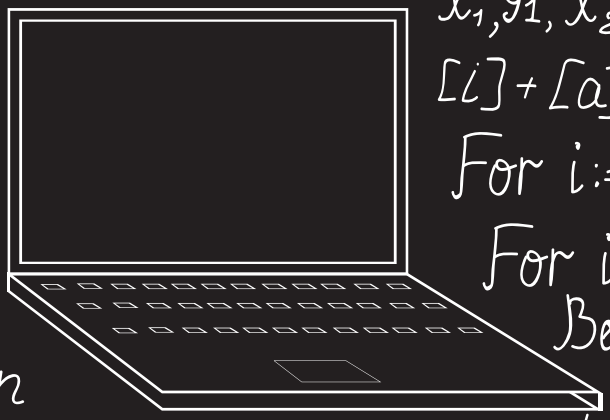
GLOBAL EDITION



Computer Systems

A Programmer's Perspective

THIRD EDITION

Randal E. Bryant • David R. O'Hallaron

ALWAYS LEARNING

PEARSON

Computer Systems

A Programmer's Perspective

Computer Systems

A Programmer's Perspective

THIRD EDITION
GLOBAL EDITION

Randal E. Bryant
Carnegie Mellon University

David R. O'Hallaron
Carnegie Mellon University

Global Edition contributions by

Manasa S.
NMAM Institute of Technology

Mohit Tahiliani
National Institute of Technology Karnataka

PEARSON

Boston Columbus Hoboken Indianapolis New York San Francisco
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Vice President and Editorial Director: Marcia J. Horton
Executive Editor: Matt Goldstein
Editorial Assistant: Kelsey Loanes
Acquisitions Editor, Global Editions: Karthik Subramanian
VP of Marketing: Christy Lesko
Director of Field Marketing: Tim Galligan
Product Marketing Manager: Bram van Kempen
Field Marketing Manager: Demetrius Hall
Marketing Assistant: Jon Bryant
Director of Product Management: Erin Gregg
Team Lead Product Management: Scott Disanno
Program Manager: Joanne Manning
Project Editor, Global Editions: K.K. Neelakantan
Senior Production Manufacturing Controller, Global Editions: Trudy Kimber
Procurement Manager: Mary Fischer
Senior Specialist, Program Planning and Support: Maura Zaldivar-Garcia
Media Production Manager, Global Editions: Vikram Kumar
Cover Designer: Lumina Datamatics
Manager, Rights Management: Rachel Youdelman
Associate Project Manager, Rights Management: William J. Opaluch
Full-Service Project Management: Paul Anagnostopoulos, Windfall Software

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the world

Visit us on the World Wide Web at:
www.pearsonglobaleditions.com

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library

10 9 8 7 6 5 4 3 2 1

ISBN 10: 1-292-10176-8

ISBN 13: 978-1-292-10176-7 (Print)

ISBN 13: 978-1-488-67207-1 (PDF)

Typeset in 10/12 Times Ten, ITC Stone Sans by Windfall Software

Printed in Malaysia

To the students and instructors of the 15-213 course at Carnegie Mellon University, for inspiring us to develop and refine the material for this book.

MasteringEngineering®

For *Computer Systems: A Programmer's Perspective,* Third Edition

---

Mastering is Pearson's proven online Tutorial Homework program, newly available with the third edition of *Computer Systems: A Programmer's Perspective*. The Mastering platform allows you to integrate dynamic homework—with many problems taken directly from the Bryant/O'Hallaron textbook—with automatic grading. Mastering allows you to easily track the performance of your entire class on an assignment-by-assignment basis, or view the detailed work of an individual student.

For more information or a demonstration of the course, visit www.MasteringEngineering.com

Contents

Preface 19

About the Authors 35

1

A Tour of Computer Systems 37

1.1 Information Is Bits + Context 39
1.2 Programs Are Translated by Other Programs into Different Forms 40
1.3 It Pays to Understand How Compilation Systems Work 42
1.4 Processors Read and Interpret Instructions Stored in Memory 43
1.4.1 Hardware Organization of a System 44
1.4.2 Running the `hello` Program 46
1.5 Caches Matter 47
1.6 Storage Devices Form a Hierarchy 50
1.7 The Operating System Manages the Hardware 50
1.7.1 Processes 51
1.7.2 Threads 53
1.7.3 Virtual Memory 54
1.7.4 Files 55
1.8 Systems Communicate with Other Systems Using Networks 55
1.9 Important Themes 58
1.9.1 Amdahl's Law 58
1.9.2 Concurrency and Parallelism 60
1.9.3 The Importance of Abstractions in Computer Systems 62
1.10 Summary 63
Bibliographic Notes 64
Solutions to Practice Problems 64

Part I Program Structure and Execution

2

Representing and Manipulating Information 67

2.1 Information Storage 70
2.1.1 Hexadecimal Notation 72
2.1.2 Data Sizes 75

2.1.3 Addressing and Byte Ordering 78
2.1.4 Representing Strings 85
2.1.5 Representing Code 85
2.1.6 Introduction to Boolean Algebra 86
2.1.7 Bit-Level Operations in C 90
2.1.8 Logical Operations in C 92
2.1.9 Shift Operations in C 93
2.2 Integer Representations 95
2.2.1 Integral Data Types 96
2.2.2 Unsigned Encodings 98
2.2.3 Two's-Complement Encodings 100
2.2.4 Conversions between Signed and Unsigned 106
2.2.5 Signed versus Unsigned in C 110
2.2.6 Expanding the Bit Representation of a Number 112
2.2.7 Truncating Numbers 117
2.2.8 Advice on Signed versus Unsigned 119
2.3 Integer Arithmetic 120
2.3.1 Unsigned Addition 120
2.3.2 Two's-Complement Addition 126
2.3.3 Two's-Complement Negation 131
2.3.4 Unsigned Multiplication 132
2.3.5 Two's-Complement Multiplication 133
2.3.6 Multiplying by Constants 137
2.3.7 Dividing by Powers of 2 139
2.3.8 Final Thoughts on Integer Arithmetic 143
2.4 Floating Point 144
2.4.1 Fractional Binary Numbers 145
2.4.2 IEEE Floating-Point Representation 148
2.4.3 Example Numbers 151
2.4.4 Rounding 156
2.4.5 Floating-Point Operations 158
2.4.6 Floating Point in C 160
2.5 Summary 162
Bibliographic Notes 163
Homework Problems 164
Solutions to Practice Problems 179

3

Machine-Level Representation of Programs 199

3.1 A Historical Perspective 202

3.2 Program Encodings 205
3.2.1 Machine-Level Code 206
3.2.2 Code Examples 208
3.2.3 Notes on Formatting 211
3.3 Data Formats 213
3.4 Accessing Information 215
3.4.1 Operand Specifiers 216
3.4.2 Data Movement Instructions 218
3.4.3 Data Movement Example 222
3.4.4 Pushing and Popping Stack Data 225
3.5 Arithmetic and Logical Operations 227
3.5.1 Load Effective Address 227
3.5.2 Unary and Binary Operations 230
3.5.3 Shift Operations 230
3.5.4 Discussion 232
3.5.5 Special Arithmetic Operations 233
3.6 Control 236
3.6.1 Condition Codes 237
3.6.2 Accessing the Condition Codes 238
3.6.3 Jump Instructions 241
3.6.4 Jump Instruction Encodings 243
3.6.5 Implementing Conditional Branches with Conditional Control 245
3.6.6 Implementing Conditional Branches with Conditional Moves 250
3.6.7 Loops 256
3.6.8 Switch Statements 268
3.7 Procedures 274
3.7.1 The Run-Time Stack 275
3.7.2 Control Transfer 277
3.7.3 Data Transfer 281
3.7.4 Local Storage on the Stack 284
3.7.5 Local Storage in Registers 287
3.7.6 Recursive Procedures 289
3.8 Array Allocation and Access 291
3.8.1 Basic Principles 291
3.8.2 Pointer Arithmetic 293
3.8.3 Nested Arrays 294
3.8.4 Fixed-Size Arrays 296
3.8.5 Variable-Size Arrays 298

3.9 Heterogeneous Data Structures 301
3.9.1 Structures 301
3.9.2 Unions 305
3.9.3 Data Alignment 309
3.10 Combining Control and Data in Machine-Level Programs 312
3.10.1 Understanding Pointers 313
3.10.2 Life in the Real World: Using the GDB Debugger 315
3.10.3 Out-of-Bounds Memory References and Buffer Overflow 315
3.10.4 Thwarting Buffer Overflow Attacks 320
3.10.5 Supporting Variable-Size Stack Frames 326
3.11 Floating-Point Code 329
3.11.1 Floating-Point Movement and Conversion Operations 332
3.11.2 Floating-Point Code in Procedures 337
3.11.3 Floating-Point Arithmetic Operations 338
3.11.4 Defining and Using Floating-Point Constants 340
3.11.5 Using Bitwise Operations in Floating-Point Code 341
3.11.6 Floating-Point Comparison Operations 342
3.11.7 Observations about Floating-Point Code 345
3.12 Summary 345
Bibliographic Notes 346
Homework Problems 347
Solutions to Practice Problems 361

4

Processor Architecture 387

4.1 The Y86-64 Instruction Set Architecture 391
4.1.1 Programmer-Visible State 391
4.1.2 Y86-64 Instructions 392
4.1.3 Instruction Encoding 394
4.1.4 Y86-64 Exceptions 399
4.1.5 Y86-64 Programs 400
4.1.6 Some Y86-64 Instruction Details 406
4.2 Logic Design and the Hardware Control Language HCL 408
4.2.1 Logic Gates 409
4.2.2 Combinational Circuits and HCL Boolean Expressions 410
4.2.3 Word-Level Combinational Circuits and HCL Integer Expressions 412
4.2.4 Set Membership 416
4.2.5 Memory and Clocking 417
4.3 Sequential Y86-64 Implementations 420
4.3.1 Organizing Processing into Stages 420

4.3.2 SEQ Hardware Structure 432
4.3.3 SEQ Timing 436
4.3.4 SEQ Stage Implementations 440
4.4 General Principles of Pipelining 448
4.4.1 Computational Pipelines 448
4.4.2 A Detailed Look at Pipeline Operation 450
4.4.3 Limitations of Pipelining 452
4.4.4 Pipelining a System with Feedback 455
4.5 Pipelined Y86-64 Implementations 457
4.5.1 SEQ+: Rearranging the Computation Stages 457
4.5.2 Inserting Pipeline Registers 458
4.5.3 Rearranging and Relabeling Signals 462
4.5.4 Next PC Prediction 463
4.5.5 Pipeline Hazards 465
4.5.6 Exception Handling 480
4.5.7 PIPE Stage Implementations 483
4.5.8 Pipeline Control Logic 491
4.5.9 Performance Analysis 500
4.5.10 Unfinished Business 504
4.6 Summary 506
4.6.1 Y86-64 Simulators 508
Bibliographic Notes 509
Homework Problems 509
Solutions to Practice Problems 516

5

Optimizing Program Performance 531

5.1 Capabilities and Limitations of Optimizing Compilers 534
5.2 Expressing Program Performance 538
5.3 Program Example 540
5.4 Eliminating Loop Inefficiencies 544
5.5 Reducing Procedure Calls 548
5.6 Eliminating Unneeded Memory References 550
5.7 Understanding Modern Processors 553
5.7.1 Overall Operation 554
5.7.2 Functional Unit Performance 559
5.7.3 An Abstract Model of Processor Operation 561
5.8 Loop Unrolling 567
5.9 Enhancing Parallelism 572
5.9.1 Multiple Accumulators 572
5.9.2 Reassociation Transformation 577

5.10 Summary of Results for Optimizing Combining Code 583
5.11 Some Limiting Factors 584
5.11.1 Register Spilling 584
5.11.2 Branch Prediction and Misprediction Penalties 585
5.12 Understanding Memory Performance 589
5.12.1 Load Performance 590
5.12.2 Store Performance 591
5.13 Life in the Real World: Performance Improvement Techniques 597
5.14 Identifying and Eliminating Performance Bottlenecks 598
5.14.1 Program Profiling 598
5.14.2 Using a Profiler to Guide Optimization 601
5.15 Summary 604
Bibliographic Notes 605
Homework Problems 606
Solutions to Practice Problems 609

6

The Memory Hierarchy 615

6.1 Storage Technologies 617
6.1.1 Random Access Memory 617
6.1.2 Disk Storage 625
6.1.3 Solid State Disks 636
6.1.4 Storage Technology Trends 638
6.2 Locality 640
6.2.1 Locality of References to Program Data 642
6.2.2 Locality of Instruction Fetches 643
6.2.3 Summary of Locality 644
6.3 The Memory Hierarchy 645
6.3.1 Caching in the Memory Hierarchy 646
6.3.2 Summary of Memory Hierarchy Concepts 650
6.4 Cache Memories 650
6.4.1 Generic Cache Memory Organization 651
6.4.2 Direct-Mapped Caches 653
6.4.3 Set Associative Caches 660
6.4.4 Fully Associative Caches 662
6.4.5 Issues with Writes 666
6.4.6 Anatomy of a Real Cache Hierarchy 667
6.4.7 Performance Impact of Cache Parameters 667
6.5 Writing Cache-Friendly Code 669
6.6 Putting It Together: The Impact of Caches on Program Performance 675

6.6.1 The Memory Mountain 675
6.6.2 Rearranging Loops to Increase Spatial Locality 679
6.6.3 Exploiting Locality in Your Programs 683
6.7 Summary 684
Bibliographic Notes 684
Homework Problems 685
Solutions to Practice Problems 696

Part II Running Programs on a System

7
Linking 705

7.1 Compiler Drivers 707
7.2 Static Linking 708
7.3 Object Files 709
7.4 Relocatable Object Files 710
7.5 Symbols and Symbol Tables 711
7.6 Symbol Resolution 715
7.6.1 How Linkers Resolve Duplicate Symbol Names 716
7.6.2 Linking with Static Libraries 720
7.6.3 How Linkers Use Static Libraries to Resolve References 724
7.7 Relocation 725
7.7.1 Relocation Entries 726
7.7.2 Relocating Symbol References 727
7.8 Executable Object Files 731
7.9 Loading Executable Object Files 733
7.10 Dynamic Linking with Shared Libraries 734
7.11 Loading and Linking Shared Libraries from Applications 737
7.12 Position-Independent Code (PIC) 740
7.13 Library Interpositioning 743
7.13.1 Compile-Time Interpositioning 744
7.13.2 Link-Time Interpositioning 744
7.13.3 Run-Time Interpositioning 746
7.14 Tools for Manipulating Object Files 749
7.15 Summary 749
Bibliographic Notes 750
Homework Problems 750
Solutions to Practice Problems 753

8

Exceptional Control Flow 757

8.1 Exceptions 759
8.1.1 Exception Handling 760
8.1.2 Classes of Exceptions 762
8.1.3 Exceptions in Linux/x86-64 Systems 765
8.2 Processes 768
8.2.1 Logical Control Flow 768
8.2.2 Concurrent Flows 769
8.2.3 Private Address Space 770
8.2.4 User and Kernel Modes 770
8.2.5 Context Switches 772
8.3 System Call Error Handling 773
8.4 Process Control 774
8.4.1 Obtaining Process IDs 775
8.4.2 Creating and Terminating Processes 775
8.4.3 Reaping Child Processes 779
8.4.4 Putting Processes to Sleep 785
8.4.5 Loading and Running Programs 786
8.4.6 Using `fork` and `execve` to Run Programs 789
8.5 Signals 792
8.5.1 Signal Terminology 794
8.5.2 Sending Signals 795
8.5.3 Receiving Signals 798
8.5.4 Blocking and Unblocking Signals 800
8.5.5 Writing Signal Handlers 802
8.5.6 Synchronizing Flows to Avoid Nasty Concurrency Bugs 812
8.5.7 Explicitly Waiting for Signals 814
8.6 Nonlocal Jumps 817
8.7 Tools for Manipulating Processes 822
8.8 Summary 823
Bibliographic Notes 823
Homework Problems 824
Solutions to Practice Problems 831

9

Virtual Memory 837

9.1 Physical and Virtual Addressing 839
9.2 Address Spaces 840

9.3 VM as a Tool for Caching 841
9.3.1 DRAM Cache Organization 842
9.3.2 Page Tables 842
9.3.3 Page Hits 844
9.3.4 Page Faults 844
9.3.5 Allocating Pages 846
9.3.6 Locality to the Rescue Again 846
9.4 VM as a Tool for Memory Management 847
9.5 VM as a Tool for Memory Protection 848
9.6 Address Translation 849
9.6.1 Integrating Caches and VM 853
9.6.2 Speeding Up Address Translation with a TLB 853
9.6.3 Multi-Level Page Tables 855
9.6.4 Putting It Together: End-to-End Address Translation 857
9.7 Case Study: The Intel Core i7/Linux Memory System 861
9.7.1 Core i7 Address Translation 862
9.7.2 Linux Virtual Memory System 864
9.8 Memory Mapping 869
9.8.1 Shared Objects Revisited 869
9.8.2 The `fork` Function Revisited 872
9.8.3 The `execve` Function Revisited 872
9.8.4 User-Level Memory Mapping with the `mmap` Function 873
9.9 Dynamic Memory Allocation 875
9.9.1 The `malloc` and `free` Functions 876
9.9.2 Why Dynamic Memory Allocation? 879
9.9.3 Allocator Requirements and Goals 880
9.9.4 Fragmentation 882
9.9.5 Implementation Issues 882
9.9.6 Implicit Free Lists 883
9.9.7 Placing Allocated Blocks 885
9.9.8 Splitting Free Blocks 885
9.9.9 Getting Additional Heap Memory 886
9.9.10 Coalescing Free Blocks 886
9.9.11 Coalescing with Boundary Tags 887
9.9.12 Putting It Together: Implementing a Simple Allocator 890
9.9.13 Explicit Free Lists 898
9.9.14 Segregated Free Lists 899
9.10 Garbage Collection 901
9.10.1 Garbage Collector Basics 902
9.10.2 Mark&Sweep Garbage Collectors 903
9.10.3 Conservative Mark&Sweep for C Programs 905

9.11 Common Memory-Related Bugs in C Programs 906
9.11.1 Dereferencing Bad Pointers 906
9.11.2 Reading Uninitialized Memory 907
9.11.3 Allowing Stack Buffer Overflows 907
9.11.4 Assuming That Pointers and the Objects They Point to Are the Same Size 908
9.11.5 Making Off-by-One Errors 908
9.11.6 Referencing a Pointer Instead of the Object It Points To 909
9.11.7 Misunderstanding Pointer Arithmetic 909
9.11.8 Referencing Nonexistent Variables 910
9.11.9 Referencing Data in Free Heap Blocks 910
9.11.10 Introducing Memory Leaks 911
9.12 Summary 911
Bibliographic Notes 912
Homework Problems 912
Solutions to Practice Problems 916

Part III Interaction and Communication between Programs

10
System-Level I/O 925

10.1 Unix I/O 926
10.2 Files 927
10.3 Opening and Closing Files 929
10.4 Reading and Writing Files 931
10.5 Robust Reading and Writing with the Rio Package 933
10.5.1 Rio Unbuffered Input and Output Functions 933
10.5.2 Rio Buffered Input Functions 934
10.6 Reading File Metadata 939
10.7 Reading Directory Contents 941
10.8 Sharing Files 942
10.9 I/O Redirection 945
10.10 Standard I/O 947
10.11 Putting It Together: Which I/O Functions Should I Use? 947
10.12 Summary 949
Bibliographic Notes 950
Homework Problems 950
Solutions to Practice Problems 951

11

Network Programming 953

11.1 The Client-Server Programming Model 954
11.2 Networks 955
11.3 The Global IP Internet 960
11.3.1 IP Addresses 961
11.3.2 Internet Domain Names 963
11.3.3 Internet Connections 965
11.4 The Sockets Interface 968
11.4.1 Socket Address Structures 969
11.4.2 The `socket` Function 970
11.4.3 The `connect` Function 970
11.4.4 The `bind` Function 971
11.4.5 The `listen` Function 971
11.4.6 The `accept` Function 972
11.4.7 Host and Service Conversion 973
11.4.8 Helper Functions for the Sockets Interface 978
11.4.9 Example Echo Client and Server 980
11.5 Web Servers 984
11.5.1 Web Basics 984
11.5.2 Web Content 985
11.5.3 HTTP Transactions 986
11.5.4 Serving Dynamic Content 989
11.6 Putting It Together: The TINY Web Server 992
11.7 Summary 1000
Bibliographic Notes 1001
Homework Problems 1001
Solutions to Practice Problems 1002

12

Concurrent Programming 1007

12.1 Concurrent Programming with Processes 1009
12.1.1 A Concurrent Server Based on Processes 1010
12.1.2 Pros and Cons of Processes 1011
12.2 Concurrent Programming with I/O Multiplexing 1013
12.2.1 A Concurrent Event-Driven Server Based on I/O Multiplexing 1016
12.2.2 Pros and Cons of I/O Multiplexing 1021
12.3 Concurrent Programming with Threads 1021
12.3.1 Thread Execution Model 1022

12.3.2 Posix Threads 1023
12.3.3 Creating Threads 1024
12.3.4 Terminating Threads 1024
12.3.5 Reaping Terminated Threads 1025
12.3.6 Detaching Threads 1025
12.3.7 Initializing Threads 1026
12.3.8 A Concurrent Server Based on Threads 1027
12.4 Shared Variables in Threaded Programs 1028
12.4.1 Threads Memory Model 1029
12.4.2 Mapping Variables to Memory 1030
12.4.3 Shared Variables 1031
12.5 Synchronizing Threads with Semaphores 1031
12.5.1 Progress Graphs 1035
12.5.2 Semaphores 1037
12.5.3 Using Semaphores for Mutual Exclusion 1038
12.5.4 Using Semaphores to Schedule Shared Resources 1040
12.5.5 Putting It Together: A Concurrent Server Based on Prethreading 1044
12.6 Using Threads for Parallelism 1049
12.7 Other Concurrency Issues 1056
12.7.1 Thread Safety 1056
12.7.2 Reentrancy 1059
12.7.3 Using Existing Library Functions in Threaded Programs 1060
12.7.4 Races 1061
12.7.5 Deadlocks 1063
12.8 Summary 1066
Bibliographic Notes 1066
Homework Problems 1067
Solutions to Practice Problems 1072

A

Error Handling 1077

A.1 Error Handling in Unix Systems 1078
A.2 Error-Handling Wrappers 1079

References 1083

Index 1089

Preface

This book (known as 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. Many 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, so this book also serves as a valuable introduction to those who go on to implement systems hardware and software. Most systems books also tend to focus on just one aspect of the system, for example, the hardware architecture, the operating system, the compiler, or the network. This book spans all of these aspects, with the unifying theme of a programmer's perspective.

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. You will be able to write programs that make better use of the capabilities provided by the operating system and systems software, that operate correctly across a wide range of operating conditions and run-time parameters, that run faster, and that avoid the flaws that make programs vulnerable to cyberattack. You will be prepared to delve deeper into advanced topics such as compilers, computer architecture, operating systems, embedded systems, networking, and cybersecurity.

Assumptions about the Reader's Background

This book focuses on systems that execute x86-64 machine code. x86-64 is the latest in an evolutionary path followed by Intel and its competitors that started with the 8086 microprocessor in 1978. Due to the naming conventions used by Intel for its microprocessor line, this class of microprocessors is referred to colloquially as "x86." As semiconductor technology has evolved to allow more transistors to be integrated onto a single chip, these processors have progressed greatly in their computing power and their memory capacity. As part of this progression, they have gone from operating on 16-bit words, to 32-bit words with the introduction of IA32 processors, and most recently to 64-bit words with x86-64.

We consider how these machines execute C programs on Linux. Linux is one of a number of operating systems having their heritage in the Unix operating system developed originally by Bell Laboratories. Other members of this class

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.

of operating systems include Solaris, FreeBSD, and MacOS X. In recent years, these operating systems have maintained a high level of compatibility through the efforts of the Posix and Standard Unix Specification standardization efforts. Thus, the material in this book applies almost directly to these "Unix-like" operating systems.

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 listing files and changing directories. If your computer runs Microsoft Windows, we recommend that you install one of the many different virtual machine environments (such as VirtualBox or VMWare) that allow programs written for one operating system (the guest OS) to run under another (the host OS).

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 [61]. Regardless of your programming background, consider K&R an essential part of your personal systems library. If your prior experience is with an interpreted language, such as Python, Ruby, or Perl, you will definitely want to devote some time to learning C before you attempt to use this book.

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 x86-64 processors. We do not assume any prior experience with hardware, machine language, or assembly-language programming.

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 firsthand. 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