

使用开源软件

自己动手写操作系统

WRITE YOUR OWN OS WITH
FREE AND OPEN SOURCE SOFTWARE

Alpha Edition



杨文博 著

生成时间： 2008 年 11 月 29 日 15:50 Revision 2

版权声明

本文遵从[署名-非商业性使用-相同方式共享 2.5 中国大陆创作共享协议](#)。

您可以自由：

- 复制、发行、展览、表演、放映、广播或通过信息网络传播本作品。

惟须遵守下列条件：

- 署名. 您必须按照作者或者许可人指定的方式对作品进行署名。
- 非商业性使用. 您不得将本作品用于商业目的。
- 相同方式共享. 如果您改变、转换本作品或者以本作品为基础进行创作，您只能采用与本协议相同的许可协议发布基于本作品的演绎作品。
- 对任何再使用或者发行，您都必须向他人清楚地展示本作品使用的许可协议条款。
- 如果得到著作权人的许可，您可以不受任何这些条件的限制。
- Nothing in this license impairs or restricts the author's moral rights.

您的合理使用以及其他权利不受上述规定的影响。

这是一份普通人可以理解的[法律文本（许可协议全文）](#)的概要。

相关链接

您可以在本书[官方网站](#)：<http://share.solrex.cn/WriteOS/> 下载到本书的最新版本和附带的全部程序源代码。

由于此版本非最终发布版，如果您对本书感兴趣，请关注作者在其[博客](#) (<http://blog.solrex.cn>) 上发布的更新公告。如果您发现本书的错误或者有好的建议，请到<http://code.google.com/p/writeos/issues/list> 检视并报告您的发现，对此作者将非常感谢。

写在前面的话

本书起源于中国电子工业出版社出版的一本书：《自己动手写操作系统》（于渊著）。我对《自己动手写操作系统》这本书中使用商业软件做为演示平台比较惊讶，因为不是每个人都买得起正版软件的，尤其是穷学生。我想《自》所面向的主要受众也应该是学生，那么一本介绍只有商业软件才能实现的编程技巧的书将会逼着穷学生去使用盗版，这是非常罪恶的行为^②。

由于本人是一个 Linux 用户，一个开源软件的拥护者，所以就试着使用开源软件实现这本书中的所有 demo，并在自己的博客上进行推广。后来我觉得，为什么我不能自己写本书呢？这样我就能插入漂亮的插图，写更详尽的介绍而不怕篇幅过长，更容易让读者接受也更容易传播，所以我就开始写这本《使用开源软件-自己动手写操作系统》。

定下写一本书的目标毕竟不像写一篇博客，我将尽量详尽的介绍我使用的方法和过程，以图能让不同技术背景的读者都能通畅地完成阅读。但是自己写并且排版一本书不是很轻松的事情，需要耗费大量时间，所以我只能抽空一点一点的将这本书堆砌起来，这也是您之所以在本书封面看到本书版本号的原因^③。

本书的最终目标是成为一本大学“计算机操作系统”课程的参考工具书，为学生提供一个 step by step 的引导去实现一个操作系统。这不是一个容易实现的目标，因为我本人现在并不自信有那个实力了解操作系统的方方面面。但是我想，立志百里行九十总好过于踟蹰不前。

《自己动手写操作系统》一书开了个好头，所以在前面部分，我将主要讨论使用开源软件实现《自》的 demo。如果您有《自》这本书，参考阅读效果会更好，不过我将尽我所能 在本书中给出清楚的讲解，尽量使您免于去参考《自》一书。

出于开放性和易编辑性考虑，本书采用 LATEX 排版，在成书前期由于专注于版面，代码比较杂乱，可读性不强，暂不开放本书 TEX 源代码下载。但您可以通过 SVN check out 所有本书相关的源代码和图片，具体方法请参见电子书主页。

如果您在阅读过程中有什么问题，发现书中的错误，或者好的建议，欢迎您使用我留下的联系方式与我联系，本人将非常感谢。

杨文博

个人主页：<http://solrex.cn>

个人博客：<http://blog.solrex.cn>

2008 年 1 月 9 日

更新历史

Rev. 1 确定书本排版样式，添加第一章，第二章。

Rev. 2 添加第三章保护模式。

目录

写在前面的话	i
序言	ix
第一章 计算机启动	1
1.1 计算机启动过程	1
1.2 磁盘抽象物理结构	2
1.2.1 硬盘	3
1.2.2 软盘	4
1.2.3 启动扇区	5
1.3 使用虚拟机	5
1.3.1 VirtualBox	6
1.3.2 Bochs	14
1.4 使用软盘镜像	14
1.4.1 制作软盘镜像	14
1.4.2 用软盘镜像启动虚拟机	14
第二章 最小的“操作系统”	19
2.1 Hello OS world!	19
2.1.1 Intel 汇编转化为 AT&T(GAS) 汇编	20
2.1.2 用连接脚本控制地址空间	20
2.1.3 用 Makefile 编译连接	22
2.1.4 用虚拟机加载执行 boot.img	24
2.2 FAT 文件系统	25
2.2.1 FAT12 文件系统	25
2.2.2 启动扇区与 BPB	26
2.2.3 FAT12 数据结构	28
2.2.4 FAT12 根目录结构	29

2.3 让启动扇区加载引导文件	30
2.3.1 一个最简单的 loader	30
2.3.2 读取软盘扇区的 BIOS 13h 号中断	30
2.3.3 搜索 loader.bin	32
2.3.4 加载 loader 入内存	35
2.3.5 向 loader 转交控制权	37
2.3.6 生成镜像并测试	38
第三章 进入保护模式	41
3.1 实模式和保护模式	41
3.1.1 一段历史	42
3.1.2 实模式	42
3.1.3 保护模式	42
3.1.4 实模式和保护模式的寻址模式	42
3.2 与保护模式初次会面	43
3.2.1 GDT 数据结构	44
3.2.2 保护模式下的 demo	46
3.2.3 加载 GDT	47
3.2.4 进入保护模式	48
3.2.5 特别的混合跳转指令	49
3.2.6 生成镜像并测试	52
3.3 段式存储	52
3.3.1 LDT 数据结构	53
3.3.2 段描述符属性	53
3.3.3 使用 LDT	55
3.3.4 生成镜像并测试	63
3.3.5 段式存储总结	64
3.4 特权级	64
3.4.1 不合法的访问请求示例	65
3.4.2 控制权转移的特权级检查	67
3.4.3 使用调用门转移	67
3.4.4 栈切换和 TSS	74
3.5 页式存储	83
3.5.1 分页机制	84
3.5.2 启动分页机制	85
3.5.3 修正内存映射的错误	89
3.5.4 体验虚拟内存	97
3.6 结语	104
第四章 中断	105

插图

1.1 硬盘	3
1.2 硬盘的抽象物理结构	3
1.3 软盘	4
1.4 启动扇区加载示意图	5
1.5 VirtualBox 个人使用协议	6
1.6 同意 VirtualBox 个人使用协议	7
1.7 VirtualBox 用户注册对话框	7
1.8 VirtualBox 主界面	8
1.9 新建一个虚拟机	8
1.10 设置虚拟机名字和操作系统类型	9
1.11 设置虚拟机内存容量	9
1.12 设置虚拟机硬盘镜像	10
1.13 新建一块虚拟硬盘	10
1.14 设置虚拟硬盘类型	11
1.15 设置虚拟硬盘镜像名字和容量	11
1.16 虚拟硬盘信息	12
1.17 使用新建的虚拟硬盘	12
1.18 虚拟机信息	13
1.19 回到 VirtualBox 主界面	13
1.20 虚拟机设置界面	14
1.21 虚拟机软盘设置	15
1.22 选择软盘镜像	15
1.23 选择启动软盘镜像	16
1.24 确认启动镜像软盘文件信息	16
1.25 查看虚拟机设置信息	17
1.26 自动键盘捕获警告信息	17
1.27 虚拟机运行时	18

2.1 《自》第一个实例代码 boot.asm	20
2.2 boot.S(chapter2/1/boot.S)	20
2.3 boot.S 的连接脚本(chapter2/1/solrex_x86.ld)	21
2.4 《自》代码 1-2(chapter2/1/boot.asm)	22
2.5 boot.S 的 Makefile(chapter2/1/Makefile)	22
2.6 使用 hexedit 打开 boot.img	23
2.7 使用 kde 图形界面工具 khexedit 打开 boot.img	24
2.8 选择启动软盘镜像 boot.img	24
2.9 虚拟机启动后打印出红色的“Hello OS world!”	25
2.10 生成启动扇区头的汇编代码(节自 chapter2/2/boot.S)	27
2.11 FAT 文件系统存储结构图	28
2.12 一个最简单的 loader(chapter2/2/loader.S)	30
2.13 一个最简单的 loader(chapter2/2/solrex_x86_dos.ld)	30
2.14 读取软盘扇区的函数(节自 chapter2/2/boot.S)	32
2.15 搜索 loader.bin 的代码片段(节自 chapter2/2/boot.S)	34
2.16 搜索 loader.bin 使用的变量定义(节自 chapter2/2/boot.S)	34
2.17 打印字符串函数 DispStr(节自 chapter2/2/boot.S)	35
2.18 寻找 FAT 项的函数 GetFATEntry(节自 chapter2/2/boot.S)	36
2.19 加载 loader.bin 的代码(节自 chapter2/2/boot.S)	37
2.20 跳转到 loader 执行(节自 chapter2/2/boot.S)	38
2.21 用 Makefile 编译(节自 chapter2/2/Makefile)	38
2.22 拷贝 LOADER.BIN 入 boot.img(节自 chapter2/2/Makefile)	39
2.23 没有装入 LOADER.BIN 的软盘启动	40
2.24 装入了 LOADER.BIN 以后再启动	40
3.1 实模式与保护模式寻址模型比较	43
3.2 段描述符	44
3.3 自动生成段描述符的宏定义(节自 chapter3/1/pm.h)	45
3.4 自动生成段描述符的宏使用示例(节自 chapter3/1/loader.S)	45
3.5 预先设置的段属性(节自 chapter3/1/pm.h)	46
3.6 第一个在保护模式下运行的 demo(节自 chapter3/1/loader.S)	47
3.7 加载 GDT(节自 chapter3/1/loader.S)	48
3.8 进入保护模式(节自 chapter3/1/loader.S)	49
3.9 混合字长跳转指令(节自 chapter3/1/loader.S)	50
3.10 chapter3/1/loader.S	52
3.11 第一次进入保护模式	52
3.12 段选择子数据结构	53

3.13 32 位全局数据段和堆栈段, 以及对应的 GDT 结构(节自chapter3/2/loader.S)	56
3.14 32 位代码段, 以及对应的 LDT 结构(节自chapter3/2/loader.S)	57
3.15 自动初始化段描述符的宏代码(节自chapter3/2/pm.h)	57
3.16 在实模式代码段中初始化所有段描述符(节自chapter3/2/loader.S)	58
3.17 在保护模式代码段中加载 LDT 并跳转执行 LDT 代码段(节自chapter3/2/loader.S)	59
3.18 chapter3/2/loader.S	63
3.19 第一次进入保护模式	63
3.20 虚拟机出现异常, 黑屏	66
3.21 虚拟退出后 VBox 主窗口显示 Abort	66
3.22 调用门描述符	68
3.23 添加调用门的目标段(节自chapter3/3/loader.S)	69
3.24 汇编宏 Gate 定义(节自chapter3/3/pm.h)	69
3.25 设置调用门描述符及选择子(节自chapter3/3/loader.S)	69
3.26 调用门选择子(节自chapter3/3/loader.S)	70
3.27 使用调用门进行简单的控制权转移	70
3.28 要运行在 ring 3 下的代码段(节自chapter3/4/loader.S)	72
3.29 为 ring 3 代码段准备的新栈(节自chapter3/4/loader.S)	72
3.30 为 ring 3 代码段和堆栈段添加的描述符和选择子(节自chapter3/4/loader.S)	72
3.31 初始化 ring 3 代码段和堆栈段描述符的代码(节自chapter3/4/loader.S)	73
3.32 hack RET 指令进行实际的跳转	73
3.33 hack RET 实现从高特权级到低特权级的跳转	73
3.34 32 位 TSS 数据结构	75
3.35 跨特权级调用时的栈切换	76
3.36 TSS 段内容及其描述符和选择子和初始化代码(节自chapter3/5/loader.S)	77
3.37 加载 TSS 段选择子到 TR 寄存器(节自chapter3/5/loader.S)	78
3.38 跨特权级的调用门转移	78
3.39 chapter3/5/loader.S	83
3.40 线性地址转换 (4KB 页)	84
3.41 邮件地址转换	85
3.42 PDE 和 PTE 的数据结构 (4KB 页)	86
3.43 添加保存页目录和页表的段(节自chapter3/6/loader.S)	87
3.44 为分页机制添加的新属性(节自chapter3/6/pm.h)	88
3.45 初始化页目录和页表, 并打开分页机制的函数(节自chapter3/6/loader.S)	88
3.46 进入保护模式后马上打开分页机制(节自chapter3/6/loader.S)	89
3.47 用来储存内存分布信息的数据段(节自chapter3/7/loader.S)	91
3.48 用中断 INT 15h 得到地址分布数据(节自chapter3/7/loader.S)	92

3.49 将地址分布信息打印到屏幕上(节自chapter3/7/loader.S)	93
3.50 chapter3/7/lib.h	95
3.51 根据可用内存大小调整内存映射范围(节自chapter3/7/loader.S)	96
3.52 调用显示内存范围和开启分页机制的函数(节自chapter3/7/loader.S)	96
3.53 修正内存映射的错误	97
3.54 两个打印自身信息的函数 Foo 和 Bar (节自chapter3/8/loader.S)	98
3.55 4KB 对齐的物理地址(节自chapter3/8/loader.S)	99
3.56 设置两个页表并开启分页机制(节自chapter3/8/loader.S)	100
3.57 添加的新段和变量(节自chapter3/8/loader.S)	101
3.58 拷贝函数、切换页表并调用同一线性地址的示例函数(节自chapter3/8/loader.S)	102
3.59 MemCpy 函数定义(节自chapter3/8/lib.h)	103
3.60 体验虚拟内存	103

序言

这里应该是各个章节的摘要和版式简介，不过因为写摘要向来是件让人心烦的事情，所以我准备把它放在最后写^④。



CHAPTER 1

计算机启动

每一个计算机软件都是由一系列的可执行文件组成的，可执行文件的内容是可以被机器识别的二进制指令和数据。一般可执行文件的运行是在操作系统的照看下加载进内存并运行的，由操作系统给它分配资源和处理器时间，并确定它的执行方式。操作系统也是由可执行文件组成的，但是操作系统的启动方式和一般应用软件是不同的，这也就是它叫做“操作系统”的原因^②。

没有操作系统的机器，一般情况下被我们称为“裸机”，意思就是只有硬件，什么都干不了。但是一个机器怎么知道自己是不是裸机呢？它总要有方式去判断机器上安装没有安装操作系统吧。下面我们就简单介绍一下计算机启动的过程：

1.1 计算机启动过程

计算机启动过程一般是指计算机从点亮到加载操作系统的一个过程。对于 IBM 兼容机（个人电脑）来讲，这个过程大致是这样的：

1. **加电** 电源开关被按下时，机器就开始供电，主板的控制芯片组会向 CPU（Central Processing Unit，中央处理器）发出并保持一个 RESET（重置）信号，让 CPU 恢复到初始状态。当芯片组检测到电源已经开始稳定供电时就会撤去 RESET 信号（松开台式机的重启键是一样的效果），这时 CPU 就从 0xfffff0 处开始执行指令。这个地址在系统 BIOS（Basic Input/Output System，基本输入输出系统）的地址范围内，大部分系统 BIOS 厂商放在这里的都只是一条跳转指令，跳到系统 BIOS 真正的启动代码处。
2. **自检** 系统 BIOS 的启动代码首先要做的事情就是进行 POST（Power-On Self Test，加电后自检），POST 的主要任务是检测系统中一些关键设备是否存在和能否正常工作，例如内存和显卡等。由于 POST 是最早进行的检测过程，此时显卡还没有初始化，如果系统 BIOS 在 POST 的过程中发现了一些致命错误，例如没有找到内存或者内存有问题（此时只会检查 640K 常规内存），那么系统 BIOS 就会直接控制喇叭发声来报告错误，声音的长短和次数代表了错误的类型。
3. **初始化设备** 接下来系统 BIOS 将查找显卡的 BIOS，存放显卡 BIOS 的 ROM 芯片的起始地址通常设在 0xC0000 处，系统 BIOS 在这个地方找到显卡 BIOS 之后就调用它的初始化代码，由显卡

BIOS 来初始化显卡，此时多数显卡都会在屏幕上显示出一些初始化信息，介绍生产厂商、图形芯片类型等内容。系统 BIOS 接着会查找其它设备的 BIOS 程序，找到之后同样要调用这些 BIOS 内部的初始化代码来初始化相关的设备。

4. **测试设备** 查找完所有其它设备的 BIOS 之后，系统 BIOS 将显示出它自己的启动画面，其中包括有系统 BIOS 的类型、序列号和版本号等内容。接着系统 BIOS 将检测和显示 CPU 的类型和工作频率，然后开始测试所有的 RAM（Random Access Memory，随机访问存储器），并同时在屏幕上显示内存测试的进度。内存测试通过之后，系统 BIOS 将开始检测系统中安装的一些标准硬件设备，包括硬盘、光驱、串口、并口、软驱等，另外绝大多数较新版本的系统 BIOS 在这一过程中还要自动检测和设置内存的定时参数、硬盘参数和访问模式等。标准设备检测完毕后，系统 BIOS 内部的支持即插即用的代码将开始检测和配置系统中安装的即插即用设备，每找到一个设备之后，系统 BIOS 都会在屏幕上显示出设备的名称和型号等信息，同时为该设备分配中断（INT）、DMA（Direct Memory Access，直接存储器存取）通道和 I/O（Input/Output，输入输出）端口等资源。
5. **更新 ESCD** 所有硬件都检测配置完毕后，多数系统 BIOS 会重新清屏并在屏幕上方显示出一个表格，其中概略地列出了系统中安装的各种标准硬件设备，以及它们使用的资源和一些相关工作参数。接下来系统 BIOS 将更新 ESCD（Extended System Configuration Data，扩展系统配置数据）。ESCD 是系统 BIOS 用来与操作系统交换硬件配置信息的一种手段，这些数据被存放在 CMOS（Complementary Metal Oxide Semiconductor，互补金属氧化物半导体）之中。
6. **启动操作系统** ESCD 更新完毕后，系统 BIOS 的启动代码将进行它的最后一项工作，即根据用户指定的启动顺序从软盘、硬盘或光驱启动操作系统。以 Windows XP 为例，系统 BIOS 将启动盘（一般是主硬盘）的第一个扇区（Boot Sector，引导扇区）读入到内存的 0x7c00 处，并检查 0x7dfe 地址的内存，如果其内容是 0xaa55，跳转到 0x7c00 处执行 MBR（Master Boot Record，主引导记录），MBR 接着从分区表（Partition Table）中找到第一个活动分区（Active Partition，一般是 C 盘分区），然后按照类似方式读取并执行这个活动分区的引导扇区（Partition Boot Sector），而引导扇区将负责读取并执行 NTLDR（NT LoaDeR，Windows NT 的加载程序），然后主动权就移交给了 Windows。

从以上介绍中我们可以看到，在第 6 步之前，电脑的启动过程完全依仗于系统 BIOS，这个程序一般是厂商写就固化在主板上的。我们所需要做的，就是第 6 步之后的内容，即：

如何写一个操作系统并把它加载到内存？

1.2 磁盘抽象物理结构

由于操作系统的启动涉及到硬件地址写入和磁盘文件寻找，为了更好理解内存地址和文件存储的相关知识，我们先来了解一下磁盘的结构。

1.2.1 硬盘



Fig 1.1: 硬盘

图 1.1 所示就是硬盘（如非特指，我们这里的“硬盘”一般指代磁介质非固态硬盘）的外观图。其中左边是硬盘盒拆开后盘片、磁头和内部机械结构的透视图，右边是普通台式机硬盘的外观图。现在的硬盘容量较以前已经有大幅度增加，一般笔记本电脑硬盘容量已经在 120G 以上，台式机硬盘容量一般也达到了 160G 大小。一般情况下，硬盘都是由坚硬金属材料（或者玻璃等）制成的涂以磁性介质的盘片构成的，一般有层叠的多片，每个盘片都有两个面，两面都可以记录信息。

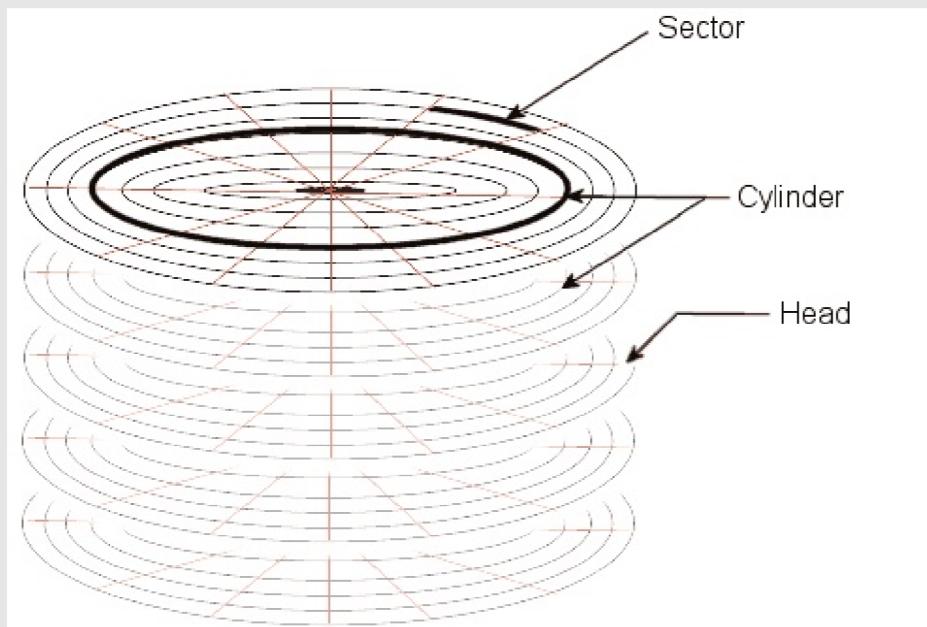


Fig 1.2: 硬盘的抽象物理结构

图 1.2 为硬盘的抽象物理结构，需要注意的是这并不是硬盘真正的物理构造，所以这里我们称其为“抽象”物理结构。因此我们下面讨论的也不是真正的硬盘技术实现，仅仅就硬盘（以及软盘等类似磁介质存储器）存储结构以程序员易于理解的角度进行简单的介绍。

如图 1.2 所示，硬盘是由很多盘片组成的，那些上下有分割的圆盘就表示一个个盘片。每个盘片被分成许多扇形的区域，每个区域叫一个扇区，通常每个扇区存储 512 字节（FAT 文件格式），盘片表面上以盘片中心为圆心，不同半径的同心圆称为磁道。硬盘中，不同盘片相同半径的磁道所组成的圆柱称为柱面。磁道与柱面都是表示不同半径的圆，在许多场合，磁道和柱面可以互换使用。每个磁盘有两

个面，每个面都有一个磁头，习惯用磁头号来区分。扇区，磁道（或柱面）和磁头数构成了硬盘结构的基本参数，使用这些参数可以得到硬盘的容量，其计算公式为：

$$\text{存储容量} = \text{磁头数} \times \text{磁道（柱面）数} \times \text{每磁道扇区数} \times \text{每扇区字节数}$$

要点：

- 硬盘有数个盘片，每盘片两个面，每面一个磁头。
- 盘片被划分为多个扇形区域即扇区。
- 同一盘片不同半径的同心圆为磁道。
- 不同盘片相同半径构成的圆柱面即柱面。
- 公式：存储容量 = 磁头数 × 磁道（柱面）数 × 每道扇区数 × 每扇区字节数。
- 信息记录可表示为：× × 磁道（柱面），× × 磁头，× × 扇区。

1.2.2 软盘

由于我们在本书中主要使用软盘作为系统启动盘，所以下面对应于硬盘介绍一下软盘的相关知识。

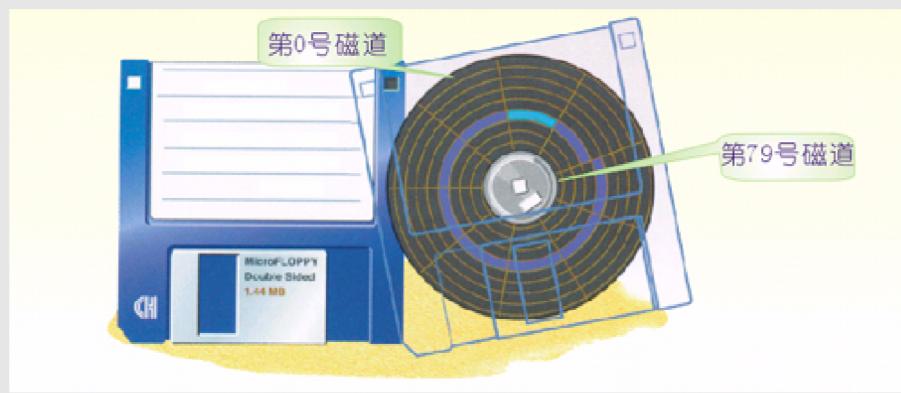


Fig 1.3: 软盘

现在通常能看到的软盘主要是 3.5 英寸软盘，3.5 英寸指的是其内部磁介质盘片的直径。从存储结构上来讲，软盘与硬盘的主要不同就是软盘只有一个盘片且其存储密度较低。

由于软盘只有一个盘片，两个面，所以 3.5 英寸软盘的容量可以根据上一小节的公式算出：

$$2(\text{磁头}) \times 80(\text{磁道}) \times 18(\text{扇区}) \times 512 \text{ bytes}(\text{扇区的大小}) = 2880 \times 512 \text{ bytes} = 1440 \text{ KB} = 1.44 \text{ MB}$$

在这里需要引起我们特别注意的就是第 0 号磁头（面），第 0 号磁道的第 0 号扇区，这里是一切的开始。

1.2.3 启动扇区

软盘是没有所谓的 MBR 的，因为软盘容量较小，没有所谓的分区，一张软盘就显示为一个逻辑磁盘。当我们使用软盘启动电脑的时候，系统从软盘中首先读取的就是第一个扇区，即前面所说的第 0 面，第 0 号磁道的第 0 号扇区，如果这个扇区的最后两个字节是 0xaa55，这里就简单叫做启动扇区（Boot Sector）。所以我们首先要做的就是：在启动扇区的开始填入需要被执行的机器指令；在启动扇区的最后两个字节中填入 0xaa55，这样这张软盘就成为了一张可启动盘。



启动扇区最后两个字节的内容为 0xaa55，这种说法是正确的——当且仅当表 2.1 中的 BPB_BytessPerSec（每扇区字节数）的值为 512。如果 BPB_BytessPerSec 的值大于 512，0xaa55 的位置不会变化，但已经不是启动扇区最后两个字节了。

整个过程如图 1.4 所示：

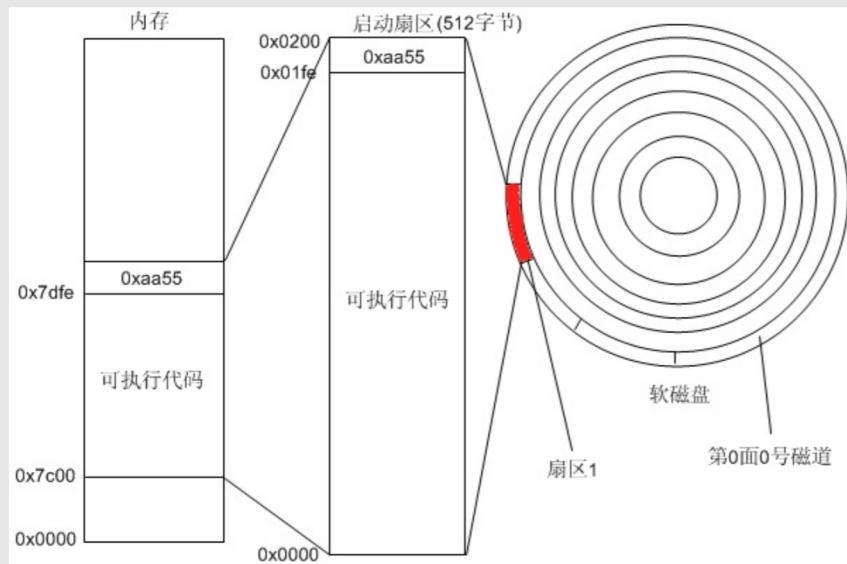


Fig 1.4: 启动扇区加载示意图

需要注意的是，软盘的启动扇区并不像一个文件一样，可以直接读取，写入启动扇区的过程是需要一些技巧的，下面我们将讨论如何去实现。

1.3 使用虚拟机

在实现一个简单的操作系统时，我们是不可能拿一台真正的机器做实验的，一是很少有人有这个条件，还有就是那样做比较麻烦。所以我们使用虚拟机来模拟一台真实的电脑，这样我们就能直接用虚拟机加载软盘镜像来启动了，而制作软盘镜像显然要比写一张真实的软盘简单许多。

在 Linux 下有很多虚拟机软件，我们选择 VirtualBox 和 Bochs 作为主要的实现平台，我们用 VirtualBox 做 demo，而 Bochs 主要用作调试。下面给出一些虚拟机设置的指导，其实用哪种虚拟机都没有关系，我们需要的只是虚拟机支持加载软盘镜像并能从软盘启动。

1.3.1 VirtualBox

VirtualBox 是遵从 GPL 协议的开源软件，它的官方网站是 <http://www.virtualbox.org>。VirtualBox 的官方网站上提供针对很多 Linux 系统平台的二进制安装包，比如针对 Red Hat 系列（Fedora, RHEL）各种版本的 RPM 安装包，针对 Debian 系（Debian, Ubuntu）各种版本的 DEB 安装包，其中 Ubuntu Linux 可以更方便地从 Ubuntu 软件仓库中直接下载安装：`sudo apt-get install virtualbox`。

安装好 VirtualBox 后，需要使用 `sudo adduser 'whoami' vboxusers`（某些系统中的添加用户命令可能是 `useradd`）将自己添加到 VirtualBox 的用户组 `vboxusers` 中去；当然，也可以使用 GNOME 或者 KDE 的图形界面用户和组的管理工具来添加组用户，也可以直接编辑 `/etc/group` 文件，将自己的用户名添加到 `vboxusers` 对应行的最后，例如 `vboxusers:x:501:solrex`，部分 Linux 可能需要注销后重新登录当前用户。

我们下面使用 CentOS 上安装的 VirtualBox 演示如何用它建立一个虚拟机。

第一次启动 VirtualBox，会首先弹出一个 VirtualBox 个人使用协议 PUEL 的对话框（某些版本的 Linux 可能不会弹出）：

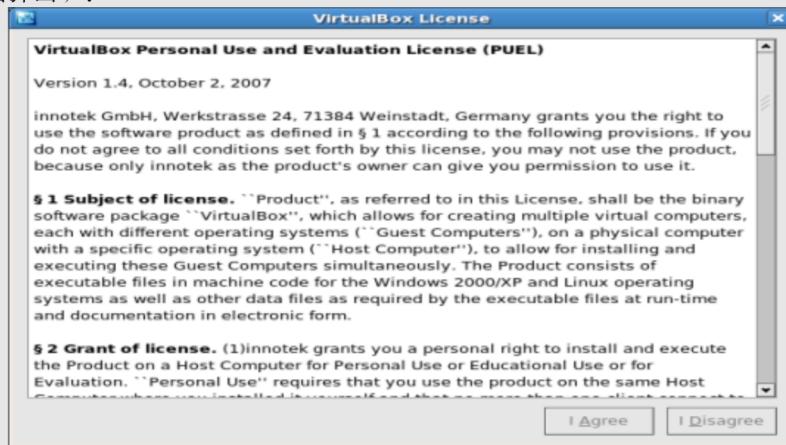


Fig 1.5: VirtualBox 个人使用协议

阅读完协议后，将下拉条拉到最低可以激活最下方的同意按钮，点击之：

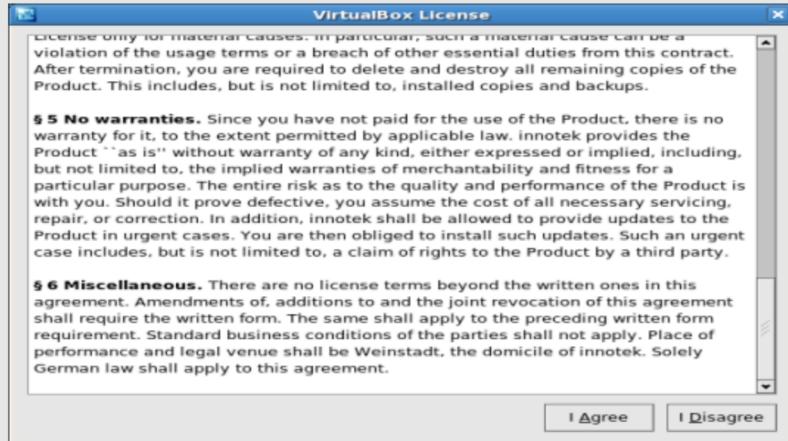


Fig 1.6: 同意 VirtualBox 个人使用协议

弹出的 VirtualBox 用户注册对话框，可忽视关闭之：



Fig 1.7: VirtualBox 用户注册对话框

接下来我们就见到了 VirtualBox 主界面：

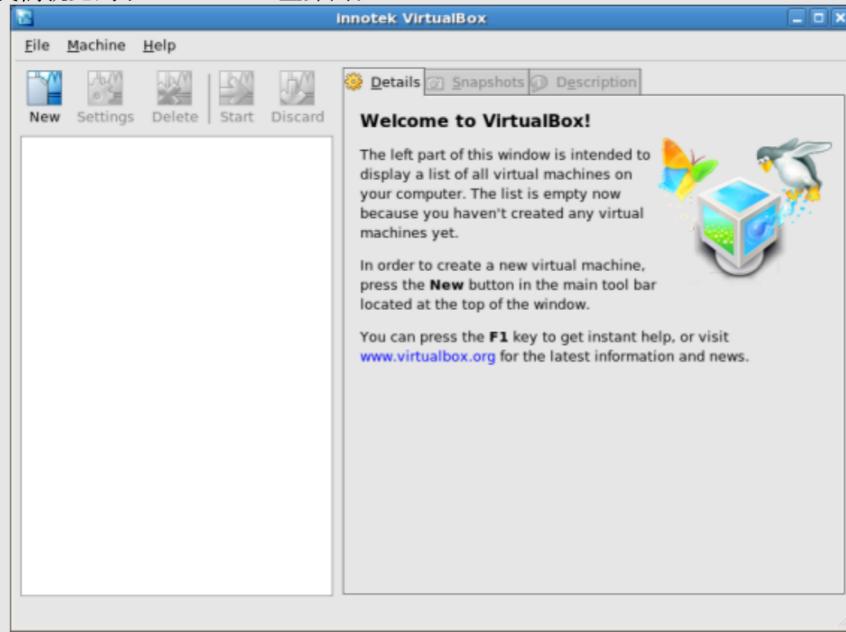


Fig 1.8: VirtualBox 主界面

点击 New 按钮新建一个虚拟机：

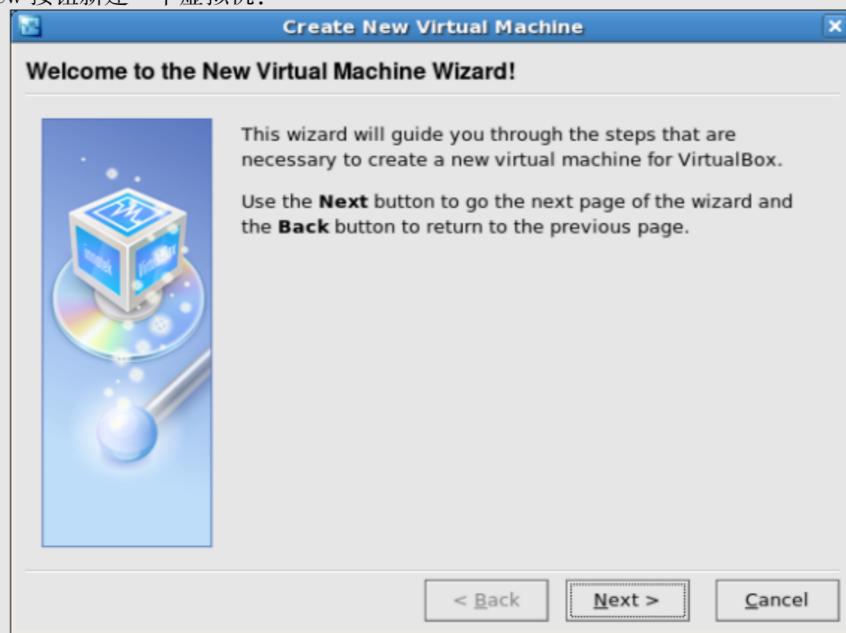


Fig 1.9: 新建一个虚拟机

我们使用 solrex 作为虚拟机的名字，系统类型未知：

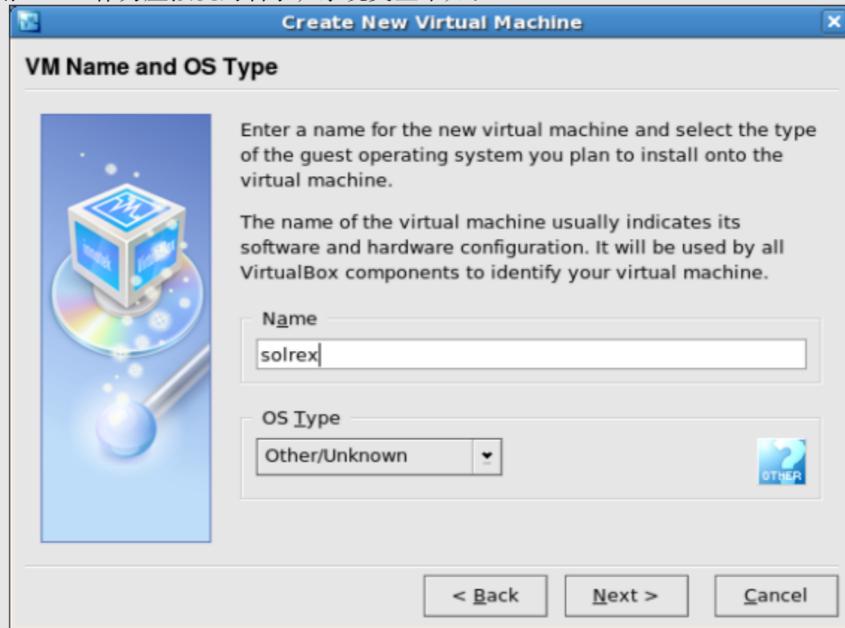


Fig 1.10: 设置虚拟机名字和操作系统类型

设置虚拟机的内存容量，这里随便设了 32M：

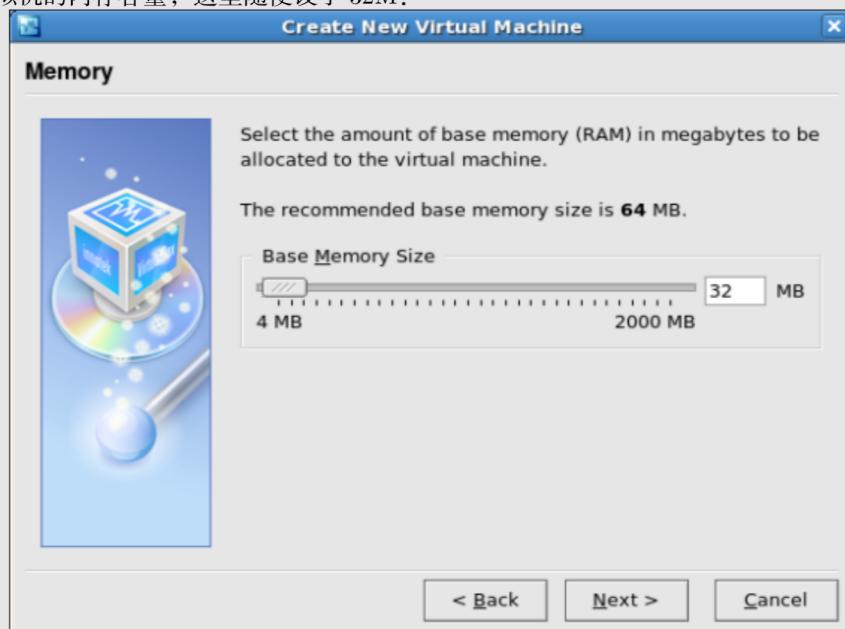


Fig 1.11: 设置虚拟机内存容量

设置虚拟机硬盘镜像：



Fig 1.12: 设置虚拟机硬盘镜像

如果没有硬盘镜像，需点“New”新建一块硬盘镜像：

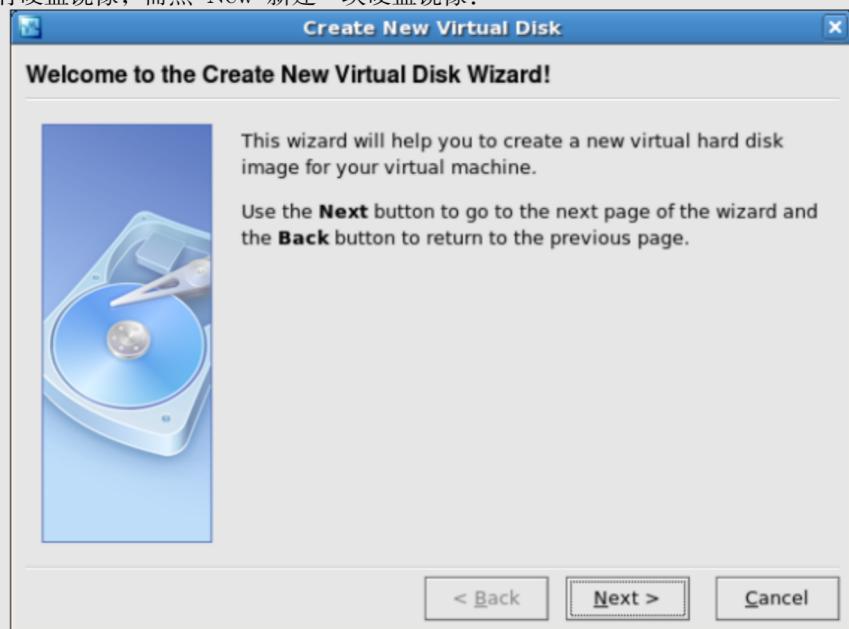


Fig 1.13: 新建一块虚拟硬盘

点“Next”，设置虚拟硬盘镜像为可自动扩充大小：



Fig 1.14: 设置虚拟硬盘类型

这里将虚拟硬盘镜像的名字设置为“solrex”，并将容量设置为“32M”：

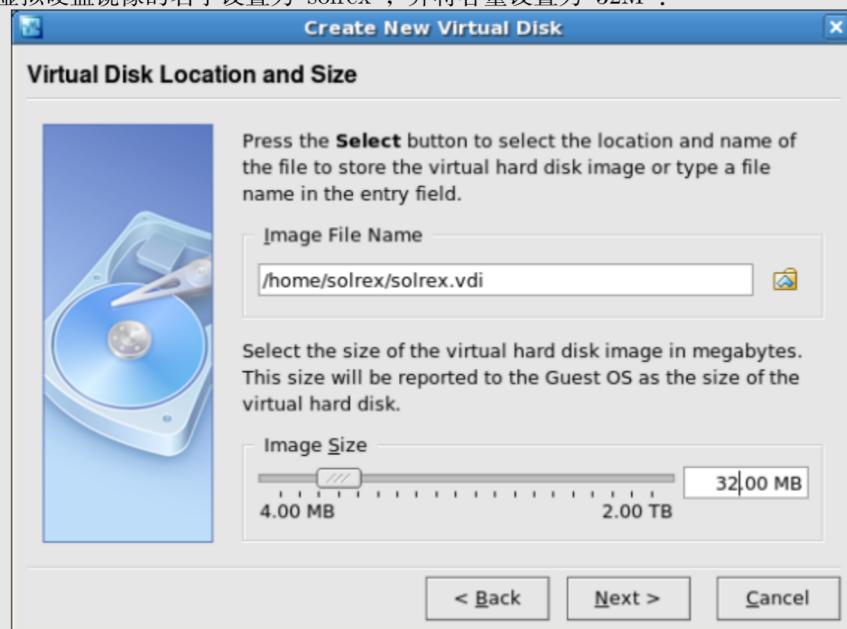


Fig 1.15: 设置虚拟硬盘镜像名字和容量

最后查看新建的虚拟硬盘信息，点击 Finish 确认新建硬盘镜像：

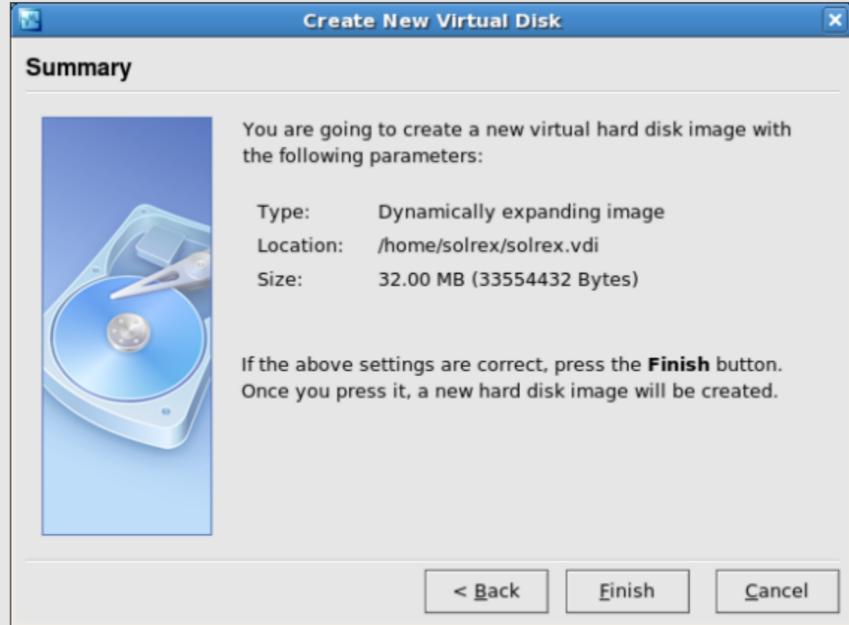


Fig 1.16: 虚拟硬盘信息

令虚拟机使用已建立的虚拟硬盘 solrex.vdi：

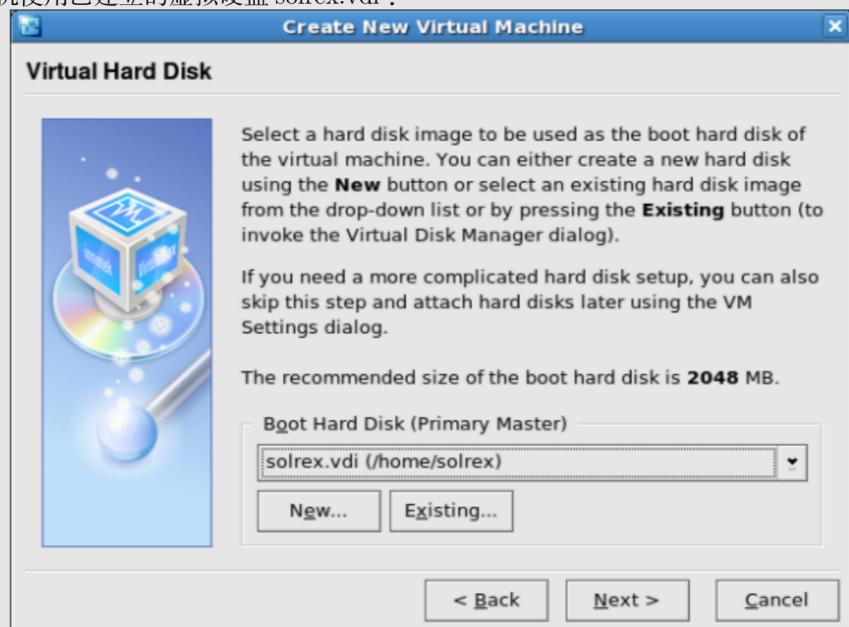


Fig 1.17: 使用新建的虚拟硬盘

最后查看新建的虚拟机信息，点击 Finish 确认新建虚拟机：

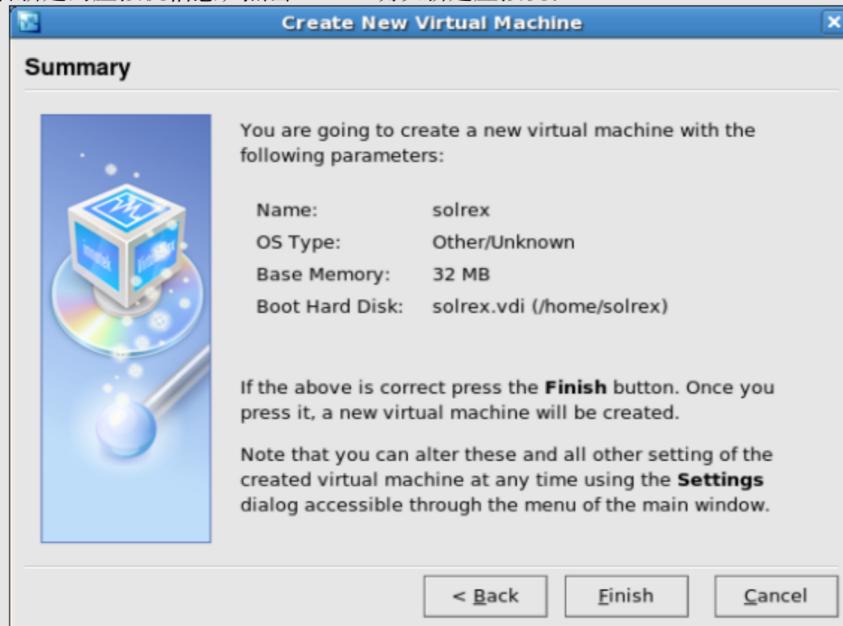


Fig 1.18: 虚拟机信息

回到 VirtualBox 主界面，左侧列表中有新建立的虚拟机 solrex：

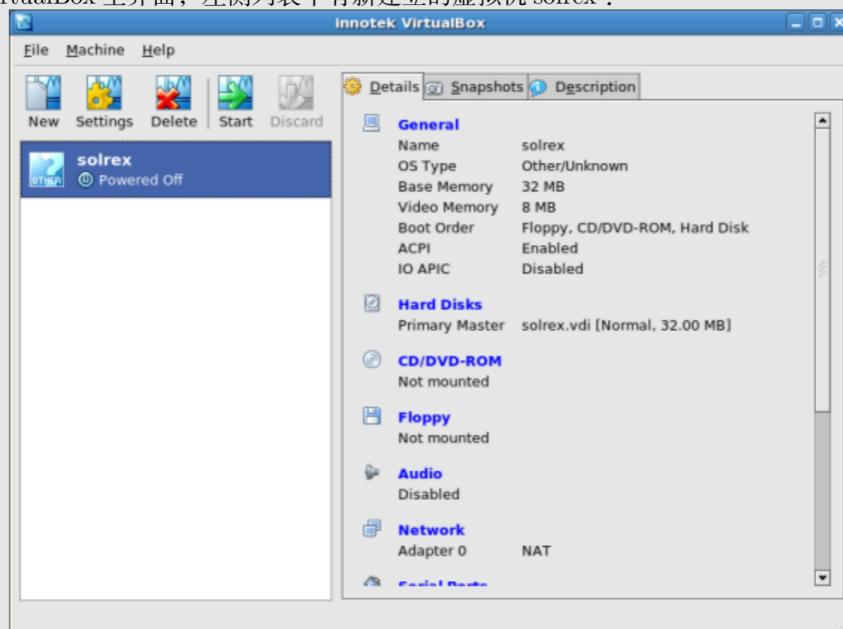


Fig 1.19: 回到 VirtualBox 主界面

1.3.2 Bochs

1.4 使用软盘镜像

1.4.1 制作软盘镜像

前面我们说过，软盘的结构比较简单，所以我们选择使用软盘镜像来启动虚拟计算机。在 Linux 下制作一个软盘镜像很简单，只需要使用：

```
$ dd if=/dev/zero of=emptydisk.img bs=512 count=2880
```

命令就可以在当前目录下生成一个名为 emptydisk.img 的空白软盘镜像，下面我们使用这个空白软盘镜像来启动虚拟机。

dd：转换和拷贝文件的工具。dd 可以设置很多拷贝时候的参数，在本例中 if=FILE 选项代表从 FILE 中读取内容；of=FILE 选项代表将导出输出到 FILE；bs=BYTES 代表每次读取和输出 BYTES 个字节；count=BLOCKS 代表从输入文件中共读取 BLOCKS 个输入块。

而这里的 /dev/zero 则是一个 Linux 的特殊文件，读取这个文件可以得到持续的 0。那么上面命令的意思就是以每块 512 字节共 2880 块全空的字符填入文件 emptydisk.img 中。我们注意到前面提及的软盘容量计算公式：

$$\begin{aligned} 2(\text{磁头}) \times 80(\text{磁道}) \times 18(\text{扇区}) \times 512 \text{ bytes} & (\text{扇区的大小}) = 2880 \times 512 \text{ bytes} = 1440 \text{ KB} \\ & = 1.44\text{MB} \end{aligned}$$

可以发现我们用上述命令得到的就是一张全空的未格式化的软盘镜像。

1.4.2 用软盘镜像启动虚拟机

在虚拟机主界面选中虚拟机后点 Settings 按钮，进入虚拟机的设置界面：

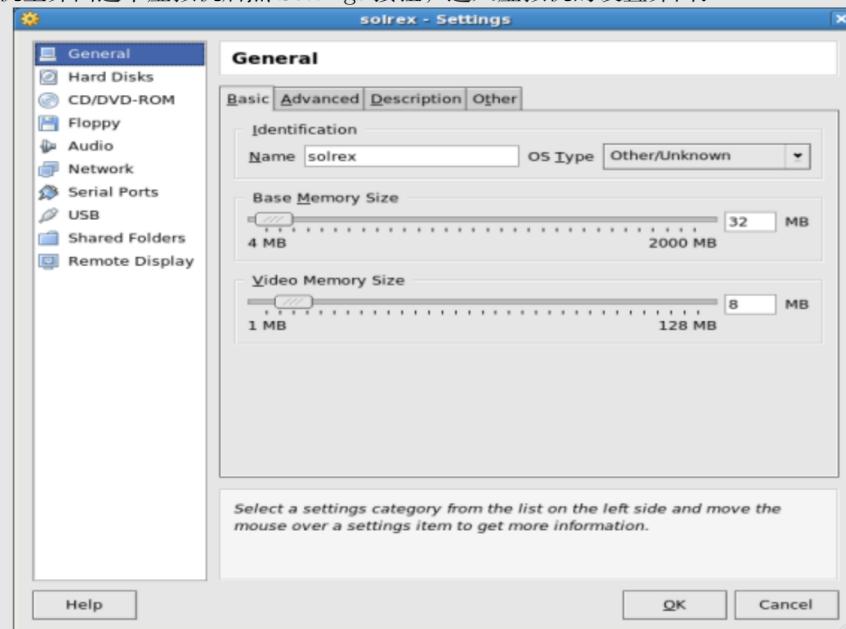


Fig 1.20: 虚拟机设置界面

在左侧列表中选择 Floppy 进入虚拟机软盘设置界面：

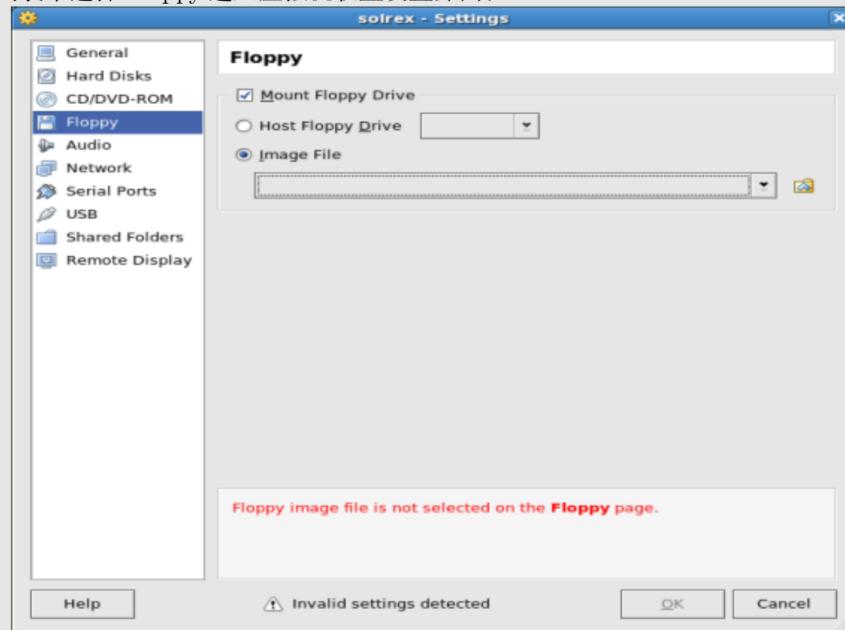


Fig 1.21: 虚拟机软盘设置

点击 Image File 最右侧的文件夹标志，进入选择软盘镜像界面：

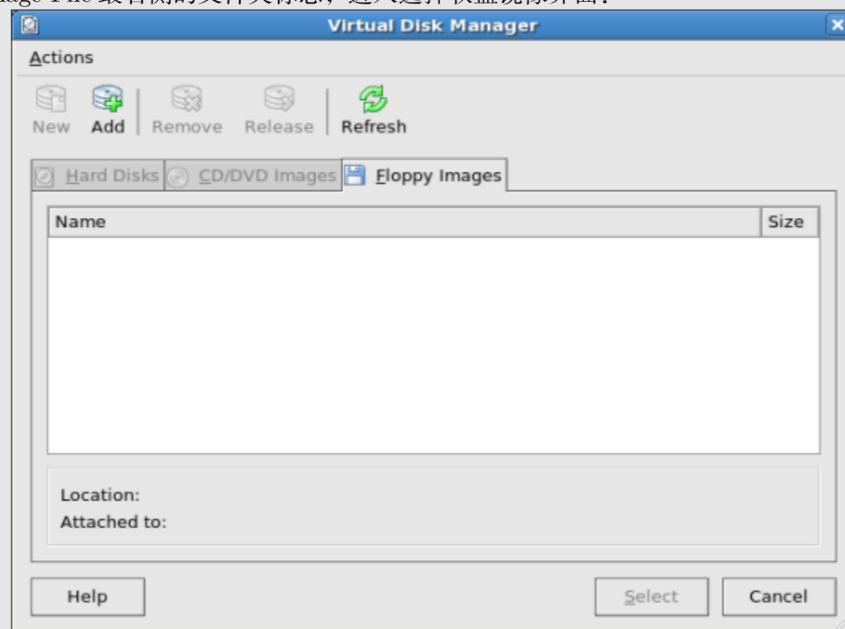


Fig 1.22: 选择软盘镜像

点击 Add 按钮添加新的软盘镜像 emptydisk.img，并点击 select 按钮选中其作为启动软盘：

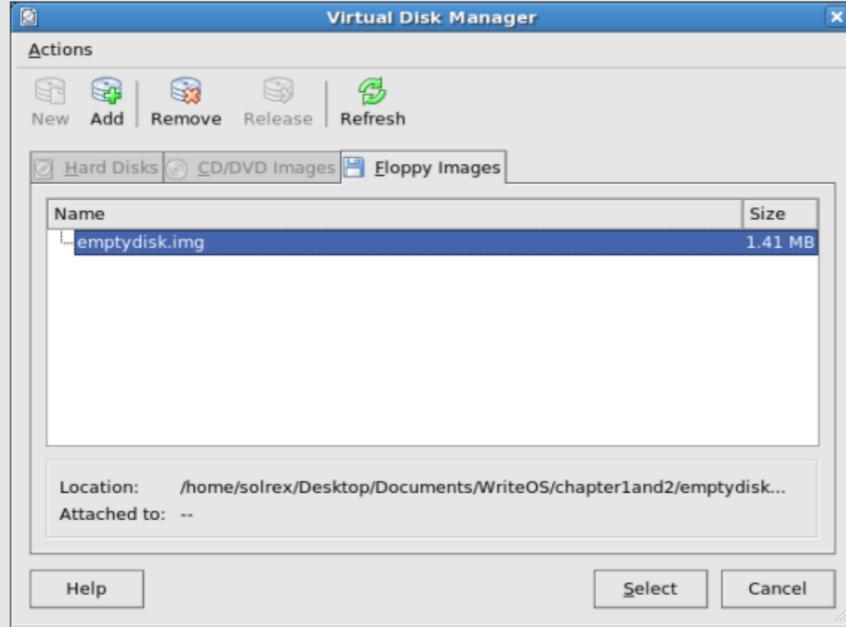


Fig 1.23: 选择启动软盘镜像

返回虚拟机软盘设置界面后，点击 OK 确认镜像文件信息：

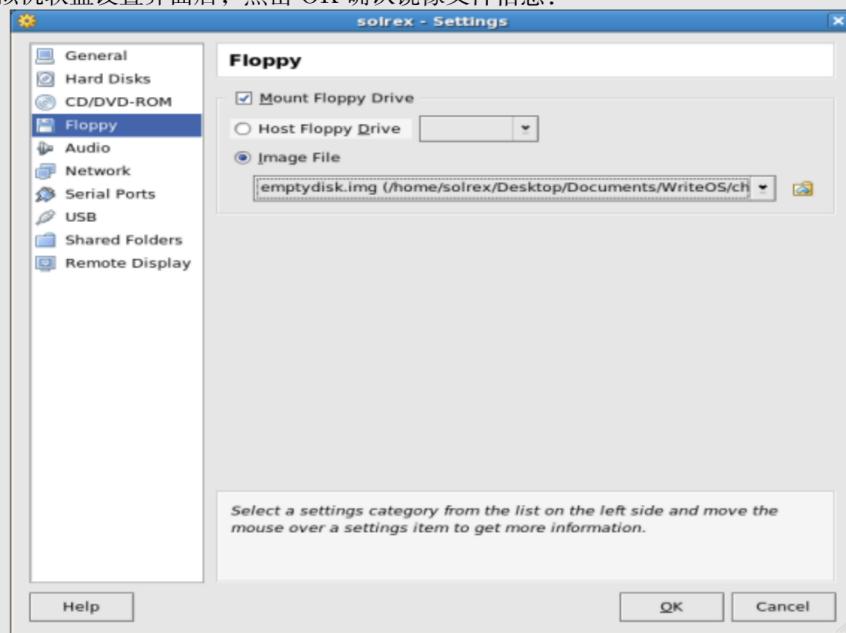


Fig 1.24: 确认启动镜像软盘文件信息

返回虚拟机主界面，查看右侧的虚拟机设置信息：

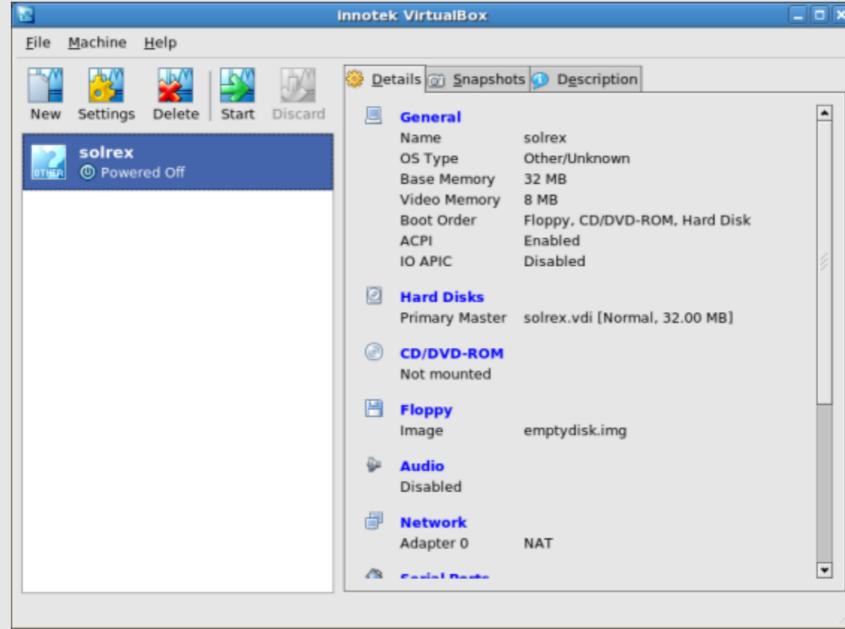


Fig 1.25: 查看虚拟机设置信息

选中虚拟机后，双击或点击 Start 按钮运行它，第一次运行可能给出如下信息：

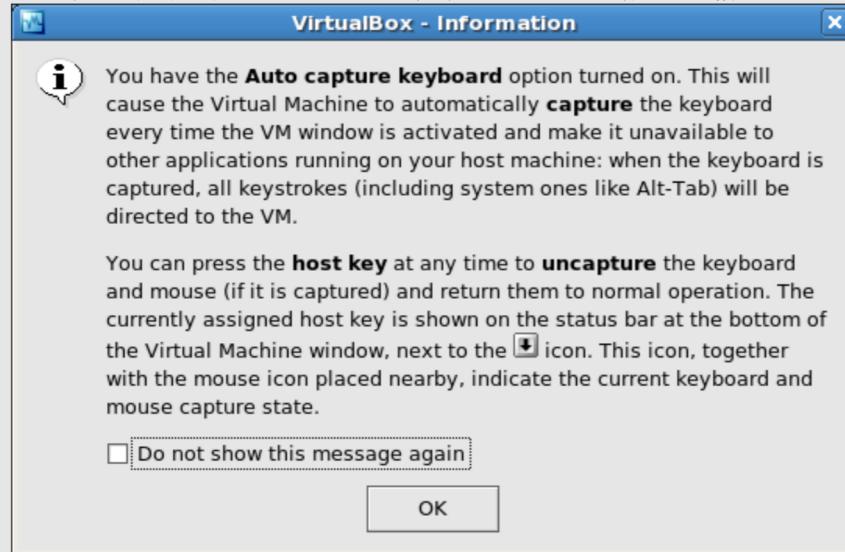


Fig 1.26: 自动键盘捕获警告信息

这个对话框的意思就是，当鼠标在虚拟机内部点击时，鼠标和键盘的消息将被虚拟机自动捕获，成为虚拟机的键盘和鼠标，可以敲击键盘右侧的 Ctrl 键解除捕获。

显示虚拟机的运行时内容：

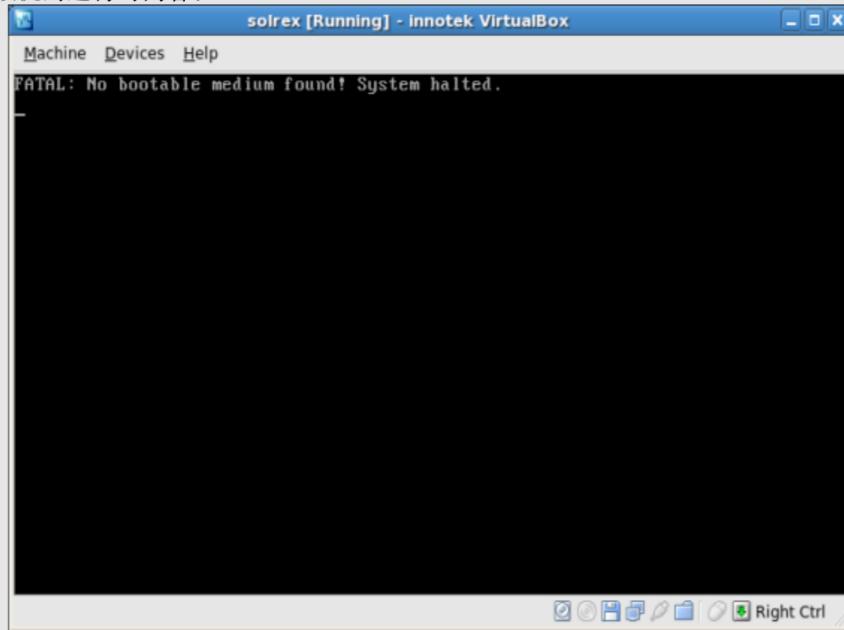


Fig 1.27: 虚拟机运行时

我们可以看到在图 1.27 中，虚拟机加载空白软盘启动后提示消息为：“FATAL: No bootable medium found! System halted.”，换成中文是找不到可启动媒体，系统停机。它的实际意思就是在前面第 1.1 节“计算机启动过程”中提到的第 6 步中虚拟机遍历了软驱、光驱、硬盘后没有找到可启动的媒体，所以就只好停机。因为我们在启动前已经在软驱中加载了软盘镜像，所以提示信息就表明那个软盘镜像不具有启动系统的功能，那么如何才能创建一个可启动的软盘呢，我们将在第 2 章介绍。

CHAPTER 2

最小的“操作系统”

任何一个完善的操作系统都是从启动扇区开始的，这一章，我们就关注如何写一个启动扇区，以及如何将其写入到软盘镜像中。

先介绍一下需要使用的工具：

- 系统：Cent OS 5.1(RHEL 5.1)
- 使用工具：gcc, binutils(as, ld, objcopy), dd, make, hexdump, vim, virtualbox

2.1 Hello OS world!



本章节内容需要和 gcc, make 相关的 Linux C 语言编程以及 PC 汇编语言的基础知识。



推荐预备阅读：CS:APP (Computer Systems: A Programmer's Perspective, 深入理解计算机系统) 第 3 章：Machine-Level Representation of Programs。

很多编程书籍给出的第一个例子往往是在终端里输出一个字符串“Hello world!”，那么要写操作系统的第一步给出的例子自然就是如何在屏幕上打印出一个字符串喽。所以，我们首先看《自己动手写操作系统》一书中给出的第一个示例代码，在屏幕上打印“Hello OS world!”：

```
1  org    07c00h      ; 告诉编译器程序加载到7c00处
2  mov    ax, cs
3  mov    ds, ax
4  mov    es, ax
5  call   DispStr     ; 调用显示字符串例程
6  jmp   $             ; 无限循环
7 DispStr:
```

```

8     mov    ax, BootMessage
9     mov    bp, ax      ; ES:BP = 串地址
10    mov   cx, 16       ; CX = 串长度
11    mov   ax, 01301h   ; AH = 13, AL = 01h
12    mov   bx, 000ch    ; 页号为0(BH = 0) 黑底红字(BL = 0Ch,高亮)
13    mov   dl, 0
14    int   10h        ; 10h 号中断
15    ret
16 BootMessage: db "Hello, OS world!"
17 times 510-($-$) db 0 ; 填充剩下的空间, 使生成的二进制代码恰好为512字节
18 dw    Oxaa55        ; 结束标志

```

Fig 2.1: 《自》第一个实例代码 boot.asm

2.1.1 Intel 汇编转化为 AT&T(GAS) 汇编

上面 boot.asm 中代码使用 Intel 风格的汇编语言写成，本也可以在 Linux 下使用同样开源的 NASM 编译，但是鉴于很少有人在 Linux 下使用此汇编语法，它在 Linux 平台上的扩展性和可调试性都不好（GCC 不兼容），而且不是采用 Linux 平台上编译习惯，所以我把它改成了使用 GNU 工具链去编译连接。这样的话，对以后使用 GNU 工具链编写其它体系结构的 bootloader 也有帮助，毕竟 NASM 没有 GAS 用户多（也许 ⊙）。

上面的汇编源程序可以改写成 AT&T 风格的汇编源代码：

```

1 .code16          #使用16位模式汇编
2 .text            #代码段开始
3     mov    %cs,%ax
4     mov    %ax,%ds
5     mov    %ax,%es
6     call   DispStr  #调用显示字符串例程
7     jmp    .         #无限循环
8 DispStr:
9     mov    $BootMessage, %ax
10    mov    %ax,%bp      #ES:BP = 串地址
11    mov    $16,%cx      #CX = 串长度
12    mov    $0x1301,%ax   #AH = 13, AL = 01h
13    mov    $0x00c,%bx    #页号为0(BH = 0) 黑底红字(BL = 0Ch,高亮)
14    mov    $0,%dl
15    int   $0x10        #10h 号中断
16    ret
17 BootMessage:.ascii "Hello, OS world!"
18 .org 510          #填充到“510”字节处
19 .word Oxaa55        #结束标志

```

Fig 2.2: boot.S(chapter2/1/boot.S)

2.1.2 用连接脚本控制地址空间

但有一个问题，我们可以使用 `nasm boot.asm -o boot.bin` 命令将 boot.asm 直接编译成二进制文件，GAS 不能。不过 GAS 的不能恰好给开发者一个机会去分步地实现从汇编源代码到二进制文件这

个过程，使编译更为灵活。下面请看 GAS 是如何通过连接脚本控制程序地址空间的：

```

11 SECTIONS
12 {
13   . = 0x7c00;
14   .text :
15   {
16     _ftext = .;      /* Program will be loaded to 0x7c00. */
17   } = 0
18 }
```

Fig 2.3: boot.S 的连接脚本 (chapter2/1/solrex_x86.ld)

连接脚本： GNU 连接器 ld 的每一个连接过程都由连接脚本控制。连接脚本主要用于，怎样把输入文件内的 section 放入输出文件内，并且控制输出文件内各部分在程序地址空间内的布局。连接器有个默认的内置连接脚本，可以用命令 `ld -verbose` 查看。选项 `-T` 选项可以指定自己的连接脚本，它将代替默认的连接脚本。

这个连接脚本的功能就是，在连接的时候，将程序入口设置为内存 `0x7c00` 的位置（BIOS 将跳转到这里继续启动过程），相当于 `boot.asm` 中的 `org 07c00h` 一句。有人可能觉得麻烦，还需要用一个脚本控制加载地址，但是《自己动手写操作系统》就给了一个很好的反例：《自》第 1.5 节代码 1-2，作者切换调试和运行模式时候需要对代码进行注释。

```

1 ;%define _BOOT_DEBUG_ ; 做 Boot Sector 时一定将此行注释掉!将此行打开后用
2 ; nasm Boot.asm -o Boot.com 做成一个.COM文件易于调试
3
4 %ifdef _BOOT_DEBUG_
5   org 0100h ; 调试状态，做成 .COM 文件，可调试
6 %else
7   org 07c00h ; Boot 状态，Bios 将把 Boot Sector 加载到 0:7C00 处并开始执行
8 %endif
9
10  mov ax, cs
11  mov ds, ax
12  mov es, ax
13  call DispStr ; 调用显示字符串例程
14  jmp $ ; 无限循环
15 DispStr:
16  mov ax, BootMessage
17  mov bp, ax ; ES:BP = 串地址
18  mov cx, 16 ; CX = 串长度
19  mov ax, 01301h ; AH = 13, AL = 01h
20  mov bx, 000ch ; 页号为0(BH = 0) 黑底红字(BL = 0Ch,高亮)
21  mov dl, 0
22  int 10h ; 10h 号中断
23  ret
24 BootMessage: db "Hello, OS world!"
25 times 510-($-$) db 0 ; 填充剩下的空间，使生成的二进制代码恰好为512字节
26 dw 0xaa55 ; 结束标志
```

Fig 2.4: 《自》代码 1-2 (chapter2/1/boot.asm)

而如果换成使用脚本控制程序地址空间，只需要编译时候调用不同脚本进行连接，就能解决这个问题。这在嵌入式编程中是很常见的处理方式，即使用不同的连接脚本一次 make 从一个源程序文件生成分别运行在开发板上和软件模拟器上的两个二进制文件。

2.1.3 用 Makefile 编译连接

下面的这个 Makefile 文件，就是我们用来自动编译 boot.S 汇编源代码的脚本文件：

```

1 CC=gcc
2 LD=ld
3 LDFILE=solrex_x86.ld      #使用上面提供的连接脚本 solrex_x86.ld
4 OBJCOPY=objcopy
5
6 all: boot.img
7
8 # Step 1: gcc 调用 as 将 boot.S 编译成目标文件 boot.o
9 boot.o: boot.S
10      $(CC) -c boot.S
11
12 # Step 2: ld 调用连接脚本 solrex_x86.ld 将 boot.o 连接成可执行文件 boot.elf
13 boot.elf: boot.o
14      $(LD) boot.o -o boot.elf -e c -T$(LDFILE)
15
16 # Step 3: objcopy 移除 boot.elf 中没有用的 section(.pdr,.comment,.note),
17 #          strip 掉所有符号信息，输出为二进制文件 boot.bin 。
18 boot.bin : boot.elf
19      @$(OBJCOPY) -R .pdr -R .comment -R.note -S -O binary boot.elf boot.bin
20
21 # Step 4: 生成可启动软盘镜像。
22 boot.img: boot.bin
23      @dd if=boot.bin of=boot.img bs=512 count=1           #用 boot.bin 生成镜像文件第一个扇区
24      # 在 bin 生成的镜像文件后补上空白，最后成为合适大小的软盘镜像
25      @dd if=/dev/zero of=boot.img skip=1 seek=1 bs=512 count=2879
26
27 clean:
28      @rm -rf boot.o boot.elf boot.bin boot.img

```

Fig 2.5: boot.S 的 Makefile(chapter2/1/Makefile)

我们将上面内容保存成 Makefile，与图 2.2 所示 boog.S 和图 2.3 所示 solrex_x86.ld 放在同一个目录下，然后在此目录下使用下面命令编译：

```

$ make
gcc -c boot.S
ld boot.o -o boot.elf -Tsolrex_x86.ld
1+0 records in
1+0 records out
512 bytes (512 B) copied, 3.1289e-05 seconds, 16.4 MB/s
2879+0 records in
2879+0 records out

```

```
1474048 bytes (1.5 MB) copied, 0.0141508 seconds, 104 MB/s
$ ls
boot.asm boot.elf boot.o Makefile solrex_x86.1d
boot.bin boot.img boot.S solrex.img
```

可以看到，我们只需执行一条命令 make 就可以编译、连接和直接生成可启动的软盘镜像文件，其间对源文件的每一步处理也都一清二楚。不用任何商业软件，也不用自己写任何转换工具，比如《自己动手写操作系统》文中提到的 HD-COPY 和 Floopy Writer 都没有使用到。

在这里需要特别注意的是图 2.5 中的 Step 4，其实对这一步的解释应该结合图 1.4 来查看。我们用 boot.S 编译生成的 boot.bin 其实只是图 1.4 中所指的软盘的启动扇区，例如 boot.S 最后一行：

```
.word 0xaa55      #结束标志
```

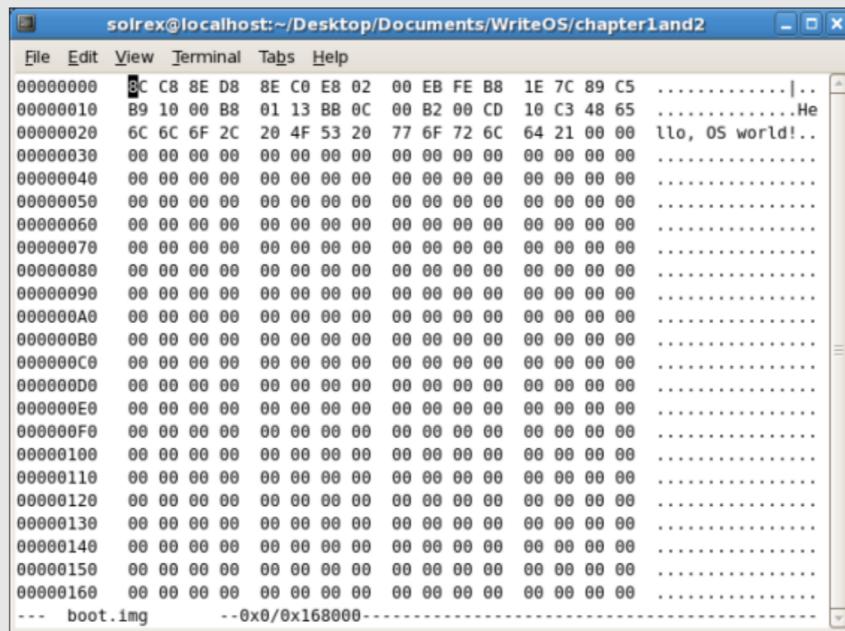
生成就是启动扇区最后的 0xaa55 那两个字节，而 boot.bin 的大小是 512 字节，正好是启动扇区的大小。那么 Step 4 的功能就是把 boot.bin 放入到一个空白软盘的启动扇区，这样呢当虚拟机启动时能识别出这是一张可启动软盘，并且执行我们在启动扇区中写入的打印代码。

为了验证软盘镜像文件的正确性也可以先用

```
$ hexdump -x -n 512 boot.img
```

将 boot.img 前 512 个字节打印出来，可以看到 boot.img dump 的内容和《自》一书附送光盘中的 TINIX.IMG dump 的内容完全相同。这里我们也显然用不到 EditPlus 或者 UltraEdit，即使需要修改二进制码，也可以使用 hexedit, ghex2, khxedit 等工具对二进制文件进行修改。

下图为使用命令行工具 hexedit 打开 boot.img 的窗口截图，从图中我们可以看到，左列是该行开头与文件头对应的偏移地址，中间一列是文件的二进制内容，最右列是文件内容的 ASCII 显示内容，可以看到，此界面与 UltraEdit 的十六进制编辑界面没有本质不同。



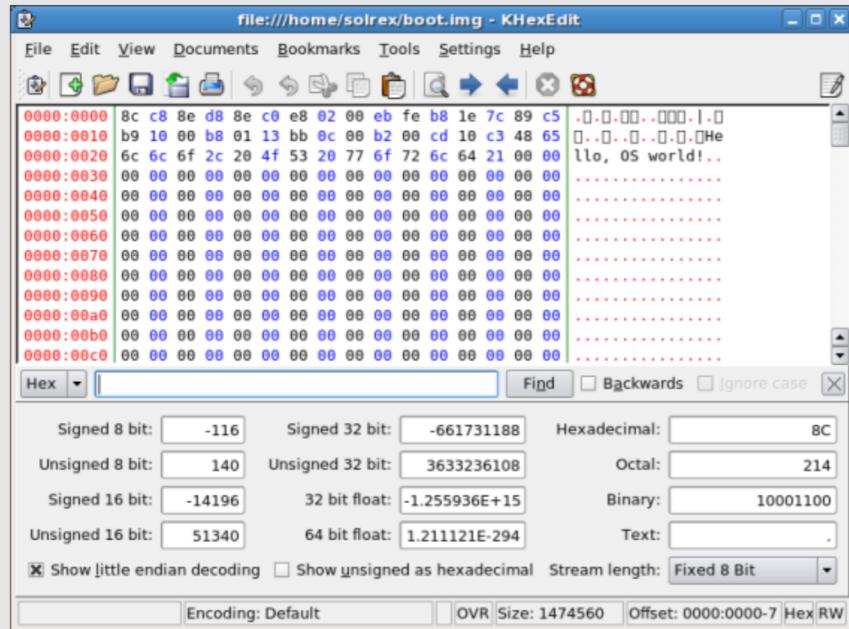


Fig 2.7: 使用 kde 图形界面工具 khexedit 打开 boot.img

2.1.4 用虚拟机加载执行 boot.img

当我们生成 boot.img 之后，仿照第 1.4.2 节中加载软盘镜像的方法，用虚拟机加载 boot.img：

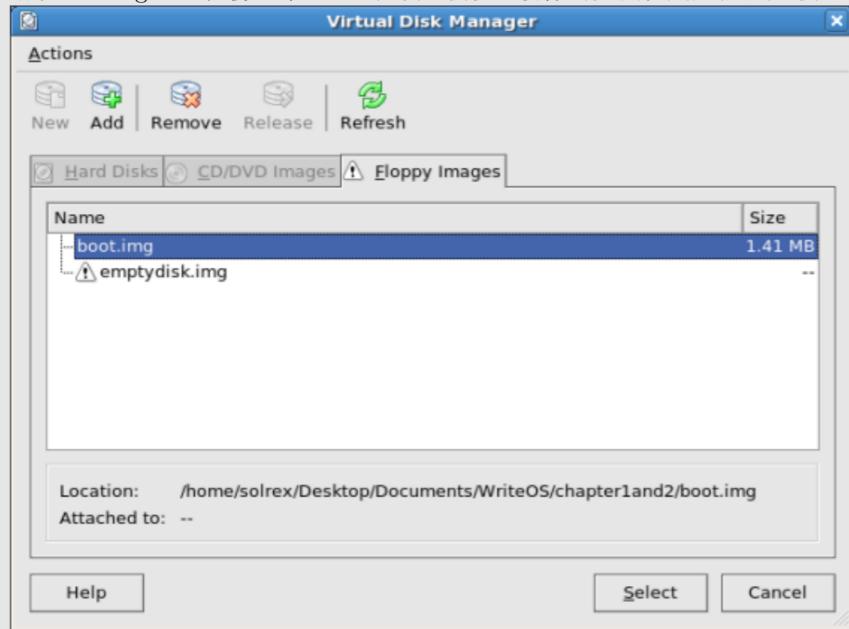


Fig 2.8: 选择启动软盘镜像 boot.img

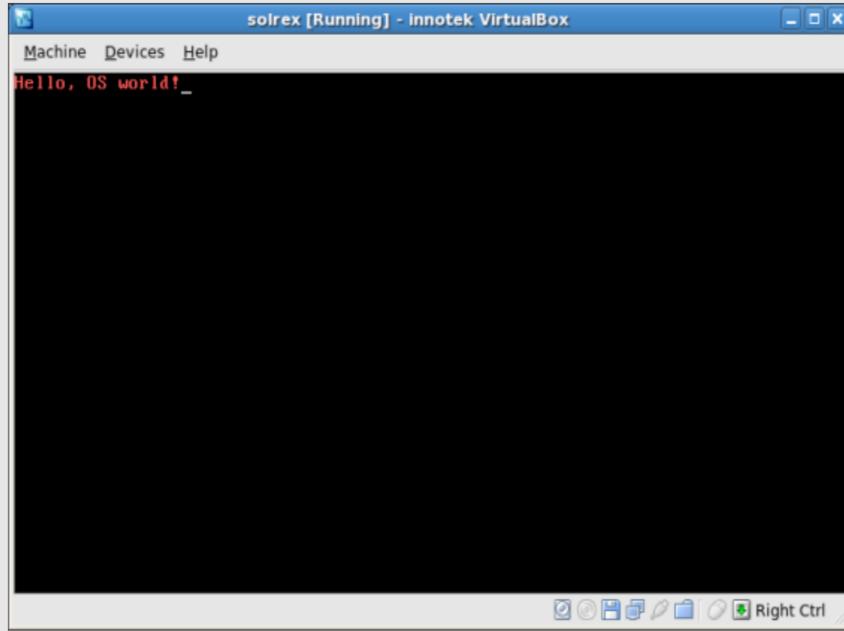


Fig 2.9: 虚拟机启动后打印出红色的“Hello OS world!”

我们看到虚拟机如我们所料的打印出了红色的“Hello OS world!”字样，这说明我们以上的程序和编译过程是正确的。

2.2 FAT 文件系统

我们在上一节中介绍的内容，仅仅是写一个启动扇区并将其放入软盘镜像的合适位置。由于启动扇区 512 字节的大小限制，我们仅仅能写入像打印一个字符串这样的非常简单的程序，那么如何突破 512 字节的限制呢？很显然的答案是我们要利用其它的扇区，将程序保存在其它扇区，运行前将其加载到内存后再跳转过去执行。那么又一个问题产生了：程序在软盘上应该怎样存储呢？

可能最直接最容易理解的存储方式就是顺序存储，即将一个大程序从启动扇区开始按顺序存储在相邻的扇区，可能这样需要的工作量最小，在启动时操作系统仅仅需要序列地将可执行代码拷贝到内存中来继续运行。可是经过简单的思考我们就可以发现这样做有几个缺陷：1. 软盘中仅能存储操作系统程序，无法存储其它内容；2. 我们必须使用二进制拷贝方式来制作软盘镜像，修改系统麻烦。

那么怎么避免这两个缺点呢？引入文件系统可以让我们在一张软盘上存储不同的文件，并提供文件管理功能，可以让我们避免上述的两个缺点。在使用某种文件系统对软盘格式化之后，我们可以像普通软盘一样使用它来存储多个文件和目录，为了使用软盘上的文件，我们给启动扇区的代码加上寻找文件和加载执行文件功能，让启动扇区将系统控制权转移给软盘上的某个文件，这样突破启动扇区 512 字节大小的限制。

2.2.1 FAT12 文件系统

FAT(File Allocation Table) 文件系统规格在 20 世纪 70 年代末和 80 年代初形成，是微软的 MS-DOS 操作系统使用的文件系统格式。它的初衷是为小于 500K 容量的软盘制定的简单文件系统，但在

将近三十年的发展过程中，它已经被一次次修改加强以支持更大的存储媒体。在目前主要有三种 FAT 文件系统类型：FAT12, FAT16 和 FAT32。这几种类型最基本的区别就像它们的名字字面区别一样，主要在于大小，即盘上 FAT 表的记录项所占的比特数。FAT12 的记录项占 12 比特，FAT16 占 16 比特，FAT32 占 32 比特。

由于 FAT12 最为简单和易实施，这里我们仅简单介绍 FAT12 文件系统，想要了解更多 FAT 文件系统知识的话，可以到 <http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx> 下载微软发布的 FAT 文件系统官方文档。

FAT12 文件系统和其它文件系统一样，都将磁盘划分为层次进行管理。从逻辑上划分，一般将磁盘划分为盘符，目录和文件；从抽象物理结构来讲，将磁盘划分为分区，簇和扇区。那么，如何将逻辑上的目录和文件映射到物理上实际的簇和扇区，就是文件系统要解决的问题。

如果让虚拟机直接读取我们上一节生成的可启动软盘镜像，或者将 boot.img 软盘用 mount -o loop boot.img mountdir/ 挂载到某个目录上，系统肯定会报出“软盘未格式化”或者“文件格式不可识别”的错误。这是因为任何系统可读取的软盘都是被格式化过的，而我们的 boot.img 是一个非常原始的软盘镜像。那么如何才能使软盘被识别为 FAT12 格式的软盘并且可以像普通软盘一样存取呢？

系统在读取一张软盘的时候，会读取软盘上存储的一些关于文件系统的信息，软盘格式化的过程也就是系统把文件系统信息写入到软盘上的过程。但是我们不能让系统来格式化我们的 boot.img，如果那样的话，我们写入的启动程序也会被擦除。所以呢，我们需要自己对软盘进行格式化。❶可能有人看到这里就会很沮丧，天那，那该有多麻烦啊！不过我相信在读完以下内容以后你会欢呼雀跃，啊哈，原来文件系统挺简单的嘛！

2.2.2 启动扇区与 BPB

FAT 文件系统的主要信息，都被提供在前几个扇区内，其中第 0 号扇区尤其重要。在这个扇区内隐藏着一个叫做 BPB(BIOS Parameter Block) 的数据结构，一旦我们把这个数据结构写对了，格式化过程也基本完成了❷。下面这个表中所示内容，主要就是启动扇区的 BPB 数据结构。

表 2.1：启动扇区的 BPB 数据结构和其它内容

名称	偏移 bytes	大小 bytes	描述	Solrex.img 文件中的值
BS_jmpBoot	0	3	跳转指令，用于跳过以下的扇区信息	jmp LABEL_START nop
BS_OEMName	3	8	厂商名	"WB. YANG"
BPB_BytsPerSec	11	2	扇区大小(字节)，应为512	512
BPB_secPerClus	13	1	簇的扇区数，应为2的幂，FAT12 为1	1
BPB_RsvdSecCnt	14	2	保留扇区，FAT12/16 应为1	1
BPB_NumFATs	16	1	FAT 结构数目，一般为2	2
BPB_RootEntCnt	17	2	根目录项目数，FAT12 为224	224
BPB_TotSec16	19	2	扇区总数，1.44M 软盘为2880	2880
BPB_Media	21	1	设备类型，1.44M 软盘为F0h	0xf0
BPB_FATSz16	22	2	FAT 占用扇区数，9	9

BPB_SecPerTrk	24	2	磁道扇区数, 18	18
BPB_NumHeads	26	2	磁头数, 2	2
BPB_HiddSec	28	4	隐藏扇区, 默认为0	0
BPB_TotSec32	32	4	如果 BPB_TotSec16 为 0, 它记录总扇区数	0
下面的扇区头信息 FAT12/FAT16 与 FAT32 不同				
BS_DrvNum	36	1	中断 0x13 的驱动器参数, 0 为软盘	0
BS_Reserved1	37	1	Windows NT 使用, 0	0
BS_BootSig	38	1	扩展引导标记 (29h), 指明此后 3 个域可用	0x29
BS_VolID	39	4	卷标序列号, 00000000h	0
BS_VolLab	43	11	卷标, 11 字节, 必须用空格20h 补齐	"Solrex 0.01"
BS_FileSysType	54	8	文件系统标志, "FAT12 "	"FAT12 "
以下为非扇区头信息部分				
启动代码及其它	62	448	启动代码、数据及填充字符	mov %cs,%ax...
启动扇区标识符	510	2	可启动扇区标志, 0xAA55	0xaa55

哇, 天那, 这个 BPB 看起来很多东西的嘛, 怎么写啊? 其实写入这些信息很简单, 因为它们都是固定不变的内容, 用下面的代码就可以实现。

```

21 /* Floppy header of FAT12 */
22     jmp    LABEL_START /* Start to boot. */
23     nop        /* nop required */
24 BS_OEMName:      .ascii  "WB. YANG"      /* OEM String, 8 bytes required */
25 BPB_BytsPerSec:   .2byte 512          /* Bytes per sector */
26 BPB_SecPerCluster: .byte 1           /* Sector per cluster */
27 BPB_ResvdSecCnt:  .2byte 1           /* Reserved sector count */
28 BPB_NumFATs:     .byte 2           /* Number of FATs */
29 BPB_RootEntCnt:   .2byte 224         /* Root entries count */
30 BPB_TotSec16:    .2byte 2880         /* Total sector number */
31 BPB_Media:       .byte 0xf0          /* Media descriptor */
32 BPB_FATSz16:     .2byte 9            /* FAT size(sectors) */
33 BPB_SecPerTrk:   .2byte 18          /* Sector per track */
34 BPB_NumHeads:    .2byte 2           /* Number of magnetic heads */
35 BPB_HiddSec:     .4byte 0            /* Number of hidden sectors */
36 BPB_TotSec32:    .4byte 0           /* If TotSec16 equal 0, this works */
37 BS_DrvNum:       .byte 0           /* Driver number of interrupt 13 */
38 BS_Reserved1:    .byte 0           /* Reserved */
39 BS_BootSig:      .byte 0x29          /* Boot signal */
40 BS_VolID:        .4byte 0           /* Volume ID */
41 BS_VolLab:       .ascii  "Solrex 0.01" /* Volume label, 11 bytes required */
42 BS_FileSysType:  .ascii  "FAT12 "     /* File system type, 8 bytes required */
43
44 /* Initial registers. */
45 LABEL_START:

```

Fig 2.10: 生成启动扇区头的汇编代码(节自chapter2/2/boot.S)

在上面的汇编代码中，我们只是顺序地用字符填充了启动扇区头的数据结构，填充的内容与表 2.1 中最后一列的内容相对应。把图 2.10 中所示代码添加到图 2.2 的第二行和第三行之间，然后再 make，就能得到一张已经被格式化，可启动也可存储文件的软盘，就是既可以使用 `mount -o loop boot.img mountdir/` 命令在普通 Linux 系统里挂载，也可用作虚拟机启动的软盘镜像文件。

2.2.3 FAT12 数据结构

在上一个小节里，我们制作出了可以当作普通软盘使用的启动软盘，这样我们就可以在这张软盘上存储多个文件了。可还有一步要求我们没有达到，怎样寻找存储的某个引导文件并将其加载到内存中运行呢？这就涉及到 FAT12 文件系统中文件的存储方式了，需要我们了解一些 FAT 数据结构和目录结构的知识。

FAT 文件系统对存储空间分配的最小单位是“簇”，因此文件在占用存储空间时，基本单位是簇而不是字节。即使文件仅有 1 字节大小，系统也必须分给它一个最小存储单元——簇。由表 2.1 中的 BPB_secPerClus 和 BPB_BytsPerSec 相乘可以得到每簇所包含的字节数，可见我们设置的是每簇包含 $1 \times 512 = 512$ 个字节，恰好是每簇包含一个扇区。

存储空间分配的最小单位确定了，那么 FAT 是如何分配和管理这些存储空间的呢？FAT 的存储空间管理是通过管理 FAT 表来实现的，FAT 表一般位于启动扇区之后，根目录之前的若干个扇区，而且一般是两个表。从根目录区的下一个簇开始，每个簇按照它在磁盘上的位置映射到 FAT 表里。FAT 文件系统的存储结构粗略上来讲如图 2.11 所示。



Fig 2.11: FAT 文件系统存储结构图

FAT 表的表项有点儿像数据结构中的单向链表节点的 next 指针，先回忆一下，单向链表节点的数据结构（C 语言）是：

```
struct node {
    char * data;
    struct node *next;
};
```

在链表中，next 指针指向的是下一个相邻节点。那么 FAT 表与链表有什么区别呢？首先，FAT 表将 next 指针集中管理，放在一起被称为 FAT 表；其次，FAT 表项指向的是固定大小的文件“簇”（data

段)，而且每个文件簇都有自己对应的 FAT 表项。由于每个文件簇都有自己的 FAT 表项，这个表项可能指向另一个文件簇，所以 FAT 表项所占字节的多少就决定了 FAT 表最大能管理多少内存，FAT12 的 FAT 表项有 12 个比特，大约能管理 2^{12} —一个文件簇。

一个文件往往要占据多个簇，只要我们知道这个文件的第一个簇，就可以到 FAT 表里查询该簇对应的 FAT 表项，该表项的内容一般就是此文件下一个簇号。如果该表项的值大于 0xff8，则表示该簇是文件最后一个簇，相当于单向链表节点的 next 指针为 NULL；如果该表项的值是 0xff7 则表示它是一个坏簇。这就是 **文件的链式存储**。

2.2.4 FAT12 根目录结构

怎样读取一个文件我们知道了，但是如何找到某个文件，即如何得到该文件对应的第一个簇呢？这就到目录结构派上用场的时候了，为了简单起见，我们这里只介绍根目录结构。

如图 2.11 所示，对于 FAT12/16，根目录存储在磁盘中固定的地方，紧跟在最后一个 FAT 表之后。根目录的扇区数也是固定的，可以根据 BPB_RootEntCnt 计算得出：

```
RootDirSectors = ((BPB_RootEntCnt * 32) + (BPB_BytsPerSec - 1)) / BPB_BytsPerSec
```

根目录的扇区号是相对于该 FAT 卷启动扇区的偏移量：

```
FirstRootDirSecNum = BPB_RsvdSecCnt + (BPB_NumFATs * BPB_FATSz16)
```

FAT 根目录其实就是一个由 32-bytes 的线性表构成的“文件”，其每一个条目代表着一个文件，这个 32-bytes 目录项的格式如图 2.2 所示。

表 2.2：根目录的条目格式

名称	偏移(bytes)	长度(bytes)	描述	举例(loader.bin)
DIR_Name	0	0xb	文件名 8 字节，扩展名 3 字节	"LOADER□□BIN"
DIR_Attr	0xb	1	文件属性	0x20
保留位	0xc	10	保留位	0
DIR_WrtTime	0x16	2	最后一次写入时间	0x7a5a
DIR_WrtDate	0x18	2	最后一次写入日期	0x3188
DIR_FstClus	0x1a	2	此目录项的开始簇编号	0x0002
DIR_FileSize	0x1c	4	文件大小	0x000000f

知道了这些，我们就得到了足够的信息去在磁盘上寻找某个文件，在磁盘根目录搜索并读取某个文件的步骤大致如下：

1. 确定根目录区的开始扇区和结束扇区；
2. 遍历根目录区，寻找与被搜索名相对应根目录项；
3. 找到该目录项对应的开始簇编号；

4. 以文件的开始簇为根据寻找整个文件的链接簇，并依次读取每个簇的内容。

2.3 让启动扇区加载引导文件

有了FAT12文件系统的相关知识之后，我们就可以跨越512字节的限制，从文件系统中加载文件并执行了。

2.3.1 一个最简单的loader

为做测试用，我们写一个最小的程序，让它显示一个字符，然后进入死循环，这样如果loader加载成功并成功执行的话，就能看到这个字符。

新建一个文件loader.S，内容如图2.12所示。

```

11 .code16
12 .text
13   mov    $0xb800,%ax
14   mov    %ax,%gs
15   mov    $0xf,%ah
16   mov    $'L',%al
17   mov    %ax,%gs:((80*0+39)*2)
18   jmp    .

```

Fig 2.12: 一个最简单的loader(chapter2/2/loader.S)

这个程序在连接时需要使用连接文件solrex_x86_dos.ld，如图2.13所示，这样能更改代码段的偏移量为0x0100。这样做的目的仅仅是为了与DOS系统兼容，可以用此代码生成在DOS下可调试的二进制文件。

```

11 SECTIONS
12 {
13   . = 0x0100;
14   .text :
15   {
16     _ftext = .;
17   } = 0
18 }

```

Fig 2.13: 一个最简单的loader(chapter2/2/solrex_x86_dos.ld)

2.3.2 读取软盘扇区的BIOS 13h号中断

我们知道了如何在磁盘上寻找一个文件，但是该如何将磁盘上内容读取到内存中去呢？我们在第2.1节中写的启动扇区不需要自己写代码来读取，是因为它每次都被加载到内存的固定位置，计算机在发现可启动标识0xaa55的时候自动就会做加载工作。但如果我们要自己从软盘上读取文件的时候，就需要使用到底层BIOS系统提供的磁盘读取功能了。这里，我们主要用到BIOS 13h号中断。

表 2.3 所示，就是 BIOS 13 号中断的参数表。从表中我们可以看到，读取磁盘驱动器所需要的参数是磁道（柱面）号、磁头号以及当前磁道上的扇区号三个分量。由第 1.2 节所介绍的磁盘知识，我们可以得到计算这三个分量的公式 2.1。

$$\frac{\text{扇区号}}{18(\text{每磁道扇区数})} = \begin{cases} \text{商 } Q = \begin{cases} \text{柱面号} = Q \gg 1 \\ \text{磁头号} = Q \& 1 \end{cases} \\ \text{余数 } R \Rightarrow \text{起始扇区号} = R + 1 \end{cases} \quad (2.1)$$

表 2.3：BIOS 13h 号中断的参数表

中断号	AH	功能	调用参数	返回参数
13	0	磁盘复位	DL = 驱动器号 00, 01 为软盘, 80h, 81h, … 为硬盘	失败： AH = 错误码
	1	读磁盘驱动器状态		AH=状态字节
	2	读磁盘扇区	AL = 扇区数 (CL) _{6,7} (CH) _{0~7} = 柱面号 (CL) _{0~5} = 扇区号 DH/DL = 磁头号/驱动器号 ES:BX = 数据缓冲区地址	读成功： AH = 0 AL = 读取的扇区数 读失败： AH = 错误码
	3	写磁盘扇区	同上	写成功： AH = 0 AL = 写入的扇区数 写失败： AH = 错误码
	4	检验磁盘扇区	AL = 扇区数 (CL) _{6,7} (CH) _{0~7} = 柱面号 (CL) _{0~5} = 扇区号 DH/DL = 磁头号/驱动器号	成功： AH = 0 AL = 检验的扇区数 失败： AH = 错误码
	5	格式化盘磁道	AL = 扇区数 (CL) _{6,7} (CH) _{0~7} = 柱面号 (CL) _{0~5} = 扇区号 DH/DL = 磁头号/驱动器号 ES:BX = 格式化参数表指针	成功： AH = 0 失败： AH = 错误码

知道了这些，我们就可以写一个读取软盘扇区的子函数了：

```

209 /* =====
210 Routine: ReadSector
211 Action: Read %cl Sectors from %ax sector(floppy) to %es:%bx(memory)

```

```

212     Assume sector number is 'x', then:
213         x/(BPB_SecPerTrk) = y,
214         x%(BPB_SecPerTrk) = z.
215     The remainder 'z' PLUS 1 is the start sector number;
216     The quotient 'y' divide by BPB_NumHeads(RIGHT SHIFT 1 bit)is cylinder
217         number;
218     AND 'y' by 1 can got magnetic header.
219 */
220 ReadSector:
221     push    %ebp
222     mov     %esp,%ebp
223     sub    $2,%esp      /* Reserve space for saving %cl */
224     mov     %cl,-2(%ebp)
225     push    %bx          /* Save bx */
226     mov     (BPB_SecPerTrk), %bl    /* %bl: the divider */
227     div    %bl           /* 'y' in %al, 'z' in %ah */
228     inc     %ah           /* z++, got start sector */
229     mov     %ah,%cl        /* %cl <- start sector number */
230     mov     %al,%dh        /* %dh <- 'y' */
231     shr    $1,%al        /* 'y'/BPB_NumHeads */
232     mov     %al,%ch        /* %ch <- Cylinder number(y>>1) */
233     and    $1,%dh        /* %dh <- Magnetic header(y&1) */
234     pop     %bx          /* Restore %bx */
235     /* Now, we got cylinder number in %ch, start sector number in %cl, magnetic
236         header in %dh. */
237     mov     (BS_DrvNum), %dl
238 GoOnReading:
239     mov     $2,%ah
240     mov     -2(%ebp),%al    /* Read %al sectors */
241     int    $0x13
242     jc    GoOnReading    /* If CF set 1, mean read error, reread. */
243     add    $2,%esp
244     pop     %ebp
245     ret
246

```

Fig 2.14: 读取软盘扇区的函数(节自chapter2/2/boot.S)

2.3.3 搜索 loader.bin

读取扇区的子函数写好了，下面我们编写在软盘中搜索 loader.bin 的代码：

```

62
63     /* Reset floppy */
64     xor     %ah,%ah
65     xor     %dl,%dl      /* %dl=0: floppy driver 0 */
66     int    $0x13          /* BIOS int 13h, ah=0: Reset driver 0 */
67
68     /* Find LOADER.BIN in root directory of driver 0 */
69     movw   $SecNoOfRootDir, (wSectorNo)
70
71 /* Read root dir sector to memory */

```

```
72 LABEL_SEARCH_IN_ROOT_DIR_BEGIN:  
73     cmpw    $0,(wRootDirSizeForLoop)    /* If searching in root dir */  
74     jz      LABEL_NO_LOADERBIN        /* can find LOADER.BIN ? */  
75     decw    (wRootDirSizeForLoop)  
76     mov     $BaseOfLoader,%ax  
77     mov     %ax,%es                  /* %es <- BaseOfLoader */  
78     mov     $OffsetOfLoader,%bx          /* %bx <- OffsetOfLoader */  
79     mov     (wSectorNo),%ax            /* %ax <- sector number in root */  
80     mov     $1,%cl  
81     call    ReadSector  
82     mov     $LoaderFileName,%si          /* %ds:%si -> LOADER BIN */  
83     mov     $OffsetOfLoader,%di          /* BaseOfLoader<<4+100*/  
84     cld  
85     mov     $0x10,%dx  
86  
87 /* Search for "LOADER BIN", FAT12 save file name in 12 bytes, 8 bytes for  
88 file name, 3 bytes for suffix, last 1 bytes for '\20'. If file name is  
89 less than 8 bytes, filled with '\20'. So "LOADER.BIN" is saved as:  
90 "LOADER BIN"(4f4c 4441 5245 2020 4942 204e).  
91 */  
92 LABEL_SEARCH_FOR_LOADERBIN:  
93     cmp     $0,%dx                  /* Read control */  
94     jz      LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR  
95     dec     %dx  
96     mov     $11,%cx  
97  
98 LABEL_CMP_FILENAME:  
99     cmp     $0,%cx  
100    jz      LABEL_FILENAME_FOUND    /* If 11 chars are all identical? */  
101    dec     %cx  
102    lodsb    /* %ds:(%si) -> %al */  
103    cmp     %es:(%di),%al  
104    jz      LABEL_GO_ON  
105    jmp     LABEL_DIFFERENT       /* Different */  
106  
107 LABEL_GO_ON:  
108    inc     %di  
109    jmp     LABEL_CMP_FILENAME    /* Go on loop */  
110  
111 LABEL_DIFFERENT:  
112    and     $0xffe0,%di            /* Go to head of this entry */  
113    add     $0x20,%di  
114    mov     $LoaderFileName,%si          /* Next entry */  
115    jmp     LABEL_SEARCH_FOR_LOADERBIN  
116  
117 LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR:  
118    addw   $1,(wSectorNo)  
119    jmp     LABEL_SEARCH_IN_ROOT_DIR_BEGIN  
120  
121 /* Not found LOADER.BIN in root dir. */  
122 LABEL_NO_LOADERBIN:  
123    mov     $2,%dh  
124    call    DispStr              /* Display string(index 2) */  
125    jmp     .                   /* Infinite loop */  
126
```

```

127 /* Found. */
128 LABEL_FILENAME_FOUND:
129     mov    $RootDirSectors,%ax
130     and    $0xffe0,%di          /* Start of current entry, 32 bytes per entry */
131     add    $0x1a,%di          /* First sector of this file */
132     mov    %es:(%di),%cx
133     push   %cx                /* Save index of this sector in FAT */
134     add    %ax,%cx
135     add    $DeltaSecNo,%cx      /* LOADER.BIN's start sector saved in %cl */
136     mov    $BaseOfLoader,%ax
137     mov    %ax,%es            /* %es <- BaseOfLoader */
138     mov    $OffsetOfLoader,%bx  /* %bx <- OffsetOfLoader */
139     mov    %cx,%ax            /* %ax <- Sector number */
140

```

Fig 2.15: 搜索 loader.bin 的代码片段(节自chapter2/2/boot.S)

这段代码的功能就是我们前面提到过的，遍历根目录的所有扇区，将每个扇区加载入内存，然后从中寻找文件名为 loader.bin 的条目，直到找到为止。找到之后，计算出 loader.bin 的起始扇区号。其中用到的变量和字符串的定义见图 2.16 中代码片段的定义。

```

12 .set  BaseOfStack,    0x7c00  /* Stack base address, inner */
13 .set  BaseOfLoader,   0x9000  /* Section loading address of LOADER.BIN */
14 .set  OffsetOfLoader, 0x0100  /* Loading offset of LOADER.BIN */
15 .set  RootDirSectors, 14      /* Root directory sector count */
16 .set  SecNoOfRootDir, 19      /* 1st sector of root directory */
17 .set  SecNoOfFAT1,    1        /* 1st sector of FAT1 */
18 .set  DeltaSecNo,     17      /* BPB_(RsvdSecCnt+NumFATs*FATSz) - 2 */

174
175 /* =====
176     Variable table
177 */
178 wRootDirSizeForLoop:    .2byte RootDirSectors
179 wSectorNo:              .2byte 0       /* Sector number to read */
180 bOdd:                  .byte 0       /* odd or even? */

181
182 /* =====
183     String table
184 */
185 LoaderFileName:         .asciz "LOADER BIN"      /* File name */
186 .set  MessageLength,9
187 BootMessage:            .ascii  "Booting***"    /* index 0 */
188 Message1:               .ascii  "Loaded in"      /* index 1 */
189 Message2:               .ascii  "No LOADER"      /* index 2 */
190

```

Fig 2.16: 搜索 loader.bin 使用的变量定义(节自chapter2/2/boot.S)

由于在代码中有一些打印工作，我们写了一个函数专门做这项工作。为了节省代码长度，被打印字符串的长度都设置为 9 字节，不够则用空格补齐，这样就相当于一个备用的二维数组，通过数字定位要

打印的字符串，很方便。打印字符串的函数 DispStr 见图 2.17，调用它的时候需要从寄存器 dh 传入参数字符串序号。

```

190
191 /* =====
192 Routine: DispStr
193 Action: Display a string, string index stored in %dh
194 */
195 DispStr:
196     mov    $MessageLength, %ax
197     mul    %dh
198     add    $BootMessage,%ax
199     mov    %ax,%bp           /* String address */
200     mov    %ds,%ax
201     mov    %ax,%es
202     mov    $MessageLength,%cx /* String length */
203     mov    $0x1301,%ax        /* ah = 0x13, al = 0x01(W) */
204     mov    $0x07,%bx          /* PageNum 0(bh = 0), bw(bl= 0x07)*/
205     mov    $0,%dl             /* Start row and column */
206     int    $0x10               /* BIOS INT 10h, display string */
207     ret
208

```

Fig 2.17: 打印字符串函数 DispStr (节自chapter2/2/boot.S)

2.3.4 加载 loader 入内存

在寻找到 loader.bin 之后，就需要把它装入内存。现在我们已经有了 loader.bin 的起始扇区号，利用这个扇区号可以做两件事：一，把起始扇区装入内存；二，通过它找到 FAT 中的条目，从而找到 loader.bin 文件所占用的其它扇区。

这里，我们把 loader.bin 装入内存中的 BaseOfLoader:OffsetOfLoader 处，但是在图 2.15 中我们将根目录区也是装载到这个位置。因为在找到 loader.bin 之后，该内存区域对我们已经没有用处了，所以它尽可以被覆盖。

我们已经知道了如何装入一个扇区，但是从 FAT 表中寻找其它的扇区还是一件麻烦的事情，所以我们写了一个函数 GetFATEEntry 来专门做这件事情，函数的输入是扇区号，输出是其对应的 FAT 项的值，见图 2.18。

```

246
247 /* =====
248 Routine: GetFATEEntry
249 Action: Find %ax sector's index in FAT, save result in %ax
250 */
251 GetFATEEntry:
252     push    %es
253     push    %bx
254     push    %ax
255     mov    $BaseOfLoader,%ax
256     sub    $0x0100,%ax

```

```

257    mov    %ax,%es      /* Left 4K bytes for FAT */
258    pop    %ax
259    mov    $3,%bx
260    mul    %bx          /* %dx:%ax = %ax*3 */
261    mov    $2,%bx
262    div    %bx          /* %dx:%ax/2 */
263    movb   %dl, (bOdd)   /* store remainder %dx in label bOdd. */
264
265 LABEL_EVEN:
266    xor    %dx,%dx      /* Now %ax is the offset of FATEEntry in FAT */
267    mov    (BPB_BytsPerSec),%bx
268    div    %bx          /* %dx:%ax/BPB_BytsPerSec */
269    push   %dx
270    mov    $0,%bx
271    add    $SecNoOfFAT1,%ax /* %ax <- FATEEntry's sector */
272    mov    $2,%cl          /* Read 2 sectors in 1 time, because FATEEntry */
273    call   ReadSector     /* may be in 2 sectors. */
274    pop    %dx
275    add    %dx,%bx
276    mov    %es:(%bx),%ax  /* read FAT entry by word(2 bytes) */
277    cmpb   $0,(bOdd)      /* remainder %dx(see above) == 0 ?*/
278    jz    LABEL_EVEN_2    /* NOTE: %ah: high address byte, %al: low byte */
279    shr    $4,%ax
280
281 LABEL_EVEN_2:
282    and    $0x0fff,%ax
283
284 LABEL_GET_FAT_ENTRY_OK:
285    pop    %bx
286    pop    %es
287    ret
288
289 .org 510      /* Skip to address 0x510. */
290 .2byte 0xaa55  /* Write boot flag to 1st sector(512 bytes) end */
291

```

Fig 2.18: 寻找 FAT 项的函数 GetFATEEntry (节自chapter2/2/boot.S)

这里有一个对扇区号判断是奇是偶的问题，因为 FAT12 的每个表项是 12 位，即 1.5 个字节，而我们这里是用字（2 字节）来读每个表项的，那么读到的表项可能在高 12 位或者低 12 位，就要用扇区号的奇偶来判断应该取哪 12 位。由于扇区号 $*3/2$ 的商就是对应表项的偏移量，余数代表着是否多半个字节，如果存在余数 1，则取高 12 位为表项值；如果余数为 0，则取低 12 位作为表项值。举个具体的例子，下面是一个真实的 FAT12 表项内容（两行是分别用字节和字来表示的结果）：

```

0000200: 00 00 00 00 40 00 FF 0F
0000200: 0000 0000 0040 OFFF

```

我们来找扇区号 3 对应的 FAT12 表项。 $3*3/2 = 4...1$ ，按照字来读取偏移 4 对应的地址，我们得到 0040。由于扇区号 3 是奇数，有余数，则应取高 12 位作为 FAT12 表项内容，将 0040 右移 4 位再算术与 0x0fff，我们得到 0x0004，即为对应的 FAT12 表项，说明扇区号 3 的后继扇区号是 4。

然后，我们就可以将 loader.bin 整个文件加载到内存中去了，见图 2.19。

```

140
141 /* Load LOADER.BIN's sector's to memory. */
142 LABEL_GOON_LOADING_FILE:
143     push    %ax
144     push    %bx
145     mov     $0x0e,%ah
146     mov     '$.',%al    /* Char to print */
147     mov     $0x0f,%bl    /* Front color: white */
148     int     $0x10        /* BIOS int 10h, ah=0xe: Print char */
149     pop     %bx
150     pop     %ax
151
152     mov     $1,%cl
153     call    ReadSector
154     pop     %ax          /* Got index of this sector in FAT */
155     call    GetFATEntry
156     cmp     $0x0fff,%ax
157     jz     LABEL_FILE_LOADED
158     push    %ax          /* Save index of this sector in FAT */
159     mov     $RootDirSectors,%dx
160     add     %dx,%ax
161     add     $DeltaSecNo,%ax
162     add     (BPB_BytsPerSec),%bx
163     jmp     LABEL_GOON_LOADING_FILE
164
165 LABEL_FILE_LOADED:
166     mov     $1,%dh
167     call    DispStr      /* Display string(index 1) */
168

```

Fig 2.19: 加载 loader.bin 的代码(节自chapter2/2/boot.S)

在图 2.19 中我们看到一个宏 DeltaSectorNo，这个宏就是为了将 FAT 中的簇号转换为扇区号。由于根目录区的开始扇区号是 19，而 FAT 表的前两个项 0,1 分别是磁盘识别字和被保留，其表项其实是从第 2 项开始的，第 2 项对应着根目录区后的第一个扇区，所以扇区号和簇号的对应关系就是：

$$\begin{aligned}\text{扇区号} &= \text{簇号} + \text{根目录区占用扇区数} + \text{根目录区开始扇区号} - 2 \\ &= \text{簇号} + \text{根目录区占用扇区数} + 17\end{aligned}$$

这就是 DeltaSectorNo 的值 17 的由来。

2.3.5 向 loader 转交控制权

我们已经将 loader 成功地加载入了内存，然后就需要进行一个跳转，来执行 loader。

```

168
169 ****
170     Jump to LOADER.BIN's start address in memory.
171 */

```

```

172     jmp      $BaseOfLoader,$OffsetOfLoader
173 /****** */
174

```

Fig 2.20: 跳转到 loader 执行(节自chapter2/2/boot.S)

2.3.6 生成镜像并测试

我们写好了汇编源代码，那么就需要将源代码编译成可执行文件，并生成软盘镜像了。

```

11 CC=gcc
12 LD=ld
13 OBJCOPY=objcopy
14
15 CFLAGS=-c
16 TRIM_FLAGS=-R .pdr -R .comment -R.note -S -O binary
17
18 LDFILE_BOOT=solrex_x86_boot.ld
19 LDFILE_DOS=solrex_x86_dos.ld
20 LDFLAGS_BOOT=-T$(LDFILE_BOOT)
21 LDFLAGS_DOS=-T$(LDFILE_DOS)
22
23 all: boot.img LOADER.BIN
24 @echo '# #####'
25 @echo '# Compiling work finished, now you can use "sudo make copy" to'
26 @echo '# copy LOADER.BIN into boot.img'
27 @echo '# #####'
28
29 boot.bin: boot.S
30 $(CC) $(CFLAGS) boot.S
31 $(LD) boot.o -o boot.elf $(LDFLAGS_BOOT)
32 $(OBJCOPY) $(TRIM_FLAGS) boot.elf $@
33
34 LOADER.BIN: loader.S
35 $(CC) $(CFLAGS) loader.S
36 $(LD) loader.o -o loader.elf $(LDFLAGS_DOS)
37 $(OBJCOPY) $(TRIM_FLAGS) loader.elf $@
38
39 boot.img: boot.bin
40 @dd if=boot.bin of=boot.img bs=512 count=1
41 @dd if=/dev/zero of=boot.img skip=1 seek=1 bs=512 count=2879
42

```

Fig 2.21: 用 Makefile 编译(节自chapter2/2/Makefile)

上面的代码比较简单，我们可以通过一个 make 命令编译生成 boot.img 和 LOADER.BIN：

```

$ make
gcc -c boot.S
ld boot.o -o boot.elf -Tsolrex_x86_boot.ld
objcopy -R .pdr -R .comment -R.note -S -O binary boot.elf boot.bin

```

```

1+0 records in
1+0 records out
512 bytes (512 B) copied, 3.5761e-05 s, 14.3 MB/s
2879+0 records in
2879+0 records out
1474048 bytes (1.5 MB) copied, 0.0132009 s, 112 MB/s
gcc -c loader.S
ld loader.o -o loader.elf -Tsolrex_x86_dos.ld
objcopy -R .pdr -R .comment -R.note -S -O binary loader.elf LOADER.BIN
#####
# Compiling work finished, now you can use "sudo make copy" to
# copy LOADER.BIN into boot.img
#####

```

由于我们的目标就是让启动扇区加载引导文件，所以需要把引导文件放入软盘镜像中。那么如何将 LOADER.BIN 放入 boot.img 中呢？我们只需要挂载 boot.img 并将 LOADER.BIN 拷贝进入被挂载的目录，为此我们在 Makefile 中添加新的编译目标 copy：

```

42
43 # You must have the authority to do mount, or you must use "su root" or
44 # "sudo" command to do "make copy"
45 copy: boot.img LOADER.BIN
46   @mkdir -p /tmp/floppy; \
47   mount -o loop boot.img /tmp/floppy/ -o fat=12; \
48   cp LOADER.BIN /tmp/floppy/; \
49   umount /tmp/floppy/; \
50   rm -rf /tmp/floppy/;
51

```

Fig 2.22: 拷贝 LOADER.BIN 入 boot.img(节自chapter2/2/Makefile)

由于挂载软盘镜像在很多 Linux 系统上需要 root 权限，所以我们没有将 copy 目标添加到 all 的依赖关系中。在执行 make copy 命令之前我们必须先获得 root 权限。

```

$ su
Password:
# make copy
mkdir -p /tmp/floppy; \
mount -o loop boot.img /tmp/floppy/ -o fat=12; \
cp LOADER.BIN /tmp/floppy/; \
umount /tmp/floppy/; \
rm -rf /tmp/floppy/;

```

如果仅仅生成 boot.img 而不将 loader.bin 装入它，用这样的软盘启动会显示找不到 LOADER：

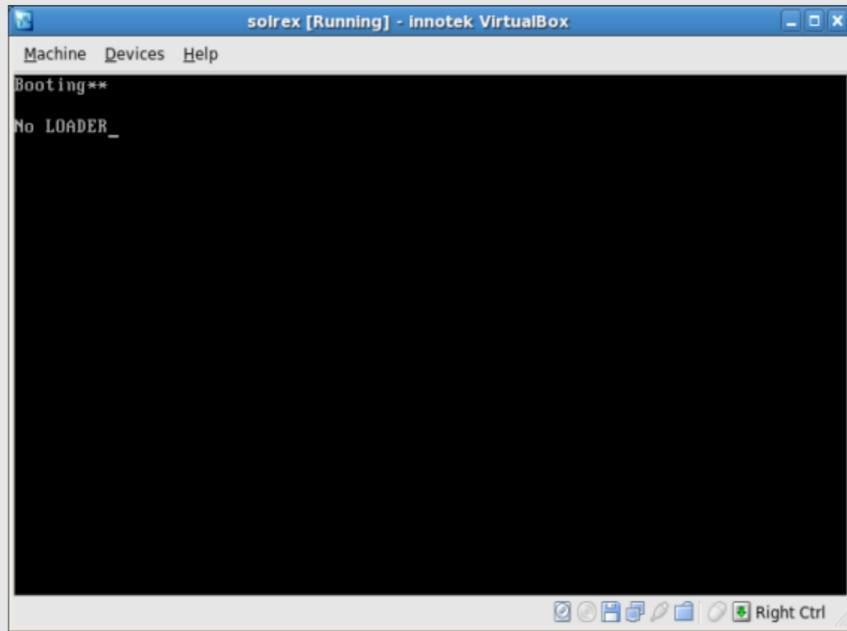


Fig 2.23: 没有装入 LOADER.BIN 的软盘启动

装入 loader.bin 之后再用 boot.img 启动，我们看到虚拟机启动并在屏幕中间打印出了一个字符 ”L“，这说明我们前面的工作都是正确的。

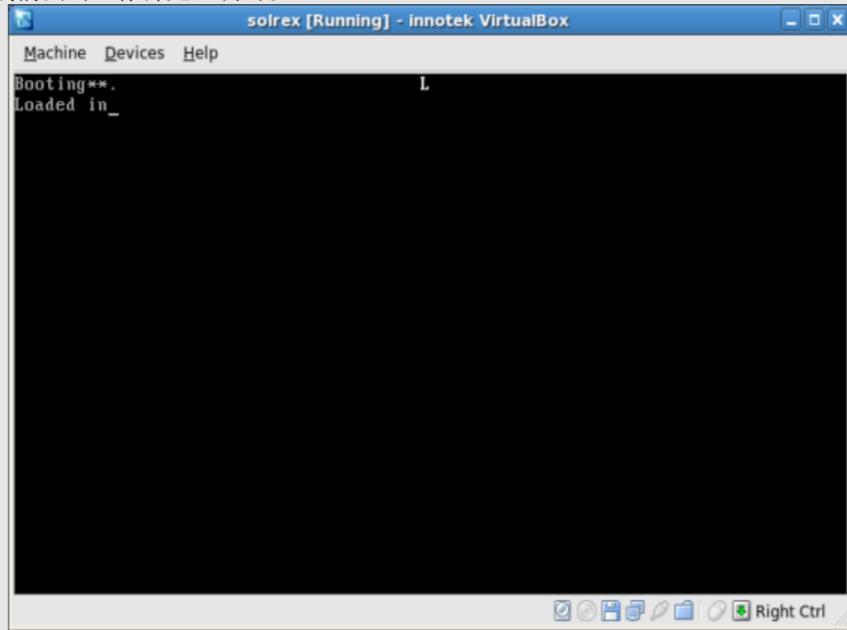


Fig 2.24: 装入了 LOADER.BIN 以后再启动

CHAPTER 3

进入保护模式

前面我们看到，通过一些很简单的代码，我们做到了启动一个微型系统，加载文件系统中的文件进入内存并运行的功能。应该注意的是，在前面的代码中我们使用的内存空间都很小。我们看一下 boot.bin 和 LOADER.BIN 的大小就能感觉出来（当然，可执行文件小未必使用内存空间小，但是这两个文件也太小了 ②）。

```
$ ls -l boot.bin LOADER.BIN
-rwxr-xr-x 1 solrex solrex 512 2008-04-26 16:34 boot.bin
-rwxr-xr-x 1 solrex solrex 15 2008-04-26 16:34 LOADER.BIN
```

boot.bin 是 512 个字节（其中还有我们填充的内容，实际指令只有 480 个字节），而 LOADER.BIN 更过分，只有 15 个字节大小。可想而知这两个文件在内存中能使用多大的空间吧。如果读者有些汇编语言经验的话，就会发现我们在前面的程序中使用的存储器寻址都是在实模式下进行的，即：由段寄存器（cs, ds: 16-bit）配合段内偏移地址（16-bit）来定位一个实际的 20-bit 物理地址，所以我们前面的程序最多支持 $2^{20} = 2^{10} * 2^{10} = 1024 * 1024$ bytes = 1MB 的寻址空间。

哇，1MB 不小了，我们的操作系统加一起连 1KB 都用不到，1MB 寻址空间足够了。但是需要考虑到的一点是，就拿我们现在用的 1.44MB 的（已经被淘汰的）软盘标准来说，如果软盘上某个文件超过 1MB，我们的操作系统就没办法处理了。那么如果以后把操作系统安装到硬盘上之后呢？我们就没办法处理稍微大一点的文件了。

所以我们要从最原始的 Intel 8086/8088 CPU 的实模式中跳出来，进入 Intel 80286 之后系列 CPU 给我们提供的保护模式。这还将为我们带来其它更多的好处，具体内容请继续往下读。

3.1 实模式和保护模式



如果您需要更详细的知识，也许您更愿意去读 Intel 的手册，本节内容主要集中 在：[Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide](#), 第 2 章和第 3 章。

3.1.1 一段历史

Intel 公司在 1978 年发布了一款 16 位字长 CPU: 8086，最高主频 5 MHz~10 MHz，集成了 29,000 个晶体管，这款在今天感觉像玩具一样的 CPU 却是奠定今天 Intel PC 芯片市场地位的最重要的产品之一。虽然它的后继者 8088，加强版的 8086（增加了一个 8 比特的外部总线）才是事实上的 IBM 兼容机（PC，个人电脑）雏形的核心，但人们仍然习惯于用 8086 作为厂商标志代表 Intel。

因为受到字长（16 位）的限制，如果仅仅使用单个寄存器寻址，8086 仅仅能访问 $64KB(2^{16})$ 的地址空间，这显然不能满足一般要求，而当时 $1MB(2^{20})$ 对于一般的应用就比较足够了，所以 8086 使用了 20 位的地址线。

在 8086 刚发布的时候，没有“实模式”这个说法，因为当时的 Intel CPU 只有一种模式。在 Intel 以后的发布中，80286 引入了“保护模式”寻址方式，将 CPU 的寻址范围扩大到 $16(2^{24}) MB$ ，但是 80286 仍然是一款 16 位 CPU，这就限制了它的广泛应用。但是“实模式”这个说法，就从 80286 开始了。

接下来的发展就更快了，1985 年发布的 i386 首先让 PC CPU 进入了 32 位时代，由此而带来的好处显而易见，寻址能力大大增强，但是多任务处理和虚拟存储器的需求仍然推动着 i386 向更完善的保护模式发展。下面我们来了解一下“实模式”和“保护模式”的具体涵义。

3.1.2 实模式

实模式（real mode），有时候也被成为实地址模式（real address mode）或者兼容模式（compatibility mode）是 Intel 8086 CPU 以及以其为基础发展起来的 x86 兼容 CPU 采用的一种操作模式。其主要的特点有：20 比特的分段访问的内存地址空间（即 1 MB 的寻址能力）；程序可直接访问 BIOS 中断和外设；硬件层不支持任何内存保护或者多任务处理。80286 之后所有 x86 CPU 在加电自举时都是首先进入实模式；80186 以及之前的 CPU 只有一种操作模式，相当于实模式。

3.1.3 保护模式

保护模式（protected mode），有时候也被成为保护的虚拟地址模式（protected virtual address mode），也是一种 x86 兼容 CPU 的工作模式。保护模式为系统软件实现虚拟内存、分页机制、安全的多任务处理的功能支持，还有其它为操作系统提供的对应用程序的控制功能支持，比如：特权级、实模式应用程序兼容、虚拟 8086 模式。

3.1.4 实模式和保护模式的寻址模式

前面提到过，实模式下的地址线是 20 位的，所以实模式下的寻址模式使用分段方式来解决 16 位字长机器提供 20 位地址空间的问题。这个分段方法需要程序员在编制程序的过程中将存储器划分成段，每个段内的地址空间是线性增长的，最大可达 $64K(2^{16})$ ，这样段内地址就可以使用 16 位表示。段基址（20-bit）的最低 4 位必须是 0，这样段基址就可以使用 16 位段地址来表示，需要时将段地址左移 4 位就得到段起始地址。除了便于寻址之外，分段还有一个好处，就是将程序的代码段、数据段和堆栈段等隔离开，避免相互之间产生干扰。

当计算某个单元的物理地址时，比如汇编语言中的一个 Label，就通过段地址（16-bit）左移 4 位得到段基址（20-bit），再加上该单元（Label）的段内偏移量（16-bit）来得到其物理地址（20-bit），如图 3.1a 所示。

一般情况下，段地址会被放在四个段寄存器中，即：代码段 CS，数据段 DS，堆栈段 SS 和附加段 ES 寄存器。这样在加载数据或者控制程序运行的时候，只需要一个偏移量参数，CPU 会自动用对应段

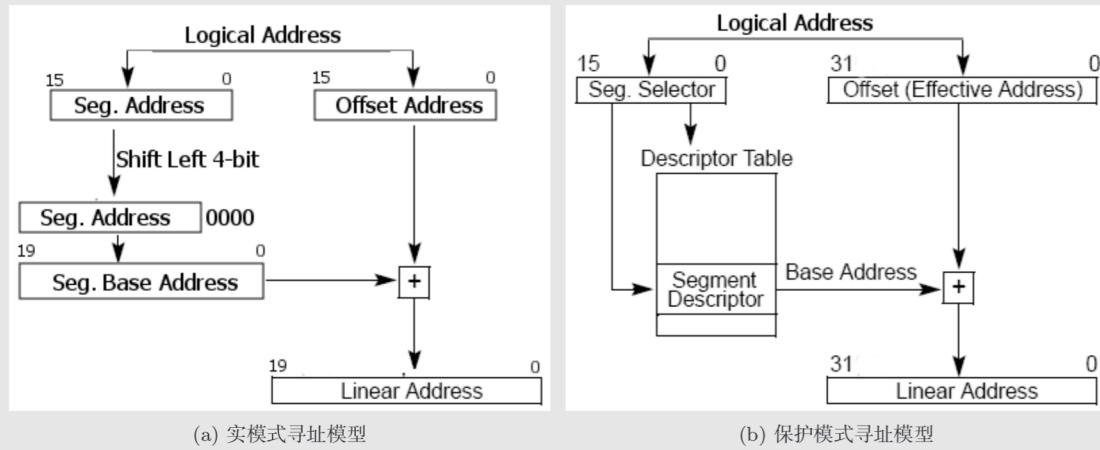


Fig 3.1: 实模式与保护模式寻址模型比较

的起始地址加上偏移量参数来得到需要的地址。（后继 CPU 又加上了两个段寄存器 FS 和 GS，不过使用方式是基本一样的。）

由此可见，实模式的寻址模式是很简单的，就是用两个 16 位逻辑地址（段地址：偏移地址）组合成一个 20 位物理地址，而保护模式的寻址方式就要稍微复杂一点了。

Intel 的 CPU 在保护模式下是可以选择打开分页机制的，但为了简单起见，我们先不开启分页机制，所以下面的讲解针对只有分段机制的保护模式展开。

在保护模式下，每个单元的物理地址仍然是由逻辑地址表示，但是这个逻辑地址不再由（段地址：偏移地址）组成了，而是由（段选择子：偏移地址）表示。这里的偏移地址也变成了 32 位的，所以段空间也比实模式下大得多。偏移地址的意思和实模式下并没有本质不同，但段地址的计算就要复杂一些了，如图 3.1b 所示。段基址（Segment Base Address）被存放在段描述符（Segment Descriptor）中，GDT（Global Descriptor Table，全局段选择子表）是保存着所有段选择子的信息，段选择子（Segment Selector）是一个指向某个段选择子的索引。

如图 3.1b 所示，当我们计算某个单元的物理地址时，只需要给出（段选择子：偏移地址），CPU 会从 GDT 中按照段选择子找到对应的段描述符，从段描述符中找出段基址，将段基址加上偏移量，就得到了该单元的物理地址。

3.2 与保护模式初次会面

介绍完了保护模式和实模式的不同，下面我们就尝试一下进入保护模式吧。在上一章我们已经实现了用启动扇区加载引导文件，所以这里我们就不用再去管启动扇区的事情了，下面的修改均在 loader.S 中进行。上一章的 loader.S 仅仅实现在屏幕的上方中间打印了一个 L，下面我们的 loader.S 要进入保护模式来打印一些新东西。

首先，我们来理清一下应该如何进入保护模式：

1. 我们需要一个 GDT。由于保护模式的寻址方式是基于 GDT 的，我们得自己写一个 GDT 数据结构并将其载入到系统中。

2. 我们需要为进入保护模式作准备。由于保护模式和实模式运行方式不同，在进入保护模式之前，我们需要一些准备工作。
3. 我们需要一段能在保护模式下运行的代码demo，以提示我们成功进入了保护模式。

下面我们就来一步步完成我们的第一个保护模式 loader。

3.2.1 GDT 数据结构

要写 GDT，首先得了解 GDT 的数据结构。GDT 实际上只是一个存储段描述符的线性表（可以理解成一个段描述符数组），对它的要求是其第一个段描述符置为空，因为处理机不会去处理第一个段描述符，所以理解 GDT 的数据结构难点主要在于理解段描述符的数据结构。

段描述符主要用来为处理机提供段位址，段访问控制和状态信息。图 3.2 显示了一个基本的段描述符结构：

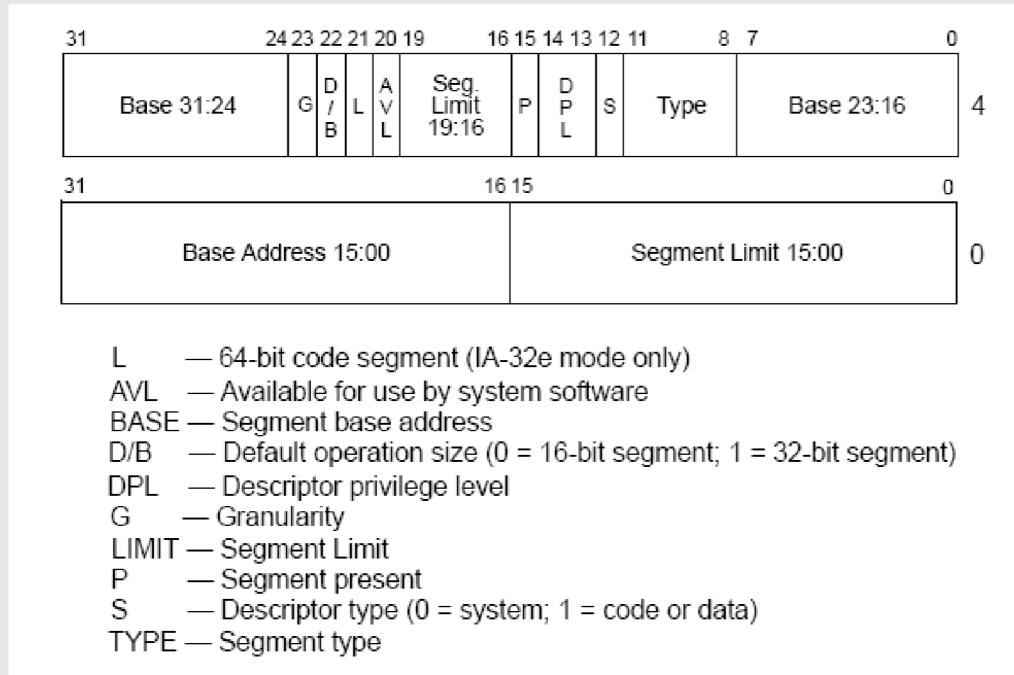


Fig 3.2: 段描述符

看到上面那么多内容，是不是感觉有点儿恐怖啊！其实简单的来看，我们现在最关注的是段基址，就是图 3.2 中标记为 Base 的部分。可以看到，段基址在段描述符中被分割为三段存储，分别是：Base 31:24, Base 23:16, Base Address 15:0，把这三段拼起来，我们就得到了一个 32 位的段基址。

有了段基址，就需要有一个界限来避免程序跑丢发生段错误，这个界限就是图 3.2 中标记为 Limit 的部分，将 Seg. Limit 19:16 和 Segment Limit 15:0 拼起来我们就得到了一个 20 位的段界限，这个界限就是应该是段需要的长度了。

下面还要说的就是那个 D/B Flag，D/B 代表 Default Operation Size，0 代表 16 位的段，1 代表 32 位的段。为了充分利用 CPU，我们当然要设置为 32 位模式了。剩下那些乱七八糟的 Flag 呢，无非就是提供段的属性（代码段还是数据段？只读还是读写？），我们将在第 3.3.2 节为大家详细介绍。

这些东西那么乱，难道要每次一点儿一点儿地计算吗？放心，程序员自有办法，请看下面的程序：

```

56 /* MACROS */
57
58 /* Segment Descriptor data structure.
59   Usage: Descriptor Base, Limit, Attr
60   Base: 4byte
61   Limit: 4byte (low 20 bits available)
62   Attr: 2byte (lower 4 bits of higher byte are always 0) */
63 .macro Descriptor Base, Limit, Attr
64     .2byte \Limit & 0xFFFF
65     .2byte \Base & 0xFFFF
66     .byte  (\Base >> 16) & 0xFF
67     .2byte ((\Limit >> 8) & 0xFO0) | (\Attr & 0xF0FF)
68     .byte  (\Base >> 24) & 0xFF
69 .endm
70

```

Fig 3.3: 自动生成段描述符的宏定义(节自chapter3/1/pm.h)

图 3.3 中所示，就是自动生成段描述符的汇编宏定义。我们只需要给宏 Descriptor 三个参数：Base（段基址），Limit（段界限[段长度]），Attr（段属性），Descriptor 就会自动将三者展开放到段描述符中对应的位置。看看我们在程序中怎么使用这个宏：

```

21
22 /* Global Descriptor Table */
23 LABEL_GDT:      Descriptor 0,           0, 0
24 LABEL_DESC_CODE32: Descriptor 0,        (SegCode32Len - 1), (DA_C + DA_32)
25 LABEL_DESC_VIDEO: Descriptor 0xB8000,    0xffff, DA_DRW
26

```

Fig 3.4: 自动生成段描述符的宏使用示例(节自chapter3/1/loader.S)

图 3.4 中，就利用 Descriptor 宏生成了三个段描述符，形成了一个 GDT。注意到没有，第一个段描述符是空的（参数全为 0）。这里 LABEL_DESC_CODE32 的段基址为 0 是因为我们无法确定它的准确位置，它将在运行期被填入。

有人可能会产生疑问，段基址和段界限什么意思我们都知道了，那段属性怎么回事呢？DA_C，DA_32，DA_DRW 都是什么东西啊？是这样的，为了避免手动一个一个置段描述符中的 Flag，我们预先定义了一些常用属性，用的时候只需要将这些属性加起来作为宏 Descriptor 的参数，就能将段描述符中的所有 flag 置上（记得 C 语言中 fopen 的参数吗？）。这些属性的定义如下（没必要细看，用的时候再找即可）：

¹¹ /* Comments below accords to "Chapter 3.4.5: Segment Descriptors" of "Intel
12 64 and IA-32 Arch. SW Developer's Manual: Volume 3A: System Programming

```

13     Guide". */
14
15 /* GDT Descriptor Attributes
16     DA_ : Descriptor Attribute
17     D   : Data Segment
18     C   : Code Segment
19     S   : System Segment
20     R   : Read-only
21     RW  : Read/Write
22     A   : Access */
23 .set    DA_32, 0x4000 /* 32-bit segment */
24
25 /* Descriptor privilege level */
26 .set    DA_DPL0, 0x00 /* DPL = 0 */
27 .set    DA_DPL1, 0x20 /* DPL = 1 */
28 .set    DA_DPL2, 0x40 /* DPL = 2 */
29 .set    DA_DPL3, 0x60 /* DPL = 3 */
30
31 /* GDT Code- and Data-Segment Types */
32 .set    DA_DR, 0x90 /* Read-Only */
33 .set    DA_DRW, 0x92 /* Read/Write */
34 .set    DA_DRWA, 0x93 /* Read/Write, accessed */
35 .set    DA_C, 0x98 /* Execute-Only */
36 .set    DA_CR, 0x9A /* Execute/Read */
37 .set    DA_CCO, 0x9C /* Execute-Only, conforming */
38 .set    DA_CCOR, 0x9E /* Execute/Read-Only, conforming */
39
40 /* GDT System-Segment and Gate-Descriptor Types */
41 .set    DA_LDT, 0x82 /* LDT */
42 .set    DA_TaskGate, 0x85 /* Task Gate */
43 .set    DA_386TSS, 0x89 /* 32-bit TSS(Available) */
44 .set    DA_386CGate, 0x8C /* 32-bit Call Gate */
45 .set    DA_386IGate, 0x8E /* 32-bit Interrupt Gate */
46 .set    DA_386TGate, 0x8F /* 32-bit Trap Gate */
47
48 /* Selector Attributes */
49 .set    SA_RPL0, 0
50 .set    SA_RPL1, 1
51 .set    SA_RPL2, 2
52 .set    SA_RPL3, 3
53 .set    SA_TIG, 0
54 .set    SA_TIL, 4
55

```

Fig 3.5: 预先设置的段属性(节自chapter3/1/pm.h)

3.2.2 保护模式下的 demo

为什么把这节提前到第 3.2.3 节前讲呢？因为要写入 GDT 正确的段描述符，首先要知道段的信息，我们就得先准备好这个段：

```

82 LABEL_SEG_CODE32:
83 .code32
84     mov    $(SelectorVideo), %ax
85     mov    %ax, %gs           /* Video segment selector(dest) */
86
87     movl   $((80 * 10 + 0) * 2), %edi
88     movb   $0xC, %ah          /* 0000: Black Back 1100: Red Front */
89     movb   '$P', %al
90
91     mov    %ax, %gs:(%edi)
92
93     /* Stop here, infinite loop. */
94     jmp    .
95
96 /* Get the length of 32-bit segment code. */
97 .set   SegCode32Len, . - LABEL_SEG_CODE32

```

Fig 3.6: 第一个在保护模式下运行的demo(节自chapter3/1/loader.S)

其实这个段的作用很简单，通过操纵视频段数据，在屏幕中间打印一个红色的”P”（和我们前面使用 BIOS 中断来打印字符的方式有所不同）。

3.2.3 加载 GDT

GDT 所需要的信息我们都知道了，GDT 表也通过图 3.4 中的代码实现了。那么，我们应该向 GDT 中填入缺少的信息，然后载入 GDT 了。将 GDT 载入处理机是用 lgdt 汇编指令实现的，但是 lgdt 指令需要存放 GDT 的基址和界限的指针作参数，所以我们还需要知道 GDT 的位置和 GDT 的界限：

```

17 /* NOTE! Wenbo-20080512: Actually here we put the normal .data section into
18   the .code section. For application SW, it is not allowed. However, we are
19   writing an OS. That is OK. Because there is no OS to complain about
20   that behavior. :) */
21
22 /* Global Descriptor Table */
23 LABEL_GDT:      Descriptor 0,          0, 0
24 LABEL_DESC_CODE32: Descriptor 0,      (SegCode32Len - 1), (DA_C + DA_32)
25 LABEL_DESC_VIDEO: Descriptor 0xB8000, 0xffff, DA_DRW
26
27 .set GdtLen, (. - LABEL_GDT) /* GDT Length */
28
29 GdtPtr: .2byte (GdtLen - 1) /* GDT Limit */
30     .4byte 0             /* GDT Base */
31
32 /* GDT Selector */
33 .set   SelectorCode32, (LABEL_DESC_CODE32 - LABEL_GDT)
34 .set   SelectorVideo,  (LABEL_DESC_VIDEO - LABEL_GDT)
35
36 /* Program starts here. */
37 LABEL_BEGIN:
38     mov    %cs, %ax     /* Move code segment address(CS) to data segment */

```

```

39    mov    %ax, %ds    /* register(DS), ES and SS. Because we have      */
40    mov    %ax, %es    /* embedded .data section into .code section in  */
41    mov    %ax, %ss    /* the start(mentioned in the NOTE above).          */
42
43    mov    $0x100, %sp
44
45    /* Initialize 32-bits code segment descriptor. */
46    xor    %eax, %eax
47    mov    %cs, %ax
48    shl    $4, %eax
49    addl   $(LABEL_SEG_CODE32), %eax
50    movw  %ax, (LABEL_DESC_CODE32 + 2)
51    shr    $16, %eax
52    movb  %al, (LABEL_DESC_CODE32 + 4)
53    movb  %ah, (LABEL_DESC_CODE32 + 7)
54
55    /* Prepared for loading GDTR */
56    xor    %eax, %eax
57    mov    %ds, %ax
58    shl    $4, %eax
59    add    $(LABEL_GDT), %eax      /* eax <- gdt base*/
60    movl  %eax, (GdtPtr + 2)
61
62    /* Load GDTR(Global Descriptor Table Register) */
63    lgdtw GdtPtr

```

Fig 3.7: 加载 GDT(节自chapter3/1/loader.S)

图 3.7 中 GdtPtr 所指，即为 GDT 的界限和基址所存放位置。某段描述符对应的 GDT 选择子，就是其段描述符相对于 GDT 基址的索引（在我们例子里 GDT 基址为 LABEL_GDT 指向的位置）。这里需要注意的是，虽然我们在代码中写：

```
.set SelectorCode32, (LABEL_DESC_CODE32 - LABEL_GDT)
```

但实际上段选择子在使用时需要右移 3 个位作为索引去寻找其对应的段描述符，段选择子的右侧 3 个位是为了标识 TI 和 RPL 的，如图 3.12 所示，这点我们将在第 3.3.1 节和第 3.3.2 节中详细介绍。但是这里为什么能直接用地址相减得到段选择子呢？因为段描述符的大小是 8 个字节，用段描述符的地址相减的话，地址差的最右侧三个位就默认置 0 了。

在图 3.7 中所示的代码，主要干了两件事：第一，将图 3.6 所示 demo 的段基址放入 GDT 中对应的段描述符中；第二，将 GDT 的基址放到 GdtPtr 所指的数据结构中，并加载 GdtPtr 所指的数据结构到 GDTR 寄存器中（使用 lgdt 指令）。

3.2.4 进入保护模式

进入保护模式前，我们需要将中断关掉，因为保护模式下中断处理的机制和实模式是不一样的，不关掉中断可能带来麻烦。使用 cli 汇编指令可以清除所有中断 flag。

由于实模式下仅有 20 条地址线：A0, A1, ..., A19，所以当我们要进入保护模式时，需要打开 A20 地址线。打开 A20 地址线有至少三种方法，我们这里采用 IBM 使用的方法，通常被称为：“Fast

A20 Gate”，即修改系统控制端口 92h，因为其端口的第 1 位控制着 A20 地址线，所以我们只需要将 0b00000010 赋给端口 92h 即可。

当前面两项工作完成后，我们就可以进入保护模式了。方法很简单，将 cr0 寄存器的第 0 位 PE 位置为 1 即可使 CPU 切换到保护模式下运行。

```

64      /* Clear Interrupt Flags */
65      cli
66
67
68      /* Open A20 line. */
69      inb    $0x92, %al
70      orb    $0b00000010, %al
71      outb   %al, $0x92
72
73      /* Enable protect mode, PE bit of CRO. */
74      movl   %cr0, %eax
75      orl    $1, %eax
76      movl   %eax, %cr0
77

```

Fig 3.8: 进入保护模式(节自chapter3/1/loader.S)

3.2.5 特别的混合跳转指令

虽然已经进入了保护模式，但由于我们的 CS 寄存器存放的仍然是实模式下 16 位的段信息，要跳转到我们的 demo 程序并不是那么简单的事情。因为 demo 程序是 32 位的指令，而我们现在仍然运行的是 16 位的指令。从 16 位的代码段中跳转到 32 位的代码段，不是一般的 near 或 far 跳转指令能解决得了的，所以这里我们需要一个特别的跳转指令。在这条指令运行之前，所有的指令都是 16 位的，在它运行之后，就变成 32 位指令的世界。

在 Intel 的手册中，把这条混合跳转指令称为 far jump(ptr16:32)，在 NASM 手册中，将这条指令称为 Mixed-Size Jump，我们就沿用 NASM 的说法，将这条指令称为混合字长跳转指令。NASM 提供了这条指令的汇编语言实现：

```
jmp dword 0x1234:0x56789ABC
```

NASM 的手册中说 GAS 没有提供这条指令的实现，我就用 .byte 伪代码直接写了二进制指令：

```

/* Mixed-Size Jump. */
.2byte 0xea66
.4byte 0x00000000
.2byte SelectorCode32

```

但是有位朋友提醒我说现在的 GAS 已经支持混合字长跳转指令（如图 3.9），看来 NASM 的手册好久没有维护喽 ⊙。

```

77
78      /* Mixed-Size Jump. */

```

```

79     ljmp $SelectorCode32, $0      /* Thanks to earthengine@gmail, I got */
80                           /* this mixed-size jump insn of gas. */
81
82 LABEL_SEG_CODE32:

```

Fig 3.9: 混合字长跳转指令(节自chapter3/1/loader.S)

执行这条混合字长的跳转指令时，CPU 就会用段选择子 SelectorCode32 去寻找 GDT 中对应的段，由于段偏移是 0，所以 CPU 将跳转到图 3.6 中 demo 程序的开头。为了方便阅读，整个 loader.S 的代码附在图 3.10 中：

```

1  /* chapter3/1/loader.S
2
3   Author: Wenbo Yang <solrex@gmail.com> <http://solrex.cn>
4
5   This file is part of the source code of book "Write Your Own OS with Free
6   and Open Source Software". Homepage @ <http://share.solrex.cn/WriteOS/>.
7
8   This file is licensed under the GNU General Public License; either
9   version 3 of the License, or (at your option) any later version. */
10
11 #include "pm.h"
12
13 .code16
14 .text
15     jmp LABEL_BEGIN      /* jump over the .data section. */
16
17 /* NOTE! Wenbo-20080512: Actually here we put the normal .data section into
18   the .code section. For application SW, it is not allowed. However, we are
19   writing an OS. That is OK. Because there is no OS to complain about
20   that behavior. :) */
21
22 /* Global Descriptor Table */
23 LABEL_GDT:           Descriptor 0,          0, 0
24 LABEL_DESC_CODE32:   Descriptor 0,          (SegCode32Len - 1), (DA_C + DA_32)
25 LABEL_DESC_VIDEO:    Descriptor 0xB8000,     0xffff, DA_DRW
26
27 .set GdtLen, (. - LABEL_GDT) /* GDT Length */
28
29 GdtPtr: .2byte (GdtLen - 1) /* GDT Limit */
30     .4byte 0             /* GDT Base */
31
32 /* GDT Selector */
33 .set    SelectorCode32, (LABEL_DESC_CODE32 - LABEL_GDT)
34 .set    SelectorVideo, (LABEL_DESC_VIDEO - LABEL_GDT)
35
36 /* Program starts here. */
37 LABEL_BEGIN:
38     mov    %cs, %ax    /* Move code segment address(CS) to data segment */
39     mov    %ax, %ds    /* register(DS), ES and SS. Because we have */
40     mov    %ax, %es    /* embedded .data section into .code section in */
41     mov    %ax, %ss    /* the start(mentioned in the NOTE above). */

```

```
43     mov    $0x100, %sp
44
45     /* Initialize 32-bits code segment descriptor. */
46     xor    %eax, %eax
47     mov    %cs, %ax
48     shl    $4, %eax
49     addl   $(LABEL_SEG_CODE32), %eax
50     movw   %ax, (LABEL_DESC_CODE32 + 2)
51     shr    $16, %eax
52     movb   %al, (LABEL_DESC_CODE32 + 4)
53     movb   %ah, (LABEL_DESC_CODE32 + 7)
54
55     /* Prepared for loading GDTR */
56     xor    %eax, %eax
57     mov    %ds, %ax
58     shl    $4, %eax
59     add    $(LABEL_GDT), %eax      /* eax <- gdt base*/
60     movl   %eax, (GdtPtr + 2)
61
62     /* Load GDTR(Global Descriptor Table Register) */
63     lgdtw GdtPtr
64
65     /* Clear Interrupt Flags */
66     cli
67
68     /* Open A20 line. */
69     inb   $0x92, %al
70     orb    $0b00000010, %al
71     outb  %al, $0x92
72
73     /* Enable protect mode, PE bit of CRO. */
74     movl   %cr0, %eax
75     orl    $1, %eax
76     movl   %eax, %cr0
77
78     /* Mixed-Size Jump. */
79     ljmpl $SelectorCode32, $0      /* Thanks to earthengine@gmail, I got */
80                           /* this mixed-size jump insn of gas. */
81
82 LABEL_SEG_CODE32:
83 .code32
84     mov    $(SelectorVideo), %ax
85     mov    %ax, %gs           /* Video segment selector(dest) */
86
87     movl   $((80 * 10 + 0) * 2), %edi
88     movb   $0xC, %ah          /* 0000: Black Back 1100: Red Front */
89     movb   '$P', %al
90
91     mov    %ax, %gs:(%edi)
92
93     /* Stop here, infinite loop. */
94     jmp    .
95
96 /* Get the length of 32-bit segment code. */
97 .set    SegCode32Len, . - LABEL_SEG_CODE32
```

Fig 3.10: chapter3/1/loader.S

3.2.6 生成镜像并测试

使用与第 2.3.6 节完全相同的方法，我们可以将代码编译并将 LOADER.BIN 拷贝到镜像文件中。利用最新的镜像文件启动 VirtualBox 我们得到图 3.11。

可以看到，屏幕的左侧中央打出了一个红色的 P，这就是我们那个在保护模式下运行的简单 demo 所做的事情，这说明我们的代码是正确的。从实模式迈入保护模式，这只是一小步，但对于我们的操作系统来说，这是一大步。从此我们不必再被限制到 20 位的地址空间中，有了更大的自由度。

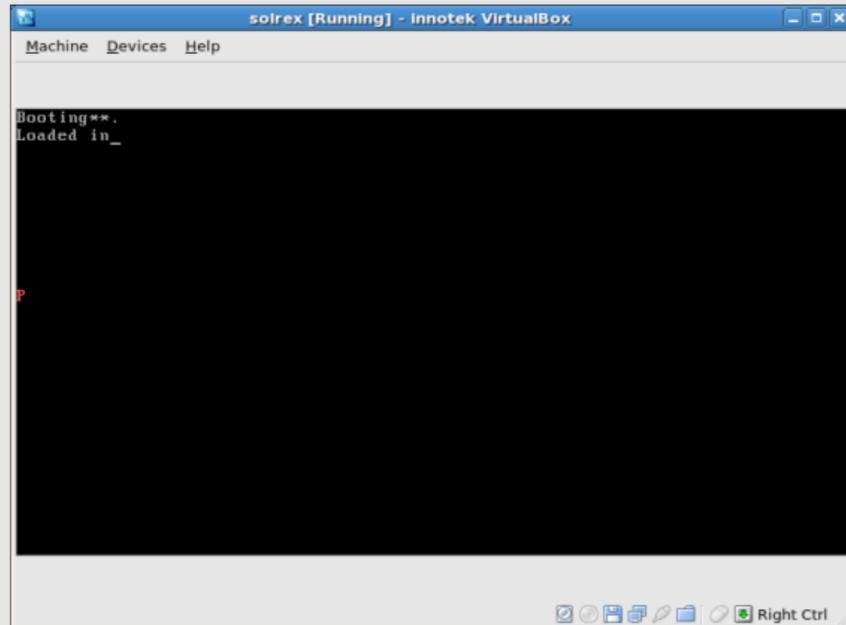


Fig 3.11: 第一次进入保护模式

3.3 段式存储

如果您仔细阅读了图 3.1b，您就会发现图中并未提到 GDT，而是使用的 Descriptor Table(DT)。这是因为对于 x86 架构的 CPU 来说，DT 总共有两个：我们上节介绍过的 GDT 和下面要介绍的 LDT。这两个描述符表构成了 x86 CPU 段式存储的基础。顾名思义，GDT 作为全局的描述符表，只能有一个，而 LDT 作为局部描述符表，就可以有很多个，这也是以后操作系统给每个任务分配自己的存储空间的基础。

3.3.1 LDT 数据结构

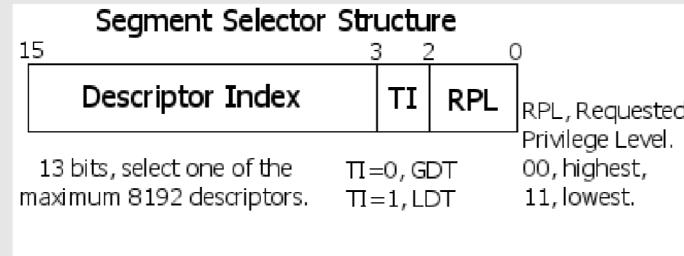


Fig 3.12: 段选择子数据结构

事实上，LDT 和 GDT 的差别非常小，LDT 段描述符的数据结构和图 3.2 所示是一样的。所不同的就是，LDT 用指令 `lldt` 来加载，并且指向 LDT 描述符项的段选择子的 TI 位置必须标识为 1，如图 3.12 所示。这样，在使用 TI flag := 1 的段选择子时，操作系统才会从当前的 LDT 而不是 GDT 中去寻找对应的段描述符。

这里值得注意的一点是：GDT 是由线性空间里的地址定义的，即 `lgdt` 指令的参数是一个线性空间的地址；而 LDT 是由 GDT 中的一个段描述符定义的，即 `lldt` 指令的参数是 GDT 中的一个段选择子。这是因为在加载 GDT 之前寻址模式是实模式的，而加载 GDT 后寻址模式变成保护模式寻址，将 LDT 作为 GDT 中的段使用，也方便操作系统在多个 LDT 之间切换。

3.3.2 段描述符属性

我们在介绍图 3.2 时，并没有完全介绍段描述符的各个 Flag 和可能的属性，这一小节就用来专门介绍段描述符的属性，按照图 3.2 中的 Flag 从左向右的顺序：

- **G:** G(Granularity, 粒度)：如果 G flag 置为 0，段的大小以字节为单位，段长度范围是 1 byte~1 MB；如果 G flag 置为 1，段的大小以 4 KB 为单位，段长度范围是 4 KB ~ 4 GB。
- **D/B:** D/B(Default operation size/Default stack pointer size and/or upper Bound, 默认操作大小)，其意思取决于段描述符是代码段、数据段或者堆栈段。该 flag 置为 0 代表代码段/数据段为 16 位的；置为 1 代表该段是 32 位的。
- **L:** L(Long, 长)，L flag 是 IA-32e(Extended Memory 64 Technology) 模式下使用的标志。该 flag 置为 1 代表该段是正常的 64 位的代码段；置为 0 代表在兼容模式下运行的代码段。在 IA-32 架构下，该位是保留位，并且永远被置为 0。
- **AVL:** 保留给操作系统软件使用的位。
- **P:** P(segment-Present, 段占用？) flag 用于标志段是否在内存中，主要供内存管理软件使用。如果 P flag 被置为 0，说明该段目前不在内存中，该段指向的内存可以暂时被其它任务占用；如果 P flag 被置为 1，说明该段在内存中。如果 P flag 为 0 的段被访问，处理机会产生一个 segment-not-present(#NP) 异常。

- **DPL:** DPL(Descriptor Privilege Level)域标志着段的特权级，取值范围是从 0~3(2-bit)，0 代表着最高的特权级。关于特权级的作用，我们将在下节讨论。
- **S:** S(descriptor type) flag 标志着该段是否系统段：置为 0 代表该段是系统段；置为 1 代表该段是代码段或者数据段。
- **Type:** Type 域是段描述符里最复杂的一个域，而且它的意义对于代码/数据段描述符和系统段/门描述符是不同的，下面我们用两张表来展示当 Type 置为不同值时的意义。

表 3.1 所示即为代码/数据段描述符的所有 Type 可能的值(0-15, 4-bit)以及对应的属性含意，表 3.2 所示为系统段/门描述符的 Type 可能的值以及对应的属性含意。这两张表每个条目的内容是自明的，而且我们在后面的讨论中将不止一次会引用这两张表的内容，所以这里对每个条目暂时不加详细阐述。

表 3.1: 代码/数据段描述符的 Type 属性列表

Type Field					Descriptor Type	Description
Decimal	11 E	10 W	9	8 A		
0	0	0	0	0	Data	Read-Only
1	0	0	0	1	Data	Read-Only, Accessed
2	0	0	1	0	Data	Read/Write
3	0	0	1	1	Data	Read/Write, Accessed
4	0	1	0	0	Data	Read-Only, Expand-down
5	0	1	0	1	Data	Read-Only, Expand-down, Accessed
6	0	1	1	0	Data	Read/Write, Expand-down
7	0	1	1	1	Data	Read/Write, Expand-down, Accessed
		C	R	A		
8	1	0	0	0	Code	Execute-Only
9	1	0	0	1	Code	Execute-Only, Accessed
10	1	0	1	0	Code	Execute/Read
11	1	0	1	1	Code	Execute/Read, Accessed
12	1	1	0	0	Code	Execute-Only, Conforming
13	1	1	0	1	Code	Execute-Only, Conforming, Accessed
14	1	1	1	0	Code	Execute/Read-Only, Conforming
15	1	1	1	1	Code	Execute/Read-Only, Conforming, Accessed

表 3.2: 系统段/门描述符的 Type 属性列表

Type Field					Description
Decimal	11	10	9	8	32-Bit Mode
0	0	0	0	0	Reserved

1	0	0	0	1	16-bit TSS(Available)
2	0	0	1	0	LDT
3	0	0	1	1	16-bit TSS(Busy)
4	0	1	0	0	16-bit Call Gate
5	0	1	0	1	Task Gate
6	0	1	1	0	16-bit Interrupt Gate
7	0	1	1	1	16-bit Trap Gate
8	1	0	0	0	Reserved
9	1	0	0	1	32-bit TSS(Available)
10	1	0	1	0	Reserved
11	1	0	1	1	32-bit TSS(Busy)
12	1	1	0	0	32-bit Call Gate
13	1	1	0	1	Reserved
14	1	1	1	0	32-bit Interrupt Gate
15	1	1	1	1	32-bit Trap Gate

3.3.3 使用 LDT

从目前的需求来看，对 LDT 并没有非介绍不可的理由，但是理解 LDT 的使用，对理解段式存储和处理器多任务存储空间分配有很大的帮助。所以我们在下面的代码中实现几个简单的例子：一，建立 32 位数据和堆栈两个段并将描述符添加到 GDT 中；二，添加一段简单代码，并以其段描述符为基础建立一个 LDT；三，在 GDT 中添加 LDT 的段描述符并初始化所有 DT；四，进入保护模式下运行的 32 位代码段后，加载 LDT 并跳转执行 LDT 中包含的代码段。

首先，建立 32 位全局数据段和堆栈段，并将其描述符添加到 GDT 中：

```

50 /* 32-bit global data segment. */
51 LABEL_DATA:
52 PMMessage: .ascii "Welcome to protect mode! ^~\0"
53 LDTMessage: .ascii "Aha, you jumped into a LDT segment.\0"
54 .set    OffsetPMMessage, (PMMessage - LABEL_DATA)
55 .set    OffsetLDTMessage, (LDTMessage - LABEL_DATA)
56 .set    DataLen,          (. - LABEL_DATA)
57
58 /* 32-bit global stack segment. */
59 LABEL_STACK:
60 .space 512, 0
61 .set    TopOfStack, (. - LABEL_STACK - 1)
62

22 /* Global Descriptor Table */
23 LABEL_GDT:      Descriptor      0,           0, 0
24 LABEL_DESC_CODE32: Descriptor    0, (SegCode32Len - 1), (DA_C + DA_32)
25 LABEL_DESC_DATA: Descriptor     0,           (DataLen - 1), DA_DRW
26 LABEL_DESC_STACK: Descriptor    0,           TopOfStack, (DA_DRWA + DA_32)
27 LABEL_DESC_VIDEO: Descriptor   0xB8000,       0xffff, DA_DRW

```

```

28 LABEL_DESC_LDT:    Descriptor      0,          (LDTLen - 1), DA_LDT
29
30 .set GdtLen, (. - LABEL_GDT) /* GDT Length */
31
32 GdtPtr: .2byte  (GdtLen - 1) /* GDT Limit */
33     .4byte  0             /* GDT Base */
34
35 /* GDT Selector(TI flag clear) */
36 .set    SelectorCode32, (LABEL_DESC_CODE32 - LABEL_GDT)
37 .set    SelectorData,   (LABEL_DESC_DATA - LABEL_GDT)
38 .set    SelectorStack, (LABEL_DESC_STACK - LABEL_GDT)
39 .set    SelectorVideo, (LABEL_DESC_VIDEO - LABEL_GDT)
40 .set    SelectorLDT,   (LABEL_DESC_LDT - LABEL_GDT)
41

```

Fig 3.13: 32 位全局数据段和堆栈段，以及对应的 GDT 结构(节自chapter3/2/loader.S)

在图 3.13 中，我们首先建立了一个全局的数据段，并在数据段里放置了两个字符串，分别用来进入保护模式后和跳转到 LDT 指向的代码段后作为信息输出。然后又建立了一个全局的堆栈段，为堆栈段预留了 512 字节的空间，并将栈顶设置为距栈底 511 字节处。然后与上节介绍的类似，将数据段和堆栈段的段描述符添加到 GDT 中，并设置好对应的段选择子。

要注意到数据段、堆栈段和代码段的段描述符属性不尽相同。数据段的段描述符属性是 DA_DRW，回忆我们前面 pm.h 的内容（图 3.5），DA_DRW 的内容是 0x92，用二进制就是 10010010，其后四位就对应着图 3.1 中的第二 2(0010) 项，说明这个段是可读写的数据段；前四位对应着 P|DPL|S 三个 flag，即 P:1，DPL:00，S:1，与第 3.3.2 节结合理解，意思就是该段在内存中，为最高的特权级，非系统段。所以我们可以看到 pm.h 中的各个属性变量定义，就是将二进制的属性值用可理解的变量名表示出来，在用的时候直接加上变量即可。

同理我们也可以分别来理解 GDT 中堆栈段和代码段描述符的属性定义。因为不同类型的属性使用的是段描述符中不同的位，所以不同类型的属性可以直接相加得到复合的属性值，例如堆栈段的 (DA_DRWA + DA_32)，其意思类似于 C++ 中 `fstream` 打开文件时可以对模式进行算术或 (`ios_base::in | ios_base::out`) 来得到复合参数。

其次，添加一段简单的代码，并以其描述符为基础建立一个 LDT：

```

114 /* 32-bit code segment for LDT */
115 LABEL_CODEA:
116 .code32
117     mov    $(SelectorVideo), %ax
118     mov    %ax, %gs
119
120     movb   $0xC, %ah           /* 0000: Black Back 1100: Red Front */
121     xor    %esi, %esi
122     xor    %edi, %edi
123     movl   $(OffsetLDTMessage), %esi
124     movl   $((80 * 12 + 0) * 2), %edi
125     cld    /* Clear DF flag. */
126
127 /* Display a string from %esi(string offset) to %edi(video segment). */
128 CODEA.1:
129     lodsb    /* Load a byte from source */

```

```

130    test    %al, %al
131    jz     CODEA.2
132    mov    %ax, %gs:(%edi)
133    add    $2, %edi
134    jmp    CODEA.1
135 CODEA.2:
136
137    /* Stop here, infinite loop. */
138    jmp    .
139 .set   CodeALen, (. - LABEL_CODEA)
140
42 /* LDT segment */
43 LABEL_LDT:
44 LABEL_LDT_DESC_CODEA: Descriptor 0, (CodeALen - 1), (DA_C + DA_32)
45
46 .set   LDTLen, (. - LABEL_LDT) /* LDT Length */
47 /* LDT Selector (TI flag set)*/
48 .set   SelectorLDTCodeA, (LABEL_LDT_DESC_CODEA - LABEL_LDT + SA_TIL)
49

```

Fig 3.14: 32 位代码段，以及对应的 LDT 结构(节自chapter3/2/loader.S)

LABEL_CODEA 就是我们为 LDT 建立的简单代码段，其作用就是操作显存在屏幕的第 12 行开始用红色的字打印出偏移 OffsetLDTMessage 指向的全局数据段中的字符串。下面就是以 LABEL_CODEA 为基础建立的 LDT，从 LDT 的结构来说，与 GDT 没有区别，但是我们不用像 GdtPtr 再建立一个 LdtPtr，因为 LDT 实际上是在 GDT 中定义的一个段，不用实模式的线性地址表示。

LDT 的选择子是与 GDT 选择子有明显区别的，图 3.12 清楚地解释了这一点，所以指向 LDT 的选择子都应该将 TI 位置 1，在图 3.14 的最后一行也实现了这一操作。

第三，在 GDT 中添加 LDT 的段描述符（在图 3.13 中我们已经能看到在 GDT 中添加好了 LDT 的段描述符），初始化所有段描述符。由于初始化段描述符属于重复性工作，我们在 pm.h 中添加一个汇编宏 InitDesc 来帮我们做这件事情。

```

84 /* Initialize descriptor.
85 Usage: InitDesc SegLabel, SegDesc */
86 .macro InitDesc SegLabel, SegDesc
87     xor    %eax, %eax
88     mov    %cs, %ax
89     shl    $4, %eax
90     addl   $(\SegLabel), %eax
91     movw   %ax, (\SegDesc + 2)
92     shr    $16, %eax
93     movb   %al, (\SegDesc + 4)
94     movb   %ah, (\SegDesc + 7)
95 .endm
96

```

Fig 3.15: 自动初始化段描述符的宏代码(节自chapter3/2/pm.h)

```

63 /* Program starts here. */
64 LABEL_BEGIN:
65     mov    %cs, %ax    /* Move code segment address(CS) to data segment */
66     mov    %ax, %ds    /* register(DS), ES and SS. Because we have      */
67     mov    %ax, %es    /* embedded .data section into .code section in   */
68     mov    %ax, %ss    /* the start(mentioned in the NOTE above).          */
69
70     mov    $0x100, %sp
71
72     /* Initialize 32-bits code segment descriptor. */
73     InitDesc LABEL_SEG_CODE32, LABEL_DESC_CODE32
74
75     /* Initialize data segment descriptor. */
76     InitDesc LABEL_DATA, LABEL_DESC_DATA
77
78     /* Initialize stack segment descriptor. */
79     InitDesc LABEL_STACK, LABEL_DESC_STACK
80
81     /* Initialize LDT descriptor in GDT. */
82     InitDesc LABEL_LDT, LABEL_DESC_LDT
83
84     /* Initialize code A descriptor in LDT. */
85     InitDesc LABEL_CODEA, LABEL_LDT_DESC_CODEA
86
87     /* Prepared for loading GDTR */
88     xor    %eax, %eax
89     mov    %ds, %ax
90     shl    $4, %eax
91     add    $(LABEL_GDT), %eax      /* eax <- gdt base*/
92     movl   %eax, (GdtPtr + 2)
93
94     /* Load GDTR(Global Descriptor Table Register) */
95     lgdtw GdtPtr
96
97     /* Clear Interrupt Flags */
98     cli
99
100    /* Open A20 line. */
101    inb   $0x92, %al
102    orb   $0b00000010, %al
103    outb  %al, $0x92
104
105    /* Enable protect mode, PE bit of CR0. */
106    movl   %cr0, %eax
107    orl    $1, %eax
108    movl   %eax, %cr0
109
110    /* Mixed-Size Jump. */
111    ljmpl $SelectorCode32, $0      /* Thanks to earthengine@gmail, I got */
                                   /* this mixed-size jump insn of gas. */
112

```

Fig 3.16: 在实模式代码段中初始化所有段描述符(节自chapter3/2/loader.S)

初始化各个段描述符的方式与上一节介绍的初始化 GDT 描述符的方式没有什么本质不同，因为属性都已经预设好，运行时只需要将段地址填入描述符中的地址域即可，代码都是重复的。我们引入宏 `InitDesc` 的帮助，能大大缩短代码长度，增强代码的可读性。

第四，进入保护模式下运行的 32 位代码段后，加载 LDT 并跳转执行 LDT 中包含的代码段：

```

141 /* 32-bit code segment for GDT */
142 LABEL_SEG_CODE32:
143     mov    $(SelectorData), %ax
144     mov    %ax, %ds          /* Data segment selector */
145     mov    $(SelectorStack), %ax
146     mov    %ax, %ss          /* Stack segment selector */
147     mov    $(SelectorVideo), %ax
148     mov    %ax, %gs          /* Video segment selector(dest) */
149
150     mov    $(TopOfStack), %esp
151
152     movb   $0xC, %ah         /* 0000: Black Back 1100: Red Front */
153     xor    %esi, %esi
154     xor    %edi, %edi
155     movl   $(OffsetPMMessages), %esi
156     movl   $((80 * 10 + 0) * 2), %edi
157     cld                  /* Clear DF flag. */
158
159 /* Display a string from %esi(string offset) to %edi(video segment). */
160 CODE32.1:
161     lodsb                /* Load a byte from source */
162     test    %al, %al
163     jz      CODE32.2
164     mov    %ax, %gs:(%edi)
165     add    $2, %edi
166     jmp    CODE32.1
167 CODE32.2:
168
169     mov    $(SelectorLDT), %ax
170     lldt   %ax
171
172    ljmp   $(SelectorLDTCodeA), $0
173
174 /* Get the length of 32-bit segment code. */
175 .set    SegCode32Len, . - LABEL_SEG_CODE32

```

Fig 3.17: 在保护模式代码段中加载 LDT 并跳转执行 LDT 代码段(节自chapter3/2/loader.S)

在 `LABEL_SEG_CODE32` 中前几行，我们可以看到非常熟悉的汇编指令，和一般汇编程序开头初始化数据/代码/堆栈段寄存器的指令非常像，只不过这里赋给几个寄存器的参数都是段选择子，而不是一般的地址。该代码段剩下的内容和前面图 3.14 中 `LABEL_CODEA` 一样，都是打印一个字符串，只不过这里选择在第 10 行（屏幕左侧中央）打印。

为了方便阅读，整个 `loader.S` 的代码附在图 3.18 中。

```

1 /* chapter3/2/loader.S
2

```

```

3     Author: Wenbo Yang <solrex@gmail.com> <http://solrex.cn>
4
5     This file is part of the source code of book "Write Your Own OS with Free
6     and Open Source Software". Homepage @ <http://share.solrex.cn/WriteOS/>.
7
8     This file is licensed under the GNU General Public License; either
9     version 3 of the License, or (at your option) any later version. */
10
11 #include "pm.h"
12
13 .code16
14 .text
15     jmp LABEL_BEGIN      /* jump over the .data section. */
16
17 /* NOTE! Wenbo-20080512: Actually here we put the normal .data section into
18   the .code section. For application SW, it is not allowed. However, we are
19   writing an OS. That is OK. Because there is no OS to complain about
20   that behavior. :) */
21
22 /* Global Descriptor Table */
23 LABEL_GDT:           Descriptor      0,          0, 0
24 LABEL_DESC_CODE32:   Descriptor      0, (SegCode32Len - 1), (DA_C + DA_32)
25 LABEL_DESC_DATA:    Descriptor      0,          (DataLen - 1), DA_DRW
26 LABEL_DESC_STACK:   Descriptor      0,          TopOfStack, (DA_DRWA + DA_32)
27 LABEL_DESC_VIDEO:   Descriptor      0xB8000,      0xffff, DA_DRW
28 LABEL_DESC_LDT:     Descriptor      0,          (LDTLen - 1), DA_LDT
29
30 .set GdtLen, (. - LABEL_GDT) /* GDT Length */
31
32 GdtPtr: .2byte (GdtLen - 1) /* GDT Limit */
33     .4byte 0             /* GDT Base */
34
35 /* GDT Selector(TI flag clear) */
36 .set   SelectorCode32, (LABEL_DESC_CODE32 - LABEL_GDT)
37 .set   SelectorData,  (LABEL_DESC_DATA - LABEL_GDT)
38 .set   SelectorStack, (LABEL_DESC_STACK - LABEL_GDT)
39 .set   SelectorVideo, (LABEL_DESC_VIDEO - LABEL_GDT)
40 .set   SelectorLDT,   (LABEL_DESC_LDT - LABEL_GDT)
41
42 /* LDT segment */
43 LABEL_LDT:
44 LABEL_LDT_DESC_CODEA: Descriptor 0, (CodeALen - 1), (DA_C + DA_32)
45
46 .set   LDTLen, (. - LABEL_LDT) /* LDT Length */
47 /* LDT Selector (TI flag set)*/
48 .set   SelectorLDTCodeA, (LABEL_LDT_DESC_CODEA - LABEL_LDT + SA_TIL)
49
50 /* 32-bit global data segment. */
51 LABEL_DATA:
52 PMMessage: .ascii "Welcome to protect mode! ^-^\0"
53 LDTMessage: .ascii "Aha, you jumped into a LDT segment.\0"
54 .set   OffsetPMMessage, (PMMessage - LABEL_DATA)
55 .set   OffsetLDTMessage, (LDTMessage - LABEL_DATA)
56 .set   DataLen,        (. - LABEL_DATA)
57

```

```
58 /* 32-bit global stack segment. */
59 LABEL_STACK:
60 .space 512, 0
61 .set TopOfStack, (. - LABEL_STACK - 1)
62
63 /* Program starts here. */
64 LABEL_BEGIN:
65     mov    %cs, %ax    /* Move code segment address(CS) to data segment */
66     mov    %ax, %ds    /* register(DS), ES and SS. Because we have      */
67     mov    %ax, %es    /* embedded .data section into .code section in   */
68     mov    %ax, %ss    /* the start(mentioned in the NOTE above).          */
69
70     mov    $0x100, %sp
71
72     /* Initialize 32-bits code segment descriptor. */
73     InitDesc LABEL_SEG_CODE32, LABEL_DESC_CODE32
74
75     /* Initialize data segment descriptor. */
76     InitDesc LABEL_DATA, LABEL_DESC_DATA
77
78     /* Initialize stack segment descriptor. */
79     InitDesc LABEL_STACK, LABEL_DESC_STACK
80
81     /* Initialize LDT descriptor in GDT. */
82     InitDesc LABEL_LDT, LABEL_DESC_LDT
83
84     /* Initialize code A descriptor in LDT. */
85     InitDesc LABEL_CODEA, LABEL_LDT_DESC_CODEA
86
87     /* Prepared for loading GDTR */
88     xor    %eax, %eax
89     mov    %ds, %ax
90     shl    $4, %eax
91     add    $(LABEL_GDT), %eax    /* eax <- gdt base*/
92     movl   %eax, (GdtPtr + 2)
93
94     /* Load GDTR(Global Descriptor Table Register) */
95     lgdtw GdtPtr
96
97     /* Clear Interrupt Flags */
98     cli
99
100    /* Open A20 line. */
101    inb   $0x92, %al
102    orb   $0b00000010, %al
103    outb  %al, $0x92
104
105    /* Enable protect mode, PE bit of CRO. */
106    movl   %cr0, %eax
107    orl    $1, %eax
108    movl   %eax, %cr0
109
110    /* Mixed-Size Jump. */
111    ljmpl $SelectorCode32, $0      /* Thanks to earthengine@gmail, I got */
112                                /* this mixed-size jump insn of gas. */
```

```

113
114 /* 32-bit code segment for LDT */
115 LABEL_CODEA:
116 .code32
117     mov      $(SelectorVideo), %ax
118     mov      %ax, %gs
119
120     movb    $0xC, %ah           /* 0000: Black Back 1100: Red Front */
121     xor     %esi, %esi
122     xor     %edi, %edi
123     movl    $(OffsetLDTMessage), %esi
124     movl    $((80 * 12 + 0) * 2), %edi
125     cld      /* Clear DF flag. */
126
127 /* Display a string from %esi(string offset) to %edi(video segment). */
128 CODEA.1:
129     lodsb      /* Load a byte from source */
130     test     %al, %al
131     jz      CODEA.2
132     mov      %ax, %gs:(%edi)
133     add     $2, %edi
134     jmp      CODEA.1
135 CODEA.2:
136
137     /* Stop here, infinite loop. */
138     jmp      .
139 .set    CodeALen, (. - LABEL_CODEA)
140
141 /* 32-bit code segment for GDT */
142 LABEL_SEG_CODE32:
143     mov      $(SelectorData), %ax
144     mov      %ax, %ds           /* Data segment selector */
145     mov      $(SelectorStack), %ax
146     mov      %ax, %ss           /* Stack segment selector */
147     mov      $(SelectorVideo), %ax
148     mov      %ax, %gs           /* Video segment selector(dest) */
149
150     mov      $(TopOfStack), %esp
151
152     movb    $0xC, %ah           /* 0000: Black Back 1100: Red Front */
153     xor     %esi, %esi
154     xor     %edi, %edi
155     movl    $(OffsetPMMMessage), %esi
156     movl    $((80 * 10 + 0) * 2), %edi
157     cld      /* Clear DF flag. */
158
159 /* Display a string from %esi(string offset) to %edi(video segment). */
160 CODE32.1:
161     lodsb      /* Load a byte from source */
162     test     %al, %al
163     jz      CODE32.2
164     mov      %ax, %gs:(%edi)
165     add     $2, %edi
166     jmp      CODE32.1
167 CODE32.2:

```

```

168     mov      $(SelectorLDT), %ax
169     lldt    %ax
170
171    ljmp    $(SelectorLDTCodeA), $0
172
173
174 /* Get the length of 32-bit segment code. */
175 .set    SegCode32Len, . - LABEL_SEG_CODE32

```

Fig 3.18: chapter3/2/loader.S

3.3.4 生成镜像并测试

使用与第 2.3.6 节完全相同的方法，我们可以将代码编译并将 LOADER.BIN 拷贝到镜像文件中。利用最新的镜像文件启动 VirtualBox 我们得到图 3.19。

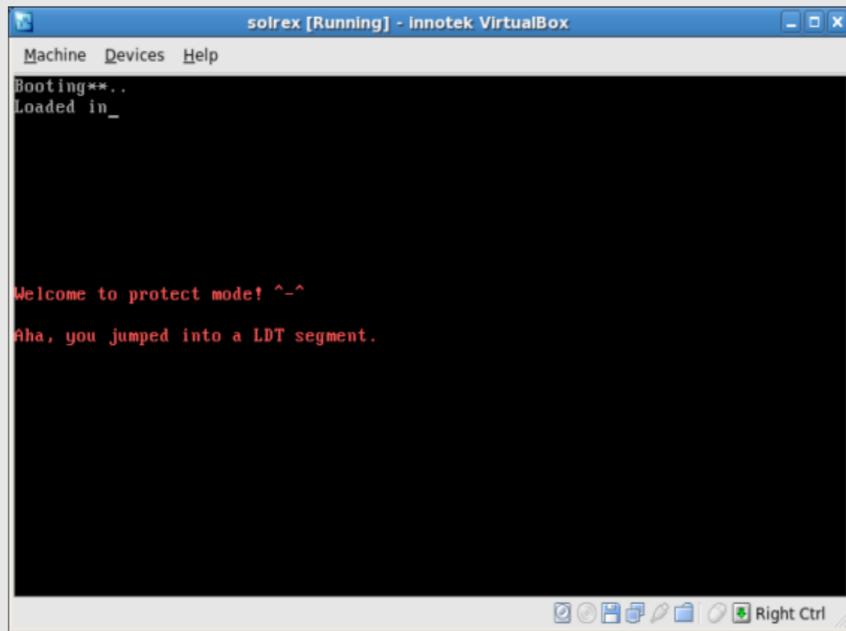


Fig 3.19: 第一次进入保护模式

可以看到，该程序首先在屏幕左侧中央（第 10 行）打印出来 "Welcome to protect mode! ^-^"，这是由 GDT 中的 32 位代码段 LABEL_SEG_CODE32 打印出来的，标志着我们成功进入保护模式；然后在屏幕的第 12 行打印出来 "Aha, you jumped into a LDT segment."，这个是由 LDT 中的 32 位代码段 LABEL_CODEA 打印出来的，标志着 LDT 的使用正确。因为这两个字符串都是被存储在 32 位全局数据段中，这两个字符串的成功打印也说明在 GDT 中添加的数据段使用正确。

3.3.5 段式存储总结

段式存储和页式存储都是最流行的计算机内存保护方式。段式存储的含义简单来说就是先将内存分为各个段，然后再分配给程序供不同用途使用，并保证对各个段的访问互不干扰。x86 主要使用段寄存器（得到的段基址）+ 偏移量来访问段中数据，也简化了寻址过程。

在 x86 的初期实模式下就使用着非常简单的段式存储方式，如图 3.1a 所示，这种模式下分段主要是为了方便寻址和隔离，没有提供其它的保护机制。x86 保护模式下采用了更高级的段式存储方式：用全局和局部描述符表存储段描述符信息，使用段选择子表示各个段描述符，如图 3.1b 所示。

由于保护模式使用段描述符来保存段信息而不是像实模式一样直接使用段地址，在段描述符中就可以添加一些属性来限制对段的访问权限，如我们在第 3.3.2 节中讨论的那样。这样，通过在访问段时检查权限和属性，就能做到对程序段的更完善保护和更好的内存管理。

x86 使用全局描述符表（GDT）和局部描述符表（LDT）来实现不同需求下对程序段的控制，操作系统使用唯一的一个 GDT 来维护一些和系统密切相关的段描述符信息，为不同的任务使用不同的 LDT 来实现对多任务内存管理的支持，简化了任务切换引起的内存切换的难度。

3.4 特权级



如果您需要更详细的知识，也许您更愿意去读 Intel 的手册，本节内容主要集中在：[Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide](#), 第 4 章。

特权级是为了保护处理器资源而引入的概念。将同一个处理器上执行的不同任务赋予不同的特权级，可以控制该任务可以访问的资源，比如内存地址范围、输入输出端口、和一些特殊指令的使用。在 x86 体系结构中，共有 4 个特权级别，0 代表最高特权级，3 代表最低特权级。由于在 x86 体系结构中，n 级可以访问的资源均可以被 0 到 n 级访问，这个模式被称作 ring 模式，相应地我们也将 x86 的对应特权级称作 ring n。

现代的 PC 操作系统的内核一般工作在 ring 0 下，拥有最高的特权级，应用程序一般工作在 ring 3 下，拥有最低的特权级。虽然 x86 体系结构提供了 4 个特权级，但操作系统并不需要全部使用到这 4 个级别，可以根据需要来选择使用几个特权级。比如 Linux/Unix 和 Windows NT，都是只使用了 0 级和 3 级，分别用于内核模式和用户模式；而 DOS 则只使用了 0 级。

为了实施对代码段和数据段的特权级检验，x86 处理机引入了以下三种特权级类型（请注意这里提到的特权级高低均为实际高低，而非数值意义上的高低）：

- **CPL(Current Privilege Level)**: 当前特权级，存储在 CS 和 SS 的 0, 1 位。它代表当前执行程序或任务的特权级，通常情况下与当前执行指令所在代码段的 DPL 相同。当程序跳转到不同特权级的代码段时，CPL 会随之修改。当访问一致代码段（Conforming Code Segment）时，对 CPL 的处理有些不同。一致代码段可以被不高于（数值上大于等于）该段 DPL 的特权级代码访问，但是，CPL 在访问一致代码段时不会跟随 DPL 的变化而更改。
- **DPL(Descriptor Privilege Level)**: 描述符特权级，定义于段描述符或门描述符中的 DPL 域（见图 3.2），它限制了可以访问此段资源的特权级别。根据被访问的段或者门的不同，DPL 的意义也不同：

- **数据段**: 数据段的 DPL 限制了可以访问该数据段的最低特权级。假如数据段的 DPL 为 1，那么只有 CPL 为 0,1 的程序才能访问该数据段。
- **非一致代码段（不使用调用门）**: 非一致代码段就是一般的代码段，它的 DPL 表示可以访问该段的特权级，程序或者任务的特权级必须与该段的 DPL 完全相同才可以访问该段。
- **调用门**: 调用门的 DPL 限制了可以访问该门的最低特权级，与数据段 DPL 的意思一样。
- **一致代码段和使用调用门访问的非一致代码段**: 这种代码段的 DPL 表示可以访问该段的最高特权级。假如一致代码段的 DPL 是 2，那么 CPL 为 0,1 的程序就无法访问该段。
- **TSS(Task State Segment)**: 任务状态段的 DPL 表示可以访问该段的最低特权级，与数据段 DPL 的意思一样。
- **RPL(Requested Privilege Level)**: 请求特权级，定义于段选择子的 RPL 域中（见图 3.12）。它限制了这个选择子可访问资源的最高特权级。比如一个段选择子的 RPL 为 2，那么使用这个段选择子只能访问 DPL 为 2 或者 3 的段，即使使用这个段选择子的程序当前特权级（CPL）为 0。就是说， $\max(CPL, RPL) \leq DPL$ 才被允许访问该段，即当 CPL 小于 RPL 时，RPL 起决定性作用，反之亦然。使用 RPL 可以避免特权级高的程序代替应用程序访问该应用程序无权访问的段。比如在系统调用时，应用程序调用系统过程，虽然系统过程的特权级高（ $CPL = 0$ ），但是被调用的系统过程仍然无法访问特权级高于应用程序的段（ $DPL < RPL = 3$ ），就避免了可能出现的安全问题。

在将段描述符对应的段选择子加载到段寄存器时，处理机通过将 CPL，段选择子的 RPL 和该段的 DPL 相比较，来判断程序是否有权访问另外一个段。如果 $CPL > \max(RPL, DPL)$ ，或者 $\max(CPL, RPL) > DPL$ ，那么该访问就是不合法的，处理机就会产生一个常规保护异常（#GP, General Protection Exception）。

3.4.1 不合法的访问请求示例

我们来看一个不合法的访问请求的例子，在上一节的 loader.S 中把 LABEL_DESC_DATA 对应的描述符的 DPL 设置为 1，然后将该数据段对应的段选择子的 RPL 设置为 3，即修改以下两行：

```
LABEL_DESC_DATA:    Descriptor          0,      (DataLen - 1), (DA_DRW + DA_DPL1)
.set    SelectorData,  (LABEL_DESC_DATA - LABEL_GDT + SA_RPL3)
```

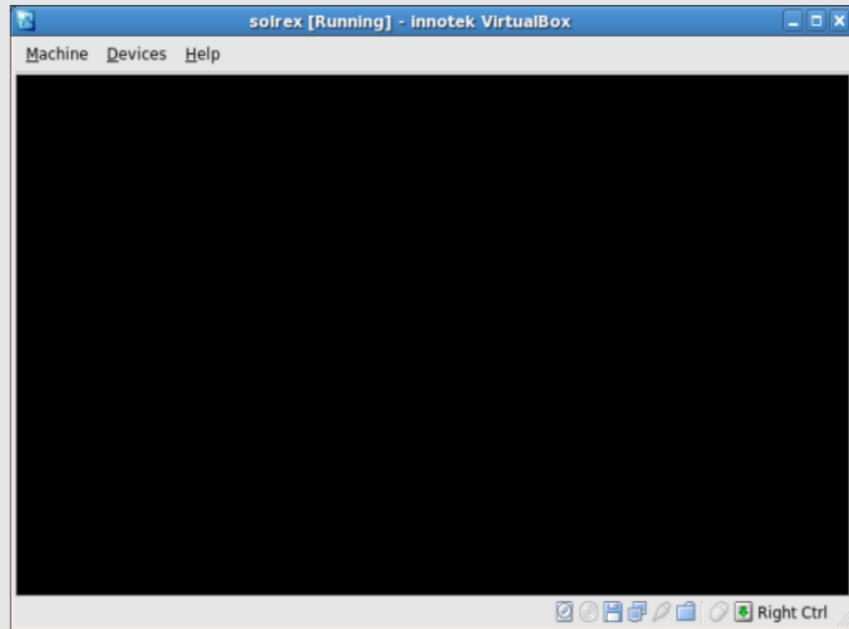


Fig 3.20: 虚拟机出现异常，黑屏

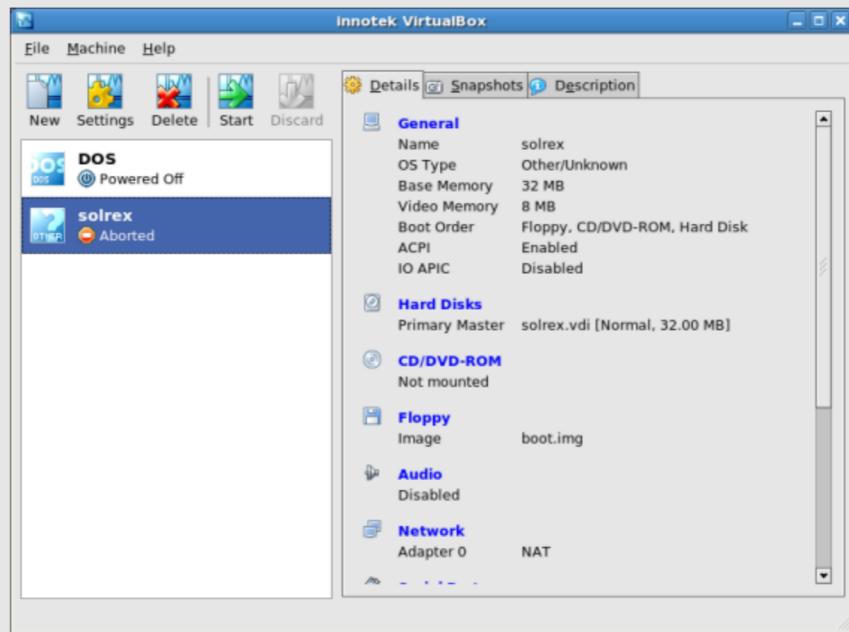


Fig 3.21: 虚拟退出后 VBox 主窗口显示 Abort

再 `make`, `sudo make copy`, 用 VirtualBox 加载生成的镜像运行一下, 就会发现虚拟机黑屏一会儿就会退出 (如图 3.20), 然后 VirtualBox 主窗口中显示该虚拟机 Aborted (如图 3.21)。这是因为我们违反特权级的规则, 使用 RPL=3 的选择子去访问 DPL=1 的段, 这个不合法的访问请求引起处理机产生常规保护异常 (#GP)。而我们又没有准备对应的异常处理模块, 当处理机找不到异常处理程序时

就只好退出了。

3.4.2 控制权转移的特权级检查

在将控制权从一个代码段转移到另一个代码段之前，目标代码段的段选择子必须被加载到 CS 中。处理器在这个过程中会查看目标代码段的段描述符以及对其界限、类型（见图 3.2）和特权级进行检查。如果没有错误发生，CS 寄存器会被加载，程序控制权被转移到新的代码段，从 EIP 指示的位置开始运行。

JMP, CALL, RET, SYSENTER, SYSEXIT, INT n 和 IRET 这些指令，以及中断和异常机制都会引起程序控制权的转移。

JMP 和 CALL 指令可以实现以下 4 种形式的转移：

- 目标操作数包含目标代码段的段选择子。
- 目标操作数指向一个包含目标代码段段选择子的调用门描述符。
- 目标操作数指向一个包含目标代码段段选择子的任务状态段。
- 目标操作数指向一个任务门，这个任务门指向一个包含目标代码段段选择子的任务状态段。

下面两个小节将描述前两种转移的实现方法，后两种控制权转移方法我们将在用到时再进行解释。

用 JMP 或 CALL 直接转移

用 JMP, CALL 和 RET 指令在段内进行近跳转并没有特权级的变化，所以对这类转移是不进行特权级检查的；用 JMP, CALL 和 RET 在段间进行远跳转涉及到其它代码段，所以要进行特权级检查。

对不通过调用门的直接转移来说，又分为两种情形：

- 访问非一致代码段：当目标是非一致代码段时（目标段段描述符的 C flag 为 0，见图 3.1），特权级检查要求调用者的 CPL 与目标代码段的 DPL 相等，而且调用者使用的目标代码段段选择子的 RPL 必须小于等于目标代码段的 DPL。我们之前的代码都属于这种情形，其中 $CPL = DPL = RPL = 0$ 。
- 访问一致代码段：当目标是一致代码段时（目标段段描述符的 C flag 为 1，见图 3.1），特权级检查要求 $CPL \geq DPL$ ，RPL 不被检查，而且转移时并不修改 CPL。

总的来说，通过 JMP 和 CALL 实行的都是一般的转移，最多从低特权级转移到高特权级的一致代码段，CPL 总是不变的。

3.4.3 使用调用门转移

调用门是 x86 体系结构下用来控制程序在不同特权级间转移的一种机制。它的目的是使低特权级的代码能够调用高特权级的代码，这一机制在使用了内存保护和特权级机制的现代操作系统中非常有用，因为它允许应用程序在操作系统控制下安全地调用内核例程或者系统接口。

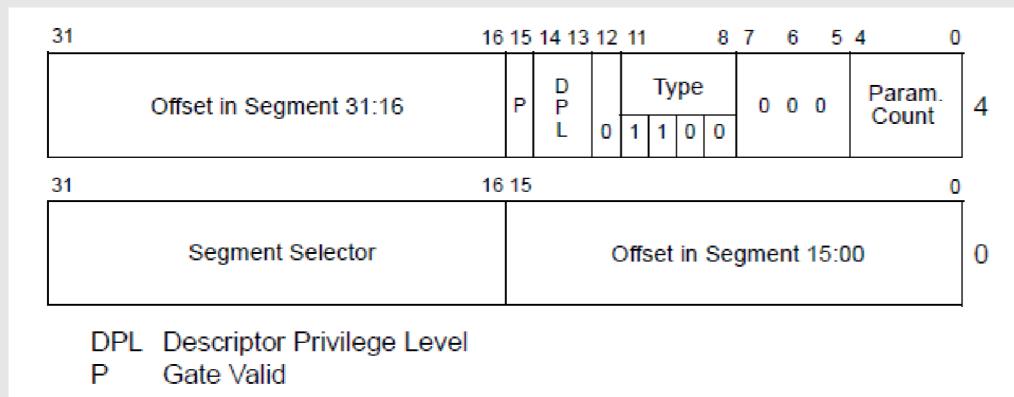


Fig 3.22: 调用门描述符

门其实也是一种描述符，和段描述符类似。调用门描述符的数据结构如图 3.22 所示。其实看起来这个调用门描述符的数据结构要比段描述符简单一些，至少从它的属性来说，没有段描述符多。我们仍然只关注最重要的部分：首先是段选择子（Segment Selector），指定了通过这个调用门访问的代码段；其次是段偏移量（Offset in Segment），指定了要访问代码段中的某个入口偏移；描述符特权级（DPL），代表此门描述符的特权级；P，代表此调用门是否可用；参数计数（Param. Count）记录了如果发生栈切换的话，有多少个选项参数会在栈间拷贝。

简单来说，调用门描述了由一个段选择子和一个偏移所指定的目标代码段中的一个地址，程序通过调用门将转移到这个地址。下面我们通过一个简单的例子来介绍一下调用门的基本使用方法。

简单的调用门转移举例

为了使用调用门，我们首先要给出一个目标段，然后用该目标段的信息初始化调用门的门描述符，最后用调用门的门选择子实现门调用。

添加一个目标段我们已经做过很多次，非常简单。首先在上一节 loader.S 最后添加一个打印一个字符的代码段 LABEL_SEG_CODECG，接着将该段的段描述符 LABEL_DESC_CODECG 添加到 GDT 中，然后为该段准备一个段选择子 SelectorCodeCG，最后加入初始化该段描述符的代码：

```

197 /* 32-bit code segment for call gate destination segment */
198 LABEL_SEG_CODECG:
199     mov    $(SelectorVideo), %ax
200     mov    %ax, %gs
201
202     movl   $((80 * 11 + 0) * 2), %edi /* line 11, column 0 */
203     movb   $0xC, %ah                /* 0000: Black Back 1100: Red Front */
204     movb   '$C', %al               /* Print a 'C' */
205
206     mov    %ax, %gs:(%edi)
207     lret
208
209 /* Get the length of 32-bit call gate destination segment code. */
210 .set    SegCodeCGLen, . - LABEL_SEG_CODECG

```

```

28 LABEL_DESC_LDT: Descriptor      0,          (LDTLen - 1), DA_LDT
29 LABEL_DESC_CODECG: Descriptor    0, (SegCodeCGLen - 1), (DA_C + DA_32)

43 .set   SelectorLDT,   (LABEL_DESC_LDT - LABEL_GDT)
44 .set   SelectorCodeCG, (LABEL_DESC_CODECG - LABEL_GDT)

92     /* Initialize call gate dest code segment descriptor. */
93     InitDesc LABEL_SEG_CODECG, LABEL_DESC_CODECG
94

```

Fig 3.23: 添加调用门的目标段(节自chapter3/3/loader.S)

总的来看，LABEL_SEG_CODECG 指向的这个段和我们以前为了打印程序运行结果所使用的段没有本质不同，为了简单起见，这里我们仅仅让它打印一个字符 ’C’ 就返回。

用目标代码段 LABEL_SEG_CODECG 的信息初始化调用门的门描述符 LABEL_CG_TEST，以及门选择子 SelectorCGTest。与汇编宏 Descriptor 类似，我们这里使用汇编宏 Gate 来初始化门描述符，宏 Gate 的定义可以在头文件 pm.h 中找到：

```

71 /* Gate Descriptor data structure.
72 Usage: Gate Selector, Offset, PCount, Attr
73   Selector: 2byte
74   Offset:   4byte
75   PCount:   byte
76   Attr:     byte */
77 .macro Gate Selector, Offset, PCount, Attr
78   .2byte    (\Offset & 0xFFFF)
79   .2byte    \Selector
80   .2byte    (\PCount & 0x1F) | ((\Attr << 8) & 0xFF00)
81   .2byte    ((\Offset >> 16) & 0xFFFF)
82 .endm

```

Fig 3.24: 汇编宏 Gate 定义(节自chapter3/3/pm.h)

```

29 LABEL_DESC_CODECG: Descriptor    0, (SegCodeCGLen - 1), (DA_C + DA_32)
30 /* Gates Descriptor */
31 LABEL_CG_TEST:     Gate   SelectorCodeCG, 0, 0, (DA_386CGate + DA_DPL0)
32
33 .set GdtLen, (. - LABEL_GDT) /* GDT Length */

43 .set   SelectorLDT,   (LABEL_DESC_LDT - LABEL_GDT)
44 .set   SelectorCodeCG, (LABEL_DESC_CODECG - LABEL_GDT)
45 .set   SelectorCGTest, (LABEL_CG_TEST - LABEL_GDT)

```

Fig 3.25: 设置调用门描述符及选择子(节自chapter3/3/loader.S)

我们可以看到，宏 Gate 的四个参数分别为：段选择子、偏移量、参数计数和属性，它们在存储空间中的分布与图 3.22 中介绍相同。由于这个例子仅仅介绍调用门的简单使用，并不涉及特权级切换，所以也不发生栈切换，这里我们将参数计数设置为 0；门描述符的属性为 (DA_386CGate + DPL) ，

表明它是一个调用门（属性定义见图 3.3），DPL 为 0，与我们一直使用的特权级相同；目标代码段选择子是 SelectorCodeCG，偏移是 0，所以如果该调用门被调用，将转移到目标代码段的开头，即 LABEL_SEG_CODECG 处开始执行。

使用远调用 lcall 指令调用该调用门的门选择子 SelectorCGTest：

```

185 CODE32.2:
186
187     lcall    $(SelectorCGTest), $0 /* Call CODECG through call gate */
188
189     mov     $(SelectorLDT), %ax
190     lldt    %ax

```

Fig 3.26: 调用门选择子(节自chapter3/3/loader.S)

由于对调用门的调用往往涉及到段间转移，所以我们通常使用 gas 的 lcall 远跳转指令和 lret 远返回指令进行调用和返回。

这样我们就完成了使用调用门进行简单控制权转移的代码，make, sudo make copy 之后，用 VBox 虚拟机载入生成的镜像，运行结果如图 3.27 所示。由于我们仅仅是在加载 LDT 之前添加了一个门调用，而且门调用的目标段在屏幕的第 11 行第 0 列打印了一个 ‘C’ 后就返回到了调用处，所以加载 LDT 的代码继续运行，就是图中所示结果。

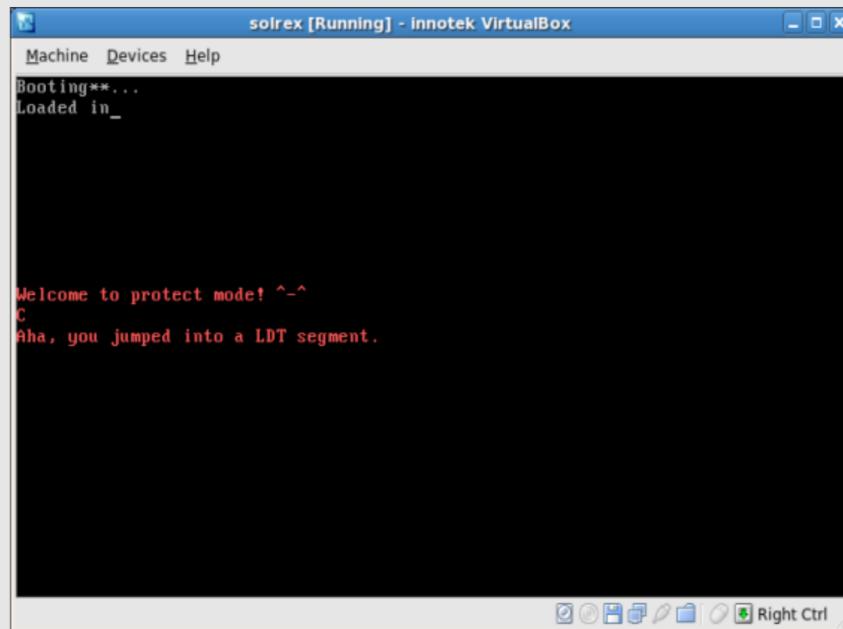


Fig 3.27: 使用调用门进行简单的控制权转移

涉及特权级变化的调用门转移

在上面例子中我们只是使用调用门取代了传统的直接跳转方式，并没有涉及到特权级的变化。显然

调用门不是用来做这种自找麻烦的事情的，其设计的主要目的是实现从低特权级代码跳转到高特权级的非一致代码的功能。

在使用调用门进行转移时，处理机会使用四个特权级值来检查控制权的转移是否合法：

1. CPL：当前特权级；
2. RPL：调用门选择子的请求特权级；
3. DPL：调用门描述符的描述符特权级；
4. DPL：目标代码段的段描述符特权级。

使用 CALL 或者 JMP 指令访问调用门进行控制权转移时，特权级检查的规则有所不同，如表 3.3 所示：

表 3.3: 调用门特权级检查规则

指令	特权级检查规则
CALL	CPL ≤ 调用门 DPL; RPL ≤ 调用门 DPL 目标段DPL ≤ CPL
JMP	CPL ≤ 调用门 DPL; RPL ≤ 调用门 DPL 当目标段是一致代码段时：目标段 DPL ≤ CPL 当目标段是非一致代码段时：目标段 DPL = CPL

这张表内容虽然不多，但也不容易很快理解。这里我们应该着重看目标段 DPL 和 CPL 的比较，这才是调用门特权级检验的特点所在。总的来说，使用调用门需要目标段的 DPL 小于或等于 CPL，意思就是要转移的目标段特权级高于当前特权级。这与我们在本节开头看到的一般转移的特权级检查有非常明显的不同。除此之外剩下的检查就是对调用门访问的检查了，这种特权级检查和访问一个数据段时进行的特权级检查的规则是一样的，我们已经熟知了。

我们已经了解了涉及特权级变化的调用门转移时处理机进行的特权级检查规则，但为了写一段从低特权级转移到高特权级的测试代码，我们仍然需要处理一个问题：如何从高特权级转移到低特权级？因为我们之前的代码一直运行在 ring 0 特权级上，要实现从低到高的转移，首先要从高特权级转到低特权级。

其实思想很简单，既然 CALL 指令能从低特权级转移到高特权级，自然而然地 RET 指令能从高特权级返回低特权级。我们只需要 hack 一下 RET 指令的使用方法即可（即用 RET 指令实现跳转到低特权级代码的功能）。

一般 CALL 和 RET 指令都是配合使用的，先用 CALL 跳转到目标地址，目标代码执行完后再用 RET 返回到 CALL 的下一条指令。为了从 ring 0 跳转到 ring 3，我们并不使用 CALL 而直接执行 RET。为了使 RET 执行返回时不出错，我们需要为 RET 准备好返回时环境，就像通常执行 CALL 指令后处理机进行的工作一样。

那么执行 CALL 指令时处理机都进行了哪些工作呢？这是一个非常复杂的问题，我们将留待下个小节再详细介绍。但是在我们的例子里（即最简单的情况下），CALL 所做的就是将 SS, ESP, CS, EIP

这四个寄存器的值顺序压到栈里，这样在 RET 指令执行的时候，处理器从堆栈 pop 出 EIP, CS, ESP, SS 的值来恢复 CALL 指令执行后的处理器现场。所以为了使 RET 跳转到我们想要执行的代码段，我们只需要手动将 ring 3 目标代码段的对应的 SS, ESP, CS, EIP 值压到栈里即可。

下面开始准备这个 demo，仍旧是在上一节代码的基础上进行添加。首先，我们准备一个 ring 3 目标代码段和新栈：

```

224 /* 32-bit code segment for running in ring 3. */
225 LABEL_SEG_CODER3:
226     mov    $SelectorVideo, %ax
227     mov    %ax, %gs
228
229     movl   $((80 * 11 + 1) * 2), %edi /* line 11, column 1 */
230     movb   $0xC, %ah                /* 0000: Black Back 1100: Red Front */
231     movb   '$3', %al               /* Print a '3' */
232
233     mov    %ax, %gs:(%edi)
234     jmp   .
235
236 /* Get the length of 32-bit ring 3 segment code. */
237 .set   SegCodeR3Len, . - LABEL_SEG_CODER3

```

Fig 3.28: 要运行在 ring 3 下的代码段(节自chapter3/4/loader.S)

```

73 /* 32-bit ring 3 stack segment. */
74 LABEL_STACKR3:
75 .space 512, 0
76 .set   TopOfStackR3, (. - LABEL_STACKR3)

```

Fig 3.29: 为 ring 3 代码段准备的新栈(节自chapter3/4/loader.S)

这个代码段的功能就是在第 11 行第 1 列打印一个 3 字。

其次，添加新段的描述符和选择子，并添加初始化代码：

```

29 LABEL_DESC_CODECG: Descriptor      0, (SegCodeCGLen - 1), (DA_C + DA_32)
30 LABEL_DESC_CODER3: Descriptor      0, (SegCodeR3Len - 1), (DA_C + DA_32 + DA_DPL3)
31 LABEL_DESC_STACKR3: Descriptor     0,           TopOfStackR3, (DA_DRWA + DA_32 + DA_DPL3)

47 .set   SelectorCGTest, (LABEL(CG_TEST) - LABEL_GDT)
48 .set   SelectorCodeR3, (LABEL_DESC_CODER3 - LABEL_GDT + SA_RPL3)
49 .set   SelectorStackR3, (LABEL_DESC_STACKR3 - LABEL_GDT + SA_RPL3)

```

Fig 3.30: 为 ring 3 代码段和堆栈段添加的描述符和选择子(节自chapter3/4/loader.S)

```

105  /* Initialize ring 3 stack segment descriptor. */
106  InitDesc LABEL_STACKR3, LABEL_DESC_STACKR3
107
108  /* Initialize ring 3 dest code segment descriptor. */
109  InitDesc LABEL_SEG_CODER3, LABEL_DESC_CODER3

```

Fig 3.31: 初始化 ring 3 代码段和堆栈段描述符的代码(节自chapter3/4/loader.S)

我们注意到，其实 ring 3 下的代码段和 ring 0 下的代码段的代码部分是没有任何区别的，区别在于它们的代码段描述符和选择子中所标明的特权级。我们将 ring 3 下的目标代码段和堆栈段的段描述符属性中加上 DA_DPL3，表明它们的段描述符特权级均为 3；在它们的段选择子属性中加上 SA_RPL3，表明它们的段选择子请求特权级均为 3。以上这两点限制了这个段只能在 ring 3 下运行。

准备好了两个段，我们将这两个段的信息作为 SS, ESP, CS, EIP 的内容依次压栈，然后再执行 lret 长返回指令，就能跳转到 ring 3 下运行目标代码段了：

```

191 CODE32.2:
192
193     pushl  $(SelectorStackR3)      /* Fake call procedure. */
194     pushl  $(TopOfStackR3)
195     pushl  $(SelectorCodeR3)
196     pushl  $0
197     lret                /* return with no call */
198
199     lcall   $(SelectorCGTest), $0 /* Call CODECG through call gate */

```

Fig 3.32: hack RET 指令进行实际的跳转

这样，我们就完成了从高特权级代码（ring 0）跳转到低特权级代码（ring 3）的过程。像往常那样使用 make, sudo make copy 编译，用虚拟机加载镜像运行结果如图 3.33 所示，程序在屏幕的第 11 行第 1 列打印出了一个红色的 '3' 字，然后进入死循环。

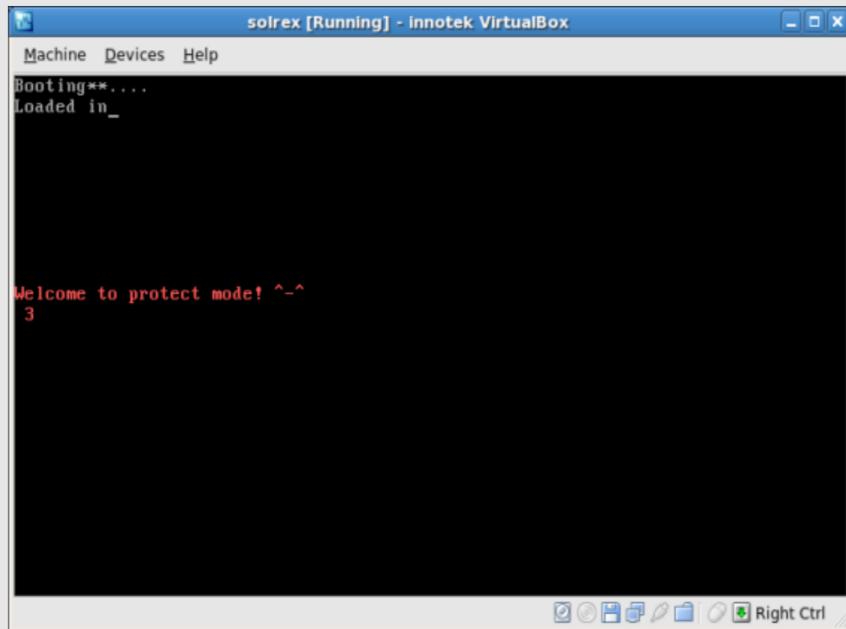


Fig 3.33: hack RET 实现从高特权级到低特权级的跳转

上面这个例子仅仅演示了如何从高特权级跳转到低特权级，而没有介绍如何转移回高特权级。直观来讲，我们只需将图 3.28 中的最后一条 JMP 指令替换成一条 CALL 调用门的指令即可。但是我们会看到，带有特权级转换的调用门转移并不是那么容易实现的。

下面我们来尝试一下直接访问调用门，首先将门描述符的 DPL 改为 3 以满足在 ring 3 代码段访问调用门的条件。

```
LABEL_CG_TEST:      Gate    SelectorCodeCG, 0, 0, (DA_386CGate + DA_DPL3)
```

然后将图 3.28 中最后一条 JMP 指令替换成对调用门的 CALL 指令：

```
lcall $(SelectorCGTest), $0 /* Call CODECG through call gate */
```

然后编译运行一下，看看能否得到我们想要的结果？答案是否定的，我们看不到屏幕上打印出 'C' 字，得到与图 3.21 相同的结果。为什么呢？主要是因为在 CALL 调用门的时候发生了程序栈的切换，由于这个栈切换发生在 CALL 指令执行的过程中，CALL 指令要访问特殊的结构来得到新的栈信息，不像 RET 指令仅仅读取我们设置好的栈，如果访问出错就会产生一个异常，处理机无法处理这个异常就只好退出。我们下面一小节介绍在 CALL 调用门和 RET 的时候究竟发生了什么事？需要什么特殊结构的辅助才能实现带有特权级切换的调用门转移？

3.4.4 栈切换和 TSS

当使用调用门转移到不同特权级下的非一致代码段时，处理机总会自动的切换到目标特权级对应的栈。栈切换是为了避免高特权级的栈空间被滥用导致栈空间不足而崩溃，以及低特权级的程序非法修改或者干扰高特权级程序的栈内容。

为了使栈切换能够成功，我们必须为任务中用到的每个特权级都定义一个独立的栈。在上一小节最后一个例子中，我们实际已经定义了两个堆栈段：LABEL_STACK 和 LABEL_STACKR3，分别被 ring 0 和 ring 3 使用，并且已经实现了从高特权级转移到低特权级程序时的栈切换。从上面例子中我们看出，使用 RET 指令时，处理机从堆栈中得到目标特权级的堆栈段选择子（SelectorStackR3）和栈顶指针（TopOfStackR3），然后分别存储到 SS 和 ESP 寄存器中，就实现了栈切换。RET 指令能如此直接地实现栈切换的关键是 CALL 指令已经将调用者所在特权级的栈信息压到被调用者所在特权级的栈里。

但是在使用 CALL 指令时，处理机不能从栈上获得被调用者所在特权级的栈信息，就需要一个辅助的数据结构来帮助它获得这些信息，这个数据结构就是 TSS(Task-State Segment, 任务状态段)。当前任务的 TSS 中存储着指向 0, 1, 2 特权级栈的指针，这个指针指向的内容包括一个堆栈段选择子和栈顶指针，如图 3.34 中 SS* 和 ESP* 所示。TSS 的结构中并不包含 ring 3 的栈信息，这是因为在使用调用门时，不可能发生 ring 0, 1, 2 的代码通过 CALL 跳转到 ring 3 的情况。这种情况只会发生在 RET 返回的时候，而这时候我们只需要将 ring 3 的栈信息压到栈里就能实现跳转（就像我们上一小节做的那样）。

当 TSS 被加载后，这些指向的和栈相关的内容会被严格限制为只读，处理机在运行过程中不会修改它们。在一个跨特权级的 CALL 调用结束返回时，TSS 中的栈信息不会被修改，所有被调用者堆栈的改变都会恢复原状。在下一次 CALL 调用时，处理机读取的栈信息与前一次没有任何不同。为每个特权级准备的堆栈段空间必须足够容纳可能的多次调用产生的多重栈帧。

由于在跨特权级的调用过程中发生了栈切换，在被调用者所在特权级无法访问调用者的栈，所以调用者栈中的一些参数、返回地址等信息需要拷贝到被调用者栈中。有多少个参数需要被拷贝就是图 3.22 中 Param. Count 域定义的。

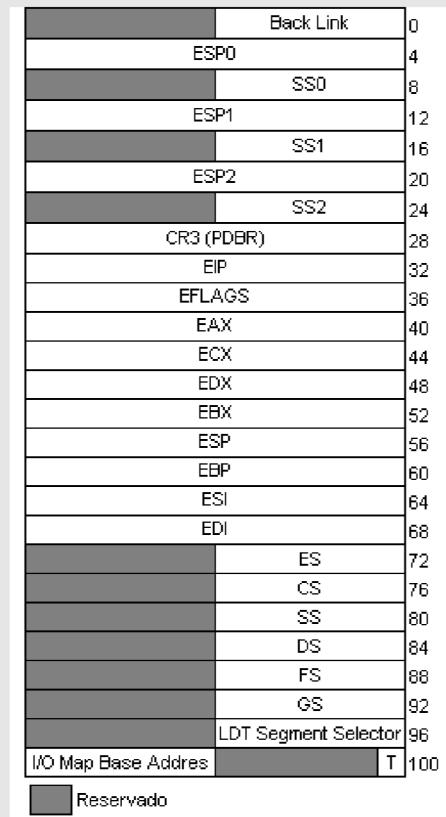


Fig 3.34: 32 位 TSS 数据结构

当跨特权级的调用发生时，处理机要进行下列操作来切换堆栈和切换到目标特权级运行：

1. 依据目标段的 DPL (新的 CPL) 从 TSS 中选择新堆栈的指针 (堆栈段选择子和栈顶指针)；
2. 从当前任务的 TSS 中读取堆栈段选择子 (SS*) 和栈顶指针 (ESP*)。在读取堆栈段选择子、栈顶指针或者堆栈段描述符时发生的任何错误，都会使处理机产生一个 #TS (非法 TSS) 异常。
3. 对堆栈段描述符进行特权级和类型检验，如果通不过检验处理机同样产生 #TS 异常；
4. 暂时保存当前 SS 和 ESP 寄存器中存储的堆栈段选择子和栈顶指针；
5. 加载新的堆栈段选择子和栈顶指针到 SS 和 ESP 中；
6. 把第 4 步保存的 SS 和 ESP 压入新栈 (被调用者栈)；
7. 从旧栈 (调用者栈) 中复制调用门描述符中 Param. Count 个参数到新栈中。如果 PCount 为 0，什么都不复制；
8. 将返回地址指针 (当前 CS 和 EIP 寄存器中的内容) 压到新栈中；
9. 加载调用门中指定的段选择子和指令指针到 CS 和 EIP 寄存器中，开始执行被调用过程。

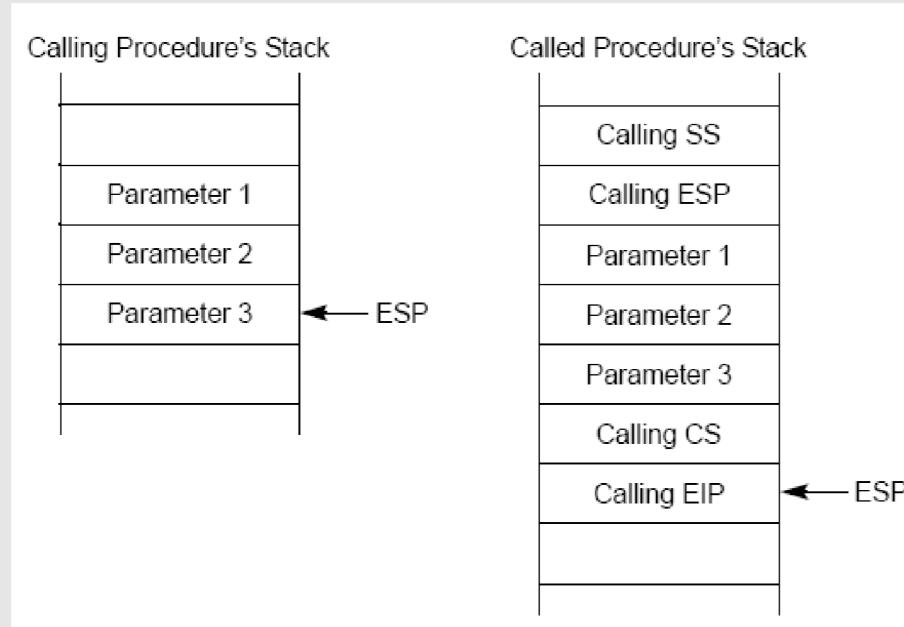


Fig 3.35: 跨特权级调用时的栈切换

其中的栈切换过程如图 3.35 所示。需要注意的是，调用门描述符中的 PCount 域仅有 5 位，就是说在栈切换过程中最多只能复制 31 个参数。如果需要传递更多参数，可以传递一个指向某个数据结构的指针，或者通过栈中保存的调用者栈信息来访问旧栈中的参数。

既然跨特权级的调用那么麻烦，跨特权级的返回也不会很简单。在跨特权级的返回指令执行时，处理器的工作包含以下步骤：

1. 检查保存在栈上的 CS 寄存器中的 RPL 域看返回时是否需要切换特权级；
2. 加载保存在栈上的 CS 和 EIP 寄存器信息（同时对段描述符和段选择子的特权级和类型进行检验）；
3. 如果 RET 指令有参数计数操作数，增加 ESP 的值跳过栈上的参数部分，此时 ESP 将指向保存的调用者的 SS 和 ESP。RET 的参数计数操作数应与调用门中描述符中的 PCount 域相同；
4. 加载 SS 和 ESP，切换到调用者堆栈，被调用者的 SS 和 ESP 将被丢弃（此时会进行堆栈段选择子、栈顶指针和堆栈段描述符的特权级和类型检验）；
5. 如果 RET 指令有参数计数操作数，增加 ESP 的值跳过栈上（调用者栈）的参数部分；
6. 检查 DS, ES, FS 和 GS 寄存器的内容，如果某个寄存器指向的段的 DPL 小于当前特权级 CPL（此规则不适用于一致代码段），该寄存器将加载一个空描述符。

在返回时栈切换的操作可视为调用时栈操作的逆过程。

至此，我们已经完整地描述了跨特权级的调用和返回过程，下面就来实现一段跨特权级调用的示例程序。

在第 3.4.3 的最后，我们尝试直接访问调用门失败，主要原因是没有准备 TSS 段，因此我们只需要在其基础上准备好 TSS 段并将其段选择子加载到 TR 寄存器中即可。

首先，准备好 TSS 段，及其段描述符和选择子，并加入初始化 TSS 段描述符的代码：

```

80 LABEL_TSS:
81     .4byte 0          /* Back Link */
82     .4byte TopOfStack /* ESP0 */
83     .4byte SelectorStack /* SSO */
84     .4byte 0          /* ESP1 */
85     .4byte 0          /* SS1 */
86     .4byte 0          /* ESP2 */
87     .4byte 0          /* SS2 */
88     .4byte 0          /* CR3(PDBR) */
89     .4byte 0          /* EIP */
90     .4byte 0          /* EFLAGS */
91     .4byte 0          /* EAX */
92     .4byte 0          /* ECX */
93     .4byte 0          /* EDX */
94     .4byte 0          /* EBX */
95     .4byte 0          /* ESP */
96     .4byte 0          /* EBP */
97     .4byte 0          /* ESI */
98     .4byte 0          /* EDI */
99     .4byte 0          /* ES */
100    .4byte 0          /* CS */
101    .4byte 0          /* SS */
102    .4byte 0          /* DS */
103    .4byte 0          /* FS */
104    .4byte 0          /* GS */
105    .4byte 0          /* LDT Segment Selector */
106    .2byte 0          /* Trap Flag: 1-bit */
107    .2byte (. - LABEL_TSS + 2) /* I/O Map Base Address */
108    .byte 0xff        /* End */
109 .set    TSSLen, (. - LABEL_TSS)

31 LABEL_DESC_STACKR3: Descriptor      0,      TopOfStackR3, (DA_DRWA + DA_32 + DA_DPL3)
32 LABEL_DESC_TSS:       Descriptor      0,      (TSSLen - 1), DA_386TSS

50 .set    SelectorStackR3, (LABEL_DESC_STACKR3 - LABEL_GDT + SA_RPL3)
51 .set    SelectorTSS,      (LABEL_DESC_TSS - LABEL_GDT)

144 /* Initialize TSS segment descriptor. */
145 InitDesc LABEL_TSS, LABEL_DESC_TSS

```

Fig 3.36: TSS 段内容及其描述符和选择子和初始化代码(节自chapter3/5/loader.S)

进入 ring 3 之前将 TSS 段选择子加载到 TR 寄存器中，这样在 ring 3 中访问调用门就可以实现跳转了。

```

229     mov      $(SelectorTSS), %ax    /* Load TSS to TR register */
230     ltr      %ax
231
232     pushl   $(SelectorStackR3)    /* Fake call procedure. */

```

Fig 3.37: 加载 TSS 段选择子到 TR 寄存器(节自chapter3/5/loader.S)

将得到的代码编译成镜像，运行结果如图 3.38 所示。我们可以看到这次在图 3.33 的基础上多打印了一个 ‘C’ 字然后陷入了死循环，说明了我们成功达到了调用门跨特权级转移的目的。这个示例程序实现了从 ring 3 代码段转移到 ring 0 代码段，然后再返回 ring 3 进入死循环。

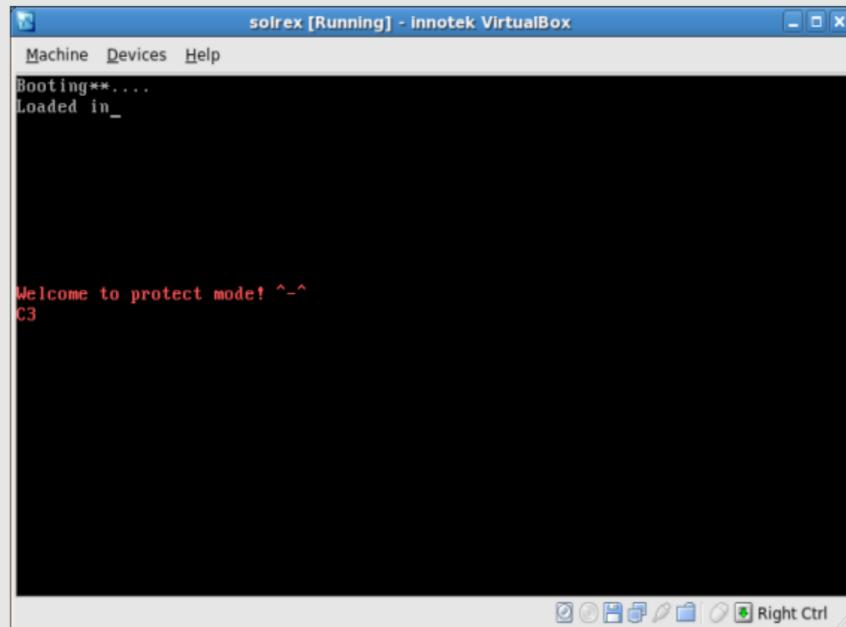


Fig 3.38: 跨特权级的调用门转移

为方便阅读，本节最终的 loader.S 全部代码如图 3.39：

```

1  /* chapter3/5/loader.S
2
3  Author: Wenbo Yang <solrex@gmail.com> <http://solrex.cn>
4
5  This file is part of the source code of book "Write Your Own OS with Free
6  and Open Source Software". Homepage @ <http://share.solrex.cn/WriteOS/>.
7
8  This file is licensed under the GNU General Public License; either
9  version 3 of the License, or (at your option) any later version. */
10
11 #include "pm.h"
12

```

```

13 .code16
14 .text
15     jmp LABEL_BEGIN      /* jump over the .data section. */
16
17 /* NOTE! Wenbo-20080512: Actually here we put the normal .data section into
18   the .code section. For application SW, it is not allowed. However, we are
19   writing an OS. That is OK. Because there is no OS to complain about
20   that behavior. :) */
21
22 /* Global Descriptor Table */
23 LABEL_GDT:           Descriptor    0,          0, 0
24 LABEL_DESC_CODE32:   Descriptor    0, (SegCode32Len - 1), (DA_C + DA_32)
25 LABEL_DESC_DATA:    Descriptor    0,          (DataLen - 1), DA_DRW
26 LABEL_DESC_STACK:   Descriptor    0,          TopOfStack, (DA_DRWA + DA_32)
27 LABEL_DESC_VIDEO:   Descriptor    0xB8000,    0xffff, (DA_DRW + DA_DPL3)
28 LABEL_DESC_LDT:     Descriptor    0,          (LDTLen - 1), DA_LDT
29 LABEL_DESC_CODECG:  Descriptor    0, (SegCodeCGLen - 1), (DA_C + DA_32)
30 LABEL_DESC_CODER3: Descriptor    0, (SegCodeR3Len - 1), (DA_C + DA_32 + DA_DPL3)
31 LABEL_DESC_STACKR3: Descriptor   0,          TopOfStackR3, (DA_DRWA + DA_32 + DA_DPL3)
32 LABEL_DESC_TSS:    Descriptor    0,          (TSSLen - 1), DA_386TSS
33 /* Gate Descriptors */
34 LABEL(CG)_TEST:    Gate        SelectorCodeCG, 0, 0, (DA_386CGate + DA_DPL3)
35
36 .set GdtLen, (. - LABEL_GDT) /* GDT Length */
37
38 GdtPtr: .2byte (GdtLen - 1) /* GDT Limit */
39     .4byte 0             /* GDT Base */
40
41 /* GDT Selector(TI flag clear) */
42 .set SelectorCode32, (LABEL_DESC_CODE32 - LABEL_GDT)
43 .set SelectorData,   (LABEL_DESC_DATA - LABEL_GDT)
44 .set SelectorStack,  (LABEL_DESC_STACK - LABEL_GDT)
45 .set SelectorVideo, (LABEL_DESC_VIDEO - LABEL_GDT)
46 .set SelectorLDT,   (LABEL_DESC_LDT - LABEL_GDT)
47 .set SelectorCodeCG, (LABEL_DESC_CODECG - LABEL_GDT)
48 .set SelectorCGTest, (LABEL(CG)_TEST - LABEL_GDT)
49 .set SelectorCodeR3, (LABEL_DESC_CODER3 - LABEL_GDT + SA_RPL3)
50 .set SelectorStackR3, (LABEL_DESC_STACKR3 - LABEL_GDT + SA_RPL3)
51 .set SelectorTSS,   (LABEL_DESC_TSS - LABEL_GDT)
52
53 /* LDT segment */
54 LABEL_LDT:
55 LABEL_LDT_DESC_CODEA: Descriptor 0, (CodeALen - 1), (DA_C + DA_32)
56
57 .set LDTLen, (. - LABEL_LDT) /* LDT Length */
58 /* LDT Selector (TI flag set)*/
59 .set SelectorLDTCodeA, (LABEL_LDT_DESC_CODEA - LABEL_LDT + SA_TIL)
60
61 /* 32-bit global data segment. */
62 LABEL_DATA:
63 PMMessage: .ascii "Welcome to protect mode! ^~\0"
64 LDTMessage: .ascii "Aha, you jumped into a LDT segment.\0"
65 .set OffsetPMMessage, (PMMessage - LABEL_DATA)
66 .set OffsetLDTMessage, (LDTMessage - LABEL_DATA)
67 .set DataLen,      (. - LABEL_DATA)

```

```

68
69 /* 32-bit global stack segment. */
70 .align 4
71 LABEL_STACK:
72 .space 512, 0
73 .set TopOfStack, (. - LABEL_STACK)
74
75 /* 32-bit ring 3 stack segment. */
76 LABEL_STACKR3:
77 .space 512, 0
78 .set TopOfStackR3, (. - LABEL_STACKR3)
79
80 LABEL_TSS:
81     .4byte 0          /* Back Link */
82     .4byte TopOfStack /* ESP0 */
83     .4byte SelectorStack /* SSO */
84     .4byte 0          /* ESP1 */
85     .4byte 0          /* SS1 */
86     .4byte 0          /* ESP2 */
87     .4byte 0          /* SS2 */
88     .4byte 0          /* CR3(PDBR) */
89     .4byte 0          /* EIP */
90     .4byte 0          /* EFLAGS */
91     .4byte 0          /* EAX */
92     .4byte 0          /* ECX */
93     .4byte 0          /* EDX */
94     .4byte 0          /* EBX */
95     .4byte 0          /* ESP */
96     .4byte 0          /* EBP */
97     .4byte 0          /* ESI */
98     .4byte 0          /* EDI */
99     .4byte 0          /* ES */
100    .4byte 0          /* CS */
101    .4byte 0          /* SS */
102    .4byte 0          /* DS */
103    .4byte 0          /* FS */
104    .4byte 0          /* GS */
105    .4byte 0          /* LDT Segment Selector */
106    .2byte 0          /* Trap Flag: 1-bit */
107    .2byte (. - LABEL_TSS + 2) /* I/O Map Base Address */
108    .byte 0xff        /* End */
109 .set TSSLen, (. - LABEL_TSS)
110
111 /* Program starts here. */
112 LABEL_BEGIN:
113     mov    %cs, %ax  /* Move code segment address(CS) to data segment */
114     mov    %ax, %ds  /* register(DS), ES and SS. Because we have */
115     mov    %ax, %es  /* embedded .data section into .code section in */
116     mov    %ax, %ss  /* the start(mentioned in the NOTE above). */
117
118     mov    $0x100, %sp
119
120     /* Initialize 32-bits code segment descriptor. */
121     InitDesc LABEL_SEG_CODE32, LABEL_DESC_CODE32
122

```

```
123    /* Initialize data segment descriptor. */
124    InitDesc LABEL_DATA, LABEL_DESC_DATA
125
126    /* Initialize stack segment descriptor. */
127    InitDesc LABEL_STACK, LABEL_DESC_STACK
128
129    /* Initialize LDT descriptor in GDT. */
130    InitDesc LABEL_LDT, LABEL_DESC_LDT
131
132    /* Initialize code A descriptor in LDT. */
133    InitDesc LABEL_CODEA, LABEL_LDT_DESC_CODEA
134
135    /* Initialize call gate dest code segment descriptor. */
136    InitDesc LABEL_SEG_CODECG, LABEL_DESC_CODECG
137
138    /* Initialize ring 3 stack segment descriptor. */
139    InitDesc LABEL_STACKR3, LABEL_DESC_STACKR3
140
141    /* Initialize ring 3 dest code segment descriptor. */
142    InitDesc LABEL_SEG_CODER3, LABEL_DESC_CODER3
143
144    /* Initialize TSS segment descriptor. */
145    InitDesc LABEL_TSS, LABEL_DESC_TSS
146
147    /* Prepared for loading GDTR */
148    xor    %eax, %eax
149    mov    %ds, %ax
150    shl    $4, %eax
151    add    $(LABEL_GDT), %eax      /* eax <- gdt base*/
152    movl   %eax, (GdtPtr + 2)
153
154    /* Load GDTR(Global Descriptor Table Register) */
155    lgdtw GdtPtr
156
157    /* Clear Interrupt Flags */
158    cli
159
160    /* Open A20 line. */
161    inb    $0x92, %al
162    orb    $0b00000010, %al
163    outb   %al, $0x92
164
165    /* Enable protect mode, PE bit of CRO. */
166    movl   %cr0, %eax
167    orl    $1, %eax
168    movl   %eax, %cr0
169
170    /* Mixed-Size Jump. */
171    ljmpl $SelectorCode32, $0      /* Thanks to earthengine@gmail, I got */
172                                /* this mixed-size jump insn of gas. */
173
174    /* 32-bit code segment for LDT */
175 LABEL_CODEA:
176 .code32
177    mov    $(SelectorVideo), %ax
```

```

178     mov    %ax, %gs
179
180     movb   $0xC, %ah           /* 0000: Black Back 1100: Red Front */
181     xor    %esi, %esi
182     xor    %edi, %edi
183     movl   $(OffsetLDTMessage), %esi
184     movl   $((80 * 12 + 0) * 2), %edi
185     cld    /* Clear DF flag. */
186
187 /* Display a string from %esi(string offset) to %edi(video segment). */
188 CODEA.1:
189     lodsb      /* Load a byte from source */
190     test    %al, %al
191     jz     CODEA.2
192     mov    %ax, %gs:(%edi)
193     add    $2, %edi
194     jmp    CODEA.1
195 CODEA.2:
196
197     /* Stop here, infinite loop. */
198     jmp    .
199 .set   CodeALen, (. - LABEL_CODEA)
200
201 /* 32-bit code segment for GDT */
202 LABEL_SEG_CODE32:
203     mov    $(SelectorData), %ax
204     mov    %ax, %ds           /* Data segment selector */
205     mov    $(SelectorStack), %ax
206     mov    %ax, %ss           /* Stack segment selector */
207     mov    $(SelectorVideo), %ax
208     mov    %ax, %gs           /* Video segment selector(dest) */
209
210     mov    $(TopOfStack), %esp
211
212     movb  $0xC, %ah           /* 0000: Black Back 1100: Red Front */
213     xor    %esi, %esi
214     xor    %edi, %edi
215     movl   $(OffsetPMMMessage), %esi
216     movl   $((80 * 10 + 0) * 2), %edi
217     cld    /* Clear DF flag. */
218
219 /* Display a string from %esi(string offset) to %edi(video segment). */
220 CODE32.1:
221     lodsb      /* Load a byte from source */
222     test    %al, %al
223     jz     CODE32.2
224     mov    %ax, %gs:(%edi)
225     add    $2, %edi
226     jmp    CODE32.1
227 CODE32.2:
228
229     mov    $(SelectorTSS), %ax    /* Load TSS to TR register */
230     ltr    %ax
231
232     pushl  $(SelectorStackR3)    /* Fake call procedure. */

```

```

233     pushl  $(TopOfStackR3)
234     pushl  $(SelectorCodeR3)
235     pushl  $0
236     lret          /* return with no call */
237
238 CODE32.3:
239     mov    $(SelectorLDT), %ax
240     lldt  %ax
241
242    ljmp  $(SelectorLDTCodeA), $0
243
244 /* Get the length of 32-bit segment code. */
245 .set   SegCode32Len, . - LABEL_SEG_CODE32
246
247 /* 32-bit code segment for call gate destination segment */
248 LABEL_SEG_CODECG:
249     mov    $(SelectorVideo), %ax
250     mov    %ax, %gs
251
252     movl  $((80 * 11 + 0) * 2), %edi /* line 11, column 0 */
253     movb  $0xC, %ah                /* 0000: Black Back 1100: Red Front */
254     movb  $'C', %al                /* Print a 'C' */
255
256     mov    %ax, %gs:(%edi)
257     lret
258
259 /* Get the length of 32-bit call gate destination segment code. */
260 .set   SegCodeCGLen, . - LABEL_SEG_CODECG
261
262 /* 32-bit code segment for running in ring 3. */
263 LABEL_SEG_CODER3:
264     mov    $(SelectorVideo), %ax
265     mov    %ax, %gs
266
267     movl  $((80 * 11 + 1) * 2), %edi /* line 11, column 1 */
268     movb  $0xC, %ah                /* 0000: Black Back 1100: Red Front */
269     movb  $'3', %al                /* Print a '3' */
270
271     mov    %ax, %gs:(%edi)
272     lcall $(SelectorCGTest), $0 /* Call CODECG through call gate */
273     jmp   .
274
275 /* Get the length of 32-bit ring 3 segment code. */
276 .set   SegCodeR3Len, . - LABEL_SEG_CODER3
277

```

Fig 3.39: chapter3/5/loader.S

3.5 页式存储

保护模式一个非常重要的特性就是提供了虚拟内存机制。虚拟内存机制使每个应用程序都以为自己拥有完整连续的内存空间，而事实上它可能是被分块存放在物理内存甚至硬盘中。这极大地方便了大型

应用程序的编程，并且能更高效地利用物理内存。在具体实现上，几乎所有的体系结构都使用分页机制作为基本方式。比如某应用程序需要使用 1GB 内存，没有虚拟内存机制的辅助，在低于 1GB 内存的电脑上该程序就无法运行，但是虚拟内存机制可以将该应用程序分成若干“页”，仅仅将程序当前使用的页调入物理内存中，剩下的页存放在硬盘上，需要时再调入，这就可以大大地减少物理内存的使用量。

我们这里所说的“页”是指一块连续的虚拟内存空间，一般情况下最小为 4K 字节，视体系结构可使用的虚拟内存大小的不同，这个数字可以更大，不过一般取 2 的整数次方。比如在 Intel 奔腾系列 CPU 中，页的大小可以取 4KB, 2MB 或者 4MB。

3.5.1 分页机制

当 IA-32 处理机启用分页机制后，程序的线性地址空间被分割成为固定大小的页。在运行时，这些页被映射到计算机的物理内存或者硬盘中。当应用程序访问某逻辑地址时，处理机首先将逻辑地址转换为线性地址，再使用分页机制将线性地址转换为实际的物理地址。如果被访问的包含该线性地址的页面当前不在物理内存中时，处理机会产生一个页错误异常（#PF, Page-Fault Exception）。此时操作系统的异常处理程序应该将该页面调入物理内存中。

回忆图 3.1b，我们已经知道了在保护模式下处理机将逻辑地址转换为线性地址的过程，下面我们以 4KB 大小的页面为例，来看处理机是如何将线性地址转换为最终的物理地址的。

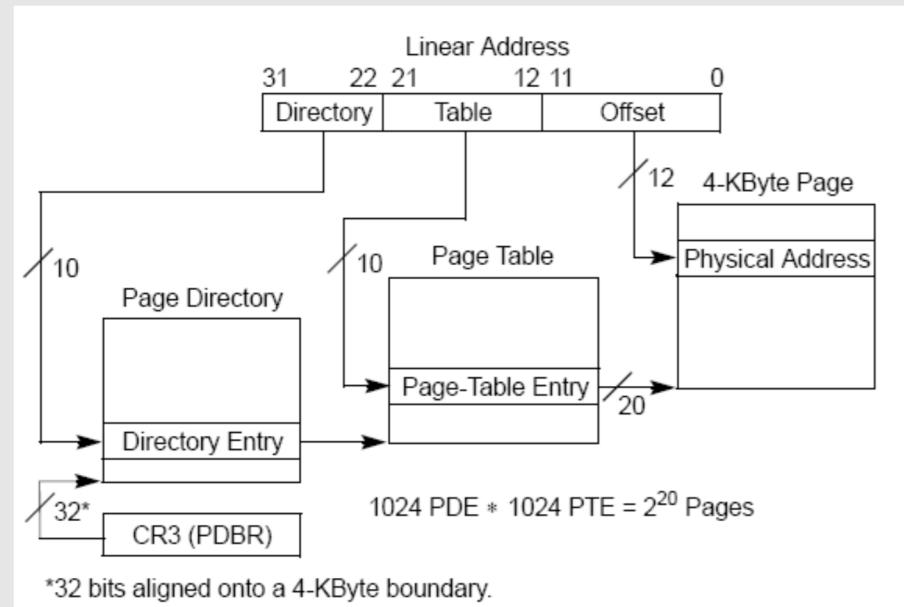


Fig 3.40: 线性地址转换 (4KB 页)

在图 3.40 里：

- CR3：是储存页目录基址的寄存器（Page Directory Base Register）。我们将设置好的页目录基址存放到 CR3 中，处理机在转换线性地址的时候就会去 CR3 中寻找页目录。

- Directory: 线性地址的第 22 到 31 位存放的是该线性地址所在页的页表记录在页目录中的偏移量，处理机通过该偏移量得到页目录项，进而得到该页表的地址。
- Table: 线性地址的第 12 到 21 位存放的是该线性地址所在页记录在页表中的偏移量，处理机通过该偏移量得到页表项，进而得到该页地址。
- Offset: 线性地址的第 0 到 11 位存放的是该线性地址在所属页表中的偏移量，处理机通过该偏移量得到该线性地址对应的物理地址。

总的来说，线性地址转换为物理地址的过程十分简单，主要是使用两层目录结构。不就是查表嘛，在我们日常生活中太经常遇到了，我们可以类比一下。比如我们要寻找南京大学 7 舍 205 室（线性地址），就需要先找到一张地图（通过 CR3 找到页目录），然后用地图先找到南京大学（在页目录中寻找页表），在南京大学里寻找 7 舍（在页表中寻找页），进去 7 舍寻找 205 室（在页中寻找偏移地址），这样我们就找到了目标邮件地址（物理地址）。

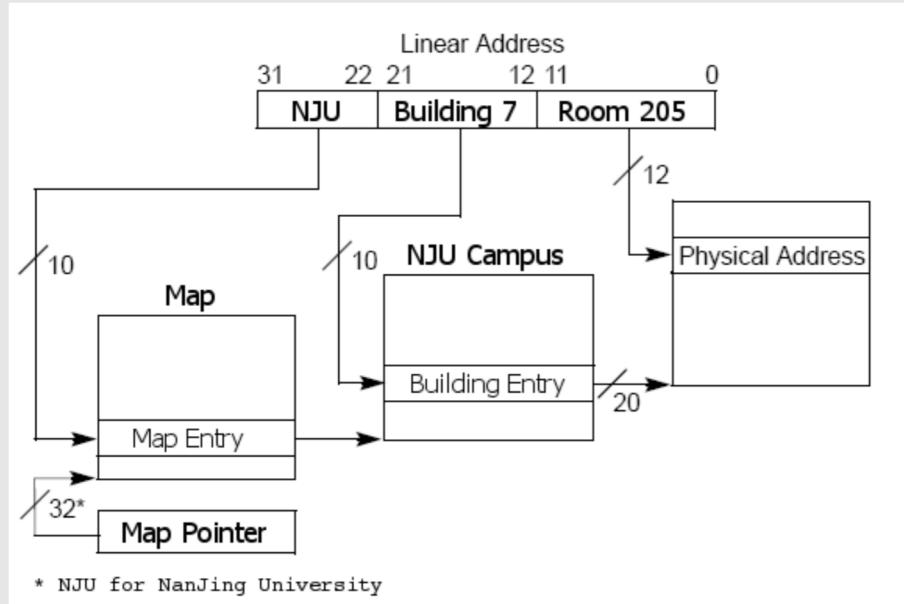


Fig 3.41: 邮件地址转换

3.5.2 启动分页机制

既然已经知道 IA-32 的分页机制，那么我们就可以尝试启动它了。但是在启动分页机制之前，仍然有一个问题需要回答：页目录项 (PDE, Page Directory Entry)、页表项 (PTE, Page Table Entry) 和页目录指针 (CR3) 的数据结构是什么？因为只有将页目录、页表和 CR3 设置正确，我们才能正确地启动和使用分页机制，所以必须知道它们的数据结构。

PDE 和 PTE

图 3.42 所示即为当页大小为 4KB 时，页目录项和页表项的数据结构。由于这里选项众多，下面我们将仅解释一下几个重要的位的用途，对其它位所代表的含义我们就不全部一一解释了。如果您想了解更多内容，请阅读 Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A: System Programming Guide 第 3 章第 7 节的有关内容。

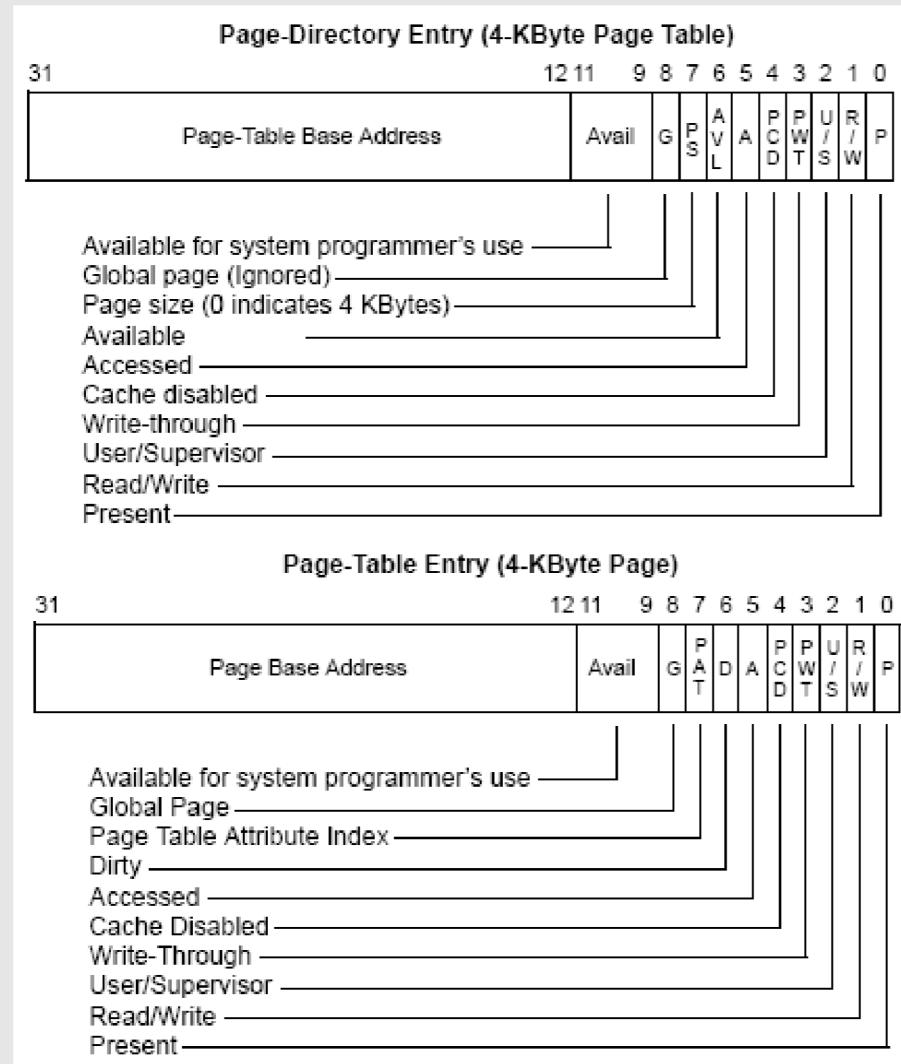


Fig 3.42: PDE 和 PTE 的数据结构 (4KB 页)

- Page(-Table) base address: 均为 20 位，指向页（表）的物理地址。由于这里只有 20 位，所以要求页（表）的物理地址要以 4KB 为边界对齐（即低 12 位为 0）。因为每个页表占用 4KB 空间。每个页表项占用 4 个字节，所以每个页表包含 1024 个页表项 ($1024 \times 4 = 4\text{KB}$)。

- Present(P) flag: 设置为 1 代表该页表项指向的页（或者该页表项指向的页表）存在于物理内存中，反之则不存在于物理内存中。处理机如果访问到该位为 0 的页表项或者页目录项，就会产生 #PF 异常。该位的值由操作系统设置，处理机不会修改该位。
- Read/Write(RW) flag: 设置为 0 表示该页表项指向的页（或者该页目录项指向的页表中的页）为只读，反之代表可读写。另外此位还与 U/S 位和 CR0 中的 WP 位相互作用，若 WP 为 0，则即使 RW 为 0，系统级程序仍然具有写权限。
- User/Supervisor(U/S) flag: 设置为 0 代表该页表项指向的页（或者该页目录项指向的页表中的页）的特权级为系统级 (CPL=0, 1, 2)，反之则为用户级 (CPL=3)。

与 PDE 和 PTE 类似，CR3 也是取页目录物理地址的高 20 位作为页目录的地址，所以同样要求页目录以 4KB 为边界对齐，因此每个页目录也包含 1024 个页目录项。CR3 的低 12 位中，第 3, 4 位分别作为 PCD 和 PWD 标志，与图 3.42 中的 PCD 和 PWD 标志作用类似，不再赘述。

开启分页机制示例代码

了解了页目录项、页表项和 CR3 的数据结构之后，我们就可以写一个实验性的代码来开启分页机制了。在上一节代码的基础上，首先我们来添加两个段，基址分别为 PageDirBase 和 PageTblBase，用来保存页目录和页表。

```

13 .set    PageDirBase, 0x200000 /* 2MB, base address of page directory */
14 .set    PageTblBase, 0x201000 /* 2MB+4KB, base address of page table */

35 LABEL_DESC_TSS:      Descriptor          0,           (TSSLen - 1), DA_386TSS
36 LABEL_DESC_PAGEDIR: Descriptor PageDirBase,        4096, DA_DRW
37 LABEL_DESC_PAGETBL: Descriptor PageTblBase,        1023, (DA_DRW | DA_LIMIT_4K) /*4M*/
38 .set    SelectorTSS,   (LABEL_DESC_TSS - LABEL_GDT)
39 .set    SelectorPageDir,(LABEL_DESC_PAGEDIR - LABEL_GDT)
40 .set    SelectorPageTbl,(LABEL_DESC_PAGETBL - LABEL_GDT)

```

Fig 3.43: 添加保存页目录和页表的段(节自chapter3/6/loader.S)

由于这两个段的基地址是我们直接写入的 magic number，不用编译时候计算，所以这里就不需要再初始化两个段的描述符了。页目录有 1024 个目录项，故占用 4KB 内存。

上文遇到的属性值 DA_LIMIT_4K 和将要遇到的 PG_P 等属性被定义在 pm.h 中：

```

23 .set    DA_32, 0x4000 /* 32-bit segment */
24 .set    DA_LIMIT_4K, 0x8000 /* 4K */

57 /* Page Attributes */
58 .set    PG_P,    1
59 .set    PG_RWR,  0
60 .set    PG_RWW,  2
61 .set    PG_USS,  0
62 .set    PG_USU,  4

```

Fig 3.44: 为分页机制添加的新属性(节自chapter3/6/pm.h)

然后添加一个函数，根据上面介绍的数据结构初始化页目录和所有页表，然后打开分页机制。由于我们只是尝试打开分页机制，为了简单起见，这个函数仅仅是将所有线性地址映射到与其相同的物理地址上，所以我们将有 1024 个页表， 1024×1024 个页。页表项所占空间将为： $1024 \times 1024 \times 4 = 4\text{MB}$ 。

```

287 SetupPaging:
288 /* Directly map linear addresses to physical addresses for simplification */
289     /* Init page directory, %ecx entries. */
290     mov    $(SelectorPageDir), %ax
291     mov    %ax, %es
292     mov    $1024, %ecx      /* Loop counter, num of page tables: 1024 */
293     xor    %edi, %edi
294     xor    %eax, %eax
295     /* Set PDE attributes(flags): P: 1, U/S: 1, R/W: 1. */
296     mov    $(PageTblBase | PG_P | PG_USU | PG_RWW), %eax
297 SP.1:
298     stosl      /* Store %eax to %es:%edi consecutively. */
299     add     $4096, %eax      /* Page tables are in sequential format. */
300     loop    SP.1          /* %ecx loops. */
301
302     /* Init page tables, %ecx pages. */
303     mov    $(SelectorPageTbl), %ax
304     mov    %ax, %es
305     mov    $(1024*1024), %ecx /* Loop counter, num of pages: 1024^2. */
306     xor    %edi, %edi
307     /* Set PTE attributes(flags): P:1, U/S: 1, R/W: 1. */
308     mov    $(PG_P | PG_USU | PG_RWW), %eax
309 SP.2:
310     stosl      /* Store %eax to %es:%edi consecutively. */
311     add     $4096, %eax      /* Pages are in sequential format. */
312     loop    SP.2          /* %ecx loops. */
313
314     mov    $(PageDirBase), %eax
315     mov    %eax, %cr3 /* Store base address of page table dir to %cr3. */
316     mov    %cr0, %eax
317     or     $0x80000000, %eax
318     mov    %eax, %cr0 /* Enable paging bit in %cr0. */
319     ret

```

Fig 3.45: 初始化页目录和页表，并打开分页机制的函数(节自chapter3/6/loader.S)

在这个函数中，首先我们初始化页目录，共有 1024 个页目录项，分别对应着 1024 个页表。由于已知第一个页表的基址为 PageTblBase，故我们只需每次增加一个页表的大小 ($1024 \times 4 = 4096$) 就得到下一个页表的基址，把这些基址赋给每个页目录项，循环 1024 次就能建立所有页目录项。在建立页目录项的时候需要加上属性 (flags)，由于我们每次增加的大小为 4096 (后 12 位都是 0)，所以在 PageTblBase 或上属性 PG_P | PG_USU | PG_RWW 后，加上 4096 并不影响这些属性，后面的页目录项也自动获得了这些属性：存在内存中、用户级别和可读写。

我们用同样的方法初始化所有页表。由于页表是连续存储的，我们可以转变初始化 1024 个页表为建立 1024×1024 个连续的页表项，由于我们将所有线性地址映射到与其相同的物理地址，所以第一个页的基址为 0，而每个页的大小为 4KB，故增加 4096 就得到下个页的基址，以此类推，我们就可以建立

起 1024^2 个页表项，对应于 1024^2 个页。每个页表项的属性（flag）为：存在内存中、用户级别和可读写。

其实这里我们是犯了一个大错误的， 1024^2 个页的大小共为 4GB，而我们为虚拟机分配的内存仅有 32MB，这些页不可能同时存在于内存中，所以我们不应该将所有页目录项和页表项的 P 标志设置为 1。不过这仅仅是为了简单起见而写的一个示例代码，而且我们的 loader 并没有访问高地址的内存，所以也不会出现什么严重的后果。

在初始化页目录和页表之后，我们就将页目录基址 PageDirBase 赋给 CR3（这里我们先忽略了 CR3 的 PCD 和 PWD 标志位）；然后将 CR0 的最高位设置为 1，就开启了 IA-32 处理机的分页机制。

最后，在进入保护模式后，马上调用上面的函数打开分页机制。

```
209 LABEL_SEG_CODE32:
210     call    SetupPaging    /* set up paging before 32-bit code */
```

Fig 3.46: 进入保护模式后马上打开分页机制(节自chapter3/6/loader.S)

将以上代码编译成镜像文件，用虚拟机运行之后，我们发现结果与图 3.38 完全相同，好像代码根本没有被修改过。这是因为我们仅仅是打开了分页机制，只是内存映射机制不同了，而负责向屏幕上打印信息的代码则和上节完全一样，所以屏显不会有任何变化。不过既然有屏显，就说明我们开启分页机制的代码成功了，否则如果内存映射错误，虚拟机的运行就会出现问题。

3.5.3 修正内存映射的错误

我们前面提到犯了一个大错误，将线性地址映射到了不存在的物理地址上。那么一个相对简单的解决办法是避免映射不存在的物理地址，也就是说我们不需要将页目录和页表中的条目全部用完。这样还有另外一个好处就是减小了页表所占用的空间，在上一节的例子中， 1024^2 个页表共占用了 4MB 的空间，而我们给虚拟机总共才分配了 32MB 的内存，这显然是极大的浪费。

为了避免映射不存在的物理地址，我们就需要知道机器的内存有多大。通常情况下我们使用功能码为 E820h 的 15h 中断来做这件事情。

INT 15h, EAX=E820h - 查询内存分布图

功能码为 E820h 的 15h 中断调用只能在实模式下使用，这个调用会返回所有安装在主机上的 RAM，以及被 BIOS 所保留的物理内存范围的内存分布。每次成功地调用这个接口都会返回一行物理地址信息，其中包括一块物理地址的范围以及这段地址的类型（可否被操作系统使用等）。

INT 15h, AX=E820h 和 INT 15h AH=88h 也可以返回内存分布信息，但是在这些旧接口返回信息与 INT 15h, EAX=E820h 不同时，应以 E820h 调用返回的信息为准。

表 3.4: INT 15h, EAX=E820h 中断的输入

EAX	功能码	E820h
EBX	后续	包含为得到下一行物理地址的“后续值”（类似于链表中的 next 指针）。 如果是第一次调用，EBX 必须为 0。
ES:DI	缓冲区指针	指向 BIOS 将要填充的地址范围描述符(Address Range Descriptor)结构。

ECX	缓冲区大小	缓冲区指针所指向的地址范围描述符结构的大小，以字节为单位。 无论 ES:DI 所指向的结构如何，BIOS 最多将会填充 ECX 字节。 BIOS 以及调用者应该最小支持 20 个字节长，未来的实现将会扩展此限制。
EDX	签名	'SMAP' - BIOS 将会使用此标志，对调用者将要请求的内存分布信息进行校验，这些信息会被 BIOS 放置到 ES:DI 所指向的结构中。

表 3.5: INT 15h, EAX=E820h 中断的输出

CF	进位标志	不进位(0)表示没有错误，否则则存在错误。
EAX	签名	'SMAP'
ES:DI	缓冲区指针	返回的地址范围描述符结构指针，和输入值相同。
ECX	缓冲区大小	BIOS 填充到地址范围描述符中的字节数量，返回的最小值是 20 个字节。
EBX	后续	存放为得到下一个地址描述符所需要的“后续值”，这个值的实际形式依赖于具体的 BIOS 实现。调用者不必关心它的具体形式，只需要在下次迭代时将其原样放置到 EBX 中，就可以通过它获取下一个地址范围描述符。注意，BIOS 将返回后续值 0 并且无进位来表示已经得到最后一个地址范围描述符。

上面两表中所提到的地址范围描述符(Address Range Descriptor Structure)结构为：

Offset(Bytes)	Name	Description
0	BaseAddrLow	Low 32 Bits of Base Address
4	BaseAddrHigh	High 32 Bits of Base Address
8	LengthLow	Low 32 Bits of Length in Bytes
12	LengthHigh	High 32 Bits of Length in Bytes
16	Type	Address type of this range.

上表中的 type 域可能的取值和意义有：

值	名称	描述
1	AddressRangeMemory	这段内存是“OS”可用的“RAM”。
2	AddressRangeReserved	这段内存正在被“OS”使用或者被“OS”保留，所以不可用。
Other	Undefined	未定义——保留给以后使用。任何“Other”值都应该被“OS”视为

有关 INT 15h, EAX=E820h 中断的更多信息，请参考 GNU GRUB 原作者 Erich Boleyn 在 GRUB 原始文档中的一个页面：<http://www.uruk.org/orig-grub/mem64mb.html>。

得到内存信息

现在我们就可以使用 INT 15h 中断来获得系统内存的分布信息了。简单地来说，就是将 INT 15h 返回的所有结果存储在一块内存区域中，然后在程序中利用这些信息完成内存的分配。因此我们首先要在数据段中为这些信息开辟一段空间：

```

68 /* 32-bit global data segment. */
69 LABEL_DATA:
70 _PMMessages: .ascii "Welcome to protect mode! ^-^\n\0"
71 _LDTMessage: .ascii "Aha, you jumped into a LDT segment.\n\0"
72 _ARDSTitle: .ascii "BaseAddrLo BaseAddrHi LengthLo LengthHi Type\n\0"
73 _RAMSizeMes: .ascii "RAM Size:\n\0"
74 _LFMes: .ascii "\n\0" /* Line Feed Message(New line) */
75 _AMECount: .4byte 0 /* Address Map Entry Counter */
76 _CursorPos: .4byte (80*2+0)*2 /* Screen Cursor position for printing */
77 _MemSize: .4byte 0 /* Usable Memory Size */
78 _ARDStruct: /* Address Range Descriptor Structure */
79     _BaseAddrLow: .4byte 0 /* Low 32 bits of base address */
80     _BaseAddrHigh: .4byte 0 /* High 32 bits of base address */
81     _LengthLow: .4byte 0 /* Low 32 bits of length in bytes */
82     _LengthHigh: .4byte 0 /* High 32 bits of length in bytes */
83     _Type: .4byte 0 /* Address type of this range: 0, 1, other */
84 _AddrMapBuf: .space 256, 0 /* Address map buffer */

85
86 .set PMMessages, (_PMMessages - LABEL_DATA)
87 .set LDTMessage, (_LDTMessage - LABEL_DATA)
88 .set ARDSTitle, (_ARDSTitle - LABEL_DATA)
89 .set RAMSizeMes, (_RAMSizeMes - LABEL_DATA)
90 .set LFMes, (_LFMes - LABEL_DATA)
91 .set AMECount, (_AMECount - LABEL_DATA)
92 .set CursorPos, (_CursorPos - LABEL_DATA)
93 .set MemSize, (_MemSize - LABEL_DATA)
94 .set ARDStruct, (_ARDStruct - LABEL_DATA)
95 .set BaseAddrLow, (_BaseAddrLow - LABEL_DATA)
96 .set BaseAddrHigh, (_BaseAddrHigh - LABEL_DATA)
97 .set LengthLow, (_LengthLow - LABEL_DATA)
98 .set LengthHigh, (_LengthHigh - LABEL_DATA)
99 .set Type, (_Type - LABEL_DATA)
100 .set AddrMapBuf, (_AddrMapBuf - LABEL_DATA)
101 .set DataLen, (. - LABEL_DATA)

```

Fig 3.47: 用来储存内存分布信息的数据段(节自chapter3/7/loader.S)

需要注意的一点是，为了方便起见，我们改变了前面章节的数据段符号的表示方法。以下划线开头的符号名字表示该符号只在实模式中使用，其对应的前面不带下划线的名字可以在保护模式中使用，这样比用前面加 Offset 的表达更清楚。

在上面这个数据段中，_AddMapBuf 就是存储系统地址分布信息的缓冲区，在这里我们设置其为 256 个字节大小，至多可以容纳 12 个 20 字节大小的地址范围描述符结构体；_ARDStruct 即为地址范围描述符结构体，共有 5 个域；_MemSize 用来保存计算出的可用内存大小；_AMECount 记录地址分布信息条数，即共返回多少个地址范围描述符结构；_CursorPos 是记录当前光标位置的全局变量，用来在不同函数中向屏幕连续打印信息。

下面我们就用 INT 15h 中断将得到的地址分布数据存储到缓冲区中：

```

154 /* Get System Address Map */
155 xor    %ebx, %ebx      /* EBX: Continuation, 0 */
156 mov    $(._AddrMapBuf), %di  /* ES:DI: Buffer Pointer, _AddrMapBuf */

```

```

157 BEGIN.loop:
158     mov    $0xe820, %eax          /* EAX: Function code, E820h */
159     mov    $20, %ecx             /* ECX: Buffer size, 20 */
160     mov    $0x534d4150, %edx      /* EDX: Signature 'SMAP' */
161     int    $0x15                /* INT 15h */
162     jc    BEGIN.getAMfail
163     add    $20, %di              /* Increase buffer pointer by 20(bytes) */
164     incl   (_AMECount)         /* Inc Address Map Entry Counter by 1 */
165     cmp    $0, %ebx              /* End of Address Map? */
166     jne    BEGIN.loop
167     jmp    BEGIN.getAMok
168 BEGIN.getAMfail:           /* Failed to get system address map */
169     movl   $0, (_AMECount)
170 BEGIN.getAMok:            /* Got system address map */
171

```

Fig 3.48: 用中断 INT 15h 得到地址分布数据(节自chapter3/7/loader.S)

得到地址分布数据的过程很简单，我们只需要设置好输入的寄存器，循环调用 INT 15h 中断，最终就能得到整个系统的地址分布信息，这些信息会被存储在 _AddMapBuf 缓冲区中，地址分布信息的条数被记录在 _AMECount 中。注意，这段代码是运行在实模式中，故应该使用前面带下划线的符号来引用数据。

为了验证得到的数据是否正确，我们可以将得到的地址分布信息打印到屏幕上，下面这个函数就是做的这个工作：

```

374 DispAddrMap:
375     push   %esi
376     push   %edi
377     push   %ecx
378
379     mov    $(AddrMapBuf), %esi /* int *p = AddrMapBuf; */
380     mov    (AMECount), %ecx   /* for (int i=0; i<AMECount; i++) { */
381 DMS.loop:
382     mov    $5, %edx          /* int j = 5; */
383     mov    $(ARDStruct), %edi /* int *q = (int *)ARDStruct; */
384 DMS.1:
385     push   (%esi)          /* do { */
386     call   DispInt          /*     printf("%xh", *p); */
387     pop    %eax
388     stosl
389     add    $4, %esi          /*     *q++ = *p; */
390     dec    %edx              /*     p++; */
391     cmp    $0, %edx          /*     j--; */
392     jnz   DMS.1            /* } while(j != 0); */
393     call   DispLF          /*     printf("\n"); */
394     cmpl  $1, (Type)        /*     if (Type == AddressRangMemory){ */
395     jne   DMS.2
396     mov    (BaseAddrLow), %eax /*         if(ARDStruct.BaseAddrLow */
397     add    (LengthLow), %eax /*             + ARDStruct.LengthLow */
398     cmp    (MemSize), %eax  /*             > MemSize){ */
399     jb    DMS.2            /*             MemSize = BaseAddrLow + LengthLow; */
400     mov    %eax, (MemSize)  /*         } */

```

```

401 DMS.2:          /*  */
402     loop    DMS.loop      /*  */
403
404     call    DispLF        /* printf("\n");           */
405     push    $(RAMSizeMes)
406     call    DispStr       /* printf("%s", RAMSizeMes); */
407     add    $4, %esp
408
409     pushl   (MemSize)
410     call    DispInt       /* printf("%x", MemSize); */
411     add    $4, %esp
412     call    DispLF        /* printf("\n");           */
413
414     pop    %ecx
415     pop    %edi
416     pop    %esi
417     ret
418
419 #include "lib.h"

```

Fig 3.49: 将地址分布信息打印到屏幕上(节自chapter3/7/loader.S)

这个函数的流程我们已经在旁边给出了 C 语言格式的注释。简单地来说，它就是遍历整个储存地址分布信息的缓冲区 `_AddMapBuf`，将缓冲区中的数据按照地址范围描述符的格式打印出来，并通过可用内存地址的最高值来计算可用内存的大小。

这个函数中使用了一些辅助的打印函数，比如 `DispInt`, `DispStr` 等，这些函数定义在 `lib.h` 头文件中，提供这样一些库函数能方便我们的编程：

```

1 /* chapter3/7/lib.h
2
3 Author: Wenbo Yang <solrex@gmail.com> <http://solrex.cn>
4
5 This file is part of the source code of book "Write Your Own OS with Free
6 and Open Source Software". Homepage @ <http://share.solrex.cn/WriteOS/>.
7
8 This file is licensed under the GNU General Public License; either
9 version 3 of the License, or (at your option) any later version. */
10
11 DispAL:
12     push    %ecx
13     push    %edx
14     push    %edi
15     mov    (CursorPos), %edi
16     mov    $0xf, %ah
17     mov    %al, %dl
18     shrb   $4, %al
19     mov    $2, %ecx
20 DispAL.begin:
21     and    $0xf, %al
22     cmp    $9, %al
23     ja     DispAL.1
24     add    $'0', %al

```

```

25      jmp    DispAL.2
26 DispAL.1:
27      sub    $0xA, %al
28      add    '$A', %al
29 DispAL.2:
30      mov    %ax, %gs:(%edi)
31      add    $2, %edi
32      mov    %dl, %al
33      loop   DispAL.begin
34      mov    %edi, (CursorPos)
35      pop    %edi
36      pop    %edx
37      pop    %ecx
38      ret
39
40 DispInt:
41      movl   4(%esp), %eax
42      shr    $24, %eax
43      call   DispAL
44      movl   4(%esp), %eax
45      shr    $16, %eax
46      call   DispAL
47      movl   4(%esp), %eax
48      shr    $8, %eax
49      call   DispAL
50      movl   4(%esp), %eax
51      call   DispAL
52      movb   $0x7, %ah
53      movb   '$h', %al
54      pushl  %edi
55      movl   (CursorPos), %edi
56      movw   %ax, %gs:(%edi)
57      addl   $4, %edi
58      movl   %edi, (CursorPos)
59      popl   %edi
60      ret
61
62 DispStr:
63      pushl  %ebp
64      movl   %esp, %ebp
65      pushl  %ebx
66      pushl  %esi
67      pushl  %edi
68      movl   8(%ebp), %esi
69      movl   (CursorPos), %edi
70      movb   $0xF, %ah
71 DispStr.1:
72      lodsb
73      testb  %al, %al
74      jz     DispStr.2
75      cmpb   $0xA, %al
76      jnz    DispStr.3
77      pushl  %eax
78      movl   %edi, %eax
79      movb   $160, %bl

```

```

80    divb    %bl
81    andl    $0xFF, %eax
82    incl    %eax
83    movb    $160, %bl
84    mulb    %bl
85    movl    %eax, %edi
86    popl    %eax
87    jmp     DispStr.1
88 DispStr.3:
89    movw    %ax, %gs:(%edi)
90    addl    $2, %edi
91    jmp     DispStr.1
92 DispStr.2:
93    movl    %edi, (CursorPos)
94    popl    %edi
95    popl    %esi
96    popl    %ebx
97    popl    %ebp
98    ret
99
100 DispLF:
101   pushl   $(LFMes)
102   call    DispStr
103   addl    $4, %esp
104   ret

```

Fig 3.50: chapter3/7/lib.h

因为我们已经得到了可用内存的大小，就可以根据可用内存的大小来调整我们的内存映射范围。

```

326 SetupPaging:
327 /* Directly map linear addresses to physical addresses for simplification */
328 /* Get usable PDE number from memory size. */
329 xor    %edx, %edx
330 mov    (MemSize), %eax          /* Memory Size */
331 mov    $0x400000, %ebx          /* Page table size(bytes), 1024*1024*4 */
332 div    %ebx                  /* temp = MemSize/4M */
333 mov    %eax, %ecx
334 test   %edx, %edx
335 jz    SP.no_remainder
336 inc    %ecx
337 SP.no_remainder:
338 push   %ecx                  /* number of PDE = ceil(temp) */
339
340 /* Init page table directories, %ecx entries. */
341 mov    $(SelectorPageDir), %ax
342 mov    %ax, %es
343 xor    %edi, %edi
344 xor    %eax, %eax
345 /* Set PDE attributes(flags): P: 1, U/S: 1, R/W: 1. */
346 mov    $(PageTblBase | PG_P | PG_USU | PG_RWW), %eax
347 SP.1:
348 stosl   /* Store %eax to %es:%edi consecutively. */

```

```

349 add    $4096, %eax      /* Page tables are in sequential format. */
350 loop   SP.1           /* %ecx loops. */
351
352 /* Init page tables, %ecx*1024 pages. */
353 mov    $(SelectorPageTbl), %ax
354 mov    %ax, %es
355 pop   %eax           /* Pop pushed ecx(number of PDE) */
356 shl    $10, %eax       /* Loop counter, num of pages: 1024*%ecx. */
357 mov    %eax, %ecx
358 xor    %edi, %edi
359 /* Set PTE attributes(flags): P:1, U/S: 1, R/W: 1. */
360 mov    $(PG_P | PG_USU | PG_RWW), %eax
361 SP.2:
362 stosl          /* Store %eax to %es:%edi consecutively. */
363 add    $4096, %eax      /* Pages are in sequential format. */
364 loop   SP.2           /* %ecx loops. */
365
366 mov    $(PageDirBase), %eax
367 mov    %eax, %cr3 /* Store base address of page table dir to %cr3. */
368 mov    %cr0, %eax
369 or     $0x80000000, %eax
370 mov    %eax, %cr0 /* Enable paging bit in %cr0. */
371 ret
372

```

Fig 3.51: 根据可用内存大小调整内存映射范围(节自chapter3/7/loader.S)

在 SetupPaging 函数中，我们首先根据可用内存计算页目录项(PDE)的数目（即页表的数目）。每个页表可以包含 1024 个页，可以指向 4MB 大小的内存，所以我们只需要用可用内存数除以 4MB，并向上取整，就能得到需要的页目录项数，然后再根据页目录项数初始化页目录项和页表项。

仔细看代码可以发现，我们这里仍然映射了一些不存在的物理地址。因为可用内存数除以 4MB 未必是整数，但是我们计算页表项数的时候使用的是 页目录项*1024(%ecx*1024)，那么在最后一个页表的末尾就可能存在一些页映射到不可用的内存地址；并且从下文中屏幕打印的信息我们可以知道，不是所有从 0 到 MemSize 的内存都是可用的，而我们在 SetupPaging 函数中却是以连续的方式初始化所有的页，那么中间必然有一些页映射到不可用的内存地址。但总的来说，在新的 SetupPaging 函数中大大减少了页表的数量，比如当 MemSize = 32MB 的时候，我们仅仅需要 8 个页表，而不是原来的 1024 个页表，因之所有页表占用的内存也从 4MB 减小到了 32KB。

至于那些映射错误，既然我们已经获得了整个系统地址分布的情况（存储在 _AddMapBuf 中），只需要根据分布情况，增加一些判断来避免映射到不存在的内存地址即可，这里就不多做演示了。

下面就可以在程序中调用我们新增加的函数：

```

266 push   $(ARDSTitle)      /* Display addr range descriptor struct title */
267 call   DispStr
268 add    $4, %esp
269 call   DispAddrMap      /* Display system address map */
270
271 call   SetupPaging      /* Setup and enable paging */

```

Fig 3.52: 调用显示内存范围和开启分页机制的函数(节自chapter3/7/loader.S)

将上面的代码编译成镜像，用虚拟机加载执行的结果如图 3.53 所示。在进入保护模式之后，首先函数 DispAddrMap 打印出虚拟机系统的内存分布信息，随后是根据内存分布信息计算出来的内存大小。接着 SetupPaging 函数根据内存大小开启处理机的分页机制。虽然我们无法直接看到分页机制启动的效果，但是接下来代码的正常执行可以告诉我们分页机制的开启是成功的。

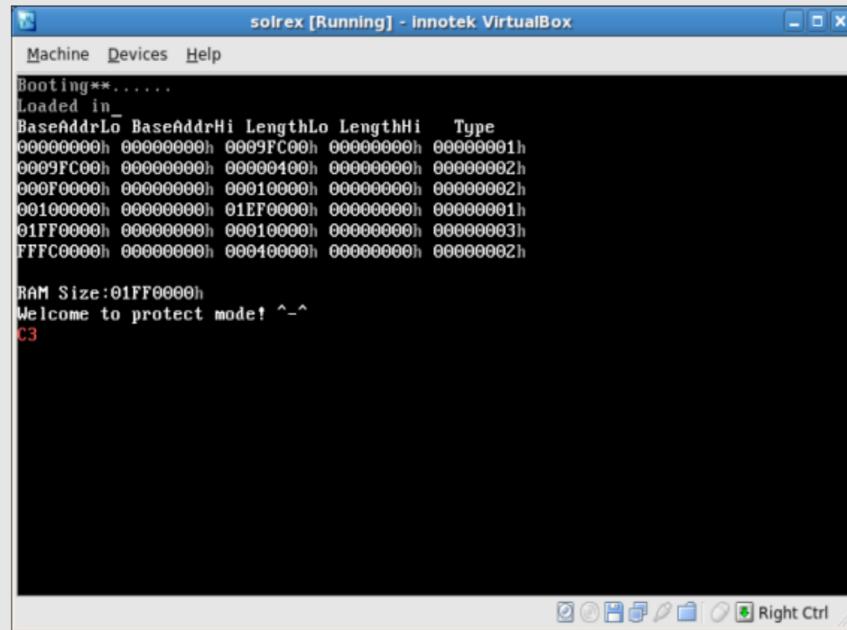


Fig 3.53: 修正内存映射的错误

打印的内存信息告诉我们，系统可用的内存大小是 0x01ff0000，大约是 31.9325MB，我们就可以根据该信息将为页表段分配内存的大小减少到 32KB：

```
LABEL_DESC_PAGETBL: Descriptor PageTblBase,           4096*8-1, DA_DRW /* 32K */
```

3.5.4 体验虚拟内存

在上面两个小节中，我们在开启分页机制时都是直接将线性地址映射到与之相同的物理地址上。这样能简化我们分页的函数，但这显然不是分页机制的最终目的。回想本节开头所提到的虚拟内存机制，它可以使每个应用程序都以为自己拥有完整连续的内存空间，即应用程序看到的只是一个完整的线性地址空间，但只有在使用某地址块时操作系统才将该地址块映射到实际的物理地址上，这个块的最小单位就是“页”。这样就会出现一种情况，比如用 GDB 调试两个不同的程序时，我们会发现两个程序使用的是同一块地址，但地址的内容却完全不同。这就是因为我们看到的只是线性地址，它们会被映射到不同的物理地址上。

想想这一机制对程序员有多大帮助吧！程序员在写应用程序时不必成天为自己怎么分配内存而担心，也不必恐惧别的程序会覆盖自己的内存空间（当然，作为 OS 程序员，这种恐惧是应当时常放在心

上的），在他的眼里，自己有 4GB(32-bit CPU) 的内存可以用——就好像自己拥有一个城市一样，太酷了！他可以想住在哪里就住在哪里，想分配什么地址就分配什么地址！虽然实际上还是会受到编译器和运行效率的限制，不过无论如何，自由度大大增加了。

下面我们就用一个小例子体验一下简单的虚拟内存机制：两次访问同一线性地址时，执行的却是不同的函数（还记得赫拉克利特的那句话吗 ⊙）。

如何做呢？首先，我们需要准备两个函数，每个函数各自打印函数自身名称信息；其次，因为我们希望两次访问同一线性地址执行不同函数，那么就需要准备两套页目录和页表，两套页表中对应该线性地址页的 PTE 中的基址（物理地址）分别指向我们上面准备的两个不同的函数，这样切换页表就能实现同一线性地址映射不同的函数上；最后呢，在示例函数中切换页表，并在切换前后调用同一个线性地址上的函数，看打印的信息是否相同。如果打印的信息不同，就说明该线性地址在页表切换前后分别映射到了不同的函数上。

准备两个打印自身信息的函数很简单：

```

502 /* Function foo, print message "Foo". */
503 foo:
504     .set    FooOffset, (. - LABEL_SEG_CODE32)
505     mov    $0xc, %ah          /* 0000: background black, 1100: font red */
506     mov    '$F', %al
507     mov    %ax, %gs:((80 * 12 + 3) * 2)    /* Line 12, column 3 */
508     mov    '$o', %al
509     mov    %ax, %gs:((80 * 12 + 4) * 2)    /* Line 12, column 4 */
510     mov    %ax, %gs:((80 * 12 + 5) * 2)    /* Line 12, column 5 */
511     lret
512     .set    FooLen, (. - foo)
513
514 /* Function bar, print message "Bar". */
515 bar:
516     .set    BarOffset, (. - LABEL_SEG_CODE32)
517     mov    $0xc, %ah          /* 0000: background black, 1100: font red */
518     mov    '$B', %al
519     mov    %ax, %gs:((80 * 12 + 7) * 2)    /* Line 12, column 7 */
520     mov    '$a', %al
521     mov    %ax, %gs:((80 * 12 + 8) * 2)    /* Line 12, column 8 */
522     mov    '$r', %al
523     mov    %ax, %gs:((80 * 12 + 9) * 2)    /* Line 12, column 9 */
524     lret
525     .set    BarLen, (. - bar)

```

Fig 3.54: 两个打印自身信息的函数 Foo 和 Bar (节自chapter3/8/loader.S)

这两个函数的内容很简单，就是在屏幕的不同位置打印自身的信息。foo 函数在屏幕的第 12 行 3-5 列打印 Foo 三个红色字符，bar 也一样，在屏幕的第 12 行 6-8 列打印 Bar 三个红色字符。至于为什么使用 lret 而不是 ret 指令返回，我们下面会说明。

编译之后，这两个函数的确是位于不同的物理地址上，但是这两个函数入口的物理地址却不是按照页面对齐的。回想一下图 3.42，PTE 中的页的物理地址要以 4KB 为边界对齐。这样我们才好将同一个页分别映射到这两个函数的入口物理地址。要想使这两个函数入口的物理地址按照 4KB 对齐，有两种方法：一种是直接使代码对齐 4KB，比如可以在每个函数入口前使用 GAS 的 .align 4096 伪指令使当前地址对齐到 4KB 上，但是这种方法会使得目标文件中填充大量的 0，我们不采用这种方法；另一

种方法是定义两个 4KB 对齐的物理地址，然后将 foo 和 bar 分别拷贝到这两个地址中。我们下面采取这个方法。定义两个 4KB 对齐的物理地址，一个线性地址和两个页表对应的页目录和页表基址：

```

13 .set    PageDirBase0, 0x200000 /* 2MB, base address of page directory */
14 .set    PageTblBase0, 0x201000 /* 2MB+4KB, base address of page table */
15 .set    PageDirBase1, 0x210000 /* 2MB+64KB, base address of page directory */
16 .set    PageTblBase1, 0x211000 /* 2MB+68KB, base address of page table */

17
18 .set    FuncLinAddr, 0x401000 /* Linear address of a function. */
19 .set    FooPhyAddr, 0x401000 /* Physical address of function foo. */
20 .set    BarPhyAddr, 0x501000 /* Physical address of function bar. */

```

Fig 3.55: 4KB 对齐的物理地址(节自chapter3/8/loader.S)

在上图中，FooPhyAddr 和 BarPhyAddr 分别是 foo 和 bar 将要被拷贝到的目标地址。FuncLinAddr 则是我们要执行函数调用所使用的线性地址。再加上有了两个页表对应的页目录和页表基址，我们就可以使用这些信息，修改上一小节使用的 SetupPaging 函数，分别设置两个页表了：

```

334 SetupPaging:
335     /* Get usable PDE number from memory size. */
336     xor    %edx, %edx
337     mov    (MemSize), %eax      /* Memory Size */
338     mov    $0x400000, %ebx      /* Page table size(bytes), 1024*1024*4 */
339     div    %ebx                /* temp = MemSize/4M */
340     mov    %eax, %ecx
341     test   %edx, %edx
342     jz    SP.no_remainder
343     inc    %ecx
344 SP.no_remainder:
345     mov    %ecx, (PageTableNum) /* number of PDE = ceil(temp) */
346
347     /* Directly map linear addresses to physical addresses. */
348     /* Init page table directories of PageDir0, %ecx entries. */
349     mov    $(SelectorFlatRW), %ax
350     mov    %ax, %es
351     mov    $(PageDirBase0), %edi
352     xor    %eax, %eax
353     /* Set PDE attributes(flags): P: 1, U/S: 1, R/W: 1. */
354     mov    $(PageTblBase0 | PG_P | PG_USU | PG_RWW), %eax
355 SP.1:
356     stosl              /* Store %eax to %es:%edi consecutively. */
357     add    $4096, %eax    /* Page tables are in sequential format. */
358     loop   SP.1          /* %ecx loops. */
359
360     /* Init page tables of PageTbl0, (PageTableNum)*1024 pages. */
361     mov    (PageTableNum), %eax /* Get saved ecx(number of PDE) */
362     shl    $10, %eax        /* Loop counter, pages: 1024*(PageTableNum). */
363     mov    %eax, %ecx
364     mov    $(PageTblBase0), %edi
365     /* Set PTE attributes(flags): P:1, U/S: 1, R/W: 1. */
366     mov    $(PG_P | PG_USU | PG_RWW), %eax

```

```

367 SP.2:
368     stosl          /* Store %eax to %es:%edi consecutively. */
369     add    $4096, %eax      /* Pages are in sequential format. */
370     loop   SP.2          /* %ecx loops. */
371
372     /* Do the same thing for PageDir1 and PageTbl1. */
373
374     /* Init page table directories of PageDir1, (PageTableNum) entries. */
375     mov    $(SelectorFlatRW), %ax
376     mov    %ax, %es
377     mov    $(PageDirBase1), %edi
378     xor    %eax, %eax
379     /* Set PDE attributes(flags): P: 1, U/S: 1, R/W: 1. */
380     mov    $(PageTblBase1 | PG_P | PG_USU | PG_RWW), %eax
381     mov    (PageTableNum), %ecx
382 SP.3:
383     stosl          /* Store %eax to %es:%edi consecutively. */
384     add    $4096, %eax      /* Page tables are in sequential format. */
385     loop   SP.3          /* %ecx loops. */
386
387     /* Init page tables of PageTbl1, (PageTableNum)*1024 pages. */
388     mov    (PageTableNum), %eax /* Get saved ecx(number of PDE) */
389     shl    $10, %eax        /* Loop counter: 1024*(PageTableNum). */
390     mov    %eax, %ecx
391     mov    $(PageTblBase1), %edi
392     /* Set PTE attributes(flags): P:1, U/S: 1, R/W: 1. */
393     mov    $(PG_P | PG_USU | PG_RWW), %eax
394 SP.4:
395     stosl          /* Store %eax to %es:%edi consecutively. */
396     add    $4096, %eax      /* Pages are in sequential format. */
397     loop   SP.4          /* %ecx loops. */
398
399     /* Locate and modify the page that includes linear address FuncLinAddr.
400      * Assume memory is larger than 8 MB. */
401     mov    $(FuncLinAddr), %eax
402     shr    $12, %eax        /* Get index of PTE which contains FuncLinAddr. */
403     shl    $2, %eax         /* PTE size is 4-bytes. */
404     add    $(PageTblBase1), %eax /* Get the pointer to that PTE. */
405     /* Modify the PTE of the page which contains FuncLinAddr. */
406     movl   $(BarPhyAddr | PG_P | PG_USU | PG_RWW), %es:(%eax)
407
408     /* Use PageDirBase0 first. */
409     mov    $(PageDirBase0), %eax
410     mov    %eax, %cr3 /* Store base address of page table dir to %cr3. */
411
412     /* Enable paging bit in %cr0. */
413     mov    %cr0, %eax
414     or     $0x80000000, %eax
415     mov    %eax, %cr0
416     ret

```

Fig 3.56: 设置两个页表并开启分页机制(节自chapter3/8/loader.S)

这里的 SetupPaging 函数比前一小节要复杂了不少，但是增加的大部分内容却是重复的，在上一

小节的 SetupPaging 函数中，我们只初始化了一个页表，这里我们初始化了两个，除了页目录基址和页表基址以外，这两个页表的初始化过程是完全相同的。所不同的是在初始化第二个页表之后，我们修改了第二个页表中包含线性地址 FuncLinAddr 那个页所对应的物理地址。所以，在第一个页表中，所有的线性地址都映射到相应的物理地址上，线性地址 FuncLinAddr 所对应的物理地址就是与它相同的 FooPhyAddr；在第二个页表中，所有的线性地址也是映射到相应的物理地址，除了线性地址 FuncLinAddr 所在的页被修改为映射到物理地址 BarPhyAddr 上。

这个函数里使用了一个新段 SelectorFlatRW 和一个新的变量 PageTableNum，它们的定义如下：

```

42 LABEL_DESC_FLAT_C: Descriptor      0,          0xfffff, (DA_CR|DA_32|DA_LIMIT_4K)
43 LABEL_DESC_FLAT_RW: Descriptor     0,          0xfffff, (DA_DRW|DA_LIMIT_4K)

63 .set    SelectorFlatC , (LABEL_DESC_FLAT_C - LABEL_GDT)
64 .set    SelectorFlatRW , (LABEL_DESC_FLAT_RW - LABEL_GDT)

90 _PageTableNum: .4byte 0           /* Number of page tables */
91 _AddrMapBuf: .space 256, 0        /* Address map buffer */

107 .set   PageTableNum,      (_PageTableNum - LABEL_DATA)
108 .set   AddrMapBuf,       (_AddrMapBuf - LABEL_DATA)

```

Fig 3.57: 添加的新段和变量(节自chapter3/8/loader.S)

其中两个新段 LABEL_DESC_FLAT_C 和 LABEL_DESC_FLAT_RW 的基址都是 0，代表整个线性地址空间。使用这两个段时，指令或者位置的段偏移就是该指令或者位置的线性地址。因此在 SetupPaging 函数中使用 LABEL_DESC_FLAT_RW 作为 ES 段，那么偏移量 PageDirBase0 所指向的线性地址就是线性地址 PageDirBase0，这样就不需要上一小节所使用的 LABEL_DESC_PAGEDIR 段了。

在 SetupPaging 函数最后，我们依然使用第一个页表，然后开启分页机制。第二个页表虽然被初始化了，但是并没有使用。

最后，我们写一个示例函数来完成拷贝函数、切换页表和调用同一线性地址的过程：

```

464 VMDemo:
465     mov    %cs, %ax
466     mov    %ax, %ds          /* Set %ds to code segment. */
467     mov    $(SelectorFlatRW), %ax
468     mov    %ax, %es          /* Set %es to flat memory segment. */

469
470     pushl  $(FooLen)
471     pushl  $(FooOffset)
472     pushl  $(FooPhyAddr)
473     call   MemCpy          /* Copy function foo to FooPhyAddr. */
474     add    $12, %esp

475
476     pushl  $(BarLen)
477     pushl  $(BarOffset)
478     pushl  $(BarPhyAddr)
479     call   MemCpy          /* Copy function bar to BarPhyAddr. */
480     add    $12, %esp

```

```

482     /* Restore data segment selector to %ds and %es. */
483     mov      $(SelectorData), %ax
484     mov      %ax, %ds
485     mov      %ax, %es
486
487     /* Setup and start paging*/
488     call    SetupPaging
489
490     /* Function call 1, should print "Foo". */
491     lcall   $(SelectorFlatC), $(FuncLinAddr)
492
493     /* Change current PDBR from PageDirBase0 to PageDirBase1. */
494     mov      $(PageDirBase1), %eax
495     mov      %eax, %cr3
496
497     /* Function call 2, should print "Bar". */
498     lcall   $(SelectorFlatC), $(FuncLinAddr)
499
500     ret

```

Fig 3.58: 拷贝函数、切换页表并调用同一线性地址的示例函数(节自chapter3/8/loader.S)

在函数 VMDemo 中，我们首先拷贝函数 foo 和 bar 的内存到目标物理地址 FooPhyAddr 和 BarPhyAddr。由于 foo 和 bar 处于代码段，如果想要拷贝成功，就需要把代码段的属性改为可读，即为 LABEL_DESC_CODE32 加上可读属性：

```

32 LABEL_GDT:           Descriptor      0,          0, 0
33 LABEL_DESC_CODE32:   Descriptor      0, (SegCode32Len - 1), (DA_CR | DA_32)

```

然后就可以将 DS 设置为代码段，ES 设置为线性地址段，将每个函数的长度、段偏移和目标物理地址（未分页前线性地址和物理地址等同）作为 MemCpy 的参数，执行函数调用就能完成拷贝。MemCpy 函数定义于库文件 lib.h 中：

```

106 MemCpy:
107     pushl  %ebp
108     mov    %esp, %ebp
109
110     pushl  %esi
111     pushl  %edi
112     pushl  %ecx
113
114     mov    8(%ebp), %edi    /* Destination */
115     mov    12(%ebp), %esi   /* Source */
116     mov    16(%ebp), %ecx   /* Counter */
117 MemCpy.1:
118     cmp    $0, %ecx    /* Loop counter */
119     jz     MemCpy.2
120     movb   %ds:(%esi), %al
121     inc    %esi
122     movb   %al, %es:(%edi)
123     inc    %edi
124     dec    %ecx

```

```

125     jmp      MemCopy.1
126 MemCopy.2:
127     mov      8(%ebp), %eax
128     pop      %ecx
129     pop      %edi
130     pop      %esi
131     mov      %ebp, %esp
132     pop      %ebp
133     ret

```

Fig 3.59: MemCopy 函数定义(节自chapter3/8/lib.h)

拷贝完成之后，仍需要恢复 DS 和 ES 为原来的数据段。然后执行 SetupPaging 开启分页机制，首先使用的是第一个页表，我们直接调用线性地址 FuncLinAddr，然后切换到第二个页表，再次调用线性地址 FuncLinAddr。我们可以看到，这两次调用的指令是一模一样的，如果没有虚拟内存机制，那么两次调用执行的效果应该是一样的。

最后在程序中加入对 VMDemo 的调用：

```

279     call      VMDemo          /* Calling Virtual Memory demo function */
280
281     push      $(PMMMessage)    /* Display PMMessage */

```

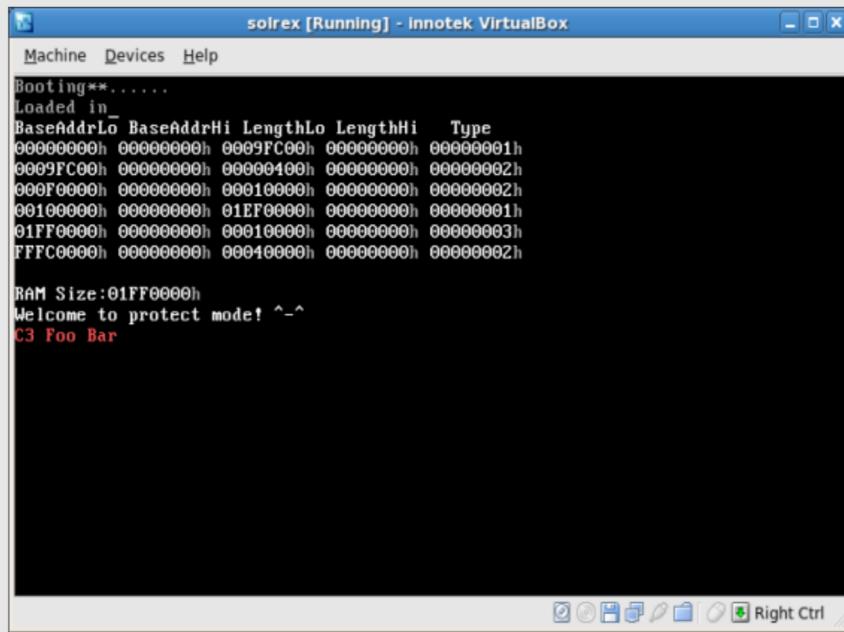


Fig 3.60: 体验虚拟内存

我们将上面的代码编译成镜像，用虚拟机加载执行，结果如图 3.60 所示。图中红色的 Foo 和 Bar 都被打印出来，说明 foo 函数和 bar 都被执行，我们的页表切换起作用了。这样我们就可以看到，在不

同的页表中，同一个线性地址可能代表不同的物理地址，使用该地址进行函数调用，结果执行的可能是不同的函数。在一般的支持虚拟内存的多进程操作系统中，所采用的也是类似的方法，按照进程可能使用的空间大小为每个进程分配一个页表，进程切换的时候会进行页表的切换，这样进程所见的内存就完全是自己的线性地址空间内存。在物理地址上增加了一层抽象层，既方便了应用程序编程，又能够更好更安全地管理内存。

3.6 结语

本章主要介绍了保护模式下的内存使用和管理方式，通过几个小例子，逐步地介绍了进入保护模式、段式存储、特权级和页式存储的一些基本概念和实际操作。一般的操作系统书籍对保护模式总是一带而过，少有介绍利用保护模式特性的方法。本章提供的这些例子虽然很小而且很不完善，但是仍然希望它们能够作为读者对保护模式更深入了解的钥匙，为读者打开进一步学习操作系统编程之门。

CHAPTER 4

中断
