HZ BOOKS
华章教育

PEARSON

经 典 原 版 书 库

深入理解计算机系统

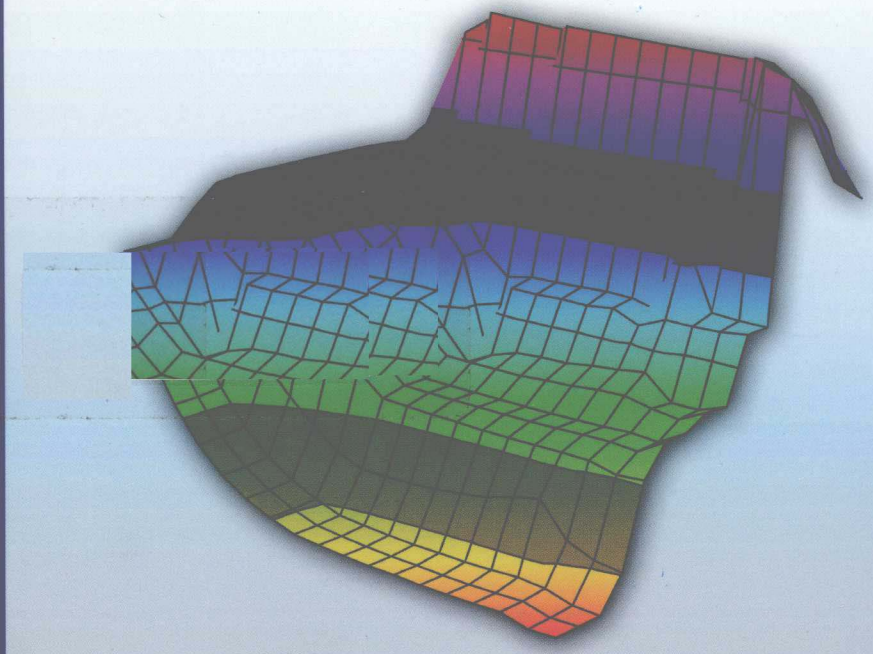
（美）Randal E. Bryant　David R. O'Hallaron 著
卡内基-梅隆大学

（英文版·第2版）

Second Edition

COMPUTER SYSTEMS

A Programmer's Perspective



Bryant • O'Hallaron

机械工业出版社
China Machine Press

经 典 原 版 书 库

深入理解计算机系统

（英文版·第2版）

Computer Systems

A Programmer's Perspective (Second Edition)

Randal E. Bryant
（美）David R. O'Hallaron 著
卡内基－梅隆大学

机械工业出版社
China Machine Press

本书版权登记号：图字：01-2010-6351

图书在版编目（CIP）数据

深入理解计算机系统（英文版 · 第 2 版）/（美）布莱恩特（Bryant, R. E.），（美）奥哈拉伦（O'Hallaron, D. R.）著 . —北京：机械工业出版社，2011.1

（经典原版书库）

书名原文：Computer Systems: A Programmer's Perspective, Second Edition

ISBN 978-7-111-32631-1

I. 深…　II. ①布…　②奥…　III. 计算机系统　IV. TP30

中国版本图书馆 CIP 数据核字（2010）第 230916 号

机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码　100037）

责任编辑：王春华

北京京师印务有限公司印刷

2011 年 1 月第 1 版第 1 次印刷

186mm × 240mm • 67.5 印张

标准书号：ISBN 978-7-111-32631-1

定价：128.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991；88361066

购书热线：（010）68326294；88379649；68995259

投稿热线：（010）88379604

读者信箱：hzjsj@hzbook.com

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自 1998 年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与 Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage 等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出 Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson 等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com

电子邮件：hzjsj@hzbook.com

联系电话：（010）88379604

联系地址：北京市西城区百万庄南街 1 号

邮政编码：100037

华章教育

华章科技图书出版中心

前言节选 | Computer Systems A Programmer's Perspective, 2E

本书的主要读者是计算机科学家、计算机工程师，以及那些想通过学习计算机系统的内在运作而能够写出更好程序的人。

我们的目的是解释所有计算机系统的本质概念，并向你展示这些概念是如何实实在在地影响应用程序的正确性、性能和实用性的。其他的系统类书籍都是从*构建者的角度*来写的，讲述如何实现硬件或是系统软件，包括操作系统、编译器和网络接口。而本书是从*程序员的角度*来写的，讲述应用程序员如何能够利用系统知识来编写出更好的程序。当然，学习一个计算机系统应该做些什么，是学习如何构建一个计算机系统的很好的出发点，所以，对于希望继续学习系统软硬件实现的人来说，本书也是一本很有价值的介绍性读物。

本书概述

本书由 12 章组成，旨在阐述计算机系统的核心概念。

- *第 1 章：计算机系统漫游*。这一章通过研究“hello, world”这个简单程序的生命周期，介绍计算机系统的主要概念和主题。
- *第 2 章：信息的表示和处理*。我们讲述了计算机的算术运算，重点描述了会对程序员有影响的无符号数和数的二进制补码（two's complement）表示的特性。我们考虑数字是如何表示的，以及由此确定对于一个给定的字长，其可能编码值的范围。我们讨论该如何表示数字，以及因此用给定的字长能编码的数值的范围。我们探讨有符号和无符号数字之间类型转换的效果，还阐述算术运算的数学特性。菜鸟级程序员经常很惊奇地了解到（用二进制补码表示的）两个正数的和或者积可能为负。另一方面，二进制补码的算术运算满足代数环的特性，因此，编译器可以很安全地把一个常量乘法转化为一系列的移位和加法。我们用 C 语言的位级操作来说明布尔代数的原理和应用。我们从两个方面讲述了 IEEE 标准的浮点格式：一是如何用它来表示数值，一是浮点运算的数学属性。
- *第 3 章：程序的机器级表示*。我们教读者如何阅读由 C 编译器生成的 IA32 和 x86-64 汇编语言。我们说明为不同控制结构，比如条件、循环和开关语句，生成的基本指令模式。我们还讲述过程的执行，包括栈分配、寄存器使用惯例和参数传递。我们讨论不同数据结构（如结构、联合（union）和数组）的分配和访问方式。我们还以分析程序在机器级的样子作为途径，来理解常见的代码安全漏洞，例如，缓冲区溢出，以及理解程序员、编译器和操作系统可以采取的减轻这些威胁的措施。
- *第 4 章：处理器体系结构*。这一章讲述基本的组合和时序逻辑元素，并展示这些元素如何在数据通路（datapath）中组合到一起来执行 IA32 指令集的一个称为“Y86”的简化子集。本

章中处理器设计的控制逻辑是用一种称为 HCL 的简单硬件描述语言来描述的。用 HCL 写的硬件设计能够编译和链接到本书提供的模拟器中，还可以根据这些设计生成 Verilog 描述，它适合合成（synthesis）到实际可以运行的硬件上去。

- 第 5 章：优化程序性能。在这一章里，我们介绍了许多提高代码性能的技术，主要思想就是让程序员通过使编译器能够生成更有效的机器代码来学习编写 C 代码。
- 第 6 章：存储器层次结构。对应用程序员来说，存储器系统是计算机系统中最直接可见的部分之一。我们讲述不同类型的随机存取存储器（RAM）和只读存储器（ROM），以及磁盘和固态硬盘的几何形状和组织构造。我们描述这些存储设备是如何放置在层次结构中的，讲述访问局部性是如何使这种层次结构成为可能的。我们通过一个独特的观点使这些理论具体化、形象化，那就是将存储器系统视为一个“存储器山”，山脊是时间局部性，而斜坡是空间局部性。最后，我们向读者阐述如何通过改善程序的时间局部性和空间局部性来提高应用程序的性能。
- 第 7 章：链接。本章讲述静态和动态链接，包括的概念有可重定位的（relocatable）和可执行的目标文件、符号解析、重定位（relocation）、静态库、共享目标库，以及与位置无关的代码。
- 第 8 章：异常控制流。在本书的这个部分，我们通过介绍异常控制流（比如，除了正常分支和过程调用以外的控制流的变化）的一般概念，打破单一程序的模型。我们给出存在于系统所有层次的异常控制流的例子，从底层的硬件异常和中断，到并发进程的上下文切换，到由于 Unix 信号传送引起的控制流突变，到 C 语言中破坏栈原则的非本地跳转（nonlocal jump）。
- 第 9 章：虚拟存储器。我们讲述虚拟存储器系统是希望读者对它是如何工作的以及它的特性有所了解。我们想让读者了解为什么不同的并发进程各自都有一个完全相同的地址范围，能共享某些页，而又独占另外一些页。我们还覆盖讲了一些管理和操纵虚拟存储器的问题。特别地，我们讨论了存储分配操作，就像 Unix 的 `malloc` 和 `free` 操作。
- 第 10 章：系统级 I/O。我们讲述 Unix I/O 的基本概念，例如文件和描述符。我们描述如何共享文件，I/O 重定向是如何工作的，还有如何访问文件的元数据。我们还开发了一个健壮的带缓冲区的 I/O 包，可以正确处理一种称为 short counts 的奇特行为，也就是库函数只读取一部分的输入数据。我们阐述 C 的标准 I/O 库，以及它与 Unix I/O 的关系，重点谈到标准 I/O 的局限性，这些局限性使之不适合网络编程。
- 第 11 章：网络编程。对编程而言，网络是非常有趣的 I/O 设备，将许多我们前面文中学习的概念，比如进程、信号、字节顺序（byte order）、存储器映射和动态存储器分配，联系在一起。网络程序还为下一章的主题——并发，提供了一个很令人信服的上下文。本章只是网络编程的一个很小的部分，使读者能够编写一个 Web 服务器。我们还讲述了位于所有网络程序底层的客户端 - 服务器模型。我们展现了一个程序员对 Internet 的观点，并且教读者如何用套接字（socket）接口来编写 Internet 客户端和服务器。最后，我们介绍超文本传输协议 HTTP，并开发了一个简单的迭代式（iterative）Web 服务器。
- 第 12 章：并发编程。这一章以 Internet 服务器设计为例介绍了并发编程。我们比较对照了三种编写并发程序的基本机制（进程、I/O 多路复用技术和线程），并且展示如何用它们来建造并发 Internet 服务器。我们探讨了用 *P*、*V* 信号操作来实现同步、线程安全和可重入

（reentrancy）、竞争条件以及死锁等的基本原则。我们还讲述了线程级编程的使用方法，来解释应用程序中的并行性，使得程序在多核的处理器上能执行得更快。

本版新增内容

本书的第 1 版于 2003 年出版。考虑到计算机技术发展如此迅速，这本书的内容还算是保持得很好。事实证明 Intel x86 的机器上运行类 Unix 操作系统，加上采用 C 语言编程，是一种能够涵盖当今许多系统的组合。硬件技术和编译器的变化，以及很多教师教授这些内容的经验，都促使我们做了大量的修改。

下面列出的是一些更加详细的改进：

- 第 2 章：信息的表示和处理。通过更加详细地解释概念以及更多的练习题和家庭作业，我们试图使这部分内容更加易懂。我们将一些比较偏理论的内容放到了网络旁注里。还讲述了一些由于计算机算术运算的溢出造成的安全漏洞。
- 第 3 章：程序的机器级表示。我们将内容的覆盖范围扩展到了包括 x86-64，也就是将 x86 处理器扩展到了 64 位字长。也使用了更新版本的 GCC 产生的代码。另外还增强了对缓冲区溢出漏洞的描述。在网络旁注里，我们给出了两类不同的浮点指令，还介绍了当编译器试图做更高等级优化的时候，做的一些奇特的变换。另外，还有一个网络旁注描述了如何在一个 C 语言程序中嵌入 x86 汇编代码。
- 第 4 章：处理器体系结构。更加详细地说明了我们的处理器设计中的异常发现和处理。在网络旁注里，我们也给出了处理器设计的 Verilog 描述映射，使得我们的设计能够合成到可运行的硬件上。
- 第 5 章：优化程序性能。我们极大地改变了对乱序处理器如何运行的描述，还提出了一种简单的技术，能够基于程序的数据流图表示中的路径来分析程序的性能。在网络旁注里，描述了 C 语言程序员如何能够利用较新的 x86 处理器中提供的 SIMD（单指令流，多数据流）指令来编程。
- 第 6 章：存储器层次结构。我们增加了固态硬盘的内容，还更新了我们的表述，使之基于 Intel Core i7 处理器的存储器层次结构。
- 第 7 章：链接。本章的变化不大。
- 第 8 章：异常控制流。我们改进了对于进程模型如何引入一些基本的并发概念的讨论，例如非确定性。
- 第 9 章：虚拟存储器。我们更新了存储器系统案例研究，采用了 64 位 Intel Core i7 处理器为例来讲述。我们还更新了 `malloc` 函数的示例实现，使之既能在 32 位也能在 64 位环境中执行。
- 第 10 章：系统级 I/O。本章的变化不大。
- 第 11 章：网络编程。本章的变化不大。
- 第 12 章：并发编程。我们增加了关于并发性一般原则的内容，还讲述了程序员如何利用线程级并行性使得程序在多核机器上能运行得更快。

此外，我们还增加和修改了很多练习题和家庭作业。

This book (CS:APP) is for computer scientists, computer engineers, and others who want to be able to write better programs by learning what is going on "under the hood" of a computer system.

Our aim is to explain the enduring concepts underlying all computer systems, and to show you the concrete ways that these ideas affect the correctness, performance, and utility of your application programs. Other systems books are written from a *builder's perspective*, describing how to implement the hardware or the systems software, including the operating system, compiler, and network interface. This book is written from a *programmer's perspective*, describing how application programmers can use their knowledge of a system to write better programs. Of course, learning what a system is supposed to do provides a good first step in learning how to build one, and so this book also serves as a valuable introduction to those who go on to implement systems hardware and software.

If you study and learn the concepts in this book, you will be on your way to becoming the rare "power programmer" who knows how things work and how to fix them when they break. Our aim is to present the fundamental concepts in ways that you will find useful right away. You will also be prepared to delve deeper, studying such topics as compilers, computer architecture, operating systems, embedded systems, and networking.

Assumptions about the Reader's Background

The presentation of machine code in the book is based on two related formats supported by Intel and its competitors, colloquially known as "x86." IA32 is the machine code that has become the de facto standard for a wide range of systems. x86-64 is an extension of IA32 to enable programs to operate on larger data and to reference a wider range of memory addresses. Since x86-64 systems are able to run IA32 code, both of these forms of machine code will see widespread use for the foreseeable future. We consider how these machines execute C programs on Unix or Unix-like (such as Linux) operating systems. (To simplify our presentation, we will use the term "Unix" as an umbrella term for systems having Unix as their heritage, including Solaris, Mac OS, and Linux.) The text contains numerous programming examples that have been compiled and run on Linux systems. We assume that you have access to such a machine and are able to log in and do simple things such as changing directories.

If your computer runs Microsoft Windows, you have two choices. First, you can get a copy of Linux (`www.ubuntu.com`) and install it as a "dual boot" option, so that your machine can run either operating system. Alternatively, by installing a copy of the Cygwin tools (`www.cygwin.com`), you can run a Unix-like shell under

Windows and have an environment very close to that provided by Linux. Not all features of Linux are available under Cygwin, however.

We also assume that you have some familiarity with C or C++. If your only prior experience is with Java, the transition will require more effort on your part, but we will help you. Java and C share similar syntax and control statements. However, there are aspects of C, particularly pointers, explicit dynamic memory allocation, and formatted I/O, that do not exist in Java. Fortunately, C is a small language, and it is clearly and beautifully described in the classic "K&R" text by Brian Kernighan and Dennis Ritchie [58]. Regardless of your programming background, consider K&R an essential part of your personal systems library.

Several of the early chapters in the book explore the interactions between C programs and their machine-language counterparts. The machine-language examples were all generated by the GNU GCC compiler running on IA32 and x86-64 processors. We do not assume any prior experience with hardware, machine language, or assembly-language programming.

New to C? Advice on the C programming language

To help readers whose background in C programming is weak (or nonexistent), we have also included these special notes to highlight features that are especially important in C. We assume you are familiar with C++ or Java.

How to Read the Book

Learning how computer systems work from a programmer's perspective is great fun, mainly because you can do it actively. Whenever you learn something new, you can try it out right away and see the result first hand. In fact, we believe that the only way to learn systems is to *do* systems, either working concrete problems or writing and running programs on real systems.

This theme pervades the entire book. When a new concept is introduced, it is followed in the text by one or more *practice problems* that you should work immediately to test your understanding. Solutions to the practice problems are at the end of each chapter. As you read, try to solve each problem on your own, and then check the solution to make sure you are on the right track. Each chapter is followed by a set of *homework problems* of varying difficulty. Your instructor has the solutions to the homework problems in an Instructor's Manual. For each homework problem, we show a rating of the amount of effort we feel it will require:

◆ Should require just a few minutes. Little or no programming required.

◆◆ Might require up to 20 minutes. Often involves writing and testing some code. Many of these are derived from problems we have given on exams.

◆◆◆ Requires a significant effort, perhaps 1–2 hours. Generally involves writing and testing a significant amount of code.

◆◆◆◆ A lab assignment, requiring up to 10 hours of effort.

code/intro/hello.c

```
1  #include <stdio.h>
2
3  int main()
4  {
5      printf("hello, world\n");
6      return 0;
7  }
```

code/intro/hello.c

Figure 1 **A typical code example.**

Each code example in the text was formatted directly, without any manual intervention, from a C program compiled with GCC and tested on a Linux system. Of course, your system may have a different version of GCC, or a different compiler altogether, and so your compiler might generate different machine code, but the overall behavior should be the same. All of the source code is available from the CS:APP Web page at `csapp.cs.cmu.edu`. In the text, the file names of the source programs are documented in horizontal bars that surround the formatted code. For example, the program in Figure 1 can be found in the file `hello.c` in directory `code/intro/`. We encourage you to try running the example programs on your system as you encounter them.

To avoid having a book that is overwhelming, both in bulk and in content, we have created a number of *Web asides* containing material that supplements the main presentation of the book. These asides are referenced within the book with a notation of the form *CHAP:TOP*, where *CHAP* is a short encoding of the chapter subject, and *TOP* is short code for the topic that is covered. For example, Web Aside DATA:BOOL contains supplementary material on Boolean algebra for the presentation on data representations in Chapter 2, while Web Aside ARCH:VLOG contains material describing processor designs using the Verilog hardware description language, supplementing the presentation of processor design in Chapter 4. All of these Web asides are available from the CS:APP Web page.

Aside What is an aside?

You will encounter asides of this form throughout the text. Asides are parenthetical remarks that give you some additional insight into the current topic. Asides serve a number of purposes. Some are little history lessons. For example, where did C, Linux, and the Internet come from? Other asides are meant to clarify ideas that students often find confusing. For example, what is the difference between a cache line, set, and block? Other asides give real-world examples. For example, how a floating-point error crashed a French rocket, or what the geometry of an actual Seagate disk drive looks like. Finally, some asides are just fun stuff. For example, what is a "hoinky"?

Book Overview

The CS:APP book consists of 12 chapters designed to capture the core ideas in computer systems:

- *Chapter 1: A Tour of Computer Systems.* This chapter introduces the major ideas and themes in computer systems by tracing the life cycle of a simple "hello, world" program.
- *Chapter 2: Representing and Manipulating Information.* We cover computer arithmetic, emphasizing the properties of unsigned and two's-complement number representations that affect programmers. We consider how numbers are represented and therefore what range of values can be encoded for a given word size. We consider the effect of casting between signed and unsigned numbers. We cover the mathematical properties of arithmetic operations. Novice programmers are often surprised to learn that the (two's-complement) sum or product of two positive numbers can be negative. On the other hand, two's-complement arithmetic satisfies the algebraic properties of a ring, and hence a compiler can safely transform multiplication by a constant into a sequence of shifts and adds. We use the bit-level operations of C to demonstrate the principles and applications of Boolean algebra. We cover the IEEE floating-point format in terms of how it represents values and the mathematical properties of floating-point operations.

  Having a solid understanding of computer arithmetic is critical to writing reliable programs. For example, programmers and compilers cannot replace the expression `(x<y)` with `(x-y < 0)`, due to the possibility of overflow. They cannot even replace it with the expression `(-y < -x)`, due to the asymmetric range of negative and positive numbers in the two's-complement representation. Arithmetic overflow is a common source of programming errors and security vulnerabilities, yet few other books cover the properties of computer arithmetic from a programmer's perspective.
- *Chapter 3: Machine-Level Representation of Programs.* We teach you how to read the IA32 and x86-64 assembly language generated by a C compiler. We cover the basic instruction patterns generated for different control constructs, such as conditionals, loops, and switch statements. We cover the implementation of procedures, including stack allocation, register usage conventions, and parameter passing. We cover the way different data structures such as structures, unions, and arrays are allocated and accessed. We also use the machine-level view of programs as a way to understand common code security vulnerabilities, such as buffer overflow, and steps that the programmer, the compiler, and the operating system can take to mitigate these threats. Learning the concepts in this chapter helps you become a better programmer, because you will understand how programs are represented on a machine. One certain benefit is that you will develop a thorough and concrete understanding of pointers.
- *Chapter 4: Processor Architecture.* This chapter covers basic combinational and sequential logic elements, and then shows how these elements can be

combined in a datapath that executes a simplified subset of the IA32 instruction set called "Y86." We begin with the design of a single-cycle datapath. This design is conceptually very simple, but it would not be very fast. We then introduce *pipelining*, where the different steps required to process an instruction are implemented as separate stages. At any given time, each stage can work on a different instruction. Our five-stage processor pipeline is much more realistic. The control logic for the processor designs is described using a simple hardware description language called HCL. Hardware designs written in HCL can be compiled and linked into simulators provided with the textbook, and they can be used to generate Verilog descriptions suitable for synthesis into working hardware.

- *Chapter 5: Optimizing Program Performance.* This chapter introduces a number of techniques for improving code performance, with the idea being that programmers learn to write their C code in such a way that a compiler can then generate efficient machine code. We start with transformations that reduce the work to be done by a program and hence should be standard practice when writing any program for any machine. We then progress to transformations that enhance the degree of instruction-level parallelism in the generated machine code, thereby improving their performance on modern "superscalar" processors. To motivate these transformations, we introduce a simple operational model of how modern out-of-order processors work, and show how to measure the potential performance of a program in terms of the critical paths through a graphical representation of a program. You will be surprised how much you can speed up a program by simple transformations of the C code.
- *Chapter 6: The Memory Hierarchy.* The memory system is one of the most visible parts of a computer system to application programmers. To this point, you have relied on a conceptual model of the memory system as a linear array with uniform access times. In practice, a memory system is a hierarchy of storage devices with different capacities, costs, and access times. We cover the different types of RAM and ROM memories and the geometry and organization of magnetic-disk and solid-state drives. We describe how these storage devices are arranged in a hierarchy. We show how this hierarchy is made possible by locality of reference. We make these ideas concrete by introducing a unique view of a memory system as a "memory mountain" with ridges of temporal locality and slopes of spatial locality. Finally, we show you how to improve the performance of application programs by improving their temporal and spatial locality.
- *Chapter 7: Linking.* This chapter covers both static and dynamic linking, including the ideas of relocatable and executable object files, symbol resolution, relocation, static libraries, shared object libraries, and position-independent code. Linking is not covered in most systems texts, but we cover it for several reasons. First, some of the most confusing errors that programmers can encounter are related to glitches during linking, especially for large software packages. Second, the object files produced by linkers are tied to concepts such as loading, virtual memory, and memory mapping.

- *Chapter 8: Exceptional Control Flow.* In this part of the presentation, we step beyond the single-program model by introducing the general concept of exceptional control flow (i.e., changes in control flow that are outside the normal branches and procedure calls). We cover examples of exceptional control flow that exist at all levels of the system, from low-level hardware exceptions and interrupts, to context switches between concurrent processes, to abrupt changes in control flow caused by the delivery of Unix signals, to the nonlocal jumps in C that break the stack discipline.

  This is the part of the book where we introduce the fundamental idea of a *process*, an abstraction of an executing program. You will learn how processes work and how they can be created and manipulated from application programs. We show how application programmers can make use of multiple processes via Unix system calls. When you finish this chapter, you will be able to write a Unix shell with job control. It is also your first introduction to the nondeterministic behavior that arises with concurrent program execution.
- *Chapter 9: Virtual Memory.* Our presentation of the virtual memory system seeks to give some understanding of how it works and its characteristics. We want you to know how it is that the different simultaneous processes can each use an identical range of addresses, sharing some pages but having individual copies of others. We also cover issues involved in managing and manipulating virtual memory. In particular, we cover the operation of storage allocators such as the Unix `malloc` and `free` operations. Covering this material serves several purposes. It reinforces the concept that the virtual memory space is just an array of bytes that the program can subdivide into different storage units. It helps you understand the effects of programs containing memory referencing errors such as storage leaks and invalid pointer references. Finally, many application programmers write their own storage allocators optimized toward the needs and characteristics of the application. This chapter, more than any other, demonstrates the benefit of covering both the hardware and the software aspects of computer systems in a unified way. Traditional computer architecture and operating systems texts present only part of the virtual memory story.
- *Chapter 10: System-Level I/O.* We cover the basic concepts of Unix I/O such as files and descriptors. We describe how files are shared, how I/O redirection works, and how to access file metadata. We also develop a robust buffered I/O package that deals correctly with a curious behavior known as *short counts*, where the library function reads only part of the input data. We cover the C standard I/O library and its relationship to Unix I/O, focusing on limitations of standard I/O that make it unsuitable for network programming. In general, the topics covered in this chapter are building blocks for the next two chapters on network and concurrent programming.
- *Chapter 11: Network Programming.* Networks are interesting I/O devices to program, tying together many of the ideas that we have studied earlier in the text, such as processes, signals, byte ordering, memory mapping, and dynamic

storage allocation. Network programs also provide a compelling context for concurrency, which is the topic of the next chapter. This chapter is a thin slice through network programming that gets you to the point where you can write a Web server. We cover the client-server model that underlies all network applications. We present a programmer's view of the Internet, and show how to write Internet clients and servers using the sockets interface. Finally, we introduce HTTP and develop a simple iterative Web server.

- *Chapter 12: Concurrent Programming.* This chapter introduces concurrent programming using Internet server design as the running motivational example. We compare and contrast the three basic mechanisms for writing concurrent programs—processes, I/O multiplexing, and threads—and show how to use them to build concurrent Internet servers. We cover basic principles of synchronization using *P* and *V* semaphore operations, thread safety and reentrancy, race conditions, and deadlocks. Writing concurrent code is essential for most server applications. We also describe the use of thread-level programming to express parallelism in an application program, enabling faster execution on multi-core processors. Getting all of the cores working on a single computational problem requires a careful coordination of the concurrent threads, both for correctness and to achieve high performance.

New to this Edition

The first edition of this book was published with a copyright of 2003. Considering the rapid evolution of computer technology, the book content has held up surprisingly well. Intel x86 machines running Unix-like operating systems and programmed in C proved to be a combination that continues to encompass many systems today. Changes in hardware technology and compilers and the experience of many instructors teaching the material have prompted a substantial revision.

Here are some of the more significant changes:

- *Chapter 2: Representing and Manipulating Information.* We have tried to make this material more accessible, with more careful explanations of concepts and with many more practice and homework problems. We moved some of the more theoretical aspects to Web asides. We also describe some of the security vulnerabilities that arise due to the overflow properties of computer arithmetic.
- *Chapter 3: Machine-Level Representation of Programs.* We have extended our coverage to include x86-64, the extension of x86 processors to a 64-bit word size. We also use the code generated by a more recent version of GCC. We have enhanced our coverage of buffer overflow vulnerabilities. We have created Web asides on two different classes of instructions for floating point, and also a view of the more exotic transformations made when compilers attempt higher degrees of optimization. Another Web aside describes how to embed x86 assembly code within a C program.

- *Chapter 4: Processor Architecture.* We include a more careful exposition of exception detection and handling in our processor design. We have also created a Web aside showing a mapping of our processor designs into Verilog, enabling synthesis into working hardware.
- *Chapter 5: Optimizing Program Performance.* We have greatly changed our description of how an out-of-order processor operates, and we have created a simple technique for analyzing program performance based on the paths in a data-flow graph representation of a program. A Web aside describes how C programmers can write programs that make use of the SIMD (single-instruction, multiple-data) instructions found in more recent versions of x86 processors.
- *Chapter 6: The Memory Hierarchy.* We have added material on solid-state disks, and we have updated our presentation to be based on the memory hierarchy of an Intel Core i7 processor.
- *Chapter 7: Linking.* This chapter has changed only slightly.
- *Chapter 8: Exceptional Control Flow.* We have enhanced our discussion of how the process model introduces some fundamental concepts of concurrency, such as nondeterminism.
- *Chapter 9: Virtual Memory.* We have updated our memory system case study to describe the 64-bit Intel Core i7 processor. We have also updated our sample implementation of `malloc` to work for both 32-bit and 64-bit execution.
- *Chapter 10: System-Level I/O.* This chapter has changed only slightly.
- *Chapter 11: Network Programming.* This chapter has changed only slightly.
- *Chapter 12: Concurrent Programming.* We have increased our coverage of the general principles of concurrency, and we also describe how programmers can use thread-level parallelism to make programs run faster on multi-core machines.

In addition, we have added and revised a number of practice and homework problems.

Origins of the Book

The book stems from an introductory course that we developed at Carnegie Mellon University in the Fall of 1998, called *15-213: Introduction to Computer Systems* (ICS) [14]. The ICS course has been taught every semester since then, each time to about 150–250 students, ranging from sophomores to masters degree students and with a wide variety of majors. It is a required course for all undergraduates in the CS and ECE departments at Carnegie Mellon, and it has become a prerequisite for most upper-level systems courses.

The idea with ICS was to introduce students to computers in a different way. Few of our students would have the opportunity to build a computer system. On the other hand, most students, including all computer scientists and computer engineers, will be required to use and program computers on a daily basis. So we

decided to teach about systems from the point of view of the programmer, using the following filter: we would cover a topic only if it affected the performance, correctness, or utility of user-level C programs.

For example, topics such as hardware adder and bus designs were out. Topics such as machine language were in, but instead of focusing on how to write assembly language by hand, we would look at how a C compiler translates C constructs into machine code, including pointers, loops, procedure calls, and switch statements. Further, we would take a broader and more holistic view of the system as both hardware and systems software, covering such topics as linking, loading, processes, signals, performance optimization, virtual memory, I/O, and network and concurrent programming.

This approach allowed us to teach the ICS course in a way that is practical, concrete, hands-on, and exciting for the students. The response from our students and faculty colleagues was immediate and overwhelmingly positive, and we realized that others outside of CMU might benefit from using our approach. Hence this book, which we developed from the ICS lecture notes, and which we have now revised to reflect changes in technology and how computer systems are implemented.

For Instructors: Courses Based on the Book

Instructors can use the CS:APP book to teach five different kinds of systems courses (Figure 2). The particular course depends on curriculum requirements, personal taste, and the backgrounds and abilities of the students. From left to right in the figure, the courses are characterized by an increasing emphasis on the programmer's perspective of a system. Here is a brief description:

- **ORG**: A computer organization course with traditional topics covered in an untraditional style. Traditional topics such as logic design, processor architecture, assembly language, and memory systems are covered. However, there is more emphasis on the impact for the programmer. For example, data representations are related back to the data types and operations of C programs, and the presentation on assembly code is based on machine code generated by a C compiler rather than hand-written assembly code.
- **ORG+**: The ORG course with additional emphasis on the impact of hardware on the performance of application programs. Compared to ORG, students learn more about code optimization and about improving the memory performance of their C programs.
- **ICS**: The baseline ICS course, designed to produce enlightened programmers who understand the impact of the hardware, operating system, and compilation system on the performance and correctness of their application programs. A significant difference from ORG+ is that low-level processor architecture is not covered. Instead, programmers work with a higher-level model of a modern out-of-order processor. The ICS course fits nicely into a 10-week quarter, and can also be stretched to a 15-week semester if covered at a more leisurely pace.

Chapter	Topic	Course				
		ORG	ORG+	ICS	ICS+	SP
1	Tour of systems	•	•	•	•	•
2	Data representation	•	•	•	•	⊙ (d)
3	Machine language	•	•	•	•	•
4	Processor architecture	•	•			
5	Code optimization		•	•	•	
6	Memory hierarchy	⊙ (a)	•	•	•	⊙ (a)
7	Linking			⊙ (c)	⊙ (c)	•
8	Exceptional control flow			•	•	•
9	Virtual memory	⊙ (b)	•	•	•	•
10	System-level I/O				•	•
11	Network programming				•	•
12	Concurrent programming				•	•

Figure 2 **Five systems courses based on the CS:APP book.** Notes: (a) Hardware only, (b) No dynamic storage allocation, (c) No dynamic linking, (d) No floating point. ICS+ is the 15-213 course from Carnegie Mellon.

- **ICS+**: The baseline ICS course with additional coverage of systems programming topics such as system-level I/O, network programming, and concurrent programming. This is the semester-long Carnegie Mellon course, which covers every chapter in CS:APP except low-level processor architecture.
- **SP**: A systems programming course. Similar to the ICS+ course, but drops floating point and performance optimization, and places more emphasis on systems programming, including process control, dynamic linking, system-level I/O, network programming, and concurrent programming. Instructors might want to supplement from other sources for advanced topics such as daemons, terminal control, and Unix IPC.

The main message of Figure 2 is that the CS:APP book gives a lot of options to students and instructors. If you want your students to be exposed to lower-level processor architecture, then that option is available via the ORG and ORG+ courses. On the other hand, if you want to switch from your current computer organization course to an ICS or ICS+ course, but are wary are making such a drastic change all at once, then you can move toward ICS incrementally. You can start with ORG, which teaches the traditional topics in a nontraditional way. Once you are comfortable with that material, then you can move to ORG+, and eventually to ICS. If students have no experience in C (for example they have only programmed in Java), you could spend several weeks on C and then cover the material of ORG or ICS.