IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Patent of:	Gregory G. Raleigh	Attorney Docket No.39843-0190IP1
U.S. Patent No.:	10,028,144	
Issue Date:	July 17, 2018	
Appl. Serial No.:	14/948,065	
Filing Date:	November 20, 2015	
Title:	SECURITY TECHNIQUES FOR DEVICE ASSISTED SERVICES	

DECLARATION OF JUNE ANN MUNFORD

SAMSUNG 1012

1. My name is June Ann Munford. I am over the age of 18, have personal knowledge of the facts set forth herein, and am competent to testify to the same.

2. I earned a Master of Library and Information Science (MLIS) from the University of Wisconsin-Milwaukee in 2009. I have over ten years of experience in the library/information science field. Beginning in 2004, I have served in various positions in the public library sector including Assistant Librarian, Youth Services Librarian and Library Director. I have attached my Curriculum Vitae as Appendix CV.

3. During my career in the library profession, I have been responsible for materials acquisition for multiple libraries. In that position, I have cataloged, purchased and processed incoming library works. That includes purchasing materials directly from vendors, recording publishing data from the material in question, creating detailed material records for library catalogs and physically preparing that material for circulation. In addition to my experience in acquisitions, I was also responsible for analyzing large collections of library materials, tailoring library records for optimal catalog search performance and creating lending agreements between libraries during my time as a Library Director.

4. I am fully familiar with the catalog record creation process in the library sector. In preparing a material for public availability, a library catalog record describing that material would be created. These records are typically written in

Machine Readable Catalog (herein referred to as "MARC") code and contain information such as a physical description of the material, metadata from the material's publisher, and date of library acquisition. In particular, the 008 field of the MARC record is reserved for denoting the date of creation of the library record itself. As this typically occurs during the process of preparing materials for public access, it is my experience that an item's MARC record indicates the date of an item's public availability.

5. Typically, in creating a MARC record, a librarian would gather various bits of metadata such as book title, publisher and subject headings among others and assign each value to a relevant numerical field. For example, a book's physical description is tracked in field 300 while title/attribution is tracked in field 245. The 008 field of the MARC record is reserved for denoting the creation of the library record itself. As this is the only date reflecting the inclusion of said materials within the library's collection, it is my experience that an item's 008 field accurately indicates the date of an item's public availability.

6. I have reviewed Exhibit SAMSUNG-1007 (Excerpts) Smith, et al., 2005. "Virtual Machines: Versa-tile Platforms for Systems and Processes," Elsevier, Inc, 2005, ISBN 1-55860-910-5 ("Smith").

7. Attached hereto as Appendix SMITH01 is a true and correct copy of Virtual Machine: Versatile Platforms for Systems and Processes as made available

to the public via the O'Reilly Learning Database. I secured screen captures of this e-book version myself from the O'Reilly website (accessible at https://learning.oreilly.com/library/view/virtual-machines/9781558609105/). The cover page found on page 1 of Appendix SMITH01 accurately describes the title, author and publisher of Virtual Machine: Versatile Platforms for Systems and Processes. In comparing Exhibit SAMSUNG-1007 to Appendix SMITH01, it is my determination that Exhibit SAMSUNG-1007 is a true and correct copy of Virtual Machine: Versatile Platforms for Systems and Processes by James E. Smith. The 'Publication Date' field of this page indicates this e-book version of the book was first made available to the public as of July 2005. O'Reilly is a publishing company that hosts e-book and multimedia editions of various textbooks and academic works on its learning.oreilly.com platform. These publications are licensed from the work's original publishers and include publication data as provided by the publisher. I have used the O'Reilly database in my work and generally consider the bibliographic data presented on this website to be as accurate as bibliographic data provided directly from a given work's publisher. The learning.oreilly.com entry for Virtual Machine: Versatile Platforms for Systems and Processes accurately describes the title, author, publisher and content of Virtual Machine: Versatile Platforms for Systems and Processes. The 'print length' field of Appendix SMITH01 describes the book as

containing 656 pages, which matches the number of pages in the book from which the excerpts found in Exhibit SAMSUNG-1007 were taken.[1]

8. Attached hereto as Appendix SMITH02 is a true and correct copy of the MARC record for Virtual Machine: Versatile Platforms for Systems and Processes as held by the University of Pittsburgh Library. I secured this record myself from the library's public catalog. The MARC record contained within Appendix SMITH02 accurately describes the title, author, publisher, and ISBN number of Virtual Machine: Versatile Platforms for Systems and Processes. In comparing Exhibit SAMSUNG-1007 to Appendix SMITH02, it is my determination that Exhibit SAMSUNG-1007 is a true and correct copy of Virtual Machine: Versatile Platforms for Systems and Processes by James E. Smith.

9. The 008 field of the MARC record in Appendix SMITH02 indicates the date of record creation. The 008 field of Appendix SMITH02 indicates the University of Pittsburgh Library first acquired Virtual Machine: Versatile Platforms for Systems and Processes as of May 9, 2005. Considering this information, it is my determination that Virtual Machine: Versatile Platforms for Systems and Processes

[1] In the print version of the book, the final numbered page is 638 and there are 18 pages before the page numbered 1.

was made available to the public via the University of Pittsburgh Library shortly after its initial acquisition as of May 9, 2005.

10. Attached hereto as Appendix SMITH03 is a true and correct copy of the MARC record for Virtual Machine: Versatile Platforms for Systems and Processes as held by the Carnegie Mellon University Library. I secured this record myself from the library's public catalog. The MARC record contained within Appendix SMITH03 accurately describes the title, author, publisher, and ISBN number of Virtual Machine: Versatile Platforms for Systems and Processes . In comparing Exhibit SAMSUNG-1007 to Appendix SMITH03, it is my determination that Exhibit is a true and correct copy of Virtual Machine: Versatile Platforms for Systems and Processes by James E. Smith.

11. The 008 field of the MARC record in Appendix SMITH03 indicates the date of record creation. The 008 field of Appendix SMITH03 indicates the Carnegie Mellon University Library first acquired Virtual Machine: Versatile Platforms for Systems and Processes as of May 9, 2005. Considering this information, it is my determination that Virtual Machine: Versatile Platforms for Systems and Processes was made available to the public via Carnegie Mellon University Library shortly after its initial acquisition as of May 9, 2005.

12. Attached hereto as Appendix SMITH04 is a true and correct copy of the MARC record for Virtual Machine: Versatile Platforms for Systems and

Processes as held by the Penn State University Library. I secured this record myself from the library's public catalog. The MARC record contained within Appendix SMITH04 accurately describes the title, author, publisher, and ISBN number of Virtual Machine: Versatile Platforms for Systems and Processes . In comparing Exhibit to Appendix SMITH04, it is my determination that Exhibit is a true and correct copy of Virtual Machine: Versatile Platforms for Systems and Processes by James E. Smith.

13. The 008 field of the MARC record in Appendix SMITH04 indicates the date of record creation. The 008 field of Appendix SMITH04 indicates the Penn State University Library first acquired Virtual Machine: Versatile Platforms for Systems and Processes as of August 2, 2007. Considering this information, it is my determination that Virtual Machine: Versatile Platforms for Systems and Processes was made available to the public shortly after its initial acquisition as of August 2, 2007.

14. Attached hereto as Appendix SMITH05 is a screen capture of the Internet Archive Wayback Machine entry for http://books.elsevier.com/us/mk/us/subindex.asp?maintarget=&isbn=&country=United+States&srccode=&ref=&subcode=&head=&pdf=&basiccode=&txtSearch=&SearchField=&operator=&order=&community=mk. I secured these screen captures myself from

https://web.archive.org/web/20051122210123/http://books.elsevier.com/us/mk/us/subindex.asp?maintarget=&isbn=&country=United+States&srccode=&ref=&subcode=&head=&pdf=&basiccode=&txtSearch=&SearchField=&operator=&order=&community=mk. This record was captured as of November 22, 2005.

15. Appendix SMITH05 is an Internet Archive capture of a web page entitled Computing - MK Bookstore as hosted on the Morgan Kaufmann website. This page contains the bibliograph elements of many books within the Morgan Kaufmann catalog. Page 4 of Appendix SMITH05 contains an entry for Virtual Machine: Versatile Platforms for Systems and Processes which indicates the book was first published as of June 3, 2005 and is currently available for purchase as of the record's capture date of November 22, 2005.

16. Attached hereto as Appendix SMITH06 is a screen capture of the Internet Archive Wayback Machine entry for http://books.elsevier.com/us/bookscat/search/details.asp?country=United+States&txtSearch=&searchfield=subjectcode&imprint=Morgan+Kaufmann&operator=%3D&textbook=&highlight=0&display=collapse&isbn=1558609105&srccode=&community=mk&ref=&mscssid=LJUXMV8TRRDW9LH72DK85TF888R692F1. I secured these screen captures myself from https://web.archive.org/web/20060324094915/http://books.elsevier.com/us/bookscat/search/details.asp?country=United+States&txtSearch=&searchfield=subjectcode

&imprint=Morgan+Kaufmann&operator=%3D&textbook=&highlight=0&display=collapse&isbn=1558609105&srccode=&community=mk&ref=&mscssid=LJUXMV8TRRDW9LH72DK85TF888R692F1. This record was captured as of March 24, 2006.

17. Appendix SMITH06 is an Internet Archive capture of a web page entitled Virtual Machines: Versatile Platforms for Systems and Processes as hosted on the Morgan Kaufmann website. This page contains a full description of Virtual Machines: Versatile Platforms for Systems and Processes as well as an 'add to cart' image which indicates users were able to purchase a copy of this book from the Morgan Kaufmann website. As this record was captured as of May 24, 2006, this page indicates Virtual Machines: Versatile Platforms for Systems and Processes was available to the public via the Morgan Kaufmann website no later than May 24, 2006.

18. Appendix SMITH01 indicates Virtual Machines: Versatile Platforms for Systems and Processes was first published in July 2005. Appendix SMITH02 indicates the University of Pittsburgh Library first acquired this book as of May 9, 2005 and made it available to the public soon after. Appendix SMITH03 indicates the Carnegie Mellon University Library first acquired this book as of May 9, 2005 and made it available to the public soon after. Appendix SMITH04 indicates the Penn State University Library first acquired this book as of August 2, 2007 and made

it available to the public soon after. Appendices SMITH05 and SMITH06 both indicate this book was published as of June 3, 2005. SMITH06 indicates this book was on sale from the Morgan Kaufmann website as of March 24, 2006. In consideration of each of these dates, it is my determination that Virtual Machines: Versatile Platforms for Systems and Processes was first published as of June 3, 2005, made available to the public by several libraries soon after this publication and was certainly available to the public no later than March 24, 2006.

19. I have been retained on behalf of the Petitioner to provide assistance in the above-illustrated matter in establishing the authenticity and public availability of the documents discussed in this declaration. I am being compensated for my services in this matter at the rate of $200.00 per hour plus reasonable expenses. My statements are objective, and my compensation does not depend on the outcome of this matter.

20. I declare under penalty of perjury that the foregoing is true and correct. I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code.

Date: 5/2/2025 By: _____
June Ann Munford

APPENDIX CV

June A. Munford
Curriculum Vitae

Education

University of Wisconsin-Milwaukee - MS, Library & Information Science, 2009
Milwaukee, WI

- Coursework included cataloging, metadata, data analysis, library systems, management strategies and collection development.
- Specialized in library advocacy, cataloging and public administration.

Grand Valley State University - BA, English Language & Literature, 2008
Allendale, MI

- Coursework included linguistics, documentation and literary analysis.
- Minor in political science with a focus in local-level economics and government.

Professional Experience

Researcher / Expert Witness, October 2017 – present
Freelance ● Pittsburgh, Pennsylvania & Grand Rapids, Michigan

- Material authentication and public accessibility determination. Declarations of authenticity and/or public accessibility provided upon research completion. Experienced with appeals and deposition process.

- Research provided on topics of public library operations, material publication history, digital database services and legacy web resources.

- Past clients include Alston & Bird, Arnold & Porter, Baker Botts, Fish & Richardson, Erise IP, Irell & Manella, O'Melveny & Myers, Perkins-Coie, Pillsbury Winthrop Shaw Pittman and Slayden Grubert Beard.

Library Director, February 2013 - March 2015
Dowagiac District Library ● Dowagiac, Michigan

- Executive administrator of the Dowagiac District Library. Located in

Southwest Michigan, this library has a service area of 13,000, an annual operating budget of over $400,000 and total assets of approximately $1,300,000.

● Developed careful budgeting guidelines to produce a 15% surplus during the 2013-2014 & 2014-2015 fiscal years while being audited.

● Using this budget surplus, oversaw significant library investments including the purchase of property for a future building site, demolition of existing buildings and building renovation projects on the current facility.

● Led the organization and digitization of the library's archival records.

● Served as the public representative for the library, developing business relationships with local school, museum and tribal government entities.

● Developed an objective-based analysis system for measuring library services - including a full collection analysis of the library's 50,000+ circulating items and their records.

November 2010 - January 2013
Librarian & Branch Manager, Anchorage Public Library ● Anchorage, Alaska

● Headed the 2013 Anchorage Reads community reading campaign including event planning, staging public performances and creating marketing materials for mass distribution.

● Co-led the social media department of the library's marketing team, drafting social media guidelines, creating original content and instituting long-term planning via content calendars.

● Developed business relationships with The Boys & Girls Club, Anchorage School District and the US Army to establish summer reading programs for children.

June 2004 - September 2005, September 2006 - October 2013
Library Assistant, Hart Area Public Library
Hart, MI

● Responsible for verifying imported MARC records and original MARC

cataloging for the local-level collection as well as the Michigan Electronic Library.

● Handled OCLC Worldcat interlibrary loan requests & fulfillment via ongoing communication with lending libraries.

Professional Involvement

Alaska Library Association - Anchorage Chapter
- Treasurer, 2012

Library Of Michigan
- Level VII Certification, 2008
- Level II Certification, 2013

Michigan Library Association Annual Conference 2014
- New Directors Conference Panel Member

Southwest Michigan Library Cooperative
- Represented the Dowagiac District Library, 2013-2015

Professional Development

Library Of Michigan Beginning Workshop, May 2008
Petoskey, MI
- Received training in cataloging, local history, collection management, children's literacy and reference service.

Public Library Association Intensive Library Management Training, October 2011
Nashville, TN
- Attended a five-day workshop focused on strategic planning, staff management, statistical analysis, collections and cataloging theory.

Alaska Library Association Annual Conference 2012 - Fairbanks, February 2012
Fairbanks, AK
- Attended seminars on EBSCO advanced search methods, budgeting, cataloging, database usage and marketing.

Depositions

2019 ● Fish & Richardson

Apple v. Qualcomm (IPR2018-001281, 39521-00421IP, IPR2018-01282 and 39521-00421IP2)

2019 ● Erise IP

Implicit, LLC v. Netscout Systems, Inc (Civil Action No. 2:18-cv-53-JRG)

2019 ● Perkins-Coie

Adobe Inc. v. RAH Color Technologies LLC (Cases IPR2019-00627, IPR2019-00628, IPR2019-00629 and IPR2019-00646)

2020 ● O'Melveny & Myers

Maxell, Ltd. v. Apple Inc. (Case No. 5:19-cv-00036-RWS)

2021 ● Pillsbury Winthrop Shaw Pittman LLP

Intel v. SRC (IPR2020-1449)

2022 ● Perkins-Coie

Realtek v. Future Link (IPR2021-01182)

2023 ● Fish & Richardson

Neuroderm Ltd. v. Abbvie, Inc (Case No. PGR2022-00040)

2023 ● Fish & Richardson

Nearmap US Inc. v. Pictometry International Corp. (IPR2022-00735)

2023 ● Fish & Richardson

Samsung Electronics v. MemoryWeb LLC (Case No. 39843-0136PS1)

2023 ● Pillsbury Winthrop Shaw Whitman LLP

Gravel Rating Systems v. Costco Wholesale Corp. (Civil Action No. 4:21-cv-149-ALM)

2024 ● Willkie-Farr

Netflix, Inc. v. VideoLabs. Inc. (IPR2023-00628)

Limited Case and Clientele History

Alston & Bird

- Ericsson

  v. Collision Communications (IPR2022-01233)

- Nokia

  v. Neptune Subsea, Xtera (Case No. 1:17-cv-01876)

- Universal Electronics Inc

  v. Roku Inc (IPR2022-00818)

Arnold & Porter

- Ivantis

  v. Glaukos (Case No. 8:18-cv-00620)

- Samsung

  v. Jawbone (Case No. 2:21-cv-00186)

Benesch Friedlander Coplan & Aronoff

- Voyis

  v. Cathx (Case No. 5:21-cv-00077-RWS)

Deschert LLP

- Smaxtec Animal Care

  v. ST Reproductive Technologies, LLC (IPR2024-00885)

Erise I.P.

- Apple

  v. Ericsson Inc. (IPR2022-00715)

  v. Future Link Systems (IPRs 6317804, 6622108, 6807505, and 7917680)

  v. INVT (Case No. 20-1881)

  v. Navblazer LLC (IPR2020-01253)

  v. Qualcomm (IPR2018-001281, 39521-00421IP, IPR2018-01282, 39521-00421IP2)

  v. Quest Nettech Corp (Case No. 2:19-cv-00118-JRG)

  v. Telefonaktiebolaget LM Ericsson (IPR2022-00275)

  v. Theta IP, LLC (IPR2024-00818)

- Fanduel

  v. CGT (Case No. 19-1393)

- Garmin

  v. Phillips North America LLC (Case No. 2:19-cv-6301-AB-KS)

- Netscout
    - v. Longhorn HD LLC (Case No. 2:20-cv-00349)
    - v. Implicit, LLC (Civil Action No. 2:18-cv-53-JRG)

- Sony Interactive Entertainment LLC
    - v. Bot M8 LLC (IPR2020-01288)
    - v. Infernal Technology LLC (Case No. 2:19-CV-00248-JRG)

- Unified Patents
    - v. GE Video Compression (Civil Action No. 2:19-cv-248)

Fish & Richardson

- Apple
    - v. AliveCor (3:21-cv-03958)
    - v. LBS Innovations (Case No. 2:19-cv-00119-JRG-RSP)
    - v. Koss Corporation (IPR2021-00305)
    - v. Masimo (IPR 50095-0012IP1, 50095-0012IP2, 50095-0013IP1, 50095-0013IP2, 50095-0006IP1, 50095-0135IP1)
    - v. Neonode (Case No. 21-cv-08872-EMC)
    - v. Qualcomm (IPR2018-001281, 39521-00421IP, IPR2018-01282, 39521-00421IP2)

- Dell
    - v. Neo Wireless (IPR2022-00616)

- Dish Network
    - v. Entropic Communications, LLC (Case No. 2:2023-CV-01043)

v. Realtime Adaptive Streaming (Case No. 1:17-CV-02097-RBJ)

v. TQ Delta LLC (Case No. 18-1798)

- Evapco Dry Cooling

  v. SPG Dry Cooling (IPR2021-00688)

- Genetec

  v. Sensormatic Electronics (Case No. 1:20-CV-00760)

- Huawei

  v. Bell Northern Research LLC (IPR2019-01174)

- Kianxis

  v. Blue Yonder (Case No. 3:20-cv-03636)

- LG Electronics

  v. Bell Northern Research LLC (Case No. 3:18-cv-2864-CAB-BLM)

- Metaswitch

  v. Sonus Networks (IPR2018-01719)

- Microsoft

  v. Throughputer Inc (IPR2022-00757)

- Mom Enterprises

  v. Ddrops Company (Case No. 1:22-cv-00332)

- MLC Intellectual Property

v. MicronTech (Case No. 3:14-cv-03657-SI)

- Nearmap Inc
  - v. EagleView Technologies (IPR2022-01009)

- Neuroderm Ltd.
  - v. Abbvie, Inc (Case No. PGR2022-00040)

- Realtek Semiconductor
  - v. Future Link (IPR2021-01182)

- Quectel
  - v. Koninklijke Philips (Case No. 1:20-cv-01710)

- Samsung
  - v. Aire Technology (IPR2022-00877)
  - v. Bell Northern Research (Case No. 2:19-cv-00286-JRG)
  - v. Communication Technologies Inc (IPR2022-01221)
  - v. Jawbone Innovations (IPR2022-00865)
  - v. MemoryWeb LLC (IPR2022-00885)
  - v. Dodots Licensing Solutions (IPR2023-00701)
  - v. Telefonaktiebolaget LM Ericsson (IPR2021-00615)

- Texas Instruments
  - v. Vantage Micro LLC (IPR2020-01261)

- Xilinx
  - v. Sentient Sensors LCC (Case No. 1:22-cv-00173)

Groombridge Wu

- Nearmap

  v. Eagle View Technologies Inc. and Pictometry International Corp. (Case No. 2:21-cv-00283-TS-DAO)

Irell & Manella

- Curium

O'Melveny & Myers

- Apple

  v. Maxell (Case No. 5:19-cv-00036-RWS)

- Disney Media & Entertainment Distribution LLC

  v. Digital Media Technology Holdings, LLC (Case No. 2:22-cv-01642)

- Micron Technology Inc.

  v. Besang Inc (IPR2023-00900)

- Samsung

  v. Daedalus Prime LLC (1335 Investigation)

Perkins-Coie

- Amazon

  v. Nokia (IPR2024-00847 and IPR2024-00848)

- Heru Industries
  - v. The UAB Research Foundation (IPR2022-01148)
- Intel Corporation
  - v. BeSang Inc. (IPR2023-00991)
  - v. BitMICRO (Case No. 5:23-cv-00625)
  - v. Telefonaktiebolaget L M Ericsson (IPR2024-00610)
- Realtek Semiconductor
  - v. Future Link (IPR2021-01182)
- Twitter Inc
  - v. VOIP-Pal.com (Case No. 3:20-cv-02397-JD)
- TCL Industries
  - v. Koninklijke Philips NV (IPR2021-00495, IPR2021-00496 and IPR2021-00497)

Pillsbury Winthrop Shaw Pittman

- Intel
  - v. FG SRC LLC (Case No. 6:20-cv-00315)
- Gravel Rating Systems
  - v. Costco (Case No. 4:21-cv-149)
  - v. Lowe's Home Centers (Case No. 4:21-cv-150)
  - v. T-Mobile USA (Case No. 4:21-cv-152)
  - v. Kohl's Inc. (Case No. 4:21-cv-258)
  - v. Under Armor (Case No. 4:21-cv-356)

Quinn Emanuel

- ServiceNow, Inc.

  v. InQuisient, Inc. (Case No. 22-900-CJB)

- Wolfspeed

  v. The Trustees of Purdue University (Case No. 1:2021-cv-00840)

Sheppard, Mullin, Richter & Hampton LLP

- Cadence Design Systems Inc.

  v. Semiconductor Design Technologies (Case No. 3:23-cv-01001)

Stern, Kessler, Goldstein & Fox

- Preliminary Research

Willkie, Farr & Gallagher LLP

- Neurent

  v. The Foundry, LLC (IPR2024-00669)

- Netflix, Inc.

  v. VideoLabs, Inc. (IPR2023-00628)

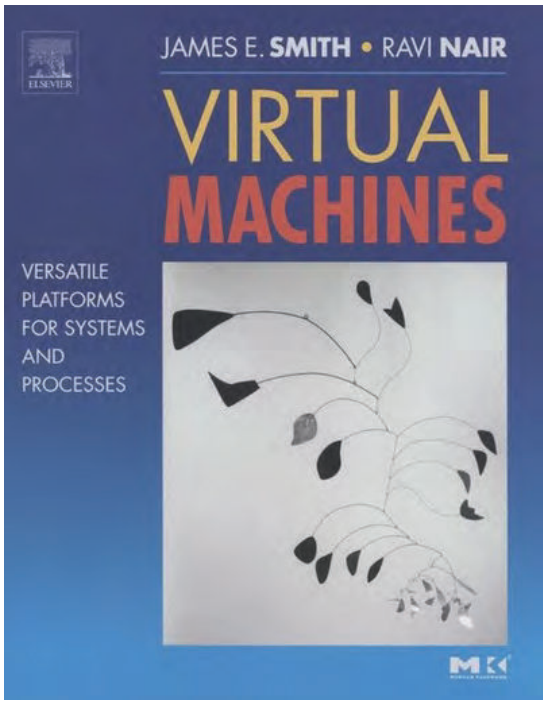
APPENDIX SMITH01

BOOK

Virtual Machines

★★★★★ 2 reviews

By Jim Smith, Ravi Nair



Start

TIME TO COMPLETE:
19h 52m

LEVEL:
Intermediate to advanced

SKILLS:
Virtualization

PUBLISHED BY:
Morgan Kaufmann

PUBLICATION DATE:
July 2005

PRINT LENGTH:
656 pages

+ Add to playlist

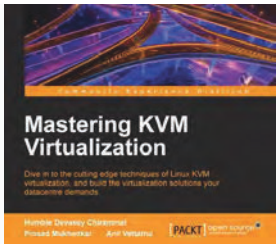
Virtual Machine technology applies the concept of virtualization to an entire machine, circumventing real machine compatibility constraints and hardware resource constraints to enable a higher degree of software portability and flexibility. Virtual machines are rapidly becoming an essential element in computer system design. They provide system security, flexibility, cross-platform compatibility, reliability, and resource efficiency. Designed to solve problems in combining and using major computer system components, virtual machine technologies play a key role in many disciplines, including operating systems, programming languages, and computer architecture. For example, at the process level, virtualizing technologies support dynamic program translation and platform-independent network computing. At the system level, they support multiple operating system environments on the same hardware platform and in servers.Historically, individual virtual machine techniques have been developed within the specific disciplines that employ them (in some cases they aren't even referred to as "virtual machines), making it difficult to see their common underlying relationships in a cohesive way. In this text, Smith and Nair take a new approach by examining virtual machines as a unified discipline. Pulling together cross-cutting technologies allows virtual machine implementations to be studied and engineered in a well-structured manner. Topics include instruction set emulation, dynamic program translation and optimization, high level virtual machines (including Java and CLI), and system virtual machines for both single-user systems and servers.

* Examines virtual machine technologies across the disciplines that use them—operating systems, programming languages and computer architecture —defining a new and unified discipline.* Reviewed by principle researchers at Microsoft, HP, and by other industry research groups.* Written by two authors who combine several decades of expertise in computer system research and development, both in academia and industry.

You might also like







BOOK

Mastering KVM Virtualization

By Humble Devassy Chirammal, Prasad Mukhedkar and Anil...

Dive in to the cutting edge techniques of Linux KVM virtualization, and build the virtualization solutions your datacentre de-

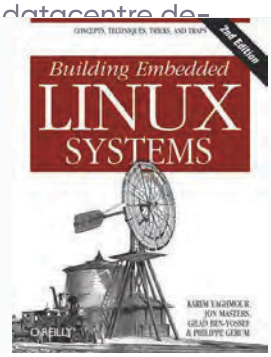


BOOK

Linux Device Drivers Development

By John Madieu

Learn to develop customized device drivers for your embedded Linux system About This Book Learn to develop customized Linux device drivers Learn

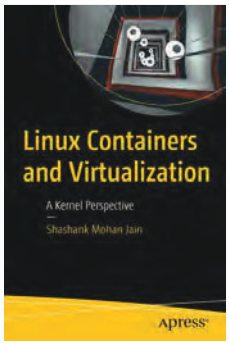


BOOK

Building Embedded Linux Systems, 2n...

By Philippe Gerum, Karim Yaghmour, Jon Masters and Gilad Ben-Yossef

There's a great deal of excitement surrounding the use of Linux in embedded systems -- for everything from cell phones to car ABS systems and water-



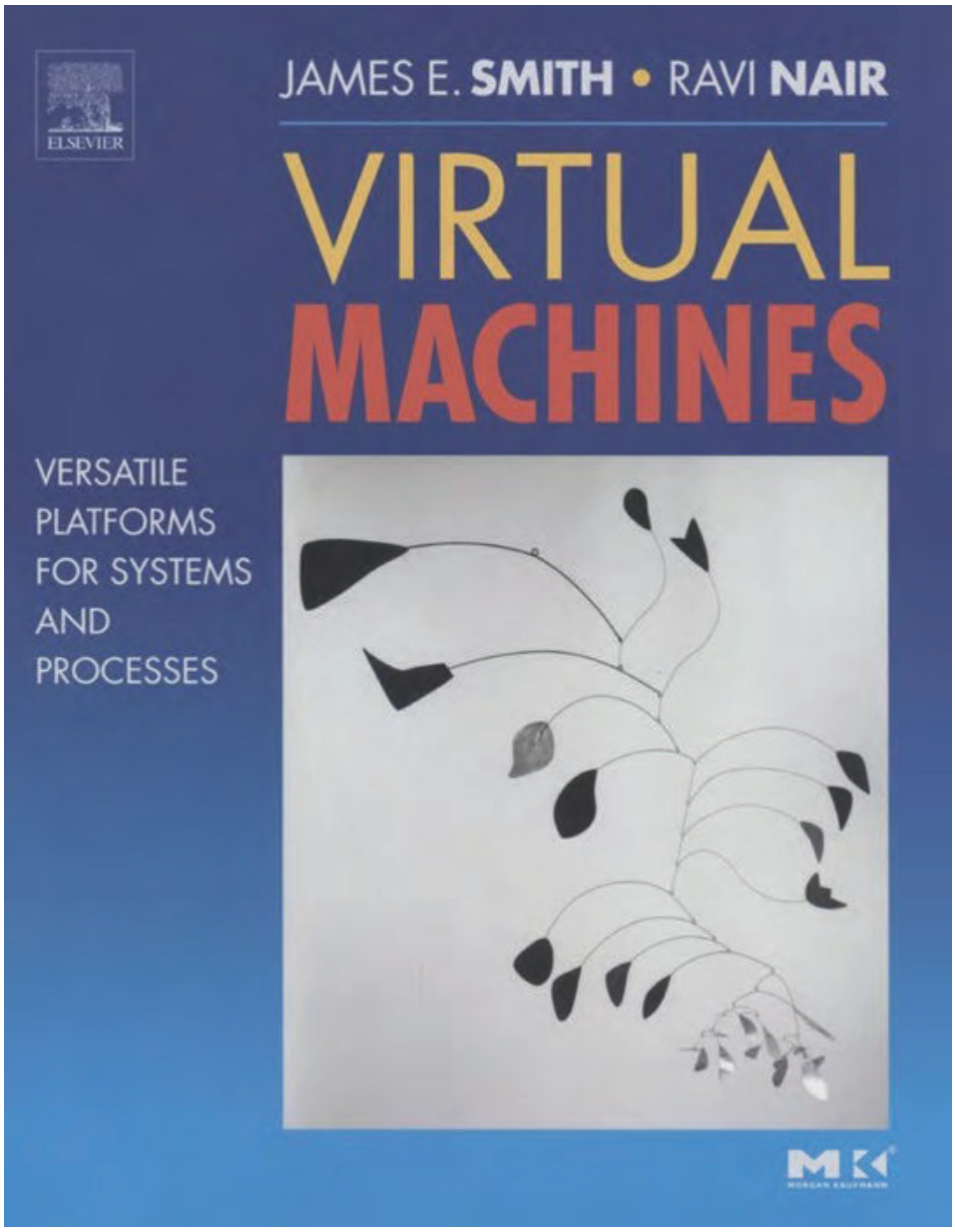
BOOK

Linux Containers and Virtualization:...

By Shashank Mohan Jain

Get a novel perspective on Linux containers and understand the world of virtualization. This book takes you down the rabbit hole to discover what lies below the

book. This book covers all about device driver develop-


ELSEVIER
JAMES E. SMITH • RAVI NAIR
VIRTUAL MACHINES
VERSATILE PLATFORMS FOR SYSTEMS AND PROCESSES
MK

O'REILLY

Virtual Machines

Versatile Platforms for Systems and Processes

James E. Smith

University of Wisconsin-Madison, USA

Ravi Nair

IBM Thomas J. Watson Research Center, USA

Morgan Kaufmann

To my mother

Publisher Denise E.M. Penrose
Publishing Services Manager Simon Crump
Senior Project Manager Angela Dooley
Editorial Assistant Kimberlee Honjo
Cover Design Hannus Design Associates
Text Design Rebecca Evans
Composition CEPHA Imaging Pvt. Ltd.
Technical Illustration Dartmouth Publishing, Inc.
Copyeditor Simon & Assocs.
Proofreader Phyllis Coyne et. al.
Indexer Northwind Editorial
Interior printer The Maple-Vail Manufacturing Group
Cover printer Phoenix Color, Inc.

Morgan Kaufmann Publishers is an imprint of Elsevier. 500 Sansome Street, Suite 400, San Francisco, CA 94111

This book is printed on acid-free paper.

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances in which Morgan Kaufmann Publishers is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any meansÑelectronic, mechanical, photocopying, scanning, or otherwiseÑwithout prior written permission of the publisher.

Permissions may be sought directly from Elsevierês Science & Technology Rights Department in Oxford, UK: phone: (+44) 1865 843830, fax: (+44) 1865 853333, e-mail **permissions@elsevier.co.uk**. You may also complete your request on-line via the Elsevier home-page (**http://elsevier.com**) by selecting çCustomer Supporté and then çObtaining Permissions.é

Cover Image: *Arc of Petals (Mobile)*, 1941 © Estate of Alexander Calder / Artists Rights Society (ARS), New York; The Solomon R. Guggenheim Foundation, New York, Peggy Guggenheim Collection, Venice, 1976.

Library of Congress Cataloging-in-Publication Data

Application Submitted

ISBN: 1-55860-910-5

For information on all Morgan Kaufmann publications, visit our Web site at **www.mkp.com** or **www.books.elsevier.com**

Printed in the United States of America

05 06 07 08 09 5 4 3 2 1

Foreword

I've been a virtual machine addict for precisely as long as I've worked with computers. My first real job, which led to my first non-trival computer program, was to implement a virtual machine for a high-level programming language. There seemed to be something magical about the ability for a computer to imitate another computer, or my idea of what a computer *ought* to be.

Now almost 20 years later, less starry-eyed and more responsible, I am concerned that my work has utility, making it faster or easier to get something done than it would be otherwise. But lucky for me, virtual machines have proven ideally suited to the needs of the computing industry, where the appreciation of the virtual machine has expanded dramatically. It's no longer only an intellectual challenge to the researcher arguably on the "lunatic fringe" of programming language implementation, or the secret weapon of a small cadre of mainframe O/S engineers and pioneers of system virtualization.

Several major trends have contributed to an explosion of interest in virtual machine technologies. In particular the rise of the World Wide Web, by far the largest and most ubiquitous cross-platform computing environment to date, created enormous and visible opportunities for virtual machine-based computing. Initially targeting the WWW, VM technology hit the mainstream as a means of safely hosting cross-platform binary programs embedded in Web pages. From that beachhead it has expanded to become the prevalent approach to open platforms from Web and back office servers to cell phones and smart cards, where the equivalent benefits—cross-platform applications that are not tied to the underlying hardware or operating system—invariably apply. Virtual machines form the basis of Sun's Java and Microsoft's .NET platforms, currently the

most popular software environments on the planet. As new markets or applications are conceived, virtual machine technologies to support them are adapted from these stalwarts.

In other markets as well, virtual machine technology is seeing a renaissance. Companies such as VMware are finding virtual machine platforms to be an ideal way to provide greater resource control, scalability and performance across variations in host hardware or operating systems. Virtual machines are likely to be common at multiple levels of the data center or server farm.

When I was asked to review this book, it seemed an opportunity to read something I might not get to otherwise. I expected to focus on the chapter covering virtual machines for high-level languages. Little did I know that I would find myself excited by less familiar topics, for instance sending back surprised comments expressing admiration for the decades-old IBM AS/400 architecture, which I'd never understood before. It wasn't just the realization of how much those coming far before me had accomplished. Seeing virtual machine technology in this broader scope made design decisions in my familiar Java virtual machine architecture, and their implications, much clearer. Such perspective is valuable even to experienced practitioners of a particular VM art.

And I found myself once again thinking how cool it all is.

Tim Lindholm
Distinguished Engineer, Sun Microsystems, Inc.
Palo Alto
February 28, 2005

Preface

Virtual machine (VM) technologies have been developed in a number of contexts—operating systems, programming languages and compilers, and computer architecture—to enable new capabilities and to solve a variety of problems in interfacing major computer system components. Virtual machines for supporting operating systems are receiving renewed interest after years of relatively little activity, because they allow effective resource sharing while maintaining a high degree of security. Virtualization is becoming popular for servers and other network applications especially where security is of crucial importance. In the area of programming languages, virtual machines provide platform independence, and they support transparent dynamic translation and optimization. In processor architectures, virtual machine technologies allow the introduction of new instruction sets, as well as dynamic optimization for power reduction and/or performance improvement.

Because of industry consolidation around a small number of standard interfaces, virtual machine technology will likely be an important enabling feature for innovations in all of the above fields. Any new instruction set, operating system, or programming language will almost certainly require some accomp

Historically, the various VM techniques have been spread across computer science and engineering disciplines. However, there are a number of underlying, cross-cutting technologies, and there is much to be gained by pulling them together so that VM implementations can be studied and engineered in a ve

This book is also about computer architecture in its purist sense. As classically defined, an *architecture* is an *interface*. Virtual machines couple interfaces and extend the flexibility and functionality of the interfaces. Understanding architecture is key to understanding virtual machines, and this book is written

The breadth of VM applications implies the audience for this book is fairly diverse. Although it is not currently recognized as a discipline with a targeted set of university courses, virtual machines makes an excellent topic for a graduate level course because it ties together the key disciplines of computer scie

The book begins by surveying the variety of VMs, putting them into perspective and building a framework for discussing VMs. The subsequent chapters describe the major types of VMs in an organized way, emphasizing the common, underlying technologies. Following is a rough outline summarizing each ch

In Chapter 1 we introduce the concept of abstraction and define the interfaces that are prevalent in today's computer systems. This is followed by a discussion of virtualization and its relationship to the interfaces. The notion of computer architecture is introduced next, followed by a survey of different types

In Chapter 2 we address issues related to the emulation of a source instruction set architecture (ISA) on a target ISA. We demonstrate the workings of a basic interpreter and show how threaded interpretation can help improve performance. The techniques developed are demonstrated using a CISC source IS

Chapter 3 discusses the implementation of process virtual machines. A process virtual machine supports an individual guest application program on a host platform consisting of an operating system and underlying hardware. We discuss the implications of VM compatibility and show how the state of a mac

Chapter 4 focuses on techniques for the optimization of translated code for better emulation performance. It discusses the framework needed to perform such optimizations and the profile information that must be gathered at program execution time in order to facilitate optimizations. Various profiling tec

Chapter 5 introduces high-level language virtual machines and traces the transition from the early Pascal P-code VMs to object-oriented VMs. The emphasis in this chapter is on the architecture of high-level language VMs, especially those features supporting object-oriented programming and security. The t

Chapter 6 continues the discussion of high-level language VMs by focusing on their implementation. As in the preceding chapter, more attention is given to Java because of its widespread usage and the variety in its implementations. Two aspects given special consideration are security and memory manage

In Chapter 7 we discuss co-designed virtual machines where a conventional ISA is implemented through a combination of an implementation-specific ISA and translation software that runs in concealed memory. We discuss techniques for mapping the state of the original ISA onto the implementation ISA an

Chapter 8 deals with the classic system virtual machines. A system virtual machine supports a complete guest operating system and all its applications on a host platform. We provide a motivation for these virtual machines and outline the basic ways for implementing system virtual machines, including nat

In Chapter 9 we shift our attention to the virtualization of multiprocessor systems. We introduce the notion of system partitioning and develop a taxonomy for different types of partitioning. We then discuss the principles behind physical partitioning and logical partitioning. The IBM LPAR feature is presen

Chapter 10 is a discussion of emerging applications for virtual machine technology. We focus on three application areas which we feel will be important in the coming years. The first is security. We discuss the vulnerability to attacks of modern computer systems and outline the principles behind intrusion d

The Appendix is essentially a condensed course in computer systems, providing background material for the main chapters. It discusses the roles of the processor, memory, and I/O in a computer system. This is followed by a discussion of ISAs, including support for user applications as well as for the operatir

The book may be used in a course in a variety of ways. Overall, the book is structured for a course focused on virtual machines as a topic in itself (an approach we advocate). For an operating system oriented treatment of virtual machines, an instructor can go straight to Chapters 8 through 10 after introduci

The specific interests of practitioners will largely determine the order in which they cover the material in the book. We have written the book in such a way that an interested reader can start at the beginning of any chapter of interest and follow the material in the complete chapter with only occasional exce

There are many people we would like to thank for having made this book possible. We would particularly like to thank the many reviewers. Michael Flynn of IBM Research, Jan Hoogerbrugge of Philips Research, Jim Larus of Microsoft Research, Tim Lindholm of Sun Microsystems, Bernd Mathisse of Sun Mic

The authors would also like to acknowledge Sriram Vajapeyam for his contributions during the early development of this material, and the students at the University of Wisconsin-Madison and Universitat Politècnica de Catalunya in Barcelona for their valuable feedback while participating in VM courses an

This book owes a lot to the guidance, persistence, and encouragement provided by our publisher, Denise Penrose, and the support provided by her excellent staff at Morgan-Kaufmann Publishers, including Kimberlee Honjo, Angela Dooley, Alyson Day, and Summer Block.

First author: I would like to thank the people at IBM Research, and Dan Prener in particular, for their support during my 2000-2001 sabbatical—the time this book had its genesis. I am especially grateful to Erik Altman for being a sounding board throughout the writing of the book. I also thank my graduate s

Second author: I would like to thank Dan Prener, Eric Kronstadt, and Jaime Moreno for their encouragement and support in undertaking this project. Thanks also to Peter Capek, Dan Prener, Peter Oden, Dick Attanasio, and Mark Mergen for many interesting tea-time discussions. Finally, I would like to thank

The authors are mutually grateful for the opportunity to renew a friendship that stretches back 30 years. We have had tremendous fun and have learnt a great deal in the process of writing this book. If you, the reader, experience just a fraction of what we have experienced, this book will have been worthwh

James E. Smith

Ravi Nair

Preface, as captured on pg. 6. This text has been transcribed from the O'Reilly website to account for display issues on the original page.

Full text:

Virtual machine (VM) technologies have been developed in a number of contexts—operating systems, programming languages and compilers, and computer architecture—to enable new capabilities and to solve a variety of problems in interfacing major computer system components. Virtual machines for supporting operating systems are receiving renewed interest after years of relatively little activity, because they allow effective resource sharing while maintaining a high degree of security. Virtualization is becoming popular for servers and other network applications especially where security is of crucial importance. In the area of programming languages, virtual machines provide platform independence, and they support transparent dynamic translation and optimization. In processor architectures, virtual machine technologies allow the introduction of new instruction sets, as well as dynamic optimization for power reduction and/or performance improvement.

Because of industry consolidation around a small number of standard interfaces, virtual machine technology will likely be an important enabling feature for innovations in all of the above fields. Any new instruction set, operating system, or programming language will almost certainly require some accompanying virtual machine technology if it is to become widely accepted. Not coincidentally, much of the impetuses for virtual machine technologies, and most of the more significant recent developments, have come from industry.

Historically, the various VM techniques have been spread across computer science and engineering disciplines. However, there are a number of underlying, cross-cutting technologies, and there is much to be gained by pulling them together so that VM implementations can be studied and engineered in a well-structured way. This book is an outgrowth of the idea that virtual machines should be studied as a unified discipline.

This book is also about computer architecture in its purist sense. As classically defined, an *architecture* is an *interface.* Virtual machines couple interfaces and extend the flexibility and functionality of the interfaces. Understanding architecture is key to understanding virtual machines, and this book is written from an architect's perspective, keeping interface issues clear and at the forefront. A goal

is for the reader to come away with a much deeper understanding of the important computer system interfaces and the role these interfaces play when the major components interact.

The breadth of VM applications implies the audience for this book is fairly diverse. Although it is not currently recognized as a discipline with a targeted set of university courses, virtual machines makes an excellent topic for a graduate level course because it ties together the key disciplines of computer science and engineering: architecture, operating systems, and programming languages. Preliminary versions of this book have already been used, quite successfully, in graduate courses at four different universities. The book can also be used as a supplementary text for a compiler course on dynamic optimization or an operating system course covering classic system VMs. Virtual machine technologies are rapidly gaining broad acceptance in industry, and practicing professionals will find the book useful for self-education on this leading edge technology. The book can also serve as a useful reference as it collects material from a number of fields into one place.

The book begins by surveying the variety of VMs, putting them into perspective and building a framework for discussing VMs. The subsequent chapters describe the major types of VMs in an organized way, emphasizing the common, underlying technologies. Following is a rough outline summarizing each chapter.

In Chapter 1 we introduce the concept of abstraction and define the interfaces that are prevalent in today's computer systems. This is followed by a discussion of virtualization and its relationship to the interfaces. The notion of computer architecture is introduced next, followed by a survey of different types of virtual machines. VMs are shown to fall into two main categories, process virtual machines and system virtual machines. We end the chapter by refining this categorization further and suggesting a taxonomy for virtual machines.

In Chapter 2 we address issues related to the emulation of a source instruction set architecture (ISA) on a target ISA. We demonstrate the workings of a basic interpreter and show how threaded interpretation can help improve performance. The techniques developed are demonstrated using a CISC source ISA, the Intel IA-32, and a RISC target ISA, the IBM PowerPC. We then introduce the notion of binary translation, and discuss the problems of code discovery and code location. This is followed by a discussion of the handling of control transfers. Many ISAs have special features (in some cases, they might be called "quirks") that must be

handled in special ways. These are discussed next. The chapter is rounded out with a case study of emulation in the Shade simulation system.

Chapter 3 discusses the implementation of process virtual machines. A process virtual machine supports an individual guest application program on a host platform consisting of an operating system and underlying hardware. We discuss the implications of VM compatibility and show how the state of a machine, consisting of the register state and the memory state, is mapped and maintained in a process virtual machine. We address the issues of self-modifying code and of protecting the region of memory used by the VM runtime software. VM emulation consists of two parts. First, the emulation of the instruction set is discussed, with reference to interpretation and binary translation discussed in Chapter 2. This leads to a discussion of code caching techniques. Next, the emulation of the interface to the host operating system is addressed. We end the chapter by describing the FX!32 system, which embodies many of the fundamental ideas discussed in the chapter.

Chapter 4 focuses on techniques for the optimization of translated code for better emulation performance. It discusses the framework needed to perform such optimizations and the profile information that must be gathered at program execution time in order to facilitate optimizations. Various profiling techniques are discussed. Because optimizations often work better on larger code blocks, the concepts of dynamic basic blocks, superblocks, traces, and tree groups are introduced. The chapter includes an extensive discussion on code re-ordering and its limitations. Various types of code optimizations, both local and inter-block, are presented. The chapter concludes with a case-study of Dynamo, a dynamic binary optimizer, which applies optimization techniques in a system where the source and target ISAs are identical.

Chapter 5 introduces high-level language virtual machines and traces the transition from the early Pascal P-code VMs to object-oriented VMs. The emphasis in this chapter is on the architecture of high-level language VMs, especially those features supporting object-oriented programming and security. The two important object-oriented VMs of today, namely the Java Virtual Machine and the Microsoft CLI, are described. The features of their bytecode, stack-oriented instruction sets are described. In both cases, the description of the instruction set is supplemented by a discussion of the overall platform that augments the virtual machines with a set of libraries and APIs.

Chapter 6 continues the discussion of high-level language VMs by focusing on their implementation. As in the preceding chapter, more attention is given to Java because of its widespread usage and the variety in its implementations. Two aspects given special consideration are security and memory management. the importance of garbage collection is discussed along with techniques for performing garbage collection. The interaction of Java objects with programs written natively outside the Java environment is discussed next. We then describe how the performance of Java can be enhanced through optimizations of code using techniques described in Chapter 4 as well as new techniques that are specific to the object-oriented paradigm. The concepts in the chapter are brought together using a case-study of the Jikes Research Virtual Machine.

In Chapter 7 we discuss co-designed virtual machines where a conventional ISA is implemented through a combination of an implementation-specific ISA and translation software that runs in concealed memory. We discuss techniques for mapping the state of the original ISA onto the implementation ISA and for maintaining the code cache containing translated code. Various sticky aspects, including the implementation of precise interrupts and page faults, are also discussed. We end the chapter with two case studies: the Transmeta Crusoe processor and the IBM AS/400 processor.

Chapter 8 deals with the classic system virtual machines. A system virtual machine supports a complete guest operating system and all its applications on a host platform. We provide a motivation for these virtual machines and outline the basic ways for implementing system virtual machines, including native and hosted VMs. We discuss techniques for the virtualization of the three main system resources: processors, memory, and I/O. The conditions for virtualizability of a processor, as first enunciated by Popek and Goldberg in the early ’70s, are developed. Also discussed are techniques for virtualization when these conditions are not satisfied by an ISA. Memory virtualization is discussed with attention given both to systems with architected page tables and architected TLBs. Then virtualization is discussed for a variety of I/O devices. We next turn our attention to hardware assists to improve the performance of VM systems with the IBM z/VM as a running example. We end the chapter with two case studies, that of a hosted VM system developed by VMware, and that of the VT-x (Vanderpool) technology developed by Intel for their IA-32 architecture.

In Chapter 9 we shift our attention to the virtualization of multiprocessor systems. We introduce the notion of system partitioning and develop a taxonomy for different types of partitioning. We then discuss the principles behind physical partitioning and logical partitioning. The IBM LPAR feature is presented as a case study in logical partitioning. Following this is a discussion about logical partitioning using hypervisors. We then turn to a system VM-based approach to multiprocessor virtualization using a research system, Cellular Disco, as a case study. We end the chapter with a discussion of multiprocessor virtualization where the guest and host platforms have different ISAs, with special attention on bridging the differences in memory models between a host and a guest.

Chapter 10 is a discussion of emerging applications for virtual machine technology. We focus on three application areas which we feel will be important in the coming years. The first is security. We discuss the vulnerability to attacks of modern computer systems and outline the principles behind intrusion detection systems. The potential of VM technology in protecting and recovering from attacks is discussed. The role of binary rewriting technology in security is also discussed with reference to the RIO system. The second application we focus on is that of migrating computing environments from one machine to another. The techniques used in two systems, the Internet Suspend/Resume system and the Stanford Collective system, are described. The commercial application of this technology in VMware's VMotion is also discussed. Our third emerging application is computational grids. We outline the motivation behind the emergence of the grid as a computing infrastructure and compare it to the motivations behind other types of virtual machines. We end the chapter by showing how classic system virtual machines are proving to be an important enabler for emerging grid systems.

The Appendix is essentially a condensed course in computer systems, providing background material for the main chapters. It discusses the roles of the processor, memory, and I/O in a computer system. This is followed by a discussion of ISAs, including support for user applications as well as for the operating system. Page tables and TLBs are discussed next. We follow this with a discussion of the major components of an operating system and the system call interface between an application and the operating system. Finally we discuss multiprocessor architectures, including cluster architectures and shared-memory multiprocessor systems. Memory coherence and memory consistency issues in shared-memory systems are also addressed.

The book may be used in a course in a variety of ways. Overall, the book is structured for a course focused on virtual machines as a topic in itself (an approach we advocate). For an operating system oriented treatment of virtual machines, an instructor can go straight to Chapters 8 through 10 after introducing the taxonomy of virtual machines in Chapter 1. Chapters 2 through 5 can then be discussed to get an idea of implementation details. A more hardware oriented course can, however, go through Chapters 1 through 4 and then skip Chapters 5 and 6, before covering the remaining chapters. A language-oriented course can go straight to Chapter 5 from Chapter 1, and then backtrack to do Chapters 2 through 4, ending with Chapter 6 to put everything together. Chapter 10 should be of interest to virtually any course using the material in the book.

The specific interests of practitioners will largely determine the order in which they cover the material in the book. We have written the book in such a way that an interested reader can start at the beginning of any chapter of interest and follow the material in the complete chapter with only occasional excursions to sections in other chapters referred to in the text.

There are many people we would like to thank for having made this book possible. We would particularly like to thank the many reviewers. Michael Hind of IBM Research, Jan Hoogerbrugge of Philips Research, Jim Larus of Microsoft Research, Tim Lindholm of Sun Microsystems, Bernd Mathiske of Sun Microsystems, and Mike Smith of Harvard University patiently went through the text of the entire book and provided us with valuable feedback, sometimes critical, but always useful. We also thank a number of reviewers who went through specific chapters or sets of chapters and provided us with their valuable insights and comments. These reviewers include Erik Altman, Peter Capek, Evelyn Duesterwald, and Michael Gschwind, all of IBM Research, Renato Figueiredo of the Univ. of Florida, Michael Franz of UC Irvine, Wei Hsu of the Univ. of Minnesota, Toni Juan of UPC-Barcelona, Alain Kägi of Intel, Beng-Hong Lim of VMware, Eliot Moss of Univ. of Massachusetts, Amherst, Frank Soltis of IBM Rochester, Richard Uhlig of Intel, Romney White of IBM Endicott, Wayne Wolf of Princeton University, and Ben Zorn of Microsoft Research. We also appreciate the discussions with Vas Bala, Ek Ekanadham, Wolfram Sauer, and Charles Webb of IBM, on various aspects of virtualization.

The authors would also like to acknowledge Sriram Vajapeyam for his contributions during the early development of this material, and the students at the

University of Wisconsin-Madison and Universitat Politècnica de Catalunya in Barcelona for their valuable feedback while participating in VM courses and conducting VM research. At the risk of omitting someone, the past and current students who have been particularly helpful are Nidhi Aggarwal, Todd Bezenek, Jason Cantin, Wooseok Chang, Ashutosh Dhodapkar, Timothy Heil, Shiliang Hu, Tejas Karkhanis, Ho-Seop Kim, Kyle Nesbit, and Subramanya Sastry.

This book owes a lot to the guidance, persistence, and encouragement provided by our publisher, Denise Penrose, and the support provided by her excellent staff at Morgan-Kaufmann Publishers, including Kimberlee Honjo, Angela Dooley, Alyson Day, and Summer Block.

First author: I would like to thank the people at IBM Research, and Dan Prener in particular, for their support during my 2000-2001 sabbatical—the time this book had its genesis. I am especially grateful to Erik Altman for being a sounding board throughout the writing of the book. I also thank my graduate students for their support and helpful suggestions. Finally, I am grateful to my children Barbara, Carolyn, and Jim, for their encouragement and patience during the writing of the book, and in general for putting up with a frequently distracted father.

Second author: I would like to thank Dan Prener, Eric Kronstadt, and Jaime Moreno for their encouragement and support in undertaking this project. Thanks also to Peter Capek, Dan Prener, Peter Oden, Dick Attanasio, and Mark Mergen for many interesting tea-time discussions. Finally, I would like to thank my wife, Indira, and my daughters, Rohini and Nandini, for their love and understanding at all times; they have given me more than I could have ever hoped for or imagined.

The authors are mutually grateful for the opportunity to renew a friendship that stretches back 30 years. We have had tremendous fun and have learnt a great deal in the process of writing this book. If you, the reader, experience just a fraction of what we have experienced, this book will have been worthwhile.

James E. Smith

Ravi Nair

Table of Contents

Instructions for online access

Cover

Title Page

Dedication

Copyright

Foreword

Preface

Chapter 1: Introduction to Virtual Machines

1.1 Computer Architecture

1.1 Computer Architecture

1.3 Process Virtual Machines

1.4 System Virtual Machines

1.5 A Taxonomy

1.6 Summary: The Versatility of Virtual Machines

1.7 The Rest of the Book

Chapter 2: Emulation

Chapter 2: Emulation

2.1 Basic Interpretation

2.2 Threaded Interpretation

2.3 Predecoding and Direct Threaded Interpretation

2.4 Interpreting a Complex Instruction Set

2.5 Binary Translation

2.6 Code Discovery and Dynamic Translation

2.7 Control Transfer Optimizations

2.8 Instruction Set Issues

2.9 Case Study: Shade and the Role of Emulation During Simulation

2.10 Summary: Performance Tradeoffs

Chapter 3: Process Virtual Machines

3.1 Virtual Machine Implementation

3.2 Compatibility

3.3 State Mapping

3.4 Memory Architecture Emulation

3.5 Instruction Emulation

3.6 Exception Emulation

3.7 Operating System Emulation

3.8 Code Cache Management

3.9 System Environment

3.10 Case Study: FX!32

3.11 Summary

Chapter 4: Dynamic Binary Optimization

4.1 Dynamic Program Behavior

4.2 Profiling

4.3 Optimizing Translation Blocks

4.4 Optimization Framework

4.5 Code Reordering

4.6 Code Optimizations

4.7 Same-ISA Optimization Systems: Special-Case Process Virtual Machines

4.8 Summary

Chapter 5: High-Level Language Virtual Machine Architecture

5.1 The Pascal P-Code Virtual Machine

5.2 Object-Oriented High-Level Language Virtual Machines

5.3 The Java Virtual Machine Architecture

5.4 Completing the Platform: APIs

5.5 The Microsoft Common Language Infrastructure: A Flexible High-Level Language Virtual Machine

5.6 Summary: Virtual ISA Features

Chapter 6: High-Level Language Virtual Machine Implementation

6.1 Dynamic Class Loading

6.2 Implementing Security

6.3 Garbage Collection

6.4 Java Native Interface

6.5 Basic Emulation

6.6 High-Performance Emulation

6.7 Case Study: The Jikes Research Virtual Machine

6.8 Summary

Chapter 7: Codesigned Virtual Machines

7.1 Memory and Register State Mapping

7.2 Self-Modifying and Self-Referencing Code

7.3 Support for Code Caching

7.4 Implementing Precise Traps

7.5 Input/Output

7.6 Applying Codesigned Virtual Machines

7.7 Case Study: Transmeta Crusoe

7.8 Case Study: IBM AS/400

7.9 Summary

Chapter 8: System Virtual Machines

8.1 Key Concepts

8.2 Resource Virtualization—Processors

8.3 Resource Virtualization—Memory

8.4 Resource Virtualization—Input/Output

8.5 Performance Enhancement of System Virtual Machines

8.6 Case Study: VMware Virtual Platform

8.7 Case Study: The Intel VT-x (Vanderpool) Technology

8.8 Summary

Chapter 9: Multiprocessor Virtualization

9.1 Partitioning of Multiprocessor Systems

9.2 Physical Partitioning

9.3 Logical Partitioning

9.4 Case Study: Cellular Disco System Virtual Machine-Based Partitioning

9.5 Virtualization with Different Host and Guest ISAs

9.6 Summary

Chapter 10: Emerging Applications

10.1 Security

10.2 Migration of Computing Environments

10.3 Grids: Virtual Organizations

10.4 Summary

Appendix A: Real Machines

A.1 Computer System Hardware

A.2 The User ISA: Computation

A.3 The System ISA: Resource Management

A.4 Operating System Organization

A.5 The Operating System Interface

A.6 System Initialization

A.7 Multiprocessor Architecture

A.8 Example Instruction Set Architectures

References

Index

Previous chapter
< Preface

Next chapter
Chapter 1: Introduction to Virtual Machines >

Chapter One

Introduction to Virtual Machines

Modern computers are among the most advanced human-engineered structures, and they are possible only because of our ability to manage extreme complexity. Computer systems contain many silicon chips, each with hundreds of millions of transistors. These are interconnected and combined with high-speed input/output (I/O) devices and networking infrastructure to form the platforms upon which software can operate. Operating systems, application programs and libraries, and graphics and networking software all cooperate to provide a powerful environment for data management, education, communication, entertainment, and many other applications.

The key to managing complexity in computer systems is their division into *levels of abstraction* separated by *well-defined interfaces.* Levels of abstraction allow implementation details at lower levels of a design to be ignored or simplified, thereby simplifying the design of components at higher levels. The details of a hard disk, for example, that it is divided into sectors and tracks, are abstracted by the operating system (OS) so that the disk appears to application software as a set of variable-size files (see **Figure 1.1**). An application programmer can then create, write, and read files, without knowledge of the way the hard disk is constructed and organized.

The levels of abstraction are arranged in a hierarchy, with lower levels implemented in hardware and higher levels in software. In the hardware levels, all the components are physical, have real properties, and their interfaces are defined so that the various parts can be physically connected. In the software levels, components are logical, with fewer restrictions based on physical characteristics. In this

with fewer restrictions based on physical characteristics. In this book, we are most concerned with the abstraction levels that are at or near the hardware/software boundary. These are the levels where software is separated from the *machine* on which it runs.

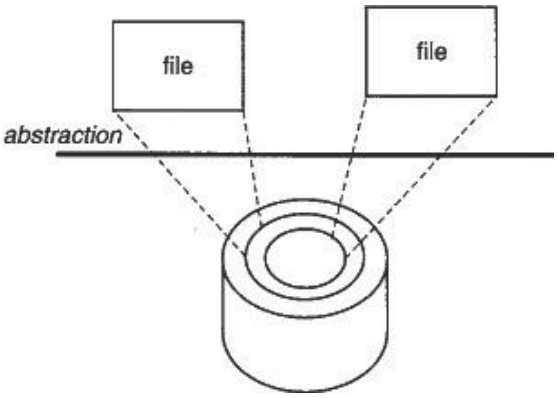


Figure 1.1 Files Are an Abstraction of a Disk. *A level of abstraction provides a simplified interface to underlying resources.*

Computer software is executed by a *machine* (a term that dates back to the beginning of computers; *platform* is the term more in vogue today). From the perspective of the operating system, a machine is largely composed of hardware, including one or more processors that run a specific instruction set, some real memory, and I/O devices. However, we do not restrict the use of the term *machine* to just the hardware components of a computer. From the perspective of application programs, for example, the machine is a combination of the operating system and those portions of the hardware accessible through user-level binary instructions.

Let us now turn to the other aspect of managing complexity: the use of *well-defined interfaces*. Well-defined interfaces allow computer design tasks to be decoupled so that teams of hardware and software designers can work more or less independently. The instruction set is one such interface. For example, designers at Intel and AMD develop microprocessors that implement the Intel IA-32[1] instruction set, while software engineers at Microsoft develop compilers that map high-level languages to the same instruction set. As long as both groups satisfy the instruction set specification, com-

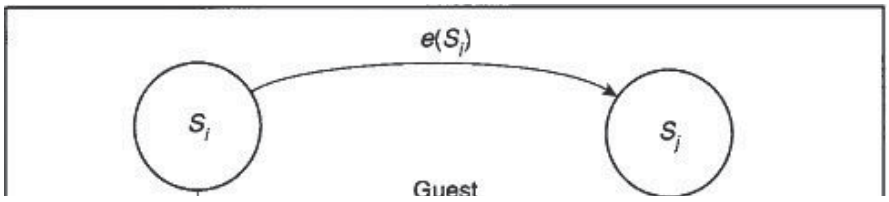
piled software will execute correctly on a machine incorporating an IA-32 microprocessor. The operating system interface, defined as a set of function calls, is another important standardized interface in computer systems. As the Intel/Microsoft example suggests, well-defined interfaces permit development of interacting computer subsystems at different companies and at different times, sometimes years apart. Application software developers do not need to be aware of detailed changes inside the operating system, and hardware and software can be upgraded according to different schedules. Software can run on different platforms implementing the same instruction set.

Despite their many advantages, well-defined interfaces can also be confining. Subsystems and components designed to specifications for one interface will not work with those designed for another. There are processors with different instruction sets (e.g., Intel IA-32 and IBM PowerPC), and there are different operating systems (e.g., Windows and Linux). Application programs, when distributed as program binaries, are tied to a specific instruction set and operating system. An operating system is tied to a computer that implements specific memory system and I/O system interfaces. As a general principle, diversity in instruction sets, operating systems, and application programming languages encourages innovation and discourages stagnation. However, in practice, diversity also leads to reduced interoperability, which becomes restrictive, especially in a world of networked computers, where it is advantageous to move software as freely as data.

Beneath the hardware/software interface, hardware resource considerations can also limit the flexibility of software systems. Memory and I/O abstractions, both in high-level languages and in operating systems, have removed many hardware resource dependences; some still remain, however. Many operating systems are developed for a specific system architecture, e.g., for a uniprocessor or a shared-memory multiprocessor, and are designed to manage

a shared-memory multiprocessor, and are designed to manage hardware resources directly. The implicit assumption is that the hardware resources of a system are managed by a single operating system. This binds all hardware resources into a single entity under a single management regime. And this, in turn, limits the flexibility of the system, not only in terms of available application software (as discussed earlier), but also in terms of security and failure isolation, especially when the system is shared by multiple users or groups of users.

Virtualization provides a way of relaxing the foregoing constraints and increasing flexibility. When a system (or subsystem), e.g., a processor, memory, or I/O device, is *virtualized,* its interface and all resources visible through the interface are mapped onto the interface and resources of a real system actually implementing it. Consequently, the real system is transformed so that it appears to be a different, virtual system or even a set of multiple virtual systems. Formally, virtualization involves the construction of an isomorphism that maps a virtual *guest* system to a real *host* (**Popek and Goldberg 1974**). This isomorphism, illustrated in **Figure 1.2**, maps the guest state to the host state (function *V* in **Figure 1.2**), and for a sequence of operations, *e,* that modifies the state in the guest (the function *e* modifies state S_i to state S_j) there is a corresponding sequence of operations *e*' in the host that performs an equivalent modification to the host's state (changes S_i' to S_j'). Although such an isomorphism can be used to characterize abstraction as well as virtualization, we distinguish the two: Virtualization differs from abstraction in that virtualization does not necessarily hide details; the level of detail in a virtual system is often the same as that in the underlying real system.



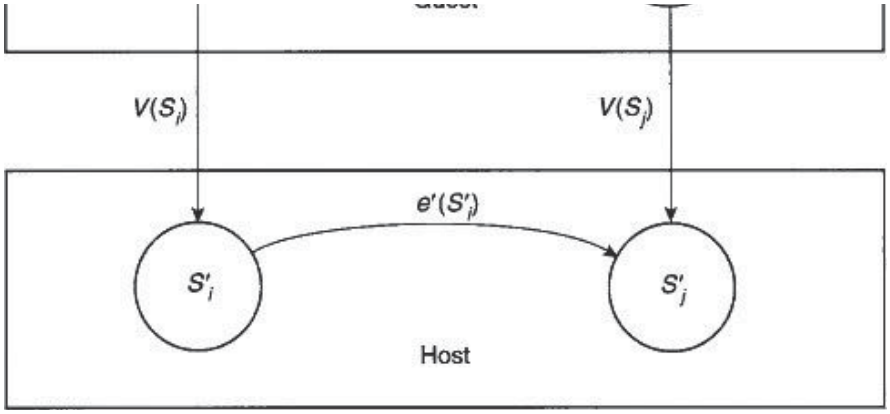


Figure 1.2 Virtualization. *Formally, virtualization is the construction of an isomorphism between a guest system and a host; e' o V(S$_i$) = V o e(S$_i$).*

Consider again the example of a hard disk. In some applications, it may be desirable to partition a single large hard disk into a number of smaller virtual disks. The virtual disks are mapped to a real disk by implementing each of the virtual disks as a single large file on the real disk (**Figure 1.3**). Virtualizing software provides a mapping between virtual disk contents and real disk contents (the function *V* in the isomorphism), using the file abstraction as an intermediate step. Each of the virtual disks is given the appearance of having a number of logical tracks and sectors (although fewer than in the large disk). A write to a virtual disk (the function *e* in the isomorphism) is mirrored by a file write and a corresponding real disk write, in the host system (the function *e'* in the isomorphism). In this example, the level of detail provided at the virtual disk interface, i.e., sector/track addressing, remains the same as for a real disk; no abstraction takes place.

The concept of virtualization can be applied not only to subsystems such as disks, but to an entire machine. A *virtual machine* (VM) is implemented by adding a layer of software to a real machine to support the desired virtual machine's architecture. For example, virtualizing software installed on an Apple Macintosh can provide a Windows/IA-32 virtual machine capable of running PC application programs. In general, a virtual machine can circumvent real machine compatibility constraints and hardware resource constraints

to enable a higher degree of software portability and flexibility.

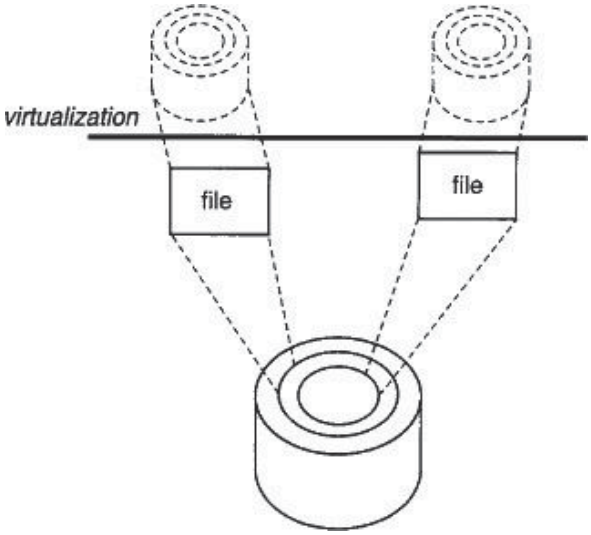


Figure 1.3 Implementing Virtual Disks. *Virtualization provides a different interface and/or resources at the same level of abstraction.*

A wide variety of virtual machines exist to provide an equally wide variety of benefits. Multiple, replicated virtual machines can be implemented on a single hardware platform to provide individuals or user groups with their own operating system environments. The different system environments (possibly with different operating systems) also provide isolation and enhanced security. A large multiprocessor server can be divided into smaller virtual servers, while retaining the ability to balance the use of hardware resources across the system.

Virtual machines can also employ emulation techniques to support cross-platform software compatibility. For example, a platform implementing the PowerPC instruction set can be converted into a virtual platform running the IA-32 instruction set. Consequently, software written for one platform will run on the other. This compatibility can be provided either at the system level (e.g., to run a Windows OS on a Macintosh) or at the program or process level (e.g., to run Excel on a Sun Solaris/SPARC platform). In addition to emulation, virtual machines can provide dynamic, on-the-fly optimization of program binaries. Finally, through emulation, virtual

mization of program binaries. Finally, through emulation, virtual machines can enable new, proprietary instruction sets, e.g., incorporating very long instruction words (VLIWs), while supporting programs in an existing, standard instruction set.

The virtual machine examples just described are constructed to match the architectures of existing real machines. However, there are also virtual machines for which there are no corresponding real machines. It has become common for language developers to invent a virtual machine tailored to a new high-level language. Programs written in the high-level language are compiled to "binaries" targeted at the virtual machine. Then any real machine on which the virtual machine is implemented can run the compiled code. The power of this approach has been clearly demonstrated with the Java high-level language and the Java virtual machine, where a high degree of platform independence has been achieved, thereby enabling a very flexible network computing environment.

Virtual machines have been investigated and built by operating system developers, language designers, compiler developers, and hardware designers. Although each application of virtual machines has its unique characteristics, there also are underlying concepts and technologies that are common across the spectrum of virtual machines. Because the various virtual machine architectures and underlying technologies have been developed by different groups, it is especially important to unify this body of knowledge and understand the base technologies that cut across the various forms of virtual machines. The goals of this book are to describe the family of virtual machines in a unified way, to discuss the common underlying technologies that support them, and to demonstrate their versatility by exploring their many applications.

1.1 Computer Architecture

As will become evident, a discussion of virtual machines is also a

As will become evident, a discussion of virtual machines is also a discussion about computer architecture in a broad sense. Virtual machines often bridge architectural boundaries, and a major consideration in constructing a virtual machine is the fidelity with which a virtual machine implements architected interfaces. Therefore, it is useful to define and summarize *computer architecture.*

The architecture of a building describes its functionality and appearance to users of the building but not the details of its construction, such as the specifics of its plumbing system or the manufacturer of the bricks. Analogously, the term *architecture,* when applied to computers, refers to the functionality and appearance of a computer system or subsystem but not the details of its implementation. The architecture is often formally described through a specification of an interface and the logical behavior of resources manipulated via the interface. The term *implementation* will be used to describe the actual embodiment of an architecture. Any architecture can have several implementations, each one having distinct characteristics, e.g., a high-performance implementation or a low-power implementation.

The levels of abstraction in a computer system correspond to *implementation layers* in both hardware and software. There is an architecture for each of these implementation layers (although the term *architecture* is not always used). **Figure 1.4** shows some of the important interfaces and implementation layers as found in a typical computer system. In software, for example, there is an interface between an application program and standard libraries (interface 2 in **Figure 1.4**). Another software interface is at the boundary of the operating system (interface 3). The interfaces in hardware include an I/O architecture that describes the signals that drive I/O device controllers (interface 11), a hardware memory architecture that describes the way addresses are translated (interface 9), an interface for the memory access signals that leave the processor (interface

12), and another for the signals that reach the DRAM chips in memory (interface 14). The OS communicates with I/O devices through a sequence of interfaces: 4, 8, 10, 11, and 13. Of these interfaces and architectures, we are most interested in those at or near the hardware/software boundary.

The *instruction set architecture* (ISA), which marks the division between hardware and software, is composed of interfaces 7 and 8 in **Figure 1.4**. The concept of an ISA was first clearly articulated when the IBM 360 family of mainframe computers was developed in the early 1960s (**Amdahl, Blaauw, and Brooks 1964**). With that project, the importance of software compatibility was fully recognized. The IBM 360 series had a number of models that could incorporate a wide range of hardware resources, thereby covering a broad spectrum of price and performance levels—yet they were designed to run the same software. To successfully accomplish this, the interface between the hardware and software had to be precisely defined and controlled, and the ISA serves this purpose.

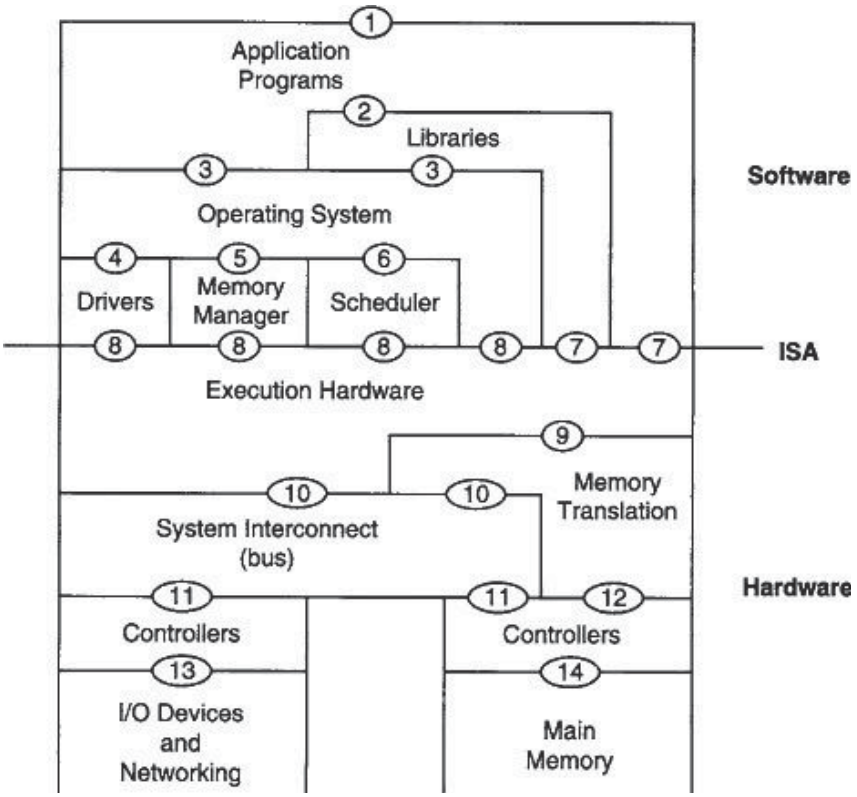




Figure 1.4 Computer System Architectures. *Implementation layers communicate vertically via the shown interfaces. This view of architecture is styled after one given by Glenford Myers (1982).*

There are two parts of an ISA that are important in the definition of virtual machines. The first part includes those aspects of the ISA that are visible to an application program. This will be referred to as the *user ISA.* The second part includes those aspects visible to supervisor software, such as the operating system, which is responsible for managing hardware resources. This is the *system ISA.* Of course, the supervisor software can also employ all the elements of the user ISA. In **Figure 1.4**, interface 7 consists of the user ISA only, and interface 8 consists of both the user ISA and the system ISA.

In this book we will also be concerned with interfaces besides the ISA. The application binary interface (ABI) consists of interfaces 3 and 7 in **Figure 1.4**. An important related interface is the application program interface (API), which consists of interfaces 2 and 7.

The *application binary interface,* which provides a program with access to the hardware resources and services available in a system, has two major components. The first is the set of all user instructions (interface 7 in **Figure 1.4**); system instructions are not included in the ABI. At the ABI level, all application programs interact with the shared hardware resources indirectly, by invoking the operating system via a *system call* interface (interface 3 in **Figure 1.4**), which is the second component of the ABI. System calls provide a specific set of operations that an operating system may perform on behalf of a user program (after checking to make sure the user program should be granted its request). The system call interface is typically implemented via an instruction that transfers control to the operating system in a manner similar to a procedure or subroutine call, except the call target address is restricted to a specific address in the operating system. Arguments for the system call are typically passed through registers and/or a stack held in memory, following

specific conventions that are part of the system call interface. A program binary compiled to a specific ABI can run unchanged only on a system with the same ISA and operating system.

The *application programming interface* is usually defined with respect to a high-level language (HLL). A key element of an API is a standard library (or libraries) that an application calls to invoke various services available on the system, including those provided by the operating system. An API, typically defined at the source code level, enables applications written to the API to be ported easily (via recompilation) to any system that supports the same API. The API specifies an abstraction of the details of implementation of services, especially those that involve privileged hardware. For example, clib is a well-known library that supports the UNIX/C programming language. The clib API provides a memory model consisting of text (for code) and a heap and stack (for data). A routine belonging to an API typically contains one or more ABI-level operating system calls. Some API library routines are simply *wrappers,* i.e., code that translates directly from the HLL calling convention to the binary convention expected by the OS. Other API routines are more complex and may include several OS calls.

1.1 Computer Architecture

To understand what a virtual machine is, we must first discuss what is meant by *machine,* and, as pointed out earlier, the meaning of "machine" is a matter of perspective. From the perspective of a *process* executing a user program, the machine consists of a logical memory address space that has been assigned to the process, along with user-level registers and instructions that allow the execution of code belonging to the process. The I/O part of the machine is visible only through the operating system, and the only way the process can interact with the I/O system is via operating system calls, often through libraries that execute as part of the process. Processes are

usually transient in nature (although not always). They are created, execute for a period of time, perhaps spawn other processes along the way, and eventually terminate. To summarize, the machine, from the prospective of a process, is a combination of the operating system and the underlying user-level hardware. The ABI provides the interface between the process and the machine (**Figure 1.5**a).

From the perspective of the operating system, an entire *system* is supported by the underlying machine. A system is a full execution environment that can simultaneously support a number of processes potentially belonging to different users. All the processes share a file system and other I/O resources. The system environment persists over time (with occasional reboots) as processes come and go. The system allocates physical memory and I/O resources to the processes and allows the processes to interact with their resources via an OS that is part of the system. Hence, the machine, from the perspective of a system, is implemented by the underlying hardware alone, and the ISA provides the interface between the system and the machine (**Figure 1.5**b).

In practical terms, a virtual machine executes software (either an individual process or a full system, depending on the type of machine) in the same manner as the machine for which the software was developed. The virtual machine is implemented as a combination of a real machine and virtualizing software. The virtual machine may have resources different from the real machine, either in quantity or in type. For example, a virtual machine may have more or fewer processors than the real machine, and the processors may execute a different instruction set than does the real machine. It is important to note that equivalent performance is usually not required as part of virtualization; often a virtual machine provides less performance than an equivalent real machine running the same software, i.e., software developed for the real machine.



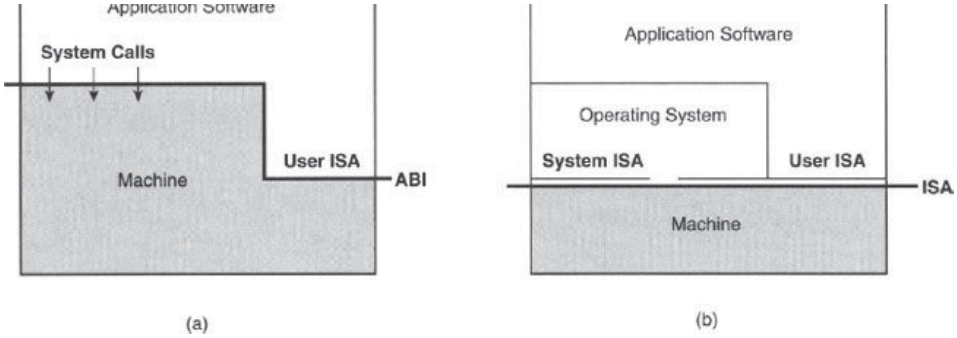


Figure 1.5 Machine Interfaces. *(a) Application binary interface (ABI); (b) instruction set architecture (ISA) interface.*

As characterized by the isomorphism described earlier, the process of virtualization consists of two parts: (1) the mapping of virtual resources or state, e.g., registers, memory, or files, to real resources in the underlying machine and (2) the use of real machine instructions and/or system calls to carry out the actions specified by virtual machine instructions and/or system calls, e.g., emulation of the virtual machine ABI or ISA.

Just as there is a process perspective and a system perspective of machines, there are also process-level and system-level virtual machines. As the name suggests, a *process virtual machine* is capable of supporting an individual process. A process virtual machine is illustrated in **Figure 1.6**. In this figure and the figures that follow, compatible interfaces are illustrated graphically as "meshing" boundaries. In process VMs, the virtualizing software is placed at the ABI interface, on top of the OS/hardware combination. The virtualizing software emulates both user-level instructions and operating system calls.

With regard to terminology (see **Figure 1.6**), we usually refer to the underlying platform as the host and to the software that runs in the VM environment as the *guest*. The real platform that corresponds to a virtual machine, i.e., the real machine being emulated by the virtual machine, is referred to as the native machine. The name given to the virtualizing software depends on the type of virtual machine being implemented. In process VMs, virtualizing software is often

referred to as the *runtime,* which is short for "runtime software."[2] The runtime is created to support a guest process and runs on top of an operating system. The VM supports the guest process as long as the guest process executes and terminates support when the guest process terminates.

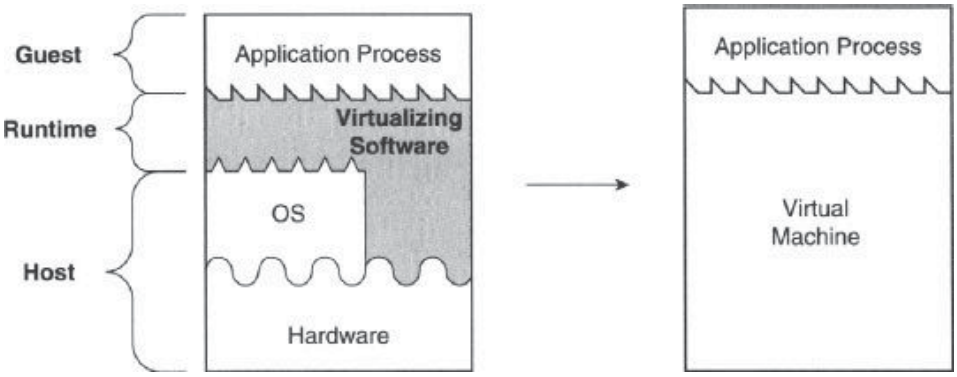


Figure 1.6 A Process Virtual Machine. *Virtualizing software translates a set of OS and user-level instructions composing one platform to another, forming a process virtual machine capable of executing programs developed for a different OS and a different ISA.*

In contrast, a *system virtual machine* provides a complete system environment. This environment can support an operating system along with its potentially many user processes. It provides a guest operating system with access to underlying hardware resources, including networking, I/O, and, on the desktop, a display and graphical user interface. The VM supports the operating system as long as the system environment is alive.

A system virtual machine is illustrated in **Figure 1.7**; virtualizing software is placed between the underlying hardware machine and conventional software. In this particular example, virtualizing software emulates the hardware ISA so that conventional software "sees" a different ISA than the one supported by hardware. In many system VMs the guest and host run the same ISA, however. In system VMs, the virtualizing software is often referred to as the *virtual machine monitor* (VMM), a term coined when the VM concept was first developed in the late 1960s.

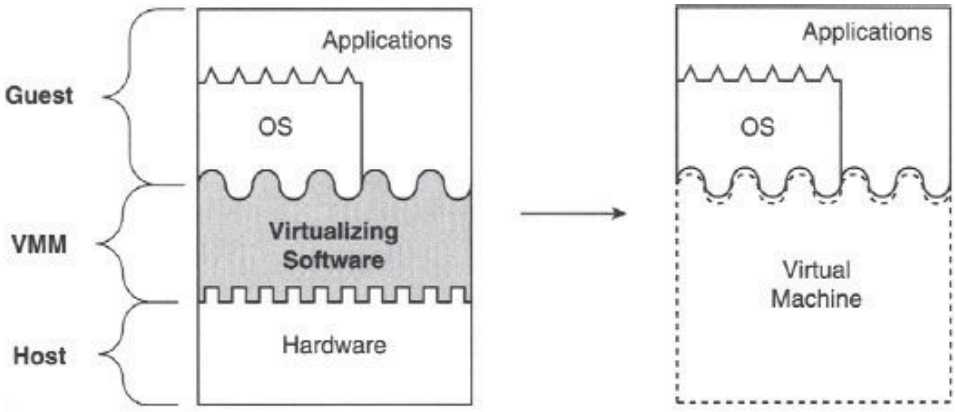


Figure 1.7 A System Virtual Machine. *Virtualizing software translates the ISA used by one hardware platform to another, forming a system virtual machine, capable of executing a system software environment developed for a different set of hardware.*

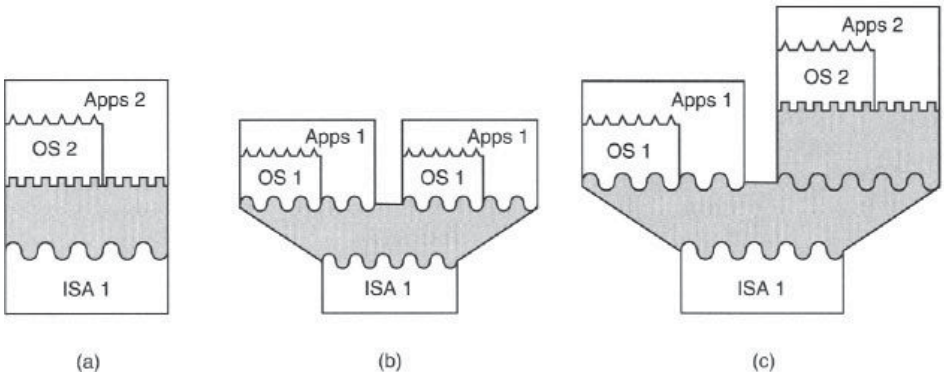


Figure 1.8 Examples of Virtual Machine Applications. *(a) Emulating one instruction set with another; (b) replicating a virtual machine so that multiple operating systems can be supported simultaneously; (c) composing virtual machine software to form a more complex, flexible system.*

Virtualizing software can be applied in several ways to connect and adapt the major computer subsystems (see **Figure 1.8**). *Emulation* adds considerable flexibility by permitting "mix and match" cross-platform software portability. In this example (**Figure 1.8**a), one ISA is emulated by another. Virtualizing software can enhance emulation with *optimization,* by taking implementation-specific information into consideration as it performs emulation. Virtualizing software can also provide resource *replication,* for example, by giving a single hardware platform the appearance of multiple platforms (**Figure 1.8**b), each capable of running a complete operating system and/or a set of applications. Finally, the virtual machine functions can be composed (**Figure 1.8**c) to form a wide variety of architec-

can be composed (Figure 1.8c) to form a wide variety of architectures, freed of many of the traditional compatibility and resource constraints.

We are now ready to describe some specific types of virtual machines. These span a fairly broad spectrum of applications, and we will discuss them according to the two main categories: process VMs and system VMs. Note that because the various virtual machines have been developed by different design communities, different terms are often used to describe similar concepts and features. In fact, it is sometimes the practice to use some term other than *virtual machine* to describe what is in reality a form of virtual machine.

1.3 Process Virtual Machines

Process-level VMs provide user applications with a virtual ABI environment. In their various implementations, process VMs can provide replication, emulation, and optimization. The following subsections describe each of these.

1.3.1 Multiprogramming

The first and most common virtual machine is so ubiquitous that we don't even think of it as being a virtual machine. The combination of the OS call interface and the user instruction set forms the machine that executes a user process. Most operating systems can simultaneously support multiple user processes through multiprogramming, where each user process is given the illusion of having a complete machine to itself. Each process is given its own address space and is given access to a file structure. The operating system time-shares the hardware and manages underlying resources to make this possible. In effect, the operating system provides a replicated process-level virtual machine for each of the concurrently executing applications.

1.3.2 Emulators and Dynamic Binary Translators

A more challenging problem for process-level virtual machines is to support program binaries compiled to a different instruction set than the one executed by the host's hardware, i.e., to *emulate* one instruction set on hardware designed for another. An example emulating process virtual machine is illustrated in **Figure 1.9**. Application programs are compiled for a *source ISA,* but the hardware implements a different *target ISA.* As shown in the example, the operating system may be the same for both the guest process and the host platform, although in other cases the operating systems may differ as well. The example in **Figure 1.9** illustrates the Digital FX!32 system (**Hookway and Herdeg 1997**). The FX!32 system can run Intel IA-32 application binaries compiled for Windows NT on an Alpha hardware platform also running Windows NT. More recent examples are the Aries system (**Zheng and Thompson 2000**) which supports PA-RISC programs on an IPF (Itanium) platform, and the Intel IA-32 EL (execution layer) which supports IA-32 programs on an IPF platform (**Baraz et al. 2003**).

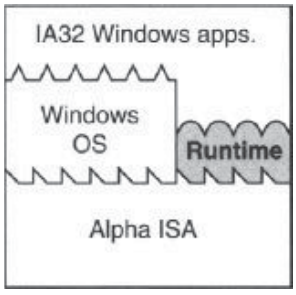


Figure 1.9 A Process VM That Emulates Guest Applications. *The Digital FX!32 system allows Windows IA-32 applications to be run on an Alpha Windows platform.*

The most straightforward emulation method is *interpretation.* An interpreter program executing the target ISA fetches, decodes, and emulates the execution of individual source instructions. This can be a relatively slow process, requiring tens of native target instructions for each source instruction interpreted

tions for each source instruction interpreted.

For better performance, *binary translation* is typically used. With binary translation, blocks of source instructions are converted to target instructions that perform equivalent functions. There can be a relatively high overhead associated with the translation process, but once a block of instructions is translated, the translated instructions can be cached and repeatedly executed much faster than they can be interpreted. Because binary translation is the most important feature of this type of process virtual machine, they are sometimes called *dynamic binary translators.*

Interpretation and binary translation have different performance characteristics. Interpretation has relatively low startup overhead but consumes significant time whenever an instruction is emulated. On the other hand, binary translation has high initial overhead when performing the translations but is fast for each repeated execution. Consequently, some virtual machines use a staged emulation strategy combined with *profiling,* i.e., the collection of statistics regarding the program's behavior. Initially, a block of source instructions is interpreted, and profiling is used to determine which instruction sequences are frequently executed. Then a frequently executed block may be binary translated. Some systems perform additional code optimizations on the translated code if profiling shows that it has a very high execution frequency. In most emulating virtual machines the stages of interpretation and binary translation can both occur over the course of a single program's execution. In the case of FX!32, translation occurs incrementally between program runs.

1.3.3 Same-ISA Binary Optimizers

Most dynamic binary translators not only translate from source to target code but also perform some code optimizations. This leads naturally to virtual machines where the instruction sets used by the

host and the guest are the same, and optimization of a program binary is the primary purpose of the virtual machine. Thus, *same-ISA dynamic binary optimizers* are implemented in a manner very similar to emulating virtual machines, including staged optimization and software caching of optimized code. Same-ISA dynamic binary optimizers are most effective for source binaries that are relatively unoptimized to begin with, a situation that is fairly common in practice. A dynamic binary optimizer can collect a profile and then use this profile information to optimize the binary code on the fly. An example of such a same-ISA dynamic binary optimizer is the Dynamo system, originally developed as a research project at Hewlett-Packard (**Bala, Duesterwald, and Banerjia 2000**).

1.3.4 High-Level Language Virtual Machines: Platform Independence

For the process VMs described earlier, cross-platform portability is clearly a very important objective. For example, the FX!32 system enabled portability of application software compiled for a popular platform (IA-32 PC) to a less popular platform (Alpha). However, this approach allows cross-platform compatibility only on a case-by-case basis and requires a great deal of programming effort. For example, if one wanted to run IA-32 binaries on a number of hardware platforms currently in use, e.g., SPARC, PowerPC, and MIPS, then an FX!32-like VM would have to be developed for each of them. The problem would be even more difficult if the host platforms run different operating systems than the one for which binaries were originally compiled.

Full cross-platform portability is more easily achieved by taking a step back and designing it into an overall software framework. One way of accomplishing this is to design a process-level VM at the same time as an application development environment is being defined. Here, the VM environment does not directly correspond to any real platform. Rather, it is designed for ease of portability and to

any real platform. Rather, it is designed for ease of portability and to match the features of a high-level language (HLL) used for application program development. These *high-level language VMs* (HLL VMs) are similar to the process VMs described earlier. However, they are focused on minimizing hardware-specific and OS-specific features because these would compromise platform independence.

High-level language VMs first became popular with the Pascal programming environment (**Bowles 1980**). In a conventional system, **Figure 1.10**a, the compiler consists of a frontend that performs lexical, syntax, and semantic analysis to generate simple intermediate code—similar to machine code but more abstract. Typically the intermediate code does not contain specific register assignments, for example. Then a code generator takes the intermediate code and generates a binary containing machine code for a specific ISA and OS. This binary file is distributed and executed on platforms that support the given ISA/OS combination. To execute the program on a different platform, however, it must be recompiled for that platform.

In HLL VMs, this model is changed (**Figure 1.10**b). The steps are similar to the conventional ones, but the point at which program distribution takes place is at a higher level. As shown in **Figure 1.10**b, a conventional compiler frontend generates abstract machine code, which is very similar to an intermediate form. In many HLL VMs, this is a rather generic stack-based ISA. This *virtual ISA* is in essence the machine code for a virtual machine. The portable virtual ISA code is distributed for execution on different platforms. For each platform, a VM capable of executing the virtual ISA is implemented. In its simplest form, the VM contains an interpreter that takes each instruction, decodes it, and then performs the required state transformations (e.g., involving memory and the stack). I/O functions are performed via a set of standard library calls that are defined as part of the VM. In more sophisticated, higher performance VMs, the abstract machine code may be compiled (binary translated) into host

machine code for direct execution on the host platform.

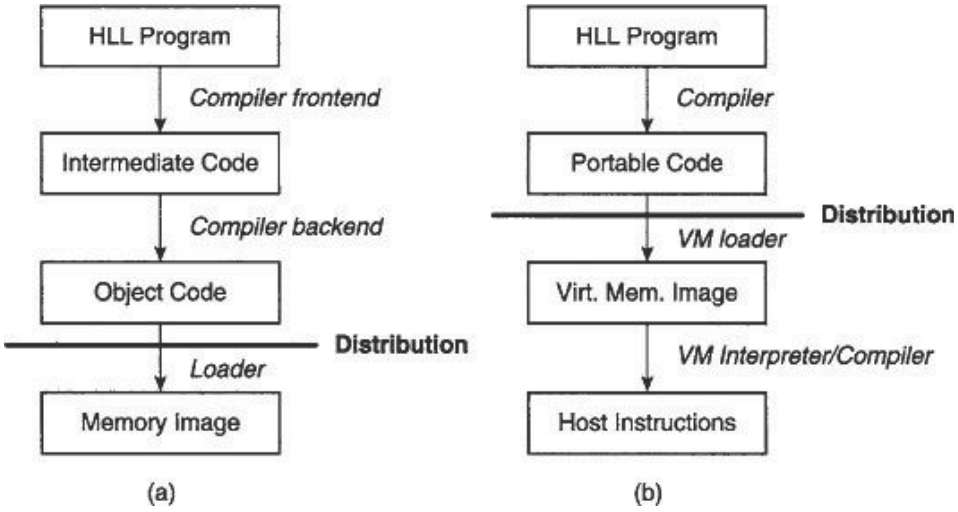


Figure 1.10 High-Level Language Environments. (*a) A conventional system, where platform-dependent object code is distributed; (b) an HLL VM environment, where portable intermediate code is "executed" by a platform-dependent virtual machine.*

An advantage of an HLL VM is that software is easily portable, once the VM is implemented on a target platform. While the VM implementation would take some effort, it is a much simpler task than developing a compiler for each platform and recompiling every application when it is ported. It is also simpler than developing a conventional emulating process VM for a typical real-world ISA.

The Sun Microsystems Java VM architecture (**Lindholm and Yellin 1999**) and the Microsoft common language infrastructure (CLI), which is the foundation of the .NET framework (**Box 2002**), are more recent, widely used examples of HLL VMs. Platform independence and high security are central to both the Java VM and CLI. The ISAs in both systems are based on *bytecodes;* that is, instructions are encoded as a sequence of bytes, where each byte is an opcode, a single-byte operand, or part of a multibyte operand. These byte-code instruction sets are stack based (to eliminate register requirements) and have an abstract data specification and memory model. In fact, the memory size is conceptually unbounded, with garbage collection as an assumed part of the implementation. Because all hardware platforms are potential targets for executing Java- or CLI-

based programs, applications are not compiled for a specific OS. Rather, a set of standard libraries is provided as part of the overall execution environment.

1.4 System Virtual Machines

System virtual machines provide a complete system environment in which many processes, possibly belonging to multiple users, can coexist. These VMs were first developed during the 1960s and early 1970s, and they were the origin of the term *virtual machine.* By using system VMs, a single host hardware platform can support multiple guest OS environments simultaneously. At the time they were first developed, mainframe computer systems were very large and expensive, and computers were almost always shared among a large number of users. Different groups of users sometimes wanted different operating systems to be run on the shared hardware, and VMs allowed them to do so. Alternatively, a multiplicity of single-user operating systems allowed a convenient way of implementing time-sharing among many users. Over time, as hardware became much less expensive and much of it migrated to the desktop, interest in these classic system VMs faded.

Today, however, system VMs are enjoying renewed popularity. This is partly due to modern-day variations of the traditional motivations for system VMs. The large, expensive mainframe systems of the past are now servers or server farms, and these servers may be shared by a number of users or user groups. Perhaps the most important feature of today's system VMs is that they provide a secure way of partitioning major software systems that run concurrently on the same hardware platform. Software running on one guest system is isolated from software running on other guest systems. Furthermore, if security on one guest system is compromised or if the guest OS suffers a failure, the software running on other guest systems is not affected. The ability to support different operating

systems is not affected. The ability to support different operating systems simultaneously, e.g., Windows and Linux (as illustrated in Figure 1.11), is another reason for their appeal, although it is probably of secondary importance to most users.

In system VMs, platform replication is the major feature provided by a VMM. The central problem is that of dividing a single set of hardware resources among multiple guest operating system environments. The VMM has access to and manages all the hardware resources. A guest operating system and application programs compiled for that operating system are then managed under (hidden) control of the VMM. This is accomplished by constructing the system so that when a guest OS performs certain operations, such as a privileged instruction that directly involves the shared hardware resources, the operation is intercepted by the VMM, checked for correctness, and performed by the VMM on behalf of the guest. Guest software is unaware of the "behind-the-scenes" work performed by the VMM.

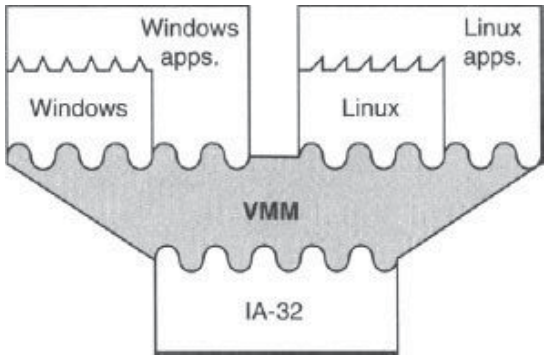


Figure 1.11 A System VM That Supports Multiple OS Environments on the Same Hardware.

1.4.1 Implementations of System Virtual Machines

From the user perspective, most system VMs provide more or less the same functionality. The thing that tends to differentiate them is the way in which they are implemented. As discussed earlier, in Section 1.3, there are a number of interfaces in a computer system

Section 1.2, there are a number of interfaces in a computer system, and this leads to a number of choices for locating the system VMM software. Summaries of two of the more important implementations follow.

Figure 1.11 illustrates the classic approach to system VM architecture (Popek and Goldberg 1974). The VMM is first placed on bare hardware, and virtual machines fit on top. The VMM runs in the most highly privileged mode, while all the guests systems run with lesser privileges. Then in a completely transparent way, the VMM can intercept and implement all the guest OS's actions that interact with hardware resources. In many respects, this system VM architecture is the most efficient, and it provides service to all the guest systems in a more or less equivalent way. One disadvantage of this type of system, at least for desktop users, is that installation requires wiping an existing system clean and starting from scratch, first installing the VMM and then installing guest operating systems on top. Another disadvantage is that I/O device drivers must be available for installation in the VMM, because it is the VMM that interacts directly with I/O devices.

An alternative system VMM implementation builds virtualizing software on top of an existing host operating system—resulting in what is called a *hosted* VM. With a hosted VM, the installation process is similar to installing a typical application program. Furthermore, virtualizing software can rely on the host OS to provide device drivers and other lower-level services; they don't have to be provided by the VMM. The disadvantage of this approach is that there can be some loss of efficiency because more layers of software become involved when OS service is required. The hosted VM approach is taken in the VMware implementation (VMware 2000), a modern system VM that runs on IA-32 hardware platforms.

1.4.2 Whole System VMs: Emulation

In the conventional system VMs described earlier, all the system software (both guest and host) and application software use the same ISA as the underlying hardware. In some important situations, however, the host and guest systems do not have a common ISA. For example, the Apple PowerPC-based systems and Windows PCs use different ISAs (and different operating systems), and they are the two most popular desktop systems today. As another example, Sun Microsystems servers use a different OS and ISA than the Windows PCs that are commonly attached to them as clients. Because software systems are so closely tied to hardware systems, this may require purchase of multiple platform types, even when unnecessary for any other reason, which complicates software support and/or restricts the availability of useful software packages to users.

This situation motivates system VMs, where a complete software system, both OS and applications, is supported on a host system that runs a different ISA and OS. These are called *whole-system VMs* because they essentially virtualize all software. Because the ISAs are different, both application and OS code require emulation, e.g., via binary translation. For whole-system VMs, the most common implementation method is to place the VMM and guest software on top of a conventional host OS running on the hardware.

Figure 1.12 illustrates a whole-system VM built on top of a conventional system with its own OS and application programs. An example of this type of VM is Virtual PC (**Traut 1997**), which enables a Windows system to run on a Macintosh platform. The VM software executes as an application program supported by the host OS and uses no system ISA operations. It is as if the VM software, the guest OS, and guest application(s) are one very large application implemented on the host OS and hardware. Meanwhile the host OS can also continue to run applications compiled for the native ISA; this feature is illustrated in the right-hand section of the drawing.

To implement a system VM of this type, the VM software must emu-

late the entire hardware environment. It must control the emulation of all the instructions, and it must convert the guest system ISA operations to equivalent OS calls made to the host OS. Even if binary translation is used, it is tightly constrained because translated code often cannot take advantage of underlying system ISA features such as virtual memory management and trap handling. In addition, problems can arise if the properties of hardware resources are significantly different in the host and the guest. Solving these mismatches is the major challenge of implementing whole-system VMs.

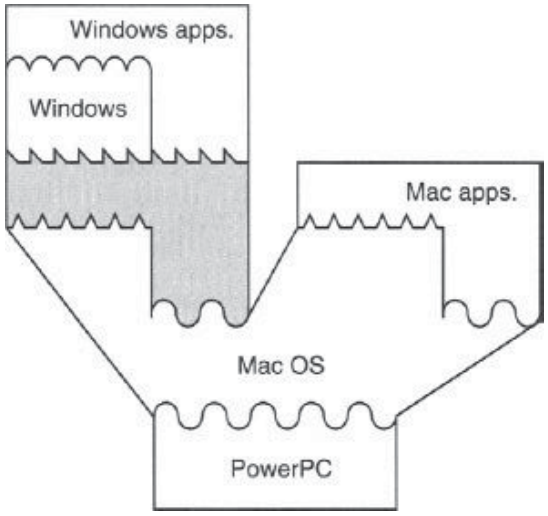


Figure 1.12 A Whole-System VM That Supports a Guest OS and Applications, in Addition to Host Applications.

1.4.3 Codesigned Virtual Machines: Hardware Optimization

In all the VM models discussed thus far, the goal has been functionality and portability—either to support multiple (possibly different) operating systems on the same host platform or to support different ISAs and operating systems on the same platform. In practice, these virtual machines are implemented on hardware already developed for some standard ISA and for which native (host) applications, libraries, and operating systems already exist. By and large, improved performance (i.e., going beyond native platform performance) has not been a goal—in fact minimizing performance losses is often the

not been a goal—in fact minimizing performance losses is often the performance goal.

Codesigned VMs have a different objective and take a different approach. These VMs are designed to enable innovative ISAs and/or hardware implementations for improved performance, power efficiency, or both. The host's ISA may be completely new, or it may be based on an existing ISA with some new instructions added and/or some instructions deleted. In a codesigned VM, there are no native ISA applications. It is as if the VM software is, in fact, part of the hardware implementation.

In some respects, codesigned virtual machines are similar to a purely hardware virtualization approach used in many high-performance superscalar microprocessors. In these designs, hardware renames architected registers to a larger number of physical registers, and complex instructions are decomposed into simpler, RISC-like instructions (**Hennessy and Patterson 2002**). In this book, however, we focus on software-implemented codesigned virtual machines; binary translation is over a larger scope and can be more flexible because it is done in software.

Because the goal is to provide a VM platform that looks exactly like a native hardware platform, the software portion of a codesigned VM uses a region of memory that is not visible to any application or system software. This concealed memory is carved out of real memory at boot time and the conventional guest software is never informed of its existence. VMM code that resides in the concealed memory can take control of the hardware at practically any time and perform a number of different functions. In its more general form, the VM software includes a binary translator that converts guest instructions into native ISA instructions and caches the translated instructions in a region of concealed memory. Hence, the guest ISA never directly executes on the hardware. Of course, interpretation can also be used to supplement binary translation, depending on

performance tradeoffs. To provide improved performance, translation is coupled with code optimization. Optimization of frequently executed code sequences is performed at translation time, and/or it is performed as an ongoing process while the program runs.

Perhaps the best-known example of a codesigned VM is the Transmeta Crusoe (**Halfhill 2000**). In this processor, the underlying hardware uses a native VLIW instruction set, and the guest ISA is the Intel IA-32. In their implementation, the Transmeta designers focused on the power-saving advantages of simpler VLIW hardware. An important computer system that relies on many codesigned VM techniques is the IBM AS/400 system (**Soltis 1996**). The AS/400 differs from the other codesigned VMs, because the primary design objective is support for an object-oriented instruction set that redefines the hardware/software interface in a novel fashion. The current AS/400 implementations are based on an extended PowerPC ISA, although earlier versions used a considerably different, proprietary CISC ISA.

1.5 A Taxonomy

We have just described a rather broad array of VMs, with different goals and implementations. To put them in perspective and organize the common implementation issues, we introduce a taxonomy illustrated in **Figure 1.13**. First, VMs are divided into the two major types: process VMs and system VMs. In the first type, the VM supports an ABI—user instructions plus system calls; in the second, the VM supports a complete ISA—both user and system instructions. Finer divisions in the taxonomy are based on whether the guest and host use the same ISA.

On the left-hand side of **Figure 1.13** are process VMs. These include VMs where the host and guest instruction sets are the same. In the figure, we identify two examples. The first is multiprogrammed sys-

tems, as already supported on most of today's systems. The second is same-ISA dynamic binary optimizers, which transform guest instructions only by optimizing them and then execute them natively.

For process VMs where the guest and host ISAs are different, we also give two examples. These are dynamic translators and HLL VMs. HLL VMs are connected to the VM taxonomy via a "dotted line" because their process-level interface is at a different, higher level than the other process VMs.

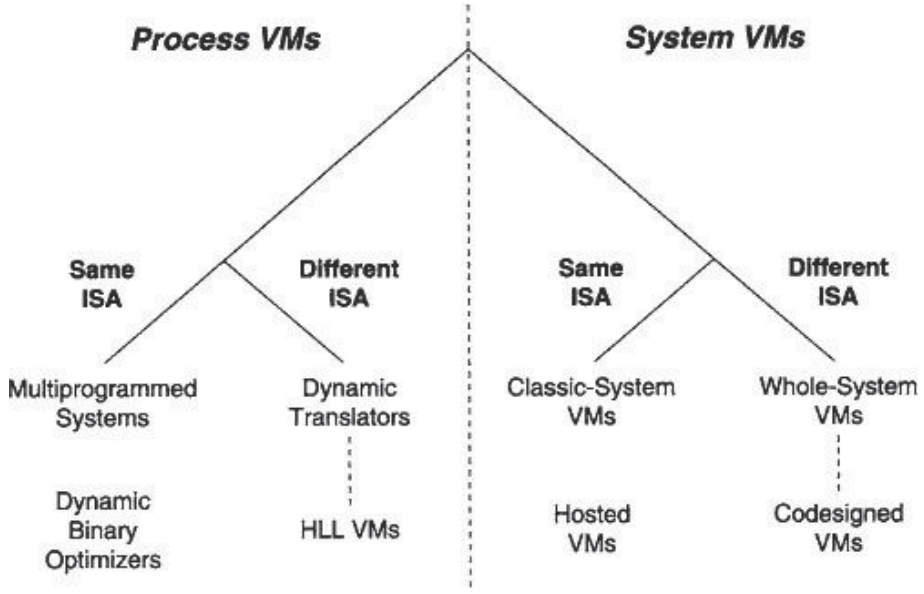


Figure 1.13 A Taxonomy of Virtual Machines.

On the right-hand side of the figure are system VMs. If the guest and host use the same ISA, examples include "classic" system VMs and hosted VMs. In these VMs, the objective is providing replicated, isolated system environments. The primary difference between classic and hosted VMs is the VMM implementation rather than the function provided to the user.

Examples of system VMs where the guest and host ISAs are different include whole-system VMs and codesigned VMs. With whole-system VMs, performance is often of secondary importance compared to accurate functionality, while with codesigned VMs, performance (and

power efficiency) are often major goals. In the figure, codesigned VMs are connected using dotted lines because their interface is typically at a lower level than other system VMs.

1.6 Summary: The Versatility of Virtual Machines

A good way to summarize this chapter is with an example of a realistic system that could conceivably be in use today (**Figure 1.14**). The example clearly illustrates the versatility of virtual machine technologies. A computer user might have a Java application running on a laptop PC. This is nothing special; it is done via a Java virtual machine developed for IA-32/Linux. However, the user happens to have Linux installed as an OS VM via VMware executing on a Windows PC. And, as it happens, the IA-32 hardware is in fact a Transmeta Crusoe, a codesigned VM implementing a VLIW ISA with binary translation (what Transmeta calls *code morphing*) to support the IA-32 ISA. By using the many VM technologies, a Java bytecode program is actually executing as native VLIW.

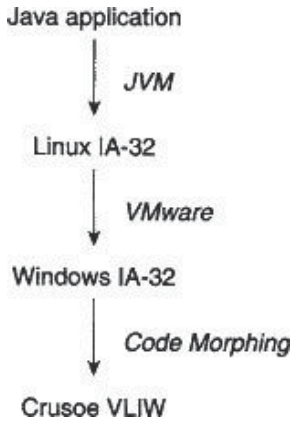


Figure 1.14 Three Levels of VMs. *A Java application running on a Java VM, running on a system VM, running on a codesigned VM.*

1.7 The Rest of the Book

The book can be divided into two major sections, along the lines of the VM taxonomy just described. **Chapters 2** through **6** deal primarily with process virtual machines, and **Chapters 7** through **9** deal primarily with system virtual machines.

It should now be evident that instruction set emulation is an important enabling technology for many virtual machine implementations. Because of its importance, we begin, in **Chapter 2**, with a detailed discussion of emulation. Emulation encompasses both interpretation, where guest instructions are emulated one by one in a simple fashion, and binary translation, where blocks of guest instructions are translated to the ISA of the host platform and saved for multiple executions. In many of the VM implementations, the need to use emulation is obvious (i.e., the guest and host ISAs are different), but, as we will see, the same techniques are important in other VMs, for nonobvious reasons (even in cases where the guest and host ISAs are the same). A case study is the Shade simulation system, which incorporates a number of emulation techniques.

Chapter 3 uses emulation as a starting point and describes the overall architecture and implementation of process VMs. Included are the management of cached binary translations and the handling of complicating issues such as precise traps and self-modifying code. The DEC/Compaq FX!32 system is used as a case study. The FX!32 system supports Windows/IA-32 guest binaries on a Windows/Alpha platform.

Performance is almost always an issue in VM implementations because performance is often lost during the emulation process. This loss can be mitigated by optimizing translated binaries, and **Chapter 4** deals with ways of dynamically optimizing translated code. First, ways of increasing the size of translated code blocks are discussed, and then specific code optimizations are covered. Code optimizations include reordering instructions, to improve pipeline efficiency, for example, and a number of classical compiler opti-

efficiency, for example, and a number of classical compiler optimizations, adapted to dynamically translated binary code. A special case occurs when the source ISA and target ISA are the same and optimization is the primary function provided by the VM. Consequently, **Chapter 4** concludes with a discussion of special-case features of these same-ISA dynamic binary optimizers and includes the HP Dynamo system as a case study.

Chapters 5 and **6** discuss high-level language virtual machines. These virtual machines are designed to enable platform independence. To give some historical perspective, **Chapter 5** begins with a description of Pascal P-code. Because modern HLL VMs are intended to support network-based computing and object-oriented programming, the features important for supporting these aspects are emphasized. The Java virtual machine architecture is described in some detail. Also included is a shorter discussion of the Microsoft common language infrastructure (CLI); where the goals and applications are somewhat broader than with Java, and the discussion is focused on those features that provide the added breadth. **Chapter 6** describes the implementation of HLL VMs, beginning with basic implementation approaches and techniques. High-performance HLL VM implementations are then described, and the IBM Jikes research virtual machine is used as a case study.

Codesigned VMs, the first system-level virtual machines to be discussed, are the topic of **Chapter 7**. The codesigned VM paradigm includes special hardware support to enhance emulation performance. Consequently, much of the chapter focuses on hardware-based performance enhancements. The chapter includes case studies of the Transmeta Crusoe and IBM AS/400 system.

Chapter 8 covers conventional system VMs—that is, VMs that support multiple operating systems simultaneously, primarily relying on software techniques. Basic mechanisms for implementing system VMs and for enhancing their performance are discussed. Some ISAs

are easier to virtualize than others, so features of instruction sets that make them more efficiently virtualized are discussed. The IBM 360–390+ series of VMs are described and used as a case study throughout the chapter. The more recently developed VMware hosted virtual machine, which is targeted at IA-32 platforms, is an important second case study.

Applying virtualization to multiprocessor systems is the topic of **Chapter 9**. Of major interest are techniques that allow the partitioning of resources in large shared-memory multiprocessors to form a number of smaller, virtual multiprocessors. These techniques can be implemented in a number of ways, ranging from those relying on microcode support to purely software solutions. Then ISA emulation in a multiprocessor context is discussed. Although most emulation techniques are the same as with uniprocessors, memory-ordering (consistency) constraints pose some new problems. In this chapter IBM logical partitioning (LPAR) and the Stanford Disco research VM are used as case studies.

Finally, **Chapter 10**, the concluding chapter, looks toward the future and considers a number of evolving VM applications that hold promise. These include support for system security, grid computing, and virtual system portability.

The book also includes an Appendix that reviews the important properties of real machines. The focus is on those aspects of architecture and implementation that are important when virtualization is performed. For some readers this will be familiar material, but other readers may benefit from browsing through the Appendix before proceeding with the book. The Appendix concludes with a brief overview of the IA-32 and PowerPC ISAs, on which many of the examples are based.

---

1. Sometimes referred to informally as *x86*.

2. Throughout this book, we will use the single-word form *runtime* as a noun to describe the virtualizing runtime software in a process VM; *run time* (two words) will be used in the more generic sense: the time during which a program is running.

Previous chapter
< Table of Contents

Next chapter
Chapter 2: Emulation >

Chapter Two

Emulation: Interpretation and Binary Translation

Many virtual machine implementations are based on emulation. We define *emulation* as the process of implementing the interface and functionality of one system or subsystem on a system or subsystem having a different interface and functionality. For example, a VT100 terminal emulator running inside a window on a PC presents to the user an interface and functionality that is nearly identical to that of a real VT100 terminal. In fact, taken in this general sense, one could argue that virtualization itself is simply a form of emulation. In this chapter, however, we give emulation a narrower meaning by applying it specifically to instruction sets.

Instruction set emulation is a key aspect of many virtual machine implementations because the virtual machine must support a program binary compiled for an instruction set that is different from the one implemented by the host processor(s). For example, Intel IA-32 program binaries are more widely available than binaries for any other instruction set. Consequently, a user may wish to use a virtual machine to execute an IA-32 program binary on some other platform, for example, an Apple Macintosh, which incorporates a PowerPC processor. For high-level virtual machines, binary classes (using the Java terminology) incorporate a stack-based, bytecode instruction set that can be emulated on any of a number of different host platforms.

In terms of instruction sets, emulation allows a machine implementing one instruction set, the *target* instruction set, to reproduce the behavior of software compiled to another instruction set, the *source*

instruction set. This is illustrated in **Figure 2.1**. Note that we use the terms *source* and *target* specifically for instruction set emulation, and we use the terms *guest* and *host* when referring to complete virtual machine environments and supporting platforms (which often involve more than just ISAs). The reader should also be aware that the literature often does not use consistent terminology when describing guest/host and source/target relationships.

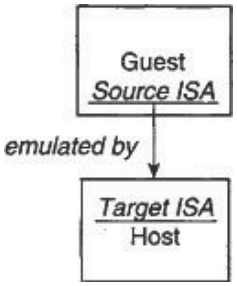


Figure 2.1 Terms Describing the Emulation Process. *Emulation allows a guest to support a source instruction set while running on a host platform executing a target instruction set.*

For many virtual machine applications, it is important that the emulation of the instruction set be performed efficiently. The lower the overhead of emulation, the more attractive the virtual machine will be. This chapter will focus on techniques for emulation of conventional instruction sets that already have hardware implementations rather than those designed specifically for virtual machine implementation, such as the Java bytecode instruction set. The latter *virtual instruction sets* can be implemented with similar emulation techniques as conventional instruction sets, but they also have special properties and therefore can take advantage of other emulation techniques. High-level language VM-specific instruction sets will be discussed in **Chapter 5**.

A complete ISA consists of many parts, including the register set and memory architecture, the instructions, and the trap and interrupt architecture. A virtual machine implementation is usually concerned with all aspects of ISA emulation. In this chapter, however, we focus primarily on emulating the operation of user-level instructions in the absence of exceptions (traps and interrupts). The emula-

tions in the absence of exceptions (traps and interrupts). The emulation of the memory addressing architecture, traps, interrupts, and other features are discussed in this chapter only as is necessary to understand instruction emulation. In **Chapters 3** and **4**, we extend the discussion to emulation of memory architecture, traps, and interrupts in the context of process virtual machines. In later chapters (primarily **Chapters 7** and **8**) we discuss emulation of system instructions and other ISA issues that are specific to system virtual machines.

To some degree, instruction emulation techniques can be applied to every type of VM we discuss in this book. Although we are interested primarily in the case where the source and target instruction sets are different, there are some virtual machine applications where the instruction sets are the same. In these situations, emulation may not be performed, strictly speaking, but the very same techniques are used for other purposes. One such application is same-ISA dynamic binary optimization, where a program binary is optimized at run time on the same platform for which it was originally compiled. Another example is in system VMs where the VMM must control and manage the execution of certain privileged guest OS instructions, and emulation-like techniques are used for this purpose.

Instruction set emulation can be carried out using a variety of methods that require different amounts of computing resources and offer different performance and portability characteristics. At one end of the spectrum is the straightforward technique of *interpretation,* while on the other is *binary translation.* Interpretation involves a cycle of fetching a source instruction, analyzing it, performing the required operation, and then fetching the next source instruction—all in software. Binary translation, on the other hand, attempts to amortize the fetch and analysis costs by translating a block of source instructions to a block of target instructions and saving the translated code for repeated use. In contrast to interpretation, bi-

nary translation has a bigger initial translation cost but a smaller execution cost. The choice of one or the other depends on the number of times a block of source code is expected to be executed by the guest software. Predictably, there are techniques that lie in between these extremes. For example, *threaded interpretation* eliminates the interpreter loop corresponding to the cycle mentioned earlier, and efficiency can be increased even further by predecoding the source instructions into a more efficiently interpretable intermediate form.

2.1 Basic Interpretation

Interpreters have a long and rich history. Some programming languages, e.g., the functional language LISP, rely on an interpreted implementation. Perl is a widely used language that is commonly implemented through interpretation. The FORTH language is probably better known for its “threaded code” interpretation model than for any other feature. A good reference is **Debaere and Van Campenhout (1990)**. In this chapter, however, we are more interested in applying interpretation techniques to program binaries (machine code) rather than high-level languages.

In our narrower context, an interpreter program emulates and operates on the complete architected state of a machine implementing the source ISA, including all architected registers and main memory (**Figure 2.2**). The image of the guest memory, including both program code and program data, is kept in a region of memory maintained by the interpreter. The interpreter’s memory also holds a table we call the *context block*, which contains the various components of the source’s architected state, such as general-purpose registers, the program counter, condition codes, and miscellaneous control registers.

