THIRD EDITION

COMPUTER SYSTEMS

A PROGRAMMER'S PERSPECTIVE

BRYANT • O'HALLARON

Computer Organization and Architecture / Computer Systems

ISBN 978-0-13-409266-9

90000

9 780134 092669

www.pearsonhighered.com

PEARSON

ALWAYS LEARNING

Computer Systems

A Programmer's Perspective

Computer Systems
A Programmer's Perspective

THIRD EDITION

Randal E. Bryant
Carnegie Mellon University

David R. O'Hallaron
Carnegie Mellon University

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 Loans
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
Procurement Manager: Mary Fischer
Senior Specialist, Program Planning and Support: Maura Zaldivar-Garcia
Cover Designer: Joyce Wells
Manager, Rights Management: Rachel Youdelman
Associate Project Manager, Rights Management: William J. Opaluch
Full-Service Project Management: Paul Anagnostopoulos, Windfall Software
Composition: Windfall Software
Printer/Binder: Courier Westford
Cover Printer: Courier Westford
Typeface: 10/12 Times 10, ITC Stone Sans

The graph on the front cover is a “memory mountain” that shows the measured read throughput of an Intel Core i7 processor as a function of spatial and temporal locality.

Pearson Education Ltd., *London*
Pearson Education Singapore, Pte. Ltd
Pearson Education Canada, Inc.
Pearson Education—Japan
Pearson Education Australia PTY, Limited
Pearson Education North Asia, Ltd., *Hong Kong*
Pearson Educacin̈ de Mexico, S.A. de C.V.
Pearson Education Malaysia, Pte. Ltd.
Pearson Education, Inc., *Upper Saddle River, New Jersey*

Library of Congress Cataloging-in-Publication Data

Bryant, Randal E.
Computer systems : a programmer’s perspective / Randal E. Bryant, Carnegie Mellon University, David R. O’Hallaron, Carnegie Mellon. University.—Third edition.
pages cm
Includes bibliographical references and index.
ISBN 978-0-13-409266-9—ISBN 0-13-409266-X
1. Computer systems. 2. Computers. 3. Telecommunication. 4. User interfaces (Computer systems) I. O’Hallaron, David R. (David Richard) II. Title.
QA76.5.B795 2016
005.3—dc23 2015000930

10 9 8 7 6 5 4 3 2 1

PEARSON

www.pearsonhighered.com

ISBN 10: 0-13-409266-X
ISBN 13: 978-0-13-409266-9

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.

Contents

Preface xix

About the Authors xxxv

1

A Tour of Computer Systems 1

1.1 Information Is Bits + Context 3
1.2 Programs Are Translated by Other Programs into Different Forms 4
1.3 It Pays to Understand How Compilation Systems Work 6
1.4 Processors Read and Interpret Instructions Stored in Memory 7
- 1.4.1 Hardware Organization of a System 8
- 1.4.2 Running the `hello` Program 10

1.5 Caches Matter 11
1.6 Storage Devices Form a Hierarchy 14
1.7 The Operating System Manages the Hardware 14
- 1.7.1 Processes 15
- 1.7.2 Threads 17
- 1.7.3 Virtual Memory 18
- 1.7.4 Files 19

1.8 Systems Communicate with Other Systems Using Networks 19
1.9 Important Themes 22
- 1.9.1 Amdahl's Law 22
- 1.9.2 Concurrency and Parallelism 24
- 1.9.3 The Importance of Abstractions in Computer Systems 26

1.10 Summary 27
Bibliographic Notes 28
Solutions to Practice Problems 28

Part I Program Structure and Execution

2

Representing and Manipulating Information 31

2.1 Information Storage 34
- 2.1.1 Hexadecimal Notation 36
- 2.1.2 Data Sizes 39

2.1.3 Addressing and Byte Ordering 42
2.1.4 Representing Strings 49
2.1.5 Representing Code 49
2.1.6 Introduction to Boolean Algebra 50
2.1.7 Bit-Level Operations in C 54
2.1.8 Logical Operations in C 56
2.1.9 Shift Operations in C 57
2.2 Integer Representations 59
2.2.1 Integral Data Types 60
2.2.2 Unsigned Encodings 62
2.2.3 Two's-Complement Encodings 64
2.2.4 Conversions between Signed and Unsigned 70
2.2.5 Signed versus Unsigned in C 74
2.2.6 Expanding the Bit Representation of a Number 76
2.2.7 Truncating Numbers 81
2.2.8 Advice on Signed versus Unsigned 83
2.3 Integer Arithmetic 84
2.3.1 Unsigned Addition 84
2.3.2 Two's-Complement Addition 90
2.3.3 Two's-Complement Negation 95
2.3.4 Unsigned Multiplication 96
2.3.5 Two's-Complement Multiplication 97
2.3.6 Multiplying by Constants 101
2.3.7 Dividing by Powers of 2 103
2.3.8 Final Thoughts on Integer Arithmetic 107
2.4 Floating Point 108
2.4.1 Fractional Binary Numbers 109
2.4.2 IEEE Floating-Point Representation 112
2.4.3 Example Numbers 115
2.4.4 Rounding 120
2.4.5 Floating-Point Operations 122
2.4.6 Floating Point in C 124
2.5 Summary 126
Bibliographic Notes 127
Homework Problems 128
Solutions to Practice Problems 143

3

Machine-Level Representation of Programs 163

3.1 A Historical Perspective 166

3.2 Program Encodings 169
3.2.1 Machine-Level Code 170
3.2.2 Code Examples 172
3.2.3 Notes on Formatting 175

3.3 Data Formats 177

3.4 Accessing Information 179
3.4.1 Operand Specifiers 180
3.4.2 Data Movement Instructions 182
3.4.3 Data Movement Example 186
3.4.4 Pushing and Popping Stack Data 189

3.5 Arithmetic and Logical Operations 191
3.5.1 Load Effective Address 191
3.5.2 Unary and Binary Operations 194
3.5.3 Shift Operations 194
3.5.4 Discussion 196
3.5.5 Special Arithmetic Operations 197

3.6 Control 200
3.6.1 Condition Codes 201
3.6.2 Accessing the Condition Codes 202
3.6.3 Jump Instructions 205
3.6.4 Jump Instruction Encodings 207
3.6.5 Implementing Conditional Branches with Conditional Control 209
3.6.6 Implementing Conditional Branches with Conditional Moves 214
3.6.7 Loops 220
3.6.8 Switch Statements 232

3.7 Procedures 238
3.7.1 The Run-Time Stack 239
3.7.2 Control Transfer 241
3.7.3 Data Transfer 245
3.7.4 Local Storage on the Stack 248
3.7.5 Local Storage in Registers 251
3.7.6 Recursive Procedures 253

3.8 Array Allocation and Access 255
3.8.1 Basic Principles 255
3.8.2 Pointer Arithmetic 257
3.8.3 Nested Arrays 258
3.8.4 Fixed-Size Arrays 260
3.8.5 Variable-Size Arrays 262

3.9 Heterogeneous Data Structures 265
3.9.1 Structures 265
3.9.2 Unions 269
3.9.3 Data Alignment 273
3.10 Combining Control and Data in Machine-Level Programs 276
3.10.1 Understanding Pointers 277
3.10.2 Life in the Real World: Using the GDB Debugger 279
3.10.3 Out-of-Bounds Memory References and Buffer Overflow 279
3.10.4 Thwarting Buffer Overflow Attacks 284
3.10.5 Supporting Variable-Size Stack Frames 290
3.11 Floating-Point Code 293
3.11.1 Floating-Point Movement and Conversion Operations 296
3.11.2 Floating-Point Code in Procedures 301
3.11.3 Floating-Point Arithmetic Operations 302
3.11.4 Defining and Using Floating-Point Constants 304
3.11.5 Using Bitwise Operations in Floating-Point Code 305
3.11.6 Floating-Point Comparison Operations 306
3.11.7 Observations about Floating-Point Code 309
3.12 Summary 309
Bibliographic Notes 310
Homework Problems 311
Solutions to Practice Problems 325

4

Processor Architecture 351

4.1 The Y86-64 Instruction Set Architecture 355
4.1.1 Programmer-Visible State 355
4.1.2 Y86-64 Instructions 356
4.1.3 Instruction Encoding 358
4.1.4 Y86-64 Exceptions 363
4.1.5 Y86-64 Programs 364
4.1.6 Some Y86-64 Instruction Details 370
4.2 Logic Design and the Hardware Control Language HCL 372
4.2.1 Logic Gates 373
4.2.2 Combinational Circuits and HCL Boolean Expressions 374
4.2.3 Word-Level Combinational Circuits and HCL Integer Expressions 376
4.2.4 Set Membership 380
4.2.5 Memory and Clocking 381
4.3 Sequential Y86-64 Implementations 384
4.3.1 Organizing Processing into Stages 384

4.3.2 SEQ Hardware Structure 396
4.3.3 SEQ Timing 400
4.3.4 SEQ Stage Implementations 404
4.4 General Principles of Pipelining 412
4.4.1 Computational Pipelines 412
4.4.2 A Detailed Look at Pipeline Operation 414
4.4.3 Limitations of Pipelining 416
4.4.4 Pipelining a System with Feedback 419
4.5 Pipelined Y86-64 Implementations 421
4.5.1 SEQ+: Rearranging the Computation Stages 421
4.5.2 Inserting Pipeline Registers 422
4.5.3 Rearranging and Relabeling Signals 426
4.5.4 Next PC Prediction 427
4.5.5 Pipeline Hazards 429
4.5.6 Exception Handling 444
4.5.7 PIPE Stage Implementations 447
4.5.8 Pipeline Control Logic 455
4.5.9 Performance Analysis 464
4.5.10 Unfinished Business 468
4.6 Summary 470
4.6.1 Y86-64 Simulators 472
Bibliographic Notes 473
Homework Problems 473
Solutions to Practice Problems 480

5

Optimizing Program Performance 495

5.1 Capabilities and Limitations of Optimizing Compilers 498
5.2 Expressing Program Performance 502
5.3 Program Example 504
5.4 Eliminating Loop Inefficiencies 508
5.5 Reducing Procedure Calls 512
5.6 Eliminating Unneeded Memory References 514
5.7 Understanding Modern Processors 517
5.7.1 Overall Operation 518
5.7.2 Functional Unit Performance 523
5.7.3 An Abstract Model of Processor Operation 525
5.8 Loop Unrolling 531
5.9 Enhancing Parallelism 536
5.9.1 Multiple Accumulators 536
5.9.2 Reassociation Transformation 541

5.10 Summary of Results for Optimizing Combining Code 547
5.11 Some Limiting Factors 548
5.11.1 Register Spilling 548
5.11.2 Branch Prediction and Misprediction Penalties 549
5.12 Understanding Memory Performance 553
5.12.1 Load Performance 554
5.12.2 Store Performance 555
5.13 Life in the Real World: Performance Improvement Techniques 561
5.14 Identifying and Eliminating Performance Bottlenecks 562
5.14.1 Program Profiling 562
5.14.2 Using a Profiler to Guide Optimization 565
5.15 Summary 568
Bibliographic Notes 569
Homework Problems 570
Solutions to Practice Problems 573

6

The Memory Hierarchy 579

6.1 Storage Technologies 581
6.1.1 Random Access Memory 581
6.1.2 Disk Storage 589
6.1.3 Solid State Disks 600
6.1.4 Storage Technology Trends 602
6.2 Locality 604
6.2.1 Locality of References to Program Data 606
6.2.2 Locality of Instruction Fetches 607
6.2.3 Summary of Locality 608
6.3 The Memory Hierarchy 609
6.3.1 Caching in the Memory Hierarchy 610
6.3.2 Summary of Memory Hierarchy Concepts 614
6.4 Cache Memories 614
6.4.1 Generic Cache Memory Organization 615
6.4.2 Direct-Mapped Caches 617
6.4.3 Set Associative Caches 624
6.4.4 Fully Associative Caches 626
6.4.5 Issues with Writes 630
6.4.6 Anatomy of a Real Cache Hierarchy 631
6.4.7 Performance Impact of Cache Parameters 631
6.5 Writing Cache-Friendly Code 633
6.6 Putting It Together: The Impact of Caches on Program Performance 639

6.6.1 The Memory Mountain 639
6.6.2 Rearranging Loops to Increase Spatial Locality 643
6.6.3 Exploiting Locality in Your Programs 647
6.7 Summary 648
Bibliographic Notes 648
Homework Problems 649
Solutions to Practice Problems 660

Part II Running Programs on a System

7 Linking 669

7.1 Compiler Drivers 671
7.2 Static Linking 672
7.3 Object Files 673
7.4 Relocatable Object Files 674
7.5 Symbols and Symbol Tables 675
7.6 Symbol Resolution 679
7.6.1 How Linkers Resolve Duplicate Symbol Names 680
7.6.2 Linking with Static Libraries 684
7.6.3 How Linkers Use Static Libraries to Resolve References 688
7.7 Relocation 689
7.7.1 Relocation Entries 690
7.7.2 Relocating Symbol References 691
7.8 Executable Object Files 695
7.9 Loading Executable Object Files 697
7.10 Dynamic Linking with Shared Libraries 698
7.11 Loading and Linking Shared Libraries from Applications 701
7.12 Position-Independent Code (PIC) 704
7.13 Library Interpositioning 707
7.13.1 Compile-Time Interpositioning 708
7.13.2 Link-Time Interpositioning 708
7.13.3 Run-Time Interpositioning 710
7.14 Tools for Manipulating Object Files 713
7.15 Summary 713
Bibliographic Notes 714
Homework Problems 714
Solutions to Practice Problems 717

8

Exceptional Control Flow 721

8.1 Exceptions 723
- 8.1.1 Exception Handling 724
- 8.1.2 Classes of Exceptions 726
- 8.1.3 Exceptions in Linux/x86-64 Systems 729

8.2 Processes 732
- 8.2.1 Logical Control Flow 732
- 8.2.2 Concurrent Flows 733
- 8.2.3 Private Address Space 734
- 8.2.4 User and Kernel Modes 734
- 8.2.5 Context Switches 736

8.3 System Call Error Handling 737

8.4 Process Control 738
- 8.4.1 Obtaining Process IDs 739
- 8.4.2 Creating and Terminating Processes 739
- 8.4.3 Reaping Child Processes 743
- 8.4.4 Putting Processes to Sleep 749
- 8.4.5 Loading and Running Programs 750
- 8.4.6 Using `fork` and `execve` to Run Programs 753

8.5 Signals 756
- 8.5.1 Signal Terminology 758
- 8.5.2 Sending Signals 759
- 8.5.3 Receiving Signals 762
- 8.5.4 Blocking and Unblocking Signals 764
- 8.5.5 Writing Signal Handlers 766
- 8.5.6 Synchronizing Flows to Avoid Nasty Concurrency Bugs 776
- 8.5.7 Explicitly Waiting for Signals 778

8.6 Nonlocal Jumps 781

8.7 Tools for Manipulating Processes 786

8.8 Summary 787

Bibliographic Notes 787

Homework Problems 788

Solutions to Practice Problems 795

9

Virtual Memory 801

9.1 Physical and Virtual Addressing 803

9.2 Address Spaces 804

9.3 VM as a Tool for Caching 805
9.3.1 DRAM Cache Organization 806
9.3.2 Page Tables 806
9.3.3 Page Hits 808
9.3.4 Page Faults 808
9.3.5 Allocating Pages 810
9.3.6 Locality to the Rescue Again 810
9.4 VM as a Tool for Memory Management 811
9.5 VM as a Tool for Memory Protection 812
9.6 Address Translation 813
9.6.1 Integrating Caches and VM 817
9.6.2 Speeding Up Address Translation with a TLB 817
9.6.3 Multi-Level Page Tables 819
9.6.4 Putting It Together: End-to-End Address Translation 821
9.7 Case Study: The Intel Core i7/Linux Memory System 825
9.7.1 Core i7 Address Translation 826
9.7.2 Linux Virtual Memory System 828
9.8 Memory Mapping 833
9.8.1 Shared Objects Revisited 833
9.8.2 The `fork` Function Revisited 836
9.8.3 The `execve` Function Revisited 836
9.8.4 User-Level Memory Mapping with the `mmap` Function 837
9.9 Dynamic Memory Allocation 839
9.9.1 The `malloc` and `free` Functions 840
9.9.2 Why Dynamic Memory Allocation? 843
9.9.3 Allocator Requirements and Goals 844
9.9.4 Fragmentation 846
9.9.5 Implementation Issues 846
9.9.6 Implicit Free Lists 847
9.9.7 Placing Allocated Blocks 849
9.9.8 Splitting Free Blocks 849
9.9.9 Getting Additional Heap Memory 850
9.9.10 Coalescing Free Blocks 850
9.9.11 Coalescing with Boundary Tags 851
9.9.12 Putting It Together: Implementing a Simple Allocator 854
9.9.13 Explicit Free Lists 862
9.9.14 Segregated Free Lists 863
9.10 Garbage Collection 865
9.10.1 Garbage Collector Basics 866
9.10.2 Mark&Sweep Garbage Collectors 867
9.10.3 Conservative Mark&Sweep for C Programs 869

9.11 Common Memory-Related Bugs in C Programs 870
9.11.1 Dereferencing Bad Pointers 870
9.11.2 Reading Uninitialized Memory 871
9.11.3 Allowing Stack Buffer Overflows 871
9.11.4 Assuming That Pointers and the Objects They Point to Are the Same Size 872
9.11.5 Making Off-by-One Errors 872
9.11.6 Referencing a Pointer Instead of the Object It Points To 873
9.11.7 Misunderstanding Pointer Arithmetic 873
9.11.8 Referencing Nonexistent Variables 874
9.11.9 Referencing Data in Free Heap Blocks 874
9.11.10 Introducing Memory Leaks 875
9.12 Summary 875
Bibliographic Notes 876
Homework Problems 876
Solutions to Practice Problems 880

Part III Interaction and Communication between Programs

10

System-Level I/O 889

10.1 Unix I/O 890
10.2 Files 891
10.3 Opening and Closing Files 893
10.4 Reading and Writing Files 895
10.5 Robust Reading and Writing with the Rio Package 897
10.5.1 Rio Unbuffered Input and Output Functions 897
10.5.2 Rio Buffered Input Functions 898
10.6 Reading File Metadata 903
10.7 Reading Directory Contents 905
10.8 Sharing Files 906
10.9 I/O Redirection 909
10.10 Standard I/O 911
10.11 Putting It Together: Which I/O Functions Should I Use? 911
10.12 Summary 913
Bibliographic Notes 914
Homework Problems 914
Solutions to Practice Problems 915

11

Network Programming 917

11.1 The Client-Server Programming Model 918
11.2 Networks 919
11.3 The Global IP Internet 924
11.3.1 IP Addresses 925
11.3.2 Internet Domain Names 927
11.3.3 Internet Connections 929
11.4 The Sockets Interface 932
11.4.1 Socket Address Structures 933
11.4.2 The `socket` Function 934
11.4.3 The `connect` Function 934
11.4.4 The `bind` Function 935
11.4.5 The `listen` Function 935
11.4.6 The `accept` Function 936
11.4.7 Host and Service Conversion 937
11.4.8 Helper Functions for the Sockets Interface 942
11.4.9 Example Echo Client and Server 944
11.5 Web Servers 948
11.5.1 Web Basics 948
11.5.2 Web Content 949
11.5.3 HTTP Transactions 950
11.5.4 Serving Dynamic Content 953
11.6 Putting It Together: The Tiny Web Server 956
11.7 Summary 964
Bibliographic Notes 965
Homework Problems 965
Solutions to Practice Problems 966

12

Concurrent Programming 971

12.1 Concurrent Programming with Processes 973
12.1.1 A Concurrent Server Based on Processes 974
12.1.2 Pros and Cons of Processes 975
12.2 Concurrent Programming with I/O Multiplexing 977
12.2.1 A Concurrent Event-Driven Server Based on I/O Multiplexing 980
12.2.2 Pros and Cons of I/O Multiplexing 985
12.3 Concurrent Programming with Threads 985
12.3.1 Thread Execution Model 986

12.3.2 Posix Threads 987
12.3.3 Creating Threads 988
12.3.4 Terminating Threads 988
12.3.5 Reaping Terminated Threads 989
12.3.6 Detaching Threads 989
12.3.7 Initializing Threads 990
12.3.8 A Concurrent Server Based on Threads 991
12.4 Shared Variables in Threaded Programs 992
12.4.1 Threads Memory Model 993
12.4.2 Mapping Variables to Memory 994
12.4.3 Shared Variables 995
12.5 Synchronizing Threads with Semaphores 995
12.5.1 Progress Graphs 999
12.5.2 Semaphores 1001
12.5.3 Using Semaphores for Mutual Exclusion 1002
12.5.4 Using Semaphores to Schedule Shared Resources 1004
12.5.5 Putting It Together: A Concurrent Server Based on Prethreading 1008
12.6 Using Threads for Parallelism 1013
12.7 Other Concurrency Issues 1020
12.7.1 Thread Safety 1020
12.7.2 Reentrancy 1023
12.7.3 Using Existing Library Functions in Threaded Programs 1024
12.7.4 Races 1025
12.7.5 Deadlocks 1027
12.8 Summary 1030
Bibliographic Notes 1030
Homework Problems 1031
Solutions to Practice Problems 1036

A

Error Handling 1041

A.1 Error Handling in Unix Systems 1042
A.2 Error-Handling Wrappers 1043

References 1047

Index 1053

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 *code/intro/hello.c*

```c
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```

code/intro/hello.c

Figure 1 A typical code example.

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.

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, 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 ("CS:APP" being our shorthand for the book's title) at csapp.cs.cmu.edu. In the text, the filenames 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 a 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.

Book Overview

The CS:APP book consists of 12 chapters designed to capture the core ideas in computer systems. Here is an overview.

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 many of the algebraic properties of integer arithmetic, 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 x86-64 machine code 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 cover the instructions that implement both integer and floating-point arithmetic. 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 pro-

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, such as how a floating-point error crashed a French rocket or the geometric and operational parameters of a commercial disk drive. Finally, some asides are just fun stuff. For example, what is a "hoinky"?

grammer, the compiler, and the operating system can take to reduce 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 x86-64 instruction set called "Y86-64." 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, position-independent code, and library interpositioning. Linking is not covered in most systems texts, but we cover it for two 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 receipt of Linux 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 Linux system calls. When you finish this chapter, you will be able to write a simple Linux 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 standard-library `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 Linux 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 study 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 simple 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, while the second had a copyright of 2011. Considering the rapid evolution of computer technology, the book content has held up surprisingly well. Intel x86 machines running C programs under Linux (and related operating systems) has proved to be a combination that continues to encompass many systems today. However, changes in hardware technology, compilers, program library interfaces, and the experience of many instructors teaching the material have prompted a substantial revision.

The biggest overall change from the second edition is that we have switched our presentation from one based on a mix of IA32 and x86-64 to one based exclusively on x86-64. This shift in focus affected the contents of many of the chapters. Here is a summary of the significant changes.

Chapter 1: A Tour of Computer Systems We have moved the discussion of Amdahl's Law from Chapter 5 into this chapter.

Chapter 2: Representing and Manipulating Information. A consistent bit of feedback from readers and reviewers is that some of the material in this chapter can be a bit overwhelming. So we have tried to make the material more accessible by clarifying the points at which we delve into a more mathematical style of presentation. This enables readers to first skim over mathematical details to get a high-level overview and then return for a more thorough reading.

Chapter 3: Machine-Level Representation of Programs. We have converted from the earlier presentation based on a mix of IA32 and x86-64 to one based entirely on x86-64. We have also updated for the style of code generated by more recent versions of GCC. The result is a substantial rewriting, including changing the order in which some of the concepts are presented. We also have included, for the first time, a presentation of the machine-level support for programs operating on floating-point data. We have created a Web aside describing IA32 machine code for legacy reasons.

Chapter 4: Processor Architecture. We have revised the earlier processor design, based on a 32-bit architecture, to one that supports 64-bit words and operations.

Chapter 5: Optimizing Program Performance. We have updated the material to reflect the performance capabilities of recent generations of x86-64 processors. With the introduction of more functional units and more sophisticated control logic, the model of program performance we developed based on a data-flow representation of programs has become a more reliable predictor of performance than it was before.

Chapter 6: The Memory Hierarchy. We have updated the material to reflect more recent technology.

Chapter 7: Linking. We have rewritten this chapter for x86-64, expanded the discussion of using the GOT and PLT to create position-independent code, and added a new section on a powerful linking technique known as *library interpositioning.*

Chapter 8: Exceptional Control Flow. We have added a more rigorous treatment of signal handlers, including async-signal-safe functions, specific guidelines for writing signal handlers, and using `sigsuspend` to wait for handlers.

Chapter 9: Virtual Memory. This chapter has changed only slightly.

Chapter 10: System-Level I/O. We have added a new section on files and the file hierarchy, but otherwise, this chapter has changed only slightly.

Chapter 11: Network Programming. We have introduced techniques for protocol-independent and thread-safe network programming using the modern `getaddrinfo` and `getnameinfo` functions, which replace the obsolete and non-reentrant `gethostbyname` and `gethostbyaddr` functions.

Chapter 12: Concurrent Programming. We have increased our coverage of using 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 throughout the text.

Origins of the Book

This 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. Over 400 students take the course each semester. The students range from sophomores to graduate students in a wide variety of majors. It is a required core 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 in CS and ECE.

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, would 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 in how computer systems are implemented.

Via the multiple editions and multiple translations of this book, ICS and many variants have become part of the computer science and computer engineering curricula at hundreds of colleges and universities worldwide.

For Instructors: Courses Based on the Book

Instructors can use the CS:APP book to teach a number of different types of systems courses. Five categories of these courses are illustrated in 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 handwritten 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.

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.

		Course				
Chapter	Topic	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.** ICS+ is the 15-213 course from Carnegie Mellon. Notes: The ⊙ symbol denotes partial coverage of a chapter, as follows: (a) hardware only; (b) no dynamic storage allocation; (c) no dynamic linking; (d) no floating point.

SP. A systems programming course. This course is similar to ICS+, but it drops floating point and performance optimization, and it 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 of 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 (e.g., they have only programmed in Java), you could spend several weeks on C and then cover the material of ORG or ICS.

Finally, we note that the ORG+ and SP courses would make a nice two-term sequence (either quarters or semesters). Or you might consider offering ICS+ as one term of ICS and one term of SP.

For Instructors: Classroom-Tested Laboratory Exercises

The ICS+ course at Carnegie Mellon receives very high evaluations from students. Median scores of 5.0/5.0 and means of 4.6/5.0 are typical for the student course evaluations. Students cite the fun, exciting, and relevant laboratory exercises as the primary reason. The labs are available from the CS:APP Web page. Here are examples of the labs that are provided with the book.

Data Lab. This lab requires students to implement simple logical and arithmetic functions, but using a highly restricted subset of C. For example, they must compute the absolute value of a number using only bit-level operations. This lab helps students understand the bit-level representations of C data types and the bit-level behavior of the operations on data.

Binary Bomb Lab. A *binary bomb* is a program provided to students as an object-code file. When run, it prompts the user to type in six different strings. If any of these are incorrect, the bomb "explodes," printing an error message and logging the event on a grading server. Students must "defuse" their own unique bombs by disassembling and reverse engineering the programs to determine what the six strings should be. The lab teaches students to understand assembly language and also forces them to learn how to use a debugger.

Buffer Overflow Lab. Students are required to modify the run-time behavior of a binary executable by exploiting a buffer overflow vulnerability. This lab teaches the students about the stack discipline and about the danger of writing code that is vulnerable to buffer overflow attacks.

Architecture Lab. Several of the homework problems of Chapter 4 can be combined into a lab assignment, where students modify the HCL description of a processor to add new instructions, change the branch prediction policy, or add or remove bypassing paths and register ports. The resulting processors can be simulated and run through automated tests that will detect most of the possible bugs. This lab lets students experience the exciting parts of processor design without requiring a complete background in logic design and hardware description languages.

Performance Lab. Students must optimize the performance of an application kernel function such as convolution or matrix transposition. This lab provides a very clear demonstration of the properties of cache memories and gives students experience with low-level program optimization.

Cache Lab. In this alternative to the performance lab, students write a general-purpose cache simulator, and then optimize a small matrix transpose kernel to minimize the number of misses on a simulated cache. We use the Valgrind tool to generate real address traces for the matrix transpose kernel.

Shell Lab. Students implement their own Unix shell program with job control, including the Ctrl+C and Ctrl+Z keystrokes and the `fg`, `bg`, and `jobs` commands. This is the student's first introduction to concurrency, and it gives them a clear idea of Unix process control, signals, and signal handling.

Malloc Lab. Students implement their own versions of `malloc`, `free`, and (optionally) `realloc`. This lab gives students a clear understanding of data layout and organization, and requires them to evaluate different trade-offs between space and time efficiency.

Proxy Lab. Students implement a concurrent Web proxy that sits between their browsers and the rest of the World Wide Web. This lab exposes the students to such topics as Web clients and servers, and ties together many of the concepts from the course, such as byte ordering, file I/O, process control, signals, signal handling, memory mapping, sockets, and concurrency. Students like being able to see their programs in action with real Web browsers and Web servers.

The CS:APP instructor's manual has a detailed discussion of the labs, as well as directions for downloading the support software.

Acknowledgments for the Third Edition

It is a pleasure to acknowledge and thank those who have helped us produce this third edition of the CS:APP text.

We would like to thank our Carnegie Mellon colleagues who have taught the ICS course over the years and who have provided so much insightful feedback and encouragement: Guy Blelloch, Roger Dannenberg, David Eckhardt, Franz Franchetti, Greg Ganger, Seth Goldstein, Khaled Harras, Greg Kesden, Bruce Maggs, Todd Mowry, Andreas Nowatzyk, Frank Pfenning, Markus Pueschel, and Anthony Rowe. David Winters was very helpful in installing and configuring the reference Linux box.

Jason Fritts (St. Louis University) and Cindy Norris (Appalachian State) provided us with detailed and thoughtful reviews of the second edition. Yili Gong (Wuhan University) wrote the Chinese translation, maintained the errata page for the Chinese version, and contributed many bug reports. Godmar Back (Virginia Tech) helped us improve the text significantly by introducing us to the notions of async-signal safety and protocol-independent network programming.

Many thanks to our eagle-eyed readers who reported bugs in the second edition: Rami Ammari, Paul Anagnostopoulos, Lucas Bärenfänger, Godmar Back, Ji Bin, Sharbel Bousemaan, Richard Callahan, Seth Chaiken, Cheng Chen, Libo Chen, Tao Du, Pascal Garcia, Yili Gong, Ronald Greenberg, Dorukhan Gülöz, Dong Han, Dominik Helm, Ronald Jones, Mustafa Kazdagli, Gordon Kindlmann, Sankar Krishnan, Kanak Kshetri, Junlin Lu, Qiangqiang Luo, Sebastian Luy, Lei Ma, Ashwin Nanjappa, Gregoire Paradis, Jonas Pfenninger, Karl Pichotta, David Ramsey, Kaustabh Roy, David Selvaraj, Sankar Shanmugam, Dominique Smulkowska, Dag Sørbø, Michael Spear, Yu Tanaka, Steven Tricanowicz, Scott Wright, Waiki Wright, Han Xu, Zhengshan Yan, Firo Yang, Shuang Yang, John Ye, Taketo Yoshida, Yan Zhu, and Michael Zink.

Thanks also to our readers who have contributed to the labs, including Godmar Back (Virginia Tech), Taymon Beal (Worcester Polytechnic Institute), Aran Clauson (Western Washington University), Cary Gray (Wheaton College), Paul Haiduk (West Texas A&M University), Len Hamey (Macquarie University), Eddie Kohler (Harvard), Hugh Lauer (Worcester Polytechnic Institute), Robert Marmorstein (Longwood University), and James Riely (DePaul University).

Once again, Paul Anagnostopoulos of Windfall Software did a masterful job of typesetting the book and leading the production process. Many thanks to Paul and his stellar team: Richard Camp (copyediting), Jennifer McClain (proofreading), Laurel Muller (art production), and Ted Laux (indexing). Paul even spotted a bug in our description of the origins of the acronym BSS that had persisted undetected since the first edition!

Finally, we would like to thank our friends at Prentice Hall. Marcia Horton and our editor, Matt Goldstein, have been unflagging in their support and encouragement, and we are deeply grateful to them.

Acknowledgments from the Second Edition

We are deeply grateful to the many people who have helped us produce this second edition of the CS:APP text.

First and foremost, we would like to recognize our colleagues who have taught the ICS course at Carnegie Mellon for their insightful feedback and encouragement: Guy Blelloch, Roger Dannenberg, David Eckhardt, Greg Ganger, Seth Goldstein, Greg Kesden, Bruce Maggs, Todd Mowry, Andreas Nowatzyk, Frank Pfenning, and Markus Pueschel.

Thanks also to our sharp-eyed readers who contributed reports to the errata page for the first edition: Daniel Amelang, Rui Baptista, Quarup Barreirinhas, Michael Bombyk, Jörg Brauer, Jordan Brough, Yixin Cao, James Caroll, Rui Carvalho, Hyoung-Kee Choi, Al Davis, Grant Davis, Christian Dufour, Mao Fan, Tim Freeman, Inge Frick, Max Gebhardt, Jeff Goldblat, Thomas Gross, Anita Gupta, John Hampton, Hiep Hong, Greg Israelsen, Ronald Jones, Haudy Kazemi, Brian Kell, Constantine Kousoulis, Sacha Krakowiak, Arun Krishnaswamy, Martin Kulas, Michael Li, Zeyang Li, Ricky Liu, Mario Lo Conte, Dirk Maas, Devon Macey, Carl Marcinik, Will Marrero, Simone Martins, Tao Men, Mark Morrissey, Venkata Naidu, Bhas Nalabothula, Thomas Niemann, Eric Peskin, David Po, Anne Rogers, John Ross, Michael Scott, Seiki, Ray Shih, Darren Shultz, Erik Silkensen, Suryanto, Emil Tarazi, Nawanan Theera-Ampornpunt, Joe Trdinich, Michael Trigoboff, James Troup, Martin Vopatek, Alan West, Betsy Wolff, Tim Wong, James Woodruff, Scott Wright, Jackie Xiao, Guanpeng Xu, Qing Xu, Caren Yang, Yin Yongsheng, Wang Yuanxuan, Steven Zhang, and Day Zhong. Special thanks to Inge Frick, who identified a subtle deep copy bug in our lock-and-copy example, and to Ricky Liu for his amazing proofreading skills.

Our Intel Labs colleagues Andrew Chien and Limor Fix were exceptionally supportive throughout the writing of the text. Steve Schlosser graciously provided some disk drive characterizations. Casey Helfrich and Michael Ryan installed and maintained our new Core i7 box. Michael Kozuch, Babu Pillai, and Jason Campbell provided valuable insight on memory system performance, multi-core systems, and the power wall. Phil Gibbons and Shimin Chen shared their considerable expertise on solid state disk designs.

We have been able to call on the talents of many, including Wen-Mei Hwu, Markus Pueschel, and Jiri Simsa, to provide both detailed comments and high-level advice. James Hoe helped us create a Verilog version of the Y86 processor and did all of the work needed to synthesize working hardware.

Many thanks to our colleagues who provided reviews of the draft manuscript: James Archibald (Brigham Young University), Richard Carver (George Mason University), Mirela Damian (Villanova University), Peter Dinda (Northwestern University), John Fiore (Temple University), Jason Fritts (St. Louis University), John Greiner (Rice University), Brian Harvey (University of California, Berkeley), Don Heller (Penn State University), Wei Chung Hsu (University of Minnesota), Michelle Hugue (University of Maryland), Jeremy Johnson (Drexel University), Geoff Kuenning (Harvey Mudd College), Ricky Liu, Sam Madden (MIT), Fred Martin (University of Massachusetts, Lowell), Abraham Matta (Boston University), Markus Pueschel (Carnegie Mellon University), Norman Ramsey (Tufts University), Glenn Reinmann (UCLA), Michela Taufer (University of Delaware), and Craig Zilles (UIUC).

Paul Anagnostopoulos of Windfall Software did an outstanding job of typesetting the book and leading the production team. Many thanks to Paul and his superb team: Rick Camp (copyeditor), Joe Snowden (compositor), MaryEllen N. Oliver (proofreader), Laurel Muller (artist), and Ted Laux (indexer).

Finally, we would like to thank our friends at Prentice Hall. Marcia Horton has always been there for us. Our editor, Matt Goldstein, provided stellar leadership from beginning to end. We are profoundly grateful for their help, encouragement, and insights.

Acknowledgments from the First Edition

We are deeply indebted to many friends and colleagues for their thoughtful criticisms and encouragement. A special thanks to our 15-213 students, whose infectious energy and enthusiasm spurred us on. Nick Carter and Vinny Furia generously provided their malloc package.

Guy Blelloch, Greg Kesden, Bruce Maggs, and Todd Mowry taught the course over multiple semesters, gave us encouragement, and helped improve the course material. Herb Derby provided early spiritual guidance and encouragement. Allan Fisher, Garth Gibson, Thomas Gross, Satya, Peter Steenkiste, and Hui Zhang encouraged us to develop the course from the start. A suggestion from Garth early on got the whole ball rolling, and this was picked up and refined with the help of a group led by Allan Fisher. Mark Stehlik and Peter Lee have been very supportive about building this material into the undergraduate curriculum. Greg Kesden provided helpful feedback on the impact of ICS on the OS course. Greg Ganger and Jiri Schindler graciously provided some disk drive characterizations

and answered our questions on modern disks. Tom Stricker showed us the memory mountain. James Hoe provided useful ideas and feedback on how to present processor architecture.

A special group of students—Khalil Amiri, Angela Demke Brown, Chris Colohan, Jason Crawford, Peter Dinda, Julio Lopez, Bruce Lowekamp, Jeff Pierce, Sanjay Rao, Balaji Sarpeshkar, Blake Scholl, Sanjit Seshia, Greg Steffan, Tiankai Tu, Kip Walker, and Yinglian Xie—were instrumental in helping us develop the content of the course. In particular, Chris Colohan established a fun (and funny) tone that persists to this day, and invented the legendary "binary bomb" that has proven to be a great tool for teaching machine code and debugging concepts.

Chris Bauer, Alan Cox, Peter Dinda, Sandhya Dwarkadas, John Greiner, Don Heller, Bruce Jacob, Barry Johnson, Bruce Lowekamp, Greg Morrisett, Brian Noble, Bobbie Othmer, Bill Pugh, Michael Scott, Mark Smotherman, Greg Steffan, and Bob Wier took time that they did not have to read and advise us on early drafts of the book. A very special thanks to Al Davis (University of Utah), Peter Dinda (Northwestern University), John Greiner (Rice University), Wei Hsu (University of Minnesota), Bruce Lowekamp (College of William & Mary), Bobbie Othmer (University of Minnesota), Michael Scott (University of Rochester), and Bob Wier (Rocky Mountain College) for class testing the beta version. A special thanks to their students as well!

We would also like to thank our colleagues at Prentice Hall. Marcia Horton, Eric Frank, and Harold Stone have been unflagging in their support and vision. Harold also helped us present an accurate historical perspective on RISC and CISC processor architectures. Jerry Ralya provided sharp insights and taught us a lot about good writing.

Finally, we would like to acknowledge the great technical writers Brian Kernighan and the late W. Richard Stevens, for showing us that technical books can be beautiful.

Thank you all.

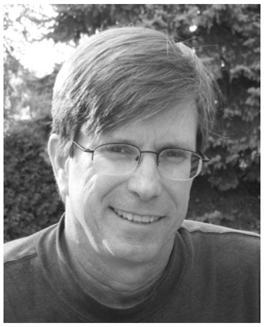
Randy Bryant
Dave O'Hallaron
Pittsburgh, Pennsylvania

About the Authors

Randal E. Bryant received his bachelor's degree from the University of Michigan in 1973 and then attended graduate school at the Massachusetts Institute of Technology, receiving his PhD degree in computer science in 1981. He spent three years as an assistant professor at the California Institute of Technology, and has been on the faculty at Carnegie Mellon since 1984. For five of those years he served as head of the Computer Science Department, and for ten of them he served as Dean of the School of Computer Science. He is currently a university professor of computer science. He also holds a courtesy appointment with the Department of Electrical and Computer Engineering.

Professor Bryant has taught courses in computer systems at both the undergraduate and graduate level for around 40 years. Over many years of teaching computer architecture courses, he began shifting the focus from how computers are designed to how programmers can write more efficient and reliable programs if they understand the system better. Together with Professor O'Hallaron, he developed the course 15-213, Introduction to Computer Systems, at Carnegie Mellon that is the basis for this book. He has also taught courses in algorithms, programming, computer networking, distributed systems, and VLSI design.

Most of Professor Bryant's research concerns the design of software tools to help software and hardware designers verify the correctness of their systems. These include several types of simulators, as well as formal verification tools that prove the correctness of a design using mathematical methods. He has published over 150 technical papers. His research results are used by major computer manufacturers, including Intel, IBM, Fujitsu, and Microsoft. He has won several major awards for his research. These include two inventor recognition awards and a technical achievement award from the Semiconductor Research Corporation, the Kanellakis Theory and Practice Award from the Association for Computer Machinery (ACM), and the W. R. G. Baker Award, the Emmanuel Piore Award, the Phil Kaufman Award, and the A. Richard Newton Award from the Institute of Electrical and Electronics Engineers (IEEE). He is a fellow of both the ACM and the IEEE and a member of both the US National Academy of Engineering and the American Academy of Arts and Sciences.



David R. O'Hallaron is a professor of computer science and electrical and computer engineering at Carnegie Mellon University. He received his PhD from the University of Virginia. He served as the director of Intel Labs, Pittsburgh, from 2007 to 2010.

He has taught computer systems courses at the undergraduate and graduate levels for 20 years on such topics as computer architecture, introductory computer systems, parallel processor design, and Internet services. Together with Professor Bryant, he developed the course at Carnegie Mellon that led to this book. In 2004, he was awarded the Herbert Simon Award for Teaching Excellence by the CMU School of Computer Science, an award for which the winner is chosen based on a poll of the students.

Professor O'Hallaron works in the area of computer systems, with specific interests in software systems for scientific computing, data-intensive computing, and virtualization. The best-known example of his work is the Quake project, an endeavor involving a group of computer scientists, civil engineers, and seismologists who have developed the ability to predict the motion of the ground during strong earthquakes. In 2003, Professor O'Hallaron and the other members of the Quake team won the Gordon Bell Prize, the top international prize in high-performance computing. His current work focuses on the notion of autograding, that is, programs that evaluate the quality of other programs.

CHAPTER 1

A Tour of Computer Systems

1.1 Information Is Bits + Context 3

1.2 Programs Are Translated by Other Programs into Different Forms 4

1.3 It Pays to Understand How Compilation Systems Work 6

1.4 Processors Read and Interpret Instructions Stored in Memory 7

1.5 Caches Matter 11

1.6 Storage Devices Form a Hierarchy 14

1.7 The Operating System Manages the Hardware 14

1.8 Systems Communicate with Other Systems Using Networks 19

1.9 Important Themes 22

1.10 Summary 27

Bibliographic Notes 28

Solutions to Practice Problems 28

A *computer system* consists of hardware and systems software that work together to run application programs. Specific implementations of systems change over time, but the underlying concepts do not. All computer systems have similar hardware and software components that perform similar functions. This book is written for programmers who want to get better at their craft by understanding how these components work and how they affect the correctness and performance of their programs.

You are poised for an exciting journey. If you dedicate yourself to learning the concepts in this book, then you will be on your way to becoming a rare "power programmer," enlightened by an understanding of the underlying computer system and its impact on your application programs.

You are going to learn practical skills such as how to avoid strange numerical errors caused by the way that computers represent numbers. You will learn how to optimize your C code by using clever tricks that exploit the designs of modern processors and memory systems. You will learn how the compiler implements procedure calls and how to use this knowledge to avoid the security holes from buffer overflow vulnerabilities that plague network and Internet software. You will learn how to recognize and avoid the nasty errors during linking that confound the average programmer. You will learn how to write your own Unix shell, your own dynamic storage allocation package, and even your own Web server. You will learn the promises and pitfalls of concurrency, a topic of increasing importance as multiple processor cores are integrated onto single chips.

In their classic text on the C programming language [61], Kernighan and Ritchie introduce readers to C using the `hello` program shown in Figure 1.1. Although `hello` is a very simple program, every major part of the system must work in concert in order for it to run to completion. In a sense, the goal of this book is to help you understand what happens and why when you run `hello` on your system.

We begin our study of systems by tracing the lifetime of the `hello` program, from the time it is created by a programmer, until it runs on a system, prints its simple message, and terminates. As we follow the lifetime of the program, we will briefly introduce the key concepts, terminology, and components that come into play. Later chapters will expand on these ideas.

code/intro/hello.c

```c
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```

code/intro/hello.c

Figure 1.1 The `hello` program. (Source: [60])

#	i	n	c	l	u	d	e	*SP*	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	*SP*	m	a	i	n	(	)	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	*SP*	*SP*	*SP*	*SP*	p	r	i	n	t	f	(	"	h	e	l
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	*SP*	w	o	r	l	d	\	n	"	)	;	\n	*SP*
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	32
SP	*SP*	*SP*	r	e	t	u	r	n	*SP*	0	;	\n	}	\n	
32	32	32	114	101	116	117	114	110	32	48	59	10	125	10	

Figure 1.2 The ASCII text representation of `hello.c`.

1.1 Information Is Bits + Context

Our `hello` program begins life as a *source program* (or *source file*) that the programmer creates with an editor and saves in a text file called `hello.c`. The source program is a sequence of bits, each with a value of 0 or 1, organized in 8-bit chunks called *bytes*. Each byte represents some text character in the program.

Most computer systems represent text characters using the ASCII standard that represents each character with a unique byte-size integer value.[1] For example, Figure 1.2 shows the ASCII representation of the `hello.c` program.

The `hello.c` program is stored in a file as a sequence of bytes. Each byte has an integer value that corresponds to some character. For example, the first byte has the integer value 35, which corresponds to the character '`#`'. The second byte has the integer value 105, which corresponds to the character '`i`', and so on. Notice that each text line is terminated by the invisible *newline* character '`\n`', which is represented by the integer value 10. Files such as `hello.c` that consist exclusively of ASCII characters are known as *text files*. All other files are known as *binary files*.

The representation of `hello.c` illustrates a fundamental idea: All information in a system—including disk files, programs stored in memory, user data stored in memory, and data transferred across a network—is represented as a bunch of bits. The only thing that distinguishes different data objects is the context in which we view them. For example, in different contexts, the same sequence of bytes might represent an integer, floating-point number, character string, or machine instruction.

As programmers, we need to understand machine representations of numbers because they are not the same as integers and real numbers. They are finite

1. Other encoding methods are used to represent text in non-English languages. See the aside on page 50 for a discussion on this.

Aside Origins of the C programming language

C was developed from 1969 to 1973 by Dennis Ritchie of Bell Laboratories. The American National Standards Institute (ANSI) ratified the ANSI C standard in 1989, and this standardization later became the responsibility of the International Standards Organization (ISO). The standards define the C language and a set of library functions known as the *C standard library*. Kernighan and Ritchie describe ANSI C in their classic book, which is known affectionately as "K&R" [61]. In Ritchie's words [92], C is "quirky, flawed, and an enormous success." So why the success?

- *C was closely tied with the Unix operating system.* C was developed from the beginning as the system programming language for Unix. Most of the Unix kernel (the core part of the operating system), and all of its supporting tools and libraries, were written in C. As Unix became popular in universities in the late 1970s and early 1980s, many people were exposed to C and found that they liked it. Since Unix was written almost entirely in C, it could be easily ported to new machines, which created an even wider audience for both C and Unix.
- *C is a small, simple language.* The design was controlled by a single person, rather than a committee, and the result was a clean, consistent design with little baggage. The K&R book describes the complete language and standard library, with numerous examples and exercises, in only 261 pages. The simplicity of C made it relatively easy to learn and to port to different computers.
- *C was designed for a practical purpose.* C was designed to implement the Unix operating system. Later, other people found that they could write the programs they wanted, without the language getting in the way.

C is the language of choice for system-level programming, and there is a huge installed base of application-level programs as well. However, it is not perfect for all programmers and all situations. C pointers are a common source of confusion and programming errors. C also lacks explicit support for useful abstractions such as classes, objects, and exceptions. Newer languages such as C++ and Java address these issues for application-level programs.

approximations that can behave in unexpected ways. This fundamental idea is explored in detail in Chapter 2.

1.2 Programs Are Translated by Other Programs into Different Forms

The `hello` program begins life as a high-level C program because it can be read and understood by human beings in that form. However, in order to run `hello.c` on the system, the individual C statements must be translated by other programs into a sequence of low-level *machine-language* instructions. These instructions are then packaged in a form called an *executable object program* and stored as a binary disk file. Object programs are also referred to as *executable object files*.

On a Unix system, the translation from source file to object file is performed by a *compiler driver:*

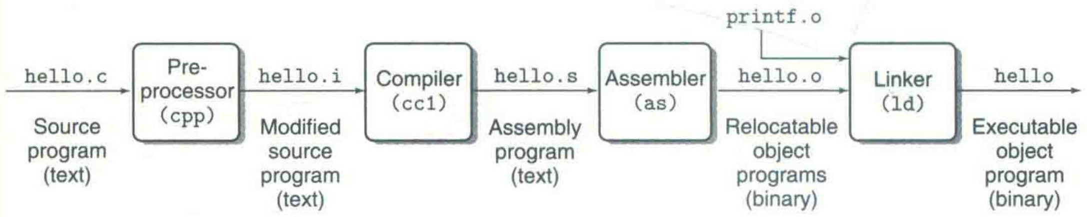


Figure 1.3 The compilation system.

```
linux> gcc -o hello hello.c
```

Here, the GCC compiler driver reads the source file `hello.c` and translates it into an executable object file `hello`. The translation is performed in the sequence of four phases shown in Figure 1.3. The programs that perform the four phases (*preprocessor*, *compiler*, *assembler*, and *linker*) are known collectively as the *compilation system*.

- *Preprocessing phase.* The preprocessor (`cpp`) modifies the original C program according to directives that begin with the '#' character. For example, the `#include <stdio.h>` command in line 1 of `hello.c` tells the preprocessor to read the contents of the system header file `stdio.h` and insert it directly into the program text. The result is another C program, typically with the `.i` suffix.
- *Compilation phase.* The compiler (`cc1`) translates the text file `hello.i` into the text file `hello.s`, which contains an *assembly-language program*. This program includes the following definition of function `main`:

```
1   main:
2     subq    $8, %rsp
3     movl    $.LC0, %edi
4     call    puts
5     movl    $0, %eax
6     addq    $8, %rsp
7     ret
```

  Each of lines 2–7 in this definition describes one low-level machine-language instruction in a textual form. Assembly language is useful because it provides a common output language for different compilers for different high-level languages. For example, C compilers and Fortran compilers both generate output files in the same assembly language.

- *Assembly phase.* Next, the assembler (`as`) translates `hello.s` into machine-language instructions, packages them in a form known as a *relocatable object program*, and stores the result in the object file `hello.o`. This file is a binary file containing 17 bytes to encode the instructions for function `main`. If we were to view `hello.o` with a text editor, it would appear to be gibberish.

Aside The GNU project

Gcc is one of many useful tools developed by the GNU (short for GNU's Not Unix) project. The GNU project is a tax-exempt charity started by Richard Stallman in 1984, with the ambitious goal of developing a complete Unix-like system whose source code is unencumbered by restrictions on how it can be modified or distributed. The GNU project has developed an environment with all the major components of a Unix operating system, except for the kernel, which was developed separately by the Linux project. The GNU environment includes the EMACS editor, GCC compiler, GDB debugger, assembler, linker, utilities for manipulating binaries, and other components. The GCC compiler has grown to support many different languages, with the ability to generate code for many different machines. Supported languages include C, C++, Fortran, Java, Pascal, Objective-C, and Ada.

The GNU project is a remarkable achievement, and yet it is often overlooked. The modern open-source movement (commonly associated with Linux) owes its intellectual origins to the GNU project's notion of *free software* ("free" as in "free speech," not "free beer"). Further, Linux owes much of its popularity to the GNU tools, which provide the environment for the Linux kernel.

- *Linking phase.* Notice that our `hello` program calls the `printf` function, which is part of the *standard C library* provided by every C compiler. The `printf` function resides in a separate precompiled object file called `printf.o`, which must somehow be merged with our `hello.o` program. The linker (`ld`) handles this merging. The result is the `hello` file, which is an executable object file (or simply *executable*) that is ready to be loaded into memory and executed by the system.

1.3 It Pays to Understand How Compilation Systems Work

For simple programs such as `hello.c`, we can rely on the compilation system to produce correct and efficient machine code. However, there are some important reasons why programmers need to understand how compilation systems work:

- *Optimizing program performance.* Modern compilers are sophisticated tools that usually produce good code. As programmers, we do not need to know the inner workings of the compiler in order to write efficient code. However, in order to make good coding decisions in our C programs, we do need a basic understanding of machine-level code and how the compiler translates different C statements into machine code. For example, is a `switch` statement always more efficient than a sequence of `if-else` statements? How much overhead is incurred by a function call? Is a `while` loop more efficient than a `for` loop? Are pointer references more efficient than array indexes? Why does our loop run so much faster if we sum into a local variable instead of an argument that is passed by reference? How can a function run faster when we simply rearrange the parentheses in an arithmetic expression?

In Chapter 3, we introduce x86-64, the machine language of recent generations of Linux, Macintosh, and Windows computers. We describe how compilers translate different C constructs into this language. In Chapter 5, you will learn how to tune the performance of your C programs by making simple transformations to the C code that help the compiler do its job better. In Chapter 6, you will learn about the hierarchical nature of the memory system, how C compilers store data arrays in memory, and how your C programs can exploit this knowledge to run more efficiently.

- *Understanding link-time errors.* In our experience, some of the most perplexing programming errors are related to the operation of the linker, especially when you are trying to build large software systems. For example, what does it mean when the linker reports that it cannot resolve a reference? What is the difference between a static variable and a global variable? What happens if you define two global variables in different C files with the same name? What is the difference between a static library and a dynamic library? Why does it matter what order we list libraries on the command line? And scariest of all, why do some linker-related errors not appear until run time? You will learn the answers to these kinds of questions in Chapter 7.
- *Avoiding security holes.* For many years, *buffer overflow vulnerabilities* have accounted for many of the security holes in network and Internet servers. These vulnerabilities exist because too few programmers understand the need to carefully restrict the quantity and forms of data they accept from untrusted sources. A first step in learning secure programming is to understand the consequences of the way data and control information are stored on the program stack. We cover the stack discipline and buffer overflow vulnerabilities in Chapter 3 as part of our study of assembly language. We will also learn about methods that can be used by the programmer, compiler, and operating system to reduce the threat of attack.

1.4 Processors Read and Interpret Instructions Stored in Memory

At this point, our `hello.c` source program has been translated by the compilation system into an executable object file called `hello` that is stored on disk. To run the executable file on a Unix system, we type its name to an application program known as a *shell:*

```
linux> ./hello
hello, world
linux>
```

The shell is a command-line interpreter that prints a prompt, waits for you to type a command line, and then performs the command. If the first word of the command line does not correspond to a built-in shell command, then the shell

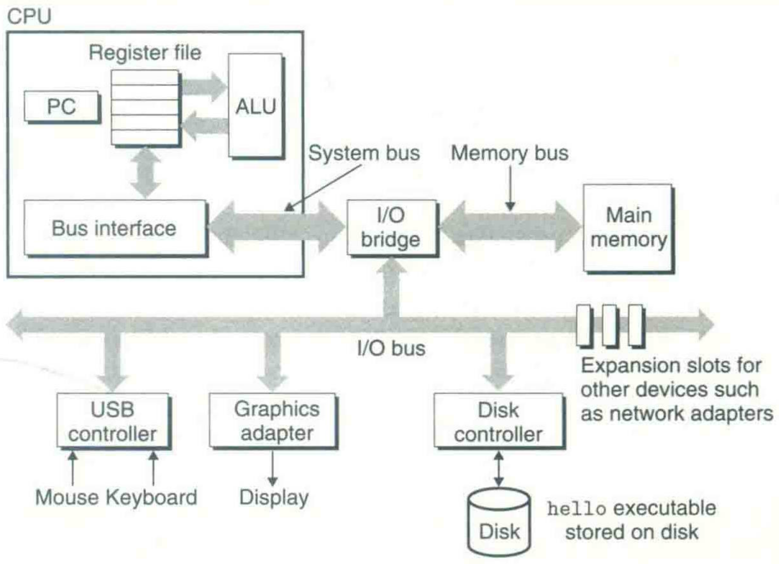


Figure 1.4
Hardware organization of a typical system. CPU: central processing unit, ALU: arithmetic/logic unit, PC: program counter, USB: Universal Serial Bus.

assumes that it is the name of an executable file that it should load and run. So in this case, the shell loads and runs the `hello` program and then waits for it to terminate. The `hello` program prints its message to the screen and then terminates. The shell then prints a prompt and waits for the next input command line.

1.4.1 Hardware Organization of a System

To understand what happens to our `hello` program when we run it, we need to understand the hardware organization of a typical system, which is shown in Figure 1.4. This particular picture is modeled after the family of recent Intel systems, but all systems have a similar look and feel. Don't worry about the complexity of this figure just now. We will get to its various details in stages throughout the course of the book.

Buses

Running throughout the system is a collection of electrical conduits called *buses* that carry bytes of information back and forth between the components. Buses are typically designed to transfer fixed-size chunks of bytes known as *words*. The number of bytes in a word (the *word size*) is a fundamental system parameter that varies across systems. Most machines today have word sizes of either 4 bytes (32 bits) or 8 bytes (64 bits). In this book, we do not assume any fixed definition of word size. Instead, we will specify what we mean by a "word" in any context that requires this to be defined.

I/O Devices

Input/output (I/O) devices are the system's connection to the external world. Our example system has four I/O devices: a keyboard and mouse for user input, a display for user output, and a disk drive (or simply disk) for long-term storage of data and programs. Initially, the executable `hello` program resides on the disk.

Each I/O device is connected to the I/O bus by either a *controller* or an *adapter*. The distinction between the two is mainly one of packaging. Controllers are chip sets in the device itself or on the system's main printed circuit board (often called the *motherboard*). An adapter is a card that plugs into a slot on the motherboard. Regardless, the purpose of each is to transfer information back and forth between the I/O bus and an I/O device.

Chapter 6 has more to say about how I/O devices such as disks work. In Chapter 10, you will learn how to use the Unix I/O interface to access devices from your application programs. We focus on the especially interesting class of devices known as networks, but the techniques generalize to other kinds of devices as well.

Main Memory

The *main memory* is a temporary storage device that holds both a program and the data it manipulates while the processor is executing the program. Physically, main memory consists of a collection of *dynamic random access memory* (DRAM) chips. Logically, memory is organized as a linear array of bytes, each with its own unique address (array index) starting at zero. In general, each of the machine instructions that constitute a program can consist of a variable number of bytes. The sizes of data items that correspond to C program variables vary according to type. For example, on an x86-64 machine running Linux, data of type `short` require 2 bytes, types `int` and `float` 4 bytes, and types `long` and `double` 8 bytes.

Chapter 6 has more to say about how memory technologies such as DRAM chips work, and how they are combined to form main memory.

Processor

The *central processing unit* (CPU), or simply *processor*, is the engine that interprets (or *executes*) instructions stored in main memory. At its core is a word-size storage device (or *register*) called the *program counter* (PC). At any point in time, the PC points at (contains the address of) some machine-language instruction in main memory.[2]

From the time that power is applied to the system until the time that the power is shut off, a processor repeatedly executes the instruction pointed at by the program counter and updates the program counter to point to the next instruction. A processor *appears* to operate according to a very simple instruction execution model, defined by its *instruction set architecture*. In this model, instructions execute

2. PC is also a commonly used acronym for "personal computer." However, the distinction between the two should be clear from the context.