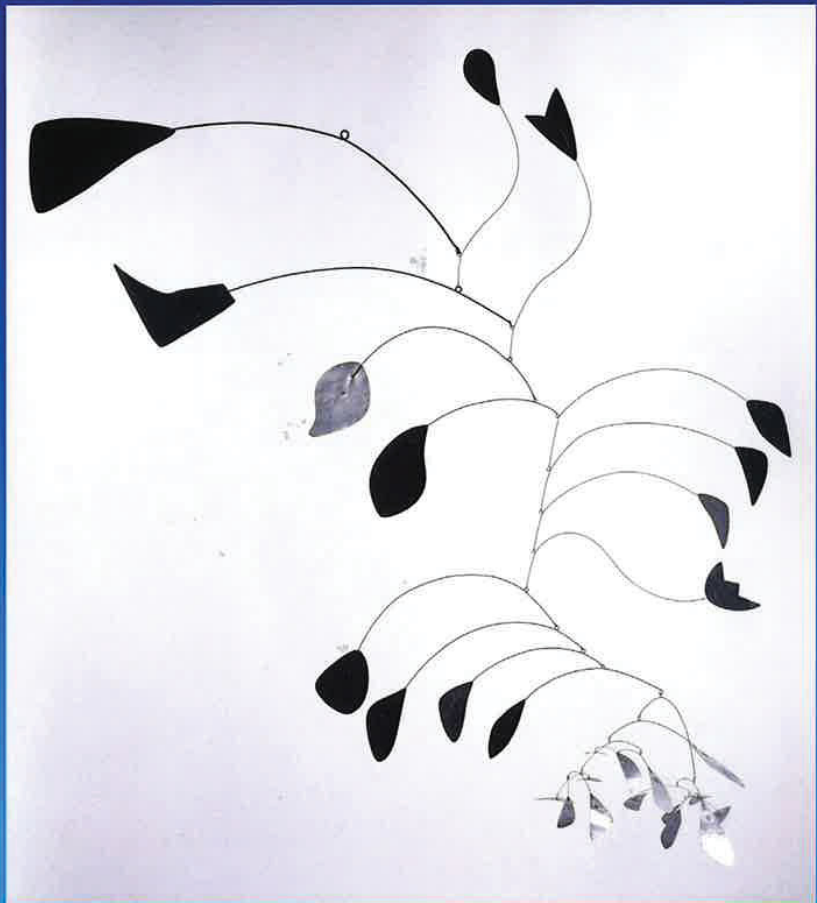




JAMES E. **SMITH** • RAVI **NAIR**

VIRTUAL MACHINES

VERSATILE
PLATFORMS
FOR SYSTEMS
AND
PROCESSES



MK[®]
MORGAN KAUFMANN

Virtual Machines

Versatile Platforms for Systems and Processes

James E. Smith

University of Wisconsin—Madison

Ravi Nair

IBM Thomas J. Watson Research Center



AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
MORGAN KAUFMANN PUBLISHERS IS AN IMPRINT OF ELSEVIER



Publisher
Publishing Services Manager
Senior Project Manager
Editorial Assistant
Cover Design
Text Design
Composition
Technical Illustration
Copyeditor
Proofreader
Indexer
Interior printer
Cover printer

Denise E.M. Penrose
Simon Crump
Angela Dooley
Kimberlee Honjo
Hannus Design Associates
Rebecca Evans
CEPHA Imaging Pvt. Ltd.
Dartmouth Publishing, Inc.
Simon & Assocs.
Phyllis Coyne et. al.
Northwind Editorial
The Maple-Vail Manufacturing Group
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.

© 2005 by Elsevier Inc. All rights reserved.

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 means—electronic, mechanical, photocopying, scanning, or otherwise—without prior written permission of the publisher.

Permissions may be sought directly from Elsevier's 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 homepage (<http://elsevier.com>) by selecting "Customer Support" and then "Obtaining Permissions."

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-13: 978-1-55860-910-5
ISBN-10: 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 and bound by CPI Group (UK) Ltd, Croydon, CR0 4YY

Transferred to Digital Print 2012

Working together to grow
libraries in developing countries

www.elsevier.com | www.bookaid.org | www.sabre.org

ELSEVIER

BOOK AID
International

Sabre Foundation

Contents

Foreword	ix
Preface	xi

Chapter One

Introduction to Virtual Machines 1

1.1	Computer Architecture	6
1.2	Virtual Machine Basics	9
1.3	Process Virtual Machines	13
1.4	System Virtual Machines	17
1.5	A Taxonomy	22
1.6	Summary: The Versatility of Virtual Machines	23
1.7	The Rest of the Book	24

Chapter Two

Emulation: Interpretation and Binary Translation 27

2.1	Basic Interpretation	29
2.2	Threaded Interpretation	32
2.3	Predecoding and Direct Threaded Interpretation	34
2.4	Interpreting a Complex Instruction Set	38
2.5	Binary Translation	49
2.6	Code Discovery and Dynamic Translation	52
2.7	Control Transfer Optimizations	64
2.8	Instruction Set Issues	68
2.9	Case Study: Shade and the Role of Emulation During Simulation	77
2.10	Summary: Performance Tradeoffs	80

Chapter Three

Process Virtual Machines	83
3.1 Virtual Machine Implementation	85
3.2 Compatibility	87
3.3 State Mapping	95
3.4 Memory Architecture Emulation	102
3.5 Instruction Emulation	114
3.6 Exception Emulation	119
3.7 Operating System Emulation	128
3.8 Code Cache Management	133
3.9 System Environment	140
3.10 Case Study: FX!32	142
3.11 Summary	145

Chapter Four

Dynamic Binary Optimization	147
4.1 Dynamic Program Behavior	153
4.2 Profiling	156
4.3 Optimizing Translation Blocks	167
4.4 Optimization Framework	180
4.5 Code Reordering	186
4.6 Code Optimizations	201
4.7 Same-ISA Optimization Systems: Special-Case Process Virtual Machines	208
4.8 Summary	218

Chapter Five

High-Level Language Virtual Machine Architecture	221
5.1 The Pascal P-Code Virtual Machine	225
5.2 Object-Oriented High-Level Language Virtual Machines	228
5.3 The Java Virtual Machine Architecture	241
5.4 Completing the Platform: APIs	261
5.5 The Microsoft Common Language Infrastructure: A Flexible High-Level Language Virtual Machine	267
5.6 Summary: Virtual ISA Features	275

Chapter Six

High-Level Language Virtual Machine Implementation	281
6.1 Dynamic Class Loading	284
6.2 Implementing Security	286

6.3	Garbage Collection	294
6.4	Java Native Interface	304
6.5	Basic Emulation	305
6.6	High-Performance Emulation	306
6.7	Case Study: The Jikes Research Virtual Machine	320
6.8	Summary	327

Chapter Seven

Codedigned Virtual Machines 329

7.1	Memory and Register State Mapping	333
7.2	Self-Modifying and Self-Referencing Code	337
7.3	Support for Code Caching	339
7.4	Implementing Precise Traps	344
7.5	Input/Output	351
7.6	Applying Codedigned Virtual Machines	352
7.7	Case Study: Transmeta Crusoe	354
7.8	Case Study: IBM AS/400	357
7.9	Summary	367

Chapter Eight

System Virtual Machines 369

8.1	Key Concepts	373
8.2	Resource Virtualization — Processors	382
8.3	Resource Virtualization — Memory	396
8.4	Resource Virtualization — Input/Output	404
8.5	Performance Enhancement of System Virtual Machines	415
8.6	Case Study: VMware Virtual Platform	426
8.7	Case Study: The Intel VT-x (Vanderpool) Technology	436
8.8	Summary	442

Chapter Nine

Multiprocessor Virtualization 445

9.1	Partitioning of Multiprocessor Systems	445
9.2	Physical Partitioning	455
9.3	Logical Partitioning	458
9.4	Case Study: Cellular Disco System Virtual Machine-Based Partitioning	475
9.5	Virtualization with Different Host and Guest ISAs	485
9.6	Summary	496

Chapter Ten

Emerging Applications 499

- 10.1 Security 501
- 10.2 Migration of Computing Environments 520
- 10.3 Grids: Virtual Organizations 535
- 10.4 Summary 552

Appendix A

Real Machines 553

- A.1 Computer System Hardware 554
- A.2 The User ISA: Computation 561
- A.3 The System ISA: Resource Management 566
- A.4 Operating System Organization 580
- A.5 The Operating System Interface 583
- A.6 System Initialization 586
- A.7 Multiprocessor Architecture 588
- A.8 Example Instruction Set Architectures 600

References 613

Index 629

m
e.
in
ce
re

id
er
I
k
0
n
al
n
re
h
d

m
c.
to
15

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 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

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 dependencies; 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 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

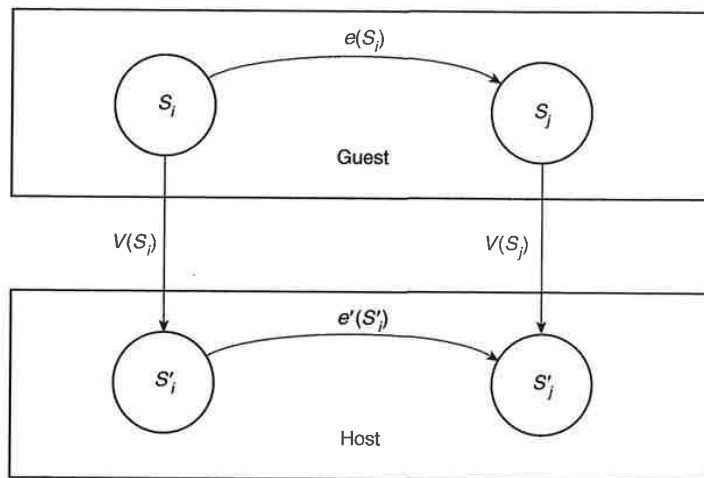


Figure 1.2 Virtualization. *Formally, virtualization is the construction of an isomorphism between a guest system and a host; $e' \circ V(S_i) = V \circ e(S_i)$.*

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.

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

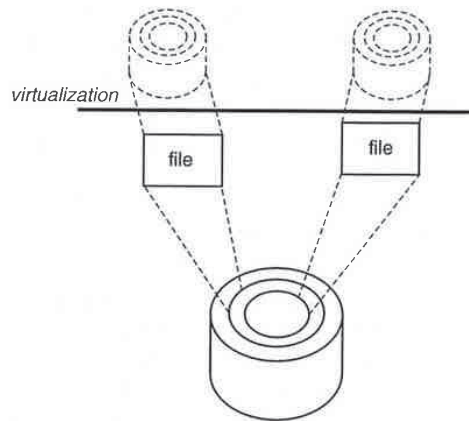


Figure 1.3 Implementing Virtual Disks. *Virtualization provides a different interface and/or resources at the same level of abstraction.*

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.

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 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

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.

1.2 Virtual Machine Basics

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 perspective 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 1.5a).

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 1.5b).

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

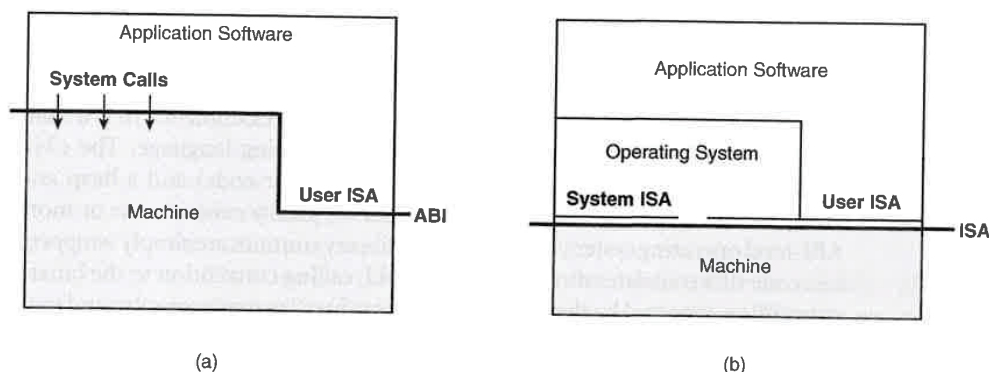


Figure 1.5 Machine Interfaces. (a) Application binary interface (ABI); (b) instruction set architecture (ISA) interface.

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.

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

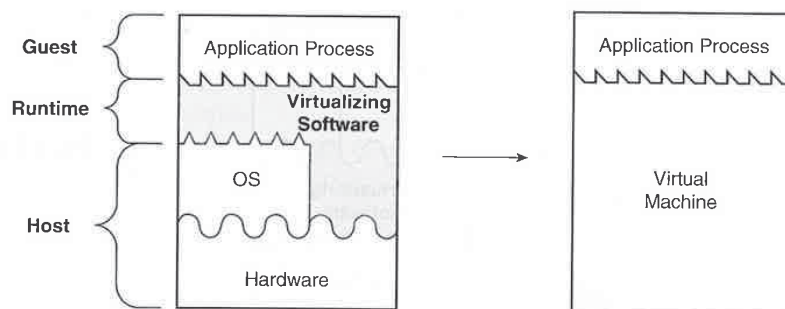


Figure 1.6 A Process Virtual Machine. Virtualizing software translates a set of OS and user-level instructions composing one platform to another, forming a process virtual machine capable of executing programs developed for a different OS and a different ISA.

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.

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.

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.

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 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.

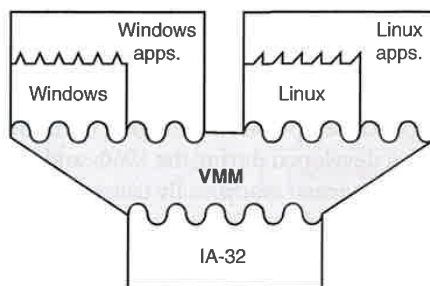


Figure 1.11 A System VM That Supports Multiple OS Environments on the Same Hardware.

Chapter Eight

System Virtual Machines

It was observed very early that many of the hardware resources in a typical computer system are underutilized. The concept of *time sharing* was developed to improve resource utilization by allowing multiple users to access a single computer system simultaneously, with each user being given the illusion of having access to a full set of system resources. In order to create this illusion, a multiprogramming operating system essentially implements a process virtual machine for each application-level program and switches resources among the programs on a time-shared basis.

System virtual machines take this concept one step further by providing a similar illusion for complete systems. A system VM environment is capable of supporting multiple system images simultaneously, each running its own operating system and associated application programs. Each operating system controls and manages a set of virtualized hardware resources, as illustrated in Figure 8.1. The virtual resources include a processor (or processors), storage resources, and peripheral devices needed to perform input and output (I/O) for the system.

In a system VM environment, real resources of the *host* platform are shared among the *guest* system VMs, with a layer of software, the virtual machine monitor (VMM), managing the allocation of, and access to, the hardware resources of the host platform. The VMM owns the real system resources and makes them available to one or more guest operating systems that alternately execute on the same hardware. Thus, a guest operating system in a virtual machine environment is given the illusion of owning the resources it subsequently allocates to its various user programs.

Each virtual resource may or may not have a corresponding physical resource. When a corresponding physical resource is available, the VMM decides how access will be provided to virtual machines requiring its

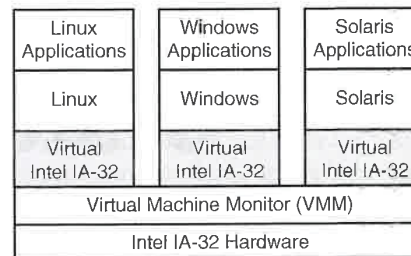


Figure 8.1 Example System VM Environment. In this example, the Intel IA-32 platform is capable of simultaneously running the Linux, Windows, and Solaris operating systems, along with their applications.

use; e.g., the resource may be partitioned or may be time shared. When a virtual resource does not have a matching physical resource, the VMM may emulate the action of the desired resource, typically via a combination of software and other resources that are physically available on the host platform.

System virtual machines have a number of useful applications, some historical, some relevant today, and many that are likely to be important in the future. Some of these applications are described next.

- **Implementing multiprogramming:** Using multiple single-user virtual machines is a simple way of providing multiprogramming and time sharing without requiring a complete multiprogramming OS. Each user in a multiuser environment is given the illusion of having an entire machine, including the CPU, memory, and peripherals. Many of the users of VMs on early mainframe computers, such as the IBM System/370 and its successors, used virtualization as an alternative to a time-sharing system. These users ran a simple, compact, and efficient single-user operating system called the *conversational monitor system* (CMS), instead of the monolithic time-sharing operating systems available at the time.
- **Multiple single-application virtual machines:** This is an extension of the multiple single-user virtual machine concept. Running each application in its own virtual machine increases the robustness of the system. Erratic behavior of one application and its associated OS on a conventional system could bring down the machine. If this happens on a virtual machine, it is less likely to affect the operation of another application running on a different virtual machine. This may be useful in situations where a new application might corrupt the entire system because of either bugs or viruses.
- **Multiple secure environments:** A system VM provides a *sandbox* that isolates one system environment from other environments, thus ensuring a

level of security that may not be provided by a single operating system. For example, a user accustomed to running applications on a private machine may be reluctant to transfer the applications to a Web server unless there is some guarantee that the user's resources and activity cannot be accessed or monitored by other users of the same server. In such a situation, a virtual machine can be used to provide an environment that, for all practical purposes, is isolated from the others and through which it is not possible for one user to observe or change another's data and activity.

- **Managed application environments:** In an organization that provides a core set of supported applications to users who also want to install their own applications, VM technology allows the core set of applications to be placed on one virtual machine, protected from other virtual machines on which users install new applications or develop their own programs.
- **Mixed-OS environments:** A single hardware platform can support two different operating systems concurrently. A user may prefer office productivity tools that run on one operating system, for example, but prefer another operating system for application development. The user can do both by installing two virtual machines on a single hardware platform, one for each operating system.
- **Legacy applications:** When a new version of an OS is released, it is not uncommon for OS developers to support seldom-used features with degraded performance while supporting new features that allow most new applications to run with improved performance. In this situation, users who need to run old applications without degraded performance can use one virtual machine to run the "legacy" application under the old OS while using another VM to take advantage of the presumably improved features in the newer version of the OS.
- **Multiplatform application development:** Software developers often need to implement and support software that works on multiple operating systems. Virtual machine technology allows software developers to test their software using multiple virtual machines on the same hardware platform. This is both more convenient and more cost effective than having multiple sets of hardware, one for each OS.
- **New system transition:** A system virtual machine allows a user to migrate to a new operating system in a gradual way. The user can try out a new operating system on one virtual machine even while the old operating system is running applications on a different virtual machine. The old operating system can be removed from the system after it has been ascertained that all relevant applications work correctly on the new operating system.

code is unavailable or because of licensing agreements. In these cases, the VMM may be implemented at the user level, a privilege level below that of the host operating system, as shown in Figure 8.5c. Such a system is commonly referred to as a *user-mode* hosted VM system. For efficiency reasons, however, it is desirable to have at least part of the VMM work in the most privileged mode. This can often be achieved by modifying the host operating system through mechanisms commonly provided to extend the functionality of an operating system, such as kernel extensions or device drivers. Such a system that has parts of the VMM operating in a privileged mode and other parts in nonprivileged mode, as depicted in Figure 8.5d, is referred to as a *dual-mode* hosted VM system. We will examine features of such systems further in Section 8.4.3.

8.1.5 IBM VM/370

The first virtual machine environment was the IBM System/360 Model 40 VM (circa 1965) (Adair et al. 1966; Creasy 1981). The intention of the developers was to build a time-sharing system that extended the then-novel concept of virtual memory to other parts of the computer system. After debating the alternatives, the design team decided that the most appropriate interface to provide to the user would be the full System/360 architecture interface (the ISA) rather than just the user-level architectural interface enhanced by system library functions (the ABI). The perceived advantage was that this would be the best way to protect users from one another while preserving compatibility as the System/360 architecture evolved. It was not until much later, with a model of the System/370, that VM became mainstream. We will refer to the IBM virtual machine in the rest of the chapter as the IBM VM/370 — many of the principles are still in use today in the z/VM running on the IBM zSeries mainframes. An informative account of the development of VM on the IBM platforms is provided in Varian (1997), while a survey of virtual machine research at that time appears in Goldberg (1974).

The virtual machine monitor of VM/370 was called the *control program* (CP). The CP design team also developed a single-user operating system called the *conversational monitor system* (CMS), mainly to demonstrate the advantages of modularization for system evolution. The CP/CMS design separated the function of resource management from the function of providing services to the user. The success of the project was in large measure due to the design of the System/370 architecture, which allowed an elegantly simple implementation of the virtual machine concept.

Even though CP and CMS are often referred to as a pair, one can exist without the other. In fact, CMS was developed on the bare machine before

Nonprivileged
modes

Privileged
modes

processor executes in
dual mode. (c–d) A
In both systems,
privileged level. The
d) when parts of

is, as we will
in the guest.

and virtual
system and
the latter, the
that of the

Figure 8.5a.
privilege mode
e VM system.

in the highest
Figure 8.5b.
and the guest
est operating
privilege lower
l of the guest
and studied

ten advanta-
that is already
in (in this case
in the VMM
and manage
proprietary
se the source