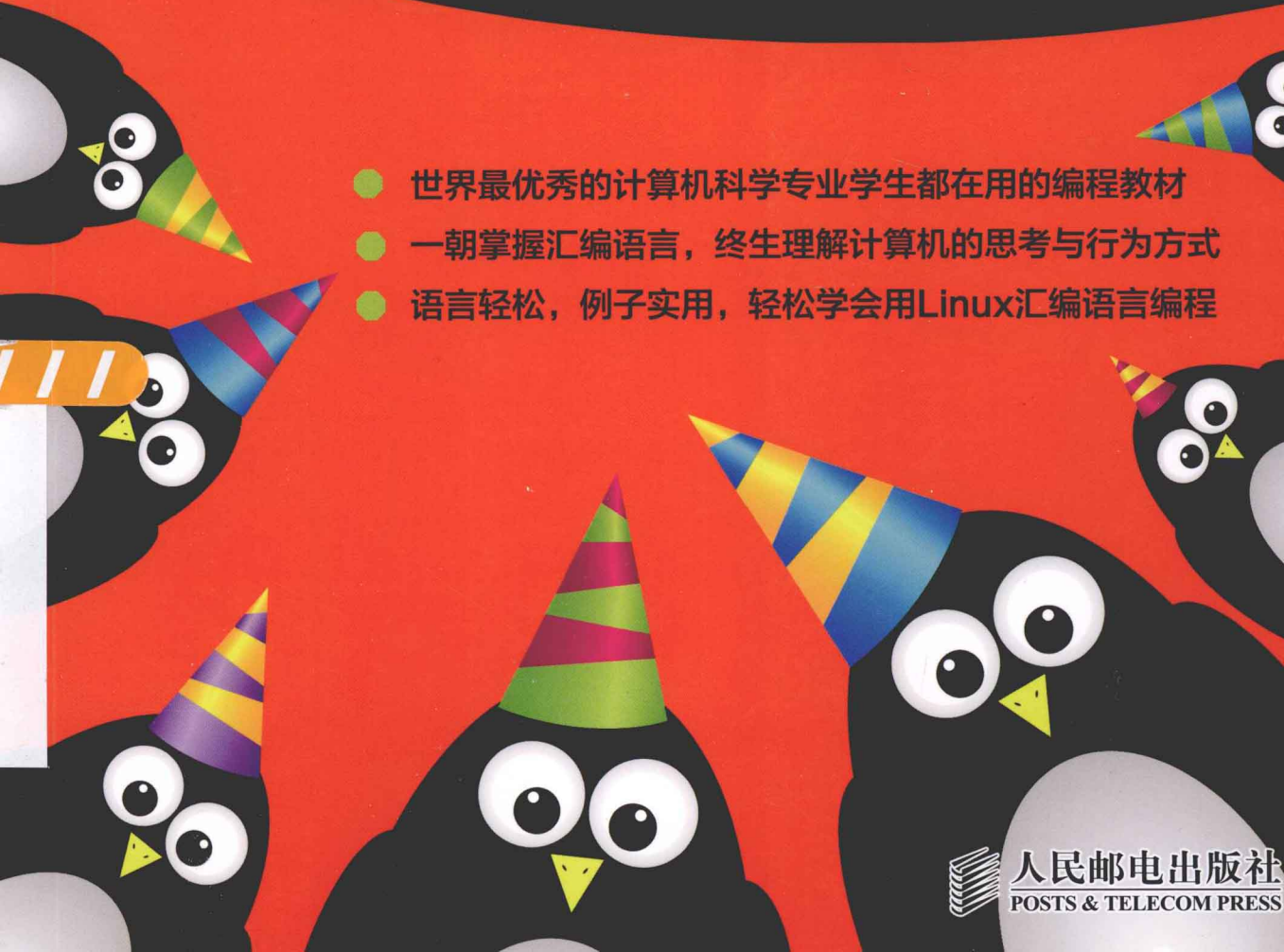
TURING 图灵程序设计丛书

Programming
from the Ground Up

深入理解程序设计
使用Linux汇编语言

[美] Jonathan Bartlett 著　郭晴霞 译



- 世界最优秀的计算机科学专业学生都在用的编程教材
- 一朝掌握汇编语言，终生理解计算机的思考与行为方式
- 语言轻松，例子实用，轻松学会用Linux汇编语言编程

人民邮电出版社
POSTS & TELECOM PRESS

Programming
from the Ground Up

深入理解程序设计
使用Linux汇编语言

是否真正理解汇编语言，常常是普通程序员和优秀程序员的分水岭。《深入理解程序设计：使用Linux汇编语言》介绍了Linux平台下的汇编语言编程，教你从计算机的角度看问题，从而了解汇编语言及计算机的工作方式，为成就自己的优秀程序员之梦夯实基础。

很多人都认为汇编语言晦涩难懂，但New Medio技术总监Jonathan Bartlett的这本书将改变人们的看法。本书首先介绍计算机的体系结构，然后从编写简单程序开始，一步一步扩充函数、文件、读写处理等知识，并平滑过渡到程序共享、存储与优化，由浅入深地介绍了Linux汇编语言编程。作者不仅会带你了解向计算机传递信息的方式方法，还让你学会向修改和使用程序的人传递信息，并最终用自己的规则构建“世界”，按自己对问题的理解和解决方案创造“世界”。

主要内容：

- 计算机体系结构（详解内存及寻址方式）；
- 函数使用及复杂度处理；
- 记录读写及修改；
- 程序共享；
- 计算机的计数原理；
- 编程初体验；
- 文件处理及缓冲区分析；
- 通过测试及错误处理打造健壮程序；
- 内存布局及处理；
- 程序优化（时机、位置及方式）。

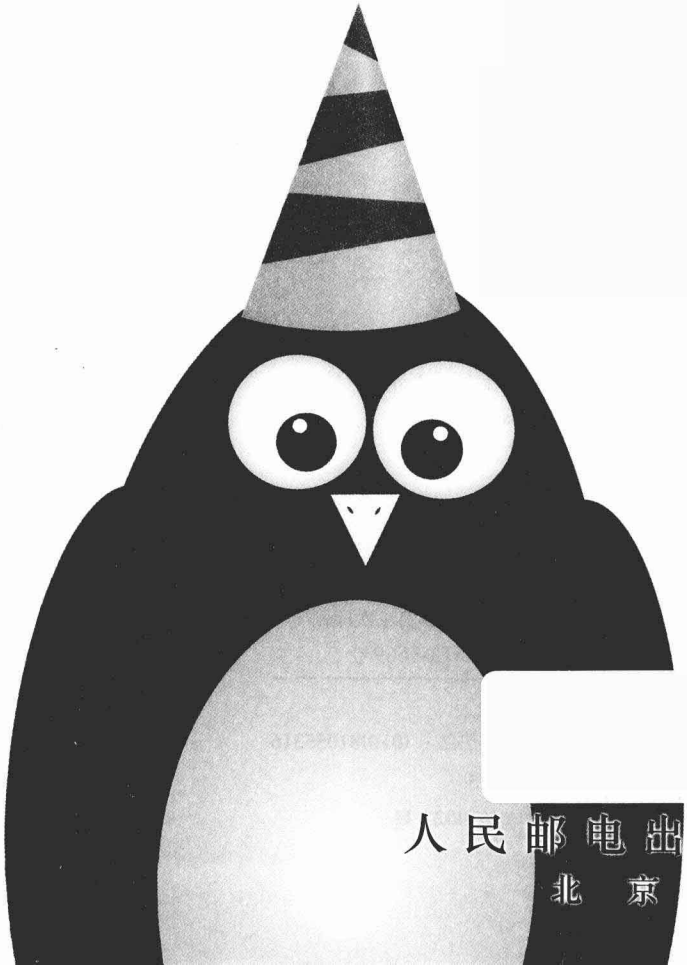
图灵社区：iTuring.cn
热线：(010)51095186转600
分类建议 计算机/程序设计/汇编语言
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-33788-7
ISBN 978-7-115-33788-7
定价：49.00元

TURING 图灵程序设计丛书

Programming
from the Ground Up

深入理解程序设计
使用Linux汇编语言



[美] Jonathan Bartlett 著
郭晴霞 译

人民邮电出版社
北京

图书在版编目（CIP）数据

深入理解程序设计 : 使用Linux汇编语言 / (美) 巴特利特 (Bartlett, J.) 著 ; 郭晴霞译. -- 北京 : 人民邮电出版社, 2014.1
(图灵程序设计丛书)
书名原文: Programming from the ground up
ISBN 978-7-115-33788-7

Ⅰ. ①深… Ⅱ. ①巴… ②郭… Ⅲ. ①Linux操作系统－程序设计 Ⅳ. ①TP316.89

中国版本图书馆CIP数据核字(2013)第281437号

内 容 提 要

《深入理解程序设计：使用 Linux 汇编语言》深入浅出地介绍 Linux 汇编语言编程，旨在让程序员真正理解汇编语言，从计算机的角度理解编程。无论其是否实际使用汇编语言，本书最终将使其能够以协调、优雅的方式解决问题，并把解决方案传授给未来的程序员。本书内容主要包括内存管理、汇编语言与C语言的接口技术、动态库，以及一些 GUI 编程知识。

本书适合初中级程序员学习参考，亦是高级程序员的案头备查书。

◆ 著 [美] Jonathan Bartlett
译 郭晴霞
责任编辑 毛倩倩
责任印制 焦志炜

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷

◆ 开本：800×1000 1/16
印张：13
字数：307千字 2014年1月第1版
印数：1－3 500册 2014年1月北京第1次印刷
著作权合同登记号 图字：01-2013-3669号

定价：49.00元

读者服务热线：(010)51095186转600 印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第 0021 号

版权声明

目　录

第1章 引 言

1.1 欢迎加入编程大军

我热爱编程，编写不但能运行而且风格良好的程序是我最喜欢的挑战。编程如同写诗。它不仅是向计算机传递信息，也是向修改和使用程序的人传递信息。有了程序，你就能用一套规则构建自己的世界，按自己对问题的理解和自己构思的解决方案来创造自己的世界。高明的程序员可以用诗篇或散文般简明的程序来构建世界。

作为最伟大的程序员之一，Donald Knuth这样向我们描述编程：编程并非告诉计算机如何做某件事，而是告诉人们程序员如何指示计算机做某件事。这里的关键在于：程序不仅仅是给计算机看的，更是给人看的。当你转向其他项目后，你的程序将由其他人修改和更新。因此，编程并非只需要和计算机交流，更意味着要和接替你的程序员沟通。程序员不仅是问题的解决者，同时也是诗人和教师。作为程序员，你的目标是以协调、优雅的方式解决手边的问题，并把自己的解决方案传授给未来的程序员。希望本书至少能有这个荣幸：向你传授一些让使用计算机变得令人兴奋的诗意和魔力。

多数入门编程书籍让我感到非常失望。读完这类书之后，我们仍然会问“计算机到底是怎么工作的”，并且不能获得圆满的答案。这些书遇到那些艰深的主题往往就语焉不详，即使这些主题非常重要。在本书中，我将会引导你理解这些难懂的问题，因为这是成为编程高手的唯一途径。我的目标是让那些对编程一无所知的人懂得如何像程序员那样思考、编程和学习。当然，读过本书之后，你不可能掌握与编程有关的一切，但将会对所有这一切是如何相互配合的有所了解。认真阅读本书后，你应该能够做到以下几点：

- 了解程序的工作原理以及一个程序如何与其他程序交互；
- 学会阅读他人的程序并了解其工作原理；
- 快速学习新的编程语言；

- 快速学习先进的计算机科学概念。

当然，我无法在这本书中将与编程有关的一切传授给你。因为计算机科学是个宽广的领域，尤其在涉及计算机编程理论与实践相结合时更是如此。然而，我会尝试让你打好基础，这样，以后你就能比较容易地向任何感兴趣的方向发展。

教授编程特别是汇编语言编程，与“先有鸡还是先有蛋”的问题颇有几分类似。在这里，有很多东西需要学习，要马上学会这一切确实有点勉为其难。但由于各部分知识是相互联系的，所以学习编程时，你只要一以贯之地耐心对待你自己和计算机，就会较快地学有所成。在学习本书的过程中，对第一次阅读时未能理解的内容，请再读一遍。如果仍然不明白，也许暂时记住它，稍后再去回顾和理解会更好。因为通常情况下，必须在对编程有更多接触后，那些概念才会变得更易理解。不要气馁，这个过程尽管漫长而艰难，但一切努力都是值得的。

本书每章的末尾都有三组复习题：第一组用于回想，检查你是否记得在该章学到的知识；第二组涉及实际应用，检验你是否能运用学到的知识解决问题；最后一组是为了检验你能否拓展视野、举一反三。最后一组题目中的有些问题要等学习了后面的章节才能回答，但它们会让你带着问题学习后续的内容。有些问题需要研究本书之外的资料才能找到答案，还有一些问题要求简单分析你的选择，对最佳方案加以说明。需要指出的是，许多问题的答案并无对错之分，但这并不意味着它们不重要。学习编程中所涉及的问题、学习如何通过调研找到答案、学习编程时如何考虑未来的变数，这些都是程序员工作的重要组成部分。

如果有实在无法解答的问题，读者可借助本书的邮件列表pgubook-readers@nongnu.org讨论书中内容并获得帮助。（通过此邮件列表，读者可提出与本书有关的任何问题。）你可以通过访问http://mail.nongnu.org/mailman/listinfo/pgubook-readers来订阅此列表。

1.2　工具

本书讲解基于x86处理器和GNU/Linux操作系统的汇编语言，因此我们给出的所有例子都使用GNU/Linux标准的GCC工具集。如果你不熟悉GNU/ Linux和GCC工具集，可以阅读本书后面的简单介绍。如果你是Linux新手，请查阅http://rute.sourceforge.net提供的指南[①]。本书的目的主要是向读者展示如何编程，而不是如何在某个特定平台上使用特定的工具集。当然，统一使用一个平台和工具集，会使本书的描述变得较为简单。

Linux新手应努力加入当地的GNU/Linux用户组。用户组的老成员通常乐于帮助新人，他们

① 这是一份很长的文档。你无需全部阅读后再开始学习本书，只需知道如何使用命令行和pico、emacs、vi或其他编辑器。

会为新手提供全面帮助，从安装Linux系统到学习如何最有效地使用Linux。GNU/Linux用户组列表可在http://www.linux.org/groups获得。

本书所有程序都已用Red Hat Linux 8.0进行了测试，同时也能运行于其他GNU/Linux版本上[①]。虽然本书程序无法运行在BSD等非Linux操作系统上，但在本书中学到的编程方法和技能，在其他任何系统上都是通用的。

如果没有安装了GNU/Linux系统的机器，读者可以找一个提供Linux shell账号（shell是Linux系统的命令行界面）的托管服务提供商。有很多低价的shell账号供应商，但你必须确保他们提供的是x86上的Linux系统服务。此外，你当地GNU/Linux用户组的人或许也能给你一个shell账号。只要能上网，通过telnet程序就可以使用你的shell账号。如果你使用的是Windows系统，单击“开始”→“运行”，再键入`telnet`即可。但因为Windows自带的telnet存在一些奇怪的问题，所以最好从http://www.chiart.greenend.co.uk/~sgtatham/putty/下载PuTTY程序。使用Macintosh系统的读者也有许多选择，NiftyTelnet就是我最喜欢用的程序。

如果没有GNU/Linux系统，也无法获得shell账号服务，那么可从http://www.knoppix.org/下载Knoppix。Knoppix是从CD引导的GNU/Linux 版本，无需进行安装。使用完毕后，只需重新启动系统并移除CD，即可回到你的常用操作系统。

说了这么多，GNU/Linux到底是什么呢？简而言之，GNU/Linux是仿照UNIX开发的操作系统。GNU/Linux的GNU部分来自GNU项目（参见http://www.gnu.org/）[②]，由包含GCC工具集在内的绝大多数Linux程序组成。GCC工具集包含了编写多种计算机语言程序所需的全部开发工具。

GNU/Linux中的Linux是操作系统*内核*的名称。内核是操作系统的核心部分，掌控系统的一切活动。可以把内核看成是系统的大门和围墙。作为大门，内核使程序以统一的方式访问硬件。如果没有内核，你就得自行编写程序才能操作所有设备。内核帮我们处理与具体设备的所有交互，我们就可以不用去操心这些事了。内核也处理文件访问以及进程间交互。例如，你从键盘输入的内容要经过以下几个步骤才会到达编辑器：首先，掌管硬件设备的内核接收到按键通知，键盘发送*扫描码*到内核；然后，内核将这些扫描码转换为实际对应的字母、数字或符号；最后，如果你使用的是窗口操作系统（如Microsoft Windows或X Window系统），窗口系统会从内核读取按键，并发送给用户显示器上获得了焦点的程序。

① 这里所说的GNU/Linux版本是指x86 GNU/Linux版本。本书的程序不适合运行在针对Power Macintosh、Alpha或其他处理器的GNU/Linux版本上。

② GNU项目由自由软件基金会发起，其主要使命是推出一个完整的自由操作系统。

示例1-1　计算机如何处理键盘信号

```
键盘 -> 内核 -> 窗口系统 -> 应用程序
```

内核也控制着程序之间的信息流。内核是程序通往周边世界的大门，控制着数据在进程间传递时的消息收发。在上面的示例中，窗口系统要将按键传递给应用程序，就离不开内核的参与。

作为“围墙”，内核会阻止程序意外覆盖其他程序的数据或访问其无权访问的文件和设备。内核的这些功能可减轻写得不好的程序给其他正在运行的程序造成的危害。

在本书中，内核就是Linux。但我们知道，只有内核什么也做不了，连启动计算机都不行。说得形象一点儿，可以把内核设想成屋里的水管：如果没有水管，水龙头就无法工作；但没有水龙头，水管也毫无用处。因此，完整的GNU/Linux操作系统是由用户应用程序（来自GNU项目及其他地方）和内核（Linux）共同构成的。

计算机语言一般分为以下三类，本书主要讲述较低级的汇编语言。

- 机器语言

机器语言是计算机可直接识别和处理的语言。在机器语言中，计算机能识别的每条命令都以数字或数字串的形式给出。

- 汇编语言

除了用易于记忆的字母序列代替数字命令外，汇编语言与机器语言差不多，另外还添加了一些便捷的辅助语法。

- 高级语言

高级语言旨在使编程更容易。汇编语言要求你工作时面向机器本身，而高级语言则让你以更接近自然语言的形式来表述程序。高级语言的一条命令，往往相当于汇编语言的几条命令。

在本书中，我们将主要学习汇编语言，但对高级语言也稍有涉猎。希望通过学习汇编语言，你能理解程序和计算机的工作原理，从而在编程之路上迈出一大步。

第2章 计算机体系结构

在学习编程前，你首先要了解计算机如何解释程序。当然，你无需拥有电子工程学位，但需要理解一些基本概念。

现代计算机的体系结构都是在冯·诺依曼体系结构（因其创始人而得名）的基础上发展起来的。冯·诺依曼体系结构将计算机划分成两个主要组成部分：CPU（中央处理单元）和内存。这种架构被用在包括个人电脑、超级计算机、大型机在内的所有现代计算机，甚至是手机中。

2.1 内存结构

为理解计算机内存的结构，可以将计算机内存与你家附近的邮局作个比较。邮局通常有一个摆满邮政信箱的房间，计算机内存就像邮局房间里的信箱一样，每个固定大小的存储单元都依次编号。例如，大小为256 M的内存大致包含2.56亿个固定大小的存储单元；如果与邮局类比，就是2.56亿个信箱。计算机内存的每个存储单元都有一个编号，所有存储单元都具有相同的固定大小。邮政信箱与计算机内存的不同之处在于：信箱中能存放各种东西，但内存的存储单元中只能存入一个数字。

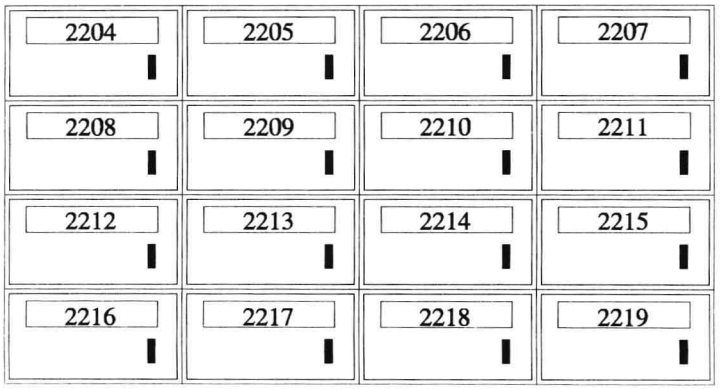


图2-1 像邮政信箱一样的存储单元

你也许想知道为什么计算机如此设计。答案是：这样比较容易实现。设想一下，如果组成计算机内存的存储单元的大小各不相同，或者能在不同的存储单元存储不同类型的数据，那么实现起来不仅困难，而且成本太高。

计算机内存有多种用途。计算机的所有计算结果都存储在内存中。事实上，所有要“存储”的内容都会先存储在内存中。以你家里的计算机为例，想想当它运行时，在它的内存里存放了哪些东西。通常有（但不限于）这些：

- 你的光标在屏幕上所在的位置；
- 屏幕上每个窗口的大小；
- 所使用的字体中每个字母的形状；
- 每个窗口上所有控件的布局；
- 工具栏上所有图标的图形；
- 所有的错误信息和对话框的文本。

除上述内容外，在采用冯·诺依曼体系结构的计算机中，不仅计算机数据应存放在内存中，而且正在控制计算机操作的程序也应存放在内存中。事实上，对于计算机来说，程序和数据的区别仅仅在于计算机使用两者的方式不同，两者的存储和访问方式是一样的。

2.2　CPU 构造

那么，计算机是怎样运行的呢？显然，仅仅能存储数据是不行的，还必须能访问、操作以及移动数据。这时候，就要用到CPU了。

CPU一次从内存中读取一条指令并执行，CPU执行指令的上述步骤就是所谓的*读取–执行周期*（又称为*指令周期*或*机器周期*）。为完成此功能，CPU需包含以下元件：

- 程序计数器；
- 指令解码器；
- 数据总线；
- 通用寄存器；
- 算术逻辑单元。

*程序计数器*用来告诉计算机从哪里提取下一条指令。我们之前曾提到，数据和程序的存储方式并无区别，只是CPU对两者的解释方式不同。程序计数器保存即将执行的下一条指令的内存地址。CPU先查看程序计数器，然后提取存放在指定内存地址的数字，接着传递给*指令解码器*，由它来解释指令。指令解码器给出的解释包括：需要进行何种处理（加法、减法、乘法、移动数据

等），以及处理过程中将会涉及哪些内存单元。计算机指令通常由实际指令和执行指令要用到的内存单元列表构成。

接着，计算机使用*数据总线*取得存储在内存单元中的用于计算的数据。数据总线是CPU和内存间的物理连线，是联系两者的纽带。计算机主板上的那些从内存延伸出来的排线，就是数据总线。

除了位于CPU外面的内存，CPU内部还包含一些被称为*寄存器*的特殊高速存储单元。CPU中有两种寄存器：*通用寄存器*和*专用寄存器*。通用寄存器是进行主要运算的地方，一般用来处理加法、减法、乘法、比较及其他运算。但计算机CPU中的通用寄存器很少，计算时用到的绝大多数信息都存储在主内存中，只有在CPU处理时才提取到通用寄存器，处理完成后再放回内存。*专用寄存器*是用于特定用途的寄存器，稍后我们再介绍。

CPU在取回需要的所有数据后，将数据和经过解码的指令传递给*算术逻辑单元*进一步处理。*算术逻辑单元*是实际执行指令的地方。它得出计算结果后，将结果经数据总线传到指令指定的相应内存单元或寄存器。

以上是对CPU处理指令过程的一个相当简化的描述。最近几年，CPU已经有了很大的发展，变得越来越复杂。尽管基本操作仍然相同，但新的CPU使用了多层高速缓存结构、超标量处理器、流水线、分支预测、乱序执行、微代码转换、协处理器以及其他复杂的优化技术。不知道这些词的意思也没关系，要是你想深入了解CPU，可在互联网上搜索一下这些词。

2.3 几个术语

计算机内存是经过编号的一系列固定大小的存储单元。每个存储单元的编号称为该存储单元的*地址*。一个存储单元的大小称为一个*字节*。在x86处理器上，一个字节对应一个0到255之间的数字[①]。

你可能想知道为什么计算机只能存储0到255之间的数，却能显示和处理文本、图形和更大的数？首先，显卡等专用硬件对每个数字都有专门的解释。当在屏幕上显示数字时，计算机根据ASCII码表将你发送给它的每一个0到255之间的数字转换为显示在屏幕上的字符，每个数字对应一个字母或一个10以内的数字[②]。例如，大写字母A用数字65表示，而数字1则用49表示。因此，要在屏幕上打印出“HELLO”，你实际上要把数字序列72、69、76、76、79发送给计算机；而要打印出数字100，你要把数字序列49、48、48发给计算机。ASCII字符及其

① 即8位二进制数所能表示的十进制数。——译者注

② 随着国际字符集和Unicode的出现，这么说就不完全正确了。这里，为了便于初学者理解，我们仍假设一个数字直接转化为一个字符。欲了解更多信息，请参阅附录D。

数字码的列表参见附录D。

对于程序员来说，数字不仅能表示ASCII字符，还能表示你希望它表示的任何信息。例如，如果我经营着一家商店，那我会用一个数字代表一件商品，每个这样的数字实际上都对应另一串数字，也就是扫描此商品时会显示的那串ASCII码。此外，我还希望用更多的数字表示价格、库存等信息。

那么，如果我们需要大于255的数字该怎么办呢？很简单，只要把多个字节组合起来，就可以表示较大的数字了。这样，两个字节就可以表示0到65 536之间的任何数字，四个字节就能表示0和4 294 967 295之间的任何数字。然而，写程序的时候要通过组合字节来增大数字颇为困难，这需要运用一点数学知识。幸运的是，计算机能帮我们表示多达4字节长的数值。事实上，4字节长的数值也是默认情况下我们要使用的数值长度。

前面提到过，计算机除了有常规内存外，还有称为*寄存器*的特殊存储单元。寄存器是计算机运算时用到的元件。可将寄存器设想为你书桌上的一块空间，用于临时存放正在使用的东西。你可能把许多资料放在文件夹和抽屉中，但当前正在使用的东西总是放在桌上。寄存器就是用来保存你当前正在操作的数字内容的。

在我们当前使用的计算机上，每个寄存器的大小为4字节。计算机中典型寄存器的大小称为计算机的*字长*。x86处理器的字长为4字节。这意味着，x86计算机能一次计算4字节，大约可以表示40亿个值。

地址的长度也是4字节（1个字长），因此也适合放入寄存器。这一方面意味着，如果计算机的内存足够大，x86处理器可以访问多达4 294 967 296字节的内存；另一方面，也意味着我们可以用存储数字的方式来存储地址。事实上，电脑不会区分地址值、数字值、ASCII码值或其他什么值。当你要显示数字时，它就是一个ASCII码值；当你要寻找数字指向的字节时，它就变成了地址。花点时间好好理解一下上述内容，这对于理解计算机程序如何工作至关重要。

由于地址中存储的值并不是常规值，而是指向内存中不同位置的数字，因此存储在内存中的地址也称为*指针*。

正如我们前面所说的，计算机指令也存储在内存中。事实上，由于指令与数据在内存中的存储方式完全相同，计算机知道某个存储单元是指令的唯一途径，就是有被称为指令指针的专用寄存器在某个时刻指向了它。如果指令指针指向一个内存字节，该字节就作为一条指令被加载。除了这种途径以外，计算机无法区分程序和其他类型的数据[①]。

① 需要指出的是，这里我们讨论的是通用计算机理论。有些处理器和操作系统实际上会以特殊标记标明可被执行的内存区。

2.4 内存详解

计算机的精确要求程序员做到同样的精确。作为一台机器，计算机并不知道你的程序要做什么。因此，你告诉计算机做什么，它才会做什么。如果你不小心输出了一个常规的数值，而不是构成数值中每个数字的ASCII码，计算机将不会阻止你，而你的屏幕上将出现一串乱码（计算机会尝试寻找此数值对应的ASCII码，然后打印出来）。如果你让计算机从包含数据而非程序指令的存储单元开始执行指令，那么就没人知道计算机会对此如何解释，尽管它肯定会尝试解释。总之，计算机会完全以你指定的顺序执行你的指令，即使这样做毫无意义。

可见，计算机会忠实地执行你的指令，无论它是否有意义。因此，作为程序员，你要确切地知道数据在内存中是如何放置的。由于计算机只能存储数字，因此文字、图片、音乐、网页、文件和其他内容在计算机中都只是长串的数字，只有对应的特定程序知道如何解释它们。

例如，如果你想将客户信息存储在内存中，那么一种方式是首先设置客户姓名和地址所要占用的最大空间量（如各占50个ASCII字符，即50字节），然后用数字表示客户年龄和客户ID。采用这种存储方式，一条客户记录占据的内存空间将如下所示：

```
记录起始处：
    客户姓名（50 字节） - 记录起始处
    客户地址（50 字节） - 记录起始处 + 50 字节
    客户年龄（1个字 - 4 字节） - 记录起始处 + 100 字节
    客户ID号（1个字 - 4 字节） - 记录起始处 + 104 字节
```

在这种存储方式中，只要给出客户记录的起始地址，你就能知道该客户的其他数据在哪里。但该存储方式将客户姓名和地址限制在50个ASCII字符以内，有可能无法满足实际需求。

如何消除这样的限制呢？我们可以采用另一种存储方式：用指针指向记录中的信息。例如，可以用指向客户姓名的指针取代客户姓名字段。采用这种存储方式，一条客户记录在内存中将如下所示：

```
记录起始处：
    客户姓名指针（1个字） - 记录起始处
    客户地址指针（1个字） - 记录起始处 + 4 字节
    客户年龄（1个字） - 记录起始处 + 8 字节
    客户ID号（1个字） - 记录起始处 + 12 字节
```

而实际的客户姓名和地址将存储在内存中的其他位置。采用上述方式，我们不仅易于确定每条记录的每部分数据相对记录起始处的位置，而且无需限制客户姓名和地址所占用的空间。不论采取上述哪种存储方式，如果记录中字段的长度是可变的，那就没办法知道下一个字段是从何处

开始的了；因为字段可变造成每条记录的长短又会有所不同，所以要找到下一条记录的开始位置更是难上加难。因此，几乎所有的记录都是固定长度的。长度可变的数据通常与记录的其余部分分开存储。

2.5　寻址方式

计算机处理器有多种不同的数据访问方式，它们被称为寻址方式。最简单的寻址方式是*立即寻址方式*，在这种寻址方式下，指令本身即包含要访问的数据。例如，如果我们想把寄存器初始化为0，那么可以使用立即寻址方式，在指令中直接给出数字0，而不是告诉计算机要到哪个地址去读取0。

*在寄存器寻址方式*中，指令中包含要访问的寄存器，而不是内存位置。

除了上述两种寻址方式外，其他余的寻址方式都与地址有关。

*在直接寻址方式*中，指令中包含要访问的内存地址。例如，直接寻址的指令可能为：请将地址2002中的数据加载到这个寄存器。按照这个指令，计算机将直接读取字节编号为2002的内存中的内容，并将其复制到寄存器。

*在变址寻址方式*中，指令中除包含一个要访问的内存地址外，还要指定一个*变址寄存器*，其中包含该地址的偏移量。例如，我们可以指定内存地址2002和一个变址寄存器。如果该变址寄存器包含数字4，那么实际用于加载数据的地址就是2006。利用这种寻址方式，如果你有起始位置为2002的一组数字，那就可以使用变址寄存器循环提取每个数字。在x86处理器中，还可以指定变址的*比例因子*，这样就能以一次一字节或一个字（4字节）的方式访问内存。比如，如果你正在访问一个完整的字，那么变址需要乘以4（即比例因子是4）才能得到第四个字相对当前地址的确切位置。如果你想访问从内存地址2002开始的第四个字节，因为是一次访问一字节，你就要把3加载到变址寄存器（我们从0开始计数），并设置比例因子为1。这样，你就会得到位置2005的数据。但如果你要访问从2002开始第四个字的位置，就要把3加载到变址寄存器，并设置比例因子为4，结果是从位置2014（即第四个字的起始位置）加载数据。花点时间认真验算一下，确保你切实了解了上述工作原理。

*在间接寻址方式*下，指令中包含一个寄存器，该寄存器中存储的是指向要访问数据的指针。比如，如果我们使用间接寻址方式，并指定值为4的`%eax`寄存器，则表示我们要使用内存位置4中的值。同样是这个指令，在直接寻址中，我们将只加载值4；但在间接寻址中，我们会用4作为地址去寻找数据。

最后，还有一种基址寻址方式。这种方式与间接寻址类似，但还须包括一个叫做偏移量的值，将其与寄存器中的值相加后再用于寻址。在本书中，我们将主要使用这个寻址方式。

在2.4节中，我们曾讨论过存储客户信息的内存结构的例子。下面我们以此为例来解释基址寻址方式，设想我们想访问某客户的年龄，也就是其记录的第八个字节的数据，而寄存器中存放着此客户信息的起始内存地址。我们可以使用基址指针寻址，指定寄存器为基址指针，8为偏移量。这与变址寻址很相似，不同之处在于：在基址指针寻址中，偏移量是常数，指针被保存在寄存器中；而在变址寻址中，偏移量存储在寄存器中，而指针是常量。

此外还存在其他寻址方式，但上面这些是最重要的。

2.6 温故知新

2.6.1 理解概念

- 试描述读取–执行周期。
- 什么是寄存器？寄存器怎样让计算变得更加便捷？
- 如何在计算机中表示大于255的数字？
- 在你使用的计算机上，寄存器的大小是多少？
- 计算机怎样知道如何解释内存中给定的一个或一组字节？
- 什么是寻址方式？它们的用途是什么？
- 指令指针的作用是什么？

2.6.2 应用概念

- 你觉得员工记录中应该包含哪些数据？这些记录在内存中应该如何存储？
- 如果我有一个指向某员工记录起始处的指针，并想访问该记录中某个特定的字段，应该使用哪种寻址方式？
- 在基址寻址方式下，如果你有一个寄存器保存着值3122，而偏移量为20，那么你要访问的地址是什么？
- 在变址寻址方式下，如果基址为6512，变址寄存器的值为5，比例因子是4，那么你要访问的地址是什么？
- 在变址寻址方式下，如果基址是123 472，变址寄存器的值为0，比例因子是4，那么你要访问的地址是什么？

- 在变址寻址方式下，如果基址为9 123 478，变址寄存器的值为20，比例因子为1，那么你要访问的地址是什么？

2.6.3 深入学习

- 计算机最少需要几种寻址方式？
- 为什么要实现一些不常用的寻址方式？
- 研究并描述流水线（或其他复杂因素）如何影响读取–执行周期。
- 研究并描述如何在固定长度指令和可变长度指令间取得折中。

第3章

编写第一个程序

在本章中，你不仅将学习如何编写和生成Linux汇编语言程序，还将学习汇编语言程序的结构以及一些汇编语言指令。在通读本章时，你可以酌情参阅附录B和附录F。

刚开始学习时，本章的这些程序可能会让你不知所措。但经过勤奋学习，特别是多次阅读程序及其注释，你将会为未来的学习打下扎实基础。为了增强学习效果，你可以尽你所能地改进这些程序。即使你的改进并不可行，但每一次失败无疑都会对你的学习有所助益。

3.1 编程初体验

我们要接触的第一个程序很简单。实际上，除了退出以外，它什么都没有做！这个程序尽管很短，但却展示了汇编语言和Linux编程的一些基础知识。你必须完全按照我下面所写的内容将程序输入编辑器，该程序的文件名为exit.s。不理解这个程序也没关系，本节只涉及如何输入并运行该程序。在3.2节中，我们将介绍它是如何工作的。

```
# 目的:      退出并向Linux内核返回一个状态码的简单程序
#

# 输入:      无
#

# 输出:      返回一个状态码。在运行程序后可通过输入echo $?来读取状态码
#

# 变量:
#          %eax保存系统调用号
#          %ebx保存返回状态
#
.section .data
```

```
.section .text
.globl _start
_start:
movl $1, %eax        # 这是用于退出程序的Linux内核命令号(系统调用)

movl $0, %ebx        # 这是我们将返回给操作系统的状态码
                     # 改变这个数字，则返回到echo $?的值会不同

int $0x80            # 这将唤醒内核，以运行退出命令
```

你上面键入的内容称为源代码。源代码是供人阅读的程序形式。为将其转换为机器可读的形式，我们需要汇编并链接程序。

第一步是汇编程序。汇编是将你键入的内容转换为机器指令的过程。我们知道，计算机只能读取一组组的数字，但人类更喜欢阅读词。汇编语言就是更方便人类阅读的计算机指令形式。而汇编就是将可供人类阅读的文件转换为机器可读的文件。如要汇编上面的简单程序，应输入以下命令

```
as exit.s -o exit.o
```

在上面的命令中，as是运行汇编程序的命令，exit.s是源文件，-o exit.o告诉汇编程序将输出放在文件exit.o中，exit.o称为目标文件。目标文件是用机器语言写成的代码。目标文件的内容通常不完全放在一处。许多大型程序有多个源文件，通常将每个源文件都转换成一个目标文件。这时，链接器程序将把多个目标文件合而为一，并且向其中添加信息，以使内核知道如何加载和运行该目标文件。在本例中，我们只有一个目标文件，所以链接器只需添加信息使之能运行即可。链接exit.o文件的命令是

```
ld exit.o -o exit
```

这里，ld是运行链接器的命令，exit.o是我们要链接的目标文件，-o exit指示链接器输出新程序到名为exit的文件。[①]如果任一命令报错，那可能是你输入了错误的程序或命令。当你修正程序后，必须重新运行所有命令。切记，对源文件做任何修改后，为在程序中实现这些更改，必须重新汇编和链接程序。要运行exit，你可以键入命令

```
./exit
```

./用来告诉计算机，该程序并非位于常用程序目录下，而是在当前目录下[②]。当你输入此命

① 如果你是Linux和UNIX新手，你可能不知道在上述系统中文件扩展名是可有可无的。事实上，相对于Windows使用.exe扩展名来表示可执行程序，UNIX的可执行文件通常没有扩展名。

② 在Linux和UNIX系统中，.指系统的当前目录。

令后，你会发现：唯一发生的事情就是光标进入到下一行。这是因为该程序的功能只是退出。但如果你在运行程序后立刻键入

```
echo $?
```

屏幕上就会出现一个0。这是因为：每个程序退出时都会返回给Linux一个*退出状态码*，告诉系统一切运行是否正常。你可以通过键入`echo $?`查看此状态码。如果一切正常，就会返回0。UNIX程序如果返回0以外的状态码，就表示失败或其他错误、警告、状态。每个状态码的具体含义通常由程序员决定。

下一节，我们将介绍本节程序每行代码的功能。

3.2 汇编语言程序概要

让我们通过刚刚输入的程序来看看汇编语言程序是怎么回事。

在汇编语言程序的开始处，许多行都是以#号开始的，这些行被称为*注释*。汇编程序不会对注释进行任何处理。实际上，注释只是程序员给该程序的未来阅读者提供的阅读辅助信息。由于你在某个程序中编写的多数代码未来将可能被其他程序员修改，养成在代码中加入注释的好习惯，将有助于修改者们理解编写此程序的目的以及此程序的原理。程序的注释通常应包括以下内容：

- 编写这段代码的目的；
- 对该程序所涉及的处理的概述；
- 该程序的特别之处及这样做的原因；[①]

在注释行之后的下一行是：

```
.section .data
```

在汇编程序中，任何以小数点符号（.）开始的指令都不会被直接翻译成机器指令，这些针对汇编程序本身的指令，由于是由汇编程序处理，实际上并不会由计算机运行，因此被称为*汇编指令*或*伪操作*。在本程序中，`.section`指令将程序分成几个部分。`.section .data`命令是数

① 很多程序对问题的处理方式都很特殊，这样做通常是有原因的，但遗憾的是，很多程序员从来不在注释中对这些原因加以记录。因此，未来的程序员只有在修改代码并看着程序崩溃后才明白是怎么回事，或者干脆不知道怎么添加注释，最后也就不加了。作为程序员，你应该毫无保留地记录程序执行时的各种特殊行为。遗憾的是，由于大多数程序缺乏这样的注释，后来的程序员要搞清楚哪些地方有特殊处理，哪些地方只是例行公事，大多只能凭经验。

据段的开始，数据段中要列出程序数据所需的所有内存存储空间。由于该程序没有使用任何数据，所以我们不需要这个段。保留这个指令只是为了保持程序的完整性，因为你将来写的每一个程序几乎都会有数据段。

上述指令之后是：

```
.section .text
```

这表示文本段开始，文本段是存放程序指令的部分。

再下一条指令是：

```
.globl _start
```

这条指令指示汇编程序：_start很重要，必须记住。_start是一个符号，这就是说它将在汇编或链接过程中被其他内容替换。符号一般用来标记程序或数据的位置，所以你可以用名字而非内存位置编号指代它们。试想一下如果你通过地址指代每个内存位置会怎么样？首先，这将令人非常困惑，因为必须记住或查找每一行代码或数据的数字内存地址。此外，每次插入一条数据或代码时，你都必须更改程序的所有地址！

.globl表示汇编程序不应在汇编之后废弃此符号，因为链接器需要它。_start是个特殊符号，总是用.globl来标记，因为它标记了该程序的开始位置。如果不这样标记这个位置，当计算机加载程序时就不知道从哪里开始运行你的程序。

下一行：

```
_start:
```

定义_start标签的值。标签是一个符号，后面跟一个冒号。标签定义一个符号的值。当汇编程序对程序进行汇编时，必须为每个数值和每条指令分配地址。标签告诉汇编程序以该符号的值作为下一条指令或下一个数据元素的位置。这样，如果数据或指令的实际物理位置更改，你就无需重写其引用，因为符号会自动获得新值。

现在我们来看真正的计算机指令。第一条指令如下所示：

```
movl $1, %eax
```

当程序运行时，该指令将数字1移入%eax寄存器。在汇编语言中，很多指令都有操作数。movl有两个操作数——源操作数和目的操作数。在本例中，源操作数是数字1，目的操作数是%eax寄存器。操作数可以是数字、内存位置引用或寄存器。不同的指令允许使用不同类型的操作数。要详细了解哪条指令使用何种类型的操作数，请参阅附录B。

大多数指令都有两个操作数，第一个是源操作数，第二个是目的操作数。注意，在这种情况下，源操作数根本不会改变。addl、subl和imull也是有两个操作数据的指令，它们将源操作数与目的操作数相加、相减或相乘，并将结果存放到目的操作数。有的指令会硬编码操作数，比如idivl要求被除数存放于%eax，且%edx的值为0，运算后将商转存入%eax，余数存入%edx。但是，除数可以是任何寄存器或内存位置的值。

x86处理器上有如下几个通用寄存器[①]（每个都可以用movl指令操作）：

- %eax
- %ebx
- %ecx
- %edx
- %edi
- %esi

除了这些通用寄存器，还有几个专用寄存器，包括：

- %ebp
- %esp
- %eip
- %eflags

专用寄存器我们将在稍后讨论，现在你只要知道存在这些寄存器就行了。[②]其中一些寄存器，如%eip和%eflags只能通过特殊指令访问，其他一些则能使用访问通用寄存器的指令来访问，但这些指令会有特殊含义、用途，或者在以特定方式使用时速度更快。

因此，movl指令将数字1移入%eax。数字1前面的美元符号$表示我们要使用立即寻址方式寻址（参见2.5节）。如果没有美元符号，指令将会进行直接寻址，加载地址1中的数字。我们希望加载实际数字1，所以必须使用立即寻址方式。

① 注意，在x86处理器上，即使是通用寄存器也有一些特殊用途，或是在x86成为32位处理器之前使用。但对于大多数指令，这些都是通用寄存器。尽管每个通用寄存器都在至少一条指令中有特殊用途，然而本书不包括有关大多数通用寄存器特殊用法的指令。

② 你也许会感觉奇怪：为什么所有寄存器都是以字母e开头的？原因是：早期的x86处理器是16位而非32位的。因此，当时的寄存器只有现在的一半长度。后来x86处理器的寄存器大小增加一倍，并保留了旧名以便我们引用寄存器的前半部分，以e开头的名字则用来引用32位的扩展寄存器。通常我们只使用扩展寄存器。较新型的处理器还提供了64位模式，大小再度翻倍，前缀r被用来指代这些更大的64位寄存器（即%rax是64位的%eax）。然而，这些处理器尚未被广泛使用，因此未在本书中介绍。

将数字1移入%eax是因为我们准备调用Linux内核，数字1表示系统调用exit。稍后我们将更深入地讨论系统调用，简单说，通过它们可以请求操作系统的帮助。正常的程序并非无所不能，许多操作如调用其他程序、处理文件及退出等都必须通过系统调用由操作系统处理。进行系统调用时（稍后我们就会这样做）必须将系统调用号加载到%eax（系统调用及相应系统调用号的完整列表，请参阅附录C）。不同的系统调用可能要求其他寄存器也必须含有值。注意，系统调用并非寄存器的唯一甚或是主要用途，只是我们在第一个程序中的一种用途。稍后的程序将使用寄存器进行一般计算。

但除了知晓进行哪个调用，操作系统通常需要更多的信息。例如，在处理文件时，操作系统需要知道你正在处理哪个文件、想要写什么数据和其他相关细节。这些额外的详细信息称为参数，存储在其他寄存器中。在进行系统调用exit的情况下，操作系统需要将状态码加载到%ebx。稍后这个值被返回给系统，它就是你键入echo $?时提取的值。因此，我们通过键入以下指令将0加载到%ebx：

```
movl $0, %ebx
```

但将这些数字加载到寄存器本身不会做任何事。系统调用之外的各类事务也要用到寄存器，它们是执行加、减、比较等所有程序逻辑的地方。Linux只需要在系统调用前将某些参数值加载到某些寄存器。通常我们需要将系统调用号加载到%eax，而对于其他寄存器，每个系统调用有不同要求。就系统调用exit而言，它要求将退出状态加载到%ebx。我们将在需要时讨论不同的系统调用。关于通用系统调用列表及其对寄存器的要求，请参见附录C。

接下来的一条指令颇具“魔力”：

```
int $0x80
```

int代表中断，0x80是要用到的中断号。[①]中断会中断正常的程序流，把控制权从我们的程序转移到Linux，因此将进行一个系统调用。[②]你可以把中断看做向蝙蝠侠（当然，如果你喜欢，也可以是Larry-Boy[③]）发信号：你需要完成某件事，于是发出信号，然后“他”就现身救场。你不用关心他是如何完成工作的（这多少是一种魔法），当他完成后，控制权再度回到你手中。在本例中，我们所做的就是要求Linux终止程序，在这种情况下，我们不会再度获得控制权。而如果我们不发出中断信号，系统调用就不会被执行。

---

① 你可能会奇怪：为什么是0x80而不是80？这是因为数字采用了十六进制形式。在十六进制中，一位数可表示16个值，而不是通常情况下的10个值，除了常用数字0~9，这一数制表示还会用到字母a~f：a表示10，b表示11，以此类推。所以0x10代表十进制数字16，以此类推。稍后我们对此深入探讨，目前你只需知道以0x开头的是十六进制数。有时我们也会通过在数字后加H来表示十六进制数，但本书中不会这样做。想了解更多信息，请参阅第10章。

② 实际上中断将控制转移到为中断号设立中断处理的程序。在Linux下，所有这些都是由Linux内核来处理。

③ 如果你不看*Veggie Tales*（蔬菜宝贝），那不妨看看，并且可以先看一下“Dave and the Giant Pickle”这集。

快速回顾系统调用 重申一下，操作系统的功能是通过系统调用来访问的；这是通过以特殊方式设置寄存器并发出`int $0x80`指令调用的。Linux通过存储在`%eax`寄存器中的值获知我们要访问哪些系统调用。每个系统调用对于需要其他寄存器存储哪些值会有要求。系统调用号1是系统调用`exit`，要求将状态码放置在`%ebx`中。

既然你已经汇编、链接、运行并检测了程序，那么现在应该进行一些基本的编辑，比如更改加载到`%ebx`的数字，并观察`echo $?`的结果。不要忘了再次运行程序前要再次汇编和链接。你可以加入一些注释。别担心，最糟的事也不过是程序无法汇编或链接，或是屏幕无响应，但这也是学习的一部分！

3.3 为程序做规划

在下一个程序中，我们将尝试找到数字列表中的最大值。计算机非常注重细节，所以为了写程序，我们将对若干细节进行规划。

- 原数字列表将存储在哪里？
- 我们要按什么程序来找最大数？
- 要执行该程序，需要多少存储空间？
- 是将所有的存储数据都装入寄存器，还是另外使用一部分内存？

你可能认为在列表中找最大数这么简单的事不需要这么多规划。通常，让某人寻找最大数是一件可以轻松完成的事情。然而，我们的大脑能够自动整合复杂的任务，而计算机在整个过程中却都需要接收指示。此外，我们通常可以毫无困难地同时考虑很多事，而且通常不会意识到自己在这样做。例如，如果快速浏览列表寻找最大数，你可能会同时记住到某时刻为止看到的最大数和已在列表中浏览到的当前位置。你的大脑会自动执行这一切操作，但对于计算机，你必须明确设置列表当前位置及当前最大数的存储空间。你还要考虑何时停止等其他问题。当你查看一张纸上的数字时，如果看完了所有数字就可以停下了。但是，计算机只包含数字，而不知道何时到达了你的列表中的最后一个数字。

对于计算机，你必须计划每一步操作。所以，让我们做一些规划。首先，我们将列表起始地址命名为`data_items`，以便参考。假定列表中的最后一个数字是0，这样我们就知道在哪里停止。我们还需要保存列表中的当前位置、正在被检测的当前列表元素以及列表的当前最大值，为它们分配寄存器如下：

- `%edi`将保存列表的当前位置；
- `%ebx`将保存列表的当前最大值；

- %eax将保存当前正在被检测的元素。

刚开始运行程序，并查看该列表中的第一项时，因为我们尚未看到任何其他项，第一项将自动成为列表的当前最大值。此外，我们将设置列表的当前位置为0，即第一个元素的位置。然后，我们将遵循以下步骤。

(1) 检查列表当前元素（%eax）是否为0（终止元素）。

(2) 如果为0则退出。

(3) 递增当前位置（%edi）。

(4) 将列表中的下一个值加载到当前值寄存器（%eax）。这里可以使用哪种寻址方式？为什么？

(5) 比较当前值（%eax）与当前最大值（%ebx）。

(6) 如果当前值大于当前最大值，就以当前值替换当前最大值。

(7) 重复以上步骤。

这就是整个过程。在这个过程中，我多次用到“如果”这个词，这些地方都是进行决策的地方。你看，计算机不会每次都执行相同的指令序列。若是某些“如果”假设成立，电脑可能会执行不同的指令集。在第二轮比较时可能不会出现最大值。在这种情况下，它会跳过第(6)步，但会回到第(7)步。除非是最后一个元素，否则程序执行都会跳过第(2)步。在更复杂的程序中，跳过的步骤大大增加。

这些“如果”是一类*流控制*指令，因为它们告诉计算机要遵循的步骤和路径。在前一个程序中，我们没有任何流控制指令，因为只有一个可能的路径——退出。但本程序更具动态性，因为它是直接由数据决定的，会根据接收到数据的遵循不同的指令路径。

这个程序可通过两种指令——条件跳转和无条件跳转——来进行流控制。条件跳转指令根据最近一次比较或计算的结果改变路径，无条件跳转无需条件就直接进入不同的路径。无条件跳转看似无用，但实际上非常必要，因为所有的指令都被安排在一条线上，如果某条路径需要回到主路径，只有无条件跳转能做到这一点。我们将在下一节遇到更多条件跳转和无条件跳转。

流程控制的另一个作用是实现循环。一个循环是一段重复执行的程序代码。在我们的示例中，程序的第一部分（将当前位置设置为0，并将当前值加载到存放当前最大值的寄存器）只执行一次，因此不是一个循环。但接下来的部分是为列表中的每个数字重复同样的操作，仅当剩下最后一个元素0时停止，这就称为一个*循环*，因为它发生了一遍又一遍。这是在循环结束处通过无条件跳转转向循环开始处实现的，这里的无条件跳转会使循环重新开始。但是，你必须记住要有一

个退出循环的条件跳转，否则循环将永远继续下去！这种情况称为无限循环。如果我们不小心忽略了步骤(1)、(2)或(3)，循环（和我们的程序）将永远不会结束。

在下一节中，我们着手编制规划好的程序。程序规划不仅听起来很复杂，在一定程度上也确实如此。首度编程时，我们往往很难将正常思维过程转换成计算机可理解的过程。我们常常忘记大脑用来处理问题的“临时存储位置”的数量。然而，随着阅读和编写的程序越来越多，你会渐渐习惯这一点，你只需要持之以恒。

3.4 查找最大值

输入以下程序，程序名为maximum.s：

```
# 目的：本程序寻找一组数据项中的最大值
#
# 变量：寄存器有以下用途：
#
# %edi - 保存正在检测的数据项索引
# %ebx - 当前已经找到的最大数据项
# %eax - 当前数据项
#
# 使用以下内存位置：
#
# data_items - 包含数据项
#              0表示数据结束
#

 .section .data

data_items:                          # 这些是数据项
 .long 3,67,34,222,45,75,54,34,44,33,22,11,66,0

 .section .text

 .globl _start
_start:
 movl $0, %edi                  # 将0移入索引寄存器
 movl data_items(,%edi,4), %eax # 加载数据的第一个字节
 movl %eax, %ebx                # 由于这是第一项，%eax就是最大值

start_loop:                     # 开始循环
 cmpl $0, %eax                  # 检测是否到达数据末尾
 je loop_exit
```

```
    incl %edi                   # 加载下一个值
    movl data_items(,%edi,4), %eax
    cmpl %ebx, %eax             # 比较值
    jle start_loop              # 若新数据项不大于原最大值，
                                # 则跳到循环起始处
    movl %eax, %ebx             # 将新值移入最大值寄存器

    jmp start_loop              # 跳到循环起始处

loop_exit:
    # %ebx是系统调用exit的状态码
    # 已经存放了最大值
            movl $1, %eax           # 1是exit()系统调用
            int  $0x80
```

现在，以如下命令汇编并链接程序：

```
as maximum.s -o maximum.o
ld maximum.o -o maximum
```

现在运行程序并查看其状态：

```
./maximum
echo $?
```

你会发现程序返回值222。让我们来看看程序内容及其具体操作。如果查看一下注释，你会发现程序的目的是在一组数字中查找最大数。（注释果然很好很强大吧！）你可能还注意到，在这个程序中数据段是有实际内容的。下面几行就是数据段：

```
data_items:                             # 这些是数据项
        .long 3,67,34,222,45,75,54,34,44,33,22,11,66,0
```

让我们仔细看一下。data_items是一个指代其后位置的标签。接下来是一条指令，该指令以.long开始。这会让汇编程序为.long之后的数字列表保留内存。data_items是指第一个数字的位置。因为data_items是标签，在我们的程序中每当需要引用这个地址时，我们就可以使用data_items符号，而汇编程序将在汇编时以数字起始处的地址取代它。例如，指令movl data_items，%eax会将值3移入%eax。除了.long，还可保留几种类型的存储位置，主要类型如下。

- .byte

每个字节类型的数字占用一个存储位置，数字范围为0~255。

- .int

每个整型数字（这种类型与int指令不同）占用两个存储位置，数字范围为0~655 35。[①]

- .long

长整型占用4个存储位置，与寄存器使用的空间相同，这也是在本程序中使用它们的原因。长整型可表示0~4 294 967 295的数字。

- .ascii

.ascii指令用于将字符输入内存。每个字符占用一个存储位置（字符在内部转换成字节）。所以，如果你给出指令.ascii "Hello there\0"，汇编程序将保留12个存储位置（12字节）。第一个字节包含H的数字代码，第二个字节包含e的数字代码，以此类推。最后一个字符以\0表示，即终止字符（这个字符永远不会显示，只是告诉程序的其余部分：字符串到此结束）。用反斜杠开始的字母和数字表示用键盘无法输入或不易在屏幕上显示的字符。例如，\n指的是换行符，这会使计算机从下一行开始输出，而\t指的是制表符。所有.ascii指令中的字符都应该用引号括起来。

在我们的示例中，汇编程序保留14个.long型位置，依次连续排放。由于每个长整型要占用4字节，因此整个列表要占用56字节。这正是我们要搜索查找最大值的列表。汇编程序使用data_items来引用列表首个元素的地址。

注意，列表中的最后一个数据项是0。我决定用0来告诉程序：已经到达列表的最后。我也可以采取其他方式，比如可以在程序中手动编码列表的大小，可以将列表长度作为列表的首项或单独放置在某个位置，也可以自定义符号来标记列表的最后一个位置。无论怎么做，我必须采用一定的方法来确定列表结束之处。计算机对此一无所知，你告诉它做什么，它才会做什么。除非我给它某种信号，否则它不会停止，而是继续处理列表结束后存储位置的数据，甚至是我们未曾放置任何数据的内存位置。

注意，我们并没有声明data_items为.globl。这是因为我们只在本程序中引用这些位置，其他任何文件或程序都不需要知道它们的位置。_start符号与此相反：Linux需要知道它在哪里，这样才能知道程序从哪里开始执行。用.globl data_items也不算错，只是这样做没有必要。总之，研究一下这行代码，添加你自己的数字。即使它们是.long，如果任何数大于255，程序也将产生奇怪的结果，因为255是允许的最大退出码。另请注意，如果你把列表中的0向前移，那么0之后的部分将被忽略。请记住，每次改变源文件时，你都必须重新汇编和链接程序。现在请你这样做并查看结果吧。

① 注意，汇编语言（或我遇到过的其他任何计算机语言）中的数字都不含逗号。因此，数字书写形式为65535，而非65,535。

好了，我们已经对数据稍有了解。现在来看看代码。你会在注释中发现我们已经对一些计划要使用的变量进行标注。变量是用于某一特定目的存储位置，通常会由程序员起一个唯一的名字。我们在上一节中谈到了变量，但并没有给它们起名字。在这个程序中，我们有以下几个变量：

- 用于存放已找到的当前最大数的变量；
- 用于表明我们正在检测列表中哪个数字的变量，称为索引；
- 保存当前正在检测的数字的变量。

在本例中，我们的变量数较少，可以全部保存在寄存器中。而如果程序较大，你必须把它们放在内存中，在准备使用时再移入寄存器。稍后我们将讨论如何做到这一点。当人们开始编程，他们通常会低估所需的变量数。人们不习惯考虑一个过程的每个细节，因此往往在第一次尝试编程时遗漏必需的变量。

在这个程序中，我们使用%ebx存放已经找到的最大值，将%edi作为当前正在查看的数据项的*索引*。现在，让我们解释一下索引是什么。当从data_items读取信息时，我们将从第一项（数据项编号为0）开始，然后进入第二项（数据项编号为1），然后是第三项（数据项编号为2），以此类推。数据项编号就是data_items的*索引*。你会发现，我们给计算机的第一条指令是：

```
movl $0, %edi
```

由于将%edi用于存放索引，而我们要开始看第一个数据项，所以将0加载到%edi。现在，下一条指令较难理解，但对于实现我们的目的非常关键，这条指令如下所示：

```
movl data_items(,%edi,4), %eax
```

要理解这一行指令，你需要记住几件事：

- data_items是数字列表起始处的位置编号；
- 存储每个数字需要4个存储位置（因为我们声明数字为.long型）；
- 此刻%edi含有0。

因此，这一行代码的大致意思是：从data_items的起始位置开始，取第一项的数字（因为%edi为0），记住每个数据占据4个存储位置，并将该数字存储到%eax。这就是在汇编语言中使用索引寻址的方式。该指令的通用格式如下：

```
movl 起始地址(,%索引寄存器,字长)
```

在我们的示例中，data_items是起始地址，%edi是索引寄存器，4是字长。我们将在3.5节

中对此作进一步讨论。

如果查看一下data_items中的数字，你将发现当前%eax中的数字是3。如果将%edi设置为1，%eax为67，如果%edi为2，%eax的内容为34，以此类推。如果我们使用4以外的数字作为存储位置的大小，那么运行结果将无法预料。[①]这个程序编得比较粗略，但如果了解每一部分的功能，你会发觉并不太难。要了解更多信息，请参见3.5节。

我们再来看下一行：

```
movl %eax, %ebx
```

现在我们要查看的第一项存储在%eax中，由于这是第一项，这也就是当前为止的最大数，将之存储在用于保存当前最大数的%ebx中。此外，尽管movl代表move（移动），但它实际上会复制值，因此%eax和%ebx都包含起始值。[②]

现在，我们将进入循环。循环就是你的程序可能会多次运行的部分。我们已经以符号start_loop来标记循环的起始位置。要采用循环是因为我们不知道要处理多少数据项，但无论有多少项数据，处理过程都是相同的。我们不希望为各种不同的列表长度分别重写程序。事实上，我们甚至不希望为比较每个列表项而单独写代码。因此，我们有单个代码段（循环），为data_items中的每个元素反复执行。在前一节中，我们介绍了这个循环需要做些什么。回顾一下：

- 检查当前被检测的值是否为0，如果为0就意味着已经到达数据结束处，应退出循环；
- 必须加载列表中的下一个值；
- 必须检测下一个值是否大于当前最大值；
- 如果是的话，必须将其复制到保存当前最大值的存储位置；
- 现在必须回到循环的起始处。

好了，现在来看代码。我们把循环的起始处标记为start_loop，这样就知道当到达循环结束处时该返回何处。接着是以下指令：

```
cmpl $0, %eax
je end_loop
```

cmpl指令对这两个值进行比较。在这里，我们比较的是数字0和存储在%eax中的数字，比较

① 这条指令实际上并未真正把4当做存储位置的大小，尽管这样解释好理解。4其实是一个比例因子。实际上，其工作原理大致如下：从data_items指定的位置开始，然后递增%edi*4个存储位置，提取那里的数字。通常，我们将数字的长度作为比例因子，但在某些情况下不会这样做。

② 同样，movl中的l代表移动长整型，因为我们移动的是占据4个存储位置的值。

指令也会影响这里尚未提及的%eflags寄存器。%eflags也称为状态寄存器，有多种用途，我们将在稍后探讨。现在你只要注意，该比较结果存储在状态寄存器中。下一行是一个流控制指令，表示如果方才比较的值相等（je指令中的e代表“equal”，即“相等”），则跳转到end_loop的位置。该指令使用状态寄存器保存最近一次比较的结果。我们已经用过je，但此外还有许多可供使用的跳转语句，如下。

- je

若值相等则跳转。

- jg

若第二个值大于第一个值则跳转。[①]

- jge

若第二个值大于等于第一个值则跳转。

- jl

若第二个值小于第一个值则跳转。

- jle

若第二个值小于等于第一个值则跳转。

- jmp

无条件跳转。该指令无需跟在比较指令之后。

完整的代码清单记录在附录B中。在本例中，如果%eax中保存的值为0，我们就执行跳转，此时完成任务并进入loop_exit。[②]

如果最后加载的元素不是0，我们将执行下一条指令：

```
incl %edi
movl data_items(,%edi,4), %eax
```

① 注意，这里的比较是为了查看第二个值是否大于第一个值。其实这里也可以比较第一个值是否大于第二个值。学习编程就会遇到很多类似的事，每个人考虑问题的角度不一样嘛。不过，这不是重点，知道这回事就好了。

② 你可以任意命名这些符号，条件是它只包含字母和下划线（_）。唯一强制规定的是_start以及以.globl声明的其他符号。但是，如果是你定义且仅供你使用的符号，那么你可随意将其命名为能充分描述符号目的的名字。（记住，其他人将会修改你的代码，并必须搞清楚这些符号的含义。）

如前所述，%edi包含data_items中值列表的索引，incl将%edi的值递增1。接下来的movl与我们之前遇到的一条类似，但是因为已经递增了%edi，%eax会获取列表的下一项值。现在，%eax中是下一个将要检测的值。好，现在让我们来检测它！

```
cmpl %ebx, %eax
jle start_loop
```

在这里，我们将存储在%eax中的当前值与存放在%ebx中的当前最大值相比较。如果当前值小于或等于当前最大值，那么无需处理，只要跳转到循环起始处就好；否则，我们需要记录该当前值为最大值：

```
movl %eax, %ebx
jmp start_loop
```

上面两条指令将当前值移动到用来存储当前最大值的%ebx中，然后继续循环。

这样循环将继续执行直到到达0才跳转至loop_exit，这一部分程序调用Linux内核并退出。想一想上一个程序，当你调用操作系统（还记得吗？这就像给蝙蝠侠发信号）时，将系统调用号（1用于exit调用）存储在%eax中，并将其他值存储在其他寄存器中。退出调用需要我们将退出状态存储在%ebx中。由于我们将最大数置于%ebx中，该寄存器中已经有退出状态了，因此只要将数字1加载到%eax并调用内核退出即可，如下所示：

```
movl $1, %eax
int  0x80
```

好了，对于这样一个小程序，解释得那么详细已经有点啰唆了。但是，嘿，你已经学到不少知识啦！现在，再次通读整个程序，请特别注意那些注释。请确保你已经充分理解每行代码的意义。如果不理解某一行代码的话，请再重读这一节，弄懂该行代码的意思。

你也可以拿过一张纸，仔细阅读程序，记录单步执行程序时每个寄存器的变化，这样就能更清楚到底发生了什么。

3.5 寻址方式

在2.5节中，我们学习了汇编语言中的寻址方式。这一节将讨论如何在汇编语言指令中表示这些寻址方式。

内存地址引用的通用格式如下所示：

```
地址或偏移 (%基址寄存器,%索引寄存器,比例因子)
```

所有字段都可选。要计算地址，你只需要执行以下计算：

```
结果地址 = 地址或偏移 + %基址或偏移量寄存器 + 比例因子 * %索引寄存器
```

地址或偏移以及比例因子都必须是常量，其余两个必须是寄存器。如果省略任何一项，那么等式中将以0代替该项。

除了立即模式，所有在2.5节中提及的寻址方式都可用这种形式表示。

● 直接寻址方式

此模式通过使用地址或偏移部分实现。示例：

```
movl ADDRESS, %eax
```

以上指令将内存地址ADDRESS加载到%eax。

● 索引寻址方式

这种模式通过使用地址或偏移以及%索引寄存器部分实现。你可以将任何通用寄存器用作索引寄存器，也可以将索引寄存器的比例因子值常量定为1、2或4，使之更适合为字节、双字节和字进行索引。例如，我们有一个名为string_start的字节串，并想访问其中第三个字节（由于从0开始计数，索引为2），%ecx中保存值2。如果你想将其加载到%eax中，可通过如下指令实现：

```
movl string_start(,%ecx,1), %eax
```

该指令从string_start处开始，将该地址与1 * %ecx相加，并将所得值加载到%eax中。

● 间接寻址方式

间接寻址方式从寄存器指定的地址加载值。例如，如果%eax保存着一个地址，我们可通过以下操作将将该地址中的值移入%ebx：

```
movl (%eax), %ebx
```

● 基址寻址方式

基址寻址方式与间接寻址方式类似，不同之处在于它将一个常量值与寄存器中的地址相加。例如，如果你有一个记录，其中年龄值位于记录起始地址后4字节处，该记录的起始地址在%eax中，那么可以通过发出以下指令将年龄提取到%ebx中：

```
movl 4(%eax), %ebx
```

● 立即寻址方式

立即寻址方式非常简单。它与我们一直在使用的通用格式不同。立即寻址方式用于将直接值加载到寄存器或存储位置。例如，如果你想加载数字12到%eax，只需执行以下操作：

```
movl $12, %eax
```

注意，为了表明立即寻址方式，我们在数字前加上一个美元符号，若非如此，就会变成直接寻址方式；此时会将位于存储位置12中的值而不是数字12本身加载到%eax。

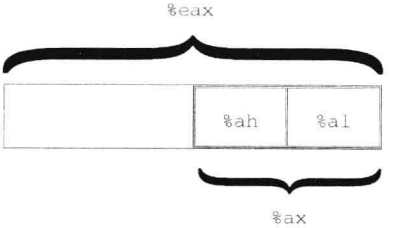
● 寄存器寻址方式

寄存器寻址方式仅仅是将数据移入或移出寄存器。在我们所有的例子中，寄存器寻址方式都用于其他操作数。

这些寻址方式非常重要，因为每次内存访问必将使用其中一种模式。立即寻址方式除外的所有模式都可用作源或目的操作数。立即寻址方式只能是源操作数。

除了这些模式，也存在不同的指令用于移动不同大小的值。例如，我们一直使用movl一次移动一个字。在许多情况下，你只想一次移动一字节的数据。这可通过指令movb来实现。然而，由于我们已经讨论过的寄存器是字大小，而不是字节大小，所以你不能使用完整的寄存器，而只能使用寄存器的一部分。

以%eax为例。如果你一次只想用两字节，那么可以只使用%ax。%ax是%eax寄存器最低有效字（即数值的后半部分），适合用于处理两字节的数值。%ax可进一步分成%al和%ah。%al是%ax寄存器的最低有效半字，%ah是最高有效半字。[①]加载值到%eax中将删除原先%al和%ah中的值（同时也将删除%ax的值，因为%ax是由前两者组成的）。同样，加载值到%al或%ah会破坏之前%eax中的值。基本上，明智的做法是只使用寄存器的一个字节或一个字，但不要同时使用两者。



%eax寄存器的布局

① 说到最高或最低有效字节，有人也许会感到迷惑。让我们以5432为例。在该数字中，54是最高有效部分，而32是最低有效部分。但对于寄存器，还不能完全这样划分，因为寄存器操作二进制数而非十进制数，但基本上是这个思想。有关此主题的更多信息，请参阅第10章。

更完整的指令列表参见附录B。

3.6 温故知新

3.6.1 理解概念

- 程序中的某一行以字符“#”开始意味着什么？
- 汇编语言文件和对象代码文件之间有何区别？
- 链接器有什么作用？
- 如何检测你最近运行的程序的结果状态码？
- 指令`movl $1, %eax`和`movl 1, %eax`有何区别？
- 哪一个寄存器保存系统调用号？
- 使用索引的目的是什么？
- 为什么索引通常从0开始？
- 如果我发出命令`movl data_items(,%edi,4), %eax`，`data_items`为地址`3634`，`%edi`保存值`13`，那么你应该把什么地址移入`%eax`？
- 请列出通用寄存器。
- `movl`和`movb`的区别是什么？
- 什么是流控制？
- 条件跳转的作用是什么？
- 编写程序时你应计划好哪些事？
- 细读每条指令，列出用于每个操作数的寻址方式。

3.6.2 应用概念

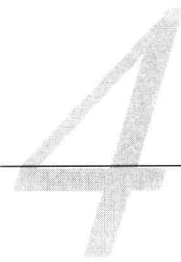
- 更改第一个程序使之返回值`3`。
- 更改最大值程序使之返回最小值。
- 更改最大值程序，使之以数字`255`取代`0`作为循环终止条件。
- 更改最大值程序，使用列表终止处地址而非`0`作为循环终止条件。
- 更改最大值程序，使用长度计数而非`0`作为循环停止条件。
- 指令`movl _start, %eax`的作用是什么？请根据你对寻址方式和`_start`含义的了解具体阐述。这条指令与`movl $_start, %eax`?的区别是什么？

3.6.3 深入学习

- 更改第一个程序，省略int指令行。汇编、链接和执行新程序。你得到的错误信息是什么？你认为出错的原因是什么？
- 到现在为止，我们讨论了确定列表结束的三种方式——使用一个特定数字、使用结束地址和使用长度计数。你觉得哪种方式最好？为什么？如果知道列表是经过排序的，那么你会采取哪种方式？为什么？

第4章

关于函数



4.1 处理复杂度

在第3章中，我们编写的程序仅包含一个代码段。但是，如果实际编写程序时也这么做，那么程序将无法维护，很难让多人合作开发和维护该项目，因为某个部分的任何改变都可能对另一个开发人员正在开发的另一部分产生不利影响。

为了有助于程序员以小组为单位分工协作，我们必须将程序划分成相互独立的部分，每一部分都能通过定义良好的接口与任一其他部分通信。这样一来，每一部分都能独立开发测试，方便程序员合作完成项目。

程序员使用*函数*将程序划分为可独立开发和测试的各部分。函数是在指定类型的数据上完成所定义的某个工作的代码段。例如，在文字处理程序中，我可以有一个名为handle_typed_character的函数，每当用户输入一个关键字时就激活。函数使用的数据很可能是按键本身和用户当前打开的文档。该函数根据其获知的按键信息来修改文件。

给予函数处理的数据项称为函数的*参数*。在文字处理程序的示例中，按下的键以及当前文档将被视为handle_typed_characters函数的参数。函数的参数列表以及处理预期结果（预期对参数作何处理）称为函数的接口。在设计函数接口时必须十分小心，因为如果要在项目中的很多地方都调用接口，那么对函数进行更改将非常困难。

一个典型的程序由数百或数千个函数构成，每个函数都执行一个定义明确的小任务。但最终还是有些事情你无法编写程序完成，它们必须由系统提供。这些由系统提供的函数称为*原函数*（或*原语*）——因为其他一切都是建立在这些函数之上的。例如，可以设想一个绘制图形化用户界面的程序：必须有一个函数来创建菜单。该函数可能会调用其他函数写入文字或图标、描绘背景、计算鼠标指针的位置等。但是，最终它们将用到一组由操作系统提供的原语，以绘制基本的线或点。我们可以这样看待编程，即将大型程序划分为较小的程序，直到到达原函数，或是在原语的

基础上逐渐构建函数，直到达成最终目标。在汇编语言中，原语通常就是系统调用，尽管系统调用并非本章中我们将讨论的真正函数。

4.2 函数的工作原理

函数由如下几个不同部分组成。

- *函数名*

函数名是一个符号，代表该函数代码的起始地址。在汇编语言中，符号是通过在函数代码前输入函数名作为标签来定义的，就像你之前用于跳转的标签。

- *函数参数*

函数参数是显示给函数以进行处理的数据项。例如，对于数学上的正弦函数，如果你让计算机寻找sin2的值，sine将是函数的名字，2将是参数。某些函数有很多参数，有些则不含参数。[①]

- *局部变量*

局部变量是函数在进行处理时使用的数据存储区，在函数返回时即被废弃。这有点像便笺纸。函数每次被激活时就获得一张新纸，当完成处理时即将其丢弃。对于某个函数的局部变量，程序中的任何其他函数都无法访问。

- *静态变量*

静态变量也是函数进行处理时用到的数据存储区，但使用后不会被废弃，每当函数代码被激活时都重复使用。程序的任何其他部分都无法访问此数据。除非绝对必要，否则我们一般不使用静态变量，因为它们可能在将来引起各种问题。

- *全局变量*

全局变量是函数进行处理时用到的、在函数之外管理的数据存储区。例如，一个简单的文本编辑器可能把当前正在操作的文件整体放入一个全局变量，这样就无需传递该文件给每个对其进行操作的函数。[②]配置的值通常也存储在全局变量中。

---

① 函数参数也可用于存放指针，指针指向函数要传回程序的数据。

② 这通常被视为不良习惯。设想，假如以这种方式编写程序，而在下一个版本中，大家又决定让程序的一个实例能编辑多个文件。那么我们就不得不更改每个函数，然后才能将当前操作的文件作为参数传递。如果一开始就将文件作为参数，那么当你更新程序时多数函数就无需更改。

- 返回地址

返回地址是一个“看不见”的参数，因为它不能直接在函数中使用。返回地址这一参数告诉函数当其执行完毕后应该再从哪里开始执行。返回地址必不可少，因为程序中许多不同的部分都会调用函数进行处理，因此函数必须能够返回到调用它的地方。在大多数编程语言中，调用函数时会自动传递这个参数。而在汇编语言中，call指令会为你处理返回地址，ret指令则负责按照该地址返回到调用函数的地方。

- 返回值

返回值是传回数据到主程序的主要方法。大多数编程语言只允许一个函数有一个返回值。

以上这些部分在大多数编程语言中都存在。然而在每种语言中如何指定每一项却各不相同。

在不同的语言中，变量存储以及计算机传输参数和返回值的方式各不相同。这种差异称为语言的调用约定，因为它描述了在调用函数时，函数预期得到什么样的数据。[①]

汇编语言能使用其偏好的任何调用约定。你甚至可以自己定一个调用约定。但如果你想与其他语言编写的函数进行互动，就必须服从其他语言的调用约定。我们将在示例中使用C编程语言的调用约定，因为这种约定使用最为广泛，也因为它是Linux平台的标准。

4.3　使用 C 调用约定的汇编语言函数

如果不了解计算机栈的工作原理，你就无法编写汇编语言函数。每个运行的计算机程序都使用称为栈的内存区来使函数正常工作。我们可以将栈设想为办公桌上的一堆文件，它可以无限增加。我们通常会把工作时用到的文件放在顶部，而当用完后就把它拿走。

计算机也有一个栈。计算机的栈处于内存地址的最顶端。你可以通过pushl指令将值压入栈顶，该指令将一个寄存器值或内存值压入栈顶。好了，我们说顶，但栈的“顶”实际上是栈内存的底部。这点会令人困惑，因为对于想到的任何一堆东西——盘子、纸等——我们都会想要从顶部添加和删除。然而，在内存中，由于考虑到内存结构，栈是从内存顶部开始向下增长的。因此，当我们提到“栈顶”，请记住这是在栈内存的底部。你也可以用指令popl将值从栈顶弹出。该指令将值从栈顶移除，并把其放入寄存器或你选择的存储位置。

① 约定是做事情的一种标准化方式，但并非必须如此。例如，一般惯例是人们见面时握手。如果我拒绝与你握手，你可能会认为我不喜欢你。遵循约定很重要，因为这样做更容易使别人了解你在做什么，也易于多个独立编程者编写的程序协同工作。

当我们将值入栈时，栈顶会移动，以容纳新增加的值。实际上，我们能不断将值入栈，栈会在内存中保持向下增长，直到达到存放代码或数据的地方。那么，我们如何知道栈当前的顶部在哪里呢？栈寄存器%esp总是包含一个指向当前栈顶的指针，无论栈顶在何处。

每当我们用pushl将数据入栈，%esp所含的指针值都会减去4，从而指向新的栈顶。（记住，每一个字的长度为4字节，并且栈是向下生长的。）如果我们想从栈中删除数据，只需使用popl指令，该指令使%esp的值增加4，并将先前栈顶的值放入你指定的寄存器。pushl和popl都有一个操作数：对于pushl，是要将其值入栈的寄存器；对于popl，是要接收弹出栈数据的寄存器。

如果我们只是想访问栈顶的值，而不想移除该值，在间接寻址方式中使用%esp即可。例如，以下代码将栈顶的内容移入%eax:

```
movl (%esp), %eax
```

如果我们只用以下指令：

```
movl %esp, %eax
```

那么%eax会保存指向栈顶的指针，而不是栈顶存放的值。将%esp置于括号中会使计算机遵循间接寻址方式，因此我们就能获得%esp所含指针指向的值。如果想访问栈顶的下一个值，我们只需发出指令：

```
movl 4(%esp), %eax
```

这条指令使用基址寻址方式（参见2.5节），在寻找指针指向的值之前将%esp与4相加。

在C语言调用约定中，栈是实现函数的局部变量、参数和返回地址的关键因素。

在执行函数之前，一个程序将函数的所有参数按逆序压入栈中。接着，程序发出一条call指令，表明程序希望开始执行哪一个函数。call指令会做两件事情：首先，它将下一条指令的地址即返回地址压入栈中；然后，修改指令指针（%eip）以指向函数起始处。因此，在函数开始执行时，栈看起来如下所示（在本例中，栈的“顶部”在底部）：

```
参数 #N
...
参数 2
参数 1
返回地址 <--- (%esp)
```

函数的每个参数都已被压入栈中，最后入栈的是返回地址。现在函数本身也有一些工作要做。

首先，函数通过pushl %ebp指令保存当前的基址指针寄存器%ebp。基址指针是一个特殊的寄存器，用于访问函数的参数和局部变量。其次，它会用movl %esp, %ebp.将栈指针%ebp复制到%esp，这使你能够把函数参数作为相对于基址指针的固定索引进行访问。你也许认为可以使用栈指针来访问函数参数，但在程序中，你可能会对栈执行其他操作，如压入其他函数的参数等。

在函数开始时将栈指针复制到基址指针寄存器可以让你一直清楚参数的位置（你将会发现局部变量也是如此），即使在将其他数据压入弹出栈的情况下也是如此。%ebp将一直是栈指针在函数开始时的位置，所以可以说是对栈帧的常量引用。（栈帧包含一个函数中使用的所有栈变量，包括参数、局部变量和返回地址。）

此时，栈看起来如下所示：

```
参数 #N    <--- N*4+4(%ebp)
...
参数 2     <--- 12(%ebp)
参数 1     <--- 8(%ebp)
返回地址   <--- 4(%ebp)
旧%ebp     <--- (%esp) 和 (%ebp)
```

可以看到，每个参数都可以用%ebp通过基址寻址方式访问。

接下来，函数为其所需的所有局部变量保留栈空间，只需将栈指针向外移动即可实现。假设要运行函数，我们需要两个字的内存，只需将栈指针向下移动两个字即可预留空间。这是通过如下指令实现的：

```
subl $8, %esp
```

该指令将%esp减去8（请记住一个字的长度为4字节）。[①]这样，我们就能将栈用于变量存储，而无需担心函数调用引起的入栈会破坏存储的变量。因为函数调用是在栈帧上分配的，而变量仅仅在函数运行期间有效，所以当函数返回时，栈帧就不复存在，这些变量也就不复存在。这就是它们被称为局部变量的原因——它们仅在函数被调用时存在。

现在我们有两个字可用于本地存储。栈现在看起来如下所示：

```
参数 #N    <--- N*4+4(%ebp)
...
参数 2     <--- 12(%ebp)
参数 1     <--- 8(%ebp)
返回地址   <--- 4(%ebp)
旧%ebp     <--- (%ebp)
```

① 提醒一下，数字8前面的美元符号表示立即寻址方式，表示我们加载8到%esp，而不是地址8中的值。

```
局部变量 1  <--- -4(%ebp)
局部变量 2  <--- -8(%ebp) 和 (%esp)
```

所以我们现在可以使用%ebp中的不同偏移量，通过基址寻址访问这个函数所需的所有数据。%ebp正是专门为这一目的设计的，这就是它被称为基址指针的原因。你可以在基址指针寻址方式中使用其他寄存器，但在x86架构中，使用%ebp寄存器速度会快得多。

访问全局变量和静态变量就像我们在前面的章节中访问内存一样。全局变量和静态变量之间的唯一区别是静态变量只用于一个函数，而全局变量可由许多函数共同使用。虽然汇编语言对它们一视同仁，但大多数其他语言会将两者区分开来。

当一个函数执行完毕后，函数会做三件事。

(1) 将其返回值存储到%eax。

(2) 将栈恢复到调用函数时的状态（移除当前栈帧，并使调用代码的栈帧重新生效）。

(3) 将控制权交还给调用它的程序。这是通过ret指令实现的，该指令将栈顶的值弹出，并将指令指针寄存器%eip设置为该弹出值。

所以，在函数将控制权返回给调用它的代码时，必须恢复前一个栈帧。也要注意，如果不这样做，ret将无法正常工作，因为我们当前栈帧的返回地址不在栈顶。因此，返回前必须将栈指针%esp和基址指针%ebp重新设置为函数开始时的值。

因此，要从函数返回，你必须使用如下指令：

```
movl %ebp, %esp
popl %ebp
ret
```

此刻，你应该将所有局部变量视为废弃。其原因是，在你移动栈指针后入栈的数据很可能覆盖之前存放在那里的内容。因此，你不应该在创建局部变量的函数的生命期之后，仍保留该局部变量的地址，否则在其栈帧的生命期结束后该地址会被覆盖。

现在控制权已经转回调用代码那里，调用代码现在可以检查%eax中的返回值。调用代码也需要弹出其入栈的所有参数，以将栈指针复位至其原先的位置（如果不再需要参数值，那你用addl指令将“4×参数个数”加到%esp即可）。[①]

① 无需一直严格遵循这点，除非你在函数调用前将寄存器保存到栈中。基址指针记录栈帧，合理地保持其一致状态。不过，这仍然是一个值得提倡的做法，如果你暂时将寄存器内容保存到栈上的话，那么这样做是绝对必要的。

寄存器内容破坏

当你调用函数时，应该假设目前寄存器中的一切内容都会被去除。可以确保函数开始后其值仍然被保留的唯一一个寄存器就是%ebp。%eax肯定会被覆盖，而其他寄存器则可能被覆盖。如果你要在调用函数前保存寄存器的内容，必须在函数参数入栈前将其入栈。然后，你可以在弹出参数后以逆序将其弹出存回寄存器。即使你知道函数不会覆盖某个寄存器，也应该保存它，因为该函数将来的版本可能会覆盖该寄存器。

其他语言的调用约定可能有所不同。例如，其他调用约定可能让函数承担保存其使用的寄存器的任务。请务必进行检查，在尝试混合使用多种语言前确保各语言的调用约定是兼容的。在汇编语言的情况下，要确保你知道如何调用其他语言的函数。

扩展规范　C语言的调用约定（也称为ABI，即应用程序二进制接口）的详细信息可在网上查到。我们为了便于新程序员理解，过分简化了规范，省略了几个重要部分。完整的详细信息，请查阅http://www.linuxbase.org/spec/refspecs/提供的文件。更具体地说，你应该查找*System V Application Binary Interface - Intel386 Architecture Processor Supplement*。

4.4　函数示例

让我们来看看在实际程序中函数是如何工作的。这里将要编写的是power函数，为计算某个数字的幂，我们给它两个参数——该数字及其指数。例如，如果我们设定参数2和3，那么函数将计算2的3次方，即2×2×2，结果为8。为了简化程序，我们只允许输入大于等于1的数字。

以下就是完整的程序，按照惯例，程序中加上了注释。文件名为power.s。

```
# 目的:  展示函数如何工作的程序
#        本程序将计算
#        2^3 + 5^2
#

# 主程序中的所有内容都存储在寄存器中，
# 因此数据段不含任何内容
.section .data

.section .text

.globl _start
_start:
 pushl $3                          # 压入第二个参数
 pushl $2                          # 压入第一个参数
```

```
 call power                          # 调用函数
 addl $8, %esp                       # 将栈指针向后移动
 pushl %eax                          # 在调用下一个函数之前
                                     # 保存第一个答案

 pushl $2                            # 压入第二个参数
 pushl $5                            # 压入第一个参数
 call power                          # 调用函数
 addl $8, %esp                       # 将栈指针向后移动

 popl %ebx                           # 第二个答案已经在
                                     # %eax中。 我们之前已经将第一个
                                     # 答案存储到栈中，
                                     # 所以现在可以将其
                                     # 弹出到%ebx

 addl %eax, %ebx                     # 将两者相加
                                     # 结果在%ebx中

 movl $1, %eax                       # 退出 (返回%ebx)
 int  $0x80

 # 目的:    本函数用于计算
 #          一个数的幂
 #
 # 输入:    第一个参数 - 底数
 #          第二个参数 - 底数的指数
 #
 # 输出:    以返回值的形式给出结果
 #
 # 注意:    指数必须大于等于1
 #
 # 变量:
 #        %ebx - 保存底数
 #        %ecx - 保存指数
 #
 #             -4(%ebp) - 保存当前结果
 #
 #              %eax用于暂时存储
 #
 .type power, @function
power:
 pushl %ebp                          # 保留旧基址指针
 movl %esp, %ebp                     # 将基址指针设为栈指针
 subl $4, %esp                       # 为本地存储保留空间
```

```
 movl 8(%ebp), %ebx                    # 将第一个参数放入%eax
 movl 12(%ebp), %ecx                   # 将第二个参数放入%ecx

 movl %ebx, -4(%ebp)                   # 存储当前结果

power_loop_start:
 cmpl $1, %ecx                         # 如果是1次方，那么我们已经获得结果
 je   end_power
 movl -4(%ebp), %eax                   # 将当前结果移入%eax
 imull %ebx, %eax                      # 将当前结果与底数相乘
 movl %eax, -4(%ebp)                   # 保存当前结果

 decl %ecx                             # 指数递减
 jmp  power_loop_start                 # 为递减后的指数进行幂计算

end_power:
 movl -4(%ebp), %eax                   # 返回值移入%eax
 movl %ebp, %esp                       # 恢复栈指针
 popl %ebp                             # 恢复基址指针
 ret
```

输入、汇编并运行该程序。尝试以不同的指数调用power函数，但请记住传递回操作系统的结果必须小于256。你也可以尝试将两个计算结果相减，尝试加入对power函数的第三次调用，并将其与前两个的差相加。

主要的程序代码颇为简单。你将参数入栈，调用函数，然后将栈指针向后方移动。结果存储在%eax中。注意，在两次power调用之间，我们将第一个值存入栈。这是因为唯一确定会被存入栈的只有%ebp寄存器。因此我们将值入栈，并在第二次函数调用完成后将值弹出。

来看看函数本身是如何编写的。注意在函数之前有关于函数功能、参数和返回值的文档，这对于使用该函数的程序员很有帮助。这就是函数的接口，让程序员了解栈上需要存放哪些值，最终在%eax中的值是什么。

接下来我们输入以下指令：

```
.type power,@function
```

这条指令告诉链接器应将符号power作为函数处理，因为这个程序只在一个文件中，即使省略这一步也同样奏效。但这是一种比较好的做法。

此后，我们定义power的标签值：

```
power:
```

如前所述，这样会将符号power定义为紧随标签之后那条指令的起始地址。这就是call power的工作原理，将控制权转移到程序的这个点，call和jmp的区别在于call还会将返回地址入栈以便函数能够返回，而jmp则不会这样做。

接下来，我们为调用函数进行设置：

```
pushl   %ebp
movl    %esp, %ebp
subl    $4, %esp
```

此刻，栈如下所示：

```
底数        <--- 12(%ebp)
指数        <--- 8(%ebp)
返回值      <--- 4(%ebp)
旧%ebp      <--- (%ebp)
当前结果    <--- -4(%ebp) 和 (%esp)
```

尽管我们能将寄存器用作临时存储，但本程序使用局部变量以展示如何为调用函数进行设置。通常不会有足够的寄存器来保存所有内容，你必须将部分内容卸载到局部变量中。有时，你的函数将需要调用另一个函数并将你部分数据的指针传递给它。不存在指向寄存器的指针，因此你必须将数据存储在局部变量中，这样才能传递指向此数据的指针。

基本上来说，程序所做的就是从基数开始，将之存为乘数（存入%ebx）和当前值（存入-4(%ebp)）。此外，它还将指数存入%ecx，然后继续将当前值与乘数相乘，递减指数，并在指数（在%ecx中）减少到1时跳出循环。

现在，你应该能够在不寻求帮助的情况下读懂程序了。你唯一需要知道的是imull指令的功能是执行整数乘法，并将结果存到第二个操作数中，而decl将给定寄存器的值减1。要了解上述及其他指令的更多信息，请参见附录B。

现在，另一个值得一试的项目就是扩展本程序，这样程序就能在指数为0时返回值。（提示，任何数的0次方都是1。）请继续尝试。如果程序开始不能获得成功，就请尝试手动检查你的程序，在纸上记录%ebp和%esp指向的值、栈以及每个寄存器中的内容。

4.5 递归函数

下一个程序会进一步拓展你的思维。该程序将计算一个数的阶乘。阶乘是某个数字与1之间的所有整数的乘积。例如，阶乘7是7×6×5×4×3×2×1，4的阶乘是4×3×2×1。现在，你可能发现一

个数的阶乘就是该数与比它小1的数的阶乘相乘。例如，4的阶乘是4乘以3的阶乘，3的阶乘是3乘以2的阶乘，2的阶乘是2乘以1的阶乘，1的阶乘为1。这种类型的定义称为递归定义。这意味着阶乘函数的定义中包括阶乘函数本身。然而，由于所有函数都必须结束，一个递归定义必须包括一个基线条件。基线条件就是递归停止的地方。如果没有基线条件，该函数将不断调用自身，直到最终用尽栈空间。在这个阶乘的示例中，基线条件是数字1。当我们遇到数字1，就再调用阶乘；刚才说过阶乘1的值为1。所以，让我们再看一下希望阶乘函数的代码是什么样的。

(1) 检查数字。

(2) 数字是否为1?

(3) 如果数字为1，答案也为1。

(4) 否则，数字与该数字减去1的阶乘相乘。

如果没有局部变量，这样做可能会出问题。在其他程序中，用全局变量存储值运作良好。然而，全局变量只为每个变量提供一个副本。而在这个程序中，在同一时间我们将有函数的多个副本在运行，每个副本都需要自己的数据副本！[①]由于局部变量存在于栈帧上，每个函数调用都有自己的栈帧，所以我们不会出问题。

让我们来看看代码是如何工作的：

```
# 目的 - 给定一个数字，本程序将计算其阶乘
#        例如，3的阶乘是3×2×1，即6
#        4的阶乘是4×3×2×1，即24，以此类推
#

# 本程序展示了如何递归调用一个函数

.section .data

# 本程序无全局数据

.section .text

.globl _start
.globl factorial  # 除非我们希望与其他程序共享该函数，
                  # 否则无需此项
_start:
 pushl $4         # 阶乘有一个参数，就是我想要
```

① 说到“同时运行”，我指的是一个函数不会在新函数被激活前完成，而不是暗示函数指令同时运行。

```
                    # 为其计算阶乘的数字。因此，该数字将入栈
 call factorial     # 运行阶乘函数
 addl $4, %esp      # 弹出入栈的参数
 movl %eax, %ebx    # 阶乘将答案返回到%eax，
                    # 但我们希望它在%ebx中，
                    # 这样可将之作为我们的退出状态
 movl $1, %eax      # 调用内核退出函数
 int $0x80

 # 这是实际的函数定义
 .type factorial,@function
factorial:
 pushl %ebp         # 标准函数 - 我们必须在返回前
                    # 恢复%ebp到其之前的状态，
                    # 因此我们必须将其入栈
 movl %esp, %ebp    # 这条指令是因为我们不想更改
                    # 栈指针，所以使用%ebp

 movl 8(%ebp), %eax # 这条指令将第一个参数移入%eax
                    # 4(%ebp)保存返回地址，而
                    # 8(%ebp)保存第一个参数
 cmpl $1, %eax      # 如果数字为1，这就是我们的基线条件
                    # 我们只要返回即可（1
                    # 已经作为返回值在%eax中）
 je end_factorial
 decl %eax          # 否则，递减值
 pushl %eax         # 为了调用factorial函数将其入栈
 call factorial     # 调用factorial函数
 movl 8(%ebp), %ebx # %eax中为返回值，因此我们
                    # 将参数重新加载至%ebx
 imull %ebx, %eax   # 将之与上一次调用
                    # factorial的结果（在%eax中）相乘
                    # 答案将存入%eax，而%eax正是
                    # 存放返回值的地方
end_factorial:
 movl %ebp, %esp    # 标准函数返回相关处理 - 我们
 popl %ebp          # 必须将%ebp和%esp恢复到
                    # 函数开始运行前的状态
 ret                # 返回到函数（这也会将返回值
                    # 弹出栈）
```

通过以下命令汇编、链接并运行程序：

```
as factorial.s -o factorial.o
ld factorial.o -o factorial
```

```
./factorial
echo $?
```

这会让你得到值24。24是4的阶乘，你可以用计算器来验证这一结果：4×3×2×1 = 24。

我猜你并不理解整个代码清单。让我们一次一行地研究一下发生了什么。

```
_start:
 pushl $4
 call factorial
```

可以看到，这个程序用于计算4的阶乘。当编写函数时，假设你在调用函数前将函数的参数放置在栈顶。记住，函数的参数是你希望函数用到的数据。在本例中，阶乘函数有1个参数——你想要为之计算阶乘的值。

pushl指令将给定的值置于栈顶，接着call指令调用函数。

接下来我们有以下指令：

```
addl $4, %esp
movl %eax, %ebx
movl $1, %eax
int $0x80
```

这些是在factorial运行完成，并已经计算出4的阶乘值之后发生的。现在我们已经清空栈。addl指令将栈指针移回我们将$4入栈之前的位置。每次在函数调用返回后都应清空栈。

下一条指令将%eax的值移入%ebx中。%eax中是什么内容？那是factorial的返回值。在我们的示例中，它就是阶乘函数的值。由于参数是4，返回值就应该是24。请记住，返回值始终存储在%eax中。我们希望将这个值作为状态码返回给操作系统。然而，Linux要求将退出码保存在%ebx，而不是%eax中，所以我们必须移动它。接着进行标准的退出系统调用。

函数调用的好处在于：

- 除了使用函数必需的参数，其他程序员无需参数的其他信息；
- 它们为编程提供了标准化构件；
- 它们可以从多个位置多次调用，由于call指令将返回地址入栈，函数总是知道如何返回到调用处。

以上是函数的主要优点。较大的程序也可以用函数将复杂的代码段分解成更小的、更简单的代码段。事实上，几乎所有的编程工作都是编写和调用函数。

现在，让我们来看看factorial函数本身是如何实现的。

在函数开始之前，我们使用这条指令：

```
 .type factorial,@function
factorial:
```

.type指令告诉链接器factorial是一个函数。除非我们需要在其他程序中使用factorial函数，否则这条指令不是必需的。我们包含此指令是为了保证完整性。factorial:将下一条指令的存储位置赋予符号factorial。这就是为什么call指令在我们发出call factorial指令时知道到哪里去调用。

函数第一段真正的指令是：

```
pushl %ebp
movl  %esp, %ebp
```

正如在先前的程序中所示，这些指令将创建该函数的栈帧。你应该在每个函数的开头都包含这两行代码。

下一条指令是：

```
movl  8(%ebp), %eax
```

该指令使用基址指针寻址将函数的第一个参数移入%eax。请记住，(%ebp)含有旧的%ebp，4(%ebp)含有返回地址，8(%ebp)是函数第一个参数的位置。你回想一下就会发现，第一次调用时参数值为4，因为这就是我们在调用函数前用pushl $4入栈的数据。当这个函数递归调用时，入栈的会是其他值。

接下来，我们要检查一下，看看是否达到基线条件（参数1）。如果达到基线条件，我们就跳转到标签end_factorial，并在那里返回（正如前面提到的，你已经将返回值放入%eax了）。这是通过以下几行代码实现的：

```
cmpl $1, %eax
je end_factorial
```

如果不是基线条件，记得我们说会怎么做吗？我们将再次调用factorial函数，参数为原参数减1。所以，首先将%eax的内容减1：

```
decl %eax
```

decl代表递减，将给定寄存器或内存位置（在此情况下是%eax）的内容减去1。incl则正好相反，它会加1。在递减%eax后，我们将其入栈，因为这将是下一次函数调用的参数。然后再次调用factorial！

```
pushl %eax
call factorial
```

好了，现在我们已经调用了factorial。还有一件事要记住，那就是函数调用后，我们永远不知道寄存器的内容是什么（%esp和%ebp除外）。因此，即使调用所得的值曾经在%eax中，现在它也已经不在%eax中了。因此，我们需要把它从首次入栈的地址（8(%ebp)）弹出。所以，我们有采用以下指令：

```
movl 8(%ebp), %ebx
```

现在，我们希望用该数与阶乘函数的结果相乘。如果你还记得前面的讨论，应该知道函数的结果保留在%eax中。因此，我们需要把%ebx与%eax相乘，这通过以下指令实现：

```
imull %ebx, %eax
```

结果也存储在%eax中，这也正是我们想要的函数返回值！由于返回值正好在该在的地方，我们只需离开函数即可。如果你还记得，在函数开始时，我们将%ebp入栈，将%esp移入%ebp以创建当前栈帧。现在，我们要进行相反的操作来清除当前栈帧，并重新激活上一个栈帧：

```
end_factorial:
 movl %ebp, %esp
 popl %ebp
```

现在，我们已经做好返回准备，所以发出以下命令：

```
ret
```

该命令将栈顶的值弹出，然后跳转到该地址。如果你还记得关于call的讨论，应该还记得我们说过call指令在跳转到函数的开始处之前，首先将下一条指令的地址入栈。所以，在这里将之弹回原处，这样就可以回到下一条指令。函数完成后，我们就得到了答案！

如同之前的程序一样，你应该再次阅读程序，并确保自己知道每一条指令的作用。如果有任何不明白之处，请仔细阅读本节以及前几节的解释。接着，拿一张纸，请记录程序每一条指令后各寄存器和栈上的值。这样做应该能加深理解，让你明白程序执行过程中到底发生了什么。

4.6　温故知新

4.6.1　理解概念

- 什么是原语？

- 什么是调用约定?
- 什么是栈?
- pushl和popl会对栈产生什么影响?它们又会对专用寄存器产生什么影响?
- 什么是局部变量?它们的用途是什么?
- 为什么局部变量在递归函数中必不可少?
- %ebp和%esp的用途是什么?
- 什么是栈帧?

4.6.2 应用概念

- 编写一个名为square的函数,使它接收一个参数,并返回该参数的平方值。
- 编写一个程序来测试你的square函数。
- 将3.4节中的最大值程序转换为函数,参数为指向值列表的一个指针,并且函数返回其中最大值。编写一个程序,以三个不同的列表调用最大值函数,并返回最后一个结果作为程序的退出状态码。
- 说明不遵循标准调用约定会产生的问题。

4.6.3 深入学习

- 假设较大的原语集可以用较小的原语集编写构成,你认为系统有较大的原语集好,还是有较小的原语集好?
- 阶乘函数可以写成非递归函数。请以非递归函数的形式编写阶乘函数。
- 找出一个你经常使用的计算机应用程序。尝试找到一个特定功能,并将该功能分解为多个函数。定义函数与程序其余部分之间的接口。
- 编写自己的调用约定,使用该调用约定重写本章中的程序。使用不同调用约定的一个示例就是用寄存器取代栈进行参数传递,以不同的顺序传递参数,将返回值存储到其他寄存器或存储位置。无论你怎样选择,请将其应用到整个程序中,并保持前后一致。
- 你能编写一个不使用栈的调用约定吗?它可能会有什么限制?
- 为了检测示例程序是否运行正确,我们应该使用什么测试用例?

第5章

文件处理

5

许多计算机编程都涉及处理文件的任务。毕竟，当我们重新启动计算机后，唯一能从之前的会话中保留下来的就是磁盘上的数据。存储在文件中的数据称为*永久数据*，因为即使程序没有运行，数据仍保留在磁盘上的文件里。

5.1 UNIX文件的概念

每个操作系统都有自己的文件处理方式。然而，Linux中用的是UNIX的处理方式，也是最简单最普遍的处理方式。无论UNIX文件是什么程序创建的，都可以作为连续的字节流进行访问。当你访问一个文件时，通过文件名打开它，操作系统会给你一个编号，称为*文件描述符*，你用它来指代该文件，直到使用完毕。接下来，你就可以使用文件描述符对该文件进行读取和写入。完成读取和写入后，关闭文件，关闭后文件描述符即失效。

在我们的程序中，文件的以下几个方面需要注意。

(1) 告诉Linux要打开文件的文件名，并以特定模式打开（读取、写入、读写、如不存在则创建等）。这是通过`open`系统调用处理的，其所需参数为一个文件名、一个表示模式的数字以及一个权限集合。`%eax`将保存系统调用号——5。文件名第一个字符的地址应存放在`%ebx`中。以数字表示的读/写意图应存放在`%ecx`中。现在，以0表示你想读的文件，以03101（必须包括最前面的0）表示你想要写的文件。[①]最后，权限集合应作为数字存储在`%edx`中。如果你不熟悉UNIX权限，使用0666（这里也必须包括最前面的0）权限即可。

(2) 接着，Linux将返回文件描述符到`%eax`。记住，你将在整个程序中用这个数字来指代这一文件。

(3) 接下来，你会对文件进行操作：读取和/或写入文件，每次都要向Linux指明要使用的文件

① 这部分内容将在10.2节详细阐述。

描述符。read是系统调用3，为了进行该调用，你必须将文件描述符存入%ebx，将存储数据的缓冲区地址存入%ecx中，将缓冲区大小放入%edx中。缓冲区将在5.2节中加以说明。read操作将返回从文件中读取的字符数，或者一个错误代码。因为错误代码是负数（负数的更多信息请参见第10章），所以很容易区分。write是系统调用4，需要的参数与read系统调用相同，唯一的区别是缓冲区应该已经填满了要写入的数据。write系统调用将把写入的字节数或错误代码存入%eax。

(4) 文件使用完毕后，你就可以告诉Linux将其关闭。此后你的文件描述符将不再有效。这是通过close，即系统调用6实现的。close唯一的参数就是文件描述符，应存储在%ebx中。

5.2 缓冲区和.bss

在上一节中，我们提到了缓冲区，但并没有解释什么是缓冲区。缓冲区是连续的字节块，用于批量数据传输。当你要求读取文件时，操作系统需要有一个地方来存储读取的数据，这个地方就称为缓冲区。一般缓冲区仅用于暂时存储数据，然后数据被从缓冲区中读出并转换成便于程序处理的形式，但我们的程序不会这么复杂。例如，你想从一个文件中读取一行文本，但不知道该行文本的长度。那么你可以从缓冲区读取大量字节/字符，寻找行结束符，并将直到该行结束符为止的所有字符复制到另一个位置。如果没有找到行结束符，你就会分配另一个缓冲区，并继续读取。在这种情况下，很可能最终会获得缓冲区中剩余的字符，而当下次需要这一文件中的数据时就可将这些字符作为寻找的起点。[①]

还有一点要注意，缓冲区的大小是固定的，由程序员设定。所以，如果你想一次读入500字节的数据，可以将500字节未使用内存位置的地址发送给read系统调用，并将数字500发送给它，这样read调用就知道数据的大小。数据可根据应用程序的需求设定大小。

要创建缓冲区，你需要保留静态或动态存储。静态存储就是我们到目前为止所说的以long或.byte指令声明的存储位置。动态存储将在9.4节中讨论。不过，用.byte声明缓冲区会存在问题。首先，输入工作让人倍感乏味。你不得不在.byte声明后键入500个数字，而除了用于占用空间，这些数字没有任何用处。其次，这会占用可执行空间的位置。在迄今为止的示例中，这种做法并没有占据太多空间，但这并不意味着在较大程序中仍如此。如果你想要用500字节，那就必须输入500字节，就浪费了可执行程序的500字节空间。但对于两者都有解决之道。到目前为止，我们已经讨论了程序的两个部分：.text段和.data段。除此之外，还存在.bss段。.bss段类似于数据段，不同的是它不占用可执行程序空间。.bss段可以保留存储位置，却不能对其进行初始化。在数据段中，你既可以保留存储位置，也能为其设置初始值；在.bss段中却不能设置

① 虽然这听起来很复杂，但在编程的时候，通常你无需直接处理缓冲区和文件描述符。在第8章中，你将了解如何使用Linux中的现有代码处理复杂的文件输入/输出。

初始值。对于缓冲区来说，这非常有用，因为我们并不需要初始化，只需要保留存储位置。为了做到这一点，我们发出以下命令：

```
.section .bss
.lcomm my_buffer, 500
```

.lcomm指令将创建一个符号my_buffer，指代我们用作缓冲区的500字节存储位置。接着执行以下指令，假定我们已经打开一个文件进行读取，并将文件描述符放置在%ebx中：

```
movl $my_buffer, %ecx
movl 500, %edx
movl 3, %eax
int  $0x80
```

上述指令将最多500字节的数据读入缓冲区。在本例中，我在my_buffer之前放置一个美元符号。记住，这样做的原因是：若没有美元符号，my_buffer将被视为以直接寻址方式访问的内存位置。美元符号则将寻址方式变为立即寻址方式，实际上就是将my_buffer表示的数字本身（即缓冲区的起始地址，也就是my_buffer的地址）加载到%ecx中。

5.3　标准文件和特殊文件

你可能会认为默认情况下程序启动时不打开任何文件，但事实上并非如此。Linux程序开始时至少有三个打开文件的描述符，分别是：

- STDIN

这是标准输入，是一个只读文件，通常代表键盘[①]，为文件描述符0。

- STDOUT

这是标准输出，是一个只写文件，通常代表屏幕显示，为文件描述符1。

- STDERR

这是标准错误，是一个只写文件，通常代表屏幕显示。处理结果通常输出到STDOUT，但在这个过程中出现的任何错误消息都输出到STDERR。这样一来，如果你希望的话，可以将两者分别存放在不同的地方。STDERR为文件描述符2。

这些“文件”都可以重定向，以一个真正的文件取代屏幕或键盘；这并不在本书讲述范围之

① 如我们前面所说，在Linux中几乎一切都是“文件”。键盘输入以及屏幕显示也被认为是文件。

内，但任何一本关于UNIX命令行的好书都会对此进行详述。程序本身甚至无需了解这种重定向的情况，因为程序可以像往常一样使用标准的文件描述符。

注意，你写的许多“文件”根本就不是文件。基于UNIX的操作系统把所有输入/输出系统都视作文件。网络连接被视为文件，串行端口被视为文件，甚至音频设备都被视为文件。进程之间的通信通常是通过称为管道的特殊文件实现的。与普通文件相比，这些文件有一些不同的打开和创建方法（例如，不使用open系统调用），但可以使用标准的read和write系统调用进行读写。

5.4 在程序中使用文件

我们将编写一个简单的程序来说明这些概念。这个程序将使用两个文件，从其中一个文件中读取数据，将所有小写字母转换为大写形式，然后写入另一个文件。在这样做之前，让我们想想需要做什么才能完成这一工作。

- 要有一个函数，它接受一个内存块，并将其内容转换为大写形式。这个函数将需要内存块地址及大小作为参数。
- 要有一反复读取数据到缓冲区的代码，并对缓冲区调用转换函数，然后将缓冲区写回另一个文件。
- 打开所需的文件以启动程序。

注意，上面逆序说明了事情的完成顺序。首先决定主要目标这种技巧在编写复杂程序时很有用，在本例中就是将字符块内容转换为大写形式。接下来考虑要完成目标需要进行的所有设置和处理：在本例中，你必须打开文件，不断地读取并写入块到磁盘。编程的一个关键是不断将问题分解为更小的问题，直到你可以轻松地解决这个问题，接着再将这些小问题重新组合为能运行的程序。[①]

你可能认为自己永远无法记住所有这些数字——系统调用号、中断号等。在这个程序中，我们还将介绍新的指令.equ，这条指令会对你有所帮助。.equ允许你为数字分配名称。例如，如果在写下LINUX_SYSCALL之后的任何时候发出.equ LINUX_SYSCALL, 0x80指令，汇编程序都将以0x80替换LINUX_SYSCALL。那么现在你可以写：

```
int $ LINUX_SYSCALL
```

这样很容易阅读，更容易记住。编程确实很复杂，但我们可以做许多与此类似的事情，以使编程更容易。

① Maureen Sprankle的《问题求解与编程概念》是探讨计算机编程中问题解决过程的一本佳作。

下面就是该程序。注意，我们所用的标签多于实际跳转所用的标签，因为其中一部分只是为了使程序保持清晰。尝试跟踪程序，看看在不同情况下会发生什么。我们将在程序之后给出深入分析。

```
# 目的:        本程序将输入文件的所有字母转换为大写字母
#              然后输出到输出文件
#
# 处理过程:    (1)打开输入文件
#              (2)打开输出文件
#              (3)如果未达到输入文件尾部:
#                 (a)将部分文件读入内存缓冲区
#                 (b)读取内存缓冲区的每个字节，如果该字节为小写字母，
#                    就将其转换为大写字母
#                 (c)将内存缓冲区写入输出文件

 .section .data

#######常数########

 # 系统调用号
 .equ SYS_OPEN, 5
 .equ SYS_WRITE, 4
 .equ SYS_READ, 3
 .equ SYS_CLOSE, 6
 .equ SYS_EXIT, 1

 # 文件打开选项（不同的值请参见/usr/include/asm/fcntl.h
 # 你可以通过将选项值相加或者进行OR操作组合使用选项）
 # 这将在10.1.2中深入阐述
 .equ O_RDONLY, 0
 .equ O_CREAT_WRONLY_TRUNC, 03101

 # 标准文件描述符
 .equ STDIN, 0
 .equ STDOUT, 1
 .equ STDERR, 2

 # 系统调用中断
 .equ LINUX_SYSCALL, 0x80

 .equ END_OF_FILE, 0 # 这是读操作的返回值
                     # 表明到达文件结束处

 .equ NUMBER_ARGUMENTS, 2
```

```
.section .bss
 #    冲区 - 从文件中将数据加载到这里，
 #           也要从这里将数据写入输出文件
 #           由于种种原因，缓冲区大小不应
 #           超过16 000字节
 .equ BUFFER_SIZE, 500
 .lcomm BUFFER_DATA, BUFFER_SIZE

 .section .text

 # 栈位置
 .equ ST_SIZE_RESERVE, 8
 .equ ST_FD_IN, -4
 .equ ST_FD_OUT, -8
 .equ ST_ARGC, 0      # 参数数目
 .equ ST_ARGV_0,    4 # 程序名
 .equ ST_ARGV_1,    8 # 输入文件名
 .equ ST_ARGV_2,    12 # 输出文件名

 .globl _start
_start:
 ###程序初始化###
 # 保存栈指针
 movl %esp, %ebp

 # 在栈上为文件描述符分配空间
 subl  $ST_SIZE_RESERVE, %esp


open_files:
open_fd_in:
 ###打开输入文件###
 # 打开系统调用
 movl $SYS_OPEN, %eax
 # 将输入文件名放入`%ebx`
 movl ST_ARGV_1(%ebp), %ebx
 # 只读标志
 movl $O_RDONLY, %ecx
 # 这实际上并不影响读操作
 movl $0666, %edx
 # 调用Linux
 int  $LINUX_SYSCALL

store_fd_in:
```

```
 # 保存给定的文件描述符
 movl %eax, ST_FD_IN(%ebp)

open_fd_out:
 ###打开输出文件###
 # 打开文件
 movl  $SYS_OPEN, %eax
 # 将输出文件名放入%ebx
 movl  ST_ARGV_2(%ebp), %ebx
 # 写入文件标志
 movl  $O_CREAT_WRONLY_TRUNC, %ecx
 # 新文件模式（如果已创建）
 movl  $0666, %edx
 # 调用Linux
 int   $LINUX_SYSCALL

store_fd_out:
 # 这里存储文件描述符
 movl  %eax, ST_FD_OUT(%ebp)

 ###主循环开始###
read_loop_begin:

 ###从输入文件中读取一个数据块###
 movl  $SYS_READ, %eax
 # 获取输入文件描述符
 movl  ST_FD_IN(%ebp), %ebx
 # 放置读取数据的存储位置
 movl  $BUFFER_DATA, %ecx
 # 缓冲区大小
 movl  $BUFFER_SIZE, %edx
 # 读取缓冲区大小返回到%eax中
 int   $LINUX_SYSCALL

 ###如到达文件结束处就退出###
 # 检查文件结束标记
 cmpl  $END_OF_FILE, %eax
 # 如果发现文件结束符或出现错误，就跳转到程序结束处
 jle   end_loop

continue_read_loop:
 ###将字符块内容转换成大写形式###
 pushl  $BUFFER_DATA      # 缓冲区位置
 pushl  %eax              # 缓冲区大小
```

```
 call  convert_to_upper
 popl  %eax                # 重新获取大小
 addl  $4, %esp            # 恢复%esp

 ###将字符块写入输出文件###
 # 缓冲区大小
 movl  %eax, %edx
 movl  $SYS_WRITE, %eax
 # 要使用的文件
 movl  ST_FD_OUT(%ebp), %ebx
 # 缓冲区位置
 movl  $BUFFER_DATA, %ecx
 int   $LINUX_SYSCALL

 ###循环继续###
 jmp read_loop_begin

end_loop:
 ###关闭文件###
 # 注意 - 这里我们无需进行错误检测
 #        因为错误情况不代表任何特殊含义
 movl  $SYS_CLOSE, %eax
 movl  ST_FD_OUT(%ebp), %ebx
 int   $LINUX_SYSCALL

 movl  $SYS_CLOSE, %eax
 movl  ST_FD_IN(%ebp), %ebx
 int   $LINUX_SYSCALL

 ###退出###
 movl  $SYS_EXIT, %eax
 movl  $0, %ebx
 int   $LINUX_SYSCALL

# 目的:   这个函数实际上将字符块内容转换为大写形式
#
# 输入:   第一个参数是要转换的内存块的位置
#         第二个参数是缓冲区的长度
#
# 输出:   这个函数以大写字符块覆盖当前缓冲区
#
# 变量:
#      %eax - 缓冲区起始地址
#      %ebx - 缓冲区长度
```

```
#      %edi - 当前缓冲区偏移量
#      %cl  - 当前正在检测的字节
#                    (%ecx的第一部分)
#

 ###常数##
 # 我们搜索的下边界
 .equ  LOWERCASE_A, 'a'
 # 我们搜索的上边界
 .equ  LOWERCASE_Z, 'z'
 # 大小写转换
 .equ  UPPER_CONVERSION, 'A' - 'a'

 ###栈相关信息###
 .equ  ST_BUFFER_LEN, 8    # 缓冲区长度
 .equ  ST_BUFFER, 12       # 实际缓冲区
convert_to_upper:
 pushl  %ebp
 movl  %esp, %ebp

 ###设置变量###
 movl  ST_BUFFER(%ebp), %eax
 movl  ST_BUFFER_LEN(%ebp), %ebx
 movl  $0, %edi

 # 如果给定的缓冲区长度为0即离开
 cmpl  $0, %ebx
 je    end_convert_loop

convert_loop:
 # 获取当前字节
 movb  (%eax,%edi,1), %cl

 # 除非该字节在'a'和'z'之间，否则读取下一字节
 cmpb  $LOWERCASE_A, %cl
 jl    next_byte
 cmpb  $LOWERCASE_Z, %cl
 jg    next_byte

 # 否则将字节转换为大写字母
 addb  $UPPER_CONVERSION, %cl
 # 并存回原处
 movb  %cl, (%eax,%edi,1)
next_byte:
 incl  %edi                # 下一字节
```

```
    cmpl   %edi, %ebx          # 继续执行程序
                               # 直到文件结束
    jne   convert_loop

end_convert_loop:
    # 无返回值，离开程序即可
    movl   %ebp, %esp
    popl   %ebp
    ret
```

输入程序为toupper.s，接着输入以下命令：

```
as toupper.s -o toupper.o
ld toupper.o -o toupper
```

上述命令会生成程序toupper，该程序将某个文件中的所有小写字母转换为大写字母。例如，要将文件toupper.s的内容转换为大写，可键入以下命令：

```
./toupper toupper.s toupper.uppercase
```

你会发现文件toupper.uppercase就是原文件的大写版。

让我们来看看程序的工作原理。

程序的第一部分被标记为CONSTANTS，在编程中，常数是程序汇编或编译时分配的值，分配后就不再改变。我的习惯是将所有常数放在程序的开始处。实际上只需要在使用之前声明即可，但把所有常数放在开始处更方便其查找。而用大写字母表示使哪些值是常数在程序中更醒目，更容易找到。[①]在汇编语言中，我们以之前提到过的.equ指令声明常数。这里，我们只是给迄今用到的数字赋值，例如系统调用号、系统中断号、文件打开选项等。

下一段被标记为BUFFERS，我们在这个程序中只使用一个缓冲区，即BUFFER_DATA。我们也定义了一个常数BUFFER_SIZE来保存缓冲区大小。如果总是在需要用到缓冲区大小时使用这个常数，而不是输入数字500，若以后缓冲区大小有改变，我们只需要更改这个常数值，而无需通读整个程序并分别修改每个值。

在继续看程序的_start段之前，先来看看程序结束处的convert_to_upper函数。这是实际上进行大小写转换的部分。

这一段以一列我们将用到的常数开始。将这些常数放在这里而不是整个程序开始处，是因为它们仅仅与这个函数有关。我们有如下定义：

① 这实际上是所有语言的程序员的标准做法。

```
.equ    LOWERCASE_A, 'a'
.equ    LOWERCASE_Z, 'z'
.equ    UPPER_CONVERSION, 'A' - 'a'
```

前两条指令只是定义搜索边界。记住，在计算机中字母是以数字表示的。因此，我们在比较、加法、减法以及其他可使用数字的操作中使用LOWERCASE_A。同时，也请注意我们定义的常数UPPER_CONVERSION。由于字母以数字表示，我们可以将它们做相减操作。用大写字母减去同一字母的小写字母，即可知道要使小写字母加上哪个数才能转换为大写字母。如果这么说不便于你的理解，请看一看ASCII代码表（见附录D）。你会发现字符A对应数字65，而字符a对应97，那么转换因子就是–32。如果将小写字母与–32相加，即可得到相应的大写字母。

在这之后，就是一些标记为STACK POSITIONS的常数。记住要在函数调用前将函数参数入栈。这些常数（为了能表示清楚，前缀为ST）定义了栈中哪个位置应有哪项数据。返回地址在位置4+%esp处，缓冲区长度在位置8+%esp处，缓冲区地址在位置12+%esp处。用符号代替数字而不使用数字本身的话，要了解使用和移动了哪些数据对我们来说更直观。

接下来是标签convert_to_upper，这是函数的入口，开始两行是标准函数行，用于存储栈指针。接下来的两行是：

```
movl ST_BUFFER(%ebp), %eax
movl ST_BUFFER_LEN(%ebp), %ebx
```

它们将函数参数移至相应的寄存器以供使用。接着，我们将0加载到%edi。这里要做的是循环取缓冲区中的每一个字节，通过以下过程实现：从存储位置%eax + %edi加载，递增%edi，并重复这一过程直到%edi等于存储在%ebx中的缓冲区长度。来看以下两行代码：

```
cmpl $0, %ebx
je   end_convert_loop
```

它们只是安全检查，以确保不会有大小为0的缓冲区。如果指定了大小为0的缓冲区，我们只需结束转换并离开。保护潜在用户，防止编程错误，这对于程序员来说是一项重要任务。你可以总是说明自己的函数不应接受大小为0的缓冲区，但如果让函数对此进行检查并在发生这种情况时可靠地退出就更好了。

接下来，循环开始了。首先，将一字节移入%cl，相应的代码是：

```
movb (%eax,%edi,1), %cl
```

该指令使用间接变址寻址方式，表示从%eax开始，向前%edi个位置（每个位置为1字节大小），并将该位置的值放入%cl。在这条指令后，程序查看该值是否在小写字母a到z的范围内，要检测这点，只要查看字符是否比a小即可。如果比a小，那就不可能是小写字母。同样，如果比z大，

也不可能是小写字母。因此，在上述两种情况下只需继续即可。如果在小写字母范围内，就与大写字母转换量相加，并将其存回缓冲区。

无论是哪种情况，接下来都通过递增%cl获取下一个值。然后，程序查看是否到达缓冲区结束处。如果未到达，就跳转到循环开始处（convert_loop标签）。如果已经到达结束处，只要继续执行到函数结束即可。由于我们是直接更改缓冲区，因此无需向调用程序返回任何值——更改已经在缓冲区中。标签end_convert_loop不是必需的，但有了这个标签你就容易看到程序的大小写转换部分在哪里。

现在我们知道转换进程的原理了。现在需要弄清楚如何从文件中获得数据以及如何向文件写数据。

在读写文件前，我们先要打开文件。UNIX的open系统调用对此进行处理，需要以下参数。

- 与以往一样，%eax含有系统调用号，在本例中为-5。
- %ebx中含有指向文件名（相应文件是要打开的那个）的字符串。该字符串必须以空字符结束。
- %ecx中包含文件打开选项。这些选项告诉Linux如何打开文件，表明文件是打开用于读、写、读写、如果不存在则创建、如果已存在则删除等。我们将在10.2中详细说明如何创建用于选项的数字。
- %edx包含用于打开文件的权限。当必须先创建文件时会用到此项，以便让Linux了解创建文件时应设置什么权限。这一项与一般UNIX权限一样，以八进制表示。[①]

在进行系统调用后，新打开文件的文件描述符存储在%eax中。

那么，打开哪些文件呢？在本例中，我们将打开在命令行中指定的文件。幸运的是，命令行参数以经由Linux存放在一个易于访问的位置，而且以经以空字符结束。当Linux程序开始时，所有指向命令行参数的指针都存储于栈中。参数数目存储于8(%esp)，程序名存储于12(%esp)，而参数存储于16(%esp)及其后的存储位置。

在C编程语言中，这称为argv数组，因此我们也将在程序中用这个名字。

我们的程序首先在%ebp中保存当前栈位置，然后在栈中保留一些空间存储文件描述符。在此之后，我们就开始打开文件。

程序打开的第一个文件是输入文件，就是第一个命令行参数。我们通过系统调用来对此进行设置。这里将文件名置于%ebx中，只读模式对应的数字置于%ecx中，默认模式$0666置于%edx

① 如果你熟悉UNIX权限，将$0666置入即可。不要遗漏开始处的0，因为它表示八进制。

中，系统调用号置于%eax中。在系统调用后，文件打开了，文件描述符被存储到%eax中。[①]接着，我们将文件描述符转移到栈中对应的位置。

对输出文件也做同样处理，唯一区别在于输出文件被创建为“只读、如不存在则创建”形式。其文件描述符也存储下来。

现在来到了主要部分——读/写循环。基本上讲，我们将从输入文件读取固定大小的数据块，调用转换函数对数据块进行转换，然后写到输出文件。尽管我们读取固定大小的数据块，但数据块的大小并不影响这个程序——我们只是对字符序列进行操作。我们能随心所欲地读取任意大小的数据块，且程序仍能正常运作。

循环的第一部分是使用read系统调用读取数据。这个调用的参数是一个用于读取的文件描述符、一个用于写入的缓冲区，以及缓冲区大小（即可写入的最大字节数）。系统调用返回实际读取的字节数或文件结束符（数字0）。

在读取一个数据块后，我们检测%eax是否含有文件结束标记。如果找到即退出循环，否则就继续执行。

在读取数据后，程序调用convert_to_upper函数，参数为刚才读入的缓冲区以及在前一个系统调用中读取的字符数。在执行这个函数后，缓冲区已经变为大写字符，程序准备好写文件了。接着我们将寄存器恢复为函数调用之前的值。

最后发出write系统调用。此调用将数据从缓冲区移出到文件，其他方面与read系统调用一样。现在，我们回到循环起始处。

在循环退出后（记住，当一次读取后探测到文件结束时即退出），程序简单地关闭其文件描述符并退出。关闭系统调用将获取%ebx中的文件描述符来关闭文件。

然后，程序就结束了！

5.5　温故知新

5.5.1　理解概念

- 描述文件描述符的生命周期。

① 注意，我们对此不做任何错误检测，但这只是为了使程序简单。在一般程序中，每一次系统调用通常都应检测是调用成功还是失败。在失败的情况下，%eax将保持错误码而非返回值。错误码是负数，因此可通过将%eax与0相比较检测到，并在小于0的情况下跳转。

- 什么是标准文件描述符？它们有什么用途？
- 什么是缓冲区？
- .data和.bss段的区别是什么？
- 与读写文件相关的系统调用是哪些？

5.5.2 应用概念

- 更改toupper程序，使之从STDIN读入，写到STDOUT，而不是使用命令行指定的文件。
- 更改缓冲区大小。
- 重新编写程序，使之使用.bss段中的存储空间代替栈来存储文件描述符。
- 编写程序创建一个名为heynow.txt的文件，并向文件中写入“Hey diddle diddle!”。

5.5.3 深入学习

- 缓冲区大小会造成什么影响？
- 这些系统调用各会返回什么错误结果？
- 使程序能根据ARGC指定的命令行参数数目的不同，成为既能接收命令行参数，又能使用STDIN或STDOUT的程序。
- 更改程序，使之能检测每次系统调用的结果，并在发生错误时向STDOUT打印错误消息。

第6章 读写简单记录

如第5章所述，许多应用程序处理持久性数据，即通过将数据存储在文件中，使数据的寿命比程序长。你可以关闭程序，然后再打开，然后回到之前打开的地方。现在，存在两种持久性数据：结构化的和非结构化的。非结构化数据就如同我们在toupper程序中所处理的数据，仅处理某个人输入的文本文件。文件内容对于程序来说不可用，因为程序无法解读用户试图通过无序文本表达的内容。

结构化数据则正好相反，是计算机擅长处理的数据。结构化数据是拆分为字段和记录的数据，且绝大部分为固定长度的字段和记录。由于数据划分为固定长度的记录和固定格式的字段，计算机就能解读数据。结构化数据可包含变长字段，但在此情况下，你最好使用数据库。[①]

本章涉及读写固定长度的简单记录。比如，我们想要存储一些认识的人的基本信息，可以设想关于他们的以下固定长度的示例记录：

- 姓——40字节；
- 名——40字节；
- 地址——240字节；
- 年龄——4字节。

本例中，除了年龄是使用4字节字的数字字段（我们可以只用单个字节，但使用一个字更便于处理），其他所有字段都是字符数据。

在编程过程中，某些定义是你经常会在一个或多个程序中反复使用的。你最好将这些定义分别独立存放入文件，这些文件仅仅在需要时才包含在汇编语言文件中。例如，在下面几个程序中，我们将访问上述记录的不同部分。这意味着为了用基址寻址方式访问各字段，我们需要各字段相

① 数据库是处理持久性结构化数据的程序。你无需编写程序从磁盘读写数据、查找数据或进行基本处理。这是结构化数据的高级别接口，尽管增加了一些开销和额外的复杂度，但对于复杂数据处理任务极为有用。学习数据库工作原理的参考书见第13章。

对于记录起始处的偏移量。以下常量描述了上述结构的各字段偏移量。将这些常量放入名为record-def.s的文件：

```
.equ RECORD_FIRSTNAME, 0
.equ RECORD_LASTNAME, 40
.equ RECORD_ADDRESS, 80
.equ RECORD_AGE, 320

.equ RECORD_SIZE, 324
```

此外，有几个常量我们在程序中曾一再定义，因此将其放入某个文件是很有用的，这样就不必总是重复输入它们。这里将其放入名为linux.s的文件。

```
# Linux常量定义

# 系统调用号
.equ SYS_EXIT, 1
.equ SYS_READ, 3
.equ SYS_WRITE, 4
.equ SYS_OPEN, 5
.equ SYS_CLOSE, 6
.equ SYS_BRK, 45

# 系统调用中断号
.equ LINUX_SYSCALL, 0x80

# 标准文件描述符
.equ STDIN, 0
.equ STDOUT, 1
.equ STDERR, 2

# 通用状态码
.equ END_OF_FILE, 0
```

我们将采用record-def.s中定义的结构编写本章的3个程序。第一个程序将生成包含几个如上定义记录的文件。第二个程序将显示文件中的记录。第三个程序将每个记录中的年龄增加一岁。

除了将在所有这些程序中使用的标准常量，我们还要在其中几个程序中使用两个函数：一个用于读记录，一个用于写记录。

这两个函数需要哪些参数才能工作呢？我们大致需要：

- 能将记录读入的缓冲区位置；
- 想写入或从中读取的文件的文件描述符。

先来看读取函数：

```
 .include "record-def.s"
 .include "linux.s"

# 目的：此函数从文件描述符读取一条记录
#
# 输入：文件描述符及缓冲区
#
# 输出：本函数将数据写入缓冲区
#       并返回状态码
#

# 栈局部变量
 .equ ST_READ_BUFFER, 8
 .equ ST_FILEDES, 12
 .section .text
 .globl read_record
 .type read_record, @function
read_record:
 pushl %ebp
 movl %esp, %ebp

 pushl %ebx
 movl ST_FILEDES(%ebp), %ebx
 movl ST_READ_BUFFER(%ebp), %ecx
 movl $RECORD_SIZE, %edx
 movl $SYS_READ, %eax
 int  $LINUX_SYSCALL

 # 注意 - %eax中含返回值，我们将该值传回调用程序
 popl %ebx

 movl %ebp, %esp
 popl %ebp
 ret
```

这是个相当简单的函数，只是从给定的文件描述符读取特定结构大小的数据，放入相应大小的缓冲区。而写函数与之类似：

```
 .include "linux.s"
 .include "record-def.s"
# 目的：本函数将一条记录写入给定文件描述符
#
# 输入：文件描述符和缓冲区
```

```
#
# 输出：本函数产生状态码
#
# 栈局部变量
 .equ ST_WRITE_BUFFER, 8
 .equ ST_FILEDES, 12
 .section .text
 .globl write_record
 .type write_record, @function
write_record:
 pushl %ebp
 movl %esp, %ebp

 pushl %ebx
 movl  $SYS_WRITE, %eax
 movl  ST_FILEDES(%ebp), %ebx
 movl  ST_WRITE_BUFFER(%ebp), %ecx
 movl  $RECORD_SIZE, %edx
 int   $LINUX_SYSCALL

 # 注意 - %eax含返回值，我们将之传回调用程序
 popl %ebx

 movl %ebp, %esp
 popl %ebp
 ret
```

现在我们已经有了基本定义，可以开始写程序了。

6.1 写入记录

这个程序简单地将一些硬编码记录写入磁盘。具体来讲，程序将：

- 打开文件；
- 写3条记录；
- 关闭文件。

输入以下代码到文件write-records.s：

```
.include "linux.s"
.include "record-def.s"

.section .data
```

```
 # 我们想写入的常量数据
 # 每个数据项以空字节（0）填充到适当的长度

 # .rept用于填充每一项。.rept告诉汇编程序将
 # .rept和.endr之间的段重复指定次数
 # 在这个程序中，此指令用于将多余的空白字符
 # 增加到每个字段末尾以将之填满
record1:
 .ascii "Fredrick\0"
 .rept 31  # 填充到40字节
 .byte 0
 .endr

 .ascii "Bartlett\0"
 .rept 31  # 填充到40字节
 .byte 0
 .endr

 .ascii "4242 S Prairie\nTulsa, OK 55555\0"
 .rept 209  # 填充到240字节
 .byte 0
 .endr

 .long 45

record2:
 .ascii "Marilyn\0"
 .rept 32  # 填充到40字节
 .byte 0
 .endr

 .ascii "Taylor\0"
 .rept 33 #Padding to 40 bytes
 .byte 0
 .endr

 .ascii "2224 S Johannan St\nChicago, IL 12345\0"
 .rept 203  # 填充到240字节
 .byte 0
 .endr

 .long 29

record3:
```

```
 .ascii "Derrick\0"
 .rept 32  # 填充到40字节
 .byte 0
 .endr

 .ascii "McIntire\0"
 .rept 31  # 填充到40字节
 .byte 0
 .endr

 .ascii "500 W Oakland\nSan Diego, CA 54321\0"
 .rept 206  # 填充到240字节
 .byte 0
 .endr

.long 36

 # 这是我们要写入文件的文件名:
 .ascii "test.dat\0"

 .equ ST_FILE_DESCRIPTOR, -4
 .globl _start
_start:
 # 复制栈指针到%ebp
 movl %esp, %ebp
 # 为文件描述符分配空间
 subl $4, %esp

 # 打开文件
 movl $SYS_OPEN, %eax
 movl $file_name, %ebx
 movl $0101, %ecx  # 本指令表明如文件不存在则创建
                   # 并打开文件用于写入
 movl $0666, %edx
 int  $LINUX_SYSCALL

 # 存储文件描述符
 movl %eax, ST_FILE_DESCRIPTOR(%ebp)

 # 写第一条记录
 pushl ST_FILE_DESCRIPTOR(%ebp)
 pushl $record1
 call  write_record
 addl  $8, %esp

 # 写第二条记录
```

```
pushl ST_FILE_DESCRIPTOR(%ebp)
pushl $record2
call  write_record
addl  $8, %esp

# 写第三条记录
pushl ST_FILE_DESCRIPTOR(%ebp)
pushl $record3
call  write_record
addl  $8, %esp

# 关闭文件描述符
movl $SYS_CLOSE, %eax
movl ST_FILE_DESCRIPTOR(%ebp), %ebx
int  $LINUX_SYSCALL

# 退出程序
movl $SYS_EXIT, %eax
movl $0, %ebx
int  $LINUX_SYSCALL
```

这是一个相当简单的程序，仅定义要写入.data段的数据，以及适当的系统调用和函数调用来完成工作。要复习所有用到过的系统调用，请参见附录C。

你也许注意到了下面这两行代码：

```
.include "linux.s"
.include "record-def.s"
```

这些声明使指定文件被粘贴到代码的当前位置。你无需在函数中使用此声明，因为链接器能将导出函数与.global指令相结合。但在另一个文件中定义的常量确实需要通过这种方式导入。

你可能也注意到了这里使用的新编译指令.rept。这条指令将文件中.rept和.endr指令之间的段重复.rept之后指定的次数。我们通常用该指令填充.data段中的值。在本例中，我们将空白字符增加到每个字段末尾，直到字段达到定义的长度。

为了生成应用程序，我们运行以下命令：

```
as write-records.s -o write-record.o
as write-record.s -o write-record.o
ld write-record.o write-records.o -o write-records
```

目前我们分别汇编两个文件，然后用链接器将之合并。要运行程序，请输入以下命令：

```
./write-records
```

这条命令会创建一个包含记录的test.dat文件。但是，由于记录包含非打印字符（即空字符），可能无法通过文本编辑器查看。因此，我们需要下一个程序来为我们读取记录。

6.2 读取记录

现在，我们将考虑读取记录的过程。这个程序将读取每个记录，并显示每条记录中的名。

由于每个人的姓名长度不同，我们需要一个函数来计算要写入的字符数。由于我们用空字符填充每个字段，因此只需对空字符之前的字符计数。[①]注意，这意味着每条记录都必须包含至少一个空字符。

下面就是代码，将其放在文件count-chars.s中：

```
# 目的：对字符进行计数，直到遇到空字符
#
# 输入：字符串地址
#
# 输出：将计数值返回到%eax
#
# 过程：
#  用到的寄存器：
#    %ecx - 字符计数
#    %al - 当前字符
#    %edx - 当前字符地址

 .type count_chars, @function
 .globl count_chars

 # 这是我们的一个参数在栈上的位置
 .equ ST_STRING_START_ADDRESS, 8
count_chars:
 pushl %ebp
 movl %esp, %ebp

 # 计数器从0开始
 movl $0, %ecx
 # 数据的起始地址
 movl ST_STRING_START_ADDRESS(%ebp), %edx

count_loop_begin:
```

① 如果你用过C语言，就会发现这与strlen函数的功能相同。

```
    # 获取当前字符
    movb (%edx), %al
    # 是否为空字符?
    cmpb $0, %al
    # 若为空字符则结束
    je count_loop_end
    # 否则，递增计数器和指针
    incl %ecx
    incl %edx
    # 返回循环起始处
    jmp count_loop_begin

count_loop_end:
    # 结束循环，将计数值移入%eax并返回
    movl %ecx, %eax

    popl %ebp
    ret
```

正如你所看到的，这是一个相当简单的函数，只是遍历所有字节并计数，直到遇到空字符。然后，它返回计数值。

我们的记录读取程序也是相当简单的。程序将完成如下步骤：

- 打开文件；
- 尝试读取一条记录；
- 若到达文件结束处则退出，否则计算名的字符数；
- 将名写到STDOUT；
- 写一个换行符到STDOUT；
- 返回并读取另一条记录。

为了编写此程序，我们需要另一个简单函数——写一个换行符到STDOUT的函数。将下面的代码放置到write-newline.s文件中：

```
    .include "linux.s"
    .globl write_newline
    .type write_newline, @function
    .section .data
newline:
    .ascii "\n"
    .section .text
    .equ ST_FILEDES, 8
write_newline:
```

```
 pushl %ebp
 movl %esp, %ebp

 movl $SYS_WRITE, %eax
 movl ST_FILEDES(%ebp), %ebx
 movl $newline, %ecx
 movl $1, %edx
 int  $LINUX_SYSCALL
 movl %ebp, %esp
 popl %ebp
 ret
```

现在，我们准备编写主程序，read-records.s的代码如下：

```
 .include "linux.s"
 .include "record-def.s"

 .section .data
file_name:
 .ascii "test.dat\0"

 .section .bss
 .lcomm record_buffer, RECORD_SIZE

 .section .text
 # 主程序
 .globl _start
_start:
 # 这些是我们将存储输入输出描述符的栈位置
 # (仅供参考：也可以用一个.data段中
 # 的内存地址代替
 .equ ST_INPUT_DESCRIPTOR, -4
 .equ ST_OUTPUT_DESCRIPTOR, -8

 # 复制栈指针到%ebp
 movl %esp, %ebp
 # 为保存文件描述符分配空间
 subl $8, %esp

 #打开文件
 movl $SYS_OPEN,  %eax
 movl $file_name, %ebx
 movl $0, %ecx              # 表示只读打开
 movl $0666, %edx
 int  $LINUX_SYSCALL
```

```
    #保存文件描述符

    movl %eax, ST_INPUT_DESCRIPTOR(%ebp)

    # 即使输出文件描述符是常数，我们也
    # 将其保存到本地变量，这样如果稍后
    # 决定不将其输出到STDOUT，很容易
    # 加以更改
    movl $STDOUT, ST_OUTPUT_DESCRIPTOR(%ebp)

record_read_loop:
    pushl ST_INPUT_DESCRIPTOR(%ebp)
    pushl $record_buffer
    call  read_record
    addl  $8, %esp

    # 返回读取的字节数
    # 如果字节数与我们请求的字节数不同，
    # 说明已到达文件结束处或出现错误，
    # 我们就要退出
    cmpl $RECORD_SIZE, %eax
    jne  finished_reading

    # 否则，打印出名，但我们首先必须知道名的大小
    pushl $RECORD_FIRSTNAME + record_buffer
    call  count_chars
    addl  $4, %esp
    movl  %eax, %edx
    movl  ST_OUTPUT_DESCRIPTOR(%ebp), %ebx
    movl  $SYS_WRITE, %eax
    movl  $RECORD_FIRSTNAME + record_buffer, %ecx
    int   $LINUX_SYSCALL

    pushl ST_OUTPUT_DESCRIPTOR(%ebp)
    call  write_newline
    addl  $4, %esp

    jmp   record_read_loop

finished_reading:
    movl $SYS_EXIT, %eax
    movl $0, %ebx
    int  $LINUX_SYSCALL
```

要生成这个程序，我们要汇编其所有组成文件并链接它们：

```
as read-record.s -o read-record.o
as count-chars.s -o count-chars.o
as write-newline.s -o write-newline.o
as read-records.s -o read-records.o
ld read-record.o count-chars.o write-newline.o \
   read-records.o -o read-records
```

第一行中的反斜线只是表明命令在下一行继续。你可以通过命令./read-records读取记录。

如上所述，这个程序打开该文件，然后运行用于读取的循环，检查文件是否结束，并写入名。也许对于你来说，下面一行代码中存在新结构：

```
pushl $RECORD_FIRSTNAME + record_buffer
```

它看起来就像我们把add指令和push结合起来一样，但实际上并非如此。你看，RECORD_FIRSTNAME和record_buffer都是常数，前者是直接常数，通过使用.equ指令创建，后者则是由汇编程序自动定义为标签（其值是紧随其后的数据的起始地址）。由于两者都是汇编程序知道的常数，因此汇编程序在实际汇编程序时能将两者相加，这样整个指令就只是立即寻址方式的单个常量入栈。

RECORD_FIRSTNAME常量是一条记录从起始地址到名字段之间的字节数。record_buffer是用于保存记录的缓冲区的名字。将以上两者相加，我们就可获得存储在record_buffer中记录的名字段地址。

6.3 修改记录

本节，我们将编写完成如下步骤的程序：

- 打开一个输入文件和一个输出文件；
- 从输入文件中读取记录；
- 递增年龄；
- 将新记录写入输出文件。

如同我们最近遇到的多数程序一样，这个程序相当直观。[①]

```
.include "linux.s"
```

① 学习编程机制后你会发现，一旦知道了自己想要做什么，大多数程序对于你来说就都非常简单了。它们大多数初始化数据，在循环中进行一些处理，最后进行数据清除。

```
 .include "record-def.s"
 .section .data
input_file_name:
 .ascii "test.dat\0"

output_file_name:
 .ascii "testout.dat\0"

 .section .bss
 .lcomm record_buffer, RECORD_SIZE

 # 局部变量的栈偏移量
 .equ ST_INPUT_DESCRIPTOR, -4
 .equ ST_OUTPUT_DESCRIPTOR, -8

 .section .text
 .globl _start
_start:
 # 复制栈指针并为局部变量分配空间
 movl %esp, %ebp
 subl $8, %esp

 # 打开用于读取的文件
 movl $SYS_OPEN, %eax
 movl $input_file_name, %ebx
 movl $0, %ecx
 movl $0666, %edx
 int  $LINUX_SYSCALL

 movl %eax, ST_INPUT_DESCRIPTOR(%ebp)

 # 打开用于写入的文件
 movl $SYS_OPEN, %eax
 movl $output_file_name, %ebx
 movl $0101, %ecx

 movl $0666, %edx
 int  $LINUX_SYSCALL

 movl %eax, ST_OUTPUT_DESCRIPTOR(%ebp)

loop_begin:
 pushl ST_INPUT_DESCRIPTOR(%ebp)
 pushl $record_buffer
 call  read_record
```

```
addl $8, %esp

# 返回读取的字节数
# 如果字节数与我们请求的字节数不同，
# 说明已到达文件结束处或出现错误，
# 我们就要退出
cmpl $RECORD_SIZE, %eax
jne  loop_end

# 递增年龄
incl record_buffer + RECORD_AGE

# 写记录
pushl ST_OUTPUT_DESCRIPTOR(%ebp)
pushl $record_buffer
call  write_record
addl  $8, %esp

jmp loop_begin

loop_end:
 movl $SYS_EXIT, %eax
 movl $0, %ebx
 int  $LINUX_SYSCALL
```

我们可以将以上代码输入名为add-year.s的文件。为生成程序，请输入以下命令[①]：

```
as add-year.s -o add-year.o
ld add-year.o read-record.o write-record.o -o add-year
```

要运行此程序，请输入以下命令[②]：

```
./add-year
```

本程序将test.dat中每一条记录的年字段值增加一年，并将新记录写到文件testout.dat。

正如你所看到的，写固定长度的记录非常简单。你只需要读取缓冲区块的数据，进行处理，然后将它们写回文件。遗憾的是，这个程序并未将新的年龄显示在屏幕上，你无法验证程序是否有效。这是因为要到第8章和第10章才会谈到显示数字的内容。阅读第8章和第10章后，你可能要回头重写程序，以显示我们修改的数值数据。

① 前提是你已经在前面的示例中生成目标文件read-record.o和write-record.o。如果你没有这么做的话，那必须先生成read-record.o和write-record.o。

② 前提是你在之前运行write-records时已经创建文件，否则需要在运行此程序之前先运行write-records。

6.4　温故知新

6.4.1　理解概念

- 什么是记录？
- 与变长记录相比，固定长度记录的优点是什么？
- 如何在多个汇编源文件中包含常量？
- 为什么要把一个项目拆分成多个源文件？
- 指令`incl record_buffer + RECORD_AGE`的作用是什么？这条指令用的是什么寻址方式？有几个操作数？哪些部分是由汇编程序处理，哪些部分是在程序运行时进行处理的？

6.4.2　应用概念

- 将另一个数据成员添加到本章定义的`person`结构，并为此重新编写读取和写入函数及程序。记住，要重新汇编和链接文件，然后才能运行程序。
- 创建一个程序，使用一个循环来写30个相同的记录到一个文件中。
- 创建一个程序，在文件中寻找年龄最大值，并将其作为程序的状态码返回。
- 创建一个程序，在文件中寻找年龄最小值，并将其作为程序的状态码返回。

6.4.3　深入学习

- 改写本章中的程序，用命令行参数来指定文件名。
- 研究如何使用`lseek`系统调用。改写`add-year`程序，打开源文件用于读取和写入（`$2`用于读/写模式），并将修改后的记录写入当初被读取的那个文件。
- 研究这些程序中进行的系统调用可能会返回的各种错误代码。选择一个程序进行改写，添加代码来检查`%eax`中的错误情况，如果发现错误就将相关消息写入`STDERR`并退出。
- 编写程序，从键盘读取数据，然后借此添加一个记录到文件。记住，你必须确保数据结束处至少有一个空字符，也必须提供方式让用户表明他们已经完成输入。因为我们还没有涉及如何将字符转换成数字，你无法从键盘读取年龄，所以必须有一个默认年龄。
- 写一个名为`compare-strings`的函数，该函数将比较最多含有5个字符的两个字符串。然后写一个程序，允许用户输入5个字符，并让程序返回所有名以这5个字符开头的记录。

第7章 开发健壮的程序

本章涉及如何开发健壮的程序。健壮的程序能够优雅地处理错误情况。无论用户做什么，健壮的程序都不会崩溃。构建健壮的程序对于编程实践来说非常重要。编写健壮的程序，要求自律和刻苦：通常需要寻找每一个可能发生的问题，并为程序采取应对计划。

7.1 将时间用在何处

程序员往往不擅长时间分配。几乎在每一个编程项目中，程序员花费的时间都是预先估计时间的2倍、4倍甚至8倍。造成这个问题的原因是多方面的，包括但不限于以下几个。

- 程序员通常不会考虑到每天中的会议或其他非编程活动的时间。
- 程序员经常低估项目的反馈次数（变更请求和批准请求来回需要多长时间）。
- 程序员并不总能理解他们工作的全部范畴。
- 程序员经常不得不为与常见项目类型完全不同的项目预估时间表，因此无法准确安排。
- 程序员往往大大低估最终获得健壮程序所需的时间。

最后一项是我们的关注点。开发健壮的程序需要大量的时间和精力。这比人们通常猜测的时间要多得多，即使有经验的程序员也是如此。程序员过度专注于解决手头的问题，因此往往会忽略可能产生的次要问题。

在`toupper`程序中，如果用户选择的文件不存在，我们并不采取任何行动。程序将继续并试图完成任务。程序不会报告任何错误消息，因此用户甚至不知道他们键入了错误的名称。比方说，目标文件在网络驱动器上，而网络暂时出现故障。操作系统将状态代码返回到`%eax`中，但我们不对其进行检查。因此，如果发生错误，用户完全不知道。这个程序完全不具备健壮性。如你所见，即使在简单的程序中，也存在很多可能发生错误的地方，而程序员必须对其加以处理。

在较大的程序中则更容易出问题。通常可能的错误条件多过可能成功的条件。因此，你应该总是预计会将大量时间用在检查状态代码、编写错误处理程序并执行类似任务使程序健壮上。如果需要用两周时间开发一个程序，你可能需要至少另外两周来保证其健壮性。记住，在屏幕上弹出的每条错误消息都需要有人进行编程。

7.2　开发健壮程序的技巧

7.2.1　用户测试

测试是程序员最重要的一个任务。对于未经测试的代码，你应该假设其无法正常工作。但是，测试不只是为了确保程序功能正常，还要确保程序不会崩溃。例如，对于一个只应处理正数的程序，需要测试如果用户输入负数、字母或0时会发生什么情况。你必须测试当用户在数字之间加入空格时，以及出现其他小概率事件时会发生什么情况。你需要确保处理用户数据的方式能让用户理解，传递数据的方式对于程序的其余部分有意义。当你的程序发现无意义的输入时，需要执行相应的操作。根据具体程序的不同，这可能包括结束程序、提示用户重新输入值、通知中央错误日志、回滚某个操作，或忽略并继续。

你不仅要自己测试程序，还需要让其他人测试。你应该让其他程序员以及程序用户帮助测试程序。如果用户觉得是问题，即使在你看来不算，你也应加以解决。如果用户不知道如何正确使用你的程序，这就是需要修复的错误。

你会发现，用户在你的程序中找到的错误往往比你多。原因是用户不知道计算机期望什么输入；而你知道计算机期望什么数据，因此更容易输入计算机能理解的数据。用户输入的数据只是对于他们有意义。为了测试，让非程序员使用你的程序，通常能让你更精确地了解程序实际上的健壮程度。

7.2.2　数据测试

在设计程序时，每个函数都需要非常明确地指明会接受或者不会接受的数据类型和范围。然后，你需要测试这些函数，确保其在接受传递来的相应数据时遵循规范。最重要的是测试*极端案例*或*边缘案例*。极端案例是最有可能造成问题或异常行为的输入。

测试数字数据时，有几个总是需要测试的极端案例：

- 数字0；
- 数字1；

- 在预期范围内的一个数字；
- 超出预期范围的一个数字；
- 在预期范围内的第一个数字；
- 在预期范围内的最后一个数字；
- 低于预期范围的第一个数字；
- 高于预期范围的第一个数字。

例如，如果我有一个程序，它应该接受5和200之间的值，我应该测试0、1、4、5、153、200、201、255（153和255分别是在范围内外随机选择的数字）。这同样适用于你的其他任何数据列表。你需要测试程序在针对只有0项、1项，还有具有大量列表项时的程序行为。此外，你也应该测试所有转折点。例如，如果针对30岁以上和以下的人的代码不同，那么你至少需要测试年龄为29、30、31岁人的情况。

你会假定某些内部函数会接收到良好数据，因为你已经在此之前进行了错误检查。然而，在开发时你经常需要检查是否有错误，因为你的其他代码可能有错误。为了在开发过程中验证数据的一致性和有效性，大多数语言有一个很容易就能检查数据正确性假设的工具；在c语言中，就是`assert`宏。你可以将`assert(a > b)`加入代码，在到达条件为假的代码时它会报错。此外，由于代码稳定后这种检查就是在浪费时间，`assert`宏允许你在编译时关闭断言。这可以确保在向公众发布的代码中，你的函数能接受良好的数据且不会降低速度。

7.2.3 模块测试

你不仅应当测试整个程序，还要测试程序的各个组成部分。开发程序时，你应该测试各个函数，向其提供你创建的数据，以确保函数能作出恰当反应。

为了有效地做到这一点，你必须开发唯一目的是调用函数进行测试的函数。这些函数称为*驱动程序*（不要与硬件驱动程序混淆）。此类函数只是加载函数，向其提供数据，并检查其结果。当你正在编制未完成的各部分程序时，这点非常有用。由于你无法同时测试程序的所有部分，可以创建一个驱动程序，单独测试每个函数。

此外，测试代码也可以调用尚未开发的函数。为了解决这个问题，你可以编写一个称为*存根*的小函数，该函数只是返回函数需要处理的值。例如，在一个电子商务应用程序中，我有一个名为`is_ready_to_checkout`的函数。在有时间真正编写此函数之前，我只是将它设置为每次调用都返回`true`，这样依赖该函数的函数将会获得答案。这使我无需完全实现`is_ready_to_checkout`函数就能测试依赖于它的函数。

7.3　有效处理错误

重要的不仅是知道如何进行测试，当检测到错误时知道该如何做同样重要。

7.3.1　万能的错误处理代码

真正健壮的软件对于每种可能的偶然事件都有唯一的错误代码。只要知道错误代码，你应该就能找到代码中发出错误信号的位置。

这点很重要，因为错误代码通常是在程序报告错误时所有用户都不得不参考的。因此，错误代码必须尽可能有用。

错误代码也应伴随描述性的错误消息。但是，只有在极少数情况下，错误消息才应该尝试预测为什么会发生相应错误。它应该仅仅叙述发生了什么事。1995年，我曾为一个因特网服务提供商工作。我们支持的一种Web浏览器试图猜测每一个网络错误的原因，而不仅仅是报告错误。如果计算机没有连接到因特网，而用户试图连接一个网站，它会说因特网服务提供商出错，服务器故障，用户应联系因特网服务提供商纠正问题。我们的电话有近1/4是来自收到这一消息的人，但实际上只需要连接到因特网再尝试使用浏览器即可。如你所见，与修复的相比，试图诊断出了什么问题会导致更多问题。因此仅报告错误代码和消息，让用户通过不同的资源进行故障排除是更好的做法。故障排除指南，并非程序本身，更适合为每个错误消息列出可能的原因及应当采取的行动。

7.3.2　恢复点

为了简化错误处理，我们往往需要把程序分割成不同单元，将其中每个单元在出现故障时都作为一个整体恢复。例如，你可能会拆分程序，将“读取配置文件”作为一个单元。如果在某个时候（如打开文件、读取文件、试图解码文件等）读取配置文件失败，那么程序会将之视为配置文件问题，并跳到这个问题的恢复点。这样一来，你能大大减少程序所需的错误处理机制，因为错误恢复是在更为普遍的层级上实现的。

注意，即使有了恢复点，错误信息仍需要具体说明问题是什么。恢复点是进行错误恢复的基本单位，而不是错误检测。错误检测仍然需要非常精确，而错误报告需要给出确切的错误代码和信息。

当使用恢复点时，你经常需要清理代码来处理不同的突发事件。例如，就我们的配置文件示例而言，恢复函数需要包括用于检查配置文件是否仍处于打开状态的代码。根据发生错误的位置，该文件可能已经打开。恢复函数需要检查此情况，以及其他可能导致系统不稳定的情况，并使程序返回到一个一致的状态。

处理恢复点最简单的方法是将整个程序封装到单个恢复点。这样，你只需要一个简单的错误报告函数，通过传递错误代码和消息即可调用。该函数将打印错误代码和消息，并退出程序。这通常不是现实情况下的最佳解决方案，但却是一个很好的回退方式，是在迫不得已的情况下采用的机制。

7.4 让程序更健壮

本节讲述如何使第6章的add-year.s程序更健壮。

由于这是一个非常简单的程序，我们仅限于对整个程序使用单个恢复点。对于恢复，我们唯一要做的就是打印错误信息并退出。完成这项任务的代码非常简单：

```
 .include "linux.s"
 .equ ST_ERROR_CODE, 8
 .equ ST_ERROR_MSG, 12
 .globl error_exit
 .type  error_exit, @function
error_exit:
 pushl %ebp
 movl  %esp, %ebp

 # 写错误代码
 movl  ST_ERROR_CODE(%ebp), %ecx
 pushl %ecx
 call  count_chars
 popl  %ecx
 movl  %eax, %edx
 movl  $STDERR, %ebx
 movl  $SYS_WRITE, %eax
 int   $LINUX_SYSCALL

 # 写错误信息
 movl  ST_ERROR_MSG(%ebp), %ecx
 pushl %ecx
 call  count_chars
 popl  %ecx
 movl  %eax, %edx
 movl  $STDERR, %ebx
 movl  $SYS_WRITE, %eax
 int   $LINUX_SYSCALL

 pushl $STDERR
```

```
call write_newline

# 退出，状态码为1
movl $SYS_EXIT, %eax
movl $1, %ebx
int  $LINUX_SYSCALL
```

请将上述代码输入名为error-exit.s的文件。要调用该函数，你必须将错误信息的地址以及错误代码入栈，然后再进行调用。

现在，让我们查找add-year程序中可能出错的地方。首先，我们并未检查open系统调用实际上是否正确完成。Linux将其状态码返回到%eax中，因此我们需要检查是否出错了：

```
# 打开供读取的文件
movl $SYS_OPEN, %eax
movl $input_file_name, %ebx
movl $0, %ecx
movl $0666, %edx
int  $LINUX_SYSCALL

movl %eax, INPUT_DESCRIPTOR(%ebp)

# 下面将检测%eax是否为负值
# 如果是非负值，那么程序将继续处理，
# 否则将处理负数对应的错误情况
cmpl $0, %eax
jl   continue_processing

# 发送错误信息
.section .data
no_open_file_code:
.ascii "0001: \0"
no_open_file_msg:
.ascii "Can't Open Input File\0"

.section .text
pushl $no_open_file_msg
pushl $no_open_file_code
call  error_exit

continue_processing:
# 程序其余部分
```

这样，在系统调用后，我们通过检测系统调用的结果是否小于0来查看是否存在错误。如果

存在错误，我们就调用错误报告并退出例程。

在每个系统调用、函数调用或可能导致错误的指令后，你都应该增加错误检测和处理代码。

通过以下指令汇编和链接文件：

```
as add-year.s -o add-year.o
as error-exit.s -o error-exit.o
ld add-year.o write-newline.o error-exit.o read-record.o writerecord.o count-chars.o
    -o add-year
```

现在尝试在缺少必要文件的情况下运行程序。可以看到，程序能干净优雅地退出了。

7.5 温故知新

7.5.1 理解概念

- 程序员不擅安排时间的原因是什么？
- 找到你最喜欢的程序，尝试以完全错误的方式使用它。比如打开错误类型的文件，选择无效选项，关闭应该打开的窗体等。计算应考虑多少种不同的错误状况。
- 什么是极端案例？你能说出一些数值型极端案例吗？
- 为什么用户测试非常重要？
- 存根和驱动有什么作用？这两者的区别是什么？
- 恢复点的作用是什么？
- 一个程序应该有多少不同的错误代码？

7.5.2 应用概念

- 通读add-year.s程序并在每个系统调用之后增加错误检测代码。
- 在到目前为止我们编好的程序中另外选一个，然后为其增加错误检测代码。
- 为add-year.s程序增加恢复机制，使之在无法打开标准文件时能从STDIN读取数据。

7.5.3 深入学习

- 如果错误报告函数失败，你应该怎么做？为什么？
- 尝试在至少一个开源程序中查找错误。找到一个错误后请尝试报告该错误。
- 尝试修正你在上一个练习中找到的错误。

第8章 与代码库共享程序

现在你应该已经意识到：即使是简单的任务，计算机也要做许多工作。因此，即使是编写代码让计算机完成简单的任务，你也必须完成大量工作。而编程任务通常并不简单，因此我们应设法简化这一过程。为此可采用以下方式：

- 以高级语言而非汇编语言编写代码；
- 保存大量预先编写好、可复制到自己程序中的代码；
- 具有任何程序若有需要即可使用的大量系统函数。

所有项目或多或少都会用到以上三种方式。第一种方式将在第11章中进一步探讨。第二种方式虽有用但却有以下缺陷：

- 被复制的代码往往必须经过大量改动才能适用；
- 包含被复制代码的每个程序中都有同样的代码，因此会浪费大量空间；
- 如果在任何被复制代码中发现错误，就必须在每个使用它的程序中进行修正。

因此，第二种方式一般不常用，仅仅用于为某一特定类型的任务复制和粘贴骨架代码、增加特定程序的细节。第三种方式最常用，它要求包括一个含有共享代码的中央存储库。这样一来，程序只需指向其所需函数的共享库，而无需让每个程序都浪费空间存储同样函数的副本。如果在某个函数中发现错误，你也只需在一个函数库文件中修正它，而所有使用该函数的应用程序都会自动更新。这种方式主要的缺点在于会产生某些依赖问题，如下。

- 如果多个应用程序都使用某个共享文件，那么我们如何知道何时可以安全删除该文件呢？例如，如果3个应用程序共享一个包含多个函数的文件，而其中两个程序已经被删除，那么系统如何知道仍然存在一个使用该代码的应用程序，该函数文件不应被删除呢？
- 一些程序无可避免地依赖于共享函数中的错误。因此，如果更新共享程序时修正了某个程序依赖的错误，就可能会导致该应用程序无法发挥其功能。

上述这些问题会导致称为DLL地狱的问题。但通常人们认为这种方式的优点大于缺点。

在编程中，这些共享代码文件称为共享库、共享对象、动态链接库、DLL文件等。在这里，我们称其为共享库。

8.1 使用共享库

这里我们要看的程序很简单，它只是将hello world输出到屏幕并退出。常用程序helloworld-nolib.s如下所示：

```
# 目的：此程序打印消息"hello world"并退出
#

 .include "linux.s"
 .section .data

helloworld:
 .ascii "hello world\n"
helloworld_end:

 .equ helloworld_len, helloworld_end - helloworld

 .section .text
 .globl _start
_start:
 movl $STDOUT, %ebx
 movl $helloworld, %ecx
 movl $helloworld_len, %edx
 movl $SYS_WRITE, %eax
 int  $LINUX_SYSCALL

 movl $0, %ebx
 movl $SYS_EXIT, %eax
 int  $LINUX_SYSCALL
```

本程序并不长。然而，看一下使用库的helloworld-lib有多短：

```
# 目的：此程序打印消息"hello world"并退出
#

 .section .data

helloworld:
 .ascii "hello world\n\0"
```

```
 .section .text
 .globl _start
_start:
 pushl $helloworld
 call printf

 pushl $0
 call exit
```

可以看到，这个程序更短！

生成使用共享库的程序与通常做法略有不同。你一般可以通过以下命令生成第一个程序：

```
as helloworld-nolib.s -o helloworld-nolib.o
ld helloworld-nolib.o -o helloworld-nolib
```

但是，为了生成第二个程序，必须运行以下命令：

```
as helloworld-lib.s -o helloworld-lib.o
ld -dynamic-linker /lib/ld-linux.so.2 \
   -o helloworld-lib helloworld-lib.o -lc
```

记住，第一行中的斜杠只是意味着命令在下一行继续。选项`-dynamic-linker/lib/ld-linux.so.2`使我们的程序能链接到库，这样可执行文件会在执行前生成，而操作系统将加载程序`/lib/ld-linux.so.2`，以加载外部库并将其链接到程序。这种程序称为*动态链接器*。

选项`-lc`表示：链接库c。该库在GNU/Linux系统上的文件名为libc.so。给定库名，在本例中为c（通常库名不止一个字符），GNU/Linux链接器将字符串`lib`加至库名之前，将字符串`.so`加到库名之后，构成库文件名。这个库包含许多函数以自动执行各种任务。我们使用其中两个，即打印字符串的`printf`和退出程序的`exit`。

注意，程序中只是通过名字指代符号`printf`和`exit`。在前面几章中，链接器将在解析所有名字为物理地址后弃用名字。当使用动态链接时，名字本身存在于可执行文件中，在运行时才由动态链接器解析。当用户运行程序时，动态链接器加载我们链接声明中列出的共享库，并找出我们程序中命名但未在链接时找到的所有函数名和变量名，将之与加载的共享库中的相应入口相匹配，最后以函数名变量名的加载地址代替相应名字。这听起来很费时间，事实上也确实要花费一定时间，但这只在程序启动时发生一次。

8.2　共享库的工作原理

在我们的第一批程序中，所有代码都包含在源文件中。这样的程序称为*静态链接可执行文件*，

因为它们包含所有程序所必需的，但内核未处理的功能。在第6章所写的程序中，我们既使用了主程序文件，也使用了包含供多个程序共用的例程的文件。在上述情况下，我们在链接时使用链接器合并所有代码，因此此时仍然属于静态链接。然而，在helloworld-lib程序中，我们开始使用共享库。当使用共享库时，程序就是动态链接的，这意味着并非运行程序所需的所有代码都实际包含在程序文件本身中，而是在外部库中。

当我们在命令中使用-lc来链接helloworld程序时，该选项告诉链接器使用库c（即libc.so）来寻找未在helloworld.o中定义过的符号。但实际上这并未增加任何代码到我们的程序，只是在程序中说明到哪里寻找。程序helloworld开始时，首先加载文件/lib/ld-linux.so.2，这是动态链接器，会查看helloworld程序，并发现该程序需要库c才能运行。因此，链接器在标准目录（即/etc/ld.so.conf下以及环境变量LD_LIBRARY_PATH中的所有目录下）查找名为libc.so的库，然后在库中查找所有所需符号（本例中为printf和exit），并加载库到程序的虚拟内存。最后，链接器以库中printf的实际位置代替程序中printf的所有实例。

运行以下命令：

```
ldd ./helloworld-nolib
```

该命令应该会报告not a dynamic executable（并非动态可执行文件）。这就好像我们说helloworld-nolib是静态链接可执行程序。但，请尝试使用以下命令：

```
ldd ./helloworld-lib
```

这次会报告如下信息：

```
        libc.so.6 => /lib/libc.so.6 (0x4001d000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x400000000)
```

在你的系统上，括号中的数字可能有所不同。这意味着程序helloworld链接到libc.so.6（.6为版本号），找到此文件的位置为/lib/libc.so.6，另外还找到/lib/ld-linux.so.2文件。这些库必须在程序运行前加载。如果你有兴趣，请在Linux版本上对不同的程序运行ldd程序，查看程序分别依赖于哪些库。

8.3 查找关于库的信息

现在你已经了解了库，但如何知道自己的系统上有哪些库，而这些库又有什么作用呢？让我们暂时跳过这个问题，先看另一个：程序员如何在文档中向他人描述函数？先看一下函数printf，其调用接口（通常称为原型）如下所示：

```
int printf(char *string, ...);
```

在Linux中，函数是以C编程语言描述的。事实上，多数Linux程序都是用C编写的。这就是为什么多数文档和二进制兼容是用C语言定义的。上述printf函数的接口是用C编程语言描述的。

此定义表示存在一个名为printf的函数。括号内的内容是函数参数。这里的第一个参数是char *string，表明有一名为string的参数（名字并不重要，只是我们说明的时候要用到），数据类型为char *，char表示单字节字符，之后的*表示函数想要的参数不是一个字符，而是一个或一串字符的地址。如果复习一下helloworld程序，你会发现该函数调用看起来如下所示：

```
pushl $hello
call printf
```

因此，我们将hello字符串的地址（而非实际字符串）入栈。你可能注意到我们并未将字符串的长度入栈。由于我们以空字符\0结束字符串，因此printf能找到字符串结束处。许多函数都采取这种方式，特别是C语言函数。在函数定义之前的int说明函数将返回到%eax中的值类型。printf将在结束时返回一个int。现在，在char *string之后是省略号，表示可以在参数string后有数量不定的参数。多数函数只有指定数量的参数。但printf可有许多参数，它会查看string参数，一旦看到字符串%s，就从栈中寻找另一个字符串以执行插入操作，而每次发现%d，就从栈中寻找一个数字插入。我们最好用一个示例来说明这点。

```
# 目的：这个程序用于说明如何调用printf
#

 .section .data

# 这个字符串称为格式字符串，是第一个参数，
# printf用这个参数来确定给定了多少个参数，
# 以及它们分别是什么类型
firststring:
 .ascii "Hello! %s is a %s who loves the number %d\n\0"
name:
 .ascii "Jonathan\0"
personstring:
 .ascii "person\0"
 # 这也可以用.equ，但为了有趣，我们决定给其一个实际内存位置
numberloved:
 .long 3
```

```
 .section .text
 .globl _start
_start:
 # 注意，参数传递顺序与函数原型中列出的顺序相反
 pushl numberloved    # 这是%d
 pushl $personstring # 这是第二个%s
 pushl $name          # 这是第一个%s
 pushl $firststring  # 这是原型中的格式字符串
 call  printf
 pushl $0
 call  exit
```

将上述代码输入名为printf-example.s的文件，然后运行以下命令：

```
as printf-example.s -o printf-example.o
ld printf-example.o -o printf-example -lc \
   -dynamic-linker /lib/ld-linux.so.2
```

接着输入命令./printf-example以运行程序，然后程序便会显示：

```
Hello! Jonathan is a person who loves the number 3
```

现在，如果你看一看代码就会发现，虽然格式字符串是第一个参数，但实际上最后才将其入栈。注意，我们应该总是以逆序将函数参数入栈。[①]你可能会想：为什么printf函数知道一共有多少参数？该函数会在字符串中搜索，计算一共找到多少%d和%s，接着从栈中抓取相应数目的参数。如果参数匹配%d，函数就将其作为数字处理，如果匹配%s，就将其作为指向以空字符结束的字符串的指针处理。除此之外，printf还有更多的功能，但这些是最常用的。正如你所见，printf能大大简化输出，但也会造成大量开销，因为它必须计数格式字符串中的字符数目，搜索整个字符串以寻找需要替换的控制字符，将相应数据从栈中弹出，转换成合适的表达形式（例如，数字必须转换成字符串），并将所有数据相结合。

我们已经看到了如何用C编程语言的原型来调用库函数。但为了有效使用它们，你需要再多了解几种读取函数可能需要的数据类型。我们主要来看以下几种。

- int

int型数据是指整数（在x86处理器上占据4字节）。

- long

① 参数以逆序入栈的原因是存在如printf这样接受可变数量参数的函数。最后入栈的参数与栈顶的相对位置已知。此后，程序就能使用这些参数决定其余的参数在栈中的位置，以及它们是什么类型的。例如，printf使用格式字符串来确定传入的其他参数共有多少。如果我们先将已知参数入栈，你就无法得知它们在栈中的位置。

long型数据也是指整数（在x86处理器上占据4字节）。

- long long

long long型数据是指比整数大的long型数据（在x86处理器上占据8字节）。

- short

short型数据是指比int型短的整数（在x86处理器上占据2字节）。

- char

char型数据是单字节整数，最常用于存储字符数据，因为ASCII字符串通常一个字符用一字节来表示。

- float

float型数据是浮点数（x86处理器上为4字节）。浮点数将在10.4.1中详细阐述。

- double

double型数据是比float型数据大的浮点型数据（x86处理器上为8字节）。

- unsigned

unsigned可用作上述类型的修饰符，使之成为无符号数，与有符号数相区别。有符号数与无符号数之间的区别将在第10章讨论。

- *

星号（*）用于表明数据并非实际值，而是指向保存给定值位置的指针（在x86处理器上为4字节）。现在，让我们假设数字20存储在存储位置my_location上，如果原型表明要传递一个int型数据，你就需要采用直接寻址方式，发出指令pushl my_location，而如果函数原型表示传递int *，就需要用立即寻址方式运行pushl $my_location，将int型值的地址入栈。除了表明某个值的地址，指针也可用于传递一系列连续的位置，从给定值指定的位置开始。这便是数组。

- struct

struct是以某个名字表示的一系列数据项。例如，你可以定义如下结构：

```
struct teststruct {
 int a;
```

```
 char *b;
};
```

每次遇到struct teststruct，你就知道这实际上是相连的两个字，第一个是一个整数，第二个是指向一个或一组字符的指针。结构不会被作为参数传递给函数，而指向结构的指针会被作为参数传递。这是因为传递结构给函数过于复杂，因为指针会占据许多存储位置。

- typedef

typedef主要用于对某种类型重命名。例如，我可以在C程序中执行typedef int myowntype;，这样每当我输入myowntype，其效果就相当于输入int。这样做的麻烦之处在于不得不查找某个函数中所有typedef和struct的原型。但是，typedef通常用于赋予某些类型更有意义、更具描述性的名字。

兼容性注解 上述所列类型大小仅用于intel（x86）兼容机。面向其他机器的大小会有所不同。此外，即使所传递参数的大小不足一个字，它仍会在栈上被作为长整型传递。

以上是关于如何阅读函数的文档。现在，让我们回到前面关于库的问题。你的多数系统库都在/usr/lib或/lib目录下。如果你只是想查看到库定义了什么符号，只需运行objdump -R FILENAME，其中FILENAME是库的完整路径。但如果要找到你需要的某个接口，这条命令的输出并无多大帮助。通常，你首先要知道想要什么库，然后阅读相应文档。大多数库都有用户手册。网站一般是库文档的最佳来源。GNU项目中的大多数库都有信息页，且提供的信息比用户手册更详尽一点。

8.4 一些有用的函数

你可能需要了解的几个有用的c库函数如下所示。

- size_t strlen (const char *s)计算以空字符结束的字符串的长度。
- int strcmp (const char *s1, const char *s2)按照字母顺序比较两个字符串。
- char * strdup (const char *s)接受一个指向字符串的指针（传入参数），并在新位置复制该字符串，并返回新位置。
- FILE[①] * fopen (const char *filename, const char *opentype)打开一个托管缓冲文件（使读写比直接用文件描述符时更方便）。[②]

① FILE是一个结构体，你无需了解其内容就可使用，只需存贮其指针并传递它给其他相关函数。

② stdin、stdout以及stderr（都是小写字符）可用在这些程序中指代相应文件的文件描述符。

- int fclose (FILE *stream)关闭通过fopen打开的文件。
- char * fgets (char *s, int count, FILE *stream) 从文件中提取一行字符到字符串s。
- int fputs (const char *s, FILE *stream)写入一个字符串到打开的特定文件。
- int fprintf (FILE *stream, const char *template, ...)与printf类似，但使用一个打开的文件，而非默认使用标准输出。

你可以在http://www.gnu.org/software/libc/manual/找到关于此库的完整用户手册。

8.5　构建一个共享库

假设我们想用第6章中的所有共享代码生成一个共享库，以便在程序中使用。首先要像往常一样进行汇编：

```
as write-record.s -o write-record.o
as read-record.s -o read-record.o
```

接下来，将它们链接为一个共享库，而不是链接为一个程序。为此，链接器命令应改变如下：

```
ld -shared write-record.o read-record.o -o librecord.so
```

这条命令将这两个文件链接为一个名为librecord.so的共享库。现在，这个文件可用于多个程序。如果我们需要更新它包含的函数，只需更新这个文件，而不必担心那些使用它的程序。

让我们来看看如何链接到这个库。为了链接write-records程序，我们要执行以下命令：

```
as write-records.s -o write-records
ld -L . -dynamic-linker /lib/ld-linux.so.2 \
   -o write-records -lrecord write-records.o
```

在此命令中，-L告诉链接器在当前目录下（通常只搜索/lib、/usr/lib和其他几个目录）寻找库。正如我们所见，选项-dynamic-linker /lib/ld-linux.so.2指定了动态链接器。选项lrecord告诉链接器在librecord.so文件中搜索函数。

现在write-records程序已经生成了，但却无法运行。如果我们尝试运行它，会得到与下面类似的错误：

```
./write-records: 当加载共享库时出错:
```

```
librecord.so: 无法打开共享目标文件: 无此文件或目录
```

这是因为动态连接器默认情况下只搜索/lib、/usr/lib和在/etc/ld.so.conf中列出的目录下搜索库。为了运行此程序，你需要将库移动到其中某个目录下，或执行以下命令：

```
LD_LIBRARY_PATH=.
export LD_LIBRARY_PATH
```

如果以上命令出错的话，你可以运行以下命令：

```
setenv LD_LIBRARY_PATH
```

现在，你可以通过正常输入./write-records来运行write-records。设置LD_LIBRARY_PATH告诉链接器将你给定的所有路径增加到动态库搜索路径。

要进一步了解动态链接，请在网上查阅下列资料：

- ld.so的用户手册页包含了大量关于Linux动态链接器工作原理的信息；
- http://www.benyossef.com/presentations/dlink/就Linux的动态链接给出了很好的介绍；
- http://www.linuxjournal.com/article.php?sid=1059 和 http://www.linuxjournal.com/article.php?sid=1060 提供了ELF文件格式的介绍，更多信息也可参见 http://www.cs.ucdavis.edu/~haungs/paper/node10.html；
- http://www.iecc.com/linker/linker10.html以ELF文件格式很好地描述了动态链接。

8.6 温故知新

8.6.1 理解概念

- 共享库的优点和缺点是什么？
- 若一个库名为foo，那么该库的文件名是什么？
- ldd命令的作用是什么？
- 假设有文件foo.o和bar.o，如果你想将它们链接在一起，并将其动态链接到库kramer。那么，生成最终可执行文件的链接命令是什么？
- 什么是typedef？
- 什么是struct？
- 类型为int或int *的数据元素其区别是什么？如何在程序中通过不同方式访问它们？
- 如果你有一个目标文件名为foo.o，用什么命令可以创建一个名为bar的共享库？
- LD_LIBRARY_PATH的作用是什么？

8.6.2　应用概念

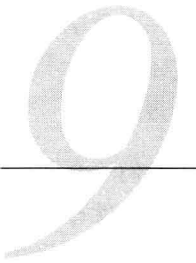
- 重写之前各章中的一个或多个程序，用printf将结果打印到屏幕，而不是将结果作为退出状态码返回。此外，设置退出状态码为0。
- 将你在4.5节中开发的factorial函数生成为共享库。然后请重新编写主程序，使之与库动态链接。
- 重写上面的程序，使之也动态链接到c库。使用c库的printf函数显示调用factorial函数的结果。
- 重写toupper程序，以使该程序将c库函数而非系统调用用于文件。

8.6.3　深入学习

- 试给出GNU/Linux动态链接器使用的所有环境变量的列表。
- 搜索目前和计算机历史上使用的不同类型的可执行文件格式，说明每种类型的长处和短处。
- 你对哪种类型的编程（图形、数据库、科学等）感兴趣？找出一个用于该领域的库，并编写一个对该库进行基本应用的程序。
- 研究如何使用LD_PRELOAD。它有何用途？试建立一个包含exit函数的共享库，并使程序在退出前写一条消息到STDERR。使用LD_PRELOAD运行各种程序，然后观察其结果是什么。

第9章

关于中间存储器



9.1 计算机如何看待内存

让我们回顾一下内存在计算机中的工作原理。你可能还需要重新阅读第2章。

计算机将内存看做一长列带编号的存储位置，一列数百万经编号的存储位置。所有一切都存储在这些位置中：程序存储在那里，数据存储在那里，一切都存储在那里。每个存储位置看起来都彼此相似，存储程序的位置与存储数据的位置并无区别。事实上，计算机不知道哪些是程序，哪些是数据，除非可执行文件告诉它从哪里开始执行。

这些存储位置称为字节。计算机最多可以将四字节组成一个字。通常，计算机一次操作一个字大小的数值数据。正如我们提到过的，指令也存储在同样的内存中。每个指令的长度是不同的，多数指令本身占据一两个存储位置，然后其参数占据一两个存储位置。例如如下指令：

```
movl data_items(,%edi,4), %ebx
```

占据了7个存储位置。前两个保存指令，第三个说明使用哪个寄存器，接下来的4个则是data_items的存储位置。在内存中，指令看起来与其他数字并无不同，指令本身也能像数字一样移入或移出寄存器，因为指令本身就是数字。

本章重点关注计算机内存的细节。在开始前，先来回顾一下我们将在本章中使用的一些基本术语。

- 字节

这是存储位置的大小。在x86处理器上，一个字节可以容纳介于0和255之间的数字。

- 字

这是一个一般寄存器的大小。在x86处理器上，一个字是4字节。大多数计算机操作一次处理一个字。

- 地址

地址是一个数字，表示内存中的一个字节。例如，计算机中的第一个字节地址为0，第二个地址为1，以此类推。[①]计算机中的每一项不在寄存器中的数据都有一个地址。占据多个字节的数据，其地址与数据第一个字节的地址相同。

通常，我们不会键入任何数字地址，而是让汇编程序来做这件事。当在代码中使用标签时，用作标签的符号就相当于其标识的地址。每当你在程序中使用该符号时，汇编程序将以相应地址替换该符号。例如，假设有以下代码：

```
 .section .data
my_data:
 .long 2, 3, 4
```

现在，每当在程序中使用my_data时，该符号就会被替换为.long指令第一个值的地址。

- 指针

指针是一个寄存器或内存字，其值是一个地址。在我们的程序中，%ebp被用作当前栈帧的指针。所有基址寻址方式都涉及指针。编程要使用大量指针，所以这是一个必须掌握的重要概念。

9.2　Linux 程序的内存布局

当程序被加载到内存时，每个.section都被加载到其自己的内存区。在每个段中声明的代码和数据将被组合起来，即使它们在源代码中是分离的。

实际指令（.text段）被加载到地址0x08048000（以0x开头的为十六进制数字，参见第10章），而.data段紧随其后被加载，再之后是.bss段。

在Linux上可以寻址的最后一个字节是存储位置0xBFFFFFFF。Linux的栈从这里开始，并向下朝其他段增长。但栈与其他段之间的距离很大。栈的初始布局如下：栈底（栈底为内存最顶端的地址，参见第4章），存在一内存字0。接着是ASCII字符表示、以空字符结束的程序名。程序名后是程序的环境变量（在本书中，这些对于我们都不重要）。接着是程序的命令行参数，是用户为了运行此程序在命令行键入的值。例如，当我们运行as时，会给出几个参数：as、sourcefile.s、-o以及objectfile.o。在这之后就是用到的参数数量。程序开始时，此处就是栈指针%esp指向的位置。数据进一步入栈会使%esp在内存中向下移动。例如，指令：

① 实际上你从来不会用到这么小的地址，但在讨论中它还是能说明问题的。

```
pushl %eax
```

相当于：

```
movl %eax, (%esp)
subl $4, %esp
```

同样，指令：

```
popl %eax
```

相当于：

```
movl (%esp), %eax
addl $4, %esp
```

程序的数据段从内存的底部开始，并向上增长。栈从内存的顶部开始，每次数据入栈都向下移动。这个栈和程序数据段之间的中间部分是你无法访问的，除非告诉内核你需要这样做。[①]如果你尝试这样做就会发生错误（错误信息通常是“分割故障“）。如果你尝试访问程序起始地址0x08048000之前的数据，会得到同样的错误。对于你的程序，其最后的可访问内存地址称为*系统中断*（也称为*当前中断*或*中断*）。

一个Linux程序启动时的内存布局

① 当栈向下增长时可以访问此部分，此时你可以通过%esp访问栈区域。然而，程序的数据段不会这样增长，其增长方式稍后你就会看到。

9.3　每个内存地址都是虚拟的

那么，为什么计算机不让你访问中断区的内存呢？要回答这个问题，我们需深入分析计算机实际上是如何处理内存的。

你可能会想：既然每个程序都被加载到内存中的相同位置，难道它们不会彼此妨碍或覆盖吗？看起来好像是这样，但作为程序编写者，你只能访问*虚拟内存*。

*物理内存*是指计算机中的实际RAM芯片及其包含的内容。现代计算机中其大小通常在16 M至512 M之间。如果我们谈及*物理内存地址*，指代的正是某一段内存在芯片上的实际位置。虚拟内存是*你的程序*看待内存的方式。Linux在加载你的程序之前，先找到一块适合你的程序、足够大的物理内存空间，然后告诉处理器假装这段内存位于内存地址0x0804800，以便将程序载入。是否感到很困惑？让我进一步解释下。

每个程序都在自己的沙盒内运行。在计算机上运行的每个程序都认为其被加载到内存地址0x0804800，其栈起始地址为0xbfffffff。当Linux加载程序时，它先找到一个未使用的内存段，然后告诉处理器将该段内存当做地址0x0804800用于该程序。程序认为自己在使用的地址称为虚拟地址，而其在芯片上实际对应的地址称为物理地址。将物理地址分配到虚拟地址的过程称为映射。

此前我们谈到了在.bss和栈之间无法访问的内存，但没有谈到为什么那里的内存无法访问。其原因是，此区域的虚拟内存地址还没有映射到物理内存地址。映射过程需要大量的时间和空间，所以如果每一个可能程序可能的虚拟地址都映射到物理地址，那么连运行一个程序的物理内存可能都不会剩下。所以，中断是未映射内存区的起始处。但有了栈，Linux将自动映射通过入栈可访问的内存。

当然，这是个关于虚拟内存的简单观点，其完整的概念高级很多。例如，虚拟内存可映射的不只是物理内存，还可以映射到磁盘。Linux上的交换分区，让Linux的虚拟内存系统不仅能将内存映射到物理RAM，还能映射到磁盘块。例如，假设物理内存只有16 M，Linux和一些基本的应用程序正在使用其中8 M，而你想运行一个需要20 M内存的程序。你能这样做吗？答案是肯定的，但前提是你已经设置了一个交换分区。当剩余的8 M物理内存都映射到虚拟内存时，Linux开始将一部分应用程序的虚拟内存映射到磁盘块。所以，如果你访问程序的某个“内存”位置，该位置可能实际上不在内存中，而在磁盘上。不过，作为程序员你不会察觉两者有何区别，因为这一切都由Linux在幕后操纵。

现在，x86处理器既不能直接从磁盘运行指令，也不能直接访问磁盘数据。要访问数据，这需要操作系统的帮助。当你尝试访问映射到磁盘的内存时，处理器发现不能直接为你的内存请求

提供服务，然后它会要求Linux介入。Linux发现内存实际上在磁盘上，因此会将某些目前在内存中的数据移动到磁盘上以腾出空间，然后将从磁盘访问的内存移回物理内存。接着，它调整处理器的“虚拟到物理”内存查找表，使之能在新的位置找到该内存数据。最后，Linux将控制权还给程序，并在程序最初试图访问数据的指令处重新启动程序。现在，这条指令能成功完成了，因为内存现在在物理RAM中。[①]

下面概述了Linux中的内存访问处理方式。

- 程序尝试从虚拟地址加载内存。
- 处理器，使用Linux提供的表，在程序运行时将虚拟内存地址转换成物理内存地址。
- 如果处理器发现所列的物理地址不是内存地址，它向Linux发送请求以加载该地址。
- Linux查找该地址。如果该地址映射到一个磁盘位置，Linux执行下一个步骤，否则就终止程序，报告分段错误。
- 如果没有足够的空间可从磁盘装入内存，Linux会将程序的其他某部分或另一个程序移到磁盘上，以腾出空间。
- 接着，Linux会将数据移动到空闲的物理内存地址。
- Linux更新处理器的“虚拟到物理”内存映射表，以反映上述更改。
- Linux恢复程序的控制权，使其重发导致这一过程发生的那条指令。
- 现在，该处理器能使用新装入内存和翻译表来处理该指令了。

对于操作系统来说上述工作量真的不小，但却在内存管理方面给用户和程序员带来了很大的灵活性。

现在，为了使这一过程更有效率，内存被拆分成组，这些组称为页。当在x86处理器上运行Linux时，一个页面是4096字节内存。所有的内存映射都一次映射一个页面。物理内存的分配、交换、映射等都以内存页为单位，而非以单个内存地址为单位进行。这意味着你在编程时都应该尽量在同一基本内存范围内进行内存访问，这样一次只需要一两个内存页。否则，Linux为了满足你的内存需求，可能不得不随时将页面移到磁盘或从磁盘移入内存。磁盘访问速度很慢，所以这样会减慢你的程序运行速度。

有时可能会有过多程序被同时加载，导致物理内存不足。这些程序不得不将更多时间花费在与磁盘交换内存上，而非用于实际程序处理，这会导致称为交换死亡（swap death）的情况，从而导致系统无响应且低效。通常只要开始终止耗用大量内存的程序，这种情况就会得到解决，但这确实让人烦恼。

① 注意，Linux不仅能让虚拟地址映射到不同的物理地址，而且能按需移动映射。

驻留集大小　程序当前在物理内存中占据的内存量称为驻留集大小（resident set size），可以通过使用该程序top查看。驻留集大小列于名为“RSS”的列下。

9.4　获取更多的内存

我们现在知道Linux将所有的虚拟内存都映射或交换为物理内存。如果你尝试访问一块尚未被映射的虚拟内存，就会触发分段错误，这将终止你的程序。也许你还记得，程序中断点是你可以使用的最后一个有效地址。现在，如果你事先知道需要多少存储量的话就太好了。你只需将需要的所有内存添加到.data或.bss段，这些内存将一直保持在那里。然而，假设你不知道会需要多少内存。例如，用文本编辑器时，你不知道用户将输入多长的文件。你可以确定一个最大文件大小，并将此限制告知用户。但如果文件很小，这样做就很浪费空间。因此，Linux支持根据应用程序的内存需求移动中断点。

如果需要更多的内存，可以告诉Linux你想要新的中断点，而Linux会将你需要的所有内存映射到现有断点与新断点之间，然后将断点移动到你指定的地方。现在程序就可以使用这段内存了。我们是通过brk系统调用告诉Linux移动中断点的。brk的系统调用号为45（这个数字将存储在%eax中）。所请求的中断点应加载到%ebx中。然后，你调用int $ 0x80发信号给Linux让其完成相应工作。在映射内存后，Linux将新的中断点返回到%eax。新的中断点实际上可能比你要求的大，因为Linux会四舍五入至最接近的页。如果没有足够的物理内存或页面交换能满足你的要求，Linux将返回0到%eax。另外，如果调用brk时%ebx为0，该调用仅仅返回最后一个可用内存地址。

这种方法的问题是必须跟踪记录我们所请求的内存。比方说，我需要移动中断点以便有足够空间来加载一个文件，然后需要再次移动中断点以加载另一个文件。假如此时我再删除第一个文件，那么内存映射中就有一大块未用空间。如果每加载一个文件，你都以这种方式移动断点，内存很快就会耗尽。所以，我们需要一个*内存管理器*。

内存管理器是一组例程，负责完成为你的程序获取内存的累活。大多数内存管理器都有两个基本功能——分配（allocate）和回收（deallocate）。[①]每当需要一定量的内存，你只需告诉allocate需要多少内存，它会返回给你一个内存地址。使用完后，你再告诉deallocate已经用完了。接下来，allocate将能够重复使用该内存。这种内存管理方式称为*动态内存分配*，能最大限度地减少内存中的“孔洞”，确保最佳地使用内存。内存管理器所使用的内存池通常称为*堆*。

① 函数名不一定都用allocate和deallocate，但功能相同的。例如，在C编程语言中，相应功能的函数名为malloc和free。

内存管理器的工作方式是记录系统中断以及你已经分配的内存位置。它们将堆中的每块内存都标记为“正在使用”或“未使用”。当你请求内存时，内存管理器查看是否有适当大小的未使用块。如果没有，它就发起brk系统调用请求更多内存。当你释放内存时，内存管理器将该内存块标记为未使用，以便将来可以检索它。在下一节中，我们来看如何建立自己的内存管理器。

9.5 一个简单的内存管理器

在这里，我会展示一个简单的内存管理器。虽然非常原始，但它很好地展示了其原理。像往常一样，先来看一下程序，然后我们再一起深入分析。它看起来很长，但其中大部分内容都是注释。

```
# 目的：用于管理内存使用的程序——按需
#       分配和释放内存
#
# 注意：使用这些例程的程序将要求一定大小的内存。在实际操作
#       中，我们使用的内存更大，但在回传指针前将之放在开始处。
#       我们增加一个大小字段，以及一个AVAILABLE/UNAVAILABLE标记。
#       因此，内存看起来如下所示
#
# ################################################
# #AVAILABLE标记#内存大小#实际内存位置#
# ################################################
#                                          ^--返回指针指向此处
#       为了方便调用程序，返回的指针仅仅指向所请求的实际内存位置
#       这也让我们无需更改调用程序即可更改结构

.section .data
#######全局变量#########

# 此处指向我们管理的内存的起始处
heap_begin:
 .long 0

# 此处指向我们管理的内存之后的一个内存位置
current_break:
 .long 0

######结构信息####
 # 内存区头空间大小
 .equ HEADER_SIZE, 8
 # 头中AVAILABLE标志的位置
 .equ HDR_AVAIL_OFFSET, 0
 # 内存区头中大小字段的位置
```

```
 .equ HDR_SIZE_OFFSET, 4

############常量############
 .equ UNAVAILABLE, 0 # 这是用于标记已分配空间的数字
 .equ AVAILABLE, 1   # 这是用于标记已回收空间的数字，此类空间可用于再分配
 .equ SYS_BRK, 45    # 用于中断系统调用的系统调用号

 .equ LINUX_SYSCALL, 0x80 # 使系统调用号更易读

 .section .text

###########函数#############

 ##allocate_init##
 # 目的：调用此函数来初始化函数（更具体地说，此函数设置
 #       heap_begin和current_break）。此函数无参数和返回值
 .globl allocate_init
 .type allocate_init,@function
allocate_init:
 pushl %ebp                    # 标准函数处理
 movl  %esp, %ebp

 # 如果发起brk系统调用时，%ebx内容为0，该调用将返回最后一个有效可用地址
 movl $SYS_BRK, %eax           # 确定中断点
 movl $0, %ebx
 int  $LINUX_SYSCALL

 incl %eax                     # %eax现为最后有效可用地址，
                               # 我们需要此地址之后的内存位置

 movl %eax, current_break      # 保存当前中断

 movl %eax, heap_begin         # 将当前中断保存为我们的首地址。这会使
                               # 分配函数在其首次运行时从Linux获取更多内存
 movl %ebp, %esp               # 退出函数
 popl %ebp
 ret
#####函数结束#######

 ##allocate##
 # 目的：此函数用于获取一段内存。它查看是否存在自由内存块，
 #       如不存在，则向Linux请求
 #
 # 参数：此函数有一个参数，就是我们要求的内存块大小
 #
 # 返回值：
 #       此函数将所分配内存的地址返回到%eax中。如果已无可用内存，
```

```
 #         就返回0到%eax
 #
 ######处理########
 # 用到的变量:
 #
 # %ecx - 保存所请求内存的大小（这是第一个也是唯一的参数）
 # %eax - 检测的当前内存区
 # %ebx - 当前中断位置
 # %edx - 当前内存区大小
 #
 # 我们检测每个以heap_begin开始的内存区，查看每一个的大小以及是否已经分配
 # 如果某个内存区大于等于所请求的大小，且可用，该函数就获取此内存区
 # 如果无法找到足够大的内存区，就像Linux请求更多内存，这种情况下
 # 此函数会向前移动current_break
 .globl allocate
 .type allocate,@function
 .equ ST_MEM_SIZE, 8  # 用于分配内存大小的栈位置
allocate:
 pushl %ebp                        # 标准函数处理
 movl %esp, %ebp

 movl ST_MEM_SIZE(%ebp), %ecx  # %ecx将保存我们需要的大小
                               #  （第一个也是唯一的参数）

 movl heap_begin, %eax         # %eax将保持当前搜索位置

 movl current_break, %ebx      # %ebx将保存当前中断

alloc_loop_begin:                  # 此处开始循环搜索每个内存区

 cmpl %ebx, %eax            # 如果两者相等，就表明需要更多内存
 je   move_break

 # 获得此内存区的大小
 movl HDR_SIZE_OFFSET(%eax), %edx
 # 如果无可用空间，则继续搜索下一块内存区
 cmpl $UNAVAILABLE, HDR_AVAIL_OFFSET(%eax)
 je   next_location

 cmpl %edx, %ecx     # 如果内存区可用，就将之与所需大小进行比较
 jle allocate_here  # 如果足够大，就跳转至allocate_here
next_location:
 addl $HEADER_SIZE, %eax  # 内存区总大小为所需大小（当前%edx中存储的值）
 addl %edx, %eax          # + 内存头8字节
                          # ( AVAILABLE/UNAVAILABLE标志4字节 + 内存区大小4字节)
                          # 因此将%edx与$8相加，结果存于%eax中，
                          # 即可获得下一个可用内存区
```

```
 jmp alloc_loop_begin      # 查看下一个位置

allocate_here:                  # 如果执行此处代码，说明要分配的内存区头在%eax中

 # 将空间标识为不可用
 movl $UNAVAILABLE, HDR_AVAIL_OFFSET(%eax)
 addl $HEADER_SIZE, %eax  # 将可用内存区头的下一个位置移入%eax
                          # (因为这是我们要返回的内容)

 movl %ebp, %esp          # 从函数中返回
 popl %ebp
 ret

move_break:                     # 如果执行到这里，说明已经耗尽所有可寻址内存，
                                # 需要请求更多内存
                                # %ebx保存当前数据结束处位置，%ecx保存数据大小

                                # 需要增加%ebx的值，使之为我们想要内存结束的地方
 addl $HEADER_SIZE, %ebx        # 因此要将其与内存区域头结构的大小相加
 addl %ecx, %ebx                # 然后将中断与所请求数据的大小相加

                                # 接着就要向Linux要求更多内存

 pushl %eax                     # 保存所需寄存器
 pushl %ecx
 pushl %ebx

 movl $SYS_BRK, %eax            # 重置中断（%ebx含所请求的中断点）
 int $LINUX_SYSCALL
                                # 在正常情况下，应返回新中断到%eax中，
              # 如果失败，返回值为0，否则新中断应
              # 大于等于我们请求的内存。在本程序中，
              # 我们并不关心实际中断设置在何处，
              # 只要%eax内容不为0，我们并不关心其实际值

 cmpl $0, %eax                  # 检测错误情况
 je error

 popl %ebx                      # 恢复保存的寄存器
 popl %ecx
 popl %eax

 # 设置该内存为不可用，因为我们将分配该内存
 movl $UNAVAILABLE, HDR_AVAIL_OFFSET(%eax)
 # 设置该内存的大小
 movl %ecx, HDR_SIZE_OFFSET(%eax)
```

```
 # 将%eax移至可用内存的实际起始处。%eax现保存着返回值
 addl $HEADER_SIZE, %eax

 movl %ebx, current_break  # 保存新中断

 movl %ebp, %esp           # 从该函数返回
 popl %ebp
 ret

error:
 movl $0, %eax             # 如果出错，就返回0
 movl %ebp, %esp
 popl %ebp
 ret
########函数结束#########

 ##deallocate##
 # 目的:
 #      此函数的目的是使用内存区域后将之返回到内存池中
 #
 # 参数:
 #      唯一的参数是我们要返回到内存池的内存的地址
 #
 # 返回值:
 #      无返回值
 #
 # 具体处理:
 #       你是否还记得，我们实际上将可用内存起始位置传递给程序，
 #       该起始位置就是内存区时间起始处加上8个存储位置
 #       我们只需倒退8个存储位置，然后标识此内存区为可用
 #       即可，这样分配函数就知道可以使用此内存区了
 .globl deallocate
 .type deallocate,@function
 # 要释放的内存区域栈位置
 .equ ST_MEMORY_SEG, 4
deallocate:
 # 因为此函数很简单，我们无需再用
 # 专门函数获取要释放的内存地址（通常该地址为8(%ebp)，
 # 但由于我们并未将%ebp入栈或将%esp内容移至%ebp，
 # 此处使用4(%esp)
 movl ST_MEMORY_SEG(%esp), %eax

 # 获得指向内存实际起始处的指针
 subl $HEADER_SIZE, %eax
```

```
    # 标识该内存区为可用
    movl $AVAILABLE, HDR_AVAIL_OFFSET(%eax)

    # 返回
    ret
########函数结束##########
```

首先引起我们注意的就是这里没有_start符号，这是因为这只是一个函数集合。内存管理器本身并非一个完整的程序——它什么都不做。内存管理器只是被其他程序使用的工具。

执行以下命令来汇编程序：

```
as alloc.s -o alloc.o
```

好了，现在让我们来看代码。

9.5.1　变量和常量

在程序的开始，我们已经设定了两个位置：

```
heap_begin:
 .long 0

current_break:
 .long 0
```

记住，被管理的内存部分通常称为堆。当我们汇编程序时，并不知道堆从哪里开始，也不知道当前中断在哪里。因此，我们为两者的地址保留空间，但暂时将地址设为0。

接着，我们用一组常量来定义堆结构。内存管理器的工作方式是：在每个被分配内存区之前，有一个描述该内存区的简短记录。这个记录含一个保留字用于放置表明该内存区是否可用的标志，还有一个表明该内存区大小的保留字。被分配的实际内存紧跟在这一记录之后。可用标志用来标记当前内存区是可用于分配还是目前正在被使用。大小字段用于判断这个内存区是否满足分配请求，以及下一个内存区的位置。以下常量用于描述此记录：

```
.equ HEADER_SIZE, 8
.equ HDR_AVAIL_OFFSET, 0
.equ HDR_SIZE_OFFSET, 4
```

这组常量表明内存区头共占用8字节，可用标志为从内存区起始处偏移0字节，大小字段为从起始处偏移4字节。若一直能善加利用这些常量，那么稍后决定将更多信息加到内存区头时，这

就不会为我们增加太多工作量。

对于AVAILABLE字段，值为0表示不可用，值1表示可用。为了方便阅读程序，我们有以下定义：

```
.equ UNAVAILABLE, 0
.equ AVAILABLE, 1
```

最后，我们的Linux系统调用定义如下：

```
.equ BRK, 45
.equ LINUX_SYSCALL, 0x80
```

1. allocate_init函数

allocate_init是一个简单的函数，其功能为设定我们前面讨论过的heap_begin和current_break变量。如果你还记得前面的讨论，就能用brk系统调用找到当前中断。所以，该函数开始如下：

```
pushl %ebp
movl  %esp, %ebp

movl  $SYS_BRK, %eax
movl  $0, %ebx
int   $LINUX_SYSCALL
```

无论如何，执行INT $ LINUX_SYSCALL之后，%eax中保存的是最后有效地址。但我们实际上想要的是首个无效地址，而非最后有效地址，所以简单地递增%eax，然后移动该值到内存位置heap_begin和current_break。接着就我们离开函数。代码如下所示：

```
incl %eax
movl %eax, current_break
movl %eax, heap_begin
movl %ebp, %esp
popl %ebp
ret
```

堆由heap_begin和current_break之间的内存构成，所以上述代码表明开始时堆为0字节。而当调用分配函数时，该函数将堆扩展到所需大小。

2. allocate函数

这是一个重要函数，让我们先看一下其概要。

(1) 从堆的起始处开始。

(2) 查看是否处于堆结束处。

(3) 若在堆结束处，则从Linux获取我们所需的内存区，将其标记为“不可用”，并返回该内存区。如果无法从Linux获取，则返回0。

(4) 若当前内存区标记为“不可用”，则前进到下一个内存区，并回到步骤(2)。

(5) 若当前内存区太小，无法提供请求的内存空间大小，则回到步骤(2)。

(6) 若能提供足够大的内存区，则将其标记为“不可用”，并返回该内存区。

现在，请记住以上内容并阅读代码，并务必阅读注释，这样你就能了解各寄存器的值。

回顾代码后，让我们一次一行来分析代码。开始处的代码如下：

```
pushl %ebp
movl  %esp, %ebp
movl  ST_MEM_SIZE(%ebp), %ecx
movl  heap_begin, %eax
movl  current_break, %ebx
```

这一部分初始化所有寄存器。前两行是标准函数处理。接下来的一步从栈中获取要分配的内存大小。内存大小是此处唯一的函数参数。然后，下一行代码将堆起始和结束地址移入寄存器中。好，现在做好了进行处理的准备工作。

下一部分标记为alloc_loop_begin。在这个循环中，我们要检查内存区，直到找到一个可用存储区或确定我们需要更多内存。以下两条指令查看是否需要更多的内存：

```
cmpl %ebx, %eax
je   move_break
```

%eax保存正在检测的当前内存区，%ebx保存的是堆结束后的一个内存位置。因此，如果下一个要检测的内存区超过堆结束位置，就意味着我们需要更多内存才能分配指定大小的内存区。我们看看move_break部分的功能：

```
move_break:
 addl  $HEADER_SIZE, %ebx
 addl  %ecx, %ebx
 pushl %eax
 pushl %ecx
 pushl %ebx
```

```
movl  $SYS_BRK, %eax
int   $LINUX_SYSCALL
```

此时，%ebx中保存的是我们想要的下一个内存区地址。因此，将之与内存区头大小和和内存区大小相加，这就是我们想要的系统中断。然后，我们将想要保存的寄存器入栈，并发起brk系统调用。之后，检查是否出错：

```
cmpl $0, %eax
je   error
```

如果没有出错，我们从栈中弹出寄存器内容，标记内存区为不可用，记录内存区大小，并确保%eax指向可用内存开始处（开始处在内存区头之后）。

```
popl %ebx
popl %ecx
popl %eax
movl $UNAVAILABLE, HDR_AVAIL_OFFSET(%eax)
movl %ecx, HDR_SIZE_OFFSET(%eax)
addl $HEADER_SIZE, %eax
```

然后，我们存储新的程序中断并返回指向被分配内存的指针。

```
movl %ebx, current_break
movl %ebp, %esp
popl %ebp
ret
```

error代码只是返回0到%eax中，所以我们不再讨论。

回顾一下循环的其余部分。如果查看的当前内存区并未超过堆结束处，那么该怎么处理呢？现在我们来看一下。

```
movl HDR_SIZE_OFFSET(%eax), %edx
cmpl $UNAVAILABLE, HDR_AVAIL_OFFSET(%eax)
je   next_location
```

上面的代码首先获取内存区大小并置于%edx中，接着查看可用标志是否被设置为UNAVAILABLE。若设为UNAVAILABLE，则表明该内存区正在使用中，必须跳过。所以，如果可用标志设置为UNAVAILABLE，就跳到标记为next_location的代码。如果可用的标志设置为AVAILABLE，那么我们就继续执行。

假设该内存空间可用，我们继续执行。接着查看这个空间是否够大以便包含所需大小的内存。该内存区的大小保存在%edx中，因此我们执行以下代码：

```
cmpl %edx, %ecx
jle  allocate_here
```

如果所请求的尺寸小于或等于当前内存区大小，我们就可以使用当前内存块。当前内存区是否大于所请求的内存无关紧要，多余的空间只是不会被用到。所以，让我们跳转到allocate_here，看看该部分代码的用途：

```
movl $UNAVAILABLE, HDR_AVAIL_OFFSET(%eax)
addl $HEADER_SIZE, %eax
movl %ebp, %esp
popl %ebp
ret
```

以上代码将内存区标记为不可用，移动指针%eax到内存区头之后的位置，并将该位置作为函数返回值。请记住，使用此函数的人甚至无需知道我们内存头记录的存在。他们只需要一个指向可用内存的指针。

现在让我们假设该内存区不够大，这种情况下该如何处理呢？此时，我们跳转至标记为next_location的代码。每当发现当前内存区不足以分配内存时就会用到这段代码，其功能是将%eax中的指针前移，指向下一个可能的内存区，并跳转至循环起始处。记住，%edx中保存着当前内存区大小，HEADER_SIZE是内存区头大小的符号。因此这段代码将移动到下一个内存区：

```
addl $HEADER_SIZE, %eax
addl %edx, %eax
jmp  alloc_loop_begin
```

现在函数运行另一个循环。

只要有一个循环，你就必须确保循环必然会结束。做到这一点的最好办法就是检查所有的可能性，并确保所有可能性最终会使循环结束。在我们的例子中，存在以下几种可能性：

- 将到达堆结束处；
- 会找到一个足够大的可用内存区；
- 将前进至下一个位置。

前两条将导致循环结束。第三条将使循环继续，但即使我们最终未能找到一个可用内存区，最终都会到达堆结束处，因为堆大小有限。因此，无论哪个条件为真，循环最终会满足终止条件。

3. deallocate函数

deallocate函数比allocate函数容易得多。这是因为该函数不做任何搜索，只是将当前内存

区标记为AVAILABLE，而allocate就会在下一次被调用时找到该内存区。此函数代码如下所示：

```
movl ST_MEMORY_SEG(%esp), %eax
subl $HEADER_SIZE, %eax
movl $AVAILABLE, HDR_AVAIL_OFFSET(%eax)
ret
```

在这个函数中，无需保存%ebp或%esp，因为我们不会改变它们，也无需在函数结束时恢复它们。这里只是从栈中读取内存区地址，回退到内存区头起始地址，将该内存区标识为AVAILABLE。该函数没有返回值，所以无需关注留在%eax中的值。

9.5.2 性能问题及其他

我们的简化版内存管理器仅适用于实验，并不实用。本节关注这个简化版分配器中存在的问题。

最大的问题就是速度。如果只需要进行少量分配，那么速度不会是大问题。但设想一下，如果你要进行1000次分配会出现什么情况。如果分配数为1000，你必须搜索999个内存区来查找是否要请求更多内存。正如你所见，速度会变得很慢。此外要注意，Linux可以在磁盘而非内存中保留内存页。而你必须检查程序所用的每一部分内存，这就意味着Linux不得不加载目前在磁盘上的所有内存页部分，查看其是否可用。你可以想象程序会变得多么慢。[①]据说此方法的运行与时间成线性关系，这意味着你要管理的每一个元素都会让程序占用更多时间。对于运行时间恒定的程序，无论你管理多少元素，其运行都占用同样的时间。就以deallocate函数为例，不管我们要管理多少元素，无论这些元素在内存中的什么位置，该函数只运行4条指令。事实上，虽然我们的allocate函数是所有内存管理器中最慢的一个，deallocate函数却是最快的一个。

另一个性能问题是我们发起brk系统调用的次数。系统调用需要很长时间来运行。它们与函数不同，因为处理器必须切换模式。你的程序是不被允许自行映射内存的，这一功能必须由Linux内核完成。因此，处理器必须切换到*内核模式*，再由Linux映射内存，然后再切换回*用户模式*，让你的应用程序继续运行。这也称为*上下文切换*。在x86处理器上，上下文切换速度相对较慢。一般来说，除非确实需要，否则你应该避免调用内核。

另一个问题是我们并没有记录Linux实际设置中断的位置。之前提到Linux实际上可能将中断设置到我们要求的内存大小之后。在这个程序中，我们甚至没有检测Linux实际上设置中断的地方，只是假设设置在我们请求的地方。这实际上不算错，但在我们已经映射了内存的情况下会引

① 这就是为什么为计算机增加内存会加快其运行速度。计算机的内存越多，其存放在磁盘上的内存页就越少，就无需常常打断程序以从磁盘上检索页面。

起不必要的brk系统调用。

来看下一个问题，如果正在寻找一个5字节的内存区，而遇到的第一个可用内存区是1000字节，那么我们只会将整个内存区标识为可用并返回该内存区。这会留下995字节已分配但未使用的内存。在这种情况下，最好将之分隔，以使这995字节稍后可以使用。在需要较大分配内存时，将连续的可用空间合并起来也不错。

9.6　使用我们的分配器

本书中，我们的程序并没有复杂到需要内存管理器的程度。因此，这里只是要使用我们的内存管理器为一个文件读/写程序分配缓冲区，而不是在.bss段中分配缓冲区。

我们要用来展示这一点的程序是第6章的read-records.s。这个程序使用一个名为record_buffer的缓冲区来处理其输入/输出需求。我们将之从在.bss中定义的缓冲区，更改为指向使用内存管理器的动态分配缓冲区的指针。因为本节只是讨论代码更改，所以你需要找到该程序的代码。

我们首先需要在声明中进行更改。现在，声明如下所示：

```
.section .bss
.lcomm, record_buffer, RECORD_SIZE
```

保持相同的名称名不副实，因为我们将之从实际缓冲区更改为一个指向缓冲区的指针。此外，该节只需要一个字大小，以便容纳一个指针。新的声明将在.data段中，如下所示：

```
record_buffer_ptr:
 .long 0
```

来看下一个变化：我们需要在程序开始后立刻初始化内存管理器。因此，在设置栈之后，需要添加以下代码：

```
call allocate_init
```

在此之后，内存管理器准备好为内存分配请求服务了。我们需要分配足够的内存来保存这些正在读取的记录。因此，这里将调用allocate来分配该内存，并保存该函数返回到record_buffer_ptr的指针。代码如下所示：

```
pushl $RECORD_SIZE
call  allocate
movl  %eax, record_buffer_ptr
```

现在，当调用read_record时，该函数需要一个指针。在原代码中，指针为对于record_buffer的立即寻址方式。现在，record_buffer_ptr只是持有指针，而不是缓冲区。因此，我们必须采用直接寻址方式来进行加载，才能获得record_buffer_ptr的值。需要删除下面这一行代码：

```
pushl $record_buffer
```

并以下面这一行代码代替：

```
pushl record_buffer_ptr
```

下一个更改是在我们尝试查找记录的FIRSTNAME字段的地址时，在原代码中该地址为$RECORD_FIRSTNAME + record_buffer，但这仅当地址和偏移量都为常数时才适用；在新代码中，该地址为存储在record_buffer_ptr中的地址偏移量。为了获得值，我们需要将指针移入寄存器，然后与$ RECORD_FIRSTNAME相加。在如下代码处：

```
pushl $RECORD_FIRSTNAME + record_buffer
```

我们需要将其替换为：

```
movl record_buffer_ptr, %eax
addl $RECORD_FIRSTNAME, %eax
pushl %eax
```

同样，我们要将以下代码：

```
movl $RECORD_FIRSTNAME + record_buffer, %ecx
```

更改为：

```
movl record_buffer_ptr, %ecx
addl $RECORD_FIRSTNAME, %ecx
```

最后要进行的更改是使用完后释放内存（在此程序中无需这样做，但这是一个很好的做法）。要做到这一点，我们只要在退出前将record_buffer_ptr传递给deallocate函数：

```
pushl record_buffer_ptr
call  deallocate
```

现在你可以通过以下命令生成程序：

```
as read-records.s -o read-records.o
ld alloc.o read-record.o read-records.o write-newline.o countchars.o
-o read-records
```

接着通过键入./read-records运行程序。

此时，动态内存分配的用途可能还不明显，但当你逐渐从学术实验转向现实中的程序时，将不断用到它。

9.7　更多信息

要详细了解Linux和其他操作系统中的内存处理，可以参考以下内容：

- 关于Linux程序的内存布局，更多信息参见文档“Startup state of a Linux/i386 ELF binary”，网址为http://linuxassembly.org/startup.html；
- 关于许多不同系统中虚拟内存的概述，请参见http://cne.gmu.edu/modules/vm/；
- 几篇深入分析Linux的虚拟内存子系统的文章，请参见http://www.nongnu.org/lkdp/files.html；
- Doug Lea为其大受欢迎的内存分配器提供了说明，请参见http://gee.cs.oswego.edu/dl/html/malloc.html；
- 关于4.4 BSD内存分配器的论文，请参见http://docs.freebsd.org/44doc/papers/malloc.html。

9.8　温故知新

9.8.1　理解概念

- 请描述Linux程序开始时的内存布局。
- 什么是堆？
- 什么是当前中断？
- 栈向哪个方向增长？
- 堆向哪个方向增长？
- 访问未映射内存时会发生什么？
- 操作系统如何防止进程覆盖其他进程的内存？
- 试描述你使用的内存当前驻留磁盘时的处理过程。
- 为什么你需要一个分配器？

9.8.2　应用概念

- 修改内存管理器，以便其在未被初始化时自动调用allocate_init。

- 修改内存管理器，使它在所请求的内存小于所选内存区时拆分内存区为多个。请实施时务必考虑新的头记录的大小。
- 修改使用缓冲区获得内存的程序，使之通过内存管理器获得缓冲区，而不是使用.bss。

9.8.3 深入学习

- 研究垃圾收集。与这里使用的内存管理方式相比，垃圾收集有什么优缺点？
- 研究引用计数。与这里使用的内存管理方式相比，引用计数有什么优缺点？
- 更改malloc和free函数的名称，并生成一个共享库。试用LD_PRELOAD来强制使用该库取代你的默认内存管理器。试增加一些write系统调用到STDOUT，来验证已经使用你的内存管理器取代了默认的内存管理器。

第 10 章

像计算机一样计数

10

10.1 计数

10.1.1 像人类一样计数

计算机的计算方式在许多方面同人类一样。所以，开始了解计算机如何计算前，先来深入了解一下我们人类如何进行计算。

你有多少根手指？注意，这不是脑筋急转弯。人通常有10根手指。这点很重要，为什么？来看看我们的计数系统。一位数什么时候会变成两位数？是的，成为数字10的时候。人类计数和做算术都使用十进制。基数10意味以10为所有事物分组。例如，我们在数羊：1，2，3，4，5，6，7，8，9，10。为什么现在我们一下子有了一个两位数，并且重用了1？这是因为我们按照10来对数字分组，因此有1组10只羊。现在来看下一个数字——11。这意味着有1组10只羊和1只不成组的羊。因此，我们继续数羊：12，13，14，15，16，17，18，19，20。现在我们有2组，每组10只羊。 21意味着2组10只羊和1只不成组的羊，22为2组（每组10只）羊和2只不成组的羊。如果我们继续计数，一直数到97、98、99及100。看，同样的情况发生了！到100时会发生什么？我们现在有10组，每组10只羊。到101时，我们有10组（每组10只）羊和1只不成组的羊。因此，我们可以用同样的方式看待所有数字。如果数了60 879只羊，这将意味着我们有：6×10×10×10组（每组10只）羊+0×10×10组（每组10只）羊+8×10组（每组10只）羊+7组（每组10只）羊+9只不成组的羊。

那么，按10分组有何重要性？没有！只是我们一直习惯于这样做，因为人类一般有10根手指。我们也可以按9或11分组计数（在这种情况下，我们将不得不发明一个新的符号）。按不同数字分组的区别在于，我们必须针对每种分组重新学习加减乘除，但规则并未改变，变化的是表示它们的方式。此外，我们会发现某些已经了解的计算技巧并不总是适用。例如，假设我们按9而不是

10分组计数。此时将小数点向右侧移动不再代表乘以10，而是代表乘以9。对于9进制而言，500意味着5×9×9。

10.1.2 像计算机一样计数

问题是，计算机该用“几根手指”来计数呢？计算机只有“两根手指”，而这就意味着所有对象都是按照2来分组的。因此，让我们用二进制来计数，即：0（0），1（1），10（2，1组2），11（3，1组2以及未分组的1），100（4，2组2），101（5，2组2以及未分组的1），110（6，2组2以及1组2），以此类推。在基数为2的计数系统中，小数点向右移动一位意味着乘以2，向左移动1位表示除以2。相应的基数为2的计数系统也称为二进制。

基数为2的好处在于基本算术运算表很短。对于基数10，乘法表为10列宽，10行高。而对于基数2，运算表极为简单：

二进制加法表

```
+ |   0   |   1
- + ----- + -----
0 |   0   |   0
- + ----- + -----
1 |   1   |  10
```

二进制乘法表

```
* |   0   |   1
- + ----- + -----
0 |   0   |   0
- + ----- + -----
1 |   0   |   1
```

如果我们将数字10010101与1100101相加，则有：

```
  10010101
+  1100101
-----------
  11111010
```

现在将两者相乘：

```
      10010101
   *   1100101
      -----------
```

```
    10010101
   00000000
  10010101
 00000000
00000000
 10010101
 10010101
---------------
11101011001001
```

10.1.3　二进制和十进制之间的转换

我们来看如何将数字从二进制（基数为2）转换为十进制（基数为10）。其实，这是一个相当简单的过程。你还记得吗？每个数字代表某些2的组。所以，我们只需将每个数字代表的分组相加，即可获得十进制数。以二进制数10010101为例。要确定其对应的十进制数，我们将其拆分如下：

```
1    0    0    1    0    1    0    1
|    |    |    |    |    |    |    |
|    |    |    |    |    |    |    单个单位 (2 ^ 0)
|    |    |    |    |    |    0组2 (2 ^ 1)
|    |    |    |    |    1组4 (2 ^ 2)
|    |    |    |    0组8 (2 ^ 3)
|    |    |    1组16 (2 ^ 4)
|    |    0组32 (2 ^ 5)
|    0组64 (2 ^ 6)
1组128 (2 ^ 7)
```

然后我们将每个相加如下：

```
1 × 128 + 0 × 64 + 0 × 32 + 1 × 16 + 0 × 8 + 1 × 4 + 0 × 2 + 1 × 1 =
128 + 16 + 4 + 1 =
149
```

所以，二进制的10010101对应十进制的149。让我们再来看看1100101。这个数可以写成如下形式：

```
1 × 64 + 1 × 32 + 0 × 16 + 0 × 8 + 1 × 4 + 0 × 2 + 1 × 1 =
64 + 32 + 4 + 1 =
101
```

所以，二进制的1100101就是十进制的101。让我们再来看一个数字——11101011001001。你可以如下所示将其转换为十进制数：

```
1 × 8192 + 1 × 4096 + 1 × 2048 + 0 × 1024 + 1 × 512 + 0 × 256
         + 1 × 128 + 1 × 64 + 0 × 32 + 0 × 16 + 1 × 8 + 0 × 4
         + 0 × 2 + 1 × 1 =

8192 + 4096 + 2048 + 512 + 128 + 64 + 8 + 1 =

15049
```

现在，如果你一直认真阅读的话，一定已经注意到，我们刚刚转换的数字就是之前乘法示例中所用的数字。所以，让我们来检查一下其结果：101 × 149 = 15 049。这么做是可行的！

现在，让我们来看看如何从十进制转换回二进制。为了进行转换，你必须将数字**除以**2的分组。例如，对于数字17，如果除以2，你会得到8及余数1。因此，这意味着有8组2，剩余1不成组。这意味着，最右边的数字将是1。现在，我们已经算出了最右边的数字，以及剩下的8组2。现在用2除8，来看看有多少组2组2。我们得到了4，没有余数。这意味着，所有8组2可以进一步分为更多的组，每组2组2。所以，我们有0组2。因此，右边第2位为0。然后，我们用4除以2获得2以及余数0，所以右边第3位是0。然后，我们将2除以2，获得1，余数0。因此，右边第4位为0。最后，我们将1除以 2，商为0，余数为1，所以接下来的一位为1。现在，没有数遗留，我们完成了转换，最终获得的数是10001。

之前，我们曾将十进制的15 049转换为二进制数11101011001001。让我们反过来做，以确保做法是正确的：

```
15049 / 2 = 7524 余 1
7524 / 2 = 3762  余 0
3762 / 2 = 1881  余 0
1881 / 2 = 940   余 1
940 / 2 = 470    余 0
470 / 2 = 235    余 0
235 / 2 = 117    余 1
117 / 2 = 58     余 1
58 / 2 = 29      余 0
29 / 2 = 14      余 1
14 / 2 = 7       余 0
7 / 2 = 3        余 1
3 / 2 = 1        余 1
1 / 2 = 0        余 1
```

然后，我们把余数并在一起即获得原来的二进制！请记住，第一次除法的余数为最右边的数字，所以你必须从下往上合并数字，得到11101011001001。

二进制数中的每一位数字称为一位，代表二进制数字。注意，计算机将其内存划分为称为字节的存储位置。在x86处理器（以及大多数其他处理器）上，每个存储位置长度为8位。我们在之

前的章节曾说过，一个字节可以容纳介于0和255之间的任何数字，原因在于可以放入8位的最大数是255。如果你亲自将二进制11111111转换成十进制数就会验证这点：

```
11111111 =

(1 × 2 ^ 7) + (1 × 2 ^ 6) + (1 × 2 ^ 5) + (1 × 2 ^ 4) + (1 × 2 ^ 3)
            + (1 × 2 ^ 2) + (1 × 2 ^ 1) + (1 × 2 ^ 0) =

128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 =

255
```

16位可容纳的最大数为65 535，32位可保存的最大数为4 294 967 295（40亿），64位可保存的最大数为18 446 744 073 709 551 615。128位可保存的最大数是340 282 366 920 938 463 463 374 607 431 768 211 456。无论如何，你现在已经大致了解了，对于x86处理器，大部分时间你会和4字节数字（32位）打交道，因为这正是寄存器的大小。

10.2　真假和二进制数

现在，我们已经看到，计算机将一切都作为1和0的序列来存储。来看看这些序列的其他用途。假设不是将二进制数序列看做数字，而是看做一组开关。例如，有4个开关控制房子的灯光。一个开关对应外面的灯，一个开关对应走廊灯，一个开关对应客厅的灯，还有一个对应卧室灯。我们可以用列表表示这些灯的开关状态，如下所示：

```
外面      走廊      客厅      卧室
 开        关        开        开
```

显然，该表显示除走廊灯外，其他灯都开着。现在用数字1和0分别替换“开”和“关”，1表示开，0代表关。所以，相同的信息表示如下：

```
外面      走廊      客厅      卧室
 1         0         1         1
```

现在，我们不再标记各灯开关，而是记住哪个位置对应哪个开关。这样一来，同样的信息可表示为：

```
1         0         1         1
```

或：

```
1011
```

这只是用计算机存储位置来表示数字以外信息的一种方式。计算机内存只能看到数字，但程序

员可以使用这些数字代表他们能想到的任何内容。但有时他们得有点创意，才能找出最佳表述方式。

你不仅能用二进制数进行普通算术运算，还能使用二进制特有的运算，即二进制或逻辑运算。标准的二进制运算有：

- 与（AND）；
- 或（OR）；
- 非（NOT）；
- 异或（XOR）。

在讨论示例前，我将先为你介绍这些二进制运算。AND需要两个一位二进制操作数，并返回一个一位数；仅当两者都为1时才返回1，否则返回0。例如，1 AND 1为1，1 AND 0为0，0 AND 1为0，0 AND 0为0。

OR需要两个一位二进制操作数，并返回一个一位数；如果两个原始操作数之一为1即返回1。例如，1 OR 1为1，1 OR 0为1， 0 OR 1为1，但0 OR 0为0。

NOT只需一个一位二进制操作数，并返回与原数相反的数，如NOT 1是0，NOT 0为1。

最后，XOR与OR类似，区别在于当两位都为1时返回0。

计算机可同时在整组寄存器上运行逻辑运算。例如，如果一个寄存器含有10001000010101010101010101111010，另一个含有10100010101010010101101100101010，你就可以对整组寄存器运行这些逻辑运算。例如，如果我们对两者进行AND运算，计算机对第1位到第32位按位运行AND操作。在这种情况下：

```
10100010101010010101101100101010 AND
10001000010101010101010101111010
--------------------------------
10000000000000010101000100101010
```

你会看到，在产生的结果中，当两个对应位都为1时结果位为1，其他位都为0。让我们来看看OR运算的结果：

```
10100010101010010101101100101010 OR
10001000010101010101010101111010
--------------------------------
10101010111111010101111101111010
```

在这种情况下，当两个对应位有任一位为1时结果即为1。再来看NOT运算：

```
NOT 10100010101010010101101100101010
------------------------------------
    01011101010101101010010011010101
```

该运算只是对每位数取反。最后，我们来看XOR运算，这种运算类似于OR运算，只是当两个相应位均为1时返回0：

```
10100010101010010101101100101010 XOR
10001000010101010101010101111010
--------------------------------
00101010111111000000111001010000
```

本例所用两个操作数与OR运算中的完全相同，这样你可以比较它们是如何进行运算的。另外，如果将数字与本身进行XOR运算，你总是会得到0，如下所示：

```
10100010101010010101101100101010 XOR
10100010101010010101101100101010
--------------------------------
00000000000000000000000000000000
```

这些操作非常有用，原因有二：

- 计算机进行逻辑运算极为迅速；
- 你可以利用逻辑运算同时比较多个真值。

你可能还不知道，不同指令执行速度不同，这是千真万确的。而这些逻辑运算在大多数处理器上都是执行最快的。例如，你看到将数字与自身XOR产生0， XOR操作速度就比加载操作快，所以很多程序员使用此指令为寄存器加载0。例如，通常用代码：

```
xorl %eax, %eax
```

来代替：

```
movl $0, %eax
```

我们将在第12章中详细讨论速度，但我希望你了解程序员是如何采用某些技巧的，特别是通过二进制运算符来加速。现在来看看如何使用这些运算符操纵真/假值。之前讨论过如何用二进制数表示任何事物，现在我们用二进制数字来表示我的父亲和我喜欢什么东西。首先，来看看我喜欢的：

```
食品：是
重金属音乐：是
穿着考究的衣服：否
足球：是
```

再来看看我父亲喜欢什么：

```
食品：是
```

```
重金属音乐：无
穿着考究的衣服：是
足球：是
```

现在使用1表示我们喜欢的东西，用0表示我们不喜欢的。现在，上述喜好可表示为：

```
我
食物：1
重金属音乐：1
穿着考究的衣服：0
足球：1

我的父亲
食物：1
重金属音乐：0
穿着考究的衣服：1
足球：1
```

现在，如果我们记住各个位置对应的内容，则上述喜好可表示如下：

```
我
1101

我的父亲
1011
```

现在假设要取我和父亲都喜欢的事物列表，你可以使用与运算，因此有：

```
1101 AND
1011
--------
1001
```

上述信息可以翻译成：

```
我们都喜欢的东西
食品：是
重金属音乐：否
穿着考究的衣服：否
足球：是
```

注意，计算机不知道1和0代表什么，这是你和你程序的工作。如果你写了一个关于上述对应关系内容的程序，你的程序在某个时刻应检查每一位，并用代码告诉用户每一位对应的内容。（如果你问计算机两个人一致的地方是什么，而计算机回答1001，这个答案可没什么用处。）此外，假设我们想知道彼此喜好不一致的地方。为此，这里将使用XOR运算，因为只有对应位之一为1

时才会返回1，而两者都为1则返回0。所以有：

```
1101 XOR
1011
--------
0110
```

我要你将此结果翻译成对应内容。

前面的运算中，AND、OR、NOT、XOR称为布尔运算符，因为它们最初是乔治·布尔的研究成果。所以，如果有人提及布尔运算符或布尔代数，你现在应该知道它们指代什么了。

除了布尔运算符，还有两个非布尔型二进制运算符，即移位和循环移位运算符。移位和循环移位，顾名思义，可以移位和循环移位，并且可以向左或向右移位。左移位将一个二进制数每位数字向左边移动一位，向最右边空出的一位填充0，并去除最左边的一位。左循环移位进行同样的操作，但将最左边数字填充到最右边空出的位置上。例如：

```
左移       10010111 = 00101110
循环左移    10010111 = 00101111
```

注意，如果循环移位一个数字的每一位，最终结果就是原始数字（如循环移位一个32位的数字32次）。但是，如果移位一个数字的每一位，你会得到0。那么，移位有什么用处呢？如果二进制数代表某些信息，你可以通过移位查看每个值。例如，假设已将我父亲的喜好情况存储在一个32位寄存器中，寄存器内容如下所示：

```
00000000000000000000000000001011
```

我们之前曾经说过，上述寄存器内容并不能作为程序的输出。因此，为了进行输出，我们需要进行移位和屏蔽。屏蔽就是去除一切无关信息的过程。而具体到这种情况下，我们要对关注的所有值进行移位，使之移到对应屏蔽位为1的地方，然后屏蔽该位的值，这样我们就只看到这个值。屏蔽是通过将二进制数和另一个我们所关注的位设置为1的数进行AND运算来实现的。例如，假设想打印输出这一信息：我的父亲是否喜欢考究的衣服？该数据是右起第二个值。所以，我们必须右移1位，这样其内容如下所示：

```
00000000000000000000000000000101
```

接着，由于我们只想看那位数字，所以通过与00000000000000000000000000000001的AND运算屏蔽该值。

```
00000000000000000000000000000101 AND
00000000000000000000000000000001
----------------------------------
00000000000000000000000000000001
```

这样，如果我的父亲喜欢考究的衣服，那么寄存器的值将为1，反之为0。接着，我们将寄存器值和1进行比较，并打印结果。代码如下所示：

```
#注意 - 假定寄存器%ebx中保存信息为
#       我父亲的喜好

movl %ebx, %eax  # 这条命令将信息复制到%eax中，
                 # 这样我们不会失去原数据

shrl $1, %eax    # 这是移位运算符，代表逻辑右移
                 # 第一个数字表示移位次数
                 # 第二个参数表示要进行移位的寄存器

# 下面一条命令进行屏蔽
andl $0b00000000000000000000000000000001, %eax

# 检查结果是1还是0
cmpl $0b00000000000000000000000000000001, %eax

je   yes_he_likes_dressy_clothes

jmp  no_he_doesnt_like_dressy_clothes
```

程序中将有两个标记，会相应打印出他是否喜欢考究的衣服的信息，然后退出。0b符号表示紧接其后是一个二进制数。在本例中其实并不需要它，因为在任何进制系统中1都是相同的，但为了清楚起见，我还是保留了0b。我们也无需列出31个0，但这里保留它们以表明使用的数据长度为32位。

当一个数字表示一个函数或系统调用的一组选项时，其中每个真/假选项就称为标志。许多系统调用都会使用这种方式，将许多选项设置在同一寄存器中。例如系统调用open，其第二个参数是告诉操作系统如何打开文件的标志列表。其中部分标志如下所示。

- O_WRONLY

这个标志是二进制的0b00000000000000000000000000000001，或八进制的01（或以任何进制表示的同一数字），表示以只写模式打开文件。

- O_RDWR

这个标志是二进制的0b00000000000000000000000000000010，或八进制的02（或以任何进制表示的同一数字），表示打开文件用于读写。

- O_CREAT

这个标志是二进制的0b00000000000000000000000001000000或八进制的0100，表示如文件不存在则创建文件。

- O_TRUNC

这个标志是二进制的0b00000000000000000000001000000000，或八进制的01000，表示如文件存在则删除文件。

- O_APPEND

这个标志是二进制的0b00000000000000000000010000000000或八进制的02000，表示在文件结束处而非开始处写入。

要使用这些标志，使用OR运算你即可获得想要的组合。例如，要以只写模式打开一个文件，且当该文件不存在时创建文件，我会选用O_WRONLY（01）和O_CREAT（0100），对这两个选项进行OR运算后获得0101。

注意，如果你不设置O_WRONLY或O_RDWR，那么该文件将自动在只读模式下打开（O_RDONLY，但这不是一个真正的标志，因为其值为0）。

许多函数和系统调用用标志来表示选项，因为如果以一位表示一个选项的话，标志（一个字）可以容纳32个可能的选项。

10.3　程序状态寄存器

我们已经了解了如何用寄存器的位给出“是/否”的答案或“真/假”陈述。计算机上有一个寄存器名为*程序状态寄存器*，该寄存器保存了大量关于在计算过程中发生了什么的信息。例如，你是否曾思考过：如果两个数相加，其结果大于寄存器所能容纳的最大数，这时会发生什么？程序状态寄存器有一个标志是进位标志，你可以测试此标志来判断最近一次计算是否导致寄存器溢出。当然，还有其他很多标志用于表示不同的情形。事实上，当你执行比较指令（cmpl）后，结果就存放在程序状态寄存器中。条件跳转指令（jge、jne等）就是通过这些结果来决定是否需要跳转。无条件跳转jmp无需任何条件即执行跳转，因此并不关心程序状态寄存器的内容。

让我们来看个示例。假如你需要存储一个大于32位的数字，这个数字为两个寄存器的长度，

即64位。那么，应该怎么处理这个数字呢？如果你想将两个64位的数字相加，应该首先将低位寄存器相加，相加后如果检测到进位，则要将高位寄存器加1。事实上，这很可能就是你学习十进制加法时的做法：如果一列相加后大于9，要向下一个高位进位。例如，将65与37相加，首先将5和7相加获得12，在最右列个位保留数字2，向十位进位1，然后将6和3相加，再加上进位1，结果为10，然后在十位数保留数字0，向百位进位1，由于百位数原本没有数字，因此直接在百位数上保留数字1。幸运的是，一般对于常用数字来说，32位已经足够大了。

其他程序状态寄存器标志参见附录B。

10.4 其他计数系统

我们至今所研究的内容都只适用于正整数。但在实际应用中用到的数字并非都是正整数，负数和小数也常常被用到。

10.4.1 浮点数

我们到现在为止打交道的一直是整数，即不带小数点的数。计算机处理带小数点的数字时通常会有一个问题，因为计算机只能存储固定长度、有限的值。而小数可能会是任何长度，包括无限长度小数（想想循环小数，如1/3对应的小数吧）。

计算机处理小数的方式是以固定精度存储（即存储高位数字）。计算机采用*指数*和*尾数*两部分来存储小数。尾数包含实际要用到的数字，指数就是数字的数量级。例如，12345.2存储为1.23452 × 10 ^ 4。尾数为1.23452，指数为4。所有的数字都存储为X.XXXXX × 10 ^ XXXX的形式。数字1被存储为1.00000 × 10 ^ 0。

由于尾数和指数长度固定，这就导致了一些有趣的问题。例如，对于计算机存储的一个整数，如果你把它加1，就会得到比原数大1的一个整数。但对于浮点数就不一定如此了。如果浮点数非常大，比如5.234 × 10 ^ 5000，加1后，尾数可能不会有变化（记住，这两个部分长度固定）。这会影响一些事情，尤其是运算顺序。比方说，我用5.234 × 10 ^ 5000进行数十亿乃至万亿次加1操作。你猜会发生什么？这个数字不会发生任何变化。但是，如果我用1与1相加足够多的次数，然后将其与5.234 × 10 ^ 5000相加，结果很可能会发生变化。

注意，大多数计算机进行浮点运算所用时间比进行整数运算多得多。所以，对于真正需要保证速度的程序，整数是最常用的。

10.4.2　负数

你认为负数在计算机上是如何表示的？一种想法是用数字的首位作为正负号标识，所以00000000000000000000000000000001将代表数字1，而10000000000000000000000000000001代表数字－1。这是行得通的，事实上一些旧的处理器正是采用了这种方式。但这种方式存在一些问题。首先，这需要更多电路才能处理以这种方式表示的有符号数。更成问题的是，用这种表示方式表示数字0会存在问题。在该系统中，既有带负号的0又有带正号的0，从而引起很多问题，如“＋0和－0是否相等”、“在不同情况下0的符号应该为正，还是为负”。

采用*二进制补码*表示的负数表示法可以克服这些问题。为了得到某个数的二进制补码，你必须执行以下步骤：

(1) 对这个数执行NOT运算；

(2) 将运算结果加1。

因此，要获得00000000000000000000000000000001的补码，你首先进行NOT操作，获得11111111111111111111111111111110，然后将其加1，结果为11111111111111111111111111111111。要获得2的补码，先取其二进制00000000000000000000000000000010，对其进行NOT运算，获得11111111111111111111111111111101，将这一结果加1，获得补码为11111111111111111111111111111110。采用二进制补码表示，你可以像对待正数一样将数字相加，会获得正确答案。例如，如果将二进制的1和－1相加，你会发现两个数字的补码都是0。此外，第一位数字仍然是符号位，因此很容易判断是正数还是负数。负数的最左一位永远是1。这改变了给定位数所表示数字的有效范围。对于有符号数，可能的值范围分为正数和负数。例如，1字节通常最大值为255，但作为有符号数，1字节存储的值范围为－128~127。

对于二进制补码表示要注意一点，如果扩展数字的位数，你不能像对待无符号数那样只是向数的左边增加0。例如，用4位表示的数字－3为1101，如果要扩展到8位寄存器，我们不能将其表示为00001101，因为这表示的是13，而不是－3。当你在二进制补码表示法中对有符号数进行扩展时，必须进行*带符号扩展*。带符号扩展是指在为数字增加位数时，必须向其左侧填充其符号位的数字。所以，如果我们扩展一个4位的负数，应向扩充位的新数字填写1；如果我们扩展一个4位数的正数，应向扩充位的新数字填写0。所以，将－3从4位扩展到8位，其结果为11111101。

x86处理器针对有符号数和无符号数有不同的指令形式，如附录B所示。例如，x86处理器有一个保留符号的右移指令`sarl`，还有不保留符号位的右移指令`shrl`。

10.5 八进制和十六进制数字

目前为止，我们讨论的记数制只有十进制和二进制，但在计算中还常常用到另外两种记数制——八进制和十六进制。事实上，这两者可能比二进制更常用。八进制仅使用数字0到7，所以八进制数10实际上是十进制的8，而八进制的121是十进制的81（1组64（8 ^ 2），2组8，以及不成组的1）。八进制的好处是每3位个二进制数就构成一位八进制数（无法将二进制数分组对应十进制数）。因此0对应000，1对应001，2对应 010，3对应011，4对应100，5对应101，6对应 110，7对应111。

在Linux中的权限就采用八进制设置。这是因为Linux的权限设置都是基于读、写和执行能力。第一位是读权限，第二位是写权限，第三位是执行权限。所以，0（000）对应无权限，6（110）对应读取和写入权限，5（101）对应读取和执行权限。然后这些数字被用于3个不同的权限集——所有者、组和其他人。数字0644表示第一个权限集具备读写权限，第二个和第三个权限集具备只读权限。第一个权限集对应文件的所有者，第二个权限集对应该文件的组所有者，最后一个权限集对应其他人。因此，0751表示该文件的所有者可以读、写并执行该文件，组成员可以读取和执行文件，而其他人只能执行该文件。

如你所见，八进制用于按照3位分组二进制位。汇编器是通过八进制数的前缀0获悉该数是八进制数。例如，010表示八进制的10，即十进制的8。如果你只写10，就代表十进制的10。最开始的0用来区别两者。*所以，一定不要在任何十进制数前加0，否则它会被当做八进制数！*

十六进制数使用数字1~15来表示每位数字。然而，由于10~15没有对应的数，十六进制使用字母a~f来表示它们。例如，字母a表示10，字母b代表11，依次类推。十六进制的10是十进制的16。在八进制中，每位数字代表3位二进制数。在十六进制中，每位数代表4位二进制数。每两位数构成一个完整的字节，8位数构成一个32位的字。所以你看，书写十六进制数比书写二进制数容易多了，因为数字的位数仅为二进制数的1/4。十六进制最重要的数字是F，这意味着所有位都进行了设置。所以，如果我要设置寄存器的所有位为1，可以运行以下命令：

```
movl $0xFFFFFFFF, %eax
```

这比以下采用二进制数字的命令要简单得多，也更不容易出错：

```
movl $0b11111111111111111111111111111111, %eax
```

还要注意的是十六进制数字以0x开头。所以，当我们运行：

```
int $0x80
```

这表明正在调用中断号128（8组16），或中断号：

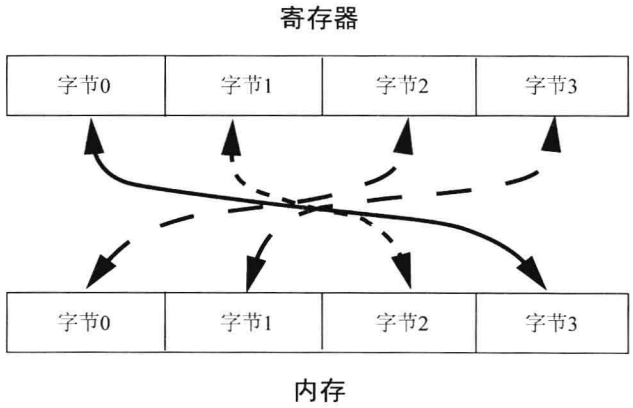
```
0b00000000000000000000000010000000
```

对于十六进制数和八进制数，你需要一些时间来适应，它们被大量用于计算机编程。你应该尝试编写几个十六进制数字，然后将之与二进制、十进制、八进制相互转换。

10.6　一个字中的字节顺序

在处理二进制位和字节时，有一件事让很多人困惑：将字节从寄存器写入内存时，总是先将最低有效部分先写出。[①]而大多数人的预期与此不同，他们认为如果寄存器中有一个字，比如说`0x5d 23 ef ee`（中间加空格是为了让你区分字节），各字节将以同样的顺序写入内存。然而，在x86处理器上，实际写入字节的顺序恰恰与此相反。在x86处理器上，内存的字节将是`0xee ef 23 5d`。字节的写入顺序恰恰与寄存器中的显示顺序相反，但字节内的二进制位顺序不变。

并非所有的处理器都采用这种方式。x86处理器是*小端字节序*处理器，这意味着它先存储“低字节”，即每个字的最低有效字节。



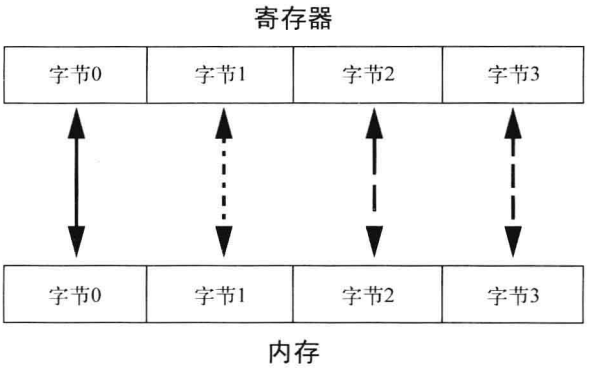
小端字节序系统上从寄存器到内存的转换

另一种处理器是大端字节序处理器，即它们保存字时先保存最高有效字节，就像我们读数字时一样。

两者的区别通常不是问题（虽然多年以来这种差异引发了很多技术上的争议）。因为当再度将字节读回寄存器时会再度翻转读取字节（如果是大端字节序处理器则不会有这种情况），程序

① 在此语境中，*有效*是指表示哪一位数字。例如，对于294，数字2是最有效的，代表百位，数字9是次有效位，而数字4是最低有效位。

员通常不会注意到字节是以哪种顺序读取的。在寄存器数据被传输到内存期间，字节交换的“魔术”在幕后自动发生。然而，下述几种情况下，字节顺序可能会引起问题。



大端字节序系统上从寄存器到内存的转换

- 如果尝试用`movl`一次读入多个字节，但却用最低有效字节逐个字节进行处理（通过使用`%al`和/或对寄存器进行移位），这样做会使字节与内存中的字节顺序不同。
- 如果为不同的架构读写文件，可能必须考虑不同架构写字节的顺序。
- 如果读取或写入网络套接字，可能协议中字节的顺序不同。

只要你意识到字节顺序问题，通常就不会受到什么影响。想更深入地了解字节顺序问题，请阅读http://www.rdrop.com/~cary/html/endian_faq.html上DAV的字节顺序常见问答，特别是Daniel Cohen的“On Holy Wars and a Plea for Peace”一文。

10.7 将数字转换成字符显示

到目前为止，除了将数字传递给退出代码，我们无法向用户展示任何存储的数字。在本节中，我们将讨论如何将正数转换成字符串来显示。

该函数命名为`integer2string`，需要两个参数：一个有待转换的整数，以及一个填充空字符（0）的字符串缓冲区。假定缓冲区足够大，能够将整个数字作为一个字符串存储。（至少有11个字符的长度，包括结尾的空字符。）

记住，我们一般都是以十进制方式看数字的。因此，要访问组成一个数的每一位十进制数字，我们需要将整个数除以10，然后显示余数的每一位。因此，整个过程将如下所示：

- 将这个数除以10；

- 余数是当前数字，将其转换为字符并存储；
- 若商是0，则转换完成；
- 否则，取商，获取缓冲区中的下一个位置，并重复以上过程。

唯一的问题：因为这个过程首先处理先获取的余数，结果将是逆序。因此，结束时我们必须逆转数字。为此，我们在计算时将数字对应的字符存储在栈上。通过这种方式，当将字符出栈并填充到缓冲区时，弹出顺序与入栈顺序正好相反。

该函数的代码应放到名为integer-to-string.s的文件，其内容如下：

```
# 目的：将一个整数转换为适合显示的十进制字符串
#
# 输入：一个足够大的缓冲区，足以存放可能的最大数
#       一个有待转换的整数
#
# 输出：缓冲区将被十进制字符串覆盖
#
# 变量：
#
# %ecx将保存已处理的字符数
# %eax将保存当前值
# %edi将保存基数（10）
#
 .equ ST_VALUE, 8
 .equ ST_BUFFER, 12

 .globl integer2string
 .type integer2string, @function
integer2string:
 # 一般函数开始
 pushl %ebp
 movl  %esp, %ebp

 # 当前字符计数
 movl $0, %ecx

 # 将值移动到所需位置
 movl ST_VALUE(%ebp), %eax

 # 当除以10时，数字10必须保存在寄存器或内存位置中
 movl $10, %edi

conversion_loop:
```

```
    # 除法实际上是在%edx:%eax两个寄存器上进行，
    # 因此首先清除%edx
    movl $0, %edx

    # 将%edx:%eax（这是默认的）除以10
    # 商存储在%eax中，余数存储在%edx （两者都是默认的）
    divl %edi

    # 商是在正确的位置上，%edx中含有余数，现在需要将其转换成数字
    # 因此，%edx中有一个0~9的数字，你也可以认为该数字是ASCII表
    # 上从字符0开始的索引。0的ASCII码加上0仍然得到0的ASCII码，0的
    # ASCII码加1为字符1的ASCII码。因此，以下指令将会让我们获得
    # %edx中数字对应的ASCII码
    addl $'0', %edx

    # 现在，我们将这个值入栈。这样，完成转换后就能以正确顺序逐个
    # 弹出字符
    # 注意，我们是将整个寄存器入栈，但字符只需要寄存器%dl部分的字节
    #（%edx寄存器的最后一个字节）
    pushl %edx

    # 递增数位计数
    incl %ecx

    # 检查%eax是否为0，若为0则执行下一条指令
    cmpl $0, %eax
    je   end_conversion_loop

    # %eax中已经有了新值

    jmp conversion_loop

end_conversion_loop:
    # 现在整个字符串都在栈中，如果我们一次弹出一个字符，
    # 就能将其复制到缓冲区中并完成任务

    # 获取%edx中指向缓冲区的指针
    movl ST_BUFFER(%ebp), %edx

copy_reversing_loop:
    # 我们将整个寄存器入栈，但只需要寄存器的最后一个字节
    # 因此，我们要弹出字符到整个%eax寄存器，但随后只移动
    # %eax寄存器的低位部分（%al）到字符串
    popl %eax
    movb %al, (%edx)
```

```
    # 递减%ecx，这样我们就能判断是否完成
    decl %ecx
    # 递增%edx，使指针指向下一个字节
    incl %edx

    # 检查是否完成
    cmpl $0, %ecx
    # 如果完成，就跳转到函数结束处
    je   end_copy_reversing_loop
    # 否则重复循环
    jmp  copy_reversing_loop

end_copy_reversing_loop:
    # 完成复制。现在写入一个空字节，并返回
    movb $0, (%edx)

    movl %ebp, %esp
    popl %ebp
    ret
```

为了展示如何在一个完整的程序中使用该函数，请使用如下代码，以及在前面各章中编写的count_chars 和write_newline函数。代码应存放在名为conversion-program.s的文件中。

```
    .include "linux.s"

    .section .data

    # 以下为其存储位置
tmp_buffer:
    .ascii "\0\0\0\0\0\0\0\0\0\0\0"

    .section .text

    .globl _start
_start:
    movl %esp, %ebp

    # 存放结果处
    pushl $tmp_buffer
    # 要转换的数字
    pushl $824
    call  integer2string
    addl  $8, %esp
```

```
# 为系统调用获取字符计数
pushl $tmp_buffer
call  count_chars
addl  $4, %esp

# 用于SYS_WRITE的字符计数存放在%edx中
movl %eax, %edx

# 进行系统调用
movl $SYS_WRITE, %eax
movl $STDOUT, %ebx
movl $tmp_buffer, %ecx

int  $LINUX_SYSCALL

# 进行带参数返回
pushl $STDOUT
call  write_newline

# 退出
movl $SYS_EXIT, %eax
movl $0, %ebx
int  $LINUX_SYSCALL
```

我们通过以下指令生成该程序：

```
as integer-to-string.s -o integer-to-number.o
as count-chars.s -o count-chars.o
as write-newline.s -o write-newline.o
as conversion-program.s -o conversion-program.o
ld integer-to-number.o count-chars.o write-newline.o conversionprogram.o
-o conversion-program
```

请输入./conversion-program以运行该程序，输出应为824。

10.8 温故知新

10.8.1 理解概念

- 请将十进制数5294转换成二进制数。
- 0x0234aeff代表什么数？试分别用二进制、八进制和十进制表示。
- 试将二进制数10111001和101011相加。

- 试将二进制数11001010110与自身相乘。
- 试将前两个问题的结果转换成十进制数。
- 试描述AND、OR、NOT、XOR的工作原理。
- 屏蔽有什么用途?
- 如果你想打开一个文件用于写入，并在文件不存在时创建该文件，那么当发起open系统调用时应使用什么标志?
- 如何在32位寄存器中表示 - 55?
- 试将上一题中的数字带符号扩展到64位寄存器。
- 试描述小端序存储和大端序存储之间的差异。

10.8.2　应用概念

- 回顾以前通过退出状态码返回数值结果的程序，试采用整数转换为字符串的函数来重写这些程序。
- 试修改integer2string代码，使返回结果为八进制，而不是十进制。
- 试修改integer2string代码，使转换基数为参数，而不是硬编码的转换基数。
- 试写一个名为is_negative的函数，所需参数为一个整数，如果参数为负就返回1，如果参数为正就返回0。

10.8.3　深入学习

- 试修改integer2string的代码，使转换基数能大于10（这需要你使用9以外的字符）。
- 试创建一个名为number2integer的函数，使其功能与integer2string相反。其所需参数为一个字符串，功能是将其转换为一个寄存器大小的整数。试用结果整数调用integer2string函数，以测试number2integer函数并显示结果。
- 试编写一个程序，用它将好恶信息存储到一个机器字中，然后比较分析两组好恶的共同点。
- 试编写一个程序从STDIN读取一串字符，并将其转换为数字。

第11章 高级语言

本章，我们来看第一个“真正”的编程语言。汇编语言是在机器级使用的语言，但大多数人觉得用汇编语言编程过于烦琐，不适合日常使用。为了简化编程任务，已有许多计算机语言问世。了解各种各样的编程语言非常有益，原因如下：

- 不同语言都基于不同的概念，有助于你学习掌握各种更好的编程方法和思路；
- 不同的语言适用于不同类型的项目；
- 不同的企业有不同的标准语言，因此了解更多的语言可使你成为更抢手的人才；
- 了解的语言越多，学习新语言就越容易。

程序员经常需要学习新的语言。一般而言，专业的程序员能够通过一个星期的研究和实践学会一门新语言。语言只是工具，而程序员不应惧怕学习使用新的工具。事实上，如果你从事计算机咨询，为了不失业，就往往需要现场学习新语言，因为用什么语言往往是由客户，而不是咨询师来决定的。本章将介绍多门语言。我建议你尽可能多地探索感兴趣的语言。事实上，每隔几个月我都会尝试学习一门新语言。

11.1 编译语言和解释语言

很多语言都属于*编译语言*。当你用汇编语言编程时，每一条指令都被对应转换成一条机器指令进行处理。而使用编译器时，一条命令能转化成一条或几百条机器指令，而这取决于编译器的先进程度。编译器甚至可能重组部分代码，使代码执行得更快。而用汇编语言编程时，你编写了什么代码，最终就执行什么命令，即所编即所得。

此外，还有一类语言：*解释语言*。这些语言要求用户运行称为*解释器*的程序，继而由解释器来运行特定的程序。此类程序通常比编译程序执行慢，因为解释器必须读取和理解（解释）代码。然而，对于好的解释器来说，这个时间可能是微不足道的。还有一类混合语言，它们会在执行二进制代码前对程序进行部分编译。这样做是因为解释器读取二进制代码的速度远远超过读取一般

的高级语言。

选择特定一种语言的原因有很多。编译程序不错，因为你无需先在用户机器上安装一个解释器。你必须有该语言的编译器，但你程序的用户则不需要。而对于解释语言，你必须确保用户机器上安装了用于解释你程序的解释器，而且计算机知道用哪个解释器来运行你的程序。但是解释语言往往更加灵活，而编译语言则更严格。

通常我们根据语言的可用工具和对编程方法的支持情况来选择语言，而不是根据某种语言是编译语言还是解释语言。事实上，许多语言既可以作为编译语言，也可以作为解释语言。

高级语言，无论是编译语言还是解释语言，都是为程序员服务，而不是为机器服务的。这使高级语言具备了各种功能，举例如下：

- 能够用一个表达式表示多个操作；
- 能够使用“大数值”——比计算机通常处理的4字节字大得多的值（例如，能够将文本字符串视为一个值，而不是一串字节）；
- 能获得较好的流控制结构，而不仅仅是使用跳转；
- 编译器能检查赋值的类型和其他断言；
- 使内存处理自动进行；
- 能够使用适宜某个问题领域的语言，而不是适宜计算机硬件的语言。

那么，人们选择某一种语言的依据是什么呢？例如，许多人选择Perl，因为其有几乎能处理现有的各种协议、各类数据的巨型函数库。一些人选择Python，因为其具有清晰的语法且适用于更直观的解决方案，其跨平台的GUI工具也非常强大。PHP使编写Web应用程序变得简单轻松。Common Lisp对于愿意学习它的人来说，比其他环境具备更强大、更多的功能。Scheme是一种简单又强大的模型。C更容易与其他语言交互。

每一种语言都各有千秋，懂得的语言越多，你就越胜任程序员的工作。了解不同语言的概念有助于你编写任何语言的程序，因为你可以选取更适合解决问题的编程语言，可以选取的工具集范围也更大。即使你使用的语言不直接支持某些功能，但也往往能模拟它们。不过，如果你没有关于编程语言的丰富经验，就无法了解可选择的所有可能性。

11.2　第一个 C 程序

这里是你的第一个C程序，它在屏幕上打印“Hello World”并退出。请键入这个程序，并将其命名为HELLO-World.c。

```
#include <stdio.h>

/* 目的：本程序旨在展示一个基本的C程序 */
/*       其功能只是在屏幕上打印 */
/*       “Hello World!”并退出 */

/* 主程序 */
int main(int argc, char **argv)
{
 /* 打印字符串到标准输出 */
 puts("Hello World!\n");

 /* 退出，状态码为0 */
 return 0;
}
```

如你所见，这是个很简单的程序。运行以下命令来编译该程序：

```
gcc -o HelloWorld Hello-World.c
```

输入以下命令以运行该程序：

```
./HelloWorld
```

让我们来看看这个程序的原理。

C语言的注释以/*开始，以*/结束。注释可以跨行，但是很多人喜欢在同一行上开始和结束注释，以避免混淆。

#include <stdio.h>是该程序的第一部分。这是一个预处理器指令。C编译分为两个阶段——预处理器和主编译器。该指令告诉预处理器寻找文件stdio.h，并将其粘贴到你的程序。预处理器负责将程序文本合并在一起，包括合并各个不同的文件，运行程序文本中的宏等。将文本合并后，预处理完成，主编译器开始工作。

现在，stdio.h的所有内容都在你的程序中，就像你自己输入了一样。文件名两边的尖括号告诉编译器到标准路径中寻找文件（通常是/usr/include和/usr/local/include）。如果文件名周围是双引号，比如#include："stdio.h"，编译器会在当前目录中寻找文件。stdio.h中包含标准输入、输出函数和变量的声明。这些声明告诉编译器哪些函数可用于输入和输出。接下来的几行内容只是关于程序的注释。

再接下来的一行是int main(int argc, char **argv)，是一个函数的起始行。C函数的声明包含函数名、参数和返回类型。这条声明表示函数名为main，返回一个int型值（整数，

在x86平台上为4字节长），有两个参数（一个int型参数，名为argc；一个char **型参数，名为argv）。你不必考虑这里的参数在栈中的位置，因为C编译器会负责处理。你也不必考虑加载到寄存器和从寄存器加载的值，因为编译器也会做此处理。

main函数是C语言的一个特殊函数，是所有C程序的开始之处（很像汇编语言程序中的_start）。这个函数总是需要两个参数，第一个参数是给予此命令的参数数目，而第二个参数是一个给定的参数列表。

下一行是一个函数调用。在汇编语言中，你必须将函数的参数入栈，然后再调用该函数。C负责处理这复杂的部分，你只需调用函数并给出括号中的参数。在本例中，我们调用函数puts，调用时带一个参数——就是我们要打印的字符串。我们只是必须用双引号括起输入的字符串，而编译器会在调用前处理好定义存储和移动指针到栈上的该存储区等一切事务。如你所见，少了很多工作。

最后，我们的函数返回数字0。在汇编语言中，我们将返回值存储在%eax中。但在C中，我们使用return命令即可，这条命令会处理所需的一切事务。main函数的返回值就是程序的退出码。

如你所见，使用高级语言大大方便了程序员的工作，也更便于我们的程序在多个平台上运行。使用汇编语言时，你的程序是与操作系统和硬件平台息息相关的，而使用编译语言和解释语言时，相同的代码通常可以运行在多种操作系统和硬件平台上。例如，这个程序可以在运行Linux、Windows、UNIX或其他大多数操作系统的x86硬件上生成和执行，也可以在运行有各种操作系统的Macintosh硬件上运行。

关于C编程语言的其他信息，请参见附录E。

11.3　Perl

Perl是一种解释语言，多用于Linux和基于UNIX的平台。尽管Perl实际上能在几乎所有平台上运行，但你通常总是在Linux和基于UNIX的平台上遇到它。以下是Perl版本的程序，输入程序后文件名应为Hello-World.pl：

```
#!/usr/bin/perl

print("Hello world!\n");
```

由于Perl是解释语言，无需编译或链接，你只需要输入以下命令并运行它：

```
perl Hello-World.pl
```

如你所见，Perl版本的程序比C版本更简短。使用Perl语言时，你无需声明任何函数或程序入口点，直接键入命令即可，解释器将运行命令。事实上，这个程序只有两行代码，其中一行是可选的。

第一行是可选代码，用于告诉UNIX机器用哪种解释器来运行程序。#!告诉计算机这是一个解释程序，/usr/bin/perl告诉计算机使用程序/usr/bin/perl来解释程序。但由于我们通过输入perl Hello-World.pl来运行该程序，实际上就已经指定了使用Perl解释器。

下一行代码调用Perl的内置打印函数，该函数有一个参数，即要打印的字符串。该程序没有显式`return`语句，但当其运行到文件末尾时会自动返回。同时，如果运行没有任何错误它就会返回0。你会发现，解释语言往往注重让你在不做额外工作的情况下，尽快获得工作代码。

Perl有一个特点并未很好地体现在这个例子中，即Perl是将字符串作为单个值进行处理的。在汇编语言中，我们必须根据计算机的内存体系结构来编程，这意味着字符串被处理为多个值的序列，指针指向第一个字母。Perl则假装字符串是可以直接存储的值，隐藏了字符串处理的复杂性。事实上，Perl的一个主要优势是处理文本的强大能力和快捷速度。

11.4 Python

Python版的程序看起来几乎与Perl版的完全一样。然而，Python语言实际上与Perl完全不同，尽管从这个简单的例子上似乎看不出来。输入以下程序到文件，文件名设为Hello-World.py：

```
#!/usr/bin/python

print "Hello World"
```

你应该能说明程序中各行代码的作用。

11.5 温故知新

11.5.1 理解概念

- ❑ 解释语言和编译语言之间的区别是什么？
- ❑ 你为什么需要学习一种新的编程语言？

11.5.2 应用概念

- ❑ 学习一种新编程语言的基本语法，用这种语言重新编码本书中的某个程序。

- 对于上题中编写的程序，有哪些细节是你选择的编程语言自动处理的？
- 修改你的程序，让其分别以新编程语言和汇编语言连续运行10 000次。然后，试运行`time`命令查看哪种语言更快。结果显示哪种语言更快？你认为原因是什么？
- 编程语言的输入/输出方法与Linux的系统调用有何不同？

11.5.3　深入学习

- 当看到有如Perl这样的简洁语言后，你认为本书为什么要从汇编语言这样一种烦琐的语言开始教授编程？
- 你认为高级语言对编程过程有何影响？
- 你认为为什么会存在这么多编程语言？
- 学习两种新的高级语言。试分析两者有何异同？它们分别采取哪种方式来解决问题？

第 12 章 优　化

优化是使应用程序运行更有效的过程。你可以对多方面进行优化，如速度、内存空间使用情况、磁盘空间使用情况等。本章的侧重点是速度优化。

12.1 何时优化

根本不采取优化胜过仓促优化。优化通常会使代码变得难以理解，因为优化会使代码更加复杂。代码的读者会很难领会你的代码为何要做某件事，这将增加项目的维护成本。即使你知道自己的程序为何这么做，以及是采取哪种方式完成的，优化后的代码也更难调试和扩展。这会大大拉长开发过程，一方面是优化代码花费的时间，一方面是花费在修改已优化代码上的时间。

雪上加霜的是，你事先甚至不知道程序的速度问题会在哪里。即使是经验丰富的程序员也无法预测程序的哪一部分是需要优化的瓶颈，所以你很可能会把时间浪费在优化错误的部分上。12.2节将讨论如何确定程序中需要优化的部分。

开发程序时必须关注以下重点：

- 对所有事做好文档记录；
- 确保每一件事都是按文档记录完成；
- 代码以模块的形式编写，易于修改。

文档记录是必不可少的，尤其是以小组形式协同进行开发工作时。程序必须能正常运作。你一定注意到了，该列表上并未列出应用程序的速度。在早期开发过程中优化并非必需进行的操作，原因如下：

- 轻微的速度问题通常可以通过硬件来解决，这往往比占用程序员的时间更经济；

- 当修改程序时，应用程序很可能会发生很大变化，因此会浪费你为优化作出的所有努力；[①]
- 速度问题通常位于代码中的少数位置，而在完成大部分程序前很难找到这些地方。

因此，我们往往在开发即将结束时进行优化，这时已经确定代码的正确性，也确定了代码确实存在性能问题。

在我参与的一个基于Web的电子商务项目中，我的重点完全放在保证程序正确性上。我的同事对此很是失望，他们烦恼的是每个页面在开始加载前都要花费12 s的处理时间（大多数网页处理过程不超过1 s）。但我下定决心首先确保程序的正确性，把优化视作最后需处理的事。当辛苦工作3个月终于确保代码正确性后，我们只花了3天来找到和消除瓶颈，使网页的平均处理时间低于0.25 s。通过遵循正确的做事顺序，我得以正确高效地完成了一个项目。

12.2　优化何处

一旦确定存在性能问题，你就需要找到问题代码。为此，你可以运行探查器（profiler）。探查器可以告诉你程序运行时用于每个函数的时间，以及每个函数的运行次数。

gprof是标准GNU/Linux分析工具，但讨论如何使用探查器不在本书讲述范围之内。探查器运行后，你就能确定哪些函数调用得最多、占据最多的时间，它们就是你应该着力优化的部分。

如果一个程序只花1%的时间在某个函数上，那么无论加速多少，整体速度提高最多也不过1%。但如果一个程序要在某个函数上用去20%的时间，那么即使对该函数改进不大，整体速度也会有明显提高。因此，利用探查器进行分析能给你所需的信息，让你能够正确选择要优化的部分。

为了优化函数，你需要了解它们的调用和使用方式。你对函数调用时机和方式了解越多，就越能作出合理优化。

优化主要有两类：局部优化和全局优化。局部优化包括针对特定硬件的优化（如以最快的方式执行特定计算）和针对特定程序的优化（如针对出现频率最高的情况编制最佳性能的一段代码）。全局优化又由结构性优化构成。例如，若尝试找到3个在不同城市的人在圣路易斯会面的最佳方式，局部优化会找到到圣路易斯的最佳路径，而全局优化则会决定采取电视电话会议的形式，以取代3人亲自出席会议的形式。全局优化往往通过结构重组避免性能问题，而不是试图找到解决性能问题的最佳方式。

---

① 对于许多新的项目来说，当开发人员深入了解了尝试解决的问题后，往往会完全重写最初使用的代码库。这样一来，对第一个代码库的任何优化都是浪费。

12.3 局部优化

以下的一些代码片段优化方法广为人知。高级语言编译器的优化器可能会自动完成其中某些优化。

- 预先计算

有时，函数的输入输出的可能组合数量有限，因此你实际上可以预先计算所有可能的答案，然后在函数被调用时找出预先计算好的答案。当然，由于必须存储所有的答案，这样做会占用一些空间，但对于小数据集，这个方式很有效，对于需要很长时间的计算来说更是如此。

- 记住计算结果

这与上一个方法类似，区别在于不是事先计算好结果，而是存储所请求的每个计算的结果。这样，当函数开始时，如果某个结果之前已经计算过，函数会直接返回以前的答案，否则就进行完整的计算，并存储计算结果供以后查询。这样做的好处是需要的存储空间较少，因为并未预先计算所有结果。有时，这称为*缓存*或*记忆*。

- 局部性原理

*局部性原理*这个术语指代你正在访问的数据项在内存中的存储区域。使用虚拟内存，你可以访问存储在磁盘上的内存页。在这种情况下，操作系统必须从磁盘上加载该内存页，并将其他内存页卸载到磁盘。例如，假设操作系统允许你使用20 K物理内存，迫使其余数据存储在磁盘上，而你的应用程序要使用60 K内存。再假设你的程序要对每一块数据进行5个操作。如果一个操作依次对每一块数据进行处理，完成后进行下一个操作，再依次对每一块数据进行处理，最终每一页数据将被从磁盘加载5次并卸载到磁盘5次。但如果你采用另一种方式，对每一个给定数据项依次进行5个操作，然后再外理下一个数据项，那么只需要从磁盘中加载每个页面一次。而将在物理内存中位置接近的操作和数据尽可能多地绑定时，你就能利用局部性原理的优势了。此外，处理器通常将芯片上的一些数据存储在缓存中。如果将所有操作限定在物理内存的一个较小范围内，你的程序甚至可能会绕过主内存，只使用芯片上的超高速缓存。这一切都已经为你做好了，你只需要尝试一次只使用小块内存，而不是大片使用整个内存。

- 使用寄存器

寄存器是计算机上访问最快的内存位置。当你存取内存时，处理器必须等待数据装入内存总线。然而，寄存器本身位于处理器上，因此访问速度极快。因此，合理使用寄存器极其重要。如果你正在使用的数据项不多，请尝试将它们全部存储在寄存器中。但并非所有高级语言都提供这个选项，因为编译器会决定哪些数据存储在寄存器中，而哪些数据不存储在寄存器中。

- 内联函数

从程序管理的角度来看，函数有很多优点：使程序更易划分成相互独立、易理解、可重复使用的部分。然而，函数调用涉及参数入栈和跳转的开销。（记得局部性原理吗？你的代码可能会被切换到磁盘上，不再驻留内存。）对于高级语言，编译器往往不可能跨函数调用边界进行优化。但有些语言支持内联函数或函数宏。这些函数的外观和行为都像真正的函数，区别在于有编译器选项可将这些代码加入调用这些函数的地方。这使得程序运行更快，但也增加了代码尺寸。也有许多函数（如递归函数）不能内联，因为这类函数直接或间接调用自身。

- 优化指令

很多时候，多个汇编语言指令可以实现相同的目的。老练的汇编语言程序员知道哪些指令最快。但对于不同的处理器，最快的指令可能并不相同。要了解更多相关信息，请参阅你正在使用的芯片相应的用户手册。我们来看看加载数字0到寄存器的过程。在大多数处理器上，执行`movl $0, %eax`并非最快的方式，最快的方法是将寄存器与其本身执行异或运算，即`xorl %eax, %eax.`，因为该指令只访问寄存器，且不传输任何数据。对于高级语言用户来说，编译器会帮忙处理这种优化，但汇编语言程序员需要非常了解自己的处理器。

- 寻址方式

不同的寻址方式速度不同。最快的是立即寻址方式和寄存器寻址方式，直接寻址方式次之，间接寻址方式再次之，基址寻址方式和索引间接寻址方式最慢。请尽可能使用较快的寻址方式。有趣的是，当你用基址寻址方式访问一块结构性内存时，访问第一个元素最快。由于其偏移量为0，你可以使用间接寻址方式代替基址寻址，这样会更快。

- 数据对齐

某些处理器访问具备字对齐内存边界（即地址能被字的大小整除）的数据时，速度比访问非对齐数据时快。所以，当在内存中设置结构时最好保持字对齐。事实上，一些非x86处理器用某些寻址方式根本不能访问非对齐的数据。

这只是少量可能产生局部优化的例子。但是请记住，除非在极端情况下，否则可维护性和代码的可读性比优化重要得多。

12.4　全局优化

全局优化有两个目标。第一个是把代码转换成更容易进行局部优化的形式。例如，假设你有一个执行缓慢、复杂计算的大过程，可能会尝试将过程的某些部分划分成数个值可预计算或记忆

存储的函数。

无状态函数（只对传递给它的参数进行操作，没有全局变量或系统调用）是计算机上最易进行优化的函数类型。程序的无状态函数部分越多，就越有得到优化的机会。在我上面提到的电子商务项目中，计算机必须找到具体的存货项目的所有相关部分。这需要大约12个数据库调用，在最坏的情况要用大约20 s。然而，这个程序的目标是交互性，而漫长的等待会破坏这一目标。但我知道，这些库存配置不改变。因此，我将各数据库调用转换成无状态函数，这样我就能记忆这些函数了。在每天开始时，为防止有人改变函数结果并造成影响，我们清除这些结果，并自动预装一些库存项目。此后每个白天，当某人首次访问库存项目时，要花费20 s时间，但之后再访问时由于数据库结果已经被记忆，所用时间不到1 s。

全局优化通常包括在函数中实现下列特性。

- *并行化*

并行化意味着算法可以高效划分给多个进程。例如，怀孕不可并行，因为无论有多少女人，怀孕仍然需要9个月。但制造一辆汽车是可并行的，因为可以让一个工人制造发动机，另一个制造车内部分。通常情况下，应用程序的并行性有一定限制。应用程序并行性程度越高，其对多处理器和集群计算机配置的利用程度就越高。

- *无状态*

正如我们已经讨论过的，无状态函数和程序完全依靠显式传递给它们的数据进行运作。大多数进程并非完全无状态，而是在一定限度内无状态。在电子商务示例中，函数并非完全无状态，而是在白天无状态。因此，我就像对待无状态函数那样优化它，但留下夜间修改的余地。无状态函数有两大优势，即多数无状态函数是可并行的，且常常受益于记忆。

全局优化需要相当的经验才能知道什么可行，什么不可行。判断如何解决代码的优化问题涉及总览所有问题，并知道修正某些问题可能会导致其他哪些问题。

12.5 温故知新

12.5.1 理解概念

- ❑ 在优先级方面，与其他编程任务相比，优化的重要性如何？
- ❑ 局部优化和全局优化之间的区别是什么？
- ❑ 试说出一些局部优化的类型。

- 你如何确定需要优化程序的哪些部分？
- 在优先级方面，与其他编程任务相比，优化的重要性如何？你知道我为什么重复这个问题吗？

12.5.2　应用概念

- 回顾本书中的每个程序，并尝试按照本章中概述的优化过程进行优化。
- 从前面的练习中选择一个程序，试计算某些特定输入对代码产生的性能影响。[①]

12.5.3　深入学习

- 寻找一个你觉得特别快的开源程序，联系开发人员询问他们对程序进行了何种优化以提高其速度。
- 寻找一个你觉得特别慢的开源程序，设想一下哪些原因导致程序运行缓慢。然后，请下载代码并尝试用`gprof`或类似的工具来分析程序。试查找代码中耗费大部分时间的部分，并对其进行优化。对比程序缓慢的原因是否与你最初设想的原因相同。
- 有了编译器，我们是否无需再进行局部优化？为什么？
- 编译器尝试跨函数调用边界优化函数时会遇到哪种问题？

① 因为这些程序通常很短，所以不会出现明显的性能问题，通过循环该程序数千次，我们可以将性能问题放大到足以计算其影响。

第13章 学无止境

恭喜已经学习到这里，你现在应该对编程设计的许多领域都有了最基本的了解。即使你再也不会用到汇编语言，但从学习中获得的宝贵观点、建立的基本理念也有助于你理解其他计算机科学知识。

学习编程的方法主要有以下3种。

- **自下而上** 这正是本书所教授的。这种方法从低级语言的编程开始，向更通用的教学内容迈进。
- **自顶向下** 这与第一种方法正好相反。这种方法的侧重点在于你想要用计算机做什么，并讲述如何细分任务直到无法再细分。
- **从中间开始** 这种方式的代表是讲述某种编程语言或API的图书。这类图书关注具体细节，不涉及概念。

不同的人喜欢不同的学习方法，但一名优秀的程序员需要全面考虑这些方法。自下而上的方法可以帮助你了解机器层面的知识，自顶向下的方法可以帮助你了解问题的各方面，从中间开始的方法可以在实际问题和答案上对你有所帮助。忽视任何方面都是错误的。

计算机编程是一个浩大的工程。程序员要准备好不断学习，超越自身，而计算机图书将帮助你做到这一点。它们不仅讲述各自的主题，还教授各种思维方法。正如Alan Perlis所说："不会影响你编程思维的语言不值得学习。"（参见http://www.cs.yale.edu/homes/perlis-alan/quotes.html。）如果你不断寻找编程和思考的更好方法，你会成为一名成功的程序员。如果你不设法提高自己，"再睡片刻，打盹片时，抱着手躺卧片时，你的贫穷就必如强盗速来，你的缺乏仿佛拿兵器的人来到。"（《箴言》，24:33-34 NIV。）也许没那么严重，但是勤学不辍总是好的。

我选择推荐这些书是因为其内容，也是因为它们在计算机科学界受到推崇。其中每一本书都有其独特之处。由于这里图书众多，你最好通过网上评论从你最感兴趣的那本书开始阅读。

13.1　自下而上

这张书单是我认为的最佳阅读顺序，不一定是从最简单到最难的罗列，而是基于主题的。

- 《深入理解程序设计》，Jonathan Bartlett著。
- 《算法导论》，Thomas H. Cormen、Charles E. Leiserson和Ronald L. Rivest著。
- 《计算机程序设计艺术》（我认为第1卷最重要），Donald Knuth著。
- *Programming Languages*，Samuel N. Kamin著。
- 《现代操作系统》，Andrew Tanenbaum著。
- 《链接器和加载器》，John Levine著。
- 《计算机组织与设计：硬件/软件接口》，David Patterson和John Hennessy著。

13.2　自顶向下

这些书按从易到难顺序排列，但你可以按照最适合自己的顺序阅读。

- 《程序设计方法》，Matthias Felleisen、Robert Bruce Findler、Matthew Flatt和Shiram Krishnamurthi著，可在http://www.htdp.org/在线阅读。
- *Simply Scheme: An Introduction to Computer Science*，Brian Harvey和Matthew Wright著。
- 《像计算机科学家一样思考Python》，Allen Downey、Jeff Elkner和Chris Meyers著，可在http://www.greenteapress.com/thinkpython/在线阅读。
- 《计算机程序的构造和解释》，Harold Abelson、Gerald Jay Sussman和Julie Sussman著，可在http://mitpress.mit.edu/sicp/在线阅读。
- 《设计模式：可复用面向对象软件的基础》，Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides著。
- *Why not How: The Rules Approach to Application Development*，Chris Date著。
- 《算法设计手册》，Steve Skiena著。
- 《程序设计语言——实践之路》，Michael Scott著。
- *Essentials of Programming Languages*，Daniel P. Friedman、Mitchell Wand和Christopher T. Haynes著。

13.3　从中间开始

以下书单中每本书都是相关主题的佳作。如果你需要学习这些语言，这些书足以满足你的需要。

- 《Perl语言编程》，Larry Wall、Tom Christiansen和Jon Orwant著。
- *Common LISP: The Language*，Guy R. Steele著。
- *ANSI Common LISP*，Paul Graham著。
- 《C程序设计语言》，Brian W. Kernighan和Dennis M. Ritchie著。
- *The Waite Group's C Primer Plus*，Stephen Prata著。
- 《C++程序设计语言》，Bjarne Stroustrup著。
- 《Java编程思想》，Bruce Eckel著，可在http://www.mindview.net/Books/TIJ/在线阅读。
- *The Scheme Programming Language*，Kent Dybvig著。
- *Linux Assembly Language Programming*，Bob Neveln著。

13.4 专题

下面这些书都是关于各自主题的佳作，解释透彻，颇具权威性。为了获得广泛的基础知识，你应该在自己通常编程范围外阅读其中几本。

- 实用编程——《编程珠玑》和《编程珠玑（续）》，Jon Louis Bentley著。
- 数据库——*Understanding Relational Databases*，Fabian Pascal著。
- 项目管理——《人月神话》，Fred P. Brooks著。
- UNIX编程——《UNIX编程艺术》，Eric S. Raymond著，可在http://www.catb.org/~esr/writings/taoup/在线阅读。
- UNIX编程——《UNIX环境高级编程》，W. Richard Stevens著。
- 网络编程——《UNIX网络编程（卷1）：套接字联网API》和《UNIX网络编程（卷2）：进程间通信》，W. Richard Stevens著。
- 泛型编程——《C++设计新思维——泛型编程与设计模式之应用》，Andrei Alexandrescu著。
- 编译器——《编译程序设计艺术：理论与实践》，Thomas Pittman和James Peters著。
- 编译器——《高级编译器设计与实现》，Steven Muchnick著。
- 开发流程——《重构：改进既有代码的设计》，Martin Fowler、Kent Beck、John Brant、William Opdyke和Don Roberts著。
- 排版——*Computers and Typesetting*（5卷），Donald Knuth著。
- 加密——《应用密码学》，Bruce Schneier著。
- Linux——*Professional Linux Programming*，Neil Matthew、Richard Stones和其他14人著。
- Linux内核——《Linux设备驱动程序》，Alessandro Rubini和Jonathan Corbet著。
- 开源编程——*The Cathedral and the Bazaar: Musings on Linux and Open Source*，Eric S. Raymond著。

- 计算机体系结构——《计算机体系结构：量化研究方法》，David Patterson和John Hennessy著。

13.5　汇编语言的更多资源

就汇编语言而言，最佳资源都在网上。

- http://www.linuxassembly.org/为Linux汇编语言程序员提供了大量资源。
- http://www.sandpile.org/——x86、x86-64以及其他兼容处理器的参考资料库。
- http://www.x86.org/——Dobb博士微处理器资源期刊。
- http://www.drpaulcarter.com/pcasm/——Paul Carter博士的PC汇编语言页面。
- http://webster.cs.ucr.edu/——汇编艺术主页。
- http://www.intel.com/design/pentium/manuals/——英特尔处理器用户手册。
- http://www.janw.easynet.be/——Jan Wagemaker的Linux汇编语言示例。
- http://www.azillionmonkeys.com/qed/asm.html——Paul Hsieh的x86汇编页面。

附录 A

GUI编程

A

A.1 GUI编程简介

本附录的目的不是教你编写GUI（图形用户界面），而仅仅是说明编写图形应用程序与编写其他应用程序一样，只是需要使用一个额外的库来处理图形部分。程序员需要对学习使用新库习以为常。大部分时间你都将在不同库之间传递数据。

A.2 GNOME库

GNOME项目是向Linux用户提供完整桌面系统的项目之一。GNOME项目包括一个（含应用程序启动器和称为applet的迷你应用程序的）面板、几个（完成文件管理、会话管理、配置等任务的）标准应用程序，以及一个用于创建适用于与系统其余部分协同工作的应用程序的API。

关于GNOME库要注意一点，即GNOME库不断地创建并给予你指向大数据结构的指针，但你永远不需要知道它们是如何在内存分布的。对GUI数据结构的全部操作都是通过函数调用来完成的。这是出色库设计的一个特点。各个版本的库有所不同，所以其数据结构保存的数据也不尽相同。如果你必须自己访问和操作数据，当库更新时，你将不得不修改自己的程序，使之适合新库，或者至少需要重新编译程序。当你通过函数访问数据时，函数会知道结构中每条数据的位置。你从库中接收到的指针是不透明的，即你无需知道指针指向的结构是什么样的，只需要知道能正确处理结构的函数。当设计库时，即使只用于一个程序，这也是一种值得提倡的做法。

本章不会详细介绍GNOME的工作原理，因此如果你想了解更多信息，请访问GNOME开发者网站http://developer.gnome.org/。这个站点包含教程、邮件列表、API文档，以及在GNOME环境下开始编程需要了解的一切内容。

A.3　用几种语言编写的简单GNOME程序

本程序只是显示一个窗口，窗口上有一个按钮，单击该按钮会退出应用程序。当单击该按钮时，程序会问你是否确定要退出，如果你单击“是”，应用程序就会关闭。若要运行此程序，请键入以下程序（文件名为gnome-example.s）：

```
# 目的：本程序是一个示例，展示利用GNOME库编写的GUI程序
#
# 输入：用户仅能单击“退出”按钮或关闭窗口
#
# 输出：应用程序将关闭
#
# 处理过程：如果用户单击“退出”按钮，程序
#           将显示一个询问用户是否确定退出的对话框
#           如果用户确定，就单击Yes按钮，程序将关闭
#           否则，程序将继续运行
#

.section .data

###GNOME定义 - 这些内容可在C语言的GNOME头文件中找到，
#              它们被转换成等价的汇编语言

# GNOME按钮名
GNOME_STOCK_BUTTON_YES:
.ascii "Button_Yes\0"
GNOME_STOCK_BUTTON_NO:
.ascii "Button_No\0"

# Gnome消息框类型
GNOME_MESSAGE_BOX_QUESTION:
.ascii "question\0"

# NULL的标准定义
.equ NULL, 0

# GNOME信号定义
signal_destroy:
.ascii "destroy\0"
signal_delete_event:
.ascii "delete_event\0"
signal_clicked:
.ascii "clicked\0"
```

```
###应用程序具体定义

# 应用程序信息
app_id:
.ascii "gnome-example\0"
app_version:
.ascii "1.000\0"
app_title:
.ascii "Gnome Example Program\0"

# 按钮与窗口的文本
button_quit_text:
.ascii "I Want to Quit the GNOME Example Program\0"
quit_question:
.ascii "Are you sure you want to quit?\0"

.section .bss

# 存储所创建的小组件的变量
.equ WORD_SIZE, 4
.lcomm appPtr, WORD_SIZE
.lcomm btnQuit, WORD_SIZE

.section .text

.globl main
.type main,@function
main:
 pushl %ebp
 movl  %esp, %ebp

 # 初始化GNOME库
 pushl 12(%ebp)              # argv
 pushl 8(%ebp)               # argc
 pushl $app_version
 pushl $app_id
 call  gnome_init
 addl  $16, %esp             # 恢复栈

 # 创建新应用程序窗口
 pushl $app_title            # 窗口标题
 pushl $app_id               # 应用程序ID
 call  gnome_app_new
 addl  $8, %esp              # 恢复栈
```

```
movl  %eax, appPtr          # 保存窗口指针

# 创建新按钮
ushl $button_quit_text      # 按钮文本
call gtk_button_new_with_label
addl $4, %esp               # 恢复栈
movl %eax, btnQuit          # 保存按钮指针

# 设定按钮显示在应用程序窗口内
pushl btnQuit
pushl appPtr
call  gnome_app_set_contents
addl  $8, %esp

# 显示按钮（但要在窗口显示后再显示按钮）
pushl btnQuit
call  gtk_widget_show
addl  $4, %esp

# 显示应用程序窗口
pushl appPtr
call  gtk_widget_show
addl  $4, %esp

# 每当delete事件发生时，让GNOME调用
# 我们的delete_handler函数
pushl $NULL                 # 将额外数据传递给我们的函数（尽管不使用这些数据）
pushl $delete_handler       # 要调用的函数地址
pushl $signal_delete_event  # 信号名
pushl appPtr                # 用于监听事件的小组件
call  gtk_signal_connect
addl  $16, %esp             # 恢复栈

# 每当destroy事件发生时，就让GNOME调用
# 我们的destroy_handler函数
pushl $NULL                 # 将额外数据传递给我们的函数（尽管不使用这些数据）
pushl $destroy_handler      # 要调用的函数地址
pushl $signal_destroy       # 信号名
pushl appPtr                # 用于监听事件的小组件
call  gtk_signal_connect
addl  $16, %esp             # 恢复栈

# 每当click事件发生时，就让GNOME调用
# 我们的click_handler函数。注意，之前对于
# 应用程序窗口监听信号，这次则对于按钮监听
```

```
 # 信号
 pushl $NULL
 pushl $click_handler
 pushl $signal_clicked
 pushl btnQuit
 call  gtk_signal_connect
 addl  $16, %esp

 # 将控制权转给GNOME。从这里开始发生的一切都是对用户事件的应对，
 # 包括调用信号处理程序。这个主函数只是设置主窗口，并将之与信号
 # 处理程序连接，由信号处理程序处理剩余的一切事务
 call gtk_main

 # 程序结束后离开
 movl $0, %eax
 leave
 ret

# 当移除小组件时发生destroy事件。 在本例中，
# 当移除应用程序窗口时，我们只需要事件跳转至退出
destroy_handler:
 pushl %ebp
 movl  %esp, %ebp

 # 这会导致gtk尽快跳出其事件循环
 call gtk_main_quit

 movl $0, %eax
 leave
 ret

# 当有人单击应用程序窗口的“×”按钮时，发生delete
# 事件。该按钮通常用于关闭窗口
delete_handler:
 movl $1, %eax
 ret

# 当组件被单击时发生click事件
click_handler:
 pushl %ebp
 movl  %esp, %ebp

 # 创建“你是否确定”对话框
 pushl $NULL # 按钮结束
 pushl $GNOME_STOCK_BUTTON_NO       # 按钮1
```

```
    pushl $GNOME_STOCK_BUTTON_YES      # 按钮0
    pushl $GNOME_MESSAGE_BOX_QUESTION # 对话框类型
    pushl $quit_question               # 对话框消息
    call  gnome_message_box_new
    addl  $16, %esp                    # 恢复栈
    # %eax现在保存指向对话框窗口的指针

    # 将窗口模式设为1，以防止其他用户在对话框显示时与之交互
    pushl $1
    pushl %eax
    call  gtk_window_set_modal
    popl  %eax
    addl  $4, %esp

    # 现在显示对话框
    pushl %eax
    call  gtk_widget_show
    popl  %eax

    # 以上操作会设置所有必要的信号处理程序，目标是
    # 只显示对话框，当其中一个按钮被单击时关闭窗口，
    # 并返回用户单击的按钮编号。按钮编号是根据
    # gnome_message_box_new函数中的入栈顺序确定的
    pushl %eax
    call gnome_dialog_run_and_close
    addl $4, %esp

    # 按钮0是Yes按钮。如果用户单击此按钮，
    # 就告诉GNOME退出事件循环，否则什么都不做
    cmpl $0, %eax
    jne  click_handler_end

    call gtk_main_quit

click_handler_end:
    leave
    ret
```

要生成此应用程序，请执行以下命令：

```
as gnome-example.s -o gnome-example.o
gcc gnome-example.o `gnome-config --libs gnomeui` \
    -o gnome-example
```

接着键入./gnome-example以运行该程序。

这个程序与多数GUI程序类似，大量将指针作为参数传递给函数。在本程序中，你利用GNOME函数创建组件，然后设置当某些事件发生时函数被调用，这些函数就称为回调函数。所有事件处理都由函数gtk_main进行，因此你无需关注事件是如何处理的，只需要设置好回调函数然后等待事件发生。

下面是对本程序中用到的GNOME函数的简短描述。

- gnome_init

传入参数为命令行参数、参数数量、应用程序ID以及应用程序版本号，功能为初始化GNOME库。

- gnome_app_new

创建一个新应用程序窗口，并返回一个指向该窗口的指针。传入参数为应用程序ID和窗口标题。

- gtk_button_new_with_label

创建一个新按钮，并返回一个指向该按钮的指针。有一个传入参数——按钮文本。

- gnome_app_set_contents

本函数的传入参数为指向gnome应用程序窗口的指针，以及指向任何你想要的组件（在本例中为一个按钮）的指针，功能为使该组件成为窗口的内容。

- gtk_widget_show

本函数必须在创建每个组件（应用程序窗口、按钮、文本输入框等）时调用以显示这些组件。但为了显示组件，首先必须显示其所有父控件。

- gtk_signal_connect

本函数绑定组件及其信号处理回调函数。本函数的传入参数为组件指针、信号名、回调函数及一个额外的数据指针。在此函数被调用后，每当给定事件被触发，回调函数就被调用，传给该回调函数产生信号的组件及额外的数据指针。在本应用程序中，我们不使用额外数据指针，因此将这个值设置为NULL，也就是0。

- gtk_main

这个函数使GNOME进入主循环。为简化应用程序编程，GNOME为我们处理主循环。GNOME将检查事件，并在事件发生时调用相应的回调函数。在信号处理程序调用gtk_main_quit之前，这一函数将继续处理事件。

- gtk_main_quit

本函数使GNOME尽快退出主循环。

- gnome_message_box_new

本函数创建一个包含问题和交互按钮的对话框。其参数为要显示的信息、信息类型（警告、问题等）、要显示的按钮列表。最后一个参数为NULL，表示没有更多要显示的按钮了。

- gtk_window_set_modal

本函数设置给定窗口为模态窗口。在GUI编程中，模态窗口确保本窗口关闭前其他窗口的事件不被处理。它往往用于对话框窗口。

- gnome_dialog_run_and_close

本函数参数为指向对话框的指针（gnome_message_box_new函数返回的指针可用于此处），将设置所有相应信号处理程序，这样函数将保持运行，直到某个按钮被按下，然后关闭对话框，并返回关于按下的是哪个按钮的信息。按钮编号就是在gnome_message_box_new函数中设置按钮的顺序。

下面是以C语言编写的同一程序。输入以下内容，并将文件命名为gnome-example-c.c：

```
/* 目的：本程序一个示例，展示利用GNOME库编写的GUI程序
 */

#include <gnome.h>

/* 程序定义 */
#define MY_APP_TITLE "Gnome Example Program"
#define MY_APP_ID "gnome-example"
#define MY_APP_VERSION "1.000"
#define MY_BUTTON_TEXT "I Want to Quit the Example Program"
#define MY_QUIT_QUESTION "Are you sure you want to quit?"

/* 必须在用到函数前声明函数 */
int destroy_handler(gpointer window,
  GdkEventAny *e,
  gpointer data);
int delete_handler(gpointer window,
  GdkEventAny *e,
  gpointer data);
int click_handler(gpointer window,
```

```
  GdkEventAny *e,
  gpointer data);

int main(int argc, char **argv)
{
 gpointer appPtr; /* 应用程序窗口 */
 gpointer btnQuit; /* 退出按钮 */

 /* 初始化GNOME库 */
 gnome_init(MY_APP_ID, MY_APP_VERSION, argc, argv);

 /* 创建新应用程序窗口 */
 appPtr = gnome_app_new(MY_APP_ID, MY_APP_TITLE);

 /* 创建新按钮 */
 btnQuit = gtk_button_new_with_label(MY_BUTTON_TEXT);

 /* 将按钮设置在应用程序窗口中显示 */
 gnome_app_set_contents(appPtr, btnQuit);

 /* 使按钮显示 */
 gtk_widget_show(btnQuit);

 /* 使应用程序窗口显示*/
 gtk_widget_show(appPtr);

 /* 绑定信号处理程序 */
 gtk_signal_connect(appPtr, "delete_event",
   GTK_SIGNAL_FUNC(delete_handler), NULL);
 gtk_signal_connect(appPtr, "destroy",
   GTK_SIGNAL_FUNC(destroy_handler), NULL);
 gtk_signal_connect(btnQuit, "clicked",
   GTK_SIGNAL_FUNC(click_handler), NULL);

 /* 将控制权转移给GNOME */
 gtk_main();

 return 0;
}

/* 用于接收destroy信号的函数 */
int destroy_handler(gpointer window,
  GdkEventAny *e,
  gpointer data)
{
```

```
 /* 离开GNOME事件循环 */
 gtk_main_quit();
 return 0;
}

/* 用于接收delete_event信号的函数 */
int delete_handler(gpointer window,
  GdkEventAny *e,
  gpointer data)
{
 return 0;
}

/* 用于接收clicked信号的函数 */
int click_handler(gpointer window,
  GdkEventAny *e,
  gpointer data)
{
 gpointer msgbox;
 int buttonClicked;

 /* 创建“你是否确定”对话框 */
 msgbox = gnome_message_box_new(
  MY_QUIT_QUESTION,
  GNOME_MESSAGE_BOX_QUESTION,
  GNOME_STOCK_BUTTON_YES,
  GNOME_STOCK_BUTTON_NO,
  NULL);
 gtk_window_set_modal(msgbox, 1);
 gtk_widget_show(msgbox);

 /* 运行对话框 */
 buttonClicked = gnome_dialog_run_and_close(msgbox);

 /* 按钮0为Yes按钮，如果按下此按钮，就告诉GNOME退出其事件循环，
 否则什么也不做 */
 if(buttonClicked == 0)
 {
  gtk_main_quit();
 }

 return 0;
}
```

键入以下命令进行编译：

```
gcc gnome-example-c.c `gnome-config --cflags \
    --libs gnomeui` -o gnome-example-c
```

键入./gnome-example-c运行该程序。

最后，让我们来看Python版程序。输入并命名文件为gnome-example.py：

```
# 目的：本程序是一个示例，展示利用GNOME库编写的GUI程序
#

# 导入GNOME库
import gtk
import gnome.ui

####首先定义回调函数####

# 在Python中，函数必须先定义再使用
# 因此我们必须先定义自己的回调函数

def destroy_handler(event):
 gtk.mainquit()
 return 0

def delete_handler(window, event):
 return 0

def click_handler(event):
 # 创建“您是否确定”对话框
 msgbox = gnome.ui.GnomeMessageBox(
   “你是否确定退出？”,
  gnome.ui.MESSAGE_BOX_QUESTION,
  gnome.ui.STOCK_BUTTON_YES,
  gnome.ui.STOCK_BUTTON_NO)
  msgbox.set_modal(1)
 msgbox.show()

 result = msgbox.run_and_close()

 # 按钮0是Yes按钮。单击此按钮将告诉GNOME
 # 退出其事件循环，否则什么都不做
 if (result == 0):
  gtk.mainquit()

 return 0
```

```
####主程序####

# 创建新应用程序窗口
myapp = gnome.ui.GnomeApp(
 "gnome-example", "Gnome Example Program")

# 创建新按钮
mybutton = gtk.GtkButton(
 "I Want to Quit the GNOME Example program")
myapp.set_contents(mybutton)

# 显示按钮
mybutton.show()

# 显示应用程序窗口
myapp.show()

# 绑定信号处理程序
myapp.connect("delete_event", delete_handler)
myapp.connect("destroy", destroy_handler)
mybutton.connect("clicked", click_handler)

# 将控制权转移给GNOME
gtk.mainloop()
```

输入python gnome-example.py运行该程序。

A.4　GUI生成器

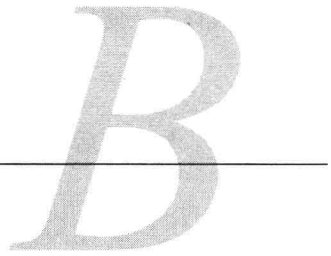
在前面的GUI示例中，我们通过为每个组件调用创建函数创建了应用程序的用户界面，并将之放置于合适的位置上。但是，对于较复杂的应用程序来说这种方式可能太麻烦。许多编程环境，包括GNOME，都含有称为GUI生成器的程序，能自动为你创建GUI。你只需编写信号处理程序和初始化程序的代码。GNOME应用程序的主要GUI生成器称为GLADE。GLADE适用于多数Linux版本。

多数编程环境都有GUI生成器。Borland有一系列工具，可在Linux和Win32系统上快速方便地生成GUI。KDE环境有一个称为QT Designer的工具，能帮助你自动开发GUI。

在开发图形应用程序上，我们有丰富的选择，希望本附录能让你对GUI编程略有了解。

附录 B

通用x86指令



B.1 阅读指令表

本附录中的指令表包括：

- 指令代码；
- 所用操作数；
- 所用标志；
- 简短描述指令功能。

操作数部分将列出可接受的所有操作数类型。如果操作数不止1个，就将每个操作数以逗号分隔。每个操作数都附带一列代码，说明操作数是否可以是立即模式值（I）、寄存器（R）或内存地址（M）。例如，`movl`指令的操作数表示为`I/R/M, R/M`，说明第一个操作数可以是任何类型值，而第二个操作数必须是寄存器或内存位置。但要注意，在x86汇编语言中，采用内存位置的操作数最多只能有一个。

标志部分列出`%eflags`寄存器中受指令影响的标志，并将涉及以下标志。

- `O`

溢出标志。如果目的操作数不够大，不足以表示指令结果，此标志为真。

- `S`

符号标志。此标志被设为最后一个结果的符号。

- `Z`

0标志。如果指令结果为0，此标志就为真。

- A

辅助进位标志。此标志不常用，根据第三位和第四位间的进位或借位情况设置。

- P

奇偶标志。如果最后一个结果的低位字节有偶数个1，此标志就为真。

- C

进位标志。用于在数学计算中表示结果是否要像另一个字节进位。如果设置了进位标志，通常就意味着目的寄存器不足以保存全部结果，由程序员决定要如何处理。（例如，将结果扩展至另一个字节、发出错误信号或完全忽略。）

还存在其他标志，但那些标志并不重要。

B.2　数据传输指令

这些指令很少用到，多数用于将数据从一个地方传输到另一个地方。

表B-1　数据传输指令

指　令	操 作 数	影响到的标志
movl	I/R/M, I/R/M	O/S/Z/A/C
本指令从一个内存位置复制一个字大小的数据到另一个内存位置。movl %eax, %ebx 将%eax的内容复制到%ebx		
movb	I/R/M, I/R/M	O/S/Z/A/C
作用与movl相同，但操作数为字节		
leal	M, I/R/M	O/S/Z/A/C
参数为标准格式中给定的内存位置，但并不加载内存位置的内容，而是加载计算得出的地址。例如，leal 5(%ebp,%ecx,1),%eax将计算得出的地址5 + %ebp + 1*%ecx加载到%eax		
popl	R/M	O/S/Z/A/C
将栈顶数据弹出至某个内存位置，相当于依次执行movl (%esp), R/M以及addl $4, %esp两条指令。popfl是弹出栈顶数据到%eflags寄存器的变种		
pushl	I/R/M	O/S/Z/A/C
将某个值入栈，相当于依次执行subl $4, %esp以及movl I/R/M, (%esp)两条指令。pushfl是将%eflags寄存器当前数据入栈的变种		
xchgl	R/M, R/M	O/S/Z/A/C
交换某个操作数的值		

B.3　整数指令

下面是对有符号整数及无符号整数进行操作的基本运算指令。

表B-2　整数指令

指　令	操　作　数	影响到的标志
adcl	I/R/M, R/M	O/S/Z/A/P/C
带进位加法。将进位位及第一个操作数与第二个操作数相加，如果存在溢出，就将溢出及进位标志设置为真。这通常用于大于一个机器字的操作。对于最低有效字的加法将使用addl，其他字的加法则使用adcl指令（以考虑前一个低位字加法的进位）。一般情况下我们不会使用本指令，而是使用addl		
addl	I/R/M, R/M	O/S/Z/A/P/C
加法。将第一个操作数与第二个操作数相加，将结果保存到第二个操作数。如果结果大于目的寄存器，那么溢出位及进位位设置为真。该指令可对有符号整数及无符号整数进行操作		
cdq		O/S/Z/A/P/C
将%eax中的字带符号扩展为由%edx:%eax组成的双字。q表明这是一个**四字**，但实际上是双字，之所以叫四字是因为这个术语是在使用16位字的时候出现的。这条指令通常用在发出idivl指令之前		
cmpl	I/R/M, R/M	O/S/Z/A/P/C
比较两个整数。本指令将第二个操作数减去第一个操作数，舍弃结果，但根据差设置标志位。通常用于条件跳转前		
decl	R/M	O/S/Z/A/P
将寄存器或内存位置的数据减1，如果对字节（而非字）操作，就使用decb指令		
divl	R/M	O/S/Z/A/P
执行无符号除法。将%edx:%eax所含的双字除以指定寄存器或内存位置的值。运算后，%eax包含商，%edx包含余数。如果商对于%eax来说过大，导致溢出，将触发中断0		
idivl	R/M	O/S/Z/A/P
执行有符号除法。操作类似于divl		
imull	R/M/I, R	O/S/Z/A/P/C
执行有符号乘法，将结果保存到第二个操作数。如果第二个操作数空缺，就默认为%eax，且完整的结果将存储在%edx:%eax中		
incl	R/M	O/S/Z/A/P
递增给定寄存器或内存位置，如果对字节（而非字）操作，就使用incb 指令		
mull	R/M/I, R	O/S/Z/A/P/C
执行无符号乘法，运算规则与imull相同		
negl	R/M	O/S/Z/A/P/C
将给定寄存器或内存位置的内容求补（二进制求补）		
sbbl	I/R/M, R/M	O/S/Z/A/P/C
借位减法，与adc用法相同，区别在于本指令为减法。通常使用subl		
subl	I/R/M, R/M	O/S/Z/A/P/C
将两个操作数相减，用第二个操作数减去第一个操作数，将结果存储到第二个操作数，本指令可用于有符号整数及无符号整数		

B.4　逻辑指令

这些指令对存储器的位进行操作，而不是对字进行操作。

表B-3　逻辑指令

指　令	操　作　数	影响到的标志
andl	I/R/M, R/M	O/S/Z/P/C
对两个操作数的内容进行逻辑与运算，并将结果存储到第二个操作数，将溢出标志及进位标志设为FALSE		
notl	R/M	
对操作数的每一位逻辑取反，也称为一个数的补数		
orl	I/R/M, R/M	O/S/Z/A/P/C
对两个操作数进行逻辑或，并将结果存储到第二个操作数，将溢出标志及进位标志设为FALSE		
rcll	I/%cl, R/M	O/C
将第一个操作数向左循环移位给定次数，第一个操作数可以是立即数或寄存器%cl。循环移位包含进位标志，因此此指令实际上对33位而非32位进行操作。本指令将设置溢出标志		
rcrl	I/%cl, R/M	O/C
向右循环移位，其他与上一条指令同		
roll	I/%cl, R/M	O/C
向左循环移位，本指令设置溢出标志和进位标志，但不会将进位位作为循环移位的一部分。向左循环移位的次数可通过立即寻址方式或是寄存器%cl的值指定		
rorl	I/%cl, R/M	O/C
向右循环移位，其他与上一条指令同		
sall	I/%cl, R/M	C
算术左移，符号位移出至进位标志，最低有效位填充0，其他位左移。这与一般的左移相同，移动位数通过立即寻址方式或是寄存器%cl指定		
sarl	I/%cl, R/M	C
算术右移，最低有效位移出至进位标志，符号位被向右移入，并保留原符号位。其他位只是向右移。移动位数通过立即寻址方式或是寄存器%cl指定		
shll	I/%cl, R/M	C
逻辑左移，将所有位左移（对符号位不做特殊处理）。将最左一位推入至进位标志，移动位数通过立即寻址方式或是寄存器%cl指定		
shrl	I/%cl, R/M	C
逻辑右移，将寄存器中的所有位右移（对符号位不做特殊处理）。将最右一位推入至进位标志，移动位数通过立即寻址方式或是寄存器%cl指定		
testl	I/R/M, R/M	O/S/Z/A/P/C
对两个操作数进行逻辑与，舍弃计算结果，但根据结果设置标志		
xorl	I/R/M, R/M	O/S/Z/A/P/C
对两个操作数进行异或，将计算结果存储到第二个操作数，将溢出标志及进位标志设为FALSE		

B.5　流控制指令

以下指令可能会改变程序流。

表B-4　流控制指令

指　令	操 作 数	影响到的标志
call	目的地址	O/S/Z/A/C
将%eip所指的下一个值入栈，并跳转至目的地址。这用于函数调用。目的地址也可以是星号后跟寄存器的形式，这种方式为间接函数调用。例如，call *%eax将调用%eax中所含地址所指的函数		
int	I	O/S/Z/A/C
引起给定数字的中断。这通常用于系统调用以及其他内核界面		
Jcc	目的地址	O/S/Z/A/C
条件分支。cc为条件代码。如果条件代码（由前一条指令设置，前一条指令很可能是比较指令）为TRUE，就跳转到给定地址；否则，执行下一条指令。条件代码如下： • [n]a[e]——大于（无符号大于）。如果不大于，就在a前加n，如果大于等于，就在a后加e • [n]b[e]——小于（无符号小于） • [n]e——等于 • [n]z——0 • [n]g[e]——大于（带符号比较） • [n]l[e]——小于（带符号比较） • [n]c——进位标志集 • [n]o——溢出标志集 • [p]p——相等标志集 • [n]s——符号标志集 • ecxz——%ecx 为0		
jmp	目的地址	O/S/Z/A/C
无条件跳转，仅仅是将%eip设置为目的地址，目的地址也可以是星号后跟寄存器的形式，这种方式为间接函数调用。例如，jmp *%eax将跳转至%eax所含地址		
ret		O/S/Z/A/C
从栈中弹出值，并将%eip设置为该值。用于从函数调用返回		

B.6　汇编器指令

以下为汇编器和链接器（而非处理器）的指令，有助于汇编器正确处理你的代码，使之易于使用。

表B-5　汇编器指令

指　令	操　作　数
.ascii	带引号的字符串
将给定带引号字符串转换为字节数据	
.byte	值
将逗号分隔的值列表作为数据插入程序	
.endr	
结束以.rept定义的重复节（section）	
.equ	标签、值
将给定标签设为等于给定值。值可以为数字、字符，或者一个结果等于数字或值的常量表达式。在此命令后，给定值将取代使用标签处	
.globl	标签
将给定标签设为全局，表示可在分开编译的目标文件中使用该标签	
.include	文件
包括给定文件，就如同在此处输入了该文件一样	
.lcomm	符号、大小
用于在.bss节说明当程序执行时应分配的存储区。用要分配的存储区地址定义符号，并确保存储区大小为给定字节数	
.long	值
在命令处将一列以逗号分隔的数字插入程序，每个数字占4字节字长	
.rept	计数
将本命令及.endr之间的内容重复给定次数	
.section	节名
切换正在使用的节。通用节包括.text（用于代码）、.data（用于嵌入程序本身的数据）和.bss（用于未初始化的全局数据）	
.type	符号、函数
告诉链接器给定符号为一个函数	

B.7　其他语法和术语的差异

本书中使用的汇编语言语法称为AT&T语法。每一个Linux发行版标配的GNU工具集都支持这种语法。但是，x86官方汇编语言的语法（称为Intel语法）与此不同。尽管x86汇编语言用于同一平台上，但与AT&T语法有所不同。两者的差异如下。

- Intel语法中，指令操作数的顺序往往相反：先列出目的操作数，再列出源操作数。
- Intel语法中，寄存器不带有前缀百分号（%）。
- Intel语法中，美元符号（$）不代表立即寻址方式。而非立即寻址则在地址前后加方括号（[]）。

- Intel语法中，指令名称不包括被移动数据的大小，而是在指令名后明确指出数据为BYTE、WORD或DWORD。
- 在Intel汇编语言中，内存地址的表示方式有很大的不同（如下文所示）。
- x86处理器系列最初出产的是16位处理器，大多数关于x86处理器的文献中所说的“字”为16位值，并称32位值为“双字”。但是，我们使用的术语“字”指的是处理器上标准寄存器的大小，在x86处理器上是32位。语法也保持了这种命名惯例，Intel语法中代表“双字”的DWORD用来表示标准寄存器大小，而我们则称为“字”。
- Intel汇编语言具备通过段/偏移量对内存寻址的能力。我们不提及这一点是因为Linux不支持分段内存，因此这点与一般的Linux编程无关。

当然还存在其他方面的差异，但相对而言差别较小。为展示部分此类差异，我们给出了以下指令：

```
movl %eax, 8(%ebx,%edi,4)
```

在Intel语法中，这条指令为：

```
mov [8 + %ebx + 1 * edi], eax
```

与AT&T的指令相比，Intel指令的内存引用更易于理解，因为阐明了地址的计算方式。但Intel语法的操作数顺序有时会非常令人费解。

B.8 更多信息

英特尔有一套处理器综合指南，参见http://www.intel.com/design/pentium/manuals/。注意，所有这些指南都使用Intel语法，而不是AT&T语法。其中最重要的信息就是3卷《英特尔IA-32架构软件开发人员手册》（*IA-32 Intel Architecture Software Developer's Manual*），如下所示。

- 第1卷：《系统编程指南》（*System Programming Guide*）
  http://developer.intel.com/design/pentium4/manuals/245470.htm
- 第2卷：《指令集参考》（*Instruction Set Reference*）
  http://developer.intel.com/design/pentium4/manuals/245471.htm
- 第3卷：《系统编程指南》（*System Programming Guide*）
  http://developer.intel.com/design/pentium4/manuals/245472.htm

此外，你可在http://www.gnu.org/software/binutils/manual/gas-2.9.1/as.html找到很多关于GNU汇编器的信息，也可在http://www.gnu.org/software/binutils/manual/ld-2.9.1/ld.html找到关于GNU链接器的手册。

附录C 重要的系统调用

以下是与Linux打交道时一些比较重要的常用系统调用，但在大多数情况下，我们最好是使用库函数，而不是直接用系统调用。这是因为系统调用旨在最简化，而库函数则旨在易于编程。关于Linux下的C函数库信息，详见http://www.gnu.org/software/libc/manual/。

记住，%eax保存系统调用号，返回值和错误代码也存储在%eax中。

表C-1 重要的Linux系统调用

%eax	指令名	%ebx	%ecx	%edx	注
1	exit	返回值（整数）			退出程序
3	read	文件描述符	缓冲区开始地址	缓冲区大小（整数）	读取至给定缓冲区
4	write	文件描述符	缓冲区开始地址	缓冲区大小（整数）	将给定缓冲区的内容写入给定文件描述符
5	open	以空字符结束的文件名	选项列表	许可模式	打开给定文件。返回该文件的描述符或错误号
6	close	文件描述符			关闭给定文件描述符
12	chdir	以空字符结束的目录名			改变程序的当前目录
19	lseek	文件描述符	偏移量	模式	在给定文件中重新定位。若为绝对定位，则模式为0；若为相对定位，则模式为1
20	getpid				返回当前进程的进程ID
39	mkdir	以空字符结束的目录名	许可模式		创建给定目录。假定该目录的所有上级目录都已经存在
40	rmdir	以空字符结束的目录名			删除给定目录
41	dup	文件描述符			返回一个新文件描述符，该描述符为一个现存文件描述符的副本，对该新文件描述符的任何操作与对原文件描述符的等效

（续）

%eax	指令名	%ebx	%ecx	%edx	注
42	pipe	管道数组			创建两个文件描述符，写入其中之一的数据即为读取另一个的数据，反之亦然。%ebx包含的指针指向存储器中的两个字，这两个字所保存的就是文件描述符
45	brk	新系统中断			设置系统中断（如数据节结束）。如果系统中断为0，就仅仅返回当前系统中断
54	ioctl	文件描述符	请求	参数	用于设置设备文件的参数。实际用法随文件描述符引用的文件或设备类型的不同而不同

http://www.lxhp.in-berlin.de/lhpsyscal.html列出了更完备的系统调用以及其他相关信息。你也可通过输入man 2 SYSCALLNAME获得更多信息，该指令将返回UNIX手册第二节关于系统调用的信息，但这是C语言系统调用的用法，不一定对你有直接用处。

关于在Linux上系统调用是如何实现的，请参阅http://www.faqs.org/docs/kernel_2_4/lki-2.html#ss2.11，上面有关于系统调用如何实现的Linux内核2.4内部机制一节。

附录 D ASCII码

为使用以下表格，请找出相应编码的字符（或转义符），并将之与其对应的最左列和最上方的数字相加。

	+0	+1	+2	+3	+4	+5	+6	+7
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL
8	BS	HT	LF	VT	FF	CR	SO	SI
16	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB
24	CAN	EM	SUB	ESC	FS	GS	RS	US
32		!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7
56	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G
72	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W
88	X	Y	Z	[	\	]	^	_
96	‘	a	b	c	d	e	f	g
104	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~	DEL

相对于称为Unicode的国际标准，ASCII实际上正逐渐被淘汰。Unicode标准能显示世界上任何文字的字符。你可能已经注意到，ASCII只支持英文字符。 但Unicode更复杂，因为它需要占用一个以上的字节来为一个字符编码。编码Unicode字符有多种方法，最常见的是UTF-8和UTF-32。UTF-8是向后兼容的，能兼容ASCII（对于英文字符存储内容与ASCII码相同，对于国际字符则扩展到多字节）。UTF-32则使每个字符都占用4字节，而不是1字节。Windows®使用的UTF-16是可变长度编码，但每个字符至少需要2字节，因此不能向后兼容ASCII。

关于国际化问题、字体和Unicode有一个很好的教程，参见Joe Spolsky的一篇佳作，即“The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets(No Excuses!)”[每个软件开发人员都一定要会的Unicode及字符集必备知识（没有借口！）]，参见http://www.joelonsoftware.com/articles/Unicode.html。

附录 E

汇编语言中的常用C语句

本附录是针对学习汇编语言的C程序员编写的，旨在使他们对如何在汇编语言中实现C结构有个大致的了解。

E.1 if语句

在C语言中，if语句由3部分组成：条件、真分支和假分支。但由于汇编语言并非块结构语言，为了实现C的块结构，你必须做一些额外的工作。来看以下C代码：

```
if(a == b)
{
 /* 此处为真分支代码 */
}
else
{
 /* 此处为假分支代码 */
}

/* 此处为真假分支重新汇合点 */
```

在汇编语言中，这可以表示为：

```
 # 将a和b移入寄存器用于比较
 movl a, %eax
 movl b, %ebx

 # 比较
 cmpl %eax, %ebx

 # 如果为真，就跳转到真分支
 je true_branch
```

```
false_branch: # 本标签并非必不可少，只是便于文档记录
 # 此处为假分支代码

 # 跳转至分支重新汇合点
 jmp reconverge

true_branch:
 # 此处为真分支代码

reconverge:
 # 此处为真假分支重新汇合点
```

如你所见，由于汇编语言是线性的，每一块都必须跳过其他块，而真假分支汇合必须由程序员来处理（而非系统自动处理）。

条件语句则编写为一系列`if`语句的形式。

E.2　函数调用

汇编语言函数的调用只需将函数参数以*逆序*入栈，并发出`call`指令。调用后，我们再将参数出栈。来看如下C代码：

```
printf("The number is %d", 88);
```

在汇编语言中，它可以表示为：

```
.section .data
text_string:
.ascii "The number is %d\0"
.section .text
pushl $88
pushl $text_string
call printf
popl %eax
popl %eax  # %eax只是虚拟变量，实际上我们对该变量不做如何操作
           # 你也可以直接调整%esp以指向正确位置
```

E.3　变量和赋值

全局变量和静态变量在`.data`或`.bss`条目中定义。局部变量通过函数开始时在栈上保留一定空间来声明，并在函数结束处归还这些空间。

有趣的是，汇编语言中的全局变量其访问方式与局部变量不同。全局变量通过直接寻址访问，而局部变量使用基址寻址方式。例如，考虑以下C代码：

```
int my_global_var;

int foo()
{
 int my_local_var;

 my_local_var = 1;
 my_global_var = 2;

 return 0;
}
```

在汇编语言中，它可以表示为：

```
 .section .data
 .lcomm my_global_var, 4

 .type foo, @function
foo:
 pushl %ebp              # 保存原基址指针
 movl %esp, $ebp         # 令栈指针指向新基址指针
 subl $4, %esp           # 为变量my_local_var保留空间
 .equ my_local_var, -4   # 现在可使用my_local_var寻找局部变量


 movl $1, my_local_var(%ebp)
 movl $2, my_global_var

 movl %ebp, %esp         # 清除函数变量并返回
 popl %ebp
 ret
```

有一点可能不太明显，即访问全局变量所用的机器周期比访问局部变量要少。但是，因为栈比全局变量更有可能在物理内存（而不是交换内存）中，这可能并不重要。

另外还有一点需要注意：在C编程语言中，编译器加载某个值到寄存器后，该值可能会留在寄存器中，直到需要将寄存器用于其他用途。也可能会存在将值从其他地方移动到寄存器的情况。例如，如果有一个变量foo最初存储在栈上，但为了进行处理，最终编译器会将之移动到寄存器上。如果当前使用的变量不太多，值可能就留在寄存器中，直到再次被用到。否则，当需要将寄存器用于存储其他值时，当前值如有改变会被复制回相应的内存位置。在C语言中，你可以使用

关键字volatile确保所有对变量的修改和引用都是对内存位置本身进行的（而不是它的寄存器副本），以避免其他进程、线程或硬件在你的函数运行时修改该值。

E.4　循环

在汇编语言中，循环与if语句的实现很相似——通过跳转来构建块。在C语言中，一个while循环是由循环体和确定是否退出循环的测试语句构成的。一个for循环与之极为类似，只是具有可选的初始化和测试值递减部分，因此稍加更改就可以变成while循环。

C语言中的while循环如下所示：

```
while(a < b)
{
 /* 进行某些操作 */
}

/* 结束循环 */
```

这在汇编语言中可表示为如下代码：

```
loop_begin:
 movl a, %eax
 movl b, %ebx
 cmpl %eax, %ebx
 jge loop_end

loop_body:
 # 进行某些操作

 jmp loop_begin

loop_end:
 # 结束循环
```

x86汇编语言对循环也直接提供了一些支持。%ecx寄存器可用作计数器，终止条件为0。loop指令会递减%ecx，并在%ecx不为0的情况下跳转到指定地址。例如，若要执行某个语句100次，在C语言中可执行如下语句：

```
for(i=0; i < 100; i++)
{
 /* 在此执行操作 */
}
```

在汇编语言中，这可以写成如下代码：

```
loop_initialize:
 movl $100, %ecx
loop_begin:
 #
 # 在此执行操作
 #

 # 递减%ecx，若%ecx不为0则继续循环
 loop loop_begin

rest_of_program:
 # 在此继续执行
```

要注意的一点是loop指令要求你倒计数至0。如果你需要递增计数或使用其他循环终止数字，那么应该使用不包括loop指令的循环形式。

对于字符串操作的循环，还存在rep指令，但我们将此留作读者练习。

E.5 结构

结构是对内存块的简单描述。例如，在C语言中可使用如下代码：

```
struct person {
 char firstname[40];
 char lastname[40];
 int age;
};
```

这些代码本身不执行任何操作，只是给予你一种使用84字节数据的方式。你可用汇编语言中的.equ指令达成同样效果，代码如下：

```
.equ PERSON_SIZE, 84
.equ PERSON_FIRSTNAME_OFFSET, 0
.equ PERSON_LASTNAME_OFFSET, 40
.equ PERSON_AGE_OFFSET, 80
```

当你要声明此类型的一个变量时，保留84字节数据即可。C语言代码如下：

```
void foo()
{
 struct person p;

 /* 在此执行操作 */
}
```

对应的汇编语言数据如下：

```
foo:
 # 标准头开始
 pushl %ebp
 movl %esp, %ebp

 # 为局部变量保留空间
 subl $PERSON_SIZE, %esp
 # 这是变量相对于%ebp的偏移量
 .equ P_VAR, 0 - PERSON_SIZE

 # 执行操作

 # 标准函数结束
 movl %ebp, %esp
 popl %ebp
 ret
```

为了访问结构成员，你必须使用基址寻址方式，偏移量为上述定义的值。例如，在C语言中通过以下语句设置某人的年龄：

```
p.age = 30;
```

对应的汇编语言为：

```
movl $30, P_VAR + PERSON_AGE_OFFSET(%ebp)
```

E.6　指针

指针很简单，它只是保存某个值的地址。让我们先来看一个全局变量：

```
int global_data = 30;
```

其对应的汇编语言是：

```
 .section .data
global_data:
 .long 30
```

在C语言中，取地址代码如下所示：

```
a = &global_data;
```

对应的汇编语言为：

```
movl $global_data, %eax
```

可以看到，在汇编语言中我们总是通过指针访问内存，这就是直接寻址方式。为了取得指针

本身，你必须采用立即寻址方式。

局部变量略为复杂，但并非难以理解。在C语言中获取局部变量地址的代码如下：

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

 a = 30;

 b = &a;

 *b = 44;
}
```

对应的汇编语言代码为：

```
foo:
 # 函数标准开头
 pushl %ebp
 movl %esp, %ebp

 # 保留两个字的内存
 subl $8, $esp
 .equ A_VAR, -4
 .equ B_VAR, -8

 # a = 30
 movl $30, A_VAR(%ebp)

 # b = &a
 movl $A_VAR, B_VAR(%ebp)
 addl %ebp, B_VAR(%ebp)

 #*b = 30
 movl B_VAR(%ebp), %eax
 movl $30, (%eax)

 # 标准函数结束
 movl %ebp, %esp
 popl %ebp
 ret
```

可以看到，要获取局部变量的地址，我们必须按基址寻址方式计算该地址。但还有一个更简

便的方法，即使用处理器提供的指令leal，该指令表示加载有效地址，会让计算机计算地址，然后在你需要时加载地址。所以，我们可以采用以下代码：

```
# b = &a
leal A_VAR(%ebp), %eax
movl %eax, B_VAR(%ebp)
```

这与采用第一种方式的代码行数相同，但更简洁。接着，要使用这个值，你只需将地址移动到一个通用寄存器，并使用间接寻址，如上例所示。

E.7　获得GCC的帮助

GCC的一个优点是能产生汇编语言代码。要转换C语言文件为汇编语言，你只需输入以下命令：

```
gcc -S file.c
```

输出的汇编语言将在file.s文件中。输出汇编代码的可读性不算高，因为绝大多数变量名被删除，或者被用数字表示的栈位置或自动生成的标签替换。刚开始，你可能想用-O0关闭优化，这样阅读时容易将汇编语言输出与源代码对应起来。

你可能还会注意到GCC会为局部变量保留更多的栈空间，并与%esp执行与运算[①]。这是为了通过双字对齐变量来提高内存和高速缓存的效率。

最后，函数结束时，我们通常在发出ret指令前通过以下指令清理栈：

```
movl %ebp, %esp
popl %ebp
```

但是，GCC输出通常只包括指令leave。这条指令只是上面两条指令的组合。本书中不使用leave，因为我们想弄清楚在处理器级别到底发生了什么。

我鼓励你将自己编写的一个C程序编译为汇编语言，并对比两者的逻辑差异。然后，加入优化并再进行一次编译和对比，这样你能了解编译器如何重新安排你的程序使之更优化。试弄清编译器选择相应安排和指令的原因。

① 注意，不同版本的GCC完成这方面任务的方式不同。

附录 F

使用GDB调试器

阅读本附录时，很可能你至少已经编写过一个带有错误的程序。在汇编语言中，即使是很小的错误也常常会使整个程序由于段错误而崩溃。在大多数编程语言中，你只要随程序的执行打印出各变量值，就能通过这些输出找出出错的地方。但在汇编语言中，调用输出函数比较麻烦。因此，为了帮助确定错误源，你必须使用*源语言调试器*。

调试器是通过单步执行程序帮助你找到错误的程序。在单步执行过程中，你能够检查内存和寄存器的内容。*源语言调试器*则是能让你将调试操作与程序源代码直接绑定的调试器。这意味着该调试器能让你在输入以符号、标签或注释结束的源代码时查看源代码。

我们将研究的调试器是GDB，也就是GNU的调试器。这个程序在几乎所有GNU/Linux发行版本中都有提供，可以调试包括汇编语言在内的多种编程语言编写的程序。

F.1 一个示例调试会话

了解调试器工作原理的最佳方式就是使用它。我们将用于调试的程序是第3章中的maximum。假设你已经输入该程序，除遗漏以下这行代码外没有其他错误：

```
incl %edi
```

当你运行程序时，程序会进入无限循环，永不退出。为确定这一错误的原因，你需要用GDB来运行程序。但是，要做到这点，首先你要让编译器在可执行文件中包含调试信息——只要在as命令中加入--gstabs选项就能做到这点。汇编时应输入以下命令：

```
as --gstabs maximum.s -o maximum.o
```

链接和平常一样，GDB使用STAB调试格式。现在，请输入gdb ./maximum运行调试器。请保证源文件在当前目录下。输出将与以下内容相似：

```
GNU gdb Red Hat Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are welcome to change it and/or
distribute copies of it under certain conditions. Type
"show copying" to see the conditions. There is
absolutely no warranty for GDB. Type "show warranty"
for details.
This GDB was configured as "i386-redhat-linux"...
(gdb)
```

根据GDB版本的不同，输出会略有不同。现在已经加载了程序，但程序还未运行。调试器正在等待你的命令。要运行程序，输入run即可，此时程序将不会返回，因为它陷入了无限循环。为了中断程序，请按下CTRL+C。程序会给出如下信息：

```
Starting program: /home/johnnyb/maximum

Program received signal SIGINT, Interrupt.
start_loop () at maximum.s:34
34              movl data_items(,%edi,4), %eax
Current language: auto; currently asm
(gdb)
```

上述信息告诉你程序被SIGINT信号中断（来自你的CTRL+C），而且中断时位于在start_loop节中，正在执行第34行。它还给出了即将执行的代码。

根据你按下CTRL+C时刻的不同，程序会停在与示例不同的行或指令上。

找到程序错误的最佳方式是跟随程序流查看分支出现错误的地方。为了跟随这个程序流，你需要不断输入stepi（单步执行指令），这会使计算机一次执行一条指令。在你输入stepi数次后，输出如下所示：

```
(gdb) stepi
35              cmpl %ebx, %eax
(gdb) stepi
36              jle start_loop
(gdb) stepi
32              cmpl $0, %eax
(gdb) stepi
33              je loop_exit
(gdb) stepi
34              movl data_items(,%edi,4), %eax
(gdb) stepi
```

```
35                cmpl %ebx, %eax
(gdb) stepi
36                jle start_loop
(gdb) step
32                cmpl $0, %eax
```

可以看出，程序执行了循环。通常，这不存在问题，因为我们编写了循环，但问题在于循环永不中止。因此，要确定问题在哪里，让我们先查看代码中应该退出循环的地方：

```
cmpl $0, %eax
je   loop_exit
```

简单来说，上面的指令查看%eax是否为0，如果为0就应该退出循环。这里要查看几件事，首先，你可能完全遗漏了退出循环的指令，而一般来说程序员犯这种错误的情况并不少见，不过这里并非如此。其次，你应该确保loop_exit在循环之外。如果我们将标签置于错误的位置，程序的行为就会很怪异，但这里亦并非如此。

这些潜在问题都不是原因。因此，下一个可能是%eax中的值不对。在GDB中有两种方法能检查寄存器的内容。首先是使用命令info register，这条命令将以十六进制显示所有寄存器的内容。但现在我们只关心%eax的内容。要只显示%eax的内容，我们可以执行print/$eax来打印十六进制值，也可以执行print/d $eax打印十进制值。注意，在GDB中，寄存器以$而不是%作为前缀。你的屏幕应该会显示如下内容：

```
(gdb) print/d $eax
$1 = 3
(gdb)
```

这意味着第一个查询的结果是3。你每次查询都将以$加上数字的形式表示。现在如果回头看代码，你会发现3是要搜索的数字列表的第一个数字。如果单步执行循环几次，你就会发现在每个循环迭代中的%eax内容都是3。这是不应当发生的，因为每次迭代%eax的内容都应该是列表中的下一个值。

好了，现在我们知道%eax在一遍又一遍地加载同一值。我们来寻找%eax的值是从哪里加载的，找到的代码应该是：

```
movl data_items(,%edi,4), %eax
```

因此，让我们先执行到准备运行这行代码之前。现在，%eax的内容取决于两个值——data_items和%edi。data_items是一个符号，因此是常数值。检查源代码，确保标签在正确的数据之前是值得推崇的做法，在我们的例子中也是如此。因此，我们需要看看%edi，要把它

打印出来，如下所示：

```
(gdb) print/d $edi
$2 = 0
(gdb)
```

这表明%edi被设置为0，这就是一直加载数组第一个元素的原因。发现这点，你应该问自己两个问题——%edi的目的是什么，它的值应该如何改变？要回答第一个问题，我们只需要看看注释。%edi保存data_items的当前索引。由于是有序查找data_items列表中的数字，因此每次循环%edi都应该递增。

快速浏览代码，可以发现根本就没有改变%edi的代码。因此，我们应该增加在每个循环迭代开始时递增%edi的代码。这恰好是我们在一开始去掉的那行代码。现在，汇编、链接并再次运行程序，你会发现程序现在能正常运作了。

希望这个练习能让你对如何使用GDB寻找程序中的错误有些了解。

F.2　断点和GDB的其他特点

上一节，我们输入了一个含有无限循环的程序，这个程序很容易用CTRL+C停止。其他程序可能会中止或带错完成，而此时CTRL+C没有帮助，因为当你按下CTRL+C的时候，程序已经运行完成了。为了解决这个问题，你需要设置*断点*。断点是在源代码中你加以标记，以指示调试器在到达该点时应该停止程序的地方。

设置断点必须在运行程序之前进行。发出run命令之前，你可以使用break命令设置断点。例如，要在第27行设置断点，可以发出命令break 27。然后，程序到达第27行时就会停止运行，并打印出当前行和指令。接着，你就可以从该断点开始单步运行程序，并检查寄存器和内存的内容。要查看程序的行数和行号，你只需使用命令l。这条命令能在打印程序的同时打印出行号。

当处理函数时，你也可以在函数名上设置断点。例如，在第4章中的阶乘程序中，我们可以通过键入break factorial为阶乘函数设置一个断点。这会使调试器在函数调用和函数设置后立即中断（会跳过%esp入栈和%esp复制）。

当单步执行代码时，你常常不希望单步执行每一个函数的每一条指令。仅在极少情况下，我们才需要单步执行经完善测试的函数，否则就是浪费时间。因此，如果你使用nexti命令，而不是stepi命令，GDB会等到函数完成再继续执行。如果使用stepi，GDB会单步执行每一个被调用函数中的每一条指令。

警　告

GDB存在一个中断处理问题。很多时候，GDB会漏掉中断之后的一条指令。该指令实际上执行了，但GDB不会单步执行这条指令。这不算什么问题，你只要注意可能会发生这种情况就行了。

F.3　GDB快速参考

本快速参考表的使用经Robert M. Dondero, Jr许可。括号中列出的参数是可选的。

表F-1　常用的GDB调试命令

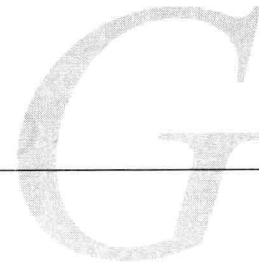
	杂　项
`quit`	退出GDB
`help [cmd]`	打印关于调试器命令cmd的描述。如果没有给出cmd，就打印主题列表
`directory [dir1] [dir2] ...`	将目录dir1、dir2等加入搜索源文件的目录列表
	运行程序
`run [arg1] [arg2] ...`	运行程序，命令行参数为arg1、arg2等
`set args arg1 [arg2] ...`	设置程序命令行参数为arg1、arg2等
`show args`	打印程序的命令行参数
	使用断点
`info breakpoints`	打印所有断点的列表及断点号（断点号用于其他断点命令）
`break linenum`	在行号*linenum*处设置断点
`break *addr`	在内存地址*addr*处设置断点
`break fn`	在函数*fn*开始处设置断点
`condition bpnum expr`	仅当表达式*expr*的值不为0时，才在断点*bpnum*处中断
`command [bpnum] cmd1 [cmd2] ...`	每当到达断点*bpnum*或当前断点时，执行命令*cmd1*、*cmd2*等
`continue`	继续执行程序
`kill`	停止执行程序
`delete [bpnum1] [bpnum2] ...`	删除断点*bpnum1*、*bpnum2*等，如果未指定就删除所有断点
`clear *addr`	清除内存地址*addr*处的断点
`clear [fn]`	清除函数*fn*处的断点或当前断点
`clear linenum`	清除行号*linenum*处的断点
`disable [bpnum1] [bpnum2] ...`	禁用断点*bpnum1*、*bpnum2*等，如果未指定就禁用所有断点

（续）

	使用断点
enable [*bpnum1*] [*bpnum2*] ...	启用断点*bpnum1*、*bpnum2*等，如果未指定就启用所有断点
	单步执行程序
nexti	“单步跳过”下一条指令（不进入函数调用）
stepi	“单步进入”下一条指令（进入函数调用）
finish	“跳出”当前函数
	检查寄存器和内存
info registers	打印所有寄存器的内容
print/*f* $*reg*	利用格式*f*打印寄存器*reg*的内容。格式可以是*x*（十六进制）、*u*（无符号十进制）、*o*（八进制）、*a*（地址）、*c*（字符）或者*f*（浮点）
x/*rsf addr*	利用重复计数*r*、大小*s*和格式*f* 打印内存地址*addr*的内容。重复计数默认值为1，大小可以是*b*（字节）、*h*（半字）、*w*（字）或者*g*（双字）。大小默认值为“字”。格式与上一条指令相同，只是多了*s*（字符串）和*i*（指令）
info display	显示设置好以在每个断点处自动显示的带编号表达式列表
display/*f* $*reg*	在每个断点外，按照格式*f* 打印寄存器*reg*的内容
display/*si* addr	在每个断点处，打印内存地址*addr*的内容，大小为*s*（与x命令的选项相同）
display/*ss* addr	在每个断点处，打印起始内存地址为*addr*，大小为*s*的字符串
undisplay *displaynum*	从显示列表上删除*displaynum*
	检查调用栈的内容
where	打印调用栈
backtrace	打印调用栈
frame	打印调用栈栈顶
up	将上下文移动至调用栈底部
down	将上下文移动至调用栈顶部

附录 G

文档历史



- 2002年12月17日，版本0.5，首次在GNU FDL下发布。
- 2003年7月18日，版本0.6，新增ASCII附录，在介绍内存的一章中完成对CPU的讨论，按照新的格式重新编排练习，并纠正了几个错误。感谢Harald Korneliussen提供建议和在ASCII表上提供的帮助。
- 2004年1月11日，版本0.7，增加介绍汇编语言与C语言转换的附录，增加了“x86指令”相关附录及GDB附录的开始部分，分别完成了“文件”及“计数”相关的两章内容，增加了关于“记录”的一章，创建了常见Linux定义的源文件，纠正了一些错误，并修正了其他很多内容。
- 2004年1月22日，版本0.8，完成GDB附录，基本完成附录B（x86指令）的“更多信息”部分，增加关于“规划程序”的章节，在复习中增加了许多问题，正式完成初稿全文。
- 2004年1月29日，版本0.9，对所有章节都做了大量编辑工作。这次更新使代码更一致，并使解释更加清晰易懂，还增加了一些插图。
- 2004年1月31日，版本1.0，改写了第9章，添加了完整的索引，进行了大量细微修正。

附录 H

GNU自由文档许可协议

H

0. 前言

本许可协议旨在赋予手册、教材或其他书面文档“自由”的含义：保证任何人都可以自由地复制与分发经过或未经改动的该文档，无论是否用于商业目的。此外，本许可协议保护作者和出版者因其作品获得名誉的权利，且他们无需对他人的改动负责。

这个许可协议是一种“反版权”（copyleft，又称公共版权），这表明相应文档的衍生作品必须同样是“自由的”。这个许可协议是为自由软件设计的GNU GPL（GNU通用公共许可协议）的补充。

我们设计本许可协议供自由软件的手册使用，因为自由软件需要自由文档：一个自由的程序附带的手册当然应与该软件同样“自由”。不过，本许可协议并不局限于程序手册，所有的文档作品都可以使用本许可协议，无论其题材如何，也无论其是否作为书本刊印。我们建议对指导性或参考性的文档作品使用本许可协议。

1. 适用性与定义

对于任何手册或其他作品，只要其中包含文档版权所有者的声明，表明文档在本许可协议的条款下发布，即适用于本许可协议。以下的“文档”都是指此类根据GFDL（GNU Free Documentation License，GNU自由文档许可协议）发布的手册或作品。任何公众一员都是本许可协议的许可对象，这里将用“你”来称呼。

文档的“修订版”（Modified Version）是指任何包含全部或部分该文档的作品，不论是完全照搬，还是经过加工修改和/或翻译成其他语言。

“附属章节”（Secondary Section）是文档指定的附录或该文档的前页部分，专门描述文档出版者或作者与文档主题或相关问题之间的关联，并且不包含文档主题直接提及的内容。(例如，如果文档是某本数学教材的一部分，附属章节或许不会谈及数学。) 其可能会是与主题或相关事

物有关的历史关联，或者是相关的法律、商业、哲理、道德或政治立场。

“固定章节”（Invariant Sections）是某些指定了标题的附属章节，其标题在表明文档以本许可协议发布的声明中声明为固定章节。

“封皮文本”（Cover Text）是在表明文档以本许可协议发布的声明中列为封面文本或封底文本的某些短段落。

文档的“透明”（Transparent）副本是指计算机可读副本，其格式符合公众可以得到的规范，其内容可直接通过通用文本编辑器、（对于像素构成的图像）通用绘图程序，或者（对于绘制的图形）被广泛使用的绘画程序直接查看或修改，也适用于输入到文本格式处理程序，或适合自动翻译成各种适用于输入到文本格式处理程序的格式。如果某个副本，其他各方面都符合透明文件定义，但副本的格式标记旨在反对或防止读者的后续修改，那么这种副本就不是“透明”的。“不透明”（Opaque）副本含义与此相反。

适用于透明副本的格式有：没有标记的纯ASCII文本、Texinfo输入格式、LaTeX输入格式、使用公开可DTD的SGML或XML，便于手动修改以及符合标准的简单HTML。不透明格式有PostScript、PDF、仅供私有版权字处理程序阅读和编辑的私有版权格式、所用的DTD和/或处理工具并非广泛可得的SGML或XML，以及一些字处理器生成并仅用于输出的机器生成HTML。

对于刊印本，“扉页”（Title Page）是扉页本身，以及随后清楚刊印本许可协议要求在扉页上出现的信息的几个页面。对于不带扉页的作品格式，扉页是指正文之前接近作品最突出标题的文本。

2. 原样复制

你可以以任何介质复制并分发文档，无论是否用于商业目的，但要保证本许可协议、版权声明和宣称本许可协议应用于文档的声明都完整复制到所有副本，且你未在本副本中增加任何附加条件。你不能对你制作或发布的副本进行技术处理以阻止他人阅读或再次复制。不过，你可以用副本换取报酬。如果你大量发布副本，就必须遵循下面第3节中的条款。

你也可以在第3节中的条款下出借或/和公开展示副本。

3. 大量复制

如果你发布的打印版文档副本数量超过100份，而文档许可声明需要封皮文本，你就必须把副本用清晰明了地印有文档许可声明所要求的封皮文字的封皮封装好，这些封皮文字包括封面文字和封底文字。封面和封底都必须写明你是该副本的发布者。封面必须印有完整的书名，书名的每个字都必须同样显著。你可以在封面添一些信息作为补充。在满足了遵循对封皮改动的限制、

维持文档标题等条件下，副本在其他方面上可以被视同原样复制的副本。

如果封面或封底必要的文字太多，从而导致看起来不清晰明了，那么你可以把优先的条目（尽量适合地）列在封面或封底，并将剩下的那些放在邻近的页面里。

如果你出版或分发不透明（即不兼容）文档副本数量超过100份，那么就应随同每份不兼容副本带一份可机读的透明副本，或在每份不透明副本里或随同该副本指明一个到完整透明副本（不含附加材料）的计算机网络地址，且其允许大众网民通过公共标准网络协议免费、匿名访问并下载。如果你采用后一种办法，就必须适当采取一些慎重的措施：开始大量发行不透明副本时，要保证透明副本在你（直接，或者通过代理商或零售商）发布最后一份对应版本的不透明副本后，至少一年内在指定的地址可以访问到。

请在对文档副本进行大量再分发前联系文档作者，尽管这并非必不可少的一步，但这样他们有机会向你提供文档的最新版本。

4. 修改

在满足上文2、3两节的条件下，你可以复制和发布文档的修订版，但必须严格遵循本许可协议发布，这样修订版就起到了原文档的作用，从而允许任何持有该修订版副本者对修订版进行发布和修改。而且，你必须在修订版中做到如下几点。

- A. 在扉页（若有封皮，则封皮和扉页都要）使用的书名要与原文档以及之前各版本（如果有的话，应该列在“文档历史”中）的名字有明显区别。如果原始发布人允许的话，你也可以使用相同的书名。
- B. 负责对修订版文档进行修改的个人或组织必须作为作者在扉页上列出，并应至少列出文档的5个主要作者（若少于5个，则全部列出）。
- C. 修订版文档发布者的名字必须作为文档发布者在扉页上列明。
- D. 保留原文档的所有版权声明。
- E. 在其他版权声明附近为你所做的修订添加适当的版权声明。
- F. 在版权声明之后，加入允许公众在本许可协议下使用修订版文档的许可声明，其形式如下面附录所示。
- G. 在许可声明里保留源文档列出的所有固定章节，以及源文档所要求的封皮文字。
- H. 不加修改地加入本许可协议的副本。
- I. 保留题为“历史”的章节及其标题，并在其中加入一项，该项至少列出扉页上的修订版的标题、年份、新作者和新发布者。如果文档中没有名为“历史”的章节，就新建这一章节，创建一条包括原文档的书名、年份、作者、发布者的条目，如原文档的扉页里给出的那样，然后再加入上句所述的描述修订版的条目。

- J. 如果原文档中给出了访问透明文档的链接，应保留此网址，以及提供原文档之前各版本中给出的对应透明文档的网址。这些链接可以在“历史”章节给出。在文档发布之前已经发布至少4年的版本给出的网址，或者该版本的发行者授权不列出网址，其链接可在作品中略去。
- K. 对于任何以“致谢”（Acknowledgements）或“献辞”（Dedications）命名的章节，保留该章节标题、保留对每个贡献者的致谢和/或献辞的内容和语气不变。
- L. 维持原文档的固定章节的标题和内容不变。章节标号等不属于章节标题的一部分（可以改动）。
- M. 删除原文档中任何题为“注记”（Endorsements）的章节。这些章节可能是只针对原文档的。
- N. 不要把任何已有章节标题改为“注记”，或改为与固定章节有冲突的标题。

如果修订版加入了符合附属章节定义的新前页或附录章节，并且不含有从原文档中复制的内容，你可以选择标记其为其全部或部分为固定章节。如果这样做的话，你就要把它们的标题加入修订版许可声明的固定章节列表中。这些标题必须与其他章节的标题不同。

你可以添加题为“注记”的章节，但它只能包含各方面对你的修订版的评注，例如同行评价的声明，或者被某组织认可为某标准权威定义的声明文本。

你可以在修订版封面文字末尾添加至多5个词的一个段落，在封底文字末尾添加至多25个词的一个段落。一个实体只能添加（或通过整理形成）一段封面文字和一段封底文字。如果你或你代理的实体已经加入或通过整理制成原文档的某一对封皮文字，你就不应再添加了；但在添加旧封皮的原发行者的明确许可下，你可以替换原来的。

原文档的作者和发布者并未通过本许可协议授权利用其名字对任一修订版进行宣传，也未表明或暗示认可任何一个修订版。

5. 合并文档

你可以在本许可协议下依据第4节里对修订版定义的条款，把原文档与其他在本协议下发布的文档合并起来发布，只要你在合并文档中原样包含所有原文档的所有固定章节，并在合并文档的许可声明中将其全部列为固定章节。

合并文档只需包含本协议的一份副本，并且重复的固定章节可用一个副本代替。如果有名称重复但内容不同的固定章节，则在此固定章节的标题名后加括号，如果已知该章节的原作者或出版者的名字，那么应当使固定章节的名字唯一：如果相应名字已知，就在后面用括号括起原作者或出版者的名字，否则括号中为一个独一无二的编号。在此合并文档许可声明中固定章节的列表

中，你应对其章节标题做相同的调整。

在合并文档里，你必须把不同原文档的“历史”章节合并成一个“历史”章节；同样需要合并的章节还有“致谢”、“献辞”。你还必须把所有“注记”章节删除。

6. 文档合集

你可以制作一个原文档和其他文档的合集，前提是其中每份文档都是在本许可协议下发布的。你可在合集中以本许可协议的一份副本替代合集中各文档里分别使用的本许可协议副本，只要在文档的其他方面遵循本许可协议的原样复制条款即可。

你可以从该合集中单独提取一个文档，并在本许可协议下单独发布，只要在该提取出的文档中加入一份本许可协议的副本，并在文档的其他所有方面都遵循本许可协议的原样复制条款即可。

7. 独立作品汇集

将原文档或文档的派生品与其他独立文档或作品汇编为一个文库或发布介质，如果未对该汇编作品声明汇编版权，那么该汇编作品整体不被视为原文档的修订版，而是称为“汇集”（aggregate），当文档被包含在汇集中时，本协议不适用于汇集中该文档的非衍生作品。如果第3节中对封皮文字的要求适用于原文档的这些副本，那么如果文档在汇集中所占的比例小于1/4，那么文档的封皮文字可以置于汇集内该文档的内封页上，否则就必须出现于封装整个汇集的封皮上。

8. 翻译

翻译被认为是一种修改，所以你可以按照第4节的条款发布文档的翻译版。用译文取代文档的固定章节需要获得版权所有者的授权，但你可以将部分或全部固定章节的译文附于原始版本之后。翻译版可以包含一份本许可协议、所有许可证声明和免责声明的译文，只要你同时保留其原始英文版即可。当译文与原文有分歧时，以原文为准。

9. 许可的终止

除非本许可协议明确规定，否则不得对文档进行复制、修改、分授许可或发布。任何其他试图复制、修改、分授许可、发布本文档的行为都是无效的，并将自动终止本许可协议所授予你的权利。然而，其他从你这里依照本许可协议得到副本或授权的人（或组织）得到的许可证都不会终止，只要他们仍然完全遵照本许可协议即可。

10. 本许可协议的未来修订版本

自由软件基金会有时会发布GNU自由文档许可协议的新修订版。新版本将会和当前版本体现

类似的精神，但在处理某些新的问题和观点的细节上会有所不同。详情参见http://www.gnu.org/copyleft/。

本许可协议的每个版本都有唯一的版本号。如果文档指定遵循某版本的本许可协议“或任何后续版本”，你可以选择遵循指定版本或自由软件基金会发布的任何后续（非草案）版本的条款和条件。如果文档没有指定本许可协议的版本，那么你可以选择遵循自由软件基金会已发布的任何（非草案）版本。

附录

要使用本许可协议发布你写的文档，请在文档中放置一份本许可协议，并在紧接扉页之后加入如下版权与许可声明：

Copyright © YEAR YOUR NAME.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled “GNU Free Documentation License”.

如果没有固定章节，将“with the Invariant Sections being LIST THEIR TITLES”替换为“with no Invariant Section”。如果无封面文字，将“Front-Cover Texts being LIST”替换为“no Front-Cover Texts”；对封底文字也做同样处理。

如果你的文档包含重要的程序代码示例，我们建议同样选择自由软件许可协议（如GNU通用公共许可协议）发布这些示例，以授权将其作为自由软件使用。

附录 I

致　　谢

我要感谢的人不计其数，但限于篇幅我只能在此提及其中有限的几位。没有他们，就没有今天的我。感谢我的家人、主日学校老师、青年牧师、学校的老师、我的朋友和上帝带入我生命的所有人，感谢这些指引我、帮助我、教导我的人们。谨以此书献给你们所有人！

不过，我还要特别向其中一些人致以诚挚的谢意。

首先，感谢伊利诺伊州香槟市葡萄园基督徒团契教会的成员，感谢众位在我和我的家庭出现危机时为我们所做的一切。我已经很久没有听到你们的消息了，但一直很想念你们。认识你们是上帝对我、我的妻子和丹尼尔的厚爱，当我们最需要的时候，你们带来了基督的爱，我们对此不胜感激。每次想到你们，我都深深感谢上帝在我们最需要的时刻将你们带到我们的面前。即使在伊利诺伊州尚未建立朋友圈的那段时间，上帝亦告诉我们其仍然看护着我们，而你们就是上帝在凡间的双手，谢谢你们。我还要特别感谢Joe、Rhonda、Pam、Dell、Herschel和Vicki。当然，还有许许多多的人帮助过我们，在此请原谅我无法一一列出。

我也想感谢我的父母，他们为我树立了在困境中坚韧不拔、自强不息的榜样，是你们的言传身教让我成为孩子们的好父亲和妻子的好丈夫。

我也要感谢我的妻子，她在我们刚开始约会时就鼓励我在一切事物中寻求神。感谢她支持我写作本书，更要感谢她支持我顺服神。

我也想感谢小灯塔学校。我们全家一直以来都从你们给我儿子的帮助中获益。

我还要感谢Joe和D.A.给了我一次加入事奉的机会，能够再次成为神的事工给了我许多帮助。在过去的几年中，你们都给予我力量，让我能撰写本书。没有你们的支持，个人危机将使我不堪重负，除了得过且过将无法思考其他，更不用说完成本书了。你们每一位都是上帝对我的恩赐，我会永远为你们祈祷。

索　引

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z