编写一个简单的操作系统——从头开始

英国伯明翰大学计算机科学学院

目录

1、简介

我们以前都使用过操作系统（例如 Window XP，Linux 等），也许我们甚至已经编写了一些程序可以在一个操作系统上运行；但是什么是操作系统呢？使用计算机时，多少由硬件完成，多少有软件完成，

兰卡斯特大学一位活泼的老师，已故的道格·谢波德教授曾经一次让我想起，当时我在抱怨一些烦人的编程问题，在他甚至还没有开始尝试任何研究，他都必须从头开始编写自己的操作系统。因此，今天看来，我们对于这些出色的的机器实际是如何工作的，在它们通常捆绑在一起的所有这些软件层以及它们日常有用性所必需的所有层上，是可以理解的。

在这里，我们将集中精力研究广泛使用的 x86 架构 CPU，使我们的计算机上所有的软件脱颖而出，并遵循 Doug 的早期足迹，并逐步学习。

- 电脑如何开机
- 如何在没有操作系统的裸机上编写低级程序
- 如何配置 CPU，一便我们开始使用其扩展功能
- 如何引导用高级语言编写代码，以便我们可以真正开始朝着自己的操作系统取得进展
- 如何创建一些基本的操作系统服务，例如设备驱动程序，文件系统，多任务处理

请注意，就实用的操作系统功能而言，本文章并非旨在进行详尽的介绍，而是旨在将来自许多来源信息的摘要汇总到一个独立且一直的文档中，从而使您可以亲身体验低级别编程，如何编写操作系统以及它们必须解决的问题类型。本文章采用的方法是独特的，因为特定的语言和工具（例如汇编，C，Make 等）不是重点，而是被视为达到目的的一种手段：我们将了解我们需要这些东西以提供帮助我们实现我们的主要目标。

这项工作不是为了替代，而是作为 Minx 项目之类的出色工作以及整个操作系统开发的基石。

2、计算机体系架构和引导过程

2.1 引导过程

现在，我们开始我们的旅程

重新启动计算机时，它必须再次启动，最初没有任何操作系统的概念。它必须以某种方式从当前连接到计算机的某些永久性存储设备（例如软盘，硬盘，USB 加密狗等）中加载操作系统。

我们很快就会发现，您的计算机的预操作系统环境无法提供丰富的的服务：在此阶段，即使是简单的文件系统也是奢侈（例如，将逻辑文件读取和写入磁盘），幸运的是，我们有基本输入/输出软件（BIOS），它是一些软件例程，这些例程从芯片最初加载到内存中，并在计算机启动时进行初始化。BIOS 提供对计算机基本设备（例如屏幕，键盘和硬盘）的自动检测和基本控制。

BIOS 完成对硬件的某些低级测试后，特别是所安装的内存是否正常工作，它必须引导存储在您的其中一台设备上的操作系统。但是，在此提醒我们，BIOS 无法简单地从磁盘加载代表您的操作系统文件，因为 BIOS 没有文件系统的概念。BIOS 必须从磁盘设备的特定物理位置（例如圆柱 2，磁头 3，扇区 5）读取磁盘的特定数据扇区（通常为 512 字节）。

因此，BIOS 找到我们操作系统最容易的地方就是其中一个磁盘的第一个扇区（即柱面 0，磁头 0，扇区 0），称为引导扇区。由于

我们的某些磁盘可能不包含操作系统（它们可能只是连接起来用于其他存储），因此 BIOS 可以确定特定磁盘的引导扇区是用于执行的引导扇区还是仅用于数据，这一点很重要。请注意，CPU 不会区分代码和数据：两者都可以解释为 CPU 指令，其中的代码只是程序员已编写成某种有用算法的指令。

同样，BIOS 在这里采用了一种简单的方法，即必须将目标引导扇区的最后两个字节设置为幻数 0xAA55。因此，BIOS 遍历每个存储设备（例如软盘驱动器、硬盘、CD 驱动器等），将引导扇区读入内存，并指示 CPU 开始执行它发现以幻数结尾的第一个引导扇区。

这是我们抓住计算机的控制权的地方。

2.2 BIOS 引导快和幻数

如果我们使用二进制编辑器（例如 TextPad 或 GHex），则可以让我们将原始字节值写入文件-而不是标准文本编辑器，该标准文本编辑器会将‘A’等字符转换为 ASCII 值-然后我们可以为自己设计一个简单而有效的的引导扇区。

```
 e9 fd ff 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 aa
```

图 2.1：机器码引导扇区，每个字节以十六进制显示

请注意，在图 2.1 中，三个重要功能是

- 最初的三个字节，以十六进制表示 0xe9,0xfd 和 0xff，实际上是机器代码指令，由 CPU 制作商定义，以执行一个无限跳转。
- 最后两个字节 0x55 和 0xaa 组成了幻数，它告诉 BIOS 这确实是启动块，而不仅仅是驱动器引导扇区上的数据。
- 文件以 0 填充（“*”表示未简洁起见省略了 0），是为了将幻数放置在 512 字节磁盘扇区的末尾。

关于字节序的重要说明。您可能想知道为什么幻数之前被描述为 16 位置 0xaa55，但是在我们的引导扇区中却被写为连续的字节 0x55 和 0xaa。这是因为 x86 体系架构是小端存储。

编译器和汇编器可以允许我们定义数据类型，从而对我们隐藏很多字节序问题，例如将 16 位值自动按其正确的字节序列化为机器码。然而，有时很有用，特别是在寻找错误时，要确切知道单个字节将在存储设备或内存中存存储位置，因此字节序非常重要。

这可能是计算机可以运行的的最小程序，但它仍然是有效程序，我们可以通过两种方式进行测试，第二种非常安全，更适合我们的实验。

- 使用当前操作系统允许的任何方式，将此启动块写入非必需储存设备（例如软盘或闪存驱动器）的第一个扇区，然后重新启动计算机。
- 使用虚拟软件（例如 VMWare 或 VirtualBox），并将引导块代码设置为虚拟机的磁盘映像，然后启动虚拟机。

如果您的计算机在引导后只是挂起，而没有诸如“找不到操作系统”之类的消息，则可以确定该代码已被加载并执行。这是我们在代码开头放置的无限循环。如果没有该循环，CPU 将会不受控制，执行内存中的每条后续指令，其中大部分指令将是随机的，未初始化的字节，直到它自身陷入某种无效状态并重新引导或偶然发现并运行 BIOS 例程格式化您的磁盘。

记住，是我们对计算机进行编程，然后计算机盲目地遵循我们的指令，获取并执行它们，直到计算机被关闭。因此，我们需要确保它执行的是精心设计的代码，而不是执行存储在内存中某个位置的随机字节数据。在这个低级别上，我们对计算机有很多权利和责任，因此我们需要学习如何控制它。

2.3 CPU 仿真

还有第三种更方便的选择来测试这些低级程序，而不必连续重新启动计算机或冒着从磁盘上擦除重要数据的风险，那就是使用诸如 Bochs 或 Qemu 之类的 CPU 仿真器。与机器虚拟化（例如 VMware，VirtualBox）试图通过直接在 CPU 上运行 Guest 指令来优化性能并因此优化托管操作系统的使用不同。仿真涉及一个行为类似于特定 CPU 体系架构程序，使用变量来表示 CPU 寄存器和高级控制结构以模拟较低级别的 jumps 等，因此速度较慢，但通常更适合于开发和调试此类系统。

请注意，为了对模拟器执行任何有用的操作，您需要给它一些代码以磁盘映像文件的形式运行。映像文件只是原始数据（即机器代码和数据），否则它们将被写入硬盘，软盘，CDROM，U 盘等介质中。实际上，某些仿真器将成功启动并运行真正的操作系统。从安装 CDROM 下载或提取的映像文件下载系统-尽管虚拟化更适合这种使用。

仿真器将低级显示设备指令转换为桌面窗口上的像素渲染，因此您可以准确地看到在真实显示器上渲染的内容。

通常，对于本文档中的练习。可以得出结论，任何在模拟器下正确运行的机器代码都将在真实体系架构上正确运行-显然必须更快

2.3.1 Bochs：A x86 CPU 仿真

Bochs 要求我们在本地目录中设置一个简单的配置文件 bochsrc，该文件描述如何模拟真实设备（例如屏幕和键盘）的细节，重要的是，在模拟计算机启动时要引导哪个软盘映像。

图 2.2 展示了一个简单的 Bochs 配置文件，我们可以用来测试上面章节编写的并保存的引导扇区文件 boot_sect.bin

```
# Tell bochs to use our boot sector code as though it were
# a floppy disk inserted into a computer at boot time.
romimage: file=$BXSHARE\BIOS-bochs-latest, address=0xf0000
megs: 16
```

图 2.2：一个简单的 Bochs 配置文件

要在 Bochs 中测试我们的引导扇区，只需要输入

$bochs

做一个简单的实验，请尝试将引导区中的 BIOS 幻数更改为无效值，然后重新运行 Bochs。

由于 Bochs 对 CPU 的仿真接近真实情况，因此在 Bochs 中测试代码后，您应该能够在真实的计算机启动它，使其运行的更快。

2.3.2 Qemu

Qemu 与 Bochs 相似，但是效率更高，并且能够仿真 x86 以外的体系架构，尽管 Qemu 的文档记录不如 Bochs 好，但是不需要配置文件意味着它更容易运行，如下所示

$qemu <your-os-boot-disk-image-file>

2.4 16 进制表示法有用性

我们已经看到了 16 进制的一些示例，因此了解为什么在较低级别的编程中经常使用 16 进制非常重要。

首先，考虑为什么十位数对我们来说是如此自然，当我们第一次看到 16 进制时，我们总是问自己：为什么不简单地用 10 进制呢？我不是这方面的专家，我想这个可能与大多数人有 10 个手指，这导致了数字表示为 10 个不同符号：0，1，2，3…8，9。

10 进制的底数为 10（即有 10 个不同的数字符号），而 16 进制的底数为 16，因此我们必须发明一些新的数字符号；懒惰的方法是只使用几个字母，给我们：0,1,2,3,…8,9,a,b,c,d,e,f，例如，单个数字 d 代表一个数 13。

为了区分 16 进制和其他数字系统，我们经常使用前缀 0x 或有时使用后缀 h，这对于不包含任何字母的数字的 16 进制数字尤其重要，例如 0x50 不等于（10 进制）50 – 0x50 实际上是 10 进制的 80。

事实是，一台计算机将一个数字表示为一个位序列（2 进制数字），因为从根本上说，它的电路只能区分两个电状态：0 和 1——就像计算机总共只有两个手指。因此，要代表大于 1 的数字，计算机可以将一系列位组合在一起，就像我们可以用两个或更多的数字来计数大于 9 的数字（例如 456、23 等）。

已经为了某些长度的位系列采用了名称，以使其更容易谈论和商定我们要处理的数字大小。大多数计算机的指令至少处理 8 位值，这些值称为字节，其他分组是 short，int 和 long，通常分别代表 16

位，32 位和 64 位值。我们还看到“字”，用于描述 CPU 当前模式的最大处理单元大小：因此在 16 位实模式下，字是指 16 位的值，在 32 位保护模式下，字是指 32 位的值，以此类推。

因此，回到 16 进制的好处是：位串写起来相当冗长，更加简练的 16 进制表示法比 10 进制转换更加容易得多，本质上是因为我们可以将转换分解成二进制数更小的 4 位段，而不是尝试将所有的位加起来求和。对于较大的位串（例如 16,32,64 等），这会变得更加困难。图 2.3 给出的例子清楚地说明了 10 进制转换的困难。

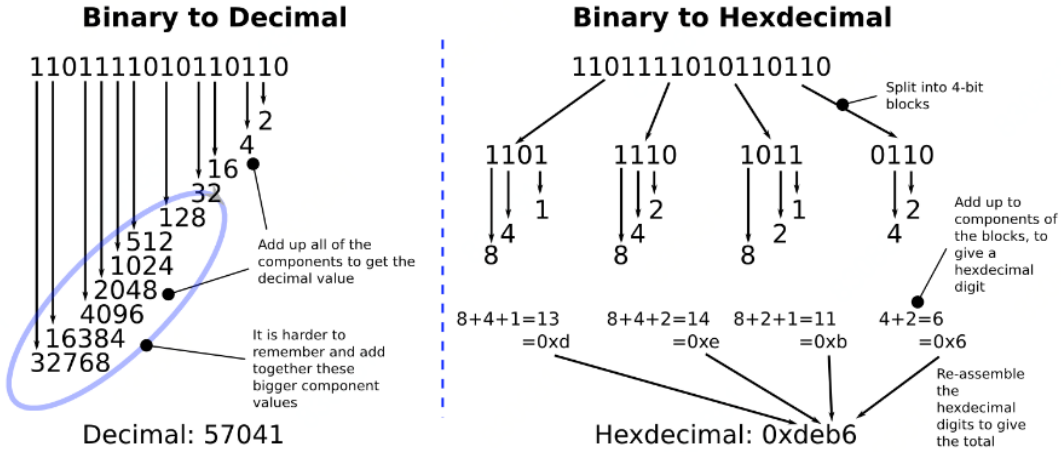


图 2.3 1101111010110110 转换为十进制和十六进制

3、引导扇区编程（16 位实模式）

即使提供了示例代码，您无疑也会发现使用二进制编辑器编写的代码令人沮丧。您必须记住或不断参考，计算机执行的机器代码。幸运的是，编写汇编程序可以将更人性化的指令转换为特定的 CPU 机器代码。在本章中，我们将探索一些日益复杂的引导扇区程序，使用自己熟悉的汇编运行在裸机上。

3.1 重新回顾引导扇区

现在，我们将使用汇编语言重新创建二进制编辑的引导扇区，这样，即使是非常底层的语言，我们也可以真正体会到它的价值。

我们可以将其编译成实际的机器代码（CPU 可以将其解释为指令的字节序列），如下所示：

```
$nasm boot_sect.asm –f bin –o boot_sect.bin
```

其中 boot_sect.asm 是我们在其中保存了图 3.1 中的源代码的文件，而 boot_sect.bin 是我们可以将作为引导扇区安装在磁盘上的汇编机器代码。

请注意，我们使用 –f bin 选项来指示 nasm 生产原始机器代码，而不是包含具有附加远信息的代码包，以用于链接在更加典型的操作系统环境中运行，我们不需要这些东西。除了低级的 BIOS 例程外，现在还仅是这台计算机运行的唯一软件。我们现在就是操作系

统，尽管在这个阶段除了无休止的循环外没有什么可提供的，但是我们很快将以此为基础。

```
;
; A simple boot sector program that loops forever.
;
loop:                   ; Define a label , "loop", that will allow
                            ; us to jump back to it , forever.
jmp loop                ; Use a simple CPU instruction that jumps
                            ; to a new memory address to
                            ; continue execution.
                            ; In our case , jump to the address
                            ; of the current
                            ; instruction.
times 510-($-$$) db 0 ; When compiled , our program must
                             ; fit into 512 bytes ,
                             ; with the last two bytes being
                             ; the magic number ,
                             ; so here , tell our assembly
                            compiler to
                             ; pad out our  program with enough
                            zero
                             ; bytes (db 0) to bring us to the
                             ; 510th byte.
dw 0xaa55               ; Last two bytes (one word) form the
                             ; magic number , so BIOS knows we are
```

图 3.1 一个简单的引导扇区汇编代码

除了将其保存到软盘的引导扇区并重新启动计算机之外，我们还可以通过运行 Bochs 方便的测试该程序：

```
$bochs
```

或者，根据我们的偏好和模拟器的可用性，我们可以使用 Qemu，如下所示：

```
$qemu boot_sect.bin
```

或者，您可以将映像文件加载到虚拟化软件中或将其写入一些可启动媒体，然后从真实的计算机启动它，请注意，将映射文件写入某个可引导媒体时，并不意味着您将文件添加到该媒体的文件系统中：您必须使用适当的工具以低级的方式直接向媒体写入（例如，直接到磁盘的扇区）。

如果我们想更准确地了解汇编程序创建的字节，可以运行以下命令，该命令以易于阅读的 16 进制格式显示文件的二进制内容。

```
$od –t x1 –A n boot_sect.bin
```

该命令的输出应该看起来很熟悉。

恭喜，您刚刚用汇编语言编写了一个引导扇区。正如我们将看到的，所有操作系统都必须以这种方式启动，然后将其自身拉入更高级别的抽象（例如，更高级别的语言，例如 C/C++）。

3.2 16 位实模式

CPU 制作商必须竭尽全力使他们的 CPU（即其特定的指令集）与早期的 CPU 兼容，以便较旧的软件（尤其是较旧的操作系统）仍可以在新的 CPU 上运行。

英特尔兼容 CPU 实施的解决方案是模拟该家族中最古老的 CPU：Intel 8086，该处理器支持 16 位指令并且不使用内存保护：内存保护对于现代操作系统的稳定性至关重要，因为它运行操作系统限制用户进程的访问，比如说，内核内存，无论是偶然还是有意进行的操作，都可能使这样的进程绕过安全机制，甚至使整个系统崩溃。

因此，为了实现向后兼容性，CPU 最初必须以 16 位实模式启动是很重要的，这要求现代操作系统显示地切换到更高级的 32 位（或 64 位）保护模式。但允许旧操作系统继续运行，没有意识到它们正在现代 CPU 上运行。稍后，我们将详细介绍从 16 位实模式到 32 位保护模式这一重要步骤。

通常，当我们说一个 CPU 是 16 位时，我们的意思是说它的指令一次最多可以使用 16 位，例如：一个 16 位的 CPU 会有一个特定的指令，它可以在一个 CPU 周期内将两个 16 位数字相加，如果一个进程需要将两个 32 位数字相加，利用 16 位加法则需要更多的周期。

首先，我们将探索这个 16 位实模式环境，因为所有操作系统都必须从这里开始，然后再看看如何切换到 32 位保护模式以及这样做的主要好处。

3.3 嗯，Hello?

现在，我们将编写一个看似简单的启动扇区程序，该程序在屏幕上显示一条短消息。为此，我们必须学习一些有关的 CPU 工作原理以及如何使用 BIOS 来帮助我们操作屏幕设备的基础知识。

首先，让我们想想我们在这里要做什么。我们想在屏幕上打印一个字符，但是我们不知道如何与屏幕设备通信，因为屏幕设备可能有很多种，它们可能有不同的接口。这就是为什么我们需要使用 BIOS，因为 BIOS 已经对硬件进行了一些自动检测，而且很明显，BIOS 之前在屏幕上打印了关于自检等信息，因此可以为我们提供帮助。

所以，接下来，我们想让 BIOS 为我们打印一个字符，但是我们如何让 BIOS 来打印呢？有没有 java 库可以打印到屏幕上——这是一个梦想，但是，我们可以肯定的是，在计算机内存中的某处会有一些 BIOS 机器代码，该代码知道如何写入屏幕。事实上，我们可能会在内存中找到 BIOS 代码并以某种方式执行它。但这比较麻烦，而且当不同的机器上的 BIOS 例程内部存在差异时，很容易出错。

这里我们可以利用计算机的基本机制：中断。

3.3.1 中断

中断是一种机制，可以让 CPU 暂时中止正在执行的操作并运行其他一些优先级更高的指令，然后再返回原始任务。可以通过软件指令（例如 int 0x10）或需要更高优先级操作的某些硬件设备（例如从网络设备读取某些传入数据）来引发中断。

每个中断都由一个唯一的数字表示，该数字是中断向量索引，BIOS 最初在内存开始处（即物理地址 0x0 处）建立了一个表，该表包含指向中断服务程序（ISR）的地址指针。ISR 只是一系列机器指令，类似于我们的引导扇区代码，它处理特定的中断（例如，可能从磁盘驱动器或网卡读取数据）。

因此，简而言之，BIOS 会将自己的一些 ISR 添加到专门处理计算机某些方面的中断向量中，例如：中断 0x10 导致调用与屏幕相关的 ISR；中断 0x13 与磁盘相关的 I/O ISR。

然而，为每个 BIOS 例程分配一个中断将是浪费，因此 BIOS 通过一个我们可以想象的 switch 语句来多路复用 ISR，通常基于引发中断之前在 CPU 通用寄存器 AX 中设置的值。

3.3.2 CPU 寄存器

正如我们在高级语言中使用变量一样，如果我们可以在特定的例程中临时存储数据，这是很有用的。为了这个目的，所有 x86 CPU

都有四个通用寄存器 AX，BX，CX 和 DX，每个寄存器可以保存一个字（两个字节，16 位）的数据，可以由 CPU 读写，与访问主存相比，延时可以忽略不计。在汇编程序中，最常见的一种或多种操作是在寄存器之间移动数据。

```
mov ax, 1234        ; store the decimal number 1234 in ax
mov cx, 0x234       ; store the hex number 0x234 in cx
mov dx, 't'             ; store the ASCII code for letter 't' in dx
mov bx, ax          ; copy the value of ax into bx, so now bx ==
1234
```

请注意，目的地是 move 操作的第一个参数，而不是第二个参数，但此约定因不同的汇编器而异。

有时使用单字节更方便，因此这些寄存器允许我们独立设置其高位和低位字节：

```
mov ax, 0           ; ax -> 0x0000 , or in binary 0000000000000000
mov ah, 0x56        ; ax -> 0x5600
mov al, 0x23        ; ax -> 0x5623
mov ah, 0x16        ; ax -> 0x162
```

3.3.3 综合使用

因此，回想一下，我们希望 BIOS 在屏幕上为我们打印一个字符，我们可以通过将 AX 设置为一些自定义值，然后触发特定的中断来调用特定的 BIOS 例程。我们需要特定的例程是 BIOS 滚动 Tele-type 例程。它将在屏幕上打印单个字符并前进光标，为下一个字符做好准备。这里有一个完整的 BIOS 例程列表，显示了要使用哪个中

断以及如何在中断之前设置寄存器。这里我们需要中断 0x10，并将 AH 设置为 0x0e（表示 tele-type 模式）。并将 AL 设置为我们希望打印的字符 ASCII 代码。

```
;
; A simple boot sector that prints a message to the screen using
; a BIOS routine.
;

mov ah, 0x0e      ; int 10/ah = 0eh -> scrolling teletype BIOS routin

mov al, 'H'
int 0x10
mov al, 'e'
int 0x10
mov al, 'l'
int 0x10
mov al, 'l'
int 0x10
mov al, 'o'
int 0x10

jmp $             ; Jump to the current address (i.e. forever).

;
; Padding and magic BIOS number.
;

times 510-($-$$) db 0 ; Pad the boot sector out with zer

dw 0xaa55         ; Last two bytes form the magic number ,
                  ; so BIOS knows we are a boot secto
```

图 3.2

图 3.2 显示了整个引导扇区程序。请注意，在这种情况下，我们只需要设置一次 AH，然后将 AL 更改为不同的字符。

```
b4 0e b0 48 cd 10 b0 65 cd 10 b0 6c cd 10 b0 6c
cd 10 b0 6f cd 10 e9 fd ff 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 aa
```

图 3.3

为了完整起见，图 3.3 显示了该引导扇区的原始机器代码，这些实际字节，它们告诉 CPU 如何操作。如果您对编写这样精巧的程序所涉及的大量工作感到惊讶，那么请记住，这些指令与 CPU 电脑非常接近，因此它们必须非常简单，但也非常的快。实际上，您正在了解您的计算机。

3.4 Hello world!

现在我们将尝试一个稍微高级一点的 hello 程序，该程序引入了更多的 CPU 基础知识，并了解了 BIOS 引导我们的扇区进入内存环境。

3.4.1 内存，地址和标签

我们前面说过 CPU 如何从内存中获取和执行指令，以及 BIOS 如何将 512 字节的引导扇区加载到内存，然后在完成初始后，告诉 CPU 跳转到代码的开头，开始执行我们的第一条指令，然后执行下一条，然后再执行下一条，依次类推。

因此，我们引导扇区的代码在内存中。但是在哪里呢？我们可以将主存想象为一个长字节序列，可以由一个地址（即索引）单独访问。因此，如果我们想找出内存第 53 个字节的内容，则 54 是我们的地址，通常用 16 进制表示为：0x36。

因此，引导扇区代码的开始（第一个机器码字节）在内存中的某个地址处，正是 BIOS 将我们引导到该位置，除非另有说明，或者我们可能会假设 BIOS 在内存的起始地址 0x0 处加载了我们的代码。不过，它并不是那么简单，因为我们知道 BIOS 早在加载代码之前就已经在计算机上进行初始化工作，并且实际上还会继续为时钟，磁盘驱动器等硬件提供中断服务。因此，这些 BIOS 例程（例如屏幕打印服务等）本身必须存储在内存中的某一个地方，并且必须在它们仍在使用时必须加以保留（即不覆盖）。另外，我们前面已经提到了，中断向量位于内存的开始位置，如果 BIOS 将我们加载到那里，我们的代码将在表上踩踏，并且在下一个中断发生时，计算机可能会崩溃并重新启动：中断号和 ISR 之间的映射将被有效切断。

事实证明，BIOS 总是喜欢将引导扇区加载到地址 0x7c00,该地址肯定不会被重要例程占用，图 3.4 给出了当我们引导扇区刚刚加载到计算机的典型低内存布局示例。虽然我们可能会指示 CPU 将数据写入内存中的任何地址，但这可能会导致不好的事情发生，因为某些内存正在被其他例程使用，例如计时器中断和磁盘设备。

3.4.2 标签 ‘X’

现在，我们将玩一个名为“查找字节”的游戏，该游戏将演示内存引用，汇编代码中标签的使用以及知道 BIOS 将我们加载到何处的重要性。我们将编写一个汇编程序，为一个字符保留一个字节数据，

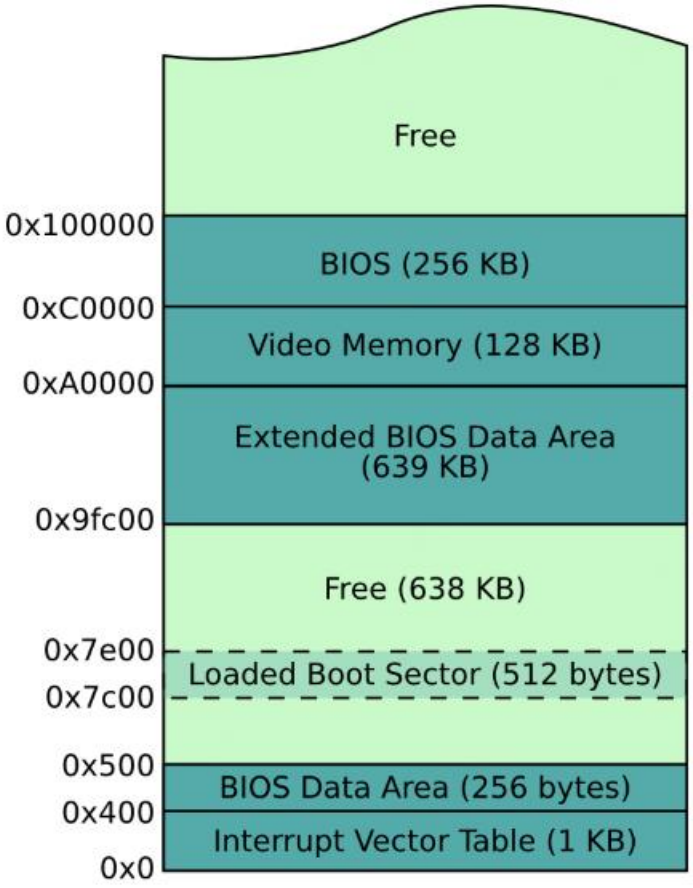


图 3.4 引导后典型的较低内存布局

然后我们尝试在屏幕上打印出这个字符。为此我们需要计算出它的绝对内存地址，以便我们可以将它加载到另一个内存区域，让 BIOS 打印它，就像上一个练习一样。

```
;
```

```
; A simple boot sector program that demonstrates addressing.
;
mov ah, 0x0e      ; int 10/ah = 0eh -> scrolling teletype BIOS
routine

; First attempt
mov al, the_secret
int 0x10          ; Does this print an X?


; Second attempt
mov al, [the_secret]
int 0x10          ; Does this print an X?

; Third attempt
mov bx, the_secret
add bx, 0x7c00
mov al, [bx]
int 0x10          ; Does this print an X?

; Fourth attempt
mov al, [0x7c1e]
int 0x10          ; Does this print an X?

jmp $             ; Jump forever

the_secret:
db "X"

; Padding and magic BIOS number.
times 510-($-$$) db 0
dw 0xaa55
```

首先，当我们在程序中声明一些数据时，我们在它们前面添加上一个标签（the_secret）,我们可以把标签放在程序中的任何地方，它们的唯一目的是给我们一个从代码开始到特定指令或数据的偏移量。

```
b4 0e b0 1e cd 10 a0 1e 00 cd 10 bb 1e 00 81 c3
00 7c 8a 07 cd 10 a0 1e 7c cd 10 e9 fd ff 58 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 aa
```

图 3.5

如果我们看下图 3.5 中的汇编机器代码，可以看到我们的“X”它有一个 16 进制 ASCII 代码 0x58，在我们用 0 填充引导扇区之前，它与代码开头的偏移量为 30（0x1e）字节。

如果我们运行该程序，我们将看到只有两次尝试成功打印“X”。

第一次尝试的问题是，它试图将偏移量加载到 al 中作为要打印的字符，但实际上我们希望将偏移量处的字符打印而不是偏移量本身，如下面所尝试的，方括号指示 CPU 要做这件事——存储地址的内容。

为什么第二次尝试失败？问题是，CPU 把偏移量当做内存的开始，而不是从我们加载代码的起始地址，它将把它放在中断向量中。第三次尝试中，我们使用 CPU add 指令将偏移量 the_secret 添加到我们认为已经加载代码 0x7c00 的 BIOS 地址，我们可以将 add 转换为高级语言语句 bx=bx+0x7c00。现在我们已经计算出正确的“X”内存地址，并且使用 mov al,[bx] 指令将该地址的内容存储在 AL 中，准备用于 BIOS 打印。

在第四次尝试中，我们试着稍微聪明点，根据我们先前对二进制代码的检查（参见图 3.5），在引导扇区被 BIOS 加载到内存后，预先计算“X”的地址。我们到达了地址 0x7c1e，这表明从引导扇区开始到

“X”是 0x1e（30）字节。最后一个例子提醒我们，为什么标签是有用的，因为没有标签，我们必须计算编译代码的偏移量，然后代码更改导致这些偏移量改变时再更新这些偏移量。

现在我们已经了解了 BIOS 确实是将引导扇区加载到了 0x7c00 处，同时我们还了解了地址和汇编代码标签之间的关系。

在代码中总是要考虑这个标签（内存偏移量）是不方便的，因此如果在代码的底部包含以下指令，告诉它您希望代码加载到内存中的确切位置，则许多的汇编程序都会在汇编过程中更正标签引用：

```
[org 0x7c00]
```

问题 1

当这个 org 指令被添加到这个引导扇区程序中时，您认为会打印什么？为了获得好的成绩，请解释为什么会这样。

3.4.3 定义字符串

假设您想在某个时候在屏幕上打印预定义的消息（例如“Booting OS”）；您将如何在汇编程序中定义这样的字符串？我们必须提醒自己，我们的计算机对字符串一无所知，而字符串仅仅是存储在内存中某一个位置的一系列数据单元（例如字节，字等）。

在汇编中，我们可以如下定义一个字符串：

```
my_string:
    db ’Booting OS’
```

实际上，我们已经看过 db 了，它表示“声明数据字节”，它告诉汇编器将随后的字节直接写到二进制输出文件中（即，不要将它解释为处理器指令）。由于我们用引号将数据引起了，因此汇编器知道将每个字符转换为 ASCII 字节码。请注意，我们经常使用标签（例如 my_string）来标记数据的开始，否则我们将无法在代码中轻松引用它。

在这个例子中，我们忽略了一件事：知道字符串的长度与知道它在哪里一样重要。由于必须由我们编写处理字符串所有代码，所以有一个一致的策略来了解字符串的长度很重要。有几种可能，但惯例是声明字符串以 null 结尾，这意味着我们总是将字符串最后一个字节声明为 0，如下所示：

```
my_string:
    db 'Booting OS',0
```

当遍历字符串时，也许是以此打印每个字符，我们可以轻松确定何时到达末尾。

3.4.4 使用栈

当讨论底层计算时，我们经常听到别人讨论栈，好像它是某种特殊的东西，栈实际上只是解决以下不便的一个简单方案：用于存储例程中临时变量的 CPU 寄存器有限，但我们通常需要更多的临时存储空间来容纳这些变量，现在，我们可以利用主存，但是在读写时

指定特定的内存地址是不方便的，特别是因为我们不关心数据存储在哪里，我们只需要要很容易检索到它就够了。还有，我们将在后面看到，栈对于传递参数以实现函数调用也很有用。

因此，CPU 提供了两条指令 push 和 pop，分别允许我们存储一个值和栈顶弹出一个值，这样就不用关心它们存储的位置。但请注意，我们不能将单个字节推送到栈上或从栈中弹出：在 16 位模式下，栈仅在 16 位边界上工作。

栈由两个特殊的 CPU 寄存器 bp 和 sp 实现，它们分别维护栈基地址（即栈的底部）和栈的顶部地址。由于栈在将数据压入栈时会扩展，因此通常会设置栈的底部远离重要内存区域（例如 BIOS 代码或我们的代码），因此如果栈太大它们也不会覆盖。关于栈的一个令人困惑的事情是，它实际上是从基指针向下增长的，因此我们发出一个 push 时，该值实际上存储在 bp 的地址一下而不是在之上，而 sp 则按值的大小递减。

接下来图 3.6 中引导扇区程序演示了栈的使用。

问题 2

图 3.6 中的代码将按什么顺序打印，为什么？在绝对地址 C 存储什么字符？您可能会发现修改代码以确认您的期望是有用的，但是一定要解释为什么是此地址。

3.4.5 控制结构

如果我们不知道如何编写一些基本的控制语句，例如 if…then …else if … else 和 while，那么使用编程语言是不舒服的，而这些结构允许执行其他的分支，并构成任何有用的例程的基础。

编译后，这些高级控制结构将简化为简单的跳转语句。实际上，我们已经看到了最简单的循环示例：

```
some_label:
    jmp some_label    ; jump to address of labe
```

或者如下，具有相同的效果

```
jmp $      ; jump to address of current instruction
```

该指令为我们提供了无条件跳转（即，死循环）；但我们通常需要根据某种条件跳转（例如，进行循环，直到我们循环 10 次等等）。

```
;
; A simple boot sector program that demonstrates the stack.
;
  mov ah, 0x0e      ; int 10/ah = 0eh -> scrolling teletype BIOS routine

  mov bp, 0x8000    ; Set the base of the stack a little above where BIOS
  mov sp, bp        ; loads our boot sector - so it won't overwrite us.

  push 'A'          ; Push some characters on the stack for later
  push 'B'          ; retreival.  Note, these are pushed on as
  push 'C'          ; 16-bit values, so the most significant byte
                    ; will be added by our assembler as 0x00.

  pop bx            ; Note, we can only pop 16-bits, so pop to bx
  mov al, bl        ; then copy bl (i.e. 8-bit char) to al
  int 0x10          ; print(al)

  pop bx            ; Pop the next value
  mov al, bl
  int 0x10          ; print(al)

  mov al, [0x7ffe]  ; To prove our stack grows downwards from bp,
                    ; fetch the char at 0x8000 - 0x2 (i.e. 16-bits)
  int 0x10          ; print(al)

  jmp $             ; Jump forever.

; Padding and magic BIOS number.

  times 510-($-$$) db 0
  dw 0xaa55
```

图 3.6 操作栈 push 和 pop

在汇编语言中，条件跳转是通过首先运行一条比较指令，然后通过发出一条特定的条件跳转指令来实现的。

```
cmp ax, 4            ; compare the value in ax to 4
je then_block        ; jump to then_block if they were equal
mov bx, 45           ; otherwise , execute this code
jmp the_end          ; important: jump over the 'then' block ,
                     ; so we don't also execute that code.

then_block:
    mov bx, 23
the_end:
```

在 C 或者 Java 这样的语言中，应该是这样的。

```
if(ax == 4) {
    bx = 23;
} else {
    bx = 45;
}
```

我们可以从汇编示例中看到，在幕后发生了一些事情，在这个例子中，将 cmp 指令与其执行的 je 指令相关联。其中 CPU 的特殊 flags 寄存器捕获 cmp 指令结果，以便随后的条件跳转指令可以确定是否跳转到指定的位置。

基于先前的 cmp x,y 指令，可以使用一下跳转指令。

```
je   target     ; jump if equal                 (i.e. x == y)
jne  target     ; jump if not equal             (i.e. x != y)
jl   target     ; jump if less than             (i.e. x < y)
jle  target     ; jump if less than or equal    (i.e. x <= y)
jg   target     ; jump if greater than          (i.e. x > y)
jge  target     ; jump if greater than or equal (i.e. x >= y
```

问题 3

用高级的语言来写条件代码，然后用汇编指令替换它。使用 cmp 和适当的跳转指令，尝试将这个伪汇编转换成完整的汇编代码，用不同的 bx 值进行测试。用你自己的话完整注释你的代码。

```
mov bx, 30
if (bx <= 4) {
    mov al, 'A'
} else if (bx < 40) {
    mov al, 'B'
} else {
    mov al, 'C'
}
mov ah, 0x0e    ; int =10/ah=0x0e -> BIOS tele -type outputint 0x10
                    ; print the character in al
jmp $

; Padding and magic number.
times 510-($-$$)
db 0dw 0xaa55
```

3.4.6 函数调用

在高级语言中，我们将大问题分解成函数，这些函数本质上是我们整个程序中反复使用的通用例程（例如打印消息，写入文件等），通常会更改传递给函数的参数，以某种方式更改结果。在 CPU 级别上，一个函数只不过是跳转到一个有用的例程地址，然后在第一次跳转之后立即跳回指令。

我们可以模拟这样的函数调用：

```
...
...
mov al, 'H'             ; Store 'H' in al so our function will print
it.
```

```
jmp my_print_function
return_to_here:    ; This label is our life -line so we can get
back
...
...
my_print_function:
  mov ah, 0x0e        ; int =10/ah=0x0e -> BIOS tele -type output
  int 0x10         ; print the character in al
  jmp return_to_here ; return from the function call
```

首先，请注意我们通过将寄存器 al 设置为被调用函数的参数。这就是在高级语言中实现参数传递的方式，调用方和被调用方必须就传递参数位置和数量达成一致。

遗憾的是，这种方法的缺陷是，在函数被调用后，我们需要显式地说明返回位置，因此无法从程序中的任意点调用此函数—在这种情况下，标签为 return_to_here 它将始终返回相同的地址。

借鉴参数传递的思想，调用者代码可以将正确的返回地址（即调用后的地址）存储在某个已知位置，然后被调用代码可以跳回该存储位置。CPU 在特殊寄存器 RIP（指令指针）中跟踪当前正在执行的指令，遗憾的是，我们不能直接访问它。然而 CPU 提供了一对指令，call 和 ret，它们完全符合我们的要求：call 的行为类似 jmp，但此外，在实际跳转前，将返回地址压入栈，ret 将返回地址从栈中弹出并跳转到它，如下所示：

```
...
...
mov al, 'H'        ; Store 'H' in al so our function will print it.
call my_print_function
...
...
my_print_function:
mov ah, 0x0e          ; int =10/ah=0x0e -> BIOS tele -type output
int 0x10              ; print the character in al
```

```
ret
```

我们的功能现在几乎是自成体系的，但是仍然存在一个棘手的问题，如果我们现在不辞辛劳地去考虑它，我们以后会感谢自己的。当我们在汇编程序中调用一个函数，如 print 函数时，该函数可能会在内部更改多个寄存器的值以执行其工作（实际上，由于寄存器是一种稀缺资源，它几乎肯定会这样做），因此当我们的程序从函数调用返回时，可能无法安全的假设，例如，我们存储在 dx 中的值仍然存在。

因此，对于一个函数来说，立即将其计划更改的所有寄存器压入栈，然后在返回之前立即再次弹出它们（即恢复寄存器的原始值），这通常是明智的。由于一个函数可以使用许多通用寄存器，因此 CPU 实现了两条便捷的指令 pusha 和 popa，它们可以方便地分别将所有寄存器压入和弹出栈，例如：

```
...
...
some_function:
pusha                   ; Push all register values to the stack
mov bx, 10
add bx, 20
mov ah, 0x0e            ; int =10/ah=0x0e -> BIOS tele -type outp
int 0x10                ; print the character in al
popa                    ; Restore original register values
ret
```

3.4.7 包含文件

即使是最简单的汇编代码，您可能仍想在多个程序中重用你的代码。nasm 允许您通过 include 引入外部文件：

```
%include "my_print_function.asm"   ; this will simply get replaced by
```

```
                                    ; the contents of the fil
...
mov al, 'H'       ; Store 'H' in al so our function will print it.
call my_print_function
```

3.4.8 综合使用

现在，我们对 CPU 和汇编语言有了足够的了解，可以编写更复杂的“Hello world”启动程序。

问题 4

将本节中的所有想法放在一起，创建一个独立函数，以打印以 null 结尾的字符串，该函数可以按一下方式使用：

```
;
; A boot sector that prints a string using our function.
;
[org 0x7c00]      ; Tell the assembler where this code will be
loaded
mov bx, HELLO_MSG ; Use BX as a parameter to our function , so
call print_string ; we can specify the address of a string.

mov bx, GOODBYE_MSG
call print_string
jmp $ ; Hang
%include "print_string.asm"

; Data
HELLO_MSG:
db 'Hello , World!', 0   ; <-- The zero on the end tells our routine

; when to stop printing characters
GOODBYE_MSG:
db 'Goodbye!', 0

; Padding and magic number.
times 510-($-$$) db 0
```

```
dw 0xaa55
```

为了获得好的分数，请确保函数在修改寄存器时非常小心，并充分注释代码以展示您的理解。

3.4.9 概要

不过，感觉我们还没有走的很远。没关系，考虑我们一直工作在原始环境，这是很正常的。如果您在这里之前的都很明白，那么我们就在路上了。

3.5 护士，给我听诊器

到目前为止，我们已经设法让计算机打印出我们已经加载到内存中的字符和字符串，但是很快我们将尝试从磁盘中加载一些数据，因此如果我们能够显示存储在二进制内存地址的十六进制值，以确认我们是否确实加载了任何内容，这将非常有用。记住，我们没有一个好的图形用户界面开发工具，它配有一个可以让我们仔细检查代码的调试器，当我们犯错时，计算机能够给我们最好的反馈，所以我们需要照顾好自己。

我们已经编写了一个打印一串字符的例程，所以我们现在将其扩展到十六进制的打印例程中——在这个无情的低级世界中一定要珍惜的例程。

让我们仔细考虑一下我们将如何做到这一点，首先仔细考虑我们将如何执行此操作。在高级语言中，我们想要这样的东西：print_hex(0x1fb6)，它将导致在屏幕上打印字符串“0x1fb6”。我们已经在第 3.4.6 节中看到了如何在汇编中调用函数以及如何将寄存器用作参数，因此让我们使用 dx 寄存器作为参数来保存我们希望打印十六进制函数打印的值：

```
mov dx, 0x1fb6     ; store the value to print in dx
call print_hex     ; call the function

; prints the value of DX as hex.
print_hex:
  ...
  ...
  Ret
```

由于我们要在屏幕上打印一个字符串，因此我们不防重新使用我们以前的打印功能来完成实际的打印部分，那么我们的主要任务是研究如何从参数 dx 中的值构建该字符串。我们绝对不想在组装时混淆过多的事情，因此，请考虑一下技巧，以使我们开始使用此功能。如果我们在代码中将完整的十进制字符串定义为一个模板变量，就像我们之前定义的“hello world”消息一样，我们只需获取字符串打印函数即可进行打印，那么我们的 print_hex 例程的任务就是更改该模板字符串的组成部分，以将十六进制值反映位 ASCII 代码：

```
mov dx, 0x1fb6            ; store the value to print in dx
call print_hex            ; call the function

; prints the value of DX as hex
print_hex:
; TODO: manipulate chars at HEX_OUT to reflect DX

mov bx, HEX_OUT           ; print the string pointed to
```

```
call print_string          ; by BX
ret

; global variables
HEX_OUT: db '0x0000 ',0
```

问题 5（高级）

完成 print_hex 函数的实现。您可能会发现 CPU 指令和 shr 很有用，您可以在 Internet 上找到有关它们的信息。请确保用自己的话语充分说明代码并加上注释。

3.6 读取磁盘

我们现在已经了解了 BIOS，并在计算机的低级别环境中进行了一些尝试，但是我们有一个小问题阻碍了我们编写操作系统的计划：BIOS 从磁盘的第一个扇区加载了引导代码，但这是完全加载的，如果我们的操作系统代码更大呢-而且我估计它会超过 512 字节。

操作系统通常不止一个扇区（512 字节），因此它们首先需要做的事情之一就是将剩余的代码从磁盘引导到内存中，然后开始执行这些代码，幸运的是，正如前面所诉，BIOS 提供了允许我们操作磁盘驱动的例程。

3.6.1 使用段访问扩展内存

当 CPU 在最初的 16 位实模式下运行时，寄存器的最大大小是 16 位，这意味着我们可以在指令引用的最高地址是 0xFFFF，按照今天的标准，这相当于微不足道的 64KB（65536 字节）。现在，也许我们想要的简单操作系统不会受到这个限制的影响，但是一个日常的操作系统永远不会舒服地坐在这样一个狭小的盒子里，所以我们理解这个问题的解决方案是很重要的。

为了解决这个限制，CPU 设计人员添加了一些特殊的寄存器 cs,ds,ss 和 es 称为段寄存器。我们可以将主存储器想象成由段寄存器索引的段，这样，当我们指定 16 位地址时，CPU 会自动将绝对地址计算为适当段的起始地址，所谓适当的段，我的意思是，除非另有明确说明，否则 CPU 将会从适合指令上下文的段寄存器中便宜我们的地址，例如：指令 mov ax.[0x45ef]中使用的地址默认为数据段的偏移量，由 ds 索引；同样，栈段 SS 用于修改栈基本指针 bp 的实际位置。

段寻址最让人困惑的是，相邻的段几乎完全重叠，但只有 16 个字节，因此不同的段和偏移量组合实际上可以指向同一个物理地址，讨论的足够多了，在没看到实例之前，我们不会真正理解这个概念。

为了计算绝对地址，CPU 将段寄存器中的值乘以 16（左移 4 位），然后再加上偏移量地址；由于我们使用的是十六进制，所以当我们将一个数字乘以 16 时，我们只需将其向左移动一个数字（例如

0x42*16=0x420）。因此，如果我们将 ds 设置为 0x4d，然后发出语句 mov ax,[0x20]，则存储在 ax 中的值实际上将从地址 0x4d0（16*0x4d+0x20）加载。

图 3.7 显示了如何设置 ds 来实现与标签寻址类似的更正，就像在第 3.4.8 节中使用[org 0x7c00]指令时一样。由于我们不使用 org 指令，因此当 BIOS 将代码加载到 0x7c00 时，汇编程序不会将标签偏移到正确的内存位置，因此首次打印“X”的尝试失败。但是，如果我们将数据段寄存器 ds 设置为 0x7c00，则 CPU 将为我们执行此偏移量（即 0x7c00*16+the_secret），因此第二次尝试将正确打印“X”。在第三次和第四次尝试中，我们执行了相同的操作，并获取得到相同的结果，但是如何在计算物理地址时显式地告诉 CPU 要使用哪个寄存器，比如使用通用寄存器 es。

请注意，CPU 电路的局限性（至少在 16 位实模式下）在这里暴露出来了，当如 mov ds，0x1234 看似正确的指令实际上是不可能的；我们可以将一个字地址直接存储到通用寄存器中（例如 mov ax,0x1234 或 mov cx，0xdf）,这并不意味着我们可以对每种类型的寄存器（例如段寄存器）执行相同的操作；因此，如图 3.7 所示，我们必须采取其他步骤才能对通用寄存器赋值。

因此，基于段的寻址允许我们访问更多的内存，最高可达 1MB（0xffff*16+0xffff）。稍后，我们将看到，当我们切换到 32 位保护模式时，如何访问更多的内存，但现在我们只需了解 16 位实模式分段寻址就足够了。

```
;
; A simple boot sector program that demonstrates segment offsetting
;
  mov ah, 0x0e          ; int 10/ah = 0eh -> scrolling teletype BIOS routine

  mov al, [the_secret]
  int 0x10              ; Does this print an X?

  mov bx, 0x7c0         ; Can't set ds directly, so set bx
  mov ds, bx            ; then copy bx to ds.
  mov al, [the_secret]
  int 0x10              ; Does this print an X?

  mov al, [es:the_secret] ; Tell the CPU to use the es (not ds) segment.
  int 0x10              ; Does this print an X?

  mov bx, 0x7c0
  mov es, bx
  mov al, [es:the_secret]
  int 0x10              ; Does this print an X?

  jmp $                 ; Jump forever.

the_secret:
  db "X"

; Padding and magic BIOS number.
  times 510-($-$$) db 0
  dw 0xaa55
```

图 3.7 用 ds 寄存器操作数据段

3.6.2 磁盘驱动器的工作原理

从机械上讲，硬盘驱动器包含一个或多个堆叠的盘片，它们在读/写磁头下旋转，就像旧的点唱机，增加了容量，唱片一张一张地叠在另一张上面，磁头在其中来回移动以覆盖整个旋转的盘片的表面。

由于一个特定的盘片在其两面都是可读可写的，所以一个读/写磁头可以浮在上面，另一个在它下面。图 3.8 显示了一个典型的硬盘驱动器内部，有一堆盘和磁头暴露在外。请注意，同样的想法也适用

于软盘驱动器，软盘驱动器通常只有一个双面软盘介质，而不是几个堆叠的硬盘。

盘片的金属涂层使它们具有以下特性：磁头可以将其表面的特定区域磁化或消磁，从而有效地将任何状态永久记录在其上。因此，重要的是能够描述磁盘表面上要读取或写入某些状态的确切位置，因此使用 Cylinder-Head-Sector（CHS）寻址，它实际上是 3D 坐标系（请参加图 3.9）。

- Cylinder：柱面描述了磁头与盘片边缘的离散距离，之所以这样命名是因为当几个盘片堆叠在一起时，您可以看到所有磁头都通过所有盘片选择了一个柱面。
- Head：磁头描述了我们感兴趣的轨迹（即圆柱内哪个特定的盘片表面）。
- Sector：圆形磁道被分为扇区，通常 512 字节，可以用扇区索引来引用。

图 3.8 磁盘驱动器内部

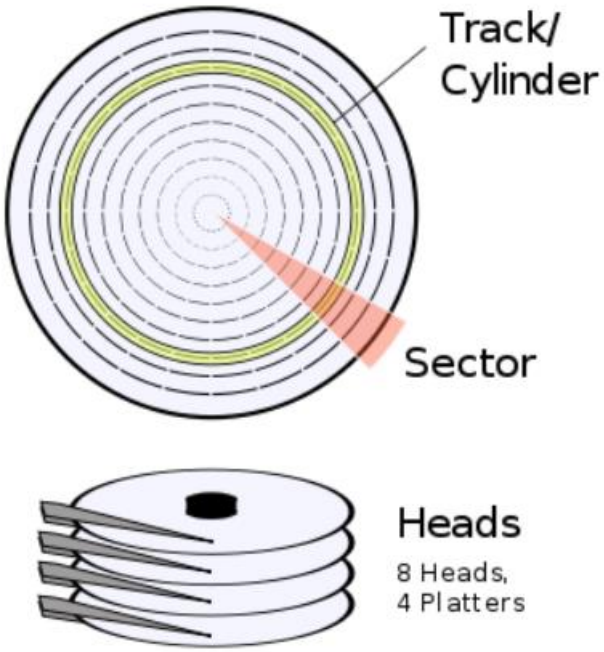


图 3.9 磁盘结构柱面，磁头，扇区

3.6.3 使用 BIOS 读取磁盘

稍后我们将看到，特定的设备需要编写特定的例程才能使用他们，例如，软盘设备要求我们正在使用前明确地打开和关闭在读写磁头下旋转磁盘电机，而大多数硬盘设备在本地芯片上有更多的自动化功能，但这些设备连接到 CPU 总线技术（如 ATA/IDE、SATA、SCSI、USB 等）同样会影响我们访问他们的方式，值得庆幸的是，BIOS 可以提供一些磁盘例程来抽象常见磁盘设备的所有差异。

将寄存器 al 设置为 0x02 后，通过引发中断 0x13 来访问我们在此处感兴趣的特定 BIOS 例程。此 BIOS 例程要求我们设置其他一些寄

存器，以详细说明要使用的磁盘设备，希望从磁盘读取的块以及将块存储在内存中的位置。使用例程最困难的部分是，我们必须使用 CHS 寻址方案指定要读取的第一块。如随后的代码段中所诉。

```
mov ah, 0x02 ; BIOS read sector function

mov dl, 0    ; Read drive 0 (i.e. first floppy drive)
mov ch, 3    ; Select cylinder 3
mov dh, 1    ; Select the track on 2nd side of floppy
             ; disk, since this count has a base of 0
mov cl, 4    ; Select the 4th sector on the track - not
             ; the 5th, since this has a base of 1.
mov al, 5    ; Read 5 sectors from the start point

; Lastly, set the address that we'd like BIOS to read the
; sectors to, which BIOS expects to find in ES:BX
; (i.e. segment ES with offset BX).
mov bx, 0xa000  ; Indirectly set ES to 0xa000
mov es, bx
mov bx, 0x1234  ; Set BX to 0x1234
; In our case, data will be read to 0xa000:0x1234, which the
; CPU will translate to physical address 0xa1234

int 0x13     ; Now issue the BIOS interrupt to do the actual read.
```

请注意，由于某种原因（例如，我们索引超出磁盘限制的扇区，试图读取有故障的扇区，未将软盘插入驱动器等），BIOS 可能无法读取磁盘对我们而言，了解如何检测到这一点很重要，否则，我们可能会认为我们已经读取了一些数据，但实际上目标地址只包含于发出 read 命令之前相同的的随机字节。对我们来说幸运的是，BIOS 更新了一些寄存器以让我们知道发生了什么，carry flag（特殊标志寄存器 CF）设置为发出一般故障信号，而 al 设置为实际读取的扇区数，而不是请求的扇区数目，发出 BIOS 中断后，我们可以执行以下简单检查：

```
...
...
int 0x13          ; Issue the BIOS interrupt to do the actual read
jc disk_error     ; jc is another jumping instruction , that jumps
                  ; only if the carry flag was set
```

```
; This jumps if what BIOS reported as the number of sector
; actually read in AL is not equal to the number we expected
cmp al, <no. sectors expected>
jne disk_error

disk_error:
mov bx, DISK_ERROR_MSG
call print_string
jmp $

; Global variables
DISK_ERROR_MSG: db "Disk read error!",
```

3.6.4 综合使用

如前所属，能够从磁盘读取更多的数据对于引导我们的操作系统至关重要，所以这里我们将本节中的所有思想放入一个有用的例程中，该例程将从指定的磁盘设备读取引导扇区之后的 n 个扇区。

```
; load DH sectors to ES:BX from drive DL
disk_load:
  push dx          ; Store DX on stack so later we can recall
                   ; how many sectors were request to be read,
                   ; even if it is altered in the meantime
  mov ah, 0x02     ; BIOS read sector function
  mov al, dh       ; Read DH sectors
  mov ch, 0x00     ; Select cylinder 0
  mov dh, 0x00     ; Select head 0
  mov cl, 0x02     ; Start reading from second sector (i.e.
                   ; after the boot sector)
  int 0x13         ; BIOS interrupt

  jc disk_error    ; Jump if error (i.e. carry flag set)

  pop dx           ; Restore DX from the stack
  cmp dh, al       ; if AL (sectors read) != DH (sectors expected)
  jne disk_error   ;   display error message
  ret

disk_error :

  mov bx, DISK_ERROR_MSG
  call print_string
  jmp $

; Variables
DISK_ERROR_MSG  db "Disk read error!", 0
```

为了测试这个程序，我们可以编写一个引导扇区程序，如下所示：

```
; Read some sectors from the boot disk using our disk_read function
[org 0x7c00]

  mov [BOOT_DRIVE], dl ; BIOS stores our boot drive in DL, so it's
                       ; best to remember this for later.

  mov bp, 0x8000       ; Here we set our stack safely out of the
  mov sp, bp           ; way, at 0x8000

  mov bx, 0x9000       ; Load 5 sectors to 0x0000(ES):0x9000(BX)
  mov dh, 5            ; from the boot disk.
  mov dl, [BOOT_DRIVE]
  call disk_load

  mov dx, [0x9000]     ; Print out the first loaded word, which
  call print_hex       ; we expect to be 0xdada, stored
                       ; at address 0x9000

  mov dx, [0x9000 + 512] ; Also, print the first word from the
  call print_hex         ; 2nd loaded sector: should be 0xface

  jmp $

%include "../print/print_string.asm" ; Re-use our print_string function
%include "../hex/print_hex.asm"      ; Re-use our print_hex function
%include "disk_load.asm"
; Include our new disk_load function

; Global variables
BOOT_DRIVE: db 0

; Bootsector padding
times 510-($-$$) db 0
dw 0xaa55

; We know that BIOS will load only the first 512-byte sector from the disk
; so if we purposely add a few more sectors to our code by repeating some
; familiar numbers, we can prove to ourselfs that we actually loaded those
; additional two sectors from the disk we booted from.
times 256 dw 0xdada
times 256 dw 0xface
```

4、进入 32 位保护模式

继续在 16 位实模式下工作是很好的，我们现在已经很好地适应，但是为了更加充分地利用 CPU，并且为了更好地理解 CPU 体系架构的发展如何有利于现代操作系统，即硬件中的内存保护，那么我们必须进入 32 位保护模式。

32 位保护模式的主要区别是：

- 寄存器被扩展到 32 位，通过寄存器名称前面加上一个前缀来访问它们的全部容量，例如：mov ebx,0x274fe8fe
- 为了方便起见，还有两个附件的通用寄存器，fs 和 gs。
- 32 位内存偏移量可用，因此偏移量可以引用高达 4GB 的内存（0xFFFFFFFF）
- CPU 支持一种更为复杂的内存分割方法，它有两大优势。
    1. 可以禁止一个代码段中的代码在特权更高的代码中执行代码，因此您可以保护内核免受用户程序攻击。
    2. CPU 可以实现为用户进程实现虚拟内存，这样进程内存的页面（即固定大小的块）可以根据需要在磁盘和内存直接透明的交换。这样可以确保有效地使用内存，因为不经常执行的代码或数据不需要占用宝贵的内存。
- 中断处理也更加复杂。

将 CPU 从 16 位实模式切换到 32 位保护模式最困难的部分是我们必须在内存中准备一个称为全局描述符表（global descriptor table

GDT）的复杂数据结构，它定义内存段及其保护模式属性。一旦我们定义了 GDT，我们就可以使用个特殊的指令将它加载到 CPU 中，然后在一个特殊的 CPU 控制寄存器中设置一个位来进行实际的切换。

如果我们不必用汇编定义 GDT，那么此过程将很容易，但是遗憾的是，如果我们以后希望加载使用高级语言（例如：C）编译的内核，那么这种低级的转换是不可避免的，因为它通常被编译为 32 位指令，而不是效率低下的 16 位指令。

噢，我几乎忘了提到一个令人震惊的事情：一旦切换到 32 位保护模式，我们将无法再使用 BIOS。如果您认为 BIOS 调用是低级的。就想退一步海阔天空。

4.1 适应没有 BIOS 的生活

这是真的：为了充分利用 CPU，我们必须放弃 BIOS 提供所有有用的例程。当我们更详细了解 32 位保护模式切换时，我们将看到，BIOS 例程被编译为仅在 16 位实模式下工作，但在 32 位保护模式下不再有效，事实上，尝试使用它们可能会使机器崩溃。

所以意味着 32 位的操作系统必须为机器的所有硬件（例如：键盘、屏幕、鼠标、磁盘驱动器等）提供自己的驱动程序。实际上，对于一个 32 位保护模式的操作系统来说，可以暂时切回 16 位模式

是允许的，因此它可以使用 BIOS，但这种技术可能比它的价值更糟糕，尤其是性能方面。

我们切换到保护模式时遇到的第一个问题是知道如何在屏幕上打印消息，这样我们就可以看到发生了什么。以前我们要求 BIOS 在屏幕上打印 ACSII 字符，但是这是如何导致在计算机屏幕的适当位置突出显示适当的像素呢？现在，只需要知道显示设备可以配置为文本模式和图形模式中的一种，并且屏幕上显示的内容时特定内存范围的视距表示。所以，为了操纵屏幕，我们必须操作它在当前模式下使用的特定内存范围。显示设备是内存映射硬件的一个例子，因为它是这样工作的。

当大多数计算机启动时，尽管它们可能实际上有更先进的图形集成电路硬件，但它们都是从一个简单的视频图形阵列（VGA）彩色文本模式演进的，具有 80x25 字符尺寸。在文本模式下，程序员不需要渲染单个像素来描述特定的字符，因为 VGA 显示设备的内存中已经定义了一个简单的字体。相反，屏幕上的每个字符单元都由内存中的两个字节表示：第一个字节是要显示的字符 ASCII 代码，第二个字节编码字符属性，例如前景和背景颜色以及字符是否应该闪烁。

因此，如果我们想在屏幕上显示一个字符，那么我们需要将它的 ASCII 代码和属性设置为当前 VGA 模式的正确内存地址，通常是地址 0xb8000。如果我们稍微修改一下原来的（16 位实模式）print_string

例程，使其不再使用 BIOS 例程，我们可以创建一个 32 位保护模式例程，直接写入视频内存，如图 4.1 所示。

请注意，虽然屏幕显示为列和行，但视频内存是有序的，例如，第 3 行第 5 列的地址可以计算如下：0xb8000+2*(row*80+col)

```
[bits 32]
; Define some constants
VIDEO_MEMORY equ 0xb8000
WHITE_ON_BLACK equ 0x0f

; prints a null-terminated string pointed to by EDX
print_string_pm:
  pusha
  mov edx, VIDEO_MEMORY  ; Set edx to the start of vid mem.

print_string_pm_loop:
  mov al, [ebx]          ; Store the char at EBX in AL
  mov ah, WHITE_ON_BLACK ; Store the attributes in AH

  cmp al, 0              ; if (al == 0), at end of string, so
  je done                ; jump to done

  mov [edx], ax          ; Store char and attributes at current
                         ; character cell.
  add ebx, 1             ; Increment EBX to the next char in string.
  add edx, 2             ; Move to next character cell in vid mem.

  jmp print_string_pm_loop  ; loop around to print the next char.

print_string_pm_done :
  popa
  ret                    ; Return from the function
```

图 4.1 将字符串直接打印到视频内存的例程（即不使用 BIOS）

我们例程的缺点是，它总是将字符打印到屏幕左上角，因此覆盖以前的消息，而不是滚动显示。我们可以花点时间来增加此汇编例程的复杂性，但是我们不要让事情变的太困难，因为我们掌握了转换到保护模式之后，我们很快就会启动用高级语言编写代码，在哪里我们可以轻松完成这些工作。

4.2 了解 GDT

在我们深入研究细节之前，重要的是了解这个 GDT 重点。这是保护模式允许的基础。回想一下第 3 节，在传统 16 位实模式中，基于段寻址的设计原理是允许程序员访问（尽管按今天的标准，这要稍微多点）超过 16 位偏移量所允许的内存。例如，假设程序员希望将 ax 的值存储在地址 0x4fe56。如果没有分段寻址，程序员所能做的最好就是这样了：

```
mov [0xffff], ax
```

离预定地址有点远，如果使用段寄存器，可以实现如下：

```
mov bx, 0x4000
mov es, bx
mov [es:0xfe56], ax
```

虽然分割内存并使用偏移量达到这些段的总体思想保持不变，但在保护模式下实现的方式已经完全改变，主要是为了提供更多的灵活性。一旦 CPU 被切换到 32 位保护模式，它将逻辑地址（即段寄存器和偏移量的组合）转换为物理地址的过程就完全不同了：不是将段寄存器的值乘以 16，然后在加上偏移量，段寄存器成为 GDT 中特定段描述符（SD）的索引。

段描述符是一个 8 字节的结构，它定义了保护模式段的一下属性。

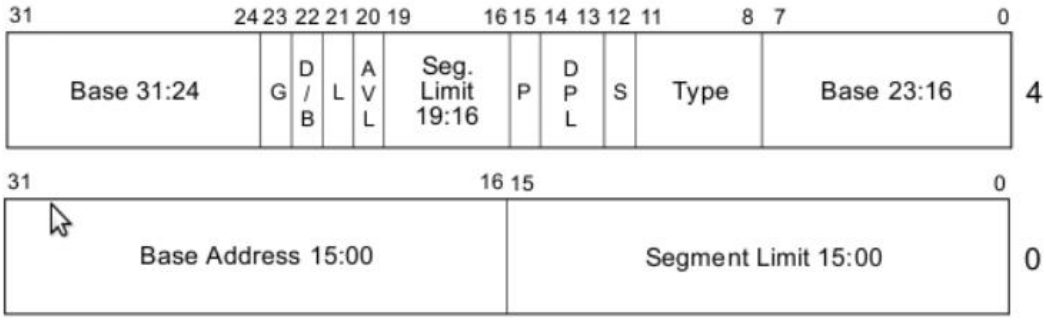
- 基地址（32 位），它定义了段在物理内存的起始地址。
- 段限制（20 位），它定义了段的大小
- 各种 flags，这会影响 CPU 如何解释段，例如在它内部运行的代码的特权级别，或者它是只读还是只写的。

图 4.2 展示了段描述符实际结构，请注意，整个结构很混乱，例如，段限制的低 16 位在结构的前面两个字节中，而较高的 4 位在机构的第 7 个字节的开头，也许这是一个玩笑，或者更新是它有历史渊源，或者是受到 CPU 硬件设计影响。

我们关心段描述符的所有可能配置的细节，有关详细说明，请参阅《英特尔开发人员手册》，但我们将学习如何使代码在 32 位保护模式下运行。

英特尔段寄存器的最简单可行配置描述为基本平坦模型，其中定义了两个重叠的段，这些段覆盖了整个 4GB 的可寻址内存，一个用于代码，另一个用于数据。在此模型中，这两个段是重叠的，这意味着没有尝试保护一个段免受另一个段的侵害，也没有任何尝试将分页功能用于虚拟内存，在早期保持最简单是值得的，特别是在以后，一旦我们进入了一种更高级的语言，我们可能会更容易地改变段描述符。

除了代码和数据段，CPU 还要求 GDT 中的第一个条目故意设置为无效的 null 描述符（即 8 个 0 字节的结构），null 描述符是一种简单的机制，用于捕捉在访问地址之前忘记设置特定段寄存器的错误，如果我们将一些段寄存器设置为 0x0，但在切换到保护模式后忘记将他们更新为适当的段描述符，则可以轻松完成此操作。如果使用 null 描述符进行寻址，那么 CPU 将引发异常，这实际上是一个中断——虽然概念上不太相似，但不要将其与高级语言（例如 Java）中的异常混淆。



L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

图 4.2 段描述符结构

我们的代码段将具有如下配置：

- 基址（Base）：0x0
- 界限（Limit）：0xfffff
- 段存在标志（P）：1 段存在内存中——用于虚拟内存
- 特权 DPL（DPL）：0 最高特权
- 描述符类型（S）：1 表示代码或数据段，0 表示陷阱
- 段类型（Type）：
  1. 代码：1 代表代码，因为这是一个代码段
  2. 一致性（C）：0，不包含意味着具有较低权限的段中的代码不能调用该段中的代码——这是保护内存的关键
  3. 可读（R）：1 可读，0 可执行

4. 已访问（A）：0 通常用于调试和虚拟内存技术，因为 CPU 在其访问时将其至为 1

- 其他标志

1. 颗粒度（G）：该字段用于确定段限长字段 Limit 值的单位，如果颗粒度标志为 0，则段限长值的单位是字节；如果为 1，则段限长值使用 4KB 单位。允许您的段跨域 4GB 内存。
2. 32 位默认值（D/B）：1 表示 32 位代码和数据段，0 表示 16 位代码和数据段，这实际上设置了操作的默认数据单元大小（例如，push 0x4 将扩展为 32 位数字，依次类推）
3. 64 位代码段（L）：0 在 32 位处理器上未使用。
4. 系统软件可用位（AVL）:0，我们可以将其设置为自己使用（例如调试），但我们不会使用它。

由于我们使用的是简单的平面模型，具有两个重叠的代码和数据段，因此改数据段将是相同，但类型（Type）标记为：

- 代码：0 表示数据段
- 向下扩展（E）：运行细分向下扩展——稍后我们对此进行解释
- 可写的（W）：1 允许数据段可写，否则它将是只读。
- 已访问（A）：0 这通常用于调试和虚拟内存技术，因为 CPU 在访问段时会设置改为。

现在我们已经看到了两个段的实际配置，探索了大多数可能的段描述符设置，那么应该更加清楚的是，保护模式在内存分配方面比实模式具有更大的灵活性。

4.3 在汇编中定义 GDT

现在我们已经了解了基本平面模型的 GDT 中要包含哪些段描述符，让我们看看如何在汇编中实际地表示 GDT，这是一项比任何其他任务都需要更多耐心的任务，当您在体验这件事的枯燥时，请记住它的重要性，我们在这里所做的将允许我们很快启动我们的操作系统，我们将用一种更高级的语言编写它，——我们的一小步将变成巨大的飞跃。

我们已经看到了如何使用 db、dw 和 dd 汇编指令在汇编代码中定义数据的示例，而这些正是我们必须用来将适当的字节放置在 GDT 的描述符条目中的示例。

实际上，由于 CPU 需要知道 GDT 的长度，我们实际上并没有直接給 CPU 提供 GDT 的起始地址，而是给它一个称为 GDT 描述符的简单结构的地址（即描述符 GDT 的东西），GDT 是一个 6 字节结构，包含：

- GDT 大小（16 位）
- GDT 地址（32 位）

请注意，当使用像这样的复杂数据结构的低级语言时，我们不能添加足够的注释。一下代码定义了我们的 GDT 和 GDT 描述符；在代码中，请注意我们如何使用 db，dw 等来填充结构的各个部分，以及使用 b 为后缀定义二进制数，可以更方便的定义具体的位：

```
; GDT
gdt_start:

gdt_null: ; the mandatory null descriptor
  dd 0x0  ; 'dd' means define double word (i.e. 4 bytes)
  dd 0x0

gdt_code: ; the code segment descriptor
  ; base=0x0, limit=0xfffff,
  ; 1st flags: (present)1 (privilege)00 (descriptor type)1 -> 1001b
  ; type flags: (code)1 (conforming)0 (readable)1 (accessed)0 -> 1010b
  ; 2nd flags: (granularity)1 (32-bit default)1 (64-bit seg)0 (AVL)0 -> 1100b
  dw 0xffff     ; Limit (bits 0-15)
  dw 0x0        ; Base (bits 0-15)
  db 0x0        ; Base (bits 16-23)
  db 10011010b ; 1st flags, type flags
  db 11001111b ; 2nd flags, Limit (bits 16-19)
  db 0x0        ; Base (bits 24-31)

gdt_data: ;the data segment descriptor
  ; Same as code segment except for the type flags:
  ; type flags: (code)0 (expand down)0 (writable)1 (accessed)0 -> 0010b
  dw 0xffff     ; Limit (bits 0-15)
  dw 0x0        ; Base (bits 0-15)
  db 0x0        ; Base (bits 16-23)
  db 10010010b ; 1st flags, type flags
  db 11001111b ; 2nd flags, Limit (bits 16-19)
  db 0x0        ; Base (bits 24-31)

gdt_end:        ; The reason for putting a label at the end of the
                ; GDT is so we can have the assembler calculate
                ; the size of the GDT for the GDT decriptor (below)

; GDT descriptior
gdt_descriptor:
  dw gdt_end - gdt_start - 1  ; Size of our GDT, always less one
                              ; of the true size
  dd gdt_start                ; Start address of our GDT

; Define some handy constants for the GDT segment descriptor offsets, which
; are what segment registers must contain when in protected mode. For example,
; when we set DS = 0x10 in PM, the CPU knows that we mean it to use the
; segment described at offset 0x10 (i.e. 16 bytes) in our GDT, which in our
; case is the DATA segment (0x0 -> NULL; 0x08 -> CODE; 0x10 -> DATA)
CODE_SEG equ gdt_code - gdt_start
DATA_SEG equ gdt_data - gdt_start
```

4.4 进行切换

一旦 GDT 和 GDT 描述符都在引导扇区内准备好，我们就可以指示 CPU 从 16 位实模式切换到 32 位保护模式。

正如我之前说过的那样，实际的转换是相当直接的代码，但是理解所涉及步骤的重要性是很重要的。

我们需要做的第一件事是使用 cli（clear interrupt）指令禁用中断，这意味着 CPU 会简单地忽略将来可能发生的任何中断，至少要等到以后启用中断前。这一点非常重要，因为与基于段寻址一样，保护模式下的中断处理与实模式下的中断处理完全不同，这使得 BIOS 在内存启动处设置的当前 IVT（中断向量表）完全没有意义。即使 CPU 可以将中断信号映射到其正确的 BIOS 例程（例如，用户按下某键时，将其值存储在缓存中）BIOS 例程也会执行 16 位代码，它将没有我们在 GDT 中定义的 32 位段的概念，因此将采用 16 位实模式分段方案的段寄存器值，最终会导致 CPU 奔溃。

下一步是告诉 CPU 我们刚刚准备好的 GDT——非常痛苦，我们使用一条指令来执行此操作，并向其传递 GDT 描述符。

```
lgdt [gdt_descriptor]
```

现在一切就绪，我们通过设置特殊 CPU 控制寄存器 cr0 的第一位来进行实际切换。现在，我们不能直接在寄存器上设置该位，因此必须将其加载到通用寄存器中，设置该位，然后将其存储回 cr0。与我们前面章节中使用 and 指令从值中排除位的方式类似，我们可以使

用 or 指令将某些位包含到值中（即，不干扰由于某些重要原因可能已经在控制器中设置的任何其他位）。

```
mov eax , cr0 ; To make the switch to protected mode , we set
or eax , 0x1  ; the first bit of CR0 , a control register
mov cr0 , eax ; Update the control register
```

cr0 更新后，CPU 处于 32 位保护模式。

最后的说法并不完全正确，因为现代处理器使用了一种称为流水线的技术，该技术使它们可以并行处理指令执行的不同阶段（我说的是单个 CPU，而不是并行 CPU），因此用的时间更少。例如，可能从内存中提取每条指令，将其解码为微代码指令，然后执行，然后将结果存储回内存；并且由于这些阶段是半独立的，因此它们都可以在同一 CPU 周期内完成，但在不同的电路中（例如，前一条指令可以被解码，而下一条指令可以被提取）。

在对 CPU 编程时，我们通常不必担心诸如流水线之类的 CPU 内部问题，但切换 CPU 模式是一种特殊情况，因为存在 CPU 可能以错误的模式处理指令执行的某些阶段的风险。在指示 CPU 切换模式之后，我们需要立即执行的是迫使 CPU 完成其管道中的所有作业，一便我们可以确信所有未来的指令都将在正确的模式下执行。

现在，当 CPU 知道接下来几条即将到来的指令时，流水线工作的非常好，应为它可以预先获取它们，但它不喜欢像 jmp 或 call 这样的指令，应为这些指令被完全执行之前，CPU 根本不知道接下来的指令是什么，特别是如果我们使用远跳转或调用。这意味着我们跳转到另一个段。因此，在指令 CPU 切换模式后，我们可以立即发

出一个远跳转，这将迫使 CPU 刷新管道（即完成当前处于管道不同阶段的所有指令）。

为了发出远跳转，而不是近跳转（即标准跳转），我们额外提供目标段，如下所示：

```
jmp <segment >:<address offset>
```

对于这次跳跃，我们需要仔细考虑我们希望在哪里着陆。假设我们在代码中定义了一个标签，比如 start_protected_mode，它标记 32 位代码的开始。正如我们刚刚讨论过的，一个近跳转，比例 jmp start_protected_mode 可能不足以刷新管道，而且此外，我们现在处于某种奇怪的边缘，因为我们当前的代码段（即 cs）在保护模式下失效。因此，我们必须将 cs 寄存器更新为 GDT 代码段描述符的偏移量。由于段描述符的长度均为 8 字节，并且由于代码段描述符是 GDT 中的第二个描述符（第一个是 null 描述符），它的偏移量将是 0x8,因此这个值就是我们现在必须将代码段寄存器设置的值。注意，根据长跳的定义，它将自动导致 CPU 将 cs 寄存器更新到目标段。为了方便地使用标签，我们让我的汇编程序计算这些段描述符偏移量，并将它们存储为常量 CODE_SEG 和 DATA_SEG，因此现在我们可以得出跳转指令。

```
jmp CODE_SEG:start_protected_mode

[bits 32]

start_protected_mode:
  ... ; By now we are assuredly in 32-bit protected mode.
  ...
```

请注意，实际上，就我们从跳跃地点到着陆地点之间的物理距离而言，我们根本不需要跳得太远，但重要的是我们的跳跃方式。

还请注意，我们需要使用[bits 32]指令来告诉汇编器，从那时起，它应以 32 位模式指令进行编码。但是请注意，这并不意味着我们不能在 16 实模式下使用 32 位指令，只是汇编程序对这些指令的编码必须与 32 位保护模式略有不同。的确，当切换到保护模式时，我们利用 32 位寄存器 eax 来设置控制为。

现在我们进入 32 位保护模式，一旦正确进入 32 为模式，要做的一件好事情就是更新所有其他段寄存器，这样它们就是指向我们的 32 位数据段（而不是现在无效的实模式段）并更新栈的位置。

我们可以将整个过程组合成一个可重用的过程，如下所示。

```
[bits 16]
; Switch to protected mode
switch_to_pm:
    cli                 ; We must switch of interrupts until we have
                        ; set -up the protected mode interrupt vector
                        ; otherwise interrupts will run riot.
    lgdt [gdt_descriptor] ; Load our global descriptor table ,
                          ; which defines
                          ; the protected mode segments
                          ;(e.g. for code and data)
    mov eax , cr0       ; To make the switch to protected mode , we set
    or eax , 0x1        ; the first bit of CR0 , a control register
    mov cr0 , eax

    jmp CODE_SEG:init_pm ; Make a far jump (i.e. to a new segment)
                         ; to our 32-bit
                         ; code. This also forces the CPU to flush
                         ; its cache of
                         ; pre -fetched and real -mode
                         ; decoded instructions , which can
                         ; cause problems
[bits 32]
```

```
; Initialise registers and the stack once in PM.
init_pm:

    mov ax, DATA_SEG  ; Now in PM, our old segments are meaningless
    mov ds, ax        ; so we point our segment registers to the
    mov ss, ax ; data selector we defined in our GDT
    mov es, ax
    mov fs, ax
    mov gs, ax

    mov ebp , 0x90000 ; Update our stack position so it is right
    mov esp , ebp ; at the top of the free space.

    call BEGIN_PM ; Finally , call some well -known label
```

4.5 综合使用

最后，我们可以将所有的例程都包含到一个引导扇区中，该扇区演示了从 16 位实模式到 32 位保护模式的切换。

```
; A boot sector that enters 32-bit protected mode
[org 0x7c00]
    mov bp, 0x9000 ; Set the stack.
    mov sp, bp

    mov bx, MSG_REAL_MODE
    call print_string

    call switch_to_pm ; Note that we never return from here.

    jmp $

%include "../print/print_string.asm
%include "gdt.asm"
%include "print_string_pm.asm"
%include "switch_to_pm.asm"

[bits 32]
; This is where we arrive after switching to and initialising
; protected mode.
BEGIN_PM:
```

```
    mov ebx , MSG_PROT_MODE
    call print_string_pm     ; Use our 32-bit print routine

    jmp $                    ; Hang

; Global variables
MSG_REAL_MODE db "Started in 16-bit Real Mode",0
MSG_PROT_MODE db "Successfully landed in 32-bit Protected Mode",0

; Bootsector padding
times 510-($-$$) db 0
dw 0xaa55
```

5、编写，构建加载内核

到目前为止，通过使用低级汇编语言与计算机进行通信，我们已经了解了许多计算机的实际工作原理，但是我们也看到了使用这种语言进行计算机开发的速度非常慢，我们需要思考即使是最简单的控制结构（例如 if(something)<do this>else<do that>）也需要非常的小心。我们不得不担心如何最好地利用有限数量的寄存器，已经如何处理堆栈。我们继续使用汇编的另一个缺点是，它与特点的 CPU

架构紧密相关，因此，将操作系统移植到另一个 CPU 架构（例如 ARM，RISC，PowerPC）将变得更加困难。

幸运的是，其他程序员也厌倦了汇编语言的编写，因此决定编写更高级的语言编译器（例如 FORTRAN，C，Pascal,BASIC 等）将更直观的代码转换为汇编语言。这些编译器的想法是将更高级别的构造（例如控制器结构和函数调用）映射到程序集模板代码上，因此缺点（通常总是存在不利之处）是泛型模板对于特定的功能可能并不是最佳的。事不宜迟，让我们看下如何将 C 代码转换为汇编语言，从而使编译器的角色更加神秘。

5.1 了解 C 编译

让我们用 C 写一些小的代码段，看看它们生产什么样的汇编代码，这是学习 C 如何工作的一个很好的方法。

5.1.1 生成原始机器代码

```
// Define an empty function that returns an integer
int my_function () {
    return 0xbaba;
}
```

将代码保存到名为 basic.c 的文件中，并按如下所示进行编译：

```
$gcc -ffreestanding -c basic.c -o basic.o
```

这将生成一个与本机环境完全无关目标文件，请勿与面向对象编程的概念混淆。编译器不是直接编译机器代码，而是输出带注释的机器代码，其中保留了对于执行而言多余的元信息（例如文本标签），以使最终代码的组合方式更具有灵活性。这种中间格式的一大优点是，当与其他库中的其他例程链家时，代码可以轻松地重定位到更大的二进制文件中，因为目标文件中的代码使用相对的而不是绝对的内部内存引用。通过以下命名，我们可以很好的了解目标文件内容：

```
$objdump -d basic.o
```

此命令的输出将显示如下图所示的内容，注意，我们看到一些汇编指令和一些关于代码的附加信息，请注意：程序集的语法与 nasm 所使用的的语法略有不同，因此请忽略这一部分，因为我们很快就会看到一种更熟悉的格式。

```
basic.o:     file format elf32-i386


Disassembly of section .text:

00000000 <my_function>:
   0: 55                    push   %ebp
   1: 89 e5                 mov    %esp,%ebp
   3: b8 ba ba 00 00        mov    $0xbaba,%eax
   8: 5d                    pop    %ebp
   9: c3                    ret
```

为了创建实际的可执行代码（即将在我们的 CPU 上运行的代码），我们必须使用链接器，该链接器的作用是将输入对象文件中描述的所有例程链接到一个可执行的二进制文件中，从而有效地将它们拼接到一起，并将这些相对地址转换为聚合代码中的绝对地址，

例如：call <function_x_label> 建变为 call 0x12345，其中的 0x12345 是链接程序决定放置由 function_x_label 表示的例程代码的输出文件中的偏移量。

不过，在我们的例子中，我们不想与任何其他目标文件中的任何例程链接——我很快将讨论这个问题——但无论如何，链接器会将带注释的机器代码文件转换为原始机器代码文件。将原始机器代码输出到文件 basic.bin 中，我们可以使用以下命令：

```
$ld -o basic.bin -Ttext 0x0 --oformat binary basic.
```

请注意，与编译器一样，连接器可以输出各种格式的可执行文件，其中一些格式可能会保留输入对象文件中的元数据。这对于由操作系统托管的可执行文件非常有用，例如我们将在 Linux 或 Windows 等平台上编写的大部分程序，因为可以保留元数据来描述如何将这些应用程序加载到内存中，以及出于调试目的，例如：进程在指令地址 0x12345678 处崩溃的信息，对程序员而言，远不及使用冗余的，不可执行的的元数据在函数有用，例如 my_function filebasic.c 的第三行。

无论如何，既然我们对编写操作系统感兴趣，那么在我们的 CPU 上运行与元数据混合的机器代码是没有好处的，因为不知道 CPU 会将每个字节都作为机器代码来执行。这就是我们为什么指定（raw）binary 的输出格式

我们使用的另一个选项-Ttext 0x0，它的工作方式与我们在早期的汇编例程中使用 org 指令相同，它允许我们告诉编译器在代码中的偏移地址（例如，对于我们在代码中指定的任何数据，例如

“hello”），当以后加载到内存中某个特定地址时，返回它们的绝对内存地址。现在这并不重要，但是当我们要讲内核加载到内存时，将其设置为我们计划加载到的地址是很重要的。

现在我们已经成功地将 C 代码编译成一个原始的机器文件，我们可以（一旦知道如何加装它）在我们的 CPU 上运行，那么让我们看看它是什么样子。幸运的是，由于汇编映射到机器代码指令非常接近，所以如果给你一个只包含机器代码的文件，你可以很容易地反汇编它，在汇编中查看它。是的，这是了解一些汇编的另一个好处，因此你可以对任何一个不包含原始代码的软件进行逆向工程，如果开发人员给你留下了一些元数据的话——他们几乎总是这样做的，那就更成功了，分解机器代码的唯一问题是，其中一些字节可能被保留为数据，但将显示为汇编指令，尽管在我们的简单 C 程序中，我们没有声明任何数据。要查看编译器实际从 C 源代码生成的机器代码，请运行一下命令：

```
$ndisasm -b 32 basic.bin > basic.dis
```

-b 32 只是告诉汇编程序将其解码为 32 位汇编指令，这是我们的编译器生成的。下图显示了 gcc 为我们的简单 C 程序生成汇编代码。

```
00000000  55              push ebp
00000001  89E5            mov ebp,esp
00000003  B8BABA0000      mov eax,0xbaba
00000008  5D              pop ebp
00000009  C3              ret
```

这里是这样的 ： gcc 生成了一些汇编代码，与我们自己编写的代码不太相似。反汇编器从左至右输出的三列指令，机器代码和等效

的汇编指令的文件偏移量。尽管我们的函数做的很简单，但是其中有些附加代码似乎正在操作栈的基址和栈顶寄存器 ebp 和 esp，C 大量使用栈来存储函数本地的变量（即函数返回时不再需要的变量），因此在输入函数时，栈的基址指针（ebp）会 push 到当前栈顶（mov ebp,esp），有效地在调用我们函数的函数栈上方创建一个本地的，最初为空的栈。此过程通常称为设置栈帧的函数，在这个过程中，它将分配任何本地的变量，但是，如果在从函数返回之前，我们未能将栈帧还原到调用方最初设置的栈帧，则调用函数将尝试访问其局部变量时陷入混乱，因此，在更新栈帧的基指针之前，我们必须存储它，而且没有比栈顶（push ebp）更好的存储它的地方了。

在准备好栈帧之后，不幸的是，在我们的简单函数中没有实际使用，我们看到了编译器如何处理返回行 0xbaba;值 0xbaba 存储在 32 位寄存器 eax 中，这是调用函数（如果有）期望找到返回值的地方，类似于我们使用某些寄存器将参数传递给我们早期的汇编例程的惯例，对于我们的示例，pring_string 例程将查找在寄存器 bx 中打印的字符串地址。

最后，在发出 ret 返回给调用者之前，该函数将原始栈基址指针从栈中弹出（pop ebp）,因此调用函数将不会意识到自己的栈帧已被调用函数更改。请注意，我们实际上并没有更改栈的顶部（esp）。因为在本例，我们的栈帧未存储任何内容，因此未修改的 esp 寄存器不想哟啊还原。

现在我们对 C 代码如何转换为汇编有了一个好主意，因此让我们进一步介绍一下编译器，直到我们充分了解如何用 C 编写一个简单的内容为止。

5.1.2 局部变量

现在，将如下代码写入一个名为 local_var.c 的文件，并像以前一样对其进行编译，链接和反汇编。

```
// Declare a local variable.
int my_function() {
  int my_var = 0xbaba;
  return my_var;
}
```

现在，编译器将生成如下的汇编代码

```
00000000  55                push ebp
00000001  89E5              mov ebp,esp
00000003  83EC10            sub esp,byte +0x10
00000006  C745FCBABA0000    mov dword [ebp-0x4],0xbaba
0000000D  8B45FC            mov eax,[ebp-0x4]
00000010  C9                leave
00000011  C3                ret
```

现在唯一的区别是，我们实际上分配了局部变量 my_var,但是这引起了编译器的有趣响应，和以前一样，栈帧已经建立了，但随后我们看到了 sub esp,byte+0x10，它从栈顶部减去 16（0x10）个字节。首先，我们必须不断提醒自己，栈在内存地址方面向下增长，因此，用更简单的术语来说，该指令意味着在栈顶部再分配 16 个字节。我们正在存储一个 4 字节（32 位）数据类型的 int，那么为什么栈上为此变量分配了 16 个字节，为什么不使用 push 来自动分配新

的栈空间呢？编译器以这种方式操作栈的愿意是一种优化，因为 CPU 常常无法在未对齐数据类型大小的内存边界上对齐的数据类型上有效地工作。由于 C 希望所有变量都正确对齐，因此它对所有栈元素都是用最大数据类型宽度（即 16 个字节），以浪费一些内存为代价。

下一条指令 mov dword[ebp-0x4],0xbaba 实际上将变量的值存储在栈上新分配的空间中，但由于先前给定的栈效率原因（即说存储数据类型的大小）而未使用 push 小于保留的栈空间。我们了解 mov 指令的一般用法，但是这里需要说明的两件事是 dword 和[ebp-0x4]的使用：

- dword 明确指出，我们在栈上存储一个双字（即 4 个字节），这是 int 数据类型的大小，所以实际存储的字节应该是 0x0000baba，但是如果不显式的话，很容易是 0xbaba（即 2 个字节）或 0x000000baba（即 8 个字节），虽然值相同，但范围不同。
- [ebp-0x4]是称为有效地址计算的现代 CPU 快捷方式的一个示例，汇编代码似乎反映了这一点。给人留下了深刻的印象。指令的这一部分引用了由计算机即时计算的地址。CPU，基于寄存器 ebp 的当前地址。乍一看，我们可能认为我们的汇编器正在操纵一个常量值，就像我们写 mov ax,0x5000+0x20 这样的。我们的汇编器将其简单地预处理为 mov ax,0x5020 一样。但是，只是运行代码后，任何寄存器的值才能知道。因

此这绝对不是预处理；它构成了 CPU 指令的一部分。通过这种寻址方式，CPU 使我们可以在每个指令周期内做更多的事情，并且是 CPU 硬件如何更好地使用程序员的一个很好的例子。在下面的三行代码中，如果没有这种地址操纵，我们可能会写效率较低的等效代码：

```
mov eax, ebp       ; EAX = EBP
sub eax, 0x4       ; EAX = EAX - 0x4
mov [eax], 0xbaba ; store 0xbaba at address EAX
```

因此，值 0xbaba 直接存储在栈的适当位置，这样它将占据基指针的上方（尽管实际位于下方，因为栈向下增长）前四个字节。

现在，作为一个计算机程序，我们的编译器可以很容易地区分不同的数字，就像我们可以区分不同的变量名一样，所以我们认为的变量 my_var，编译器将认为是地址 ebp-0x4（即栈的前 4 个字节）。我们在下一条指令 mov eax,[ebp-0x4]中看到了这一点，这意味着将“my_var 的内容存储在 eax 中”，同样使用高效的地址计算；并且我们从前面的函数中知道，eax 用于将变量返回给函数的调用者。

现在，在 ret 指令之前，我们看到了新的东西，leave 指令，实际上 leave 指令是一下步骤的代替方法，该步骤恢复了调用方的原始栈：

```
mov esp , ebp ; Put the stack back to as we found it.
pop ebp
```

尽管只有一条指令，但并非总是 leave 比单独的指令更有效，因为我们的编译器选择了使用此指令。所以我们将把这个特定的讨论留给其他人。

5.1.3 函数调用

我们看看如下的 C 代码，它有两个函数，其中一个函数 caller_function 调用另一个函数 callee_function，并为其传递了一个整数参数。被调用函数仅返回其传递的参数。

```
void caller_function() {
  callee_function(0xdede);
}

int callee_function(int my_arg) {
  return my_arg;
}
```

如果我们编译和反汇编 C 代码，将得到如下内容。

```
00000000  55                push ebp
00000001  89E5              mov ebp,esp
00000003  83EC08            sub esp,byte +0x8
00000006  C70424DEDE0000    mov dword [esp],0xdede
0000000D  E802000000        call dword 0x14
00000012  C9                leave
00000013  C3                ret
00000014  55                push ebp
00000015  89E5              mov ebp,esp
00000017  8B4508            mov eax,[ebp+0x8]
```

```
0000001A  5D                pop ebp
0000001B  C3                ret
```

首先，请注意如何通过查找总是作为函数的最后一条指令 ret 指令来区分两个函数的汇编代码。接下来请注意上层函数如何使用汇编指令调用。我们知道 call 汇编指令将用于跳转到通常希望从中返回的另一个例程。这必须是我们的 caller_function，即在机器代码的

偏移量 0x14 处调用 caller_function。这里最有趣的行是调用前的行，因为它们一某种方式确保将参数 my_arg 传递给了 callee_function。建立栈框架以后。如我们所见所有函数都通用。Caller_function 在栈顶部分配 8 个字节（sub esp,byte+0x8），然后将传递的值 0xdade 存储到栈空间中（mov dword[exp],0xdede）。

因此，让我们么看看 callee_function 如何访问该参数，从偏移量 0x14 开始，我们看到了 callee_function 像往常一样建立了栈帧，但随后查看了它存储在 eax 寄存器中的内容，从我们先前分析中知道，该寄存器用于保存函数的返回值，它存储在地址[ebp+0x8]。在这里，我们必须再次提醒自己一个令人困惑的事情，即栈空间是在内存中向下增长的，因此从逻辑上讲，向上增长的栈更加合理，ebp+0x8 比我们栈基数低 8 个字节，因此我们实际上正式进入了调用我们以获取参数值的函数栈帧中。当然这是我们期望的，因为调用者将该值放在其栈的顶部，然后我们将栈放在其栈的顶部以建立我们新栈的栈帧。

知道任何高级语言编译器在汇编中连接其他代码时所使用的调用约定非常有用。例如，C 的默认调用约定是将参数以相反的顺序推入栈，因为第一个参数位于栈的顶部。混淆参数的顺序肯定会导致程序执行不正确，并可能崩溃。

5.1.4 指针，地址和数据

当我们使用高级语言工作时，我们很容易发现自己忘记了以下事实：变量只是对分配的内存地址的引用，在该地址中已保留了足够的空间来容纳其特定的数据类型。这是因为，在大多数情况下，当我们处理变量时，我们实际上只对它们所保存的值感兴趣。而不是对它们在内存中的位置感兴趣，考虑一下 C 代码：

```
int a = 3;
int b = 4;
int total = a + b;
```

现在，我们对计算机如何执行这些简单的 C 指令有了更多的了解，我们可以做出一个充分了解的假设，即指令 int a=3；将涉及两个主要步骤。首先，至少 4 个字节（32 位）将保留在栈上保留该值，然后，将 3 存储在保留地址中。第二行也是如此，在 int total=a+b 行；将为变量 total 保留更多空间，并在其中存储由标签 a 和 b 指向的地址内容的增加。

现在，假设我们想将值存储在特定的内存位置中；例如，就像我们在汇编中所做的那样，当 BIOS 不再可用时，将字符直接写入地址 0x8000 的视频内存中。当我们希望存储的任何值都必须位于编译器已确定的地址中时，我们将如何在 C 中执行此操作？确实，某些搞级语言根本不允许我们以这种方式访问内存，这从根本上打破了该语言的抽象，幸运的是，C 允许我们使用指针变量，这些变量是用于存储地址（而不是值）的数据类型，并且可以取消引用以将数据指向或指向它们的任何地方。

现在，从技术上讲，所有指针变量都是相同的数据类型（例如32 位内存地址），但是通常我们计划从指针指向的地址读写特定的数据类型，所以我们告诉编译器，比方说这是一个指向 char 的指针，那是一个指向 int 的指针。这真的很方便，这样我们就不必总是告诉编译器它应该从某个指针中的地址读写多少字节，定义和使用指针的语法如下。

```
// Here, the star following the type means that this is not a variable to hold
// a char (i.e. a single byte) but a pointer to the ADDRESS of a char,
// which, being an address, will actually require the allocation of at least
// 32 bits.
char* video_address = 0xb8000;

// If we'd like to store a character at the address pointed to, we make the
// assignment with a star-prefixed pointer variable.  This is known as
// dereferencing a pointer, because we are not changing the address held by
// the pointer variable but the contents of that address.
*video_address = 'X';

// Just to emphasise the purpose of the star, an ommision of it, such as:
video_address = 'X';
// would erroneously store the ASCII code of 'X' in the pointer variable,
// such that it may later be interpretted as an address.
```

在 C 代码中，我们经常看到用于字符串的 char *变量。让我们考虑下这是为什么。如果我们要存储单个 int 或 char，那么我们知道它们都是固定大小的数据类型（即我们知道它们将使用多少个字节），但是字符串是数据类型 char 的数组（通常是 char），可以是任何长度。因此，由于单个数据类型不能容纳整个字符串，因此只能使用其中的一个元素，因此我们可以使用指向 char 的指针，并将其设置为字符串的第一个字符内存地址。这实际上使我们在汇编例程中所做的工作。例如 print_string，在该例程中，我们在代码内的某个位置分配了一个字符串（例如“hello,World”），然后为了打印特定的字符串，我们通过 bx 寄存器传递了第一个字符的地址。

让我们看一个设置字符串变量时编译器执行的示例，在如下图中，我们定义了一个简单的函数，除了将字符串分配给变量外，该函数不执行其他任何操作。

```
void my_function() {
  char* my_string = "Hello";
}
```

和以前一样，我们可以反汇编得到如下图所示内容。

```
00000000  55                push ebp
00000001  89E5              mov ebp,esp
00000003  83EC10            sub esp,byte +0x10
00000006  C745FA48656C6C    mov dword [ebp-0x6],0x6c6c6548
0000000D  66C745FE6F00      mov word [ebp-0x2],0x6f
00000013  C9                leave
00000014  C3                ret
```

首先，为了了解我们的方位，我们需要寻找 ret 指令，该指令标志着函数的结束。我们看到函数的前两个指令通常设置了栈帧。下一条指令如前所述，sub esp，byte+0x10，在栈上分配 16 个字节来存储我们的局部变量。现在，下一条指令 mov dword[ebp-0x4],0xf 应该具有一种熟悉的形式，因为它在我们的变量中存储了一个值。但是为什么它存储数字 0xf——我们没有告诉它这样做，是吗？存储此可疑值后，我们看到函数 politley 将栈还原为调用者栈帧（离开），然后返回（退出）。但是请注意，函数结束后还有 5 条指令！您认为 10 进制指令在做什么？也许它将 eax 的值减少 1，但是为什么呢？那么其余的说明什么呢。

在这种情况下，我们需要进行完整性检查，并记住：反汇编程序无法区分代码和数据；该代码中的某处必须使我们定义的字符串的

数据。现在，我们知道我们的功能由代码的前半部分组成，因为这些指令对我我们很有意义，并且它们以 ret 结尾。如果现在我们假设代码的其余部分实际上我们的数据，那么存储在我们变量中的可疑值 0xf 就很有意义。因为它是从代码开始到数据开始的偏移量：指针变量被设置为数据地址。为了保证我们的直觉，如果我们在 ASCII 表中查询字符串“Hello”的字符值，我们会发现它们为 0x48、0x65、0x6c 和 0x6f。现在，这变得很清楚了，因为如果我们查看反汇编程序输出的中间列，就会发现这些事那些看起来似乎没有意义的奇怪指令的机器代码字节；我们还看到最后一个字节 0x0，C 会自动将其添加到字符串的末尾，因此，就像在我们的汇编例程 print_string 中一样，在处理过程中，我们可以轻松确定何时到达字符串的末尾。

5.2 执行内核代码

理论讲得够多了，让我们引导并执行用 C 编写的最简单的内核。这一步将使用我们迄今所学的所有知识，并将为更快地开发我们的操作系统功能铺平道路。

涉及的步骤如下：

- 编写和编译内核
- 编写并汇编引导扇区代码

- 创建一个内核映像，它不仅包含我们的引导扇区，还包括我们编译的内核代码。
- 将我们的内核代码加载到内存中。
- 切换到 32 位保护模式
- 开始执行我们的内核代码

5.2.1 编写内核

这不会花很长时间，因为目前我们的内核的主要功能是让我们知道它已成功加载和执行。我们稍后可以详细介绍内核，因此起初保持简单很重要。将如下代码保存到名为 Kernel.c 的文件中。

```
void main() {
  // Create a pointer to a char, and point it to the first text cell of
  // video memory (i.e. the top-left of the screen)
  char* video_memory = (char*) 0xb8000;
  // At the address pointed to by video_memory, store the character 'X'
  // (i.e. display 'X' in the top-left of the screen).
  *video_memory = 'X';
}
```

将其编译为原始二进制文件，如下所示：

```
$gcc -ffreestanding -c kernel.c -o kernel.o
$ld -o kernel.bin -Ttext 0x1000 kernel.o --oformat
binary
```

请注意，现在，我们告诉链接器，一旦我们将代码加载到内存中，它将知道从该源代码偏移本地地址引用，就像我们在引导扇区中使用[org 0x7c00]，因为这是 BIOS 加载然后开始执行它的地方。

5.2.2 创建一个引导扇区来引导内核

我们现在要编写一个引导扇区，该引导扇区必须从磁盘引导（即加载并开始执行）我们的内核。由于内核被编译为 32 位指令，因此在执行内核代码之前，我们将不得不切换到 32 位保护模式。我们知道 BIOS 只会在引导时加载引导区（即磁盘的前 512 个字节），而不会加载内核，但是在前面节中，我们了解了如何使用 BIOS 磁盘例程进行引导。扇区从磁盘加载其他扇区，我们隐约意识到，进入保护模式后，缺少 BIOS 将使我们很难使用磁盘：我们将不得不自己编写软盘或硬盘驱动程序。

为了简化从哪个磁盘以及从哪个扇区加载内核代码的问题，可以将操作系统的引导扇区和内核移植到内核映像中，然后将其写入引导磁盘的初始扇区，从而引导扇区代码始终位于内核映像的开头。编译了本节中描述的引导扇区后，就可以使用以下文件链接命令创建内核映像：

```
cat bootsect.bin kernel.bin > os-image
```

在 Bochs 中运行这个命令之前，确保 Bochs 配置文件将引导磁盘设置为内核映像，如下所示

```
floppya: 1_44=os-image, status=inserted
boot: a
```

您可能会想知道的一个问题是，为什么我们从引导磁盘加载多达 15 段（即 512*15 字节），而我们的内核映像比这个小的多，实际上它的大小小于一个扇区，所以加载一个扇区就可以完成任务了。原因

很简单，从磁盘读取这些额外的扇区并没有害，即使它们还没有用数据初始化，但当我们在这个阶段尝试检测到我们在稍后添加到内核代码时没有读取足够的扇区，从而增加内核代码的内存占用大小时，这可能会造成伤害，计算机会在没有警告的情况下挂起，也许在一个程序被分割到一个未加载的扇区边界——一个丑陋的 bug。

如果在屏幕的左上角显示了“X”，那就恭喜你了，虽然对于普通计算机用户来说，它看起来毫无意义，但这意味着我们从开始时就迈出了一大步：我们现在已经引导进入了更高层次的语言，并且您可以开始减少对汇编代码的担心，而更多地关注我们自己如何开发操作系统，并更多地了解 C，这是学习 C 的最好方法：将 C 视作为一种高级语言，而不是从更高抽象的角度（例如 Java 或脚本语言 Python,PHP）开始看待。

5.2.3 找到进入内核的方法

从一个非常简单的内核开始绝对是一个好主意，但是这样做却忽略了一个潜在的问题：当启动内核时，我们不计后果地跳转到内核代码的第一条指令，并由此开始执行；但是我们在前面章节中看到了 C 编译器如何决定将代码和数据放置在输出文件中的任何位置。由于我们的简单内核只有一个函数，并且基于我们先前对编译器如何生成机器代码的观察，我们可以假设第一条机器代码指令是内核入

口函数 main 的第一条指令，但是假设我们的内核代码看起来像如下图这样：

```
void some_function() {
}

void main() {
  char* video_memory = 0xb8000;
  *video_memory = 'X';
  // Call some function
  some_function();
}
```

现在，编译器可能会在 some_function 的指令之前执行预期入口函数 main 的指令，并且由于我们的引导程序代码将从第一条指令开始盲目执行，因此它将命中 some_function 的第一条 ret 指令，并在没有输入 main 的情况下返回到引导扇区代码。问题是，在正确的位置进入我们的内核代码太依赖于内核元代码中的元素（例如函数）的顺序以及编译器和链接器的异想天开，因此我们需要使其更加健壮。

许多操作系统用来正确进入内核的一个技巧是编写一个非常简单的汇编例程，它总是附加在内核机器代码的开头，其唯一的目的是调用内核入口函数，使用汇编的原因是因为我们知道它在机器代码中是如何被翻译的，因此我们可以确保第一条指令最终将导致到达内核的哨兵功能。

这是链接器如何工作的一个很好的例子，因为我们尚未真正利用此重要工具。链接器将目标文件作为输入，然后将它们连接在一起，但是将所有标签解析为正确的地址。例如，如果一个目标文件中有一段代码调用了另一个目标文件中定义的函数 some_function,那么将目标文件中的代码物理链接到一个文件中之后，标签

code:some_function 将解析为特定的例程在组合代码中结束位置的偏移量。

如下显示了一个用于输入内核的简单组装例程：

```
; Ensures that we jump straight into the kernel ’s entry function.
[bits 32] ; We’re in protected mode by now , so use 32-bit
instructions.
[extern main] ; Declate that we will be referencing the
              ; external symbol ’main ’,
              ; so the linker can substitute the final address
call main ; invoke main() in our C kernel
jmp $ ; Hang forever when we return from the kernel
```

从 call main 行可以看到，代码只是调用一个名为 main 的函数，但是 main 不作为标签存在与此代码中，因为它预期存在与其他目标文件中，这样它将在链接时解析为正确的地址；这种期望由文件顶部指令[extern main]表示，如果没有找到这样的标签，链接器将失败。

以前，我们将程序集编译为原始二进制格式，因为我们希望将其作为启动扇区代码在 CPU 上执行，但是由于这段代码无法独立执行，并且无法解析该标签，因此我们必须按如下方式对其进行编译：目标文件，因此保留了有关必须解析的标签信息。

```
$nasm kernel_entry.asm -f elf -o kernel_entry.o
```

选项-elf 指示汇编器输出特定格式的目标文件，该格式为可执行可链接格式（ELF），这是 out C 编译器输出的默认格式。

现在，只需简单地将 kernel.o 文件与其自身链接起来的 kernel.bin,我们可以将其与 kernel_entry.o 链接，如下所示：

```
$ld -o kernel.bin -Ttext 0x1000 kernel_entry.o kernel.o
--oformat binary
```

链接器尊重我们在命令行上提供给它的文件顺序，因此上一个命令将确保我们的 kernel_entry.o 将位于 kernel.o 中的代码之前。

和以前一样，我们可以使用一下命令重建内核映像文件。

```
cat bootsect.bin kernel.bin > os-image
```

现在我们可以在在 Bochs 中对此进行测试，但是可以更加放心地将引导快找到进入内核正确入口点的方式。

5.3 使用 Make 自动化构建

到现在为止，您应该受够了每次更改一段代码时都必须重新键入很多命令，以获取有关更正或尝试过的新想法的反馈，同样幸运的是其他程序员也受够了并已经开发了许多工具软件来使软件的构建过程自动化。在这里，我们将考虑使用 make，他是许多构建工具的前身，并且用于在其他操作系统和应用程序中构建 Linux 和 Minix。make 的基本原理是，我们将配置文件（通常称为 Makefile）中指定如何将一个文件转换为另一个文件，这样一个文件的生成可以描述为依赖于一个或多个其他文件的存在。例如我们可以在 Makefile 中编写一下规则，它将明确告诉 make 如何将 C 文件编译为目标文件。

```
kernel.o : kernel.c
   gcc -ffreestanding -c kernel.c -o kernel.o
```

这样做的好处是，在于 Makefile 相同的目录中，我们现在可以键入：

```
$make kernel.o
```

仅当 kernel.o 不存在或文件修改时间比 kernel.o 更早时，它才会重新编译 C 源文件。但只有当我们添加了一些相互依赖的规则时，我们才能看到 make 如何真正帮助我们节省时间和不必要的命令执行。

```
# Build the kernel binary
kernel.bin: kernel_entry.o kernel.o
  ld -o kernel.bin -Ttext 0x1000 kernel_entry.o kernel.o --oformat binary

# Build the kernel object file
kernel.o : kernel.c
  gcc -ffreestanding -c kernel.c -o kernel.o

# Build the kernel entry object file.
kernel_entry.o : kernel_entry.asm
  nasm kernel_entry.asm -f elf -o kernel_entry.o
```

如果我们使用如上图中的 Makefile 运行 make kernel.bin，make 将知道，在可以运行命令生成 kernel.bin 之前，它必须从其源文件构建其两个依赖项 kernel.o 和 kernel_entry.o。kernel.c 和 kernel_entry.asm 在其运行的命令的以下输出中：

```
nasm kernel_entry.asm -f elf -o kernel_entry.o
gcc -ffreestanding -c kernel.c -o kernel.o
ld -o kernel.bin -Ttext 0x1000 kernel_entry.o kernel.o --oformat
binary
```

然后，如果再次运行 make,我们将看到 make 报告构建目标 kernel.bin 是最新的。但是，如果我们修改例如 kernel.c 将其保存，然后运行 make kernel.bin，我们将看到只有必要的命令由 make 运行，如下所示：

```
gcc -ffreestanding -c kernel.c -o kernel.o
ld -o kernel.bin -Ttext 0x1000 kernel_entry.o kernel.o --oformat
binary
```

为了减少 Makefile 中的重复，从而提高 Makefile 的可维护性，我们可以使用特殊的 Makefile 变量$<、$@和$^，如下所示

```
# $^ is substituted with all of the target's dependancy files
kernel.bin: kernel_entry.o kernel.o
  ld -o kernel.bin -Ttext 0x1000 $^ --oformat binary

# $< is the first dependancy and $@ is the target file
kernel.o : kernel.c
  gcc -ffreestanding -c $< -o $@

# Same as the above rule.
kernel_entry.o : kernel_entry.asm
  nasm $< -f elf -o $@
```

不指定实际目标通常很有用，因为它们不会生成文件，如下显示我们运行 make clean 时，所有生成的文件都会从目录中删除，只留下源文件。

```
clean:
  rm *.bin *.o
```

如果您只想将源文件分发给朋友，将目录置于版本控制下，或者想要测试对 Makefile 的修改是否可以正确地从头开始构建所有目标，则以这种方式清理目录非常有用。

如果 make 在没有目标的情况下运行，则主文件中的第一个目标将被视为默认目标，因此您经常会看到一个虚拟目标，如下所示，位于 Makefile 顶部：

```
# Default make target.
all: kernel.bin

# $^ is substituted with all of the target's dependancy files
kernel.bin: kernel_entry.o kernel.o
  ld -o kernel.bin -Ttext 0x1000 $^ --oformat binary

# $< is the first dependancy and $@ is the target file
kernel.o : kernel.c
  gcc -ffreestanding -c $< -o $@

# Same as the above rule.
kernel_entry.o : kernel_entry.asm
  nasm $< -f elf -o $@
```

请注意，通过 kernel.bin 作为对所有目标的依赖关系，我们确保 kernel.bin 及其所有依赖关系都是为默认目标构建的。

现在，我们可以将用于构建内核和可加载内核映像的所有命令放入一个有用的 Makefile 中，这将使我们只需要键入 make run 即可再 Bochs 中测试对代码的更改或更正。

```
all: os-image

# Run bochs to simulate booting of our code.
run: all
  bochs

# This is the actual disk image that the computer loads,
# which is the combination of our compiled bootsector and kernel
os-image: boot_sect.bin kernel.bin
  cat $^ > os-image
```

```
# This builds the binary of our kernel from two object files:
#  - the kernel_entry, which jumps to main() in our kernel
#  - the compiled C kernel
kernel.bin: kernel_entry.o kernel.o
  ld -o kernel.bin -Ttext 0x1000 $^ --oformat binary

# Build our kernel object file.
kernel.o : kernel.c
  gcc -ffreestanding -c $< -o $@

# Build our kernel entry object file.
kernel_entry.o : kernel_entry.asm
  nasm $< -f elf -o $@

# Assemble the boot sector to raw machine code
#   The -I options tells nasm where to find our useful assembly
#   routines that we include in boot_sect.asm
boot_sect.bin : boot_sect.asm
  nasm $< -f bin -I '../../16bit/' -o $@


# Clear away all generated files.
clean:
  rm -fr *.bin *.dis *.o os-image *.map

# Disassemble our kernel - might be useful for debugging.
kernel.dis : kernel.bin
  ndisasm -b 32 $< > $@
```

5.3.1 组织操作系统的代码库

我们现在得到了一个非常简单的 C 内核，它在屏幕的一角打印出一个“X”。内核被编译成 32 位指令，并被 CPU 成功地执行，这一事实本身就意味着我们已经走了很远的路；但现在是时候为今后的工作做好准备了。我们需要为我们的代码建立一个合适的结构，并附带一个 Makefile，该文件允许我们轻松地想操作系统添加具有新功能的的新源文件，并使用 Bochs 等仿真器以增量方式检查这些添加。

与 Linux 和 Minix 等内核类似，我们可以将代码库组织到一下文件夹中：

- boot：任何与引导和引导扇区相关的内容都可以放在此目录下，比如 boot_sect.asm，以及我们的引导扇区汇编程序（例如 print_string.asm，switch_to_pm.asm 等等）。
- kernel：内核的主文件 kernel.c 和其他与内核先关的。与设备驱动无关的代码将放这里。
- drivers：任何特定于硬件的驱动程序代码都会放在这里。

现在，在我们的 Makefile 中，我们不必指定要构建的每个文件（例如 kernel/kernel.o、drivers/screen.o、drivers/keyboard.o 等），我们可以使用如下特殊通配符声明：

```
# Automatically expand to a list of existing files that
# match the patterns
C_SOURCES = $(wildcard kernel/*.c drivers/*.c)
```

然后，我们可以使用另一个 make 声明将源文件名转换为对象文件名，如下所示：

```
# Create a list of object files to build, simple by replacing
# the '.c' extension of filenames in C_SOURCES with '.o'
OBJ = ${C_SOURCES:.c=.o}
```

现在我们可以将内核对象文件链接在一起，以构建内核二进制文件，如下所示：

```
# Link kernel object files into one binary, making sure the
# entry code is right at the start of the binary.
kernel.bin: kernel/kernel_entry.o ${OBJ}
        ld -o $@ -Ttext 0x1000 $^ --oformat binary
```

Make 的一个特性将与我们动态包含对象文件密切相关，它告诉 make 如何根据文件名的简单模式从另一个文件类型构建一个文件类型，如下所示：

```
# Generic rule for building 'somefile.o' from 'somefile.c'
%.o : %.c
        gcc -ffreestanding -c $< -o $@
```

替代方法将是很多重复，如下所示：

```
kernel/kernel.o : kernel/kernel.c
        gcc -ffreestanding -c $< -o $@

drivers/screen.o : drivers/screen.c
        gcc -ffreestanding -c $< -o $@

drivers/keyboard.o : drivers/keyboard.c
        gcc -ffreestanding -c $< -o $@

...
```

很好，现在我们已经充分了解了 make 了，我们可以继续开发我们的内核，而不必一遍又一遍地重新输入很多命令来检查某些东西是否正常工作。如下所示了一个完整的 Makefile，它将适合我们的内核进行处理。

```
# Automatically generate lists of sources using wildcards.
C_SOURCES = $(wildcard kernel/*.c drivers/*.c)
HEADERS = $(wildcard kernel/*.h drivers/*.h)

# TODO: Make sources dep on all header files.

# Convert the *.c filenames to *.o to give a list of object files to build
OBJ = ${C_SOURCES:.c=.o}

# Defaul build target
all: os-image
```

```
# Run bochs to simulate booting of our code.
run: all
  bochs

# This is the actual disk image that the computer loads
# which is the combination of our compiled bootsector and kernel
os-image: boot/boot_sect.bin kernel.bin
  cat $^ > os-image


# This builds the binary of our kernel from two object files:
#  - the kernel_entry, which jumps to main() in our kernel
#  - the compiled C kernel
kernel.bin: kernel/kernel_entry.o ${OBJ}
  ld -o $@ -Ttext 0x1000 $^ --oformat binary

# Generic rule for compiling C code to an object file
# For simplicity, we C files depend on all header files.
%.o : %.c ${HEADERS}
  gcc -ffreestanding -c $< -o $@

# Assemble the kernel_entry.
%.o : %.asm
  nasm $< -f elf -o $@

%.bin : %.asm
  nasm $< -f bin -I '../../16bit/' -o $@

clean:
  rm -fr *.bin *.dis *.o os-image
  rm -fr kernel/*.o boot/*.bin drivers/*.o
```

5.4 C 入门

C 语言有一些怪异之处，可能会使新的程序员感到不安。

5.4.1 预处理和指令

在 C 文件被编译成对象文件之前，预处理器会扫描它以查找预处理程序的指令和变量，然后通常用代码（例如宏和常量值）来替换

它们，或者干脆不进行任何替换。预处理器不是编译 C 代码所必需的，而是提供一些方便，是代码更易于管理。

```
#define PI 3.141592
...
float radius = 3.0;
float circumference = 2 * radius * PI;
...
```

预处理器将输出一下代码，准备进行编译：

```
...
float radius = 3.0;
float circumference = 2 * radius * 3.141592;
...
```

预处理器对于输出条件代码也很有用，但不是在运行时做出决定的（例如 if 语句），而不是在编译时。例如，考虑一下预处理程序指令来包含或排除调试代码：

```
...
#ifdef DEBUG
print("Some debug message\n");
#endif
...
```

现在，如果定义了预处理器变量 DEBUG，那么这样的调试代码将被包括在内；否则，就不包含了。编译 C 文件时，可以在命令行上定义变量，如下所示：

```
$gcc -DDEBUG -c some_file.c -o some_file.o
```

此类命令行变量声明通常用于应用程序（尤其是操作系统）的编译时配置，其中可能包括或排除整个代码段，也许是为了减少小型嵌入式设备上内核的内存占用。

5.4.2 函数声明和头文件

当编译器对可能在编译的文件中定义或可能未定义的函数调用进行封装时，如果尚未遇到函数的返回类型和参数的描述，则编译器可能做出错误的假设并生成错误的机器代码指令，回顾前面章节，编译器必须为传递给函数的参数准备栈，但是如果栈不是函数期望的栈，则栈可能会损坏。因此，重要的是，至少要在使用函数之前给出函数接口的声明（如果不是整个函数定义）。此声明称为函数的原型。

```
int add(int a, int b) {
  return a + b;
}

void main() {
  // This is okay, because our compiler has seen the full
  // definition of add.
  int result = add(5, 3);

  // This is not okay, since compiler does not know the return
  // type or anything about the arguments.
  result = divide(34.3, 12.76);

  // This is not okay, because our compiler knows nothing about
```

```
  // this function's interface.
  int output = external_function(5, "Hello", 4.5);
}

float divide(float a, float b) {
  return a / b;
}
```

这可以固定如下：

```
// These function prototypes inform the compiler about
// the function interfaces.
float divide(float a, float b);  // <-- note the semi-colon
int external_function(int a, char* message, float b);

int add(int a, int b) {
  return a + b;
}

void main() {
  // This is okay, because our compiler has seen the full
  // definition of add.
  int result = add(5, 3);

  // This is okay now: compiler knows the interface.
  result = divide(34.3, 12.76);

  // This is okay now: compiler knows the interface.
  int output = external_function(5, "Hello", 4.5);
}

float divide(float a, float b) {
  return a / b;
}
```

现在，由于将从编译到其他目标文件中的代码中调用某些函数，因此它们还需要声明这些函数的相同原型。这将导致重复的原型声明，难以维护。因此许多 C 程序使用#include 预处理程序指令在编译之前插入包含所需原型的通用代码。这种通用代码称为头文件，我们可以将其视为已编译目标文件的接口，其用法如下：

6、开发基本设备驱动程序和文件系统

6.1 硬件输入/输出

通过写入屏幕，我们实际上已经遇到了一种更好的的硬件 I/O 方式，称为内存映射 I/O，直接写入主存中某个地址范围的数据被写入设备内存缓存区，但现在是时候了解 CPU 和硬件之间的交互作用。

让我们以现在流行的 TFT 显示器为例。屏幕的表面被分成一个由背光单元组成的矩阵。通过在极化薄膜之间夹一层液晶，通过施加电场可以改变通过每个单元的光的量，因为液晶的性质，当受到电场时，它们的取向可以以一致的方式改变，当晶体的方向改变时，它们会改变光波的振动方向，这样一些光就会被屏幕表面的偏振薄膜挡住。对于彩色显示，每个单元格进一步分为三个区域，这些区域覆盖有红色，蓝色和绿色的过滤器。

因此，硬件的工作就是确保适当的单元或子单元的颜色区域受到适当的电场的作用，从而在屏幕上重建所需的图像。硬件的这一面最好留给专业的电子工程师，但在设备或主板上会有一个控制器芯片，理想的情况下具有定义明确的功能（如芯片数据表中所述），CPU 可以与之交互以控制硬件。实际上，出于向后兼容的原因，TFT 显示器通常模仿老式的 CRT 显示器，因此可以由主板的标准 VGA 控制器驱动，该控制器产生一个复杂的模拟信号，引导电子束在涂有

荧光粉的屏幕上扫描，而且由于没有真正的 CRT 光束来引导，TFT 显示器巧妙地将此信号解释为数字图像。

在内部，控制器芯片通常具有几个可由 CPU 读写的寄存器，这些寄存器的状态告诉控制器该做什么（例如，将哪个引脚设置为高电平或低电平来驱动硬件，或者执行什么内部功能）。例如，从 Intel 广泛使用的 82077AA 单片机软盘控制器的数据表中，我们看到有一个引脚（引脚 57，标记为 ME0）驱动第一个软盘设备的电机（因为一个控制器可以驱动多个这样的设备）：当引脚打开时，电机旋转，当关闭时，电机不旋转。这个特定引脚的状态直接连接到控制器内部寄存器的一个特定位上，即数字输出寄存器（DOR）。然后，可以通过设置一个值来设置寄存器的状态，在芯片的数据引脚上设置适当的位（在本例中是位 4），并使用芯片的寄存器选择引脚 A0-A2，通过内部地址 0x2 选择 DOR 寄存器。

6.1.1 I/O 总线

尽管从历史上讲，CPU 会直接与设备控制器通信，但随着 CPU 速度的不断提高，这将需要 CPU 人为地降低到与最慢设备相同的速度，因此 CPU 直接向高速顶级总线的控制器芯片发出 I/O 指令更为实际。然后，总线控制器负责以兼容速率将指令中断到特定设备的控制器。然后为了避免顶级总线不得不为较慢的设备减速，另一种总

线技术的控制器可以作为一个设备添加，这样我们就可以到达现代计算机中的总线层次结构。

6.1.2 I/O 编程

所以问题是，我们如何以编程方式读写设备控制器的寄存器（即告诉我们的设备该做什么）？在 Intel 体系结构系统中，设备控制器的寄存器被映射到一个 I/O 地址空间中，该空间与主内存地址空间分离，然后使用 in 和 out 指令的变量来读取和写入映射到特定控制器寄存器的 I/O 地址。例如，前面描述的软盘控制器通常将其 DOR 寄存器映射到 I/O 地址 0x3F2，因此我们可以使用以下指令启动第一个驱动器的电机。

```
mov dx, 0x3f2     ; Must use DX to store port address
in al, dx         ; Read contents of port (i.e. DOR) to AL
or al, 00001000b ; Switch on the motor bit
out dx, al        ; Update DOR of the device.
```

在较旧的系统中，如工业标准体系机构（ISA）总线，端口地址将静态分配给设备，但使用现代即插即用总线（如外围组件互联 PCI），BIOS 可以在引导操作系统之前将 I/O 地址动态分配给大多数设备。这种动态分配要求设备通过总线来传输配置信息，以描述硬件，例如：需要为寄存器保留多少 I/O 端口；需要多少内存映射空间；以及硬件类型的唯一 ID，以便操作系统以后可以找到适当的驱动程序。

端口 I/O 的问题在于我们无法使用 C 语言表达这些低级指令，因此我们必须了解一些有关内联汇编的知识，大多数编译器都允许您将汇编代码片段插入函数主体中，用 gcc 实现如下：

```
unsigned char port_byte_in(unsigned short port) {
  // A handy C wrapper function that reads a byte from the specified port
  //  "=a" (result) means: put AL register in variable RESULT when finishe
  //  "d" (port) means: load EDX with port
  unsigned char result;
  __asm__("in %%dx, %%al" : "=a" (result) : "d" (port));
  return result;
}
```

注意，in %%dx,%%al 中的实际汇编指令在我们看来有点奇怪，因为 gcc 采用了不同的汇编语法（称为 GAS），其中目标操作数与我们更熟悉的 nasm 语法相反，另外%用于表示寄存器，这需要一个难看的%%，因为%是 C 编译器的转义字符，所以%%的意思：是转义转义字符，这样它就出现在字符串中。

由于这些底层端口 I/O 函数将被内核中的大多数硬件驱动程序使用，所以让我们将她们收集到 kernel/low_level.c 文件中，我可以将其定义为如下图：

```
unsigned char port_byte_in(unsigned short port) {
  // A handy C wrapper function that reads a byte from the specified port
  //  "=a" (result) means: put AL register in variable RESULT when finished
  //  "d" (port) means: load EDX with port
  unsigned char result;
  __asm__("in %%dx, %%al" : "=a" (result) : "d" (port));
  return result;
}

void port_byte_out(unsigned short port, unsigned char data) {
  // "a" (data) means: load EAX with data
  // "d" (port) means: load EDX with port
  __asm__("out %%al, %%dx" : :"a" (data), "d" (port));
}

unsigned short port_word_in(unsigned short port) {
  unsigned short result;
  __asm__("in %%dx, %%ax" : "=a" (result) : "d" (port));
  return result;
}

void port_word_out(unsigned short port, unsigned short data) {
  __asm__("out %%ax, %%dx" : :"a" (data), "d" (port));
}
```

6.1.3 直接内存访问

由于端口 I/O 涉及到读取或写入单个字节或字，在磁盘设备和内存之间传输大量数据可能会占用大量 CPU 时间。这个问题使得 CPU 有必要将这个繁琐的任务传递给其他人，即直接内存访问（DMA）控制器。

DMA 的一个很好的类比是建筑师希望将墙从一个地方移到另一个地方。建筑师很清楚要做什么，但除了移动每一块转之外，还有其他重要的事情要考虑，因此指示建筑工人一块一块地移动砖块，并在墙完工或存在妨碍完工的错误时提醒（即打断）他。

6.2 屏幕驱动

到目前为止，我们的内核能够在屏幕的一角打印“X”，尽管足以让我们知道我们的内核已经成功加载并执行，但并不能告诉我们计算机上正在发生的一切。

我们知道，可以通过在地址 0xb8000 的显示缓存区中的某个位置写入字符来在屏幕上显示字符，但我们不想在整个内核中做这种低级操作。如果我们能够创建一个屏幕抽象，运行我们写 print(“hello”)，或者 clean_screen()那会更好；如果它能在我们打印到最后一行显示线时它可以滚动，那将锦上添花。这种抽象不仅使在内核的其他代码中显示信息更加容易，而且还使我们可以在以轻松地将一个显示驱动程序替换为另一个显示驱动程序，也许如果某台计算机不支持我们目前假设的彩色 VGA 文本。

6.2.1 了解显示设备

与我们不久将要介绍的其他硬件相比，显示设备相当简单，因为作为内存映射设备，我们可以在不了解控制消息和硬件 I/O 的情况下正常工作，屏幕上需要 I/O 控制（即通过 I/O 端口）来操作的一个有用的设备是光标，它闪烁以标记下一个字符将被写入屏幕上的位置。这对用户很有用，因为它可以让用户注意到输入文本的提示，

但我们也将它作用内部标记，无论光标是否可见，这样程序员就不必总是指定屏幕上显示字符串的位置的坐标。例如：如果我们写print(“hello”)，每个字符都将被写入。

6.2.2 基本的屏幕驱动器实现

虽然我们可以在 kernel.c 中编写所有这些代码，其中包含内核的入口函数 main()，但最好将这些特定于功能的代码组织到它自己的文件中，该文件可以编译并链接到我们的内核代码中，但其效果与将它们全部放入一个文件中的效果相同。让我们在 drivers 文件夹中创建一个新的驱动程序实现文件 screen.c 和驱动程序接口文件 screen.h。由于我们在 Makefile 中使用了包含通配符的文件，screen.c（以及该文件夹中的任何其他 c 文件）将被自动编译并链接到我们的内核。

首先，让我们在 screen.h 中定义一下常量，使代码更易于阅读。

```
#define VIDEO_ADDRESS 0xb8000
#define MAX_ROWS 25
#define MAX_COLS 80
// Attribute byte for our default colour scheme.
#define WHITE_ON_BLACK 0x0f

// Screen device I/O ports
#define REG_SCREEN_CTRL 0x3D4
#define REG_SCREEN_DATA 0x3D5
```

然后，让我们考虑如何编写一个函数 print_char(...)，它在屏幕的特定列和行中显示单个字符。我们将在内部（即私下）在我们的驱动程序中使用这个函数，这样我们的驱动程序的公共接口函数（即

我们希望外部代码使用的函数）将在此基础上构建，我们现在知道 video 内存只是一个特定范围的内存地址范围，其中每个字符单元由两个字节表示，第一个字节是字符 ASCII 码，第二个字节是属性字节，它允许我们设置字符单元的配色方案。如下显示了如何通过使用函数来定义这些操作：get_cursor()，set_cursor()，get_screen_offset()，和 handle_scrolling()。

```
/* Print a char on the screen at col, row, or at cursor position */
void print_char(char character, int col, int row, char attribute_byte) {
  /* Create a byte (char) pointer to the start of video memory */
  unsigned char *vidmem = (unsigned char *) VIDEO_ADDRESS;

  /* If attribute byte is zero, assume the default style. */
  if (!attribute_byte) {
    attribute_byte = WHITE_ON_BLACK;
  }

  /* Get the video memory offset for the screen location */
  int offset;
  /* If col and row are non-negative, use them for offset. */
  if (col >= 0 && row >= 0) {
    offset = get_screen_offset(col, row);
  /* Otherwise, use the current cursor position. */
  } else {
    offset = get_cursor();
  }

  // If we see a newline character, set offset to the end of
  // current row, so it will be advanced to the first col
  // of the next row.
  if (character == '\n') {
```

```
    int rows = offset / (2*MAX_COLS);
    offset = get_screen_offset(79, rows);
  // Otherwise, write the character and its attribute byte to
  // video memory at our calculated offset.
  } else {
    vidmem[offset] = character;
    vidmem[offset+1] = attribute_byte;
  }

  // Update the offset to the next character cell, which is
  // two bytes ahead of the current cell.
  offset += 2;
  // Make scrolling adjustment, for when we reach the bottom
  // of the screen.
  offset = handle_scrolling(offset);
  // Update the cursor position on the screen device.
  set_cursor(offset);
}
```

首先让我们来解决最简单的功能：get_screen_offset，此函数将行和列坐标映射到从视频内存开始的特定显示字符单元的内存偏移量。映射很简单，但我们必须记住，每个单元包含了两个字节。例如，如果我想在显示器的第 3 行第 4 列设置一个字符，则该字符单元的偏移量为（十进制）488（(3*80*(行宽)+4)*2=488）。因此，我们的 get_screen_offset 函数将类似于下图。

```
  // This is similar to get_cursor, only now we write
  // bytes to those internal device registers.
  port_byte_out(REG_SCREEN_CTRL, 14);
  port_byte_out(REG_SCREEN_DATA, (unsigned char)(offset >> 8));
  port_byte_out(REG_SCREEN_CTRL, 15);
```

现在让我们看下光标控制函数，get_cursor()和 set_cursor()，它们将通过一组 I/O 端口操作显示控制器的寄存器。使用特定的视频设备 I/O 端口读取和写入其内部光标相关寄存器，这些函数的实现将类似于下图：

```
  cursor_offset -= 2*MAX_COLS;

  // Return the updated cursor position.
  return cursor_offset;
}

int get_cursor() {
  // The device uses its control register as an index
  // to select its internal registers, of which we are
  // interested in:
```

```
  //   reg 14: which is the high byte of the cursor's offset
  //   reg 15: which is the low byte of the cursor's offset
  // Once the internal register has been selected, we may read or
  // write a byte on the data register.
  port_byte_out(REG_SCREEN_CTRL, 14);
  int offset = port_byte_in(REG_SCREEN_DATA) << 8;
  port_byte_out(REG_SCREEN_CTRL, 15);
  offset += port_byte_in(REG_SCREEN_DATA);
  // Since the cursor offset reported by the VGA hardware is the
  // number of characters, we multiply by two to convert it to
  // a character cell offset.
  return offset*2;
}

void set_cursor(int offset) {
  offset /= 2;  // Convert from cell offset to char offset.
  // This is similar to get_cursor, only now we write
  // bytes to those internal device registers.
```

所以现在我们有了一个函数，它允许我们在屏幕的特定位置打印子一个字符，这个函数封装了所有凌乱的硬件特定的东西。通常，我们不想将每个字符打印到屏幕上，而是要打印一整串字符，所以让我们创建一个更友好的函数 print_at(…)，它将指针指向字符的第一个字符（即 char*），然后从给定的坐标一个接一个地打印每个后续字符。如果坐标（-1，-1）传递给函数，则它将从当前光标开始位置打印。我们的 print_at(….)函数类似于如下图：

```
void print_at(char* message, int col, int row) {
  // Update the cursor if col and row not negative.
  if (col >= 0 && row >= 0) {
    set_cursor(get_screen_offset(col, row));
  }
  // Loop through each char of the message and print it.
  int i = 0;
  while(message[i] != 0) {
    print_char(message[i++], col, row, WHITE_ON_BLACK);
  }
}
```

纯粹是为了方便起见，为了使我们不必输入 print_at('hello',-1,-1)，我们可以顶一个函数 print()它仅使用一个参数，如下图：

```
void print(char* message) {
  print_at(message, -1, -1);
}
```

另一个有用但不太难的功能是 clear_screen(...)，它允许我们通过在每个位置书写空白字符来整理屏幕，如下图显示：

```
void clear_screen() {
  int row = 0;
  int col = 0;

  /* Loop through video memory and write blank characters. */
  for (row=0; row<MAX_ROWS; row++) {
    for (col=0; col<MAX_COLS; col++) {
      print_char(' ', col, row, WHITE_ON_BLACK);
    }
  }

  // Move the cursor back to the top left.
  set_cursor(get_screen_offset(0, 0));
}
```

6.2.3 滚动屏幕

如果您希望当光标到达屏幕底部时屏幕会自动滚动，那么您的大脑一定已经退回到了更高级别的计算领域，这是可以原谅的，因为屏幕滚动看起来是很自然的事情，我们只是理所当然地认为，但是在此级别工作，我们可以完全控制硬件，因此必须自己实现此功能。

为了使屏幕在到达底部时看起来可以滚动，我们必须将每个字符单元格向上移动一行，然后清除最后一行，以准备写入新行（即本应在屏幕末尾以外写入的行）。这意味着第一行将被第二行覆盖，因

此第一行将永远丢失，这一点我们不关心，因为我们的目标是允许用户在他们的计算机上看到最新的活动日志。

一种实现滚动的好方法是在增加 print_char 中的光标位置后立即调用一个函数，我们将其定义为 handle_scrolling。这样一来，handle_scrolling 就可以确保每当光标的视频内存偏移量增加到屏幕的最后一行之外时，这些行就会滚动并且光标会重新定位在最后一个可见行（即新行）中。

移动一行等同于将其所有字节（即一行中的 80 个字符单元中每个字符两个字节）复制到上一行的地址。这是向操作系统添加通用的 memory_copy 函数的绝好机会。由于我们可能会在操作系统的其他区域使用此函数，因此请将其添加到文件 kernel/util.c 中，我们的 memory_copy 函数将获取源和目标的地址以及要复制的字节数，然后通过循环将复制每个字节。如下图：

```
/* Copy bytes from one place to another. */
void memory_copy(char* source, char* dest, int no_bytes) {
```

```
  int i;
  for (i=0; i<no_bytes; i++) {
    *(dest + i) = *(source + i);
  }
}
```

现在我们可以使用 memory_copy,滚动屏幕

```
/* Advance the text cursor, scrolling the video buffer if necessary. */
int handle_scrolling(int cursor_offset) {

  // If the cursor is within the screen, return it unmodified.
  if (cursor_offset < MAX_ROWS*MAX_COLS*2) {
    return cursor_offset;
  }

  /* Shuffle the rows back one. */
  int i;
  for (i=1; i<MAX_ROWS; i++) {
    memory_copy(get_screen_offset(0,i) + VIDEO_ADDRESS,
                get_screen_offset(0,i-1) + VIDEO_ADDRESS,
                MAX_COLS*2
    );
  }

  /* Blank the last line by setting all bytes to 0 */
  char* last_line = get_screen_offset(0,MAX_ROWS-1) + VIDEO_ADDRESS;
  for (i=0; i < MAX_COLS*2; i++) {
    last_line[i] = 0;
  }

  // Move the offset back one row, such that it is now on the last
  // row, rather than off the edge of the screen.
  cursor_offset -= 2*MAX_COLS;

  // Return the updated cursor position.
  return cursor_offset;
}
```

6.3 处理中断

6.4 键盘驱动

6.5 磁盘驱动

6.6 文件系统

7、进程管理

7.1 单进程

7.2 多进程