

3e



MODERN OPERATING SYSTEMS

THIRD EDITION

ANDREW S. TANENBAUM

*Vrije Universiteit
Amsterdam, The Netherlands*



UPPER SADDLE RIVER, NEW JERSEY 07458

Library of Congress Cataloging-in-Publication Data on File.

Editorial Director, Computer Science, Engineering, and Advanced Mathematics: *Marcia J. Horton*
Executive Editor: *Tracy Dunkelberger*
Editorial Assistant: *Melinda Haggerty*
Associate Editor: *ReeAnne Davies*
Senior Managing Editor: *Scott Disanno*
Production Editor: *Irwin Zucker*
Cover Concept: *Andrew S. Tanenbaum and Tracy Dunkelberger*
Cover Design: *Tamara Newnam*
Cover Illustrator: *Steve Lefkowitz*
Interior design: *Andrew S. Tanenbaum*
Typesetting: *Andrew S. Tanenbaum*
Art Director: *Kenny Beck*
Art Editor: *Gregory Dulles*
Media Editor: *David Alick*
Manufacturing Manager: *Alan Fischer*
Manufacturing Buyer: *Lisa McDowell*
Marketing Manager: *Mack Patterson*



© 2008 Pearson Education, Inc.
Pearson Prentice Hall
Pearson Education, Inc.
Upper Saddle River, NJ 07458

All rights reserved. No part of this book may be reproduced in any form or by any means, without permission in writing from the publisher.

Pearson Prentice Hall™ is a trademark of Pearson Education, Inc.

The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Printed in the United States of America

10 9 8 7 6

ISBN 0-13-600663-9
978-0-13-600663-2

Pearson Education Ltd., London
Pearson Education Australia Pty. Ltd., Sydney
Pearson Education Singapore, Pte. Ltd.
Pearson Education North Asia Ltd., Hong Kong
Pearson Education Canada, Inc., Toronto
Pearson Educación de México, S.A. de C.V.
Pearson Education—Japan, Tokyo
Pearson Education Malaysia, Pte. Ltd.
Pearson Education, Inc., Upper Saddle River, New Jersey

1

INTRODUCTION

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

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

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

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

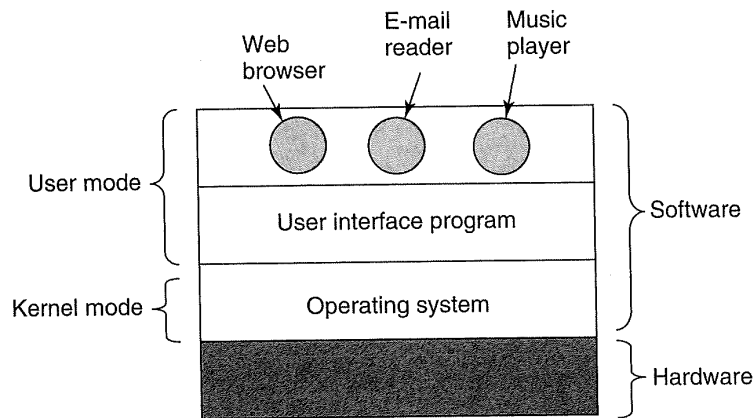


Figure 1-1. Where the operating system fits in.

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

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

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

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

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

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

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

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

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

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

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

1.1 WHAT IS AN OPERATING SYSTEM?

It is hard to pin down what an operating system is other than saying it is the software that runs in kernel mode—and even that is not always true. Part of the problem is that operating systems perform two basically unrelated functions: providing application programmers (and application programs, naturally) a clean abstract set of resources instead of the messy hardware ones and managing these

hardware resources. Depending on who is doing the talking, you might hear mostly about one function or the other. Let us now look at both.

1.1.1 The Operating System as an Extended Machine

The **architecture** (instruction set, memory organization, I/O, and bus structure) of most computers at the machine language level is primitive and awkward to program, especially for input/output. To make this point more concrete, consider how floppy disk I/O is done using the NEC PD765 compatible controller chips used on most Intel-based personal computers. (Throughout this book we will use the terms “floppy disk” and “diskette” interchangeably.) We use the floppy disk as an example, because, although it is obsolete, it is much simpler than a modern hard disk. The PD765 has 16 commands, each specified by loading between 1 and 9 bytes into a device register. These commands are for reading and writing data, moving the disk arm, and formatting tracks, as well as initializing, sensing, resetting, and recalibrating the controller and the drives.

The most basic commands are read and write, each of which requires 13 parameters, packed into 9 bytes. These parameters specify such items as the address of the disk block to be read, the number of sectors per track, the recording mode used on the physical medium, the intersector gap spacing, and what to do with a deleted-data-address-mark. If you do not understand this mumbo jumbo, do not worry; that is precisely the point—it is rather esoteric. When the operation is completed, the controller chip returns 23 status and error fields packed into 7 bytes. As if this were not enough, the floppy disk programmer must also be constantly aware of whether the motor is on or off. If the motor is off, it must be turned on (with a long startup delay) before data can be read or written. The motor cannot be left on too long, however, or the floppy disk will wear out. The programmer is thus forced to deal with the trade-off between long startup delays versus wearing out floppy disks (and losing the data on them).

Without going into the *real* details, it should be clear that the average programmer probably does not want to get too intimately involved with the programming of floppy disks (or hard disks, which are worse). Instead, what the programmer wants is a simple, high-level abstraction to deal with. In the case of disks, a typical abstraction would be that the disk contains a collection of named files. Each file can be opened for reading or writing, then read or written, and finally closed. Details such as whether or not recording should use modified frequency modulation and what the current state of the motor is should not appear in the abstraction presented to the application programmer.

Abstraction is the key to managing complexity. Good abstractions turn a nearly impossible task into two manageable ones. The first one of these is defining and implementing the abstractions. The second one is using these abstractions to solve the problem at hand. One abstraction that almost every computer user understands is the file. It is a useful piece of information, such as a digital photo,

saved e-mail message, or Web page. Dealing with photos, e-mails, and Web pages is easier than the details of disks, such as the floppy disk described above. The job of the operating system is to create good abstractions and then implement and manage the abstract objects thus created. In this book, we will talk a lot about abstractions. They are one of the keys to understanding operating systems.

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

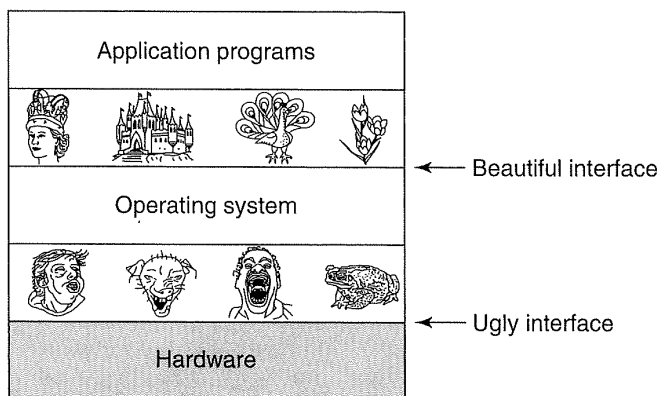


Figure 1-2. Operating systems turn ugly hardware into beautiful abstractions.

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

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

1.1.2 The Operating System as a Resource Manager

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

Modern operating systems allow multiple programs to run at the same time. Imagine what would happen if three programs running on some computer all tried to print their output simultaneously on the same printer. The first few lines of printout might be from program 1, the next few from program 2, then some from program 3, and so forth. The result would be chaos. The operating system can bring order to the potential chaos by buffering all the output destined for the printer on the disk. When one program is finished, the operating system can then copy its output from the disk file where it has been stored for the printer, while at the same time the other program can continue generating more output, oblivious to the fact that the output is not really going to the printer (yet).

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

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

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

1.2 HISTORY OF OPERATING SYSTEMS

Operating systems have been evolving through the years. In the following sections we will briefly look at a few of the highlights. Since operating systems have historically been closely tied to the architecture of the computers on which they run, we will look at successive generations of computers to see what their operating systems were like. This mapping of operating system generations to computer generations is crude, but it does provide some structure where there would otherwise be none.

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

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

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

1.2.1 The First Generation (1945–55) Vacuum Tubes

After Babbage’s unsuccessful efforts, little progress was made in constructing digital computers until World War II, which stimulated an explosion of activity. Prof. John Atanasoff and his graduate student Clifford Berry built what is now

1.7 OPERATING SYSTEM STRUCTURE

Now that we have seen what operating systems look like on the outside (i.e., the programmer's interface), it is time to take a look inside. In the following sections, we will examine six different structures that have been tried, in order to get some idea of the spectrum of possibilities. These are by no means exhaustive, but they give an idea of some designs that have been tried in practice. The six designs are monolithic systems, layered systems, microkernels, client-server systems, virtual machines, and exokernels.

1.7.1 Monolithic Systems

By far the most common organization, in this approach the entire operating system runs as a single program in kernel mode. The operating system is written as a collection of procedures, linked together into a single large executable binary program. When this technique is used, each procedure in the system is free to call any other one, if the latter provides some useful computation that the former needs. Having thousands of procedures that can call each other without restriction often leads to an unwieldy and difficult to understand system.

To construct the actual object program of the operating system when this approach is used, one first compiles all the individual procedures (or the files containing the procedures) and then binds them all together into a single executable file using the system linker. In terms of information hiding, there is essentially none—every procedure is visible to every other procedure (as opposed to a structure containing modules or packages, in which much of the information is hidden away inside modules, and only the officially designated entry points can be called from outside the module).

Even in monolithic systems, however, it is possible to have some structure. The services (system calls) provided by the operating system are requested by putting the parameters in a well-defined place (e.g., on the stack) and then executing a trap instruction. This instruction switches the machine from user mode to kernel mode and transfers control to the operating system, shown as step 6 in Fig. 1-17. The operating system then fetches the parameters and determines which system call is to be carried out. After that, it indexes into a table that contains in slot k a pointer to the procedure that carries out system call k (step 7 in Fig. 1-17).

This organization suggests a basic structure for the operating system:

1. A main program that invokes the requested service procedure.
2. A set of service procedures that carry out the system calls.
3. A set of utility procedures that help the service procedures.

In this model, for each system call there is one service procedure that takes care of it and executes it. The utility procedures do things that are needed by several

service procedures, such as fetching data from user programs. This division of the procedures into three layers is shown in Fig. 1-24.

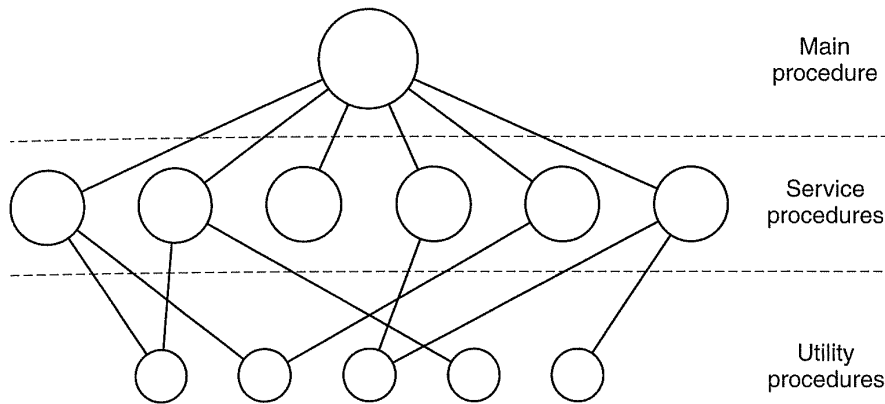


Figure 1-24. A simple structuring model for a monolithic system.

In addition to the core operating system that is loaded when the computer is booted, many operating systems support loadable extensions, such as I/O device drivers and file systems. These components are loaded on demand.

1.7.2 Layered Systems

A generalization of the approach of Fig. 1-24 is to organize the operating system as a hierarchy of layers, each one constructed upon the one below it. The first system constructed in this way was the THE system built at the Technische Hogeschool Eindhoven in the Netherlands by E. W. Dijkstra (1968) and his students. The THE system was a simple batch system for a Dutch computer, the Electrologica X8, which had 32K of 27-bit words (bits were expensive back then).

The system had six layers, as shown in Fig. 1-25. Layer 0 dealt with allocation of the processor, switching between processes when interrupts occurred or timers expired. Above layer 0, the system consisted of sequential processes, each of which could be programmed without having to worry about the fact that multiple processes were running on a single processor. In other words, layer 0 provided the basic multiprogramming of the CPU.

Layer 1 did the memory management. It allocated space for processes in main memory and on a 512K word drum used for holding parts of processes (pages) for which there was no room in main memory. Above layer 1, processes did not have to worry about whether they were in memory or on the drum; the layer 1 software took care of making sure pages were brought into memory whenever they were needed.

Layer 2 handled communication between each process and the operator console (that is, the user). On top of this layer each process effectively had its own

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

Figure 1-25. Structure of the THE operating system.

operator console. Layer 3 took care of managing the I/O devices and buffering the information streams to and from them. Above layer 3 each process could deal with abstract I/O devices with nice properties, instead of real devices with many peculiarities. Layer 4 was where the user programs were found. They did not have to worry about process, memory, console, or I/O management. The system operator process was located in layer 5.

A further generalization of the layering concept was present in the MULTICS system. Instead of layers, MULTICS was described as having a series of concentric rings, with the inner ones being more privileged than the outer ones (which is effectively the same thing). When a procedure in an outer ring wanted to call a procedure in an inner ring, it had to make the equivalent of a system call, that is, a TRAP instruction whose parameters were carefully checked for validity before allowing the call to proceed. Although the entire operating system was part of the address space of each user process in MULTICS, the hardware made it possible to designate individual procedures (memory segments, actually) as protected against reading, writing, or executing.

Whereas the THE layering scheme was really only a design aid, because all the parts of the system were ultimately linked together into a single executable program, in MULTICS, the ring mechanism was very much present at run time and enforced by the hardware. The advantage of the ring mechanism is that it can easily be extended to structure user subsystems. For example, a professor could write a program to test and grade student programs and run this program in ring n , with the student programs running in ring $n + 1$ so that they could not change their grades.

1.7.3 Microkernels

With the layered approach, the designers have a choice where to draw the kernel-user boundary. Traditionally, all the layers went in the kernel, but that is not necessary. In fact, a strong case can be made for putting as little as possible in

kernel mode because bugs in the kernel can bring down the system instantly. In contrast, user processes can be set up to have less power so that a bug there may not be fatal.

Various researchers have studied the number of bugs per 1000 lines of code (e.g., Basilli and Perricone, 1984; and Ostrand and Weyuker, 2002). Bug density depends on module size, module age, and more, but a ballpark figure for serious industrial systems is ten bugs per thousand lines of code. This means that a monolithic operating system of five million lines of code is likely to contain something like 50,000 kernel bugs. Not all of these are fatal, of course, since some bugs may be things like issuing an incorrect error message in a situation that rarely occurs. Nevertheless, operating systems are sufficiently buggy that computer manufacturers put reset buttons on them (often on the front panel), something the manufacturers of TV sets, stereos, and cars do not do, despite the large amount of software in these devices.

The basic idea behind the microkernel design is to achieve high reliability by splitting the operating system up into small, well-defined modules, only one of which—the microkernel—runs in kernel mode and the rest run as relatively powerless ordinary user processes. In particular, by running each device driver and file system as a separate user process, a bug in one of these can crash that component, but cannot crash the entire system. Thus a bug in the audio driver will cause the sound to be garbled or stop, but will not crash the computer. In contrast, in a monolithic system with all the drivers in the kernel, a buggy audio driver can easily reference an invalid memory address and bring the system to a grinding halt instantly.

Many microkernels have been implemented and deployed (Accetta et al., 1986; Haertig et al., 1997; Heiser et al., 2006; Herder et al., 2006; Hildebrand, 1992; Kirsch et al., 2005; Liedtke, 1993, 1995, 1996; Pike et al., 1992; and Zuberi et al., 1999). They are especially common in real-time, industrial, avionics, and military applications that are mission critical and have very high reliability requirements. A few of the better-known microkernels are Integrity, K42, L4, PikeOS, QNX, Symbian, and MINIX 3. We will now give a brief overview of MINIX 3, which has taken the idea of modularity to the limit, breaking most of the operating system up into a number of independent user-mode processes. MINIX 3 is a POSIX conformant, open-source system freely available at www.minix3.org (Herder et al., 2006a; Herder et al., 2006b).

The MINIX 3 microkernel is only about 3200 lines of C and 800 lines of assembler for very low-level functions such as catching interrupts and switching processes. The C code manages and schedules processes, handles interprocess communication (by passing messages between processes), and offers a set of about 35 kernel calls to allow the rest of the operating system to do its work. These calls perform functions like hooking handlers to interrupts, moving data between address spaces, and installing new memory maps for newly created processes. The process structure of MINIX 3 is shown in Fig. 1-26, with the kernel call

handlers labeled *Sys*. The device driver for the clock is also in the kernel because the scheduler interacts closely with it. All the other device drivers run as separate user processes.

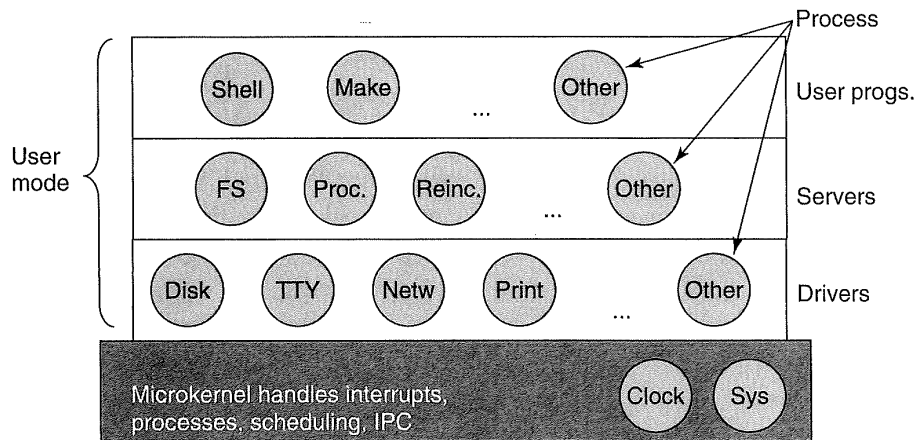


Figure 1-26. Structure of the MINIX 3 system.

Outside the kernel, the system is structured as three layers of processes all running in user mode. The lowest layer contains the device drivers. Since they run in user mode, they do not have physical access to the I/O port space and cannot issue I/O commands directly. Instead, to program an I/O device, the driver builds a structure telling which values to write to which I/O ports and makes a kernel call telling the kernel to do the write. This approach means that the kernel can check to see that the driver is writing (or reading) from I/O it is authorized to use. Consequently, (and unlike a monolithic design), a buggy audio driver cannot accidentally write on the disk.

Above the drivers is another user-mode layer containing the servers, which do most of the work of the operating system. One or more file servers manage the file system(s), the process manager creates, destroys, and manages processes, and so on. User programs obtain operating system services by sending short messages to the servers asking for the POSIX system calls. For example, a process needing to do a read sends a message to one of the file servers telling it what to read.

One interesting server is the **reincarnation server**, whose job is to check if the other servers and drivers are functioning correctly. In the event that a faulty one is detected, it is automatically replaced without any user intervention. In this way the system is self healing and can achieve high reliability.

The system has many restrictions limiting the power of each process. As mentioned, drivers can only touch authorized I/O ports, but access to kernel calls is also controlled on a per process basis, as is the ability to send messages to other processes. Processes can also grant limited permission for other processes to have the kernel access their address spaces. As an example, a file system can grant

permission for the disk driver to let the kernel put a newly read in disk block at a specific address within the file system's address space. The sum total of all these restrictions is that each driver and server has exactly the power to do its work and nothing more, thus greatly limiting the damage a buggy component can do.

An idea somewhat related to having a minimal kernel is to put the **mechanism** for doing something in the kernel but not the **policy**. To make this point better, consider the scheduling of processes. A relatively simple scheduling algorithm is to assign a priority to every process and then have the kernel run the highest-priority process that is runnable. The mechanism—in the kernel—is to look for the highest-priority process and run it. The policy—assigning priorities to processes—can be done by user-mode processes. In this way policy and mechanism can be decoupled and the kernel can be made smaller.

1.7.4 Client-Server Model

A slight variation of the microkernel idea is to distinguish two classes of processes, the **servers**, each of which provides some service, and the **clients**, which use these services. This model is known as the **client-server** model. Often the lowest layer is a microkernel, but that is not required. The essence is the presence of client processes and server processes.

Communication between clients and servers is often by message passing. To obtain a service, a client process constructs a message saying what it wants and sends it to the appropriate service. The service then does the work and sends back the answer. If the client and server run on the same machine, certain optimizations are possible, but conceptually, we are talking about message passing here.

An obvious generalization of this idea is to have the clients and servers run on different computers, connected by a local or wide-area network, as depicted in Fig. 1-27. Since clients communicate with servers by sending messages, the clients need not know whether the messages are handled locally on their own machines, or whether they are sent across a network to servers on a remote machine. As far as the client is concerned, the same thing happens in both cases: requests are sent and replies come back. Thus the client-server model is an abstraction that can be used for a single machine or for a network of machines.

Increasingly many systems involve users at their home PCs as clients and large machines elsewhere running as servers. In fact, much of the Web operates this way. A PC sends a request for a Web page to the server and the Web page comes back. This is a typical use of the client-server model in a network.

1.7.5 Virtual Machines

The initial releases of OS/360 were strictly batch systems. Nevertheless, many 360 users wanted to be able to work interactively at a terminal, so various groups, both inside and outside IBM, decided to write timesharing systems for it. The

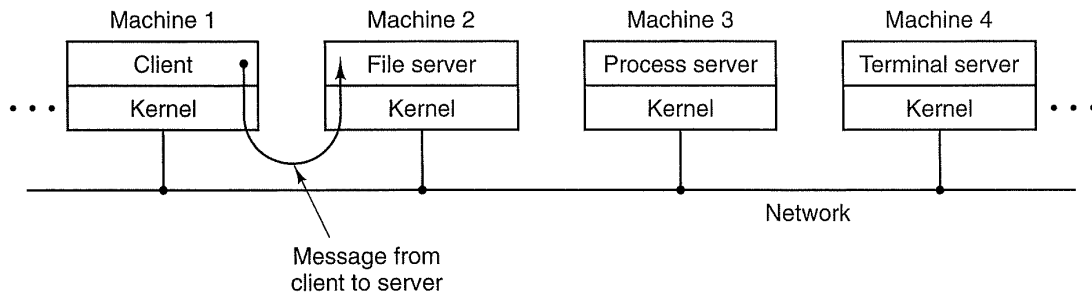


Figure 1-27. The client-server model over a network.

official IBM timesharing system, TSS/360, was delivered late, and when it finally arrived it was so big and slow that few sites converted to it. It was eventually abandoned after its development had consumed some \$50 million (Graham, 1970). But a group at IBM's Scientific Center in Cambridge, Massachusetts, produced a radically different system that IBM eventually accepted as a product. A linear descendant of it, called **z/VM**, is now widely used on IBM's current mainframes, the zSeries, which are heavily used in large corporate data centers, for example, as e-commerce servers that handle hundreds or thousands of transactions per second and use databases whose sizes run to millions of gigabytes.

VM/370

This system, originally called CP/CMS and later renamed VM/370 (Seawright and MacKinnon, 1979), was based on an astute observation: a timesharing system provides (1) multiprogramming and (2) an extended machine with a more convenient interface than the bare hardware. The essence of VM/370 is to completely separate these two functions.

The heart of the system, known as the **virtual machine monitor**, runs on the bare hardware and does the multiprogramming, providing not one, but several virtual machines to the next layer up, as shown in Fig. 1-28. However, unlike all other operating systems, these virtual machines are not extended machines, with files and other nice features. Instead, they are *exact* copies of the bare hardware, including kernel/user mode, I/O, interrupts, and everything else the real machine has.

Because each virtual machine is identical to the true hardware, each one can run any operating system that will run directly on the bare hardware. Different virtual machines can, and frequently do, run different operating systems. On the original VM/370 system, some ran OS/360 or one of the other large batch or transaction processing operating systems, while other ones ran a single-user, interactive system called **CMS (Conversational Monitor System)** for interactive timesharing users. The latter was popular with programmers.

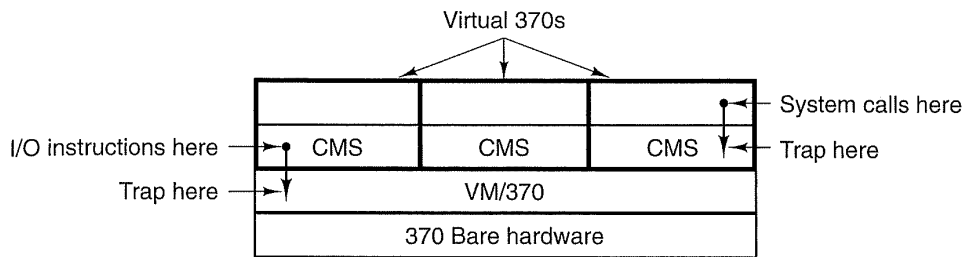


Figure 1-28. The structure of VM/370 with CMS.

When a CMS program executed a system call, the call was trapped to the operating system in its own virtual machine, not to VM/370, just as it would if it were running on a real machine instead of a virtual one. CMS then issued the normal hardware I/O instructions for reading its virtual disk or whatever was needed to carry out the call. These I/O instructions were trapped by VM/370, which then performed them as part of its simulation of the real hardware. By completely separating the functions of multiprogramming and providing an extended machine, each of the pieces could be much simpler, more flexible, and much easier to maintain.

In its modern incarnation, z/VM is usually used to run multiple complete operating systems rather than stripped-down single-user systems like CMS. For example, the zSeries is capable of running one or more Linux virtual machines along with traditional IBM operating systems.

Virtual Machines Rediscovered

While IBM has had a virtual machine product available for four decades, and a few other companies, including Sun Microsystems and Hewlett-Packard, have recently added virtual machine support to their high-end enterprise servers, the idea of virtualization has largely been ignored in the PC world until recently. But in the past few years, a combination of new needs, new software, and new technologies have combined to make it a hot topic.

First the needs. Many companies have traditionally run their mail servers, Web servers, FTP servers, and other servers on separate computers, sometimes with different operating systems. They see virtualization as a way to run them all on the same machine without having a crash of one server bring down the rest.

Virtualization is also popular in the Web hosting world. Without it, Web hosting customers are forced to choose between **shared hosting** (which just gives them a login account on a Web server, but no control over the server software) and dedicated hosting (which gives them their own machine, which is very flexible but not cost effective for small to medium Websites). When a Web hosting

company offers virtual machines for rent, a single physical machine can run many virtual machines, each of which appears to be a complete machine. Customers who rent a virtual machine can run whatever operating system and software they want to, but at a fraction of the cost of a dedicated server (because the same physical machine supports many virtual machines at the same time).

Another use of virtualization is for end users who want to be able to run two or more operating systems at the same time, say Windows and Linux, because some of their favorite application packages run on one and some run on the other. This situation is illustrated in Fig. 1-29(a), where the term “virtual machine monitor” has been renamed type 1 **hypervisor** in recent years.

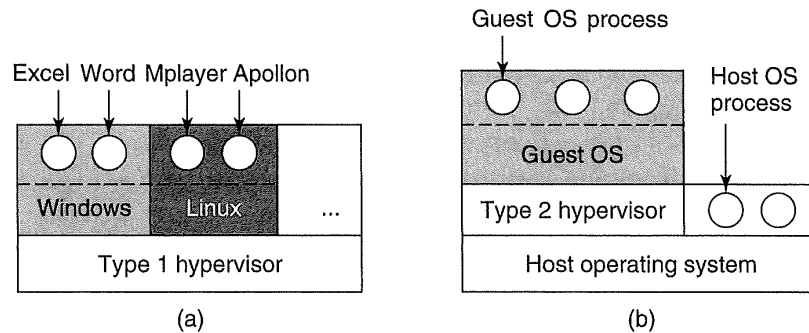


Figure 1-29. (a) A type 1 hypervisor. (b) A type 2 hypervisor.

Now the software. While no one disputes the attractiveness of virtual machines, the problem was implementation. In order to run virtual machine software on a computer, its CPU must be virtualizable (Popek and Goldberg, 1974). In a nutshell, here is the problem. When an operating system running on a virtual machine (in user mode) executes a privileged instruction), such as modifying the PSW or doing I/O, it is essential that the hardware trap to the virtual machine monitor so the instruction can be emulated in software. On some CPUs—notably the Pentium, its predecessors, and its clones—attempts to execute privileged instructions in user mode are just ignored. This property made it impossible to have virtual machines on this hardware, which explains the lack of interest in the PC world. Of course, there were interpreters for the Pentium that ran on the Pentium, but with a performance loss of typically 5–10x, they were not useful for serious work.

This situation changed as a result of several academic research projects in the 1990s, notably Disco at Stanford (Bugnion et al., 1997), which led to commercial products (e.g., VMware Workstation) and a revival of interest in virtual machines. VMware Workstation is a type 2 hypervisor, which is shown in Fig. 1-29(b). In contrast to type 1 hypervisors, which run on the bare metal, type 2 hypervisors run as application programs on top of Windows, Linux, or some other operating system, known as the **host operating system**. After a type 2 hypervisor is started, it

reads the installation CD-ROM for the chosen **guest operating system** and installs on a virtual disk, which is just a big file in the host operating system's file system.

When the guest operating system is booted, it does the same thing it does on the actual hardware, typically starting up some background processes and then a GUI. Some hypervisors translate the binary programs of the guest operating system block by block, replacing certain control instructions with hypervisor calls. The translated blocks are then executed and cached for subsequent use.

A different approach to handling control instructions is to modify the operating system to remove them. This approach is not true virtualization, but **paravirtualization**. We will discuss virtualization in more detail in Chap. 8.

The Java Virtual Machine

Another area where virtual machines are used, but in a somewhat different way, is for running Java programs. When Sun Microsystems invented the Java programming language, it also invented a virtual machine (i.e., a computer architecture) called the **JVM (Java Virtual Machine)**. The Java compiler produces code for JVM, which then typically is executed by a software JVM interpreter. The advantage of this approach is that the JVM code can be shipped over the Internet to any computer that has a JVM interpreter and run there. If the compiler had produced SPARC or Pentium binary programs, for example, they could not have been shipped and run anywhere as easily. (Of course, Sun could have produced a compiler that produced SPARC binaries and then distributed a SPARC interpreter, but JVM is a much simpler architecture to interpret.) Another advantage of using JVM is that if the interpreter is implemented properly, which is not completely trivial, incoming JVM programs can be checked for safety and then executed in a protected environment so they cannot steal data or do any damage.

1.7.6 Exokernels

Rather than cloning the actual machine, as is done with virtual machines, another strategy is partitioning it, in other words, giving each user a subset of the resources. Thus one virtual machine might get disk blocks 0 to 1023, the next one might get blocks 1024 to 2047, and so on.

At the bottom layer, running in kernel mode, is a program called the **exokernel** (Engler et al., 1995). Its job is to allocate resources to virtual machines and then check attempts to use them to make sure no machine is trying to use somebody else's resources. Each user-level virtual machine can run its own operating system, as on VM/370 and the Pentium virtual 8086s, except that each one is restricted to using only the resources it has asked for and been allocated.

The advantage of the exokernel scheme is that it saves a layer of mapping. In the other designs, each virtual machine thinks it has its own disk, with blocks

running from 0 to some maximum, so the virtual machine monitor must maintain tables to remap disk addresses (and all other resources). With the exokernel, this remapping is not needed. The exokernel need only keep track of which virtual machine has been assigned which resource. This method still has the advantage of separating the multiprogramming (in the exokernel) from the user operating system code (in user space), but with less overhead, since all the exokernel has to do is keep the virtual machines out of each other's hair.

1.8 THE WORLD ACCORDING TO C

Operating systems are normally large C (or sometimes C++) programs consisting of many pieces written by many programmers. The environment used for developing operating systems is very different from what individuals (such as students) are used to when writing small Java programs. This section is an attempt to give a very brief introduction to the world of writing an operating system for small-time Java programmers.

1.8.1 The C Language

This is not a guide to C, but a short summary of some of the key differences between C and Java. Java is based on C, so there are many similarities between the two. Both are imperative languages with data types, variables, and control statements, for example. The primitive data types in C are integers (including short and long ones), characters, and floating-point numbers. Composite data types can be constructed using arrays, structures, and unions. The control statements in C are similar to those in Java, including if, switch, for, and while statements. Functions and parameters are roughly the same in both languages.

One feature that C has that Java does not is explicit pointers. A **pointer** is a variable that points to (i.e., contains the address of) a variable or data structure. Consider the statements

```
char c1, c2, *p;  
c1 = 'c';  
p = &c1;  
c2 = *p;
```

which declare *c1* and *c2* to be character variables and *p* to be a variable that points to (i.e., contains the address of) a character. The first assignment stores the ASCII code for the character 'c' in the variable *c1*. The second one assigns the address of *c1* to the pointer variable *p*. The third one assigns the contents of the variable pointed to by *p* to the variable *c2*, so after these statements are executed, *c2* also contains the ASCII code for 'c'. In theory, pointers are typed, so you are not supposed to assign the address of a floating-point number to a character pointer, but

Eager et al. (1986) constructed an analytical queueing model of this algorithm. Using this model, it was established that the algorithm behaves well and is stable under a wide range of parameters, including various threshold values, transfer costs, and probe limits.

Nevertheless, it should be observed that under conditions of heavy load, all machines will constantly send probes to other machines in a futile attempt to find one that is willing to accept more work. Few processes will be off loaded, but considerable overhead may be incurred in the attempt to do so.

A Receiver-Initiated Distributed Heuristic Algorithm

A complementary algorithm to the one given above, which is initiated by an overloaded sender, is one initiated by an underloaded receiver, as shown in Fig. 8-25(b). With this algorithm, whenever a process finishes, the system checks to see if it has enough work. If not, it picks some machine at random and asks it for work. If that machine has nothing to offer, a second, and then a third machine is asked. If no work is found with N probes, the node temporarily stops asking, does any work it has queued up, and tries again when the next process finishes. If no work is available, the machine goes idle. After some fixed time interval, it begins probing again.

An advantage of this algorithm is that it does not put extra load on the system at critical times. The sender-initiated algorithm makes large numbers of probes precisely when the system can least tolerate it—when it is heavily loaded. With the receiver-initiated algorithm, when the system is heavily loaded, the chance of a machine having insufficient work is small. However, when this does happen, it will be easy to find work to take over. Of course, when there is little work to do, the receiver-initiated algorithm creates considerable probe traffic as all the unemployed machines desperately hunt for work. However, it is far better to have the overhead go up when the system is underloaded than when it is overloaded.

It is also possible to combine both of these algorithms and have machines try to get rid of work when they have too much, and try to acquire work when they do not have enough. Furthermore, machines can perhaps improve on random polling by keeping a history of past probes to determine if any machines are chronically underloaded or overloaded. One of these can be tried first, depending on whether the initiator is trying to get rid of work or acquire it.

8.3 VIRTUALIZATION

In some situations, an organization has a multicomputer but does not actually want it. A common example is where a company has an e-mail server, a Web server, an FTP server, some e-commerce servers, and others. These all run on different computers in the same equipment rack, all connected by a high-speed network, in other words, a multicomputer. In some cases, all these servers run on

separate machines because one machine cannot handle the load, but in many other cases the primary reason not to run all these services as processes on the same machine is reliability: management simply does not trust the operating system to run 24 hours a day, 365 or 366 days a year, with no failures. By putting each service on a separate computer, if one of the servers crashes, at least the other ones are not affected. While fault tolerance is achieved this way, this solution is expensive and hard to manage because so many machines are involved.

What to do? Virtual machine technology, often just called **virtualization**, which is more than 40 years old, has been proposed as a solution, as we discussed in Sec. 1.7.5. This technology allows a single computer to host multiple virtual machines, each potentially running a different operating system. The advantage of this approach is that a failure in one virtual machine does not automatically bring down any others. On a virtualized system, different servers can run on different virtual machines, thus maintaining the partial failure model that a multicomputer has, but at a much lower cost and with easier maintainability.

Of course, consolidating servers like this is like putting all of your eggs in one basket. If the server running all the virtual machines fails, the result is even more catastrophic than a single dedicated server crashing. The reason virtualization works however, is that most service outages are not due to faulty hardware, but due to bloated, unreliable, buggy software, especially operating systems. With virtual machine technology, the only software running in kernel mode is the hypervisor, which has two orders of magnitude fewer lines of code than a full operating system, and thus two orders of magnitude fewer bugs.

Running software in virtual machines has other advantages in addition to strong isolation. One of them is that having fewer physical machines saves money on hardware and electricity and takes up less office space. For a company such as Amazon, Yahoo, Microsoft, or Google, which may have hundreds of thousands of servers doing a huge variety of different tasks, reducing the physical demands on their data centers represents a huge cost savings. Typically, in large companies, individual departments or groups think of an interesting idea and then go out and buy a server to implement it. If the idea catches on and hundreds or thousands of servers are needed, the corporate data center expands. It is often hard to move the software to existing machines because each application often needs a different version of the operating system, its own libraries, configuration files, and more. With virtual machines, each application can take its own environment with it.

Another advantage of virtual machines is that checkpointing and migrating virtual machines (e.g., for load balancing across multiple servers) is much easier than migrating processes running on a normal operating system. In the latter case, a fair amount of critical state information about every process is kept in operating system tables, including information relating to open files, alarms, signal handlers, and more. When migrating a virtual machine, all that has to be moved is the memory image, since all the operating system tables move too.

Another use for virtual machines is to run legacy applications on operating systems (or operating system versions) no longer supported or which do not work on current hardware. These can run at the same time and on the same hardware as current applications. In fact, the ability to run at the same time applications that use different operating systems is a big argument in favor of virtual machines.

Yet another important use of virtual machines is software development. A programmer who wants to make sure his software works on Windows 98, Windows 2000, Windows XP, Windows Vista, several versions of Linux, FreeBSD, OpenBSD, NetBSD, and Mac OS X no longer has to get a dozen computers and install different operating systems on all of them. Instead he merely create a dozen virtual machines on a single computer and installs different operating systems on each one. Of course, the programmer could have partitioned the hard disk and installed a different operating system in each partition, but this approach is more difficult. First of all, standard PCs support only four primary disk partitions, no matter how big the disk is. Second, although a multiboot program could be installed in the boot block, it would be necessary to reboot the computer to work on a new operating system. With virtual machines, all of them can run at once, since they are really just glorified processes.

8.3.1 Requirements for Virtualization

As we saw in Chap. 1, there are two approaches to virtualization. One kind of hypervisor, dubbed a **type 1 hypervisor** (or **virtual machine monitor**) is illustrated in Fig. 1-29(a). In reality, it is the operating system, since it is the only program running in kernel mode. Its job is to support multiple copies of the actual hardware, called **virtual machines**, similar to the processes a normal operating system supports. In contrast, a **type 2 hypervisor**, shown in Fig. 1-29(b), is a completely different kind of animal. It is just a user program running on, say, Windows or Linux that “interprets” the machine’s instruction set, which also creates a virtual machine. We put “interprets” in quotes because usually chunks of code are processed in a certain way and then cached and executed directly to improve performance, but in principle, full interpretation would work, albeit slowly. The operating system running on top of the hypervisor in both cases is called the **guest operating system**. In the case of a type 2 hypervisor, the operating system running on the hardware is called the **host operating system**.

It is important to realize that in both cases, the virtual machines must act just like the real hardware. In particular, it must be possible to boot them like real machines and install arbitrary operating systems on them, just as can be done on the real hardware. It is the task of the hypervisor to provide this illusion and to do it efficiently (without being a complete interpreter).

The reason for the two types has to do with defects in the Intel 386 architecture that were slavishly carried forward into new CPUs for 20 years in the name of backward compatibility. In a nutshell, every CPU with kernel mode and user

mode has a set of instructions that may only be executed in kernel mode, such as instructions that do I/O, change the MMU settings, and so on. In their classic work on virtualization, Popek and Goldberg (1974) called these **sensitive instructions**. There is also a set of instructions that cause a trap if executed in user mode. Popek and Goldberg called these **privileged instructions**. Their paper stated for the first time that a machine is virtualizable only if the sensitive instructions are a subset of the privileged instructions. In simpler language, if you try to do something in user mode that you should not be doing in user mode, the hardware should trap. Unlike the IBM/370, which had this property, the 386 did not. Quite a few sensitive 386 instructions were ignored if executed in user mode. For example, the POPF instruction replaces the flags register, which changes the bit that enables/disables interrupts. In user mode, this bit is simply not changed. As a consequence, the 386 and its successors could not be virtualized, so they could not support a type 1 hypervisor.

Actually, the situation is slightly worse than sketched. In addition to the problems with instructions that fail to trap in user mode, there are instructions that can read sensitive state in user mode without causing a trap. For example, on the Pentium, a program can determine whether it is running in user mode or kernel mode by reading its code segment selector. An operating system that did this and discovered that it was actually in user mode, might make an incorrect decision based on this information.

This problem was solved when Intel and AMD introduced virtualization in their CPUs starting in 2005. On the Intel Core 2 CPUs it is called **VT (Virtualization Technology)**; On the AMD Pacific CPUs it is called **SVM (Secure Virtual Machine)**. We will use the term VT in a generic sense below. Both were inspired by the IBM VM/370 work, but they are slightly different. The basic idea is to create containers in which virtual machines can be run. When a guest operating system is started up in a container, it continues to run there until it causes an exception and traps to the hypervisor, for example, by executing an I/O instruction. The set of operations that trap is controlled by a hardware bitmap set by the hypervisor. With these extensions the classical trap-and-emulate virtual machine approach becomes possible.

8.3.2 Type 1 Hypervisors

Virtualizability is an important issue, so let us examine it a more closely. In Fig. 8-26 we see a type 1 hypervisor supporting one virtual machine. Like all type 1 hypervisors, it runs on the bare metal. The virtual machine runs as a user process in user mode, and as such, is not allowed to execute sensitive instructions. The virtual machine runs a guest operating system that thinks it is in kernel mode, although, of course, it is really in user mode. We will call this **virtual kernel mode**. The virtual machine also runs user processes, which think they are in user mode (and really are in user mode).

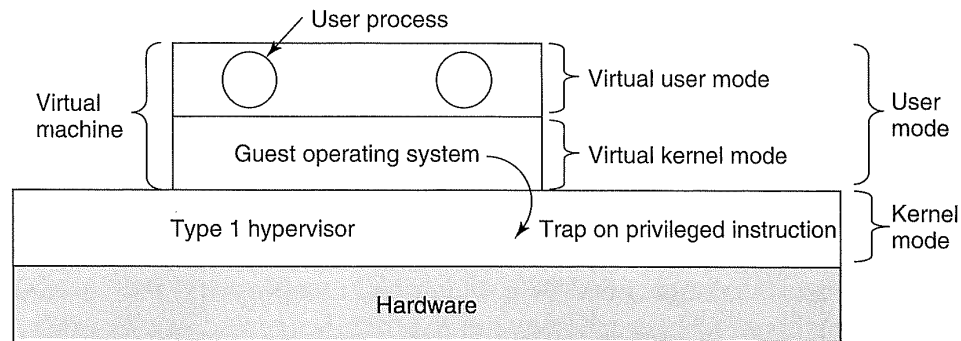


Figure 8-26. When the operating system in a virtual machine executes a kernel-only instruction, it traps to the hypervisor if virtualization technology is present.

What happens when the operating system (which thinks it is kernel mode) executes a sensitive instruction (one allowed only in kernel mode)? On CPUs without VT, the instruction fails and the operating system usually crashes. This makes true virtualization impossible. One could certainly argue that all sensitive instructions should always trap when executed in user mode, but that is not how the 386 and its non-VT successors worked.

On CPUs with VT, when the guest operating system executes a sensitive instruction, a trap to the kernel occurs, as illustrated in Fig. 8-26. The hypervisor can then inspect the instruction to see if it was issued by the guest operating system in the virtual machine or by a user program in the virtual machine. In the former case, it arranges for the instruction to be carried out; in the latter case, it emulates what the real hardware would do when confronted with a sensitive instruction executed in user mode. If the virtual machine does not have VT, the instruction is typically ignored; if it does have VT, it traps to the guest operating system running in the virtual machine.

8.3.3 Type 2 Hypervisors

Building a virtual machine system is relatively straightforward when VT is available, but what did people do before that? Clearly, running a full operating system in a virtual machine would not work because (some of) the sensitive instructions would just be ignored, causing the system to fail. Instead what happened was the invention of what are now called **type 2 hypervisors**, as illustrated in Fig. 1-29(b). The first of these was **VMware** (Adams and Agesen, 2006; and Waldspurger, 2002), which was the outgrowth of the DISCO research project at Stanford University (Bugnion et al., 1997). VMware runs as an ordinary user program on top of a host operating system such as Windows or Linux. When it starts for the first time, it acts like a newly booted computer and expects to find a CD-

ROM containing an operating system in the CD-ROM drive. It then installs the operating system to its **virtual disk** (really just a Windows or Linux file) by running the installation program found on the CD-ROM. Once the guest operating system is installed on the virtual disk, it can be booted at run.

Now let us look at how VMware works in a bit more detail. When executing a Pentium binary program, whether obtained from the installation CD-ROM or from the virtual disk, it scans the code first looking for **basic blocks**, that is, straight runs of instructions ending in a jump, call, trap, or other instruction that changes the flow of control. By definition, no basic block contains any instruction that modifies the program counter except the last one. The basic block is inspected to see if it contains any sensitive instructions (in the Popek and Goldberg sense). If so, each one is replaced with a call to a VMware procedure that handles it. The final instruction is also replaced with a call into VMware.

Once these steps have been taken, the basic block is cached inside VMware and then executed. A basic block not containing any sensitive instructions will execute exactly as fast under VMware as it will on the bare machine—because it is running on the bare machine. Sensitive instructions are caught this way and emulated. This technique is known as **binary translation**.

After the basic block has completed executing, control is returned to VMware, which locates its successor. If the successor has already been translated, it can be executed immediately. If it has not been, it is first translated, cached, then executed. Eventually, most of the program will be in the cache and run at close to full speed. Various optimizations are used, for example, if a basic block ends by jumping to (or calling) another one, the final instruction can be replaced by a jump or call directly to the translated basic block, eliminating all overhead associated with finding the successor block. Also, there is no need to replace sensitive instructions in user programs; the hardware will just ignore them anyway.

It should now be clear why type 2 hypervisors work, even on unvirtualizable hardware: all sensitive instructions are replaced by calls to procedures that emulate these instructions. No sensitive instructions issued by the guest operating system are ever executed by the true hardware. They are turned into calls to the hypervisor, which then emulates them.

One might naively expect that CPUs with VT would greatly outperform the software techniques used by the type 2 hypervisors, but measurements show a mixed picture (Adams and Agesen, 2006). It turns out that the trap-and-emulate approach used by VT hardware generates a lot of traps, and traps are very expensive on modern hardware because they ruin CPU caches, TLBs, and branch prediction tables internal to the CPU. In contrast, when sensitive instructions are replaced by calls to VMware procedures within the executing process, none of this context switching overhead is incurred. As Adams and Agesen show, depending on the workload, sometimes software beats hardware. For this reason, some type 1 hypervisors do binary translation for performance reasons, even though the software will execute correctly without it.

8.3.4 Paravirtualization

Both type 1 and type 2 hypervisors work with unmodified guest operating systems, but have to jump through hoops to get reasonable performance. A different approach that is becoming popular is to modify the source code of the guest operating system so that instead of executing sensitive instructions at all, it makes **hypervisor calls**. In effect the guest operating system is acting like a user program making system calls to the operating system (the hypervisor). When this route is taken, the hypervisor must define an interface consisting of a set of procedure calls that guest operating systems can use. This set of calls forms what is effectively an **API (Application Programming Interface)** even though it is an interface for use by guest operating systems, not application programs.

Going one step further, by removing all the sensitive instructions from the operating system and just having it make hypervisor calls to get system services like I/O, we have turned the hypervisor into a microkernel, like that of Fig. 1-26. A guest operating system from which (some) sensitive instructions have been intentionally removed is said to be **paravirtualized** (Barham et al., 2003; and Whitaker et al., 2002). Emulating peculiar hardware instructions is an unpleasant and time-consuming task. It requires a call into the hypervisor and then emulating the exact semantics of a complicated instruction. It is far better just to have the guest operating system call the hypervisor (or microkernel) to do I/O, and so on. The main reason the first hypervisors just emulated the complete machine was the lack of availability of source code for the guest operating system (e.g., for Windows) or the vast number of variants (e.g., for Linux). Perhaps in the future, the hypervisor/microkernel API will be standardized, and subsequent operating systems will be designed to call it instead of using sensitive instructions. Doing so would make virtual machine technology easier to support and use.

The difference between true virtualization and paravirtualization is illustrated in Fig. 8-27. Here we have two virtual machines being supported on VT hardware. On the left, is an unmodified version of Windows as the guest operating system. When a sensitive instruction is executed, the hardware causes a trap to the hypervisor, which then emulates it and returns. On the right, is a version of Linux modified so that it no longer contains any sensitive instructions. Instead, when it needs to do I/O or change critical internal registers (such as the one pointing to the page tables), it makes a hypervisor call to get the work done, just like an application program making a system call in standard Linux.

In Fig. 8-27 we have shown the hypervisor as being divided into two parts separated by a dashed line. In reality, there is only one program running on the hardware. One part of it is responsible for interpreting trapped sensitive instructions, in this case, from Windows. The other part of it just carries out hypervisor calls. In the figure the latter part is labeled “microkernel.” If the hypervisor is intended to run only paravirtualized guest operating systems, there is no need for the emulation of sensitive instructions and we have a true microkernel, which just

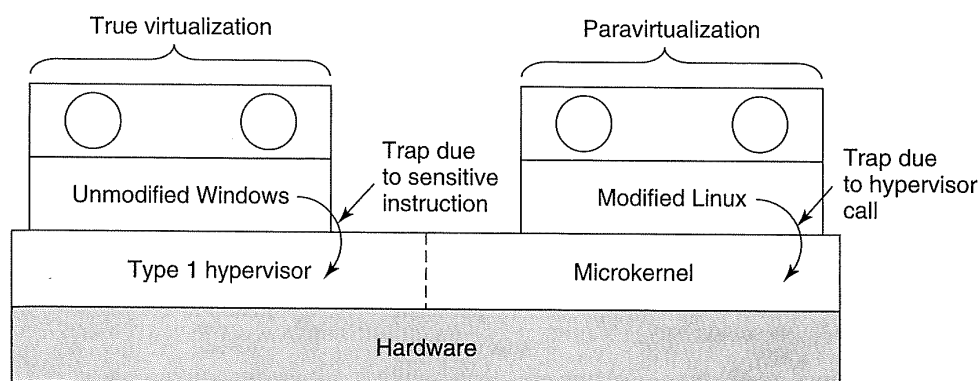


Figure 8-27. A hypervisor supporting both true virtualization and paravirtualization.

provides very basic services such as process dispatching and managing the MMU. The boundary between a type 1 hypervisor and a microkernel is vague already and will get even less clear as hypervisors begin acquiring more and more functionality and hypervisor calls, as seems likely. This subject is controversial, but it is increasingly clear that the program running in kernel mode on the bare hardware should be small and reliable and consist of thousands of lines of code, not millions of lines of code. The topic has been discussed by various researchers (Hand et al., 2005; Heiser et al. 2006; Hohmuth et al., 2004; and Roscoe et al., 2007).

Paravirtualizing the guest operating system raises a number of issues. First, if the sensitive instructions are replaced with calls to the hypervisor, how can the operating system run on the native hardware? After all, the hardware does not understand these hypervisor calls. And second, what if there are multiple hypervisors available in the marketplace, such as VMware, the open-source Xen originally from the University of Cambridge, and Microsoft's Viridian, all with somewhat different hypervisor APIs? How can the kernel be modified to run on all of them?

Amsden et al. (2006) have proposed a solution. In their model, the kernel is modified to call special procedures whenever it needs to do something sensitive. Together these procedures, called the **VMI (Virtual Machine Interface)** form a low-level layer that interfaces with the hardware or hypervisor. These procedures are designed to be generic and not tied to the hardware or to any particular hypervisor.

An example of this technique is given in Fig. 8-28 for a paravirtualized version of Linux they call VMI Linux (VMIL). When VMI Linux runs on the bare hardware, it has to be linked with a library that issues the actual (sensitive) instruction needed to do the work, as shown in Fig. 8-28(a). When running on a hypervisor, say VMware or Xen, the guest operating system is linked with different libraries that make the appropriate (and different) hypervisor calls to the

underlying hypervisor. In this way, the core of the operating system remains portable yet is hypervisor friendly and still efficient.

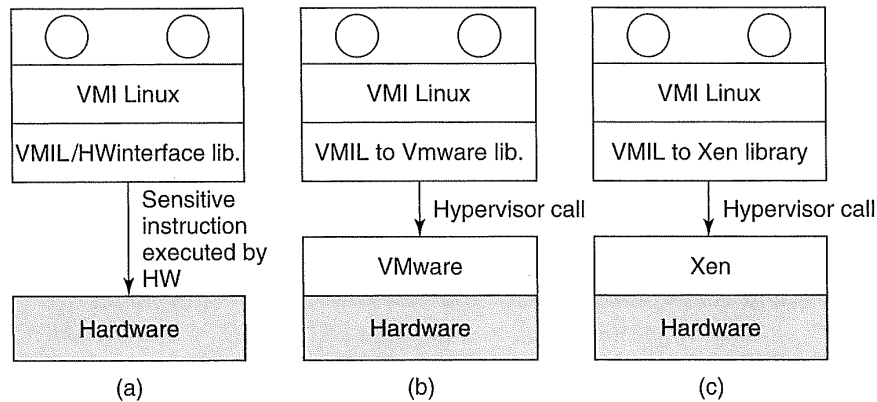


Figure 8-28. VMI Linux running on (a) the bare hardware (b) VMware (c) Xen.

Other proposals for a virtual machine interface have also been made. Another popular one is called **paravirt ops**. The idea is conceptually similar to what we described above, but different in the details.

8.3.5 Memory Virtualization

So far we have just addressed the issue of how to virtualize the CPU. But a computer system has more than just a CPU. It also has memory and I/O devices. They have to be virtualized, too. Let us see how that is done.

Modern operating systems nearly all support virtual memory, which is basically a mapping of pages in the virtual address space onto pages of physical memory. This mapping is defined by (multilevel) page tables. Typically the mapping is set in motion by having the operating system set a control register in the CPU that points to the top-level page table. Virtualization greatly complicates memory management.

Suppose, for example, a virtual machine is running, and the guest operating system in it decides to map its virtual pages 7, 4, and 3 onto physical pages 10, 11, and 12, respectively. It builds page tables containing this mapping and loads a hardware register to point to the top-level page table. This instruction is sensitive. On a VT CPU, it will trap; with VMware it will cause a call to a VMware procedure; on a paravirtualized operating system, it will generate a hypervisor call. For simplicity, let us assume it traps into a type 1 hypervisor, but the problem is the same in all three cases.

What does the hypervisor do now? One solution is to actually allocate physical pages 10, 11, and 12 to this virtual machine and set up the actual page tables to map the virtual machine's virtual pages 7, 4, and 3 to use them. So far, so good.

Now suppose a second virtual machine starts and maps its virtual pages 4, 5, and 6 onto physical pages 10, 11, and 12 and loads the control register to point to its page tables. The hypervisor catches the trap, but what should it do? It cannot use this mapping because physical pages 10, 11, and 12 are already in use. It can find some free pages, say 20, 21, and 22 and use them, but it first has to create new page tables mapping the virtual pages 4, 5, and 6 of virtual machine 2 onto 20, 21, and 22. If another virtual machine starts and tries to use physical pages 10, 11, and 12, it has to create a mapping for it. In general, for each virtual machine the hypervisor needs to create a **shadow page table** that maps the virtual pages used by the virtual machine onto the actual pages the hypervisor gave it.

Worse yet, every time the guest operating system changes its page tables, the hypervisor must change the shadow page tables as well. For example, if the guest OS remaps virtual page 7 onto what it sees as physical page 200 (instead of 10), the hypervisor has to know about this change. The trouble is that the guest operating system can change its page tables by just writing to memory. No sensitive operations are required, so the hypervisor does not even know about the change and certainly cannot update the shadow page tables used by the actual hardware.

A possible (but clumsy) solution, is for the hypervisor to keep track of which page in the guest's virtual memory contains the top-level page table. It can get this information the first time the guest attempts to load the hardware register that points to it because this instruction is sensitive and traps. The hypervisor can create a shadow page table at this point and also map the top-level page table and the page tables it points to as read only. Subsequent attempts by the guest operating system to modify any of them will cause a page fault and thus give control to the hypervisor, which can analyze the instruction stream, figure out what the guest OS is trying to do, and update the shadow page tables accordingly. It is not pretty, but it is doable in principle.

This is an area in which future versions of VT could provide assistance by doing a two-level mapping in hardware. The hardware could first map the virtual page to the guest's idea of the physical page, then map that address (which the hardware sees as a virtual address) onto the physical address, all without causing any traps. In this way no page tables would have to be marked as read only and the hypervisor would merely have to provide a mapping between each guest's virtual address space and physical memory. When switching virtual machines, it would just change this mapping, the same way a normal operating system changes the mapping when switching processes.

In a paravirtualized operating system, the situation is different. Here the paravirtualized OS in the guest knows that when it is finished changing some process' page table, it had better inform the hypervisor. Consequently, it first changes the page table completely, then issues a hypervisor call telling the hypervisor about the new page table. Thus instead of getting a protection fault on every update to the page table, there is one hypervisor call when the whole thing has been updated, obviously a more efficient way to do business.

8.3.6 I/O Virtualization

Having looked at CPU and memory virtualization, the next step is to examine I/O virtualization. The guest operating system typically will start out probing the hardware to find out what kinds of I/O devices are attached. These probes will trap to the hypervisor. What should the hypervisor do? One approach is for it to report back that the disks, printers, and so on are the ones that the hardware actually has. The guest will then load device drivers for these devices and try to use them. When the device drivers try to do actual I/O, they will read and write the device's hardware device registers. These instructions are sensitive and will trap to the hypervisor, which could then copy the needed values to and from the hardware registers, as needed.

But here, too, we have a problem. Each guest OS thinks it owns an entire disk partition, and there may be many more virtual machines (hundreds) than there are disk partitions. The usual solution is for the hypervisor to create a file or region on the actual disk for each virtual machine's physical disk. Since the guest OS is trying to control a disk that the real hardware has (and which the hypervisor understands), it can convert the block number being accessed into an offset into the file or disk region being used for storage and do the I/O.

It is also possible for the disk that the guest is using to be different from the real one. For example, if the actual disk is some brand-new high-performance disk (or RAID) with a new interface, the hypervisor could advertise to the guest OS that it has a plain old IDE disk and let the guest OS install an IDE disk driver. When this driver issues IDE disk commands, the hypervisor converts them into commands to drive the new disk. This strategy can be used to upgrade the hardware without changing the software. In fact, this ability of virtual machines to remap hardware devices was one of the reasons VM/370 became popular: companies wanted to buy new and faster hardware but did not want to change their software. Virtual machine technology made this possible.

Another I/O problem that must be solved somehow is the use of DMA, which uses absolute memory addresses. As might be expected, the hypervisor has to intervene here and remap the addresses before the DMA starts. However, hardware is starting to appear with an **I/O MMU**, which virtualizes the I/O the same way the MMU virtualizes the memory. This hardware eliminates the DMA problem.

A different approach to handling I/O is to dedicate one of the virtual machines to running a standard operating system and reflect all I/O calls from the other ones to it. This approach is enhanced when paravirtualization is used, so the command being issued to the hypervisor actually says what the guest OS wants (e.g., read block 1403 from disk 1) rather than being a series of commands writing to device registers, in which case the hypervisor has to play Sherlock Holmes and figure out what it is trying to do. Xen uses this approach to I/O, with the virtual machine that does I/O called **domain 0**.

I/O virtualization is an area in which type 2 hypervisors have a practical advantage over type 1 hypervisors: the host operating system contains the device drivers for all the weird and wonderful I/O devices attached to the computer. When an application program attempts to access a strange I/O device, the translated code can call the existing device driver to get the work done. With a type 1 hypervisor, the hypervisor must either contain the driver itself, or make a call to a driver in domain 0, which is somewhat similar to a host operating system. As virtual machine technology matures, future hardware is likely to allow application programs to access the hardware directly in a secure way, meaning that device drivers can be linked directly with application code or put in separate user-mode servers, thereby eliminating the problem.

8.3.7 Virtual Appliances

Virtual machines offer an interesting solution to a problem that has long plagued users, especially users of open-source software: how to install new application programs. The problem is that many applications are dependent on numerous other applications and libraries, which are themselves dependent on a host of other software packages, and so on. Furthermore, there may be dependencies on particular versions of the compilers, scripting languages, and the operating system.

With virtual machines now available, a software developer can carefully construct a virtual machine, load it with the required operating system, compilers, libraries, and application code, and freeze the entire unit, ready to run. This virtual machine image can then be put on a CD-ROM or a Website for customers to install or download. This approach means that only the software developer has to understand all the dependencies. The customers get a complete package that actually works, completely independent of which operating system they are running and which other software, packages, and libraries they have installed. These “shrink-wrapped” virtual machines are often called **virtual appliances**.

8.3.8 Virtual Machines on Multicore CPUs

The combination of virtual machines and multicore CPUs opens a whole new world in which the number of CPUs available can be set in software. If there are, say, four cores, and each one can be used to run, for example, up to eight virtual machines, a single (desktop) CPU can be configured as a 32-node multicomputer if need be, but it can also have fewer CPUs, depending on the needs of the software. Never before has it been possible for an application designer to first choose how many CPUs he wants and then write the software accordingly. This clearly represents a new phase in computing.

Although it is not so common yet, it is certainly conceivable that virtual machines could share memory. All that has to be done is map physical pages into the

address spaces of multiple virtual machines. If this can be done, a single computer becomes a virtual multiprocessor. Since all the cores in a multicore chip share the same RAM, a single quad-core chip could easily be configured as a 32-node multiprocessor or a 32-node multicomputer, as needed.

The combination of multicore, virtual machines, and hypervisors and micro-kernels is going to radically change the way people think about computer systems. Current software cannot deal with the idea of the programmer determining how many CPUs are needed, whether they should be set up as a multicomputer or a multiprocessor, and how minimal kernels of one kind or another fit into the picture. Future software will have to deal with these issues.

8.3.9 Licensing Issues

Most software is licensed on a per-CPU basis. In other words, when you buy a program, you have the right to run it on just one CPU. Does this contract give you the right to run the software on multiple virtual machines all running on the same physical machine? Many software vendors are somewhat unsure of what to do here.

The problem is much worse in companies that have a license allowing them to have n machines running the software at the same time, especially when virtual machines come and go on demand.

In some cases, software vendors have put an explicit clause in the license forbidding the licensee from running the software on a virtual machine or on an unauthorized virtual machine. Whether any of these restrictions will hold up in court and how users respond to them remains to be seen.

8.4 DISTRIBUTED SYSTEMS

Having now completed our study of multiprocessors, multicomputers, and virtual machines, it is time to turn to the last type of multiple processor system, the **distributed system**. These systems are similar to multicomputers in that each node has its own private memory, with no shared physical memory in the system. However, distributed systems are even more loosely coupled than multicomputers.

To start with, the nodes of a multicomputer generally have a CPU, RAM, a network interface, and perhaps a hard disk for paging. In contrast, each node in a distributed system is a complete computer, with a full complement of peripherals. Next, the nodes of a multicomputer are normally in a single room, so they can communicate by a dedicated high-speed network, whereas the nodes of a distributed system may be spread around the world. Finally, all the nodes of a multicomputer run the same operating system, share a single file system, and are under a common administration, whereas the nodes of a distributed system may each run a

different operating system, each of which has its own file system, and be under a different administration. A typical example of a multicomputer is 512 nodes in a single room at a company or university working on, say, pharmaceutical modeling, whereas a typical distributed system consists of thousands of machines loosely cooperating over the Internet. Figure 8-29 compares multiprocessors, multicomputers, and distributed systems on the points mentioned above.

Item	Multiprocessor	Multicomputer	Distributed System
Node configuration	CPU	CPU, RAM, net interface	Complete computer
Node peripherals	All shared	Shared exc. maybe disk	Full set per node
Location	Same rack	Same room	Possibly worldwide
Internode communication	Shared RAM	Dedicated interconnect	Traditional network
Operating systems	One, shared	Multiple, same	Possibly all different
File systems	One, shared	One, shared	Each node has own
Administration	One organization	One organization	Many organizations

Figure 8-29. Comparison of three kinds of multiple CPU systems.

Multicomputers are clearly in the middle using these metrics. An interesting question is: "Are multicomputers more like multiprocessors or more like distributed systems?" Oddly enough, the answer depends strongly on your perspective. From a technical perspective, multiprocessors have shared memory and the other two do not. This difference leads to different programming models and different mindsets. However, from an applications perspective, multiprocessors and multicomputers are just big equipment racks in a machine room. Both are used for solving computationally intensive problems, whereas a distributed system connecting computers all over the Internet is typically much more involved in communication than in computation and is used in a different way.

To some extent, loose coupling of the computers in a distributed system is both a strength and a weakness. It is a strength because the computers can be used for a wide variety of applications, but it is also a weakness, because programming these applications is difficult due to the lack of any common underlying model.

Typical Internet applications include access to remote computers (using *telnet*, *ssh*, and *rlogin*), access to remote information (using the World Wide Web and FTP, the File Transfer Protocol), person-to-person communication (using e-mail and chat programs), and many emerging applications (e.g., e-commerce, telemedicine, and distance learning). The trouble with all these applications is that each one has to reinvent the wheel. For example, e-mail, FTP, and the World Wide Web all basically move files from point *A* to point *B*, but each one has its own way of doing it, complete with its own naming conventions, transfer protocols, replication techniques, and everything else. Although many Web browsers

hide these differences from the average user, the underlying mechanisms are completely different. Hiding them at the user interface level is like having a person at a full-service travel agent Web site order a trip from New York to San Francisco, and only later discover whether she has purchased a plane, train, or bus ticket.

What distributed systems add to the underlying network is some common paradigm (model) that provides a uniform way of looking at the whole system. The intent of the distributed system is to turn a loosely connected bunch of machines into a coherent system based on one concept. Sometimes the paradigm is simple and sometimes it is more elaborate, but the idea is always to provide something that unifies the system.

A simple example of a unifying paradigm in a slightly different context is found in UNIX, where all I/O devices are made to look like files. Having keyboards, printers, and serial lines all operated on the same way, with the same primitives, makes it easier to deal with them than having them all conceptually different.

One way a distributed system can achieve some measure of uniformity in the face of different underlying hardware and operating systems is to have a layer of software on top of the operating system. The layer, called **middleware**, is illustrated in Fig. 8-30. This layer provides certain data structures and operations that allow processes and users on far-flung machines to interoperate in a consistent way.

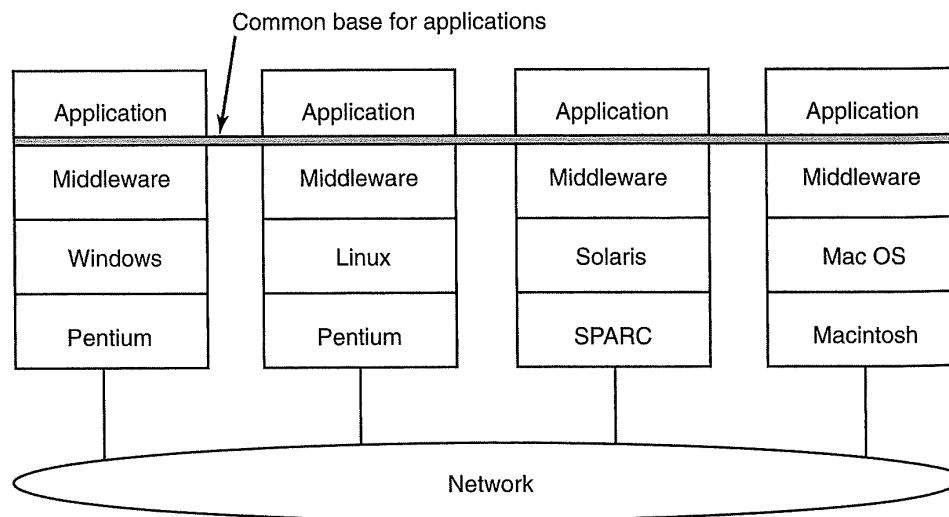


Figure 8-30. Positioning of middleware in a distributed system.

In a sense, middleware is like the operating system of a distributed system. That is why it is being discussed in a book on operating systems. On the other hand, it is *not* really an operating system, so the discussion will not go into much detail. For a comprehensive, book-length treatment of distributed systems, see

Distributed Systems (Tanenbaum and van Steen, 2006). In the remainder of this chapter, we will look quickly at the hardware used in a distributed system (i.e., the underlying computer network), then its communication software (the network protocols). After that we will consider a variety of paradigms used in these systems.

8.4.1 Network Hardware

Distributed systems are built on top of computer networks, so a brief introduction to the subject is in order. Networks come in two major varieties, **LANs** (**Local Area Networks**), which cover a building or a campus, and **WANs** (**Wide Area Networks**), which can be citywide, countrywide, or even worldwide. The most important kind of LAN is Ethernet, so we will examine that as an example LAN. As our example WAN, we will look at the Internet, even though technically, the Internet is not one network, but a federation of thousands of separate networks. However, for our purposes, it is sufficient to think of it as one WAN.

Ethernet

Classic Ethernet, which is described in IEEE Standard 802.3, consists of a coaxial cable to which a number of computers are attached. The cable is called the **Ethernet**, in reference to the *luminiferous ether* through which electromagnetic radiation was once thought to propagate. (When the nineteenth-century British physicist James Clerk Maxwell discovered that electromagnetic radiation could be described by a wave equation, scientists assumed that space must be filled with some ethereal medium in which the radiation was propagating. Only after the famous Michelson-Morley experiment in 1887, which failed to detect the ether, did physicists realize that radiation could propagate in a vacuum.)

In the very first version of Ethernet, a computer was attached to the cable by literally drilling a hole halfway through the cable and screwing in a wire leading to the computer. This was called a **vampire tap**, and is shown symbolically in Fig. 8-31(a). The taps were hard to get right, so before long, proper connectors were used. Nevertheless, electrically, all the computers were connected as if the cables on their network interface cards were soldered together.

To send a packet on an Ethernet, a computer first listens to the cable to see if any other computer is currently transmitting. If not, it just begins transmitting a packet, which consists of a short header followed by a 0- to 1500-byte payload. If the cable is in use, the computer simply waits until the current transmission finishes, then it begins sending.

If two computers start transmitting simultaneously, a collision results, which both of them detect. Both respond by terminating their transmissions, waiting a random amount of time between 0 and T μ sec and then starting again. If another collision occurs, all colliding computers randomize the wait into the interval 0 to $2T$ μ sec, and then try again. On each further collision, the maximum wait interval



MODERN^{3e} OPERATING SYSTEMS

ANDREW S. TANENBAUM

The widely anticipated revision of this worldwide best-seller incorporates the latest developments in operating systems technologies. Hundreds of pages of new content have been added, and older material has been removed. This authoritative book covers both the principles and practice of modern operating systems, with an emphasis on Linux, Windows Vista®, and embedded, real-time, and multimedia systems.

Andrew S. Tanenbaum brings his clear and entertaining style to the “big picture” concepts every OS designer needs to master. Drawing on his long experience as designer or co-designer of three operating systems, he shares deep knowledge and practical detail that few other books can match. Among the many topics Tanenbaum covers in depth: processes, threads, memory management, file systems, I/O, deadlocks, interface design, multimedia, performance tradeoffs, and the newest trends in operating system design.

FEATURES

- NEW** — Up-to-the-minute coverage of Windows Vista and the newest Linux/Unix operating systems
- NEW** — Chapter-length case study on the Symbian operating system for mobile devices
- NEW** — Extensively updated and enhanced security coverage
- NEW** — Reorganized to present key abstractions earlier
- NEW** — Coverage of new research that is likely to shape tomorrow's operating systems
- NEW** — Updated and additional programming exercises added to every chapter
- NEW** — Online OS experiments using popular Windows and Open Source tools
- NEW** — OS Simulation exercises

About the Author

Andrew S. Tanenbaum is Professor of Computer Science at Vrije Universiteit, Amsterdam, The Netherlands, where he focuses on advanced research on systems and security. Dr. Tanenbaum is a Fellow of both the IEEE and ACM, and a Member of the Royal Dutch Academy of Arts and Sciences. He has published nearly 150 scientific papers and lectured in over a dozen countries on operating systems and related topics. He was awarded the 2007 IEEE James H. Mulligan, Jr. Education Medal; the award was given “For contributions to education in computing, especially computer organization, networking and operating systems.”

Contributions made by:

Ada Gavrilovska-Habl, Georgia Institute of Technology
Michael Jipping, Hope College
David Probert, Microsoft Corporation

OTHER BEST-SELLING TITLES BY THE AUTHOR AVAILABLE FROM PRENTICE HALL

Distributed Systems: Principles and Paradigms, Second Edition,
by Andrew S. Tanenbaum and Maarten van Steen, 2006
Operating Systems Design and Implementation, Third Edition,
by Andrew S. Tanenbaum and Albert S. Woodhull, 2006
Structured Computer Organization, Fifth Edition,
by Andrew S. Tanenbaum, 2005
Computer Networks, Fourth Edition, by Andrew S. Tanenbaum, 2002



Upper Saddle River, NJ 07458
www.nrenhall.com

