

GLOBAL
EDITION



Modern Operating Systems

FOURTH EDITION

Andrew S. Tanenbaum • Herbert Bos

ALWAYS LEARNING

PEARSON

ONLINE ACCESS

Thank you for purchasing a new copy of *Modern Operating Systems, Fourth Edition, Global Edition*. Your textbook includes twelve months of prepaid access to the book's Companion Website. This prepaid subscription provides you with full access to the following student support areas:

- Online Chapters
- Lab Experiments
- Online Exercises
- Simulation Exercises

Use a coin to scratch off the coating and reveal your student access code.
Do not use a knife or other sharp object as it may damage the code.

To access the *Modern Operating Systems, Fourth Edition, Global Edition*, Companion Website for the first time, you will need to register online using a computer with an Internet connection and a web browser. The process takes just a couple of minutes and only needs to be completed once.

1. Go to **www.pearsonglobaleditions.com/Tanenbaum**
2. Click on your book.
3. Click on **Companion Website**.
4. Click on the **Register** button.
5. On the registration page, enter your student access code* found beneath the scratch-off panel. Do not type the dashes. You can use lower- or uppercase.
6. Follow the on-screen instructions. If you need help at any time during the online registration process, simply click the **Need Help?** icon.
7. Once your personal Login Name and Password are confirmed, you can begin using the *Modern Operating Systems* Companion Website!

To log in after you have registered:

You only need to register for this Companion Website once. After that, you can log in any time at **www.pearsonglobaleditions.com/Tanenbaum** by providing your Login Name and Password when prompted.

*Important: The access code can only be used once. This subscription is valid for twelve months upon activation and is not transferable. If this access code has already been revealed, it may no longer be valid. If this is the case, you can purchase a subscription by going to **www.pearsonglobaleditions.com/Tanenbaum** and following the on-screen instructions.

MODERN OPERATING SYSTEMS

FOURTH EDITION

GLOBAL EDITION

Trademarks

AMD, the AMD logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc.

Android and Google Web Search are trademarks of Google Inc.

Apple and Apple Macintosh are registered trademarks of Apple Inc.

ASM, DESPOOL, DDT, LINK-80, MAC, MP/M, PL/1-80 and SID are trademarks of Digital Research.

BlackBerry®, RIM®, Research In Motion® and related trademarks, names and logos are the property of Research In Motion Limited and are registered and/or used in the U.S. and countries around the world.

Blu-ray Disc™ is a trademark owned by Blu-ray Disc Association.

CD Compact Disk is a trademark of Phillips.

CDC 6600 is a trademark of Control Data Corporation.

CP/M and CP/NET are registered trademarks of Digital Research.

DEC and PDP are registered trademarks of Digital Equipment Corporation.

eCosCentric is the owner of the eCos Trademark and eCos Logo, in the US and other countries. The marks were acquired from the Free Software Foundation on 26th February 2007. The Trademark and Logo were previously owned by Red Hat.

The GNOME logo and GNOME name are registered trademarks or trademarks of GNOME Foundation in the United States or other countries.

Firefox® and Firefox® OS are registered trademarks of the Mozilla Foundation.

Fortran is a trademark of IBM Corp.

FreeBSD is a registered trademark of the FreeBSD Foundation.

GE 645 is a trademark of General Electric Corporation.

Intel Core is a trademark of Intel Corporation in the U.S. and/or other countries.

Java is a trademark of Sun Microsystems, Inc., and refers to Sun's Java programming language.

Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

MS-DOS and Windows are registered trademarks of Microsoft Corporation in the United States and/or other countries.

TI Silent 700 is a trademark of Texas Instruments Incorporated.

UNIX is a registered trademark of The Open Group.

Zilog and Z80 are registered trademarks of Zilog, Inc.

MODERN OPERATING SYSTEMS

FOURTH EDITION

GLOBAL EDITION

ANDREW S. TANENBAUM

HERBERT BOS

Vrije Universiteit

Amsterdam, The Netherlands

PEARSON

Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montréal Toronto
Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Vice President and Editorial Director, ECS: *Marcia Horton*
Executive Editor: *Tracy Johnson*
Program Management Team Lead: *Scott Disanno*
Program Manager: *Carole Snyder*
Project Manager: *Camille Trentacoste*
Operations Specialist: *Linda Sager*
Head of Learning Asset Acquisition, Global Edition: *Laura Dent*
Assistant Acquisitions Editor, Global Edition: *Aditee Agarwal*
Project Editor, Global Edition: *Amrita Naskar*
Media Producer, Global Edition: *Vikram Kumar*
Senior Manufacturing Controller, Production, Global Edition: *Trudy Kimber*
Cover art: *Pavel K/Shutterstock*
Media Project Manager: *Renata Butera*

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

© Pearson Education Limited 2015

The rights of Andrew S. Tanenbaum and Herbert Bos 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 Modern Operating Systems, 4th edition, ISBN 978-0-13-359162-0, by Andrew S. Tanenbaum and Herbert Bos, published by Pearson Education © 2015.

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 EC 1N 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.

ISBN 10: 1-292-06142-1
ISBN 13: 978-1-292-06142-9

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
14 13 12 11 10

Printed and bound by Courier Westford in The United States of America

*To Suzanne, Barbara, Daniel, Aron, Nathan, Marvin, Matilde, and Olivia.
The list keeps growing. (AST)*

To Marieke, Duko, Jip, and Spot. Fearsome Jedi, all. (HB)

CONTENTS

PREFACE

xxiii

1 INTRODUCTION

1

- 1.1 WHAT IS AN OPERATING SYSTEM? 3
 - 1.1.1 The Operating System as an Extended Machine 4
 - 1.1.2 The Operating System as a Resource Manager 5
- 1.2 HISTORY OF OPERATING SYSTEMS 6
 - 1.2.1 The First Generation (1945–55): Vacuum Tubes 7
 - 1.2.2 The Second Generation (1955–65): Transistors and Batch Systems 8
 - 1.2.3 The Third Generation (1965–1980): ICs and Multiprogramming 9
 - 1.2.4 The Fourth Generation (1980–Present): Personal Computers 14
 - 1.2.5 The Fifth Generation (1990–Present): Mobile Computers 19
- 1.3 COMPUTER HARDWARE REVIEW 20
 - 1.3.1 Processors 21
 - 1.3.2 Memory 24
 - 1.3.3 Disks 27
 - 1.3.4 I/O Devices 28
 - 1.3.5 Buses 31
 - 1.3.6 Booting the Computer 34

- 1.4 THE OPERATING SYSTEM ZOO 35
 - 1.4.1 Mainframe Operating Systems 35
 - 1.4.2 Server Operating Systems 35
 - 1.4.3 Multiprocessor Operating Systems 36
 - 1.4.4 Personal Computer Operating Systems 36
 - 1.4.5 Handheld Computer Operating Systems 36
 - 1.4.6 Embedded Operating Systems 36
 - 1.4.7 Sensor-Node Operating Systems 37
 - 1.4.8 Real-Time Operating Systems 37
 - 1.4.9 Smart Card Operating Systems 38

- 1.5 OPERATING SYSTEM CONCEPTS 38
 - 1.5.1 Processes 39
 - 1.5.2 Address Spaces 41
 - 1.5.3 Files 41
 - 1.5.4 Input/Output 45
 - 1.5.5 Protection 45
 - 1.5.6 The Shell 45
 - 1.5.7 Ontogeny Recapitulates Phylogeny 46

- 1.6 SYSTEM CALLS 50
 - 1.6.1 System Calls for Process Management 53
 - 1.6.2 System Calls for File Management 56
 - 1.6.3 System Calls for Directory Management 57
 - 1.6.4 Miscellaneous System Calls 59
 - 1.6.5 The Windows Win32 API 60

- 1.7 OPERATING SYSTEM STRUCTURE 62
 - 1.7.1 Monolithic Systems 62
 - 1.7.2 Layered Systems 63
 - 1.7.3 Microkernels 65
 - 1.7.4 Client-Server Model 68
 - 1.7.5 Virtual Machines 68
 - 1.7.6 Exokernels 72

- 1.8 THE WORLD ACCORDING TO C 73
 - 1.8.1 The C Language 73
 - 1.8.2 Header Files 74
 - 1.8.3 Large Programming Projects 75
 - 1.8.4 The Model of Run Time 76

- 1.9 RESEARCH ON OPERATING SYSTEMS 77
- 1.10 OUTLINE OF THE REST OF THIS BOOK 78
- 1.11 METRIC UNITS 79
- 1.12 SUMMARY 80

2 PROCESSES AND THREADS

85

- 2.1 PROCESSES 85
 - 2.1.1 The Process Model 86
 - 2.1.2 Process Creation 88
 - 2.1.3 Process Termination 90
 - 2.1.4 Process Hierarchies 91
 - 2.1.5 Process States 92
 - 2.1.6 Implementation of Processes 94
 - 2.1.7 Modeling Multiprogramming 95
- 2.2 THREADS 97
 - 2.2.1 Thread Usage 97
 - 2.2.2 The Classical Thread Model 102
 - 2.2.3 POSIX Threads 106
 - 2.2.4 Implementing Threads in User Space 108
 - 2.2.5 Implementing Threads in the Kernel 111
 - 2.2.6 Hybrid Implementations 112
 - 2.2.7 Scheduler Activations 113
 - 2.2.8 Pop-Up Threads 114
 - 2.2.9 Making Single-Threaded Code Multithreaded 115
- 2.3 INTERPROCESS COMMUNICATION 119
 - 2.3.1 Race Conditions 119
 - 2.3.2 Critical Regions 121
 - 2.3.3 Mutual Exclusion with Busy Waiting 121
 - 2.3.4 Sleep and Wakeup 127
 - 2.3.5 Semaphores 130
 - 2.3.6 Mutexes 132

- 2.3.7 Monitors 137
- 2.3.8 Message Passing 144
- 2.3.9 Barriers 146
- 2.3.10 Avoiding Locks: Read-Copy-Update 148
- 2.4 SCHEDULING 148
 - 2.4.1 Introduction to Scheduling 149
 - 2.4.2 Scheduling in Batch Systems 156
 - 2.4.3 Scheduling in Interactive Systems 158
 - 2.4.4 Scheduling in Real-Time Systems 164
 - 2.4.5 Policy Versus Mechanism 165
 - 2.4.6 Thread Scheduling 165
- 2.5 CLASSICAL IPC PROBLEMS 167
 - 2.5.1 The Dining Philosophers Problem 167
 - 2.5.2 The Readers and Writers Problem 169
- 2.6 RESEARCH ON PROCESSES AND THREADS 172
- 2.7 SUMMARY 173

3 MEMORY MANAGEMENT

181

- 3.1 NO MEMORY ABSTRACTION 182
- 3.2 A MEMORY ABSTRACTION: ADDRESS SPACES 185
 - 3.2.1 The Notion of an Address Space 185
 - 3.2.2 Swapping 187
 - 3.2.3 Managing Free Memory 190
- 3.3 VIRTUAL MEMORY 194
 - 3.3.1 Paging 195
 - 3.3.2 Page Tables 198
 - 3.3.3 Speeding Up Paging 201
 - 3.3.4 Page Tables for Large Memories 205

- 3.4 PAGE REPLACEMENT ALGORITHMS 209
 - 3.4.1 The Optimal Page Replacement Algorithm 209
 - 3.4.2 The Not Recently Used Page Replacement Algorithm 210
 - 3.4.3 The First-In, First-Out (FIFO) Page Replacement Algorithm 211
 - 3.4.4 The Second-Chance Page Replacement Algorithm 211
 - 3.4.5 The Clock Page Replacement Algorithm 212
 - 3.4.6 The Least Recently Used (LRU) Page Replacement Algorithm 213
 - 3.4.7 Simulating LRU in Software 214
 - 3.4.8 The Working Set Page Replacement Algorithm 215
 - 3.4.9 The WSClock Page Replacement Algorithm 219
 - 3.4.10 Summary of Page Replacement Algorithms 221

- 3.5 DESIGN ISSUES FOR PAGING SYSTEMS 222
 - 3.5.1 Local versus Global Allocation Policies 222
 - 3.5.2 Load Control 225
 - 3.5.3 Page Size 225
 - 3.5.4 Separate Instruction and Data Spaces 227
 - 3.5.5 Shared Pages 228
 - 3.5.6 Shared Libraries 229
 - 3.5.7 Mapped Files 231
 - 3.5.8 Cleaning Policy 232
 - 3.5.9 Virtual Memory Interface 232

- 3.6 IMPLEMENTATION ISSUES 233
 - 3.6.1 Operating System Involvement with Paging 233
 - 3.6.2 Page Fault Handling 234
 - 3.6.3 Instruction Backup 235
 - 3.6.4 Locking Pages in Memory 236
 - 3.6.5 Backing Store 237
 - 3.6.6 Separation of Policy and Mechanism 239

- 3.7 SEGMENTATION 240
 - 3.7.1 Implementation of Pure Segmentation 243
 - 3.7.2 Segmentation with Paging: MULTICS 243
 - 3.7.3 Segmentation with Paging: The Intel x86 247

- 3.8 RESEARCH ON MEMORY MANAGEMENT 252

- 3.9 SUMMARY 253

4 FILE SYSTEMS**263**

- 4.1 FILES 265
 - 4.1.1 File Naming 265
 - 4.1.2 File Structure 267
 - 4.1.3 File Types 268
 - 4.1.4 File Access 269
 - 4.1.5 File Attributes 271
 - 4.1.6 File Operations 271
 - 4.1.7 An Example Program Using File-System Calls 273
- 4.2 DIRECTORIES 276
 - 4.2.1 Single-Level Directory Systems 276
 - 4.2.2 Hierarchical Directory Systems 276
 - 4.2.3 Path Names 277
 - 4.2.4 Directory Operations 280
- 4.3 FILE-SYSTEM IMPLEMENTATION 281
 - 4.3.1 File-System Layout 281
 - 4.3.2 Implementing Files 282
 - 4.3.3 Implementing Directories 287
 - 4.3.4 Shared Files 290
 - 4.3.5 Log-Structured File Systems 293
 - 4.3.6 Journaling File Systems 294
 - 4.3.7 Virtual File Systems 296
- 4.4 FILE-SYSTEM MANAGEMENT AND OPTIMIZATION 299
 - 4.4.1 Disk-Space Management 299
 - 4.4.2 File-System Backups 306
 - 4.4.3 File-System Consistency 312
 - 4.4.4 File-System Performance 314
 - 4.4.5 Defragmenting Disks 319
- 4.5 EXAMPLE FILE SYSTEMS 320
 - 4.5.1 The MS-DOS File System 320
 - 4.5.2 The UNIX V7 File System 323
 - 4.5.3 CD-ROM File Systems 325
- 4.6 RESEARCH ON FILE SYSTEMS 331
- 4.7 SUMMARY 332

5 INPUT/OUTPUT 337

- 5.1 PRINCIPLES OF I/O HARDWARE 337
 - 5.1.1 I/O Devices 338
 - 5.1.2 Device Controllers 339
 - 5.1.3 Memory-Mapped I/O 340
 - 5.1.4 Direct Memory Access 344
 - 5.1.5 Interrupts Revisited 347

- 5.2 PRINCIPLES OF I/O SOFTWARE 351
 - 5.2.1 Goals of the I/O Software 351
 - 5.2.2 Programmed I/O 352
 - 5.2.3 Interrupt-Driven I/O 354
 - 5.2.4 I/O Using DMA 355

- 5.3 I/O SOFTWARE LAYERS 356
 - 5.3.1 Interrupt Handlers 356
 - 5.3.2 Device Drivers 357
 - 5.3.3 Device-Independent I/O Software 361
 - 5.3.4 User-Space I/O Software 367

- 5.4 DISKS 369
 - 5.4.1 Disk Hardware 369
 - 5.4.2 Disk Formatting 375
 - 5.4.3 Disk Arm Scheduling Algorithms 379
 - 5.4.4 Error Handling 382
 - 5.4.5 Stable Storage 385

- 5.5 CLOCKS 388
 - 5.5.1 Clock Hardware 388
 - 5.5.2 Clock Software 389
 - 5.5.3 Soft Timers 392

- 5.6 USER INTERFACES: KEYBOARD, MOUSE, MONITOR 394
 - 5.6.1 Input Software 394
 - 5.6.2 Output Software 399

- 5.7 THIN CLIENTS 416

- 5.8 POWER MANAGEMENT 417
 - 5.8.1 Hardware Issues 418

- 5.8.2 Operating System Issues 419
- 5.8.3 Application Program Issues 425

- 5.9 RESEARCH ON INPUT/OUTPUT 426
- 5.10 SUMMARY 428

6 DEADLOCKS 435

- 6.1 RESOURCES 436
 - 6.1.1 Preemptable and Nonpreemptable Resources 436
 - 6.1.2 Resource Acquisition 437
- 6.2 INTRODUCTION TO DEADLOCKS 438
 - 6.2.1 Conditions for Resource Deadlocks 439
 - 6.2.2 Deadlock Modeling 440
- 6.3 THE OSTRICH ALGORITHM 443
- 6.4 DEADLOCK DETECTION AND RECOVERY 443
 - 6.4.1 Deadlock Detection with One Resource of Each Type 444
 - 6.4.2 Deadlock Detection with Multiple Resources of Each Type 446
 - 6.4.3 Recovery from Deadlock 448
- 6.5 DEADLOCK AVOIDANCE 450
 - 6.5.1 Resource Trajectories 450
 - 6.5.2 Safe and Unsafe States 452
 - 6.5.3 The Banker's Algorithm for a Single Resource 453
 - 6.5.4 The Banker's Algorithm for Multiple Resources 454
- 6.6 DEADLOCK PREVENTION 456
 - 6.6.1 Attacking the Mutual-Exclusion Condition 456
 - 6.6.2 Attacking the Hold-and-Wait Condition 456
 - 6.6.3 Attacking the No-Preemption Condition 457
 - 6.6.4 Attacking the Circular Wait Condition 457
- 6.7 OTHER ISSUES 458
 - 6.7.1 Two-Phase Locking 458
 - 6.7.2 Communication Deadlocks 459

6.7.3 Livelock	461
6.7.4 Starvation	463
6.8 RESEARCH ON DEADLOCKS	464
6.9 SUMMARY	464
7 VIRTUALIZATION AND THE CLOUD	471
7.1 HISTORY	473
7.2 REQUIREMENTS FOR VIRTUALIZATION	474
7.3 TYPE 1 AND TYPE 2 HYPERVISORS	477
7.4 TECHNIQUES FOR EFFICIENT VIRTUALIZATION	478
7.4.1 Virtualizing the Unvirtualizable	479
7.4.2 The Cost of Virtualization	482
7.5 ARE HYPERVISORS MICROKERNELS DONE RIGHT?	483
7.6 MEMORY VIRTUALIZATION	486
7.7 I/O VIRTUALIZATION	490
7.8 VIRTUAL APPLIANCES	493
7.9 VIRTUAL MACHINES ON MULTICORE CPUS	494
7.10 LICENSING ISSUES	494
7.11 CLOUDS	495
7.11.1 Clouds as a Service	496
7.11.2 Virtual Machine Migration	496
7.11.3 Checkpointing	497
7.12 CASE STUDY: VMWARE	498
7.12.1 The Early History of VMware	498
7.12.2 VMware Workstation	499

- 7.12.3 Challenges in Bringing Virtualization to the x86 500
 - 7.12.4 VMware Workstation: Solution Overview 502
 - 7.12.5 The Evolution of VMware Workstation 511
 - 7.12.6 ESX Server: VMware's type 1 Hypervisor 512
- 7.13 RESEARCH ON VIRTUALIZATION AND THE CLOUD 514

8 MULTIPLE PROCESSOR SYSTEMS 517

- 8.1 MULTIPROCESSORS 520
 - 8.1.1 Multiprocessor Hardware 520
 - 8.1.2 Multiprocessor Operating System Types 530
 - 8.1.3 Multiprocessor Synchronization 534
 - 8.1.4 Multiprocessor Scheduling 539
- 8.2 MULTICOMPUTERS 544
 - 8.2.1 Multicomputer Hardware 545
 - 8.2.2 Low-Level Communication Software 550
 - 8.2.3 User-Level Communication Software 552
 - 8.2.4 Remote Procedure Call 556
 - 8.2.5 Distributed Shared Memory 558
 - 8.2.6 Multicomputer Scheduling 563
 - 8.2.7 Load Balancing 563
- 8.3 DISTRIBUTED SYSTEMS 566
 - 8.3.1 Network Hardware 568
 - 8.3.2 Network Services and Protocols 571
 - 8.3.3 Document-Based Middleware 576
 - 8.3.4 File-System-Based Middleware 577
 - 8.3.5 Object-Based Middleware 582
 - 8.3.6 Coordination-Based Middleware 584
- 8.4 RESEARCH ON MULTIPLE PROCESSOR SYSTEMS 587
- 8.5 SUMMARY 588

9	SECURITY	593
9.1	THE SECURITY ENVIRONMENT	595
9.1.1	Threats	596
9.1.2	Attackers	598
9.2	OPERATING SYSTEMS SECURITY	599
9.2.1	Can We Build Secure Systems?	600
9.2.2	Trusted Computing Base	601
9.3	CONTROLLING ACCESS TO RESOURCES	602
9.3.1	Protection Domains	602
9.3.2	Access Control Lists	605
9.3.3	Capabilities	608
9.4	FORMAL MODELS OF SECURE SYSTEMS	611
9.4.1	Multilevel Security	612
9.4.2	Covert Channels	615
9.5	BASICS OF CRYPTOGRAPHY	619
9.5.1	Secret-Key Cryptography	620
9.5.2	Public-Key Cryptography	621
9.5.3	One-Way Functions	622
9.5.4	Digital Signatures	622
9.5.5	Trusted Platform Modules	624
9.6	AUTHENTICATION	626
9.6.1	Authentication Using a Physical Object	633
9.6.2	Authentication Using Biometrics	636
9.7	EXPLOITING SOFTWARE	639
9.7.1	Buffer Overflow Attacks	640
9.7.2	Format String Attacks	649
9.7.3	Dangling Pointers	652
9.7.4	Null Pointer Dereference Attacks	653
9.7.5	Integer Overflow Attacks	654
9.7.6	Command Injection Attacks	655
9.7.7	Time of Check to Time of Use Attacks	656
9.8	INSIDER ATTACKS	657
9.8.1	Logic Bombs	657
9.8.2	Back Doors	658
9.8.3	Login Spoofing	659

- 9.9 MALWARE 660
 - 9.9.1 Trojan Horses 662
 - 9.9.2 Viruses 664
 - 9.9.3 Worms 674
 - 9.9.4 Spyware 676
 - 9.9.5 Rootkits 680
- 9.10 DEFENSES 684
 - 9.10.1 Firewalls 685
 - 9.10.2 Antivirus and Anti-Antivirus Techniques 687
 - 9.10.3 Code Signing 693
 - 9.10.4 Jailing 694
 - 9.10.5 Model-Based Intrusion Detection 695
 - 9.10.6 Encapsulating Mobile Code 697
 - 9.10.7 Java Security 701
- 9.11 RESEARCH ON SECURITY 703
- 9.12 SUMMARY 704

10 CASE STUDY 1: UNIX, LINUX, AND ANDROID 713

- 10.1 HISTORY OF UNIX AND LINUX 714
 - 10.1.1 UNICS 714
 - 10.1.2 PDP-11 UNIX 715
 - 10.1.3 Portable UNIX 716
 - 10.1.4 Berkeley UNIX 717
 - 10.1.5 Standard UNIX 718
 - 10.1.6 MINIX 719
 - 10.1.7 Linux 720
- 10.2 OVERVIEW OF LINUX 723
 - 10.2.1 Linux Goals 723
 - 10.2.2 Interfaces to Linux 724
 - 10.2.3 The Shell 725
 - 10.2.4 Linux Utility Programs 728
 - 10.2.5 Kernel Structure 730
- 10.3 PROCESSES IN LINUX 733
 - 10.3.1 Fundamental Concepts 733
 - 10.3.2 Process-Management System Calls in Linux 735

- 10.3.3 Implementation of Processes and Threads in Linux 739
- 10.3.4 Scheduling in Linux 746
- 10.3.5 Booting Linux 751
- 10.4 MEMORY MANAGEMENT IN LINUX 753
 - 10.4.1 Fundamental Concepts 753
 - 10.4.2 Memory Management System Calls in Linux 756
 - 10.4.3 Implementation of Memory Management in Linux 758
 - 10.4.4 Paging in Linux 764
- 10.5 INPUT/OUTPUT IN LINUX 767
 - 10.5.1 Fundamental Concepts 767
 - 10.5.2 Networking 769
 - 10.5.3 Input/Output System Calls in Linux 770
 - 10.5.4 Implementation of Input/Output in Linux 771
 - 10.5.5 Modules in Linux 774
- 10.6 THE LINUX FILE SYSTEM 775
 - 10.6.1 Fundamental Concepts 775
 - 10.6.2 File-System Calls in Linux 780
 - 10.6.3 Implementation of the Linux File System 783
 - 10.6.4 NFS: The Network File System 792
- 10.7 SECURITY IN LINUX 798
 - 10.7.1 Fundamental Concepts 798
 - 10.7.2 Security System Calls in Linux 800
 - 10.7.3 Implementation of Security in Linux 801
- 10.8 ANDROID 802
 - 10.8.1 Android and Google 803
 - 10.8.2 History of Android 803
 - 10.8.3 Design Goals 807
 - 10.8.4 Android Architecture 809
 - 10.8.5 Linux Extensions 810
 - 10.8.6 Dalvik 814
 - 10.8.7 Binder IPC 815
 - 10.8.8 Android Applications 824
 - 10.8.9 Intents 836
 - 10.8.10 Application Sandboxes 837
 - 10.8.11 Security 838
 - 10.8.12 Process Model 844
- 10.9 SUMMARY 848

11 CASE STUDY 2: WINDOWS 8**857**

- 11.1 HISTORY OF WINDOWS THROUGH WINDOWS 8.1 857
 - 11.1.1 1980s: MS-DOS 857
 - 11.1.2 1990s: MS-DOS-based Windows 859
 - 11.1.3 2000s: NT-based Windows 859
 - 11.1.4 Windows Vista 862
 - 11.1.5 2010s: Modern Windows 863
- 11.2 PROGRAMMING WINDOWS 864
 - 11.2.1 The Native NT Application Programming Interface 867
 - 11.2.2 The Win32 Application Programming Interface 871
 - 11.2.3 The Windows Registry 875
- 11.3 SYSTEM STRUCTURE 877
 - 11.3.1 Operating System Structure 877
 - 11.3.2 Booting Windows 893
 - 11.3.3 Implementation of the Object Manager 894
 - 11.3.4 Subsystems, DLLs, and User-Mode Services 905
- 11.4 PROCESSES AND THREADS IN WINDOWS 908
 - 11.4.1 Fundamental Concepts 908
 - 11.4.2 Job, Process, Thread, and Fiber Management API Calls 914
 - 11.4.3 Implementation of Processes and Threads 919
- 11.5 MEMORY MANAGEMENT 927
 - 11.5.1 Fundamental Concepts 927
 - 11.5.2 Memory-Management System Calls 931
 - 11.5.3 Implementation of Memory Management 932
- 11.6 CACHING IN WINDOWS 942
- 11.7 INPUT/OUTPUT IN WINDOWS 943
 - 11.7.1 Fundamental Concepts 944
 - 11.7.2 Input/Output API Calls 945
 - 11.7.3 Implementation of I/O 948
- 11.8 THE WINDOWS NT FILE SYSTEM 952
 - 11.8.1 Fundamental Concepts 953
 - 11.8.2 Implementation of the NT File System 954
- 11.9 WINDOWS POWER MANAGEMENT 964

- 11.10 SECURITY IN WINDOWS 8 966
 - 11.10.1 Fundamental Concepts 967
 - 11.10.2 Security API Calls 969
 - 11.10.3 Implementation of Security 970
 - 11.10.4 Security Mitigations 972
- 11.11 SUMMARY 975

12 OPERATING SYSTEM DESIGN

981

- 12.1 THE NATURE OF THE DESIGN PROBLEM 982
 - 12.1.1 Goals 982
 - 12.1.2 Why Is It Hard to Design an Operating System? 983
- 12.2 INTERFACE DESIGN 985
 - 12.2.1 Guiding Principles 985
 - 12.2.2 Paradigms 987
 - 12.2.3 The System-Call Interface 991
- 12.3 IMPLEMENTATION 993
 - 12.3.1 System Structure 993
 - 12.3.2 Mechanism vs. Policy 997
 - 12.3.3 Orthogonality 998
 - 12.3.4 Naming 999
 - 12.3.5 Binding Time 1001
 - 12.3.6 Static vs. Dynamic Structures 1001
 - 12.3.7 Top-Down vs. Bottom-Up Implementation 1003
 - 12.3.8 Synchronous vs. Asynchronous Communication 1004
 - 12.3.9 Useful Techniques 1005
- 12.4 PERFORMANCE 1010
 - 12.4.1 Why Are Operating Systems Slow? 1010
 - 12.4.2 What Should Be Optimized? 1011
 - 12.4.3 Space-Time Trade-offs 1012
 - 12.4.4 Caching 1015
 - 12.4.5 Hints 1016
 - 12.4.6 Exploiting Locality 1016
 - 12.4.7 Optimize the Common Case 1017

- 12.5 PROJECT MANAGEMENT 1018
 - 12.5.1 The Mythical Man Month 1018
 - 12.5.2 Team Structure 1019
 - 12.5.3 The Role of Experience 1021
 - 12.5.4 No Silver Bullet 1021

- 12.6 TRENDS IN OPERATING SYSTEM DESIGN 1022
 - 12.6.1 Virtualization and the Cloud 1023
 - 12.6.2 Manycore Chips 1023
 - 12.6.3 Large-Address-Space Operating Systems 1024
 - 12.6.4 Seamless Data Access 1025
 - 12.6.5 Battery-Powered Computers 1025
 - 12.6.6 Embedded Systems 1026

- 12.7 SUMMARY 1027

13 READING LIST AND BIBLIOGRAPHY 1031

- 13.1 SUGGESTIONS FOR FURTHER READING 1031
 - 13.1.1 Introduction 1031
 - 13.1.2 Processes and Threads 1032
 - 13.1.3 Memory Management 1033
 - 13.1.4 File Systems 1033
 - 13.1.5 Input/Output 1034
 - 13.1.6 Deadlocks 1035
 - 13.1.7 Virtualization and the Cloud 1035
 - 13.1.8 Multiple Processor Systems 1036
 - 13.1.9 Security 1037
 - 13.1.10 Case Study 1: UNIX, Linux, and Android 1039
 - 13.1.11 Case Study 2: Windows 8 1040
 - 13.1.12 Operating System Design 1040

- 13.2 ALPHABETICAL BIBLIOGRAPHY 1041

INDEX 1071

PREFACE

The fourth edition of this book differs from the third edition in numerous ways. There are large numbers of small changes everywhere to bring the material up to date as operating systems are not standing still. The chapter on Multimedia Operating Systems has been moved to the Web, primarily to make room for new material and keep the book from growing to a completely unmanageable size. The chapter on Windows Vista has been removed completely as Vista has not been the success Microsoft hoped for. The chapter on Symbian has also been removed, as Symbian no longer is widely available. However, the Vista material has been replaced by Windows 8 and Symbian has been replaced by Android. Also, a completely new chapter, on virtualization and the cloud has been added. Here is a chapter-by-chapter rundown of the changes.

- Chapter 1 has been heavily modified and updated in many places but with the exception of a new section on mobile computers, no major sections have been added or deleted.
- Chapter 2 has been updated, with older material removed and some new material added. For example, we added the futex synchronization primitive, and a section about how to avoid locking altogether with Read-Copy-Update.
- Chapter 3 now has more focus on modern hardware and less emphasis on segmentation and MULTICS.
- In Chapter 4 we removed CD-Roms, as they are no longer very common, and replaced them with more modern solutions (like flash drives). Also, we added RAID level 6 to the section on RAID.

- Chapter 5 has seen a lot of changes. Older devices like CRTs and CD-ROMs have been removed, while new technology, such as touch screens have been added.
- Chapter 6 is pretty much unchanged. The topic of deadlocks is fairly stable, with few new results.
- Chapter 7 is completely new. It covers the important topics of virtualization and the cloud. As a case study, a section on VMware has been added.
- Chapter 8 is an updated version of the previous material on multiprocessor systems. There is more emphasis on multicore and manycore systems now, which have become increasingly important in the past few years. Cache consistency has become a bigger issue recently and is covered here, now.
- Chapter 9 has been heavily revised and reorganized, with considerable new material on exploiting code bugs, malware, and defenses against them. Attacks such as null pointer dereferences and buffer overflows are treated in more detail. Defense mechanisms, including canaries, the NX bit, and address-space randomization are covered in detail now, as are the ways attackers try to defeat them.
- Chapter 10 has undergone a major change. The material on UNIX and Linux has been updated but the major addition here is a new and lengthy section on the Android operating system, which is very common on smartphones and tablets.
- Chapter 11 in the third edition was on Windows Vista. That has been replaced by a chapter on Windows 8, specifically Windows 8.1. It brings the treatment of Windows completely up to date.
- Chapter 12 is a revised version of Chap. 13 from the previous edition.
- Chapter 13 is a thoroughly updated list of suggested readings. In addition, the list of references has been updated, with entries to 223 new works published after the third edition of this book came out.
- Chapter 7 from the previous edition has been moved to the book's Website to keep the size somewhat manageable).
- In addition, the sections on research throughout the book have all been redone from scratch to reflect the latest research in operating systems. Furthermore, new problems have been added to all the chapters.

Numerous teaching aids for this book are available. Instructor supplements can be found at www.pearsonglobaleditions.com/Tanenbaum They include

PowerPoint sheets, software tools for studying operating systems, lab experiments for students, simulators, and more material for use in operating systems courses. Instructors using this book in a course should definitely take a look. The Companion Website for this book is also located at www.pearsonglobaleditions.com/Tanenbaum. The specific site for this book is password protected. To use the site, follow the instructions on the student access card that came with your text to create a user account and log in. Student resources include:

- An online chapter on Multimedia Operating Systems
- Lab Experiments
- Online Exercises
- Simulation Exercises

A number of people have been involved in the fourth edition. First and foremost, Prof. Herbert Bos of the Vrije Universiteit in Amsterdam has been added as a coauthor. He is a security, UNIX, and all-around systems expert and it is great to have him on board. He wrote much of the new material except as noted below.

Our editor, Tracy Johnson, has done a wonderful job, as usual, of herding all the cats, putting all the pieces together, putting out fires, and keeping the project on schedule. We were also fortunate to get our long-time production editor, Camille Trentacoste, back. Her skills in so many areas have saved the day on more than a few occasions. We are glad to have her again after an absence of several years. Carole Snyder did a fine job coordinating the various people involved in the book.

The material in Chap. 7 on VMware (in Sec. 7.12) was written by Edouard Bugnion of EPFL in Lausanne, Switzerland. Ed was one of the founders of the VMware company and knows this material as well as anyone in the world. We thank him greatly for supplying it to us.

Ada Gavrilovska of Georgia Tech, who is an expert on Linux internals, updated Chap. 10 from the Third Edition, which she also wrote. The Android material in Chap. 10 was written by Dianne Hackborn of Google, one of the key developers of the Android system. Android is the leading operating system on smartphones, so we are very grateful to have Dianne help us. Chap. 10 is now quite long and detailed, but UNIX, Linux, and Android fans can learn a lot from it. It is perhaps worth noting that the longest and most technical chapter in the book was written by two women. We just did the easy stuff.

We haven't neglected Windows, however. Dave Probert of Microsoft updated Chap. 11 from the previous edition of the book. This time the chapter covers Windows 8.1 in detail. Dave has a great deal of knowledge of Windows and enough vision to tell the difference between places where Microsoft got it right and where it got it wrong. Windows fans are certain to enjoy this chapter.

The book is much better as a result of the work of all these expert contributors. Again, we would like to thank them for their invaluable help.

We were also fortunate to have several reviewers who read the manuscript and also suggested new end-of-chapter problems. These were Trudy Levine, Shivakant Mishra, Krishna Sivalingam, and Ken Wong. Steve Armstrong did the PowerPoint sheets for instructors teaching a course using the book.

Normally copyeditors and proofreaders don't get acknowledgements, but Bob Lentz (copyeditor) and Joe Ruddick (proofreader) did exceptionally thorough jobs. Joe in particular, can spot the difference between a roman period and an italics period from 20 meters. Nevertheless, the authors take full responsibility for any residual errors in the book. Readers noticing any errors are requested to contact one of the authors.

Finally, last but not least, Barbara and Marvin are still wonderful, as usual, each in a unique and special way. Daniel and Matilde are great additions to our family. Aron and Nathan are wonderful little guys and Olivia is a treasure. And of course, I would like to thank Suzanne for her love and patience, not to mention all the *druiven*, *kersen*, and *sinaasappelsap*, as well as other agricultural products. (AST)

Most importantly, I would like to thank Marieke, Duko, and Jip. Marieke for her love and for bearing with me all the nights I was working on this book, and Duko and Jip for tearing me away from it and showing me there are more important things in life. Like Minecraft. (HB)

Andrew S. Tanenbaum
Herbert Bos

ABOUT THE AUTHORS

Andrew S. Tanenbaum has an S.B. degree from M.I.T. and a Ph.D. from the University of California at Berkeley. He is currently a Professor of Computer Science at the Vrije Universiteit in Amsterdam, The Netherlands. He was formerly Dean of the Advanced School for Computing and Imaging, an interuniversity graduate school doing research on advanced parallel, distributed, and imaging systems. He was also an Academy Professor of the Royal Netherlands Academy of Arts and Sciences, which has saved him from turning into a bureaucrat. He also won a prestigious European Research Council Advanced Grant.

In the past, he has done research on compilers, operating systems, networking, and distributed systems. His main research focus now is reliable and secure operating systems. These research projects have led to over 175 refereed papers in journals and conferences. Prof. Tanenbaum has also authored or co-authored five books, which have been translated into 20 languages, ranging from Basque to Thai. They are used at universities all over the world. In all, there are 163 versions (language + edition combinations) of his books.

Prof. Tanenbaum has also produced a considerable volume of software, notably MINIX, a small UNIX clone. It was the direct inspiration for Linux and the platform on which Linux was initially developed. The current version of MINIX, called MINIX 3, is now focused on being an extremely reliable and secure operating system. Prof. Tanenbaum will consider his work done when no user has any idea what an operating system crash is. MINIX 3 is an ongoing open-source project to which you are invited to contribute. Go to www.minix3.org to download a free copy of MINIX 3 and give it a try. Both x86 and ARM versions are available.

Prof. Tanenbaum's Ph.D. students have gone on to greater glory after graduating. He is very proud of them. In this respect, he resembles a mother hen.

Prof. Tanenbaum is a Fellow of the ACM, a Fellow of the IEEE, and a member of the Royal Netherlands Academy of Arts and Sciences. He has also won numerous scientific prizes from ACM, IEEE, and USENIX. If you are unbearably curious about them, see his page on Wikipedia. He also has two honorary doctorates.

Herbert Bos obtained his Masters degree from Twente University and his Ph.D. from Cambridge University Computer Laboratory in the U.K.. Since then, he has worked extensively on dependable and efficient I/O architectures for operating systems like Linux, but also research systems based on MINIX 3. He is currently a professor in Systems and Network Security in the Dept. of Computer Science at the Vrije Universiteit in Amsterdam, The Netherlands. His main research field is system security. With his students, he works on novel ways to detect and stop attacks, to analyze and reverse engineer malware, and to take down botnets (malicious infrastructures that may span millions of computers). In 2011, he obtained an ERC Starting Grant for his research on reverse engineering. Three of his students have won the Roger Needham Award for best European Ph.D. thesis in systems.

MODERN OPERATING SYSTEMS

1

INTRODUCTION

A modern computer consists of one or more processors, some main memory, disks, printers, a keyboard, a mouse, a display, network interfaces, and various other input/output devices. All in all, a complex system.oo If every application programmer had to understand how all these things work in detail, no code would ever get written. Furthermore, managing all these components and using them optimally is an exceedingly challenging job. For this reason, computers are equipped with a layer of software called the **operating system**, whose job is to provide user programs with a better, simpler, cleaner, model of the computer and to handle managing all the resources just mentioned. Operating systems are the subject of this book.

Most readers will have had some experience with an operating system such as Windows, Linux, FreeBSD, or OS X, but appearances can be deceiving. The program that users interact with, usually called the **shell** when it is text based and the **GUI (Graphical User Interface)**—which is pronounced “gooey”—when it uses icons, is actually not part of the operating system, although it uses the operating system to get its work done.

A simple overview of the main components under discussion here is given in Fig. 1-1. Here we see the hardware at the bottom. The hardware consists of chips, boards, disks, a keyboard, a monitor, and similar physical objects. On top of the hardware is the software. Most computers have two modes of operation: kernel mode and user mode. The operating system, the most fundamental piece of software, runs in **kernel mode** (also called **supervisor mode**). In this mode it has

complete access to all the hardware and can execute any instruction the machine is capable of executing. The rest of the software runs in **user mode**, in which only a subset of the machine instructions is available. In particular, those instructions that affect control of the machine or do **I/O** (Input/Output) are forbidden to user-mode programs. We will come back to the difference between kernel mode and user mode repeatedly throughout this book. It plays a crucial role in how operating systems work.

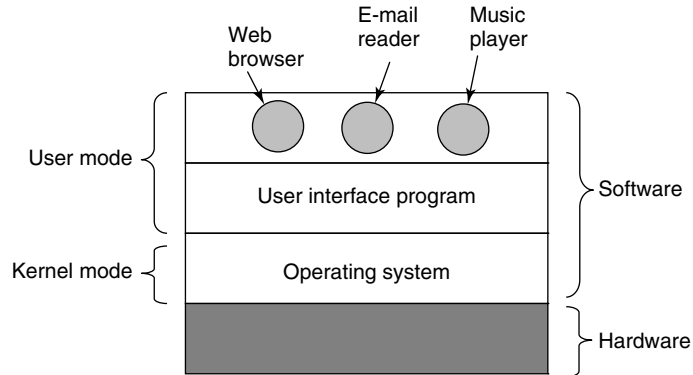


Figure 1-1. Where the operating system fits in.

The user interface program, shell or GUI, is the lowest level of user-mode software, and allows the user to start other programs, such as a Web browser, email reader, or music player. These programs, too, make heavy use of the operating system.

The placement of the operating system is shown in Fig. 1-1. It runs on the bare hardware and provides the base for all the other software.

An important distinction between the operating system and normal (user-mode) software is that if a user does not like a particular email reader, he† is free to get a different one or write his own if he so chooses; he is not free to write his own clock interrupt handler, which is part of the operating system and is protected by hardware against attempts by users to modify it.

This distinction, however, is sometimes blurred in embedded systems (which may not have kernel mode) or interpreted systems (such as Java-based systems that use interpretation, not hardware, to separate the components).

Also, in many systems there are programs that run in user mode but help the operating system or perform privileged functions. For example, there is often a program that allows users to change their passwords. It is not part of the operating system and does not run in kernel mode, but it clearly carries out a sensitive function and has to be protected in a special way. In some systems, this idea is carried to an extreme, and pieces of what is traditionally considered to be the operating

† “He” should be read as “he or she” throughout the book.

system (such as the file system) run in user space. In such systems, it is difficult to draw a clear boundary. Everything running in kernel mode is clearly part of the operating system, but some programs running outside it are arguably also part of it, or at least closely associated with it.

Operating systems differ from user (i.e., application) programs in ways other than where they reside. In particular, they are huge, complex, and long-lived. The source code of the heart of an operating system like Linux or Windows is on the order of five million lines of code or more. To conceive of what this means, think of printing out five million lines in book form, with 50 lines per page and 1000 pages per volume (larger than this book). It would take 100 volumes to list an operating system of this size—essentially an entire bookcase. Can you imagine getting a job maintaining an operating system and on the first day having your boss bring you to a bookcase with the code and say: “Go learn that.” And this is only for the part that runs in the kernel. When essential shared libraries are included, Windows is well over 70 million lines of code or 10 to 20 bookcases. And this excludes basic application software (things like Windows Explorer, Windows Media Player, and so on).

It should be clear now why operating systems live a long time—they are very hard to write, and having written one, the owner is loath to throw it out and start again. Instead, such systems evolve over long periods of time. Windows 95/98/Me was basically one operating system and Windows NT/2000/XP/Vista/Windows 7 is a different one. They look similar to the users because Microsoft made very sure that the user interface of Windows 2000/XP/Vista/Windows 7 was quite similar to that of the system it was replacing, mostly Windows 98. Nevertheless, there were very good reasons why Microsoft got rid of Windows 98. We will come to these when we study Windows in detail in Chap. 11.

Besides Windows, the other main example we will use throughout this book is UNIX and its variants and clones. It, too, has evolved over the years, with versions like System V, Solaris, and FreeBSD being derived from the original system, whereas Linux is a fresh code base, although very closely modeled on UNIX and highly compatible with it. We will use examples from UNIX throughout this book and look at Linux in detail in Chap. 10.

In this chapter we will briefly touch on a number of key aspects of operating systems, including what they are, their history, what kinds are around, some of the basic concepts, and their structure. We will come back to many of these important topics in later chapters in more detail.

1.1 WHAT IS AN OPERATING SYSTEM?

It is hard to pin down what an operating system is other than saying it is the software that runs in kernel mode—and even that is not always true. Part of the problem is that operating systems perform two essentially unrelated functions:

providing application programmers (and application programs, naturally) a clean abstract set of resources instead of the messy hardware ones and managing these hardware resources. Depending on who is doing the talking, you might hear mostly about one function or the other. Let us now look at both.

1.1.1 The Operating System as an Extended Machine

The **architecture** (instruction set, memory organization, I/O, and bus structure) of most computers at the machine-language level is primitive and awkward to program, especially for input/output. To make this point more concrete, consider modern **SATA (Serial ATA)** hard disks used on most computers. A book (Anderson, 2007) describing an early version of the interface to the disk—what a programmer would have to know to use the disk—ran over 450 pages. Since then, the interface has been revised multiple times and is more complicated than it was in 2007. Clearly, no sane programmer would want to deal with this disk at the hardware level. Instead, a piece of software, called a **disk driver**, deals with the hardware and provides an interface to read and write disk blocks, without getting into the details. Operating systems contain many drivers for controlling I/O devices.

But even this level is much too low for most applications. For this reason, all operating systems provide yet another layer of abstraction for using disks: files. Using this abstraction, programs can create, write, and read files, without having to deal with the messy details of how the hardware actually works.

This abstraction is the key to managing all this complexity. Good abstractions turn a nearly impossible task into two manageable ones. The first is defining and implementing the abstractions. The second is using these abstractions to solve the problem at hand. One abstraction that almost every computer user understands is the file, as mentioned above. It is a useful piece of information, such as a digital photo, saved email message, song, or Web page. It is much easier to deal with photos, emails, songs, and Web pages than with the details of SATA (or other) disks. The job of the operating system is to create good abstractions and then implement and manage the abstract objects thus created. In this book, we will talk a lot about abstractions. They are one of the keys to understanding operating systems.

This point is so important that it is worth repeating in different words. With all due respect to the industrial engineers who so carefully designed the Macintosh, hardware is ugly. Real processors, memories, disks, and other devices are very complicated and present difficult, awkward, idiosyncratic, and inconsistent interfaces to the people who have to write software to use them. Sometimes this is due to the need for backward compatibility with older hardware. Other times it is an attempt to save money. Often, however, the hardware designers do not realize (or care) how much trouble they are causing for the software. One of the major tasks of the operating system is to hide the hardware and present programs (and their programmers) with nice, clean, elegant, consistent, abstractions to work with instead. Operating systems turn the ugly into the beautiful, as shown in Fig. 1-2.

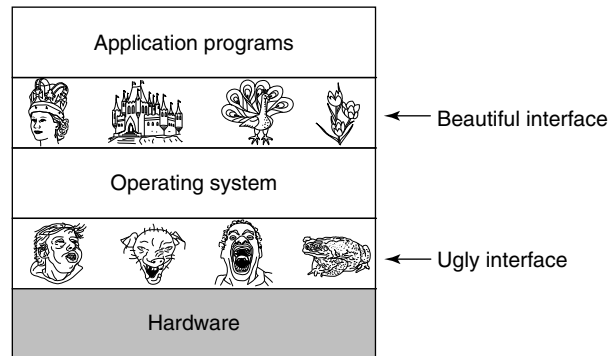


Figure 1-2. Operating systems turn ugly hardware into beautiful abstractions.

It should be noted that the operating system's real customers are the application programs (via the application programmers, of course). They are the ones who deal directly with the operating system and its abstractions. In contrast, end users deal with the abstractions provided by the user interface, either a command-line shell or a graphical interface. While the abstractions at the user interface may be similar to the ones provided by the operating system, this is not always the case. To make this point clearer, consider the normal Windows desktop and the line-oriented command prompt. Both are programs running on the Windows operating system and use the abstractions Windows provides, but they offer very different user interfaces. Similarly, a Linux user running Gnome or KDE sees a very different interface than a Linux user working directly on top of the underlying X Window System, but the underlying operating system abstractions are the same in both cases.

In this book, we will study the abstractions provided to application programs in great detail, but say rather little about user interfaces. That is a large and important subject, but one only peripherally related to operating systems.

1.1.2 The Operating System as a Resource Manager

The concept of an operating system as primarily providing abstractions to application programs is a top-down view. An alternative, bottom-up, view holds that the operating system is there to manage all the pieces of a complex system. Modern computers consist of processors, memories, timers, disks, mice, network interfaces, printers, and a wide variety of other devices. In the bottom-up view, the job of the operating system is to provide for an orderly and controlled allocation of the processors, memories, and I/O devices among the various programs wanting them.

Modern operating systems allow multiple programs to be in memory and run at the same time. Imagine what would happen if three programs running on some computer all tried to print their output simultaneously on the same printer. The first

few lines of printout might be from program 1, the next few from program 2, then some from program 3, and so forth. The result would be utter chaos. The operating system can bring order to the potential chaos by buffering all the output destined for the printer on the disk. When one program is finished, the operating system can then copy its output from the disk file where it has been stored for the printer, while at the same time the other program can continue generating more output, oblivious to the fact that the output is not really going to the printer (yet).

When a computer (or network) has more than one user, the need for managing and protecting the memory, I/O devices, and other resources is even more since the users might otherwise interfere with one another. In addition, users often need to share not only hardware, but information (files, databases, etc.) as well. In short, this view of the operating system holds that its primary task is to keep track of which programs are using which resource, to grant resource requests, to account for usage, and to mediate conflicting requests from different programs and users.

Resource management includes **multiplexing** (sharing) resources in two different ways: in time and in space. When a resource is time multiplexed, different programs or users take turns using it. First one of them gets to use the resource, then another, and so on. For example, with only one CPU and multiple programs that want to run on it, the operating system first allocates the CPU to one program, then, after it has run long enough, another program gets to use the CPU, then another, and then eventually the first one again. Determining how the resource is time multiplexed—who goes next and for how long—is the task of the operating system. Another example of time multiplexing is sharing the printer. When multiple print jobs are queued up for printing on a single printer, a decision has to be made about which one is to be printed next.

The other kind of multiplexing is space multiplexing. Instead of the customers taking turns, each one gets part of the resource. For example, main memory is normally divided up among several running programs, so each one can be resident at the same time (for example, in order to take turns using the CPU). Assuming there is enough memory to hold multiple programs, it is more efficient to hold several programs in memory at once rather than give one of them all of it, especially if it only needs a small fraction of the total. Of course, this raises issues of fairness, protection, and so on, and it is up to the operating system to solve them. Another resource that is space multiplexed is the disk. In many systems a single disk can hold files from many users at the same time. Allocating disk space and keeping track of who is using which disk blocks is a typical operating system task.

1.2 HISTORY OF OPERATING SYSTEMS

Operating systems have been evolving through the years. In the following sections we will briefly look at a few of the highlights. Since operating systems have historically been closely tied to the architecture of the computers on which they

run, we will look at successive generations of computers to see what their operating systems were like. This mapping of operating system generations to computer generations is crude, but it does provide some structure where there would otherwise be none.

The progression given below is largely chronological, but it has been a bumpy ride. Each development did not wait until the previous one nicely finished before getting started. There was a lot of overlap, not to mention many false starts and dead ends. Take this as a guide, not as the last word.

The first true digital computer was designed by the English mathematician Charles Babbage (1792–1871). Although Babbage spent most of his life and fortune trying to build his “analytical engine,” he never got it working properly because it was purely mechanical, and the technology of his day could not produce the required wheels, gears, and cogs to the high precision that he needed. Needless to say, the analytical engine did not have an operating system.

As an interesting historical aside, Babbage realized that he would need software for his analytical engine, so he hired a young woman named Ada Lovelace, who was the daughter of the famed British poet Lord Byron, as the world’s first programmer. The programming language Ada[®] is named after her.

1.2.1 The First Generation (1945–55): Vacuum Tubes

After Babbage’s unsuccessful efforts, little progress was made in constructing digital computers until the World War II period, which stimulated an explosion of activity. Professor John Atanasoff and his graduate student Clifford Berry built what is now regarded as the first functioning digital computer at Iowa State University. It used 300 vacuum tubes. At roughly the same time, Konrad Zuse in Berlin built the Z3 computer out of electromechanical relays. In 1944, the Colossus was built and programmed by a group of scientists (including Alan Turing) at Bletchley Park, England, the Mark I was built by Howard Aiken at Harvard, and the ENIAC was built by William Mauchley and his graduate student J. Presper Eckert at the University of Pennsylvania. Some were binary, some used vacuum tubes, some were programmable, but all were very primitive and took seconds to perform even the simplest calculation.

In these early days, a single group of people (usually engineers) designed, built, programmed, operated, and maintained each machine. All programming was done in absolute machine language, or even worse yet, by wiring up electrical circuits by connecting thousands of cables to plugboards to control the machine’s basic functions. Programming languages were unknown (even assembly language was unknown). Operating systems were unheard of. The usual mode of operation was for the programmer to sign up for a block of time using the signup sheet on the wall, then come down to the machine room, insert his or her plugboard into the computer, and spend the next few hours hoping that none of the 20,000 or so vacuum tubes would burn out during the run. Virtually all the problems were simple

straightforward mathematical and numerical calculations, such as grinding out tables of sines, cosines, and logarithms, or computing artillery trajectories.

By the early 1950s, the routine had improved somewhat with the introduction of punched cards. It was now possible to write programs on cards and read them in instead of using plugboards; otherwise, the procedure was the same.

1.2.2 The Second Generation (1955–65): Transistors and Batch Systems

The introduction of the transistor in the mid-1950s changed the picture radically. Computers became reliable enough that they could be manufactured and sold to paying customers with the expectation that they would continue to function long enough to get some useful work done. For the first time, there was a clear separation between designers, builders, operators, programmers, and maintenance personnel.

These machines, now called **mainframes**, were locked away in large, specially air-conditioned computer rooms, with staffs of professional operators to run them. Only large corporations or major government agencies or universities could afford the multimillion-dollar price tag. To run a **job** (i.e., a program or set of programs), a programmer would first write the program on paper (in FORTRAN or assembler), then punch it on cards. He would then bring the card deck down to the input room and hand it to one of the operators and go drink coffee until the output was ready.

When the computer finished whatever job it was currently running, an operator would go over to the printer and tear off the output and carry it over to the output room, so that the programmer could collect it later. Then he would take one of the card decks that had been brought from the input room and read it in. If the FORTRAN compiler was needed, the operator would have to get it from a file cabinet and read it in. Much computer time was wasted while operators were walking around the machine room.

Given the high cost of the equipment, it is not surprising that people quickly looked for ways to reduce the wasted time. The solution generally adopted was the **batch system**. The idea behind it was to collect a tray full of jobs in the input room and then read them onto a magnetic tape using a small (relatively) inexpensive computer, such as the IBM 1401, which was quite good at reading cards, copying tapes, and printing output, but not at all good at numerical calculations. Other, much more expensive machines, such as the IBM 7094, were used for the real computing. This situation is shown in Fig. 1-3.

After about an hour of collecting a batch of jobs, the cards were read onto a magnetic tape, which was carried into the machine room, where it was mounted on a tape drive. The operator then loaded a special program (the ancestor of today's operating system), which read the first job from tape and ran it. The output was written onto a second tape, instead of being printed. After each job finished, the operating system automatically read the next job from the tape and began running

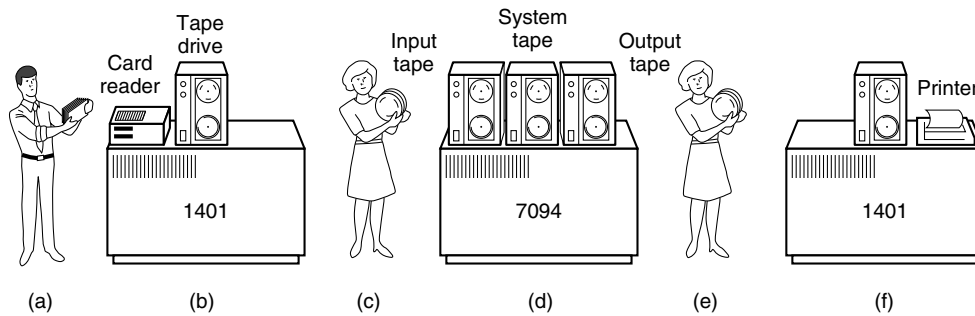


Figure 1-3. An early batch system. (a) Programmers bring cards to 1401. (b) 1401 reads batch of jobs onto tape. (c) Operator carries input tape to 7094. (d) 7094 does computing. (e) Operator carries output tape to 1401. (f) 1401 prints output.

it. When the whole batch was done, the operator removed the input and output tapes, replaced the input tape with the next batch, and brought the output tape to a 1401 for printing **off line** (i.e., not connected to the main computer).

The structure of a typical input job is shown in Fig. 1-4. It started out with a \$JOB card, specifying the maximum run time in minutes, the account number to be charged, and the programmer's name. Then came a \$FORTRAN card, telling the operating system to load the FORTRAN compiler from the system tape. It was directly followed by the program to be compiled, and then a \$LOAD card, directing the operating system to load the object program just compiled. (Compiled programs were often written on scratch tapes and had to be loaded explicitly.) Next came the \$RUN card, telling the operating system to run the program with the data following it. Finally, the \$END card marked the end of the job. These primitive control cards were the forerunners of modern shells and command-line interpreters.

Large second-generation computers were used mostly for scientific and engineering calculations, such as solving the partial differential equations that often occur in physics and engineering. They were largely programmed in FORTRAN and assembly language. Typical operating systems were FMS (the Fortran Monitor System) and IBSYS, IBM's operating system for the 7094.

1.2.3 The Third Generation (1965–1980): ICs and Multiprogramming

By the early 1960s, most computer manufacturers had two distinct, incompatible, product lines. On the one hand, there were the word-oriented, large-scale scientific computers, such as the 7094, which were used for industrial-strength numerical calculations in science and engineering. On the other hand, there were the