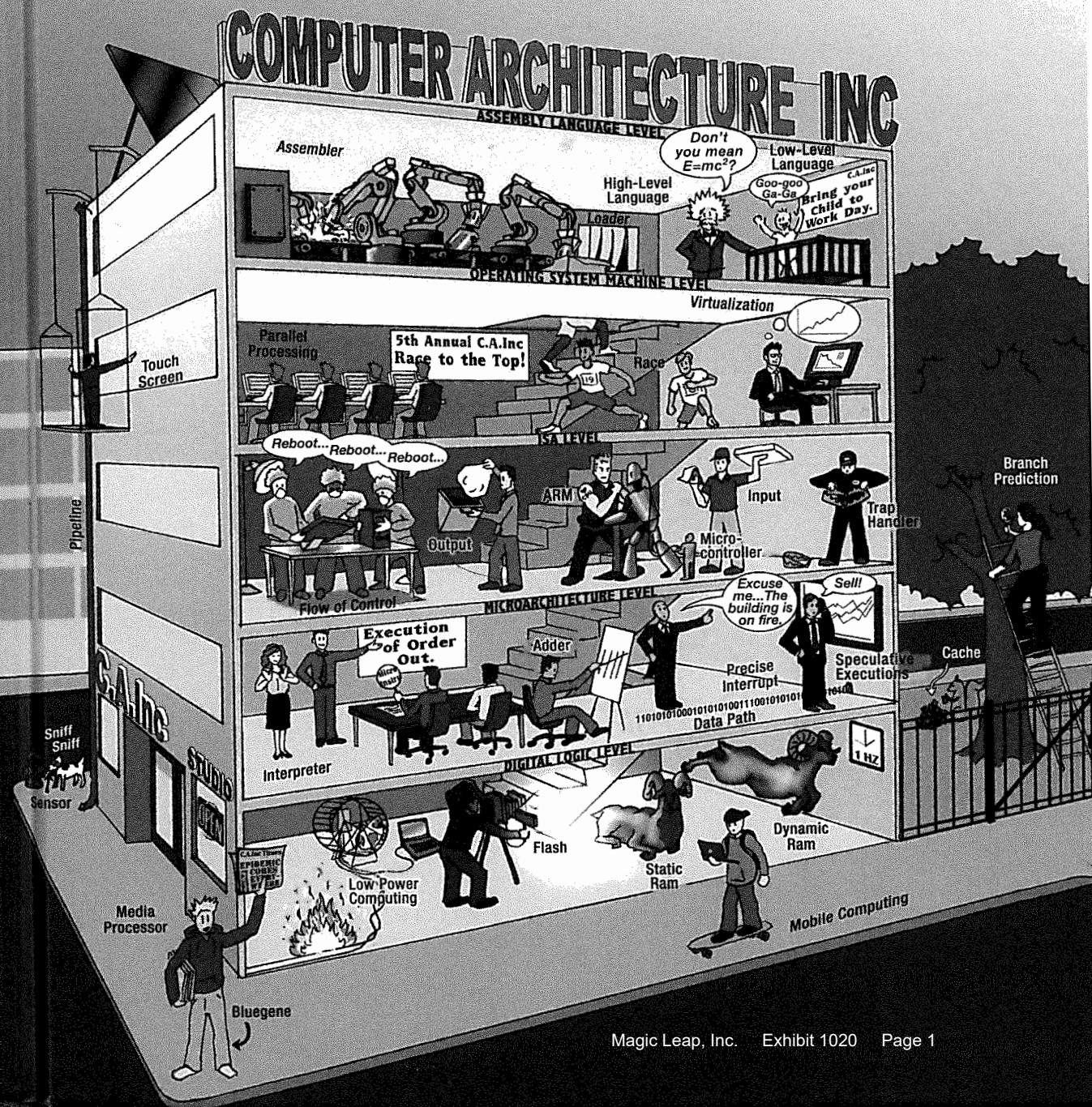


STRUCTURED COMPUTER ORGANIZATION

SIXTH EDITION

Andrew S. Tanenbaum • Todd Ausfin



STRUCTURED COMPUTER ORGANIZATION

SIXTH EDITION

ANDREW S. TANENBAUM

*Vrije Universiteit
Amsterdam, The Netherlands*

TODD AUSTIN

*University of Michigan
Ann Arbor, Michigan, United States*

PEARSON

Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Editorial Director, ECS: Marcia Horton
Executive Editor: Tracy Johnson (Dunkelberger)
Associate Editor: Carole Snyder
Director of Marketing: Christy Lesko
Marketing Manager: Yez Alayan
Senior Marketing Coordinator: Kathryn Ferranti
Director of Production: Erin Gregg
Managing Editor: Jeff Holcomb
Associate Managing Editor: Robert Engelhardt
Manufacturing Buyer: Lisa McDowell
Art Director: Anthony Gemmellaro
Cover Illustrator: Jason Consalvo
Manager, Rights and Permissions: Michael Joyce
Media Editor: Daniel Sandin
Media Project Manager: Renata Butera
Printer/Binder: LSC Communications
Cover Printer: LSC Communications

Credits and acknowledgments borrowed from other sources and reproduced, with permission, in this textbook appear in the Credits section in the end matter of this text.

Copyright © 2013, 2006, 1999 Pearson Education, Inc., publishing as Prentice Hall. All rights reserved. Printed in the United States of America. This publication is protected by Copyright, and permission should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission(s) to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to 201-236-3290.

Many of the designations by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed in initial caps or all caps.

Library of Congress Cataloging-in-Publication Data

Tanenbaum, Andrew S.,

Structured computer organization / Andrew S. Tanenbaum, Todd Austin. -- 6th ed.
p. cm.

Includes bibliographical references and index.

ISBN-13: 978-0-13-291652-3

ISBN-10: 0-13-291652-5

1. Computer programming. 2. Computer organization. I. Austin, Todd. II. Title.

QA76.6.T38 2013

005.1--dc23

2012021627

5 2021

PEARSON

ISBN 10: 0-13-291652-5

ISBN 13: 978-0-13-291652-3

Magic Leap, Inc. Exhibit 1020 Page 3

the semaphore until the first process has either completed the operation or been suspended trying. Furthermore, with semaphores, no wakeups are lost. In contrast, the if statements of Figure 6-26 are not indivisible. Between the evaluation of the condition and the execution of the selected statement, another process can send a wakeup signal.

In effect the problem of process synchronization has been eliminated by declaring the up and down system calls made by *up* and *down* to be indivisible. In order for these operations to be indivisible, the operating system must prohibit two or more processes from using the same semaphore at the same time. At the very least, once an up or down system call has been made, no other user code will be run until the call has been completed. On single-processor systems, semaphores are sometimes implemented by disabling interrupts during semaphore operations. On multiple-processor systems, this trick does not work.

Synchronization using semaphores is a technique that works for arbitrarily many processes. Several processes may be sleeping, attempting to complete a down system call on the same semaphore. When some other process finally performs an up on that semaphore, one of the waiting processes is allowed to complete its down and continue running. The semaphore value remains 0 and the other processes continue waiting.

An analogy may make the nature of semaphores clearer. Imagine a picnic with 20 volleyball teams divided into 10 games (processes), each playing on its own court, and a large basket (the semaphore) for the volleyballs. Unfortunately, only seven volleyballs are available. At any instant, there are between zero and seven volleyballs in the basket (the semaphore has a value between 0 and 7). Putting a ball in the basket is an up because it increases the value of the semaphore; taking a ball out of the basket is a down because it decreases the value.

At the start of the picnic, each court sends a player to the basket to get a volleyball. Seven of them successfully manage to get a volleyball (complete the down); three are forced to wait for a volleyball (i.e., fail to complete the down). Their games are suspended temporarily. Eventually, one of the other games finishes and puts a ball into the basket (executes an up). This operation allows one of the three players waiting around the basket to get a ball (complete an unfinished down), allowing one game to continue. The other two games remain suspended until two more balls are put into the basket. When two more balls come back (two more ups are executed), the last two games can proceed.

6.5 EXAMPLE OPERATING SYSTEMS

In this section we will continue discussing our example systems, the Core i7 and the OMAP4430 ARM CPU. For each one we will look at an operating system used on that processor. For the Core i7 we will use Windows; for the OMAP4430 ARM CPU we will use UNIX. Since UNIX is simpler and in many ways more

```

public class m {
    final public static int BUF_SIZE = 100;           // buffer runs from 0 to 99
    final public static long MAX_PRIME = 100000000000L; // stop here
    public static int in = 0, out = 0;                // pointers to the data
    public static long buffer[] = new long[BUF_SIZE]; // primes stored here
    public static producer p;                         // name of the producer
    public static consumer c;                         // name of the consumer
    public static int filled = 0, available = 100;    // semaphores

    public static void main(String args[] ) {         // main class
        p = new producer( );                         // create the producer
        c = new consumer( );                         // create the consumer
        p.start( );                                  // start the producer
        c.start( );                                  // start the consumer
    }

    // This is a utility function for circularly incrementing in and out
    public static int next(int k) {if (k < BUF_SIZE - 1) return(k+1); else return(0);}
}

class producer extends Thread {                     // producer class
    native void up(int s); native void down(int s);  // methods on semaphores
    public void run( ) {                             // producer code
        long prime = 2;                             // scratch variable

        while (prime < m.MAX_PRIME) {
            prime = next_prime(prime);                // statement P1
            down(m.available);                         // statement P2
            m.buffer[m.in] = prime;                   // statement P3
            m.in = m.next(m.in);                      // statement P4
            up(m.filled);                             // statement P5
        }
    }

    private long next_prime(long prime){ ... }        // function that computes next prime
}

class consumer extends Thread {                     // consumer class
    native void up(int s); native void down(int s);  // methods on semaphores
    public void run( ) {                             // consumer code
        long emirp = 2;                             // scratch variable

        while (emirp < m.MAX_PRIME) {
            down(m.filled);                           // statement C1
            emirp = m.buffer[m.out];                   // statement C2
            m.out = m.next(m.out);                     // statement C3
            up(m.available);                           // statement C4
            System.out.println(emirp);                 // statement C5
        }
    }
}

```

Figure 6-29. Parallel processing using semaphores.

elegant, we will begin with it. Also, UNIX was designed and implemented first and had a major influence on Windows 7, so this order makes more sense than the reverse.

6.5.1 Introduction

In this section we will give a brief introduction to our two example operating systems, UNIX and Windows 7, focusing on the history, structure, and system calls.

UNIX

UNIX was developed at Bell Labs in the early 1970s. The first version was written by Ken Thompson in assembler for the PDP-7 minicomputer. This was soon followed by a version for the PDP-11, written in a new language called C that was devised and implemented by Dennis Ritchie. In 1974, Ritchie and his colleague Ken Thompson published a landmark paper about UNIX (Ritchie and Thompson, 1974). For the work described in this paper they were later given the prestigious ACM Turing Award (Ritchie, 1984, Thompson, 1984). The publication of this paper stimulated many universities to ask Bell Labs for a copy of UNIX. Since Bell Labs' parent company, AT&T, was a regulated monopoly at the time and was not permitted to be in the computer business, it had no objection to licensing UNIX to universities for a modest fee.

In one of those coincidences that often shape history, the PDP-11 was the computer of choice at nearly all university computer science departments, and the operating systems that came with the PDP-11 were widely regarded as being dreadful by professors and students alike. UNIX quickly filled the void, not in the least because it was supplied with the complete source code, so people could, and did, tinker with it endlessly.

One of the many universities that acquired UNIX early on was the University of California at Berkeley. Because the complete source code was available, Berkeley was able to modify the system substantially. Foremost among the changes was a port to the VAX minicomputer and the addition of paged virtual memory, the extension of file names from 14 characters to 255 characters, and the inclusion of the TCP/IP networking protocol, which is now used on the Internet (largely due to the fact that it was in Berkeley UNIX).

While Berkeley was making all these changes, AT&T itself continued to develop UNIX, leading to System III in 1982 and then System V in 1984. By the late 1980s, two different, and quite incompatible, versions of UNIX were in widespread use: Berkeley UNIX and System V. This split in the UNIX world, together with the fact that there were no standards for binary program formats, greatly inhibited the commercial success of UNIX because it was impossible for software vendors to write and package UNIX programs with the expectation that they would run on any UNIX system (as was routinely done then with MS-DOS). After much

bickering, a standard called **POSIX (Portable Operating System-IX)** was created by the IEEE Standards Board. POSIX is also known by its IEEE Standards number, P1003. It later became an International Standard.

The standard is divided into many parts, each one covering a different area of UNIX. The first part, P1003.1, defines the system calls; the second part, P1003.2, defines the basic utility programs, and so on. The P1003.1 standard defines about 60 system calls that all conformant systems must support. These are the basic calls for reading and writing files, creating new processes, and so on. Nearly all UNIX systems now support the P1003.1 system calls. However many UNIX systems also support extra system calls, especially those defined by System V and/or those in Berkeley UNIX. Typically these add up to 200 system calls.

In 1987, one author of this book (Tanenbaum) released the source code for a tiny version of UNIX, called MINIX, for use at universities (Tanenbaum, 1987). One of the students who studied MINIX at his university in Helsinki and ran it on his home PC was Linus Torvalds. After becoming thoroughly familiar with MINIX, Torvalds decided to write his own clone of MINIX, which was called Linux and has become quite popular.

Many operating systems running today on ARM platforms are based on Linux. Both MINIX and Linux are POSIX conformant, and nearly everything said about UNIX in this chapter also applies to them unless stated otherwise.

A rough breakdown of the Linux system calls by category is given in Fig. 6-30. The file- and directory-management system calls are the largest and the most important categories. Linux is mostly POSIX P1003.1 compliant, although the developers did deviate from the specification in some areas. In general, however, it is not difficult to get POSIX-compliant programs to build and run on Linux.

Category	Some examples
File management	Open, read, write, close, and lock files
Directory management	Create and delete directories; move files around
Process management	Spawn, terminate, trace, and signal processes
Memory management	Share memory among processes; protect pages
Getting/setting parameters	Get user, group, process ID; set priority
Dates and times	Set file access times; use interval timer; profile execution
Networking	Establish/accept connection; send/receive message
Miscellaneous	Enable accounting; manipulate disk quotas; reboot the system

Figure 6-30. A rough breakdown of the UNIX system calls.

One area that is largely due to Berkeley UNIX rather than System V is networking. Berkeley invented the concept of a **socket**, which is the endpoint of a network connection. The four-pin wall plugs to which telephones can be connected served as the model for this concept. It is possible for a UNIX process to create a socket, attach to it, and establish a connection to a socket on a distant

machine. Over this connection it can then exchange data in both directions, typically using the TCP/IP protocol. Since networking technology has been in UNIX for decades and is very stable and mature, a substantial fraction of the servers on the Internet run UNIX.

Since there are many implementations of UNIX, it is difficult to say much about the structure of the operating system since each one is somewhat different from all the others. However, in general, Fig. 6-31 applies to most of them. At the bottom, there is a layer of device drivers that shield the file system from the bare hardware. Originally, each device driver was written as an independent entity, separate from all the others. This arrangement led to a lot of duplicated effort, since many drivers must deal with flow control, error handling, priorities, separating data from control, and so on. This observation led Dennis Ritchie to develop a framework called **streams** for writing drivers in a modular way. With a stream, it is possible to establish a two-way connection from a user process to a hardware device and to insert one or more modules along the path. The user process pushes data into the stream, which then is processed or transformed by each module until it gets to the hardware. The inverse processing occurs for incoming data.

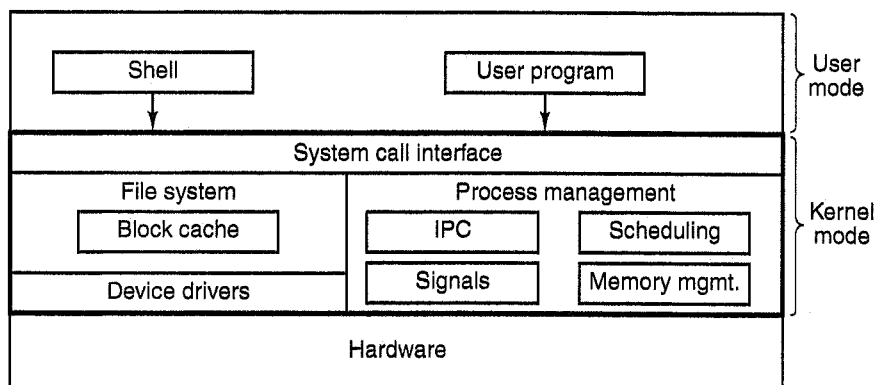


Figure 6-31. The structure of a typical UNIX system.

On top of the device drivers comes the file system. It manages file names, directories, disk block allocation, protection, and much more. Part of the file system is a **block cache**, for holding the blocks most recently read in from disk, in case they are needed again soon. A variety of file systems have been used over the years, including the Berkeley fast file system (McKusick et al., 1984), and log-structured file systems (Rosenblum and Ousterhout, 1991, and Seltzer et al., 1993).

The other part of the UNIX kernel is the process-management portion. Among its various other functions, it handles IPC (InterProcess Communication), which allows processes to communicate with one another and synchronize to avoid race conditions. A variety of mechanisms are provided. The process-management code

also handles process scheduling, which is based on priorities. Signals, which are a form of (asynchronous) software interrupt, are also managed here. Finally, memory management is done here as well. Most UNIX systems support demand-paged virtual memory, sometimes with a few extra features, such as the ability of multiple processes to share common regions of address space.

From its inception, UNIX has tried to be a small system, in order to enhance reliability and performance. The first versions of UNIX were entirely text based, using terminals that could display 24 or 25 lines of 80 ASCII characters. The user interface was handled by a user-level program called the **shell**, which offered a command-line interface. Since the shell was not part of the kernel, adding new shells to UNIX was easy, and over time a number of increasingly sophisticated ones were invented.

Later on, when graphics terminals came into existence, a windowing system for UNIX, called **X Windows**, was developed at M.I.T. Still later, a full-fledged GUI (**Graphical User Interface**), called **Motif**, was put on top of X Windows. These GUIs eventually developed into full-blown desktop environments with beautifully rendered window management, productivity tools, and utilities. Examples of these desktop environments include GNOME and KDE. In keeping with the UNIX philosophy of having a small kernel, nearly all the code of X Windows and its accompanying GUIs run in user mode, outside the kernel.

Windows 7

When the original IBM PC was launched in 1981, it came equipped with a 16-bit real-mode, single-user, command-line-oriented operating system called MS-DOS 1.0. This operating system consisted of 8 KB of memory-resident code. Two years later, a much more powerful 24-KB system, MS-DOS 2.0, appeared. It contained a command-line processor (shell), with a number of features borrowed from UNIX. When IBM released the 286-based PC/AT in 1984, it came equipped with MS-DOS 3.0, by now 36 KB. Over the years, MS-DOS continued to acquire new features, but it was still a command-line-oriented system.

Inspired by the success of the Apple Macintosh, Microsoft decided to give MS-DOS a graphical user interface that it called **Windows**. The first three versions of Windows, culminating in Windows 3.x, were not true operating systems but graphical user interfaces on top of MS-DOS, which was still in control of the machine. All programs ran in the same address space and a bug in any one of them could bring the whole system to a grinding halt.

The release of Windows 95 in 1995 still did not eliminate MS-DOS, although it introduced a new version, 7.0. Together, Windows 95 and MS-DOS 7.0 contained most of the features of a full-blown operating system, including virtual memory, process management, and multiprogramming. However, Windows 95 was not a full 32-bit program. It contained large chunks of old 16-bit code (as well as some 32-bit code) and still used the MS-DOS file system, with nearly all its limitations.

The only major changes to the file system were the addition of long file names in place of the 8 + 3 character file names allowed in MS-DOS and the ability to have more than 65,536 blocks on a disk.

Even with the release of Windows 98 in 1998, MS-DOS was still there (now called version 7.1) and running 16-bit code. Although a bit more functionality migrated from the MS-DOS part to the Windows part, and a disk layout suitable for larger disks was now standard, under the hood, Windows 98 was not very different from Windows 95. The main difference was the user interface, which integrated the desktop, the Internet, and to some extent, even television more closely. It was precisely this integration that attracted the attention of the U.S. Dept. of Justice, which then sued Microsoft claiming that it was an illegal monopoly. Windows 98 was followed by the short-lived Windows Millennium Edition (ME), which was a slightly improved Windows 98.

While all these developments were going on, Microsoft was also busy with a completely new 32-bit operating system being written from the ground up. This new system was called **Windows New Technology**, or **Windows NT**. It was initially hyped as the replacement for all other operating systems for Intel-based PCs (as well as the MIPS PowerPC chips), but it was somewhat slow to catch on and was later redirected to the upper end of the market, where it found a niche on large servers. The second version of NT was called Windows 2000 and became the mainstream version, also for the desktop market. The successor to Windows 2000 was Windows XP, but the changes here were relatively minor (better backward compatibility and a few more features). In 2007, the followup Windows Vista was released. Vista implemented many graphical enhancements over Windows XP, and it added many new user applications, such as a media center. Vista's adoption was slow because of its poor performance and high resource demands. A mere two years later, Windows 7 was released, which by all accounts is a tuned-up version of Windows Vista. Windows 7 runs much better on older hardware, and it requires significantly less hardware resources.

Windows 7 is sold in six different versions. Three are for home users in various countries, two are aimed at business users, and one combines all the features of all versions. These versions are nearly identical and differ primarily in focus, advanced features, and optimizations made. We will focus on the core features and not make any further distinction between these versions.

Before getting into the interface Windows 7 presents to programmers, let us take a very quick look at its internal structure, which is illustrated in Fig. 6-32. It consists of a number of modules that are structured in layers and work together to implement the operating system. Each module has some particular function and a well-defined interface to the other modules. Nearly all the modules are written in C, although part of the graphics device interface is written in C++ and tiny bits of the lowest layers are written in assembly language.

At the bottom is a thin layer called the **hardware abstraction layer**. Its job is to present the rest of the operating system with abstract hardware devices, devoid

of the warts and idiosyncracies with which real hardware is so richly endowed. Among the devices modeled are off-chip caches, timers, I/O buses, interrupt controllers, and DMA controllers. By exposing these to the rest of the operating system in idealized form, it becomes easier to port Windows 7 to other hardware platforms, since most of the modifications required are concentrated in one place.

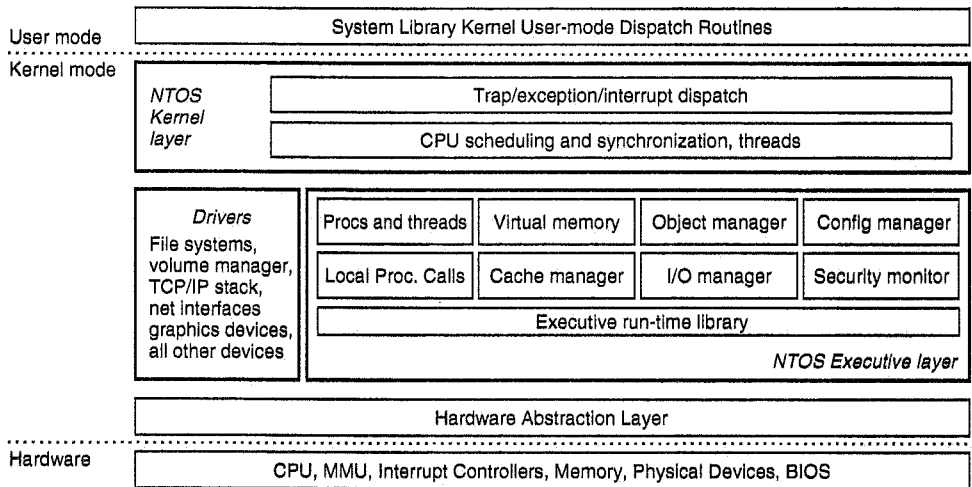


Figure 6-32. The structure of Windows 7.

Above the HAL, the code is divided into two major parts, the **NTOS executive** and the **Windows drivers**, which includes the file systems, networking, and graphics code. On top of that is the kernel layer. All of this code runs in protected kernel mode.

The executive manages the fundamental abstractions used in Windows 7, including threads, processes, virtual memory, kernel objects, and configurations. Also here are the managers for local procedure calls, the file cache, I/O, and security.

The kernel layer handles trap and exception handling, as well as scheduling and synchronization.

Outside the kernel are the user programs and the system library used to interface to the operating system. In contrast to UNIX systems, Microsoft does not encourage user programs to make direct system calls. Instead they are expected to call procedures in the library. To provide standardization across different versions of Windows (e.g., XP, Vista, and Windows 7), Microsoft defined a set of calls called the **Win32 API (Application Programming Interface)**. These are library procedures that either make system calls to get the work done, or, in some case, do the work right in the user-space library procedure. Although many Windows 7 library calls have been added since Win32 was defined, these are the core calls and it is them we will focus on. Later, when Windows was ported to 64-bit machines,

Microsoft changed the name of Win32 to cover both the 32-bit and 64-bit versions, but for our purposes, looking at the 32-bit version is sufficient.

The Win32 API philosophy is completely different from the UNIX philosophy. In the latter, the system calls are all publicly known and form a minimal interface: removing even one of them would reduce the functionality of the operating system. The Win32 philosophy is to provide a very comprehensive interface, often with three or four ways of doing the same thing, and including many functions that clearly should not be (and are not) system calls, such as an API call to copy an entire file.

Many Win32 API calls create kernel objects of one kind or another, including files, processes, threads, pipes, etc. Every call creating a kernel object returns a result called a **handle** to the called. This handle can be subsequently used to perform operations on the object. Handles are specific to the process that created the object referred to by the handle. They cannot be passed directly to another process and used there (just as UNIX file descriptors cannot be passed to other processes and used there). However, under certain circumstances, it is possible to duplicate a handle and pass it to other processes in a protected way, allowing them controlled access to objects belonging to other processes. Every object can have a security descriptor associated with it, telling in detail who may and may not perform what kinds of operations on the object.

Windows 7 is sometimes said to be object oriented because the only way to manipulate kernel objects is by invoking methods (API functions) on their handles, which are returned when the objects are created. On the other hand, it lacks some of the most basic properties of object-oriented systems such as inheritance and polymorphism.

6.5.2 Examples of Virtual Memory

In this section we will look at virtual memory in both UNIX and Windows 7. For the most part, they are fairly similar from the programmer's point of view.

UNIX Virtual Memory

The UNIX memory model is simple. Each process has three segments: code, data, and stack, as illustrated in Fig. 6-33. In a machine with a single, linear address space, the code is generally placed near the bottom of memory, followed by the data. The stack is placed at the top of memory. The code size is fixed, but the data and stack may each grow, in opposite directions. This model is easy to implement on almost any machine and is the model used by Linux variants that run on OMAP4430 ARM CPUs.

Furthermore, if the machine has paging, the entire address space can be paged, without user programs even being aware of it. The only thing they notice is that it