct Page \*)end;

return;

}

//超过4MB空间，neededmemory变为还需开辟空间，4MB大小已映射，故减去4MB

neededmemory -= (4\*1024\*1024- (uint32\_t) end + KERNBASE );

//4KB地址对齐

if (0 != neededmemory & 0xfff)

neededmemory=((neededmemory>>12)+1)<<12;

//页目录表需记录条数

int num4M = neededmemory / (4 \* 1024 \* 1024);

if (0 != neededmemory & 0xfffff) //4MB

num4M++;

//页表需记录条数

num4KB = neededmemory >> 12;

kernnelpageend=(char \*)(KERNBASE+4\*1024\*1024+neededmemory);

//4KB对齐

if (0 != (uint32\_t)kernnelpageend % (4 \* 1024))

kernnelpageend=(char \* )((((uint32\_t) kernnelpageend)>>12 + 1)<<12);

uint32\_t \*pagetable = (uint32\_t \*) end;

int countpt=0;

for (countpt = 0; countpt < num4KB ; countpt++) {

pagetable[countpt] = 4 \* 1024 \* 1024 + countpt \* 1024 \* 4 + (PTE\_P | PTE\_U | PTE\_W);

}

for (int i = 1; i <= num4M ; i++) {

kernelbootpgdir[i + ((KERNBASE + 4 \* 1024 \* 1024) >> 22)] =

((uint32\_t)(pagetable) + (i - 1) \* 1024) - KERNBASE + (PTE\_P | PTE\_U | PTE\_W);

}

pageRecord=(struct Page \* )(pagetable+countpt);

if(0!= ((uint32\_t)pageRecord & 0x3ff))

{//4KB对齐 不可以侵占最后一个4KB页表

pageRecord=(((uint32\_t)pageRecord)>>12 + 1)<<12;

}

else

{

pageRecord=(((uint32\_t)pageRecord)>>12 )<<12;

}

}

static void pageRecordInit()

{

struct e820map \* memory\_record=(struct e820map \*)(0x8000+KERNBASE);

//获取所有可用的内存空间并组成4KB大小的页

int countpage=0;

for (int i=0;i<memory\_record->nr\_map;i++) {

uint32\_t seg\_start = memory\_record->map[i].addr;

uint32\_t seg\_end = memory\_record->map[i].addr + memory\_record->map[i].size;

if (memory\_record->map[i].type == E820\_ARM) {

//将地址对齐至4KB

if (0 != (seg\_start & 0xfff)) {

seg\_start = ((seg\_start & 0xFFFFF000)+1) << 12;

}

seg\_end = seg\_end & 0xFFFFF000;

}

else {

//将地址对齐至4KB

if (0 != (seg\_end & 0xfff)) {

seg\_end =( (seg\_end & 0xFFFFF000) +1) << 12;

}

seg\_start = seg\_start & 0xFFFFF000;

cprintf("%d %d\n",seg\_start,seg\_end);

}

if ((seg\_end - seg\_start) > PGSIZE) {

int pagenum = (seg\_end - seg\_start) / PGSIZE;

for (int j = 0; j < pagenum; j++) {

pageRecord[countpage + j].ref = 0;

}

countpage += pagenum;

}

}

//将已经使用的页进行标记

//0~4MB物理内存被映射两次，ref值为2

//4MB~ kernnelpageend-KERNBASE 物理内存被映射一次，ref为1

for (int i=0;i<=(uint32\_t)(kernnelpageend-KERNBASE)/PGSIZE;i++)

pageRecord[i].ref=2;

}

//查询pageRecord数组，开辟连续的 @num\_n 个物理页，返回第一个物理页的逻辑地址

char \* kernelgetpages(int num\_n)

{

char flag=1;

for (int i=1024;i<128\*1024/4;i++)

if(pageRecord[i].ref==0)

{

int j=0;

flag=1;

for (j=0;j<num\_n;j++)

if(pageRecord[i+j].ref!=0)

{flag=0;break;}

if(1==flag)

{

for (j=0;j<num\_n;j++)

pageRecord[i+j].ref++;

//根据pageRecord的下标i推算出逻辑地址

return ((char \*)(i\*PGSIZE+KERNBASE));

}

i += j;

}

return (char \* )0;

}

//内核开辟@num \* 4KB内存

char \* kmalloc4KB(int num\_n)

{

char \* startaddress=kernelgetpages( num\_n);

cprintf("start address is %d ",startaddress);

if(startaddress==0)

{

cprintf("kmalloc memory is space! kamlloc failed\n");

return startaddress;

}

//内核页目录表换算为逻辑地址，startaddress高10位填入

//通过页目录表中索引得 页表物理地址 转换为虚拟地址（+KERNBASE），填页表相关项

for (int i=0;i<num\_n;i++)

{

//待映射的逻辑页地址

char \* waitingmaplogicaddress=startaddress+i\*PGSIZE;

//待映射页的PDT内偏移值：映射页逻辑地址的高10位

int PDToffset= (uint32\_t)waitingmaplogicaddress>>22;

//待映射页的页表内偏移值：映射页逻辑地址的中10位

int PToffset = ((uint32\_t)waitingmaplogicaddress & 0x3FF000)>>12;

//waitingmaplogicaddress得出页目录偏移值

if(((int \*)\_\_boot\_pgdir) [PDToffset]==0)

{

//开辟一个页表tmppage

//tmppage地址还未填写页表，直接操作内存奔溃

char \* tmppage =kernelgetpages(1);

//将这个新的地址tmppage暂时映射到pagepionter下,tmppage是新开辟的页表

((int \* )pagepionter)[0]=(((uint32\_t)tmppage-KERNBASE) & 0xfffff000) + (PTE\_P | PTE\_U | PTE\_W);

invlpg(0xffc00000);

//新开辟的页表转换为物理地址，补齐权限写入PDT中

((int \*)\_\_boot\_pgdir) [PDToffset]=(((uint32\_t)tmppage-KERNBASE) & 0xfffff000)+ (PTE\_P | PTE\_U | PTE\_W);

//cprintf("%d\n",((int \*)\_\_boot\_pgdir) [PDToffset]);

//0xffc0 0000地址是一个特殊的用于操作新开辟页表的地址，

for (int n=0;n<1024;n++)

{ //将该页表1024项记录全部清空

((int \*)0xffc00000)[n]= 0;

}

}

else

{

//通过PDT中的物理地址+KERNBASE得到PDToffset索引页表的逻辑地址 注意地12位是标记

((int \* )pagepionter)[0]= ((int \*)\_\_boot\_pgdir)[PDToffset];

//刷新TLB记录，不然出现TLB记录幻影

invlpg(0xffc00000);

}

//将待映射页映射入页表 the kmallocnum is more than space 4MB this code will break down

((int \*)0xffc00000)[PToffset]=( ((uint32\_t)waitingmaplogicaddress -KERNBASE) & 0xfffff000 )+ (PTE\_P | PTE\_U | PTE\_W);

}

return startaddress;

}

//桶分配小字节内存

char \* bucketmalloc(int byte)

{

struct BucketDes \* p=0;

int type=1024;

//使用那个尺寸块

if(byte<=1024-1)

{

p=kbucket.size1024chain;

type=1024;

}

if(byte<=512-1)

{

p=kbucket.size512chain;

type=512;

}

if(byte<=256-1)

{

p=kbucket.size256chain;

type=256;

}

if(byte<=128-1)

{

p=kbucket.size128chain;

type=128;

}

if(byte<=64-1)

{

p=kbucket.size64chain;

type=64;

}

if(byte<=32-1)

{

p=kbucket.size32chain;

type=32;

}

if(byte<=16-1)

{

p=kbucket.size16chain;

type=16;

}

//标记能否有合适块

char flag=0;

struct BucketDes \*pre=p;//for循环判断条件对只有一个节点的链表会判断错误，使用pre来防范

for (;p!=0 && p->next != 0 ;p=p->next)

if(p->rest>0)

{flag=1;break;}

if(pre!=0 && pre->rest>0) //链表第一个节点很特殊

{p=pre;flag=1;}

char \* resaddress=0;

if(1==flag)

{//有合适的块空间

int currentaddress=(p->freeoffset);//通过BucketDes确定页内块起始偏移量

resaddress=p->page+currentaddress; //得到该块的逻辑地址

p->freeoffset=resaddress[0]\*16; //该块未分配前，第一个字节处存储的是下一个空闲块的页内偏移量，依靠这个偏移量更新BucketDes的freeoffset值

p->rest--; //可用块减一

resaddress[0]=type/16; //该块的第一个字节存储该块大小（除16是为了能让一个字节就存储下，不然需要耗费两个字节）

resaddress++; //真正可用地址需要加一

}

else

{//没有合适的块，开辟一个页，依靠type初始化里面的块，主要初始化空闲链表

//首先需要开辟一个BucketDes并接入对应的kbucket.next

struct BucketDes\* newbucketDes= (struct BucketDes \*)mallocBucketDes();

//开一新的4KB页

char \* newpage=kmalloc4KB(1);

//初始化新页的空闲链表

for(int i=0;i<PGSIZE/type;i++)

{

newpage[i\*type]= (i+1)\*type/16;

}

//最后一块的空闲指针用-1标记

newpage[PGSIZE-type]=-1;

if(p==0)

//接入对应的kbucket.next

switch (type) {

case 16: kbucket.size16chain=newbucketDes;break;

case 32: kbucket.size32chain=newbucketDes;break;

case 64: kbucket.size64chain=newbucketDes;break;

case 128: kbucket.size128chain=newbucketDes;break;

case 256: kbucket.size256chain=newbucketDes;break;

case 512: kbucket.size512chain=newbucketDes;break;

case 1024: kbucket.size1024chain=newbucketDes;break;

}

else

p->next=newbucketDes; //将新的BucketDes拼接在p后面

//初始化这个描述块

newbucketDes->next=0;

newbucketDes->page=newpage;

newbucketDes->freeoffset=0;

newbucketDes->rest=PGSIZE/type;

//划分出type大小并修改空闲块链表，同bucketmalloc(int byte)函数

int currentaddress=(newbucketDes->freeoffset);

resaddress=newbucketDes->page+currentaddress;

newbucketDes->freeoffset=resaddress[0]\*16;

newbucketDes->rest--;

resaddress[0]=type/16;

resaddress++;

}

return resaddress;

}

//开辟一个块描述符，sizeof（struct BucketDes）=16 ,故本质是开辟一个32字节空间

char \* mallocBucketDes()

{

struct BucketDes \* p=0;

int type=1024;

int byte=sizeof (struct BucketDes);

if(byte<=32-1)

{

p=kbucket.size32chain;

type=32;

}

if(byte<=16-1)

{

p=kbucket.size16chain;

type=16;

}

char flag=0;

struct BucketDes \* pre=p;

//查看32字节块中是否用空间

for (;p!=0 && p->next != 0; p=p->next)

if(p->rest>0)

{flag=1;break;}

if(pre!=0 && pre->rest>0) //链表第一个节点很特殊

{p=pre;flag=1;}

char \* resaddress=0;

if(1==flag)

{//有合适的32字节块，同bucketmalloc(int byte)对应的操作

int currentaddress=(p->freeoffset);

resaddress=p->page+currentaddress;

p->freeoffset=resaddress[0]\*16;

p->rest--;

resaddress[0]=type/16;

resaddress++;

}else

{//需要开辟新的4KB页并初始化

char \*newpage=kmalloc4KB(1);

for (int i=0;i<PGSIZE/type;i++)

{//初始化空闲块链表

newpage[i\*type]= (i+1)\*type/16;

}

int freeoffset=newpage[0];

if(p==0)

switch (type) {

case 16: kbucket.size16chain=((struct BucketDes \* )(newpage+1)); break;

case 32: kbucket.size32chain=((struct BucketDes \* )(newpage+1)); break;

}

else

p->next=((struct BucketDes \* )(newpage+1)); //在新的页中选定第一个块存储32字节的块描述符

//下面都是使用newpage+1逻辑地址 通过指针类型转换写入描述快初始化数值

newpage[0]=type/16;//存储该块占用空间值/16，压缩空间

resaddress=((struct BucketDes \* )(newpage+1));

((struct BucketDes \* )(newpage+1))->next=0;

((struct BucketDes \* )(newpage+1))->freeoffset=freeoffset\*16;

((struct BucketDes \* )(newpage+1))->rest=PGSIZE/type-1;

((struct BucketDes \* )(newpage+1))->page=newpage;

}

return resaddress;

}

第四章 进程

进程，计算机专业的同学都很熟悉，学习过操作系统课程后，对于其理解没有过多困难，这里只提一下PCB和各种调度策略，没有其他需要讲解的知识，大家通过代码都可以读懂，在进程中，基本没有涉及底层硬件操作，所有代码以C为主。

## PCB

进程切换是需要保存和恢复一些CPU寄存器；（当一个进程A挂起，进程B占用CPU资源。AB使用同一个寄存器空间，A先得保存数据到PCB中，B进程数据然后覆盖寄存器中A数据，执行）

## 调度策略：

决定如何切换各个进程，通过PCB感知各个进程的状态，完成读写

网络参考博客：

<https://blog.csdn.net/better0332/article/details/3416749>

<https://blog.csdn.net/feijj2002_/article/details/4597174>

<https://www.cnblogs.com/chenwb89/p/operating_system_004.html>

<https://blog.csdn.net/zhd19910223/article/details/42059029>

<http://blog.itpub.net/10752043/viewspace-993092/>

<https://blog.csdn.net/csd3176/article/details/100211326>

<https://blog.csdn.net/weixin_30279751/article/details/96549717>

<https://www.cnblogs.com/chenwb89/p/operating_system_004.html>

<https://blog.csdn.net/feijj2002_/article/details/4597174>

<https://blog.csdn.net/better0332/article/details/3416749>

https://www.cnblogs.com/t1mes/articles/7905336.html

<https://blog.csdn.net/longintchar/article/details/51439689>

<http://blog.chinaunix.net/uid-20718384-id-3418279.html>

<https://blog.csdn.net/judyge/article/details/52303020>

https://blog.csdn.net/q936330007/article/details/52344191

<https://www.cnblogs.com/nullecho/p/10266467.html>

https://blog.csdn.net/qq\_33339479/article/details/80423291

https://blog.csdn.net/qq\_35642036/article/details/82809812

致谢

这篇学习记录文章与代码能够最终完成，全部仰赖于网络上各位技术高超的博客主们，是他们热衷于分享自己所学得的知识并传播，再次向他们感谢，感谢他们的开源精神与无私奉献精神！！！

感谢我的本科毕设导师，鲍老师！他带我入门操作系统！

感谢我的学校，东北大学！感谢对我本硕的培养！

感谢在麒麟软件公司·长沙工作的本科同学：英俊潇洒风流倜傥玉树临风，人见人爱花见花开，集才华与智慧于一身的宇宙无敌小帅哥张昕宇同学，他总能发给我一些技术资料！他也是做操作系统的！

附录

## BIOS中断:

1、显示服务(Video Service——INT 10H)

00H —设置显示器模式0CH —写图形象素

01H —设置光标形状0DH —读图形象素

02H —设置光标位置0EH —在Teletype模式下显示字符

03H —读取光标信息0FH —读取显示器模式

04H —读取光笔位置10H —颜色

05H —设置显示页11H —字体

06H、07H —初始化或滚屏12H —显示器的配置

08H —读光标处的字符及其属性13H —在Teletype模式下显示字符串

09H —在光标处按指定属性显示字符1AH —读取/设置显示组合编码

0AH —在当前光标处显示字符1BH —读取功能/状态信息

0BH —设置调色板、背景色或边框1CH —保存/恢复显示器状态

(1)、功能00H

功能描述：设置显示器模式

入口参数：AH＝00H

AL＝显示器模式，见下表所示

出口参数：无

可用的显示模式如下所列：

显示模式显示模式属性显示模式显示模式属性

00H40×2516色 文本01H40×2516色 文本

02H80×2516色 文本03H80×2516色 文本

04H320×2004色05H320×2004色

06H640×2002色07H80×252色 文本

08H160×20016色09H320×20016色

0AH640×2004色0BH保留

0CH保留0DH320×20016色

0EH640×20016色0FH640×3502(单色)

10H640×3504色10H640×35016色

11H640×4802色12H640×48016色

13H640×480256色

对于超级VGA显示卡，我们可用AX＝4F02H和下列BX的值来设置其显示模式。

BX显示模式属性BX显示模式属性

100H640×400256色101H640×480256色

102H800×60016色103H800×600256色

104H1024×76816色105H1024×768256色

106H1280×102416色107H1280×1024256色

108H80×60文本模式109H132×25文本模式

10AH132×43文本模式10BH132×50文本模式

10CH132×60文本模式

(2)、功能01H

功能描述：设置光标形状

入口参数：AH＝01H

CH低四位＝光标的起始行

CL低四位＝光标的终止行

出口参数：无

(3)、功能02H

功能描述：用文本坐标下设置光标位置

入口参数：AH＝02H

BH＝显示页码

DH＝行(Y坐标)

DL＝列(X坐标)

出口参数：无

(4)、功能03H

功能描述：在文本坐标下，读取光标各种信息

入口参数：AH＝03H

BH＝显示页码

出口参数：CH＝光标的起始行

CL＝光标的终止行

DH＝行(Y坐标)

DL＝列(X坐标)

(5)、功能04H

功能描述：获取当前状态和光笔位置

入口参数：AH＝04H

出口参数：AH＝00h——光笔未按下/未触发，01h——光笔已按下/已触发

BX＝象素列(图形X坐标)

CH＝象素行(图形Y坐标，显示模式：04H~06H)

CX＝象素行(图形Y坐标，显示模式：0DH~10H)

DH＝字符行(文本Y坐标)

DL＝字符列(文本X坐标)

(6)、功能05H

功能描述：设置显示页，即选择活动的显示页

入口参数：AH＝05H

AL＝显示页

对于CGA、EGA、MCGA和VGA，其显示页如下表所列： 模式页数显示器类型

00H、01H0~7CGA、EGA、MCGA、VGA

02H、03H0~3CGA

02H、03H0~7EGA、MCGA、VGA

07H0~7EGA、VGA

0DH0~7EGA、VGA

0EH0~3EGA、VGA

0FH0~1EGA、VGA

10H0~1EGA、VGA

对于PCjr：

AL＝80H——读取CRT/CPU页寄存器

81H——设置CPU页寄存器

82H——设置CRT页寄存器

83H——设置CRT/CPU页寄存器

BH＝CRT页(子功能号82H和83H)

BL＝CPU页(子功能号81H和83H)

出口参数：对于前者，无出口参数，但对PCjr在子功能80H~83H调用下，有：BH＝CRT页寄存器，BL＝CPU页寄存器

(7)、功能06H和07H

功能描述：初始化屏幕或滚屏

入口参数：AH＝06H——向上滚屏，07H——向下滚屏

AL＝滚动行数(0——清窗口)

BH＝空白区域的缺省属性

(CH、CL)＝窗口的左上角位置(Y坐标，X坐标)

(DH、DL)＝窗口的右下角位置(Y坐标，X坐标)

出口参数：无

(8)、功能08H

功能描述：读光标处的字符及其属性

入口参数：AH＝08H

BH＝显示页码

出口参数：AH＝属性

AL＝字符

(9)、功能09H

功能描述：在当前光标处按指定属性显示字符

入口参数：AH＝09H

AL＝字符

BH＝显示页码

BL＝属性(文本模式)或颜色(图形模式)

CX＝重复输出字符的次数

出口参数：无

(10)、功能0AH

功能描述：在当前光标处按原有属性显示字符

入口参数：AH＝0AH

AL＝字符

BH＝显示页码

BL＝颜色(图形模式，仅适用于PCjr)

CX＝重复输出字符的次数

出口参数：无

(11)、功能0BH

功能描述：设置调色板、背景色或边框

入口参数：AH＝0BH

设置颜色：BH＝00H，BL＝颜色

选择调色板：BH＝01H，BL＝调色板(320×200、4种颜色的图形模式)

出口参数：无

(12)、功能0CH

功能描述：写图形象素

入口参数：AH＝0CH

AL＝象素值

BH＝页码

(CX、DX)＝图形坐标列(X)、行(Y)

出口参数：无

(13)、功能0DH

功能描述：读图形象素

入口参数：AH＝0DH

BH＝页码

(CX、DX)＝图形坐标列(X)、行(Y)

出口参数：AL＝象素值

(14)、功能0EH

功能描述：在Teletype模式下显示字符

入口参数：AH＝0EH

AL＝字符

BH＝页码

BL＝前景色(图形模式)

出口参数：无

(15)、功能0FH

功能描述：读取显示器模式

入口参数：AH＝0FH

出口参数：AH＝屏幕字符的列数

AL＝显示模式(参见功能00H中的说明)

BH＝页码

(16)、功能10H

功能描述：颜色中断。其子功能说明如下： 功能号　子功能名称功能号　子功能名称

00H — 设置调色板寄存器01H — 设置边框颜色

02H — 设置调色板和边框03H — 触发闪烁/亮显位

07H — 读取调色板寄存器08H — 读取边框颜色

09H — 读取调色板和边框10H — 设置颜色寄存器

12H — 设置颜色寄存器块13H — 设置颜色页状态

15H — 读取颜色寄存器17H — 读取颜色寄存器块

1AH — 读取颜色页状态1BH — 设置灰度值

(17)、功能11H

功能描述：字体中断。其子功能说明如下： 子功能号子功能名称

00H装入用户字体和可编程控制器

10H装入用户字体和可编程控制器

01H装入8×14 ROM字体和可编程控制器

11H装入8×14 ROM字体和可编程控制器

02H装入8×8 ROM字体和可编程控制器

12H装入8×8 ROM字体和可编程控制器

03H设置块指示器

04H装入8×16 ROM字体和可编程控制器

14H装入8×16 ROM字体和可编程控制器

20H设置INT 1Fh字体指针

21H为用户字体设置INT 43h

22H为8×14 ROM字体设置INT 43H

23H为8×8 ROM字体设置INT 43H

24H为8×16 ROM字体设置INT 43H

30H读取字体信息

(18)、功能12H

功能描述：显示器的配置中断。其子功能说明如下： 功能号 功能名称 功能号 功能名称

10H — 读取配置信息20H — 选择屏幕打印

30H — 设置扫描行31H — 允许/禁止装入缺省调色板

32H — 允许/禁止显示33H — 允许/禁止灰度求和

34H — 允许/禁止光标模拟35H — 切换活动显示

36H — 允许/禁止屏幕刷新

(19)、功能13H

功能描述：在Teletype模式下显示字符串

入口参数：AH＝13H

BH＝页码

BL＝属性(若AL=00H或01H)

CX＝显示字符串长度

(DH、DL)＝坐标(行、列)

ES:BP＝显示字符串的地址 AL＝显示输出方式

0——字符串中只含显示字符，其显示属性在BL中。显示后，光标位置不变

1——字符串中只含显示字符，其显示属性在BL中。显示后，光标位置改变

2——字符串中含显示字符和显示属性。显示后，光标位置不变

3——字符串中含显示字符和显示属性。显示后，光标位置改变

出口参数：无

(20)、功能1AH

功能描述：读取/设置显示组合编码，仅PS/2有效，在此从略

(21)、功能1BH

功能描述：读取功能/状态信息，仅PS/2有效，在此从略

(22)、功能1CH

功能描述：保存/恢复显示器状态，仅PS/2有效，在此从略

2、直接磁盘服务(Direct Disk Service——INT 13H)

00H —磁盘系统复位0EH —读扇区缓冲区

01H —读取磁盘系统状态0FH —写扇区缓冲区

02H —读扇区10H —读取驱动器状态

03H —写扇区11H —校准驱动器

04H —检验扇区12H —控制器RAM诊断

05H —格式化磁道13H —控制器驱动诊断

06H —格式化坏磁道14H —控制器内部诊断

07H —格式化驱动器15H —读取磁盘类型

08H —读取驱动器参数16H —读取磁盘变化状态

09H —初始化硬盘参数17H —设置磁盘类型

0AH —读长扇区18H —设置格式化媒体类型

0BH —写长扇区19H —磁头保护

0CH —查寻1AH —格式化ESDI驱动器

0DH —硬盘系统复位

(1)、功能00H

功能描述：磁盘系统复位

入口参数：AH＝00H

DL＝驱动器，00H~7FH：软盘；80H~0FFH：硬盘

出口参数：CF＝0——操作成功，AH＝00H，否则，AH＝状态代码，参见功能号01H中的说明

(2)、功能01H

功能描述：读取磁盘系统状态

入口参数：AH＝01H

DL＝驱动器，00H~7FH：软盘；80H~0FFH：硬盘

出口参数：AH＝00H，AL＝状态代码，其定义如下：

00H — 无错 01H — 非法命令

02H — 地址目标未发现03H — 磁盘写保护(软盘)

04H — 扇区未发现05H — 复位失败(硬盘)

06H — 软盘取出(软盘)07H — 错误的参数表(硬盘)

08H — DMA越界(软盘)09H — DMA超过64K界限

0AH — 错误的扇区标志(硬盘)0BH — 错误的磁道标志(硬盘)

0CH — 介质类型未发现(软盘)0DH — 格式化时非法扇区号(硬盘)

0EH — 控制数据地址目标被发现(硬盘)0FH — DMA仲裁越界(硬盘)

10H — 不正确的CRC或ECC编码11H — ECC校正数据错(硬盘)

　CRC:Cyclic Redundancy Check code

　ECC:Error Checking & Correcting code

20H — 控制器失败40H — 查找失败

80H — 磁盘超时(未响应)AAH — 驱动器未准备好(硬盘)

BBH — 未定义的错误(硬盘)CCH — 写错误(硬盘)

E0H — 状态寄存器错(硬盘)FFH — 检测操作失败(硬盘)

(3)、功能02H

功能描述：读扇区

入口参数：AH＝02H

AL＝扇区数

CH＝柱面

CL＝扇区

DH＝磁头

DL＝驱动器，00H~7FH：软盘；80H~0FFH：硬盘

ES:BX＝缓冲区的地址

出口参数：CF＝0——操作成功，AH＝00H，AL＝传输的扇区数，否则，AH＝状态代码，参见功能号01H中的说明

(4)、功能03H

功能描述：写扇区

入口参数：AH＝03H

AL＝扇区数

CH＝柱面

CL＝扇区

DH＝磁头

DL＝驱动器，00H~7FH：软盘；80H~0FFH：硬盘

ES:BX＝缓冲区的地址

出口参数：CF＝0——操作成功，AH＝00H，AL＝传输的扇区数，否则，AH＝状态代码，参见功能号01H中的说明

(5)、功能04H

功能描述：检验扇区

入口参数：AH＝04H

AL＝扇区数

CH＝柱面

CL＝扇区

DH＝磁头

DL＝驱动器，00H~7FH：软盘；80H~0FFH：硬盘

ES:BX＝缓冲区的地址

出口参数：CF＝0——操作成功，AH＝00H，AL＝被检验的扇区数，否则，AH＝状态代码，参见功能号01H中的说明

(6)、功能05H

功能描述：格式化磁道

入口参数：AH＝05H

AL＝交替(Interleave)

CH＝柱面

DH＝磁头

DL＝驱动器，00H~7FH：软盘；80H~0FFH：硬盘

ES:BX＝地址域列表的地址

出口参数：CF＝0——操作成功，AH＝00H，否则，AH＝状态代码，参见功能号01H中的说明

(7)、功能06H

功能描述：格式化坏磁道

入口参数：AH＝06H

AL＝交替

CH＝柱面

DH＝磁头

DL＝80H~0FFH：硬盘

ES:BX＝地址域列表的地址

出口参数：CF＝0——操作成功，AH＝00H，否则，AH＝状态代码，参见功能号01H中的说明

(8)、功能07H

功能描述：格式化驱动器

入口参数：AH＝07H

AL＝交替

CH＝柱面

DL＝80H~0FFH：硬盘

出口参数：CF＝0——操作成功，AH＝00H，否则，AH＝状态代码，参见功能号01H中的说明

(9)、功能08H

功能描述：读取驱动器参数

入口参数：AH＝08H

DL＝驱动器，00H~7FH：软盘；80H~0FFH：硬盘

出口参数：CF＝1——操作失败，AH＝状态代码，参见功能号01H中的说明，否则， BL＝01H — 360K

＝02H — 1.2M

＝03H — 720K

＝04H — 1.44M

CH＝柱面数的低8位

CL的位7-6＝柱面数的该2位

CL的位5-0＝扇区数

DH＝磁头数

DL＝驱动器数

ES:DI＝磁盘驱动器参数表地址

(10)、功能09H

功能描述：初始化硬盘参数

入口参数：AH＝09H

DL＝80H~0FFH：硬盘(还有有关参数表问题，在此从略)

出口参数：CF＝0——操作成功，AH＝00H，否则，AH＝状态代码，参见功能号01H中的说明

(11)、功能0AH

功能描述：读长扇区，每个扇区随带四个字节的ECC编码

入口参数：AH＝0AH

AL＝扇区数

CH＝柱面

CL＝扇区

DH＝磁头

DL＝80H~0FFH：硬盘

ES:BX＝缓冲区的地址

出口参数：CF＝0——操作成功，AH＝00H，AL＝传输的扇区数，否则，AH＝状态代码，参见功能号01H中的说明

(12)、功能0BH

功能描述：写长扇区，每个扇区随带四个字节的ECC编码

入口参数：AH＝0BH

AL＝扇区数

CH＝柱面

CL＝扇区

DH＝磁头

DL＝80H~0FFH：硬盘

ES:BX＝缓冲区的地址

出口参数：CF＝0——操作成功，AH＝00H，AL＝传输的扇区数，否则，AH＝状态代码，参见功能号01H中的说明

(13)、功能0CH

功能描述：查寻

入口参数：AH＝0CH

CH＝柱面的低8位

CL(7-6位)＝柱面的高2位

DH＝磁头

DL＝80H~0FFH：硬盘

出口参数：CF＝0——操作成功，AH＝00H，否则，AH＝状态代码，参见功能号01H中的说明

(14)、功能0DH

功能描述：硬盘系统复位

入口参数：AH＝0DH

DL＝80H~0FFH：硬盘

出口参数：CF＝0——操作成功，AH＝00H，否则，AH＝状态代码，参见功能号01H中的说明

(15)、功能0EH

功能描述：读扇区缓冲区

入口参数：AH＝0EH

ES:BX＝缓冲区的地址

出口参数：CF＝0——操作成功，否则，AH＝状态代码，参见功能号01H中的说明

(16)、功能0FH

功能描述：写扇区缓冲区

入口参数：AH＝0FH

ES:BX＝缓冲区的地址

出口参数：CF＝0——操作成功，否则，AH＝状态代码，参见功能号01H中的说明

(17)、功能10H

功能描述：读取驱动器状态

入口参数：AH＝10H

DL＝80H~0FFH：硬盘

出口参数：CF＝0——操作成功，AH＝00H，否则，AH＝状态代码，参见功能号01H中的说明

(18)、功能11H

功能描述：校准驱动器

入口参数：AH＝11H

DL＝80H~0FFH：硬盘

出口参数：CF＝0——操作成功，AH＝00H，否则，AH＝状态代码，参见功能号01H中的说明

(19)、功能12H

功能描述：控制器RAM诊断

入口参数：AH＝12H

出口参数：CF＝0——操作成功，否则，AH＝状态代码，参见功能号01H中的说明

(20)、功能13H

功能描述：控制器驱动诊断

入口参数：AH＝13H

出口参数：CF＝0——操作成功，否则，AH＝状态代码，参见功能号01H中的说明

(21)、功能14H

功能描述：控制器内部诊断

入口参数：AH＝14H

出口参数：CF＝0——操作成功，否则，AH＝状态代码，参见功能号01H中的说明

(22)、功能15H

功能描述：读取磁盘类型

入口参数：AH＝15H

DL＝驱动器，00H~7FH：软盘；80H~0FFH：硬盘

出口参数：CF＝1——操作失败，AH＝状态代码，参见功能号01H中的说明， 否则，AH＝00H — 未安装驱动器

＝01H — 无改变线支持的软盘驱动器

＝02H — 带有改变线支持的软盘驱动器

＝03H — 硬盘，CX:DX＝512字节的扇区数

(23)、功能16H

功能描述：读取磁盘变化状态

入口参数：AH＝16H

DL＝00H~7FH：软盘

出口参数：CF＝0——磁盘未改变，AH＝00H，否则，AH＝06H，参见功能号01H中的说明

(24)、功能17H

功能描述：设置磁盘类型

入口参数：AH＝17H

DL＝00H~7FH：软盘 AL＝00H — 未用

＝01H — 360K在360K驱动器中

＝02H — 360K在1.2M驱动器中

＝03H — 1.2M在1.2M驱动器中

＝04H — 720K在720K驱动器中

出口参数：CF＝0——操作成功，AH＝00H，否则，AH＝状态编码，参见功能号01H中的说明

(25)、功能18H

功能描述：设置格式化媒体类型

入口参数：AH＝18H

CH＝柱面数

CL＝每磁道的扇区数

DL＝00H~7FH：软盘

出口参数：CF＝0——操作成功，AH＝00H，ES:DI＝介质类型参数表地址，否则，AH＝状态编码，参见功能号01H中的说明

(26)、功能19H

功能描述：磁头保护，仅在PS/2中有效，在此从略

(27)、功能1AH

功能描述：格式化ESDI驱动器，仅在PS/2中有效，在此从略

3、串行口服务(Serial Port Service——INT 14H)

00H —初始化通信口03H —读取通信口状态

01H —向通信口输出字符04H —扩充初始化通信口

02H —从通信口读入字符

(1)、功能00H

功能描述：初始化通信口

入口参数：AH＝00H

DX＝初始化通信口号(0＝COM1，1＝COM2，……)

AL＝初始化参数，参数的说明如下： 波特率奇偶位停止位字的位数

76543210

000 = 110X0 = None0 = 1 bit10 = 7 bits

001 = 15001 = Odd1 = 2 bits11 = 8 bits

010 = 30011 = Even

011 = 600

100 = 1200

101 = 2400

110 = 4800

111 = 9600

对于PS/2，可用INT 14H之功能04H和05H来初始化其通信速率大于9600。

出口参数：AH＝通信口状态，各状态位为1时的含义如下： 位7—超时

位6—传递移位寄存器为空

位5—传递保持寄存器为空

位4—发现终止位3—发现帧错误

位2—发现奇偶错

位1—发现越界错

位0—接受数据准备好

AL＝Modem状态

位7—接受单线信号诊断

位6—环指示器

位5—数据发送准备好

位4—清除数据，再发送位3—改变在接受线上的信号诊断

位2—后边界环指示器

位1—改变“数据准备好”状态

位0—改变“清除—发送”状态

(2)、功能01H

功能描述：向通信口输出字符

入口参数：AH＝01H

AL＝字符

DX＝初始化通信口号(0＝COM1，1＝COM2，……)

出口参数：AL的值不变

AH的位7＝0——操作成功，通信口状态，AH的位6~0是其状态位

(3)、功能02H

功能描述：从通信口读入字符

入口参数：AH＝02H

DX＝初始化通信口号(0＝COM1，1＝COM2，……)

出口参数：AL＝接受的字符

AH的位7＝0——操作成功，通信口状态，AH的位6~0是其状态位

(4)、功能03H

功能描述：读取通信口状态

入口参数：AH＝03H

DX＝初始化通信口号(0＝COM1，1＝COM2，……)

出口参数：AH＝通信口状态，AL＝Modem状态，参见功能号00H中的说明

(5)、功能04H

功能描述：扩充初始化通信口，仅在PS/2中有效，在此从略

4、杂项系统服务(Miscellaneous System Service——INT 15H)

00H —开盒式磁带机马达85H —系统请求(SysReq)键

01H —关盒式磁带机马达86H —延迟

02H —读盒式磁带机87H —移动扩展内存块

03H —写盒式磁带机88H —读取扩展内存大小

0FH —格式化ESDI驱动器定期中断89H —进入保护模式

21H —读/写自检(POST)错误记录90H —设备等待

4FH —键盘截听91H —设备加电自检

80H —设备打开C0H —读取系统环境

81H —设备关闭C1H —读取扩展BIOS数据区地址

82H —进程终止C2H —鼠标图形

83H —事件等待C3H —设置WatcHdog超时

84H —读游戏杆C4H —可编程选项选择

(1)、功能00H

功能描述：开盒式磁带机马达

入口参数：AH＝00H

出口参数：CF＝0——操作成功，否则，AH＝状态(86H，若未安装盒式磁带机)

(2)、功能01H

功能描述：关盒式磁带机马达

入口参数：AH＝01H

出口参数：CF＝0——操作成功，否则，AH＝状态(86H，若未安装盒式磁带机)

(3)、功能02H

功能描述：读盒式磁带机

入口参数：AH＝02H

CX＝读入的字节数

ES:BX＝存放数据的缓冲区地址

出口参数：CF＝0——操作成功，DX＝实际读入的字节数，ES:BX指向最后一个字节的后面地址，否则，AH＝状态码，其值含义如下： 01H

—— CRC校验码错80H —— 非法命令

02H —— 位信号混乱86H —— 未安装盒式磁带机

04H —— 无发现数据

(4)、功能03H

功能描述：写盒式磁带机

入口参数：AH＝03H

CX＝要写入的字节数

ES:BX＝已存数据的缓冲区地址

出口参数：CF＝0——操作成功，CX＝00H，ES:BX指向最后一个字节的后面地址，否则，AH＝状态码，其值含义如下： 80H ——

非法命令86H —— 未安装盒式磁带机

(5)、功能0FH

功能描述：格式化ESDI驱动器定期中断，仅在PS/2中有效，在此从略

(6)、功能21H

功能描述：读/写自检(POST)错误记录，仅在PS/2中有效，在此从略

(7)、功能4FH

功能描述：键盘截听，仅在PS/2中有效，在此从略

(8)、功能80H

功能描述：打开设备

入口参数：AH＝80H

BX＝设备号

CX＝进程号

出口参数：CF＝0——操作成功，AH＝00H，否则，AH＝状态码

(9)、功能81H

功能描述：关闭设备

入口参数：AH＝81H

BX＝设备号

CX＝进程号

出口参数：CF＝0——操作成功，AH＝00H，否则，AH＝状态码

(10)、功能82H

功能描述：进程终止

入口参数：AH＝81H

BX＝进程号

出口参数：CF＝0——操作成功，AH＝00H，否则，AH＝状态码

(11)、功能83H

功能描述：事件等待

入口参数：AH＝83H 若需要事件等待，则：AL＝00H

CX:DX＝千分秒

ES:BX＝信号量字节的地址

否则，调用参数为AL＝01H

出口参数：若调用时，AL＝00H，操作成功——CF＝0，否则，CF＝1

(12)、功能84H

功能描述：读游戏杆

入口参数：AH＝84H

DX＝00H——读取开关设置

　＝01H——读取阻力输入

出口参数：CF＝1H——操作失败，否则，

　DX＝00H时，AL＝开关设置(位7～4)

　DX＝01H时，AX、BX、CX和DX分别为A(x)、A(y)、B(x)和B(y)的值

(13)、功能85H

功能描述：系统请求(SysReq)键

入口参数：AH＝85H

AL＝00H——键按下

　＝01H——键放开

出口参数：CF＝0——操作成功，AH＝00H，否则，AH＝状态码

(14)、功能86H

功能描述：延迟

入口参数：AH＝86H

CX:DX＝千分秒

出口参数：CF＝0——操作成功，AH＝00H

(15)、功能87H

功能描述：从常规内存和扩展内存之间移动扩展内存块

入口参数：AH＝87H

CX＝移动的字数

ES:SI＝GDT(Global Descriptor Table)的地址，其结构定义如下： 偏移量存储的信息

00h-0Fh保留，但现全为0

10h-11h段的长度(2CX-1或更大)

12h-14h24位源地址

15h访问权限字节(其值为93h)

16h-17h保留，但现全为0

18h-19h段的长度(2CX-1或更大)

1Ah-1Ch 24位目标源地址

1Dh访问权限字节(其值为93h)

1Eh-2Fh保留，但现全为0

出口参数：CF＝0——操作成功，AH＝00H，否则，AH＝状态码，其含义如下：

　　01H —— RAM奇偶错

　　02H —— 异常中断错

　　03H —— 20号线门地址失败

(16)、功能88H

功能描述：读取扩展内存大小

入口参数：AH＝88H

出口参数：AX＝扩展内存字节数(以K为单位)

(17)、功能89H

功能描述：进入保护模式，CPU从实模式进入保护模式

入口参数：AH＝89H

BH＝IRQ0的中断号

BL＝IRQ8的中断号

ES:SI＝GDT的地址(参见功能号87H)

出口参数：CF＝1——操作失败，AH＝0FFH，否则，AH＝00H，CS、DS、ES和SS都是用户定义的选择器

(18)、功能90H

功能描述：设备等待

入口参数：AH＝90H AL＝驱动器类型，具体的驱动器类型定义如下：

＝00H~7FH——串行再重用设备

＝80H~0BFH——可重入式设备

＝0C0H~0FFH——等待访问设备，没有自检功能

00h — 磁盘

02h — 键盘

80h — 网络

FDh — 软盘马达启动01h — 软盘

03h — 点设备(Pointing Device)

FCh — 硬盘复位

FEh — 打印机

ES:BX＝对驱动器类型80H~0FFH的请求块地址

出口参数：CF＝1——操作失败，否则，AH＝00H

(19)、功能91H

功能描述：设备加电自检

入口参数：AH＝91H

AL＝00H~7FH——串行再重用设备

　＝80H~0BFH——可重入式设备

出口参数：AH＝00H

(20)、功能0C0H

功能描述：读取系统环境

入口参数：AH＝0C0H

出口参数：ES:BX＝配置表地址，配置表的定义如下： 偏移量含义说明

00h-01h表的大小(字节数)

02h系统模型

03h系统子模型

04hBIOS版本号

05h配置标志，其各位为1时的说明如下：

　　位7—DMA通道3使用

　　位6—存在从属8259

　　位5—实时时钟有效

　　位4—键盘截听有效

　　位3—等待外部事件有效

　　位2—扩展BIOS数据区

　　位1—微通道设施

　　位0—保留

06h-09h保留

(21)、功能C1H

功能描述：读取扩展BIOS数据区地址，仅在PS/2中有效，在此从略

(22)、功能C2H

功能描述：鼠标图形，仅在PS/2中有效，在此从略

(23)、功能C3H

功能描述：设置WatcHdog超时，仅在PS/2中有效，在此从略

(24)、功能C4H

功能描述：可编程选项选择，仅在PS/2中有效，在此从略

5、键盘服务(Keyboard Service——INT 16H)

00H、10H —从键盘读入字符03H —设置重复率

01H、11H —读取键盘状态04H —设置键盘点击

02H, 12H —读取键盘标志05H —字符及其扫描码进栈

(1)、功能00H和10H

功能描述：从键盘读入字符

入口参数：AH＝00H——读键盘

＝10H——读扩展键盘，可根据0000:0496H单元的内容判断：扩展键盘是否有效

出口参数：AH＝键盘的扫描码

AL＝字符的ASCII码

(2)、功能01H和11H

功能描述：读取键盘状态

入口参数：AH＝01H——检查普通键盘

＝11H——检查扩展键盘

出口参数：ZF＝1——无字符输入，否则，AH＝键盘的扫描码，AL＝ASCII码。

(3)、功能02H和12H

功能描述：读取键盘标志

入口参数：AH＝02H——普通键盘的移位标志

＝12H——扩展键盘的移位标志

出口参数：AL＝键盘标志(02H和12H都有效)，其各位之值为1时的含义如下： 位7—INS开状态位3—ALT键按下

位6—CAPS LOCK开状态位2—CTRL键按下

位5—NUM LOCK开状态位1—左SHIFT键按下

位4—SCROLL LOCK开状态位0—右SHIFT键按下

AH＝扩展键盘的标志(12H有效)，其各位之值为1时的含义如下：

位7—SysReq键按下位3—右ALT键按下

位6—CAPS LOCK键按下位2—右CTRL键按下

位5—NUM LOCK键按下位1—左ALT键按下

位4—SCROLL键按下位0—左CTRL键按下

(4)、功能03H

功能描述：设置重复率

入口参数：AH＝03H 对于PC/AT和PS/2：AL＝05H

BH＝重复延迟

BL＝重复率

对于PCjr：AL＝00H——装入缺省的速率和延迟

＝01H——增加初始延迟

＝02H——重复频率降低一半

＝03H——增加延迟和降低一半重复频率

＝04H——关闭键盘重复功能

出口参数：无

(5)、功能04H

功能描述：设置键盘点击

入口参数：AH＝04H AL＝00H——关闭键盘点击功能

＝01H——打开键盘点击功能

出口参数：无

(6)、功能05H

功能描述：字符及其扫描码进栈

入口参数：AH＝05H

CH＝字符的描述码

CL＝字符的ASCII码

出口参数：CF＝1——操作成功，AL＝00H，否则，AL＝01H

6、并行口服务(Parallel Port Service——INT 17H)

00H —向打印机输出字符

01H —初始化打印机端口

02H —读取打印机状态

(1)、功能00H

功能描述：向打印机输出字符

入口参数：AH＝00H

AL＝输出的字符

DX＝打印机号(0—LPT1，1—LPT2，2—LPT3，……)

出口参数：AH＝打印机状态。其各位为1时的含义如下： 位7—打印机空闲 位3—I/O错误

位6—打印机响应位2—保留

位5—无纸位1—保留

位4—打印机被选位0—打印机超时

(2)、功能01H

功能描述：初始化打印机端口

入口参数：AH＝01H

DX＝打印机号(0—LPT1，1—LPT2，2—LPT3，……)

出口参数：AH＝打印机状态。各位定义如下功能00H所示

(3)、功能02H

功能描述：读取打印机状态

入口参数：AH＝02H

DX＝打印机号(0—LPT1，1—LPT2，2—LPT3，……)

出口参数：AH＝打印机状态。各位定义如下功能00H所示

7、时钟服务(Clock Service——INT 1AH)

00H —读取时钟“滴答”计数06H —设置闹钟

01H —设置时钟“滴答”计数07H —闹钟复位

02H —读取时间0AH —读取天数计数

03H —设置时间0BH —设置天数计数

04H —读取日期 80H —设置声音源信息

05H —设置日期

(1)、功能00H

功能描述：读取时钟“滴答”计数

入口参数：AH＝00H

出口参数：AL＝00H——未过午夜，否则，表示已过午夜

CX:DX＝时钟“滴答”计数

(2)、功能01H

功能描述：设置时钟“滴答”计数

入口参数：AH＝01H

CX:DX＝时钟“滴答”计数

出口参数：无

(3)、功能02H

功能描述：读取时间

入口参数：AH＝02H

出口参数：CH＝BCD码格式的小时

CL＝BCD码格式的分钟

DH＝BCD码格式的秒

DL＝00H——标准时间，否则，夏令时

CF＝0——时钟在走，否则，时钟停止

(4)、功能03H

功能描述：设置时间

入口参数：AH＝03H

CH＝BCD码格式的小时

CL＝BCD码格式的分钟

DH＝BCD码格式的秒

DL＝00H——标准时间，否则，夏令时

出口参数：无

(5)、功能04H

功能描述：读取日期

入口参数：AH＝04H

出口参数：CH＝BCD码格式的世纪

CL＝BCD码格式的年

DH＝BCD码格式的月

DL＝BCD码格式的日

CF＝0——时钟在走，否则，时钟停止

(6)、功能05H

功能描述：设置日期

入口参数：AH＝05H

CH＝BCD码格式的世纪

CL＝BCD码格式的年

DH＝BCD码格式的月

DL＝BCD码格式的日

出口参数：无

(7)、功能06H

功能描述：设置闹钟

入口参数：AH＝06H

CH＝BCD码格式的小时

CL＝BCD码格式的分钟

DH＝BCD码格式的秒

出口参数：CF＝０——操作成功，否则，闹钟已设置或时钟已停止

(8)、功能07H

功能描述：闹钟复位

入口参数：AH＝07H

出口参数：无

(9)、功能0AH

功能描述：读取天数计数，仅在PS/2有效，在此从略

(10)、功能0BH

功能描述：设置天数计数，仅在PS/2有效，在此从略

(11)、功能80H

功能描述：设置声音源信息

入口参数：AH＝80H

AL＝声音源

＝00H——8253可编程计时器，通道2

＝01H——盒式磁带输入

＝02H——I/O通道上的”Audio In”

＝03H——声音产生芯片

出口参数：无

8、直接系统服务(Direct System Service)

INT 00H —“0”作除数

INT 01H —单步中断

INT 02H —非屏蔽中断(NMI)

INT 03H —断点中断

INT 04H —算术溢出错误

INT 05H —打印屏幕和BOUND越界

INT 06H —非法指令错误

INT 07H —处理器扩展无效

INT 08H —时钟中断

INT 09H —键盘输入

INT 0BH —通信口(COM2:)

INT 0CH —通信口(COM1:)

INT 0EH —磁盘驱动器输入/输出

INT 11H —读取设备配置

INT 12H —读取常规内存大小(返回值AX为内存容量，以K为单位)

INT 18H —ROM BASIC

INT 19H —重启动系统

INT 1BH —CTRL+BREAK处理程序

INT 1CH —用户时钟服务

INT 1DH —指向显示器参数表指针

INT 1EH —指向磁盘驱动器参数表指针

INT 1FH —指向图形字符模式表指针

## IO端口地址分配表

端口地址范围                     分配说明  
0x000-0x01f                       8237A DMA控制器1  
0x020-0x03f                       8259A 可编程中断控制器1  
0x040-0x05f                       8253/8254|A 定时计数器  
0x060-0x06f                       8042键盘控制器  
0x070-0x07f                       访问CMOS RAM/实时时钟RTC（Real Time Clock)端口  
0x080-0x09f                       DMA页面寄存器访问端口  
0x0a0-0x0bf                       8259 可编程中断控制器2  
0x0c0-0x0df                       8237A DMA控制器2  
0x0f0-0x0ff                         协处理器访问端口  
0x170-0x177                       IDE硬盘控制器1  
0x1f0-0x1f7                        IDE硬盘控制器2  
0x278-ox27f                       并行打印机端口2  
0x2f8-0x2ff                         串行控制器2  
0x378-0x38f                       并行打印机端口1   
0x3b0-0x3bf                       单色MDA显示控制器  
0x3c0-0x3cf                        彩色CGA显示控制器  
0x3d0-0x3df                       彩色EGA/VGA显示控制器  
0x3f0-0x3f7                        软盘控制器  
0x3f8-0x3ff                        串行控制器1

000-00F 8237 DMA controller:  
  
000 Channel 0 address register  
001 Channel 0 word count  
002 Channel 1 address register  
003 Channel 1 word count  
004 Channel 2 address register  
005 Channel 2 word count  
006 Channel 3 address register  
007 Channel 3 word count  
008 Status/command register  
009 Request register  
00A Mask register  
00B Mode register  
00C Clear MSB/LSB flip flop  
00D Master clear temp register  
00E Clear mask register  
00F Multiple mask register

010-01F 8237 DMA Controller (PS2 model 60 & 80), reserved (AT)  
  
020-02F 8259A Master Programmable Interrupt Controller:  
  
020 8259 Command port (see 8259)  
021 8259 Interrupt mask register (see 8259)  
  
030-03F 8259A Slave Programmable Interrupt Controller (AT,PS2)  
  
040-05F 8253 or 8254 Programmable Interval Timer:  
  
040 8253 channel 0, counter divisor  
041 8253 channel 1, RAM refresh counter  
042 8253 channel 2, Cassette and speaker functions  
043 8253 mode control (see 8253)  
044 8254 PS/2 extended timer  
047 8254 Channel 3 control byte  
  
060-067 8255 Programmalbe Peripheral Interface (PC,XT,PCjr):  
  
060 8255 Port A keyboard input/output buffer (output PCjr)  
061 8255 Port B output  
062 8255 Port C input  
063 8255 Command/Mode control register  
  
060-06f 8042 Keyboard Controller (AT,PS2):  
  
060 8042 Keyboard input/output buffer register  
061 8042 system control port (for compatability with 8255)  
064 8042 Keyboard command/status register  
  
070 CMOS RAM/RTC, also NMI enable/disable (AT,PS2, see RTC)  
071 CMOS RAM data (AT,PS2)  
  
080 Manufacturer checkpoint port  
080-090 DMA Page Registers:  
  
081 High order 4 bits of DMA channel 2 address  
082 High order 4 bits of DMA channel 3 address  
083 High order 4 bits of DMA channel 1 address  
  
090-097 POS/Programmable Option Select (PS2):  
  
090 Central arbitration control Port  
091 Card selection feedback  
092 System control and status register  
094 System board enable/setup register  
095 Reserved  
096 Adapter enable/setup register  
097 Reserved  
  
0A0 NMI Mask Register (PC,XT) (write 80h to enable NMI, 00h disable)  
0A0-0BF Second 8259 Programmalbe Interrupt Controller (AT,PS2):  
  
0A0 Second 8259 Command port (see 8259)  
0A1 Second 8259 Interrupt mask register (see 8259)  
  
0C0 TI SN76496 Programmable Tone/Noise Generator (PCjr)  
0C0-0DF 8237 DMA Controller 2 (AT):  
  
0C2 DMA channel 3 selector (see ports 6 & 82)  
  
0E0-0EF Reserved  
  
0F0-0FF Math Coprocessor (AT,PS2)  
0F0-0F5 PCjr Disk Controller:  
  
0F0 Disk Controller  
0F2 Disk Controller control port  
0F4 Disk Controller status register  
0F5 Disk Controller data port  
  
0F8-0FF Reserved for future microprocessor extensions  
  
100-10F POS Programmable Option Select (PS2):  
  
100 POS Register 0, Adapter ID byte (LSB)  
101 POS Register 1, Adapter ID byte (MSB)  
102 POS Register 2, Option select data byte 1  
        Bit 0 is card enable (CDEN)  
103 POS Register 3, Option select data byte 2  
104 POS Register 4, Option select data byte 3  
105 POS Register 5, Option select data byte 4  
        Bit 7 is (-CHCK)  
        Bit 6 is reserved  
106 POS Register 6, subaddress extension (LSB)  
107 POS Register 7, subaddress extension (MSB)  
  
110-1EF System I/O channel  
  
170-17F Fixed disk 1 (AT):  
  
170 disk 1 data  
171 disk 1 error  
172 disk 1 sector count  
173 disk 1 sector number  
174 disk 1 cylinder low  
175 disk 1 cylinder high  
176 disk 1 drive/head  
177 disk 1 status  
  
1F0-1FF Fixed disk 0 (AT):  
  
1F0 disk 0 data  
1F1 disk 0 error  
1F2 disk 0 sector count  
1F3 disk 0 sector number  
1F4 disk 0 cylinder low  
1F5 disk 0 cylinder high  
1F6 disk 0 drive/head  
1F7 disk 0 status  
  
200-20F Game Adapter

210-217 Expansion Card Ports (XT):  
  
210 Write: latch expansion bus data  
    read: verify expansion bus data  
211 Write: clear wait,test latch  
    Read: MSB of data address  
212 Read: LSB of data address  
213 Write: 0=enable, 1=/disable expansion unit  
214-215 Receiver Card Ports  
214 write: latch data, read: data  
215 read: MSB of address, next read: LSB of address  
  
21F Reserved  
  
220-26F Reserved for I/O channel

270-27F Third parallel port (see PARALLEL PORT):  
  
278 data port  
279 status port  
27A control port  
  
280-2AF Reserved for I/O channel

2A2-2A3 MSM58321RS clock

2B0-2DF Alternate EGA, or 3270 PC video (XT, AT)

2E0 Alternate EGA/VGA

E1 GPIB Adapter (AT)

2E2-2E3 Data acquisition adapter (AT)

2E8-2EF COM4 non PS2 UART (Reserved by IBM) (see UART)

2F0-2F7 Reserved

2F8-2FF COM2 Second Asynchronous Adapter  
Primary Asynchronous Adapter for PCjr

300-31F Prototype Experimentation Card  
Periscope hardware debugger

320-32F Hard Disk Controller (XT):  
  
320 Read from/Write to controller  
321 Read: Controller Status, Write: controller reset  
322 Write: generate controller select pulse  
323 Write: Pattern to DMA and interrupt mask register  
324 disk attention/status  
  
330-33F Reserved for XT/370  
  
340-35F Reserved for I/O channel  
  
360-36F PC Network  
  
370-377 Floppy disk controller (except PCjr):  
  
372 Diskette digital output  
374 Diskette controller status  
375 Diskette controller data  
376 Diskette controller data  
377 Diskette digital input  
  
378-37F Second Parallel Printer (see PARALLEL PORT):  
  
378 data port  
379 status port  
37A control port  
  
380-38F Secondary Binary Synchronous Data Link Control (SDLC) adapter:  
  
380 On board 8255 port A, internal/external sense  
381 On board 8255 port B, external modem interface  
382 On board 8255 port C, internal control and gating  
383 On board 8255 mode register  
384 On board 8253 channel square wave generator  
385 On board 8253 channel 1 inactivity time-out  
386 On board 8253 channel 2 inactivity time-out  
387 On board 8253 mode register  
388 On board 8273 read: status; Write: Command  
389 On board 8273 write: parameter; read: response  
38A On board 8273 transmit interrupt status  
38B On board 8273 receiver interrupt status  
38C On board 8273 data  
  
390-39F Cluster Adapter

3A0-3AF Primary Binary Synchronous Data Link Control (SDLC) adapter:  
  
3A0 On board 8255 port A, internal/external sense  
3A1 On board 8255 port B, external modem interface  
3A2 On board 8255 port C, internal control and gating  
3A3 On board 8255 mode register  
3A4 On board 8253 counter 0 unused  
3A5 On board 8253 counter 1 inactivity time-outs  
3A6 On board 8253 counter 2 inactivity time-outs  
3A7 On board 8253 mode register  
3A8 On board 8251 data  
3A9 On board 8251 command/mode/status register  
  
3B0-3BF Monochrome Display Adapter (write only, see 6845):  
  
3B0 port address decodes to 3B4  
3B1 port address decodes to 3B5  
3B2 port address decodes to 3B4  
3B3 port address decodes to 3B5  
3B4 6845 index register, selects which register [0-11h]  
    is to be accessed through port 3B5  
3B5 6845 data register [0-11h] selected by port 3B4,  
    registers 0C-0F may be read. If a read occurs without  
    the adapter installed, FFh is returned. (see 6845)  
3B6 port address decodes to 3B4  
3B7 port address decodes to 3B5  
3B8 6845 Mode control register  
3B9 reserved for color select register on color adapter  
3BA status register (read only)  
3BB reserved for light pen strobe reset  
  
3BC-3BF Primary Parallel Printer Adapter (see PARALLEL PORT):  
  
3BC parallel 1, data port  
3BD parallel 1, status port  
3BE parallel 1, control port  
  
3C0-3CF EGA/VGA:  
  
3C0 VGA attribute and sequencer register  
3C1 Other video attributes  
3C2 EGA, VGA, CGA input status 0  
3C3 Video subsystem enable  
3C4 CGA, EGA, VGA sequencer index  
3C5 CGA, EGA, VGA sequencer  
3C6 VGA video DAC PEL mask  
3C7 VGA video DAC state  
3C8 VGA video DAC PEL address  
3C9 VGA video DAC  
3CA VGA graphics 2 position  
3CC VGA graphics 1 position  
3CD VGA feature control  
3CE VGA graphics index  
3CF Other VGA graphics  
  
3D0-3DF Color Graphics Monitor Adapter (ports 3D0-3DB are write only, see 6845):  
  
3D0 port address decodes to 3D4  
3D1 port address decodes to 3D5  
3D2 port address decodes to 3D4  
3D3 port address decodes to 3D5  
3D4 6845 index register, selects which register [0-11h]  
    is to be accessed through port 3D5  
3D5 6845 data register [0-11h] selected by port 3D4,  
    registers 0C-0F may be read. If a read occurs without  
    the adapter installed, FFh is returned. (see 6845)  
3D6 port address decodes to 3D4  
3D7 port address decodes to 3D5  
3D8 6845 Mode control register (CGA, EGA, VGA, except PCjr)  
3D9 color select palette register (CGA, EGA, VGA, see 6845)  
3DA status register (read only, see 6845, PCjr VGA access)  
3DB Clear light pen latch (any write)  
3DC Preset Light pen latch  
3DF CRT/CPU page register (PCjr only)  
  
3E8-3EF COM3 non PS2 UART (Reserved by IBM) (see UART)

3F0-3F7 Floppy disk controller (except PCjr):  
  
3F0 Diskette controller status A  
3F1 Diskette controller status B  
3F2 controller control port  
3F4 controller status register  
3F5 data register (write 1-9 byte command, see INT 13)  
3F6 Diskette controller data  
3F7 Diskette digital input  
  
3F8-3FF COM1 Primary Asynchronous Adapter (see UART)3F0 Diskette controller status A:  
  
3220-3227 PS2 COM3  
3228-322F PS2 COM4  
4220-4227 PS2 COM5  
4228-422F PS2 COM6  
5220-5227 PS2 COM7  
5228-522F PS2 COM8