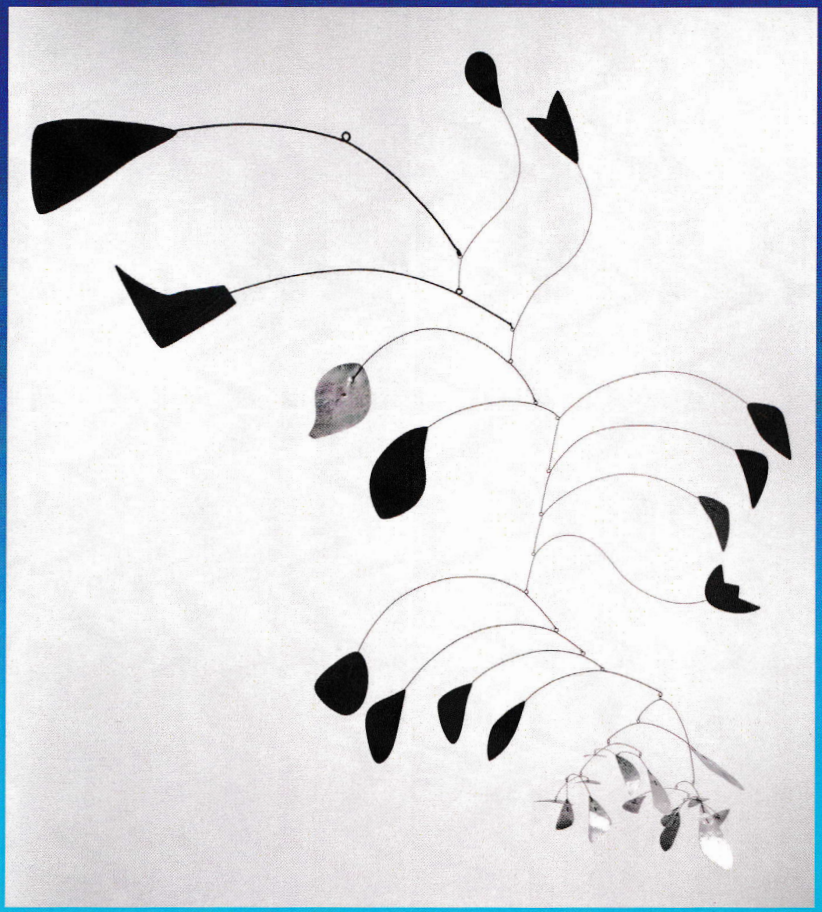
JAMES E. SMITH • RAVI NAIR

VIRTUAL MACHINES

VERSATILE PLATFORMS FOR SYSTEMS AND PROCESSES



ELSEVIER

MORGAN KAUFMANN

SAMSUNG 1008

Virtual Machines

Versatile Platforms for Systems and Processes

James E. Smith
University of Wisconsin—Madison

Ravi Nair
IBM Thomas J. Watson Research Center



ELSEVIER

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

MORGAN KAUFMANN PUBLISHERS

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.

© 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
Codesigned 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 Codesigned 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

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

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 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.

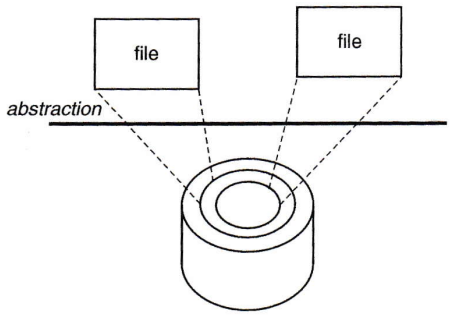


Figure 1.1 Files Are an Abstraction of a Disk. *A level of abstraction provides a simplified interface to underlying resources.*

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, compiled 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

1. Sometimes referred to informally as *x86.*

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 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

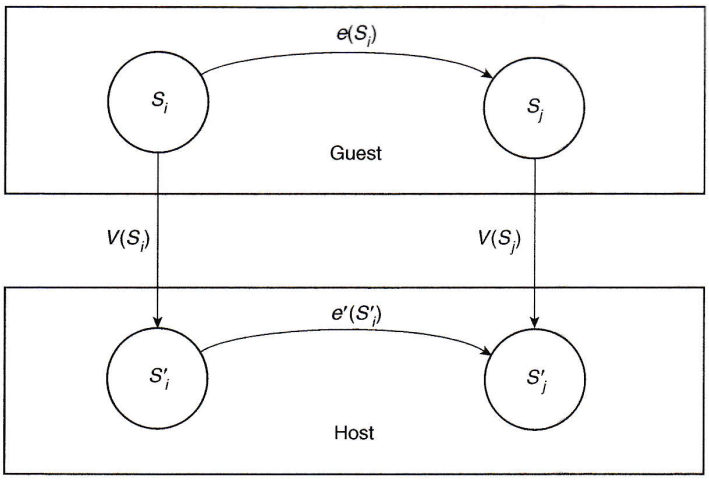


Figure 1.2 Virtualization. *Formally, virtualization is the construction of an isomorphism between a guest system and a host;* $e' \circ V(S_i) = V \circ e(S_i)$.

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

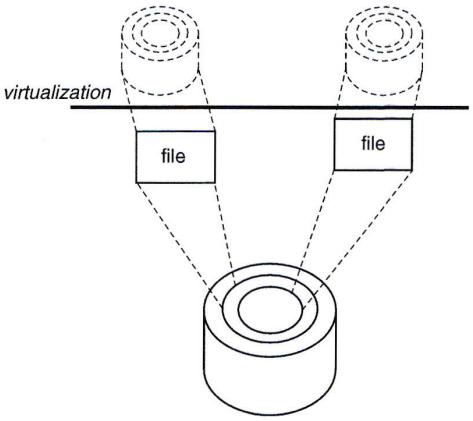


Figure 1.3 Implementing Virtual Disks. *Virtualization provides a different interface and/or resources at the same level of abstraction.*

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

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 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

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 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 1.5a).

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

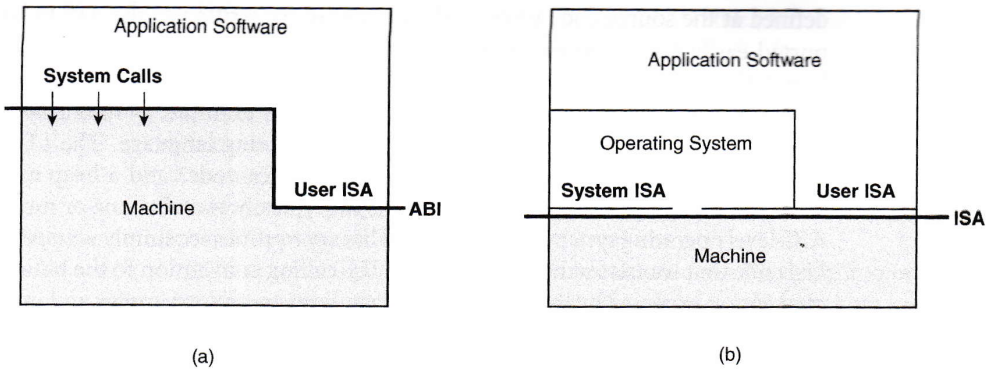


Figure 1.5 Machine Interfaces. *(a) Application binary interface (ABI); (b) instruction set architecture (ISA) interface.*

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

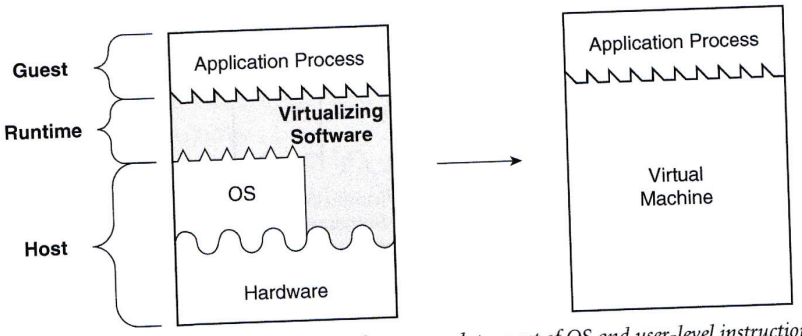


Figure 1.6 A Process Virtual Machine. *Virtualizing software translates a set of OS and user-level instructions composing one platform to another, forming a process virtual machine capable of executing programs developed for a different OS and a different ISA.*

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.

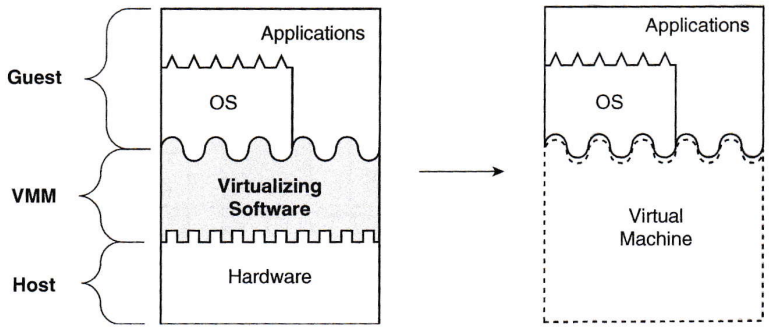


Figure 1.7 A System Virtual Machine. *Virtualizing software translates the ISA used by one hardware platform to another, forming a system virtual machine, capable of executing a system software environment developed for a different set of hardware.*

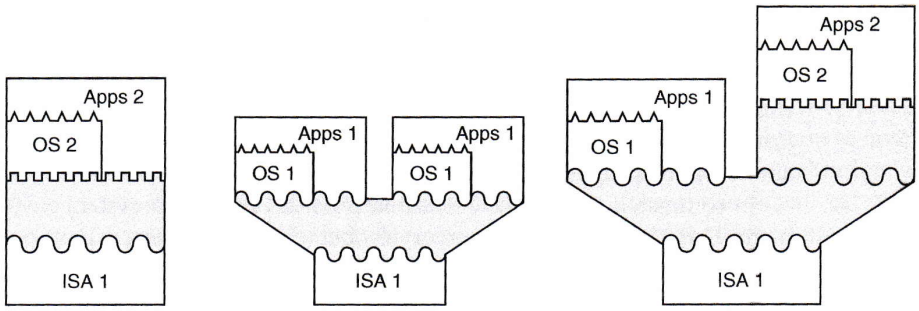


Figure 1.8 Examples of Virtual Machine Applications. *(a) Emulating one instruction set with another; (b) replicating a virtual machine so that multiple operating systems can be supported simultaneously; (c) composing virtual machine software to form a more complex, flexible system.*

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 1.8a), 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 1.8b), each

capable of running a complete operating system and/or a set of applications. Finally, the virtual machine functions 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

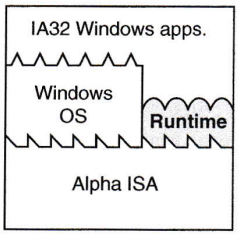


Figure 1.9 A Process VM That Emulates Guest Applications. *The Digital FX!32 system allows Windows IA-32 applications to be run on an Alpha Windows platform.*

the 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).

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.

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 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 1.10a, 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 1.10b). 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 1.10b, 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*

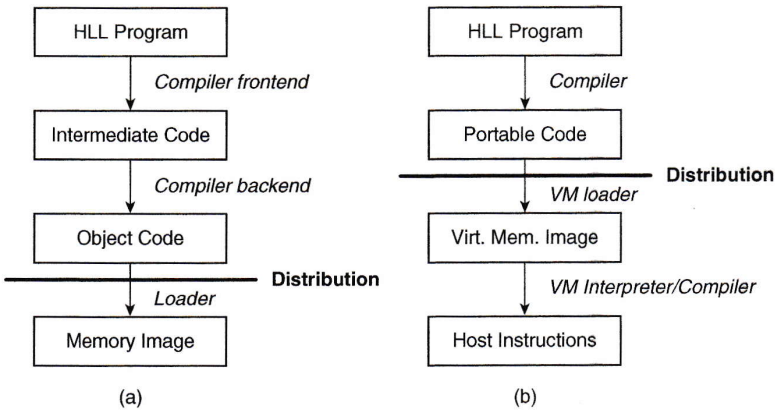


Figure 1.10 High-Level Language Environments. *(a) A conventional system, where platform-dependent object code is distributed; (b) an HLL VM environment, where portable intermediate code is "executed" by a platform-dependent virtual machine.*

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.

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 bytecode 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 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.

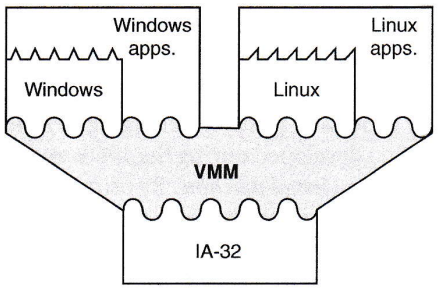


Figure 1.11 A System VM That Supports Multiple OS Environments on the Same Hardware.

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 1.2, 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 emulate the entire hardware environment. It must control the emulation of all the

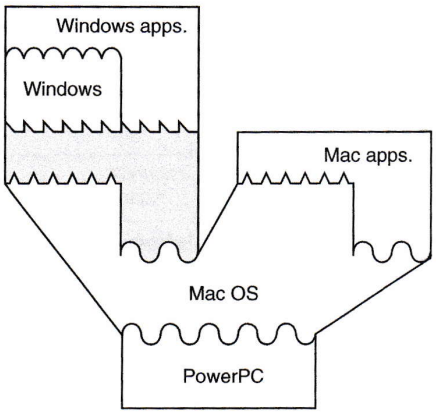


Figure 1.12 A Whole-System VM That Supports a Guest OS and Applications, in Addition to Host Applications.

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.

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 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

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

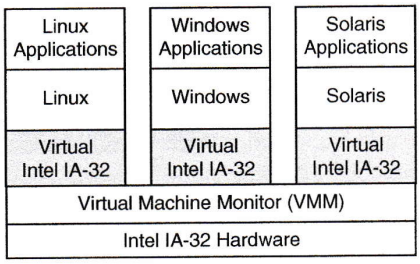


Figure 8.1 Example System VM Environment. *In this example, the Intel IA-32 platform is capable of simultaneously running the Linux, Windows, and Solaris operating systems, along with their applications.*

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.

- **System software development:** In large systems, it is sometimes necessary to develop (or test) new system software while the same system is being used for running important applications. In this situation it could be very costly if a bug in the code under development were to bring down the entire system. This situation can be avoided by encapsulating the development environment on a virtual machine separate from the virtual machine where the production-level applications are running.
- **Operating system training:** It is often necessary to demonstrate techniques and effects of changing parameters or policies when training system management personnel. Such training is better done on a copy of the operating system running on a virtual machine than on the operating system running on the real machine so that other users of the operating system do not suffer from unexpected effects.
- **Help desk support:** Help desk service personnel can bring up a virtual machine that emulates the hardware configuration of a client in order to determine the nature and cause of problems the client may be encountering. It is not necessary to have all possible configurations of hardware physically available — virtual machine technology allows a single hardware platform to be set to any of several different configurations.
- **Operating system instrumentation:** Running the operating system on a virtual machine allows the virtual machine monitor to instrument accesses to hardware resources selectively. Not only can all events of a particular type, e.g., page faults, be counted, they can also be logged with detailed information about their nature, their origin, and how the requests were satisfied. Moreover, the programming for all such instrumentation and measurements can be encapsulated outside the operating system, communicating solely with the virtual machine monitor. Today, it is quite common for operating system researchers to conduct most of their experiments using virtual machines rather than directly on the hardware platform. Keefe (1968) describes early work on IBM mainframes on system evaluation using a virtual machine. The User Mode Linux system (UMLinux 2003) also uses virtual machine technology to test the fault tolerance of a Linux system.
- **Event monitoring:** Some virtual machines provide capabilities that cannot be performed on a native system. For example, virtual machines lend themselves better to providing traces of execution or dumps of the machine state at points of interest. Similarly, the ability of virtual machines to replay a system execution from some saved state is useful for analyzing the behavior of a system to understand unexpected behavior.

- **System encapsulation:** System virtual machines provide a convenient way of encapsulating the state of the entire machine. This is useful for checkpointing machine state, with a view to resuming execution at a different time or even on a different host machine.

We are nearly ready to describe system VMs in more detail. Before doing so, however, we first point out that this chapter considers a very important class of system VMs but not all system VMs. In particular, as described in the taxonomy given in Section 1.5, this chapter will focus on system VMs where the ISA of the host and guest are the same. Furthermore, most of the attention will be given to uniprocessor systems. Important aspects of VM implementation related to other types of system VMs can be found elsewhere in this book, as summarized shortly.

Multiprocessor virtualization is a simple extension of uniprocessor virtualization. However, there are some special considerations when a multiprocessor is being virtualized. One consideration is that with a single processor, virtualization must be done by time sharing the single processor resource. With multiprocessors, however, virtualization can be achieved by partitioning the multiple hardware processors among the virtual systems. The choice between time-sharing processors and partitioning (or using both) opens up a number of additional design alternatives when multiprocessors are being virtualized.

Another important consideration for shared-memory multiprocessors, particularly when the host and guest ISAs are different, is the memory models supported by the guest and host platforms. The two aspects of memory models that are crucial are memory coherence and memory consistency. Both of these involve the way in which memory operations performed by one processor are observed by other processors in the systems. If the host system does not support the same coherence and/or consistency models as the guest assumes, then the virtualization process may need to take special steps to ensure compatibility. We discuss considerations that are specific to multiprocessor virtualization in Chapter 9.

When the guest and host ISAs are different, software in the VMM can emulate the virtual ISA using the techniques described in Chapters 2–4. Chapter 9 will address some of the system VM issues that arise where the ISA of the guest is different from that of the host.

8.1 Key Concepts

In this section we provide an introduction to the various parts of a hardware platform, with a view to understanding how virtualization can be achieved.

However, the VMM maps these ASIDs to two different ASIDs (4 and 9) so that they can be distinguished in the real TLB.

8.4 Resource Virtualization — Input/Output

Virtualization of the input/output subsystem is one of the more difficult parts of implementing a system virtual machine environment. Each I/O device type has its own characteristics and needs to be controlled in its own special way. The difficulty of virtualizing I/O devices is compounded by the fact that the number of devices of a given type is often very large and continues to grow. Moreover, the number of I/O device types is also increasing. Virtualization of such a large number of devices is a challenge for any new VMM.

On the other hand, techniques to share I/O devices have been developed since the early days of time sharing. The proliferation of I/O devices is also a problem for conventional operating systems, which have developed abstractions to support a wide variety of devices and device types. It is possible to adapt many of these abstraction techniques for use in virtual machine systems.

The virtualization strategy for a given I/O device type consists of constructing a virtual version of the device and then virtualizing the I/O activity directed at the device. A virtual device given to a guest VM is typically (but not necessarily) supported by a similar, underlying physical device. When a guest VM makes a request to use the virtual device, the request is intercepted by the VMM. The VMM converts the request to the equivalent request for the underlying physical device and it is carried out. The next two subsections discuss techniques for virtualizing devices and carrying out virtual I/O activity.

8.4.1 Virtualizing Devices

A number of techniques can be used for virtualizing I/O devices, and the specific technique for virtualizing a given device depends on whether it is shared and, if so, the ways in which it can be shared. Following are the common categories of devices.

Dedicated Devices

Some I/O devices, by their very nature, must be dedicated to a particular guest VM or at least are switched from one guest to another on a very long time scale. Examples of dedicated I/O devices are the display, keyboard, mouse, and speakers of a virtual machine user, as shown in Figure 8.2. In this case, the device itself does not necessarily have to be virtualized. Requests to and from

the device could theoretically bypass the VMM and go directly to the guest operating system. However, in practice, the requests are usually routed through the VMM because the guest OS is running in the nonprivileged user mode. Interrupts from the device are first handled by the VMM, which determines that it came from a device dedicated to a particular guest VM and queues up the interrupt to be handled by the VM when it is activated by the VMM.

Partitioned Devices

For some devices, such as the disk, it is convenient to partition the available resources among the virtual machines. A very large disk, for example, can be partitioned into several smaller *virtual disks* that are then made available to the virtual machine as dedicated devices. A virtual disk is treated by a virtual machine exactly as a physical disk, except with a smaller capacity.

To emulate an I/O request for the virtual device such as a disk, the VMM must translate the parameters (e.g., the track and sector locations) into corresponding parameters for the underlying physical device(s), using a map maintained for this purpose, and then reissue the request to the physical device (e.g., the disk controller). Similarly, status information coming back from the physical device is used to update status information in the map and is transformed into appropriate parameters for the virtual device before being delivered to the guest VM.

Shared Devices

Some devices, such as a network adapter, can be shared among a number of guest VMs at a fine time granularity. Each guest may have its own virtual state related to usage of the device, e.g., a virtual network address. This state information is maintained by the VMM for each guest VM. A request by a guest VM to use the device is translated by the VMM to a request for the physical device through a virtual device driver. For a network, the VMM's virtualization routine will translate a request from some virtual machine to a request on a network port using its own physical network address. Similarly, the incoming requests through various ports will be translated into requests for virtual network addresses associated with the different virtual machines.

Spooled Devices

A spooled device is shared, but at a much higher granularity than a device such as a network adapter. An example of a device that is often spooled is a printer. Conceptually, the printer is solely under the control of the program that is

printing until the completion of the printing task. This must be the case even if the program is swapped in and out several times during the printing process. *Spooling* allows this to be accomplished (Madnick and Donovan 1974). In the case of a printer, spooling involves the buffering of the lines to be printed from each program in an area of memory dedicated to that program. A buffer is closed only when the program that owns it sends a special signal to the printer. The operating system schedules the printing of the spool buffers as they close, ensuring that the output of each program is printed completely before the printer is allocated to the next program.

Virtualization of spooled devices can be performed by using a two-level spool table approach, as depicted in Figure 8.17. The first level is within the operating system, with one table for each process active on the system, while

Virtual Machine 1 Spool Table

Program	Status	Location	Real loc.	Size
A	Printed	1000	11000	400
B	Completed	2000	12000	200
C	Running	3000	13000	200
D	Completed	4000	14000	500

Virtual Machine 2 Spool Table

Program	Status	Location	Real loc.	Size
P	Running	1000	21000	400
Q	Completed	2000	22000	800

VMM Spool Table

VM	Program	Status	Real loc.	Size
1	A	Printed	30000	400
2	Q	Printing	31000	800
1	B	Waiting	31800	200
1	D	Waiting	30400	500

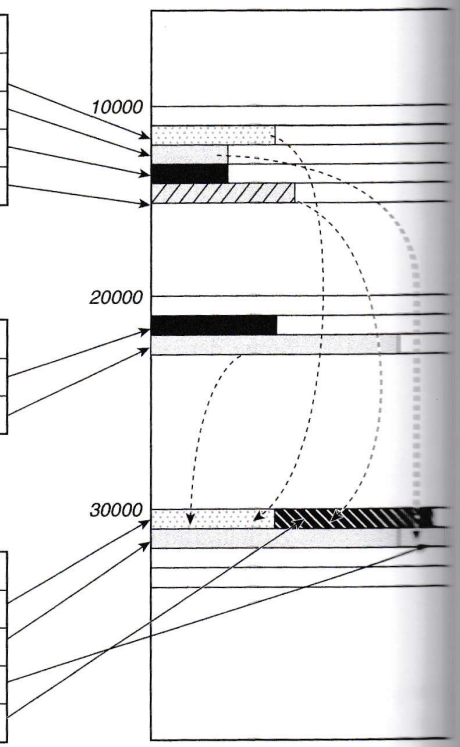


Figure 8.17 Spooling in a Virtual Machine System. *As programs are "completed," their spool areas are copied over to the VMM spool area. The corresponding entry in the VMM spool table is set to "waiting." At the end of printing, this status is changed in both tables to "printed." The spool area for the printed jobs can be recovered and reused.*

the second level is in the virtual machine monitor, with one table for each guest virtual machine. An operating system's request to print a spool buffer is intercepted by the VMM, which copies the buffer contents into one of its own spool buffers. This allows the VMM to schedule requests from different virtual machines on the same printer.

Various techniques could be employed to optimize the movement of large spool data between the virtual machine and the VMM. For example, the VMM could simply record the location of the spool data and its size, delaying the completion signal to the guest operating system until after the buffer has actually been printed. Another possibility is to have a single physical pool of buffers owned by the VMM, with each virtual machine having a virtual set of spool buffers allocated by the VMM. The practicality of this latter scheme depends on the ability of the VMM to detect such allocation of buffers, and on the ability of the guest operating system(s) to invoke services provided by the VMM.

The virtualization of printers of this kind was more appropriate to the older line printers, which were generally expensive and attached closely to the main computer. Today it is more common to find multiple printers accessible over a network. These printers already have the software to buffer and spool jobs from various nodes in the network. Virtualization of the printer is simple in such an environment as long as the virtual machine is connected to the network and has it own distinct network ID. The printer software in this case is unable to distinguish between requests from real machines and requests from virtual machines.

Nonexistent Physical Devices

Virtualization makes it possible to provide support for virtual devices "attached" to a virtual machine for which there is no corresponding physical device. For example, each virtual machine could be associated with a network adapter that is used for communicating with other virtual machines on the same platform, without requiring an actual network adapter to be available on the platform. In such a situation the virtual machine monitor can simply intercept the I/O requests coming from the device drivers and emulate the transmission of the network packet, buffering the packet at the device driver of the appropriate destination virtual machine and causing an interrupt to be handled by that virtual machine.

8.4.2 Virtualizing I/O Activity

The overall I/O process is illustrated in Figure 8.18. As discussed in Appendix Section A.4.3, an operating system abstracts most of the peculiarities of

VMotion, 532–535
VMware Virtual Platform, 426–435
Volatile memory location, 187–188
VT-x (Vanderpool), 436–442

Wiggins/Redstone system, 209
Windows operating system, 130–132
Win32, 133, 141
World Wide Grid Forum, 540
Wrappers, 9
Write barriers, 302
Write-protect table, 338

Xen system, 422

VIRTUAL MACHINES

VERSATILE PLATFORMS FOR SYSTEMS AND PROCESSES

JAMES E. SMITH, University of Wisconsin – Madison
RAVI NAIR, IBM Thomas J. Watson Research Center

Viva versatile VMs!
Hardware is hard, inflexible, produced by gnomes with sub-micron tools. Virtual machines wrap a layer of software around this hardware, and suddenly computers become flexible, malleable, and start doing new tricks: running multiple operating systems, executing several instruction sets, allowing running programs to switch machines, or even rejecting unsafe code. Virtual machines are changing the way in which computer architects, operating system designers, programming language implementers, and security experts think about computers and computing. Smith and Nair's book is an up-to-date overview of virtual machines and their many uses.
—Jim Larus, Microsoft Research

Virtual machines are everywhere. Jim Smith and Ravi Nair make abundantly clear the deep impact of this technology on the design of modern computer architectures, programming languages, operating systems, and security techniques. I highly recommend this book for anyone interested in the future of computing systems.
—Michael D. Smith, Harvard University

Historically, individual virtual machine techniques have been developed within the specific disciplines that employ them, making it difficult to see their common underlying relationships in a cohesive way. This book offers 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.

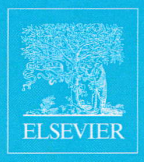
Virtual Machines carefully examines core topics of VM technology including instruction set emulation, dynamic program translation and optimization, high level language virtual machines and system virtual machines for both single-user systems and servers. It offers graduate students, researchers, and practicing professionals a unique resource that gathers and synthesizes the work from a number of fields into a single authoritative resource, defining a new and unified discipline.

Features

- Illustrates principles with references to several systems including the Shade simulation system, the FX!32 system, Dynamo/RIO, contemporary language VMs like Java and CLI, the Jikes Research Virtual Machine, the Transmeta Crusoe processor, IBM's AS/400 and z/VM systems, VMware's hosted VM, Intel's VT-x virtualization technology, and the multiprocessor virtualization system, Cellular Disco.
- Discusses emerging applications including security, the migration of computing environments between machines, and computational grids.

Computer Architecture

ISBN-13: 978-1-55860-910-5
ISBN-10: 1-55860-910-5



MORGAN KAUFMANN PUBLISHERS
AN IMPRINT OF ELSEVIER
www.mkp.com



LPN N I4SFT121R