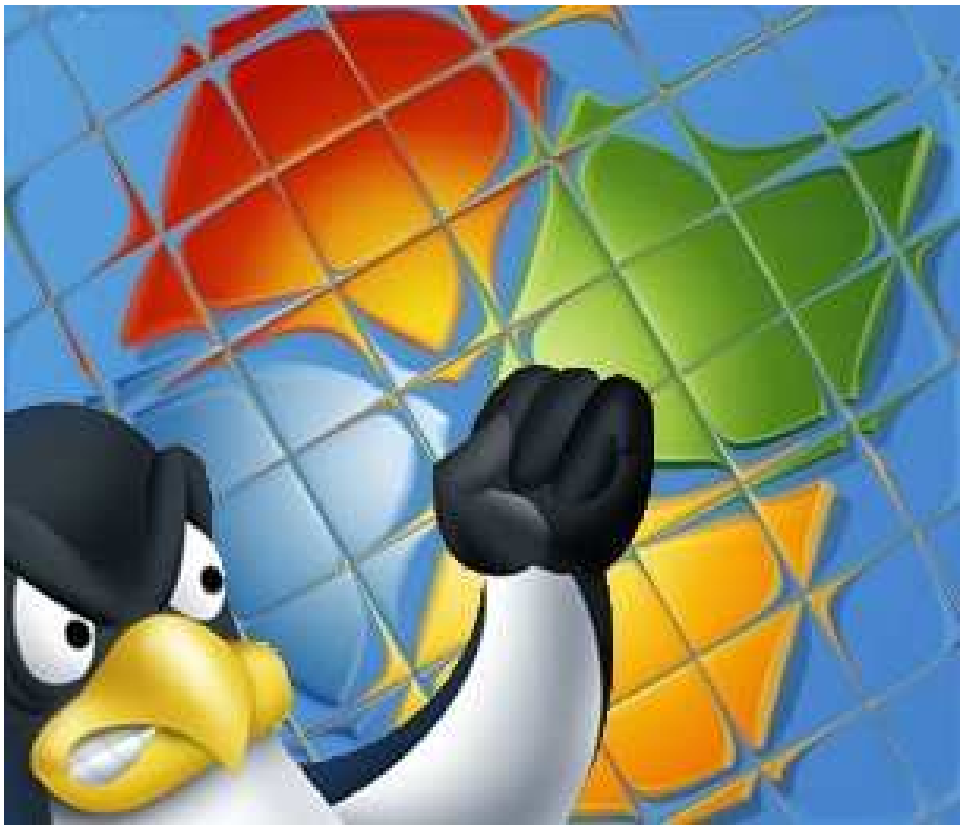
使用Open Source 軟體

自己動手寫作業系統

Write Your Own OS with Free and Open Source Software

Alpha Edition



楊文博 著

文件建立時間： 2011 年 10 月 31 日 02:04 Revision 2

版權聲明

相關鏈接

您可以在本書 官方網站： http://share.solrex.cn/WriteOS/ 下載到本書的最新版本和附帶的全部 source code。

由於此版本非最終發布版, 如果您對本書感興趣，請關注作者在其 blog (http://blog.solrex.cn) 上發布的更新公告。如果您發現本書的錯誤或者有好的建議, 請到 http://code.google.com/p/writeos/issues/list 檢視並報告您的發現, 對此作者將非常感謝。

寫在前面的話

本書起源於中國電子工業出版社出版的一本書：《自己動手寫作業系統》（于淵著）。我對《自己動手寫作業系統》這本書中使用商業軟體做為演示平台比較驚訝，因為不是每個人都買得起正版的軟體，尤其是窮學生。我想《自》所面對的主要讀者也應該是學生，那麼一本介紹只有商業軟體才能實作編程技巧的書將會逼著窮學生去使用盜版，這是非常罪惡的行為 ☹。

由於本人是一個 Linux 使用者，一個open source 軟體的擁護者，所以就試著使用open source 軟體實作這本書中的所有 demo ，並在自己的blog上進行推廣。後來我覺得，為什麼我不能自己寫本書呢？這樣我就能插入漂亮的插圖，寫更詳盡的介紹而不怕篇幅過長，更容易讓讀者接受也更容易傳播，所以我就開始寫這本《 使用源件-自己手操作系 》。

定下寫一本書的目標畢竟不像寫一篇blog 文章，我將盡量詳盡的介紹我使用的方法和過程，以圖能讓不同技術背景的讀者都能通暢地完成閱讀。但是自己寫並且排版一本書不是很輕鬆的事情，需要耗費大量時間，所以我只能抽空一點一點的將這本書堆砌起來，這也是您之所以在本書封面看到本書版本號的原因 ☺。

本書的最終目標是成為一本大學"計算機作業系統"課程的參考工具書，為學生提供一個 step by step 的引導去實作一個作業系統。這不是一個容易實現的目標，因為我本人現在並不自信有那個實力了解作業系統的所有細節。但是我想，立志百里行九十總好過于躑躅不前。

《自己動手寫作業系統》一書開了個好頭，所以在前面部分，我將主要討論使用open source 軟體實作《自》的 demo 。如果您有《自》這本書，參考閱讀效果會更好，不過我將盡我所能在本書中給出清楚的講解，盡量使您免於去參考《自》一書。

出於開放性和易編輯性考慮，本書採用 LaTeX 排版，在成書前期由於專注於版面，source code 比較雜亂，可讀性不强，暫不開放本書 TeX source code 下載。但您可以通過 SVN check out 所有本書相關的soure code 和圖片，具體方法請參見電子書主頁。

如果您在閱讀過程中有什麼問題，發現書中的錯誤，或者好的建議，歡迎您使用我留下的聯系方式與我聯系，本人將非常感謝。

楊文博
個人主頁：http://solrex.cn
個人博客：http://blog.solrex.cn
2008 年 1 月 9 日

更新歷史

Rev. 1 確定書本排版樣式，添加第一章，第二章。

Rev. 2 添加第三章保護模式。

目錄

插圖

序言

這裡應該是各個章節的摘要和版式簡介，不過因為寫摘要向來是件讓人心煩的事情，所以我準備把它放在最後寫 ☺。

CHAPTER 1

計算機啓動

每一個計算機軟體都是由一系列的可執行檔案組成的，可執行檔案的內容是可以被機器識別的二進制指令和數據。一般可執行檔案的執行是在作業系統的管理下載入記憶體並執行的，由作業系統給它分配資源和處理器時間，並確定它的執行方式。作業系統也是由可執行檔案組成的，但是作業系統的啓動方式和一般應用軟體是不同的，這也就是它叫做"作業系統"的原因 ☺。

沒有作業系統的機器，一般情況下被我們稱為"裸機"，意思就是只有硬體，什麼都幹不了。但是一個機器怎麼知道自己是不是裸機呢？它總要有方式去判斷機器上安裝沒有安裝作業系統吧。下面我們就簡單介紹一下計算機啓動的過程：

1.1 計算機啓動過程

計算機啓動過程一般是指計算機從點亮到載入作業系統的一個過程。對于 IBM 兼容機（個人電腦）來講，這個過程大致是這樣的：

1. **加電** 電源開關被按下時，機器就開始供電，主板的控制芯片組會向 CPU （Central Processing Unit，中央處理器）發出並保持一個 RESET （重置）信號，讓 CPU 恢復到初始狀態。當芯片組檢測到電源已經開始穩定供電時就會撤去 RESET 信號（鬆開台式機的重啓鍵是一樣的效果），這時 CPU 就從 `0xffff0` 處開始執行指令。這個位址在系統 BIOS （Basic Input/Output System，基本輸入輸出系統）的位址範圍內，大部分系統 BIOS 廠商放在這裡的都只是一道跳轉指令，跳到系統 BIOS 真正的啓動程式碼處。

2. **自檢** 系統 BIOS 的啓動代碼首先要做的事情就是進行 POST （Power-On Self Test，加電後自檢），POST 的主要任務是檢測系統中一些關鍵設備是否存在和能否正常工作，例如記憶體和顯示卡等。由于 POST 是最早進行的檢測過程，此時顯示卡還沒有初始化，如果系統 BIOS 在 POST 的過程中發現了一些致命錯誤，例如沒有找到記憶體或者記憶體有問題（此時只會檢查 640K 常規記憶體），那麼系統 BIOS 就會直接控制喇叭發聲來報告錯誤，聲音的長短和次數代表了錯誤的類型。

3. **初始化設備** 接下來系統 BIOS 將查找顯示卡的BIOS，存放顯示卡 BIOS 的 ROM 芯片的起始位址通常設在 `0xC0000` 處，系統 BIOS 在這個地方找到顯示卡 BIOS 之後就調用它的初始化代碼，由

顯示卡 BIOS 來初始化顯示卡，此時多數顯示卡都會在屏幕上顯示出一些初始化信息，介紹生產廠商、圖形芯片類型等內容。系統 BIOS 接著會查找其它設備的 BIOS 程序，找到之後同樣要調用這些 BIOS 內部的初始化代碼來初始化相關的設備。

4. **測試設備** 查找完所有其它設備的 BIOS 之後，系統 BIOS 將顯示出它自己的啓動畫面，其中包括有系統 BIOS 的類型、序列號和版本號等內容。接著系統 BIOS 將檢測和顯示 CPU 的類型和工作頻率，然後開始測試所有的 RAM （Random Access Memory，隨機訪問存儲器），並同時在屏幕上顯示記憶體測試的進度。記憶體測試通過之後，系統 BIOS 將開始檢測系統中安裝的一些標準硬體設備，包括硬碟、光驅、串口、並口、軟驅等，另外絶大多數較新版本的系統 BIOS 在這一過程中還要自動檢測和設置記憶體的定時參數、硬碟參數和訪問模式等。標準設備檢測完畢後，系統 BIOS 內部的支持即插即用的代碼將開始檢測和配置系統中安裝的即插即用設備，每找到一個設備之後，系統 BIOS 都會在屏幕上顯示出設備的名稱和型號等信息，同時為該設備分配中斷（INT）、DMA （Direct Memory Access，直接存儲器存取）通道和 I/O （Input/Output，輸入輸出）端口等資源。

5. **更新 ESCD** 所有硬體都檢測配置完畢後，多數系統 BIOS 會重新清屏並在屏幕上方顯示出一個表格，其中概略地列出了系統中安裝的各種標準硬體設備，以及它們使用的資源和一些相關工作參數。接下來系統 BIOS 將更新 ESCD （Extended System Configuration Data，擴展系統配置數據）。 ESCD 是系統 BIOS 用來與作業系統交換硬體配置信息的一種手段，這些數據被存放在 CMOS （Complementary Metal Oxide Semiconductor，互補金屬氧化物半導體）之中。

6. **啓動作業系統** ESCD 更新完畢後，系統 BIOS 的啓動代碼將進行它的最後一項工作，即根據用戶指定的啓動順序從軟碟、硬碟或光驅啓動作業系統。以 Windows XP 為例，系統 BIOS 將啓動盤（一般是主硬碟）的第一個磁區（Boot Sector，引導磁區）讀入到記憶體的 `0x7c00` 處，並檢查 `0x7dfe` 位址的記憶體，如果其內容是 `0xaa55`，跳轉到 `0x7c00` 處執行 MBR （Master Boot Record，主引導記錄），MBR 接著從分區表（Partition Table）中找到第一個活動分區（Active Partition，一般是 C 盤分區），然後按照類似方式讀取並執行這個活動分區的引導磁區（Partition Boot Sector），而引導磁區將負責讀取並執行 NTLDR （NT LoaDeR，Windows NT 的載入程序），然後主動權就移交給了 Windows 。

從以上介紹中我們可以看到，在第 6 步之前，電腦的啓動過程完全依仗于系統 BIOS ，這個程序一般是廠商寫就固化在主板上的。我們所需要做的，就是第 6 步之後的內容，即：

如何寫一個作業系統並把它載入到記憶體?

1.2 磁盤抽象物理結構

由于作業系統的啓動涉及到硬體位址寫入和磁盤文件尋找，為了更好理解記憶體位址和文件存儲的相關知識，我們先來了解一下磁盤的結構。

1.2.1 硬碟

Fig 1.1: 硬碟

圖 1.1 所示就是硬碟（如非特指，我們這裡的“硬碟”一般指代磁介質非固態硬碟）的外觀圖。其中左邊是硬碟盒拆開後盤片、磁頭和內部機械結構的透視圖，右邊是普通台式機硬碟的外觀圖。現在的硬碟容量較以前已經有大幅度增加，一般筆記本電腦硬碟容量已經在 120G 以上，台式機硬碟容量一般也達到了 160G 大小。一般情況下，硬碟都是由堅硬金屬材料（或者玻璃等）制成的塗以磁性介質的盤片構成的，一般有層疊的多片，每個盤片都有兩個面，兩面都可以記錄信息。

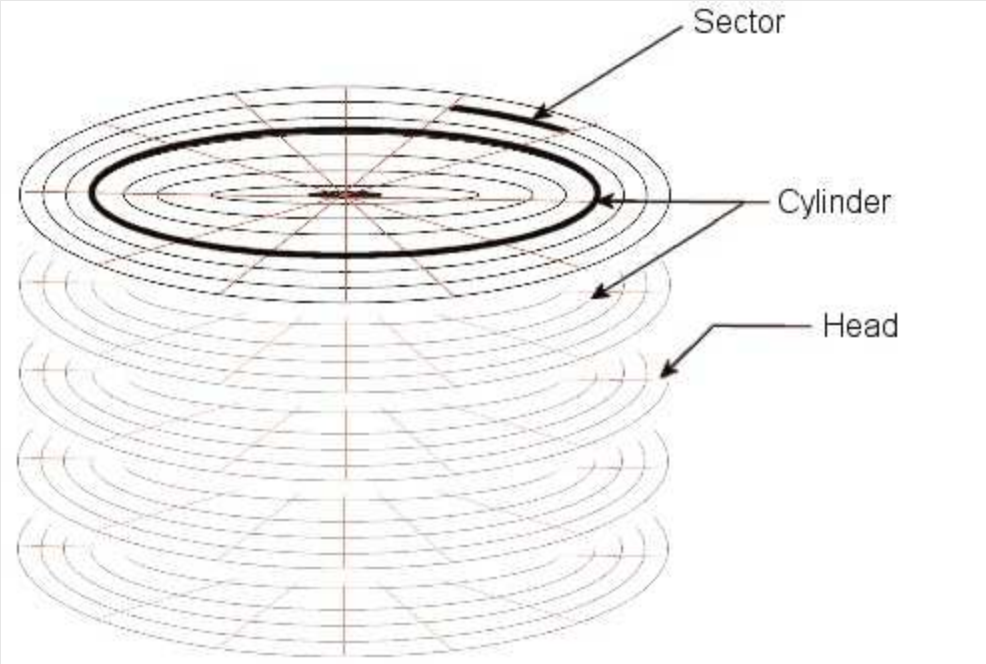


Fig 1.2: 硬碟的抽象物理結構

圖 1.2 為硬碟的抽象物理結構，需要注意的是這並不是硬碟真正的物理構造，所以這裡我們稱其為“抽象”物理結構。因此我們下面討論的也不是真正的硬碟技術實現，僅僅就硬碟（以及軟碟等類似磁介質存儲器）存儲結構以程序員易于理解的角度進行簡單的介紹。

如圖 1.2 所示，硬碟是由很多盤片組成的，那些上下有分割的圓盤就表示一個個盤片。每個盤片被分成許多扇形的區域，每個區域叫一個磁區，通常每個磁區存儲 512 byte（ FAT 文件格式），盤片表面上以盤片中心為圓心，不同半徑的同心圓稱為磁道。硬碟中，不同盤片相同半徑的磁道所組成的圓柱稱為柱面。磁道與柱面都是表示不同半徑的圓，在許多場合，磁道和柱面可以互換使用。每個磁盤有兩

個面，每個面都有一個磁頭，習慣用磁頭號來區分。磁區，磁道（或柱面）和磁頭數構成了硬碟結構的基本參數，使用這些參數可以得到硬碟的容量，其計算公式為：

存儲容量 = 磁頭數 × 磁道（柱面）數 × 每磁道磁區數 × 每磁區byte數

要點：

- 硬碟有數個盤片，每盤片兩個面，每面一個磁頭。
- 盤片被劃分為多個扇形區域即磁區。
- 同一盤片不同半徑的同心圓為磁道。
- 不同盤片相同半徑構成的圓柱面即柱面。
- 公式：存儲容量 = 磁頭數 × 磁道（柱面）數 × 每道磁區數 × 每磁區byte數。
- 信息記錄可表示為：××磁道（柱面），××磁頭，××磁區。

1.2.2 軟碟

由于我們在本書中主要使用軟碟作為系統啓動盤，所以下面對應于硬碟介紹一下軟碟的相關知識。

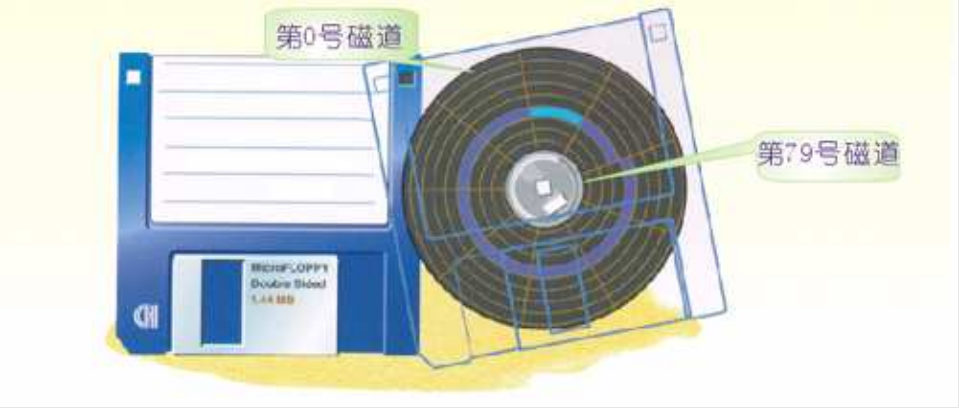


Fig 1.3: 軟碟

現在通常能看到的軟碟主要是 3.5 英寸軟碟，3.5 英寸指的是其內部磁介質盤片的直徑。從存儲結構上來講，軟碟與硬碟的主要不同就是軟碟只有一個盤片且其存儲密度較低。

由于軟碟只有一個盤片，兩個面，所以 3.5 英寸軟碟的容量可以根據上一小節的公式算出：

2(磁頭) × 80(磁道) × 18(磁區) × 512 bytes(磁區的大小) = 2880 x 512 bytes = 1440 KB = 1.44MB

在這裡需要引起我們特別注意的就是第 0 號磁頭（面），第 0 號磁道的第 0 號磁區，這裡是一切的開始。

1.2.3 啓動磁區

軟碟是沒有所謂的 MBR 的，因為軟碟容量較小，沒有所謂的分區，一張軟碟就顯示為一個邏輯磁盤。當我們使用軟碟啓動電腦的時候，系統從軟碟中首先讀取的就是第一個磁區，即前面所說的第 0 面，第 0 號磁道的第 0 號磁區，如果這個磁區的最後兩個byte是 0xaa55 ，這裡就簡單叫做啓動磁區（Boot Sector）。所以我們首先要做的就是：在啓動磁區的開始填入需要被執行的機器指令；在啓動磁區的最後兩個byte中填入 0xaa55，這樣這張軟碟就成為了一張可啓動盤。

> 啓動磁區最後兩個byte的內容為 0xaa55 ，這種說法是正確的—當且僅當表 2.1 中的 BPB_BytesPerSec（每磁區byte數）的值為 512 。如果 BPB_BytesPerSec 的值大于 512 ， 0xaa55 的位置不會變化，但已經不是啓動磁區最後兩個byte了。

整個過程如圖 1.4 所示：

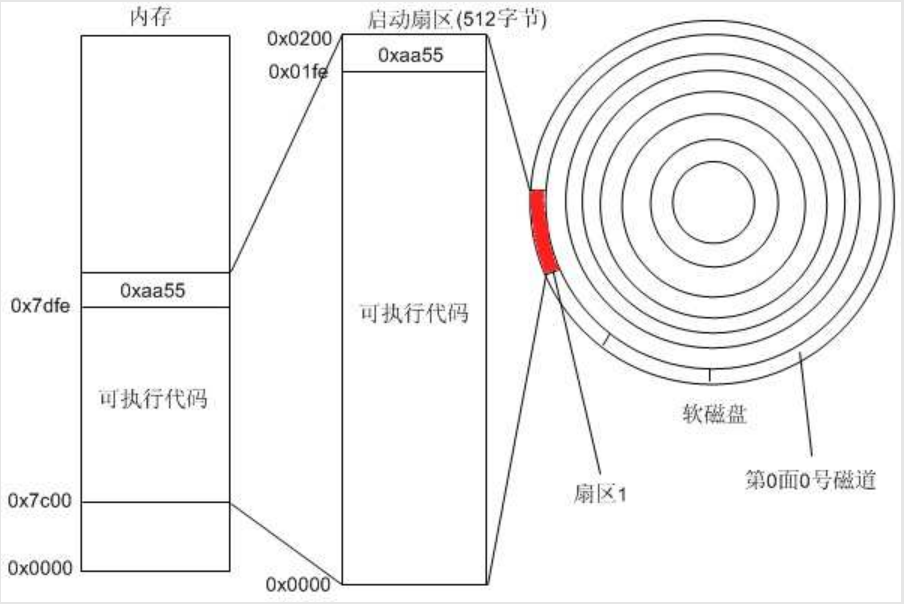


Fig 1.4: 啓動磁區載入示意圖

需要注意的是，軟碟的啓動磁區並不像一個文件一樣，可以直接讀取，寫入啓動磁區的過程是需要一些技巧的，下面我們將討論如何去實現。

1.3 使用虛擬機

在實現一個簡單的作業系統時，我們是不可能拿一台真正的機器做實驗的，一是很少有人有這個條件，還有就是那樣做比較麻煩。所以我們使用虛擬機來模擬一台真實的電腦，這樣我們就能直接用虛擬機載入軟碟鏡像來啓動了，而制作軟碟鏡像顯然要比寫一張真實的軟碟簡單許多。

在 Linux 下有很多虛擬機軟體，我們選擇 VirtualBox 和 Bochs 作為主要的實現平台，我們用 VirtualBox 做 demo ，而 Bochs 主要用作調試。下面給出一些虛擬機設置的指導，其實用哪種虛擬機都沒有關系，我們需要的只是虛擬機支持載入軟碟鏡像並能從軟碟啓動。

1.3.1 VirtualBox

VirtualBox 是遵從 GPL 協議的開源軟體，它的官方網站是 http://www.virtualbox.org 。VirtualBox 的官方網站上提供針對很多 Linux 系統平台的二進制安裝包，比如針對 Red Hat 系列（Fedora, RHEL）各種版本的 RPM 安裝包，針對 Debian 系（Debian, Ubuntu）各種版本的 DEB 安裝包，其中 Ubuntu Linux 可以更方便地從 Ubuntu 軟體倉庫中直接下載安裝：`sudo apt-get install virtualbox` 。

安裝好 VirtualBox 後，需要使用 `sudo adduser ‘whoami‘ vboxusers`（某些系統中的添加用戶命令可能是 useradd ）將自己添加到 VirtualBox 的用戶組 vboxusers 中去；當然，也可以使用 GNOME 或者 KDE 的圖形界面用戶和組的管理工具來添加組用戶，也可以直接編輯 /etc/group 文件，將自己的用戶名添加到 vboxusers 對應行的最後，例如 `vboxusers:x:501:solrex` ，部分 Linux 可能需要注銷後重新登錄當前用戶。

我們下面使用 CentOS 上安裝的 VirtualBox 演示如何用它建立一個虛擬機。

第一次啓動 VirtualBox ，會首先彈出一個 VirtualBox 個人使用協議 PUEL 的對話框（某些版本的 Linux 可能不會彈出）：

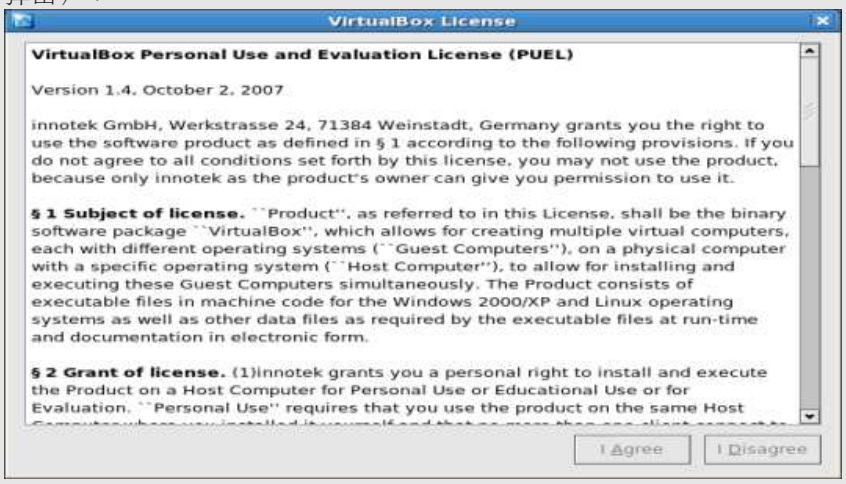


Fig 1.5: VirtualBox 個人使用協議

閱讀完協議後，將下拉條拉到最低可以激活最下方的同意按鈕，點擊之：

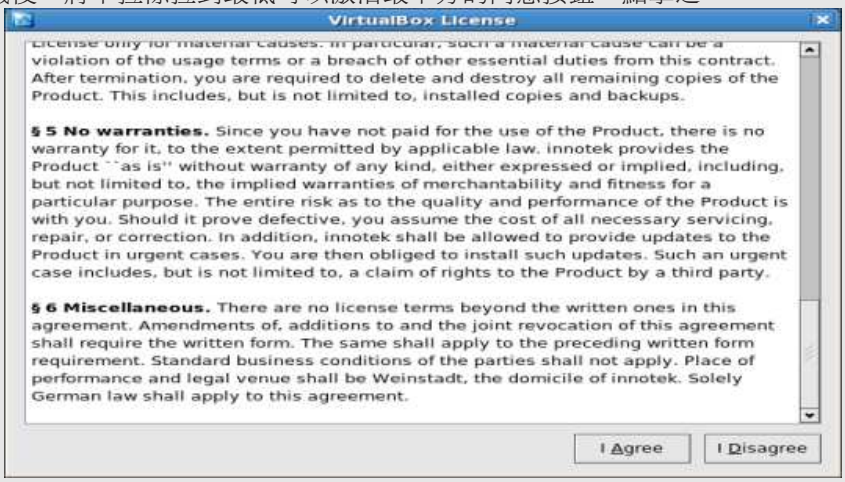


Fig 1.6: 同意 VirtualBox 個人使用協議

彈出的 VirtualBox 用戶注冊對話框，可忽視關閉之：

Fig 1.7: VirtualBox 用戶注冊對話框

接下來我們就見到了 VirtualBox 主界面：

Fig 1.8: VirtualBox 主界面

點擊 New 按鈕新建一個虛擬機：

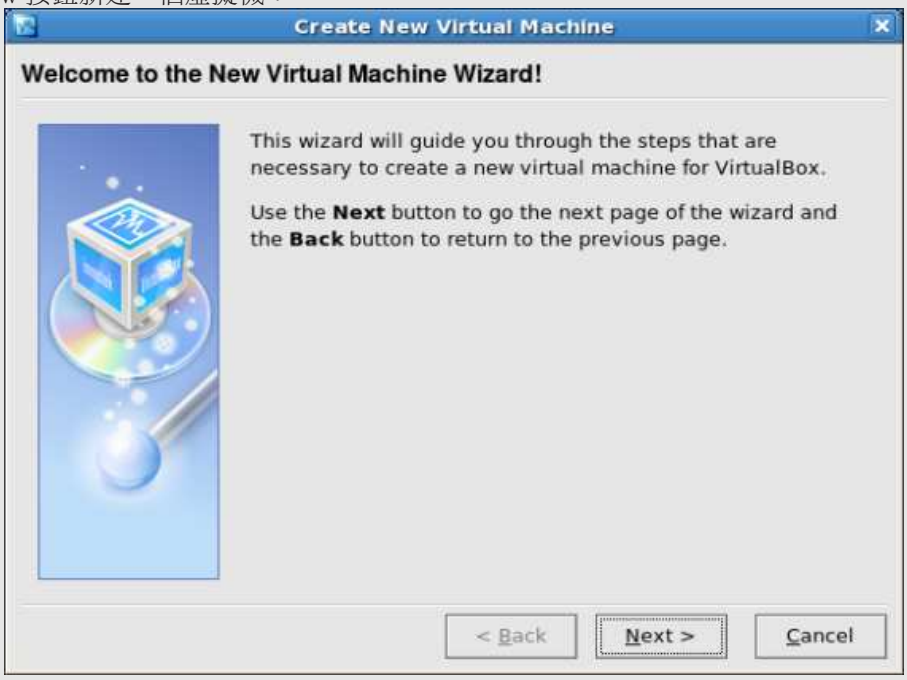


Fig 1.9: 新建一個虛擬機

我們使用 solrex 作為虛擬機的名字，系統類型未知：

Fig 1.10: 設置虛擬機名字和作業系統類型

設置虛擬機的記憶體容量，這裡隨便設了 32M：

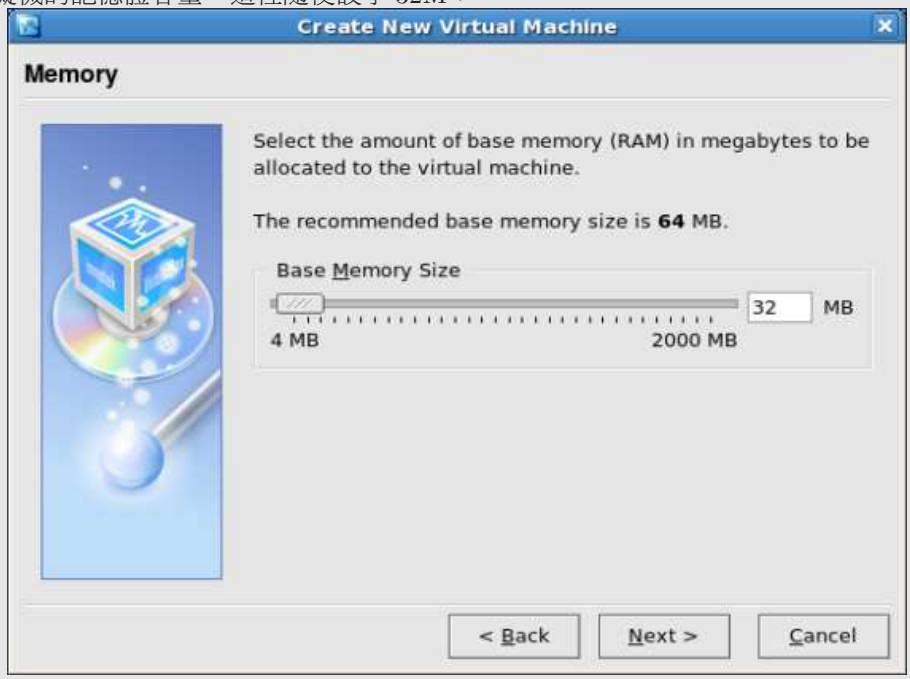


Fig 1.11: 設置虛擬機記憶體容量

設置虛擬機硬碟鏡像：

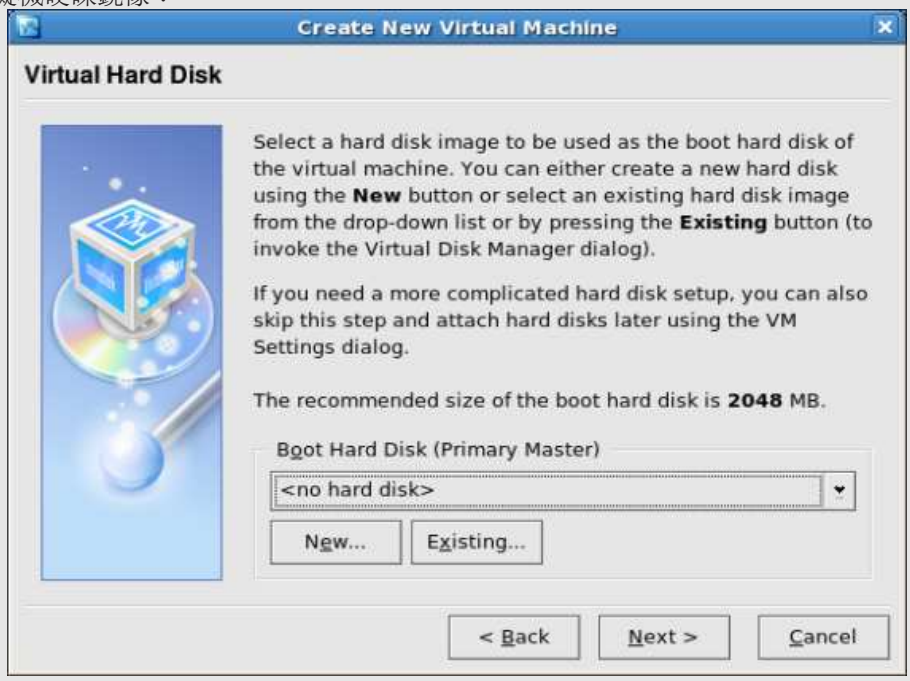


Fig 1.12: 設置虛擬機硬碟鏡像

如果沒有硬碟鏡像，需點“New”新建一塊硬碟鏡像：

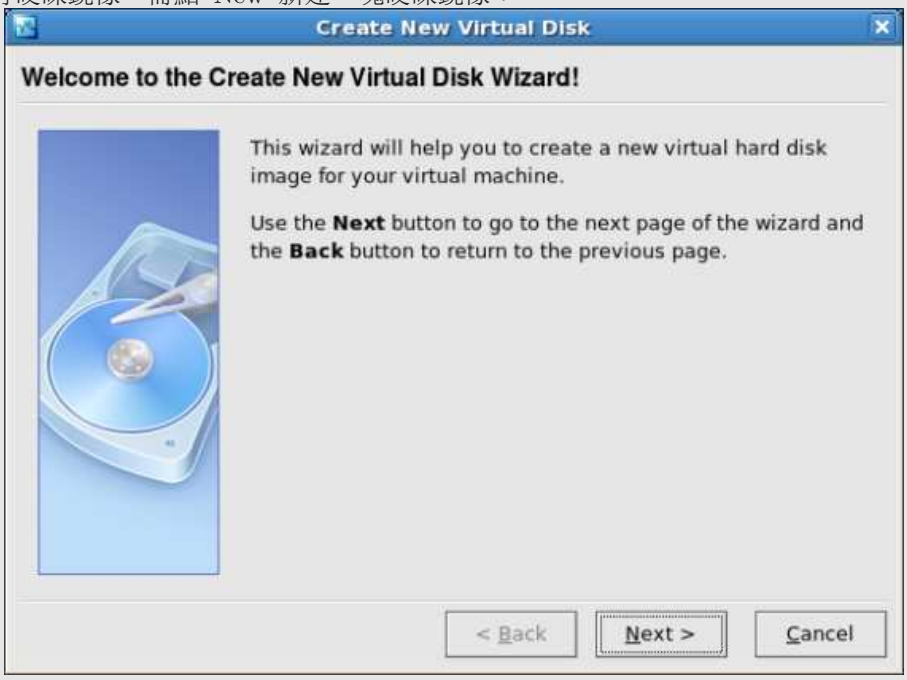


Fig 1.13: 新建一塊虛擬硬碟

點“Next”，設置虛擬硬碟鏡像為可自動擴充大小：

Fig 1.14: 設置虛擬硬碟類型

這裡將虛擬硬碟鏡像的名字設置為“solrex”，並將容量設置為“32M”：

Fig 1.15: 設置虛擬硬碟鏡像名字和容量

最後查看新建的虛擬硬碟信息，點擊 Finish 確認新建硬碟鏡像：

Fig 1.16: 虛擬硬碟信息

令虛擬機使用已建立的虛擬硬碟 solrex.vdi ：

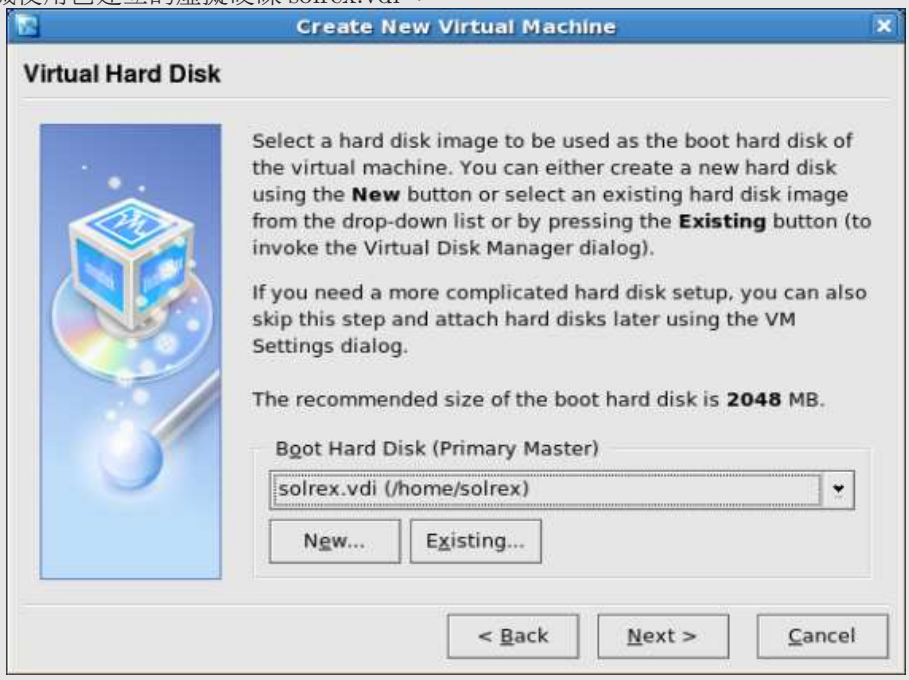


Fig 1.17: 使用新建的虛擬硬碟

最後查看新建的虛擬機信息，點擊 Finish 確認新建虛擬機：

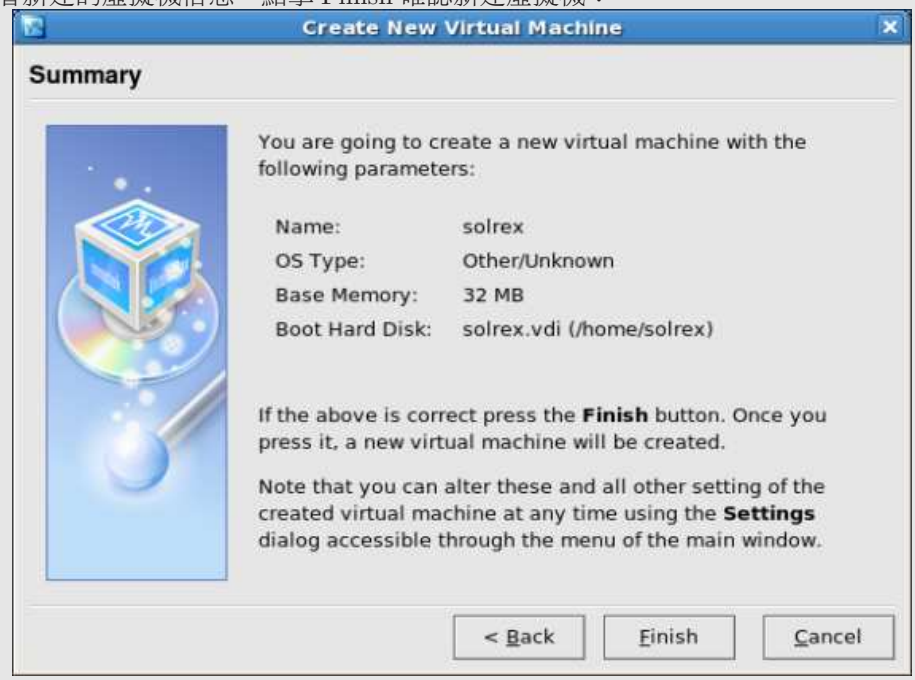


Fig 1.18: 虛擬機信息

回到 VirtualBox 主界面，左側列表中有新建立的虛擬機 solrex ：

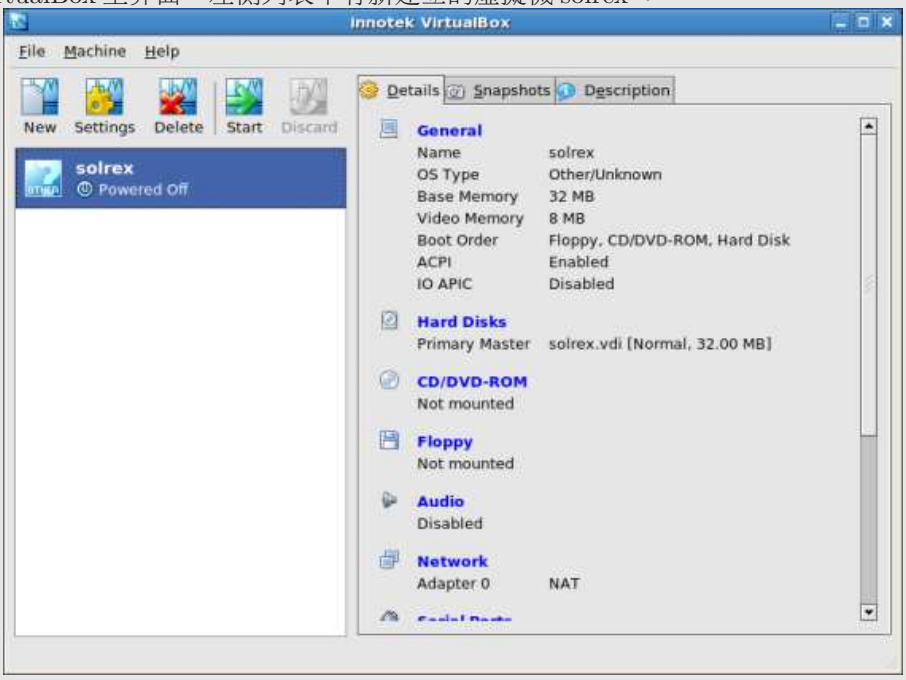


Fig 1.19: 回到 VirtualBox 主界面

1.3.2 Bochs

1.4 使用軟碟鏡像

1.4.1 制作軟碟鏡像

前面我們說過，軟碟的結構比較簡單，所以我們選擇使用軟碟鏡像來啓動虛擬計算機。在 Linux 下制作一個軟碟鏡像很簡單，只需要使用：

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

命令就可以在當前目錄下生成一個名為 `emptydisk.img` 的空白軟碟鏡像，下面我們使用這個空白軟碟鏡像來啓動虛擬機。

> **dd** ：轉換和拷貝文件的工具。dd 可以設置很多拷貝時候的參數，在本例中 if=FILE 選項代表從 FILE 中讀取內容； of=FILE 選項代表將導出輸出到 FILE ； bs=BYTES 代表每次讀取和輸出 BYTES 個byte； count=BLOCKS 代表從輸入文件中共讀取 BLOCKS 個輸入塊。
>
> 而這裡的 /dev/zero 則是一個 Linux 的特殊文件，讀取這個文件可以得到持續的 0 。那麼上面命令的意思就是以每塊 512 byte共 2880 塊全空的字符填入文件 emptydisk.img 中。我們注意到前面提及的軟碟容量計算公式：
>
> 2(磁頭) × 80(磁道) × 18(磁區) × 512 bytes(磁區的大小) = 2880 x 512 bytes = 1440 KB = 1.44MB
>
> 可以發現我們用上述命令得到的就是一張全空的未格式化的軟碟鏡像。

1.4.2 用軟碟鏡像啓動虛擬機

在虛擬機主界面選中虛擬機後點 Settings 按鈕，進入虛擬機的設置界面：

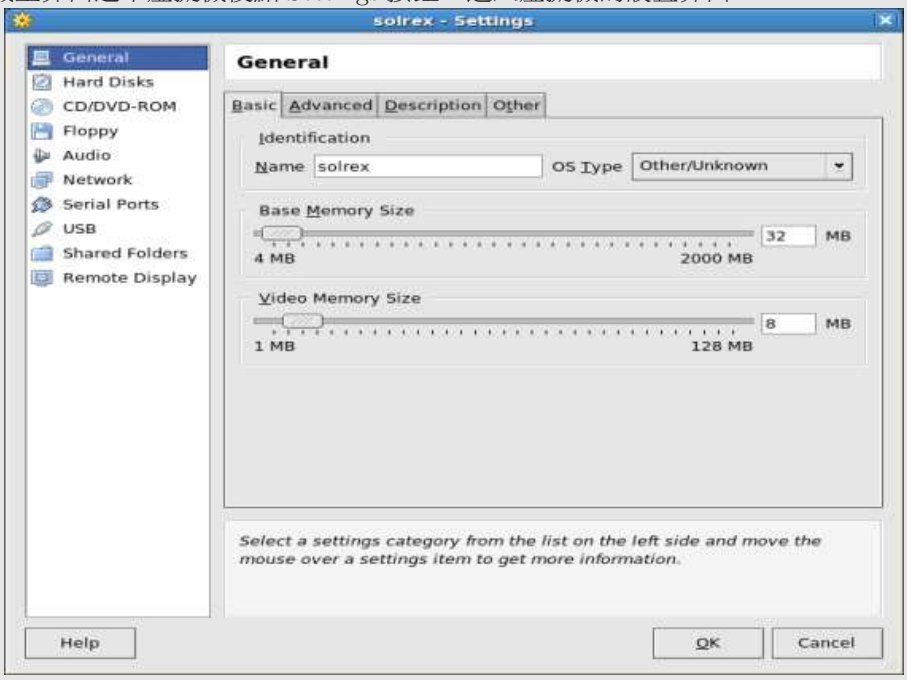


Fig 1.20: 虛擬機設置界面

在左側列表中選擇 Floppy 進入虛擬機軟碟設置界面：

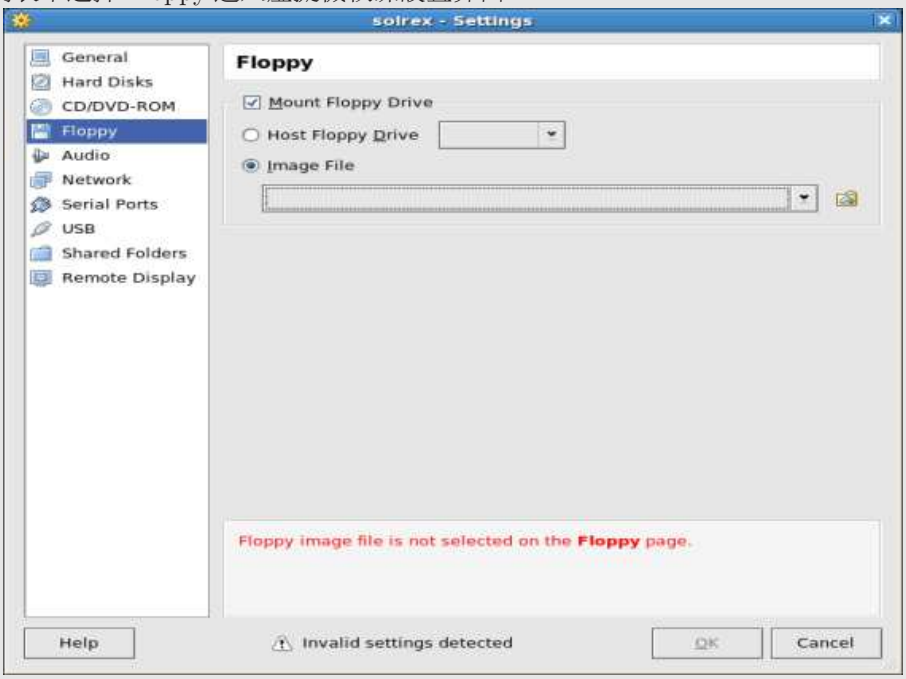


Fig 1.21: 虛擬機軟碟設置

點擊 Image File 最右側的文件夾標志，進入選擇軟碟鏡像界面：

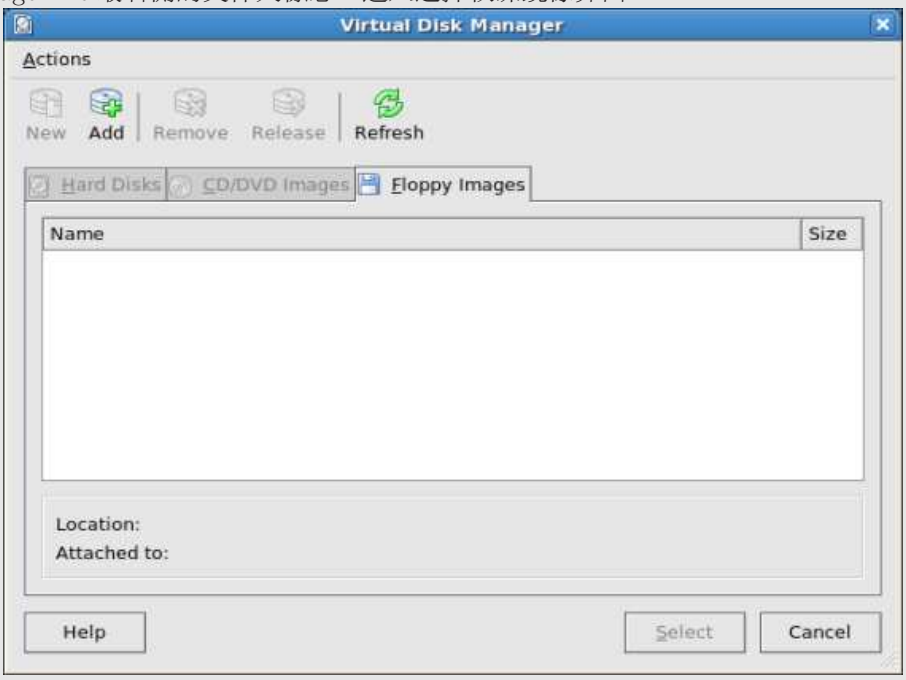


Fig 1.22: 選擇軟碟鏡像

點擊 Add 按鈕添加新的軟碟鏡像 emptydisk.img，並點擊 select 按鈕選中其作為啓動軟碟：

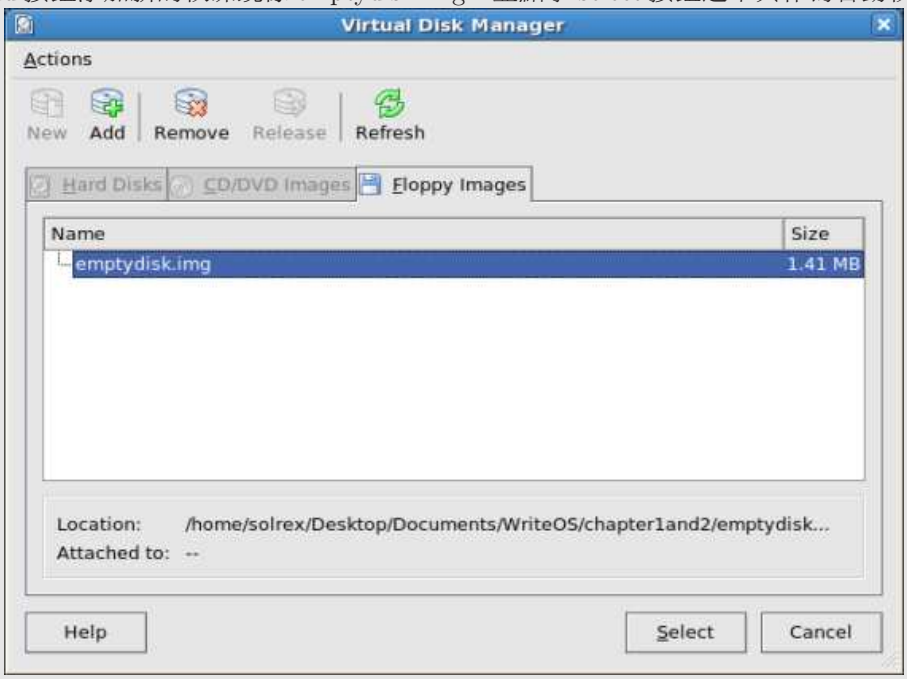


Fig 1.23: 選擇啓動軟碟鏡像

返回虛擬機軟碟設置界面後，點擊 OK 確認鏡像文件信息：

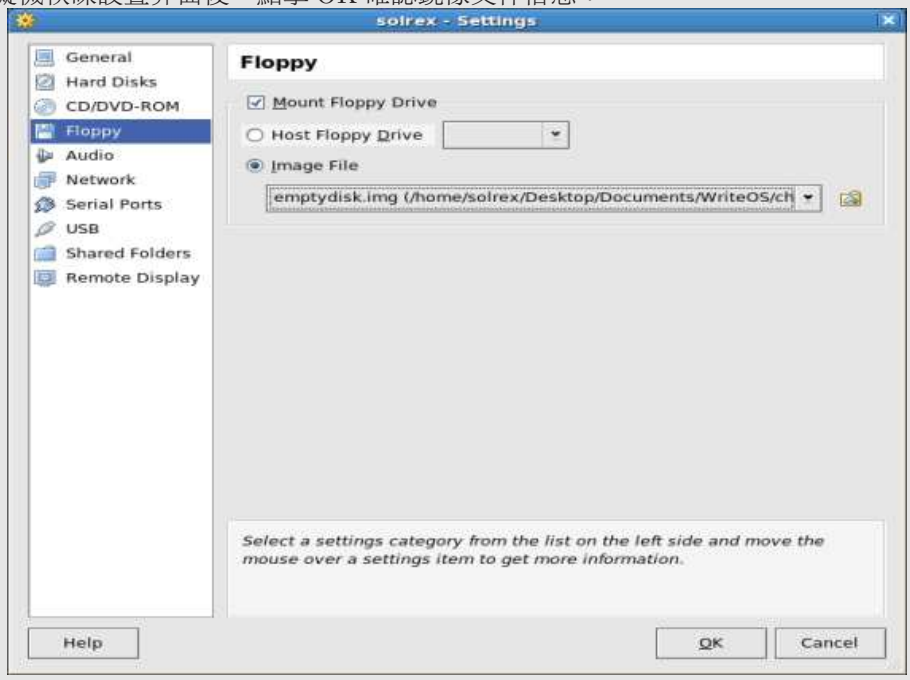


Fig 1.24: 確認啓動鏡像軟碟文件信息

返回虛擬機主界面，查看右側的虛擬機設置信息：

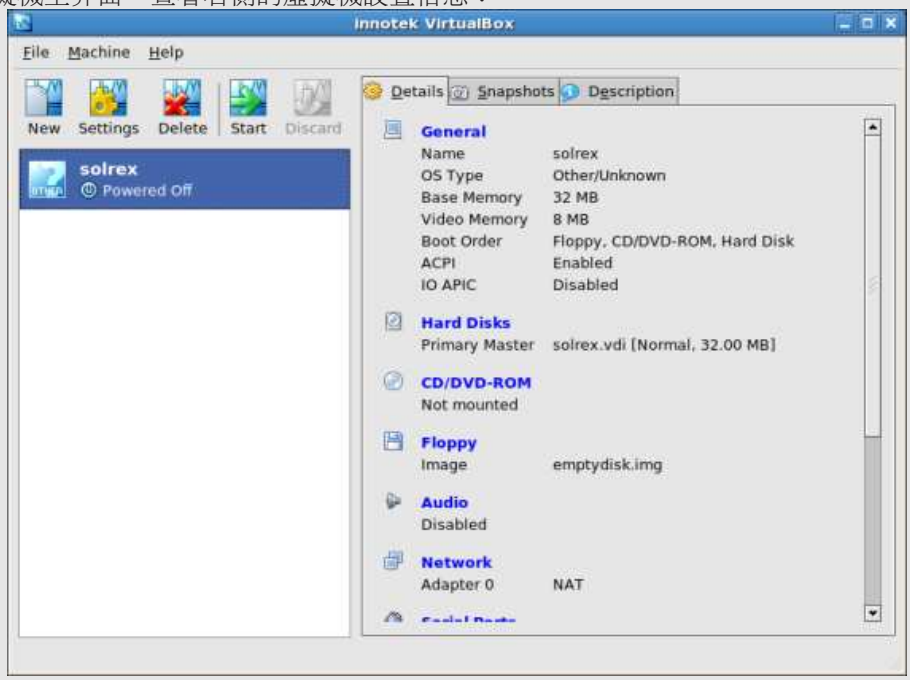


Fig 1.25: 查看虛擬機設置信息

選中虛擬機後，雙擊或點擊 Start 按鈕執行它，第一次執行可能給出如下信息：

Fig 1.26: 自動鍵盤捕獲警告信息

這個對話框的意思就是，當鼠標在虛擬機內部點擊時，鼠標和鍵盤的消息將被虛擬機自動捕獲，成為虛擬機的鍵盤和鼠標，可以敲擊鍵盤右側的 Ctrl 鍵解除捕獲。

顯示虛擬機的執行時內容：

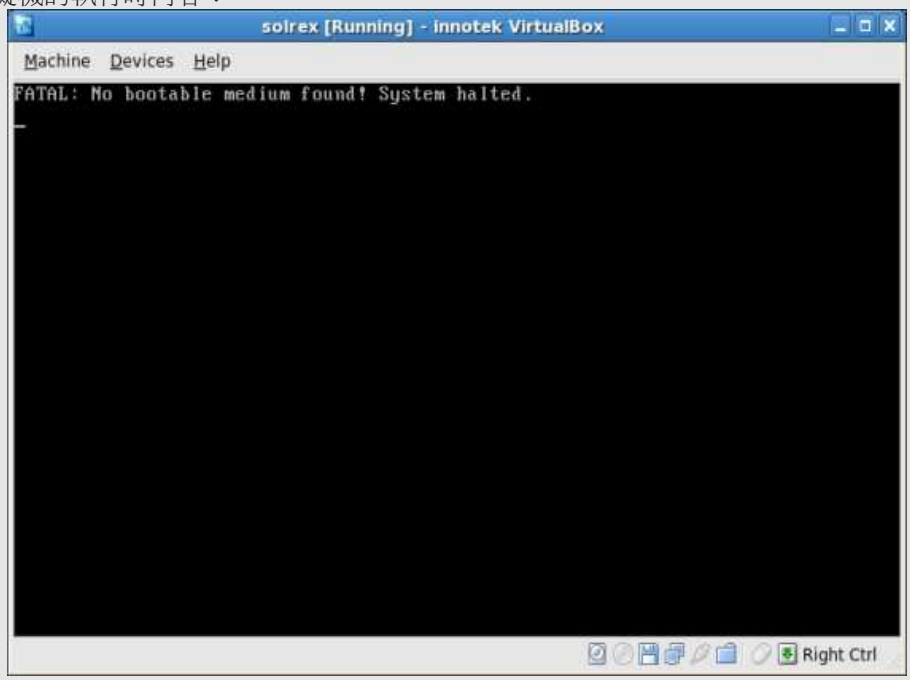


Fig 1.27: 虛擬機執行時

我們可以看到在圖 1.27 中，虛擬機載入空白軟碟啓動後提示消息為：“FATAL: No bootable medium found! System halted.”，換成中文是找不到可啓動媒體，系統停機。它的實際意思就是在前面第 1.1 節“計算機啓動過程”中提到的第 6 步中虛擬機遍歷了軟驅、光驅、硬碟後沒有找到可啓動的媒體，所以就只好停機。因為我們在啓動前已經在軟驅中載入了軟碟鏡像，所以提示信息就表明那個軟碟鏡像不具有啓動系統的功能，那麼如何才能創建一個可啓動的軟碟呢，我們將在第 2 章介紹。

CHAPTER 2

最小的「作業系統」

任何一個完善的作業系統都是從啓動磁區開始的, 這一章, 我們就關注如何寫一個啓動磁區, 以及如何將其寫入到軟碟片鏡像中。

先介紹一下需要使用的工具：

- 系統： 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!

本章節內容需要和 gcc, make 相關的 Linux C 語言編程以及 PC 匯編語言的基礎知識。

推薦預備閱讀：CS:APP（Computer Systems: A Programmer's Perspective, 深入理解計算機系統）第 3 章：Machine-Level Representation of Programs。

很多編程書籍給出的第一個例子往往是在終端裡輸出一個字符串"Hello world!", 那麼要寫作業系統的第一步給出的例子自然就是如何在屏幕上打印出一個字符串嘍。所以, 我們首先看《自己動手寫作業系統》一書中給出的第一個示例代碼, 在屏幕上打印"Hello OS world!"：

```
1     org    07c00h        ; 告訴編譯器程序加載到7c00處
2     mov    ax, cs
3     mov    ds, ax
4     mov    es, ax
5     call   DispStr       ; 呼叫顯示字符串例程
6     jmp    $             ; 無限循環
7 DispStr:
```

```
8     mov    ax, BootMessage
9     mov    bp, ax        ; ES:BP = 串位址
10    mov    cx, 16        ; CX = 串長度
11    mov    ax, 01301h    ; AH = 13,  AL = 01h
12    mov    bx, 000ch     ; 頁號為0(BH = 0) 黑底紅字(BL = 0Ch,高亮)
13    mov    dl, 0
14    int    10h           ; 10h 號中斷
15    ret
16 BootMessage:     db    "Hello, OS world!"
17 times 510-($-$$) db    0 ; 填充剩下的空間, 使生成的二進制代碼恰好為512 byte
18 dw    0xaa55             ; 結束標志
```

Fig 2.1: 《自》第一個實例代碼 boot.asm

2.1.1 Intel 語法轉化為 AT&T(GAS) 語法

上面 boot.asm 中代碼使用 Intel 風格的匯編語言寫成, 本也可以在 Linux 下使用同樣開源的 NASM 編譯, 但是鑑于很少有人在 Linux 下使用此匯編語法, 它在 Linux 平台上的擴展性和可調試性都不好（ GCC 不兼容）, 而且不是採用 Linux 平台上編譯習慣, 所以我把它改成了使用 GNU 工具鏈去編譯連接。這樣的話, 對以後使用 GNU 工具鏈編寫其它體系結構的 bootloader 也有幫助, 畢竟 NASM 沒有 GAS 用戶多（也許 ☺）。

上面的匯編源程序可以改寫成 AT&T 風格的匯編源代碼：

```
1 .code16               #使用16位模式匯編
2 .text                 #代碼段開始
3     mov    %cs,%ax
4     mov    %ax,%ds
5     mov    %ax,%es
6     call   DispStr     #呼叫顯示字符串例程
7     jmp    .           #無限循環
8 DispStr:
9     mov    $BootMessage, %ax
10    mov    %ax,%bp         #ES:BP = 串位址
11    mov    $16,%cx         #CX = 串長度
12    mov    $0x1301,%ax     #AH = 13,  AL = 01h
13    mov    $0x00c,%bx      #頁號為0(BH = 0) 黑底紅字(BL = 0Ch,高亮)
14    mov    $0,%dl
15    int    $0x10           #10h 號中斷
16    ret
17 BootMessage:.ascii "Hello, OS world!"
18 .org 510             #填充到~510~ byte 處
19 .word 0xaa55         #結束標志
```

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

2.1.2 用 linker script 控制位址空間

但有一個問題,我們可以使用 `nasm boot.asm -o boot.bin` 命令將 boot.asm 直接編譯成二進制檔案, GAS 不能。不過 GAS 的不能恰好給開發者一個機會去分步地實現從匯編源代碼到二進制檔案這個

過程, 使編譯更為靈活。下面請看 GAS 是如何通過 linker script 控制程序位址空間的：

```
SECTIONS
{
  . = 0x7c00;
  .text :
  {
    _ftext = .;     /* Program will be loaded to 0x7c00. */
  } = 0
}
```

Fig 2.3: boot.S 的 linker script (chapter2/1/solrex_x86.ld)

> **linker script** ： GNU 連接器 ld 的每一個連接過程都由 linker script 控制。 linker script 主要用于, 怎樣把輸入檔案內的 section 放入輸出檔案內, 並且控制輸出檔案內各部分在程序位址空間內的布局。連接器有個默認的內置 linker script , 可以用命令 ld –verbose 查看。選項 -T 選項可以指定自己的 linker script , 它將代替默認的 linker script 。

這個 linker script 的功能就是, 在連接的時候, 將程序入口設置為記憶體 0x7c00 的位置（ BOIS 將跳轉到這裡繼續啓動過程）, 相當于 boot.asm 中的 org 07c00h 一句。有人可能覺得麻煩, 還需要用一個腳本控制加載位址, 但是《自己動手寫作業系統》就給了一個很好的反例：《自》第 1.5 節代碼 1-2 ,作者切換調試和運行模式時候需要對代碼進行注釋。

```
;%define _BOOT_DEBUG_   ; 做 Boot Sector 時一定將此行注釋掉!將此行打開後用
                        ; nasm Boot.asm -o Boot.com 做成一個.COM檔案易于調試

%ifdef _BOOT_DEBUG_
    org  0100h      ; 調試狀態, 做成 .COM 檔案, 可調試
%else
    org  07c00h     ; Boot 狀態, Bios 將把 Boot Sector 加載到 0:7C00 處並開始執行
%endif

    mov    ax, cs
    mov    ds, ax
    mov    es, ax
    call   DispStr      ; 呼叫顯示字符串例程
    jmp    $            ; 無限循環
DispStr:
    mov    ax, BootMessage
    mov    bp, ax       ; ES:BP = 串位址
    mov    cx, 16       ; CX = 串長度
    mov    ax, 01301h   ; AH = 13,  AL = 01h
    mov    bx, 000ch    ; 頁號為0(BH = 0) 黑底紅字(BL = 0Ch,高亮)
    mov    dl, 0
    int    10h          ; 10h 號中斷
    ret
BootMessage:     db    "Hello, OS world!"
times 510-($-$$) db    0 ; 填充剩下的空間, 使生成的二進制代碼恰好為512 byte
dw    0xaa55             ; 結束標志
```

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

而如果換成使用腳本控制程序位址空間, 只需要編譯時候呼叫不同腳本進行連接, 就能解決這個問題。這在嵌入式編程中是很常見的處理方式, 即使用不同的 linker script 一次 make 從一個源程序檔案生成分別運行在開發板上和軟件模擬器上的兩個二進制檔案。

2.1.3 用 Makefile 編譯連接

下面的這個 Makefile 檔案, 就是我們用來自動編譯 boot.S 匯編源代碼的腳本檔案：

```
1 CC=gcc
2 LD=ld
3 LDFILE=solrex_x86.ld    #使用上面提供的~linker script~solrex_x86.ld
4 OBJCOPY=objcopy
5
6 all: boot.img
7
8 # Step 1: gcc 呼叫 as 將 boot.S 編譯成目標檔案 boot.o
9 boot.o: boot.S
10        $(CC) -c boot.S
11
12 # Step 2: ld 呼叫 linker script solrex_x86.ld 將 boot.o 連接成可執行檔案 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 中沒有用的 section(.pdr,.comment,.note),
17 #         strip 掉所有符號信息, 輸出為二進制檔案 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: 生成可啓動軟碟片鏡像。
22 boot.img: boot.bin
23        @dd if=boot.bin of=boot.img bs=512 count=1             #用 boot.bin 生成鏡像檔案第一個磁區
24        # 在 bin 生成的鏡像檔案後補上空白, 最後成為合適大小的軟碟片鏡像
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)

我們將上面內容保存成 Makefile , 與圖 2.2 所示 boog.S 和圖 2.3 所示 solrex_x86.ld 放在同一個目錄下, 然後在此目錄下使用下面命令編譯：

```
$ 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.ld
boot.bin  boot.img  boot.S  solrex.img
```

可以看到, 我們只需執行一條命令 make 就可以編譯、連接和直接生成可啓動的軟碟片鏡像檔案, 其間對源檔案的每一步處理也都一清二楚。不用任何商業軟件, 也不用自己寫任何轉換工具, 比如《自己動手寫作業系統》文中提到的 HD-COPY 和 Floopy Writer 都沒有使用到。

在這裡需要特別注意的是圖 2.5 中的 Step 4, 其實對這一步的解釋應該結合圖 1.4 來查看。我們用 boot.S 編譯生成的 boot.bin 其實只是圖 1.4 中所指的軟碟片的啓動磁區, 例如 boot.S 最後一行：

```
.word 0xaa55        #結束標志
```

生成就是啓動磁區最後的 0xaa55 那兩個byte，而 boot.bin 的大小是 512 byte，正好是啓動磁區的大小。那麼 Step 4 的功能就是把 boot.bin 放入到一個空白軟碟片的啓動磁區, 這樣呢當虛擬機啓動時能識別出這是一張可啓動軟碟片, 並且執行我們在啓動磁區中寫入的打印代碼。

為了驗證軟碟片鏡像檔案的正確性也可以先用

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

將 boot.img 前 512 個byte 打印出來, 可以看到 boot.img dump 的內容和《自》一書附送光盤中的 TINIX.IMG dump 的內容完全相同。這裡我們也顯然用不到 EditPlus 或者 UltraEdit , 即使需要修改二進制碼, 也可以使用 hexedit, ghex2, khexedit 等工具對二進制檔案進行修改。

下圖為使用命令行工具 hexedit 打開 boot.img 的窗口截圖, 從圖中我們可以看到, 左列是該行開頭與檔案頭對應的偏移位址, 中間一列是檔案的二進制內容, 最右列是檔案內容的 ASCII 顯示內容, 可以看到, 此界面與 UltraEdit 的十六進制編輯界面沒有本質不同。

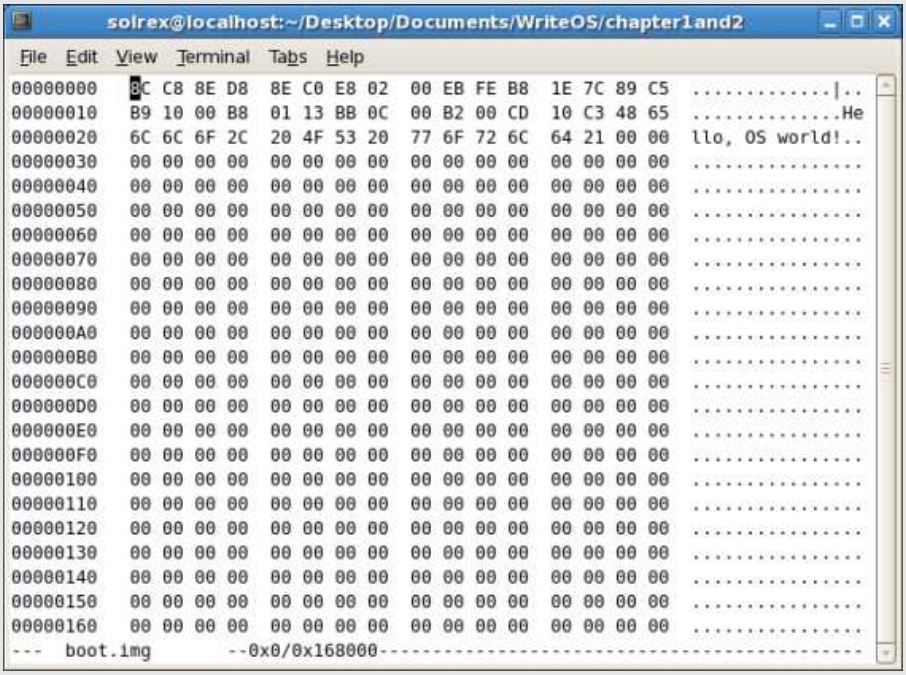


Fig 2.6: 使用 hexedit 打開 boot.img

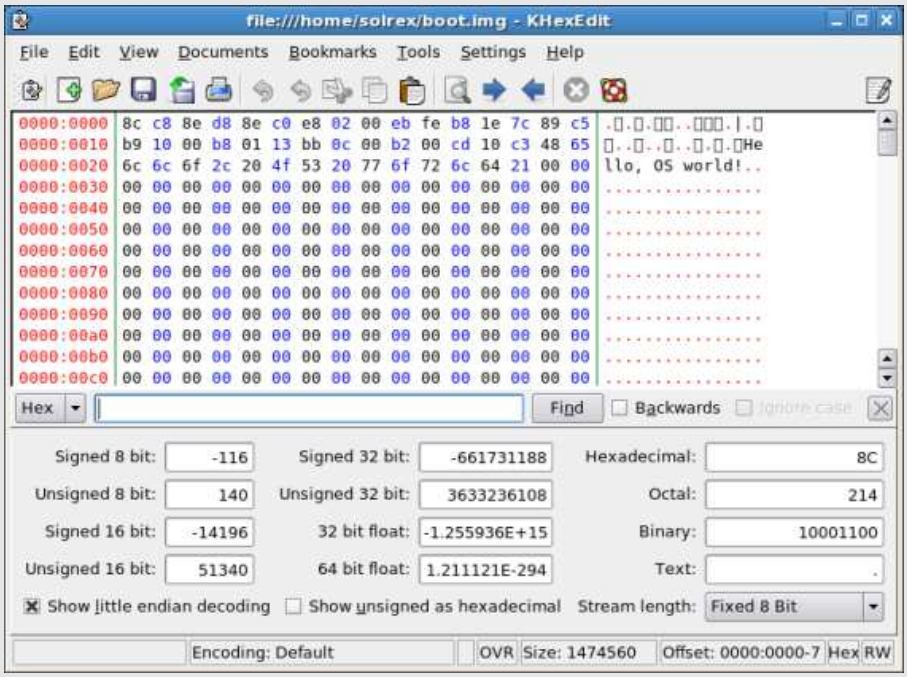


Fig 2.7: 使用 kde 圖形界面工具 khexedit 打開 boot.img

2.1.4 用虛擬機加載執行 boot.img

當我們生成 boot.img 之後, 仿照第 1.4.2 節中加載軟碟片鏡像的方法, 用虛擬機加載 boot.img:

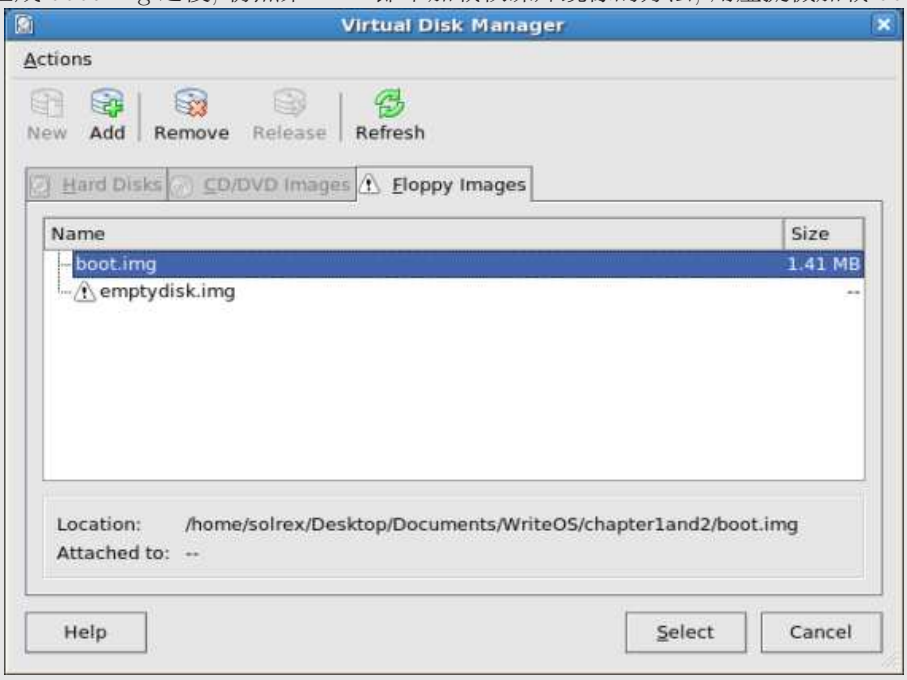


Fig 2.8: 選擇啓動軟碟片鏡像 boot.img

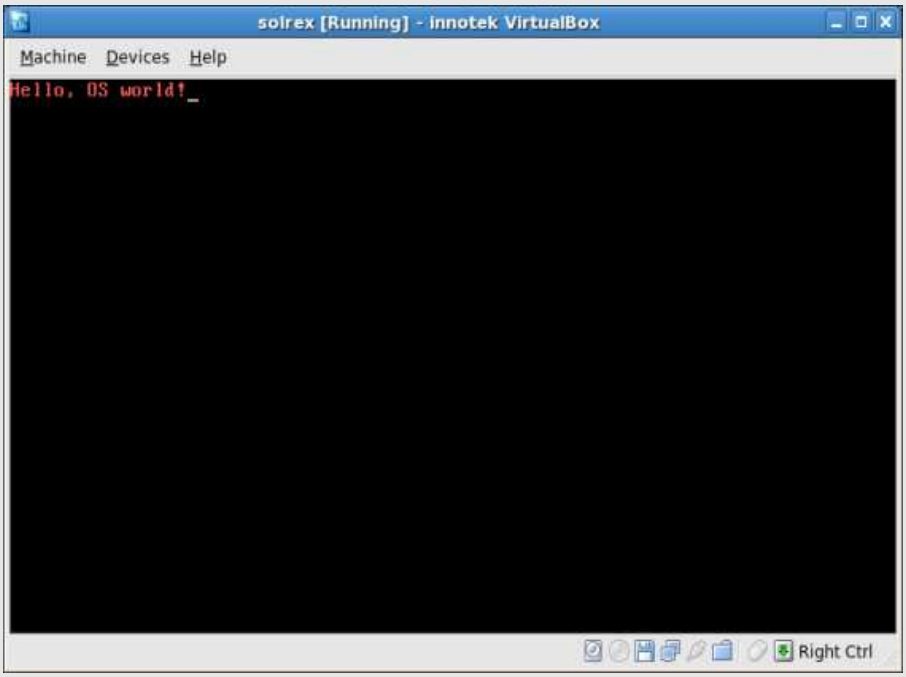


Fig 2.9: 虛擬機啓動後打印出紅色的“Hello OS world!”

我們看到虛擬機如我們所料的打印出了紅色的“Hello OS world!”字樣, 這説明我們以上的程序和編譯過程是正確的。

2.2 FAT 檔案系統

我們在上一節中介紹的内容, 僅僅是寫一個啓動磁區並將其放入軟碟片鏡像的合適位置。由于啓動磁區 512 byte 的大小限制, 我們僅僅能寫入像打印一個字符串這樣的非常簡單的程序, 那麼如何突破 512 byte 的限制呢？很顯然的答案是我們要利用其它的磁區, 將程序保存在其它磁區, 運行前將其加載到記憶體後再跳轉過去執行。那麼又一個問題産生了：程序在軟碟片上應該怎樣存儲呢？

可能最直接最容易理解的存儲方式就是順序存儲, 即將一個大程序從啓動磁區開始按順序存儲在相鄰的磁區, 可能這樣需要的工作量最小, 在啓動時作業系統僅僅需要序列地將可執行代碼拷貝到記憶體中來繼續運行。可是經過簡單的思考我們就可以發現這樣做有幾個缺陷：1. 軟碟片中僅能存儲作業系統程序, 無法存儲其它内容；2. 我們必須使用二進制拷貝方式來制作軟碟片鏡像, 修改系統麻煩。

那麼怎麼避免這兩個缺點呢？引入檔案系統可以讓我們在一張軟碟片上存儲不同的檔案, 並提供檔案管理功能, 可以讓我們避免上述的兩個缺點。在使用某種檔案系統對軟碟片格式化之後, 我們可以像普通軟碟片一樣使用它來存儲多個檔案和目錄, 為了使用軟碟片上的檔案, 我們給啓動磁區的代碼加上尋找檔案和加載執行檔案功能, 讓啓動磁區將系統控制權轉移給軟碟片上的某個檔案, 這樣突破啓動磁區 512 byte 大小的限制。

2.2.1 FAT12 檔案系統

FAT(File Allocation Table) 檔案系統規格在 20 世紀 70 年代末和 80 年代初形成, 是微軟的 MS-DOS 作業系統使用的檔案系統格式。它的初衷是為小于 500K 容量的軟碟片制定的簡單檔案系統, 但在將近

三十年的發展過程中, 它已經被一次次修改加强以支持更大的存儲媒體。在目前主要有三種 FAT 檔案系統類型：FAT12, FAT16 和 FAT32。這幾種類型最基本的區別就像它們的名字字面區別一樣, 主要在于大小, 即盤上 FAT 表的記錄項所佔的bit 數。FAT12 的記錄項佔 12 bit , FAT16 佔 16 bit , FAT32 佔 32 bit 。

由于 FAT12 最為簡單和易實施, 這裡我們僅簡單介紹 FAT12 檔案系統, 想要了解更多 FAT 檔案系統知識的話, 可以到 http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx 下載微軟發布的 FAT 檔案系統官方文檔。

FAT12 檔案系統和其它檔案系統一樣, 都將磁盤劃分為層次進行管理。從邏輯上劃分, 一般將磁盤劃分為盤符, 目錄和檔案；從抽象物理結構來講, 將磁盤劃分為分區, 簇和磁區。那麼, 如何將邏輯上的目錄和檔案映射到物理上實際的簇和磁區, 就是檔案系統要解決的問題。

如果讓虛擬機直接讀取我們上一節生成的可啓動軟碟片鏡像, 或者將 boot.img 軟碟片用 `mount -o loop boot.img mountdir/` 掛載到某個目錄上, 系統肯定會報出“軟碟片未格式化”或者“檔案格式不可識別”的錯誤。這是因為任何系統可讀取的軟碟片都是被格式化過的, 而我們的 boot.img 是一個非常原始的軟碟片鏡像。那麼如何才能使軟碟片被識別為 FAT12 格式的軟碟片並且可以像普通軟碟片一樣存取呢？

系統在讀取一張軟碟片的時候, 會讀取軟碟片上存儲的一些關于檔案系統的信息, 軟碟片格式化的過程也就是系統把檔案系統信息寫入到軟碟片上的過程。但是我們不能讓系統來格式化我們的 boot.img , 如果那樣的話, 我們寫入的啓動程序也會被擦除。所以呢, 我們需要自己對軟碟片進行格式化。☻可能有人看到這裡就會很沮喪, 天那, 那該有多麻煩啊！不過我相信在讀完以下內容以後你會歡呼雀躍, 啊哈, 原來檔案系統挺簡單的嘛！

2.2.2 啓動磁區與 BPB

FAT 檔案系統的主要信息, 都被提供在前幾個磁區內, 其中第 0 號磁區尤其重要。在這個磁區內隱藏著一個叫做 BPB(BIOS Parameter Block) 的資料結構, 一旦我們把這個資料結構寫對了, 格式化過程也基本完成了☺。下面這個表中所示內容, 主要就是啓動磁區的 BPB 資料結構。

表 2.1: 啓動磁區的 BPB 資料結構和其它內容

名稱	偏移 bytes	大小 bytes	描述	Solrex.img 檔案中的值
BS_jmpBoot	0	3	跳轉指令，用于跳過以下的磁區信息	jmp LABEL_START nop
BS_OEMName	3	8	廠商名	"WB. YANG"
BPB_BytesPerSec	11	2	磁區大小（byte ），應為512	512
BPB_secPerClus	13	1	簇的磁區數，應為2 的冪，FAT12 為1	1
BPB_RsvdSecCnt	14	2	保留磁區，FAT12/16 應為1	1
BPB_NumFATs	16	1	FAT 結構數目，一般為2	2
BPB_RootEntCnt	17	2	根目錄項目數，FAT12 為224	224
BPB_TotSec16	19	2	磁區總數，1.44M 軟碟片為2880	2880
BPB_Media	21	1	設備類型，1.44M 軟碟片為F0h	0xf0

BPB_FATSz16	22	2	FAT 佔用磁區數，9	9
BPB_SecPerTrk	24	2	磁道磁區數，18	18
BPB_NumHeads	26	2	磁頭數，2	2
BPB_HiddSec	28	4	隱藏磁區，默認為0	0
BPB_TotSec32	32	4	如果 BPB_TotSec16 為 0，它記錄總磁區數	0
下面的磁區頭信息 FAT12/FAT16 與 FAT32 不同				
BS_DrvNum	36	1	中斷 0x13 的驅動器參數，0 為軟碟片	0
BS_Reserved1	37	1	Windows NT 使用，0	0
BS_BootSig	38	1	擴展引導標記 (29h)，指明此後 3 個域可用	0x29
BS_VolID	39	4	卷標序列號，00000000h	0
BS_VolLab	43	11	卷標，11 byte ，必須用空格20h 補齊	"Solrex 0.01"
BS_FilSysType	54	8	檔案系統標志，"FAT12 "	"FAT12 "
以下為非磁區頭信息部分				
啓動代碼及其它	62	448	啓動代碼、數據及填充字符	mov %cs,%ax...
啓動磁區標識符	510	2	可啓動磁區標志，0xAA55	0xaa55

哇, 天那, 這個 BPB 看起來很多東西的嘛, 怎麼寫啊？其實寫入這些信息很簡單, 因為它們都是固定不變的内容, 用下面的代碼就可以實現。

```
/* Floppy header of FAT12 */
    jmp     LABEL_START /* Start to boot. */
    nop                 /* nop required */
BS_OEMName:          .ascii  "WB. YANG"    /* OEM String, 8 bytes required */
BPB_BytsPerSec:      .2byte  512           /* Bytes per sector */
BPB_SecPerCluster:   .byte   1             /* Sector per cluster */
BPB_ResvdSecCnt:     .2byte  1             /* Reserved sector count */
BPB_NumFATs:         .byte   2             /* Number of FATs */
BPB_RootEntCnt:      .2byte  224           /* Root entries count */
BPB_TotSec16:        .2byte  2880          /* Total sector number */
BPB_Media:           .byte   0xf0          /* Media descriptor */
BPB_FATSz16:         .2byte  9             /* FAT size(sectors) */
BPB_SecPerTrk:       .2byte  18            /* Sector per track */
BPB_NumHeads:        .2byte  2             /* Number of magnetic heads */
BPB_HiddSec:         .4byte  0             /* Number of hidden sectors */
BPB_TotSec32:        .4byte  0             /* If TotSec16 equal 0, this works */
BS_DrvNum:           .byte   0             /* Driver number of interrupt 13 */
BS_Reserved1:        .byte   0             /* Reserved */
BS_BootSig:          .byte   0x29          /* Boot signal */
BS_VolID:            .4byte  0             /* Volume ID */
BS_VolLab:           .ascii  "Solrex 0.01" /* Volume label, 11 bytes required */
BS_FileSysType:      .ascii  "FAT12   "    /* File system type, 8 bytes required */

/* Initial registers. */
LABEL_START:
```

Fig 2.10: 啓動磁區頭的組合語言程式碼(節自chapter2/2/boot.S)

在上面的組合語言程式碼中, 我們只是順序地用字符填充了啓動磁區頭的資料結構, 填充的内容與表 2.1 中最後一列的内容相對應。把圖 2.10 中所示代碼添加到圖 2.2 的第二行和第三行之間, 然後再 make , 就能得到一張已經被格式化, 可啓動也可存儲檔案的軟碟片, 就是既可以使用 `mount -o loop boot.img mountdir/` 命令在普通 Linux 系統裡掛載, 也可用作虛擬機啓動的軟碟片鏡像檔案。

2.2.3 FAT12 資料結構

在上一個小節裡, 我們制作出了可以當作普通軟碟片使用的啓動軟碟片, 這樣我們就可以在這張軟碟片上存儲多個檔案了。可是還有一步要求我們沒有達到, 怎樣尋找存儲的某個引導檔案並將其加載到記憶體中運行呢？這就涉及到 FAT12 檔案系統中檔案的存儲方式了, 需要我們了解一些 FAT 資料結構和目錄結構的知識。

FAT 檔案系統對存儲空間分配的最小單位是「簇」, 因此檔案在佔用存儲空間時, 基本單位是簇而不是byte。即使檔案僅僅有 1 byte 大小, 系統也必須分給它一個最小存儲單元——簇。由表 2.1 中的 BPB_secPerClus 和 BPB_BytsPerSec 相乘可以得到每簇所包含的byte 數, 可見我們設置的是每簇包含 1*512=512 個byte, 恰好是每簇包含一個磁區。

存儲空間分配的最小單位確定了, 那麼 FAT 是如何分配和管理這些存儲空間的呢？FAT 的存儲空間管理是透過管理 FAT 表來實現的, FAT 表一般位于啓動磁區之後, 根目錄之前的若幹個磁區, 而且一般是兩個表。從根目錄區的下一個簇開始, 每個簇按照它在磁盤上的位置映射到 FAT 表裡。FAT 檔案系統的存儲結構粗略上來講如圖 2.11 所示。

Fig 2.11: FAT 檔案系統存儲結構圖

FAT 表的表項有點兒像資料結構中的單向鏈表節點的 next 指標, 先回憶一下, 單向鏈表節點的資料結構（C 語言）是：

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

在鏈表中，next 指標指向的是下一個相鄰節點。那麼 FAT 表與鏈表有什麼區別呢？首先，FAT 表將 next 指標集中管理，放在一起被稱為 FAT 表；其次，FAT 表項指向的是固定大小的檔案“簇”(data 段)，而且每個檔案簇都有自己對應的 FAT 表項。由于每個檔案簇都有自己的 FAT 表項，這個表項可能指向另一個檔案簇，所以 FAT 表項所佔byte 的多少就決定了 FAT 表最大能管理多少內存，FAT12 的 FAT 表項有 12 個bit，大約能管理 $2^{12}-$ 個檔案簇。

一個檔案往往要佔據多個簇，只要我們知道這個檔案的第一個簇，就可以到 FAT 表裡查詢該簇對應的 FAT 表項，該表項的內容一般就是此檔案下一個簇號。如果該表項的值大于 `0xff8`，則表示該簇是檔案最後一個簇，相當于單向鏈表節點的 next 指標為 NULL；如果該表項的值是 `0xff7` 則表示它是一個壞簇。這就是 **檔案的鏈式存儲** 。

2.2.4 FAT12 根目錄結構

怎樣讀取一個檔案我們知道了，但是如何找到某個檔案，即如何得到該檔案對應的第一個簇呢？這就到目錄結構派上用場的時候了，為了簡單起見，我們這裡只介紹根目錄結構。

如圖 2.11 所示，對于 FAT12/16，根目錄存儲在磁盤中固定的地方，緊跟在最後一個 FAT 表之後。根目錄的磁區數也是固定的，可以根據 BPB_RootEntCnt 計算得出：

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

根目錄的磁區號是相對于該 FAT 卷啓動磁區的偏移量：

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

FAT 根目錄其實就是一個由 32-bytes 的線性表構成的“檔案”，其每一個條目代表著一個檔案，這個 32-bytes 目錄項的格式如圖 2.2 所示。

表 2.2: 根目錄的條目格式

名稱	偏移(bytes)	長度(bytes)	描述	舉例(loader.bin)
DIR_Name	0	0xb	檔案名 8 byte ， 擴展名 3 byte	"LOADER□□BIN"
DIR_Attr	0xb	1	檔案屬性	0x20
保留位	0xc	10	保留位	0
DIR_WrtTime	0x16	2	最後一次寫入時間	0x7a5a
DIR_WrtDate	0x18	2	最後一次寫入日期	0x3188
DIR_FstClus	0x1a	2	此目錄項的開始簇編號	0x0002
DIR_FileSize	0x1c	4	檔案大小	0x000000f

知道了這些，我們就得到了足夠的信息去在磁盤上尋找某個檔案，在磁盤根目錄搜索並讀取某個檔案的步驟大致如下：

1. 確定根目錄區的開始磁區和結束磁區；

2. 遍歷根目錄區, 尋找與被搜索名相對應根目錄項；

3. 找到該目錄項對應的開始簇編號；

4. 以檔案的開始簇為根據尋找整個檔案的鏈接簇, 並依次讀取每個簇的內容。

2.3 讓啓動磁區加載引導檔案

有了 FAT12 檔案系統的相關知識之後, 我們就可以跨越 512 byte 的限制, 從檔案系統中加載檔案並執行了。

2.3.1 一個最簡單的 loader

為做測試用, 我們寫一個最小的程序, 讓它顯示一個字符, 然後進入死循環, 這樣如果 loader 加載成功並成功執行的話, 就能看到這個字符。

新建一個檔案 loader.S, 內容如圖 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: 一個最簡單的 loader(chapter2/2/loader.S)

這個程序在連接時需要使用連接檔案 solrex_x86_dos.ld, 如圖 2.13 所示, 這樣能更改代碼段的偏移量為 0x0100。這樣做的目的僅僅是為了與 DOS 系統兼容, 可以用此代碼生成在 DOS 下可調試的二進制檔案。

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

Fig 2.13: 一個最簡單的 loader(chapter2/2/solrex_x86_dos.ld)

2.3.2 讀取軟碟片磁區的 BIOS 13h 號中斷

我們知道了如何在磁盤上尋找一個檔案, 但是該如何將磁盤上內容讀取到記憶體中去呢？我們在第 2.1 節中寫的啓動磁區不需要自己寫代碼來讀取, 是因為它每次都被加載到記憶體的固定位置, 計算機在發現可啓動標識 0xaa55 的時候自動就會做加載工作。但如果我們想自己從軟碟片上讀取檔案的時候, 就需要使用到底層 BIOS 系統提供的磁盤讀取功能了。這裡, 我們主要用到 BIOS 13h 號中斷。

表 2.3 所示, 就是 BIOS 13 號中斷的參數表。從表中我們可以看到, 讀取磁盤驅動器所需要的參數是磁道（柱面）號、磁頭號以及當前磁道上的磁區號三個分量。由第 1.2 節所介紹的磁盤知識, 我們可以得到計算這三個分量的公式 2.1 。

$$\frac{\text{磁區號}}{18(\text{每磁道磁區數})} = \begin{cases} \text{商 Q} = \begin{cases} \text{柱面號} = Q \ >> \ 1 \\ \text{磁頭號} = Q \ \& \ 1 \end{cases} \\ \text{餘數 R} \Longrightarrow \ \text{起始磁區號} = R + 1 \end{cases} \tag{2.1}$$

表 2.3: BIOS 13h 號中斷的參數表

中斷號	AH	功能	呼叫參數	返回參數
13	0	磁盤復位	DL = 驅動器號 00, 01 為軟碟片, 80h, 81h, ··· 為硬盤	失敗： AH = 錯誤碼
	1	讀磁盤驅動器狀態		AH=狀態byte
	2	讀磁盤磁區	AL = 磁區數 $(CL)_{6,7}(CH)_{0\sim7}$ = 柱面號 $(CL)_{0\sim5}$ = 磁區號 DH/DL = 磁頭號/驅動器號 ES:BX = 數據緩衝區位址	讀成功： AH = 0 AL = 讀取的磁區數 讀失敗： AH = 錯誤碼
	3	寫磁盤磁區	同上	寫成功： AH = 0 AL = 寫入的磁區數 寫失敗： AH = 錯誤碼
	4	檢驗磁盤磁區	AL = 磁區數 $(CL)_{6,7}(CH)_{0\sim7}$ = 柱面號 $(CL)_{0\sim5}$ = 磁區號 DH/DL = 磁頭號/驅動器號	成功： AH = 0 AL = 檢驗的磁區數 失敗：AH = 錯誤碼
	5	格式化盤磁道	AL = 磁區數 $(CL)_{6,7}(CH)_{0\sim7}$ = 柱面號 $(CL)_{0\sim5}$ = 磁區號 DH/DL = 磁頭號/驅動器號 ES:BX = 格式化參數表指標	成功： AH = 0 失敗： AH = 錯誤碼

知道了這些, 我們就可以寫一個讀取軟碟片磁區的子函數了：

```
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: 讀取軟碟片磁區的函數(節自chapter2/2/boot.S)

2.3.3 搜索 loader.bin

讀取磁區的子函數寫好了, 下面我們編寫在軟碟片中搜索 loader.bin 的代碼：

```
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 的代碼片段(節自chapter2/2/boot.S)

這段代碼的功能就是我們前面提到過的, 遍歷根目錄的所有磁區, 將每個磁區加載入記憶體, 然後從中尋找檔案名為 loader.bin 的條目, 直到找到為止。找到之後, 計算出 loader.bin 的起始磁區號。其中用到的變量和字符串的定義見圖 2.16 中代碼片段的定義。

```
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 使用的變量定義(節自chapter2/2/boot.S)

由于在代碼中有一些打印工作, 我們寫了一個函數專門做這項工作。為了節省代碼長度, 被打印字符串的長度都設置為 9 byte , 不夠則用空格補齊, 這樣就相當于一個備用的二維數組, 通過數字定位要打印的字符串, 很方便。打印字符串的函數 DispStr 見圖 2.17 , 呼叫它的時候需要從寄存器 dh 傳入參數字符串序號。

```
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: 打印字符串函數 DispStr (節自chapter2/2/boot.S)

2.3.4 加載 loader 入記憶體

在尋找到 loader.bin 之後, 就需要把它裝入記憶體。現在我們已經有了 loader.bin 的起始磁區號, 利用這個磁區號可以做兩件事：一, 把起始磁區裝入記憶體；二, 通過它找到 FAT 中的條目, 從而找到 loader.bin 檔案所佔用的其它磁區。

這裡, 我們把 loader.bin 裝入記憶體中的 BaseOfLoader:OffsetOfLoader 處, 但是在圖 2.15 中我們將根目錄區也是裝載到這個位置。因為在找到 loader.bin 之後, 該記憶體區域對我們已經沒有用處了, 所以它盡可以被覆蓋。

我們已經知道了如何裝入一個磁區, 但是從 FAT 表中尋找其它的磁區還是一件麻煩的事情, 所以我們寫了一個函數 GetFATEntry 來專門做這件事情, 函數的輸入是磁區號, 輸出是其對應的 FAT 項的值, 見圖 2.18。

```
246
247 /* ================================================================
```

```
248   Routine: GetFATEntry
249   Action: Find %ax sector's index in FAT, save result in %ax
250 */
251 GetFATEntry:
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 FATEntry 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 <- FATEntry's sector */
272     mov     $2,%cl            /* Read 2 sectors in 1 time, because FATEntry */
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: 尋找 FAT 項的函數 GetFATEntry (節自chapter2/2/boot.S)

這裡有一個對磁區號判斷是奇是偶的問題, 因為 FAT12 的每個表項是 12 位, 即 1.5 個byte , 而我們這裡是用字（2 byte ）來讀每個表項的, 那麼讀到的表項可能在高 12 位或者低 12 位, 就要用磁區號的奇偶來判斷應該取哪 12 位。由于磁區號*3/2 的商就是對應表項的偏移量, 餘數代表著是否多半個byte , 如果存在餘數 1 , 則取高 12 位為表項值；如果餘數為 0, 則取低 12 位作為表項值。舉個具體的例子, 下面是一個真實的 FAT12 表項內容（兩行是分別用byte 和字來表示的結果）：

```
0000200: 00 00 00 00 40 00 FF 0F
```

```
0000200:  0000  0000  0040  0FFF
```

我們來找磁區號 3 對應的 FAT12 表項。3*3/2 = 4...1, 按照字來讀取偏移 4 對應的位址, 我們得到 0040。由于磁區號 3 是奇數, 有餘數, 則應取高 12 位作為 FAT12 表項內容, 將 0040 右移 4 位再算術與 0x0fff, 我們得到 0x0004, 即為對應的 FAT12 表項, 說明磁區號 3 的後繼磁區號是 4 。

然後, 我們就可以將 loader.bin 整個檔案加載到記憶體中去了, 見圖 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: 加載 loader.bin 的代碼(節自chapter2/2/boot.S)

在圖 2.19 中我們看到一個宏 DeltaSectorNo , 這個宏就是為了將 FAT 中的簇號轉換為磁區號。由于根目錄區的開始磁區號是 19 , 而 FAT 表的前兩個項 0,1 分別是磁盤識別字和被保留, 其表項其實是從第 2 項開始的, 第 2 項對應著根目錄區後的第一個磁區, 所以磁區號和簇號的對應關系就是：

$$\begin{aligned} \text{磁區號} &= \text{簇號} + \text{根目錄區佔用磁區數} + \text{根目錄區開始磁區號 - 2} \\ &= \text{簇號} + \text{根目錄區佔用磁區數} + 17 \end{aligned}$$

這就是 DeltaSectorNo 的值 17 的由來。

2.3.5 向 loader 轉交控制權

我們已經將 loader 成功地加載入了記憶體, 然後就需要進行一個跳轉, 來執行 loader 。

```
168
169 /*************************************************************
170    Jump to LOADER.BIN's start address in memory.
171 */
172     jmp     $BaseOfLoader,$OffsetOfLoader
173 /*************************************************************/
174
```

Fig 2.20: 跳轉到 loader 執行(節自chapter2/2/boot.S)

2.3.6 生成鏡像並測試

我們寫好了匯編源代碼, 那麼就需要將源代碼編譯成可執行檔案, 並生成軟碟片鏡像了。

```
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 編譯(節自chapter2/2/Makefile)

上面的代碼比較簡單, 我們可以通過一個 make 命令編譯生成 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
########################################################################
```

由于我們的目標就是讓啓動磁區加載引導檔案, 所以需要把引導檔案放入軟碟片鏡像中。那麼如何將 LOADER.BIN 放入 boot.img 中呢？我們只需要掛載 boot.img 並將 LOADER.BIN 拷貝進入被掛載的目錄, 為此我們在 Makefile 中添加新的編譯目標 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: 拷貝 LOADER.BIN 入 boot.img(節自chapter2/2/Makefile)

由于掛載軟碟片鏡像在很多 Linux 系統上需要 root 權限, 所以我們沒有將 copy 目標添加到 all 的依賴關系中。在執行 make copy 命令之前我們必須先獲得 root 權限。

```
$ 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/;
```

如果僅僅生成 boot.img 而不將 loader.bin 裝入它, 用這樣的軟碟片啓動會顯示找不到 LOADER：

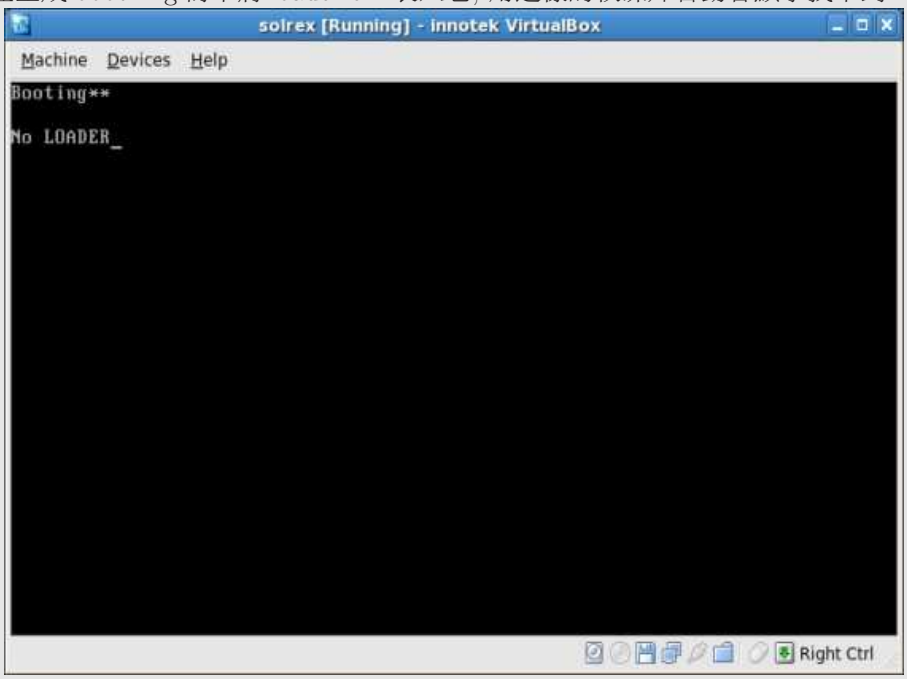


Fig 2.23: 沒有裝入 LOADER.BIN 的軟碟片啓動

裝入 loader.bin 之後再用 boot.img 啓動, 我們看到虛擬機啓動並在屏幕中間打印出了一個字符「L」, 這説明我們前面的工作都是正確的。

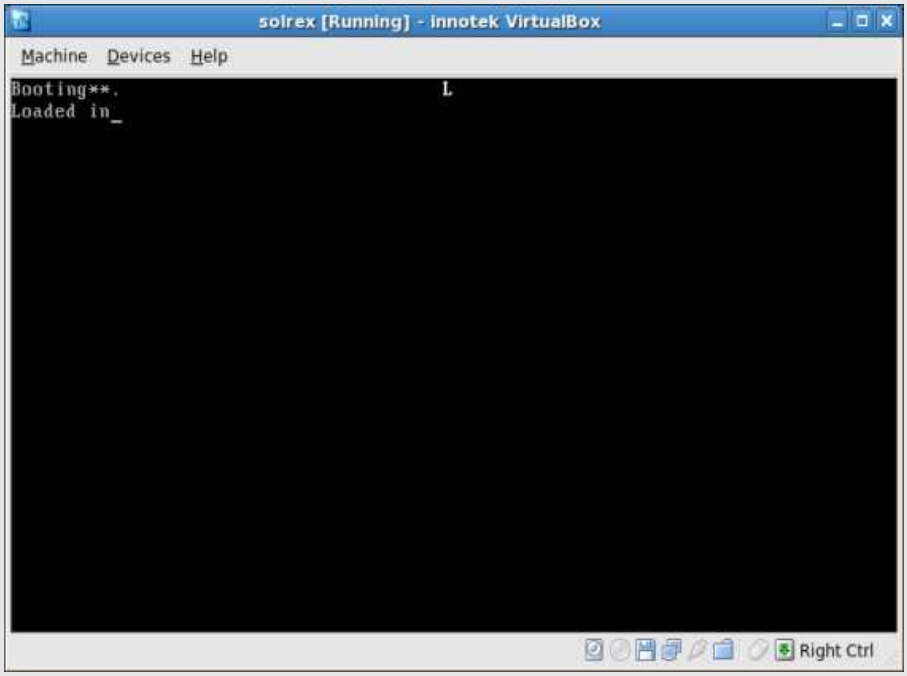


Fig 2.24: 裝入了 LOADER.BIN 以後再啓動

CHAPTER 3

進入保護模式

前面我們看到, 通過一些很簡單的程式碼, 我們做到了啓動一個微型系統, 載入檔案系統中的檔案到記憶體並運行的功能。應該注意的是, 在前面的程式碼中我們使用的記憶體空間都很小。我們看一下 boot.bin 和 LOADER.BIN 的大小就能感覺出來（當然, 可執行檔案小未必使用記憶體空間小, 但是這兩個檔案也太小了 ☺）。

```
$ 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 個 byte（其中還有我們填充的內容, 實際指令只有 480 個 byte ）, 而 LOADER.BIN 更過分, 只有 15 個 byte 大小。可想而知這兩個文件在記憶體中能使用多大的空間吧。如果讀者有些組合語言經驗的話, 就會發現我們在前面的程序中使用的存儲器尋址都是在真實模式下進行的, 即：由段暫存器（cs, ds: 16-bit）配合段內偏移位址（16-bit）來定位一個實際的 20-bit 物理位址, 所以我們前面的程序最多支持 $2^{20} = 2^{10} * 2^{10} = 1024 * 1024$ bytes $=$ 1MB 的尋址空間。

哇, 1MB 不小了, 我們的作業系統加一起連 1KB 都用不到, 1MB 尋址空間足夠了。但是需要考慮到的一點是, 就拿我們現在用的 1.44MB的（已經被淘汰的）軟盤標準來說, 如果軟盤上某個文件超過 1MB , 我們的作業系統就沒辦法處理了。那麼如果以後把作業系統安裝到硬盤上之後呢？我們就沒辦法處理稍微大一點的文件了。

所以我們要從最原始的 Intel 8086/8088 CPU 的真實模式中跳出來, 進入 Intel 80286 之後系列 CPU 給我們提供的保護模式。這還將為我們帶來其它更多的好處, 具體內容請繼續往下讀。

3.1 真實模式和保護模式

如果您需要更詳細的知識, 也許您更願意去讀Intel 的手冊, 本節內容主要集中在：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 一段歷史

Intel 公司在 1978 年發布了一款 16 位字長 CPU: 8086 , 最高頻率 5 MHz~10 MHz , 集成了 29,000 個晶體管, 這款在今天感覺像玩具一樣的 CPU 卻是奠定今天 Intel PC 芯片市場地位的最重要的產品之一。雖然它的後繼者 8088 , 加强版的 8086 （增加了一個 8 比特的外部總線）才是事實上的 IBM 兼容機（PC, 個人電腦）雛形的核心, 但人們仍然習慣于用 8086 作為廠商標志代表 Intel 。

因為受到字長（16 位）的限制, 如果僅僅使用單個暫存器尋址, 8086 僅僅能訪問 64KB(2^{16}) 的位址空間, 這顯然不能滿足一般要求, 而當時 1MB(2^{20}) 對于一般的應用就比較足夠了, 所以 8086 使用了 20 bit 的位址線。

在 8086 剛發布的時候, 沒有「真實模式」這個說法, 因為當時的 Intel CPU 只有一種模式。在 Intel 以後的發布中, 80286 引入了“保護模式”尋址方式, 將 CPU 的尋址範圍擴大到 16(2^{24}) MB , 但是 80286 仍然是一款 16 位 CPU , 這就限制了它的廣泛應用。但是“真實模式”這個說法, 就從 80286 開始了。

接下來的發展就更快了, 1985 年發布的 i386 首先讓 PC CPU 進入了 32 位時代, 由此而帶來的好處顯而易見, 尋址能力大大增强, 但是多任務處理和虛擬存儲器的需求仍然推動著 i386 向更完善的保護模式發展。下面我們來了解一下“真實模式”和“保護模式”的具體涵義。

3.1.2 真實模式

真實模式（real mode）, 有時候也被成為真實位址模式（real address mode）或者兼容模式（compatibility mode）是 Intel 8086 CPU 以及以其為基礎發展起來的 x86 兼容 CPU 採用的一種操作模式。其主要的特點有：20 比特的分段訪問的記憶體位址空間（即 1 MB 的尋址能力）；程序可直接訪問 BIOS 中斷和外設；硬件層不支持任何記憶體保護或者多任務處理。 80286 之後所有 x86 CPU 在加電自舉時都是首先進入真實模式； 80186 以及之前的 CPU 只有一種操作模式, 相當于真實模式。

3.1.3 保護模式

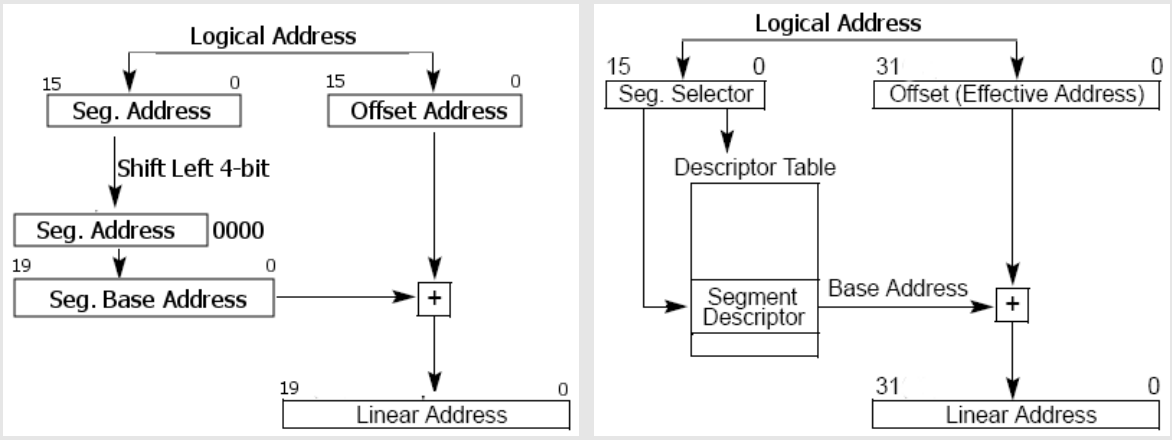
保護模式（protected mode）, 有時候也被成為保護的虛擬位址模式（protected virtual address mode）, 也是一種 x86 兼容 CPU 的工作模式。保護模式為系統軟件實現虛擬記憶體、分頁機制、安全的多任務處理的功能支持, 還有其它為作業系統提供的對應用程序的控制功能支持, 比如：權限、真實模式應用程序兼容、虛擬 8086 模式。

3.1.4 真實模式和保護模式的尋址模式

前面提到過, 真實模式下的位址線是 20 bit 的, 所以真實模式下的尋址模式使用分段方式來解決 16 位字長機器提供 20 bit 位址空間的問題。這個分段方法需要程序員在編制程序的過程中將存儲器劃分成段, 每個段內的位址空間是線性增長的, 最大可達 64K(2^16), 這樣段位址就可以使用 16 位表示。段基址（ 20-bit ）的最低 4 位必須是 0 , 這樣段基址就可以使用 16 位段位址來表示, 需要時將段位址左移 4 位就得到段起始位址。除了便于尋址之外, 分段還有一個好處, 就是將程序的程式碼段、資料段和堆堆疊段等隔離開, 避免相互之間產生干擾。

當計算某個單元的物理位址時, 比如組合語言中的一個 Label , 就通過段位址（ 16-bit ）左移 4 位得到段基址（ 20-bit ）, 再加上該單元（ Label ）的段偏移量（ 16-bit ）來得到其物理位址（ 20-bit ）, 如圖 3.1a 所示。

一般情況下, 段位址會被放在四個段暫存器中, 即：程式碼段 CS, 資料段 DS, 堆堆疊段 SS 和附加段 ES 暫存器。這樣在加載數據或者控制程序運行的時候, 只需要一個偏移量參數, CPU 會自動用對應段的



(a) 真實模式尋址模型 (b) 保護模式尋址模型

Fig 3.1: 真實模式與保護模式尋址模型比較

起始位址加上偏移量參數來得到需要的位址。（後繼 CPU 又加上了兩個段暫存器 FS 和 GS , 不過使用方式是基本一樣的。）

由此可見, 真實模式的尋址模式是很簡單的, 就是用兩個 16 位邏輯位址（段位址：偏移位址）組合成一個 20 bit 物理位址, 而保護模式的尋址方式就要稍微復雜一點了。

> Intel 的 CPU 在保護模式下是可以選擇打開分頁機制的, 但為了簡單起見, 我們先不開啓分頁機制, 所以下面的講解針對只有分段機制的保護模式展開。

在保護模式下, 每個單元的物理位址仍然是由邏輯位址表示, 但是這個邏輯位址不再由（段位址：偏移位址）組成了, 而是由（段選擇子：偏移位址）表示。這裡的偏移位址也變成了 32 位的, 所以段空間也比真實模式下大得多。偏移位址的意思和真實模式下並沒有本質不同, 但段位址的計算就要復雜一些了, 如圖 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, 全局段選擇子表）是保存著所有段選擇子的信息, 段選擇子（Segment Selector）是一個指向某個段選擇子的索引。

如圖 3.1b 所示, 當我們計算某個單元的物理位址時, 只需要給出（段選擇子：偏移位址）, CPU 會從 GDT 中按照段選擇子找到對應的段描述符, 從段描述符中找出段基址, 將段基址加上偏移量, 就得到了該單元的物理位址。

3.2 與保護模式初次會面

介紹完了保護模式和真實模式的不同, 下面我們就嘗試一下進入保護模式吧。在上一章我們已經實現了用啓動磁區加載引導文件, 所以這裡我們就不用再去管啓動磁區的事情了, 下面的修改均在 loader.S 中進行。上一章的 loader.S 僅僅實現在屏幕的上方中間打印了一個 `L` , 下面我們的 loader.S 要進入保護模式來打印一些新東西。

首先, 我們來理清一下應該如何進入保護模式：

1. 我們需要一個 GDT。由于保護模式的尋址方式是基于 GDT 的, 我們得自己寫一個 GDT 資料結構並將其載入到系統中。

2. 我們需要為進入保護模式作準備。由于保護模式和真實模式運行方式不同, 在進入保護模式之前, 我們需要一些準備工作。

3. 我們需要一段能在保護模式下運行的程式碼demo, 以提示我們成功進入了保護模式。

下面我們就來一步步完成我們的第一個保護模式 loader 。

3.2.1 GDT 資料結構

要寫 GDT, 首先得了解 GDT 的資料結構。GDT 實際上只是一個存儲段描述符的線性表（可以理解成一個段描述符數組）, 對它的要求是其第一個段描述符置為空, 因為處理機不會去處理第一個段描述符, 所以理解 GDT 的資料結構難點主要在于理解段描述符的資料結構。

段描述符主要用來為處理機提供段位址, 段訪問控制和狀態信息。圖 3.2 顯示了一個基本的段描述符結構：

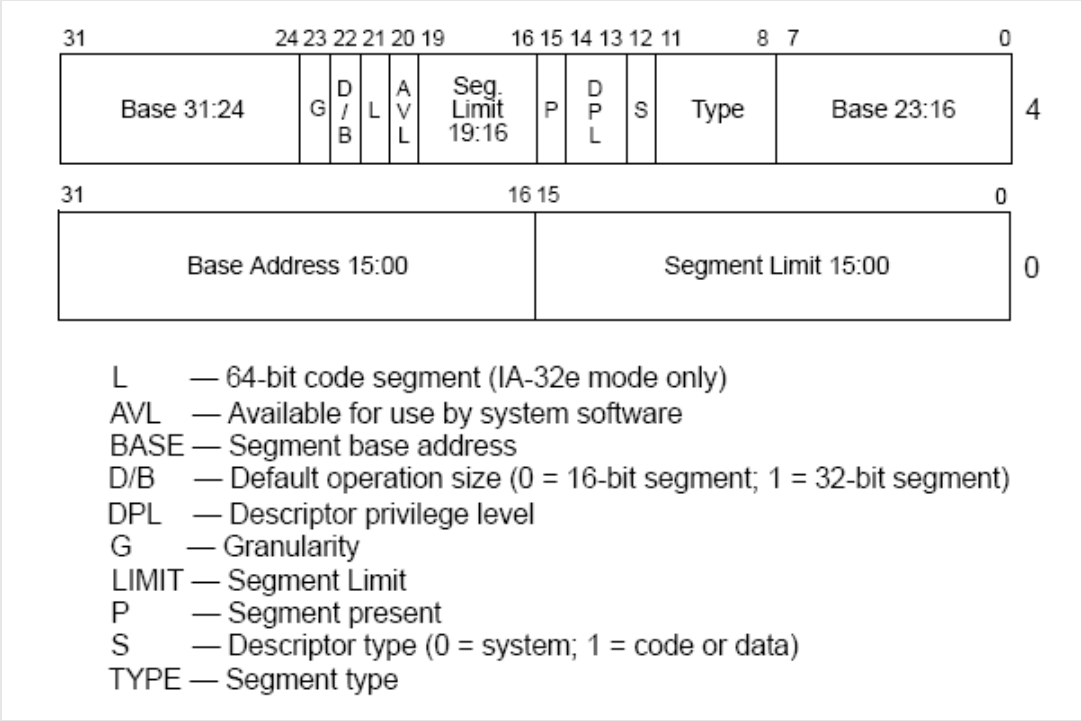


Fig 3.2: 段描述符

看到上面那麼多內容, 是不是感覺有點兒恐怖啊！其實簡單的來看, 我們現在最關注的是段基址, 就是圖 3.2 中標記為 Base 的部分。可以看到, 段基址在段描述符中被分割為三段存儲, 分別是：Base 31:24, Base 23:16, Base Address 15:0, 把這三段拼起來, 我們就得到了一個 32 位的段基址。

有了段基址, 就需要有一個界限來避免程序跑丟發生段錯誤, 這個界限就是圖 3.2 中標記為 Limit 的部分, 將 Seg. Limit 19:16 和 Segment Limit 15:0 拼起來我們就得到了一個 20 bit 的段界限, 這個界限就是應該是段需要的長度了。

下面還要說的就是那個 D/B Flag , D/B 代表 Default Operation Size , 0 代表 16 位的段, 1 代表 32 位的段。為了充分利用 CPU , 我們當然要設置為 32 位模式了。剩下那些亂七八糟的 Flag 呢, 無非就是提供段的屬性（程式碼段還是資料段？只讀還是讀寫？）, 我們將在第 3.3.2 節為大家詳細介紹。

這些東西那麼亂, 難道要每次一點兒一點兒地計算嗎？放心, 程序員自有辦法, 請看下面的程序：

```
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) & 0xF00) | (\Attr & 0xF0FF)
68     .byte   (\Base >> 24) & 0xFF
69 .endm
70
```

Fig 3.3: 自動生成段描述符的宏定義(節自chapter3/1/pm.h)

圖 3.3 中所示, 就是自動生成段描述符的匯編宏定義。我們只需要給宏 Descriptor 三個參數：Base（段基址）, Limit（段界限[段長度]）, Attr（段屬性）, Descriptor 就會自動將三者展開放到段描述符中對應的位置。看看我們在程序中怎麼使用這個宏：

```
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: 自動生成段描述符的宏使用示例(節自chapter3/1/loader.S)

圖 3.4 中, 就利用 Descriptor 宏生成了三個段描述符, 形成了一個 GDT。注意到沒有, 第一個段描述符是空的（參數全為 0）。這裡 LABEL_DESC_CODE32 的段基址為 0 是因為我們無法確定它的準確位置, 它將在運行期被填入。

有人可能會產生疑問, 段基址和段界限什麼意思我們都知道了, 那段屬性怎麼回事呢？ DA_C, DA_32, DA_DRW 都是什麼東西啊？是這樣的, 為了避免手動一個一個置段描述符中的 Flag , 我們預先定義了一些常用屬性, 用的時候只需要將這些屬性加起來作為宏 Descriptor 的參數, 就能將段描述符中的所有 flag 置上（記得 C 語言中 fopen 的參數嗎？）。這些屬性的定義如下（沒必要細看, 用的時候再找即可）：

```
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: 預先設置的段屬性(節自chapter3/1/pm.h)

3.2.2 保護模式下的 demo

為什麼把這節提前到第 3.2.3 節前講呢？因為要寫入 GDT 正確的段描述符, 首先要知道段的信息, 我們就得先準備好這個段：

```
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: 第一個在保護模式下運行的demo(節自chapter3/1/loader.S)

其實這個段的作用很簡單, 通過操縱視頻段數據, 在屏幕中間打印一個紅色的"P"（和我們前面使用 BIOS 中斷來打印字符的方式有所不同）。

3.2.3 加載 GDT

GDT 所需要的信息我們都知道了, GDT 表也通過圖 3.4 中的程式碼實現了。那麼, 我們應該向 GDT 中填入缺少的信息, 然後載入 GDT 了。將 GDT 載入處理機是用 `lgdt` 匯編指令實現的, 但是 `lgdt` 指令需要存放 GDT 的基址和界限的指標作參數, 所以我們還需要知道 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: 加載 GDT(節自chapter3/1/loader.S)

圖 3.7 中 GdtPtr 所指, 即為 GDT 的界限和基址所存放位置。某段描述符對應的 GDT 選擇子, 就是其段描述符相對于 GDT 基址的索引（在我們例子裡 GDT 基址為 LABEL_GDT 指向的位置）。這裡需要注意的是, 雖然我們在程式碼中寫：

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

但實際上段選擇子在使用時需要右移 3 個位作為索引去尋找其對應的段描述符, 段選擇子的右側 3 個位是為了標識 TI 和 RPL 的, 如圖 3.12 所示, 這點我們將在第 3.3.1 節和第 3.3.2 節中詳細介紹。但是這裡為什麼能直接用位址相減得到段選擇子呢？因為段描述符的大小是 8 個 byte , 用段描述符的位址相減的話, 位址差的最右側三個位就默認置 0 了。

在圖 3.7 中所示的程式碼, 主要幹了兩件事：第一, 將圖 3.6 所示 demo 的段基址放入 GDT 中對應的段描述符中；第二, 將 GDT 的基址放到 GdtPtr 所指的資料結構中, 並加載 GdtPtr 所指的資料結構到 GDTR 暫存器中（使用 lgdt 指令）。

3.2.4 進入保護模式

進入保護模式前, 我們需要將中斷關掉, 因為保護模式下中斷處理的機制和真實模式是不一樣的, 不關掉中斷可能帶來麻煩。使用 cli 匯編指令可以清除所有中斷 flag。

由于真實模式下僅有 20 條位址線：A0, A1, ..., A19, 所以當我們要進入保護模式時, 需要打開 A20 位址線。打開 A20 位址線有至少三種方法, 我們這裡採用 IBM 使用的方法, 通常被稱為：“Fast A20 Gate”, 即修改系統控制端口 92h , 因為其端口的第 1 位控制著 A20 位址線, 所以我們只需要將 0b00000010 賦給端口 92h 即可。

當前面兩項工作完成後, 我們就可以進入保護模式了。方法很簡單, 將 cr0 暫存器的第 0 位 PE 位置為 1 即可使 CPU 切換到保護模式下運行。

```
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 CR0. */
74      movl    %cr0, %eax
75      orl     $1, %eax
76      movl    %eax, %cr0
77
```

Fig 3.8: 進入保護模式(節自chapter3/1/loader.S)

3.2.5 特別的混合跳轉指令

雖然已經進入了保護模式, 但由于我們的 CS 暫存器存放的仍然是真實模式下 16 位的段信息, 要跳轉到我們的 demo 程序並不是那麼簡單的事情。因為 demo 程序是 32 位的指令, 而我們現在仍然運行的是 16 位的指令。從 16 位的程式碼段中跳轉到 32 位的程式碼段, 不是一般的 near 或 far 跳轉指令能解決得了的, 所以這裡我們需要一個特別的跳轉指令。在這條指令運行之前, 所有的指令都是 16 位的, 在它運行之後, 就變成 32 位指令的世界。

在 Intel 的手冊中, 把這條混合跳轉指令稱為 far jump(ptr16:32) , 在 NASM 手冊中, 將這條指令稱為 Mixed-Size Jump , 我們就沿用 NASM 的說法, 將這條指令稱為混合字長跳轉指令。NASM 提供了這條指令的組合語言實做：

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

NASM 的手冊中說 GAS 沒有提供這條指令的實做, 我就用 .byte 偽程式碼直接寫了二進制指令：

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

但是有位朋友提醒我說現在的 GAS 已經支持混合字長跳轉指令（如圖 3.9）, 看來 NASM 的手冊好久沒有維護嘍 ☺ 。

```
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:
```

Fig 3.9: 混合字長跳轉指令(節自chapter3/1/loader.S)

執行這條混合字長的跳轉指令時, CPU 就會用段選擇子 SelectorCode32 去尋找 GDT 中對應的段, 由于段偏移是 0 , 所以 CPU 將跳轉到圖 3.6 中 demo 程序的開頭。為了方便閱讀, 整個 loader.S 的程式碼附在圖 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).        */
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
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 CR0. */
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 生成鏡像並測試

使用與第 2.3.6 節完全相同的方法, 我們可以將程式碼編譯並將 LOADER.BIN 拷貝到鏡像文件中。利用最新的鏡像文件啓動 VirtualBox 我們得到圖 3.11 。

可以看到, 屏幕的左側中央打出了一個紅色的 P , 這就是我們那個在保護模式下運行的簡單 demo 所做的事情, 這說明我們的程式碼是正確的。從真實模式邁入保護模式, 這只是一小步, 但對于我們的作業系統來說, 這是一大步。從此我們不必再被限制到 20 bit 的位址空間中, 有了更大的自由度。

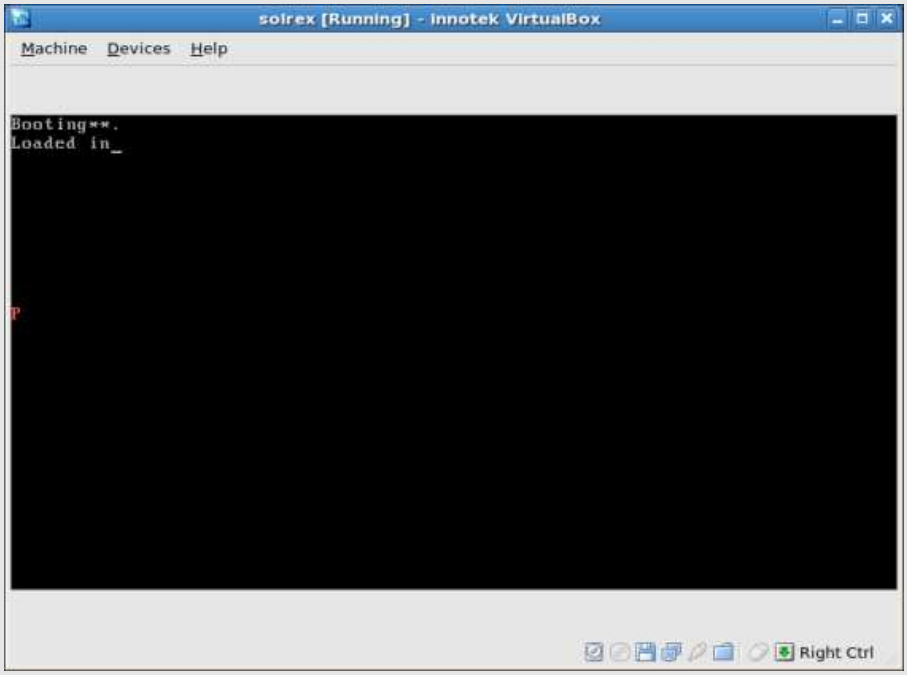


Fig 3.11: 第一次進入保護模式

3.3 段式存儲

如果您仔細閱讀了圖 3.1b , 您就會發現圖中並未提到 GDT , 而是使用的 Descriptor Table(DT) 。這是因為對于 x86 架構的 CPU 來說, DT 總共有兩個：我們上節介紹過的 GDT 和下面要介紹的 LDT 。這兩個描述符表構成了 x86 CPU 段式存儲的基礎。顧名思義, GDT 作為全局的描述符表, 只能有一個, 而 LDT 作為局部描述符表, 就可以有很多個, 這也是以後作業系統給每個任務分配自己的存儲空間的基礎。

3.3.1 LDT 資料結構

事實上, LDT 和 GDT 的差別非常小, LDT 段描述符的資料結構和圖 3.2 所示是一樣的。所不同的就是, LDT 用指令 `lldt` 來加載, 並且指向 LDT 描述符項的段選擇子的 TI 位置必須標識為 1 , 如圖 3.12

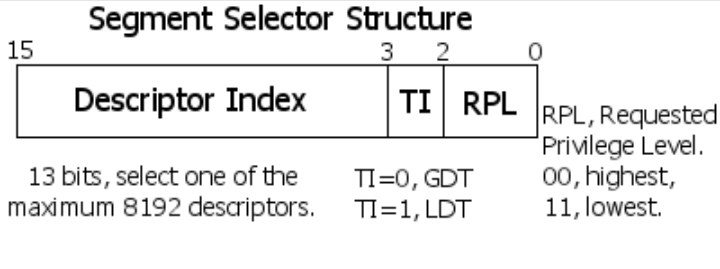


Fig 3.12: 段選擇子資料結構

所示。這樣, 在使用 TI flag := 1 的段選擇子時, 作業系統才會從當前的 LDT 而不是 GDT 中去尋找對應的段描述符。

這裡值得注意的一點是：GDT 是由線性空間裡的位址定義的, 即 `lgdt` 指令的參數是一個線性空間的位址；而 LDT 是由 GDT 中的一個段描述符定義的, 即 `lldt` 指令的參數是 GDT 中的一個段選擇子。這是因為在加載 GDT 之前尋址模式是真實模式的, 而加載 GDT 後尋址模式變成保護模式尋址, 將 LDT 作為 GDT 中的段使用, 也方便作業系統在多個 LDT 之間切換。

3.3.2 段描述符屬性

我們在介紹圖 3.2 時, 並沒有完全介紹段描述符的各個 Flag 和可能的屬性, 這一小節就用來專門介紹段描述符的屬性, 按照圖 3.2 中的 Flag 從左向右的順序：

- **G**: G(Granularity, 粒度)：如果 G flag 置為 0 , 段的大小以 byte 為單位, 段長度範圍是 `1 byte`∼`1 MB` ；如果 G flag 置為 1 , 段的大小以 4 KB 為單位, 段長度範圍是 `4 KB ∼ 4 GB` 。
- **D/B**：D/B(Default operation size/Default stack pionter size and/or upper Bound, 默認操作大小), 其意思取決于段描述符是程式碼段、資料段或者堆堆疊段。該 flag 置為 0 代表程式碼段/資料段為 16 位的；置為 1 代表該段是 32 位的。
- **L**：L(Long, 長), L flag 是 IA-32e(Extended Memory 64 Technology) 模式下使用的標志。該 flag 置為 1 代表該段是正常的 64 位的程式碼段；置為 0 代表在兼容模式下運行的程式碼段。在 IA-32 架構下, 該位是保留位, 並且永遠被置為 0 。
- **AVL**：保留給作業系統軟件使用的位。
- **P**：P(segment-Present, 段佔用？) flag 用于標志段是否在記憶體中, 主要供記憶體管理軟件使用。如果 P flag 被置為 0 , 說明該段目前不在記憶體中, 該段指向的記憶體可以暫時被其它任務佔用；如果 P flag 被置為 1 , 說明該段在記憶體中。如果 P flag 為 0 的段被訪問, 處理機會產生一個 segment-not-present(#NP) 異常。
- **DPL**：DPL(Descriptor Privilege Level)域標志著段的權限, 取值範圍是從 0∼3(2-bit) , 0 代表著最高的權限。關于權限的作用, 我們將在下節討論。
- **S**：S(descriptor type) flag 標志著該段是否系統段：置為 0 代表該段是系統段；置為 1 代表該段是程式碼段或者資料段。

- **Type**：Type 域是段描述符裡最復雜的一個域, 而且它的意義對于程式碼/資料段描述符和系統段/門描述符是不同的, 下面我們用兩張表來展示當 Type 置為不同值時的意義。

  表 3.1 所示即為程式碼/資料段描述符的所有 Type 可能的值(0-15, 4-bit)以及對應的屬性含意, 表 3.2 所示為系統段/門描述符的 Type 可能的值以及對應的屬性含意。這兩張表每個條目的內容是自明的, 而且我們在後面的討論中將不止一次會引用這兩張表的內容, 所以這裡對每個條目暫時不加詳細闡述。

表 3.1: 程式碼/資料段描述符的 Type 屬性列表

Type Field					Descriptor Type	Description
Decimal	11	10 E	9 W	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: 系統段/門描述符的 Type 屬性列表

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

從目前的需求來看, 對 LDT 並沒有非介紹不可的理由, 但是理解 LDT 的使用, 對理解段式存儲和處理機多任務存儲空間分配有很大的幫助。所以我們在下面的程式碼中實現幾個簡單的例子：一, 建立 32 位數據和堆堆疊兩個段並將描述符添加到 GDT 中；二, 添加一段簡單程式碼, 並以其段描述符為基礎建立一個 LDT；三, 在 GDT 中添加 LDT 的段描述符並初始化所有 DT ；四, 進入保護模式下運行的 32 位程式碼段後, 加載 LDT 並跳轉執行 LDT 中包含的程式碼段。

首先, 建立 32 位全局資料段和堆堆疊段, 並將其描述符添加到 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 位全局資料段和堆堆疊段, 以及對應的 GDT 結構(節自chapter3/2/loader.S)

在圖 3.13 中, 我們首先建立了一個全局的資料段, 並在資料段裡放置了兩個字符串, 分別用來進入保護模式後和跳轉到 LDT 指向的程式碼段後作為信息輸出。然後又建立了一個全局的堆堆疊段, 為堆堆疊段預留了 512 byte 的空間, 並將堆疊頂設置為距堆疊底 511 byte 處。然後與上節介紹的類似, 將資料段和堆堆疊段的段描述符添加到 GDT 中, 並設置好對應的段選擇子。

要注意到資料段、堆堆疊段和程式碼段的段描述符屬性不盡相同。資料段的段描述符屬性是 DA_DRW , 回憶我們前面 pm.h 的內容（圖 3.5 ）, DA_DRW 的內容是 0x92, 用二進制就是 10010010, 其後四位就對應著圖 3.1 中的第二 2(0010) 項, 說明這個段是可讀寫的資料段；前四位對應著 P|DPL|S 三個 flag , 即 P:1, DPL:00, S:1 , 與第 3.3.2 節結合理解, 意思就是該段在記憶體中, 為最高的權限, 非系統段。所以我們可以看到 pm.h 中的各個屬性變量定義, 就是將二進制的屬性值用可理解的變量名表示出來, 在用的時候直接加上變量即可。

同理我們也可以分別來理解 GDT 中堆堆疊段和程式碼段描述符的屬性定義。因為不同類型的屬性使用的是段描述符中不同的位, 所以不同類型的屬性可以直接相加得到復合的屬性值, 例如堆堆疊段的 (DA_DRWA + DA_32) , 其意思類似于 C++ 中 fstream 打開文件時可以對模式進行算術或（ios_base::in | ios_base::out）來得到復合參數。

其次, 添加一段簡單的程式碼, 並以其描述符為基礎建立一個 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 位程式碼段, 以及對應的 LDT 結構(節自chapter3/2/loader.S)

LABEL_CODEA 就是我們為 LDT 建立的簡單程式碼段, 其作用就是操作顯存在屏幕的第 12 行開始用紅色的字打印出偏移 OffsetLDTMessage 指向的全局資料段中的字符串。下面就是以 LABEL_CODEA 為基礎建立的 LDT , 從 LDT 的結構來說, 與 GDT 沒有區別, 但是我們不用像 GdtPtr 再建立一個 LdtPtr , 因為 LDT 實際上是在 GDT 中定義的一個段, 不用真實模式的線性位址表示。

LDT 的選擇子是與 GDT 選擇子有明顯區別的, 圖 3.12 清楚地解釋了這一點, 所以指向 LDT 的選擇子都應該將 TI 位置 1 , 在圖 3.14 的最後一行也實現了這一操作。

第三, 在 GDT 中添加 LDT 的段描述符（在圖 3.13 中我們已經能看到在 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屬于重復性工作, 我們在 pm.h 中添加一個匯編宏 InitDesc 來幫我們做這件事情。

```
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: 自動初始化段描述符的宏程式碼(節自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 */
112                                   /* this mixed-size jump insn of gas.  */
```

Fig 3.16: 在真實模式程式碼段中初始化所有段描述符(節自chapter3/2/loader.S)

初始化各個段描述符的方式與上一節介紹的初始化 GDT 描述符的方式沒有什麼本質不同, 因為屬性都已經預設好, 運行時只需要將段位址填入描述符中的位址域即可, 程式碼都是重復的。我們引入宏InitDesc的幫助, 能大大縮短程式碼長度, 增强程式碼的可讀性。

第四, 進入保護模式下運行的 32 位程式碼段後, 加載 LDT 並跳轉執行 LDT 中包含的程式碼段：

```
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    $(OffsetPMMessage), %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: 在保護模式程式碼段中加載 LDT 並跳轉執行 LDT 程式碼段(節自chapter3/2/loader.S)

在 LABEL_SEG_CODE32 中前幾行, 我們可以看到非常熟悉的匯編指令, 和一般匯編程序開頭初始化數據/程式碼/堆堆疊段暫存器的指令非常像, 只不過這裡賦給幾個暫存器的參數都是段選擇子, 而不是一般的位址。該程式碼段剩下的內容和前面圖 3.14 中 LABEL_CODEA 一樣, 都是打印一個字符串, 只不過這裡選擇在第 10 行（屏幕左側中央）打印。

為了方便閱讀, 整個 loader.S 的程式碼附在圖 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 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 */
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    $(OffsetPMMessage), %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 3.18: chapter3/2/loader.S

3.3.4 生成鏡像並測試

使用與第 2.3.6 節完全相同的方法, 我們可以將程式碼編譯並將 LOADER.BIN 拷貝到鏡像文件中。利用最新的鏡像文件啓動 VirtualBox 我們得到圖 3.19 。

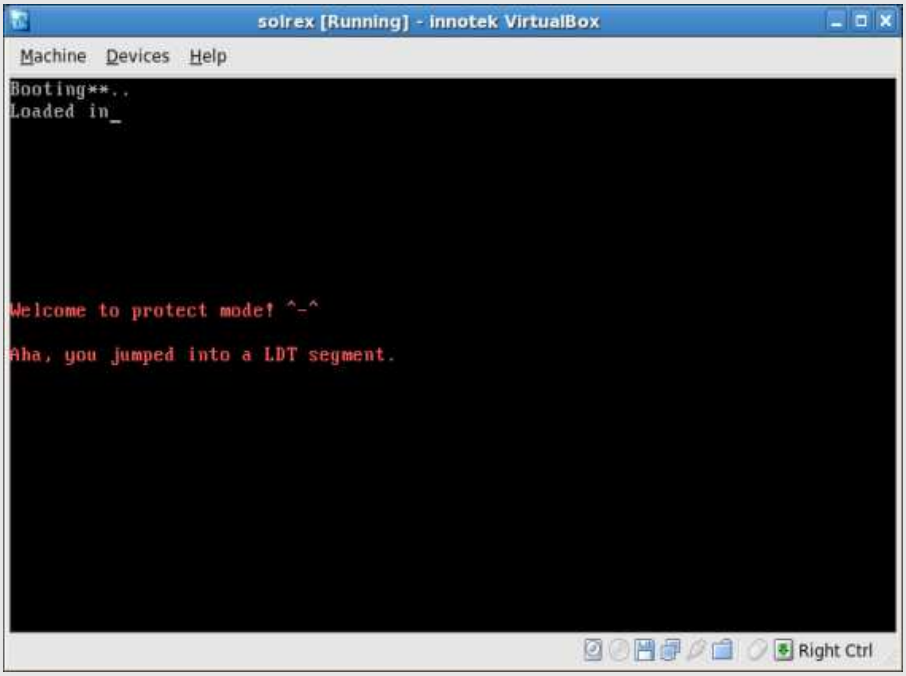


Fig 3.19: 第一次進入保護模式

可以看到, 該程序首先在屏幕左側中央（第 10 行）打印出來 `"Welcome to protect mode!^-^"`, 這是由 GDT 中的 32 位程式碼段 `LABEL_SEG_CODE32` 打印出來的, 標志著我們成功進入保護模式；然後在屏幕的第 12 行打印出來 `"Aha, you jumped into a LDT segment."`, 這個是由 LDT 中的 32 位程式碼段 `LABEL_CODEA` 打印出來的, 標志著 LDT 的使用正確。因為這兩個字符串都是被存儲在 32 位全局資料段中, 這兩個字符串的成功打印也説明在 GDT 中添加的資料段使用正確。

3.3.5 段式存儲總結

段式存儲和頁式存儲都是最流行的計算機記憶體保護方式。段式存儲的含義簡單來説就是先將記憶體分為各個段, 然後再分配給程序供不同用途使用, 並保證對各個段的訪問互不干擾。x86 主要使用段暫存器（得到的段基址） + 偏移量來訪問段中數據, 也簡化了尋址過程。

在 x86 的初期真實模式下就使用著非常簡單的段式存儲方式, 如圖 3.1a 所示, 這種模式下分段主要是為了方便尋址和隔離, 沒有提供其它的保護機制。x86 保護模式下採用了更高級的段式存儲方式：用全局和局部描述符表存儲段描述符信息, 使用段選擇子表示各個段描述符, 如圖 3.1b 所示。

由于保護模式使用段描述符來保存段信息而不是像真實模式一樣直接使用段位址, 在段描述符中就可以添加一些屬性來限制對段的訪問權限, 如我們在第 3.3.2 節中討論的那樣。這樣, 通過在訪問段時檢查權限和屬性, 就能做到對程序段的更完善保護和更好的記憶體管理。

x86 使用全局描述符表（GDT）和局部描述符表（LDT）來實現不同需求下對程序段的控制, 作業系統使用唯一的一個 GDT 來維護一些和系統密切相關的段描述符信息, 為不同的任務使用不同的 LDT 來實現對多任務記憶體管理的支持, 簡化了任務切換引起的記憶體切換的難度。

3.4 權限

如果您需要更詳細的知識, 也許您更願意去讀Intel 的手冊, 本節內容主要集中在：Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, 第 4 章.

權限是為了保護處理機資源而引入的概念。將同一個處理機上執行的不同任務賦予不同的權限, 可以控制該任務可以訪問的資源, 比如記憶體位址範圍、輸入輸出端口、和一些特殊指令的使用。在 x86 體系結構中, 共有 4 個權限別, 0 代表最高權限, 3 代表最低權限。由于在 x86 體系結構中, n 級可以訪問的資源均可以被 0 到 n 級訪問, 這個模式被稱作 ring 模式, 相應地我們也將 x86 的對應權限稱作 ring n。

現代的 PC 作業系統的核心一般工作在 ring 0 下, 擁有最高的權限, 應用程序一般工作在 ring 3 下, 擁有最低的權限。雖然 x86 體系結構提供了 4 個權限, 但作業系統並不需要全部使用到這 4 個級別, 可以根據需要來選擇使用幾個權限。比如 Linux/Unix 和 Windows NT , 都是只使用了 0 級和 3 級, 分別用于核心模式和用戶模式；而 DOS 則只使用了 0 級。

為了實施對程式碼段和資料段的權限檢驗, x86 處理機引入了以下三種權限類型（請注意這裡提到的權限高低均為實際高低, 而非數值意義上的高低）：

- **CPL(Current Privilege Level)**：當前權限, 存儲在 CS 和 SS 的 0, 1 位。它代表當前執行程序或任務的權限, 通常情況下與當前執行指令所在程式碼段的 DPL 相同。當程序跳轉到不同權限的程式碼段時, CPL 會隨之修改。當訪問一致程式碼段（Conforming Code Segment）時, 對 CPL 的處理有些不同。一致程式碼段可以被不高于（數值上大于等于）該段 DPL 的權限程式碼訪問, 但是, CPL 在訪問一致程式碼段時不會跟隨 DPL 的變化而更改。
- **DPL(Descriptor Privilege Level)**：描述符權限, 定義于段描述符或門描述符中的 DPL 域（見圖 3.2）, 它限制了可以訪問此段資源的權限別。根據被訪問的段或者門的不同, DPL 的意義也不同：
  - **資料段**：資料段的 DPL 限制了可以訪問該資料段的最低權限。假如資料段的 DPL 為 1, 那麼只有 CPL 為 0,1 的程序才能訪問該資料段。
  - **非一致程式碼段（不使用 call gate ）**：非一致程式碼段就是一般的程式碼段, 它的 DPL 表示可以訪問該段的權限, 程序或者任務的權限必須與該段的 DPL 完全相同才可以訪問該段。
  - **call gate** ： call gate 的 DPL 限制了可以訪問該門的最低權限, 與資料段 DPL 的意思一樣。

  - **一致程式碼段和使用 call gate 訪問的非一致程式碼段**：這種程式碼段的 DPL 表示可以訪問該段的最高權限。假如一致程式碼段的 DPL 是 2, 那麼 CPL 為 0,1 的程序就無法訪問該段。
  - **TSS(Task State Segment)**：任務狀態段的 DPL 表示可以訪問該段的最低權限, 與資料段 DPL 的意思一樣。

- **RPL(Requested Privilege Level)**：請求權限, 定義于段選擇子的 RPL 域中（見圖 3.12）。它限制了這個選擇子可訪問資源的最高權限。比如一個段選擇子的 RPL 為 2 , 那麼使用這個段選擇子只能訪問 DPL 為 2 或者 3 的段, 即使使用這個段選擇子的程序當前權限（CPL）為 0 。就是說, $\max(CPL, RPL) \leq DPL$ 才被允許訪問該段, 即當 CPL 小于 RPL 時, RPL 起決定性作用, 反之亦然。使用 RPL 可以避免權限高的程序代替應用程序訪問該應用程序無權訪問的段。比如在系統呼叫時, 應用程序呼叫系統過程, 雖然系統過程的權限高（$CPL = 0$）, 但是被呼叫的系統過程仍然無法訪問權限高于應用程序的段（$DPL < RPL = 3$）, 就避免了可能出現的安全問題。

在將段描述符對應的段選擇子加載到段暫存器時, 處理機通過將 CPL, 段選擇子的 RPL 和該段的 DPL 相比較, 來判斷程序是否有權訪問另外一個段。如果 $CPL > \max(RPL, DPL)$, 或者$\max(CPL, RPL) > DPL$, 那麼該訪問就是不合法的, 處理機就會產生一個常規保護異常（#GP, General Protection Exception）。

3.4.1 不合法的訪問請求示例

我們來看一個不合法的訪問請求的例子, 在上一節的 loader.S 中把 LABEL_DESC_DATA 對應的描述符的 DPL 設置為 1, 然後將該資料段對應的段選擇子的 RPL 設置為 3, 即修改以下兩行：

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

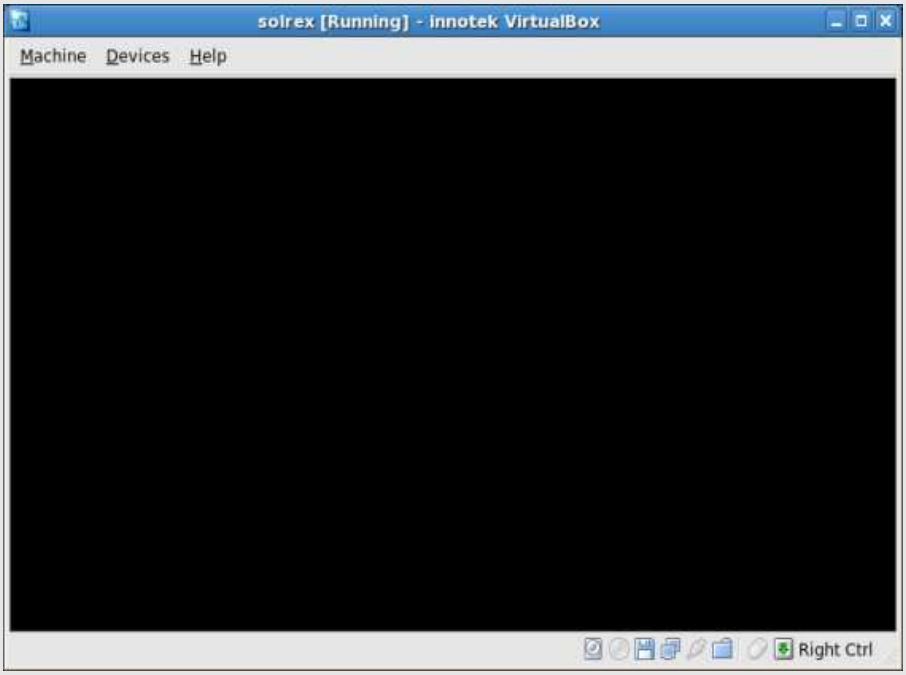


Fig 3.20: 虛擬機出現異常, 黑屏

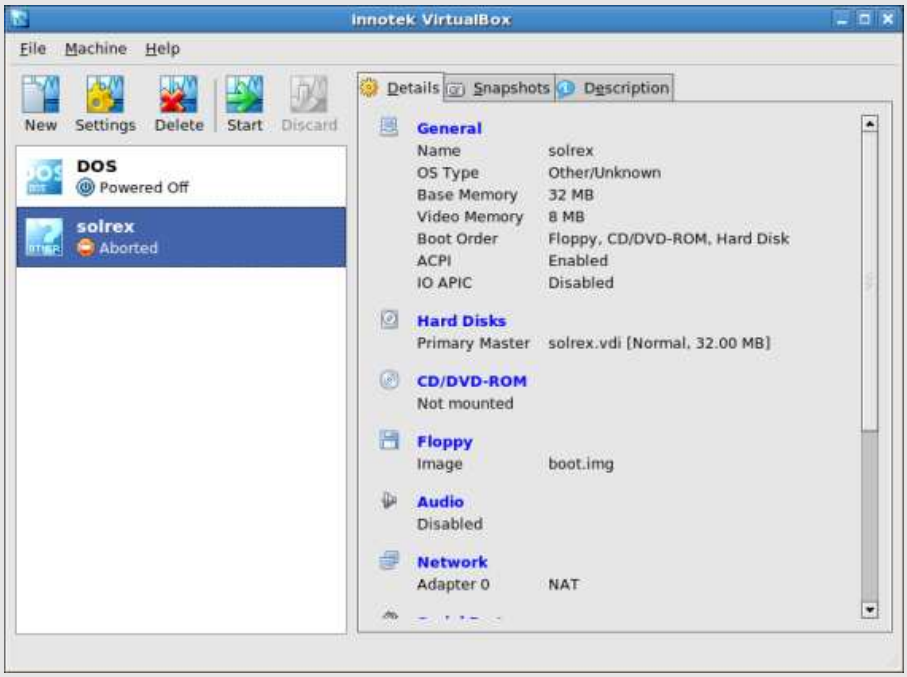


Fig 3.21: 虛擬退出後 VBox 主窗口顯示 Abort

再 `make, sudo make copy`, 用 VirtualBox 加載生成的鏡像運行一下, 就會發現虛擬機黑屏一會兒就會退出（如圖 3.20）, 然後 VirtualBox 主窗口中顯示該虛擬機 Aborted（如圖 3.21）。這是因為我們違反權限的規則, 使用 RPL=3 的選擇子去訪問 DPL=1 的段, 這個不合法的訪問請求引起處理機產生常規保護異常（#GP）。而我們又沒有準備對應的異常處理模塊, 當處理機找不到異常處理程序時就只好退出了。

3.4.2 控制權轉移的權限檢查

在將控制權從一個程式碼段轉移到另一個程式碼段之前, 目標程式碼段的段選擇子必須被加載到 CS 中。處理器在這個過程中會查看目標程式碼段的段描述符以及對其界限、類型（見圖 3.2）和權限進行檢查。如果沒有錯誤發生, CS 暫存器會被加載, 程序控制權被轉移到新的程式碼段, 從 EIP 指示的位置開始運行。

JMP, CALL, RET, SYSENTER, SYSEXIT, INT n 和 IRET 這些指令, 以及中斷和異常機制都會引起程序控制權的轉移。

JMP 和 CALL 指令可以實現以下 4 種形式的轉移：

- 目標操作數包含目標程式碼段的段選擇子。
- 目標操作數指向一個包含目標程式碼段段選擇子的 call gate 描述符。
- 目標操作數指向一個包含目標程式碼段段選擇子的任務狀態段。
- 目標操作數指向一個任務門, 這個任務門指向一個包含目標程式碼段段選擇子的任務狀態段。

下面兩個小節將描述前兩種轉移的實現方法, 後兩種控制權轉移方法我們將在用到時再進行解釋。

3.4.2.1 用 JMP 或 CALL 直接轉移

用 JMP, CALL 和 RET 指令在段內進行近跳轉並沒有權限的變化, 所以對這類轉移是不進行權限檢查的；用 JMP, CALL 和 RET 在段間進行遠跳轉涉及到其它程式碼段, 所以要進行權限檢查。

對不通過 call gate 的直接轉移來說, 又分為兩種情形：

- 訪問非一致程式碼段：當目標是非一致程式碼段時（目標段段描述符的 C flag 為 0 , 見圖 3.1）, 權限檢查要求呼叫者的 CPL 與目標程式碼段的 DPL 相等, 而且呼叫者使用的目標程式碼段段選擇子的 RPL 必須小于等于目標程式碼段的 DPL。我們之前的程式碼都屬于這種情形, 其中 $CPL = DPL = RPL = 0$。
- 訪問一致程式碼段：當目標是一致程式碼段時（目標段段描述符的 C flag 為 1 , 見圖 3.1）, 權限檢查要求 $CPL \geq DPL$, RPL 不被檢查, 而且轉移時並不修改 CPL。

總的來說, 通過 JMP 和 CALL 實行的都是一般的轉移, 最多從低權限轉移到高權限的一致程式碼段, CPL 總是不變的。

3.4.3 使用 call gate 轉移

call gate 是 x86 體系結構下用來控制程序在不同權限間轉移的一種機制。它的目的是使低權限的程式碼能夠呼叫高權限的程式碼, 這一機制在使用了記憶體保護和權限機制的現代作業系統中非常有用, 因為它允許應用程序在作業系統控制下安全地呼叫核心例程或者系統接口。

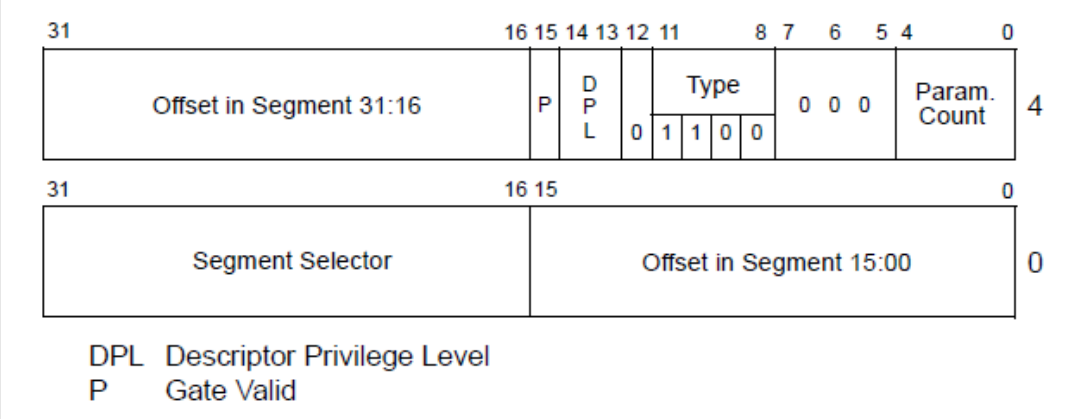


Fig 3.22: call gate 描述符

門其實也是一種描述符, 和段描述符類似。 call gate 描述符的資料結構如圖 3.22 所示。其實看起來這個 call gate 描述符的資料結構要比段描述符簡單一些, 至少從它的屬性來說, 沒有段描述符多。我們仍然只關注最重要的部分：首先是段選擇子（Segment Selector）, 指定了通過這個 call gate 訪問的程式碼段；其次是段偏移量（Offset in Segment）, 指定了要訪問程式碼段中的某個入口偏移；描述符權限（DPL）, 代表此門描述符的權限；P, 代表此 call gate 是否可用；參數計數（Param. Count）記錄了如果發生堆疊切換的話, 有多少個選項參數會在堆疊間拷貝。

簡單來說, call gate 描述了由一個段選擇子和一個偏移所指定的目標程式碼段中的一個位址, 程序通過 call gate 將轉移到這個位址。下面我們通過一個簡單的例子來介紹一下 call gate 的基本使用方法。

3.4.3.1 簡單的 call gate 轉移舉例

為了使用 call gate , 我們首先要給出一個目標段, 然後用該目標段的信息初始化 call gate 的門描述符, 最後用 call gate 的門選擇子實現門呼叫。

添加一個目標段我們已經做過很多次, 非常簡單。首先在上一節 loader.S 最後添加一個打印一個字符的程式碼段 LABEL_SEG_CODECG, 接著將該段的段描述符 LABEL_DESC_CODECG 添加到 GDT 中, 然後為該段準備一個段選擇子 SelectorCodeCG, 最後加入初始化該段描述符的程式碼：

```
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 call gate 的目標段(節自chapter3/3/loader.S)

總的來看, LABEL_SEG_CODECG 指向的這個段和我們以前為了打印程序運行結果所使用的段沒有本質不同, 為了簡單起見, 這裡我們僅僅讓它打印一個字符 'C' 就返回。

用目標程式碼段 LABEL_SEG_CODECG 的信息初始化 call gate 的門描述符 LABEL_CG_TEST, 以及門選擇子 SelectorCGTest。與匯編宏 Descriptor 類似, 我們這裡使用匯編宏 Gate 來初始化門描述符, 宏 Gate 的定義可以在頭文件 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: 匯編宏 Gate 定義(節自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: 設置 call gate 描述符及選擇子(節自chapter3/3/loader.S)

我們可以看到, 宏 Gate 的四個參數分別為：段選擇子、偏移量、參數計數和屬性, 它們在存儲空間中的分布與圖 3.22 中介紹相同。由于這個例子僅僅介紹 call gate 的簡單使用, 並不涉及權限切換, 所以也不發生堆疊切換, 這裡我們將參數計數設置為 0；門描述符的屬性為 (DA_386CGate + DPL) , 表明它是一個 call gate （屬性定義見圖 3.3）, DPL 為 0 , 與我們一直使用的權限相同；目標程式碼段選擇子是 SelectorCodeCG , 偏移是 0 , 所以如果該 call gate 被呼叫, 將轉移到目標程式碼段的開頭, 即 LABEL_SEG_CODECG 處開始執行。

使用遠呼叫 lcall 指令呼叫該 call gate 的門選擇子 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: call gate 選擇子(節自chapter3/3/loader.S)

由于對 call gate 的呼叫往往涉及到段間轉移, 所以我們通常使用 gas 的 lcall 遠跳轉指令和 lret 遠返回指令進行呼叫和返回。

這樣我們就完成了使用 call gate 進行簡單控制權轉移的程式碼, make, sudo make copy 之後, 用 VBox 虛擬機載入生成的鏡像, 運行結果如圖 3.27 所示。由于我們僅僅是在加載 LDT 之前添加了一個門呼叫, 而且門呼叫的目標段在屏幕的第 11 行第 0 列打印了一個 'C' 後就返回到了呼叫處, 所以加載 LDT 的程式碼繼續運行, 就是圖中所示結果。

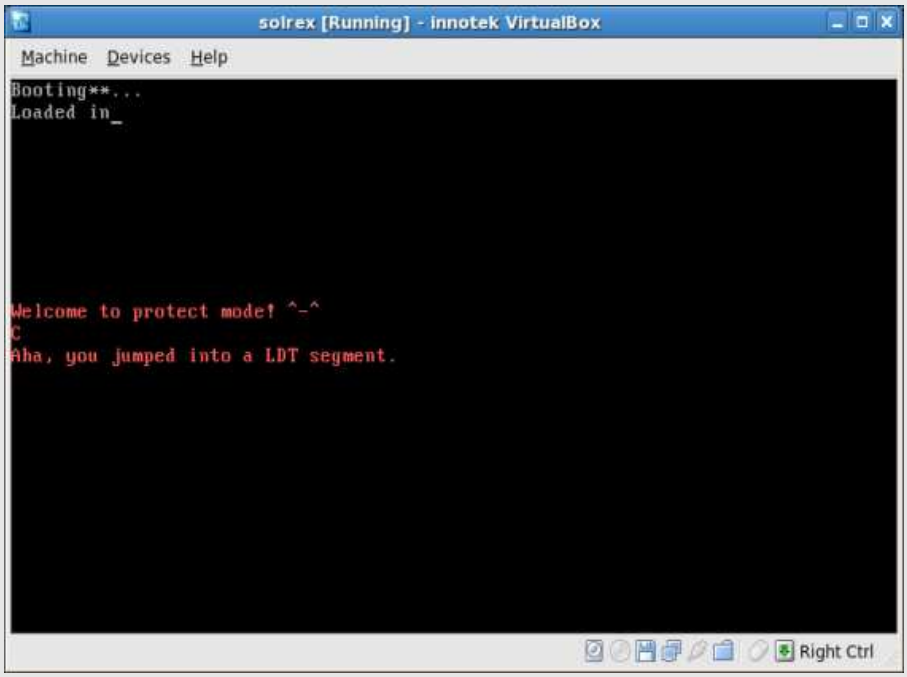


Fig 3.27: 使用 call gate 進行簡單的控制權轉移

3.4.3.2 涉及權限變化的 call gate 轉移

在上面例子中我們只是使用 call gate 取代了傳統的直接跳轉方式, 並沒有涉及到權限的變化。顯然 call gate 不是用來做這種自找麻煩的事情的, 其設計的主要目的是實現從低權限程式碼跳轉到高權限的非一致程式碼的功能。

在使用 call gate 進行轉移時, 處理機會使用四個權限值來檢查控制權的轉移是否合法：

1. CPL：當前權限；
2. RPL： call gate 選擇子的請求權限；
3. DPL： call gate 描述符的描述符權限；
4. DPL：目標程式碼段的段描述符權限。

使用 CALL 或者 JMP 指令訪問 call gate 進行控制權轉移時, 權限檢查的規則有所不同, 如表 3.3 所示：

表 3.3: call gate 權限檢查規則

指令	權限檢查規則
CALL	CPL $\leq$ call gate DPL; RPL $\leq$ call gate DPL 目標段DPL $\leq$ CPL
JMP	CPL $\leq$ call gate DPL; RPL $\leq$ call gate DPL 當目標段是一致程式碼段時：目標段 DPL $\leq$ CPL 當目標段是非一致程式碼段時：目標段 DPL = CPL

這張表內容雖然不多, 但也不容易很快理解。這裡我們應該著重看目標段 DPL 和 CPL 的比較, 這才是 call gate 權限檢驗的特點所在。總的來說, 使用 call gate 需要目標段的 DPL 小于或等于 CPL, 意思

那麼執行 CALL 指令時處理機都進行了哪些工作呢？這是一個非常復雜的問題, 我們將留待下個小節再詳細介紹。但是在我們的例子裡（即最簡單的情況下）, CALL 所做的就是將 SS, ESP, CS, EIP 這四個暫存器的值順序壓到堆疊裡, 這樣在 RET 指令執行的時候, 處理機從堆堆疊 pop 出 EIP, CS, ESP, SS 的值來恢復 CALL 指令執行後的處理機現場。所以為了使 RET 跳轉到我們想要執行的程式碼段, 我們只需要手動將 ring 3 目標程式碼段的對應的 SS, ESP, CS, EIP 值壓到堆疊裡即可。

下面開始準備這個 demo, 仍舊是在上一節程式碼的基礎上進行添加。首先, 我們準備一個 ring 3 目標程式碼段和新堆疊：

```
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: 要運行在 ring 3 下的程式碼段(節自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: 為 ring 3 程式碼段準備的新堆疊(節自chapter3/4/loader.S)

這個程式碼段的功能就是在第 11 行第 1 列打印一個 3 字。

其次, 添加新段的描述符和選擇子, 並添加初始化程式碼：

```
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: 為 ring 3 程式碼段和堆堆疊段添加的描述符和選擇子(節自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 程式碼段和堆堆疊段描述符的程式碼(節自chapter3/4/loader.S)

我們注意到, 其實 ring 3 下的程式碼段和 ring 0 下的程式碼段的程式碼部分是沒有任何區別的, 區別在于它們的程式碼段描述符和選擇子中所標明的權限。我們將 ring 3 下的目標程式碼段和堆堆疊段的段描述符屬性中加上 DA_DPL3 , 表明它們的段描述符權限均為 3 ；在它們的段選擇子屬性中加上 SA_RPL3 , 表明它們的段選擇子請求權限均為 3 。以上這兩點限制了這個段只能在 ring 3 下運行。

準備好了兩個段, 我們將這兩個段的信息作為 SS, ESP, CS, EIP 的內容依次壓堆疊, 然後再執行 lret 長返回指令, 就能跳轉到 ring 3 下運行目標程式碼段了：

```
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 指令進行實際的跳轉

這樣, 我們就完成了從高權限程式碼（ring 0）跳轉到低權限程式碼（ring 3）的過程。像往常那樣使用 make, sudo make copy 編譯, 用虛擬機加載鏡像運行結果如圖 3.33 所示, 程序在屏幕的第 11 行第 1 列打印出了一個紅色的 '3' 字, 然後進入死循環。

上面這個例子僅僅演示了如何從高權限跳轉到低權限, 而沒有介紹如何轉移回高權限。直觀來講, 我們只需將圖 3.28 中的最後一條 JMP 指令替換成一條 CALL call gate 的指令即可。但是我們會看到, 帶有權限轉換的 call gate 轉移並不是那麼容易實現的。

下面我們來嘗試一下直接訪問 call gate , 首先將門描述符的 DPL 改為 3 以滿足在 ring 3 程式碼段訪問 call gate 的條件。

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

然後將圖 3.28 中最後一條 JMP 指令替換成對 call gate 的 CALL 指令：

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

然後編譯運行一下, 看看能否得到我們想要的結果？答案是否定的, 我們看不到屏幕上打印出 'C' 字, 得到與圖 3.21 相同的結果。為什麼呢？主要是因為在 CALL call gate 的時候發生了程序堆疊的切換, 由于這個堆疊切換發生在 CALL 指令執行的過程中, CALL 指令要訪問特殊的結構來得到新的堆疊信息, 不像 RET 指令僅僅讀取我們設置好的堆疊, 如果訪問出錯就會產生一個異常, 處理機無法處理這個異常就只好退出。我們下面一小節介紹在 CALL call gate 和 RET 的時候究竟發生了什麼事？需要什麼特殊結構的輔助才能實現帶有權限切換的 call gate 轉移？

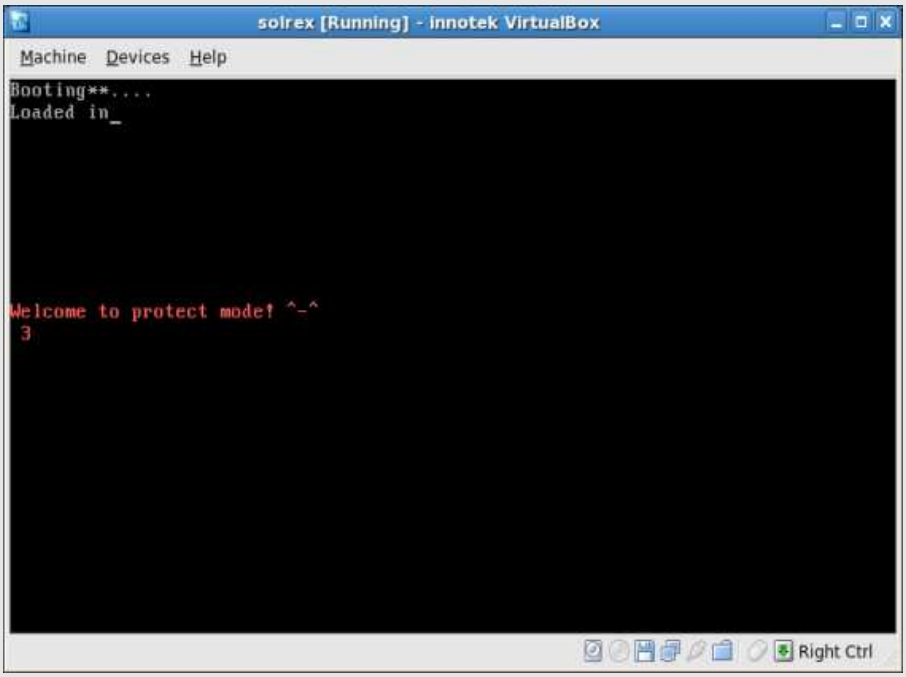


Fig 3.33: hack RET 實現從高權限到低權限的跳轉

3.4.4 堆疊切換和 TSS

當使用 call gate 轉移到不同權限下的非一致程式碼段時, 處理機總會自動的切換到目標權限對應的堆疊。堆疊切換是為了避免高權限的堆疊空間被濫用導致堆疊空間不足而崩潰, 以及低權限的程序非法修改或者干擾高權限程序的堆疊內容。

為了使堆疊切換能夠成功, 我們必須為任務中用到的每個權限都定義一個獨立的堆疊。在上一小節最後一個例子中, 我們實際已經定義了兩個堆堆疊段： `LABEL_STACK` 和 `LABEL_STACKR3`, 分別被 ring 0 和 ring 3 使用, 並且已經實現了從高權限轉移到低權限程序時的堆疊切換。從上面例子中我們看出, 使用 RET 指令時, 處理機從堆堆疊中得到目標權限的堆堆疊段選擇子（`SelectorStackR3`）和堆疊頂指標（`TopOfStackR3`）, 然後分別存儲到 SS 和 ESP 暫存器中, 就實現了堆疊切換。RET 指令能如此直接地實現堆疊切換的關鍵是 CALL 指令已經將呼叫者所在權限的堆疊信息壓到被呼叫者所在權限的堆疊裡。

但是在使用 CALL 指令時, 處理機不能從堆疊上獲得被呼叫者所在權限的堆疊信息, 就需要一個輔助的資料結構來幫助它獲得這些信息, 這個資料結構就是 TSS(Task-State Segment, 任務狀態段) 。當前任務的 TSS 中存儲著指向 0, 1, 2 權限堆疊的指標, 這個指標指向的內容包括一個堆堆疊段選擇子和堆疊頂指標, 如圖 3.34 中 SS* 和 ESP* 所示。TSS 的結構中並不包含 ring 3 的堆疊信息, 這是因為在使用 call gate 時, 不可能發生 ring 0, 1, 2 的程式碼通過 CALL 跳轉到 ring 3 的情況。這種情況只會發生在 RET 返回的時候, 而這時候我們只需要將 ring 3 的堆疊信息壓到堆疊裡就能實現跳轉（就像我們上一小節做的那樣）。

當 TSS 被加載後, 這些指向的和堆疊相關的內容會被嚴格限制為只讀, 處理機在運行過程中不會修改它們。在一個跨權限的 CALL 呼叫結束返回時, TSS 中的堆疊信息不會被修改, 所有被呼叫者堆堆疊的改變都會恢復原狀。在下一次 CALL 呼叫時, 處理機讀取的堆疊信息與前一次沒有任何不同。為每個

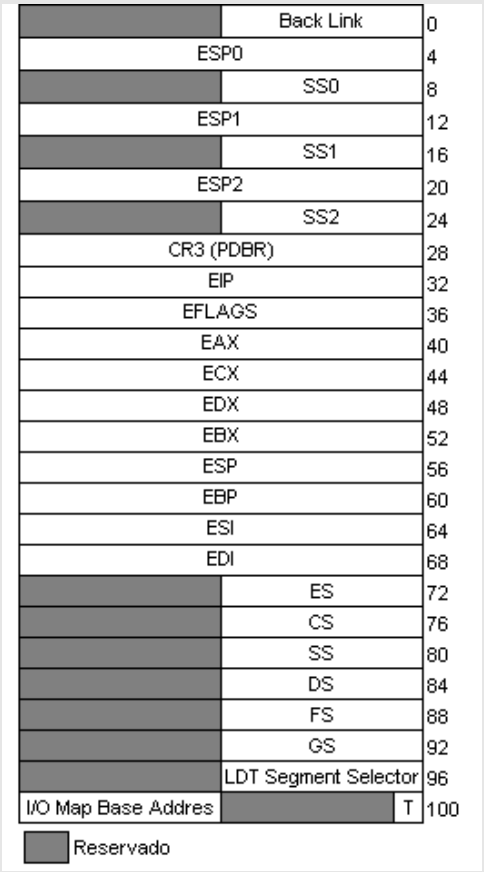


Fig 3.34: 32 位 TSS 資料結構

權限準備的堆堆疊段空間必須足夠容納可能的多次呼叫產生的多重堆疊幀。

由于在跨權限的呼叫過程中發生了堆疊切換, 在被呼叫者所在權限無法訪問呼叫者的堆疊, 所以呼叫者堆疊中的一些參數、返回位址等信息需要拷貝到被呼叫者堆疊中。有多少個參數需要被拷貝就是圖 3.22 中 Param. Count 域定義的。

當跨權限的呼叫發生時, 處理機要進行下列操作來切換堆堆疊和切換到目標權限運行：

1. 依據目標段的 DPL（新的 CPL）從 TSS 中選擇新堆堆疊的指標（堆堆疊段選擇子和堆疊頂指標）；
2. 從當前任務的 TSS 中讀取堆堆疊段選擇子（SS*）和堆疊頂指標（ESP*）。在讀取堆堆疊段選擇子、堆疊頂指標或者堆堆疊段描述符時發生的任何錯誤, 都會使處理機產生一個 #TS（非法 TSS）異常。
3. 對堆堆疊段描述符進行權限和類型檢驗, 如果通不過檢驗處理機同樣產生 #TS 異常；
4. 暫時保存當前 SS 和 ESP 暫存器中存儲的堆堆疊段選擇子和堆疊頂指標；
5. 加載新的堆堆疊段選擇子和堆疊頂指標到 SS 和 ESP中；
6. 把第 4 步保存的 SS 和 ESP 壓入新堆疊（被呼叫者堆疊）；

7. 從舊堆疊（呼叫者堆疊）中復制 call gate 描述符中 Param. Count 個參數到新堆疊中。如果 PCount 為 0 , 什麼都不復制；
8. 將返回位址指標（當前 CS 和 EIP 暫存器中的內容）壓到新堆疊中；
9. 加載 call gate 中指定的段選擇子和指令指標到 CS 和 EIP 暫存器中, 開始執行被呼叫過程。

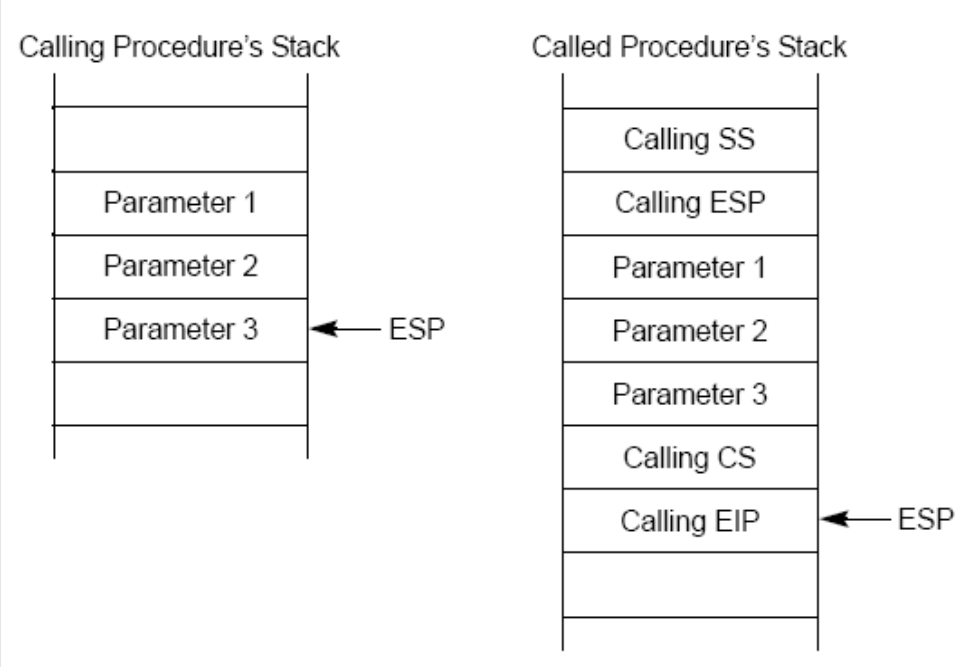


Fig 3.35: 跨權限呼叫時的堆疊切換

其中的堆疊切換過程如圖 3.35 所示。需要注意的是, call gate 描述符中的 PCount 域僅有 5 位, 就是說在堆疊切換過程中最多只能復制 31 個參數。如果需要傳遞更多參數, 可以傳遞一個指向某個資料結構的指標, 或者通過堆疊中保存的呼叫者堆疊信息來訪問舊堆疊中的參數。

既然跨權限的呼叫那麼麻煩, 跨權限的返回也不會很簡單。在跨權限的返回指令執行時, 處理機的工作包含以下步驟：

1. 檢查保存在堆疊上的 CS 暫存器中的 RPL 域看返回時是否需要切換權限；
2. 加載保存在堆疊上的 CS 和 EIP 暫存器信息（同時對段描述符和段選擇子的權限和類型進行檢驗）；
3. 如果 RET 指令有參數計數操作數, 增加 ESP 的值跳過堆疊上的參數部分, 此時 ESP 將指向保存的呼叫者的 SS 和 ESP。RET 的參數計數操作數應與 call gate 中描述符中的 PCount 域相同;
4. 加載 SS 和 ESP, 切換到呼叫者堆堆疊, 被呼叫者的 SS 和 ESP 將被丟棄（此時會進行堆堆疊段選擇子、堆疊頂指標和堆堆疊段描述符的權限和類型檢驗）；
5. 如果 RET 指令有參數計數操作數, 增加 ESP 的值跳過堆疊上（呼叫者堆疊）的參數部分；

6. 檢查 DS, ES, FS 和 GS 暫存器的內容, 如果某個暫存器指向的段的 DPL 小于當前權限 CPL（此規則不適用于一致程式碼段）, 該暫存器將加載一個空描述符。

在返回時堆疊切換的操作可視為呼叫時堆疊操作的逆過程。

至此, 我們已經完整地描述了跨權限的呼叫和返回過程, 下面就來實現一段跨權限呼叫的示例程序。

在第 3.4.3 的最後, 我們嘗試直接訪問 call gate 失敗, 主要原因是沒有準備 TSS 段, 因此我們只需要在其基礎上準備好 TSS 段並將其段選擇子加載到 TR 暫存器中即可。

首先, 準備好 TSS 段, 及其段描述符和選擇子, 並加入初始化 TSS 段描述符的程式碼：

```
80 LABEL_TSS:
81     .4byte  0           /* Back Link */
82     .4byte  TopOfStack  /* ESP0 */
83     .4byte  SelectorStack /* SS0 */
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 段內容及其描述符和選擇子和初始化程式碼(節自chapter3/5/loader.S)

進入 ring 3 之前將 TSS 段選擇子加載到 TR 暫存器中, 這樣在 ring 3 中訪問 call gate 就可以實現跳轉了。

```
227 CODE32.2:
228
229     mov     $(SelectorTSS), %ax     /* Load TSS to TR register */
230     ltr     %ax
231
232     pushl   $(SelectorStackR3)      /* Fake call procedure. */
```

Fig 3.37: 加載 TSS 段選擇子到 TR 暫存器(節自chapter3/5/loader.S)

將得到的程式碼編譯成鏡像, 運行結果如圖 3.38 所示。我們可以看到這次在圖 3.33 的基礎上多打印了一個 'C' 字然後陷入了死循環, 說明了我們成功達到了 call gate 跨權限轉移的目的。這個示例程序實現了從 ring 3 程式碼段轉移到 ring 0 程式碼段, 然後再返回 ring 3 進入死循環。

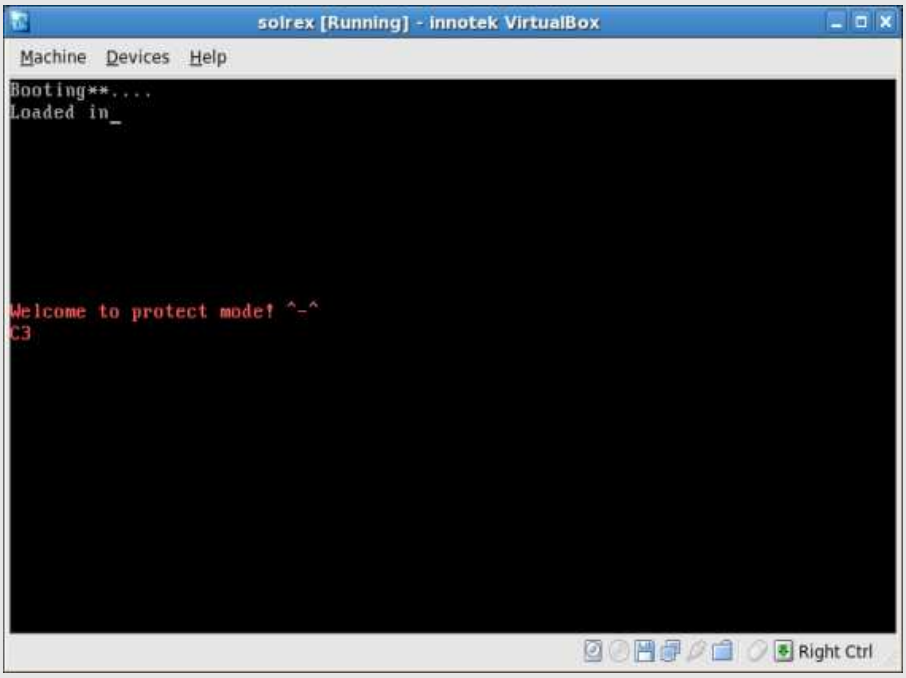


Fig 3.38: 跨權限的 call gate 轉移

為方便閱讀, 本節最終的 loader.S 全部程式碼如圖 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 /* SS0 */
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 CR0. */
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    $(OffsetPMMessage), %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 分頁式記憶體管理

保護模式一個非常重要的特性就是提供了虛擬記憶體機制。虛擬記憶體機制使每個應用程序都以為自己擁有完整連續的記憶體空間, 而事實上它可能是被分塊存放在物理記憶體甚至硬盤中。這極大地方便了大型應用程序的編程, 並且能更高效地利用物理記憶體。在具體實現上, 幾乎所有的體系結構都使用分頁機制作為基本方式。比如某應用程序需要使用 1GB 記憶體, 沒有虛擬記憶體機制的輔助, 在低于 1GB 記憶體的電腦上該程序就無法運行, 但是虛擬記憶體機制可以將該應用程序分成若幹“頁”, 僅僅將程序當前使用的頁調入物理記憶體中, 剩下的頁存放在硬盤上, 需要時再調入, 這就可以大大地減少物理記憶體的使用量。

我們這裡所說的“頁”是指一塊連續的虛擬記憶體空間, 一般情況下最小為 4K byte , 視體系結構可使用的虛擬記憶體大小的不同, 這個數字可以更大, 不過一般取 2 的整數次方。比如在 Intel 奔騰系列 CPU 中, 頁的大小可以取 4KB, 2MB 或者 4MB。

3.5.1 分頁機制

當 IA-32 處理機啓用分頁機制後, 程序的線性位址空間被分割成為固定大小的頁。在運行時, 這些頁被映射到計算機的物理記憶體或者硬盤中。當應用程序訪問某邏輯位址時, 處理機首先將邏輯位址轉換為線性位址, 再使用分頁機制將線性位址轉換為實際的物理位址。如果被訪問的包含該線性位址的頁面當前不在物理記憶體中時, 處理機會產生一個頁錯誤異常（ #PF, Page-Fault Exception）。此時作業系統的異常處理程序應該將該頁面調入物理記憶體中。

回憶圖 3.1b , 我們已經知道了在保護模式下處理機將邏輯位址轉換為線性位址的過程, 下面我們以 4KB 大小的頁面為例, 來看處理機是如何將線性位址轉換為最終的物理位址的。

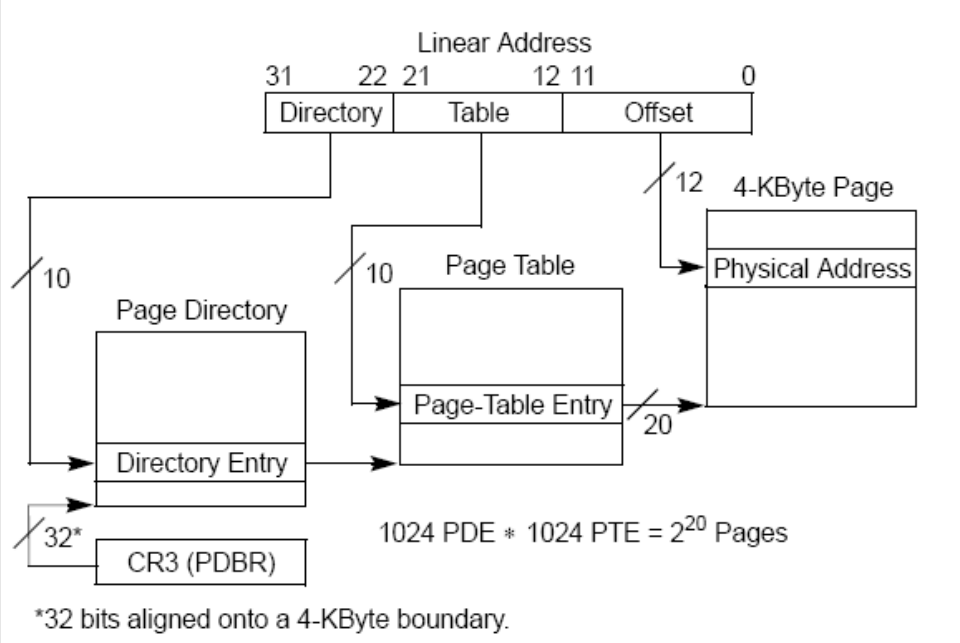


Fig 3.40: 線性位址轉換（4KB 頁）

在圖 3.40 裡：

- CR3：是儲存頁目錄基址的暫存器（Page Directory Base Register）。我們將設置好的頁目錄基址存放到 CR3 中, 處理機在轉換線性位址的時候就會去 CR3 中尋找頁目錄。
- Directory：線性位址的第 22 到 31 位存放的是該線性位址所在頁的頁表記錄在頁目錄中的偏移量, 處理機通過該偏移量得到頁目錄項, 進而得到該頁表的位址。
- Table：線性位址的第 12 到 21 位存放的是該線性位址所在頁記錄在頁表中的偏移量, 處理機通過該偏移量得到頁表項, 進而得到該頁位址。
- Offset：線性位址的第 0 到 11 位存放的是該線性位址在所屬頁表中的偏移量, 處理機通過該偏移量得到該線性位址對應的物理位址。

總的來說, 線性位址轉換為物理位址的過程十分簡單, 主要是使用兩層目錄結構。不就是查表嘛, 在我們日常生活中太經常遇到了, 我們可以類比一下。比如我們要尋找南京大學 7 舍 205 室（線性位址）, 就需要先找到一張地圖（通過 CR3 找到頁目錄）, 然後用地圖先找到南京大學（在頁目錄中尋找頁表）, 在南京大學裡尋找 7 舍（在頁表中尋找頁）, 進去 7 舍尋找 205 室（在頁中尋找偏移位址）, 這樣我們就找到了目標郵件位址（物理位址）。

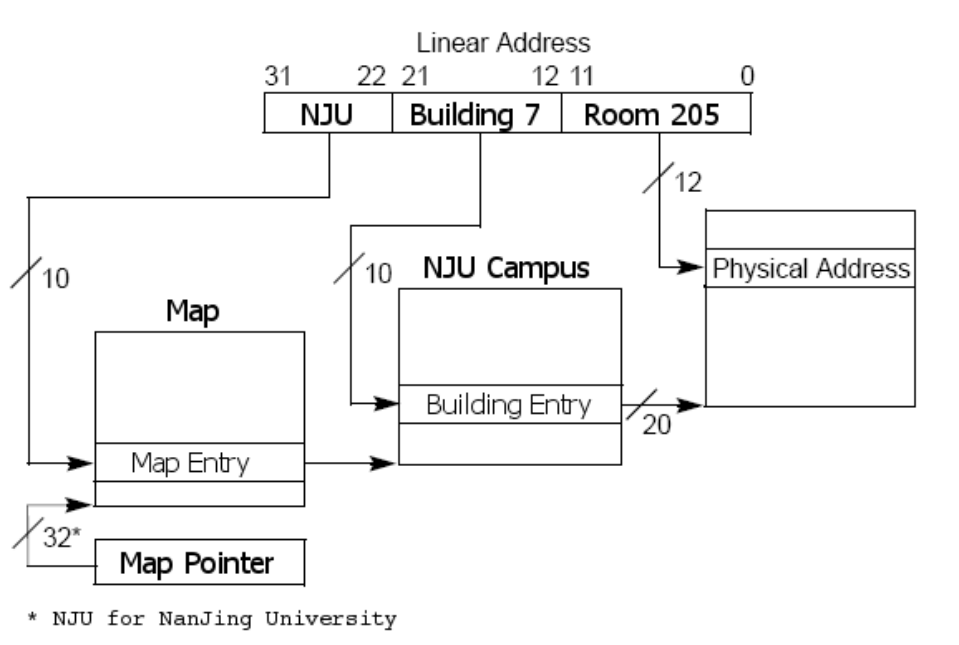


Fig 3.41: 郵件位址轉換

3.5.2 啓動分頁機制

既然已經知道 IA-32 的分頁機制, 那麼我們就可以嘗試啓動它了。但是在啓動分頁機制之前, 仍然有一個問題需要回答：頁目錄項（PDE, Page Directory Entry）、頁表項（PTE, Page Table Entry）和頁目錄指標（CR3）的資料結構是什麼？因為只有將頁目錄、頁表和 CR3 設置正確, 我們才能正確地啓動和使用分頁機制, 所以必須知道它們的資料結構。

3.5.2.1 PDE 和 PTE

圖 3.42 所示即為當頁大小為 4KB 時, 頁目錄項和頁表項的資料結構。由于這裡選項衆多, 下面我們僅解釋一下幾個重要的位的用途, 對其它位所代表的含義我們就不全部一一解釋了。如果您想了解更多內容, 請閱讀 Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide 第 3 章第 7 節的有關內容。

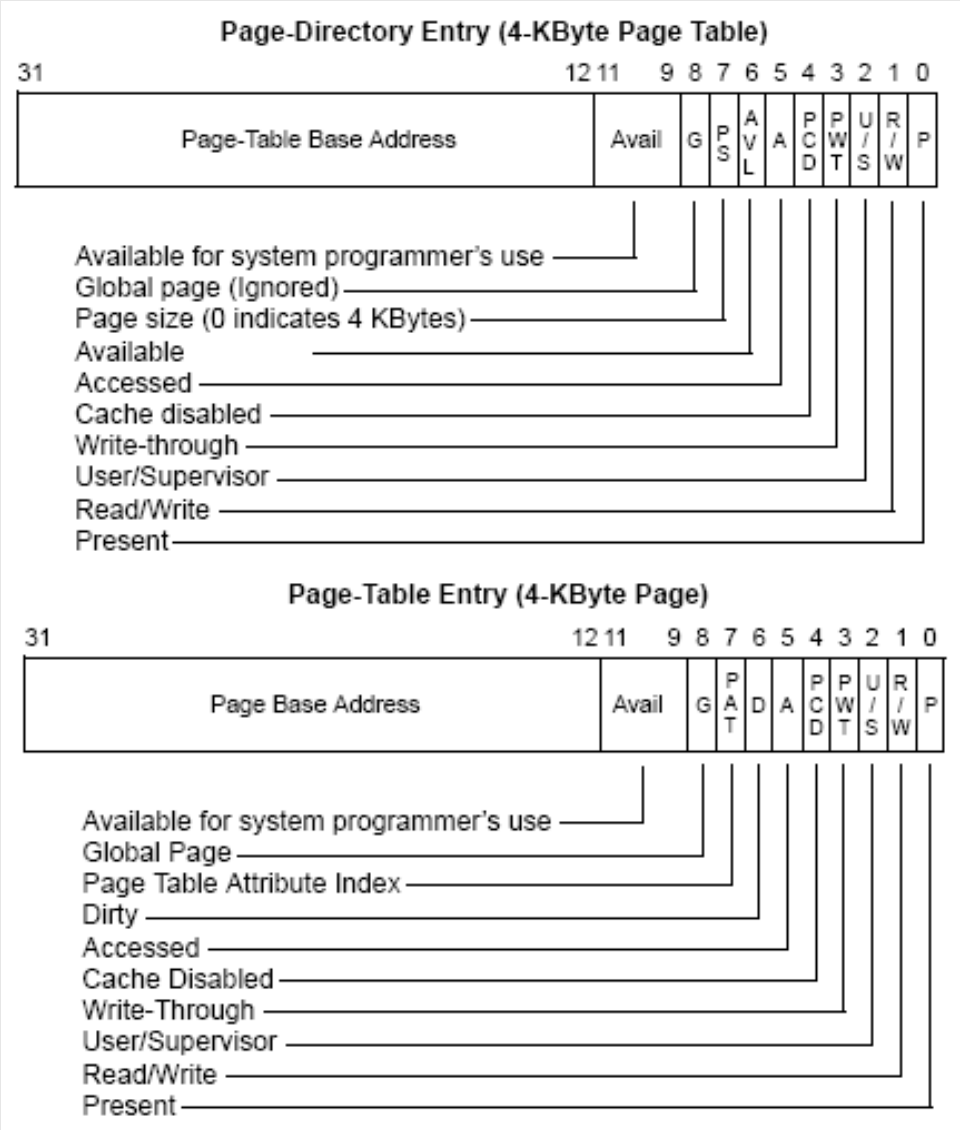


Fig 3.42: PDE 和 PTE 的資料結構（4KB 頁）

- Page(-Table) base address：均為 20 bit , 指向頁（表）的物理位址。由于這裡只有 20 bit , 所以要求頁（表）的物理位址要以 4KB 為邊界對齊（即低 12 位為 0）。因為每個頁表佔用 4KB 空間。每個頁表項佔用 4 個 byte , 所以每個頁表包含 1024 個頁表項（1024*4=4KB）。

- Present(P) flag：設置為 1 代表該頁表項指向的頁（或者該頁表項指向的頁表）存在于物理記憶體中, 反之則不存在于物理記憶體中。處理機如果訪問到該位為 0 的頁表項或者頁目錄項, 就會產生 #PF 異常。該位的值由作業系統設置, 處理機不會修改該位。
- Read/Write(RW) flag：設置為 0 表示該頁表項指向的頁（或者該頁目錄項指向的頁表中的頁）為只讀, 反之代表可讀寫。另外此位還與 U/S 位和 CR0 中的 WP 位相互作用, 若 WP 為 0 , 則即使 RW 為 0 , 系統級程序仍然具有寫權限。
- User/Supervisor(U/S) flag：設置為 0 代表該頁表項指向的頁（或者該頁目錄項指向的頁表中的頁）的權限為系統級（CPL=0, 1, 2）, 反之則為用戶級（CPL=3）。

與 PDE 和 PTE 類似, CR3 也是取頁目錄物理位址的高 20 bit 作為頁目錄的位址, 所以同樣要求頁目錄以 4KB 為邊界對齊, 因此每個頁目錄也包含 1024 個頁目錄項。CR3 的低 12 位中, 第 3, 4 位分別作為 PCD 和 PWD 標志, 與圖 3.42 中的 PCD 和 PWD 標志作用類似, 不再贅述。

3.5.2.2 開啓分頁機制示例程式碼

了解了頁目錄項、頁表項和 CR3 的資料結構之後, 我們就可以寫一個實驗性的程式碼來開啓分頁機制了。在上一節程式碼的基礎上, 首先我們來添加兩個段, 基址分別為 PageDirBase 和 PageTblBase , 用來保存頁目錄和頁表。

```
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*/

56 .set    SelectorTSS,    (LABEL_DESC_TSS - LABEL_GDT)
57 .set    SelectorPageDir,(LABEL_DESC_PAGEDIR - LABEL_GDT)
58 .set    SelectorPageTbl,(LABEL_DESC_PAGETBL - LABEL_GDT)
```

Fig 3.43: 添加保存頁目錄和頁表的段(節自chapter3/6/loader.S)

由于這兩個段的基位址是我們直接寫入的 magic number , 不用編譯時候計算, 所以這裡就不需要再初始化兩個段的描述符了。頁目錄有 1024 個目錄項, 故佔用 4KB 記憶體。

上文遇到的屬性值 DA_LIMIT_4K 和將要遇到的 PG_P 等屬性被定義在 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: 為分頁機制添加的新屬性(節自chapter3/6/pm.h)

然後添加一個函數, 根據上面介紹的資料結構初始化頁目錄和所有頁表, 然後打開分頁機制。由于我們只是嘗試打開分頁機制, 為了簡單起見, 這個函數僅僅是將所有線性位址映射到與其相同的物理位址上, 所以我們將有 1024 個頁表, 1024*1024 個頁。頁表項所佔空間將為： 1024*1024*4 = 4MB。

```
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: 初始化頁目錄和頁表, 並打開分頁機制的函數(節自chapter3/6/loader.S)

在這個函數中, 首先我們初始化頁目錄, 共有 1024 個頁目錄項, 分別對應著 1024 個頁表。由于已知第一個頁表的基址為 PageTblBase , 故我們只需每次增加一個頁表的大小（1024*4 = 4096）就得到下一個頁表的基址, 把這些基址賦給每個頁目錄項, 循環 1024 次就能建立所有頁目錄項。在建立頁目錄項的時候需要加上屬性（flags）, 由于我們每次增加的大小為 4096（後 12 位都是 0）, 所以在 PageTblBase 或上屬性 PG_P | PG_USU | PG_RWW 後, 加上 4096 並不影響這些屬性, 後面的頁目錄項也自動獲得了這些屬性：存在記憶體中、用戶級別和可讀寫。

我們用同樣的方法初始化所有頁表。由于頁表是連續存儲的, 我們可以轉變初始化 1024 個頁表為建立 1024*1024 個連續的頁表項, 由于我們將所有線性位址映射到與其相同的物理位址, 所以第一個頁的基址為 0, 而每個頁的大小為 4KB , 故增加 4096 就得到下個頁的基址, 以此類推, 我們就可以建立起 1024^2 個頁表項, 對應于 1024^2 個頁。每個頁表項的屬性（flag）為：存在記憶體中、用戶級別和可讀寫。

其實這裡我們是犯了一個大錯誤的, 1024^2 個頁的大小共為 4GB, 而我們為虛擬機分配的記憶體僅僅有 32MB, 這些頁不可能同時存在于記憶體中, 所以我們不應該將所有頁目錄項和頁表項的 P 標志設置為 1。不過這僅僅是為了簡單起見而寫的一個示例程式碼, 而且我們的 loader 並沒有訪問高位址的記憶體, 所以也不會出現什麼嚴重的後果。

在初始化頁目錄和頁表之後, 我們就將頁目錄基址 PageDirBase 賦給 CR3 （這裡我們先忽略了 CR3 的 PCD 和 PWD 標志位）；然後將 CR0 的最高位設置為 1 , 就開啓了 IA-32 處理機的分頁機制。

最後, 在進入保護模式後, 馬上呼叫上面的函數打開分頁機制。

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

Fig 3.46: 進入保護模式後馬上打開分頁機制(節自chapter3/6/loader.S)

將以上程式碼編譯成鏡像文件, 用虛擬機運行之後, 我們發現結果與圖 3.38 完全相同, 好像程式碼根本沒有被修改過。這是因為我們僅僅是打開了分頁機制, 只是記憶體映射機制不同了, 而負責向屏幕上打印信息的程式碼則和上節完全一樣, 所以屏顯不會有任何變化。不過既然有屏顯, 就說明我們開啓分頁機制的程式碼成功了, 否則如果記憶體映射錯誤, 虛擬機的運行就會出現問題。

3.5.3 修正記憶體映射的錯誤

我們前面提到犯了一個大錯誤, 將線性位址映射到了不存在的物理位址上。那麼一個相對簡單的解決辦法是避免映射不存在的物理位址, 也就是說我們不需要將頁目錄和頁表中的條目全部用完。這樣還有另外一個好處就是減小了頁表所佔用的空間, 在上一節的例子中, 1024^2 個頁表共佔用了 4MB 的空間, 而我們給虛擬機總共才分配了 32MB 的記憶體, 這顯然是極大的浪費。

為了避免映射不存在的物理位址, 我們就需要知道機器的記憶體有多大。通常情況下我們使用功能碼為 E820h 的 15h 中斷來做這件事情。

3.5.3.1 INT 15h, EAX=E820h - 查詢記憶體分布圖

功能碼為 E820h 的 15h 中斷呼叫只能在真實模式下使用, 這個呼叫會返回所有安裝在主機上的 RAM,以及被 BIOS 所保留的物理記憶體範圍的記憶體分布。每次成功地呼叫這個接口都會返回一行物理位址信息, 其中包括一塊物理位址的範圍以及這段位址的類型（可否被操作系使用等）。

INT 15h, AX=E820h 和 INT 15h AH=88h 也可以返回記憶體分布信息, 但是在這些舊接口返回信息與 INT 15h, EAX=E820h 不同時, 應以 E820h 呼叫返回的信息為準。

表 3.4: INT 15h, EAX=E820h 中斷的輸入

EAX	功能碼	E820h
EBX	後續	包含為得到下一行物理位址的“後續值”（類似于鏈表中的 next 指標）。 如果是第一次呼叫, EBX 必須為 0。
ES:DI	緩衝區指標	指向 BIOS 將要填充的位址範圍描述符(Address Range Descriptor)結構。
ECX	緩衝區大小	緩衝區指標所指向的位址範圍描述符結構的大小, 以 byte 為單位。 無論 ES:DI 所指向的結構如何, BIOS 最多將會填充 ECX byte 。

EDX	簽名	BIOS 以及呼叫者應該最小支持 20 個 byte 長,未來的實現將會擴展此限制。'SMAP' - BIOS 將會使用此標志, 對呼叫者將要請求的記憶體分布信息進行校驗, 這些信息會被BIOS放置到ES:DI所指向的結構中。

表 3.5: INT 15h, EAX=E820h 中斷的輸出

CF	進位標志	不進位(0)表示沒有錯誤, 否則則存在錯誤。
EAX	簽名	'SMAP'
ES:DI	緩衝區指標	返回的位址範圍描述符結構指標, 和輸入值相同。
ECX	緩衝區大小	BIOS 填充到位址範圍描述符中的 byte 數量, 返回的最小值是 20 個 byte 。
EBX	後續	存放為得到下一個位址描述符所需要的"後續值", 這個值的實際形式倚賴于具體的 BIOS 實現。呼叫者不必關心它的具體形式, 只需要在下次迭代時將其原樣放置到 EBX 中, 就可以通過它獲取下一個位址範圍描述符。注意, BIOS 將返回後續值 0 並且無進位來表示已經得到最後一個位址範圍描述符。

上面兩表中所提到的位址範圍描述符(Address Range Descriptor Structure)結構為：

```
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 域可能的取值和意義有：

```
值      名稱                    描述
1       AddressRangeMemory      這段記憶體是~OS~可用的~RAM。
2       AddressRangeReserved    這段記憶體正在被~OS~使用或者被~OS~保留,~所以不可用。
Other   Undefined               未定義~-~保留給以後使用。任何~Other~值都應該被~OS~視為
```

有關 INT 15h, EAX=E820h 中斷的更多信息, 請參考 GNU GRUB 原作者 Erich Boleyn 在 GRUB 原始文檔中的一個頁面：http://www.uruk.org/orig-grub/mem64mb.html。

3.5.3.2 得到記憶體信息

現在我們就可以使用 INT 15h 中斷來獲得系統記憶體的分布信息了。簡單地來說, 就是將 INT 15h 返回的所有結果存儲在一塊記憶體區域中, 然後在程序中利用這些信息完成記憶體的分配。因此我們首先要在資料段中為這些信息開闢一段空間：

```
68 /* 32-bit global data segment. */
69 LABEL_DATA:
70 _PMMessage:     .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:\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    PMMessage,        (_PMMessage - 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: 用來儲存記憶體分布信息的資料段(節自chapter3/7/loader.S)

需要注意的一點是, 為了方便起見, 我們改變了前面章節的資料段符號的表示方法。以下劃線開頭的符號名字表示該符號只在真實模式中使用, 其對應的前面不帶下劃線的名字可以在保護模式中使用, 這樣比用前面加 Offset 的表達更清楚。

在上面這個資料段中, _AddMapBuf 就是存儲系統位址分布信息的緩衝區, 在這裡我們設置其為 256 個 byte 大小, 至多可以容納 12 個 20 byte 大小的位址範圍描述符結構體； _ARDStruct 即為位址範圍描述符結構體, 共有 5 個域； _MemSize 用來保存計算出的可用記憶體大小； _AMECount 記錄位址分布信息條數, 即共返回多少個位址範圍描述符結構； _CursorPos 是記錄當前光標位置的全局變量, 用來在不同函數中向屏幕連續打印信息。

下面我們就用 INT 15h 中斷將得到的位址分布數據存儲到緩衝區中：

```
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: 用中斷 INT 15h 得到位址分布數據(節自chapter3/7/loader.S)

得到位址分布數據的過程很簡單, 我們只需要設置好輸入的暫存器, 循環呼叫 INT 15h 中斷, 最終就能得到整個系統的位址分布信息, 這些信息會被存儲在 _AddMapBuf 緩衝區中, 位址分布信息的條數被記錄在 _AMECount 中。注意, 這段程式碼是運行在真實模式中, 故應該使用前面帶下劃線的符號來引用數據。

為了驗證得到的數據是否正確, 我們可以將得到的位址分布信息打印到屏幕上, 下面這個函數就是做的這個工作：

```
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                        /*     *q++ = *p;                           */
389     add     $4, %esi             /*     p++;                                 */
390     dec     %edx                 /*     j--;                                 */
391     cmp     $0, %edx
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: 將位址分布信息打印到屏幕上(節自chapter3/7/loader.S)

這個函數的流程我們已經在旁邊給出了 C 語言格式的注釋。簡單地來說, 它就是遍歷整個儲存位址分布信息的緩衝區 _AddMapBuf , 將緩衝區中的數據按照位址範圍描述符的格式打印出來, 並通過可用記憶體位址的最高值來計算可用記憶體的大小。

這個函數中使用了一些輔助的打印函數, 比如 DispInt, DispStr 等, 這些函數定義在 lib.h 頭文件中, 提供這樣一些庫函數能方便我們的編程：

```
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

因為我們已經得到了可用記憶體的大小, 就可以根據可用記憶體的大小來調整我們的記憶體映射範圍。

```
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: 根據可用記憶體大小調整記憶體映射範圍(節自chapter3/7/loader.S)

在 SetupPaging 函數中, 我們首先根據可用記憶體計算頁目錄項(PDE)的數目（即頁表的數目）。每個頁表可以包含 1024 個頁, 可以指向 4MB 大小的記憶體, 所以我們只需要用可用記憶體數除以 4MB , 並向上取整, 就能得到需要的頁目錄項數, 然後再根據頁目錄項數初始化頁目錄項和頁表項。

仔細看程式碼可以發現, 我們這裡仍然映射了一些不存在的物理位址。因為可用記憶體數除以 4MB 未必是整數, 但是我們計算頁表項數的時候使用的是 頁目錄項*1024(%ecx*1024), 那麼在最後一個頁表的末尾就可能存在一些頁映射到不可用的記憶體位址；並且從下文中屏幕打印的信息我們可以知道, 不是所有從 0 到 MemSize 的記憶體都是可用的, 而我們在 SetupPaging 函數中卻是以連續的方式初始化所有的頁, 那麼中間必然有一些頁映射到不可用的記憶體位址。但總的來說, 在新的 SetupPaging 函數中大大減少了頁表的數量, 比如當 MemSize = 32MB 的時候, 我們僅僅需要 8 個頁表, 而不是原來的 1024 個頁表, 因之所有頁表佔用的記憶體也從 4MB 減小到了 32KB。

至于那些映射錯誤, 既然我們已經獲得了整個系統位址分布的情況（存儲在 _AddMapBuf 中）, 只需要根據分布情況, 增加一些判斷來避免映射到不存在的記憶體位址即可, 這裡就不多做演示了。

下面就可以在程序中呼叫我們新增加的函數：

```
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: 呼叫顯示記憶體範圍和開啓分頁機制的函數(節自chapter3/7/loader.S)

將上面的程式碼編譯成鏡像, 用虛擬機加載執行的結果如圖 3.53 所示。在進入保護模式之後, 首先函數 DispAddrMap 打印出虛擬機系統的記憶體分布信息, 隨後是根據記憶體分布信息計算出來的記憶體

大小。接著 SetupPaging 函數根據記憶體大小開啓處理機的分頁機制。雖然我們無法直接看到分頁機制啓動的效果, 但是接下來程式碼的正常執行可以告訴我們分頁機制的開啓是成功的。

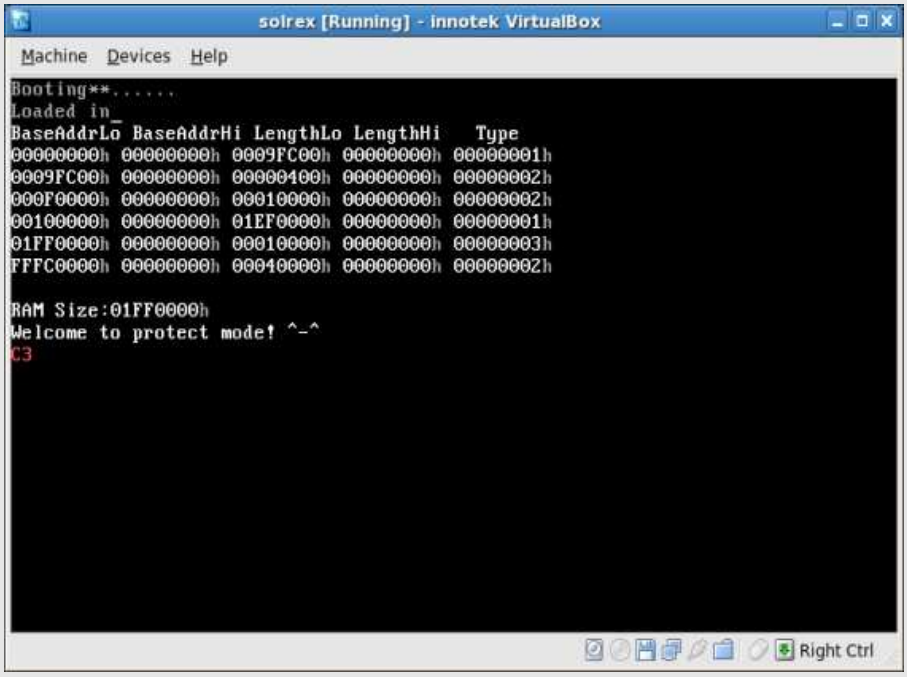


Fig 3.53: 修正記憶體映射的錯誤

打印的記憶體信息告訴我們, 系統可用的記憶體大小是 0x01ff0000 , 大約是 31.9325MB , 我們就可以根據該信息將為頁表段分配記憶體的大小減少到 32KB：

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

3.5.4 體驗虛擬記憶體

在上面兩個小節中, 我們在開啓分頁機制時都是直接將線性位址映射到與之相同的物理位址上。這樣能簡化我們分頁的函數, 但這顯然不是分頁機制的最終目的。回想本節開頭所提到的虛擬記憶體機制, 它可以使每個應用程序都以為自己擁有完整連續的記憶體空間, 即應用程序看到的只是一個完整的線性位址空間, 但只有在使用某位址塊時作業系統才將該位址塊映射到實際的物理位址上, 這個塊的最小單位就是“頁”。這樣就會出現一種情況, 比如用 GDB 調試兩個不同的程序時, 我們會發現兩個程序使用的是同一塊位址, 但位址的內容卻完全不同。這就是因為我們看到的只是線性位址, 它們會被映射到不同的物理位址上。

想想這一機制對程序員有多大幫助吧！程序員在寫應用程序時不必成天為自己怎麼分配記憶體而擔心, 也不必恐懼別的程序會覆蓋自己的記憶體空間（當然, 作為 OS 程序員, 這種恐懼是應當時常放在心上的）, 在他的眼裡, 自己有 4GB(32-bit CPU) 的記憶體可以用—就好像自己擁有一個城市一樣, 太酷了！他可以想住在哪裡就住在哪裡, 想分配什麼位址就分配什麼位址！雖然實際上還是會受到編譯器和運行效率的限制, 不過無論如何, 自由度大大增加了。

下面我們就用一個小例子體驗一下簡單的虛擬記憶體機制：兩次訪問同一線性位址時, 執行的卻是不同的函數（還記得赫拉克利特的那句話嗎 ☺ ）。

如何做呢？首先, 我們需要準備兩個函數, 每個函數各自打印函數自身名稱信息；其次, 因為我們希望兩次訪問同一線性位址執行不同函數, 那麼就需要準備兩套頁目錄和頁表, 兩套頁表中對應該線性位址頁的 PTE 中的基址（物理位址）分別指向我們上面準備的兩個不同的函數, 這樣切換頁表就能實現同一線性位址映射不同的函數上；最後呢, 在示例函數中切換頁表, 並在切換前後呼叫同一個線性位址上的函數, 看打印的信息是否相同。如果打印的信息不同, 就說明該線性位址在頁表切換前後分別映射到了不同的函數上。

準備兩個打印自身信息的函數很簡單：

```
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: 兩個打印自身信息的函數 Foo 和 Bar (節自chapter3/8/loader.S)

這兩個函數的內容很簡單, 就是在屏幕的不同位置打印自身的信息。foo 函數在屏幕的第 12 行 3-5 列打印 Foo 三個紅色字符, bar 也一樣, 在屏幕的第 12 行 6-8 列打印 Bar 三個紅色字符。至于為什麼使用 lret 而不是 ret 指令返回, 我們下面會說明。

編譯之後, 這兩個函數的確是位于不同的物理位址上, 但是這兩個函數入口的物理位址卻不是按照頁面對齊的。回想一下圖 3.42 , PTE 中的頁的物理位址要以 4KB 為邊界對齊。這樣我們才好將同一個頁分別映射到這兩個函數的入口物理位址。要想使這兩個函數入口的物理位址按照 4KB 對齊, 有兩種方法：一種是直接使程式碼對齊 4KB, 比如可以在每個函數入口前使用 GAS 的 `.align 4096` 偽指令使當前位址對齊到 4KB 上, 但是這種方法會使得目標文件中填充大量的 0 , 我們不採用這種方法；另一種方法是定義兩個 4KB 對齊的物理位址, 然後將 foo 和 bar 分別拷貝到這兩個位址中。我們下面採取這個方法。定義兩個 4KB 對齊的物理位址, 一個線性位址和兩個頁表對應的頁目錄和頁表基址：

```
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 對齊的物理位址(節自chapter3/8/loader.S)

在上圖中, FooPhyAddr 和 BarPhyAddr 分別是 foo 和 bar 將要被拷貝到的目標位址。 FuncLinAddr 則是我們要執行函數呼叫所使用的線性位址。再加上有了兩個頁表對應的頁目錄和頁表基址, 我們就可以使用這些信息, 修改上一小節使用的 SetupPaging 函數, 分別設置兩個頁表了：

```
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: 設置兩個頁表並開啓分頁機制(節自chapter3/8/loader.S)

這裡的 SetupPaging 函數比前一小節要復雜了不少, 但是增加的大部分內容卻是重復的, 在上一小節的 SetupPaging 函數中, 我們只初始化了一個頁表, 這裡我們初始化了兩個, 除了頁目錄基址和頁表基址以外, 這兩個頁表的初始化過程是完全相同的。所不同的是在初始化第二個頁表之後, 我們修改了第二個頁表中包含線性位址 FuncLinAddr 那個頁所對應的物理位址。所以, 在第一個頁表中, 所有的線性位址都映射到相應的物理位址上, 線性位址 FuncLinAddr 所對應的物理位址就是與它相同的 FooPhyAddr ;

在第二個頁表中, 所有的線性位址也是映射到相應的物理位址, 除了線性位址 FuncLinAddr 所在的頁被修改為映射到物理位址 BarPhyAddr 上。

這個函數裡使用了一個新段 SelectorFlatRW 和一個新的變量 PageTableNum, 它們的定義如下：

```
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: 添加的新段和變量(節自chapter3/8/loader.S)

其中兩個新段 LABEL_DESC_FLAT_C 和 LABEL_DESC_FLAT_RW 的基址都是 0 , 代表整個線性位址空間。使用這兩個段時, 指令或者位置的段偏移就是該指令或者位置的線性位址。因此在 SetupPaging 函數中使用 LABEL_DESC_FLAT_RW 作為 ES 段, 那麼偏移量 PageDirBase0 所指向的線性位址就是線性位址 PageDirBase0 , 這樣就不需要上一小節所使用的 LABEL_DESC_PAGEDIR 段了。

在 SetupPaging 函數最後, 我們依然使用第一個頁表, 然後開啓分頁機制。第二個頁表雖然被初始化了, 但是並沒有使用。

最後, 我們寫一個示例函數來完成拷貝函數、切換頁表和呼叫同一線性位址的過程：

```
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
481
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: 拷貝函數、切換頁表並呼叫同一線性位址的示例函數(節自chapter3/8/loader.S)

在函數 VMDemo 中, 我們首先拷貝函數 foo 和 bar 的記憶體到目標物理位址 FooPhyAddr 和 BarPhyAddr。由于 foo 和 bar 處于程式碼段, 如果想要拷貝成功, 就需要把程式碼段的屬性改為可讀, 即為 LABEL_DESC_CODE32 加上可讀屬性：

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

然後就可以將 DS 設置為程式碼段, ES 設置為線性位址段, 將每個函數的長度、段偏移和目標物理位址（未分頁前線性位址和物理位址等同）作為 MemCpy 的參數, 執行函數呼叫就能完成拷貝。MemCpy 函數定義于庫文件 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     MemCpy.1
126 MemCpy.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: MemCpy 函數定義(節自chapter3/8/lib.h)

拷貝完成之後, 仍需要恢復 DS 和 ES 為原來的資料段。然後執行 SetupPaging 開啓分頁機制, 首先使用的是第一個頁表, 我們直接呼叫線性位址 FuncLinAddr , 然後切換到第二個頁表, 再次呼叫線性位址 FuncLinAddr。我們可以看到, 這兩次呼叫的指令是一模一樣的, 如果沒有虛擬記憶體機制, 那麼兩次呼叫執行的效果應該是一樣的。

最後在程序中加入對 VMDemo 的呼叫：

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

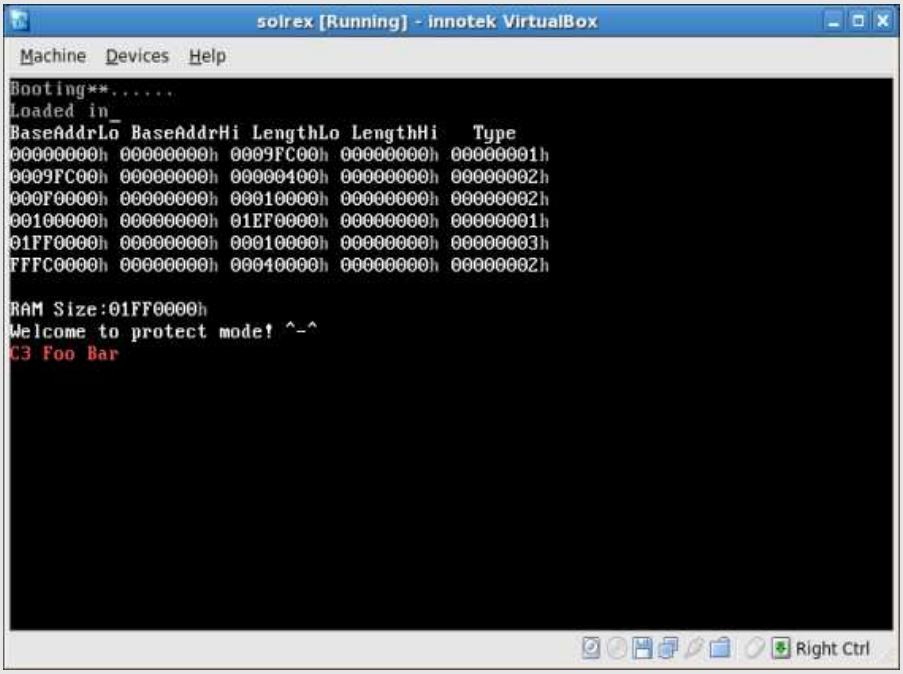


Fig 3.60: 體驗虛擬記憶體

我們將上面的程式碼編譯成鏡像, 用虛擬機加載執行, 結果如圖 3.60 所示。圖中紅色的 Foo 和 Bar 都被打印出來, 說明 foo 函數和 bar 都被執行, 我們的頁表切換起作用了。這樣我們就可以看到, 在不同的頁表中, 同一個線性位址可能代表不同的物理位址, 使用該位址進行函數呼叫, 結果執行的可能是不同的函數。在一般的支持虛擬記憶體的多進程作業系統中, 所採用的也是類似的方法, 按照進程可能使用的空間大小為每個進程分配一個頁表, 進程切換的時候會進行頁表的切換, 這樣進程所見的記憶體就完全是自己的線性位址空間記憶體。在物理位址上增加了一層抽象層, 既方便了應用程序編程, 又能夠更好更安全地管理記憶體。

3.6 結語

本章主要介紹了保護模式下的記憶體使用和管理方式, 通過幾個小例子, 逐步地介紹了進入保護模式、段式存儲、權限和頁式存儲的一些基本概念和實際操作。一般的作業系統書籍對保護模式總是一帶而過, 少有介紹利用保護模式的特性的方法。本章提供的這些例子雖然很小而且很不完善, 但是仍然希望它們能夠作為讀者對保護模式更深入了解的鑰匙, 為讀者打開進一步學習作業系統編程之門。

CHAPTER 4

中斷