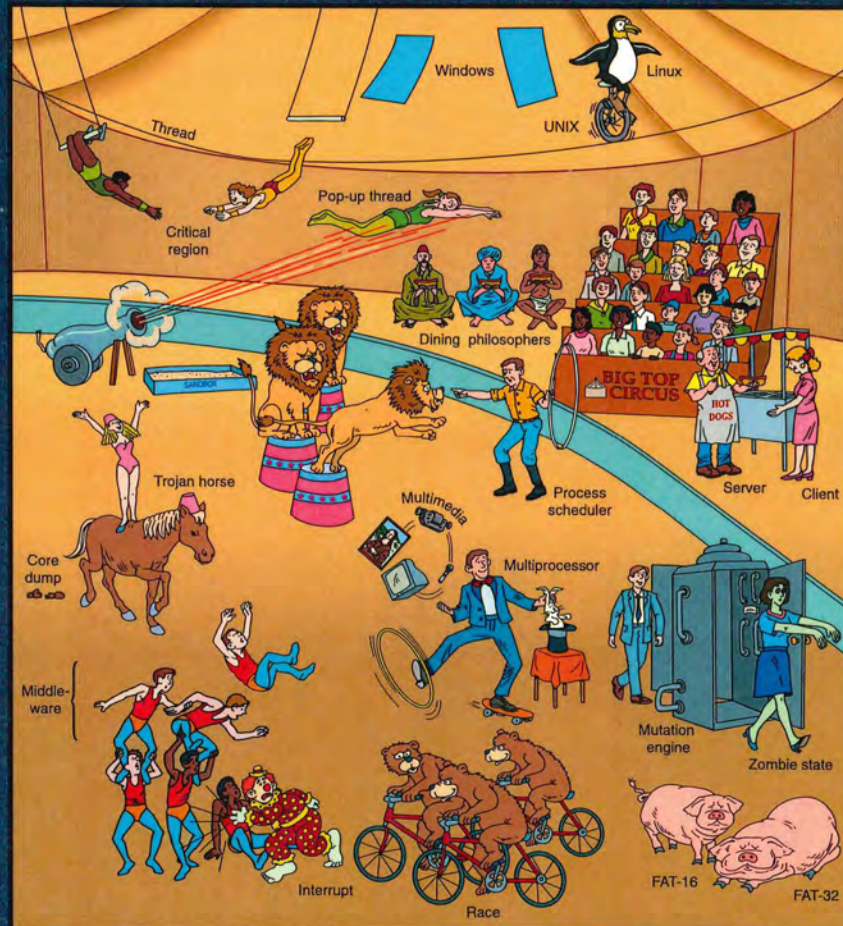


MODERN OPERATING SYSTEMS

SECOND EDITION



Andrew S. Tanenbaum

MODERN OPERATING SYSTEMS

SECOND EDITION

ANDREW S. TANENBAUM

*Vrije Universiteit
Amsterdam, The Netherlands*



PRENTICE HALL
UPPER SADDLE RIVER, NEW JERSEY 07458

Library of Congress Cataloging-in-Publication Data

Tanenbaum, Andrew S.

Modern operating systems / Andrew S. Tanenbaum.--2nd ed.
p. cm.

Includes bibliographical references and index.

ISBN 0-13-031358-0

1. Operating systems (Computers) I. Title

QA76.76.O63 T359 2001

005.4'3--dc21

Vice President and Editorial Director, ECS: *Marcia Horton*

Publisher: *Alan Apt*

Associate Editor: *Toni D. Holm*

Editorial Assistant: *Amy K. Todd*

Vice President and Director of Production and Manufacturing, ESM: *David W. Riccardi*

Executive Managing Editor: *Vince O'Brien*

Managing Editor: *David A. George*

Senior Production Editor: *Camille Trentacoste*

Director of Creative Services: *Paul Belfanti*

Creative Director: *Carole Anson*

Art Director: *Heather Scott*

Cover Art: *Don Martinetti*

Cover Design: *Joseph Sengotta*

Assistant to Art Director: *John Christiana*

Interior Illustration: *Patricia Gutiérrez*

Interior Design and Typesetting; Cover Illustration Concept: *Andrew S. Tanenbaum*

Manufacturing Manager: *Trudy Piscioti*

Manufacturing Buyer: *Pat Brown*

Marketing Manager: *Jennie Burger*



© 2001 by Prentice-Hall, Inc.
Upper Saddle River, New Jersey 07458

The authors 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 authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in this book. The authors 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.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks and registered trademarks. Where those designations appear in this book, and Prentice Hall and the authors were aware of a trademark claim, the designations have been printed in initial caps or all caps. All product names mentioned remain trademarks or registered trademarks of their respective owners.

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.

Printed in the United States of America

1098765432

ISBN 0-13-031358-0

Prentice-Hall International (UK) Limited, *London*

Prentice-Hall of Australia Pty. Limited, *Sydney*

Prentice-Hall Canada Inc., *Toronto*

Prentice-Hall Hispanoamericana, S.A., *Mexico*

Prentice-Hall of India Private Limited, *New Delhi*

Prentice-Hall of Japan, Inc., *Tokyo*

Pearson Education Asia Pte. Ltd., *Singapore*

Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

1

INTRODUCTION

A modern computer system consists of one or more processors, some main memory, disks, printers, a keyboard, a display, network interfaces, and other input/output devices. All in all, a complex system. Writing programs that keep track of all these components and use them correctly, let alone optimally, is an extremely difficult job. For this reason, computers are equipped with a layer of software called the **operating system**, whose job is to manage all these devices and provide user programs with a simpler interface to the hardware. These systems are the subject of this book.

The placement of the operating system is shown in Fig. 1-1. At the bottom is the hardware, which, in many cases, is itself composed of two or more levels (or layers). The lowest level contains physical devices, consisting of integrated circuit chips, wires, power supplies, cathode ray tubes, and similar physical devices. How these are constructed and how they work are the provinces of the electrical engineer.

Next comes the **microarchitecture level**, in which the physical devices are grouped together to form functional units. Typically this level contains some registers internal to the CPU (Central Processing Unit) and a data path containing an arithmetic logic unit. In each clock cycle, one or two operands are fetched from the registers and combined in the arithmetic logic unit (for example, by addition or Boolean AND). The result is stored in one or more registers. On some machines, the operation of the data path is controlled by software, called the **microprogram**. On other machines, it is controlled directly by hardware circuits.

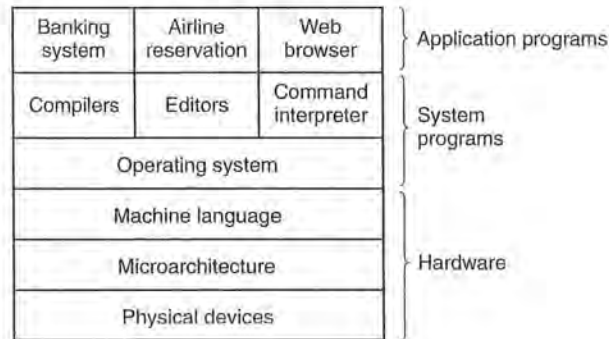


Figure 1-1. A computer system consists of hardware, system programs, and application programs.

The purpose of the data path is to execute some set of instructions. Some of these can be carried out in one data path cycle; others may require multiple data path cycles. These instructions may use registers or other hardware facilities. Together, the hardware and instructions visible to an assembly language programmer form the **ISA (Instruction Set Architecture)** level. This level is often called **machine language**.

The machine language typically has between 50 and 300 instructions, mostly for moving data around the machine, doing arithmetic, and comparing values. In this level, the input/output devices are controlled by loading values into special **device registers**. For example, a disk can be commanded to read by loading the values of the disk address, main memory address, byte count, and direction (read or write) into its registers. In practice, many more parameters are needed, and the status returned by the drive after an operation is highly complex. Furthermore, for many I/O (Input/Output) devices, timing plays an important role in the programming.

To hide this complexity, an operating system is provided. It consists of a layer of software that (partially) hides the hardware and gives the programmer a more convenient set of instructions to work with. For example, read block from file is conceptually simpler than having to worry about the details of moving disk heads, waiting for them to settle down, and so on.

On top of the operating system is the rest of the system software. Here we find the command interpreter (shell), window systems, compilers, editors, and similar application-independent programs. It is important to realize that these programs are definitely not part of the operating system, even though they are typically supplied by the computer manufacturer. This is a crucial, but subtle, point. The operating system is (usually) that portion of the software that runs in **kernel mode** or **supervisor mode**. It is protected from user tampering by the hardware (ignoring for the moment some older or low-end microprocessors that do not have

hardware protection at all). Compilers and editors run in **user mode**. If a user does not like a particular compiler, he† is free to 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 normally 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). Still, for traditional computers, the operating system is what runs in kernel mode.

That said, 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 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.

Finally, above the system programs come the application programs. These programs are purchased or written by the users to solve their particular problems, such as word processing, spreadsheets, engineering calculations, or storing information in a database.

1.1 WHAT IS AN OPERATING SYSTEM?

Most computer users have had some experience with an operating system, but it is difficult to pin down precisely what an operating system is. Part of the problem is that operating systems perform two basically unrelated functions, extending the machine and managing resources, and depending on who is doing the talking, you hear mostly about one function or the other. Let us now look at both.

1.1.1 The Operating System as an Extended Machine

As mentioned earlier, 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, let us briefly look at how floppy disk I/O is done using the NEC

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

PD765 compatible controller chips used on most Intel-based personal computers. (Throughout this book we will use the terms “floppy disk” and “diskette” interchangeably.) 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 just as complex and quite different). 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 user.

The program that hides the truth about the hardware from the programmer and presents a nice, simple view of named files that can be read and written is, of course, the operating system. Just as the operating system shields the programmer from the disk hardware and presents a simple file-oriented interface, it also conceals a lot of unpleasant business concerning interrupts, timers, memory management, and other low-level features. In each case, the abstraction offered by the operating system is simpler and easier to use than that offered by the underlying hardware.

In this view, the function of the operating system is to present the user with the equivalent of an **extended machine** or **virtual machine** that is easier to program than the underlying hardware. How the operating system achieves this goal is a long story, which we will study in detail throughout this book. To summarize it in a nutshell, the operating system provides a variety of services that programs can obtain using special instructions called system calls. We will examine some of the more common system calls later in this chapter.

1.1.2 The Operating System as a Resource Manager

The concept of the operating system as primarily providing its users with a convenient interface 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.

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 to 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 who is 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 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 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 and Plugboards

After Babbage’s unsuccessful efforts, little progress was made in constructing digital computers until World War II. Around the mid-1940s, Howard Aiken at Harvard, John von Neumann at the Institute for Advanced Study in Princeton, J. Presper Eckert and William Mauchley at the University of Pennsylvania, and Konrad Zuse in Germany, among others, all succeeded in building calculating engines. The first ones used mechanical relays but were very slow, with cycle times measured in seconds. Relays were later replaced by vacuum tubes. These machines were enormous, filling up entire rooms with tens of thousands of vacuum tubes, but they were still millions of times slower than even the cheapest personal computers available today.

In these early days, a single group of people designed, built, programmed, operated, and maintained each machine. All programming was done in absolute

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. 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.

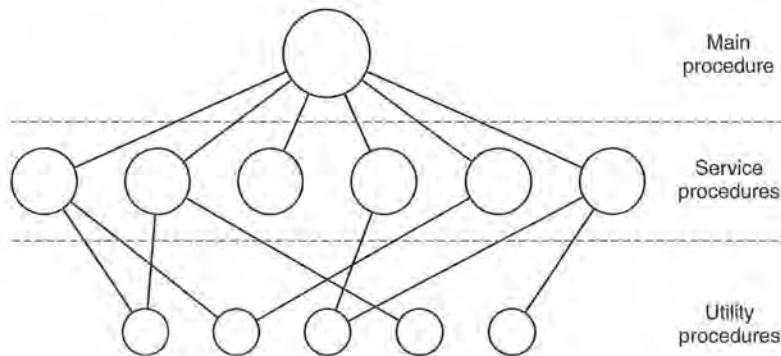


Figure 1-24. A simple structuring model for a monolithic system.

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 6 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

Windows has directories and they are created with `CreateDirectory` and `RemoveDirectory`, respectively. There is also a notion of a current directory, set by `SetCurrentDirectory`. The current time is acquired using `GetLocalTime`.

The Win32 interface does not have links to files, mounted file systems, security, or signals, so the calls corresponding to the UNIX ones do not exist. Of course, Win32 has a huge number of other calls that UNIX does not have, especially for managing the GUI. And Windows 2000 has an elaborate security system and also supports file links.

One last note about Win32 is perhaps worth making. Win32 is not a terribly uniform or consistent interface. The main culprit here was the need to be backward compatible with the previous 16-bit interface used in Windows 3.x.

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 five 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 five designs are monolithic systems, layered systems, virtual machines, exokernels, and client-server systems.

1.7.1 Monolithic Systems

By far the most common organization, this approach might well be subtitled "The Big Mess." The structure is that there is no structure. The operating system is written as a collection of procedures, each of which can call any of the other ones whenever it needs to. When this technique is used, each procedure in the system has a well-defined interface in terms of parameters and results, and each one is free to call any other one, if the latter provides some useful computation that the former needs.

To construct the actual object program of the operating system when this approach is used, one first compiles all the individual procedures, or files containing the procedures, and then binds them all together into a single object 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 at least a little 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

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	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.

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. Above this layer each process effectively had its own 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 object 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 Virtual Machines

The initial releases of OS/360 were strictly batch systems. Nevertheless, many 360 users wanted to have timesharing, so various groups, both inside and outside IBM decided to write timesharing systems for it. The 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, and which is now widely used on its remaining mainframes.

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. 1-26. 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.

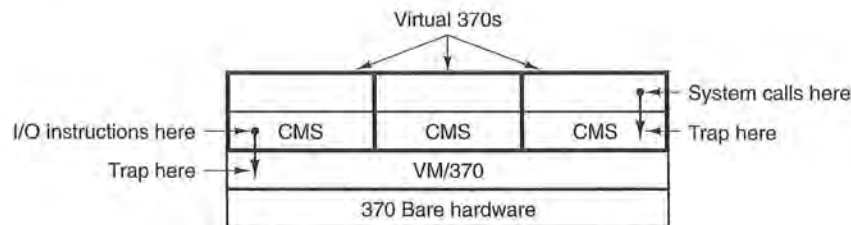


Figure 1-26. The structure of VM/370 with CMS.

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. Some run one of the descendants of OS/360 for batch or transaction processing, while other ones run a single-user, interactive system called **CMS (Conversational Monitor System)** for interactive timesharing users.

When a CMS program executes a system call, the call is 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 issues the normal hardware I/O instructions for reading its virtual disk or whatever is needed to carry out the call. These I/O instructions are trapped by VM/370, which then performs 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 can be much simpler, more flexible, and easier to maintain.

The idea of a virtual machine is heavily used nowadays in a different context: running old MS-DOS programs on a Pentium (or other 32-bit Intel CPU). When designing the Pentium and its software, both Intel and Microsoft realized that there would be a big demand for running old software on new hardware. For this reason, Intel provided a virtual 8086 mode on the Pentium. In this mode, the machine acts like an 8086 (which is identical to an 8088 from a software point of view), including 16-bit addressing with a 1-MB limit.

This mode is used by Windows and other operating systems for running MS-DOS programs. These programs are started up in virtual 8086 mode. As long as they execute normal instructions, they run on the bare hardware. However, when a program tries to trap to the operating system to make a system call, or tries to do protected I/O directly, a trap to the virtual machine monitor occurs.

Two variants on this design are possible. In the first one, MS-DOS itself is loaded into the virtual 8086's address space, so the virtual machine monitor just reflects the trap back to MS-DOS, just as would happen on a real 8086. When MS-DOS later tries to do the I/O itself, that operation is caught and carried out by the virtual machine monitor.

In the other variant, the virtual machine monitor just catches the first trap and does the I/O itself, since it knows what all the MS-DOS system calls are and thus knows what each trap is supposed to do. This variant is less pure than the first one, since it only emulates MS-DOS correctly, and not other operating systems, as the first one does. On the other hand, it is much faster, since it saves the trouble of starting up MS-DOS to do the I/O. A further disadvantage of actually running MS-DOS in virtual 8086 mode is that MS-DOS fiddles around with the interrupt enable/disable bit quite a lot, all of which must be emulated at considerable cost.

It is worth noting that neither of these approaches are really the same as VM/370, since the machine being emulated is not a full Pentium, but only an 8086. With the VM/370 system, it is possible to run VM/370, itself, in the virtual machine. With the Pentium, it is not possible to run, say, Windows in the virtual 8086 because no version of Windows runs on an 8086; a 286 is the minimum for even the oldest version, and 286 emulation is not provided (let alone Pentium emulation). However, by modifying the Windows binary slightly, this emulation is possible and even available in commercial products.

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.

1.7.4 Exokernels

With VM/370, each user process gets an exact copy of the actual computer. With virtual 8086 mode on the Pentium, each user process gets an exact copy of a different computer. Going one step further, researchers at M.I.T. have built a system that gives each user a clone of the actual computer, but with a subset of the resources (Engler et al., 1995). 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**. 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.

1.7.5 Client-Server Model

VM/370 gains much in simplicity by moving a large part of the traditional operating system code (implementing the extended machine) into a higher layer, CMS. Nevertheless, VM/370 itself is still a complex program because simulating a

number of virtual 370s in their entirety is not *that* simple (especially if you want to do it reasonably efficiently).

A trend in modern operating systems is to take the idea of moving code up into higher layers even further and remove as much as possible from kernel mode, leaving a minimal **microkernel**. The usual approach is to implement most of the operating system in user processes. To request a service, such as reading a block of a file, a user process (now known as the **client process**) sends the request to a **server process**, which then does the work and sends back the answer.

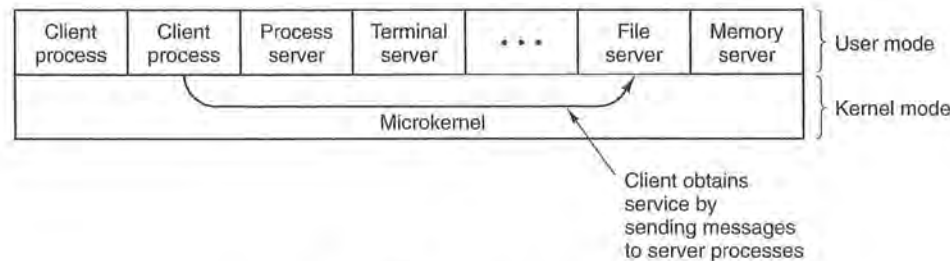


Figure 1-27. The client-server model.

In this model, shown in Fig. 1-27, all the kernel does is handle the communication between clients and servers. By splitting the operating system up into parts, each of which only handles one facet of the system, such as file service, process service, terminal service, or memory service, each part becomes small and manageable. Furthermore, because all the servers run as user-mode processes, and not in kernel mode, they do not have direct access to the hardware. As a consequence, if a bug in the file server is triggered, the file service may crash, but this will not usually bring the whole machine down.

Another advantage of the client-server model is its adaptability to use in distributed systems (see Fig. 1-28). If a client communicates with a server by sending it messages, the client need not know whether the message is handled locally in its own machine, or whether it was sent across a network to a server on a remote machine. As far as the client is concerned, the same thing happens in both cases: a request was sent and a reply came back.

The picture painted above of a kernel that handles only the transport of messages from clients to servers and back is not completely realistic. Some operating system functions (such as loading commands into the physical I/O device registers) are difficult, if not impossible, to do from user-space programs. There are two ways of dealing with this problem. One way is to have some critical server processes (e.g., I/O device drivers) actually run in kernel mode, with complete access to all the hardware, but still communicate with other processes using the normal message mechanism.

The other way is to build a minimal amount of **mechanism** into the kernel but leave the **policy** decisions up to servers in user space (Levin et al., 1975). For

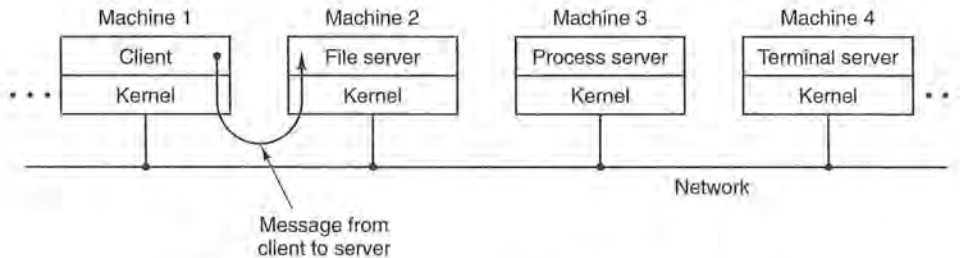


Figure 1-28. The client-server model in a distributed system.

example, the kernel might recognize that a message sent to a certain special address means to take the contents of that message and load it into the I/O device registers for some disk, to start a disk read. In this example, the kernel would not even inspect the bytes in the message to see if they were valid or meaningful; it would just blindly copy them into the disk's device registers. (Obviously, some scheme for limiting such messages to authorized processes only must be used.) The split between mechanism and policy is an important concept; it occurs again and again in operating systems in various contexts.

1.8 RESEARCH ON OPERATING SYSTEMS

Computer science is a rapidly advancing field and it is hard to predict where it is going. Researchers at universities and industrial research labs are constantly thinking up new ideas, some of which go nowhere but some of which become the cornerstone of future products and have massive impact on the industry and users. Telling which is which turns out to be easier to do in hindsight than in real time. Separating the wheat from the chaff is especially difficult because it often takes 20-30 years from idea to impact.

For example, when President Eisenhower set up the Dept. of Defense's Advanced Research Projects Agency (ARPA) in 1958, he was trying to keep the Army from killing the Navy and the Air Force over the Pentagon's research budget. He was not trying to invent the Internet. But one of the things ARPA did was fund some university research on the then-obscure concept of packet switching, which quickly led to the first experimental packet-switched network, the ARPANET. It went live in 1969. Before long, other ARPA-funded research networks were connected to the ARPANET, and the Internet was born. The Internet was then happily used by academic researchers for sending email to each other for 20 years. In the early 1990s, Tim Berners-Lee invented the World Wide Web at the CERN research lab in Geneva and Marc Andreessen wrote a graphical browser for it at the University of Illinois. All of a sudden the Internet was full of chatting teenagers. President Eisenhower is probably rolling over in his grave.

9.2 BASICS OF CRYPTOGRAPHY

A little knowledge of cryptography may be useful for understanding parts of this chapter and some subsequent ones. However, a serious discussion of cryptography is beyond the scope of this book. Many excellent books on computer security discuss the topic at length. The interested reader is referred to some of these (e.g., Kaufman et al., 1995; and Pfleeger, 1997). Below we give a very quick discussion of cryptography for readers not familiar with it at all.

The purpose of cryptography is to take a message or file, called the **plaintext**, and encrypt it into the **ciphertext** in such a way that only authorized people know how to convert it back to the plaintext. For all others, the ciphertext is just an incomprehensible pile of bits. Strange as it may sound to beginners in the area, the encryption and decryption algorithms (functions) should always be public. Trying to keep them secret never works and gives the people trying to keep the secrets a false sense of security. In the trade, this tactic is called **security by obscurity** and is employed only by security amateurs. Oddly enough, this category also includes many huge multinational corporations that really should know better.

Instead, the secrecy depends on parameters to the algorithms called **keys**. If P is the plaintext file, K_E is the encryption key, C is the ciphertext, and E is the encryption algorithm (i.e., function), then $C = E(P, K_E)$. This is the definition of encryption. It says that the ciphertext is obtained by using the (known) encryption algorithm, E , with the plaintext, P , and the (secret) encryption key, K_E , as parameters.

Similarly, $P = D(C, K_D)$ where D is the decryption algorithm and K_D is the decryption key. This says that to get the plaintext, P , back from the ciphertext, C , and the decryption key, K_D , one runs the algorithm D with C and K_D as parameters. The relation between the various pieces is shown in Fig. 9-2.

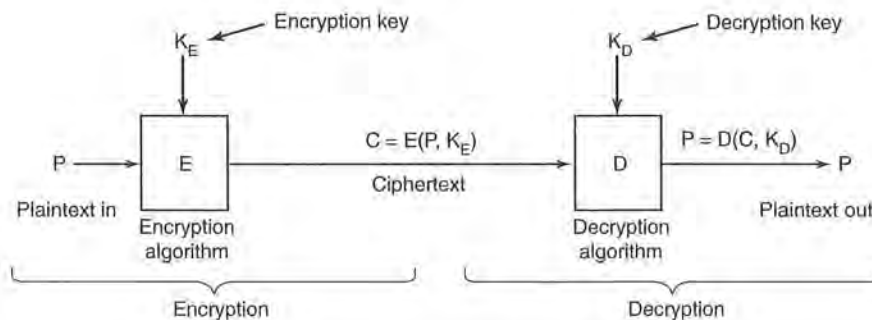


Figure 9-2. Relationship between the plaintext and the ciphertext.

9.2.1 Secret-Key Cryptography

To make this clearer, consider an encryption algorithm in which each letter is replaced by a different letter, for example, all *As* are replaced by *Qs*, all *Bs* are replaced by *Ws*, all *Cs* are replaced by *Es*, and so on like this:

plaintext:	ABCDEFGHIJKL MNOPQRST UVWXYZ
ciphertext:	QWERTYUIOPASDFGHJKLZXCVBNM

This general system is called a **monoalphabetic substitution**, with the key being the 26-letter string corresponding to the full alphabet. The encryption key in this example is *QWERTYUIOPASDFGHJKLZXCVBNM*. For the key above, the plaintext *ATTACK* would be transformed into the ciphertext *QZZQEA*. The decryption key tells how to get back from the ciphertext to the plaintext. In this example, the decryption key is *KXVMCNOHQRSZYIJADLEGWBUFT* because an *A* in the ciphertext is a *K* in the plaintext, a *B* in the ciphertext is an *X* in the plaintext, etc.

At first glance this might appear to be a safe system because although the cryptanalyst knows the general system (letter for letter substitution), he does not know which of the $26! \approx 4 \times 10^{26}$ possible keys is in use. Nevertheless, given a surprisingly small amount of ciphertext, the cipher can be broken easily. The basic attack takes advantage of the statistical properties of natural languages. In English, for example, *e* is the most common letter, followed by *t*, *o*, *a*, *n*, *i*, etc. The most common two letter combinations, called **digrams**, are *th*, *in*, *er*, *re*, etc. Using this kind of information, breaking the cipher is easy.

Many cryptographic systems, like this one, have the property that given the encryption key it is easy to find the decryption key, and vice versa. Such systems are called **secret-key cryptography** or **symmetric-key cryptography**. Although monoalphabetic substitution ciphers are worthless, other symmetric key algorithms are known and are relatively secure if the keys are long enough. For serious security, probably 1024-bit keys should be used, giving a search space of $2^{1024} \approx 2 \times 10^{308}$ keys. Shorter keys may thwart amateurs, but not major governments.

9.2.2 Public-Key Cryptography

Secret key systems are efficient because the amount of computation required to encrypt or decrypt a message is manageable, but have a big drawback: the sender and receiver must both be in possession of the shared secret key. They may even have to get together physically for one to give it to the other. To get around this problem, **public-key cryptography** is used (Diffie and Hellman, 1976). This system has the property that distinct keys are used for encryption and decryption and that given a well-chosen encryption key, it is virtually impossible

to discover the corresponding decryption key. Under these circumstances, the encryption key can be made public and only the private decryption key kept secret.

Just to give a feel for public-key cryptography, consider the following two questions:

Question 1: How much is $314159265358979 \times 314159265358979$?

Question 2: What is the square root of $3912571506419387090594828508241$?

Most sixth graders given a pencil, paper, and the promise of a really big ice cream sundae for the correct answer could answer question 1 in an hour or two. Most adults given a pencil, paper, and the promise of a lifetime 50% tax cut could not solve question 2 at all without using a calculator, computer, or other external help. Although squaring and square rooting are inverse operations, they differ enormously in their computational complexity. This kind of asymmetry forms the basis of public-key cryptography. Encryption makes use of the easy operation but decryption without the key requires you to perform the hard operation.

A public key system called **RSA** exploits the fact that multiplying big numbers is much easier for a computer to do than factoring big numbers, especially when all arithmetic is done using modulo arithmetic and all the numbers involved have hundreds of digits (Rivest et al., 1978). This system is widely used in the cryptographic world. Systems based on discrete logarithms are also used (El Gamal, 1985). The main problem with public-key cryptography is that it is a thousand times slower than symmetric cryptography.

The way public-key cryptography works is that everyone picks a (public key, private key) pair and publishes the public key. The public key is the encryption key; the private key is the decryption key. Usually, the key generation is automated, possibly with a user-selected password fed into the algorithm as a seed. To send a secret message to a user a correspondent encrypts the message with the receiver's public key. Since only the receiver has the private key, only the receiver can decrypt the message.

9.2.3 One-Way Functions

There are various situations that we will see later in which it is desirable to have some function, f , which has the property that given f and its parameter x , computing $y = f(x)$ is easy to do, but given only $f(x)$, finding x is computationally infeasible. Such a function typically mangles the bits in complex ways. It might start out by initializing y to x . Then it could have a loop that iterates as many times as there are 1 bits in x , with each iteration permuting the bits of y in an iteration-dependent way, adding in a different constant on each iteration, and generally mixing the bits up very thoroughly.

9.2.4 Digital Signatures

Frequently it is necessary to sign a document digitally. For example, suppose a bank customer instructs the bank to buy some stock for him by sending the bank an email message. An hour after the order has been sent and executed, the stock crashes. The customer now denies ever having sent the email. The bank can produce the email, of course, but the customer can claim the bank forged it in order to get a commission. How does a judge know who is telling the truth?

Digital signatures make it possible to sign email messages and other digital documents in such a way that they cannot be repudiated by the sender later. One common way is to first run the document through a one-way hashing algorithm that is very hard to invert. The hashing function typically produces a fixed-length result independent of the original document size. The most popular hashing functions used are **MD5 (Message Digest)**, which produces a 16-byte result (Rivest, 1992) and **SHA (Secure Hash Algorithm)**, which produces a 20-byte result (NIST, 1995).

The next step assumes the use of public-key cryptography as described above. The document owner then applies his private key to the hash to get $D(\text{hash})$. This value, called the **signature block**, is appended to the document and sent to the receiver, as shown in Fig. 9-3. The application of D to the hash is sometimes referred to as decrypting the hash, but it is not really a decryption because the hash has not been encrypted. It is just a mathematical transformation on the hash.

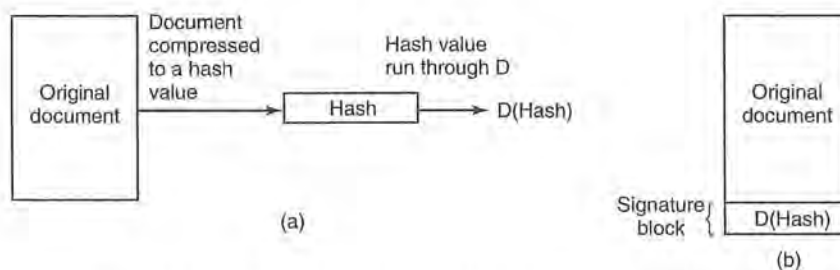


Figure 9-3. (a) Computing a signature block. (b) What the receiver gets.

When the document and hash arrive, the receiver first computes the hash of the document using MD5 or SHA, as agreed upon in advance. The receiver then applies the sender's public key to the signature block to get $E(D(\text{hash}))$. In effect, it encrypts the decrypted hash, canceling it out and getting the hash back. If the computed hash does not match the hash from the signature block, the document, the signature block, or both have been tampered with (or changed by accident). The value of this scheme is that it applies (slow) public-key cryptography only to a relatively small piece of data, the hash. Note carefully that this method works only if for all x

$$E(D(x)) = x$$

It is not guaranteed a priori that all encryption functions will have this property since all that we originally asked for was that

$$D(E(x)) = x$$

that is, E is the encryption function and D is the decryption function. To get the signature property in addition, the order of application must not matter, that is, D and E must be commutative functions. Fortunately, the RSA algorithm has this property.

To use this signature scheme, the receiver must know the sender's public key. Some users publish their public key on their Web page. Others do not because they may be afraid of an intruder breaking in and secretly altering their key. For them, an alternative mechanism is needed to distribute public keys. One common method is for message senders to attach a **certificate** to the message, which contains the user's name and public key and digitally signed by a trusted third party. Once the user has acquired the public key of the trusted third party, he can accept certificates from all senders who use this trusted third party to generate their certificates.

Above we have described how public-key cryptography can be used for digital signatures. It is worth mentioning that schemes that do not involve public-key cryptography also exist.

9.3 USER AUTHENTICATION

Now that we have some cryptographic background, let us start looking at security issues in operating systems. When a user logs into a computer, the operating system normally wishes to determine who the user is, a process called **user authentication**.

User authentication is one of those things we meant by "ontogeny recapitulates phylogeny" in Sec. 1.2.5. Early mainframes, such as the ENIAC, did not have an operating system, let alone a login procedure. Later mainframe batch and timesharing systems generally did have a login procedure for authenticating jobs and users.

Early minicomputers (e.g., PDP-1 and PDP-8) did not have a login procedure, but with the spread of UNIX on the PDP-11 minicomputer, logging in was again needed. Early personal computers (e.g., Apple II and the original IBM PC) did not have a login procedure, but more sophisticated personal computer operating systems, such as Windows 2000, again require a secure login. Using a personal computer to access servers on a LAN (local area network) or one's account at an e-commerce Web site always requires logging in. Thus the subject of secure login has gone through several cycles, and is once again an important subject.

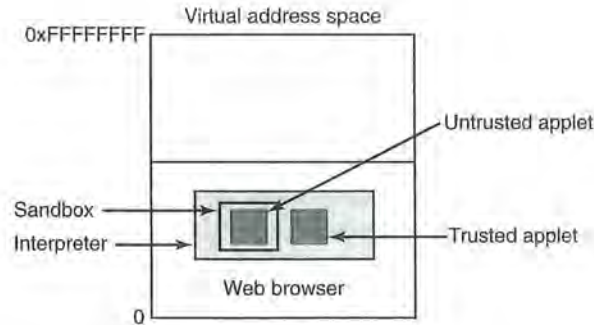


Figure 9-19. Applets can be interpreted by a Web browser.

applet is not trusted (e.g., it came in over the Internet), it could be put in what is effectively a sandbox to restrict its behavior.

High-level scripting languages can also be interpreted. Here no machine addresses are used, so there is no danger of a script trying to access memory in an impermissible way. The downside of interpretation in general is that it is very slow compared to running native compiled code.

Code

Yet another way to deal with applet security is to know where they came from and only accept applets from trusted sources. With this approach, a user can maintain a list of trusted applet vendors and only run applets from those vendors. Applets from all other sources are rejected as too dicey. In this approach, no actual security mechanisms are present at run time. Applets from trustworthy vendors are run as is and code from other vendors is not run at all or in a restricted way (sandboxed or interpreted with little or no access to user files and other system resources).

To make this scheme work, as a minimum, there has to be a way for a user to determine that an applet was written by a trustworthy vendor and not modified by anyone after being produced. This is done using a digital signature, which allows the vendor to sign the applet in such a way that future modifications can be detected.

Code signing is based on public-key cryptography. An applet vendor, typically a software company, generates a (public key, private key) pair, making the former public and zealously guarding the latter. To sign an applet, the vendor first computes a hash function of the applet to get a 128-bit or 160-bit number, depending on whether MD5 or SHA is used. It then signs the hash value by encrypting it with its private key (actually, decrypting it using the notation of Fig. 9-3). This signature accompanies the applet wherever it goes.

When the user gets the applet, the browser computes the hash function itself. It then decrypts the accompanying signature using the vendor's public key and compares what the vendor claims the hash function is with what the browser itself computed. If they agree, the applet is accepted as genuine. Otherwise it is rejected as a forgery. The mathematics involved makes it exceedingly difficult for anyone to tamper with the applet in such a way as its hash function will match the hash function that is obtained by decrypting the genuine signature. It is equally difficult to generate a new false signature that matches without having the private key. The process of signing and verifying is illustrated in Fig. 9-20.

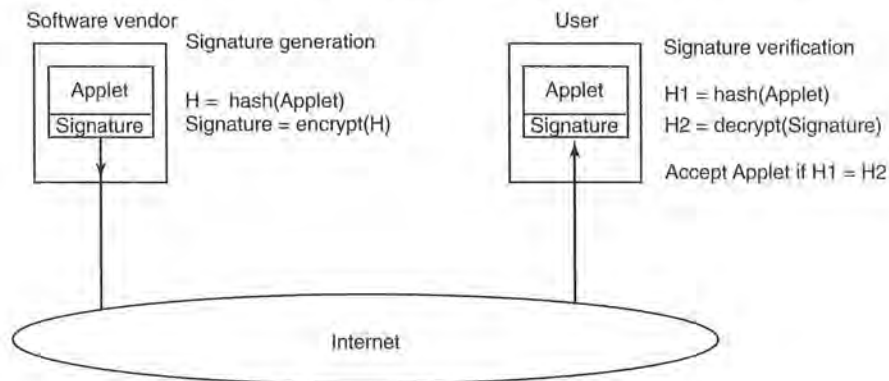


Figure 9-20. How code signing works.

9.5.7 Java Security

The Java programming language and accompanying run-time system were designed to allow a program to be written and compiled once and then shipped over the Internet in binary form and run on any machine supporting Java. Security was a part of the Java design from the beginning. In this section we will describe how it works.

Java is a type-safe language, meaning that the compiler will reject any attempt to use a variable in a way not compatible with its type. In contrast, consider the following C code:

```
naughty_func()
{
    char *p;
    p = rand();
    *p = 0;
}
```

It generates a random number and stores it in the pointer p . Then it stores a 0 byte at the address contained in p , overwriting whatever was there, code or data. In

Java, constructions that mix types like this are forbidden by the grammar. In addition, Java has no pointer variables, casts, user-controlled storage allocation (such as *malloc* and *free*) and all array references are checked at run time.

Java programs are compiled to an intermediate binary code called **JVM (Java Virtual Machine) byte code**. JVM has about 100 instructions, most of which push objects of a specific type onto the stack, pop them from the stack, or combine two items on the stack arithmetically. These JVM programs are typically interpreted, although in some cases they can be compiled into machine language for faster execution. In the Java model, applets sent over the Internet for remote execution are JVM programs.

When an applet arrives, it is run through a JVM byte code verifier that checks if the applet obeys certain rules. A properly compiled applet will automatically obey them, but there is nothing to prevent a malicious user from writing a JVM applet in JVM assembly language. The checks include

1. Does the applet attempt to forge pointers?
2. Does it violate access restrictions on private class members?
3. Does it try to use a variable of one type as another type?
4. Does it generate stack overflows or underflows?
5. Does it illegally convert variables of one type to another?

If the applet passes all the tests, it can be safely run without fear that it will access memory other than its own.

However, applets can still make system calls by calling Java methods (procedures) provided for that purpose. The way Java deals with that has evolved over time. In the first version of Java, **JDK (Java Development Kit) 1.0**, applets were divided into two classes: trusted and untrusted. Applets fetched from the local disk were trusted and allowed to make any system calls they wanted. In contrast, applets fetched over the Internet were untrusted. They were run in a sandbox, as shown in Fig. 9-19, and allowed to do practically nothing.

After some experience with this model, Sun decided that it was too restrictive. In JDK 1.1, code signing was employed. When an applet arrived over the Internet, a check was made to see if it was signed by a person or organization the user trusted (as defined by the user's list of trusted signers). If so, the applet was allowed to do whatever it wanted. If not, it was run in a sandbox and severely restricted.

After more experience, this proved unsatisfactory as well, so the security model was changed again. JDK 1.2 provides a configurable fine-grain security policy that applies to all applets, both local and remote. The security model is complicated enough that an entire book can be written describing it (Gong, 1999), so we will just briefly summarize some of the highlights.

Each applet is characterized by two things: where it came from and who signed it. Where it came from is its URL; who signed it is which private key was used for the signature. Each user can create a security policy consisting of a list of rules. A rule may list a URL, a signer, an object, and an action that the applet may perform on the object if the applet's URL and signer match the rule. Conceptually, the information provided is shown in the table of Fig. 9-21, although the actual formatting is different and is related to the Java class hierarchy.

URL	Signer	Object	Action
www.taxprep.com	TaxPrep	/usr/susan/1040.xls	Read
*		/usr/tmp/*	Read, Write
www.microsoft.com	Microsoft	/usr/susan/Office/—	Read, Write, Delete

Figure 9-21. Some examples of protection that can be specified with JDK 1.2.

One kind of action permits file access. The action can specify a specific file or directory, the set of all files in a given directory, or the set of all files and directories recursively contained in a given directory. The three lines of Fig. 9-21 correspond to these three cases. In the first line, the user, Susan, has set up her permissions file so that applets originating at her tax preparer's machine, *www.taxprep.com* and signed by the company, have read access to her tax data located in the file *1040.xls*. This is the only file they can read and no other applets can read this file. In addition, all applets from all sources, whether signed or not, can read and write files in */usr/tmp*.

Furthermore, Susan also trusts Microsoft enough to allow applets originating at its site and signed by Microsoft to read, write, and delete all the files below the *Office* directory in the directory tree, for example, to fix bugs and install new versions of the software. To verify the signatures, Susan must either have the necessary public keys on her disk or must acquire them dynamically, for example in the form of a certificate signed by a company she trusts and whose public key she already has.

Files are not the only resources that can be protected. Network access can also be protected. The objects here are specific ports on specific computers. A computer is specified by an IP address or DNS name; ports on that machine are specified by a range of numbers. The possible actions include asking to connect to the remote computer and accepting connections originated by the remote computer. In this way, an applet can be given network access, but restricted to talking only to computers explicitly named in the permissions list. Applets may dynamically load additional code (classes) as needed, but user-supplied class loaders can precisely control on which machines such classes may originate. Numerous other security features are also present.

SECOND EDITION



MODERN 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 material on a wealth of subjects have been added. This authoritative, example-based reference offers practical, hands-on information in constructing and understanding modern operating systems. Continued in this second edition are the "big picture" concepts, presented in the clear and entertaining style that only Andrew S. Tanenbaum can provide. Tanenbaum's long experience as the designer or co-designer of three operating systems brings a knowledge of the subject and wealth of practical detail that few other books can match.

FEATURES

- NEW—New chapters on computer security, multimedia operating systems, and multiple processor systems.
- NEW—Extensive coverage of Linux, UNIX®, and Windows 2000™ as examples.
- NEW—A unique chapter on operating system design not found in other books.
- NEW—Now includes coverage of graphical user interfaces, multiprocessor operating systems, trusted systems, viruses, network terminals, CD-ROM file systems, power management on laptops, RAID, soft timers, stable storage, fair-share scheduling, three-level scheduling, and new paging algorithms.
- NEW—Most chapters have a new section on current research on the chapter's topic.
- NEW—Focus on *single-processor* computer systems; a new book for a follow-up course on distributed systems is also available from Prentice Hall.
- NEW—Over 200 references to books and papers published since the first edition.
- NEW—The Web site for this book contains PowerPoint slides, simulators, figures in various formats, and other teaching aids.

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

Distributed Systems by Andrew S. Tanenbaum and Maarten van Steen, 2002

Operating Systems: Design and Implementation 2/e by Andrew S. Tanenbaum and Albert S. Woodhull, 1997

Structured Computer Organization 4/e by Andrew S. Tanenbaum, 1999

Computer Networks 3/e by Andrew S. Tanenbaum, 1996

InformIT
www.informit.com

**Pearson
Education**

Prentice Hall
Upper Saddle River, NJ 074
<http://www.prenhall.com>

ISBN 0-13-021358-2



X0045EEKOP

Modern Operating Systems
Used - G