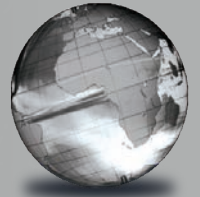


GLOBAL EDITION



# Computer Systems

## A Programmer's Perspective

THIRD EDITION

Randal E. Bryant • David R. O'Hallaron

# Computer Systems

A Programmer's Perspective

*This page is intentionally left blank.*

# 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](http://www.pearsonglobaleditions.com)

© Pearson Education Limited 2016

The rights of Randal E. Bryant and David R. O'Hallaron to be identified as the authors of this work have been asserted by them in accordance with the Copyright, Designs and Patents Act 1988.

*Authorized adaptation from the United States edition, entitled Computer Systems: A Programmer's Perspective, 3rd edition, ISBN 978-0-13-409266-9, by Randal E. Bryant and David R. O'Hallaron published by Pearson Education © 2016.*

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without either the prior written permission of the publisher or a license permitting restricted copying in the United Kingdom issued by the Copyright Licensing Agency Ltd, Saffron House, 6-10 Kirby Street, London EC1N 8TS.

All trademarks used herein are the property of their respective owners. The use of any trademark in this text does not vest in the author or publisher any trademark ownership rights in such trademarks, nor does the use of such trademarks imply any affiliation with or endorsement of this book by such owners.

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](http://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
<b>4.4</b>	<b>General Principles of Pipelining</b>	<b>448</b>
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
<b>4.5</b>	<b>Pipelined Y86-64 Implementations</b>	<b>457</b>
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
<b>4.6</b>	<b>Summary</b>	<b>506</b>
4.6.1	Y86-64 Simulators	508
	Bibliographic Notes	509
	Homework Problems	509
	Solutions to Practice Problems	516

## 5

---

### Optimizing Program Performance 531

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

---

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

---

code/intro/hello.c

**Figure 1** A typical code example.

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. Cov-

ering 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.