TU, DO HOANG

操作系统：从0到1

Contents

Preface

你好！

你可能至少问过自己一次，操作系统是如何从头开始编写的。你可能甚至拥有多年的编程经验，但你对操作系统的理解可能仍然是一系列抽象概念，而不是基于实际实现的。对于那些从未构建过操作系统的人来说，操作系统可能就像魔法一样：一种神秘的东西，可以控制硬件，同时通过他们最喜欢的编程语言的API处理程序员的请求。学习如何构建操作系统似乎令人畏惧且困难；无论你学了多少，你永远不会觉得你了解得足够。你现在可能正在阅读这本书，以更好地理解操作系统，成为一个更好的软件工程师。

如果情况如此，这本书就是为你准备的。通过阅读这本书，你将能够找到那些至关重要的缺失部分，并使你能够从头开始实现自己的操作系统！是的，从头开始，不通过任何现有的操作系统层来证明自己是一名操作系统开发者。你可能想知道：“学习Linux的内部结构不是更实用吗？”。

是...

并且没有。

学习Linux可以帮助你在日常工作中提高工作效率。然而，如果你选择那条路线，你仍然无法达到编写实际操作系统的最终目标。通过编写自己的操作系统，你将获得仅从学习中学不到的知识。

运行 Linux。

以下是自己编写操作系统的部分好处列表：

- 您将学习计算机在硬件层面的工作原理，并将学习编写软件直接管理该硬件。
- 您将学习操作系统的基本原理，这使您能够适应任何操作系统，而不仅仅是Linux
- 要在 Linux 内部结构上进行适当的黑客攻击，你至少需要自己编写一个操作系统。这就像应用程序编程一样：要编写一个大型应用程序，你需要从简单的开始。
- 您将开启通往各种低级编程领域的途径，例如逆向工程、漏洞利用、构建虚拟机、游戏机仿真等。汇编语言将成为您进行低级分析最不可或缺的工具。（但这并不意味着您必须用汇编语言编写您的操作系统！）
- 编写操作系统很有趣!

Why another book on Operating Systems?

有许多由著名教授和专家编写的关于这个主题的书籍和课程。我有什么资格写一本关于如此高级主题的书呢？虽然确实存在许多优质资源，但我发现它们还不够。它们中有任何一本向你展示了如何在不依赖现有操作系统的情况下编译你的C代码和C运行时库吗？大多数关于操作系统设计和实现的书籍只讨论软件方面；操作系统如何与硬件通信被跳过了。重要的硬件细节被跳过了，自学成才的人很难在互联网上找到相关的资源。这本书的目标是填补这一空白：你不仅将学习如何直接编程硬件，还将学习如何阅读硬件供应商的官方文档来编程它。你不再需要寻找资源来帮助你解释硬件手册和文档：你可以自己完成。最后，我以自学者的视角写了这本书。我将这本书编写得尽可能独立，这样你就可以花更多

学习时间更长，猜测或在网上寻找信息的时间更少。

这本书的核心重点之一是指导您阅读供应商的官方文档，以实现您的软件。来自英特尔等硬件供应商的官方文档对于实现操作系统或其他直接控制硬件的软件至关重要。至少，操作系统开发者需要能够理解这些文档，并根据一组硬件要求实现软件。因此，第一章专门讨论相关文档及其重要性。

这本书的另一个显著特点是它以“Hello World”为中心。大多数示例都围绕“Hello World”程序的变体展开，这将使您熟悉核心概念。在尝试编写操作系统之前，必须学习这些概念。任何超出简单“Hello World”示例的内容都会妨碍对概念的教授，从而延长开始编写操作系统所需的时间。

让我们深入探讨。通过这本书，我希望提供足够的基础知识，这将为你打开理解其他资源的门。这本书将对刚刚完成他们第一个C/C++课程的学生非常有帮助。想象一下，向潜在雇主展示你已经构建了一个操作系统是多么酷的事情。

Prerequisites

▷ 基本电路知识

- 电力基本概念：原子、电子、质子、中子、电流流动。– 欧姆定律

如果您对这些概念不熟悉，可以在这里快速学习它们：
`http://www.allaboutcircuits.com/textbook/`，通过阅读第1章和第2章。

▷ C程序设计。特别是：

– 变量和函数声明/定义 – while和for循环 – 指针和函数指针 – C中的基本算法和数据结构

▷ Linux 基础：

- 了解如何使用命令行导航目录- 了解如何使用选项调用命令- 了解如何将输出管道传输到另一个程序

▷ 触摸打字。由于我们将使用Linux，触摸打字很有帮助。我知道打字速度与解决问题的能力无关，但至少你的打字速度应该足够快，以免影响并降低学习体验。

通常，我假设读者具备基本的C语言编程知识，并且可以使用IDE构建和运行程序。

What you will learn in this book

▷ 从硬件数据表中阅读，从头开始编写操作系统的方法。在现实世界中，你无法快速查阅谷歌来获得答案。▷独立编写代码。复制粘贴代码毫无意义。真正的学习发生在你独立解决问题的时候。提供了一些示例以帮助您开始工作，但大多数问题需要你自己去征服。然而，在你努力尝试后，解决方案在网上是可用的。▷从硬件到软件，每一层计算机是如何相互关联的总体图景。▷如何将Linux用作开发环境以及用于低级编程的常用工具。▷程序是如何构建的，以便操作系统可以运行。

▷ 如何使用 gdb 和 QEMU 调试直接在硬件上运行的程序。▷ 在裸金属 x86_64 上链接和加载，使用纯 C。没有标准库。没有运行时开销。

What this book is not about

▷ 电气工程：本书仅讨论了电子和电气工程的一些概念，仅限于软件在裸金属上运行的程度。▷ 如何使用Linux或任何操作系统类型的书籍：尽管Linux被用作开发环境和演示高级操作系统概念的媒介，但这本书并不是以Linux为重点。▷ Linux内核开发：关于这个主题已经有许多高质量的书籍。▷ 专注于算法的操作系统书籍：本书更多地关注实际的硬件平台——Intel x86_64——以及如何编写利用硬件平台支持的操作系统。

书籍的组织

Part 1 提供学习操作系统的基础。

▷ 第一章简要说明了领域文档的重要性。文档对于学习体验至关重要，因此它们应有一章。▷第二章解释了从硬件到软件的抽象层次。目的是提供对代码如何物理运行的洞察。▷第三章提供了计算机的一般架构，然后介绍了一个您将用于编写操作系统的示例计算机模型。

- ▷ 第四章通过使用英特尔手册介绍了x86汇编语言，以及常用指令。本章提供了高级语法如何对应低级汇编的详细示例，使您能够舒适地阅读生成的汇编代码。在调试操作系统时，阅读汇编代码是必要的。

- ▷ 第五章详细剖析了ELF。只有通过理解程序在二进制级别的结构，你才能构建一个在裸机上运行的程序。

- ▷ 第6章介绍了具有丰富示例的 gdb 调试器，用于常用命令。在使读者熟悉 gdb 之后，它提供了关于调试器如何工作的见解。这些知识对于在裸金属上构建可调试程序至关重要。

Part 2 介绍如何编写引导加载程序以引导内核。因此得名 *"Groundwork"*。掌握这部分内容后，读者可以继续下一部分，即编写操作系统的指南。然而，如果读者不喜欢这种展示方式，他或她可以另寻他处，例如 OSDev Wiki：`http://wiki.osdev.org/`。

- ▷ 第7章介绍了引导加载程序是什么，如何在汇编中编写一个，以及如何在QEMU（一个硬件仿真器）上加载它。这个过程涉及到输入重复且长的命令，因此应用GNU Make来提高生产力，通过自动化重复部分并简化与项目的交互。本章还演示了在上下文中使用GNU Make的方法。

- ▷ 第8章通过解释组合目标文件时的重定位过程来介绍链接。除了用C编写的引导加载程序和操作系统之外，这是构建裸机上的可调试程序所需的最后一部分拼图，包括用汇编编写的引导加载程序和用C编写的操作系统。

Part 3 提供有关如何编写操作系统的指导，因为您应该自己实现操作系统并为其自豪。该指南包括从硬件到软件的必要概念的更简单和连贯的解释，以实现

操作系统功能。没有此类指导，您将浪费时间收集散布在各种文档和
互联网上的信息。然后它提供了一个如何将概念映射到代码的计划。

Acknowledgments

谢谢，我亲爱的家人。谢谢，各位贡献者。

Part I

初步

1

Domain documents

1.1 Problem domains

在现实世界中，软件工程不仅关注软件，还关注其试图解决的问题域。

> 一个 *problem domain* 是 *the part of the world*，其中计算机是用于削减效果，以及产生这些效果的可用手段，无论是直接还是间接。 (Kovitz, 1999)

problem domain

A *problem domain* 是软件工程师为了编写能够实现预期效果的正确代码而需要了解的编程之外的一切。“直接”意味着包括软件可以控制以产生预期效果的一切，例如键盘、打印机、显示器、其他软件等。“间接”意味着不属于软件但与问题领域相关的一切，例如当某些事件发生时，软件需要通知的适当人员，根据软件生成的日程表移动到正确教室的学生。要编写财务应用程序，软件工程师需要学习足够的财务概念来理解客户的 *requirements*。

需求

并且正确实施这些要求。

> 需求是机器通过其编程在问题域中施加的效果。

编程本身并不太复杂；编程以解决一个问题域，是1. 不仅软件工程师需要理解如何

1 我们在这里将“编程”的概念定义为能够用一种语言编写代码的人，但不一定需要知道任何或所有软件工程知识。

要实现软件，还要解决它试图解决的问题领域，这可能需要深入的专业知识。软件工程师还必须选择适用于他试图解决的问题领域的正确编程技术，因为许多在一个领域有效的技术可能在另一个领域不适用。例如，许多类型的应用不需要高性能的代码，但需要快速上市。在这种情况下，解释型语言非常受欢迎，因为它可以满足这种需求。然而，对于编写大型3D游戏或操作系统，编译型语言占主导地位，因为它可以生成此类应用所需的最高效的代码。

经常，对于软件工程师来说，学习非平凡领域（可能需要学士学位或以上才能理解这些领域）是过于困难的。此外，对于*domain expert*来说，学习足够的编程知识来将问题领域分解成软件工程师可以实施的小部分要容易得多。有时，领域专家会亲自实现软件。

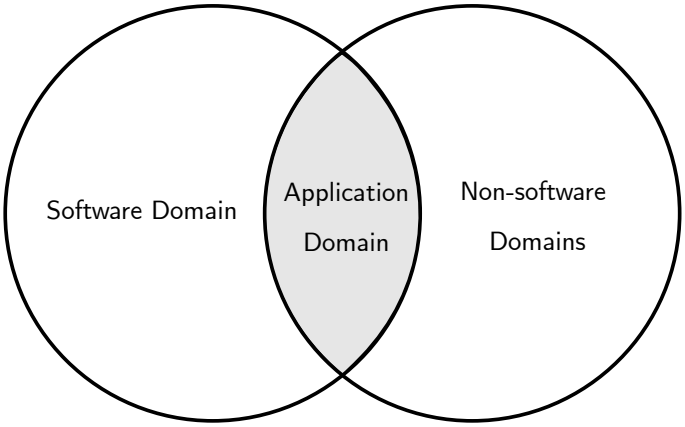


图1.1.1：问题领域：软件和非软件。

一个这样的场景示例是本书中介绍的领域：*operating system*。实现操作系统需要一定程度的电气工程（EE）知识。如果计算机科学（CS）课程不包含最低限度的EE课程，那么该课程的学生几乎没有机会实现一个可工作的操作系统。即使他们能实现一个，他们要么需要投入大量时间自学，要么在预编代码中填充。

定义的框架仅用于理解高级算法。因此，电子工程（EE）学生实现操作系统（OS）更容易，因为他们只需要学习几门核心计算机科学（CS）课程。事实上，通常只需要 *“C programming”* 和 *“Algorithms and Data Structures”* 课程就足够让他们开始编写设备驱动程序的代码，并将其推广到 *op- erating system.*

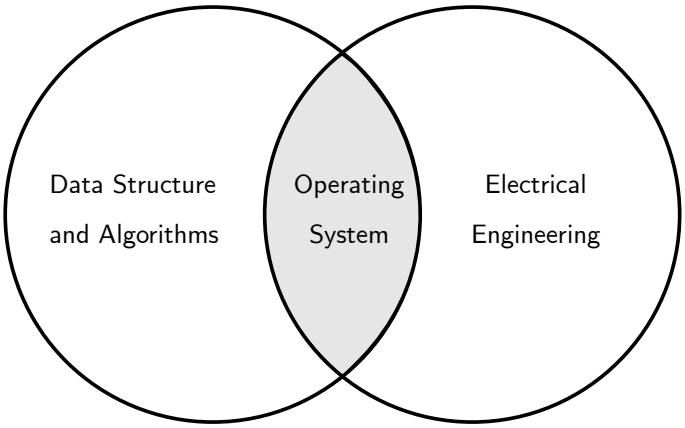


图1.1.2：操作系统域。

一点需要注意的是，软件本身就是其问题域。问题域不一定在软件及其自身之间划分。编译器、3D图形、游戏、密码学、人工智能等，都是软件工程领域的部分（实际上它更多的是计算机科学领域而不是软件工程领域）。一般来说，软件专属领域创建软件供其他软件使用。操作系统也是一个领域，但它与其他领域（如电气工程）重叠。为了有效地实现操作系统，需要学习足够的外部领域知识。软件工程师需要学习多少呢？至少，软件工程师应该足够了解硬件工程师准备的文档，以便使用（即编程）他们的设备。

学习一种编程语言，即使是C或汇编，并不意味着软件工程师可以自动擅长硬件编程或任何相关的底层编程领域。一个人可能花费10年、20年甚至他的一生来编写C/C++代码，但他仍然无法编写一个操作系统，仅仅是因为缺乏相关领域知识。就像学习英语并不意味着一个人可以自动擅长阅读用英语写的数学书籍。很多

超过所需。仅了解一到两种编程语言是不够的。如果一个程序员以编写软件为生，如果他不想让在业余时间学习编程的领域专家取代他的工作，他最好在软件之外的专业领域内专精于一到两个问题领域。

1.2 文档实现问题域

文档对于学习问题领域（实际上，任何事情）至关重要，因为信息可以以可靠的方式传递。显然，这种书面文本已经使用了数千年，用于从一代传到下一代传递知识。文档是非平凡项目的组成部分。没有文档:

- 新的人会发现加入项目要难得多。
- 维护项目更难，因为人们可能会忘记他们系统中重要未解决的错误或怪癖。
- 客户理解他们将要使用的产品具有挑战性。然而，文档不需要以书籍格式编写。它可以是从HTML格式到数据库格式，通过图形用户界面显示的任何内容。重要信息必须存储在某个安全且易于访问的地方。

有许多类型的文档。然而，为了便于理解问题域，需要编写这两份文档：*software requirement document*和 *software specification*。

1.2.1 Software Requirement Document

Software requirement

Software requirement document 包括一个要求和列表以及问题描述领域（Kovitz，1999）。

一个软件解决一个商业问题。但是，要解决的问题是由客户提出的。许多这些请求列出了一系列我们的软件需要满足的要求。然而，在交付软件时，一个列举的功能列表很少是有用的。正如在所述的

上一节，棘手之处不在于编程本身，而在于根据问题域进行编程。软件设计和实现的大部分内容依赖于对问题域的了解。对领域理解得越好，软件质量越高。例如，建造房屋已有数千年的历史，并且被充分理解，因此建造高质量的房屋很容易；软件也是如此。难以理解的代码通常是由于作者对问题域的无知。在本书的背景下，我们寻求理解各种硬件设备的底层工作原理。

因为软件质量取决于对问题域的理解，软件需求文档的量应包括问题域描述。

请注意，软件需求不是：

What vs How “什么”和“如何”是模糊的术语。什么是“什么”？它只包括名词吗？如果是这样，如果客户要求他的软件执行特定的操作步骤，比如网站上的客户购买流程，它现在包括“动词”了吗？然而，“如何”不应该是逐步操作吗？任何东西都可以是“什么”，任何东西都可以是“如何”。

Sketches 软件需求文档完全是关于问题域的。它不应该是实现的高级描述。一些问题可能看起来直接从其域描述映射到实现结构是直接的。例如：

- ▷ 用户可以从一个 ***drop-down menu*** 中选择书籍列表。
- ▷ 书籍存储在 ***linked list***”。
- ▷ 等等

在将来，将不再使用下拉菜单，所有书籍将直接以缩略图形式列在页面上。书籍可能将以图的形式重新实现，每个节点代表一本书，用于查找相关书籍，因为下一个版本将添加推荐功能。需求文档需要再次更新，以删除所有过时的实现细节，因此需要额外的努力来维护需求文档。

文档，当与实施同步的努力过多时，开发者放弃文档，然后每个人都开始抱怨文档是多么无用。

通常情况下，没有直接的单一映射。例如，普通计算机用户期望操作系统是运行具有GUI的程序或他们喜欢的电脑游戏的东西。但对于这样的需求，操作系统被实现为多层，每一层都隐藏了从上层隐藏的细节。要实现操作系统，需要来自多个领域的广泛知识，尤其是如果操作系统运行在非PC设备上。

最好在需求文档中包含与问题域相关的信息。测试需求文档质量的一个好方法是将其提供给领域专家进行校对，以确保他能够彻底理解材料。需求文档也作为帮助文档很有用，或者用于编写一个更容易的文档。

1.2.2 Software Specification

Software specification 文档声明了与期望行为相关的规则

Software specification

输出设备到所有可能的输入设备行为，以及问题域其他部分必须遵守的任何规则。Kovitz（1999）

简单来说，软件规范是接口设计，包括对问题域的约束，例如软件可以接受某些类型的输入，例如软件设计为接受英语，但不接受其他语言。对于硬件设备，总是需要规范，因为软件依赖于其硬编码的行为。实际上，大多数情况下，硬件规范都是明确定义的，其中包含最细微的细节。必须这样，因为一旦硬件在物理上制造出来，就无法回头，如果存在缺陷，将对公司造成财务和声誉的双重打击。

注意，类似于需求文档，规范仅关注接口设计。如果实现细节泄露进来，它将成为负担

同步实际实现与规范，并很快将被废弃。

另一个重要说明是，尽管规范文档很重要，但它不必在实现*before*时产生。它可以按任何顺序准备：在完整实现之前或之后；或者与实现同时进行，当某些部分完成时，接口已准备好记录在规范中。无论采用何种方法，重要的是最终要有一个完整的规范。

1.3 文档编写 x86 操作系统的指南

当问题域与软件域不同时，需求文档和规范通常分开。然而，如果问题域在软件内部，规范通常包括两者，并且两者的内容可以相互混合。如前几节所示，为了实现操作系统，我们需要收集相关文档以获得足够的领域知识。

以下文档如下：

- 英特尔® 64 和 IA-32 架构软件开发者手册（第 1 卷、第 2 卷、第 3 卷）
- 英特尔® 3系列Express芯片组家族数据表
- 系统V应用程序二进制接口

除了英特尔官方网站外，本书网站也提供方便2的文档。

英特尔文档明确划分了需求和规范部分，但使用了不同的名称。对应需求文档的部分是一个称为 *"Functional Description"* 的章节，主要由领域描述组成；对于规范，*"Register Description"* 章节描述了所有编程接口。两份文档均不包含不必要的实现细节3。英特尔文档也非常出色...

2 英特尔可能会更改其网站上的文档链接，因此本书不包含任何文档链接，以避免读者产生混淆。

3 正如应当的那样，这些细节是商业机密。

示例说明如何编写良好的需求/规范，如本章所述。

除了英特尔文档外，相关章节中还将介绍其他文档。

2

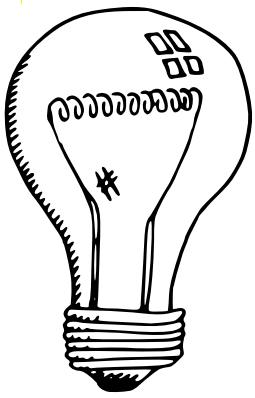
From hardware to software: Layers of abstraction

本章介绍了硬件和软件如何连接在一起，以及软件如何以物理形式表示的直观理解。

2.1 比特的物理实现

所有电子设备，从简单到复杂，都操纵这种流动以在现实世界中实现预期的效果。计算机也不例外。当我们编写软件时，我们间接地在物理层面上操纵电流，以便底层机器产生预期的效果。为了理解这个过程，我们考虑一个简单的灯泡。灯泡可以通过开关在开和关两种状态之间切换，周期性地：关表示数字0，开表示数字1。

图2.1.1：一盏灯泡



然而，一个问题是需要人工手动干预。所需的是基于电压水平的自动开关，如上所述。为了实现电气信号的自动切换，需要一种称为*transistor*的设备，由威廉·肖克利、约翰·巴丁和沃尔特·布拉顿发明。这项发明启动了整个计算机行业。

在核心，一个 *transistor* 只是一个其值可以变化的电阻
在输入电压值上。

transistor

具有此属性，晶体管可以用作电流放大器（电压越高，电阻越小）或根据电压水平开关电信号的开和关（阻塞和解除电子流动）。在0伏时，电流无法通过晶体管，因此它像一个开路的电路（灯泡关闭），因为电阻值足以阻止电流流动。同样，在+3.5伏时，由于电阻值降低，电流可以通过晶体管，有效地启用电子流动，因此它像一个闭合开关的电路。

图2.1.2：现代晶体管

一个比特有两种状态：0 和 1，这是所有数字系统和软件的基石。类似于可以打开和关闭的灯泡，比特由电源提供的这个电流制成：比特 0 表示为 0 v（无电子流动），比特 1 是 +3.5 v 到 +5 v（电子流动）。晶体管可以正确实现比特，因为它可以根据电压水平调节电子流动。

如果您想获得更深入的说明
电子移动，你应该看看YouTube上Ben Eater的“半导体如何工作”视频。

2.1.1 MOSFET transistors

经典的晶体管发明开辟了微数码设备的新世界。在发明之前，真空管——也就是更高级的灯泡——被用来表示0和1，需要人工开关。*MOSFET*，或***M****etal–****O****xide–****S****emiconductor* ***F****ield-****E****ffect*

MOSFET

T*ransistor*, 1959年由贝尔实验室的Dawon Kahng和Martin M. (John) Atalla发明，是经典晶体管的改进版本，更适合数字设备，因为它在两个状态0和1之间需要更短的切换时间，更稳定，功耗更低，更容易生产。

有两种MOSFET与两种晶体管类似：n-MOSFET和p-MOSFET。n-MOSFET和p-MOSFET也简称为NMOS和PMOS晶体管。

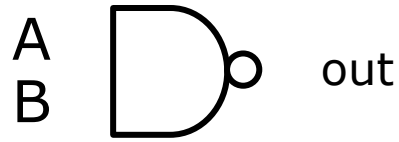
2.2 超越晶体管：数字逻辑门

所有数字设备都是用逻辑门设计的。一个 *logic gate* 是一种设备
该实现布尔函数。每个逻辑门包括一个数字

logic gate

输入和输出。所有计算机操作都是基于逻辑门的组合，而逻辑门只是布尔函数的组合。

图2.2.1：示例：NAND门



2.2.1 The theory behind logic gates

逻辑门仅接受二进制输入1并产生二进制输出。在[1]

其他词，逻辑门是转换二进制值的函数。幸运的是

一个专门处理二进制值的数学分支已经存在，称为*Boolean Algebra*，由乔治·布尔在19th世纪开发。以坚实的数学理论为基础，创建了逻辑门.。由于逻辑门实现布尔函数，一组布尔函数是*functionally complete*，如果这个集合可以构造所有其他布尔

输入为0或1。

，

functionally complete

函数可以从中构造。后来，查尔斯·桑德斯·皮尔士（在1880-1881年期间）证明了仅使用NOR或NAND布尔函数就足以创建所有其他布尔逻辑函数。因此，NOR和NAND门是功能完备的皮尔士（1933）。门只是布尔逻辑函数的实现，因此NAND或NOR门足以实现***all***其他逻辑门。CMOS电路可以实现的 simplest gates 是反相器（NOT门），而NAND门则由此而来。有了NAND门，我们就可以自信地实现其他所有功能。这就是为什么晶体管和CMOS电路的发明以及随后的CMOS电路革命改变了计算机行业。

我们应该认识到并欣赏所有编程语言中可用的布尔函数是多么强大。

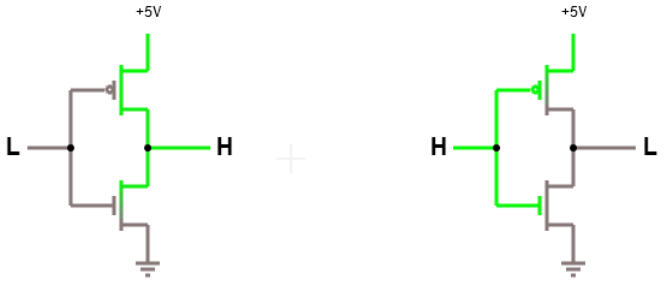
如果您想了解为什么以及如何从NAND门可以创建所有布尔函数和计算机，我建议这门课程 *Build a Modern Computer from First Principles: From Nand to Tetris* 可在Coursera上找到：`https://www.coursera.org/learn/build-a-computer`。课程结束后，更进一步，你应该在Edx上学习系列课程 *Computational Structures*。

2.2.2 Logic Gate implementation: CMOS circuit

每个逻辑门下面都有一个称为 ***CMOS*** - ***C****omplementary* 的电路

MOS*FET*CMOS由两个互补晶体管组成，*NMOS*和*PMOS*.。最简单的CMOS电路是一个反相器或一个*NOT*门：

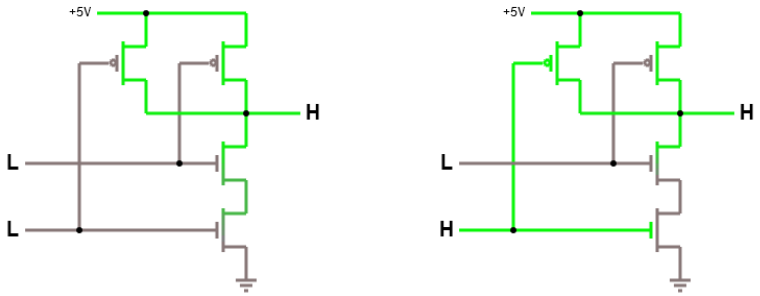
CMOS



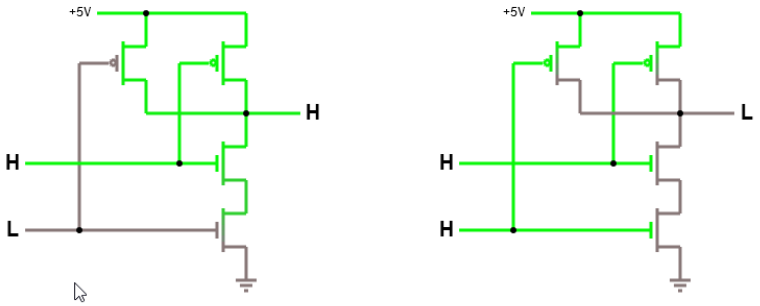
(a) When input is low **(b)** When input is high

图2.2.2：逆变器中的电子流动。输入在左侧，输出在右侧。上部元件是PMOS，下部元件是NMOS，两者都连接到输入和输出。（来源：使用`http://www.falstad.com/circuit/`创建）

From NOT gate, a NAND gate can be created:



(a) Input = 00, Ouput = 1 **(b)** Input = 01, Ouput = 1



(c) Input = 10, Output = 1 **(d)** Input = 11, Output = 0

图2.2.3：NAND门的电子流动。

从NAND门，我们得到所有其他门。如所示，如此简单的电路在日常程序语言中执行逻辑运算符，例如NOT运算符~直接由反相器电路执行，运算符&由AND电路执行，依此类推。代码不是在魔法黑盒上运行的。相反，代码执行是精确且透明的，通常就像运行一些硬连线电路一样简单。当

我们编写软件，我们只是在物理层面上简单地操纵电流，以运行适当的电路以产生所需的结果。然而，整个过程似乎与任何涉及电流的思想都没有关系。这就是真正的魔法，很快就会解释。

CMOS的一个有趣特性是***a k-input gate uses k PMOS and k NMOS transistors*** (Wakerly，1999)。所有逻辑门都是由NMOS和PMOS晶体管成对构建的，门是简单到复杂的所有数字设备的构建模块，包括任何计算机。多亏了这种模式，可以区分实际的物理电路实现和逻辑实现。数字设计是通过设计逻辑门，然后将其“编译”成物理电路来完成的。实际上，我们将在后面看到，逻辑门成为描述电路如何工作的语言。了解CMOS的工作原理对于理解计算机的设计至关重要，从而了解计算机的工作原理[2]。

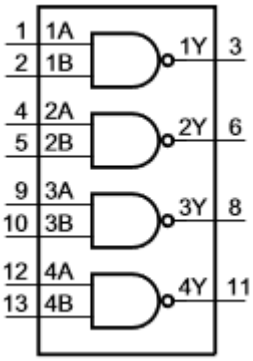
最后，一个具有其电线和晶体管的实现电路以物理形式存储在一个称为*chip*的封装中。*chip*是一个集成电路被蚀刻的衬底。然而，在消费市场上，芯片也指一个完全封装的集成电路。根据上下文，其理解不同。

2 再次，如果你想了解逻辑门如何使计算机工作，请考虑之前在 Coursera 和 Edx 上推荐的课程。

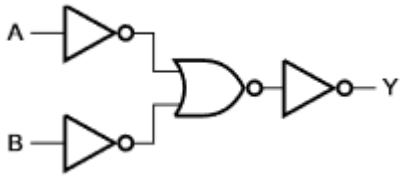
图2.2.4：74HC00芯片物理视图

示例2.2.1. 74HC00是一种带有四个2输入NAND门的芯片。该芯片具有8个输入引脚和4个输出引脚，1个引脚用于连接到电压源，1个引脚用于连接到地。这个设备是我们可以触摸并使用的NAND门的物理实现。但芯片上不仅有单个门，还有4个可以组合的门。每个组合实现不同的逻辑功能，有效地创建了其他逻辑门。正是这个特性使芯片变得流行。

每个上面的门只是一个简单的NAND电路，如之前所示，具有电子流动。然而，许多这些NAND门芯片组合可以构建一台简单的计算机。在物理层面上，软件只是电子流动。

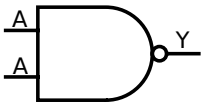


(a) Logic diagram of 74HC00

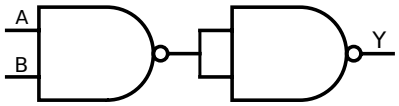


(b) Logic diagram of one NAND gate

Figure 2.2.574HC00 逻辑 diagrams (Source74HC00 数据表, `http://www.scrpdf.com/pdf/Semiconductors_new/Logic/74HCT/74HC_HCT00.pdf`)

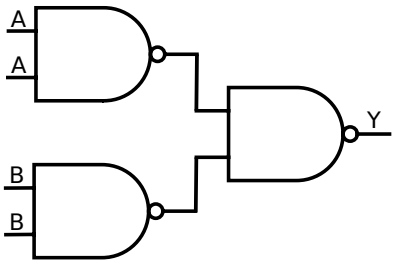


(a) NOT gate

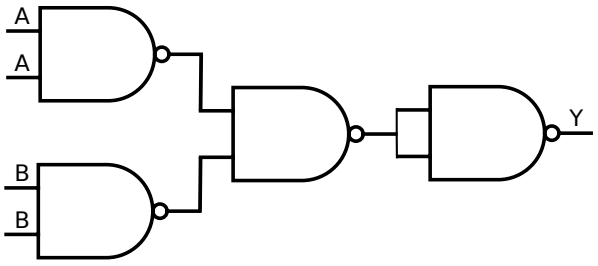


(b) AND gate

Figure 2.2.6: G由...构建的ates NAND gates, 每个接受2个input signals and 生成 1 个输出 signal.



(c) OR gate



(d) NOR gate

如何使用74HC00创建上述门？很简单：因为每个门都有2个输入引脚和1个输出引脚，我们可以将一个NAND门的输出写入另一个NAND门的输入，从而将NAND门串联起来，产生上述的图。

2.3 逻辑门之外：机器语言

2.3.1 Machine language

基于门构建，因为门只接受一系列的0和1，硬件设备只理解0和1。然而，设备以系统化的方式接受0和1。*Machine language*是一组独特的

Machine language

位模式，设备可以识别并执行相应的操作。一个 *machine instruction* 是设备可以识别的唯一位模式。在计算机系统中，具有其语言的设备被称为 ***CPU*** - ***C****entral* ***P****rocessing* ***U****nit*，它控制计算机内部的所有活动。例如，在 x86 架构中，模式 `10100000` 表示告诉 CPU 添加两个数字，或 `000000101` 使计算机停止。在计算机的早期，人们必须完全用二进制编写。

为什么这样的比特模式会导致设备执行某些操作？原因是每条指令下面都有一个实现该指令的小电路。类似于计算机程序中通过名称调用函数/子程序，比特模式是CPU内部一个执行的小函数的名称，当CPU找到它时就会执行。

注意，CPU 并非唯一具有其语言的设备。CPU 只是一个表示控制计算机系统的硬件设备的名称。一个硬件设备可能不是 CPU，但仍具有其语言。具有自己机器语言的设备是 *programmable device*，因为用户可以使用该语言来命令设备执行不同的操作。例如，打印机有其一套指令，用于指导其如何打印一页。

示例2.3.1。用户可以使用74HC00芯片而无需了解其内部结构，只需了解使用该设备的接口。首先，我们需要了解其布局：

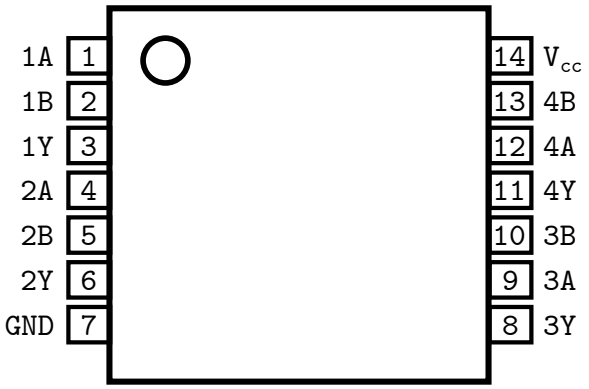


图2.3.1：74HC00引脚布局（来源：74HC00数据手册，http://www.nxp.com/documents/data_sheet/74HC_HCT00.pdf）

然后，每个引脚的功能：

Symbol	Pin	Description
1A to 4A	1, 4, 9, 12	data input
1B to 4B	2, 5, 10, 13	data input
1Y to 4Y	3, 6, 8, 11	data output
GND	7	ground (0 V)
V_{cc}	14	supply voltage

表2.3.1：引脚描述（来源：74HC00数据表，http://www.nxp.com/documents/data_sheet/74HC_HCT00.pdf）

最后，如何使用引脚：

Input		Output
nA	**nB**	**nY**
L	L	H
L	X	H
X	L	H
H	H	L

表2.3.2：功能描述

功能描述提供了一个包含所有可能的引脚输入和输出的真值表，该表还描述了设备中所有引脚的用法。用户不需要了解实现细节，但可以通过这样的表格来使用设备。我们可以说，上面的真值表是设备的机器语言。由于设备是数字的，其语言是一系列二进制字符串：

▷ n 是一个数字，可以是 1、2、3 或 4

▷ H = 高压电平；L = 低压电平；X = 不关心。

▷ 该设备有8个输入引脚，这意味着它接受8位二进制字符串。

▷ 该设备有4个输出引脚，这意味着它从8位输入产生4位二进制字符串。

输入字符串的数量是设备所能理解的，输出字符串的数量是设备所能说出的。它们共同构成了设备的语言。尽管这个设备很简单，但它所能接受的语言包含相当多的二进制字符串：$2^8 + 2^4 = 272$。然而，这个数量对于像CPU这样的复杂设备来说只是微不足道的一小部分，后者有数百个引脚。

当保持原样时，74HC00只是一个具有两个4位输入的NAND器件[3]。

[3] 或者简单地称为4位NAND门，因为它最多只能接受4位输入。

	Input								**Output**			
Pin	1A	1B	2A	2B	3A	3B	4A	4B	1Y	2Y	3Y	4Y
Value	1	1	0	0	1	1	0	0	0	1	0	1

输入和输出如图所示：

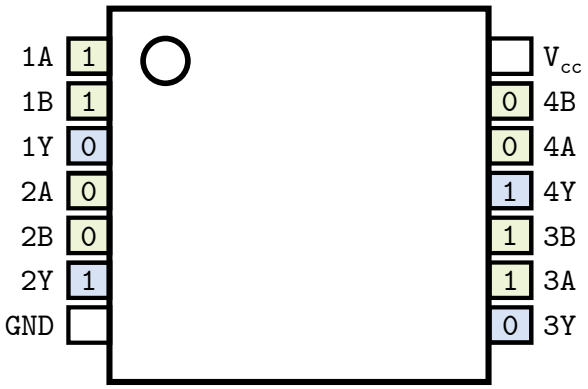
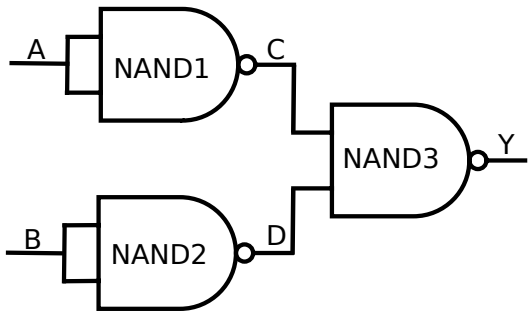
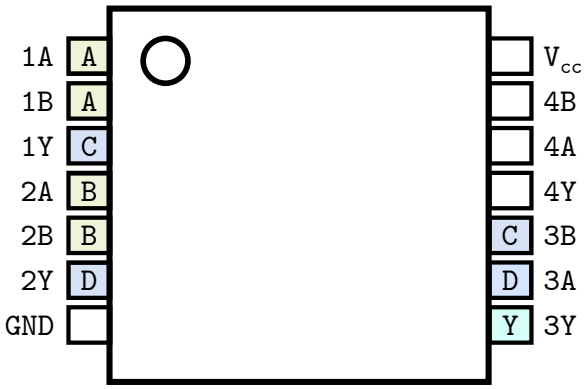


图2.3.2：接收对应二进制字符串的数字信号时的引脚。绿色信号是输入；蓝色信号是输出。

另一方面，如果实现了或门，我们只能从74HC00构建一个2输入或门，因为它需要3个与非门：2个输入与非门和1个输出与非门。每个输入与非门仅代表或门的1位输入。在以下图中，每个输入与非门的引脚始终设置为相同的值（要么两个输入都是A，要么两个输入都是B），以表示最终或门的单比特输入：



(a) 2位OR门逻辑图，由3个4引脚的NAND门构建，仅用于2位输入。



(b) Pin 3A and 3B take the values from 1Y and 2Y.

图2.3.3：2位或门实现

表2.3.3：或逻辑图的真值表。

A	B	C	D	Y
0	0	1	1	0
0	1	1	0	1
1	0	0	1	1
1	1	0	0	1

要实现一个4位或门，我们需要总共四个配置为或门的74HC00芯片，如图2.3.4所示，封装成一个单独的芯片。

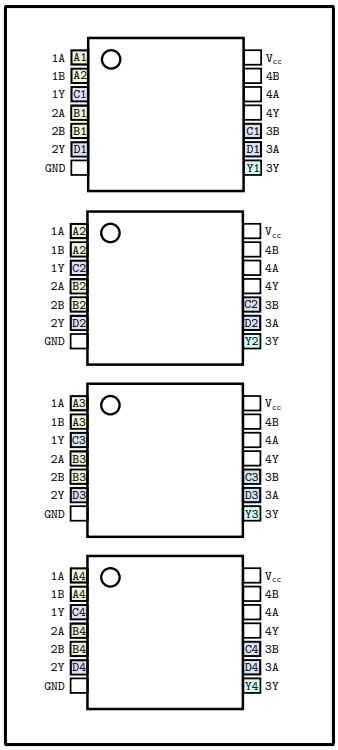


图2.3.4：由四个74HC00器件构成的4位OR芯片

2.3.2 Assembly Language

汇编语言是二进制机器代码的符号表示，通过给位模式起助记符名称。当程序员必须编写0和1时，这是一大改进。例如，程序员只需写`hlt`来停止计算机。这种抽象使得CPU执行的指令更容易记住，因此可以记住更多的指令，花费在查找CPU手册以找到位形式指令的时间更少，结果代码编写得更快。

理解汇编语言对于低级编程领域至关重要，即便到今天也是如此。程序员想要理解的指令越多，对机器架构的理解就越深。

示例2.3.2。我们可以构建一个包含2条汇编指令的设备：

```
or   <op1>, <op2>
nand <op1>, <op2>
```

▷ `or` 接受两个4位操作数。这相当于由4个74HC00芯片构建的4输入或门设备。

▷ `nand` 接受两个4位操作数。这对应于一个74HC00芯片，保持不变。

本质上，示例2.3.1中的门实现了指令。到目前为止，我们只指定输入和输出，并将其手动喂入设备。也就是说，要执行一个操作：

▷ 通过手选择设备。

▷ 手动将电信号放入引脚。

首先，我们希望自动化设备选择的过程。也就是说，我们希望简单地编写汇编指令，而实现该指令的设备将被正确选择。解决这个问题很简单：

▷ 为每条指令分配一个二进制代码索引，称为 *operation code* 或 *opcode*，并将其作为输入的一部分嵌入。每条指令的值如表2.3.4所示。

表2.3.4：指令-操作码映射。

Instruction	Binary Code
nand	00
or	01

每个输入现在在开头包含额外的数据：一个操作码。例如，指令：

```
nand 1100, 1100
```

对应于二进制字符串：0011001100。前两位 00 编码一个 nand 指令，如上表所示。

▷ 添加另一个设备以选择设备，基于特定于指令的二进制代码。

这样的设备被称为 *decoder*，是CPU中的一个重要组件，用于决定使用哪个电路。在上面的例子中，当将 0011001100 输入解码器时，因为操作码是 00，数据会被发送到NAND设备进行计算。

最后，编写汇编代码只是编写设备能理解的二进制字符串的一种更简单的方式。当我们编写汇编代码并将其保存到文本文件中时，一个名为*assembler*的程序将文本文件***assembler***转换为设备能理解的二进制字符串。那么，汇编器最初是如何存在的呢？假设这是世界上第一个汇编器，那么它是用二进制代码编写的。在下一个版本中，生活变得更简单：程序员用汇编代码编写汇编器，然后使用第一个版本来编译它自己。然后，这些二进制字符串被存储在另一个设备中，稍后可以检索并发送到解码器。一个*storage device*

storage device

这是存储机器指令的设备，它是一个用于保存0和1状态的电路数组。

解码器由类似于其他数字设备的逻辑门构建而成。然而，存储设备可以是任何可以存储0和1并且可检索的设备。存储设备可以是使用磁性来存储信息的磁化设备，或者它可以是由可以改变并记住当施加电压时的状态的电路制成的。无论使用什么技术，只要设备可以存储数据并且可以访问以检索数据，就足够了。事实上，现代设备如此复杂，不可能也不必要理解每个实现细节。相反，我们只需要学习设备暴露的接口，例如引脚。

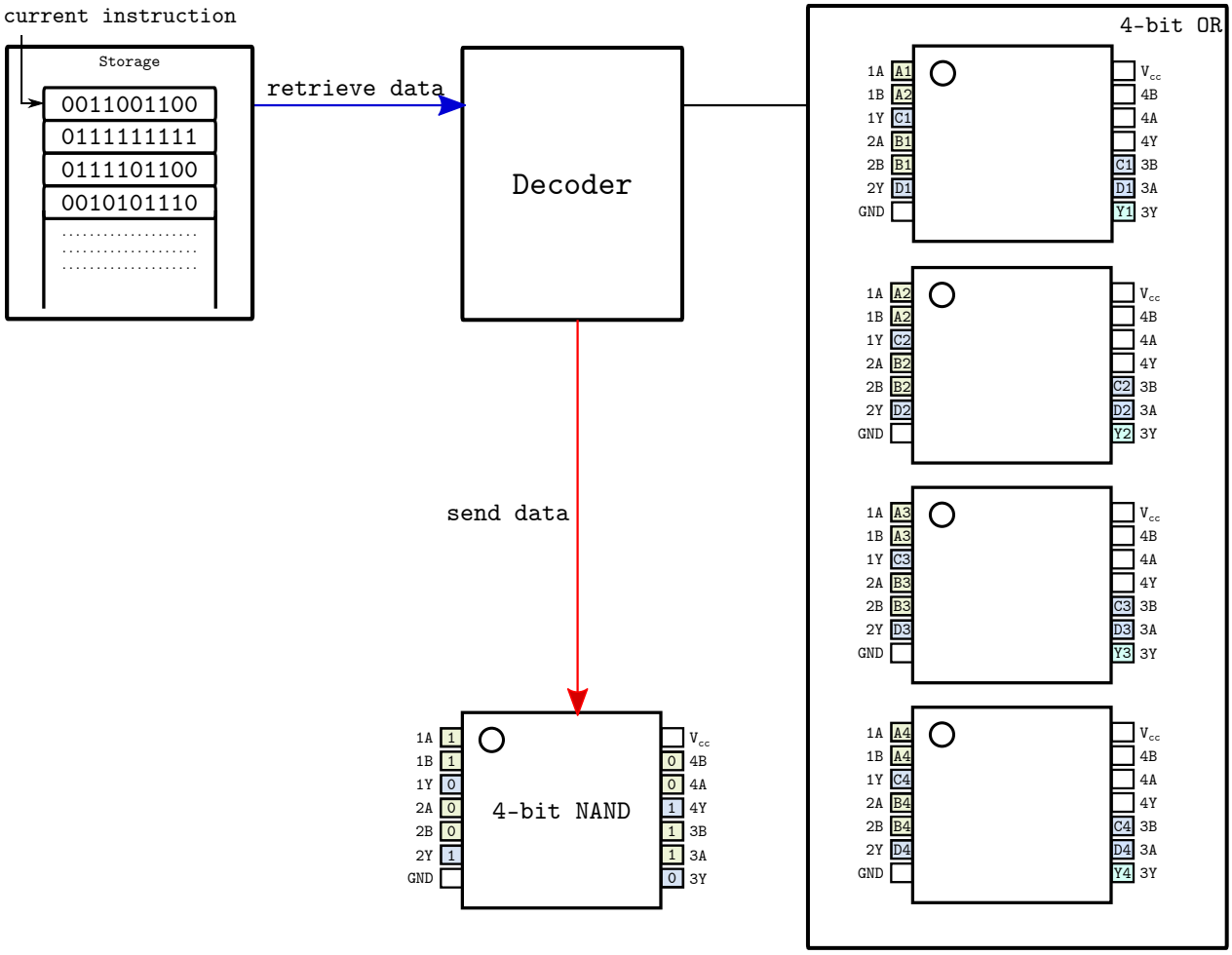


图2.3.5：解码器检索由箭头指向的当前指令，并选择执行 nand 指令的NAND设备。

计算机本质上实现了这个过程：

▷ *Fetch* 存储设备的一条指令。

▷ *Decode* 指令。

▷ *Execute* 指令。

简而言之，一个获取-解码-执行周期。上述设备极其基础，但它已经代表了一个具有*fetch – decode – execute*周期的计算机。可以通过添加更多设备和为指令分配更多操作码来实现更多指令，然后相应地更新解码器。阿波罗导航计算机，一种从1961年到1972年为阿波罗太空计划生产的数字计算机，完全由NOR门组成——创建时选择NAND门之外的另一种选择。

其他逻辑门。同样地，如果我们继续改进我们的假设设备，它最终会变成一台全功能的计算机。

2.3.3 Programming Languages

汇编语言比编写0和1更高级。随着时间的推移，人们意识到许多汇编代码块具有重复的使用模式。如果能够不在所有地方重新编写所有重复的代码块，而是简单地用更容易使用的文本形式引用这些代码块，那就太好了。例如，一个汇编代码块检查一个变量是否大于另一个变量，如果是，则执行一个代码块，否则执行另一个代码块；在C语言中，这样的汇编代码块由一个类似于人类语言的`if`语句表示。

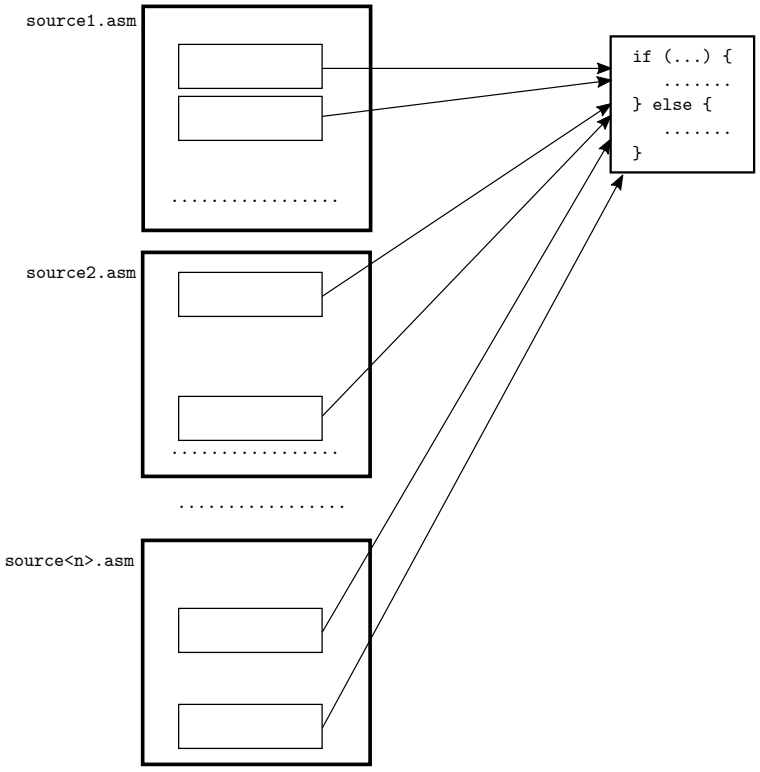


图2.3.6：重复的组装模式被概括成一种新的语言。

人们创建了文本形式来表示常见的汇编代码块，例如上面的 `if` 语法，然后编写一个程序将文本形式转换为汇编代码。将此类文本形式转换为机器码的程序称为 *compiler*：

compiler

任何编程语言可以实现的软件逻辑，硬件

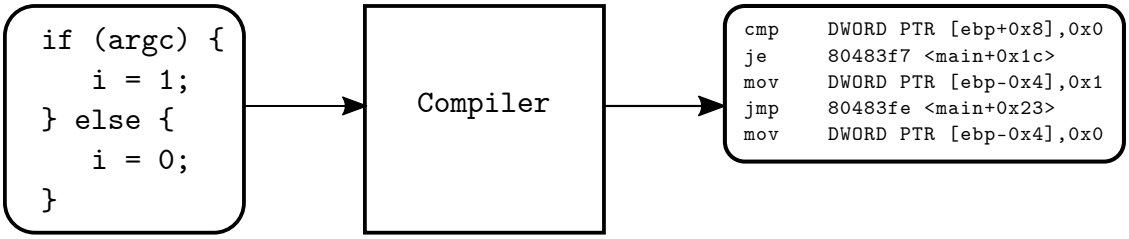


图2.3.7：从高级语言回到低级语言。

也可以实现。反之亦然：任何在电路中实现的硬件逻辑都可以用编程语言重新实现。简单的原因是，编程语言、汇编语言、机器语言或逻辑门只是表达计算的语法。软件无法实现硬件无法实现的事情，因为编程语言只是使用底层硬件的更简单方式。最终，编程语言被翻译成CPU有效的机器指令。否则，代码无法运行，因此是无用的软件。相反，软件可以做硬件（运行软件的硬件）能做的一切，因为编程语言只是使用硬件的更简单方式。

实际上，尽管所有语言在能力上都是等效的，但并非所有语言都能表达彼此的程序。编程语言在光谱的两端之间变化：高级和低级。

编程语言级别越高，它就越远离硬件。在一些高级编程语言中，例如Python，尽管程序员能够像低级编程语言一样进行相同的计算，但他们无法操作底层硬件。原因是高级语言想要隐藏硬件细节，以使程序员从处理与当前问题域无关的不相关细节中解放出来。然而，这种便利并非免费：它要求软件携带额外的代码来管理硬件细节（例如内存），从而使代码运行速度变慢，并使得硬件编程变得困难或不可能。编程语言施加的抽象越多，编写低级软件（如硬件驱动程序或操作系统）就越困难。这就是为什么C通常被选为编写操作系统的语言，因为C只是底层硬件的一个薄包装，使得

理解一个硬件设备在执行某段C代码时是如何运行的非常简单。

每种编程语言都代表了一种对程序思考的方式。高级编程语言有助于关注与硬件完全不相关的领域，在这些领域中，程序员的表现比计算机性能更重要。低级编程语言有助于关注机器的内部工作原理，因此最适合与控制硬件相关的领域。这就是为什么存在这么多语言的原因。使用正确的工具来完成正确的工作，以实现最佳结果。

2.4 抽象

Abstraction 这是一种隐藏与上下文问题无关复杂性的技术。例如，除了最低层之外不使用任何其他层来编写程序：使用电路。不仅一个人需要深入了解电路的工作原理，这使得设计电路变得更加晦涩，因为设计者必须查看原始电路，但又要从更高的层次，如逻辑门，进行思考。这是一个分散注意力的过程，因为设计者必须不断将想法转化为电路。设计者可以简单地直接思考他的高级想法，然后稍后将其转化为电路。这不仅更有效率，而且更准确，因为设计者可以将所有精力集中在用高级思维验证设计上。当新设计师到来时，他可以轻松理解高级设计，从而可以继续开发或维护现有系统。

2.4.1 Why abstraction works

在所有层中，抽象自身显现：{v*}

▷ 逻辑门抽象掉了CMOS的细节。▷机器语言抽象掉了逻辑门的细节。▷汇编语言抽象掉了机器语言的细节。▷编程语言抽象掉了汇编语言的细节。

我们观察到下层构建上层时的重复模式：

▷ 一层有重复的图案。然后，从这个重复的图案中提取出来，在其上构建一种语言。

▷ 更高层去除层特定（非重复）细节，以关注重复细节。

▷ 重复的细节使用了比底层语言更新、更简单的语言。

要认识到的是，每一层只是 *a more convenient language to* ***describe*** *the lower layer*。只有当用高层的语言完全创建了一个描述后，它才会用低层的语言进行 *implemented*。

▷ CMOS层有一个重复的模式，确保逻辑门能够可靠地转换为CMOS电路：***a k-input gate uses k PMOS and k NMOS transistors*** (Wakerly，1999)。由于数字设备仅使用CMOS，因此出现了一种语言来描述高级概念，同时隐藏CMOS电路：逻辑门。

▷ 逻辑门隐藏了电路的语言，专注于如何实现原始布尔函数并将它们组合以创建新函数。所有逻辑门都接收输入并以二进制数字生成输出。多亏了这种重复的模式，逻辑门被隐藏在新的语言：汇编语言中，这是一组预定义的二进制模式，导致底层门执行操作。

▷ 很快，人们意识到许多重复的模式源于汇编语言。表达相同或类似想法的汇编代码块在汇编源文件中重复出现。有许多这样的想法可以可靠地翻译成汇编代码。因此，这些想法被提取出来，构建成今天每个程序员学习的通用高级编程语言。

重复模式是抽象的关键。正是由于重复模式，抽象才能发挥作用。没有它们，就无法构建语言，因此

没有抽象。幸运的是，人类已经发展了一套研究模式的系统学科：数学。正如英国数学家G. H. Hardy（2005年）所说：

> 一个数学家，就像画家或诗人一样，是模式的创造者。如果他的模式比他们的更持久，那是因为它们是用思想制作的。

这不是一个数学公式，它表示一个模式吗？一个变量代表由约束给出的具有相同属性的值？数学为识别和描述自然界中现有的模式提供了一个形式化的系统。因此，这个系统当然可以应用于数字世界，而数字世界只是现实世界的一个子集。数学可以用作一种通用语言，帮助层之间的翻译更容易，并有助于理解层。

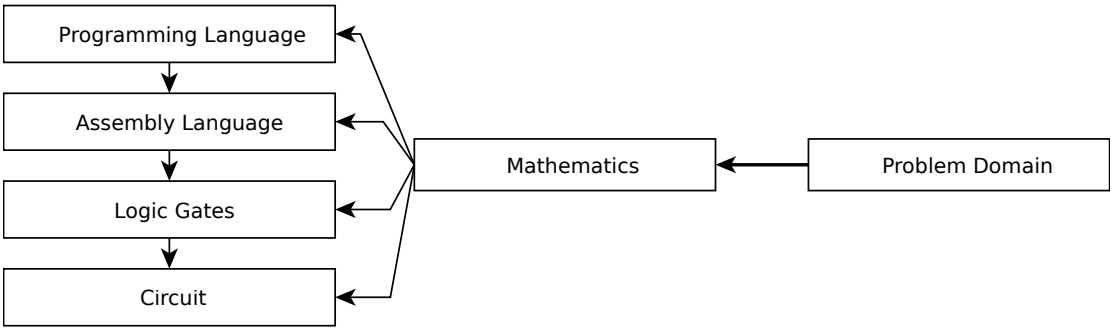


图2.4.1：数学作为所有层次的通用语言。由于所有层次都可以用它们的技术表达数学，每个层次都可以被翻译成另一个层次。

2.4.2 Why abstraction reduces complexity

通过构建语言进行抽象无疑通过剥离与问题无关的细节来提高生产力。想象一下，编写程序时没有任何其他布局，除了最低层：电路。这就是复杂性的产生方式：当用低级语言表达高级思想时，就像上面例子所展示的那样。不幸的是，目前软件的情况就是这样，因为编程语言更强调软件而不是问题域。也就是说，没有先验知识，用一种语言编写的代码无法表达其目标域的知识。换句话说，*a language is expressive if its syntax is designed to express the problem domain it is trying to solve*。考虑这个例子：也就是说，它将做的*what*

the *how* it will do.

示例2.4.1. Graphviz (`http://www.graphviz.org/`) 是一种可视化软件，它提供了一种称为 `dot` 的语言来描述图：

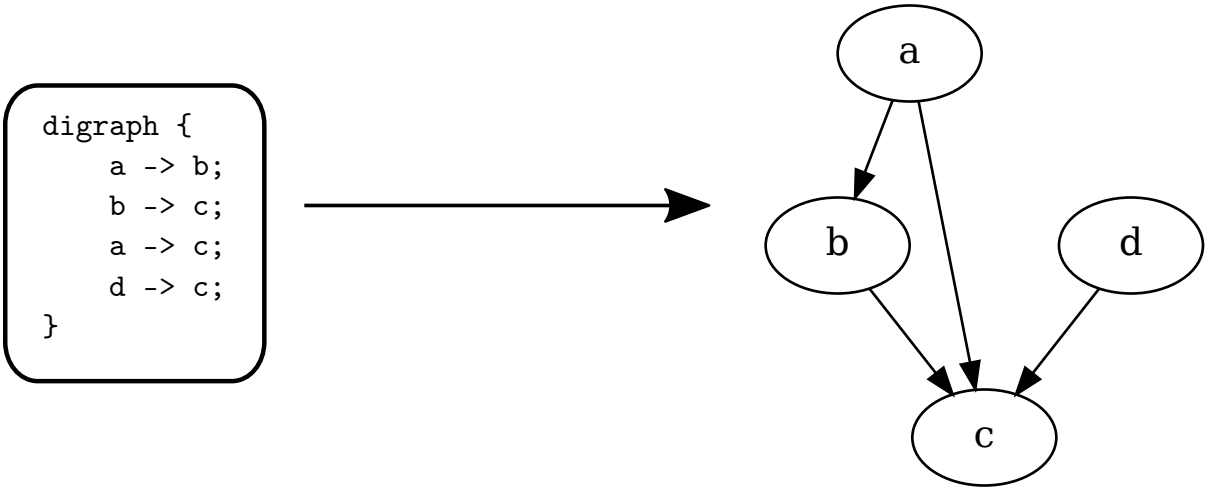


图2.4.2：从图描述到图。

如所见，代码完美地表达了图是如何连接的。即使是非程序员也能轻松理解和使用这种语言。在C语言中的实现会更麻烦，而且这还假设已经有了绘制图的函数。要画一条线，在C语言中我们可能会写如下：

```
draw_line(a, b);
```

然而，与以下相比，它仍然很冗长：

```
a -> b;
```

此外，`a` 和 `b` 必须在 C 中定义，与 `dot` 语言中的隐式节点相比。然而，如果不考虑冗长性，C 仍然有限制：它不能改变其语法以适应问题域。一种特定领域的语言可能甚至更加冗长，但它使领域更容易理解。如果必须用 C 表达问题域，那么它就受到 C 语法的限制。由于 C 不是

专门针对问题域的语言，但它是*general-purpose*编程语言，领域知识隐藏在实现细节中。因此，需要一个C程序员来解码和提取领域知识。如果不能提取领域知识，那么软件就无法进一步开发。

示例2.4.2。Linux中充满了由许多特定领域语言控制的程序，并放置在`/etc`目录中，例如一个网络服务器。而不是重新编写软件，为它制作了一种领域无关的语言。

一般来说，能够表达问题域的代码必须被领域专家所理解。即使在软件领域内，从重复的编程模式中构建一种语言也是有用的。这有助于人们意识到代码中存在这样的模式，从而使软件更容易维护，因为软件结构作为一门语言是可见的。只有能够根据问题域自我变形以适应的编程语言才能实现这一目标。这种语言被称为*programmable programming language*。不幸的是，这种将软件结构可视化的方法并不受程序员青睐，因为必须为此创建一种新的语言以及支持它的新工具链。因此，软件结构和领域知识被隐藏在通用语言语法编写的代码中，如果一个程序员不熟悉甚至不知道代码模式的存在，那么理解代码是毫无希望的。一个典型的例子是阅读控制硬件的C代码，例如操作系统：如果一个程序员对硬件一无所知，那么他不可能用C语言阅读和编写操作系统代码，即使他可能已经写了20年的应用C代码。

通过抽象，软件工程师也可以理解设备的内部工作原理，而无需具备物理电路设计的专业知识，使软件工程师能够编写控制设备的代码。逻辑实现与物理实现之间的分离还意味着，即使在底层技术发生变化时，门设计也可以被重用。

已更改。例如，在某个遥远的未来，生物计算机可能成为现实，门电路可能不是用CMOS实现，而是某种生物细胞，例如活细胞；在任一技术：电或生物，只要逻辑门在物理上实现，就可以实现相同的计算机设计。

3

Computer Architecture

编写底层代码时，程序员必须了解计算机的架构。这类似于当一个人在软件框架中编写程序时，他必须知道框架解决哪些类型的问题，以及如何通过提供的软件接口使用框架。但在定义计算机架构之前，我们必须确切地了解什么是计算机，因为许多人仍然认为计算机是我们放在桌上的普通电脑，或者最多是服务器。计算机有各种形状和大小，是人们从未想象过它们是计算机的设备，并且代码可以在这样的设备上运行。

3.1 什么是计算机?

一个 *computer* 是由至少一个处理器（CPU）、***computer*** 存储设备和输入/输出接口组成的硬件设备。所有计算机可以分为两种类型:

Single-purpose computer 这是一个在*hardware level*构建的用于特定任务的计算机。例如，专用应用编码器/解码器、计时器、图像/视频/声音处理器。

General-purpose computer 是一种可以编程的计算机（无需修改其硬件）以模拟专用计算机的各种功能

计算机。

3.1.1 Server

一个 *server* 是一种通用高性能计算机，具有巨大的...

server

来源为提供大规模服务以服务广泛受众。听众是那些将个人电脑连接到服务器的人。

图3.1.1：刀片服务器。每个刀片服务器都是一种模块化设计的计算机，旨在优化物理空间和能源的使用。刀片服务器的机箱被称为*chassis*。（来源：维基百科，作者：Victorgrigas）

3.1.2 Desktop Computer

一个 *desktop computer* 是一种通用计算机，具有输入和输出功能。

desktop computer

为人类用户设计的系统，具有足够的资源以供常规使用。输入系统通常包括鼠标和键盘，而输出系统通常由一个可以显示大量像素的显示器组成。计算机被封装在一个足够大的机箱中，可以放置各种计算机组件，如处理器、主板、电源、硬盘等。

图3.1.2：一台典型的台式计算机。

3.1.3 Mobile Computer

一个 *mobile computer* 类似于资源较少的台式电脑
但可以随身携带。

mobile computer

(a) A laptop **(b)** A tablet **(c)** A mobile phone

图3.1.3：移动计算机

3.1.4 Game Consoles

游戏机类似于台式电脑，但针对游戏进行了优化。游戏机不是使用键盘和鼠标，而是使用游戏控制器作为输入系统，这是一种带有少量按钮的设备，用于控制屏幕上的对象；输出系统是电视。机箱与台式电脑相似，但更小。游戏机使用定制处理器和图形处理器，但与台式电脑中的类似。例如，第一代Xbox使用定制的英特尔奔腾III处理器。

(a) A Play Station 4 **(b)** A Xbox One (c) 一台Wii U

图3.1.4：当前世代游戏机

手持式游戏机类似于游戏机，但将输入和输出系统以及计算机集成在一个单一包装中。

(a) A Nintendo DS

(b) A PS Vita

图3.1.5：一些手持式控制台

3.1.5 Embedded Computer

一个 *embedded computer* 是一种单板或单片计算机，具有限制-集成到更大硬件设备中的专用资源。

embedded computer

一个 *microcontroller* 是一种用于控制的嵌入式计算机其他硬件设备。微控制器安装在一块芯片上。微控通用计算机，但资源有限，因此只能执行一个或几个专门的任务。这些计算机用于单一目的，但它们仍然是通用的，因为可以编程它们执行不同的任务，取决于需求，而不改变底层硬件。

图3.1.6：嵌入在PC主板上的英特尔82815图形和内存控制器集线器。（来源：维基百科，作者：Qurren）

olTranslated Text: ol

另一种嵌入式计算机是 *system-on-chip*。一个 *system-on- chip* 是一个单芯片上的完整计算机。尽管微控制器安装在芯片上，但其目的是不同的：控制某些硬件。微控制器通常更简单且硬件资源更有限，因为它在运行时仅专注于一个目的，而片上系统是一个通用计算机，可以服务于多个目的。片上系统可以像普通台式计算机一样运行，能够加载操作系统并运行各种应用程序。片上系统通常出现在智能手机中，例如用于Ipad2和iPhone 4S的苹果A5 SoC，或用于许多Android手机的高通Snapdragon。

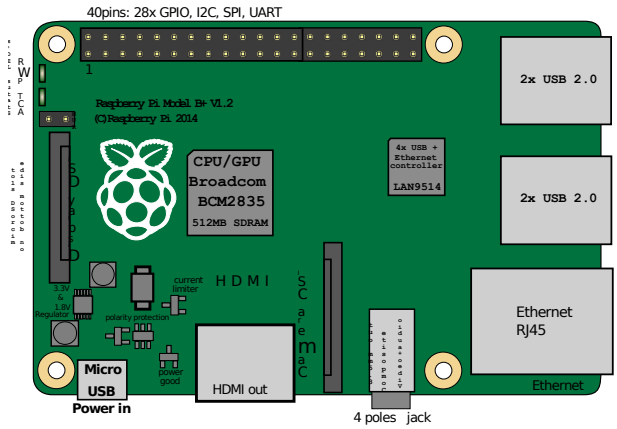
图3.1.7：PIC微控制器。（来源：Microchip）

microcontroller

Figure 3.1.8: Apple A5 SoC

无论是一个微控制器还是片上系统，都必须有一个环境，这些设备可以连接到其他设备。这个环境是一个称为 *PCB – **P**rinted **C**ircuit **B**oard.* 的电路板 – *printed cir- cuit board* 是一个包含线路和焊盘的物理板，用于在电气和电子组件之间实现电子流动。没有PCB，就无法将这些设备组合起来创建更大的设备。只要这些

设备隐藏在更大的设备内部，并为在更高层次上运行的更高层次目的的大设备做出贡献，它们是嵌入式设备。因此，为嵌入式设备编写程序被称为 *embedded programming*。嵌入式计算机用于自动控制的设备，包括电动工具、玩具、植入式医疗设备、办公机器、引擎控制系统、家用电器、遥控器和其他类型的嵌入式系统。



(a) 功能视图。SoC是Broadcom BCM2835。微控制器是以太网控制器LAN9514。（来源：维基百科，作者：Efa2）

(b) Physical View

图3.1.9：Raspberry PiB+ Rev 1.2，一款包含系统芯片和微控制器的单板计算机。

微控制器和系统级芯片之间的界限模糊。如果硬件不断进化变得更强大，那么微控制器就可以获得足够的资源，在它上面运行多个专用目的的最小操作系统。相比之下，系统级芯片足够强大，可以处理微控制器的任务。然而，将系统级芯片用作微控制器并不是一个明智的选择，因为价格会显著上升，但我们也会浪费硬件资源，因为为微控制器编写的软件需要的计算资源很少。

3.1.6 Field Gate Programmable Array

Field Programmable Gate Array (*FPGA*) 是一个硬件数组可配置门，使其在出厂后电路结构可编程1。回忆上一章，

Field Programmable Gate Array

这是为什么它被称为 ***Field*** 可编程阵列。它在应用现场是可更改的。

每个74HC00芯片都可以配置为一个门，通过组合多个74HC00芯片可以构建更复杂的设备。在类似

方式，每个FPGA设备包含成千上万的称为*logic blocks*的芯片，这比可以配置为执行布尔逻辑函数的74HC00芯片更复杂。这些逻辑块可以连接在一起以创建高级硬件功能。这个高级功能通常是一个需要高速处理的专用算法。

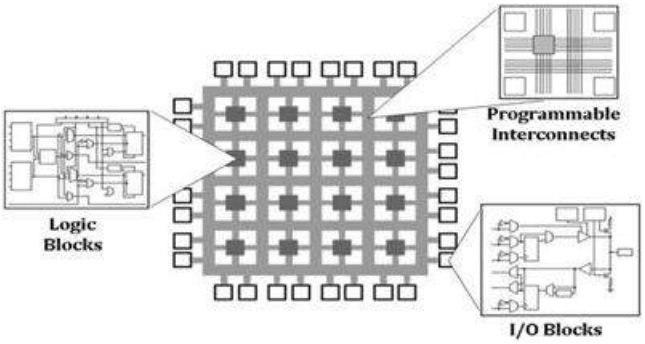


图3.1.10：FPGA架构（来源：National Instruments）

数字设备可以通过组合逻辑门来设计，无需考虑实际的电路元件，因为物理电路只是CMOS电路的多个实例。数字硬件，包括计算机中的各种组件，可以通过编写代码来设计，就像普通程序员一样，使用一种语言来描述门是如何连接在一起的。这种语言被称为*Hardware Description Language*。随后，硬件描述被编译成连接电子组件的描述，称为*netlist*，这是门连接的更详细描述。

FPGA与其他嵌入式计算机的区别在于，FPGA中的程序是在数字逻辑级别实现的，而像微控制器或片上系统设备这样的嵌入式计算机中的程序是在汇编代码级别实现的。为FPGA设备编写的算法是逻辑门中算法的描述，然后FPGA设备根据描述配置自身以运行该算法。为微控制器编写的算法是处理器可以理解和相应执行的汇编指令。

FPGA适用于在常规计算机上运行专用操作不合适且成本高昂的情况，例如实时医疗图像处理、巡航控制系统、电路原型设计、视频编码等。

编码/解码等。这些应用需要高速处理，而普通处理器无法实现，因为处理器在执行许多非专用指令时浪费了大量的时间——这些指令可能多达数千条或更多——以实现专用操作，因此在物理层面上需要更多的电路来完成相同的操作。FPGA设备没有这样的开销；相反，它直接在硬件中运行一个专用的操作。

3.1.7 Application-Specific Integrated Circuit

ASIC是一种为特定目的而设计的芯片，而不是通用用途。ASIC不包含可以重新配置以适应任何操作的通用逻辑块数组，就像FPGA一样；相反，ASIC中的每个逻辑块都是为电路本身制作和优化的。FPGA可以被认为是ASIC的原型阶段，而ASIC是电路生产的最终阶段。ASIC比FPGA更加专业化，因此可以实现更高的性能。然而，ASIC的制造成本非常高，一旦电路制作完成，如果发生设计错误，所有东西都将被丢弃，这与可以简单地重新编程的FPGA设备不同，因为它们具有通用门阵列。

3.2 计算机体系结构

上一节探讨了各种计算机类别。无论形状和大小，每台计算机都是为从高级到低级的建筑师设计的。

$$Computer\ Architecture = Instruction\ Set\ Architecture + Computer\ Organization + Hardware$$

在最高级别是指令集架构。

计算机组织位于中间层次。

在最低层是硬件。

3.2.1 Instruction Set Architecture

一个 *instruction set* 是微处理器能够理解和执行的基本命令和指令集。

一个 ***I****nstruction* ***S****et* ***A****rchitecture* 或 ***ISA*** 是实现指令集的环境设计。本质上，类似于高级语言解释器的运行时环境。该设计包括CPU的所有指令、寄存器、中断、内存模型（程序如何安排使用内存）、寻址模式、I/O等。CPU具有的功能（例如，更多的指令）越多，所需的电路就越多。

3.2.2 Computer organization

Computer organization 这是计算机设计的功能视图。

Computer organization

在这个视图中，计算机的硬件组件以带有输入和输出的框的形式呈现，它们相互连接并形成计算机的设计。两台计算机可能具有相同的ISA，但组织结构不同。例如，AMD和Intel处理器都实现了x86 ISA，但构成ISA环境的每个处理器的硬件组件并不相同。

计算机组织可能因制造商的设计而异，但它们都源自冯·诺依曼架构[2]：

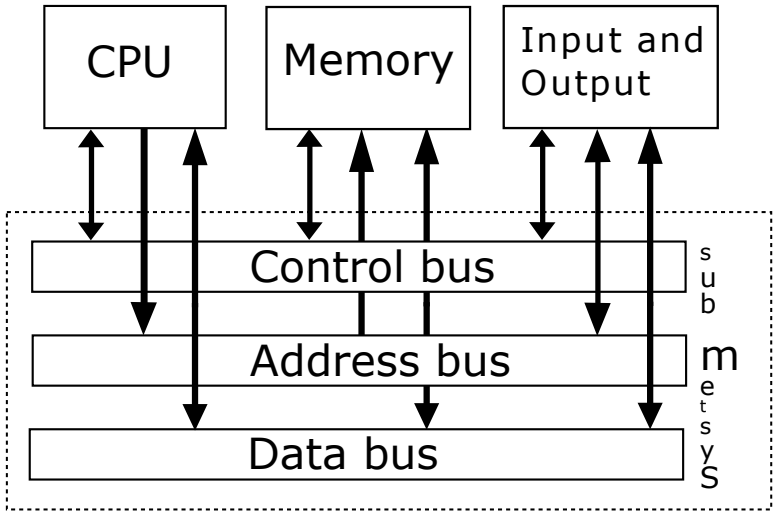


图3.2.1：冯·诺依曼架构

CPU 从主内存连续获取指令并执行。

2 *John von Neumann* 是一位数学家和物理学家，他发明了一种计算机架构。

Memory 存储程序代码和数据。

Bus 是用于在上述组件之间发送原始比特的电线。

I/O Devices 设备，即向计算机输入的设备，例如键盘、鼠标、传感器等，以及从计算机获取输出的设备，例如显示器接收从CPU发送的信息以显示它，LED根据CPU计算的模式打开/关闭等。

冯·诺依曼计算机通过将指令存储在主内存中运行，CPU反复将这些指令逐个取入其内部存储进行执行。数据通过CPU、内存和I/O设备之间的数据总线传输，而存储在设备中的位置则由CPU通过地址总线传输。这种架构完全实现了$fetch - decode - execute$周期。

早期的计算机只是冯·诺依曼架构的精确实现，CPU、内存和I/O设备通过相同的总线进行通信。如今，计算机拥有更多总线，每个总线都专门处理一种类型的流量。然而，在核心上，它们仍然是冯·诺依曼架构。为冯·诺依曼计算机编写操作系统，程序员需要能够理解和编写控制核心组件的代码：CPU、内存、I/O设备和总线。

CPU, 或 ***C****entral* ***P****rocessing* ***U****nit*，是任何计算机系统的核心和大脑。理解CPU对于从头编写操作系统至关重要：

- 要使用这些设备，程序员需要控制CPU以使用其他设备的编程接口。CPU是唯一的方式，因为CPU是程序员可以使用的唯一直接设备，也是唯一能理解程序员编写的代码的设备。

- 在CPU中，许多操作系统概念已经直接在硬件中实现，例如任务切换、分页。内核程序员需要了解如何使用硬件特性，以避免在软件中重复这些概念，从而浪费计算机资源。

- CPU 内置的 OS 功能提升了 OS 性能和开发者生产力，因为这些功能是实际硬件，最低级别，开发者可以自由实现这些功能。

▷ 为了有效地使用CPU，程序员需要理解CPU制造商提供的文档。例如，Intel® 64和IA-32架构软件开发手册。

▷ 了解一种CPU架构后，学习其他CPU架构会更容易。

一个CPU是ISA的实现，实际上是汇编语言的实现（并且根据CPU架构的不同，语言可能也会变化）。汇编语言是提供给软件工程师以控制CPU、从而控制计算机的一种接口。但是，如何仅通过访问CPU来控制每个计算机设备呢？简单的答案是，CPU可以通过这两个接口与其他设备通信，从而指挥它们：

寄存器是高速数据访问和通信的硬件组件。 ***Registers***

与其他硬件设备通信。寄存器允许软件通过写入设备的寄存器来直接控制硬件，或者在从设备的寄存器中读取时接收来自硬件设备的信息。

并非所有寄存器都用于与其他设备通信。在CPU中，大多数寄存器用作临时数据的快速存储。CPU可以与之通信的其他设备始终有一组寄存器用于与CPU接口。

端口是硬件设备中用于通信的专用寄存器。 ***Port***

与其他设备通信。当数据写入端口时，它会导致硬件设备根据写入端口的值执行某些操作。端口与寄存器的区别在于，端口不存储数据，而是将数据委托给其他电路。

这两个接口非常重要，因为它们是唯一用于用软件控制硬件的接口。编写设备驱动程序本质上是在学习每个寄存器的功能以及如何正确使用它们来控制设备。

Memory 这是一个存储信息的存储设备。内存由 ***Memory***

许多单元格。每个单元格是一个字节，具有其地址编号，因此CPU可以

使用这样的地址编号来访问内存中的确切位置。内存是存储和检索软件指令（以机器语言的形式）以供CPU执行的地方；内存还存储某些软件所需的数据。冯·诺依曼机器中的内存不区分哪些字节是数据，哪些字节是软件指令。这取决于软件来决定，如果数据字节被检索并作为指令执行，CPU仍然会执行，如果这些字节代表有效的指令，但会产生不理想的结果。对于CPU来说，没有代码和数据；它们只是它要操作的不同类型的数据：一个告诉它如何以特定方式做某事，另一个是它执行此类行动所需的必要材料。

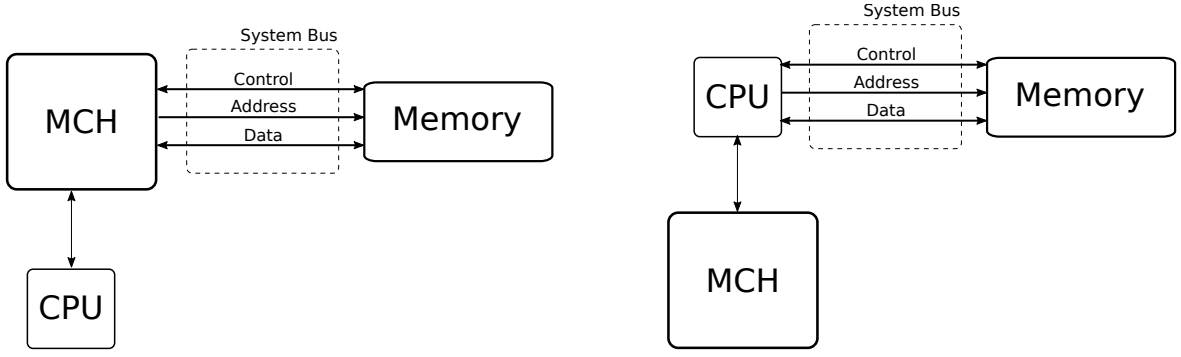
RAM由一个称为*memory controller*的设备控制。目前

，

大多数处理器都集成了此设备，因此CPU有一个专用内存总线连接处理器和RAM。在较旧的CPU3上，如何-

3 在2009年生产的CPU之前

永远，这个设备位于一个称为MCH或***M****emory****C****ontroller* ***H****ub*的芯片上。在这种情况下，CPU不是直接与RAM通信，而是与MCH芯片通信，然后这个芯片访问内存来读取或写入数据。第一个选项提供了更好的性能，因为在CPU和内存之间的通信中没有中间人。



(a) Old CPU

(b) Modern CPU

图3.2.2：CPU - 内存通信

在物理层面，RAM被实现为一个由单元格组成的网格，每个单元格包含一个晶体管和一个称为*capacitor*的电气设备，该设备存储

capacitor

短时间内的充电。晶体管控制对电容器的访问；当开启时，它允许从小型电容器中读取或写入电荷。电容器上的电荷逐渐耗散，

需要包含一个刷新电路，以定期从单元格中读取值，并在从外部电源放大后将其写回。

Bus 是一个在计算机组件之间传输数据的子系统 ***Bus***

或计算机之间。在物理上，总线只是连接所有组件的电气电线，每根电线传输一大块数据。电线的总数称为 *bus width*，并且依赖于

bus width

在CPU可以支持多少根线上。如果一个CPU一次只能接受16位，那么总线就有16根线从组件连接到CPU，这意味着CPU一次只能检索16位数据。

3.2.3 Hardware

硬件是计算机的一种特定实现。一系列处理器实现相同的指令集架构并使用几乎相同的组织结构，但在硬件实现上有所不同。例如，Core i7系列为台式计算机提供了一个更强大但消耗更多能量的模型，而笔记本电脑的另一个模型性能较低但能效更高。要为硬件设备编写软件，如果文档可用，我们很少需要了解硬件实现。计算机组织和特别是指令集架构对操作系统程序员更为相关。因此，下一章致力于深入研究x86指令集架构。

3.3 x86 架构

一个 *chipset* 是具有多个功能的芯片。从历史上看，芯片组实际上是一组单独的芯片，每个芯片负责一个功能，例如内存控制器、图形控制器、网络控制器、电源控制器等。随着硬件的发展，这些芯片集被整合到一个单独的芯片中，从而更加节省空间、能源和成本。在台式计算机中，各种硬件设备通过称为 *motherboard* 的 PCB 相互连接。每个 CPU 都需要一个能够容纳它的兼容主板。每个主板由其芯片组型号定义，

确定CPU可以控制的环境。这个环境通常包括

- 一个或多个CPU插槽
- 芯片组，由两块芯片组成，分别是北桥芯片和南桥芯片

  北桥芯片负责CPU、主内存和显卡之间的高性能通信。南桥芯片负责与I/O设备以及其他非性能敏感设备进行通信。

- 内存棒插槽
- 一个或更多显卡插槽。
- 通用插槽用于其他设备，例如网卡、声卡。
- 端口号用于I/O设备，例如键盘、鼠标、USB。

编写一个完整的操作系统，程序员需要了解如何编程这些设备。毕竟，操作系统会自动管理硬件，以便应用程序可以这样做。然而，在所有组件中，学习如何编程CPU是最重要的，因为它是任何计算机都存在的组件，无论计算机的类型是什么。因此，本书的主要焦点将是如何编程x86 CPU。即使仅关注这个设备，也可以编写一个相当好的最小操作系统。原因是并非所有计算机都包含像普通台式计算机那样的所有设备。例如，嵌入式计算机可能只有CPU和有限的内部内存，以及用于获取输入和产生输出的引脚；然而，为这样的设备编写了操作系统。

然而，学习如何编程x86 CPU是一项艰巨的任务，为此编写了3本主要手册：第1卷近500页，第2卷超过2000页，第3卷超过1000页。对于一个程序员来说，掌握x86 CPU编程的每一个方面是一项令人印象深刻的成就。

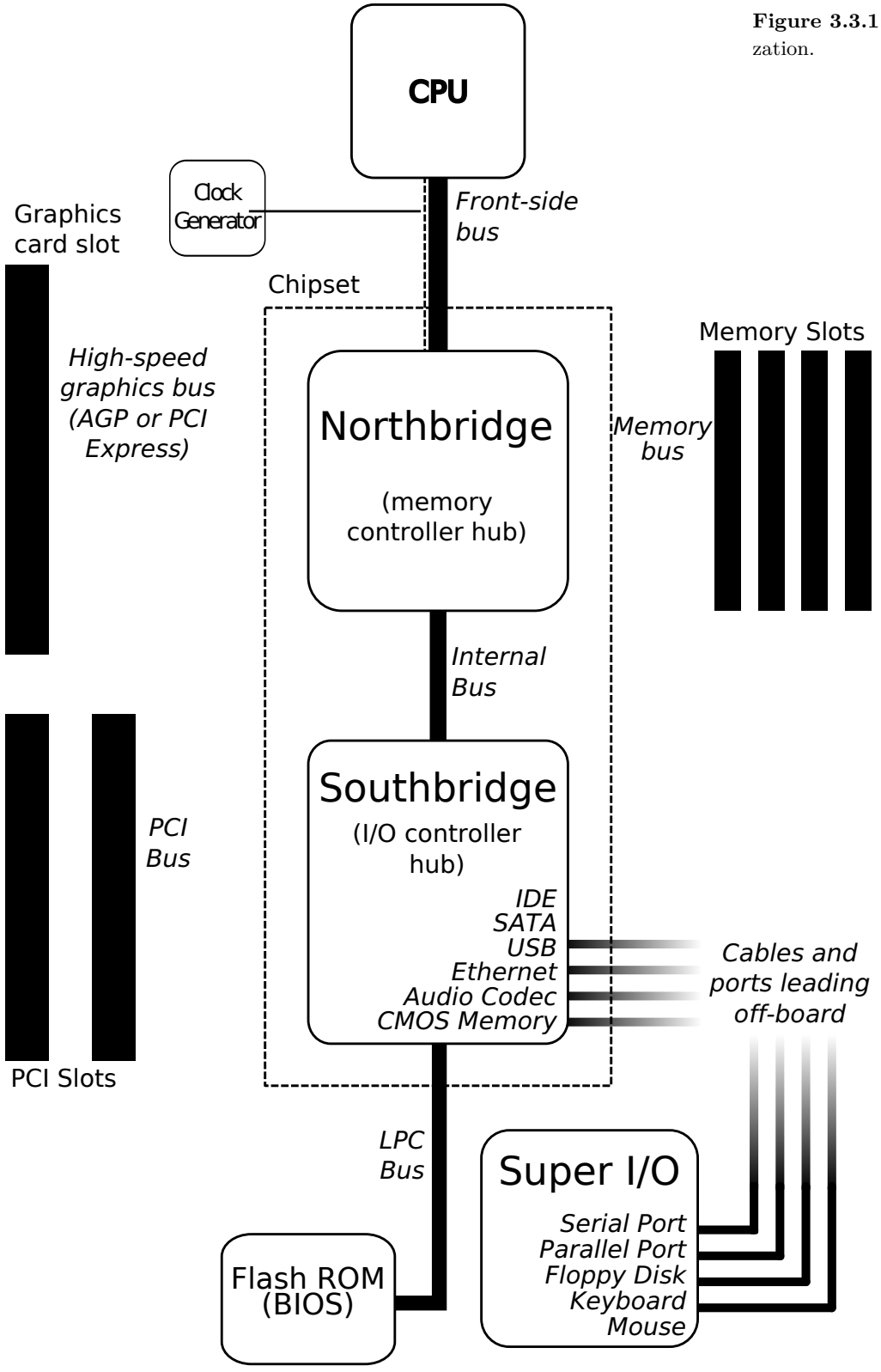


Figure 3.3.1: 主板组织-zation.

3.4 Intel Q35 Chipset

Q35是英特尔于2007年9月发布的芯片组。Q35被用作高级计算机组织的例子，因为稍后我们将使用QEMU来模拟Q35系统，这是QEMU可以模拟的最新英特尔系统。尽管于2007年发布，但Q35相对于当前硬件来说相对较新，这些知识仍然可以用于当前芯片组型号。使用Q35芯片组，模拟的CPU也相对较新，具有当前CPU中呈现的功能，因此我们可以使用英特尔最新的软件手册。

图3.3.1在 facing 页上是一个典型的当前主板组织，其中Q35具有相似的组织。

3.5 x86执行环境

一个 *execution environment* 是一个提供使代码可执行功能的环境。执行环境需要解决以下问题:

- 支持的操作? 数据传输、算术、控制、浮点等。
- 操作数存储在哪里? 寄存器、内存、堆栈、累加器
- 每个指令有多少个显式操作数? 0、1、2或3
- 如何指定操作数位置? 寄存器、立即数、间接等。
- 支持哪些操作数类型和大小? byte, int, float, double, string, vector等。
- 等等

本章剩余部分，请继续阅读至英特尔手册第1卷第3章, *"Basic Execution Environment"*。

4

x86 Assembly and C

在这一章中，我们将探讨汇编语言及其与C语言的关系。但我们为什么要这样做呢？难道不是更信任编译器，而且现在没有人再写汇编语言了吗？

不一定。当然，当前的编译器处于技术前沿，是值得信赖的，我们不需要用汇编语言编写代码，*most of the time*。编译器可以生成代码，但如前所述，高级语言是低级语言模式集合。它并不涵盖硬件平台提供的所有内容。因此，并非每个汇编指令都可以由编译器生成，所以我们仍然需要为这些情况编写汇编代码以访问特定于硬件的功能。由于特定于硬件的功能需要编写汇编代码，调试需要阅读它。我们可能花费更多的时间阅读而不是编写。与直接与硬件交互的低级代码一起工作，汇编代码是不可避免的。此外，了解编译器如何生成汇编代码可以提高程序员的效率。例如，如果一项工作或学校作业要求我们编写汇编代码，我们只需用C编写它，然后让`gcc`为我们完成编写汇编代码的艰苦工作。我们只需收集生成的汇编代码，根据需要进行修改，然后完成作业。

我们将广泛学习 `objdump`，以及如何使用英特尔文档来帮助理解 x86 汇编代码。

4.1 objdump

objdump 这是一个显示对象文件信息的程序。在以后调试手动链接的错误布局时，它将很有用。现在，我们使用 objdump 来检查高级源代码如何映射到汇编代码。目前，我们忽略输出，先学习如何使用命令。假设我们有一个名为 hello 的可执行二进制文件，它是由一个打印“Hello World”的 hello.c 编译而成的，使用 objdump 非常简单：

```
$ objdump -d hello
```

-d 选项仅显示可执行部分的组装内容。一个 *section* 是包含程序代码或数据的内存块。代码部分可由CPU执行，而数据部分不可执行。非可执行部分，如用于存储程序数据)的 .data 和 .bss (以及调试部分等，将不会显示。我们将在第5章第107页学习更多关于部分的内容。另一方面：

```
$ objdump -D hello
```

在 -D 选项显示所有段的汇编内容。如果 -D，则隐式假设 -d。objdump 主要用于检查汇编代码，因此 -d 是最有用的，因此默认设置为 -d。

输出超出了终端屏幕。为了便于阅读，请将所有输出发送到 less：

```
$ objdump -d hello | less
```

要混合源代码和汇编，必须使用 -g 选项编译二进制文件以包含源代码，然后添加 -S 选项：

```
$ objdump -S hello | less
```

默认情况下，objdump使用的语法是AT&T语法。要将它更改为熟悉的Intel语法：

```
$ objdump -M intel -D hello | less
```

当使用 -M 选项时，必须显式提供选项 -D 或 -d。接下来，我们将使用 objdump 来检查编译后的 C 数据和代码在机器代码中的表示。

最后，我们将编写一个32位内核，因此我们需要编译一个32位二进制文件，并在32位模式下检查它：

```
$ objdump -M i386,intel -D hello | less
```

-M i386 告诉objdump使用32位布局显示汇编内容。了解32位和64位之间的区别对于编写内核代码至关重要。我们将在编写内核时稍后探讨这个问题。

4.2 阅读输出

在输出开始时显示目标文件的格式：

```
hello: file format elf64-x86-64
```

在行之后是一系列拆解部分：

```
Disassembly of section .interp:
...
Disassembly of section .note.ABI-tag:
...
Disassembly of section .note.gnu.build-id:
...
...
etc
```

最后，每个拆解部分都显示其实际内容——即一系列汇编指令——以下格式：

```
4004d6:       55                          push   rbp
```

▷ 第一列是汇编指令的地址。在上面的例子中，地址是 0x4004d6。

▷ 第二列是原始十六进制值中的汇编指令。在上面的例子中，值是 0x55。

▷ 第三列是汇编指令。根据部分的不同，汇编指令可能是有意义的，也可能没有意义。例如，如果汇编指令位于 .text 部分，则汇编指令是实际程序代码。另一方面，如果汇编指令显示在 .data 部分，则我们可以安全地忽略显示的指令。原因是 objdump 不知道哪些十六进制值是代码，哪些是数据，因此它盲目地将每个十六进制值翻译成汇编指令。在上面的例子中，汇编指令是 push %rbp。

▷ 可选的第四列是注释 - 当有地址引用时出现 - 以告知地址的来源。例如，蓝色的注释：

```
lea r12,[rip+0x2008ee] # 600e10 <__frame_dummy_init_array_entry>
```

要通知，所引用的地址从 [rip+0x2008ee] 是 0x600e10，其中变量 __frame_dummy_init_array_entry 存储于此。

在一个拆解的部分，它也可能包含 *labels*。标签是赋予汇编指令的名称。标签向人类读者表明汇编块的目的，使其更容易理解。例如，.text 部分携带许多这样的标签，以表示程序中代码的起始位置；下面的 .text 部分包含两个函数：_start 和 deregister_tm_clones。_start 函数从地址 4003e0 开始，在函数名称左侧进行注释。在 _start 标签下方也是地址 4003e0 的指令。这整个意味着标签只是一个内存地址的名称。函数 deregister_tm_clones 也与该部分中每个函数相同的格式。

```
00000000004003e0 <_start>:
  4003e0:       31 ed                   xor    ebp,ebp
  4003e2:       49 89 d1                mov    r9,rdx
  4003e5:       5e                      pop    rsi
...more assembly code....
0000000000400410 <deregister_tm_clones>:
  400410:       b8 3f 10 60 00          mov    eax,0x60103f
  400415:       55                      push   rbp
  400416:       48 2d 38 10 60 00       sub    rax,0x601038
...more assembly code....
```

4.3 英特尔手册

了解和使用汇编语言的最佳方式是精确理解底层计算机架构以及每条机器指令的作用。为此，最可靠的来源是参考厂商提供的文档。毕竟，硬件厂商是制造他们机器的人。要理解英特尔指令集，我们需要文档“*Intel 64 and IA-32 architectures software developer's manual combined volumes 2A, 2B, 2C, and 2D: Instruction set reference, A-Z*”。该文档可在此处获取：`https://software.intel.com/en-us/articles/intel-sdm`。

▷ 第一章提供了关于手册的简要信息，以及书中使用的注释符号。

▷ 第二章深入解释了汇编指令的解剖结构，我们将在下一节中探讨。

▷ 第三章 - 5 提供了 x86_64 架构每条指令的详细信息。

▷ 第6章提供了关于安全模式扩展的信息。我们不需要使用这一章。

第一卷“*Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*”描述了英特尔处理器的基本架构和编程环境。在书中，第

5 提供了所有英特尔指令的总结，通过将指令分类列出。我们只需要学习列出的通用指令 *chapter 5.1* 以构建我们的操作系统。*Chapter 7* 描述了每个类别的目的。逐渐地，我们将学习所有这些指令。

练习4.3.1。阅读第2卷第1.3节，排除第1.3.5节和第1.3.7节。

4.4 尝试汇编代码

后续部分检查汇编指令的解剖结构。要完全理解，有必要编写代码并查看以十六进制数字形式实际显示的代码。为此，我们使用 `nasm` 汇编器编写几行汇编代码并查看生成的代码。

示例4.4.1. 假设我们想查看这条指令生成的机器代码：

```
jmp eax
```

然后，我们使用一个编辑器，例如Emacs，然后创建一个新文件，编写代码并将其保存到文件中，例如`test.asm`。然后，在终端中运行以下命令：

```
$ nasm -f bin test.asm -o test
```

`-f` 选项指定最终输出文件的文件格式，例如 ELF。但在此情况下，格式为 `bin`，这意味着此文件只是一个没有额外信息的平面二进制输出。也就是说，编写的汇编代码直接转换为机器代码，没有像 ELF 这样的文件格式元数据的开销。实际上，编译后，我们可以使用以下命令检查输出：

```
$ hd test
```

hd (简称为hexdump)的程序可以以十六进制格式显示文件内容。并且得到以下输出:

尽管其名称简称为十六进制转储，hd 可以以不同的基数显示，例如二进制，而不仅仅是十六进制。

```
00000000  66 ff e0                                       |f..|
00000003
```

文件仅包含3个字节: 66 ff e0, 相当于指令jmp eax。

示例4.4.2。如果我们使用elf作为文件格式:

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

学习和理解汇编将更具挑战性
指令包含所有添加的噪声1:

1 hd.的输出

```
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 03 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  40 00 00 00 00 00 00 00  34 00 00 00 00 00 28 00  |@.......4.....(.|
00000030  05 00 02 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000060  00 00 00 00 00 00 00 00  01 00 00 00 01 00 00 00  |................|
00000070  06 00 00 00 00 00 00 00  10 01 00 00 02 00 00 00  |................|
00000080  00 00 00 00 00 00 00 00  10 00 00 00 00 00 00 00  |................|
00000090  07 00 00 00 03 00 00 00  00 00 00 00 00 00 00 00  |................|
000000a0  20 01 00 00 21 00 00 00  00 00 00 00 00 00 00 00  | ...!...........|
000000b0  01 00 00 00 00 00 00 00  11 00 00 00 02 00 00 00  |................|
000000c0  00 00 00 00 00 00 00 00  50 01 00 00 30 00 00 00  |........P...0...|
000000d0  04 00 00 00 03 00 00 00  04 00 00 00 10 00 00 00  |................|
000000e0  19 00 00 00 03 00 00 00  00 00 00 00 00 00 00 00  |................|
000000f0  80 01 00 00 0d 00 00 00  00 00 00 00 00 00 00 00  |................|
00000100  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

```
00000110  ff e0 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000120  00 2e 74 65 78 74 00 2e  73 68 73 74 72 74 61 62  |..text..shstrtab|
00000130  00 2e 73 79 6d 74 61 62  00 2e 73 74 72 74 61 62  |..symtab..strtab|
00000140  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000160  01 00 00 00 00 00 00 00  00 00 00 00 04 00 f1 ff  |................|
00000170  00 00 00 00 00 00 00 00  00 00 00 00 03 00 01 00  |................|
00000180  00 74 65 73 74 2e 61 73  6d 00 00 00 00 00 00 00  |.disp8-5.asm....|
00000190
```

因此，在这种情况下，最好只使用平面二进制格式，逐条实验指令。

使用如此简单的流程，我们准备好调查每个汇编指令的结构。

注意：使用二进制格式默认将 nasm 放入16位模式。要生成32位代码，我们必须在 nasm 源文件的开头添加此行：

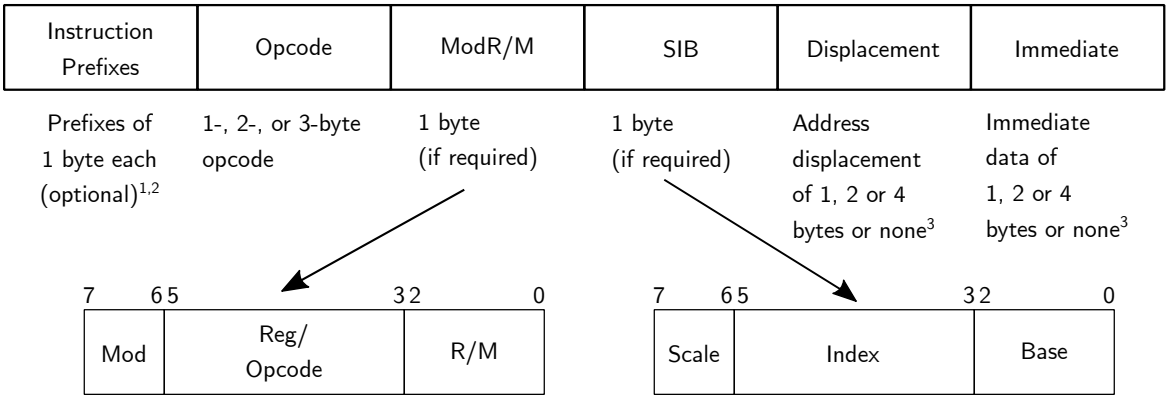
```
bits 32
```

4.5 汇编指令的解剖

第二章的指令参考手册提供了对指令格式的深入了解。但是，信息量太大，可能会让初学者感到不知所措。本节在阅读手册中的实际章节之前，提供了一个更简单的指令。

回忆一下，汇编指令只是一个固定大小的位序列。指令的长度是可变的，并且取决于指令的复杂程度。每个指令共有的东西是上面图中描述的通用格式，它将指令的位划分为更小的部分，这些部分编码了不同类型的信息。这些部分是：

Instruction Prefixes 出现于指令的开头。前缀是可选的。程序员可以选择使用前缀或不使用，因为在实践中，所谓的“前缀”只是另一个汇编指令，以



1. The REX prefix is optional, but if used must be immediately before the opcode; see Section 2.2.1, "REX Prefixes" in the manual for additional information.
2. For VEX encoding information, see Section 2.3, "Intel® Advanced Vector Extensions (Intel® AVX)" in the manual.
3. Some rare instructions can take an 8B immediate or 8B displacement.

图4.5.1：Intel 64和IA-32架构指令格式

在另一个汇编指令之前插入，以便该前缀适用。包含2或3字节操作码的指令默认包含前缀。

Opcode 这是一个唯一数字，用于标识一条指令。每个操作码都有一个人类可读的助记符名称，例如，指令 `add` 的一个操作码是 `04`。当 CPU 在其指令缓存中看到数字 `04` 时，它看到指令 `add` 并相应地执行。操作码可以是 1、2 或 3 个字节长，并在需要时在 `ModR/M` 字节中包含一个额外的 3 位字段。

示例4.5.1。本指令：

```
jmp [0x1234]
```

生成机器代码：

```
ff 26 34 12
```

第一个字节，`0xff` 是操作码，它对 `jmp` 指令是唯一的。

ModR/M 指定指令的操作数。操作数可以是

寄存器，一个内存位置或一个立即值。指令的这一部分由3个更小的部分组成：

▷ *mod* 字段或 *modifier* 字段与 r/m 字段结合，总共5位信息编码32个可能值：8个寄存器和24种寻址模式。▷ *reg/opcode* 字段编码寄存器操作数，或扩展 Opcode 字段3位。▷ *r/m* 字段编码寄存器操作数，或可与 mod 字段结合编码寻址模式。

表格4.5.1和4.5.2列出了所有可能的256个ModR/M字节值，以及每个值如何映射到寻址模式和寄存器，在16位和32位模式下。

r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP[1]	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
(In decimal) /digit (Opcode)			0	1	2	3	4	5	6	7
(In binary) REG =			000	001	010	011	100	101	110	111
Effective Address	**Mod**	**R/M**	**Values of ModR/M Byte (In Hexadecimal)**							
[BX + SI]	00	000	00	08	10	18	20	28	30	38
[BX + DI]		001	01	09	11	19	21	29	31	39
[BP + SI]		010	02	0A	12	1A	22	2A	32	3A
[BP + DI]		011	03	0B	13	1B	23	2B	33	3B
[SI]		100	04	0C	14	1C	24	2C	34	3C
[DI]		101	05	0D	15	1D	25	2D	35	3D
disp16[2]		110	06	0E	16	1E	26	2E	36	3E
[BX]		111	07	0F	17	1F	27	2F	37	3F
[BX + SI] + disp8[3]	01	000	40	48	50	58	60	68	70	78
[BX + DI] + disp8		001	41	49	51	59	61	69	71	79
[BP + SI] + disp8		010	42	4A	52	5A	62	6A	72	7A
[BP + DI] + disp8		011	43	4B	53	5B	63	6B	73	7B
[SI] + disp8		100	44	4C	54	5C	64	6C	74	7C
[DI] + disp8		101	45	4D	55	5D	65	6D	75	7D
[BP] + disp8		110	46	4E	56	5E	66	6E	76	7E
[BX] + disp8		111	47	4F	57	5F	67	6F	77	7F
[BX + SI] + disp16	10	000	80	88	90	98	A0	A8	B0	B8
[BX + DI] + disp16		001	81	89	91	99	A1	A9	B1	B9
[BP + SI] + disp16		010	82	8A	92	9A	A2	AA	B2	BA
[BP + DI] + disp16		011	83	8B	93	9B	A3	AB	B3	BB
[SI] + disp16		100	84	8C	94	9C	A4	AC	B4	BC
[DI] + disp16		101	85	8D	95	9D	A5	AD	B5	BD
[BP] + disp16		110	86	8E	96	9E	A6	AE	B6	BE
[BX] + disp16		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM1/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AHMM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

1. 默认段寄存器为SS，用于包含BP索引的有效地址，DS用于其他有效地址。

2. disp16命名法表示一个16位位移，该位移跟在ModR/M字节之后，并将其加到索引上。

3. disp8命名法表示一个8位位移，该位移跟在ModR/M字节之后，然后进行符号扩展并加到索引上。

Table 4.5.1: 16-Bit Addressing Forms with the ModR/M Byte

r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
(In decimal) /digit (Opcode)			0	1	2	3	4	5	6	7
(In binary) REG =			000	001	010	011	100	101	110	111
Effective Address	**Mod**	**R/M**	**Values of ModR/M Byte (In Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX] + disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX] + disp8		001	41	49	51	59	61	69	71	79
[EDX] + disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX] + disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--] + disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP] + disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI] + disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI] + disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX] + disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX] + disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX] + disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX] + disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--] + disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP] + disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI] + disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI] + disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

1. The [--][--] 命名法表示 SIB 后跟 ModR/M 字节。

2. disp32命名法表示一个32位位移，该位移跟在ModR/M字节之后（如果存在SIB字节，则跟在SIB字节之后），并将其加到索引上。

3. disp8命名法表示一个8位位移，该位移跟在ModR/M字节之后（如果存在SIB字节，则跟在SIB字节之后），然后进行符号扩展并加到索引上。

Table 4.5.2: 32-Bit Addressing Forms with the ModR/M Byte

如何阅读表格：

在指令中，操作码旁边是一个 ModR/M 字节。然后，查找此表中的字节值以获取行和列中相应的操作数。

示例4.5.2。一条指令使用此寻址方式 de:

```
jmp [0x1234]
```

然后，机器码是：

```
ff 26 34 12
```

0xff 是操作码。旁边，0x26是ModR/M字节。在16位表中查找，第一个操作数在行中，相当于一个disp16，

这意味着一个16位偏移。由于指令没有第二个操作数，该列可以忽略。

记住，使用 bin 格式默认生成 16 位代码

示例4.5.3。一条指令使用这种寻址模式：

```
add eax, ecx
```

然后，机器码是：

```
66 01 c8
```

这个指令的有趣之处在于 0x66 不是操作码。0x01 是操作码。那么，0x66 又是什么呢？回想一下，对于每条汇编指令，都会有一个可选的指令前缀，这就是 0x66。根据英特尔手册，第1卷：

> 操作数大小覆盖前缀允许程序在16位和32位操作数大小之间切换。任一大小都可以是默认值；使用前缀选择非默认大小。

如果CPU切换到32位模式，当它执行带有0x66前缀的指令时，指令操作数仅限于16位宽度。

另一方面，如果CPU处于16位环境中，因此，32位被视为非标准，因此，指令操作数临时升级到32位宽度，而未使用前缀的指令使用16位操作数。

旁边，c8是ModR/M字节。在16位表中查找c8值，行表示第一个操作数是ax，列表示第二个

操作数是 cx；该列不能忽略，因为第二个操作数在指令中。

记住，使用二进制格式默认生成16位代码

为什么第一个操作数在行中，而第二个在列中？让我们以示例值 c8 为例，将 ModR/M 字节分解为位：

mod		**reg/opcode**			**r/m**		
1	1	0	0	1	0	0	0

mod字段将寻址模式分为4个不同的类别。进一步与r/m字段结合，可以从24行中选择一种精确的寻址模式。如果指令只需要一个操作数，则可以忽略该列。然后reg/opcode字段最终提供额外的寄存器或不同的变体，如果指令需要的话。

SIB 是 ***S****cale-****I****ndex-****B****ase* 字节。此字节将计算内存位置的编码方式转换为一个数组的元素。SIB 是基于此公式计算有效地址的名称：

$$\texttt{Effective address} = \texttt{scale} * \texttt{index} + \texttt{base}$$

- ▷ Index 是一个数组的偏移量。
- ▷ Scale 是一个 Index 的因子。Scale 是值 1、2、4 或 8 之一；任何其他值都是无效的。要与其他值 2、4 或 8 进行缩放，必须将缩放因子设置为 1，并且必须手动计算偏移量。例如，如果我们想获取数组中 $\texttt{n}^{\texttt{th}}$ 元素的地址，并且每个元素长度为 12 字节。因为每个元素长度为 12 字节而不是 1、2、4 或 8，Scale 被设置为 1，并且编译器需要计算偏移量：

$$\texttt{Effective address} = 1 * (12 * \texttt{n}) + \texttt{base}$$

为什么我们还要使用SIB，当我们可以手动计算偏移量呢？答案是，在上面的场景中，必须执行额外的 `mul` 指令来获取偏移量，而 `mul` 指令消耗超过1个字节，而SIB只消耗1个字节。更重要的是，如果元素在循环中被反复访问多次，例如数百万次，那么额外的 `mul` 指令可能会损害性能，因为CPU必须花费时间执行数百万个这些额外的 `mul` 指令。

2、4和8不是随机选择的。它们对应于16位（或2字节）、32位（或4字节）和64位（或8字节）的数字，这些数字常用于密集的数值计算。

▷ `Base` 是起始地址。

以下是列出所有256个 `SIB` 字节值的表格，查找规则类似于 `ModR/M` 表格：

示例4.5.4。此指令：

```
jmp [eax*2 + ebx]
```

生成以下代码：

```
00000000  67 ff 24 43
```

首先，第一个字节`0x67`是*not*一个操作码但不是*prefix*。这个数字是一个地址大小覆盖前缀的预定义前缀。在预定义前缀之后，是操作码`0xff`和字节`ModR/M` `0x24`。从`ModR/M`的值可以推断出存在一个随后的`SIB`字节。`SIB`字节是`0x43`。

在 `SIB` 表中查找，行表示 `eax` 已放大2倍，列表示要添加的基数位于 `ebx`。

Displacement 是从基线起始的偏移 例如。

示例4.5.5。此指令：

```
jmp [0x1234]
```

r32(/r) (In decimal) /digit (Opcode) (In binary) REG =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	EBP 5 101	ESI 6 110	EDI 7 111
Effective Address	**SS**	**R/M**	**Values of SIB Byte (In Hexadecimal)**							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	99	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

1. [*] 命名法表示 MOD 为 00B 时的 disp32，没有基址。否则，[*] 表示 disp8 或 disp32 + [EBP] 。这提供了以下地址模式:

MOD bits	Effective Address
00	[scaled index] + disp32
01	[scaled index] + disp8 + [EBP]
10	[scaled index] + disp32 + [EBP]

Table 4.5.3: 32-Bit Addressing Forms with the SIB Byte

generates machine code is:

```
ff 26 34 12
```

0x1234, ĭn raw machine code ġénérátéd as 34 12, s the dis- placement an d stands right next to 0x26, s the ModR/M byte.

示例4.5.6。此指令：

```
jmp [eax * 4 + 0x1234]
```

生成机器代码：

```
67 ff 24 85 34 12 00 00
```

▷ 0x67 是一个地址大小覆盖前缀。它的意思是，如果一条指令运行默认的地址大小，例如16位，使用前缀可以使指令使用非默认的地址大小，例如32位或64位。由于二进制应该是16位的，0x67将指令更改为32位模式。

▷ 0xff 是操作码。

▷ 0x24 是 ModR/M 字节。根据表4.5.2，该值表明随后是一个SIB字节。

▷ 0x85 是 SIB 字节。根据表4.5.3，字节 0x85 可以分解为位，如下所示：

SS		**R/M**			**REG**		
1	0	0	0	0	1	0	1

上述值通过SS、R/M列获得，最后是REG的8列分别。将总比特数组合成值10000101，十六进制值为0x85。默认情况下，如果位移后的寄存器未指定，则将其设置为EBP寄存器，因此6th列（比特模式101）始终被选中。如果示例使用另一个寄存器：

示例4.5.7。例如：

```
jmp [eax * 4 + eax + esi]
```

SIB 字节变为 0x86 而不是 ，位于第 7th 列。请再次与表 4.5.3 进行验证。

▷ `34 12 00 00` 位移。如所见，位移大小为4字节，相当于32位，因为地址大小覆盖前缀。

Immediate 当一条指令接受一个固定值，例如 `0x1234`，作为操作数时，此可选字段包含该值。请注意，此字段与位移不同：该值不一定用作偏移量，而是任何任意值。

示例4.5.8。此指令：

```
mov eax, 0x1234
```

生成代码：

```
66 b8 34 12 00 00
```

▷ `0x66` 是操作数大小的覆盖前缀。类似于地址大小覆盖前缀，此前缀使操作数大小可以非默认。▷ `0xb8`是`mov`指令的指令码之一。
▷ `0x1234`是要存储在寄存器`eax`中的值。它只是直接存储到寄存器的一个值，没有其他。另一方面，位移值是某些地址计算的一个偏移量。

练习4.5.1。阅读第2卷第2.1节以获取更多详细信息。

练习4.5.2。快速浏览第1卷第5.1节。阅读第1卷第7章。如果有不理解的专业术语，例如分割，不要担心，这些术语将在后面的章节中解释或忽略。

4.6 详细理解指令

在指令参考手册（第2卷）中，从第3章开始，每个x86指令都进行了详细说明。当需要指令的确切行为时，我们总是首先查阅此文档。然而，在使用文档之前，我们必须首先了解写作规范。每个指令都有以下共同的结构来组织信息：

Opcode table 列出所有汇编指令的可能操作码。

每个表包含以下字段，并且可以有一行或多行:

Opcode	Instruction	Op/En	64/32-bit Mode	CPUID Feature flag	Description

Opcode shows a unique hexadecimal number assigned to an instruction. There can be more than one opcode for an instruction, each encodes a variant of the instruction. For example, one variant requires one operand, but another requires two. In this column, there can be other notations aside from hexadecimal numbers. For example, `/r` indicates that the `ModR/M` byte of the instruction contains a `reg` operand and an `r/m` operand. The detail listing is in section *3.1.1.1* and *3.1.1.2* in the Intel's manual, volume 2.

Instruction gives the syntax of the assembly instruction that a programmer can use for writing code. Aside from the mnemonic representation of the opcode, e.g. `jmp`, other symbols represent operands with specific properties in the instruction. For example, `rel8` represents a relative address from 128 bytes before the end of the instruction to 127 bytes after the end of instruction; similarly `rel16/rel32` also represents relative addresses, but with the operand size of 16/32-bit instead of 8-bit like `rel8`. For a detailed listing, please refer to section `3.1.1.3` of volume 2.

Op/En is short for ***Op****erand/****En****coding*. An operand encoding specifies how a `ModR/M` byte encodes the operands that an instruction requires. If a variant of an instruction requires operands, then an additional table named *"Instruction Operand Encoding"* is added for explaining the operand encoding, with the following structure:

Op/En	Operand 1	Operand 2	Operand 3	Operand 4

大多数指令需要一到两个操作数。我们使用这些指令为我们的操作系统服务，并跳过需要三个或四个操作数的指令。操作数可以是可读的、可写的或两者都是。符号 `(r)` 表示可读操作数，而 `(w)` 表示可写操作数。例如，当 `Operand 1` 字段包含 `ModRM:r/m` 时

(r), 这意味着第一个操作数编码在 ModR/M 字节的 r/m 字段中，并且只有 readable。

64/32-bit mode 指示操作码序列是否在64位模式下受支持，以及可能在32位模式下受支持。

CPUID Feature Flag 指示特定CPU功能必须可用才能启用指令。如果CPU不支持所需功能，则指令无效。

在Linux中，命令:

```
cat /proc/cpuinfo
```

列出可用的CPU及其功能的信息在 flags 字段中。

Compat/Leg Mode 许多指令没有此字段，而是用 Compat/Leg Mode 代替，代表 *Compati*

*or Legacy Mode*此模式使指令的64位变体能够在16位或32位模式下正常运行。

表4.6.1: Compat/Leg Mode 中的符号

Notation	Description
Valid	Supported
I	Not supported
N.E.	The 64-bit opcode cannot be encoded as it overlaps with existing 32-bit opcode.

Description 简要解释当前行中指令的变体。

Description 指定指令的目的以及指令如何详细工作。

Operation 伪代码实现指令。如果描述模糊，本节是理解汇编指令的次优来源。语法在第二卷第 *3.1.1.9* 节中描述。

Flags affected 列出 EFLAGS 寄存器中系统标志的可能更改。

Exceptions 列出当指令无法正确运行时可能出现的错误。本节对操作系统调试很有价值。异常分为以下几类:

▷ 保护模式异常

▷ 真实地址模式异常 ▷ 虚拟8086
模式异常 ▷ 浮点异常 ▷ SIMD浮点
异常 ▷ 兼容模式异常

▷ 64-bit Mode Exception

对于我们的操作系统，我们只使用 *Protected Mode Exceptions* 和 *Real-Address Mode Exceptions*。详细信息见第 *3.1.1.13* 和 *3.1.1.14* 节，第 2 卷。

4.7 示例：jmp 指令

让 查看我们的老牌 jmp 指令。首先，操作码 表格：

Opcode	Instruction	Op/ En	64-bit Mode	Compat/Leg Mode	Description
EB cb	JMP rel8	D	Valid	Valid	Jump short, RIP = RIP + 8-bit displacement sign extended to 64-bits
E9 cw	JMP rel16	D	N.S.	Valid	Jump near, relative, displacement relative to next instruction. Not supported in 64-bit mode.
E9 cd	JMP rel32	D	Valid	Valid	Jump near, relative, RIP = RIP + 32-bit displacement sign extended to 64-bits
FF /4	JMP r/m16	M	N.S.	Valid	Jump near, absolute indirect, address = zero- extended r/m16. Not supported in 64-bit mode
FF /4	JMP r/m32	M	N.S.	Valid	Jump near, absolute indirect, address given in r/m32. Not supported in 64-bit mode
FF /4	JMP r/m64	M	Valid	N.E	Jump near, absolute indirect, RIP = 64-Bit offset from register or memory
EA cd	JMP ptr16:16	D	Inv.	Valid	Jump far, absolute, address given in operand
EA cp	JMP ptr16:32	D	Inv.	Valid	Jump far, absolute, address given in operand
FF /5	JMP m16:16	D	Valid	Valid	Jump far, absolute indirect, address given in m16:16
FF /5	JMP m16:32	D	Valid	Valid	Jump far, absolute indirect, address given in m16:32
REX.W + FF /5	JMP m16:64	D	Valid	N.E.	Jump far, absolute indirect, address given in m16:64

表4.7.1：jmp操作码表

每行列出 jmp 指令的一个变体。第一列包含操作码 EB cb，以及等效的符号形式 jmp rel8。在这里，rel8 表示从指令末尾开始的 128 字节偏移量。指令的末尾是指令最后一个字节之后的下一个字节。为了更具体地说明，考虑以下汇编代码：

```
main:
  jmp main
  jmp main2
  jmp main
```

```
main2:
  jmp 0x1234
```

生成机器代码：

	main ↓					main2 ↓				
Address	00	01	02	03	04	05	06	07	08	09
Opcode	eb	fe	eb	02	eb	fa	e9	2b	12	00

表4.7.2：每个操作码的内存地址

第一条 `jmp main` 指令生成到 `eb fe` 并占用地址 00 和 01；第一条 `jmp main` 的结束地址在地址 02，超过第一条 `jmp main` 的最后一个字节，该字节位于地址 01。值 `fe` 等同于 -2，因为 `eb` 指令码只使用一个字节（8位）进行相对寻址。偏移量是 -2，第一条 `jmp main` 的结束地址是 02，将它们相加我们得到 00，这是跳转的目标地址。

同样，`jmp main2`指令生成到`eb 02`，这意味着偏移量是+2；`jmp main2`的结束地址在04，加上偏移量我们得到目标地址是06，这是由标签`main2`标记的起始指令。

相同的规则可以应用于 `rel16` 和 `rel32` 编码。在示例代码中，`jmp 0x1234` 使用 `rel16` (，这意味着 2 字节偏移) 并生成到 `e9 2b 12`。如表 4.7.1 所示，`e9` 指令码取一个 `cw` 操作数，它是一个 2 字节偏移（章节 *3.1.1.1*，卷 2）。注意这里的一个奇怪问题：偏移值是 `2b 12`，而它应该是 `34 12`。没有错误。记住，`rel8/rel16/rel32` 是一个 *offset*，不是一个 *address*。偏移是一个从某一点的距离。由于没有给出标签，而是给出一个数字，因此偏移是从程序开始计算的。在这种情况下，程序开始是地址 00，结束地址 `jmp 0x1234` 是地址 $_{09}2$，因此偏移是按 `0x1234 - 0x9` 计算的

2 表示消耗了 9 个字节，从地址 0 开始。

`= 0x122b`那解决了这个谜团！

`jmp` 指令与操作码 `FF /4` 允许跳转到存储在通用寄存器或内存位置的 *near, absolute* 地址；或者简而言之，如描述中所述，*absolute indirect*。符号 `/4` 是表 4.5.13 中数字 4 所在的列。例如：

3 列包含以下字段：
```
AH
SP
ESP
M45
XMM4
4
100
```

```
jmp [0x1234]
```

生成为:

```
ff 26 34 12
```

由于这是16位代码，我们使用表4.5.1。查表得知，ModR/M值26表示disp16，这意味着从当前索引4的起始处偏移16位，这是存储在DS寄存器中的基址。

在这种情况下，jmp [0x1234] 隐含地理解为 jmp [ds:0x1234]，这意味着目标地址距离数据段起始处 0x1234 字节。

4 查看表格下的注释。

jmp指令，操作码为FF /5，允许跳转到存储在*memory location* (中的*far, ab- solute*地址，而不是/4，即存储在寄存器)中；简而言之，*a far pointer*。要生成此类指令，需要使用关键字far来告知nasm我们正在使用远指针:

```
jmp far [eax]
```

生成为:

```
67 ff 28
```

由于 28 是表 4.5.25 第 5 列的值，它指的是

到 [eax]，我们成功生成了一条远跳转指令。CPU 执行该指令后，程序计数器 eip 和代码段寄存器 cs 被设置为 eax 所指向的内存地址，CPU 从 cs 和 eip 中的新地址开始获取代码。为了更具体地说明，这里有一个例子:

5 记住前缀 67 表示指令用作32位。只有当汇编器生成代码时，默认环境假设为16位时，才会添加前缀。

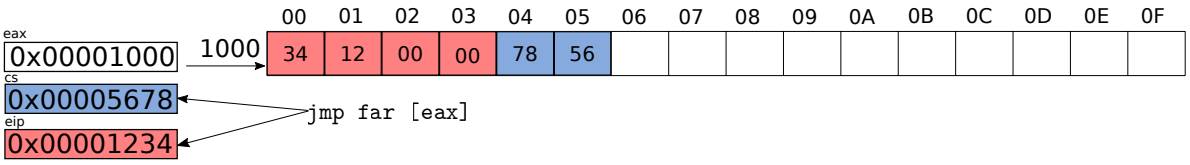


图4.7.1: 远 jmp 示例，目标内存存储在地址 0x1000，该地址存储在 eax 以进行解引用。CPU 执行指令后，代码段寄存器 cs 和指令指针 eip

远地址对于16位段和32位地址总共消耗6个字节的容量，该地址编码为表4.7.1中的m16:32。

从上图可以看出，蓝色部分是段地址，将其值0x5678加载到cs寄存器中；红色部分是该段内的内存地址，将其值0x1234加载到eip寄存器中，并从这里开始执行。

最后，具有 EA 操作码的 jmp 指令跳转到直接绝对地址。例如，该指令：

```
jmp 0x5678:0x1234
```

生成为：

```
ea 34 12 78 56
```

地址 0x5678:0x1234 正在操作码旁边，与需要 eax 寄存器中间接地址的 FF /5 指令不同。

我们跳过具有 REX 前缀的跳转指令，因为它是一个64位指令。

4.8 检查编译数据

在这个部分，我们将探讨C语言中的数据定义如何映射到其汇编形式。生成的代码是从.bss部分提取的。这意味着显示的汇编代码除了显示这样的

一个值有一个等效的汇编操作码，它表示一个指令。

6 实际上，代码只是数据的一种类型，通常用于劫持正在运行的程序以执行此类代码。然而，在这本书中我们并不需要它。

代码汇编列表不是随机的，而是基于第1卷的 *Chapter 4*、*"Data Type"*。该章节列出了x86硬件操作的基本数据类型，通过学习生成的汇编代码，可以理解C语言如何将语法映射到硬件，然后程序员可以了解为什么C语言适合操作系统编程。本节中使用的特定 objdump 命令将是：

```
$ objdump -z -M intel -S -D -j .data -j .bss <object
file> | less
```

注意：使用三个点符号隐藏零字节：... 要显示所有零字节，我们添加 -z 选项。

4.8.1 Fundamental data types

x86架构处理的最基本类型基于大小，每个类型的大小是前一个的两倍：1字节（8位）、2字节（16位）、4字节（32位）、8字节（64位）和16字节（128位）。

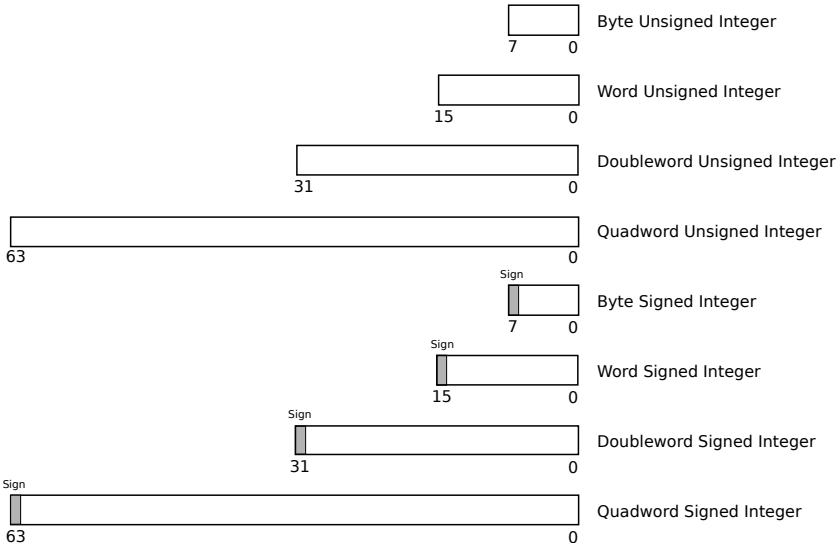


图4.8.1：基本数据类型

这些类型是最简单的：它们只是不同大小的内存块，使CPU能够高效地访问内存。从手册中，*section 4.1.1*，第1卷：

> 单词、双词和四词不需要在内存的自然边界上对齐。单词、双词和四词的自然边界分别是偶数地址、能被4整除的地址和能被8整除的地址。然而，为了提高程序的性能，数据结构（尤其是栈）应尽可能在自然边界上对齐。这是因为处理器需要两次内存访问来执行未对齐的内存访问；对齐的访问只需要一次内存访问。跨越4字节边界的字或双字操作数，或跨越8字节边界的四字操作数被认为是未对齐的，并且需要两个单独的内存总线周期来访问。

> 某些操作双四倍字的操作指令要求内存操作数对齐在自然边界上。如果指定了未对齐的操作数，这些指令将生成一个一般保护异常（`#GP`）。双四倍字的自然边界是任何能被16整除的地址。其他操作双四倍字的操作指令

允许未对齐访问（不会生成一般保护异常）。然而，要从内存中访问未对齐数据，需要额外的内存总线周期。

在C语言中，以下原始类型（必须包括 stdint.h）映射到基本类型：

Source

```
#include <stdint.h>

uint8_t byte = 0x12;
uint16_t word = 0x1234;
uint32_t dword = 0x12345678;
uint64_t qword = 0x123456789abcdef;
unsigned __int128 dqword1 = (__int128) 0x123456789abcdef;
unsigned __int128 dqword2 = (__int128) 0x123456789abcdef << 64;

int main(int argc, char *argv[]) {
       return 0;
}
```

Assembly

```
0804a018 <byte>:
 804a018:       12 00                   adc    al,BYTE PTR [eax]
0804a01a <word>:
 804a01a:       34 12                   xor    al,0x12
0804a01c <dword>:
 804a01c:       78 56                   js     804a074 <_end+0x48>
 804a01e:       34 12                   xor    al,0x12
0804a020 <qword>:
 804a020:       ef                      out    dx,eax
 804a021:       cd ab                   int    0xab
 804a023:       89 67 45                mov    DWORD PTR [edi+0x45],esp
 804a026:       23 01                   and    eax,DWORD PTR [ecx]
0000000000601040 <dqword1>:
  601040:       ef                      out    dx,eax
  601041:       cd ab                   int    0xab
  601043:       89 67 45                mov    DWORD PTR [rdi+0x45],esp
  601046:       23 01                   and    eax,DWORD PTR [rcx]
```

```
  601048:       00 00                   add    BYTE PTR [rax],al
  60104a:       00 00                   add    BYTE PTR [rax],al
  60104c:       00 00                   add    BYTE PTR [rax],al
  60104e:       00 00                   add    BYTE PTR [rax],al
0000000000601050 <dqword2>:
  601050:       00 00                   add    BYTE PTR [rax],al
  601052:       00 00                   add    BYTE PTR [rax],al
  601054:       00 00                   add    BYTE PTR [rax],al
  601056:       00 00                   add    BYTE PTR [rax],al
  601058:       ef                      out    dx,eax
  601059:       cd ab                   int    0xab
  60105b:       89 67 45                mov    DWORD PTR [rdi+0x45],esp
  60105e:       23 01                   and    eax,DWORD PTR [rcx]
```

gcc 生成变量 byte、word、dword、qword、dqword1,

dword2, датирано по-рано, с техните съответни стойности п одчертани с еднакви цветове; променливи от същия тип съ що са подчертани с еднакъв цвят. Тъй като това е раздел с д анни, списъкът с汇编 няма значение. Когато byte се деклари ра с uint8_t, gcc гарантира, че размерът на byte е винаги 1 ба йт. Но, внимателен читател може да забележи стойността 00 до стойността 12 в променливата byte. Това е нормално, т ъй като gcc избягва неуреденост в паметта, като добавя доп ълнителни *padding bytes*. За да се направи по-лесно за забеле жка, ние разглеждаме readelf изхода на .data раздела:

```
$ readelf -x .data hello
```

输出是（颜色标记哪些值属于哪些变量）：

```
Hex dump of section '.data':
  0x00601020 00000000 00000000 00000000 00000000 ................
  0x00601030 12003412 78563412 efcdab89 67452301 ..4.xV4.....gE#.
  0x00601040 efcdab89 67452301 00000000 00000000 ....gE#.........
  0x00601050 00000000 00000000 efcdab89 67452301 ............gE#.
```

如readelf输出所示，变量根据其类型和程序声明的顺序分配存储空间

mer（颜色对应变量）。Intel 是小端机器，这意味着较小的地址存储较小值的字节，较大的地址存储较大值的字节。例如，`0x1234` 显示为 `34 12`；也就是说，`34` 首先出现在地址 `0x601032`，然后 `12` 出现在 `0x601033`。字节内的十进制值保持不变，所以我们看到 `34 12` 而不是 `43 21`。一开始这很令人困惑，但你会很快习惯的。

此外，当 `char` 类型已经是 1 字节时，这不是重复了吗？为什么我们还要添加 `int8_t`？事实上，`char` 类型的大小并不保证是 1 字节，而只是保证至少是 1 字节。在 C 语言中，字节被定义为 char 的大小，而 char 被定义为底层硬件平台的最小可寻址单元。有些硬件设备的最小可寻址单元是 16 位或更大，这意味着 char 的大小是 2 字节，而在这样的平台上，“字节”实际上是 2 个 8 位字节的单位。

并非所有架构都支持双四字长类型。然而，gcc仍然提供了对128位数的支持，并在CPU支持它时生成代码（即，CPU必须是64位的）。通过指定类型为`__int128`或`unsigned __int128`的变量，我们得到一个128位变量。如果CPU不支持64位模式，gcc将抛出错误。

C中的数据类型，代表基本数据类型，也称为 *unsigned numbers*。除了数值计算外，无符号数还用作在内存中结构化数据的工具；我们将在本书后面看到这种应用，当各种数据结构被组织成位组时。

在所有上述示例中，当将较小尺寸变量的值赋给较大尺寸变量时，值容易适应较大变量。相反，当将较大尺寸变量的值赋给较小尺寸变量时，会出现两种情况：

- 该值大于具有较小布局的变量的最大值，因此需要截断到变量的大小，导致值不正确。

- 该值小于具有较小布局的变量的最大值，因此适合该变量。

然而，该值可能在运行时未知，可以是值，最好不让编译器处理这种隐式转换，而由程序员显式控制。否则，它将导致难以捕捉的微妙错误，因为错误值可能很少被用来重现错误。

4.8.2 Pointer Data Types

指针是存储内存地址的变量。x86使用两种类型的指针：

Near pointer 是一个16位/32位段偏移量，也称为 *effec- tive address*。

Far pointer 也是一个类似于近指针的偏移量，但具有显式的段选择器。

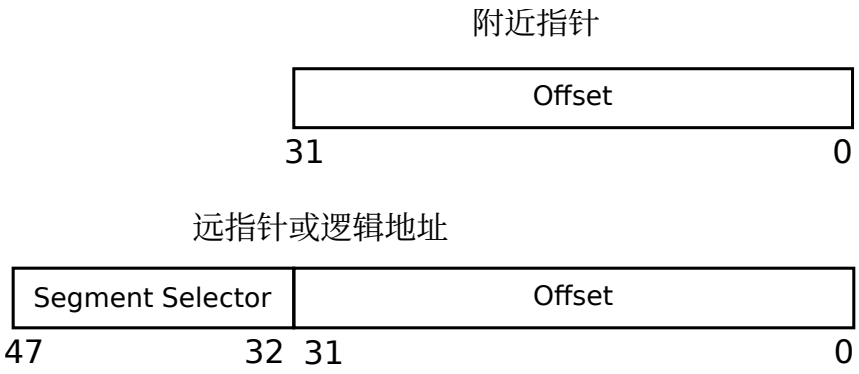


图4.8.2：数值数据类型

C只提供对近指针的支持，因为远指针是平台相关的，例如x86。在应用程序代码中，您可以假设当前段的地址从0开始，因此偏移量实际上是0到最大地址之间的任何内存地址。

Source

```
#include <stdint.h>


int8_t i = 0;
int8_t *p1 = (int8_t *) 0x1234;
int8_t *p2 = &i;
```

```
int main(int argc, char *argv[]) {
       return 0;
}
```

Assembly

```
0000000000601030 <p1>:
  601030:       34 12                   xor    al,0x12
  601032:       00 00                   add    BYTE PTR [rax],al
  601034:       00 00                   add    BYTE PTR [rax],al
  601036:       00 00                   add    BYTE PTR [rax],al
0000000000601038 <p2>:
  601038:       41 10 60 00             adc    BYTE PTR [r8+0x0],spl
  60103c:       00 00                   add    BYTE PTR [rax],al
  60103e:       00 00                   add    BYTE PTR [rax],al
Disassembly of section .bss:
0000000000601040 <__bss_start>:
  601040:       00 00                   add    BYTE PTR [rax],al
0000000000601041 <i>:
  601041:       00 00                   add    BYTE PTR [rax],al
  601043:       00 00                   add    BYTE PTR [rax],al
  601045:       00 00                   add    BYTE PTR [rax],al
  601047:       00                      .byte 0x0
```

指示器 `p1` 持有一个直接地址，其值为 `0x1234`。指示器 `p2` 持有变量 `i` 的地址。请注意，这两个指示器的大小都是 8 字节（如果为 32 位，则为 4 字节）。

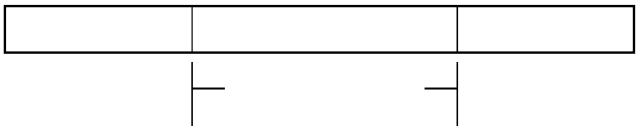
4.8.3 Bit Field Data Type

一个 *bit field* 是一个连续的位序列。位字段允许在位级别上进行数据结构化。例如，32位数据可以包含多个位字段，代表多个不同的信息片段，例如位0-4指定数据结构的大小，位5-6指定权限等等。位级别的数据结构在底层编程中很常见。

Source

```
struct bit_field {
    int data1:8;
```

.



最低有效
位 位字段 字段长度

图4.8.3：数值数据类型（来源：图4-6，第1卷）

```
    int data2:8;
    int data3:8;
    int data4:8;
};

struct bit_field2 {
    int data1:8;
    int data2:8;
    int data3:8;
    int data4:8;
    char data5:4;
};

struct normal_struct {
    int data1;
    int data2;
    int data3;
    int data4;
};

struct normal_struct ns = {
    .data1 = 0x12345678,
    .data2 = 0x9abcdef0,
    .data3 = 0x12345678,
    .data4 = 0x9abcdef0,
};
```

```
int i = 0x12345678;

struct bit_field bf = {
    .data1 = 0x12,
    .data2 = 0x34,
    .data3 = 0x56,
    .data4 = 0x78
};

struct bit_field2 bf2 = {
    .data1 = 0x12,
    .data2 = 0x34,
    .data3 = 0x56,
    .data4 = 0x78,
    .data5 = 0xf
};

int main(int argc, char *argv[]) {
    return 0;
}
```

Assembly 每个变量及其值在as-中都被赋予一种独特的颜色

以下为列表:

```
0804a018 <ns>:
 804a018: 78 56                 js     804a070 <_end+0x34>
 804a01a: 34 12                 xor    al,0x12
 804a01c: f0 de bc 9a 78 56 34  lock fidivr WORD PTR [edx+ebx*4+0x12345678]
 804a023: 12
 804a024: f0 de bc 9a 78 56 34  lock fidivr WORD PTR [edx+ebx*4+0x12345678]
 804a02b: 12
0804a028 <i>:
 804a028: 78 56                 js     804a080 <_end+0x44>
 804a02a: 34 12                 xor    al,0x12
0804a02c <bf>:
 804a02c: 12 34 56              adc    dh,BYTE PTR [esi+edx*2]
```

```
 804a02f: 78 12                  js     804a043 <_end+0x7>
0804a030 <bf2>:
 804a030: 12 34 56               adc    dh,BYTE PTR [esi+edx*2]
 804a033: 78 0f                  js     804a044 <_end+0x8>
 804a035: 00 00                  add    BYTE PTR [eax],al
 804a037: 00                     .byte 0x0
```

样本代码创建了4个变量：ns、i、bf、bf2。normal_struct和bit_field结构体的定义都指定了4个整数。bit_field在其成员名旁边指定了附加信息，由冒号分隔，例如.data1 : 8。这些额外信息是每个位组的位宽。这意味着，即使定义为int，.data1也只消耗8位信息。如果在.data1之后指定了额外的数据成员，将发生两种情况：

- 如果新数据成员在 .data 之后剩余的位中，即 24 位7，内，那么 bit_field 结构的总大小仍然是 4 字节，或者 32 位。
- 如果新的数据成员不匹配，则剩余的24位（3字节）仍然被分配。然而，新的数据成员将分配全新的存储空间，而不使用之前的24位。

7 由于 .data1 被声明为 int 类型，仍然分配了 32 位，但 .data1 只能访问 8 位信息。

在示例中，4个数据成员：.data1、.data2、.data3和.data4，每个都可以访问8位信息，并且一起可以访问首次由.data1声明的整数的所有4个字节。正如生成的汇编代码所示，bf的值遵循C代码中编写的自然顺序：12 34 56 78，因为每个值都是独立的成员。相比之下，i的值是一个整体数字，因此它受小端序规则的约束，因此包含78 56 34 12的值。请注意，在804a02f处是bf中最后一个字节的地址，但旁边是一个数字12，尽管78是其中的最后一个数字。这个额外的数字12不属于bf的值。objdump只是混淆了78是一个操作码；78对应于js指令，它需要一个操作数。因此，objdump抓取78之后的下一个字节并将其放在那里。objdump是在所有汇编代码之后显示的工具。更好的工具是在下一章中我们将学习的gdb。但就本章而言，objdump足够了。

与 bf 不同，ns 中的每个数据成员都完全按整数分配，每个 4 字节，总共 16 字节。正如我们所见，位字段和普通结构不同：位字段结构数据在位级别，而普通结构在字节级别工作。

最后，bf2[8]的结构与bf[9]相同，除了它包含一个更多数据成员：.data5，定义为一种 char。因此，为 .data5 分配了另外 4 个字节，尽管它只能访问 4 位信息，bf2 的最终值是：12 34 56 78 0f 00 00 00。剩余的 3 个字节必须通过指针访问，或者转换为可以完全访问所有 4 个字节的另一种数据类型。

[8] 位_字段2
[9] 位字段

练习4.8.1. 当bit_field结构体和bf变量的定义更改时会发生什么：

```
struct bit_field {
    int data1:8;
};
struct bit_field bf = {
    .data1 = 0x1234,
};
```

什么将是 .data1 的值?

练习4.8.2. 当bit_field2结构体的定义改变为时，会发生什么?

```
struct bit_field2 {
    int data1:8;
    int data5:32;
};
```

什么是类型 bit_field2? 变量的布局

4.8.4 String Data Types

尽管名称相同，x86定义的字符串与C中的字符串不同。x86将字符串定义为*"continuous sequences of bits, bytes, words, or doublewords"*。另一方面，C将字符串定义为以零为最后一个元素的1字节字符数组，以

创建一个 *null-terminated string*。这意味着 x86 中的字符串是数组，而不是 C 字符串。程序员可以使用 `char` 或 `uint8_t`、`short` 或 `uint16_t` 以及 `int` 或 `uint32_t` 定义字节数组、字或双字数组，但不能定义位数组。然而，这种功能可以很容易地实现，因为位数组本质上可以是任何字节数组、字或双字数组，但它在位级别上操作。

以下代码演示了如何定义数组（字符串）数据类型：

Source

```
#include <stdint.h>


uint8_t a8[2] = {0x12, 0x34};
uint16_t a16[2] = {0x1234, 0x5678};
uint32_t a32[2] = {0x12345678, 0x9abcdef0};
uint64_t a64[2] = {0x123456789abcdef0, 0x123456789abcdef0
    };


int main(int argc, char *argv[])
{
    return 0;
}
```

Assembly

```
0804a018 <a8>:
 804a018: 12 34 00               adc    dh,BYTE PTR [eax+eax*1]
 804a01b: 00 34 12               add    BYTE PTR [edx+edx*1],dh
0804a01c <a16>:
 804a01c: 34 12                  xor    al,0x12
 804a01e: 78 56                  js     804a076 <_end+0x3a>
0804a020 <a32>:
 804a020: 78 56                  js     804a078 <_end+0x3c>
 804a022: 34 12                  xor    al,0x12
 804a024: f0 de bc 9a f0 de bc   lock fidivr WORD PTR [edx+ebx*4-0x65432110]
 804a02b: 9a
0804a028 <a64>:
 804a028: f0 de bc 9a 78 56 34   lock fidivr WORD PTR [edx+ebx*4+0x12345678]
 804a02f: 12
 804a030: f0 de bc 9a 78 56 34   lock fidivr WORD PTR [edx+ebx*4+0x12345678]
```

```
804a037: 12
```

尽管 a8 是一个包含2个元素的数组，每个元素都是1字节长，但它仍然分配了4字节。再次，为了确保最佳性能的自然对齐，gcc 补充了额外的零字节。如汇编列表所示，a8 的实际值是 12 34 00 00，其中 a8[0] 等于 12，a8[1] 等于 34。

然后是 a16，包含2个元素，每个元素都是2字节长。由于2个元素总共是4字节，这是自然对齐的，gcc 不填充任何字节。a16 的值是 34 12 78 56，其中 a16[0] 等于 34 12 且 a16[1] 等于 78 56。请注意，objdump 再次混淆了，因为 de 是指令 fidivr (short of reverse divide) 的操作码，该操作码需要另一个操作数，所以 objdump 抓取它认为有意义的下一个字节来创建“一个操作数”。只有高亮显示的值属于 a32。

接下来是 a32，包含 2 个元素，每个元素 4 字节。类似于上面的数组，a32[0] 的值为 78 56 34 12，a32[1] 的值为 f0 de bc 9a，这正是 C 代码中分配的值。

最后是 a64，也有 2 个元素，但每个元素 8 字节。a64 的总大小为 16 字节，这是自然对齐的，因此没有添加填充字节。a64[0] 和 a64[1] 的值相同：f0 de bc 9a 78 56 34 12，被错误地解释为 fidivr 指令。

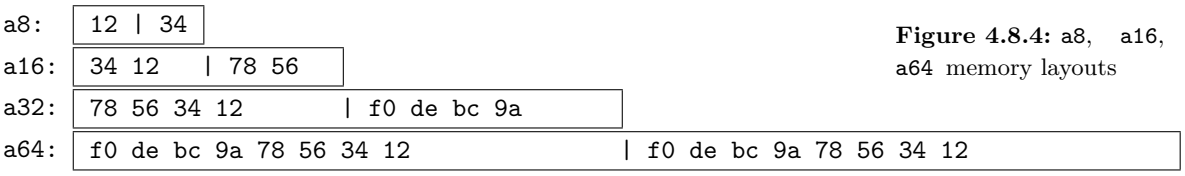


Figure 4.8.4: a8, a16, a32 and a64 memory layouts

然而，超出一维数组直接映射到硬-
ware 字符串类型，C 提供了自己的语法来表示多维数组：

Source

```
#include <stdint.h>


uint8_t a2[2][2] = {
    {0x12, 0x34},
    {0x56, 0x78}
};
```

```
uint8_t a3[2][2][2] = {
    {{0x12, 0x34},
     {0x56, 0x78}},
    {{0x9a, 0xbc},
     {0xde, 0xff}},
};

int main(int argc, char *argv[]) {
    return 0;
}
```

Assembly

```
0804a018 <a2>:
 804a018: 12 34 56                adc    dh,BYTE PTR [esi+edx*2]
 804a01b: 78 12                   js     804a02f <_end+0x7>
0804a01c <a3>:
 804a01c: 12 34 56                adc    dh,BYTE PTR [esi+edx*2]
 804a01f: 78 9a                   js     8049fbb <_DYNAMIC+0xa7>
 804a021: bc                      .byte 0xbc
 804a022: de ff                   fdivrp st(7),st
```

技术上，多维数组就像普通数组：最终，总大小被转换为平坦分配的字节。一个 2 x 2 数组分配了 4 字节；一个 $2 \times 2 \times 2$ 数组分配了 8 字节，如 a2[10] 和 a3 的汇编列表所示。在底层汇编代码中，

10 再次，objdump 感到困惑，将数字 12 放在 78 旁边，列入 a3 列表中。

表示在 a[4] 和 a[2][2] 之间是相同的。然而，在高级 C 代码中，差异巨大。多维数组的语法使程序员能够以更高级的概念进行思考，而不是手动将高级概念转换为低级代码，并同时在大脑中处理高级概念。

示例4.8.1。以下二维数组可以存储长度为10的2个名字列表：

```
char names[2][10] = {
    "John␣Doe",
    "Jane␣Doe"
```

```
};
```

要访问一个名称，我们只需调整列索引11，例如 names[0]，names[1]. ö

字符 “o” 等等。

没有这样的语法，我们需要创建一个20字节的数组，例如names[20]，每次我们想要访问一个字符，例如检查名称中是否包含数字，我们都需要手动计算索引。这会分散注意力，因为我们不断地需要在实际问题与翻译问题之间切换思考。

由于这是一个重复的模式，C语言通过定义和操作多维数组的语法来抽象这个问题。通过这个例子，我们可以清楚地看到通过语言抽象可以给我们带来的力量。如果程序员能够拥有这样的能力来定义适合当前问题的任何语法，那就太理想了。并不是很多语言提供这样的能力。幸运的是，通过C宏，我们可以部分实现这个目标。

在所有情况下，无论数组具有何种维度，都保证生成连续的字节内存。

练习4.8.3。多维数组与指针数组，甚至指针的指针之间有什么区别？

11 左索引被称为列索引，因为它根据列改变索引。

12 与列索引相同，右侧索引被称为行索引，因为它根据行改变索引。

4.9 检查编译代码

本节将探讨编译器如何将高级代码转换为CPU可执行的汇编代码，并了解常见的汇编模式如何帮助创建高级语法。在 objdump 中添加了 -S 选项，以更好地展示高级和低级代码之间的联系。

在此部分，将选项 --no-show-raw-insn 添加到 objdump 命令中，以省略指令码以提高清晰度：

```
$ objdump --no-show-raw-insn -M intel -S -D <object
file> | less
```

4.9.1 Data Transfer

上一节探讨了各种类型的数据是如何创建的，以及它们如何在内存中布局。一旦为变量分配了内存存储，它们就必须可访问和可写。数据传输指令在内存和寄存器之间、寄存器之间移动数据（字节、字、双字或四字），有效地从存储源读取并写入另一个存储源。

Source

```
#include <stdint.h>

int32_t i = 0x12345678;

int main(int argc, char *argv[]) {
       int j = i;
       int k = 0xabcdef;

       return 0;
}
```

Assembly 080483db <main>:

```
    #include <stdint.h>
    int32_t i = 0x12345678;
    int main(int argc, char *argv[]) {
     80483db:       push   ebp
     80483dc:       mov    ebp,esp
     80483de:       sub    esp,0x10
            int j = i;
     80483e1:       mov    eax,ds:0x804a018
     80483e6:       mov    DWORD PTR [ebp-0x8],eax
            int k = 0xabcdef;
     80483e9:       mov    DWORD PTR [ebp-0x4],0xabcdef
            return 0;
     80483f0:       mov    eax,0x0
    }
     80483f5:       leave
```

```
80483f6:       ret
80483f7:       xchg   ax,ax
80483f9:       xchg   ax,ax
80483fb:       xchg   ax,ax
80483fd:       xchg   ax,ax
80483ff:       nop
```

通用数据移动是通过 mov 指令执行的。请注意，尽管指令被称为 mov，但它实际上是从一个目的地复制数据到另一个目的地。

红色指令将寄存器 esp 中的数据复制到寄存器 ebp。此 mov 指令在寄存器之间移动数据，并分配了操作码 89。

蓝色指令将数据从一个内存位置（i变量）复制到另一个（j变量）。不存在从内存到内存的数据移动；它需要两个mov指令，一个用于将数据从内存位置复制到寄存器，另一个用于将数据从寄存器复制到目标内存位置。

粉色指令将立即值复制到内存中。最后，绿色指令将立即数据复制到寄存器中。

4.9.2 Expressions

Source

```
int expr(int i, int j)
{
    int add = i + j;
    int sub = i - j;
    int mul = i * j;
    int div = i / j;
    int mod = i % j;
    int neg = -i;
    int and = i & j;
    int or = i | j;
    int xor = i ^ j;
    int not = ~i;
    int shl = i << 8;
```

```
    int shr = i >> 8;
    char equal1 = (i == j);
    int equal2 = (i == j);
    char greater = (i > j);
    char less = (i < j);
    char greater_equal = (i >= j);
    char less_equal = (i <= j);
    int logical_and = i && j;
    int logical_or = i || j;
    ++i;
    --i;
    int i1 = i++;
    int i2 = ++i;
    int i3 = i--;
    int i4 = --i;

    return 0;
}

int main(int argc, char *argv[]) {
    return 0;
}
```

Assembly 整个汇编列表真的很长。因此，我们逐个表达式进行检查。

Expression: `int add = i + j;`

```
80483e1:       mov    edx,DWORD PTR [ebp+0x8]
80483e4:       mov    eax,DWORD PTR [ebp+0xc]
80483e7:       add    eax,edx
80483e9:       mov    DWORD PTR [ebp-0x34],eax
```

汇编代码很简单：变量 `i` 和 `j` 分别存储在 `eax` 和 `edx` 中，然后与 `add` 指令中的值相加，最终结果存储到 `eax` 中。然后，结果保存到局部变量 `add`，它在位置 `[ebp-0x34]`。

Expression: `int sub = i - j;`

```
80483ec:        mov    eax,DWORD PTR [ebp+0x8]
80483ef:        sub    eax,DWORD PTR [ebp+0xc]
80483f2:        mov    DWORD PTR [ebp-0x30],eax
```

类似于 `add` 指令，x86 提供了 `sub` 指令用于减法。因此，`gcc` 将减法转换为 `sub` 指令，其中 `eax` 被重新加载为 `i`，因为 `eax` 仍然携带前一个表达式的结果。然后，从 `i` 中减去 `j`。减法之后，值被保存到变量 `sub`，位置为 `[ebp-0x30]`。

Expression: `int mul = i * j;`

```
80483f5:        mov    eax,DWORD PTR [ebp+0x8]
80483f8:        imul   eax,DWORD PTR [ebp+0xc]
80483fc:        mov    DWORD PTR [ebp-0x34],eax
```

类似于 `sub` 指令，仅重新加载 `eax`，因为它携带了先前计算的结果。`imul` 执行有符号乘[13]。`eax` 首先加载 `i`，然后与 `j` 相乘并将结果存储回 `eax`，然后存储到变量 `mul` 的位置 `[ebp-0x34]`.

[13] 无符号乘法通过 `mul` 指令执行。

Expression: `int div = i / j;`

```
80483ff:        mov    eax,DWORD PTR [ebp+0x8]
8048402:        cdq
8048403:        idiv   DWORD PTR [ebp+0xc]
8048406:        mov    DWORD PTR [ebp-0x30],eax
```

类似于 `imul`，`idiv` 执行符号除法。但是，与上面的 `imul` 不同，`idiv` 只需要一个操作数：

1. 首先，`i`被重新加载到`eax`中。

2. 然后，`cdq` 将 `eax` 中的双字值转换为四字值，存储在寄存器对 `edx:eax` 中，通过将 `eax` 中的有符号位（31^{th}位）复制到 `edx` 的每个位位置。对 `edx:eax` 是被除数，即变量 `i`，而 `idiv` 的操作数是除数，即变量 `j`。

3. 计算完成后，结果存储到对 `edx:eax` 寄存器中，商在 `eax` 中，余数在 `edx` 中。商存储在变量 `div` 中，位置为 `[ebp-0x30]`。

Expression: `int mod = i % j;`

```
8048409:       mov    eax,DWORD PTR [ebp+0x8]
804840c:       cdq
804840d:       idiv   DWORD PTR [ebp+0xc]
8048410:       mov    DWORD PTR [ebp-0x2c],edx
```

相同的 `idiv` 指令也执行模运算，因为它也计算余数并将其存储在变量 `mod` 中，位置 `[ebp-0x2c]`。

Expression: `int neg = -i;`

```
8048413:       mov    eax,DWORD PTR [ebp+0x8]
8048416:       neg    eax
8048418:       mov    DWORD PTR [ebp-0x28],eax
```

`neg` 替换操作数（目标操作数）的值为它的二进制补码（此操作相当于从0减去操作数）。在此示例中，`eax`中的值`i`被`-i`替换，使用`neg`指令。然后，新值存储在变量`neg`的`[ebp-0x28]`位置。

Expression: `int and = i & j;`

```
804841b:        mov    eax,DWORD PTR [ebp+0x8]
804841e:        and    eax,DWORD PTR [ebp+0xc]
8048421:        mov    DWORD PTR [ebp-0x24],eax
```

`and` 在两个操作数上执行位运算 `AND`，并将结果存储在目标操作数中，该目标操作数是变量 `and` 在 `[ebp-0x24]`。

Expression: `int or = i | j;`

```
8048424:       mov    eax,DWORD PTR [ebp+0x8]
8048427:       or     eax,DWORD PTR [ebp+0xc]
804842a:       mov    DWORD PTR [ebp-0x20],eax
```

类似于 `and` 指令，`or` 对两个操作数执行位运算 `OR`，并将结果存储在目标操作数中，在这种情况下是变量 `or` 在 `[ebp-0x20]` 处。

Expression: `int xor = i ^ j;`

```
804842d:       mov    eax,DWORD PTR [ebp+0x8]
8048430:       xor    eax,DWORD PTR [ebp+0xc]
8048433:       mov    DWORD PTR [ebp-0x1c],eax
```

类似于 and/or 指令，xor 对两个操作数执行位运算 XOR，并将结果存储在目标操作数中，该目标操作数是位于 [ebp-0x1c] 的变量 xor。

Expression: `int not = ~i;`

```
8048436:       mov    eax,DWORD PTR [ebp+0x8]
8048439:       not    eax
804843b:       mov    DWORD PTR [ebp-0x18],eax
```

not 在目标操作数上执行位运算 NOT（每个1设置为0，每个0设置为1），并将结果存储在目标操作数位置，即变量 not 在 [ebp-0x18] 处。

Expression: `int shl = i << 8;`

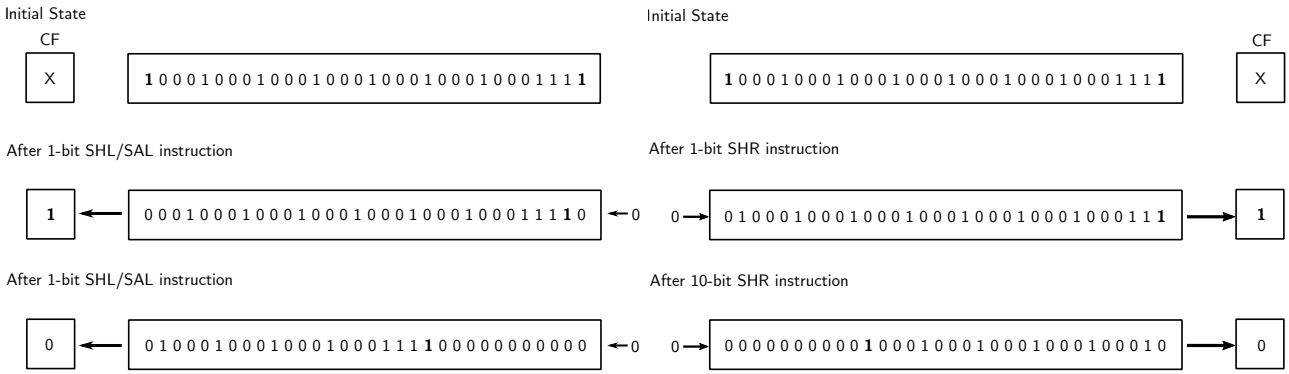
```
804843e:       mov    eax,DWORD PTR [ebp+0x8]
8048441:    shl    eax,0x8
8048444:    mov    DWORD PTR [ebp-0x14],eax
```

shl (位移逻辑左)将目标操作数中的位向左移动由源操作数指定的位数。在这种情况下，eax存储i，shl将eax向左移动8位。shl的另一个名称是sal（（算术左移）。两者可以同义使用。最后，结果存储在变量shl的[ebp-0x14]位置。以下是shl/sal和shr指令的视觉演示：向左移动后，最右边的位在EFLAGS寄存器中设置进位标志。

Expression: `int shr = i >> 8;`

```
8048447:       mov    eax,DWORD PTR [ebp+0x8]
804844a:    sar    eax,0x8
804844d:    mov    DWORD PTR [ebp-0x10],eax
```

sar 与 shl/sal 类似，但向右移位并扩展符号位。对于右移，shr 和 sar 是两个不同的指令。shr 与 sar 的区别在于它不扩展符号位。最后，结果存储在变量 shr 的 [ebp-0x10] 处。



(a) SHL/SAL (Source: Figure 7-6, Volume 1)

(b) SHR (Source: Figure 7-7, Volume 1)

图4.9.1：移位指令（红色是起始位，蓝色是结束位。）

在图4.9.1(b)中，注意最初符号位是1，但在1位和10位移位之后，移出的位被填充为零。

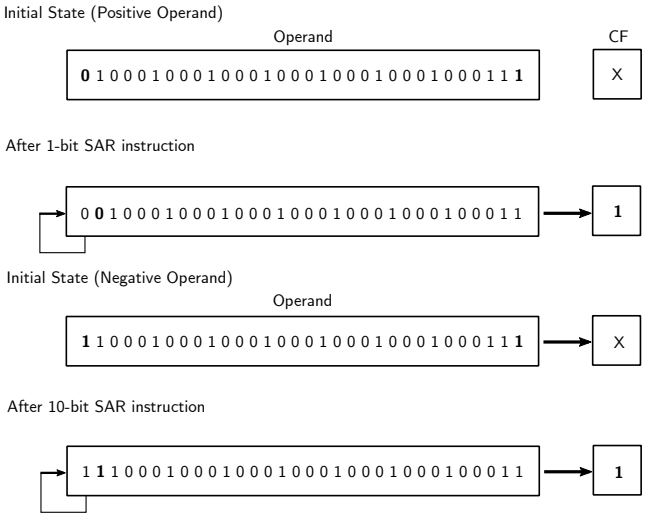


图4.9.2：SAR指令操作（来源：图7-8，第1卷）

使用 `sar`，符号位（最高位）被保留。也就是说，如果符号位为0，新位总是得到值0；如果符号位为1，新位总是得到值1。

Expression: `char equal1 = (i == j);`

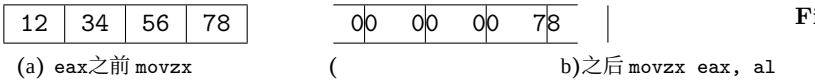
```
8048450:       mov    eax,DWORD PTR [ebp+0x8]
8048453:       cmp    eax,DWORD PTR [ebp+0xc]
8048456:       sete   al
8048459:       mov    BYTE PTR [ebp-0x41],al
```

cmp 并且set指令的变体及其变体的变体构成了所有逻辑比较。在这个表达式中，cmp比较变量i和j；然后如果之前的比较cmp相等，sete将值1存储到al寄存器中，否则存储0。set指令的变体的一般名称称为SETcc。后缀cc表示在EFLAGS寄存器中测试的条件。第1卷附录B，*"EFLAGS Condition Codes"*，列出了可以使用此指令测试的条件。最后，结果存储在[ebp-0x41]处的变量equal1中。

Expression: `int equal2 = (i == j);`

```
804845c:       mov    eax,DWORD PTR [ebp+0x8]
804845f:       cmp    eax,DWORD PTR [ebp+0xc]
8048462:       sete   al
8048465:       movzx  eax,al
8048468:       mov    DWORD PTR [ebp-0xc],eax
```

类似于相等比较，此表达式也用于比较相等，但结果存储在int类型中。因此，增加了一条指令：movzx指令，它是mov的变体，将结果复制到目标操作数并将剩余的字节填充为0。在这种情况下，由于eax是4字节宽，在复制al中的第一个字节后，将eax的剩余字节填充为0，以确保eax具有与al相同的值。



(a) eax之前 movzx　　(　　b)之后 movzx eax, al

Figure 4.9.3: movzx instruction

Expression: `char greater = (i > j);`

```
804846b:       mov    eax,DWORD PTR [ebp+0x8]
804846e:       cmp    eax,DWORD PTR [ebp+0xc]
8048471:       setg   al
8048474:       mov    BYTE PTR [ebp-0x40],al
```

类似于等于比较，但用 setg 进行大于比较。

Expression: `char less = (i < j);`

```
8048477:       mov    eax,DWORD PTR [ebp+0x8]
```

```
804847a:       cmp    eax,DWORD PTR [ebp+0xc]
804847d:       setl   al
8048480:       mov    BYTE PTR [ebp-0x3f],al
```

应用 `setl` 进行较少的比较。

Expression: `char greater_equal = (i >= j);`

```
8048483:       mov    eax,DWORD PTR [ebp+0x8]
8048486:       cmp    eax,DWORD PTR [ebp+0xc]
8048489:       setge  al
804848c:       mov    BYTE PTR [ebp-0x3e],al
```

应用 `setge` 进行大于或等于比较。

Expression: `char less_equal = (i <= j);`

```
804848f:       mov    eax,DWORD PTR [ebp+0x8]
8048492:       cmp    eax,DWORD PTR [ebp+0xc]
8048495:       setle  al
8048498:       mov    BYTE PTR [ebp-0x3d],al
```

应用 `setle` 进行小于或等于比较。

Expression: `int logical_and = (i && j);`

```
804849b:       cmp    DWORD PTR [ebp+0x8],0x0
804849f:       je     80484ae <expr+0xd3>
80484a1:       cmp    DWORD PTR [ebp+0xc],0x0
80484a5:       je     80484ae <expr+0xd3>
80484a7:       mov    eax,0x1
80484ac:       jmp    80484b3 <expr+0xd8>
80484ae:       mov    eax,0x0
80484b3:       mov    DWORD PTR [ebp-0x8],eax
```

逻辑 `AND` 操作符 `&&` 是一种完全由软件[14] 以更简单的指令构成的语法。来自 as- 的算法

组件代码很简单：

1. 首先，检查`i`是否等于`0`，使用位于`0x804849b`的指令。

   (a) 如果为真，跳转到 `0x80484ae` 并将 eax 设置为 0。 (b) 将变量 `logical_and` 设置为 0，因为它是在 `0x80484ae` 之后的下一个指令。

[14] That is, there is no equivalent assembly instruction implemented in hardware.

2. 如果 i 不等于 0，则检查 j 是否等于 0，使用位于 0x80484a1 的指令。

(a) 如果为真，跳转到 0x80484ae 并将 eax 设置为 0。 (b) 将变量 logical_and 设置为 0，因为它位于 0x80484ae 之后的下一个指令。3. 如果 i 和 j 都不是 0，则结果肯定是 1，或者 true。 (a) 根据位于 0x80484a7 的指令相应地设置它。 (b) 然后跳转到位于 0x80484b3 的指令，将变量 logical_and 在 [ebp-0x8] 处设置为 1。

Expression: `int logical_or = (i || j);`

```
80484b6:       cmp    DWORD PTR [ebp+0x8],0x0
80484ba:       jne    80484c2 <expr+0xe7>
80484bc:       cmp    DWORD PTR [ebp+0xc],0x0
80484c0:       je     80484c9 <expr+0xee>
80484c2:       mov    eax,0x1
80484c7:       jmp    80484ce <expr+0xf3>
80484c9:       mov    eax,0x0
80484ce:       mov    DWORD PTR [ebp-0x4],eax
```

逻辑 OR 操作符 || 与逻辑与类似。理解算法留给读者作为练习。

Expression: `++i;` 并且 `--i;`（或 `i++` 和 `i--`）

```
80484d1:       add    DWORD PTR [ebp+0x8],0x1
80484d5:       sub    DWORD PTR [ebp+0x8],0x1
```

增量减量语法与逻辑 AND 和逻辑 OR 类似，因为它是由现有指令 add 构成的。区别在于 CPU 实际上确实有一个内置指令，但 gcc 决定不使用该指令，因为 inc 和 dec 会导致 *partial flag register stall*，当指令修改标志寄存器的一部分，而后续指令依赖于标志的结果时发生（*section 3.5.2.6*，英特尔优化手册，2016b）。手册甚至建议用 add 和 sub 指令替换 inc 和 dec（第 *3.5.1.1* 节，英特尔优化手册，2016b）。

Expression: `int i1 = i++;`

```
80484d9:       mov    eax,DWORD PTR [ebp+0x8]
80484dc:       lea    edx,[eax+0x1]
80484df:       mov    DWORD PTR [ebp+0x8],edx
80484e2:       mov    DWORD PTR [ebp-0x10],eax
```

首先，`i` 在 `80484d9` 处被复制到 `eax`。然后，`eax + 0x1` 的值作为一个 *effective address* 在 `80484dc` 处被复制。`lea` (*load effective address*) 指令将一个内存地址复制到一个寄存器。根据第2卷，源操作数是使用处理器的一种寻址模式指定的内存地址。这意味着，源操作数必须由16位/32位ModR/M字节表中的寻址模式定义指定，4.5.1和4.5.2。

在将增量值加载到 `edx` 后，`i` 的值在 `80484df` 处增加 1。最后，通过 `80484e2` 处的指令，将 *previous* `i` 值存储回 `i1` 在 `[ebp-0x8]` 处。

Expression: `int i2 = ++i;`

```
80484e5:       add    DWORD PTR [ebp+0x8],0x1
80484e9:       mov    eax,DWORD PTR [ebp+0x8]
80484ec:       mov    DWORD PTR [ebp-0xc],eax
```

主要区别于这种增量语法和之前的一个是：

▷ `add` 用于代替 `lea` 以直接增加 `i`。

▷ 新增加的 `i` 被存储到 `i2` 中，而不是旧值。▷ 表达式仅需要 3 条指令，而不是 4 条。

这个前缀增量语法比之前使用的后缀语法更快。如果增量只在小循环中使用一次或几次，使用哪个版本可能没有太大关系，但如果循环运行数百万次或更多，那就很重要了。此外，根据不同情况，使用其中一个比另一个更方便，例如，如果 `i` 是访问数组的索引，我们希望使用旧值来访问前一个数组元素，而使用新增加的 `i` 来访问当前元素。

Expression: `int i3 = i--;`

```
80484ef:       mov    eax,DWORD PTR [ebp+0x8]
```

```
80484f2:       lea    edx,[eax-0x1]
80484f5:       mov    DWORD PTR [ebp+0x8],edx
80484f8:       mov    DWORD PTR [ebp-0x8],eax
```

与 i++ 语法类似，留作读者练习。

Expression: `int i4 = --i;`

```
80484fb:       sub    DWORD PTR [ebp+0x8],0x1
80484ff:       mov    eax,DWORD PTR [ebp+0x8]
8048502:       mov    DWORD PTR [ebp-0x4],eax
```

类似于 ++i 语法，留作读者练习。

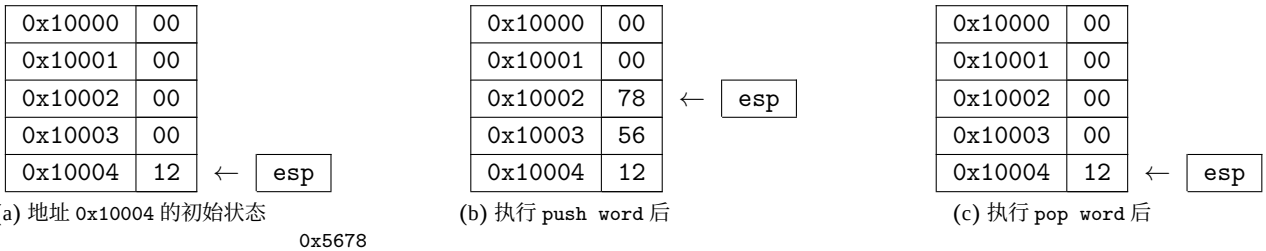
练习4.9.1. 阅读第*3.5.2.4, "Partial Register Stalls"*节以了解通用寄存器停滞。

练习4.9.2。阅读第1卷中从*7.3.1*到*7.3.7*的章节。

4.9.3 Stack

堆栈是一系列连续的内存位置，用于存储离散数据集合。当添加新元素时，堆栈 *grows down* 在内存中向较小地址方向移动，当删除元素时，*shrinks up* 向较大地址方向移动。x86 使用 esp 寄存器指向堆栈顶部，即最新元素。堆栈可以位于主内存的任何位置，因为 esp 可以设置为任何内存地址。x86 提供两种操作来操作堆栈：

▷ push 指令及其变体在栈顶添加一个新元素 ▷ pop，指令及其变体从栈中移除最顶部的元素。



(a) 地址 0x10004 的初始状态　(b) 执行 push word 后　(c) 执行 pop word 后

图4.9.4：栈操作

4.9.4 Automatic variables

局部变量是存在于某个作用域内的变量。作用域由一对花括号界定：`{..}`。定义局部变量最常见的作用域是函数作用域。然而，作用域可以是未命名的，在未命名的作用域内创建的变量在其作用域及其内部作用域之外不存在。

示例4.9.1. 函数作用域：

```
void foo() {
    int a;
    int b;
}
```

`a` `b` 是函数 `foo` 的局部变量。

示例 4.9.2. 未命名范围：

```
int foo() {
    int i;

    {
        int a = 1;
        int b = 2;
        {
            return i = a + b;
        }
    }
}
```

`a` 并且 `b` 在其定义的地方是局部的，并且在其内部子作用域中返回 `i = a + b`。然而，它们在创建 `i` 的函数作用域中不存在。

当创建局部变量时，它被压入栈中；当局部变量超出作用域时，它从栈中弹出，从而被销毁。当调用者向被调用者传递参数时，它被压入栈中；当被调用者返回调用者时，参数被弹出

栈。局部变量和参数在进入函数时自动分配，在退出函数后销毁，这就是为什么它被称为 *automatic variables*。

一个基帧指针指向当前函数帧的起始位置，并保存在 ebp 寄存器中。每当调用一个函数时，它都会在栈上为其分配专用的存储空间，称为 *stack frame*。栈帧是函数所有局部变量和参数放置在栈上的位置15。

15 数据和仅数据被专用于每个栈帧的栈上分配。此处不驻留任何代码。

当函数需要一个局部变量或参数时，它使用 ebp 来访问变量：

▷ 所有局部变量都在 ebp 指针之后分配。因此，要访问局部变量，需要从 ebp 中减去一个数字以到达变量的位置。

▷ 所有参数在 ebp 指针之前分配。要访问一个参数，需要将一个数字加到 ebp 上以到达参数的位置。

▷ ebp本身指针指向其调用者的返回地址。

Previous Frame					Current Frame						
Function Arguments						ebp	Local variables				
A1	*A2*	*A3*	.....	*An*	*Return Address*	Old ebp	*L1*	*L2*	*L3*	.....	*Ln*

图4.9.5：函数参数和局部变量

A = 参数 L = 局部变量以下是一个具体示例：

Source

```
int add(int a, int b) {
    int i = a + b;

    return i;
}
```

Assembly

```
080483db <add>:
#include <stdint.h>
int add(int a, int b) {
 80483db:       push   ebp
```

```
 80483dc:       mov    ebp,esp
 80483de:       sub    esp,0x10
    int i = a + b;
 80483e1:       mov    edx,DWORD PTR [ebp+0x8]
 80483e4:       mov    eax,DWORD PTR [ebp+0xc]
 80483e7:       add    eax,edx
 80483e9:       mov    DWORD PTR [ebp-0x4],eax
    return i;
 80483ec:       mov    eax,DWORD PTR [ebp-0x4]
}
 80483ef:       leave
 80483f0:       ret
```

在汇编列表中，`[ebp-0x4]` 是局部变量 `i`，因为它分配了 *after* `ebp`，长度为 4 字节（一个 `int`）。另一方面，

`a` 并且 `b` 是参数，可以使用 `ebp` 访问：

▷ `[ebp+0x8]` 访问 `a`。

▷ `[ebp+0xc]` 访问 `b`。

对于访问参数，规则是堆栈上变量越靠近，

到 `ebp`，越接近函数名。

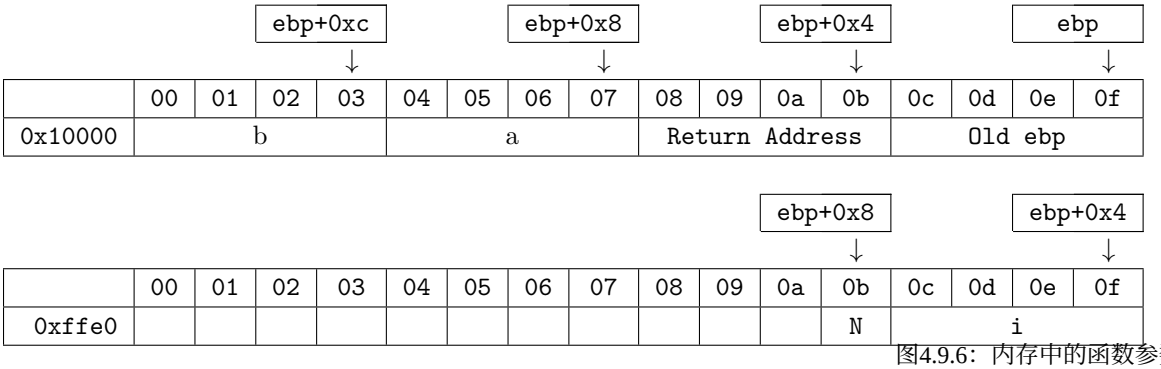


图4.9.6：内存中的函数参数和局部变量

`N` = 下一个局部变量从这里开始

从图中可以看出，`a` 和 `b` 在内存中的布局与 C 中书写的顺序完全一致，相对于返回地址。

4.9.5 Function Call and Return

Source

```
#include <stdio.h>

int add(int a, int b) {
    int local = 0x12345;

    return a + b;
}

int main(int argc, char *argv[]) {
    add(1,1);

    return 0;
}
```

Assembly 对于每个函数调用，gcc 使用 push 指令以相反的顺序将参数压入栈中。也就是说，压入栈的参数顺序与在高级 C 代码中编写的顺序相同，以确保参数之间的相对顺序，正如前一个章节中关于函数参数和局部变量布局的描述。然后，gcc 生成一个 call 指令，该指令随后隐式地压入返回地址，在将控制权转移到 add 函数之前：

```
080483f2 <main>:
int main(int argc, char *argv[]) {
 80483f2:       push   ebp
 80483f3:       mov    ebp,esp
    add(1,2);
 80483f5:       push   0x2
 80483f7:       push   0x1
 80483f9:       call   80483db <add>
 80483fe:       add    esp,0x8
    return 0;
 8048401:       mov    eax,0x0
}
 8048406:       leave
```

```
8048407:       ret
```

在完成对 add 函数的调用后，通过将 0x8 添加到栈指针 esp（来恢复栈，这相当于 2 条 pop 指令）。最后，执行一条 leave 指令，并通过一条 ret 指令返回 main。一条 ret 指令将程序执行转移回调用者，回到 call 指令之后的指令 add。ret 能够返回到该位置的原因是 call 指令隐式推送的返回地址，即 call 指令之后的地址；每当 CPU 执行 ret 指令时，它都会检索堆栈上所有参数之后的返回地址：

函数结束时，gcc放置一条leave指令来清理为局部变量分配的所有空间，并将帧指针恢复为调用者的帧指针。

```
080483db <add>:
#include <stdio.h>
int add(int a, int b) {
 80483db:       push   ebp
 80483dc:       mov    ebp,esp
 80483de:       sub    esp,0x10
    int local = 0x12345;
 80483e1:       DWORD PTR [ebp-0x4],0x12345
    return a + b;
 80483e8:       mov    edx,DWORD PTR [ebp+0x8]
 80483eb:       mov    eax,DWORD PTR [ebp+0xc]
 80483ee:       add    eax,edx
}
 80483f0:       leave
 80483f1:       ret
```

练习4.9.3。上述由gcc生成的函数调用代码实际上是x86定义的标准方法。阅读第6章，*“Produce Calls, Interrupts, and Exceptions”*,英特尔手册第1卷。

4.9.6 Loop

循环只是将指令指针重置为已执行的指令，并从那里重新开始。循环只是 jmp 指令的一个应用。然而，由于循环是一个普遍的模式，它在 C 语言中获得了自己的语法。

Source

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    for (int i = 0; i < 10; i++) {
    }

    return 0;
}
```

Assembly

```
080483db <main>:
#include <stdio.h>
int main(int argc, char *argv[]) {
 80483db:       push   ebp
 80483dc:       mov    ebp,esp
 80483de:       sub    esp,0x10
   for (int i = 0; i < 10; i++) {
 80483e1:       mov    DWORD PTR [ebp-0x4],0x0
 80483e8:       jmp    80483ee <main+0x13>
 80483ea:       add    DWORD PTR [ebp-0x4],0x1
 80483ee:       cmp    DWORD PTR [ebp-0x4],0x9
 80483f2:       jle    80483ea <main+0xf>
   }
   return 0;
 80483f4: b8 00 00 00 00        mov    eax,0x0
}
 80483f9: c9                    leave
 80483fa: c3                    ret
 80483fb: 66 90                 xchg   ax,ax
 80483fd: 66 90                 xchg   ax,ax
```

```
80483ff: 90                     nop
```

颜色标记对应的高级代码到汇编代码：

1. 红色指令将 i 初始化为 0。

2. 绿色指令通过使用 jle 将 i 与 10 进行比较，并将其与 9 进行比较。如果为真，则跳转到 80483ea 进行另一次迭代。3. 蓝色指令将 i 增加 1，使得当满足终止条件时循环能够终止。

练习4.9.4.为什么增量指令（蓝色指令）出现在比较指令（绿色指令）之前?

练习4.9.5。可以为 while 和 do...while 生成哪些汇编代码?

4.9.7 Conditional

再次，C语言中的条件语句使用 if...else... 构造只是底层 jmp 指令的另一种应用。它也是一个无处不在的模式，在C语言中获得了自己的语法。

Source

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i = 0;

    if (argc) {
        i = 1;
    } else {
        i = 0;
    }

    return 0;
}
```

Assembly

```
int main(int argc, char *argv[]) {
80483db:        push   ebp
```

```
80483dc:        mov    ebp,esp
80483de:        sub    esp,0x10
   int i = 0;
80483e1:        mov    DWORD PTR [ebp-0x4],0x0
   if (argc) {
80483e8:        cmp    DWORD PTR [ebp+0x8],0x0
80483ec:        je     80483f7 <main+0x1c>
       i = 1;
80483ee:        mov    DWORD PTR [ebp-0x4],0x1
80483f5:        jmp    80483fe <main+0x23>
   } else {
       i = 0;
80483f7:        mov    DWORD PTR [ebp-0x4],0x0
   }
   return 0;
80483fe:        mov    eax,0x0
}
8048403:        leave
8048404:        ret
```

生成的汇编代码遵循相同的顺序，与对应-

高级语法：

- 红色指令表示 `if` 分支。
- 蓝色指令表示 `else` 分支。
- 绿色指令是 `if` 和 `else` 分支的出口点。

`if` 分支首先使用 `cmp` 指令比较 `argc` 是否等于 0)。如果为真，则继续执行 `80483f7` 处的 `else` 分支。否则，`if` 分支继续执行其分支代码，即 `80483ee` 处的下一条指令，用于将 `1` 复制到 `i`。最后，跳过 `else` 分支，继续执行 `80483fe`，这是 `if..else...` 构造之后的下一条指令。

`else` 当从 `if` 分支的 `cmp` 指令为真时进入分支。`else` 分支从 `80483f7` 开始，它是 `else` 分支的第一个指令。该指令将 `0` 复制到 `i`，并自然地继续执行到 `if...else...` 构造之后的下一个指令，而不进行任何跳转。

5

The Anatomy of a Program

每个程序由代码和数据组成，只有这两个组件构成了一个程序。然而，如果一个程序仅由它自己的代码和数据组成，从操作系统的角度来看（以及人类），它不知道在程序中哪个二进制块是程序，哪个只是原始数据，程序从哪里开始执行，哪个内存区域应该被保护，哪个可以自由修改。因此，每个程序都携带额外的元数据来与操作系统通信，说明如何处理该程序。

当源文件编译时，生成的机器代码存储到一个*object file*中，它只是一个二进制块。一个或多个目标文件

object file

可以组合产生一个 *executable binary*，它是一个完整的

executable binary

程序可在操作系统上运行。

`readelf` 这是一个识别并显示二进制文件（无论是对象文件还是可执行二进制文件）的ELF元数据的程序。***ELF***或***E****xecutable and* ***L****inkable* ***F****ormat*是可执行文件开头的内容，为操作系统提供加载到主内存并运行可执行文件所必需的信息。ELF可以被视为类似书籍的目录。在书中，目录列出主要章节、子章节的页码，有时甚至列出图表和表格以便于查找。同样，ELF列出用于代码和数据的各个部分，以及每个符号的内存地址以及其他信息。

构成。

ELF可执行文件由以下组成：▷ 一个 *ELF header*：可执行文件的第一部分，描述了 ***ELF header*** 文件的组织。▷ 一个 *program header table*：是一个固定大小的结构数组，描述了可执行文件的段。▷ 一个 *section header table*：是一个固定大小的结构数组，描述了可执行文件的节。

▷ *Segments and sections*s是ELF二进制文件的主要内容，其中代码和数据被划分为不同目的的块。

Segments and sections

一个 *segment* 是由零个或多个部分组成的，并在运行时由操作系统直接加载。

一个 *section* 是一个二进制块，可以是以下之一：– 当程序运行时在内存中可用的实际程序代码和数据。– 仅在链接过程中使用的关于其他部分的元数据，并从最终可执行文件中消失。

链接器使用部分来构建段。

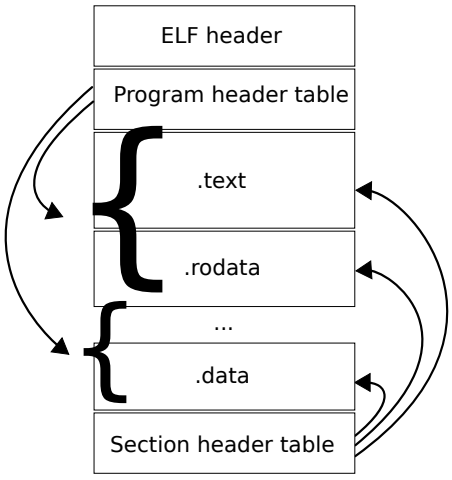


图5.0.1：ELF - 链接视图与可执行视图（来源：维基百科）

稍后我们将使用GCC将内核编译为ELF可执行文件，并明确指定段是如何创建以及它们在哪里加载的

在内存中使用一个 *linker script*，一个文本文件来指导链接器如何生成二进制文件。目前，我们将详细检查 ELF 可执行文件的结构。

5.1 参考文件：

ELF规范在Linux中捆绑为man页：

ELF specification

```
$ man elf
```

这是一个理解和实现ELF的有用资源。然而，在完成本章后，使用它将更容易，因为其中混合了实现细节。

默认规范是一个通用的规范，其中每个ELF实现都遵循。然而，每个平台都提供了独特的额外功能。x86的ELF规范目前由H.J. Lu在Github上维护：`https://github.com/hjl-tools/x86-psABI/wiki/X86-psABI`。

平台相关的细节在通用ELF规范中被称为“处理器特定”。我们不会探讨这些细节，但会研究通用细节，这些细节足以为我们操作系统的ELF二进制图像制作提供所需信息。

5.2 ELF头

要查看ELF头信息：

```
$ readelf -h hello
```

输出：

Output

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
```

```
    Version:                           1 (current)
    OS/ABI:                            UNIX - System V
    ABI Version:                       0
    Type:                              EXEC (Executable file)
    Machine:                           Advanced Micro Devices X86-64
    Version:                           0x1
    Entry point address:               0x400430
    Start of program headers:          64 (bytes into file)
    Start of section headers:          6648 (bytes into file)
    Flags:                             0x0
    Size of this header:               64 (bytes)
    Size of program headers:           56 (bytes)
    Number of program headers:         9
    Size of section headers:           64 (bytes)
    Number of section headers:         31
  Section header string table index: 28
```

让我们逐个字段进行了解:

Magic 显示唯一标识文件的原始字节的ELF可执行二进制文件。每个字节提供简要信息。

在示例中，我们有以下魔法字节:

输出

```
    Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
```

Examine byte by byte:

Byte	Description
`7f 45 4c 46`	Predefined values. The first byte is always `7F`, the remaining 3 bytes represent the string `"ELF"`.
`02`	See `Class` field below.
`01`	See `Data` field below.
`01`	See `Version` field below.
`00`	See `OS/ABI` field below.
`00 00 00 00 00 00 00 00`	Padding bytes. These bytes are unused and are always set to 0. Padding bytes are added for proper alignment, and is reserved for future use when more information is needed.

Class 字节在 `Magic` 字段中。它指定了文件的类别或容量。

可能的值:

Value	Description
`0`	Invalid class
`1`	32-bit objects
`2`	64-bit objects

Data 一个字节在 `Magic` 字段中。它指定了处理器的数据编码-特定对象文件中的数据。

可能的值:

Value	Description
`0`	Invalid data encoding
`1`	Little endian, 2's complement
`2`	Big endian, 2's complement

Version 一个字节在 `Magic` 中。它指定了 ELF 头版本 数字。

可能的值:

Value	Description
0	Invalid version
1	Current version

OS/ABI 一个字节在 `Magic` 字段中。它指定了目标操作系统 `ABI`。最初，它是一个填充字节。

可能的值：请参阅最新的ABI文档，因为它是一个不同操作系统的长列表。

Type 标识对象文件类型。

Value	Description
`0`	No file type
`1`	Relocatable file
`2`	Executable file
`3`	Shared object file
`4`	Core file
`0xff00`	Processor specific, lower bound
`0xffff`	Processor specific, upper bound

`0xff00` 到 `0xffff` 的值保留给处理器定义对其有意义的附加文件类型。

Machine 指定ELF文件的所需架构值，例如x86_64、MIPS、SPARC等。在示例中，该机器的架构为`x86_64`。

可能值：请参阅最新的ABI文档，因为它是一个不同架构的长列表。

Version 指定当前 *object file* (的版本号，而不是 ELF 头的版本，因为上述 `Version` 字段指定了)。

Entry point address 指定要执行的第一个代码的内存地址。在正常的应用程序中，`main`函数的地址是默认值，但可以通过显式指定函数名到`gcc`来指定任何函数。对于我们将要编写的操作系统，这是我们启动内核时需要检索的最重要字段，其他所有内容都可以忽略。

Start of program headers 程序头表偏移量，以字节为单位。在示例中，这个数字是 64 字节，这意味着 65th 字节，或 `<start address> + 64`，是程序头表的起始地址。也就是说，如果程序在内存地址 0x10000 处加载，则起始地址是 Magic 字段的第一字节，其中存储着值 0x7f，程序头表的起始地址是 `0x10000 + 0x40 = 0x10040`。

Start of section headers 节标题表偏移量（以字节为单位），类似于程序标题的开始。在示例中，它是文件中的 6648 字节。

Flags 保留与文件相关的处理器特定标志。当程序加载时，在x86机器中，根据此值设置EFLAGS寄存器。在示例中，该值为0x0，这意味着EFLAGS寄存器处于清除状态。

Size of this header 指定ELF头总大小（以字节为单位）。在示例中，它是64字节，相当于程序头起始位置。请注意，这两个数字不一定相等，因为程序头表可能放置在ELF头很远的地方。在ELF可执行二进制文件中，唯一固定的组件是ELF头，它出现在文件的起始位置。

Size of program headers 指定 *each* 程序头的大小（以字节为单位）。在示例中，它为 64 字节。

Number of program headers 指定程序头的总数。在示例中，该文件共有 9 个程序头。

Size of section headers 指定 *each* 节标题的大小（以字节为单位）。在示例中，它是 64 字节。

Number of section headers 指定总部分标题数。在示例中，文件总共有 31 个部分标题。在部分标题表中，表中的第一个条目始终是一个空部分。

Section header string table index 指定了在节标题表中指向包含所有内容的节的标题的索引

空终止字符串。在示例中，索引是 28，这意味着它是表的第28th个条目。

5.3 章节标题表

我们知道，代码和数据组成一个程序。然而，并非所有类型的代码和数据都有相同的目的。因此，而不是一大块代码和数据，它们被分成更小的块，并且每个块都必须满足这些条件（根据gABI）：

▷ 每个对象文件中的每个部分都有且只有一个描述它的部分头。但是，可能存在没有部分的部分头。▷每个部分占用文件中一个连续的（可能为空）字节序列。这意味着，没有两个字节区域属于同一个部分。▷文件中的部分可能不重叠。文件中的任何字节都不会属于多个部分。▷对象文件可能有非活动空间。各种头和部分可能不会“覆盖”对象文件中的每个字节。非活动数据的内容未指定。

要从可执行二进制文件中获取所有标题，例如 `hello`，请使用以下命令：

```
$ readelf -S hello
```

这里是一个示例输出（如果不懂输出内容不用担心。只需快速浏览以熟悉它。我们很快就会对其进行剖析）：

Output

```
There are 31 section headers, starting at offset 0x19c8:
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
```

```
[ 0]                   NULL             0000000000000000  00000000
     0000000000000000  0000000000000000           0     0     0
[ 1] .interp           PROGBITS         0000000000400238  00000238
     000000000000001c  0000000000000000   A       0     0     1
[ 2] .note.ABI-tag     NOTE             0000000000400254  00000254
     0000000000000020  0000000000000000   A       0     0     4
[ 3] .note.gnu.build-i NOTE             0000000000400274  00000274
     0000000000000024  0000000000000000   A       0     0     4
[ 4] .gnu.hash         GNU_HASH         0000000000400298  00000298
     000000000000001c  0000000000000000   A       5     0     8
[ 5] .dynsym           DYNSYM           00000000004002b8  000002b8
     0000000000000048  0000000000000018   A       6     1     8
[ 6] .dynstr           STRTAB           0000000000400300  00000300
     0000000000000038  0000000000000000   A       0     0     1
[ 7] .gnu.version      VERSYM           0000000000400338  00000338
     0000000000000006  0000000000000002   A       5     0     2
[ 8] .gnu.version_r    VERNEED          0000000000400340  00000340
     0000000000000020  0000000000000000   A       6     1     8
[ 9] .rela.dyn         RELA             0000000000400360  00000360
     0000000000000018  0000000000000018   A       5     0     8
[10] .rela.plt         RELA             0000000000400378  00000378
     0000000000000018  0000000000000018  AI       5    24     8
[11] .init             PROGBITS         0000000000400390  00000390
     000000000000001a  0000000000000000  AX       0     0     4
[12] .plt              PROGBITS         00000000004003b0  000003b0
     0000000000000020  0000000000000010  AX       0     0     16
[13] .plt.got          PROGBITS         00000000004003d0  000003d0
     0000000000000008  0000000000000000  AX       0     0     8
[14] .text             PROGBITS         00000000004003e0  000003e0
     0000000000000192  0000000000000000  AX       0     0     16
[15] .fini             PROGBITS         0000000000400574  00000574
     0000000000000009  0000000000000000  AX       0     0     4
[16] .rodata           PROGBITS         0000000000400580  00000580
     0000000000000004  0000000000000004  AM       0     0     4
```

```
  [17] .eh_frame_hdr     PROGBITS         0000000000400584  00000584
       000000000000003c  0000000000000000   A       0     0     4
  [18] .eh_frame         PROGBITS         00000000004005c0  000005c0
       0000000000000114  0000000000000000   A       0     0     8
  [19] .init_array       INIT_ARRAY       0000000000600e10  00000e10
       0000000000000008  0000000000000000  WA       0     0     8
  [20] .fini_array       FINI_ARRAY       0000000000600e18  00000e18
       0000000000000008  0000000000000000  WA       0     0     8
  [21] .jcr              PROGBITS         0000000000600e20  00000e20
       0000000000000008  0000000000000000  WA       0     0     8
  [22] .dynamic          DYNAMIC          0000000000600e28  00000e28
       00000000000001d0  0000000000000010  WA       6     0     8
  [23] .got              PROGBITS         0000000000600ff8  00000ff8
       0000000000000008  0000000000000008  WA       0     0     8
  [24] .got.plt          PROGBITS         0000000000601000  00001000
       0000000000000020  0000000000000008  WA       0     0     8
  [25] .data             PROGBITS         0000000000601020  00001020
       0000000000000010  0000000000000000  WA       0     0     8
  [26] .bss              NOBITS           0000000000601030  00001030
       0000000000000008  0000000000000000  WA       0     0     1
  [27] .comment          PROGBITS         0000000000000000  00001030
       0000000000000034  0000000000000001  MS       0     0     1
  [28] .shstrtab         STRTAB           0000000000000000  000018b6
       000000000000010c  0000000000000000           0     0     1
  [29] .symtab           SYMTAB           0000000000000000  00001068
       0000000000000648  0000000000000018          30    47     8
  [30] .strtab           STRTAB           0000000000000000  000016b0
       0000000000000206  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

The first line:

```
There are 31 section headers, starting at offset 0x19c8
```

总结文件中总共有多少节，以及它开始的地址。然后，按节列出，以下为每个节的标题，也是每个节输出的格式：

Output

```
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
```

每个部分有两行，字段不同：

Nr 每个部分的索引。

Name 每个部分的名称。

Type 此字段（在部分标题中）标识每个部分的类型。类型用于对部分进行分类。

Address 每个部分的起始 *virtual* 地址。请注意，当程序在支持虚拟内存的操作系统上运行时，地址才是虚拟的。在我们的操作系统上，我们在裸机上运行，所有地址都将为物理地址。

Offset 这是一个以字节为单位的距离，从文件的第一个字节到对象的起始位置，例如在ELF二进制文件上下文中一个节或段的开头。

Size 每个部分的字节数大小。

EntSize 某些部分包含固定大小的条目表，例如符号表。对于此类部分，此成员给出每个条目的大小（以字节为单位）。如果该部分不包含固定大小的条目表，则此成员包含0。

Flags 描述一个部分的属性。标志与类型一起定义了部分的目的。两个部分可以是同一类型，但服务于不同的目的。例如，尽管 `.data` 和 `.text` 具有相同的类型，但 `.data` 存储程序的初始化数据，而

`.text` 程序持有可执行指令。因此，`.data` 被赋予读和写权限，但没有执行权限。任何在 `.data` 中执行代码的尝试都会被运行中的操作系统拒绝：在 Linux 中，这种无效部分的使用会导致一个 *segmentation fault*。

ELF提供信息以使具有此类保护机制的操作系统。然而，在裸金属上运行时，没有任何东西可以阻止做任何事情。我们的操作系统可以在数据部分执行代码，反之亦然，写入代码部分。

表5.3.1：部分标志

Flag	**Descriptions**
`W`	Bytes in this section are writable during execution.
`A`	Memory is allocated for this section during process execution. Some control sections do not reside in the memory image of an object file; this attribute is off for those sections.
`X`	The section contains executable instructions.
`M`	The data *in the section* may be merged to eliminate duplication. Each element in the section is compared against other elements in sections with the same name, type and flags. Elements that would have identical values at program run-time may be merged.
`S`	The data elements in the section consist of null-terminated character strings. The size of each character is specified in the section header's ***`EntSize`*** field.
`l`	Specific large section for x86_64 architecture. This flag is not specified in the Generic ABI but in x86_64 ABI.
`I`	The ***`Info`*** field of this section header holds an index of a section header. Otherwise, the number is the index of something else.
`L`	Preserve section ordering when linking. If this section is combined with other sections in the output file, it must appear in the same relative order with respect to those sections, as the linked-to section appears with respect to sections the linked-to section is combined with. Apply when the ***`Link`*** field of this section's header references another section (the linked-to section)
`G`	This section is a member (perhaps the only one) of a section group.
`T`	This section holds **T**hread-**L**ocal **S**torage, meaning that each thread has its own distinct instance of this data. A thread is a distinct execution flow of code. A program can have multiple threads that pack different pieces of code and execute separately, at the same time. We will learn more about threads when writing our kernel.

E	Link editor is to exclude this section from executable and shared library that it builds when those objects are not to be further relocated.
x	Unknown flag to `readelf`. It happens because the linking process can be done manually with a linker like `GNU ld` (we will later later). That is, section flags can be specified manually, and some flags are for a customized ELF that the open-source `readelf` doesn't know of.
O	This section requires special OS-specific processing (beyond the standard linking rules) to avoid incorrect behavior. A link editor encounters sections whose headers contain OS-specific values it does not recognize by Type or Flags values defined by ELF standard, the link editor should combine those sections.
o	All bits included in this flag are reserved for operating system-specific semantics.
p	All bits included in this flag are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

Link and Info 这些数字引用了章节、符号表条目、哈希表条目的索引。Link字段仅包含章节的索引，而Info字段包含章节、符号表条目或哈希表条目的索引，具体取决于章节的类型。在编写我们的操作系统时，我们将通过链接器脚本显式链接由gcc生成的目标文件来手工制作内核映像。我们将通过指定它们将在最终映像中出现的地址来指定章节的内存布局。但我们将不分配任何章节标志，让链接器来处理。然而，了解哪个标志做什么是有用的。

Align 这是一个强制段落偏移量可被该值整除的值。仅允许0和2的正整数次幂。值0和1表示该段落没有对齐约束。

示例5.3.1. `.interp`部分的输出：

Output

```
    [Nr] Name              Type             Address           Offset
         Size              EntSize          Flags  Link  Info  Align
    [ 1] .interp           PROGBITS         0000000000400238  00000238
         000000000000001c  0000000000000000   A       0     0     1
```

Nr 是 1。

Type 是 PROGBITS，这意味着本节是程序的一部分。

Address 是 0x0000000000400238，这意味着程序在运行时加载到这个虚拟内存地址。

Offset 是 0x00000238 *bytes* 到文件中。

Size 是 0x000000000000001c 字节。

EntSize 是 0，这意味着本节没有任何固定大小的条目。

Flags A (可分配)，这意味着本节在运行时消耗内存。

Info and Link 0 和 0，这意味着本节链接到任何表中的无章节或条目。

Align 是 1，这意味着没有对齐。

示例5.3.2. .text部分的输出：

Output

```
  [14] .text             PROGBITS         00000000004003e0  000003e0
       0000000000000192  0000000000000000  AX       0     0     16
```

Nr is 14.

Type is PROGBITS, which means this section is part of the program.

Address is 0x00000000004003e0, which means the program is loaded at this virtual memory address at runtime.

Offset is 0x000003e0 *bytes* into file.

Size is 0x0000000000000192 in bytes.

EntSize is 0, which means this section does not have any fixed-size entry.

Flags are A (Allocatable) and X (Executable), which means this section consumes memory and can be executed as code at runtime.

Info and Link are 0 and 0, which means this section links to no section or entry in any table.

Align is 16, which means the starting address of the section should be divisible by 16, or 0x10. Indeed, it is: 0x3e0/0x10 = 0x3e.

5.4 深入理解章节

在这个部分，我们将通过逐个查看每个部分来学习部分类型的各种细节以及特殊部分的目的，例如.bss，.text，.data等。我们还将使用以下命令将每个部分的内容作为十六进制转储进行检查：

```
$ readelf -x <section name|section number> <file>
```

例如，如果您想检查索引为25的章节（样本输出中的 .bss 章节）的内容，在文件 hello 中：

```
$ readelf -x 25 hello
```

等效地，使用名称代替索引可以工作：

```
$ readelf -x .data hello
```

如果某个部分包含字符串，例如字符串符号表，则可以将标志 -x 替换为 -p。

NULL 标记一个部分标题为非活动状态且没有关联的部分。NULL 部分始终是部分标题表的第一条记录。这意味着，任何有用的部分都是从1开始的。

示例5.4.1. NULL部分的样本输出：

Output

```
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000 00000000
       0000000000000000  0000000000000000           0     0     0
```

检查内容，该部分为空：

输出

```
Section " has no data to dump.
```

NOTE 标记一个包含特殊信息的部分，其他程序将通过供应商或系统构建商检查其一致性、兼容性等。

示例5.4.2。在样本输出中，我们有2 NOTE 部分：

Output

```
[Nr] Name              Type             Address           Offset
     Size              EntSize          Flags  Link  Info  Align
[ 2] .note.ABI-tag     NOTE             0000000000400254  00000254
     0000000000000020  0000000000000000   A       0     0     4
[ 3] .note.gnu.build-i NOTE             0000000000400274  00000274
     0000000000000024  0000000000000000   A       0     0     4
```

检查第2节，使用以下命令：

```
$ readelf -x 2 hello
```

我们有：

Output

```
Hex dump of section '.note.ABI-tag':
  0x00400254 04000000 10000000 01000000 474e5500 ............GNU.
  0x00400264 00000000 02000000 06000000 20000000 ............ ...
```

PROGBITS 指示包含程序主要内容的部分，ei-该代码或数据。

示例5.4.3。有许多PROGBITS部分：

Output

```
[Nr] Name              Type             Address           Offset
     Size              EntSize          Flags  Link  Info  Align
[ 1] .interp           PROGBITS         0000000000400238  00000238
     000000000000001c  0000000000000000   A       0     0     1
...
[11] .init             PROGBITS         0000000000400390  00000390
     000000000000001a  0000000000000000  AX       0     0     4
[12] .plt              PROGBITS         00000000004003b0  000003b0
     0000000000000020  0000000000000010  AX       0     0     16
[13] .plt.got          PROGBITS         00000000004003d0  000003d0
```

```
     0000000000000008  0000000000000000  AX       0     0     8
[14] .text             PROGBITS         00000000004003e0  000003e0
     0000000000000192  0000000000000000  AX       0     0     16
[15] .fini             PROGBITS         0000000000400574  00000574
     0000000000000009  0000000000000000  AX       0     0     4
[16] .rodata           PROGBITS         0000000000400580  00000580
     0000000000000004  0000000000000004  AM       0     0     4
[17] .eh_frame_hdr     PROGBITS         0000000000400584  00000584
     000000000000003c  0000000000000000   A       0     0     4
[18] .eh_frame         PROGBITS         00000000004005c0  000005c0
     0000000000000114  0000000000000000   A       0     0     8
...
[23] .got              PROGBITS         0000000000600ff8  00000ff8
     0000000000000008  0000000000000008  WA       0     0     8
[24] .got.plt          PROGBITS         0000000000601000  00001000
     0000000000000020  0000000000000008  WA       0     0     8
[25] .data             PROGBITS         0000000000601020  00001020
     0000000000000010  0000000000000000  WA       0     0     8
[27] .comment          PROGBITS         0000000000000000  00001030
     0000000000000034  0000000000000001  MS       0     0     1
```

F或我们的操作系统，我们只需要以下部分　　在：

.text 本节包含程序的所有编译代码。

.data 本节包含程序的初始化数据。由于数据使用实际值初始化，gcc 在可执行二进制文件中分配了实际字节的区域。

.rodata 本节包含只读数据，例如程序中的固定大小字符串，例如 “Hello World”，以及其他数据。

.bss 本节，简称 ***B****lock* ***S****tarted by* ***S****ymbol*，包含程序的未初始化数据。与其他节不同，磁盘上可执行二进制文件的映像中未为此节分配空间。该节仅在程序加载到主内存时分配。

其他部分主要用于动态链接，即在运行时进行代码链接以供多个程序共享。为了启用此功能，必须提供一个作为运行时环境的操作系统。由于我们在裸机上运行我们的操作系统，我们实际上正在创建这样的环境。为了简单起见，我们不会将动态链接添加到我们的操作系统。

SYMTAB and DYNSYM 这些部分包含符号表。一个 *symbol table* 是描述程序中符号的条目数组。一个 *symbol* 是分配给程序中实体的名称。这些实体的类型也是符号的类型，这些是实体的可能类型：

示例5.4.4。在样本输出中，第5节和第29节是符号表：

Output

```
[Nr] Name              Type             Address           Offset
     Size              EntSize          Flags  Link  Info  Align
[ 5] .dynsym           DYNSYM           00000000004002b8  000002b8
     0000000000000048  0000000000000018   A       6     1     8
...
[29] .symtab           SYMTAB           0000000000000000  00001068
     0000000000000648  0000000000000018          30    47     8
```

要显示符号表：

```
$ readelf -s hello
```

输出包含2个符号表，对应于两个部分

以上，`.dynsym` 和 `.symtab`：

Output

```
Symbol table '.dynsym' contains 4 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
```

```
Symbol table '.symtab' contains 67 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
    .....................................
    59: 0000000000601040     0 NOTYPE  GLOBAL DEFAULT   26 _end
    60: 0000000000400430    42 FUNC    GLOBAL DEFAULT   14 _start
    61: 0000000000601038     0 NOTYPE  GLOBAL DEFAULT   26 __bss_start
    62: 0000000000400526    32 FUNC    GLOBAL DEFAULT   14 main
    63: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
    64: 0000000000601038     0 OBJECT  GLOBAL HIDDEN    25 __TMC_END__
    65: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
    66: 00000000004003c8     0 FUNC    GLOBAL DEFAULT   11 _init
```

TLS The 符号与一个线程局部存储实体相关联。

Num 是表中条目索引。

Value 这是符号所在的虚拟内存地址。

Size 是符号关联的实体的大小。

Type 这是一个根据表格的符号类型。

NOTYPE 符号的类型未指定。

OBJECT The 符号与数据对象相关联。在C语言中，任何变量定义都是OBJECT类型。

FUNC 该符号与一个函数或其他可执行代码相关联。

SECTION The 符号与一个部分相关联，主要存在用于重定位。

FILE 该符号是与其可执行二进制文件关联的源文件的名称。

COMMON The 符号表示一个未初始化的变量。也就是说，当C语言中的变量被定义为没有初始值的全局变量，或者使用extern关键字定义的外部变量。换句话说，这些变量保持在.bss部分。

Bind 符号的作用范围。

LOCAL 这些符号仅在定义它们的对象文件中可见。在C语言中，static修饰符将符号（例如变量/函数）标记为仅限于定义它的文件。

示例5.4.5。如果我们使用static修饰符定义变量和函数：

hello.c

```
static int global_static_var = 0;

static void local_func() {
}

int main(int argc, char *argv[])
{
    static int local_static_var = 0;

    return 0;
}
```

然后我们在编译后得到以下 static 变量作为局部符号列出：

```
$ gcc -m32 hello.c -o hello
$ readelf -s hello
```

Output

```
   Symbol table '.dynsym' contains 5 entries:
      Num:    Value  Size Type    Bind   Vis      Ndx Name
        0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
        1: 00000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.0 (2)
        2: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
        3: 00000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.0 (2)
        4: 080484bc     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used
   Symbol table '.symtab' contains 72 entries:
      Num:    Value  Size Type    Bind   Vis      Ndx Name
        0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND

           ......... output omitted .........

       38: 0804a020     4 OBJECT  LOCAL  DEFAULT   26 global_static_var
       39: 0804840b     6 FUNC    LOCAL  DEFAULT   14 local_func
```

```
    40: 0804a024     4 OBJECT  LOCAL  DEFAULT   26 local_static_var.1938
..... output omitted .....
```

GLOBAL *are* 符号在链接时可以被其他对象文件访问。这些符号主要是非-static函数和非-static全局数据。extern修饰符将符号标记为在其他地方外部定义但可在最终可执行二进制文件中访问，因此一个extern变量也被视为GLOBAL。

示例5.4.6。类似于上面的 LOCAL 示例，输出列出了许多 GLOBAL 符号，例如 main:

```
  Num:    Value  Size Type    Bind   Vis      Ndx Name
..... output omitted .....
 66: 080483e1    10 FUNC    GLOBAL DEFAULT   14 main
..... output omitted .....
```

WEAK 符号，其定义可以被重新定义。通常，具有多个定义的符号会被编译器报告为错误。然而，当定义被显式标记为弱定义时，这个约束就放宽了，这意味着默认实现可以在链接时被不同的定义所替换。

示例5.4.7。假设我们有一个函数 add 的默认实现:

hello.c

```
#include <stdio.h>

__attribute__((weak)) int add(int a, int b) {
    printf("warning: function is not implemented.\n")
        ;
    return 0;
}

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

```
    printf("add(1,2) is %d\n", add(1,2));
    return 0;
}
```

`__attribute__((weak))` 是一个函数属性。一个 *function attribute* 是编译器处理函数与普通函数不同的额外信息。在这个例子中，`weak`属性使函数`add`成为弱函数，这意味着默认实现可以在链接时被不同的定义替换。函数属性是编译器的特性，不是标准C。

function attribute

如果我们在不同的文件中不提供不同的函数定义（必须在不同的文件中，否则 `gcc` 会报告错误），则应用默认实现。当调用函数 `add` 时，它只打印消息：`"warning: function not implemented"` 并返回 0：

```
$ ./hello
warning: function is not implemented.
add(1,2) is 0
```

然而，如果我们向另一个文件提供不同的定义，例如 `math.c`：

math.c

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

将两个文件合并编译：

```
$ gcc math.c hello.c -o hello
```

然后，当运行 `hello` 时，不会打印警告消息，并返回正确值。

弱符号是一种提供默认实现机制的机制，但在链接时如果可用更好的实现（例如更专业和优化的实现）则可替换。

Vis 符号的可视性。以下值可用：

表5.4.1：符号可见性

Value	Description
DEFAULT	The visibility is specified by the binding type of asymbol. ▷ Global and weak symbols are visible outside of their defining component (executable file or shared object). ▷ Local symbols are hidden. See HIDDEN below.
HIDDEN	A symbol is hidden when the name is not visible to any other program outside of its running program.
PROTECTED	A symbol is protected when it is shared outside of its running program or shared libary and cannot be overridden. That is, there can only be one definition for this symbol across running programs that use it. No program can define its own definition of the same symbol.
INTERNAL	Visibility is processor-specific and is defined by processor-specific ABI.

Ndx 是符号所在的节区的索引。除了表示节区索引的固定索引数字外
，索引还具有以下特殊值：

表5.4.2：符号索引

Value	Description
ABS	The index will not be changed by any symbol relocation.
COM	The index refers to an unallocated common block.
UND	The symbol is undefined in the current object file, which means the symbol depends on the actual definition in another file. Undefined symbols appears when the object file refers to symbols that are available at runtime, from shared library.
LORESERVE HIRESERVE	LORESERVE is the lower boundary of the reserve indexes. Its value is 0xff00. HIREVERSE is the upper boundary of the reserve indexes. Its value is 0xffff. The operating system reserves exclusive indexes between LORESERVE and HIRESERVE, which do not map to any actual section header.
XINDEX	The index is larger than LORESERVE. The actual value will be contained in the section SYMTAB_SHNDX, where each entry is a mapping between a symbol, whose Ndx field is a XINDEX value, and the actual index value.

Others	Sometimes, values such as `ANSI_COM`, `LARGE_COM`, `SCOM`, `SUND` appear. This means that the index is processor-specific.

Name 是符号名称。

示例5.4.8。一个C应用程序程序总是从符号 `main` 开始。在 `.symtab` 部分的符号表中，`main` 的条目是：

Output

```
    Num:                Value  Size Type    Bind   Vis      Ndx Name
     62:     0000000000400526    32 FUNC    GLOBAL DEFAULT   14 main
```

该条目显示：

▷ `main` 是表中的62th条目。▷ `main`从地址0x0000000000400526开始。▷ `main`占用32字节。▷ `main`是一个函数。▷ `main`在全局范围内。▷ `main`对使用它的其他对象文件可见。▷ `main`在14th部分中，它是`.text`。这是逻辑的，因

ce

`.text` 包含所有程序代码。

STRTAB 维护一个以空字符终止的字符串表，称为 *string table*。该部分的第一个和最后一个字节始终是空字符。存在字符串表部分，因为字符串可以被多个部分重用以表示符号和部分名称，因此像 `readelf` 或 `objdump` 这样的程序可以在人类可读的文本中显示程序中的各种对象，例如变量、函数、部分名称，而不是其原始十六进制地址。

示例5.4.9。在示例输出中，部分28和30是STRTAB类型：

Output

```
[Nr] Name              Type             Address           Offset
     Size              EntSize          Flags  Link  Info  Align
[28] .shstrtab         STRTAB           0000000000000000  000018b6
     000000000000010c  0000000000000000           0     0     1
[30] .strtab           STRTAB           0000000000000000  000016b0
     0000000000000206  0000000000000000           0     0     1
```

.shstrtab 包含所有章节名称。

.strtab 包含符号，例如变量名、函数名、结构体名等，在C程序中，但不包含固定大小的空终止C字符串；C字符串保存在.rodata部分。

示例 5.4.10. 该部分中的字符串可以使用以下命令进行检查：

```
$ readelf -p 29 hello
```

输出显示了所有部分名称，以及到 .shstrtab 表的偏移量（也是字符串索引）到左侧：

Output

```
String dump of section '.shstrtab':
  [     1]  .symtab
  [     9]  .strtab
  [    11]  .shstrtab
  [    1b]  .interp
  [    23]  .note.ABI-tag
  [    31]  .note.gnu.build-id
  [    44]  .gnu.hash
  [    4e]  .dynsym
  [    56]  .dynstr
  [    5e]  .gnu.version
  [    6b]  .gnu.version_r
  [    7a]  .rela.dyn
  [    84]  .rela.plt
```

```
[    8e]  .init
[    94]  .plt.got
[    9d]  .text
[    a3]  .fini
[    a9]  .rodata
[    b1]  .eh_frame_hdr
[    bf]  .eh_frame
[    c9]  .init_array
[    d5]  .fini_array
[    e1]  .jcr
[    e6]  .dynamic
[    ef]  .got.plt
[    f8]  .data
[    fe]  .bss
[   103]  .comment
```

字符串表的实际实现是一个连续的以空字符结尾的字符串数组。字符串的索引是其第一个字符在数组中的位置。例如，在上面的字符串表中，.symtab在数组中的索引是1（空字符位于索引0）。.symtab的长度是7，加上空字符，它在8个字节的位置。

tal. 因此，.strtab 从索引 9 开始，以此类推。

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00000000	\0	.	s	y	m	t	a	b	\0	.	s	t	r	t	a	b

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00000010	\0	.	s	h	s	t	r	t	a	b	\0	.	i	n	t	e

.... and so on ....

图5.4.1：.shstrtab.内存中的字符串表。红色数字是字符串的起始索引。

同样，.strtab的输出：

Output

```
String dump of section '.strtab':
  [     1]  crtstuff.c
  [     c]  __JCR_LIST__
  [    19]  deregister_tm_clones
```

```
[    2e]  __do_global_dtors_aux
[    44]  completed.7585
[    53]  __do_global_dtors_aux_fini_array_entry
[    7a]  frame_dummy
[    86]  __frame_dummy_init_array_entry
[    a5]  hello.c
[    ad]  __FRAME_END__
[    bb]  __JCR_END__
[    c7]  __init_array_end
[    d8]  _DYNAMIC
[    e1]  __init_array_start
[    f4]  __GNU_EH_FRAME_HDR
[   107]  _GLOBAL_OFFSET_TABLE_
[   11d]  __libc_csu_fini
[   12d]  _ITM_deregisterTMCloneTable
[   149]  j
[   14b]  _edata
[   152]  __libc_start_main@@GLIBC_2.2.5
[   171]  __data_start
[   17e]  __gmon_start__
[   18d]  __dso_handle
[   19a]  _IO_stdin_used
[   1a9]  __libc_csu_init
[   1b9]  __bss_start
[   1c5]  main
[   1ca]  _Jv_RegisterClasses
[   1de]  __TMC_END__
[   1ea]  _ITM_registerTMCloneTable
```

HASH 包含一个符号哈希表，该表支持符号表访问。

DYNAMIC 包含动态链接信息。

NOBITS 与 PROGBITS 相似，但占据的空间为零。

Example 5.4.11. `.bss` section holds uninitialized data, which means

节区的字节可以具有任何值。直到操作系统实际将节区加载到主内存中，没有必要在磁盘上为二进制映像分配空间以减小二进制文件的大小。以下是示例输出的.bss的详细信息：

Output

```
[Nr] Name              Type             Address           Offset
     Size              EntSize          Flags  Link  Info  Align
[26] .bss              NOBITS           0000000000601038  00001038
     0000000000000008  0000000000000000  WA       0     0     1
[27] .comment          PROGBITS         0000000000000000  00001038
     0000000000000034  0000000000000001  MS       0     0     1
```

在上述输出中，该部分的大小仅为8字节，而两个部分的偏移量相同，这意味着 .bss 在磁盘上的可执行二进制文件中不消耗任何字节。

注意，.comment部分没有起始地址。这意味着当可执行二进制文件加载到内存中时，此部分将被丢弃。

REL 包含没有显式加数的重定位条目。此类将在8.1中详细解释。

RELA 包含具有显式加数的重定位条目。此类将在8.1中详细解释。

INIT_ARRAY 这是一个程序初始化的函数指针数组。当应用程序运行时，在到达 main() 之前，首先执行 .init 和本节中的初始化代码。此数组中的第一个元素是一个被忽略的函数指针。

它可能在我们可以在 main() 函数中包含初始化代码时不合逻辑。然而，对于没有 main() 的共享对象文件，本节确保从对象文件执行的初始化代码在任何其他代码之前执行，以确保主代码能够正常运行的环境。这也使对象文件更具模块化，因为主应用程序代码不需要负责为使用特定对象文件初始化适当的环境，而是由对象文件本身负责。这种清晰的划分使代码更简洁。

然而，为了简单起见，我们不会在我们的操作系统中使用任何 .init 和 INIT_ARRAY 部分，因为初始化环境是操作系统领域的一部分。

示例5.4.12. 要使用INIT_ARRAY，我们只需将一个函数标记为具有属性constructor：

hello.c

```c
#include <stdio.h>

__attribute__((constructor)) static void init1(){
    printf("%s\n", __FUNCTION__);
}

__attribute__((constructor)) static void init2(){
    printf("%s\n", __FUNCTION__);
}


int main(int argc, char *argv[])
{
    printf("hello␣world\n");

    return 0;
}
```

程序自动调用构造函数，而不显式调用它：

```
$ gcc -m32 hello.c -o hello
$ ./hello
init1
init2
hello world
```

示例5.4.13。可选地，可以从101开始为构造函数分配优先级。0到100的优先级已被保留

对于 gcc。如果我们想让 init2 在 init1 之前运行，我们给它更高的优先级：

hello.c

```
#include <stdio.h>

__attribute__((constructor(102))) static void init1(){
    printf("%s\n", __FUNCTION__);
}

__attribute__((constructor(101))) static void init2(){
    printf("%s\n", __FUNCTION__);
}



int main(int argc, char *argv[])
{
    printf("hello␣world\n");

    return 0;
}
```

调用顺序应完全按照指定进行：

```
$ gcc -m32 hello.c -o hello
$ ./hello
init2
init1
hello world
```

示例 5.4.14。我们可以使用另一种方法添加初始化函数：

hello.c

```
#include <stdio.h>
```

```
void init1() {
    printf("%s\n", __FUNCTION__);
}

void init2() {
    printf("%s\n", __FUNCTION__);
}

/* Without typedef, init is a definition of a function
    pointer.
   With typedef, init is a declaration of a type.*/
typedef void (*init)();

__attribute__((section(".init_array"))) init init_arr[2]
    = {init1, init2};

int main(int argc, char *argv[])
{
    printf("hello␣world!\n");

    return 0;
}
```

属性 `section("...")` 将函数放入特定部分而不是默认的 `.text`。在这个例子中，它是 `.init_array`。部分名称不一定与 ELF 文件中的标准标题（如 `.text` 或 `.init_array`）相同，可以是任何名称。非标准部分名称通常用于控制编译程序的最终二进制布局。当学习 `GNU ld` 链接器和链接过程时，我们将更详细地探讨这种技术。再次强调，程序会自动调用构造函数，而无需显式调用它：

```
$ gcc -m32 hello.c -o hello
$ ./hello
init1
init2
hello world!
```

FINI_ARRAY 是一个程序终止的函数指针数组，在退出 `main()` 之后调用。如果应用程序异常终止，例如通过 `abort()` 调用或崩溃，则忽略 `.finit_array`。

示例5.4.15。如果存在一个或多个可用的析构函数，则在退出main()后自动调用析构函数:

hello.c

```c
#include <stdio.h>


__attribute__((destructor)) static void destructor(){
    printf("%s\n", __FUNCTION__);
}


int main(int argc, char *argv[])
{
    printf("hello world\n");


    return 0;
}
```

```
$ gcc -m32 hello.c -o hello
$ ./hello
hello world
destructor
```

PREINIT_ARRAY 是一个在 `INIT_ARRAY` 中所有其他初始化函数之前调用的函数指针数组。

示例5.4.16. 要使用.preinit_array，将函数放入本节的唯一方法是使用属性section()：

hello.c

```
#include <stdio.h>

void preinit1() {
    printf("%s\n", __FUNCTION__);
}

void preinit2() {
    printf("%s\n", __FUNCTION__);
}

void init1() {
    printf("%s\n", __FUNCTION__);
}

void init2() {
    printf("%s\n", __FUNCTION__);
}



typedef void (*preinit)();
typedef void (*init)();

__attribute__((section(".preinit_array"))) preinit
    preinit_arr[2] = {preinit1, preinit2};
__attribute__((section(".preinit_array"))) init init_arr
    [2] = {init1, init2};

int main(int argc, char *argv[])
{
    printf("hello␣world!\n");
```

```
    return 0;
}
```

```
$ gcc -m32 hello2.c -o hello2
$ ./hello2
preinit1
preinit2
init1
init2
hello world!
```

GROUP 定义一个节组，该节组在不同对象文件中显示相同，但在合并到最终可执行二进制文件时，只保留一个副本，其余在其它对象文件中的都被丢弃。此节仅在C++对象文件中相关，因此我们不再进一步检查。

SYMTAB_SHNDX 这是一个包含扩展部分索引的节，这些索引与符号表相关联。当符号表中的条目 Ndx 值超过 LORESERVE 值时，此节才会出现。然后，该节将符号与部分头部的实际索引值之间进行映射。

在理解了节类型之后，我们可以理解 Link 和 Info 字段中的数字：

练习5.4.1. 验证 SYMTAB 节的 Link 字段值是 STRTAB 节的索引。

练习5.4.2.验证SYMTAB节中Info字段的值是最后一个局部符号+的索引1。这意味着，在符号表中，从Info字段列出的索引开始，不再出现局部符号。

练习5.4.3.验证REL节中Info字段的值是SYMTAB节的索引。

练习5.4.4.验证REL节中Link字段的值是应用重定位的节索引。例如，如果节是.rel.text，则重定位节应该是.text。

Type	Link	Info
DYNAMIC	Entries in this section uses the section index of the dynamic string table.	0
HASH GNU_HASH	The section index of the symbol table to which the hash table applies.	0
REL RELA	The section index of the associated symbol table.	The section index to which the relocation applies.
SYMTAB DYNSYM	The section index of the associated string table.	One greater than the symbol table index of the last local symbol.
GROUP	The section index of the associated symbol table.	The symbol index of an entry in the associated symbol table. The name of the specified symbol table entry provides a signature for the section group.
SYMTAB_SHNDX	The section header index of the associated symbol table.	

表5.4.3: *Link*和*Info* depend on的解释含义section types.

5.5 程序头表

一个 *program header table* 是定义程序在运行时内存布局的程序头数组。

一个 *program header* 是程序段的描述。

一个 *program segment* 是相关部分的集合。一个段包含零个或多个部分。操作系统在加载程序 *only use segments* 时，不是部分。要查看程序头表的信息，我们使用 `-l` 选项与 `readelf:`。

```
$ readelf -l <binary file>
```

类似于一个部分，程序头也有类型:

PHDR 指定程序头表本身的位置和大小，包括文件和程序内存映像中

INTERP 指定调用作为链接运行时库的解释器的空终止路径名位置和大小。

LOAD 指定一个可加载段。也就是说，此段被加载到主内存中。

DYNAMIC 指定动态链接信息。

NOTE 指定辅助信息的位置和大小。

TLS 指定了 *Thread-Local Storage template*，它是由所有带有标志 TLS 的部分组合而成的。

GNU_STACK 指示程序堆栈是否应可执行。Linux内核使用此类型。

表5.5.1：段权限

Permission	Description
R	Readable
W	Writable
E	Executable

一个段也有权限，它是这三个值的组合：

▷ 读取(R) ▷
写入(W) ▷ 执行
(E)

Example 5.5.1 获取程序头部的命令 r 表格：

```
$ readelf -l hello
```

输出：

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x400430
There are 9 program headers, starting at offset 64
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R E    8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x000000000000070c 0x000000000000070c  R E    200000
```

```
  LOAD           0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
                 0x0000000000000228 0x0000000000000230  RW     200000
  DYNAMIC        0x0000000000000e28 0x0000000000600e28 0x0000000000600e28
                 0x00000000000001d0 0x00000000000001d0  RW     8
  NOTE           0x0000000000000254 0x0000000000400254 0x0000000000400254
                 0x0000000000000044 0x0000000000000044  R      4
  GNU_EH_FRAME   0x00000000000005e4 0x00000000004005e4 0x00000000004005e4
                 0x0000000000000034 0x0000000000000034  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
                 0x00000000000001f0 0x00000000000001f0  R      1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr
.gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .plt.got .text .fini
.rodata .eh_frame_hdr .eh_frame
   03     .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .jcr .dynamic .got
```

在样本输出中，LOAD段出现两次：

Output

```
LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
               0x000000000000070c 0x000000000000070c  R E    200000
LOAD           0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
               0x0000000000000228 0x0000000000000230  RW     200000
```

为什么？注意权限：

▷ 上级 LOAD 具有读取和执行权限。这是一个 *text* seg-

文本段包含只读指令和只读数据。

▷ 下级 LOAD 具有读写权限。这是一个 *data* 段。这意味着这个段可以被读取和写入，但不允许用作可执行代码，出于安全原因。

然后，LOAD 包含以下部分：

Output

```
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr
.gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .plt.got .text .fini
.rodata .eh_frame_hdr .eh_frame
   03     .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
```

第一个数字是程序头表中程序头的索引，其余文本是段内所有节的列表。不幸的是，readelf不打印索引，因此用户需要手动跟踪哪个段对应哪个索引。第一个段从索引0开始，第二个段从索引1开始，依此类推。LOAD是索引2和3的段。从两个节列表中可以看出，大多数节都是可加载的，并在运行时可用。

5.6 段落与部分

如前所述，操作系统加载的是程序段，而不是节。然而，一个问题产生了：为什么操作系统不使用节呢？毕竟，节也包含与程序段类似的信息，例如类型、要加载的虚拟内存地址、大小、属性、标志和对齐。正如之前解释的那样，段是操作系统的视角，而节是链接器的视角。为了理解这一点，查看段的结构，我们可以很容易地看到：

▷ 一个段是部分集合。这意味着部分根据它们的属性逻辑分组。例如，所有部分

在一个 LOAD 段中总是由操作系统加载；所有部分具有相同的权限，无论是可执行部分的 RE (读取 +执行)，还是数据部分的 RW (读取 + 写入)。

▷ 通过将部分组合成一个段，操作系统可以更容易地一次性批量加载部分，只需加载段的起始和结束，而不是逐个加载部分。

▷ 由于一个段用于加载程序，一个节用于链接程序，因此一个段中的所有节是 *within its start and end virtual memory addresses of a segment*。

为了更清楚地看到最后一点，考虑一个链接两个目标文件的例子。假设我们有两个源文件：

hello.c

```c
#include <stdio.h>


int main(int argc, char *argv[])
{
    printf("Hello World\n");
    return 0;
}
```

并且：

math.c

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

现在，将两个源文件编译为 *object files*：

```
$ gcc -m32 -c math.c
$ gcc -m32 -c hello.c
```

然后，我们检查 math.o 的部分：

```
$ readelf -S math.o
```

Output

```
There are 11 section headers, starting at offset 0x1a8:
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 00000d 00  AX  0   0  1
  [ 2] .data             PROGBITS        00000000 000041 000000 00  WA  0   0  1
  [ 3] .bss              NOBITS          00000000 000041 000000 00  WA  0   0  1
  [ 4] .comment          PROGBITS        00000000 000041 000035 01  MS  0   0  1
  [ 5] .note.GNU-stack   PROGBITS        00000000 000076 000000 00      0   0  1
  [ 6] .eh_frame         PROGBITS        00000000 000078 000038 00   A  0   0  4
  [ 7] .rel.eh_frame     REL             00000000 00014c 000008 08   I  9   6  4
  [ 8] .shstrtab         STRTAB          00000000 000154 000053 00      0   0  1
  [ 9] .symtab           SYMTAB          00000000 0000b0 000090 10     10   8  4
  [10] .strtab           STRTAB          00000000 000140 00000c 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

如输出所示，每个段的虚拟内存地址都设置为0。在此阶段，每个对象文件只是一个包含代码和数据的*block of binary*。它的存在是为了作为最终产品的材料容器，即可执行二进制文件。因此，`hello.o`中的虚拟地址都是零。

此阶段不存在任何段：

```
$ readelf -l math.o
There are no program headers in this file.
```

The same happens to other object file:

Output

```
There are 13 section headers, starting at offset 0x224:
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 00002e 00  AX  0   0  1
  [ 2] .rel.text         REL             00000000 0001ac 000010 08   I 11   1  4
  [ 3] .data             PROGBITS        00000000 000062 000000 00  WA  0   0  1
  [ 4] .bss              NOBITS          00000000 000062 000000 00  WA  0   0  1
  [ 5] .rodata           PROGBITS        00000000 000062 00000c 00   A  0   0  1
  [ 6] .comment          PROGBITS        00000000 00006e 000035 01  MS  0   0  1
  [ 7] .note.GNU-stack   PROGBITS        00000000 0000a3 000000 00      0   0  1
  [ 8] .eh_frame         PROGBITS        00000000 0000a4 000044 00   A  0   0  4
  [ 9] .rel.eh_frame     REL             00000000 0001bc 000008 08   I 11   8  4
  [10] .shstrtab         STRTAB          00000000 0001c4 00005f 00      0   0  1
  [11] .symtab           SYMTAB          00000000 0000e8 0000b0 10     12   9  4
  [12] .strtab           STRTAB          00000000 000198 000013 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

```
$ readelf -l hello.o
There are no program headers in this file.
```

仅当目标文件组合成最终的可执行二进制文件时，节才被完全实现：

```
$ gcc -m32 math.o hello.o -o hello
$ readelf -S hello.
```

Output

```
There are 31 section headers, starting at offset 0x1804:
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
```

```
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048154 000154 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048168 000168 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            08048188 000188 000024 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH        080481ac 0001ac 000020 04   A  5   0  4
  [ 5] .dynsym           DYNSYM          080481cc 0001cc 000050 10   A  6   1  4
  [ 6] .dynstr           STRTAB          0804821c 00021c 00004a 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          08048266 000266 00000a 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         08048270 000270 000020 00   A  6   1  4
  [ 9] .rel.dyn          REL             08048290 000290 000008 08   A  5   0  4
  [10] .rel.plt          REL             08048298 000298 000010 08  AI  5  24  4
  [11] .init             PROGBITS        080482a8 0002a8 000023 00  AX  0   0  4
  [12] .plt              PROGBITS        080482d0 0002d0 000030 04  AX  0   0 16
  [13] .plt.got          PROGBITS        08048300 000300 000008 00  AX  0   0  8
  [14] .text             PROGBITS        08048310 000310 0001a2 00  AX  0   0 16
  [15] .fini             PROGBITS        080484b4 0004b4 000014 00  AX  0   0  4
  [16] .rodata           PROGBITS        080484c8 0004c8 000014 00   A  0   0  4
  [17] .eh_frame_hdr     PROGBITS        080484dc 0004dc 000034 00   A  0   0  4
  [18] .eh_frame         PROGBITS        08048510 000510 0000ec 00   A  0   0  4
  [19] .init_array       INIT_ARRAY      08049f08 000f08 000004 00  WA  0   0  4
  [20] .fini_array       FINI_ARRAY      08049f0c 000f0c 000004 00  WA  0   0  4
  [21] .jcr              PROGBITS        08049f10 000f10 000004 00  WA  0   0  4
  [22] .dynamic          DYNAMIC         08049f14 000f14 0000e8 08  WA  6   0  4
  [23] .got              PROGBITS        08049ffc 000ffc 000004 04  WA  0   0  4
  [24] .got.plt          PROGBITS        0804a000 001000 000014 04  WA  0   0  4
  [25] .data             PROGBITS        0804a014 001014 000008 00  WA  0   0  4
  [26] .bss              NOBITS          0804a01c 00101c 000004 00  WA  0   0  1
  [27] .comment          PROGBITS        00000000 00101c 000034 01  MS  0   0  1
  [28] .shstrtab         STRTAB          00000000 0016f8 00010a 00      0   0  1
  [29] .symtab           SYMTAB          00000000 001050 000470 10     30  48  4
  [30] .strtab           STRTAB          00000000 0014c0 000238 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
```

```
O (extra OS processing required) o (OS specific), p (processor specific)
```

每个可加载部分都被分配了一个地址，用绿色突出显示。每个部分获得自己的地址的原因是，在现实中，gcc *does not combine an object by itself, but invokes the linker* ld。链接器 ld 使用它在系统中找到的默认脚本来构建可执行二进制文件。在默认脚本中，将一个段分配一个起始地址 0x8048000，并且部分属于它。然后：

▷ 1^{st} section address = starting segment address + section offset = 0x8048000 + 0x154 = 0x08048154

▷ 2^{nd} section address = starting segment address + section offset = 0x8048000 + 0x168 = 0x08048168

▷ 等等，直到最后一个可加载部分。

确实，段落的结束地址也是最终段的结束地址。我们可以通过列出所有段来看到这一点：

```
$ readelf -l hello
```

例如检查从 0x08048000 开始到 0x08048000 + 0x005fc = 0x080485fc 结束的 LOAD 段：

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x8048310
There are 9 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00120 0x00120 R E 0x4
  INTERP         0x000154 0x08048154 0x08048154 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x005fc 0x005fc R E 0x1000
  LOAD           0x000f08 0x08049f08 0x08049f08 0x00114 0x00118 RW  0x1000
  DYNAMIC        0x000f14 0x08049f14 0x08049f14 0x000e8 0x000e8 RW  0x4
  NOTE           0x000168 0x08048168 0x08048168 0x00044 0x00044 R   0x4
  GNU_EH_FRAME   0x0004dc 0x080484dc 0x080484dc 0x00034 0x00034 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
```

```
  GNU_RELRO      0x000f08 0x08049f08 0x08049f08 0x000f8 0x000f8 R   0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr
.gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .plt.got .text .fini
.rodata .eh_frame_hdr .eh_frame
   03     .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .jcr .dynamic .got
```

最后一段在第一个 LOAD 段中是 .eh_frame。.eh_frame 段从 0x0804851 开始，因为起始地址是 0x08048000，文件中的偏移量是 0x510。.eh_frame 的结束地址应该是：0x08048000 + 0x510 + 0xec = 0x080485fc，因为段大小是 0xec。这正好与上面第一个 LOAD 段的结束地址相同：0x08048000 + 0x5ec = 0x080485fc。

第八章将详细探讨整个过程。

6

Runtime inspection and debug

一个 *debugger* 是一个允许检查正在运行的程序的程序。 ***debugger***

调试器可以启动并运行程序，然后停止在特定行以检查程序在该点的状态。调试器停止（但未挂起）的点称为 *breakpoint*。

我们将使用 ***GDB - GNU Debugger*** 来调试我们的内核。`gdb` 是程序名称。`gdb` 可以做四种主要的事情:

- 启动您的程序，指定可能影响其行为的任何内容。
- 使您的程序在指定条件下停止。
- 检查程序停止时发生了什么
- 更改您程序中的内容，以便您可以尝试纠正一个错误的影响，并继续了解另一个错误

6.1 一个示例程序

必须存在一个用于调试的现有程序。经典的“Hello World”程序在本章的教育目的中就足够了:

`hello.c`

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello World!\n");
    return 0;
}
```

我们使用带有调试信息的选项 -g 编译它：

```
$ gcc -m32 -g hello.c -o hello
```

最后，我们以程序作为参数启动gdb：

```
$ gdb hello
```

6.2 程序静态检查

在运行时检查程序之前，gdb 首先加载它。在加载到内存中（但未运行）时，可以检索大量有用的信息以供检查。本节中的命令可以在程序运行之前使用。然而，它们在程序运行时也可以使用，并且可以显示更多信息。

6.2.1 Command: info target/info file/info files

此命令打印调试目标的信息。一个 *target* 是调试程序。

示例6.2.1. 从hello程序中命令的输出，一个本地目标的详细情况：

```
(gdb) info target
```

Output

```
Symbols from "/tmp/hello".
Local exec file:
‘/tmp/hello’, file type elf32-i386.
Entry point: 0x8048310
0x08048154 - 0x08048167 is .interp
0x08048168 - 0x08048188 is .note.ABI-tag
0x08048188 - 0x080481ac is .note.gnu.build-id
0x080481ac - 0x080481cc is .gnu.hash
0x080481cc - 0x0804821c is .dynsym
0x0804821c - 0x08048266 is .dynstr
0x08048266 - 0x08048270 is .gnu.version
0x08048270 - 0x08048290 is .gnu.version_r
0x08048290 - 0x08048298 is .rel.dyn
0x08048298 - 0x080482a8 is .rel.plt
0x080482a8 - 0x080482cb is .init
0x080482d0 - 0x08048300 is .plt
0x08048300 - 0x08048308 is .plt.got
0x08048310 - 0x080484a2 is .text
0x080484a4 - 0x080484b8 is .fini
0x080484b8 - 0x080484cd is .rodata
0x080484d0 - 0x080484fc is .eh_frame_hdr
0x080484fc - 0x080485c8 is .eh_frame
0x08049f08 - 0x08049f0c is .init_array
0x08049f0c - 0x08049f10 is .fini_array
0x08049f10 - 0x08049f14 is .jcr
0x08049f14 - 0x08049ffc is .dynamic
0x08049ffc - 0x0804a000 is .got
0x0804a000 - 0x0804a014 is .got.plt
0x0804a014 - 0x0804a01c is .data
0x0804a01c - 0x0804a020 is .bss
```

显示的输出报告:

▷ Path of a symbol file. A *symbol file* is the file that contains the debugging information. Usually, this is the same file as the binary, but it is

通常将可执行二进制文件和其调试信息分开到2个文件中，特别是用于远程调试。例如，就是这一行：

```
Symbols from "/tmp/hello".
```

▷ 调试程序的路径及其文件类型。在示例中，是这一行：

```
Local exec file:
‘/tmp/hello’, file type elf32-i386.
```

▷ 调试程序的入口点。也就是说，程序运行的第一个代码。在示例中，它是这一行：

```
Entry point: 0x8048310
```

▷ 一个包含起始和结束地址的章节列表。例如，它是剩余的输出。

示例6.2.2。如果调试程序在不同的机器上运行，它是一个远程目标，gdb只打印简要信息：

```
(gdb) info target
```

Output

```
Remote serial target in gdb-specific protocol:
Debugging a target over a serial line.
```

6.2.2 Command: maint info sections

此命令类似于 info target，但提供了关于程序部分的额外信息，特别是每个段的文件偏移量和标志。

示例 6.2.3。以下是针对 hello 程序运行时的输出：

```
(gdb) maint info sections
```

Output

```
Exec file:
    '/tmp/hello', file type elf64-x86-64.
 [0]     0x00400238->0x00400254 at 0x00000238: .interp ALLOC LOAD READONLY DATA HAS_CONTENTS
 [1]     0x00400254->0x00400274 at 0x00000254: .note.ABI-tag ALLOC LOAD READONLY DATA HAS_CONTENTS
 [2]     0x00400274->0x00400298 at 0x00000274: .note.gnu.build-id ALLOC LOAD READONLY DATA HAS_CONTENTS
 [3]     0x00400298->0x004002b4 at 0x00000298: .gnu.hash ALLOC LOAD READONLY DATA HAS_CONTENTS
 [4]     0x004002b8->0x00400318 at 0x000002b8: .dynsym ALLOC LOAD READONLY DATA HAS_CONTENTS
 [5]     0x00400318->0x00400355 at 0x00000318: .dynstr ALLOC LOAD READONLY DATA HAS_CONTENTS
 [6]     0x00400356->0x0040035e at 0x00000356: .gnu.version ALLOC LOAD READONLY DATA HAS_CONTENTS
 [7]     0x00400360->0x00400380 at 0x00000360: .gnu.version_r ALLOC LOAD READONLY DATA HAS_CONTENTS
....remaining output omitted....
```

输出类似于 `info target`，但更详细。在章节名称旁边是章节标志，它是章节的属性。在这里，我们可以看到带有 LOAD 标志的章节来自 LOAD 段。该命令可以与章节标志结合以进行过滤输出：

ALLOBJ 显示所有已加载对象文件的区域，包括共享库。当程序已运行时才显示共享库。

section names 仅显示命名部分。

示例 6.2.4. 命令：

```
(gdb) maint info sections .text .data .bss
```

仅显示 .text、.data 和 .bss 部分：

Output

```
Exec file:
    '/tmp/hello', file type elf64-x86-64.
 [13]     0x00400430->0x004005c2 at 0x00000430: .text ALLOC LOAD READONLY CODE HAS_CONTENTS
 [24]     0x00601028->0x00601038 at 0x00001028: .data ALLOC LOAD DATA HAS_CONTENTS
 [25]     0x00601038->0x00601040 at 0x00001038: .bss ALLOC
```

section-flags 仅显示具有指定部分标志的章节。注意

这些部分标志特定于 gdb，尽管它基于之前定义的部分属性。目前，gdb 理解以下标志：

ALLOC 章节在加载过程中将分配空间。除包含调试信息的章节外，对所有章节进行设置。

LOAD 章节将从文件加载到子进程内存中。为预初始化的代码和数据设置，为.bss段清除。*RELOC* 章节在加载前需要重定位。*READONLY* 子进程不能修改该章节。

CODE 章节仅包含可执行代码。

DATA 章节仅包含数据（无执行代码）。

ROM 章节将驻留在ROM中。

CONSTRUCTOR 章节包含构造函数/析构函数列表的数据。

HAS_CONTENTS 章节不为空。

NEVER_LOAD 指令链接器不要输出该部分。

COFF_SHARED_LIBRARY 通知链接器该部分包含COFF共享库信息。COFF是一种对象文件格式，类似于ELF。虽然ELF是可执行二进制文件的文件格式，但COFF是对象文件的文件格式。

IS_COMMON 章节包含常用符号。

示例 6.2.5。我们可以使用命令仅显示包含代码的部分：

```
(gdb) maint info sections CODE
```

输出：

Output

```
Exec file:
    '/tmp/hello', file type elf64-x86-64.
 [10]     0x004003c8->0x004003e2 at 0x000003c8: .init ALLOC LOAD READONLY CODE HAS_CONTENTS
 [11]     0x004003f0->0x00400420 at 0x000003f0: .plt ALLOC LOAD READONLY CODE HAS_CONTENTS
 [12]     0x00400420->0x00400428 at 0x00000420: .plt.got ALLOC LOAD READONLY CODE HAS_CONTENTS
 [13]     0x00400430->0x004005c2 at 0x00000430: .text ALLOC LOAD READONLY CODE HAS_CONTENTS
 [14]     0x004005c4->0x004005cd at 0x000005c4: .fini ALLOC LOAD READONLY CODE HAS_CONTENTS
```

6.2.3 Command: `info functions`

此命令列出所有函数名称及其加载的地址。名称可以使用正则表达式进行筛选。

示例6.2.6. 运行命令，我们得到以下输出：

```
(gdb) info functions
```

Output

```
All defined functions:
File hello.c:
int main(int, char **);
Non-debugging symbols:
0x00000000004003c8  _init
0x0000000000400400  puts@plt
0x0000000000400410  __libc_start_main@plt
0x0000000000400430  _start
0x0000000000400460  deregister_tm_clones
0x00000000004004a0  register_tm_clones
0x00000000004004e0  __do_global_dtors_aux
0x0000000000400500  frame_dummy
0x0000000000400550  __libc_csu_init
0x00000000004005c0  __libc_csu_fini
0x00000000004005c4  _fini
```

6.2.4 Command: `info variables`

此命令列出所有全局和静态变量名，或通过正则表达式过滤。

示例6.2.7。如果我们将全局变量`int i`添加到示例源程序中并重新编译然后运行命令，我们得到以下输出：

```
(gdb) info variables
```

Output

```
All defined variables:
File hello.c:
int i;
Non-debugging symbols:
0x00000000004005d0  _IO_stdin_used
0x00000000004005e4  __GNU_EH_FRAME_HDR
0x0000000000400708  __FRAME_END__
0x0000000000600e10  __frame_dummy_init_array_entry
0x0000000000600e10  __init_array_start
0x0000000000600e18  __do_global_dtors_aux_fini_array_entry
0x0000000000600e18  __init_array_end
0x0000000000600e20  __JCR_END__
0x0000000000600e20  __JCR_LIST__
0x0000000000600e28  _DYNAMIC
0x0000000000601000  _GLOBAL_OFFSET_TABLE_
0x0000000000601028  __data_start
0x0000000000601028  data_start
0x0000000000601030  __dso_handle
0x000000000060103c  __bss_start
0x000000000060103c  _edata
0x000000000060103c  completed
0x0000000000601040  __TMC_END__
0x0000000000601040  _end
```

6.2.5 Command: `disassemble/disas`

This command displays the assembly code of the executable file.

示例 6.2.8. gdb 可以显示函数的汇编代码:

```
(gdb) disassemble main
```

Output

```
Dump of assembler code for function main:
   0x0804840b <+0>:  lea    ecx,[esp+0x4]
   0x0804840f <+4>:  and    esp,0xfffffff0
   0x08048412 <+7>:  push   DWORD PTR [ecx-0x4]
   0x08048415 <+10>: push   ebp
   0x08048416 <+11>: mov    ebp,esp
   0x08048418 <+13>: push   ecx
   0x08048419 <+14>: sub    esp,0x4
   0x0804841c <+17>: sub    esp,0xc
   0x0804841f <+20>: push   0x80484c0
   0x08048424 <+25>: call   0x80482e0 <puts@plt>
   0x08048429 <+30>: add    esp,0x10
   0x0804842c <+33>: mov    eax,0x0
   0x08048431 <+38>: mov    ecx,DWORD PTR [ebp-0x4]
   0x08048434 <+41>: leave
   0x08048435 <+42>: lea    esp,[ecx-0x4]
   0x08048438 <+45>: ret
End of assembler dump.
```

Example 6.2.9. 如果源包含，将更有用。 ded:

```
(gdb) disassemble /s main
```

Output

```
Dump of assembler code for function main:
hello.c:
4 {
   0x0804840b <+0>: lea     ecx,[esp+0x4]
   0x0804840f <+4>: and     esp,0xfffffff0
   0x08048412 <+7>: push    DWORD PTR [ecx-0x4]
   0x08048415 <+10>: push   ebp
   0x08048416 <+11>: mov    ebp,esp
   0x08048418 <+13>: push   ecx
   0x08048419 <+14>: sub    esp,0x4
```

```
5     printf("Hello World!\n");
   0x0804841c <+17>: sub    esp,0xc
   0x0804841f <+20>: push   0x80484c0
   0x08048424 <+25>: call   0x80482e0 <puts@plt>
   0x08048429 <+30>: add    esp,0x10
6     return 0;
   0x0804842c <+33>: mov    eax,0x0
7 }
   0x08048431 <+38>: mov    ecx,DWORD PTR [ebp-0x4]
   0x08048434 <+41>: leave
   0x08048435 <+42>: lea    esp,[ecx-0x4]
   0x08048438 <+45>: ret
End of assembler dump.
```

现在，高级源代码（以绿色文本显示）被包括在汇编转储中。每一行都有其下相应的汇编代码作为支持。

示例6.2.10。如果添加选项/r，将包含十六进制原始指令，就像objdump默认显示汇编代码一样：

```
(gdb) disassemble /rs main
```

Output

```
Dump of assembler code for function main:
hello.c:
4 {
   0x0804840b <+0>:  8d 4c 24 04    lea    ecx,[esp+0x4]
   0x0804840f <+4>:  83 e4 f0       and    esp,0xfffffff0
   0x08048412 <+7>:  ff 71 fc       push   DWORD PTR [ecx-0x4]
   0x08048415 <+10>: 55   push   ebp
   0x08048416 <+11>: 89 e5   mov    ebp,esp
   0x08048418 <+13>: 51   push   ecx
   0x08048419 <+14>: 83 ec 04       sub    esp,0x4
5     printf("Hello World!\n");
   0x0804841c <+17>: 83 ec 0c       sub    esp,0xc
```

```
   0x0804841f <+20>: 68 c0 84 04 08 push   0x80484c0
   0x08048424 <+25>: e8 b7 fe ff ff call   0x80482e0 <puts@plt>
   0x08048429 <+30>: 83 c4 10       add    esp,0x10
6     return 0;
   0x0804842c <+33>: b8 00 00 00 00 mov    eax,0x0
7 }
   0x08048431 <+38>: 8b 4d fc       mov    ecx,DWORD PTR [ebp-0x4]
   0x08048434 <+41>: c9 leave
   0x08048435 <+42>: 8d 61 fc       lea    esp,[ecx-0x4]
   0x08048438 <+45>: c3 ret
End of assembler dump.
```

示例6.2.11。特定文件中的函数也可以指定:

```
(gdb) disassemble /sr 'hello.c'::main
```

Output

```
Dump of assembler code for function main:
hello.c:
4 {
   0x0804840b <+0>:  8d 4c 24 04   lea    ecx,[esp+0x4]
   0x0804840f <+4>:  83 e4 f0      and    esp,0xfffffff0
   0x08048412 <+7>:  ff 71 fc      push   DWORD PTR [ecx-0x4]
   0x08048415 <+10>: 55    push   ebp
   0x08048416 <+11>: 89 e5 mov    ebp,esp
   0x08048418 <+13>: 51    push   ecx
   0x08048419 <+14>: 83 ec 04       sub    esp,0x4
5     printf("Hello World!\n");
   0x0804841c <+17>: 83 ec 0c       sub    esp,0xc
   0x0804841f <+20>: 68 c0 84 04 08 push   0x80484c0
   0x08048424 <+25>: e8 b7 fe ff ff call   0x80482e0 <puts@plt>
   0x08048429 <+30>: 83 c4 10       add    esp,0x10
6     return 0;
   0x0804842c <+33>: b8 00 00 00 00 mov    eax,0x0
7 }
```

```
   0x08048431 <+38>: 8b 4d fc       mov    ecx,DWORD PTR [ebp-0x4]
   0x08048434 <+41>: c9 leave
   0x08048435 <+42>: 8d 61 fc       lea    esp,[ecx-0x4]
   0x08048438 <+45>: c3 ret
End of assembler dump.
```

文件名必须包含在单引号内，并且函数必须以双冒号作为前缀，例如 `'hello.c'::main` 以指定对文件 `hello.c` 中的函数 `main` 进行反汇编。

6.2.6 Command: x

此命令检查给定内存范围的内容。

示例 6.2.12。我们可以检查 `main` 中的原始内容：

```
(gdb) x main
```

输出

```
0x804840b <main>: 0x04244c8d
```

默认情况下，如果没有任何参数，该命令仅打印单个内存地址的内容。在这种情况下，那就是 `main` 中的起始内存地址。

示例6.2.13。使用格式参数，该命令可以按特定格式打印内存范围。

```
(gdb) x/20b main
```

Output

```
0x804840b <main>:    0x8d 0x4c 0x24 0x04 0x83 0xe40xf0 0xff
0x8048413 <main+8>:  0x71 0xfc 0x55 0x89 0xe5 0x510x83 0xec
0x804841b <main+16>: 0x04 0x83 0xec 0x0c
```

`/20b main` 参数表示命令打印20字节，其中main从内存开始。

通用的格式参数形式是：`/<repeated count><format letter>`

如果未提供重复计数，则默认由 gdb 提供计数

1. 格式字母是以下值之一：

Letter	Description
o	Print the memory content in *octal* format.
x	Print the memory content in hex format.
d	Print the memory content in decimal format.
u	Print the memory content in *unsigned decimal* format.
t	Print the memory content in *binary* format.
f	Print the memory content in *float* format.
a	Print the memory content as *memory addresses*.
i	Print the memory content as a series of assembly instructions, similar to `disassemble` command.
c	Print the memory content as an array of ASCII characters.
s	Print the memory content as a string

根据情况，某些格式比其他格式更有优势。例如，如果一个内存区域包含浮点数，那么使用格式 f 比将数字视为分开的 1 字节十六进制数更好。

6.2.7 Command: print/p

检查原始内存很有用，但通常更好的是有一个更易于阅读的输出。此命令恰好执行此任务：它美化了表达式。一个表达式可以是一个全局变量、当前堆栈帧中的局部变量、一个函数、一个寄存器、一个数字等。

6.3 程序运行时检查

调试器的主要用途是在程序运行时检查其状态。gdb提供了一组有用的命令，用于检索有用的运行时信息。

6.3.1 Command: run

此命令开始运行程序。

示例 6.3.1. 运行 hello 程序：

```
(gdb) r
```

Output

```
Starting program: /tmp/hello
Hello World!
[Inferior 1 (process 1002) exited normally]
```

程序运行成功并打印了消息“Hello World”。然而，如果所有gdb能做的只是运行一个程序，那就没有用了。

6.3.2 Command: break/b

此命令在高级源代码中的某个位置设置断点。当 gdb 运行到由断点标记的特定位置时，它将停止执行，以便程序员检查程序当前状态。

示例 6.3.2. 可以在编辑器显示的行上设置断点。假设我们想在程序的第三行设置断点，这是 main 函数的开始：

hello.c

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5     printf("Hello World!\n");
6     return 0;
7 }
```

当运行程序时，而不是从开始到结束运行，gdb 停留在第3行：

```
(gdb) b 3
```

输出

```
Breakpoint 1 at 0x400535: file hello.c, line 3.
```

```
(gdb) r
```

输出

```
Starting program: /tmp/hello
Breakpoint 1, main (argc=1, argv=0x7fffffffdfb8) at hello.c:5
5     printf("Hello World!\n");
```

断点位于第3行，但gdb在第5行停止。原因是第3行不包含代码，而是一个函数签名；gdb只会在可以执行代码的地方停止。函数中的代码从第5行开始，调用printf，因此gdb在那里停止。

示例6.3.3。一行代码并不总是指定断点的可靠方式，因为源代码可能会更改。如果gdb应该在main函数处始终停止怎么办？在这种情况下，更好的方法是直接使用函数名：

```
b main
```

然后，无论源代码如何更改，gdb 总是在 main 函数处停止。

示例6.3.4。有时，调试程序不包含调试信息，或者 gdb 正在调试汇编代码。在这种情况下，可以将内存地址指定为停止点。要获取函数地址，可以使用 print 命令：

```
(gdb) print main
```

输出

```
$3 = {int (int, char **)} 0x400526 <main>
```

了解主函数的地址，我们可以轻松设置一个断点，使用

内存地址：

```
b *0x400526
```

示例6.3.5. gdb 也可以在任何源文件中设置断点。假设 hello 程序不仅由一个文件组成，而是由多个文件组成，例如 hello1.c、hello2.c、hello3.c... 在这种情况下，只需在行号之前添加文件名即可：

```
b hello.c:3
```

示例6.3.6。在特定文件中，也可以设置函数名：

```
b hello.c:main
```

6.3.3 Command: next/n

此命令执行当前行并在下一行停止。当当前行是函数调用时，会跳过它。

示例6.3.7。在设置断点于main后，运行程序并在第一次到达printf时停止：

```
(gdb) r
```

输出

```
Starting program: /tmp/hello
Breakpoint 1, main (argc=1, argv=0x7fffffffdfb8) at hello.c:5
5     printf("Hello World!\n");
```

然后，要进入下一个语句，我们使用 next 命令：

```
(gdb) n
```

Output

```
Hello World!
6     return 0;
```

在输出中，第一行显示执行第5行后的输出；然后，下一行显示当前gdb停止的位置，即第6行。

6.3.4 Command: step/s

此命令执行当前行并在下一行停止。当当前行是函数调用时，进入该函数并停在调用函数的第一个下一行。

示例6.3.8。假设我们有一个新的函数 add[1]:

hello.c

```
#include <stdio.h>

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

int main(int argc, char *argv[])
{
    add(1, 2);
    printf("Hello World!\n");
    return 0;
}
```

1 为什么我们要添加一个新的函数和函数调用，而不是使用现有的 printf 调用？进入共享库函数很棘手，因为为了使调试工作，必须安装和加载调试信息。演示这个简单的命令不值得麻烦。

如果使用 step 命令代替 next 在函数调用 printf 中，gdb进入函数:

```
(gdb) r
```

Output

```
Starting program: /tmp/hello
Breakpoint 1, main (argc=1, argv=0xffffd154) at hello.c:11
11      add(1, 2);
```

```
(gdb) s
```

输出

```
add (a=1, b=2) at hello.c:6
6     return a + b;
```

执行命令 s 后，gdb 步入 add 函数，其中第一条语句是一个 return。

6.3.5 Command: ni

在核心上，gdb 在汇编指令上操作。逐行调试源代码只是一个增强，使其对程序员更友好。C语言中的每条语句都翻译为一条或多条汇编指令，如objdump和disassemble命令所示。有了调试信息，gdb知道属于一条高级代码行的指令数量；逐行调试只是当从当前行移动到下一行时执行该行的汇编指令。

此命令执行当前行的 *one* 汇编指令。直到当前行的所有汇编指令执行完毕，gdb 才会移动到下一行。如果当前指令是 call，则跳过它执行下一条指令。

示例6.3.9。当断点位于 printf 调用处且使用 ni 时，它将逐条执行汇编指令：

```
(gdb) disassemble /s main
```

Output

```
Dump of assembler code for function main:
hello.c:
4 {
   0x0804840b <+0>:  lea    ecx,[esp+0x4]
   0x0804840f <+4>:  and    esp,0xfffffff0
   0x08048412 <+7>:  push   DWORD PTR [ecx-0x4]
   0x08048415 <+10>: push   ebp
```

```
   0x08048416 <+11>: mov    ebp,esp
   0x08048418 <+13>: push   ecx
   0x08048419 <+14>: sub    esp,0x4
5     printf("Hello World!\n");
   0x0804841c <+17>: sub    esp,0xc
   0x0804841f <+20>: push   0x80484c0
   0x08048424 <+25>: call   0x80482e0 <puts@plt>
   0x08048429 <+30>: add    esp,0x10
6     return 0;
=> 0x0804842c <+33>: mov    eax,0x0
7 }
   0x08048431 <+38>: mov    ecx,DWORD PTR [ebp-0x4]
   0x08048434 <+41>: leave
   0x08048435 <+42>: lea    esp,[ecx-0x4]
   0x08048438 <+45>: ret
End of assembler dump.
```

```
(gdb) r
```

Output

```
Starting program: /tmp/hello
Breakpoint 1, main (argc=1, argv=0xffffd154) at hello.c:5
5     printf("Hello World!\n");
```

```
(gdb) ni
```

Output

```
0x0804841f 5     printf("Hello World!\n");
```

```
(gdb) ni
```

Output

```
0x08048424 5     printf("Hello World!\n");
```

```
(gdb) ni
```

Output

```
Hello World!
0x08048429 5     printf("Hello World!\n");
```

```
(gdb)
```

输出

```
6     return 0;
```

进入 ni 后，gdb 执行当前指令并显示 *next* 指令。这就是为什么从输出中，gdb 只显示了 3 个地址：0x0804841f、0x08048424 和 0x08048429。位于 0x0804841c 的指令，即 printf 的第一条指令，没有显示，因为它是在 gdb 停止的第一条指令。假设 gdb 在 0x0804841c 处停止在 printf 的第一条指令，当前指令可以使用 x 命令显示：

```
(gdb) x/i $eip
```

输出

```
=> 0x804841c <main+17>: sub    esp,0xc
```

6.3.6 Command: si

类似于 ni，此命令执行当前汇编指令属于当前行。但如果当前指令是 call，则进入它到被调用函数的第一个后续指令。

示例6.3.10。回想一下从 printf 生成的汇编代码中包含一个 call 指令：

```
(gdb) disassemble /s main
```

Output

```
Dump of assembler code for function main:
hello.c:
4 {
   0x0804840b <+0>: lea    ecx,[esp+0x4]
   0x0804840f <+4>: and    esp,0xfffffff0
   0x08048412 <+7>: push   DWORD PTR [ecx-0x4]
   0x08048415 <+10>: push   ebp
   0x08048416 <+11>: mov    ebp,esp
   0x08048418 <+13>: push   ecx
   0x08048419 <+14>: sub    esp,0x4
5     printf("Hello World!\n");
   0x0804841c <+17>: sub    esp,0xc
   0x0804841f <+20>: push   0x80484c0
   0x08048424 <+25>: call   0x80482e0 <puts@plt>
   0x08048429 <+30>: add    esp,0x10
6     return 0;
=> 0x0804842c <+33>: mov    eax,0x0
7 }
   0x08048431 <+38>: mov    ecx,DWORD PTR [ebp-0x4]
   0x08048434 <+41>: leave
   0x08048435 <+42>: lea    esp,[ecx-0x4]
   0x08048438 <+45>: ret
End of assembler dump.
```

我们再次逐条指令地尝试步进，但这次是通过在 0x08048424 运行 si，其中 call 存在：

```
(gdb) si
```

Output

```
0x0804841f 5         printf("Hello World!\n");
```

```
(gdb) si
```

输出
```
    0x08048424 5         printf("Hello World!\n");
```

```
(gdb) x/i $eip
```

输出
```
=> 0x8048424 <main+25>: call   0x80482e0 <puts@plt>
```

```
(gdb) si
```

输出
```
    0x080482e0 in puts@plt ()
```

下一条指令紧跟在 0x8048424 之后，是 puts 函数中 0x080482e0 的第一条指令。换句话说，gdb 跳入了 puts 而不是跳过它。

6.3.7 Command: `until`

此命令执行，直到下一行大于当前行。

示例6.3.11。假设我们有一个执行长循环的函数:

hello.c
```
#include <stdio.h>

int add1000() {
    int total = 0;

    for (int i = 0; i < 1000; ++i){
        total += i;
    }

    printf("Done adding!\n");
```

```
    return total;
}

int main(int argc, char *argv[])
{
    add1000(1, 2);
    printf("Hello World!\n");
    return 0;
}
```

Using `next` command, we need to press 1000 times for finishing the loop. Instead, a faster way is to use `until`:

```
(gdb) b add1000
```

Output

```
Breakpoint 1 at 0x8048411: file hello.c, line 4.
```

```
(gdb) r
```

Output

```
Starting program: /tmp/hello
Breakpoint 1, add1000 () at hello.c:4
4     int total = 0;
```

```
(gdb) until
```

Output

```
5     for (int i = 0; i < 1000; ++i){
```

```
(gdb) until
```

Output

```
6         total += i;
```

```
(gdb) until
```

输出

```
5     for (int i = 0; i < 1000; ++i){
(gdb) until
```

```

```

输出

```
8     printf("Done adding!\n");
```

执行第一个 `until`，gdb 在第5行停止，因为第5行大于第4行。

执行第二个 `until`，gdb 在第6行停止，因为第6行大于第5行。

执行第三个 `until`，gdb 在第5行停止，因为循环仍在继续。因为第5行小于第6行，使用第四个 `until`，gdb 继续执行，直到不再回到第5行，并在第8行停止。这是一种在中间跳过循环的好方法，而不是设置不必要的断点。

示例 6.3.12. `until` 可以提供一个参数以显式执行到特定行：

```
(gdb) r
```

输出

```
Starting program: /tmp/hello
4     int total = 0;输出
    Breakpoint 1, add1000 () at hello.c:4
add1000 () at hello.c:8

8     printf("Done adding!\n");
(gdb) until 8
```

```

```

输出

```

```

6.3.8 Command: *finish*

此命令执行到函数结束并显示返回值。finish实际上是until的一个更方便的版本。

示例6.3.13。使用前一个示例中的 add1000 函数，并用 finish 代替 until：

```
(gdb) r
```

Output

```
Starting program: /tmp/hello
Breakpoint 1, add1000 () at hello.c:4
4     int total = 0;
```

```
(gdb) finish
```

Output

```
Run till exit from #0  add1000 () at hello.c:4
Done adding!
0x08048466 in main (argc=1, argv=0xffffd154) at hello.c:15
15     add1000(1, 2);
Value returned is $1 = 499500
```

6.3.9 Command: *bt*

此命令打印所有栈帧的 *backtrace*。一个 *backtrace* 是当前活动函数的 ***backtrace*** 列表：

示例6.3.14。假设我们有一个函数调用链：

hello.c

```
void d(int d) { };
void c(int c) { d(0); }
void b(int b) { c(1); }
void a(int a) { b(2); }

int main(int argc, char *argv[])
```

```
{
    a(3);
    return 0;
}
```

`bt` can visualize such a chain in action:

```
(gdb) b a
```

Output

```
Breakpoint 1 at 0x8048404: file hello.c, line 9.
```

```
(gdb) r
```

Output

```
Starting program: /tmp/hello
Breakpoint 1, a (a=3) at hello.c:9
9 void a(int a) { b(2); }
```

```
(gdb) s
```

Output

```
b (b=2) at hello.c:7
7 void b(int b) { c(1); }
```

```
(gdb) s
```

Output

```
c (c=1) at hello.c:5
5 void c(int c) { d(0); }
```

```
(gdb) s
```

输出

```
d (d=0) at hello.c:3
#2  0x080483fb in b (b=2) at hello.c:7
#3  0x0804840b in a (a=3) at hello.c:9
#4  0x0804841b in main (argc=1, argv=0xffffd154) at hello.c:13
```

```
(gdb) bt
```

输出

```
#0  d (d=0) at hello.c:3
#1  0x080483eb in c (c=1) at hello.c:5
3 void d(int d) { };
```

最近调用放在顶部，最远调用接近底部。在这种情况下，d 是当前最活跃的功能，因此它具有索引 0。接下来是 c，2nd 活跃功能，具有索引 1，以此类推，直到函数 b、函数 a，最后是底部的函数 main，这是最远的函数。这就是我们读取堆栈跟踪的方式。

*6.3.10 Command: **up***

此命令比当前帧提前一帧。

示例6.3.15。我们不必停留在d函数中，可以上升到c函数并查看其状态：

```
(gdb) bt
```

Output

```
#0  d (d=0) at hello.c:3
#1  0x080483eb in c (c=1) at hello.c:5
#2  0x080483fb in b (b=2) at hello.c:7
#3  0x0804840b in a (a=3) at hello.c:9
#4  0x0804841b in main (argc=1, argv=0xffffd154) at hello.c:13
```

```
(gdb) up
```

Output

```
#1  0x080483eb in c (c=1) at hello.c:3
3 void b(int b) { c(1); }
```

The output displays the current frame is moved to `c` and where the call to `c` is made, which is in function `b` at line 3.

6.3.11 Command: `down`

Similar to `up`, this command goes down one frame later then the current frame.

Example 6.3.16. After inspecting `c` function, we can go back to `d`:

```
(gdb) bt
```

Output

```
#0  d (d=0) at hello.c:3
#1  0x080483eb in c (c=1) at hello.c:5
#2  0x080483fb in b (b=2) at hello.c:7
#3  0x0804840b in a (a=3) at hello.c:9
#4  0x0804841b in main (argc=1, argv=0xffffd154) at hello.c:13
```

```
(gdb) up
```

Output

```
#1  0x080483eb in c (c=1) at hello.c:3
3 void b(int b) { c(1); }
```

```
(gdb) down
```

Output

```
#0  d (d=0) at hello.c:1
1 void d(int d) { };
```

6.3.12 Command: `info registers`

此命令列出常用寄存器中的当前值。此命令在调试汇编和操作系统代码时很有用，因为我们可以检查机器的当前状态。

示例6.3.17. 执行命令后，我们可以看到常用的寄存器：

```
(gdb) info registers
```

Output

```
eax            0xf7faddbc -134554180
ecx            0xffffd0c0 -12096
edx            0xffffd0e4 -12060
ebx            0x0 0
esp            0xffffd0a0 0xffffd0a0
ebp            0xffffd0a8 0xffffd0a8
esi            0xf7fac000 -134561792
edi            0xf7fac000 -134561792
eip            0x804841c 0x804841c <main+17>
eflags         0x286 [ PF SF IF ]
cs             0x23 35
ss             0x2b 43
ds             0x2b 43
es             0x2b 43
fs             0x0 0
gs             0x63 99
```

以上寄存器足以在后续部分编写我们的操作系统。

6.4 调试器如何工作：简要介绍

6.4.1 How breakpoints work

When a programmer places a breakpoint somewhere in his code, what actually happens is that the *first* opcode of the *first* instruction of a state-

指令ment被替换为另一条指令，`int 3`用操作码CCh替换：

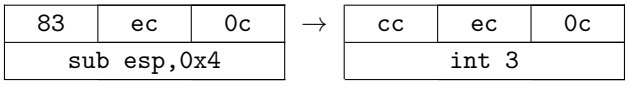


图6.4.1：操作码替换，含 `int 3`

`int 3` 仅占用一个字节，使其在调试时效率高。当执行 `int 3` 指令时，操作系统调用其断点中断处理程序。处理程序随后检查哪个进程达到断点，暂停它并通知调试器它已暂停调试进程。调试进程仅被暂停，这意味着调试器可以自由地检查其内部状态，就像外科医生在麻醉病人身上进行手术一样。然后，调试器将 `int 3` 操作码替换为原始操作码，并正常执行原始指令。

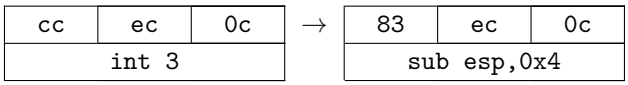


图6.4.2：执行 `int 3` 后恢复原始操作码

示例6.4.1。很容易看到 `int 3` 的作用。首先，我们添加一个 `int 3` 指令中我们需要 gdb 停止的地方：

hello.c

```
#include <stdio.h>


int main(int argc, char *argv[])
{
    asm("int 3");
    printf("Hello World\n");
    return 0;
}
```

`int 3` 在 `printf` 之前，因此预计 gdb 将在 `printf` 处停止。接下来，我们启用调试并使用 Intel 语法进行编译：

```
$ gcc -masm=intel -m32 -g hello.c -o hello
```

最后，开始 gdb：

```
$ gdb hello
```

运行时未设置任何断点，gdb 如预期地停在 printf 调用处：

```
(gdb) r
```

输出

```
Starting program: /tmp/hello
Program received signal SIGTRAP, Trace/breakpoint trap.
main (argc=1, argv=0xffffd154) at hello.c:6
6     printf("Hello World\n");
```

蓝色文本表示 gdb 遇到了断点，确实它停在了正确的位置：在 printf 调用之前，int 3 先于它出现。

6.4.2 Single stepping

当断点实现时，实现单步执行很容易：调试器只需在下一条指令中放置另一个 int 3 操作码。因此，当程序员在一条指令处设置断点时，调试器会自动设置下一条指令，从而实现逐条指令的调试。同样，逐行源代码调试只是将两个语句中的第一个操作码放置为两个 int 3 操作码。

6.4.3 How a debugger understands high level source code

DWARF是一种调试文件格式，许多编译器和调试器使用它来支持源级调试。DWARF包含映射可执行二进制文件中的实体与源文件之间的信息。程序实体可以是数据或代码。DIE，或 ***D****ebugging* ***I****nformation* ***E****ntry*，是程序实体的描述。DIE由一个标签组成，该标签指定DIE描述的实体，以及一个描述实体的属性列表。在所有属性中，这两个属性使源级调试成为可能：

Debugging Information Entry

▷ 在源文件中实体出现的位置：哪个文件以及哪一行实体出现。

▷ 实体在可执行二进制文件中的出现位置：在运行时实体被加载到的内存地址。有了精确的地址，gdb可以检索数据实体的正确值，或者放置正确的断点并相应地停止代码实体。没有这些地址的信息，gdb将不知道要检查的实体在哪里。

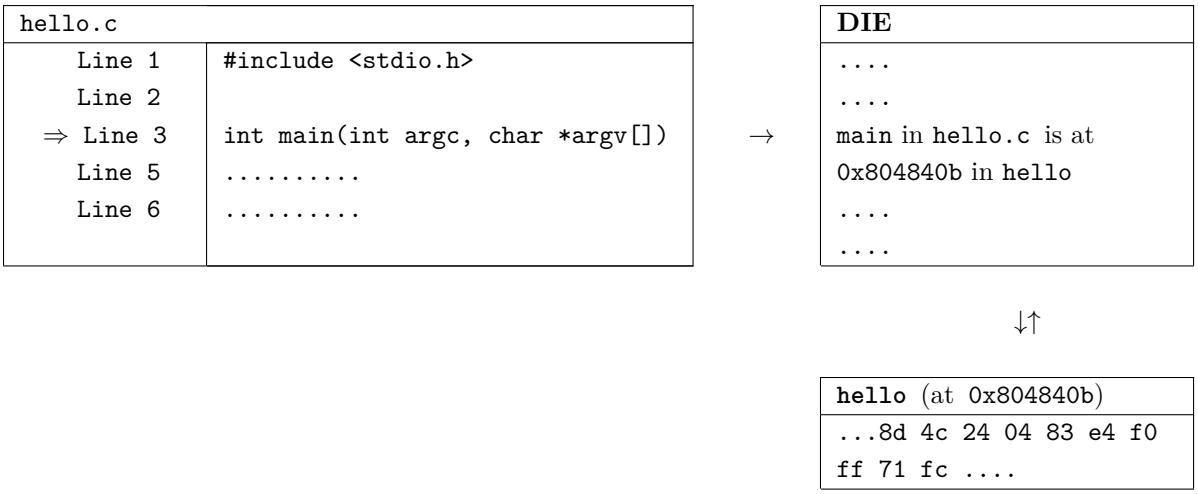


图6.4.3：使用DIE的源-二进制映射

除了DIEs之外，另一种二进制到源映射是 *line num- ber table*。行号表将源代码中的一行与可执行二进制中该行的起始内存地址进行映射。

总结来说，为了成功启用源级调试，调试器需要知道源文件的精确位置和运行时的加载地址。ELF二进制映像的布局与加载地址之间的地址匹配非常重要，因为调试信息依赖于运行时的正确加载地址。也就是说，它假设编译时记录在二进制映像中的地址与运行时相同，例如，如果.text段的加载地址记录在可执行二进制文件中的0x800000，那么当二进制文件实际运行时，.text应该真正加载到0x800000，以便gdb能够正确地将运行指令与高级代码语句匹配。地址不匹配会使调试信息失效，因为实际代码在一个地址上显示为另一个地址上的代码。没有这种知识，我们将

无法构建一个可以用 gdb 调试的操作系统。

示例6.4.2。当一个可执行二进制文件包含调试信息时，`readelf`可以以可读的格式显示此类信息。使用那个古老的hello world程序：

hello.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello World\n");

    return 0;
}
```

使用调试信息进行编译：

```
$ gcc -m32 -g hello.c -o hello
```

使用二进制文件准备就绪后，我们可以使用以下命令查看行号表：

```
$ readlelf -wL hello
```

`-w` 选项打印所有调试信息。与子选项结合使用时，仅显示特定信息。例如，使用 `-L`，仅显示行号表：

Output

```
Decoded dump of debug contents of section .debug_line:
CU: hello.c:
File name                            Line number    Starting address
hello.c                                        6           0x804840b
hello.c                                        7           0x804841c
hello.c                                        9           0x804842c
```

```
    hello.c                                    10          0x8048431
```

从上述输出：

CU 简写为 ***C****ompilation* ***U****nit*，一个单独编译的源文件。在示例中，我们只有一个文件，`hello.c`。

File name 显示当前编译单元的文件名。

Line number 是源文件中非空行的行号。在示例中，第8行是空行，因此它没有出现。

Starting address 这是可执行二进制文件中实际开始行的内存地址。

在如此清晰的信息下，这就是gdb能够轻松在行上设置断点的原理。要放置变量和函数的断点，是时候查看DIEs了。要从可执行二进制文件中获取DIEs信息，请运行以下命令：

```
$ readelf -wi hello
```

`-wi` 选项列出所有DIE条目。这是一个典型的DIE条目：

```
 <0><b>: Abbrev Number: 1 (DW_TAG_compile_unit)
    <c>   DW_AT_producer    : (indirect string, offset: 0xe): GNU C11 5.4.0 20160609 -masm=intel -m32
    <10>   DW_AT_language    : 12 (ANSI C99)
    <11>   DW_AT_name        : (indirect string, offset: 0xbe): hello.c
    <15>   DW_AT_comp_dir    : (indirect string, offset: 0x97): /tmp
    <19>   DW_AT_low_pc      : 0x804840b
    <1d>   DW_AT_high_pc     : 0x2e
    <21>   DW_AT_stmt_list   : 0x0
```

Red 这个最左边的数字表示DIE条目当前嵌套级别。0是外层级别的DIE，其实体是编译单元。这意味着后续嵌套级别更高的DIE条目都是此标签，即编译单元的子项。这很有道理，因为所有实体都必须源自源文件。

Blue 这些以十六进制格式表示的数字指示 .debug_info 部分的偏移量。每个有意义的信息都与其偏移量一起显示。当一个属性引用另一个属性时，使用偏移量来精确地标识所引用的属性。

Green 这些带有 DW_AT_ 前缀的名称是描述实体的DIE所附加的属性。显著的属性:

DW_AT_name

DW_AT_comp_dir 编译单元的文件名以及编译发生的目录。没有文件名和路径，即使有调试信息，gdb也无法显示高级源代码。调试信息仅包含源代码和二进制之间的映射，而不是源代码本身。

DW_AT_low_pc

DW_AT_high_pc 当前实体的起始和结束位置，该实体是编译单元，在可执行二进制文件中。DW_AT_low_pc中的值是起始地址。DW_AT_high_pc是编译单元的大小，当与DW_AT_low_pc相加时，得到实体的结束地址。在此示例中，从hello.c编译的代码从0x804840b开始，到0x804840b + 0x2e = 0x8048439.结束。为确保无误，我们使用objdump进行验证:

Output

```
 int main(int argc, char *argv[])
{
  804840b:       8d 4c 24 04             lea    ecx,[esp+0x4]
  804840f:       83 e4 f0                and    esp,0xfffffff0
  8048412:       ff 71 fc                push   DWORD PTR [ecx-0x4]
  8048415:       55                      push   ebp
  8048416:       89 e5                   mov    ebp,esp
  8048418:       51                      push   ecx
  8048419:       83 ec 04                sub    esp,0x4
     printf("Hello World\n");
  804841c:       83 ec 0c                sub    esp,0xc
  804841f:       68 c0 84 04 08          push   0x80484c0
```

```
 8048424:       e8 b7 fe ff ff          call   80482e0 <puts@plt>
 8048429:       83 c4 10                add    esp,0x10
    return 0;
 804842c:       b8 00 00 00 00          mov    eax,0x0
}
 8048431:       8b 4d fc                mov    ecx,DWORD PTR [ebp-0x4]
 8048434:       c9                      leave
 8048435:       8d 61 fc                lea    esp,[ecx-0x4]
 8048438:       c3                      ret
 8048439:       66 90                   xchg   ax,ax
 804843b:       66 90                   xchg   ax,ax
 804843d:       66 90                   xchg   ax,ax
 804843f:       90                      nop
```

这是真的：`main` 从 804840b 开始，在 8048439 结束，紧随 `ret` 指令之后的 8048438。8048439 之后的指令只是由 `gcc` 插入的对齐填充字节，不属于 `main`。请注意，`objdump` 的输出显示了 `main` 之后的更多代码。这些代码不计入，因为它们位于 `hello.c` 之外，由 `gcc` 为操作系统添加。`hello.c` 只包含一个函数：`main`，这也是为什么 `hello.c` 也以相同的方式开始和结束，与 `main` 相同。

Pink 这个数字显示标签的缩写形式。缩写是DIE的形式。当使用`-wi`显示调试信息时，DIE及其值会显示。`-wa`选项在`.debug_abbrev`部分显示缩写：

Output

```
Contents of the .debug_abbrev section:
  Number TAG (0x0)
   1      DW_TAG_compile_unit    [has children]
    DW_AT_producer     DW_FORM_strp
    DW_AT_language     DW_FORM_data1
    DW_AT_name         DW_FORM_strp
    DW_AT_comp_dir     DW_FORM_strp
```

```
        DW_AT_low_pc        DW_FORM_addr
        DW_AT_high_pc       DW_FORM_data4
        DW_AT_stmt_list     DW_FORM_sec_offset
        DW_AT value: 0      DW_FORM value: 0
    .... more abbreviations ....
```

输出类似于DIE输出，只有属性名称，没有任何值。我们也可以说缩写是DIE的一个*type*，因为缩写代表了特定DIE的结构。许多DIE共享相同的缩写或结构，因此它们属于同一类型。缩写编号指定了DIE在上述缩写表中的类型。缩写可以提高编码效率（减少二进制大小），因为每个DIE不需要携带它们的结构信息作为属性-值2对，而只需简单地引用

对一个缩写进行正确解码。

例如，数据格式如 YAML 或 JSON 将其属性名称与其值一起编码。这简化了编码，但增加了开销。

这里展示了hello中所有DIEs的表示，以树的形式：

在图6.4.4中，`DW_TAG_subprogram`代表一个如`main`的函数。它的子节点是`argc`和`argv`的DIEs。有了这样精确的信息，对于`gdb`来说，将源代码与二进制代码匹配是一项简单的工作。

如果可执行二进制文件中存在多个编译单元，DIE条目将根据编译顺序从`gcc`进行排序。例如，假设我们还有另一个`test.c`源文件3，并将其编译为-

3 它可以包含任何内容。只是一个示例文件。

与hello一起：

```
$ gcc -masm=intel -m32 -g test.c hello.c -o hello
```

然后，在 `test.c` 中的所有 DIE 条目在 `hello.c`: 中的 DIE 条目之前显示

```
<0><b>: Abbrev Number: 1 (DW_TAG_compile_unit)

  <c>   DW_AT_producer    : (indirect string, offset: 0x0): GNU C11 5.4.0 20160609
 -masm=intel -m32 -mtune=generic -march=i686 -g -fstack-protector-strong
     <10>   DW_AT_language    : 12       (ANSI C99)
```

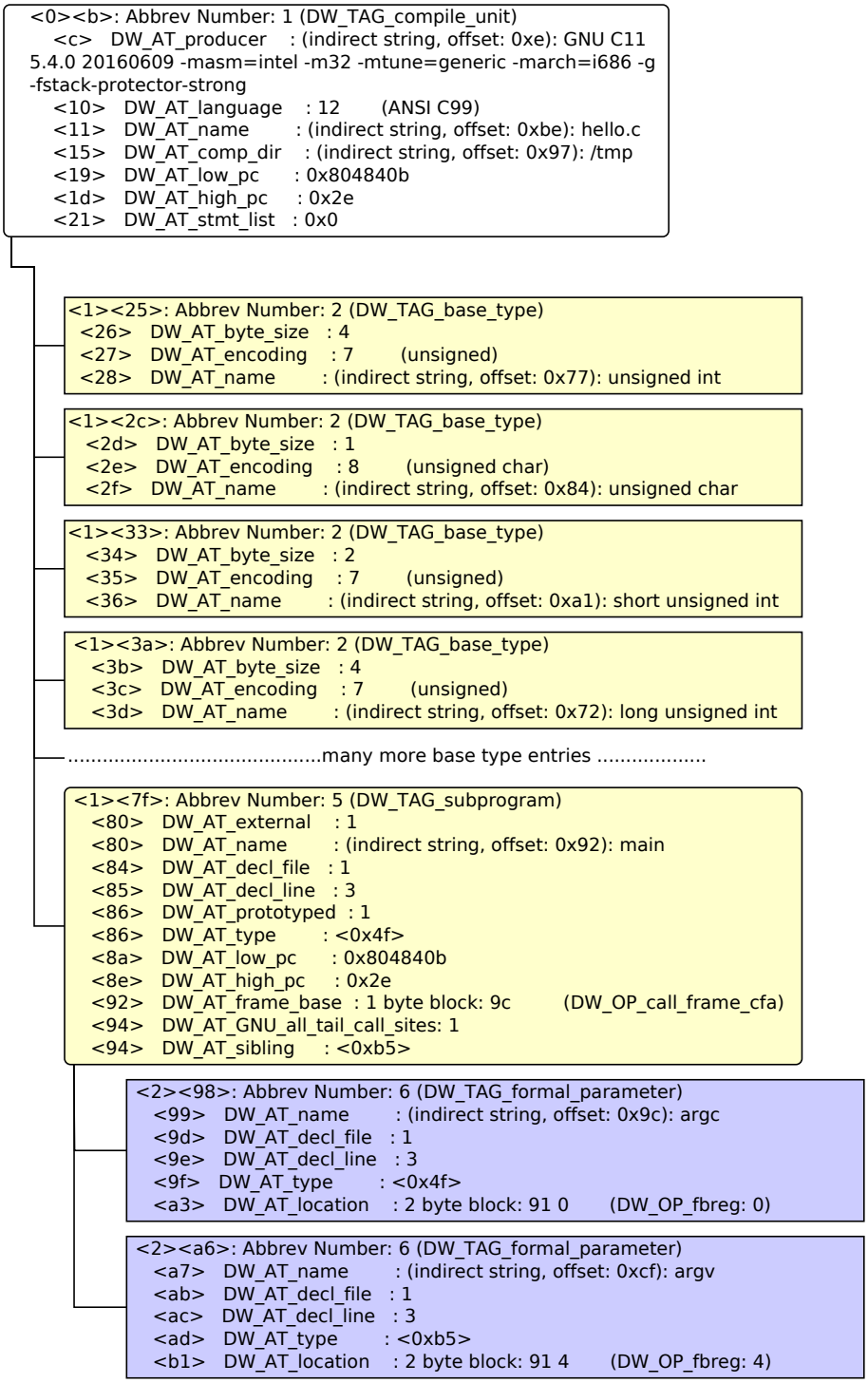


图6.4.4：将DIE条目可视化为一棵树

```
    <11>   DW_AT_name        : (indirect string, offset: 0x64): test.c
    <15>   DW_AT_comp_dir    : (indirect string, offset: 0x5f): /tmp
    <19>   DW_AT_low_pc      : 0x804840b
    <1d>   DW_AT_high_pc     : 0x6
    <21>   DW_AT_stmt_list   : 0x0
 <1><25>: Abbrev Number: 2 (DW_TAG_subprogram)
    <26>   DW_AT_external    : 1
    <26>   DW_AT_name        : bar
    <2a>   DW_AT_decl_file   : 1
    <2b>   DW_AT_decl_line   : 1
    <2c>   DW_AT_low_pc      : 0x804840b
    <30>   DW_AT_high_pc     : 0x6
    <34>   DW_AT_frame_base  : 1 byte block: 9c         (DW_OP_call_frame_cfa)
    <36>   DW_AT_GNU_all_call_sites: 1
....after all DIEs in test.c listed....
<0><42>: Abbrev Number: 1 (DW_TAG_compile_unit)
    <43>   DW_AT_producer    : (indirect string, offset: 0x0): GNU C11 5.4.0 20160609
-masm=intel -m32 -mtune=generic -march=i686 -g -fstack-protector-strong
    <47>   DW_AT_language    : 12       (ANSI C99)
    <48>   DW_AT_name        : (indirect string, offset: 0xc5): hello.c
    <4c>   DW_AT_comp_dir    : (indirect string, offset: 0x5f): /tmp
    <50>   DW_AT_low_pc      : 0x8048411
    <54>   DW_AT_high_pc     : 0x2e
    <58>   DW_AT_stmt_list   : 0x35
....then all DIEs in hello.c are listed....
```

第二部分

基础工作

7

Bootloader

一个 *bootloader* 加载操作系统，或一个运行并通信的应用程序 1。

直接与硬件交互。要运行操作系统，首先要编写引导加载程序。在本章中，我们将编写一个基本的引导加载程序，因为我们的主要重点是编写操作系统，而不是引导加载程序。更有趣的是，本章将介绍适用于编写引导加载程序以及操作系统的相关工具和技术。

1 许多嵌入式设备不使用操作系统。在嵌入式系统中，引导加载程序简单地包含在引导固件中，不需要引导加载程序。

7.1 x86 引导过程

在POST过程完成后，CPU的程序计数器被设置为地址`FFFF:0000h`以执行BIOS代码。*BIOS - **B**asic **I**nput/**O**utput **S**ystem*是一种固件，它执行硬件初始化并提供一组通用子例程以控制输入/输出设备。BIOS检查所有可用的存储设备（软盘和硬盘），通过检查第一个扇区的最后两个字节是否具有`0x55`、`0xAA`的引导记录签名来确定是否有可引导的设备。如果是这样，BIOS将第一个扇区加载到地址`7C00h`，将程序计数器设置为该地址，并让CPU从该地址执行代码。

第一个区域被称为 ***M**aster **B**oot **R**ecord,* 或 *MBR*。第一个区域中的程序被称为 *MBR Bootloader*。

7.2 使用BIOS服务

BIOS在启动阶段为控制硬件提供许多基本服务。一项服务是一组控制特定硬件设备或返回当前系统信息的例程。每个服务都有一个中断号。要调用BIOS例程，必须使用带有中断号的中断指令`int`。每个BIOS服务为其例程定义自己的编号；要调用例程，必须将特定数字写入每个服务所需的寄存器。所有BIOS中断的列表可在Ralf Brown的中断列表中找到：`http://www.cs.cmu.edu/ ~ralf/files.html`。

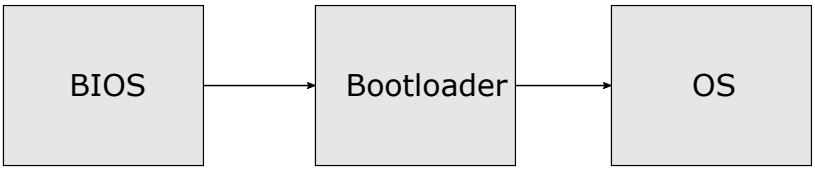


图7.2.1：引导过程。

Example: 中断调用 `13h` (磁盘服务)需要读取的扇区数、磁道号、扇区号、磁头号和驱动器号来从存储设备读取。扇区的内容存储在由寄存器对 `ES:BX` 定义的记忆地址中。参数存储在如下寄存器中：

```
1  ; Store sector content in the buffer 10FF:0000
2  mov dx, 10FFh
3  mov es, dx
4  xor bx, bx
5  mov al, 2 ; read 2 sector
6  mov ch, 0 ; read track 0
7  mov cl, 2 ; 2nd sector is read
8  mov dh, 0 ; head number
9  mov dl, 0 ; drive number. Drive 0 is floppy drive.
10 mov ah, 0x02 ; read floppy sector function
11 int 0x13 ; call BIOS - Read the sector
```

BIOS仅在实模式下可用。然而，当切换到保护模式时，BIOS将无法再使用，并且操作

系统代码负责控制硬件设备。这时操作系统独立运行：它必须提供自己的内核驱动程序以与硬件通信。

7.3 启动过程

1. BIOS通过跳转到0000:7c00h将控制权传递给MBR引导加载程序，假设引导加载程序已经存在。

2. 通过正确初始化段寄存器来设置启动机器环境，以启用平面内存模型。

3. 加载内核：

   (a) 从磁盘读取内核。

   (b) 将其保存在主内存中的某个位置。

   (c) 跳转到内核的起始代码地址并执行。

4. 如果发生错误，打印一条消息通知用户出了些问题并停止执行。

7.4 示例引导加载程序

这里是一个简单的引导加载程序，它什么也不做，除了不会使机器崩溃，而是优雅地停止。如果虚拟机没有停止，而是文本反复闪烁，这意味着引导加载程序没有正确加载，机器崩溃了。机器崩溃是因为它一直执行到物理内存的近端（实模式下为1 MB），即 FFFF:0000h，这会重新启动整个BIOS引导过程。这实际上是一个重置，但不是完全的，因为之前运行的机器环境仍然被保留。因此，它被称为 *warm reboot*。与热启动相反的是 *cold reboot*，在计算机从无电状态启动时，机器环境会重置到初始设置。

bootloader.asm

```
1 ;*****************************************
```

```
; bootloader.asm
; A Simple Bootloader
;****************************************
org 0x7c00
bits 16
start: jmp boot

;; constant and variable definitions
msg db "Welcome to My Operating System!", 0ah, 0dh, 0h

boot:
  cli ; no interrupts
  cld ; all that we need to init
  hlt ; halt the system

; We have to be 512 bytes. Clear the rest of the bytes with
    0
times 510 - ($-$$) db 0
dw 0xAA55    ; Boot Signiture
```

7.5 编译和加载

我们使用 nasm 编译代码并将其写入磁盘镜像:

```
$ nasm -f bin bootloader.asm -o bootloader
```

然后，我们创建一个1.4MB软盘和:

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

Output

```
2880+0 records in
2880+0 records out
1474560 bytes (1.5 MB, 1.4 MiB) copied, 0.00625622 s, 236 MB/s
```

然后，我们将引导加载程序写入1st扇区：

```
$ dd conv=notrunc if=bootloader of=disk.img bs=512
count=1 seek=0
```

Output

```
1+0 records in
1+0 records out
512 bytes copied, 0.000102708 s, 5.0 MB/s
```

选项 conv=notrunc 保留软盘的原始大小。没有此选项，1.4 MB 的磁盘镜像将被全新的 disk.img 完全替换，仅有 512 字节，我们不希望发生这种情况。

过去，开发操作系统很复杂，因为程序员需要了解他使用的特定硬件。尽管x86无处不在，但不同型号之间的细微差别使得为某台机器编写的代码无法在另一台机器上运行。此外，如果你使用相同的物理计算机编写操作系统，运行之间的时间非常长，而且调试也很困难。幸运的是，今天我们可以统一生成具有特定规格的虚拟机，从而完全避免兼容性问题，这使得编写和测试操作系统变得更加容易，因为每个人都可以重现相同的机器环境。

我们将使用 *QEMU*，一个通用且开源的机器仿真器和虚拟化器。QEMU 可以仿真各种类型的机器，不仅限于 x86_64。调试很容易，因为您可以通过 QEMU 内置的 GDB 服务器将 GDB 连接到虚拟机来调试在其上运行的代码。QEMU 可以将 disk.img 作为启动设备，例如软盘：

```
$ qemu-system-i386 -machine q35 -fda disk.img -gdb
tcp::26000 -S
```

▷ 使用选项 -machine q35，QEMU 模拟了英特尔的一个 q35 机器模型。[2]

[2] The following command lists all supported emulated machines from QEMU:

```
qemu-system-i386 -machine help
```

▷ 使用选项 `-fda disk.img`，QEMU 使用 `disk.img` 作为软盘镜像。

▷ 使用选项 `-gdb tcp::26000`，QEMU 允许 `gdb` 通过端口 26000 的 tcp 套接字连接到虚拟机进行远程调试。

▷ 使用选项 `-S`，QEMU 在开始运行之前等待 `gdb` 连接。

执行命令后，将打开一个新的控制台窗口，显示虚拟机的屏幕输出。打开另一个终端，运行 `gdb` 并将当前架构设置为 `i8086`，因为我们正在16位模式下运行:

```
(gdb) set architecture i8086
```

Output

```
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB. Attempting to continue with the default i8086 settings.
The target architecture is assumed to be i8086
```

Then, connect `gdb` to the waiting virtual machine with this command:

```
(gdb) target remote localhost:26000
```

Output

```
Remote debugging using localhost:26000
0x0000fff0 in ?? ()
```

Then, place a breakpoint at `0x7c00`:

```
(gdb) b *0x7c00
```

Output

```
Breakpoint 1 at 0x7c00
```

注意星号前的内存地址。没有星号，`gdb`将地址视为程序中的符号，而不是地址。

然后，为了方便，我们使用分割布局来一起查看汇编代码和寄存器：

```
(gdb) layout asm
(gdb) layout reg
```

最后，运行程序：

```
(gdb) c
```

如果虚拟机成功运行引导加载程序，QEMU屏幕应该看起来像这样：

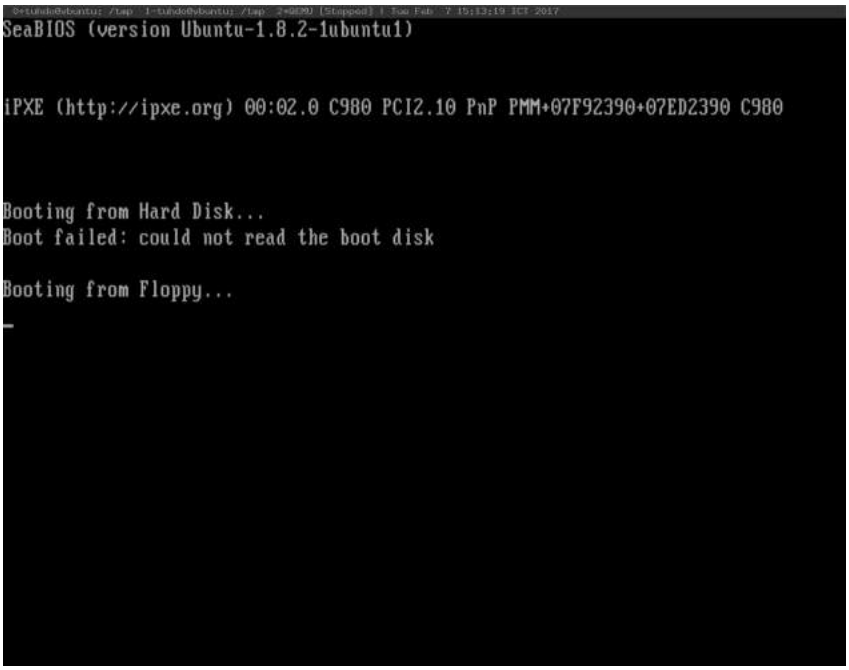


Figure 7.5.1: Boot succeeded.

7.5.1 Debugging

如果由于某种原因，样本引导加载程序无法到达此类屏幕，并且 gdb 在 0x7c00 处没有停止，那么以下情况很可能是：

▷ 引导加载程序无效：在软盘引导时出现消息“引导失败：不是可引导磁盘”。请确保引导签名位于第一个扇区的最后2个字节。

▷ 机器找不到启动盘：在软盘启动时出现“启动失败：不是可启动磁盘”的消息。请确保引导加载程序正确写入第一个扇区。可以通过使用 hd 检查磁盘来验证：

```
$ hd disk.img | less
```

如果前512个字节全部为零，那么很可能是引导加载程序被错误地写入到另一个扇区。

▷ 机器崩溃：当这种情况发生时，它会重置回开始位置 FFFF:0000h。如果 QEMU 机器在没有等待 gdb 的情况下启动，那么控制台输出窗口会不断闪烁，因为机器被反复重置。这很可能是引导加载程序代码中的某些指令导致了故障。

练习7.5.1. 打印欢迎信息

我们成功加载了引导加载程序。但是，它需要做一些除了使我们的机器停止之外的有用的事情。最容易做的事情是在屏幕上打印一些东西，就像所有编程语言的介绍都是从“Hello World”开始的。我们的引导加载程序打印“欢迎使用我的操作系统”。在这一部分，我们将构建一个简单的I/O库

[3] Or whatever message you want.

允许我们在屏幕上的任何位置设置光标并在那里打印文本。

首先，创建一个 io.asm 文件用于 I/O 相关例程。然后，编写以下例程：

1. MovCursor

目的：将光标移动到屏幕上的特定位置并记住此位置。

参数：

▷ bh = Y坐标
▷ bl = X坐标。

返回：无

2. `PutChar`

   目的：在屏幕上打印字符，位置由之前由 `MovCursor` 设置的光标位置确定。

   参数：

   ▷ `al` = 字符要打印 ▷ `bl` = 文本颜色 ▷ `cx` = 字符重复的次数

   Return: 无

3. `Print`

   目的：打印一个字符串。

   参数：

   ▷ `ds:si` = 零终止字符串

   Return: 无

测试这些例程，将每个放入引导加载程序源代码中，编译并运行。要调试，运行GDB并在特定例程处设置断点。最终结果是`Print`应在屏幕上显示欢迎信息。

7.6 从引导加载程序加载程序

现在我们已经了解了如何使用BIOS服务的感觉，是时候做一些更复杂的事情了。我们将把内核放置在2^{nd}扇区之后，我们的引导加载程序从2^{nd}扇区开始读取30个扇区。为什么是30个扇区？我们的内核将逐渐增长，因此我们将保留30个扇区，并在内核大小每次增加一个扇区时节省修改引导加载程序的时间。

引导加载程序的主要责任是从某个存储设备（例如硬盘）读取操作系统，然后将其加载到主内存中，并将控制权转移到已加载的操作系统，类似于

如何BIOS读取和加载引导加载程序。目前，我们的引导加载程序所做的只是由BIOS加载的一个汇编程序。要使我们的引导加载程序成为真正的引导加载程序，它必须出色地完成上述两项任务：*read*和*load*一个操作系统。

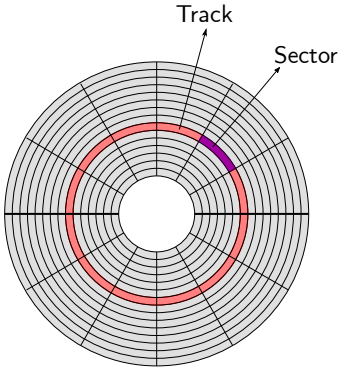
7.6.1 Floppy Disk Anatomy

要从存储设备中读取，我们必须了解设备的工作原理以及控制它的接口。首先，软盘是一种存储设备，类似于RAM，但即使在计算机关闭时也能存储信息，因此被称为*persistent storage device*。软盘

persistent storage device

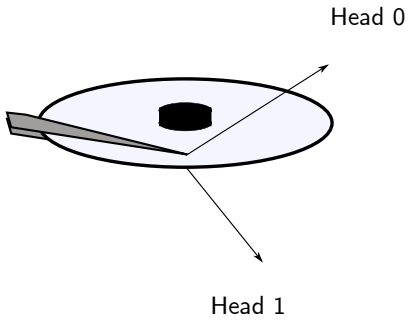
磁盘也是一个持久存储设备，因此它提供高达1.4MB的存储空间，或1,474,560字节。从软盘读取时，可以读取的最小单位是一个*sector*，即一组512个连续的字节。一组18个扇区是一个*track*。软盘的每一面由80个磁道组成。读取软盘需要软盘驱动器。软盘驱动器内部包含一个带有2个*heads*的臂，每个磁头读取软盘的一侧；磁头0写入软盘的上侧，磁头1写入软盘的下侧。

图7.6.1：区域和轨道。



当软盘驱动器向全新的软盘写入数据时，首先写入上侧的0磁道，由0磁头完成。当上侧的0磁道写满后，由1磁头使用下侧的0磁道。当0磁道的上侧和下侧都写满后，它将回到0磁头再次写入数据，但这次是1磁道的上侧，依此类推，直到设备上没有剩余空间。相同的程序也适用于从软盘读取数据。

图7.6.2：两面软盘盘片。



7.6.2 Read and load sectors from a floppy disk

首先，我们需要编写一个示例程序来写入2nd扇区，这样我们就可以在软盘读取上进行实验：

sample.asm

```
1 ;*****************************************
2 ; sample.asm
3 ; A Sample Program
```

```
4 ;*****************************************
5 mov eax, 1
6 add eax, 1
```

这样一个程序已经足够好了。为了简化并且为了演示的目的，我们将使用同一个包含引导加载程序的软盘来存放我们的操作系统。操作系统镜像从2nd扇区开始，因为1st扇区已经被引导加载程序占用。我们使用dd编译并写入到2nd扇区：

```
$ nasm -f bin sample.asm -o sample
$ dd if=sample of=disk.img bs=512 count=1 seek=1
```

1st sector	**2nd sector**	.....	**30th sector**
bootloader	sample	....	(empty)

图7.6.3：软盘上的引导加载程序和示例程序。

接下来，我们需要修复引导加载程序以从软盘读取并加载多个任意扇区。在这样做之前，需要基本了解软盘。要从磁盘读取数据，使用中断13和AH = 02是将磁盘扇区读取到内存中的常规操作：

```
AH = 02
AL = number of sectors to read (1-128 dec.)
CH = track/cylinder number (0-1023 dec., see below)
CL = sector number (1-17 dec.)
DH = head number (0-15 dec.)
DL = drive number (0=A:, 1=2nd floppy, 80h=drive 0, 81h=drive 1)
ES:BX = pointer to buffer
Return:
   AH = status (see INT 13,STATUS)
   AL = number of sectors read
   CF = 0 if successful
      = 1 if error
```

应用上述程序，引导加载程序可以读取2nd扇区：

bootloader.asm

```
;*****************************************
; Bootloader.asm
; A Simple Bootloader
;*****************************************
org 0x7c00
bits 16
start: jmp boot

;; constant and variable definitions
msg db "Welcome to My Operating System!", 0ah, 0dh, 0h

boot: cli ; no interrupts
cld ; all that we need to init

mov ax, 0x50
;; set the buffer
mov es, ax
xor bx, bx

mov al, 2      ; read 2 sector
mov ch, 0      ; track 0
mov cl, 2      ; sector to read (The second sector)
mov dh, 0      ; head number
mov dl, 0      ; drive number

mov ah, 0x02   ; read sectors from disk
int 0x13       ; call the BIOS routine
jmp 0x50:0x0   ; jump and execute the sector!

hlt ; halt the system

; We have to be 512 bytes. Clear the rest of the bytes
```

```
    with 0
35  times 510 - ($-$$) db 0
36  dw 0xAA55     ; Boot Signiture
```

上述代码跳转到地址 0x50:00 (，该地址是 0x500)。为了测试代码，将其加载到 QEMU 虚拟机上，并通过 gdb 连接，然后在 0x500 处设置断点。如果 gdb 在该地址停止，并且汇编列表中的代码与 sample.asm 中的相同，则引导加载程序成功加载了程序。这是一个重要的里程碑，因为我们确保我们的操作系统被正确加载并运行。

7.7 使用脚本提高生产力

7.7.1 Automate build with GNU Make

到这一点为止，整个开发过程感觉重复：每次进行更改时，都要再次输入相同的命令。命令也很复杂。Ctrl+r有所帮助，但仍感觉繁琐。

GNU Make 是一个控制并自动化构建复杂软件过程的程序。对于像单个 C 源文件这样的小型程序，调用 *gcc* 是快速且简单的。然而，很快你的软件将变得更加复杂，涉及多个目录中的多个文件，手动构建和链接文件将变得繁琐。为了解决这个问题，创建了一个工具来自动化这个问题，被称为 *build system*。GNU Make 就是这样的工具之一。虽然存在各种构建系统，但 GNU Make 在 Linux 世界中最为流行，因为它用于构建 Linux 内核。

对于make的全面介绍，请参阅官方《Make简介》：https://www.gnu.org/software/make/manual/html_node/Introduction.html#Introduction。对我们项目来说，这些就足够了。您还可以从官方手册页面下载不同格式的手册，例如PDF：https://www.gnu.org/software/ make/manual/ 。

使用Makefile，我们可以构建更简单的命令并节省时间：

Makefile

```
all: bootloader bootdisk

bootloader:
    nasm -f bin bootloader.asm -o bootloader.o

kernel:
    nasm -f bin sample.asm -o sample.o

bootdisk: bootloader.o kernel.o
    dd if=/dev/zero of=disk.img bs=512 count=2880
    dd conv=notrunc if=bootloader.o of=disk.img bs=512
        count=1 seek=0
    dd conv=notrunc if=sample.o of=disk.img bs=512 count=1
        seek=1
```

现在，只需一个命令，我们就可以从头到尾构建一个带有引导加载程序在1st扇区以及示例程序在2nd扇区的磁盘镜像：

```
$ make bootdisk
```

Output

```
nasm -f bin bootloader.asm -o bootloader.o
nasm -f bin sample.asm -o bootloader.o
dd if=/dev/zero of=disk.img bs=512 count=2880
2880+0 records in
2880+0 records out
1474560 bytes (1.5 MB, 1.4 MiB) copied, 0.00482188 s, 306 MB/s
dd conv=notrunc if=bootloader.o of=disk.img bs=512 count=1 seek=0
0+1 records in
0+1 records out
10 bytes copied, 7.0316e-05 s, 142 kB/s
dd conv=notrunc if=sample.o of=disk.img bs=512 count=1 seek=1
0+1 records in
0+1 records out
```

```
10 bytes copied, 0.000208375 s, 48.0 kB/s
```

查看Makefile，我们可以看到一些问题：

首先，名称 disk.img 遍布各处。当我们想要更改磁盘映像名称，例如 floppy_disk.img 时，所有包含名称 disk.img 的地方都必须手动更改。为了解决这个问题，我们使用一个变量，并且将 disk.img 的每一次出现都替换为对该变量的引用。这样，只需更改一个地方——变量定义——其他所有地方都会自动更新。以下变量被添加：

```
BOOTLOADER=bootloader.o
OS=sample.o
DISK_IMG=disk.img.o
```

第二个问题是，名称 bootloader 和 sample 出现在源文件的文件名中，例如 bootloader.asm 和 sample.asm，以及二进制文件的文件名中，例如 bootloader 和 sample。与 disk.img 类似，当名称更改时，该名称的所有引用都必须手动更改，包括源文件和二进制文件的名称，例如，如果我们把 bootloader.asm 改为 loader.asm，那么目标文件 bootloader.o 需要改为 loader.o。为了解决这个问题，我们不是手动更改文件名，而是创建一个规则来自动生成一个扩展名到另一个扩展名的文件名。在这种情况下，我们希望任何以 .asm 开头的源文件都有其等效的二进制文件，没有任何扩展名，例如 bootloader.asm → bootloader.o。这种转换很常见，因此 GNU Make 提供了内置函数：wildcard 和 patsubst 来解决这类问题：

```
BOOTLOADER_SRCS := $(wildcard *.asm)
BOOTLOADER_OBJS := $(patsubst %.asm, %.o, $(BOOTLOADER_SRCS
    ))
```

wildcard 匹配当前目录中的任何 .asm 文件，然后将匹配到的文件列表赋值给变量 BOOTLOADER_SRCS。在这种情况下，BOOTLOADER_SRCS 被赋予以下值：

```
bootloader.asm sample.asm
```

patsubst 替换以 .asm 开头的任何文件名到文件名 .o，例如 bootloader.asm → bootloader.o。在 patsubsts 运行后，我们得到 BOOTLOADER_OBJS 中的对象文件列表：

```
bootloader.o sample.o
```

最后，需要从 .asm 到 .o 的构建配方：

```
%.o: %.asm
    nasm -f bin $< -o $@
```

▷ $< 是一个特殊变量，它引用了食谱的输入：%.asm。

▷ $@ 是一个特殊变量，它引用了菜谱的输出：%.o。

当执行配方时，变量会被替换为实际值。例如，如果转换是 bootloader.asm → bootloader.o，那么在配方中替换占位符时实际执行的命令是：

```
nasm -f bin bootloader.asm -o bootloader.o
```

使用该配方，所有 .asm 文件都通过 nasm 命令自动构建成 .o 文件，我们不再需要为每个对象文件单独编写配方。将所有内容与新变量结合，我们得到一个更好的 Makefile：

Makefile

```
1 BOOTLOADER=bootloader.o
2 OS=sample.o
3 DISK_IMG=disk.img
4
5 BOOTLOADER_SRCS := $(wildcard *.asm)
6 BOOTLOADER_OBJS := $(patsubst %.asm, %.o, $(BOOTLOADER_SRCS
      ))
```

```
7
8  all: bootdisk
9
10 %.o: %.asm
11     nasm -f bin $< -o $@
12
13 bootdisk: $(BOOTLOADER_OBJS)
14     dd if=/dev/zero of=$(DISK_IMG) bs=512 count=2880
15     dd conv=notrunc if=$(BOOTLOADER) of=$(DISK_IMG) bs=512
           count=1 seek=0
16     dd conv=notrunc if=$(OS) of=$(DISK_IMG) bs=512 count=1
           seek=1
```

从现在开始，任何 .asm 文件都将自动编译，无需为每个文件指定明确的配方。

目标文件与源文件位于同一目录中，这使得在处理源树时更加困难。理想情况下，目标文件和源文件应位于不同的目录中。我们希望有一个更好的目录布局，如图7.7.1所示。

图7.7.1：更好的项目布局

```
.
├── bootloader
│   ├── bootloader.asm
│   └── Makefile
├── build
│   ├── bootloader
│   │   └── bootloader.o
│   ├── disk.img
│   └── os
│       └── sample.o
├── Makefile
└── os
    ├── Makefile
    └── sample.asm
```

布局可以使用 tree 命令显示：$ tree

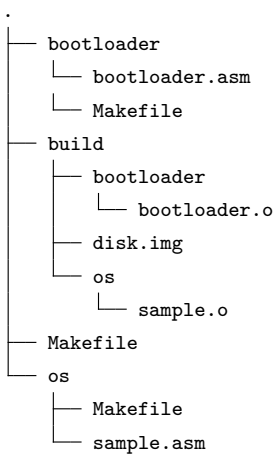
bootloader/ 目录包含引导加载程序源文件；os/包含我们将要编写的操作系统的源文件；build/包含引导加载程序、操作系统和最终磁盘映像disk.img的对象文件。请注意，bootloader/目录也有自己的Makefile。这个Makefile将负责构建bootloader/目录中的所有内容，而顶级Makefile则从构建引导加载程序的负担中解脱出来，但只构建磁盘映像。bootloader/目录中的Makefile内容应为：

bootloader/Makefile

```
1  BUILD_DIR=../build/bootloader
2
3  BOOTLOADER_SRCS := $(wildcard *.asm)
4  BOOTLOADER_OBJS := $(patsubst %.asm, $(BUILD_DIR)/%.o, $(
       BOOTLOADER_SRCS))
5
```

```
6 all: $(BOOTLOADER_OBJS) 7
8 $(BUILD_DIR)/%.o: %.asm 9
nasm -f bin $< -o $@
```

图7.7.2：bootloader/中的Makefile



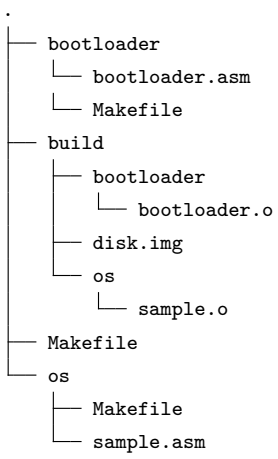
基本上，与顶层Makefile中的引导加载程序相关的所有内容都被提取到这个Makefile中。当make运行此Makefile时，bootloader.o应该被构建并放入../build/目录。作为一个好的实践，所有对../build/的引用都应该通过BUILD_DIR变量进行。从.asm → .o转换的配方也更新了正确的路径，否则将无法工作。

▷ %.asm 引用当前目录中的汇编源文件 ectory.

▷ $(BUILD_DIR)/%.o 引用构建目录中路径 ../build/ 下的输出目标文件。

整个配方实现了从 <source_file.asm> → ../build/<object_file.o> 的转换。请注意，所有路径都必须正确。如果我们尝试在不同的目录中构建对象文件，例如当前目录，它将不会工作，因为没有这样的配方来在这样一个路径下构建对象。

图7.7.3：os/中的Makefile



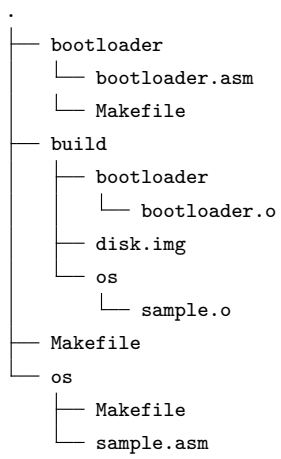
我们也为 os/ 目录创建了一个类似的 Makefile：

os/Makefile

```
1 BUILD_DIR=../build/os
2
3 OS_SRCS := $(wildcard *.asm)
4 OS_OBJS := $(patsubst %.asm, $(BUILD_DIR)/%.o, $(OS_SRCS))
5
6 all: $(OS_OBJS)
7
8 $(BUILD_DIR)/%.o: %.asm
9     nasm -f bin $< -o $@
```

现在，它看起来几乎与引导加载程序的Makefile完全相同。在下一章中，我们将为C代码更新它。然后，我们更新顶层Makefile：

图7.7.4：顶级Makefile



Makefile

```
1  BUILD_DIR=build
2  BOOTLOADER=$(BUILD_DIR)/bootloader/bootloader.o
3  OS=$(BUILD_DIR)/os/sample.o
4  DISK_IMG=disk.img
5
6  all: bootdisk
7
8  .PHONY: bootdisk bootloader os
9
10 bootloader:
11     make -C bootloader
12
13 os:
14     make -C os
15
16 bootdisk: bootloader os
17     dd if=/dev/zero of=$(DISK_IMG) bs=512 count=2880
18     dd conv=notrunc if=$(BOOTLOADER) of=$(DISK_IMG) bs=512
           count=1 seek=0
19     dd conv=notrunc if=$(OS) of=$(DISK_IMG) bs=512 count=1
           seek=1
```

构建过程现在真正模块化了：

- `bootloader` 现在 `os` 构建 已委托给相应组件的子 Makefile。`-C` 选项告诉 `make` 在提供的目录中使用 Makefile 执行。在这种情况下，目录是 `bootloader/` 和 `os/`。

- 目标 `all` 的顶级 Makefile 只负责 `bootdisk` 目标，这是本 Makefile 的主要目标。

在很多情况下，目标不总是文件名，而只是一个在请求时始终要执行的菜谱的名称。如果文件名与目标同名且文件是最新的，make不会执行目标。为了解决这个问题，.PHONY指定某些目标不是文件。然后，所有虚假目标在请求时都会运行，无论同名文件是否存在。

为了节省输入启动QEMU虚拟机命令的时间，我们还在顶层Makefile中添加了一个目标：

```
qemu:
    qemu-system-i386 -machine q35 -fda $(DISK_IMG) -gdb tcp
        ::26000 -S
```

最后一个问题是项目清理。目前，需要手动删除目标文件，这是一个重复的过程。相反，让每个组件的Makefile负责清理其目标文件，然后顶层Makefile通过调用组件Makefile来完成项目清理任务。每个Makefile在末尾添加了一个clean目标：

▷ 引导加载程序 Make 文件：

```
clean:
    rm $(BUILD_DIR)/*
```

▷ OS Makefile:

```
clean:
    rm $(BUILD_DIR)/*
```

▷ 顶级 Makefile：

```
clean:
    make -C bootloader clean
    make -C os clean
```

只需在项目根目录下调用 make clean，所有对象文件都将被删除。

7.7.2 GNU Make Syntax summary

GNU Make，在其核心，是一种针对构建自动化的领域特定语言。像任何编程语言一样，它需要一种定义数据和代码的方法。在Makefile中，变量携带数据。变量的值要么是硬编码的，要么是从调用Bash等shell中评估得到的。Make中的所有变量值具有相同的类型：文本字符串。数字3不是一个数字，而是符号3的文本表示。以下是在Makefile中定义数据的一些常见方法：

Syntax	**Description**

`A = 1`
`B = 2`
`C = $$(expr $(A) + $(B))`
⇒ `A` is `1`, `B` is `2`, `C` is `3`.
	Declare a variable and assign a textual value to it. the double dollar sign `$$` means the enclosing expression evaluating by a shell, defined by `/bin/sh`. In this case, the enclosing expression is `(expr $(A) + $(B))` and is evaluated by Bash.

`PATH = /bin`
`PATH := $PATH:/usr/bin`
⇒ `PATH` is `/bin/:/usr/bin`
	Declare a variable and assign to it. However, the difference is that the `=` syntax does not allow refer to a variable to use itself as a value in the right hand side, while this syntax does.

`PATH = /bin`
`PATH += /usr/bin`
⇒ `PATH` is `/bin/:/usr/bin`
	Append a new value at the end of a variable. Equivalent to:
	`PATH := $PATH:/usr/bin`

`CFLAGS ?= -o` ⇒ `CFLAGS` is assigned the value `-o` if it was not defined.	This syntax is called **conditional reference**. Set a variable to a value if it is undefined. This is useful if a user wants to supply different value for a variable from the command line e.g. add debugging option to `CFLAGS`. Otherwise, Make uses the default defined by `?=`.

`SRCS = lib1.c lib2.c main.c`
`OBJS := $(SRC:.o=.c)`
⇒ `OBJS` has the value `lib1.o lib2.o main.o`
	This syntax is called **substitution reference**. A part of referenced variable is replaced with something else. In this case, all the `.c` extension is replaced by `.o` extension, thus creating a list of object files for `OBJS` variable from the list of source files from `SRCS` variable.

代码在GNU Make中是一系列它可以运行的食谱。每个食谱类似于编程语言中的一个函数，可以像常规函数一样调用。每个食谱包含一系列由shell（例如Bash）执行的shell命令。食谱具有以下格式：

```
target: prerequisites
        command
```

每个 `target` 都类似于一个函数名。每个 `prerequisite` 是调用另一个目标的调用。每个命令是 Make 的内置命令或由 shell 可执行的命令。在进入 `target` 的主体之前，必须满足所有先决条件；也就是说，每个先决条件都不能返回任何错误。如果返回任何错误，Make 将终止整个构建过程并在命令行上打印错误。

每次运行 `make`，默认情况下如果没有提供目标，它将从 `all` 目标开始，通过每个先决条件，最后执行 `all` 的主体。`all` 在其他编程语言中类似于 `main`。然而，如果提供了 `make` 目标，它将从该目标开始，而不是 `main`。此功能有助于自动化项目中的多个方面。例如，一个目标是构建项目，一个目标是生成文档，例如测试报告，另一个目标是运行整个测试套件，`all` 运行所有主要目标。

7.7.3 Automate debugging steps with GDB script

为了方便，我们将 GDB 配置保存到项目根目录下的 `.gdbinit` 文件中。此配置仅是一组 GDB 命令和一些额外的命令。当 `gdb` 运行时，它首先加载 `.gdbinit`

文件在主目录中，然后在当前目录的 .gdbinit 文件。为什么我们不应该把命令放在 ~/.gdbinit 中？因为这些命令仅针对这个项目特定，例如，并非所有程序都需要远程连接。

我们的第一个配置：

.gdbinit

```
1 define hook-stop
2     # Translate the segment:offset into a physical address
3     printf "[%4x:%4x] ", $cs, $eip
4     x/i $cs*16+$eip
5 end
```

上述脚本以 [segment:offset] 格式显示内存地址，这对于调试我们的引导加载程序和操作系统代码是必要的。

使用Intel语法更好：

```
set disassembly-flavor intel
```

以下命令设置了一个更方便的调试汇编代码的布局：

```
layout asm
layout reg
```

我们目前正在调试引导加载程序代码，因此首先将其设置为16位是个好主意：

```
set architecture i8086
```

每次启动QEMU虚拟机时，gdb必须始终连接到端口26000。为了避免手动连接虚拟机的麻烦，请添加以下命令：

```
target remote localhost:26000
```

调试引导加载程序需要在0x7c00处设置断点，这是我们的引导加载程序代码开始的地方：

```
b *0x7c00
```

现在，每当 gdb 启动时，它将自动根据代码设置正确的架构，自动连接到虚拟机4，显示输出-

将代码放入方便的布局并设置必要的断点。所需做的就是运行程序。

4 QEMU 虚拟机应该在启动 gdb 之前已经启动。

8

Linking and loading on bare metal

Relocation 是替换符号引用为其实际对象文件中的符号定义。符号引用是符号的内存地址。

Relocation

如果定义难以理解，可以考虑一个类似的类比：搬家。假设一个程序员买了一栋新房，新房是空的。他必须购买家具和电器来满足日常需求，因此他列出了一个购买物品清单以及它们放置的位置。为了可视化新物品的放置，他绘制了房子的蓝图以及所有物品的相应位置。然后他前往商店购买商品。每次他访问商店并看到匹配的物品时，他就告诉店主记下来。完成选择后，他就告诉店主取一个全新的物品而不是展示的物品，然后给出送货到新房的地址。最后，当商品到达时，他将物品放置在最初计划的位置。

现在房屋搬迁已明确，物体搬迁类似：

▷ 项目列表表示重定位表，其中每个符号（项目）的内存位置是预先确定的。▷每个项目代表一对*symbol definition*及其*symbol address*。

▷ 每个商店代表一个编译后的对象文件。

▷ 每个展品代表一个符号定义和对象文件中的引用。

▷ 新地址，所有商品都交付于此，代表最终的可执行二进制文件或最终目标文件。由于显示的商品不对外出售，店主会提供全新的商品。同样，目标文件也不会合并在一起，而是全部复制到一个新的文件中，即目标/可执行文件。

▷ 最后，根据从开始制作的购物清单将商品放置在相应位置。同样，符号定义被适当地放置在其各自的章节中，并且最终对象/可执行文件的符号引用被替换为符号定义的实际内存地址。

8.1 理解readelf中的重定位

之前，当我们探索对象部分时，存在以 .rel 开始的部分。这些部分是重定位表，它们映射了符号与其在最终对象文件或最终可执行二进制文件中的位置之间的关系。

[1] A .rel section is equivalent to a list of items in the house analogy.

假设在另一个对象文件中定义了一个函数 foo，因此 main.c 将其声明为 extern：

main.c

```
int i;
void foo();
int main(int argc, char *argv[])
{
    i = 5;
    foo();
    return 0;
}

void foo() {}
```

当我们将 main.c 编译成目标文件时使用此命令：

```
$ gcc -m32 -masm=intel -c main.c
```

T然而，我们可以使用此命令检查重定位表 和
：`$ readelf -r main.o`

```
```

输出：

Output

```
Relocation section '.rel.text' at offset 0x1cc contains 2 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000013  00000801 R_386_32          00000004   i
0000001c  00000a02 R_386_PC32        0000002e   foo
Relocation section '.rel.eh_frame' at offset 0x1dc contains 2 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000020  00000202 R_386_PC32        00000000   .text
0000004c  00000202 R_386_PC32        00000000   .text
```

8.1.1 Offset

一个 *offset* 是二进制文件中某个部分的定位点，其中实际 ***offset***
内存地址的符号定义被替换。具有 `.rel` 前缀的部分确定要偏移到哪个部分。例如，`.rel.text` 是需要纠正地址的符号的 *table* 重定位，在 `.text` 部分的特定偏移 `.text` 部分中。在示例输出中：

输出

```
0000001c  00000a02 R_386_PC32        0000002e   foo
```

蓝色数字表示存在一个符号 `foo` 的引用，它在 `.text` 节区中 `1c` 字节处。为了更清楚地看到它，我们使用选项 -g 重新编译 main.c 到文件 `main_debug.o`，然后对其运行 `objdump` 并得到：

Output

```
Disassembly of section .text:
00000000 <main>:
int i;
void foo();
```

```
int main(int argc, char *argv[])
{
   0:   8d 4c 24 04             lea    ecx,[esp+0x4]
   4:   83 e4 f0                and    esp,0xfffffff0
   7:   ff 71 fc                push   DWORD PTR [ecx-0x4]
   a:   55                      push   ebp
   b:   89 e5                   mov    ebp,esp
   d:   51                      push   ecx
   e:   83 ec 04                sub    esp,0x4
    i = 5;
  11:   c7 05 00 00 00 00 05    mov    DWORD PTR ds:0x0,0x5
  18:   00 00 00
    foo();
  1b:   e8 fc ff ff ff          call   1c <main+0x1c>
    return 0;
  20:   b8 00 00 00 00          mov    eax,0x0
}
  25:   83 c4 04                add    esp,0x4
  28:   59                      pop    ecx
  29:   5d                      pop    ebp
  2a:   8d 61 fc                lea    esp,[ecx-0x4]
  2d:   c3                      ret
....irrelevant content omitted....
```

字节在 1b 处是操作码 e8，call 指令；字节在 1c 处是值 fc。为什么 e8 的操作数值是 0xfffffffc，即 -4，但翻译后的指令 call 1c？这将在接下来的几节中解释，但你应该停下来思考一下原因。

8.1.2 Info

Info指定符号表中符号的索引和要执行的重定位类型。

输出

```
0000001c  00000a02 R_386_PC32        0000002e   foo
```

粉色数字是符号 foo 在符号表中的索引，绿色数字是重定位类型。这些数字以十六进制格式编写。在示例中，0a 表示十进制中的 10，符号 foo 确实位于索引 10：

输出

```
10: 0000002e     6 FUNC    GLOBAL DEFAULT    1 foo
```

8.1.3 Type

类型表示文本形式的类型值。查看foo的类型：

输出

```
0000001c  00000a02 R_386_PC32        0000002e   foo
```

绿色数字以数字形式输入，R_386_PC32 是分配给该值的名称。每个值代表一种计算重定位方法。例如，使用类型 R_386_PC32，应用以下重定位公式（Inteli386 psABI）：

$$Relocated\,Offset = S + A - P$$

为了理解该公式，有必要了解符号值。

8.1.4 Sym.Value

此字段显示 *symbol value*。符号值是指分配给符号的值，其含义取决于 Ndx 字段：

一个其部分索引为 COMMON, 的符号，其符号值包含对齐约束。

示例8.1.1。在符号表中，变量i被标识为COM (未初始化变量)：2

2 列出符号表的命令是（假设目标文件是 hello.o）：

```
readelf -s hello.o
```

Output

```
Symbol table '.symtab' contains 16 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS hello2.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    5
     6: 00000000     0 SECTION LOCAL  DEFAULT    7
     7: 00000000     0 SECTION LOCAL  DEFAULT    8
     8: 00000000     0 SECTION LOCAL  DEFAULT   10
     9: 00000000     0 SECTION LOCAL  DEFAULT   12
    10: 00000000     0 SECTION LOCAL  DEFAULT   14
    11: 00000000     0 SECTION LOCAL  DEFAULT   15
    12: 00000000     0 SECTION LOCAL  DEFAULT   13
    13: 00000004     4 OBJECT  GLOBAL DEFAULT  COM i
    14: 00000000    46 FUNC    GLOBAL DEFAULT    1 main
    15: 0000002e     6 FUNC    GLOBAL DEFAULT    1 foo
```

因此，其符号值是一个内存对齐，用于分配一个符合最终内存地址对齐的适当内存地址。在 `i` 的情况下，其值为 4，因此 `i` 在最终二进制文件中的起始内存地址将是 4 的倍数。

一个符号，其中 `Ndx` 识别一个特定部分,，其符号值包含部分偏移量。

示例8.1.2。在符号表中，`main`和`foo`属于第1节：

Output

```
    14: 00000000    46 FUNC    GLOBAL DEFAULT    1 main
    15: 0000002e     6 FUNC    GLOBAL DEFAULT    1 foo
```

这是 `.text`[3] 部分[4]：

[3] `.text` 存储程序代码和只读数据。[4] 列出段的命令是（假设目标文件是 `hello.o`）：

```
readelf -S hello.o
```

Output

```
There are 20 section headers, starting at offset 0x558:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 000034 00  AX  0   0  1
  [ 2] .rel.text         REL             00000000 000414 000010 08   I 18   1  4
  [ 3] .data             PROGBITS        00000000 000068 000000 00  WA  0   0  1
  [ 4] .bss              NOBITS          00000000 000068 000000 00  WA  0   0  1
  [ 5] .debug_info       PROGBITS        00000000 000068 000096 00      0   0  1
..... remaining output omitted for clarity....
```

在最终的可执行文件和共享对象文件中,, 而不是上述值, 符号值包含一个内存地址。

示例8.1.3。将hello.o编译成最终的可执行文件hello后, 符号表现在包含每个符号的内存地址: 5:

5 编译对象文件 `hello.o` 成可执行文件 `hello` 的命令:

```
gcc -g -m32 -masm=intel hello.o -o hello
```

Output

```
Symbol table '.symtab' contains 75 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 08048154     0 SECTION LOCAL  DEFAULT    1
     2: 08048168     0 SECTION LOCAL  DEFAULT    2
     3: 08048188     0 SECTION LOCAL  DEFAULT    3
....output omitted...
    64: 08048409     6 FUNC    GLOBAL DEFAULT   14 foo
    65: 0804a020     0 NOTYPE  GLOBAL DEFAULT   26 _end
    66: 080482e0     0 FUNC    GLOBAL DEFAULT   14 _start
    67: 08048488     4 OBJECT  GLOBAL DEFAULT   16 _fp_hw
    68: 0804a01c     4 OBJECT  GLOBAL DEFAULT   26 i
    69: 0804a018     0 NOTYPE  GLOBAL DEFAULT   26 __bss_start
    70: 080483db    46 FUNC    GLOBAL DEFAULT   14 main
...ouput omitted...
```

Unlike the values of the symbols `foo`, `i` and `main` as in the hello.o object file, the complete memory addresses are in place.

现在只需了解重定位类型。之前，我们提到了类型 `R_386_PC32`。以下公式适用于重定位（Inteli386 psABI）：

$$RelocatedOffset = S + A - P$$

哪里

S 表示符号的值。在最终的可执行二进制文件中，它是符号的地址。

A 表示加数，一个额外添加到符号值的值。

P 表示要固定的内存地址。

Relocate Offset 是移动位置6与之间的距离

实际符号定义的内存位置或内存地址。

6 其中要固定的引用内存地址。

但是为什么我们浪费时间计算距离而不是用直接内存地址替换呢？原因是x86架构没有使用任何使用绝对内存地址的寻址模式，如表4.5.2中列出。x86中的所有寻址模式都是相对的。在某些汇编语言中，可以使用绝对地址，因为这只是一个语法糖，后来由汇编器将其转换为x86硬件提供的相对寻址模式之一。

示例8.1.4。对于`foo`符号：

Output

```
0000001c  00000a02 R_386_PC32        0000002e   foo
```

The distance between the usage of `foo` in `main.o` and its definition, applying the formula $S + A - P$ is: $\texttt{2e} + \texttt{0} - \texttt{1c} = \texttt{12}$. That is, the place where memory fixing starts is `0x12` or 18 bytes away from *the definition* of the symbol `foo`. However, to make an instruction works properly, we must also subtract 4 from `0x12` and results in `0xe`. Why the extra `-4`? Because the relative address starts at *the end* of an instruction, *not the*

*address where memory fixing starts*因此，我们还必须排除覆盖地址的4个字节。

确实，查看对象文件 `hello.o` 的 `objdump` 输出：

Output

```
Disassembly of section .text:
00000000 <main>:
   0:   8d 4c 24 04             lea    ecx,[esp+0x4]
   4:   83 e4 f0                and    esp,0xfffffff0
   7:   ff 71 fc                push   DWORD PTR [ecx-0x4]
   a:   55                      push   ebp
   b:   89 e5                   mov    ebp,esp
   d:   51                      push   ecx
   e:   83 ec 04                sub    esp,0x4
  11:   c7 05 00 00 00 00 05    mov    DWORD PTR ds:0x0,0x5
  18:   00 00 00
  1b:   e8 fc ff ff ff          call   1c <main+0x1c>
  20:   b8 00 00 00 00          mov    eax,0x0
  25:   83 c4 04                add    esp,0x4
  28:   59                      pop    ecx
  29:   5d                      pop    ebp
  2a:   8d 61 fc                lea    esp,[ecx-0x4]
  2d:   c3                      ret
0000002e <foo>:
  2e:   55                      push   ebp
  2f:   89 e5                   mov    ebp,esp
  31:   90                      nop
  32:   5d                      pop    ebp
  33:   c3                      ret
```

内存固定开始的地址在操作码 `e8` 之后，使用模拟值 `fc ff ff ff`，十进制表示为 `-4`。然而，汇编代码中显示的值是 `1c`。紧接在 `e8` 之后的内存地址。原因是指令 `e8` 从 `1b` 开始，到 $_{20}$ 结束。

指令的结束是其最后一个操作数之后的内存地址。整个指令 `e8` 从地址 `1b` 扩展到地址 `1f`。

`-4` 表示指令末尾向前4个字节，即：$20 - 4 = 1c$。链接后，最终可执行文件的输出显示实际的内存定位：

Output

```
080483db <main>:
 80483db:       8d 4c 24 04             lea    ecx,[esp+0x4]
 80483df:       83 e4 f0                and    esp,0xfffffff0
 80483e2:       ff 71 fc                push   DWORD PTR [ecx-0x4]
 80483e5:       55                      push   ebp
 80483e6:       89 e5                   mov    ebp,esp
 80483e8:       51                      push   ecx
 80483e9:       83 ec 04                sub    esp,0x4
 80483ec:       c7 05 1c a0 04 08 05    mov    DWORD PTR ds:0x804a01c,0x5
 80483f3:       00 00 00
 80483f6:       e8 0e 00 00 00          call   8048409 <foo>
 80483fb:       b8 00 00 00 00          mov    eax,0x0
 8048400:       83 c4 04                add    esp,0x4
 8048403:       59                      pop    ecx
 8048404:       5d                      pop    ebp
 8048405:       8d 61 fc                lea    esp,[ecx-0x4]
 8048408:       c3                      ret
08048409 <foo>:
 8048409:       55                      push   ebp
 804840a:       89 e5                   mov    ebp,esp
 804840c:       90                      nop
 804840d:       5d                      pop    ebp
 804840e:       c3                      ret
 804840f:       90                      nop
```

在最终输出中，之前位于 `1b` 的操作码 `e8` 现在从地址 `80483f6` 开始。使用其对象文件中的相同计算方法，将模拟值 `fc ff ff ff` 替换为实际值 `0e 00 00 00`：操作码 `e8` 位于 `80483f6`。`foo` 的定义位于 `8048409`。从 `e8` 之后的地址偏移量是 $8048409 + 0 - 80483f7 - 4 = 0e$。然而，为了可读性，汇编以 `call 8048409 <foo>` 的形式显示，因为 `GNU as`[8] 汇编器允许指定实际内存地址

[8] 或任何当前使用的汇编器。

符号定义的。此类地址随后转换为相对寻址模式，节省程序员计算偏移量的麻烦

manually.

8.1.5 Sym. Name

此字段显示要重新定位的符号名称。该符号名称与C等高级语言中书写的一致。

8.2 使用链接脚本构建 ELF 可执行文件

一个 *linker* 是一个将分离的对象文件组合成最终程序的程序 ***linker***

二进制文件。当调用 `gcc` 时，它运行 `ld` 将目标文件转换为最终的可执行文件。

一个 *linker script* 是一个文本文件，它指导链接器应该如何... ***linker script***

编译对象文件。当 `gcc` 运行时，它使用其默认链接脚本来构建编译的二进制文件的内存布局。标准化的内存布局称为 *object file format*，例如，ELF 包括程序头、节头及其属性。默认链接脚本是为在当前操作系统环境中运行而制作的9。在裸机

9 查看默认脚本，使用 `--verbose` 选项：`ld --verbose`

金属，默认脚本不能直接使用，因为它不是为这种环境设计的。因此，程序员需要为这种环境提供自己的链接脚本。

每个链接脚本都由一系列具有以下格式的命令组成：

```
COMMAND
{
   sub-command 1
   sub-command 2
   .... more sub-command....
}
```

每个子命令仅针对顶级命令。最简单的链接脚本只需要一个命令：`SECTION`，它从对象文件中消耗输入段并生成最终二进制文件的输出段10。

[10] Recall that sections are chunks of code or data, or both.

8.2.1 Example linker script

Here is a minimal example of a linker script:

main.lds

```
SECTIONS /* Command */
{
   . = 0x10000; /* sub-command 1 */
   .text : { *(.text) } /* sub-command 2 */
   . = 0x8000000; /* sub-command 3 */
   .data : { *(.data) } /* sub-command 4 */
   .bss : { *(.bss) } /* sub-command 5 */
}
```

代码剖析:

Code	Description
`SECTION`	Top-level command that declares a list of custom program sections. `ld` provides a set of such commands.
`. = 0x10000;`	Set location counter to the address `0x10000`. Location counter specifies the base address for subsequent commands. In this example, subsequent commands will use `0x10000` onward.
`.text : { *(.text) }`	Since location counter is set to `0x10000`, the output `.text` in the final binary file will starts at the address `0x10000`. This command combines all `.text` sections from all object files with `*(.text)` syntax into a final `.text` section. The `*` is the wildcard which matches any file name.
`. = 0x8000000;`	Again, the location counter is set to `0x8000000`. Subsequent commands will use this address for working with sections.
`.data : { *(.data) }`	All `.data` section are combined into one `.data` section in the final binary file.
`.bss : { *(.bss) }`	All `.bss` section are combined into one `.bss` section in the final binary file.

地址 `0x10000` 和 `0x8000000` 被称为 *Virtual Memory Address*。

一个 *virtual memory address* 是内存中加载段的地址 - *virtual memory address*

当程序运行时。要使用链接脚本，我们将其保存为文件，例如 `main.lds`[11]；然后，我们需要一个示例程序文件，例如 `main.c`:

11 `.lds` 是链接脚本的扩展。

main.c

```
void test() {}
```

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


    return 0;
}
```

然后，我们编译文件并显式使用链接脚本调用 ld：

```
$ gcc -m32 -g -c main.c
$ ld -m elf_i386 -o main -T main.lds main.o
```

在 ld 命令中，选项类似于 gcc：

Option	Description
-m	Specify object file format that ld produces. In the example, elf_i386 means a 32-bit ELF is to be produced.
-o	Specify the name of the final executable binary.
-T	Specify the linker script to use. In the example, it is main.lds.

剩余的输入是链接的对象文件列表。在完成

当执行mand ld，将生成最终的可执行二进制文件 - main。如果我们尝试运行它：

```
$ ./main
Segmentation fault
```

原因是当手动链接时，必须显式设置入口地址，否则 ld 默认将其设置为 .text 节的开始。我们可以从 readelf 输出中进行验证：

```
$ readelf -h main
```

Output

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
```

```
OS/ABI:                            UNIX - System V
ABI Version:                       0
Type:                              EXEC (Executable file)
Machine:                           Advanced Micro Devices X86-64
Version:                           0x1
Entry point address:               0x10000
Start of program headers:          64 (bytes into file)
Start of section headers:          2098144 (bytes into file)
Flags:                             0x0
Size of this header:               64 (bytes)
Size of program headers:           56 (bytes)
Number of program headers:         3
Size of section headers:           64 (bytes)
Number of section headers:         14
Section header string table index: 11
```

入口地址设置为 0x10000，它是 .text 节的开始。使用 objdump 检查地址：

```
$ objdump -z -M intel -S -D prog | less
```

我们注意到地址 0x10000 并不在 main 函数开始处

程序运行：

Output

```
Disassembly of section .text:
00010000 <test>:
int a = 5;
int i;
void test(){}
   10000:       55                      push   ebp
   10001:       89 e5                   mov    ebp,esp
   10003:       90                      nop
   10004:       5d                      pop    ebp
   10005:       c3                      ret
00010006 <main>:
```

```
int main(int argc, char *argv[])
{
    10006:       55                      push   ebp
    10007:       89 e5                   mov    ebp,esp

     return 0;
    10009:       b8 00 00 00 00          mov    eax,0x0
}
    1000e:       5d                      pop    ebp
    1000f:       c3                      ret
```

起始.text部分在0x10000处的函数是test，不是main！为了使程序能在main处正常运行，我们需要在链接器脚本中将以下行设置在文件开头：

```
ENTRY(main)
```

重新编译可执行二进制文件 main。这次，readelf 的输出不同：

Output

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x10006
  Start of program headers:          52 (bytes into file)
  Start of section headers:          9168 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
```

```
    Size of program headers:           32 (bytes)
    Number of program headers:         3
    Size of section headers:           40 (bytes)
    Number of section headers:         14
    Section header string table index: 11
```

程序现在在启动时在地址 0x10006 执行代码。

0x10006 这是 main 开始的地方！为了确保我们真正从 main 开始，我们使用 gdb 运行程序，在 main 和 test 函数处设置两个断点：

```
$ gdb ./main
```

输出

```
.... output omitted ....
(gdb) b test
Reading symbols from ./main...done.
```

```
(gdb) b test
```

输出

```
Breakpoint 1 at 0x10003: file main.c, line 1.
```

```
(gdb) b main
```

输出

```
Breakpoint 2 at 0x10009: file main.c, line 5.
```

```
(gdb) r
```

输出

```
Starting program: /tmp/main
5     return 0;
Breakpoint 2, main (argc=-11493, argv=0x0) at main.c:5
```

如输出所示，gdb 首先在 2nd 断点处停止。

现在，我们正常运行程序，不使用 gdb：

```
$ ./main
Segmentation fault
```

我们仍然得到一个段错误。这是可以预料的，因为我们运行了一个没有操作系统C运行时支持的定制二进制文件。在 `main` 函数的最后一条语句：`return 0`，简单地返回到一个随机位置12。C运行时确保程序正确退出。在

12 返回地址位于当前 `ebp` 之上。然而，当我们进入 `main` 时，没有返回值被压入栈中。因此，当执行返回时，它只是检索 `ebp` 之上的任何值并将其用作返回地址。

Linux，当 `main` 返回时，会隐式调用 `_exit()` 函数。要解决这个问题，我们只需将程序改为正确退出：

hello.c

```
1 void test() {}
2 int main(int argc, char *argv[])
3 {
4     asm("mov eax, 0x1\n"
5         "mov ebx, 0x0\n"
6         "int 0x80");
7 }
```

内联汇编是必需的，因为Linux中为系统调用定义了中断 `0x80`。由于程序没有使用库，除了使用汇编之外，没有其他方式可以调用系统函数。然而，在编写我们的操作系统时，我们不需要这样的代码，因为没有正确退出的环境。

现在我们可以精确控制程序最初运行的地点，因此从引导加载程序启动内核变得容易。在我们进入下一节之前，注意如何将 `readelf` 和 `objdump` 应用到在程序运行之前进行调试。

8.2.2 Understand the custom ELF structure

在示例中，我们成功从一个自定义链接脚本创建了一个可运行的ELF可执行二进制文件，而不是使用gcc提供的默认脚本。为了方便查看其结构：

```
$ readelf -e main
```

-e 选项是3个选项的组合 -h -l -S:

Output

```
....... ELF header output omitted .......
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00010000 001000 000010 00  AX  0   0  1
  [ 2] .eh_frame         PROGBITS        00010010 001010 000058 00   A  0   0  4
  [ 3] .debug_info       PROGBITS        00000000 001068 000087 00      0   0  1
  [ 4] .debug_abbrev     PROGBITS        00000000 0010ef 000074 00      0   0  1
  [ 5] .debug_aranges    PROGBITS        00000000 001163 000020 00      0   0  1
  [ 6] .debug_line       PROGBITS        00000000 001183 000038 00      0   0  1
  [ 7] .debug_str        PROGBITS        00000000 0011bb 000078 01  MS  0   0  1
  [ 8] .comment          PROGBITS        00000000 001233 000034 01  MS  0   0  1
  [ 9] .shstrtab         STRTAB          00000000 00133a 000074 00      0   0  1
  [10] .symtab           SYMTAB          00000000 001268 0000c0 10     11  10  4
  [11] .strtab           STRTAB          00000000 001328 000012 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x00010000 0x00010000 0x00068 0x00068 R E 0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
 Section to Segment mapping:
  Segment Sections...
   00     .text .eh_frame
   01
```

The structure is incredibly simple. Both the segment and section listings can be contained within one screen. This is not the case with default ELF executable binary. From the output, there are only 11 sections,

仅有两个在运行时加载：.text 和 .eh_frame，因为这两个部分分别分配了实际的内存地址，0x10000 和 0x10010，其余部分分配了 0 *in the final executable binary*[13]，这意味着它们在运行时不会被加载。这使得

13 与对象文件相反，其中内存地址始终为0，仅在链接过程中分配实际值。

感觉，因为这些部分与版本14、调试15和链接-相关

ing16.

14 它是 .comment 部分。它可以与评论 `readelf -p` 查看。[15] 以 .debug 前缀开始的那些。

16 符号表和字符串表。

程序段头表甚至更简单。它只包含2个段：LOAD和GNU_STACK。默认情况下，如果链接脚本没有提供构建程序段的指令，ld提供合理的默认段。就像在这种情况下，.text应该在LOAD段中。GNU_STACK段是Linux内核用来控制程序栈状态的GNU扩展。在我们从头开始编写自己的操作系统时，我们不需要这个段，以及.eh_frame，它是用于异常处理的。为了实现这些目标，我们需要创建自己的程序头，而不是让ld处理这个任务，并指示ld移除.eh_frame。

.comment main

8.2.3 Manipulate the program segments

First, we need to craft our own program header table by using the following syntax:

```
PHDRS
{
  <name> <type> [ FILEHDR ] [ PHDRS ] [ AT ( address ) ]
        [ FLAGS ( flags ) ] ;
}
```

PHDRS 命令，类似于 SECTION 命令，但用于声明具有预定义语法的自定义程序段列表。

name 是用于在 SECTION 命令中声明的部分中稍后引用的标题名称。

type 是ELF段类型，如第5.5节所述，添加了前缀PT_。例如，而不是由readelf显示的NULL或LOAD，它是PT_NULL或PT_LOAD。

示例8.2.1。仅使用name和type，我们可以创建任意数量的程序段。例如，我们可以添加NULL程序段并删除GNU_STACK段：

main.lds

```
PHDRS
{
    null PT_NULL;
    code PT_LOAD;
}

SECTIONS
{
    . = 0x10000;
    .text : { *(.text) } :code
    . = 0x8000000;
    .data : { *(.data) }
    .bss : { *(.bss) }
}
```

PHDRS命令的内容表明最终的可执行二进制文件包含2个程序段：NULL和LOAD。NULL段被命名为null，LOAD段被命名为code以表示此LOAD段包含程序代码。然后，要将一个节放入一个段中，我们使用语法:<phdr>，其中phdr是之前给定的段名。在这个例子中，.text节被放入code段。我们编译并查看结果（假设main.o之前已编译）：

```
$ ld -m elf_i386 -o main -T main.lds main.o
$ readelf -l main
```

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x10000
```

```
There are 2 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  NULL           0x000000 0x00000000 0x00000000 0x00000 0x00000     0x4
  LOAD           0x001000 0x00010000 0x00010000 0x00010 0x00010 R E 0x1000
 Section to Segment mapping:
  Segment Sections...
   00
   01     .text .eh_frame
```

这两个段现在分别是 NULL 和 LOAD，而不是 LOAD 和 GNU_STACK。

示例8.2.2。我们可以添加任意数量的相同类型的段，只要
它们被赋予不同的名称：

main.lds

```
1  PHDRS
2  {
3      null1 PT_NULL;
4      null2 PT_NULL;
5      code1 PT_LOAD;
6      code2 PT_LOAD;
7  }
8
9  SECTIONS
10 {
11     . = 0x10000;
12     .text : { *(.text) } :code1
13     .eh_frame : { *(.eh_frame) } :code2
14     . = 0x8000000;
15     .data : { *(.data) }
16     .bss : { *(.bss) }
17 }
```

After amending the PHDRS content earlier with this new segment listing, we put .text into code1 segment and .eh_frame into code2 segment, we compile and see the new segments:

```
$ ld -m elf_i386 -o main -T main.lds main.o
$ readelf -l main
```

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x10000
There are 4 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  NULL           0x000000 0x00000000 0x00000000 0x00000 0x00000     0x4
  NULL           0x000000 0x00000000 0x00000000 0x00000 0x00000     0x4
  LOAD           0x001000 0x00010000 0x00010000 0x00010 0x00010 R E 0x1000
  LOAD           0x001010 0x00010010 0x00010010 0x00058 0x00058 R   0x1000
 Section to Segment mapping:
  Segment Sections...
   00
   01
   02     .text
   03     .eh_frame
```

现在 .text 和 .eh_frame 在不同的段中。

FILEHDR 是一个可选关键字，当添加时指定程序段包含可执行二进制文件的ELF文件头。然而，此属性仅应添加到第一个程序段，因为它会极大地改变段的大小和起始地址，因为ELF头始终位于二进制文件的开头，回想一下，一个段从其第一个内容地址开始，在大多数情况下（除了这种情况，即文件头），这是第一个部分。

Example 8.2.3. Adding the FILEHDR keyword changes the size of NULL segment:

main.lds

```
PHDRS
{
    null PT_NULL FILEHDR;
    code PT_LOAD;
}
..... content is the same .....
```

我们再次链接并查看结果：

```
$ ld -m elf_i386 -o main -T main.lds main.o
$ readelf -l main
```

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x10000
There are 2 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  NULL           0x000000 0x00000000 0x00000000 0x00034 0x00034 R   0x4
  LOAD           0x001000 0x00010000 0x00010000 0x00068 0x00068 R E 0x1000
 Section to Segment mapping:
  Segment Sections...
   00
   01     .text .eh_frame
```

在之前的示例中，NULL部分的文件大小和内存大小始终为0，现在它们都是34字节，这是ELF头的大小。

示例8.2.4。如果我们将FILEHDR分配给一个非起始段，其大小和起始地址将显著改变：

main.lds

```
PHDRS
{
    null PT_NULL;
```

```
    code PT_LOAD FILEHDR;
}
..... content is the same .....
```

```
$ ld -m elf_i386 -o main -T main.lds main.o
$ readelf -l main
```

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x10000
There are 2 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  NULL           0x000000 0x00000000 0x00000000 0x00000 0x00000     0x4
  LOAD           0x000000 0x0000f000 0x0000f000 0x01068 0x01068 R E 0x1000
 Section to Segment mapping:
  Segment Sections...
   00
   01     .text .eh_frame
```

前一个例子中 LOAD 段的大小仅为 0x68，与其中 .text 和 .eh_frame 部分的总大小相同。但现在，它变成了 0x01068，增加了 0x1000 字节。这些额外字节的原因是什么？一个简单的答案：段对齐。从输出中可以看出，此段的对齐方式为 0x1000；这意味着无论哪个地址是此段的起始地址，它都必须能被 0x1000 整除。因此，LOAD 的起始地址为 0xf000，因为它能被 0x1000 整除。

另一个问题出现了：为什么起始地址是 0xf000 而不是 0x10000？.text 是第一部分，它从 0x10000 开始，因此段应该从 0x10000 开始。原因是我们将 FILEHDR 作为段的一部分，它必须扩展以包含 ELF 文件头，该文件头位于 ELF 可执行二进制文件的起始位置。为了满足这个约束和对齐约束，0xf000 是最接近的地址。请注意，虚拟和物理内存地址是地址

在运行时，而不是磁盘上段的位置。正如 `FileSiz` 字段所示，该段仅在磁盘上占用 `0x1068` 字节。图 8.2.1 阐述了有无 `FILEHDR` 关键字时的内存布局差异。

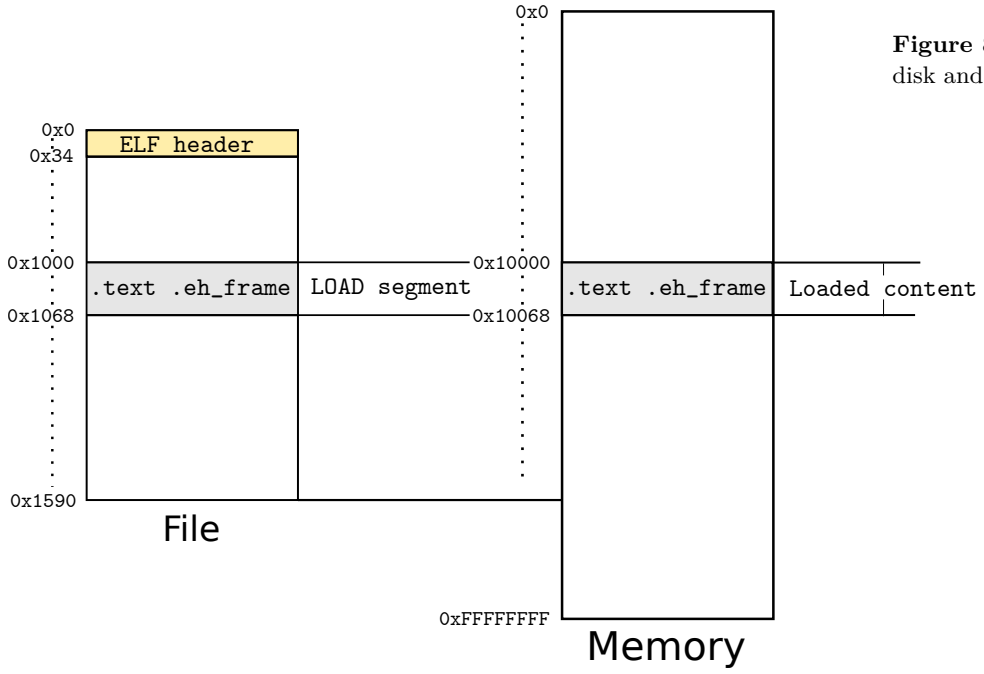
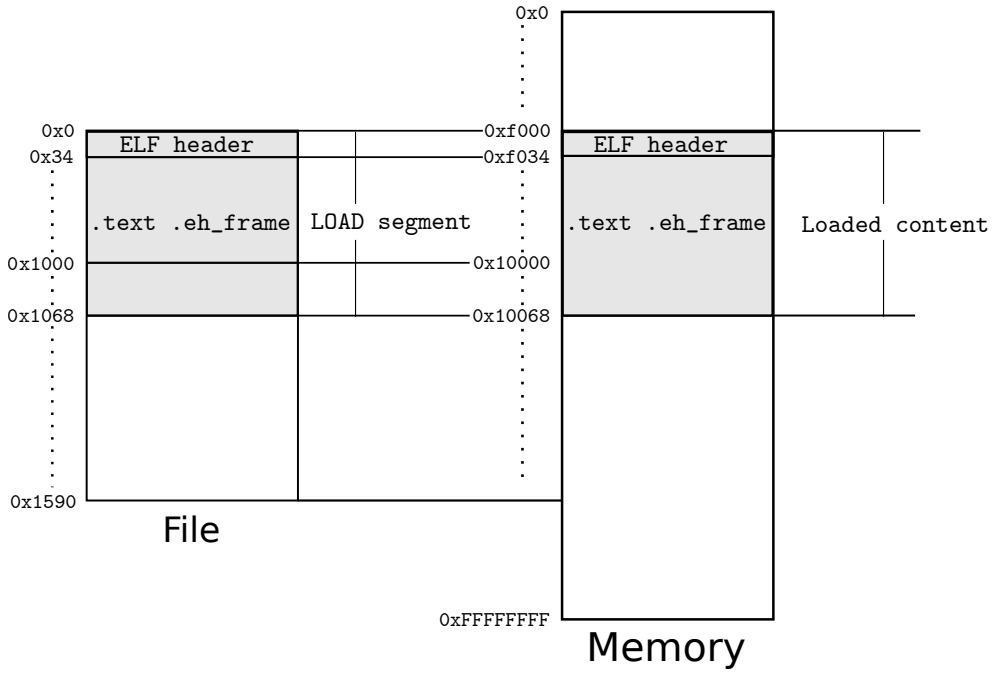


Figure 8.2.1: LOAD段开启disk and i内存。

(a) Without `FILEHDR`.



(b) With `FILEHDR`.

PHDRS 是一个可选关键字，当添加时指定一个程序段是一个程序段表头。

示例8.2.5. 由gcc生成的默认可执行二进制文件的第一段是一个PHDR，因为程序段头表紧接在ELF头之后。这也是一个方便将ELF头放入其中的段，使用FILEHDR关键字。我们用PHDR段替换了之前未使用的NULL段：

main.lds

```
PHDRS
{
    headers PT_PHDR FILEHDR PHDRS;
    code PT_LOAD FILEHDR;
}
..... content is the same .....
```

```
$ ld -m elf_i386 -o main -T main.lds main.o
$ readelf -l main
```

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x10000
There are 2 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000000 0x00000000 0x00000000 0x00074 0x00074 R   0x4
  LOAD           0x001000 0x00010000 0x00010000 0x00068 0x00068 R E 0x1000
 Section to Segment mapping:
  Segment Sections...
   00
   01     .text .eh_frame
```

如输出所示，第一个段落的类型为 PHDR。其大小是0x74，包括：

▷ 0x34 字节数用于ELF头。

▷ 0x40 程序段表头表字节数，包含2个条目，每个条目长度为0x20字节（32字节）。

上述数字与ELF头输出一致：

Output

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
....... output omitted ......
  Size of this header:               52 (bytes)   --> 0x34 bytes
  Size of program headers:           32 (bytes)   --> 0x20 bytes each program header
  Number of program headers:         2            --> 0x40 bytes in total
  Size of section headers:           40 (bytes)
  Number of section headers:         12
  Section header string table index: 9
```

AT (address) 指定将段放置的加载内存地址。每个段或节都有一个 *virtual memory address* 和一个 *load memory address*：

▷ 一个 *virtual memory address* 是一个段或一个的起始地址 *virtual memory address*
内存中运行时的程序部分。内存地址被称为虚拟的，因为它不映射到与地址号码相对应的实际内存单元，而是映射到任何随机内存单元，这取决于底层操作系统如何转换地址。例如，虚拟内存地址 0x1 可能映射到具有物理地址 0x1000 的内存单元。

▷ 一个 *load memory address* 是物理内存地址，其中 a *load memory address*
程序已加载但尚未运行。

加载内存地址由 AT 语法指定。通常这两种类型的地址相同，物理地址可以忽略。

nored. 当加载和运行时，故意将其分为两个不同的阶段，需要不同的地址区域。

例如，可以设计一个程序将其加载到ROM[17]中固定地址。但将其加载到RAM以供裸机应用程序或操作系统使用时，程序需要一个能够适应目标应用程序或操作系统的寻址方案的加载地址。

17 只读存储器

示例8.2.6。我们可以使用AT语法为段LOAD指定一个加载内存地址：

main.lds

```
PHDRS
{
    headers PT_PHDR FILEHDR PHDRS AT(0x500);
    code PT_LOAD;
}
..... content is the same .....
```

```
$ ld -m elf_i386 -o main -T main.lds main.o
$ readelf -l main
```

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x4000
There are 2 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000000 0x00000000 0x00000500 0x00074 0x00074 R   0x4
  LOAD           0x001000 0x00004000 0x00002000 0x00068 0x00068 R E 0x1000
 Section to Segment mapping:
  Segment Sections...
   00
   01     .text .eh_frame
```

它取决于操作系统是否使用地址。对于我们的操作系统，虚拟内存地址和加载是相同的，因此显式加载地址不是我们关心的问题。

FLAGS (flags) 为段分配权限。每个标志是一个表示权限的整数，可以通过或操作组合。可能的值：

Permission	Value	Description
R	1	Readable
W	2	Writable
E	4	Executable

示例8.2.7。我们可以创建一个具有读取、写入和执行权限的LOAD段：

main.lds

```
PHDRS
{
    headers PT_PHDR FILEHDR PHDRS AT(0x500);
    code PT_LOAD FILEHDR FLAGS(0x1 | 0x2 | 0x4);
}
..... content is the same .....
```

```
$ ld -m elf_i386 -o main -T main.lds main.o
$ readelf -l main
```

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x0
There are 2 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000000 0x00000000 0x00000500 0x00074 0x00074 R   0x4
```

```
 LOAD           0x001000 0x00000000 0x00000000 0x00010 0x00010 RWE 0x1000
Section to Segment mapping:
 Segment Sections...
  00
  01     .text .eh_frame
```

LOAD 段现在获得所有 RWE 权限，如上所示。

最后，我们想要删除 .eh_frame 或任何不想要的章节，
我们添加了一个名为 /DISCARD/ 的特殊部分：

main.lds

```
... program segment header table remains the same ...

SECTIONS
{
    /* . = 0x10000; */
    .text : { *(.text) } :code
    . = 0x8000000;
    .data : { *(.data) }
    .bss : { *(.bss) }
    /DISCARD/ : { *(.eh_frame) }
}
```

任何包含 /DISCARD/ 的部分在最终可执行文件中都会消失
二进制：

```
$ ld -m elf_i386 -o main -T main.lds main.o
$ readelf -l main
```

输出

```
Elf file type is EXEC (Executable file)
Entry point 0x0
There are 2 program headers, starting at offset 52
Program Headers:
```

```
 Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
 PHDR           0x000000 0x00000000 0x00000500 0x00074 0x00074 R   0x4
 LOAD           0x001000 0x00000000 0x00000000 0x00010 0x00010 R E 0x1000
Section to Segment mapping:
 Segment Sections...
  00
  01     .text
```

如所见，`.eh_frame`无处可寻。

8.3 C运行时：托管与独立

`.init`、`.init_array`、`.fini_array`和`.preinit_array`部分的作用是初始化一个支持C标准库的C运行时环境。当C被认为是一种编译型语言时，为什么它还需要运行时环境呢？原因在于许多标准函数依赖于底层操作系统，而操作系统本身就是一个庞大的运行时环境。例如，与I/O相关的函数，如使用`gets()`从键盘读取、使用`open()`从文件读取、使用`printf()`在屏幕上打印、使用`malloc()`、`free()`等管理系统内存等。

A C实现无法在没有运行操作系统的环境中提供此类例程，这是一个*hosted environment*。一个*hosted environment*是一个运行时环境，它：

▷ 提供C库的默认实现，包括系统依赖的数据和例程。

▷ 执行资源分配以准备程序运行的环境。

此过程类似于硬件初始化过程：

▷ 当首次开机时，台式计算机从主板上的只读存储器中加载其基本系统例程。▷然后，它开始初始化环境，例如为CPU和设备中的各种寄存器设置默认值，在执行任何操作之前。

代码。

与之一致，一个 *freestanding environment* 是一个不提供系统依赖数据和例程的环境。因此，几乎不存在C库，并且该环境可以运行用纯C语法编写的编译代码。为了一个独立环境成为宿主环境，它必须实现标准C系统例程。但对于一个 *conforming* 独立环境，它只需要根据GCC手册)提供的这些头文件：`<float.h>`、`<limits.h>`、`<stadarg.h>`和`<stddef.h>` (。

对于典型的桌面x86程序，C运行时环境由编译器初始化，因此程序可以正常运行。然而，对于直接在其上运行的嵌入式平台，情况并非如此。在桌面操作系统中使用的典型C运行时环境不能用于嵌入式平台，因为架构差异和资源限制。因此，软件编写者必须实现一个适合目标平台的自定义C运行时环境。对于嵌入式平台，

在编写我们的操作系统时，第一步是创建一个独立的环境，然后再创建一个托管环境。

8.4 可调试裸机引导加载程序

当前，引导加载程序被编译为平面二进制文件。尽管 `gdb` 可以显示汇编代码，但它并不总是与源代码相同。在汇编源代码中存在变量名称和标签。这些符号在编译为平面二进制文件时丢失，使得调试更加困难。另一个问题是编写的汇编源代码与显示的汇编源代码不匹配。编写的代码可能包含汇编器特定的更高级语法，这些语法被生成为 `gdb` 显示的更低级汇编代码。最后，在有调试信息的情况下，可以使用命令 `next/n` 和 `prev/p` 来代替 `ni` 和 `si`。

要启用调试信息，我们修改引导加载程序Makefile：

1. 引导加载程序必须编译为 ELF 二进制文件。打开 Makefile

在引导加载程序目录下，并更改以下 $(BUILD_DIR)/%.o: %.asm 菜单中的这一行：

```
nasm -f bin $< -o $@
```

到这一行：

```
nasm -f elf $< -F dwarf -g -o $@
```

在更新的配方中，bin格式被elf格式替换，以便正确生成调试信息.-F选项指定调试信息格式，在本例中为dwarf。最后，-g选项导致nasm实际上以所选格式生成调试信息。

2. 然后，ld消耗ELF引导加载程序二进制文件并产生另一个ELF引导加载程序二进制文件，具有与运行时引导加载程序实际地址匹配的.text节区的正确起始内存地址，当QEMU虚拟机在0x7c00处加载它时。我们需要ld，因为当由nasm编译时，假设起始地址为0，而不是0x7c00。

3. 最后，我们使用 objcopy 来分离仅提取平面二进制内容作为原始引导加载程序，通过向 $(BUILD_DIR)/%.o: %.asm 添加此行：

```
objcopy -O binary $(BUILD_DIR)/bootloader.o.elf $@
```

objcopy，正如其名所示，是一个复制和翻译对象文件的程序。在这里，我们复制原始的ELF引导加载程序并将其翻译成平面二进制文件。

更新后的食谱应如下所示：

```
$(BUILD_DIR)/%.o: %.asm
    nasm -f elf $< -F dwarf -g -o $@
    ld -m elf_i386 -T bootloader.lds $@ -o $@.elf
    objcopy -O binary $(BUILD_DIR)/bootloader.o.elf $@
```

现在我们测试具有调试信息的引导加载程序：

1. 启动QEMU虚拟机：

```
$ make qemu
```

2. 从存储在 `bootloader.o.elf` 中的调试信息开始 `gdb`：

```
$ gdb build/bootloader/bootloader.o.elf
```

在进入 `gdb` 后，按 `Enter` 键，如果使用样本 `.gdbinit` 部分 7.7.3，输出应如下所示：

输出

```
---Type <return> to continue, or q <return> to quit---
[f000:fff0] 0x0000fff0 in ?? ()
Breakpoint 1 at 0x7c00: file bootloader.asm, line 6.
(gdb)
```

`gdb` 现在理解了地址 `0x7c00` 的指令在汇编源文件中的位置，多亏了调试信息。

8.5 硬件裸机上的可调试程序

构建调试就绪的可执行二进制文件的过程类似于引导加载程序，但更为复杂。回想一下，为了调试器能够正常工作，其调试信息必须包含内存地址和源代码之间的正确地址映射。gcc在DIE条目中存储此类映射信息，其中它告诉gdb哪个代码地址对应于源文件中的一行，以便断点能够正常工作。

但是首先，我们需要一个示例C源文件，一个非常简单的文件：

os.c

```
void main() {}
```

因为这是一个独立的环境，涉及系统函数如 `printf()` 的标准库将无法工作，因为没有C运行时。在此阶段，目标是正确跳转到main，并在 `gdb` 中正确显示源代码，因此目前不需要花哨的C代码。

下一步是更新 `os/Makefile`：

```
BUILD_DIR=../build
OS=$(BUILD_DIR)/os

CFLAGS+=-ffreestanding -nostdlib -gdwarf-4 -m32 -ggdb3

OS_SRCS := $(wildcard *.c)
OS_OBJS := $(patsubst %.c, $(BUILD_DIR)/%.o, $(OS_SRCS))

all: $(OS)

$(BUILD_DIR)/%.o: %.c
    gcc $(CFLAGS) -c $< -o $@

$(OS): $(OS_OBJS)
    ld -m elf_i386 -Tos.lds $(OS_OBJS) -o $@

clean:
    rm $(OS_OBJS)
```

我们更新了Makefile，以下为更改内容：

- 添加一个 `CFLAGS` 变量，用于将选项传递给 `gcc`。
- 替代早期构建汇编源代码的规则，它被一个带有构建C源文件菜谱的C版本所取代。无论添加多少选项，`CFLAGS`变量都使菜谱中的`gcc`命令看起来更简洁。
- 添加一个链接命令，用于使用自定义链接脚本 `os.lds` 构建操作系统的最终可执行二进制文件。

一切看起来都很好，除了链接脚本部分。为什么需要它？链接脚本需要用于控制操作系统二进制文件在内存中的物理内存地址，以便链接器可以跳转到操作系统代码并执行它。为了满足这一要求，gcc使用的默认链接脚本将不起作用，因为它假设编译后的可执行文件在现有的操作系统内部运行，而我们是正在编写操作系统本身。

下一个问题是，链接脚本中的内容将是什么？为了回答这个问题，我们必须了解使用链接脚本要实现的目标：

▷ 为了引导加载程序正确跳转并执行操作系统代码。

▷ 对于 gdb 正确调试与操作系统源代码。

为了实现目标，我们必须为操作系统设计一个合适的内存布局。回想一下，在第7章中开发的引导加载程序已经可以加载从示例汇编程序 sample.asm 编译的简单二进制文件。为了加载操作系统，我们可以简单地加载从 sample.asm 编译的二进制文件和上面从 os.c 编译的二进制文件。

如果只有这么简单就好了。这个想法是正确的，但还不够。目标意味着以下约束：

1. 操作系统代码是用C语言编写的，编译成ELF可执行二进制文件。这意味着引导加载程序需要从ELF头中检索正确的入口地址。

2. 要正确使用 gdb 进行调试，调试信息必须包含指令地址和源代码之间的正确映射。

感谢在前面章节中对ELF和DWARF的理解，我们当然可以修改引导加载程序并创建一个满足上述约束的可执行二进制文件。我们将逐一解决这些问题。

8.5.1 Loading an ELF binary from a bootloader

之前我们检查了一个ELF头包含一个程序的入口地址。根据 `man elf`，这个信息距离ELF头的开始有0x18字节：

```
typedef struct {
              unsigned char e_ident[EI_NIDENT];
              uint16_t e_type;
              uint16_t e_machine;
              uint32_t e_version;
              ElfN_Addr e_entry;
              ElfN_Off e_phoff;
              ElfN_Off e_shoff;
              uint32_t e_flags;
              uint16_t e_ehsize;
              uint16_t e_phentsize;
              uint16_t e_phnum;
              uint16_t e_shentsize;
              uint16_t e_shnum;
              uint16_t e_shstrndx;
          } ElfN_Ehdr;
```

从结构体开始到 `e_entry` 开始的偏移量是：

▷ 16 字节 of `e_ident[EI_NIDENT]`：

```
#define EI_NIDENT 16
```

▷ 2 字节 e_type ▷ 2 字节 e_machine ▷ 4 字节 e_version

$$\texttt{Offset} = 16 + 2 + 2 + 4 = 24 = \texttt{0x18}$$

e_entry 是类型 ElfN_Addr，其中 N 要么是 32，要么是 64。我们正在编写32位操作系统，在这种情况下 $N = 32$，因此 ElfN_Addr 是 Elf32_Addr，它长度为4字节。

示例8.5.1。对于任何程序，例如这个简单的程序：

hello.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("hello␣world!\n");
    return 0;
}
```

我们可以使用可读的格式检索条目地址
使用 readelf 的翻译：

```
$ gcc hello.c -o hello
$ readelf -h hello
```

Output

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  .... output omitted ....
  Entry point address:               0x400430
  .... output omitted ....
```

或者以原始二进制形式，hd：

```
$ hd hello | less
```

Output

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  30 04 40 00 00 00 00 00  |..>.....0.@.....|
.........
```

偏移 0x18 是 e_entry 最不显著字节的起始位置，

这是 0x30，接着是 04 40 00，反序在一起构成地址 0x00400430。

现在我们知道ELF头中入口地址的位置，修改7.6.2节中制作的引导加载程序以检索并跳转到该地址就很容易了：

bootloader.asm

```
;*****************************************
; Bootloader.asm
; A Simple Bootloader
;*****************************************
bits 16
start: jmp boot

;; constant and variable definitions
msg db "Welcome to My Operating System!", 0ah, 0dh, 0h

boot:
  cli ; no interrupts
  cld ; all that we need to init

  mov  ax, 50h

  ;; set the buffer
  mov es, ax
  xor bx, bx

  mov al, 2     ; read 2 sector
  mov ch, 0 ; we are reading the second sector past us,
                                    ; so its still on track
                                        0
  mov cl, 2     ; sector to read (The second sector)
  mov dh, 0     ; head number
  mov dl, 0     ; drive number. Remember Drive 0 is floppy
      drive.
```

```
  mov ah, 0x02   ; read floppy sector function
  int 0x13     ; call BIOS - Read the sector
  jmp [500h + 18h]    ; jump and execute the sector!

  hlt ; halt the system

  ; We have to be 512 bytes. Clear the rest of the bytes
      with 0
  times 510 - ($-$$) db 0
  dw 0xAA55    ; Boot Signiture
```

就像这样简单！首先，我们在 0x500 加载操作系统二进制文件，然后从 0x500 中检索偏移量 0x18 的入口地址，首先计算表达式 $500h + 18h = 518h$ 以获取实际的内存地址，然后通过解引用检索内容。

第一部分已完成。对于下一部分，我们需要为引导加载程序构建一个ELF操作系统镜像。第一步是创建一个链接脚本：

main.lds

```
ENTRY(main);

PHDRS
{
  headers PT_PHDR FILEHDR PHDRS;
  code PT_LOAD;
}

SECTIONS
{
  .text 0x500: { *(.text) } :code
  .data : { *(.data) }
  .bss : { *(.bss) }
  /DISCARD/ : { *(.eh_frame) }
```

```
}
```

脚本简单明了，几乎与之前相同。唯一的不同之处是：

▷ `main` 被明确指定为入口点，通过指定 `ENTRY(main)`。

▷ `.text` 以 `0x500` 作为其 *virtual memory address* 明确指定，因为我们是在 `0x500` 加载操作系统镜像。

在放置脚本后，我们使用 `make` 编译，应该可以顺利运行：

```
$ make clean; make
$ readelf -l build/os/os
```

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x500
There are 2 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000000 0x00000000 0x00000000 0x00074 0x00074 R   0x4
  LOAD           0x000500 0x00000500 0x00000500 0x00040 0x00040 R E 0x1000
 Section to Segment mapping:
  Segment Sections...
   00
   01     .text
```

所有看起来都很好，直到我们运行它。我们首先启动QEMU虚拟-通用机器：

```
$ make qemu
```

然后，启动 `gdb` 并加载调试信息（也在同一个二进制文件中）并在 `main` 处设置断点：

```
(gdb) symbol-file build/os/os
Reading symbols from build/os/os...done.
(gdb) b main
Breakpoint 2 at 0x500
```

然后我们开始程序:

```
(gdb) symbol-file build/os/os
Reading symbols from build/os/os...done.
(gdb) b main
Breakpoint 2 at 0x500
```

保持编程运行，直到它在 `main` 处停止:

```
(gdb) c
Continuing.
[ 0:7c00]
Breakpoint 1, 0x00007c00 in ?? ()
(gdb) c
Continuing.
[ 0: 500]
Breakpoint 2, main () at main.c:1
```

在这个点上，我们切换布局到C源代码而不是寄存器:

```
(gdb) layout split
```

`layout split` 创建一个由3个小窗口组成的布局:

- ▷ 源窗口在顶部。
- ▷ 中间的汇编窗口。
- ▷ 命令窗口在底部。

在命令之后，布局应该看起来像这样:

Output

```
┌──main.c─────────────────────────────────────────────────────────────────────┐
B+>│1       void main(){}                                                      │
   │2                                                                          │
   │3                                                                          │
   │4                                                                          │
   │5                                                                          │
   │6                                                                          │
   │7                                                                          │
   │8                                                                          │
   │9                                                                          │
   │10                                                                         │
   │11                                                                         │
   │12                                                                         │
   │13                                                                         │
   │14                                                                         │
   │15                                                                         │
   │16                                                                         │
   └─────────────────────────────────────────────────────────────────────────────┘

B+>┌──────────────────────────────────────────────────────────────────────────────┐
   │0x500 <main>    jg     0x547                                                 │
   │0x502 <main+2>  dec    sp                                                    │
   │0x503 <main+3>  inc    si                                                    │
   │0x504 <main+4>  add    WORD PTR [bx+di],ax                                   │
   │0x506           add    WORD PTR [bx+si],ax                                   │
   │0x508           add    BYTE PTR [bx+si],al                                   │
   │0x50a           add    BYTE PTR [bx+si],al                                   │
   │0x50c           add    BYTE PTR [bx+si],al                                   │
   │0x50e           add    BYTE PTR [bx+si],al                                   │
   │0x510           add    al,BYTE PTR [bx+si]                                   │
   │0x512           add    ax,WORD PTR [bx+si]                                   │
   │0x514           add    WORD PTR [bx+si],ax                                   │
   │0x516           add    BYTE PTR [bx+si],al                                   │
   │0x518           add    BYTE PTR [di],al                                      │
   │0x51a           add    BYTE PTR [bx+si],al                                   │
```

```
0x51c           xor    al,0x0
0x51e           add    BYTE PTR [bx+si],al
remote Thread 1 In: main                                L1    PC: 0x500
[f000:fff0] 0x0000fff0 in ?? ()
Breakpoint 1 at 0x7c00
(gdb) symbol-file build/os/os
Reading symbols from build/os/os...done.
(gdb) b main
Breakpoint 2 at 0x500: file main.c, line 1.
(gdb) c
Continuing.
[   0:7c00]
Breakpoint 1, 0x00007c00 in ?? ()
(gdb) c
Continuing.
[   0: 500]
Breakpoint 2, main () at main.c:1
(gdb) layout split
(gdb)
```

这里出了些问题。这并不是像第4.9.5节所知的功能调用生成的汇编代码。这绝对是错误的，用 objdump 验证过：

```
$ objdump -D build/os/os | less
```

Output

```
/home/tuhdo/workspace/os/build/os/os:     file format elf32-i386
Disassembly of section .text:
00000500 <main>:
 500:   55                      push   %ebp
 501:   89 e5                   mov    %esp,%ebp
 503:   90                      nop
```

```
    504:   5d                      pop    %ebp
    505:   c3                      ret
   .... remaining output omitted ....
```

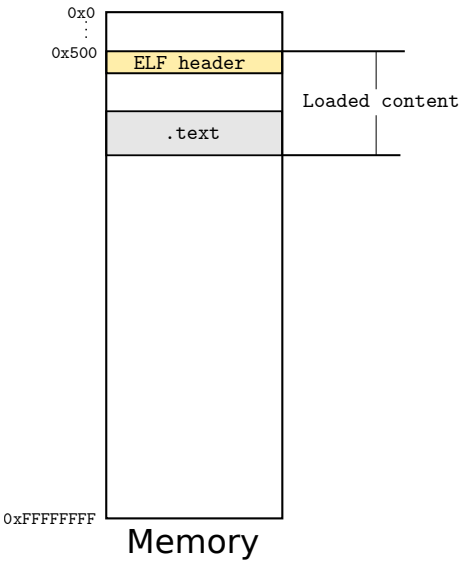
汇编代码的 main 完全不同。这就是为什么理解汇编代码及其与高级语言的关系很重要。没有这种知识，我们就会将 gdb 作为一个简单的源级调试器使用，而不必麻烦地查看从拆分布局的汇编代码。结果，无法发现代码无法正常工作的真正原因。

8.5.2 Debugging the memory layout

为什么在 gdb 显示的 main 中出现了错误的汇编代码？只有一个原因：引导加载程序跳转到了错误的地址。但是为什么地址是错误的呢？我们在地址 0x500 处创建了 .text 部分，其中 main 代码的第一个字节用于执行，并指示引导加载程序在偏移 0x18 处检索地址，然后跳转到入口地址。

然后，引导加载程序可能在错误的地址加载操作系统地址。但是，我们明确地将加载地址设置为50h:00，即0x500，因此使用了正确的地址。引导加载程序加载2nd扇区后，内存状态应如图8.5.1所示：

图8.5.1：加载2nd扇区后的内存状态。



这里的问题是：0x500是ELF头的开始。引导加载器实际上加载了2nd扇区，其中存储了整个可执行文件，到0x500。显然，.text部分，其中main存在，离0x500很远。由于可执行二进制文件的内存入口地址是0x500，.text应该在0x500 + 0x500 = 0xa00。然而，记录在ELF头中的入口地址仍然是0x500，因此引导加载器跳转到了那里而不是0xa00。这是必须解决的问题之一。

其他问题是调试信息与内存地址之间的映射。因为调试信息是使用假设的偏移量 0x500 编译的，它是 .text 节的开始，但由于实际加载，偏移量又推了另一个 0x500 字节，使得地址实际上位于 0xa00。

此内存不匹配使调试信息变得无用。

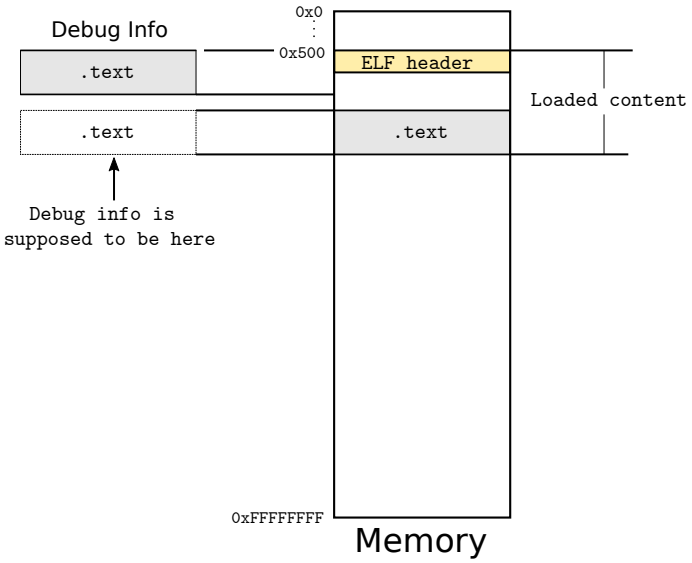


图8.5.2：调试信息中的符号-内存映射错误。

总结来说，我们有两个问题需要克服：

▷ 修复条目地址以考虑加载到内存时的额外偏移量。

▷ 修复调试信息，以考虑加载到内存时的额外偏移量。

首先，我们需要知道编译后的可执行二进制文件的实际布局：1

```
$ readelf -l build/os/os
```

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x500
There are 2 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000000 0x00000000 0x00000000 0x00074 0x00074 R   0x4
  LOAD           0x000500 0x00000500 0x00000500 0x00040 0x00040 R E 0x1000
 Section to Segment mapping:
  Segment Sections...
```

```
    00
    01     .text
```

注意 Offset 和 VirtAddress 字段：它们的值相同。这是问题所在，因为调试信息中的条目地址和内存地址依赖于 VirtAddr 字段，但 Offset 的值相同会破坏 VirtAddr[18] 的有效性，因为它意味着

实际内存地址将始终大于 VirtAddr。

18 偏移量是文件开始，地址0，到段或节的起始地址之间的字节数。

如果我们尝试调整链接脚本 os.lds 中 .text 部分的虚拟内存地址，无论我们设置什么值，都会将 Offset 设置为相同的值，直到我们将其设置为等于或大于 0x1074 的某个值：

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x1074
There are 2 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000000 0x00000000 0x00000000 0x00074 0x00074 R   0x4
  LOAD           0x000074 0x00001074 0x00001074 0x00006 0x00006 R E 0x1000
 Section to Segment mapping:
  Segment Sections...
   00
   01     .text
```

如果我们调整虚拟地址为 0x1073，则 Offset 和 VirtAddr 都会调整仍然共享相同的值：

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x1073
There are 2 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000000 0x00000000 0x00000000 0x00074 0x00074 R   0x4
  LOAD           0x001073 0x00001073 0x00001073 0x00006 0x00006 R E 0x1000
 Section to Segment mapping:
```

```
Segment Sections...
 00
 01     .text
```

关键回答此类现象的答案是 `Align` 场。值 `0x1000` 表示段偏移地址应该能被 `0x1000` 整除，或者如果段之间的距离能被 `0x1000` 整除，链接器将移除这样的距离以节省二进制大小。我们可以进行一些实验来验证这个说法[19]：

19 所有输出均由以下命令生成：
`$ readelf -l build/os/os`

▷ 通过将 `.text` 的虚拟地址设置为 `0x0` 到 `0x73`（在 `os.lds`）中，偏移量从 `0x1000` 到 `0x1073` 开始，相应地。例如，将其设置为 `0x0`：

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x0
There are 2 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000000 0x00000000 0x00000000 0x00074 0x00074 R   0x4
  LOAD           0x001000 0x00000000 0x00000000 0x00006 0x00006 R E 0x1000
 Section to Segment mapping:
  Segment Sections...
   00
   01     .text
```

默认情况下，如果我们没有指定任何虚拟地址，偏移量保持在 `0x1000`，因为 `0x1000` 是满足对齐约束的完美偏移量。从 `0x1` 到 `0x73` 的任何增加都会使段错位，但链接器仍然保留它，因为它被告知这样做。

▷ 通过将 `.text` 的虚拟地址设置为 `0x74`（在 `os.lds`）：

输出

```
Elf file type is EXEC (Executable file)
Entry point 0x74
There are 2 program headers, starting at offset 52
```

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000000 0x00000000 0x00000000 0x00074 0x00074 R   0x4
  LOAD           0x000074 0x00000074 0x00000074 0x00006 0x00006 R E 0x1000
 Section to Segment mapping:
  Segment Sections...
   00
   01     .text
```

PHDR 0x74 字节大小，因此如果 LOAD 从 0x1074 开始，PHDR 段和 LOAD 段之间的距离是 0x1074 − 0x74 = 0x1000 字节。为了节省空间，它移除了额外的 0x1000 字节。

▷ 通过将 .text 的虚拟地址设置为 os.lds) 中 0x75 和 0x1073 (之间的任何值，偏移量将取指定的确切值，如上例中将 0x1073 设置为所示。

▷ 通过将 .text 的虚拟地址设置为任何等于或大于 0x1074 的值：它将从 0x74 重新开始，其中距离等于 0x1000 字节。

现在我们得到了如何控制 Offset 和 VirtAddr 的值以产生所需二进制布局的线索。我们需要做的是将 Align 字段更改为具有较小值的值以实现更精细的控制。可能使用如下二进制布局即可实现：

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x600
There are 2 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000000 0x00000000 0x00000000 0x00074 0x00074 R   0x4
  LOAD           0x000100 0x00000600 0x00000600 0x00006 0x00006 R E 0x100
 Section to Segment mapping:
  Segment Sections...
   00
   01     .text
```

二进制在内存中将类似于图8.5.3:

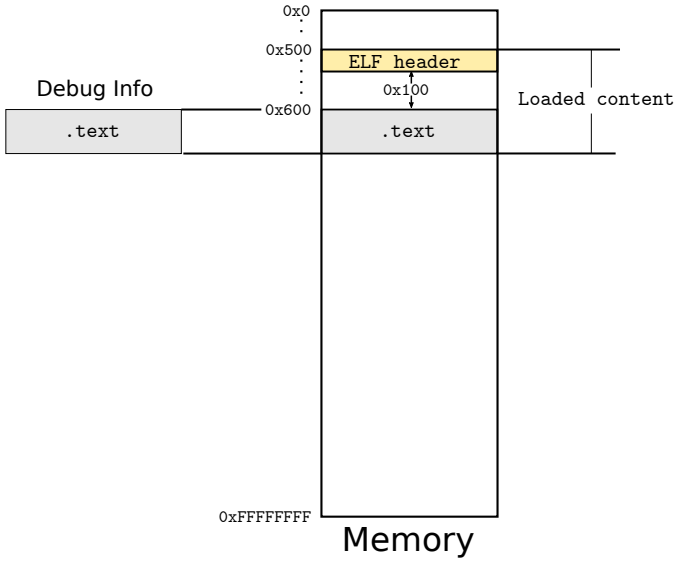


图8.5.3: 一个很好的 二进制布局。

如果我们从文件开头将 `Offset` 字段设置为 `0x100`，并将 `VirtAddr` 设置为 `0x600`，则在内存加载时，.text 的实际内存为 `0x500` + `0x100` = `0x600`；`0x500` 是引导加载程序加载到物理内存的内存位置，`0x100` 是从 ELF 头部末尾到 `.text` 的偏移量。然后，入口地址和调试信息将从上面的 `VirtAddr` 字段取值 `0x600`，这与实际的物理布局完全匹配。我们可以通过以下方式更改 `os.lds`:

main.lds

```
ENTRY(main);

PHDRS
{
  headers PT_PHDR FILEHDR PHDRS;
  code PT_LOAD;
}

SECTIONS
{
  .text 0x600: ALIGN(0x100) { *(.text) } :code
```

```
  .data : { *(.data) }
  .bss : { *(.bss) }
  /DISCARD/ : { *(.eh_frame) }
}
```

ALIGN 关键字，正如其含义所示，告诉链接器对一个部分进行对齐，因此包含它的段。然而，要使 ALIGN 关键字产生任何效果，必须禁用自动对齐。根据 man ld：

Output

```
     -n
     --nmagic
          Turn off page alignment of sections, and disable linking against shared
          libraries.  If the output format supports Unix style magic numbers, mark the
          output as "NMAGIC"
```

这是，默认情况下，每个部分都按操作系统页面对齐，大小为 4096，或 0x1000 字节。-n 或 -nmagic 选项禁用此行为，这是需要的。我们修改了在 os/Makefile 中使用的 ld 命令：

os/Makefile

```
..... above content omitted ....
$(OS): $(OS_OBJS)
    ld -m elf_i386 -nmagic -Tos.lds $(OS_OBJS) -o $@
```

最后，我们还需要更新顶层Makefile，以便将多个扇区写入磁盘镜像中的操作系统二进制文件，因为其大小超过了一个扇区：

```
 $ ls -l build/os/os
-rwxrwxr-x 1 tuhdo tuhdo 9060 Feb 13 21:37 build/os/os
```

我们更新了规则，以便自动计算行业：

os/Makefile

```
..... above content omitted ....
bootdisk: bootloader os
```

```
dd if=/dev/zero of=$(DISK_IMG) bs=512 count=2880
dd conv=notrunc if=$(BOOTLOADER) of=$(DISK_IMG) bs=512
    count=1 seek=0
dd conv=notrunc if=$(OS) of=$(DISK_IMG) bs=512 count=$$
    (($(shell stat --printf="%s" $(OS))/512)) seek=1
```

更新完所有内容后，重新编译可执行二进制文件，我们分别得到所需的偏移量和虚拟内存 0x100 和 0x600：

Output

```
Elf file type is EXEC (Executable file)
Entry point 0x600
There are 2 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000000 0x00000000 0x00000000 0x00074 0x00074 R   0x4
  LOAD           0x000100 0x00000600 0x00000600 0x00006 0x00006 R E 0x100
 Section to Segment mapping:
  Segment Sections...
   00
   01     .text
```

8.5.3 Testing the new binary

首先，我们启动QEMU机器：

```
$ make qemu
```

在另一 er 终端，我们开始 gdb，加载调试信息并设置 断点在 main：

```
$ gdb
```

以下输出应生成：

输出

```
---Type <return> to continue, or q <return> to quit---
[f000:fff0] 0x0000fff0 in ?? ()
Breakpoint 1 at 0x7c00
Breakpoint 2 at 0x600: file main.c, line 1.
```

然后，让 gdb 运行直到它遇到 main 函数，然后我们切换到源和汇编之间的分割布局：

```
(gdb) layout split
```

最终终端输出应如下所示：

Output

```
┌──main.c──────────────────────────────────────────────────────────────────┐
B+>│1       void main(){}                                                  │
   │2                                                                      │
   │3                                                                      │
   │4                                                                      │
   │5                                                                      │
   │6                                                                      │
   │7                                                                      │
   │8                                                                      │
   │9                                                                      │
   │10                                                                     │
   │11                                                                     │
   │12                                                                     │
   │13                                                                     │
   │14                                                                     │
   │15                                                                     │
   │16                                                                     │
   └───────────────────────────────────────────────────────────────────────┘
B+>│0x600 <main>    push   bp                                              │
   │0x601 <main+1>  mov    bp,sp                                           │
   │0x603 <main+3>  nop                                                    │
```

```
0x604 <main+4>  pop    bp
0x605 <main+5>  ret
0x606           aaa
0x607           add    BYTE PTR [bx+si],al
0x609           add    BYTE PTR [si],al
0x60b           add    BYTE PTR [bx+si],al
0x60d           add    BYTE PTR [bx+si],al
0x60f           add    BYTE PTR [si],al
0x611           add    ax,bp
0x613           push   ss
0x614           add    BYTE PTR [bx+si],al
0x616           or     al,0x67
0x618           adc    al,BYTE PTR [bx+si]
0x61a           add    BYTE PTR [bx+si+0x2],al
remote Thread 1 In: main                              L1    PC: 0x600
(gdb) c
Continuing.
[   0:7c00]
Breakpoint 1, 0x00007c00 in ?? ()
(gdb) c
Continuing.
[   0: 600]
Breakpoint 2, main () at main.c:1
(gdb) layout split
```

现在，显示的汇编与 `objdump` 中的相同，只是寄存器是16位的。这是正常的，因为 `gdb` 在16位模式下运行，而 `objdump` 以32位模式显示代码。为确保，我们通过使用 `x` 命令来验证原始操作码：

```
(gdb) x/16xb 0x600
```

Output

```
0x600 <main>:   0x55    0x89    0xe5    0x90    0x5d    0xc3    0x37
        0x00
0x608:  0x00    0x00    0x04    0x00    0x00    0x00    0x00    0x00
```

从汇编窗口，`main` 停止在地址 `0x605`。因此，从 `0x600` 到 `0x605` 的对应字节在命令 `x/16xb 0x600` 的输出中以红色突出显示。然后，从 `objdump` 输出的原始操作码：

```
$ objdump -z -M intel -S -D build/os/os | less
```

Output

```
build/os/os:     file format elf32-i386
Disassembly of section .text:
00000600 <main>:
void main(){}
 600:   55                      push   ebp
 601:   89 e5                   mov    ebp,esp
 603:   90                      nop
 604:   5d                      pop    ebp
 605:   c3                      ret
Disassembly of section .debug_info:
...... output omitted ......
```

两个程序显示的原始操作码相同。在这种情况下，这证明了gdb正确跳转到了main地址，以进行适当的调试。这是一个极其重要的里程碑。能够在裸机上进行调试将极大地帮助编写操作系统，因为调试器允许程序员在每一步检查运行机器的内部状态，逐步验证他的代码，逐步建立起一个坚实的基础理解。一些专业程序员不喜欢调试器，但这是因为他们已经足够深入地理解了他们的领域，不需要依赖调试器来验证他们的代码。在遇到新领域时，调试器是一个不可或缺的学习工具，因为它具有可验证性。

然而，即使有调试器的帮助，编写操作系统仍然不是一件轻松的事情。调试器可能在某个时间点提供对机器的访问，但它不会提供原因。要找出根本原因，则取决于程序员的技能。本书后面，我们将学习如何使用其他调试技术，例如使用QEMU日志功能来调试CPU异常。

第三部分

内核编程

9

x86 Descriptors

9.1 基本操作系统概念

首先和最重要的是，操作系统管理硬件资源。很容易从冯·诺依曼图中看到操作系统的核心功能：*CPU management:* 允许程序共享CPU以实现多任务处理。*Memory management:* 为程序运行分配足够的存储空间。*Devices management:* 检测并与不同设备通信。任何操作系统都应该擅长上述基本任务。

操作系统的重要特性之一是提供软件接口层，该层隐藏硬件接口，以便与在该操作系统上运行的应用程序进行交互。此类层的优势：

▷ 可重用性：也就是说，相同的软件API可以在不同的程序中重用，从而简化软件开发过程 ▷ 关注点分离：错误要么出现在应用程序中，要么出现在操作系统（OS）中；程序员需要隔离错误所在的位置。▷ 简化软件开发过程：提供了一个易于使用的软件接口层，并提供了对硬件资源的统一访问。

设备，而不是直接使用特定设备的硬件接口。

9.1.1 Hardware Abstraction Layer

存在许多硬件设备，因此最好让硬件工程师决定设备如何与操作系统通信。为了实现这一目标，操作系统仅提供一套在自身与设备驱动程序编写者之间达成一致的软件接口，被称为 *Hardware Abstraction Layer*。

在C语言中，此软件接口通过结构函数指针实现。

使用Linux示例说明

9.1.2 System programming interface

System programming interfaces 标准接口是操作系统提供给应用程序使用其服务的方式。例如，如果一个程序希望读取磁盘上的文件，那么它必须调用类似于 *open()* 的函数，并让操作系统处理与硬盘交互以检索文件的细节。

9.1.3 The need for an Operating System

从某种程度上说，操作系统是一种开销，但却是必要的，因为用户需要告诉计算机该做什么。当计算机系统中的资源（CPU、GPU、内存、硬盘等）变得庞大且复杂时，手动管理所有资源就变得繁琐。

想象我们不得不手动在3GB RAM的计算机上加载程序。我们必须在各种固定地址上加载程序，并且必须手动计算每个程序的大小以避免浪费内存资源，并且足够大以防止程序相互覆盖。

或者，当我们想要通过键盘给计算机输入时，如果没有操作系统，应用程序也必须携带代码以方便与键盘硬件通信；然后每个应用程序都独立处理这种键盘通信。为什么对于这样的标准功能，应用程序之间会有这样的重复？如果您编写一个

计数软件，为什么程序员应该关心编写键盘驱动程序，这与问题域完全无关?

这是为什么操作系统的一个关键任务就是隐藏硬件设备的复杂性，通过提供一个标准化的接口集，程序就可以从维护自身硬件通信代码的负担中解放出来，从而减少潜在的错误并加快开发时间。

为了有效地编写操作系统，程序员需要很好地理解程序员为编写操作系统而依赖的底层计算机架构。第一个原因是，许多操作系统概念都由架构支持，例如虚拟内存的概念得到了x86架构的良好支持。如果对底层计算机架构理解不充分，操作系统开发者注定要在自己的操作系统中重新发明它，并且这种软件实现的解决方案比硬件版本运行得更慢。

9.2 驱动程序

驱动程序是使操作系统能够与硬件设备通信并使用其功能的程序。例如，键盘驱动程序使操作系统能够从键盘获取输入；或者网络驱动程序允许网卡向互联网发送和接收数据包。

如果您只编写应用程序，可能会想知道软件如何控制硬件设备？如第二章所述，通过硬件-软件接口：通过向设备的寄存器写入或向设备的端口写入，通过使用CPU的指令。

9.3 用户空间和内核空间

Kernel space 指向仅内核可以访问的操作系统的工作环境。内核空间包括与硬件的直接通信或操作特权内存区域（例如内核代码和数据）。

与之一致，*userspace* 指的是运行在操作系统之上的较低权限进程，并由操作系统监督。要访问内核功能，用户

程序必须通过操作系统提供的标准化系统编程接口。

9.4 内存段 9.5 段描述符 9.6 段描述符类型

9.6.1 Code and Data descriptors

9.6.2 Task Descriptor

9.6.3 Interrupt Descriptor

9.7 描述符范围

9.7.1 Global Descriptor

9.7.2 Local Descriptor

9.8 段选择器

9.9 增强功能：具有描述符的引导加载程序

10

Process

10.1 Concepts

10.2 Process

10.2.1 Task

一个 *task* 是操作系统需要执行的工作单元，类似于人类每天需要完成的任务。从用户的角度来看，计算机需要执行的任务可以是网页浏览、文档编辑、游戏、发送和接收电子邮件等。由于CPU只能顺序执行，一个指令接一个指令（从主内存中获取），因此必须有一种方法可以同时执行许多有意义的任务。因此，计算机必须在任务之间共享资源，例如寄存器、堆栈、内存等，因为我们有多个任务，但资源和资源是单一且有限的。

10.2.2 Process

Process 这是一个跟踪任务执行状态的数据结构。任务是一个通用概念，进程是任务的实现。在通用操作系统（OS）中，任务通常是一个程序。例如，当你运行Firefox时，会创建一个进程结构来跟踪为Firefox分配的堆栈和堆的位置，Firefox的

代码区域是以及EIP将要执行的指令等。典型的流程结构看起来像这样
:

[插入流程图]

进程是一个虚拟计算机，但比虚拟化软件（如Virtual Box）中的虚拟机要原始得多，这其实是个好事。想象一下，如果每个任务都需要运行一个完整的虚拟机，那将多么浪费机器资源.. 从运行进程的角度来看，其代码执行就像它直接在硬件上运行一样。每个进程都有自己的寄存器值集合，这些值由操作系统跟踪，以及自己的连续虚拟内存空间（在物理内存中实际上是离散的）。进程中的代码被赋予虚拟内存地址以进行读写操作。

说明：- 一个过程看起来像一个小型的冯·诺依曼 - 带有连续的内存，每个都有颜色；每个过程单元格映射到物理内存中的远程内存单元格 {v *}]

一个进程可以运行到操作系统通知它暂时停止以供其他任务使用硬件资源。暂停的进程可以等待来自操作系统的进一步通知。整个切换过程如此之快，以至于计算机用户认为他的计算机实际上在并行运行任务。负责在任务之间进行切换的程序被称为“调度器”。

10.2.3 Scheduler

一个操作系统需要执行广泛的不同的功能，例如网页浏览、文档编辑、游戏等。一个 *scheduler* 决定哪些任务在先于其他任务运行，以及运行多长时间，以高效的方式进行。调度器使您的计算机成为一个 *time sharing system*，因为任务共享CPU执行时间，没有任何一个进程可以垄断CPU（在实践中，这种情况仍然经常发生）。没有调度器，一次只能执行一个任务。

10.2.4 Context switch

当进程准备被切换出，以便另一个进程取代其位置时，某些硬件资源，即当前打开的文件、当前注册的

ter 值等必须备份，以便稍后恢复该进程的执行。

10.2.5 Priority

Priority 这是一个操作系统决定在其它任务之前哪个任务应该运行的重要指标，以便为每个任务分配适当的CPU执行时间。

10.2.6 Preemptive vs Non-preemptive

一个 *preemptive* 操作系统可以中断正在执行的过程并切换到另一个过程。

一个 *non-preemtive* 操作系统，一个任务会一直运行直到完成。

10.2.7 Process states

State 这是一个过程的特定条件，由调度器的动作触发。一个过程在其生命周期中会经历各种状态。一个过程通常具有以下状态：

Run 指示CPU正在此进程中执行代码。

Sleep (或挂起)：表示CPU正在执行其他进程。

Destroyed: 流程已完成，等待完全销毁。

10.2.8 procfs

10.3 线程

Threads 工作单元是具有共享执行环境的进程的一部分。进程使用自己的代码创建一个全新的执行环境：

[进程与线程之间的插图，每个进程是一个大矩形框，嵌套的线程小框指向不同的代码区域]

而不是在内存中创建一个全新的进程结构，操作系统只是让线程使用创建它的父进程的一些资源。线程有自己的寄存器、程序计数器和堆栈

指针及其自己的调用栈。其余的一切都在线程之间共享，例如地址空间、堆、静态数据和代码段以及文件描述符。因为线程只是重用现有资源且不涉及上下文切换，所以创建和切换进程要快得多。

然而，请注意，上述方案只是线程概念的实现。您可以完全将线程视为进程（因此您可以称所有进程为线程，反之亦然）。或者，您可以选择备份一些资源，同时保留一些资源共享。区分线程和进程取决于操作系统设计者。线程通常作为进程的一个组件实现。

在Linux中，一个线程仅仅是与其父进程共享资源的进程；因此，Linux线程也被称为*轻量级进程*。或者换句话说，Linux中的线程仅仅是单线程进程的实现，它执行其主程序代码。Linux中的多线程程序只是一个与单线程子进程共享的进程，每个子进程指向其父进程的不同代码区域。

[待办：将上述表格转换为图表]

在Windows上，线程和进程是两个独立的实体，因此上述针对Linux的描述不适用。然而，基本思想是：线程共享执行环境，持有。

10.4 任务：x86进程概念 10.5 任务 数据结构

10.5.1 Task State Segment

10.5.2 Task Descriptor

10.6 流程实施

10.6.1 Requirements

10.6.2 Major Plan

10.6.3 Stage 1: Switch to a task from bootloader

10.6.4 Stage 2: Switch to a task with one function from kernel

10.6.5 Stage 3: Switch to a task with many functions from kernel

To implement the concept of a process, a kernel needs to be able to save and restore its machine states for different tasks.

Description [Describe task switching mechanism involving LDT and GDT]

qasdfasdf asd

Constraints

Design

Implementation plan

10.7 里程碑：代码重构

11

Interrupt

12

Memory management

12.0.1 Address Space

Address space 是所有可寻址内存位置的集合。物理内存地址中有两种地址空间类型:

- 一个用于记忆:
- 一个用于I/O:

每个进程都有自己的地址空间，可以随心所欲地做任何事情，只要物理内存没有耗尽。这个地址空间被称为*virtual memory*。

12.0.2 Virtual Memory

物理内存是一种具有简单映射关系的传染性内存位置，该映射关系存在于物理内存地址及其对应的内存位置之间，由内存控制器解码。另一方面，*虚拟内存*在内存地址与其对应的物理内存位置之间没有直接映射，尽管从用户空间程序的角度看它似乎具有传染性。相反，虚拟内存地址由操作系统转换为实际的物理内存地址。因此，即使在虚拟内存空间中地址相邻，它们也会在物理内存中分散分布。

为什么需要虚拟内存？因为虚拟内存通过给每个程序一个拥有自己独立“物理”内存的错觉，从而降低了编程的复杂性。没有虚拟内存，程序必须知道并同意彼此的内存区域，以避免意外破坏彼此。

[描绘一个没有虚拟内存的世界]

虚拟内存还能使操作系统更加安全，因为应用程序无法直接操作主内存，所以恶意程序不会通过破坏主内存和可能破坏硬件设备来造成破坏，通过访问硬件I/O端口。

另一个好处是虚拟内存可以扩展到物理内存之外，通过将数据存储到硬盘上。通过交换一些未使用的内存（即休眠进程的不活跃内存），系统可以获得一些空闲内存以继续运行，因此不会破坏数据。否则，操作系统被迫随机终止一个用户进程以释放一些内存，您可能会丢失属于被终止进程的未保存工作。然而，由于冯·诺依曼瓶颈，这个过程可能会显著减慢整个系统。在内存稀缺的过去，这很有用。

13

File System

File system 这是一个机制，用于如何有意义地管理存储设备中的原始字节。也就是说，存储设备中特定位置的多个字节可以分配给某个目的，例如存储原始ASCII文档，之后可以正确检索到确切的字节块。文件系统管理许多这样的字节组。将文件系统视为一个数据库，它将高级信息与硬盘上的特定位置进行映射，类似于如何将业务信息映射到表中的特定行。与文件系统相关的高级信息组织为*文件*和*目录*。

[文件系统和数据库表之间的插图，以了解它们的相似之处]

File 是一个包含两个组件的实体：元数据和实际原始数据。*Metadata*是描述与文件相关的原始数据属性的信息；原始数据是文件的实际内容。*Directory*是一个包含一组文件和子目录的文件。它们共同创建了一个类似于Windows或Linux中常见的文件层次结构系统。

13.0.1 Example: Ex2 filesystem

Index

Bibliography

G. H. Hardy. *A Mathematician's Apology*，第10章，第13页。阿尔伯塔大学数学科学协会，2005年。

英特尔。*IntelÂ® 64 and IA-32 Architectures Optimization Reference Manual*。英特尔，2016b。

本杰明·L科维茨。*Practical Software Requirements*，第3章，第53页。Manning，1999。

查尔斯·桑德斯·皮尔士。*Collected Papers v. 4*，第A布尔代数与一个常数的章节。1933。

约翰·F沃克利。*Digital Design: Principles and Practices*，第3章，第86页。普伦蒂斯·霍尔，1999年。