PKFOKAM INSTITUTE OF EXCELLENCE

Course support

OPERATING SYSTEM

Narcisse TALLA
PhD, Fotso Victor University Institute of Technology
University of Dschang, Cameroon
ntalla@gmail.com

September 2015

SUMMARY

CHAPTER 1. INTRODUCTION

1.1 PRELIMINARY

1.1.1 Essential reading

- R. Bryant and D. O'Hallaron, *Computer Systems : A Programmers Perspective, Second Edition*, Addison-Wesley (2010).
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, MIT Press (2009).
- H. Deitel, P. Deitel,and, D. Choffnes, *Operating Systems, Third Edition*, Prentice Hall (2004).
- J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach, Fifth Edition*, Morgan Kaufmann (2012).
- R. Love, *Linux Kernel Development, Third Edition*, Developer's Library (2010).
- R. McDougall and J. Mauro, *Solaris Internals, Second Edition,* Prentice Hall (2007).
- M. E. Russinovich and D. A. Solomon, *Windows Internals: Including Windows Server 2008 and Windows Vista,Fifth Edition*, Microsoft Press (2009).
- A. Singh,Mac OS X Internals: A Systems Approach, AddisonWesley (2007).
- W. Stallings, *Operating Systems, Seventh Edition*, Prentice Hall (2011).
- A. S. Tanenbaum, *Modern Operating Systems, Third Edition*, Prentice Hall (2007).
- S. Tarkoma and E. Lagerspetz, "*Arching over the Mobile Computing Chasm: Platforms and Runtimes*", IEEE Computer, Volume 44, (2011), pages 22–28.

1.1.2 Global overview

An operating system is a program that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware. An amazing aspect of operating systems is how they vary in accomplishing these tasks. Mainframe operating systems are designed primarily to optimize utilization of hardware. Personal computer (PC) operating systems support complex games, business applications, and everything in between. Operating systems for mobile computers provide an environment in which a user can easily interface with the computer to execute programs. Thus, some operating systems are designed to be convenient, others to be efficient, and others to be some combination of the two. Before we can explore the details of computer system operation, we need to know something about system structure. We thus discuss the basic functions of system startup, I/O, and storage early in this chapter. We also describe the basic computer architecture that makes it possible to write a functional operating system.

Because an operating system is large and complex, it must be created piece by piece. Each of these pieces should be a well-delineated portion of the system, with carefully defined inputs, outputs, and functions. In this chapter, we provide a general overview of the major components of a contemporary computer system as well as the functions provided by the operating system.

Additionally, we cover several other topics to help set the stage for the remainder of this text: data structures used in operating systems, computing environments, and open-source operating systems.

1.1.3 Learning outcomes

Having followed this course and consulted relevant documents, each student should be able to

- describe the basic organization of computer systems.
- provide a grand tour of the major components of operating systems.
- give an overview of the many types of computing environments.
- explore several open-source operating systems.

1.2 INTRODUCTION

1.2.1 What Operating Systems Do

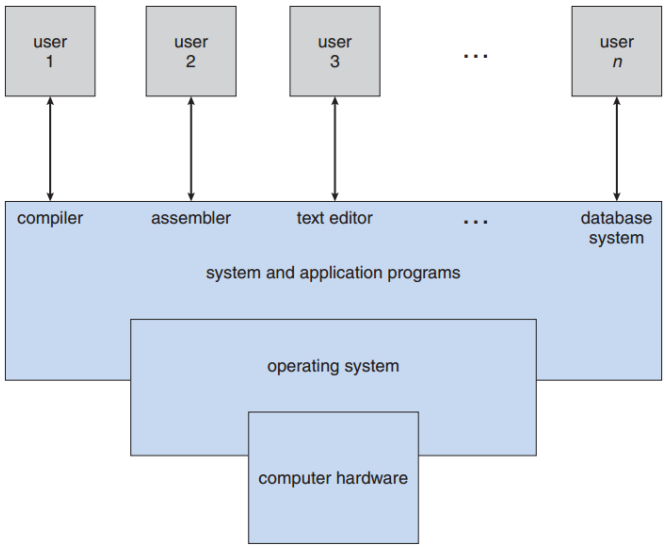


Figure 1.1. Abstract view of the components of a computer system

We begin our discussion by looking at the operating system's role in the overall computer system. A computer system can be divided roughly into four components: thehardware,the operating system,the application programs, and theusers(Figure 1.1).

Thehardware—thecentral processing unit(CPU), thememory, and the input/output(I/O) devices—provides the basic computing resources for the system. Theapplication programs—such as word processors, spreadsheets, compilers, and Web browsers—define the ways in which these resources are used to solve users' computing problems. The operating system controls the hardware and coordinates its use among the various application programs for the various users.

We can also view a computer system as consisting of hardware, software, and data. The operating system provides the means for proper use of these resources in the operation

of the computer system. An operating system is similar to a government. Like a government, it performs no useful function by itself. It simply provides an environment within which other programs can do useful work. To understand more fully the operating system's role, we next explore operating systems from two viewpoints: that of the user and that of the system.

1.2.1.1 User View

The user's view of the computer varies according to the interface being used. Most computer users sit in front of a PC, consisting of a monitor, keyboard, mouse, and system unit. Such a system is designed for one user to monopolize its resources. The goal is to maximize the work (or play) that the user is performing. In this case, the operating system is designed mostly for ease of use, with some attention paid to performance and none paid to resource utilization—how various hardware and software resources are shared. Performance is, of course, important to the user; but such systems are optimized for the single-user experience rather than the requirements of multiple users.

In other cases, a user sits at a terminal connected to a main frame or a minicomputer. Other users are accessing the same computer through other terminals. These users share resources and may exchange information. The operating system in such cases is designed to maximize resource utilization— to assure that all available CPU time, memory, and I/O are used efficiently and that no individual user takes more than her fair share.

In still other cases, users sit at workstationsconnected to networks of other workstations andservers. These users have dedicated resources at their disposal, but they also share resources such as networking and servers, including file, compute, and print servers. Therefore, their operating system is designed to compromise between individual usability and resource utilization.

Recently, many varieties of mobile computers, such as smartphones and tablets, have come into fashion. Most mobile computers are standalone units for individual users. Quite often, they are connected to networks through cellular or other wireless technologies. Increasingly, these mobile devices are replacing desktop and laptop computers for people who are primarily interested in using computers for e-mail and web browsing. The user interface for mobile computers generally features a touch

screen, where the user interacts with the system by pressing and swiping fingers across the screen rather than using a physical keyboard and mouse.

Some computers have little or no user view. For example, embedded computers in home devices and automobiles may have numeric keypads and may turn indicator lights on or off to show status, but they and their operating systems are designed primarily to run without user intervention.

1.2.1.2 System View

From the computer's point of view, the operating system is the program most intimately involved with the hardware. In this context, we can view an operating system as are source allocator. A computer system has many resources that may be required to solve a problem: CPU time, memory space, file-storage space, I/O devices, and so on. The operating system acts as the manager of these resources. Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them to specific programs and users so that it can operate the computer system efficiently and fairly. As we have seen, resource allocation is especially important where many users access the same mainframe or minicomputer.

A slightly different view of an operating system emphasizes the need to control the various I/O devices and user programs. An operating system is a control program. A control program manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

1.2.1.3 Defining Operating Systems

By now, we can that the term operating system covers many roles and functions. That is the case, at least in part, because of the myriad designs and uses of computers. Computers are present within toasters, cars, ships, spacecraft, homes, and businesses. They are the basis for game machines, music players, cable TV tuners, and industrial control systems. Although computers have a relatively short history, they have evolved rapidly. Computing started as an experiment to determine what could be done and quickly moved to fixed-purpose systems for military uses, such as code breaking and trajectory plotting, and governmental uses, such as census calculation. Those early computers evolved into general-purpose, multifunction mainframes, and that's when

operating systems were born. In the 1960s, Moore's Law predicted that the number of transistors on an integrated circuit would double every eighteen months, and that prediction has held true. Computers gained in functionality and shrunk in size, leading to a vast number of uses and a vast number and variety of operating systems.

How, then, can we define what an operating system is? In general, we have no completely adequate definition of an operating system. Operating systems exist because they offer a reasonable way to solve the problem of creating a usable computing system. The fundamental goal of computer systems is to execute user programs and to make solving user problems easier. Computer hardware is constructed toward this goal. Since bare hardware alone is not particularly easy to use, application programs are developed. These programs require certain common operations, such as those controlling the I/O devices.

The common functions of controlling and allocating resources are then brought together into one piece of software: the operating system.

In addition, we have no universally accepted definition of what is part of the operating system. A simple viewpoint is that it includes everything a vendor ships when you order "the operating system." The features included, however, vary greatly across systems. Some systems take up less than a megabyte of space and lack even a full-screen editor, whereas others require gigabytes of space and are based entirely on graphical windowing systems. A more common definition, and the one that we usually follow, is that the operating system is the one program running at all times on the computer—usually called the kernel. (Along with the kernel, there are two other types of programs: system programs, which are associated with the operating system but are not necessarily part of the kernel, and application programs, which include all programs not associated with the operation of the system.)

The matter of what constitutes an operating system became increasingly important as personal computers became more widespread and operating systems grew increasingly sophisticated. In 1998, the United States Department of Justice filed suit against Microsoft, in essence claiming that Microsoft included too much functionality in its operating systems and thus prevented application vendors from competing. (For example, a Web browser was an integral part of the operating systems.) As a result, Microsoft was found guilty of using its operating-system monopoly to limit competition.

Today, however, if we look at operating systems for mobile devices, we see that once again the number of features constituting the operating system is increasing. Mobile operating systems often include not only a core kernel but also middleware—a set of software frameworks that provide additional services to application developers. For example, each of the two most prominent mobile operating systems—Apple's iOSand Google's Android—features a core kernel along with middleware that supports databases, multimedia, and graphics (to name only a few).

1.2.2 Computer-System Organization

Before we can explore the details of how computer systems operate, we need general knowledge of the structure of a computer system. In this section, we look at several parts of this structure. The section is mostly concerned with computer-system organization, so you can skim or skip it if you already understand the concepts.

1.2.2.1 Computer-System Operation

A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common bus that provides access to shared memory (Figure 1.2). Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, or video displays). The CPU and the device controllers can execute in parallel, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory.

For a computer to start running—for instance, when it is powered up or rebooted—it needs to have an initial program to run. This initial program, or bootstrap program, tends to be simple. Typically, it is stored within the computer hardware in read-only memory (ROM) or electrically erasable programmable read-only memory (EEPROM), known by the general term firmware. It initializes all aspects of the system, from CPU registers to device controllers to memory contents. The bootstrap program must know how to load the operating system and how to start executing that system. To accomplish this goal, the bootstrap program must locate the operating-system kernel and load it into memory.

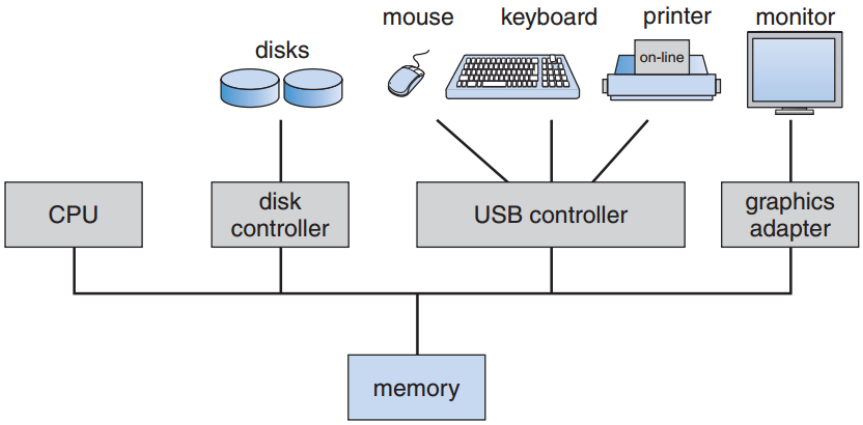


Figure 1.2. A modern computer system

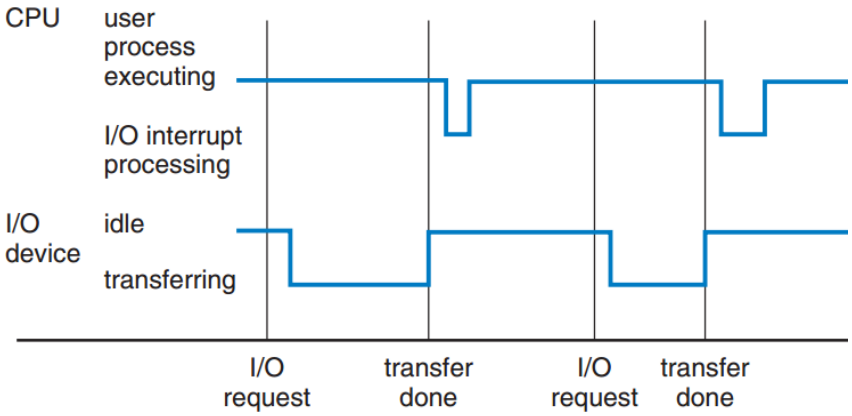


Figure 1.3. Interrupt timeline for a single process doing output

Once the kernel is loaded and executing, it can start providing services to the system and its users. Some services are provided outside of the kernel, by system programs that are loaded into memory at boot time to become system processes, or system daemons that run the entire time the kernel is running.

On UNIX, the first system process is “init,”and it starts many other daemons. Once this phase is complete, the system is fully booted, and the system waits for some event to occur. The occurrence of an event is usually signaled by an interrupt from either the hardware or the software. Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus. Software may trigger an interrupt by executing a special operation called a system call (also called amonitor call).

When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. The fixed location usually contains the starting address where the service routine for the interrupt is located. The interrupt service routine executes; on completion, the CPU resumes the interrupted computation. A timeline of

this operation is shown in Figure 1.3. Interrupts are an important part of a computer architecture. Each computer design has its own interrupt mechanism, but several functions are common.

The interrupt must transfer control to the appropriate interrupt service routine. The straightforward method for handling this transfer would be to invoke a generic routine to examine the interrupt information. The routine, in turn, would call the interrupt-specific handler. However, interrupts must be handled quickly. Since only a predefined number of interrupts is possible, a table of pointers to interrupt routines can be used instead to provide the necessary speed. The interrupt routine is called indirectly through the table, with no intermediate routine needed. Generally, the table of pointers is stored in low memory (the first hundred or so locations). These locations hold the addresses of the interrupt service routines for the various devices. This array, or interrupt vector, of addresses is then indexed by a unique device number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device. Operating systems as different as Windows and UNIX dispatch interrupts in this manner.

The interrupt architecture must also save the address of the interrupted instruction. Many old designs simply stored the interrupt address in a fixed location or in a location indexed by the device number. More recent architectures store the return address on the system stack. If the interrupt routine needs to modify the processor state—for instance, by modifying register values—it must explicitly save the current state and then restore that state before returning. After the interrupt is serviced, the saved return address is loaded into the program counter, and the interrupted computation resumes as though the interrupt had not occurred.

1.2.3 Storage Structure

The CPU can load instructions only from memory, so any programs to run must be stored there. General-purpose computers run most of their programs from rewritable memory, called main memory (also called random-access memory, or RAM). Main memory commonly is implemented in a semiconductor technology called dynamic random-access memory (DRAM).

A typical instruction–execution cycle, as executed on a system with a von Neumann architecture, first fetches an instruction from memory and stores that instruction in the instruction register. The instruction is then decoded and may cause operands to be fetched from memory and stored

in some internal register. After the instruction on the operands has been executed, the result may be stored back in memory.

Ideally, we want the programs and data to reside in main memory permanently. This arrangement is not usually possible for the following two reasons:

1. Main memory is usually too small to store all needed programs and data permanently.
2. Main memory is a volatile storage device that loses its contents when power is turned off.

Thus, most computer systems provide secondary storage as an extension of main memory. The most common secondary-storage device is a hard disk drive (HDD), which provides storage for both programs and data. Most programs (system and application) are stored on a disk until they are loaded into memory.

Many programs then use the disk as both the source and the destination of their processing. Hence, the proper management of disk storage is of central importance to a computer system.

The wide variety of storage systems can be organized in a hierarchy, according to speed and cost.

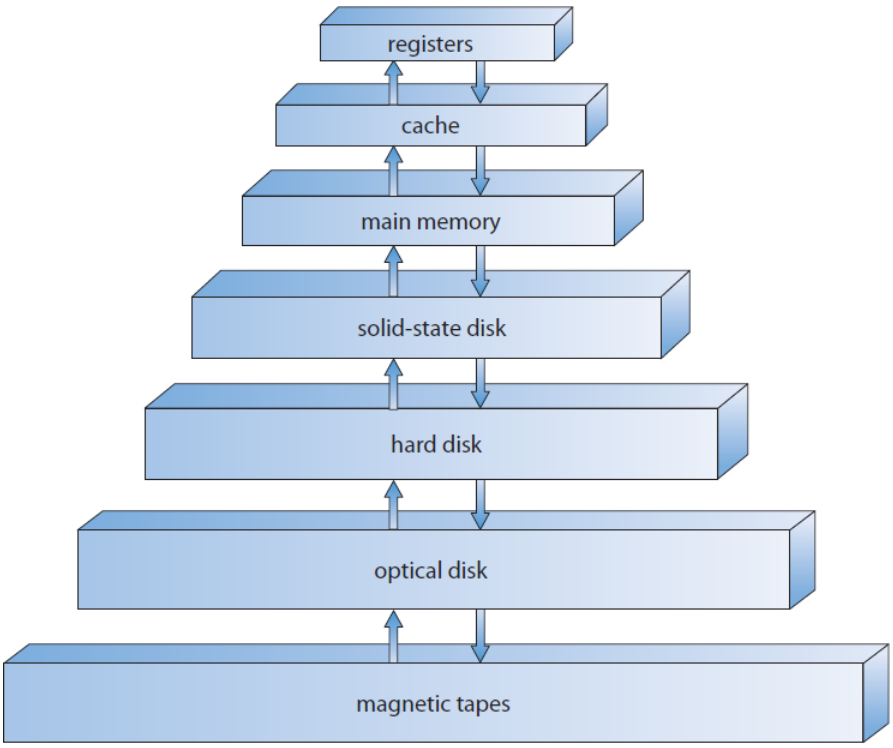


Figure 1.4. Storage-device hierarchy.

The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases. The top four levels of memory in the previous Figure 1.4 may be constructed using

semiconductor memory. The storage systems above the solid-state disk are volatile, whereas those including the solid-state disk and below are nonvolatile. **Solid-state disks** have several variants but in general are faster than hard disks and are nonvolatile.

The design of a complete memory system must balance all the factors just discussed: it must use only as much expensive memory as necessary while providing as much inexpensive, nonvolatile memory as possible. Caches can be installed to improve performance where a large disparity in access time or transfer rate exists between two components.

1.2.4 Input/Output (I/O) Structure

Storage is only one of many types of I/O devices within a computer. A large portion of operating system code is dedicated to managing I/O, both because of its importance to the reliability and performance of a system and because of the varying nature of the devices.

To start an I/O operation, the device driver loads the appropriate registers within the device controller. The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage.

1.3 Computer-System Architecture

A computer system can be organized in a number of different ways, which we can categorize roughly according to the number of general-purpose processors used.

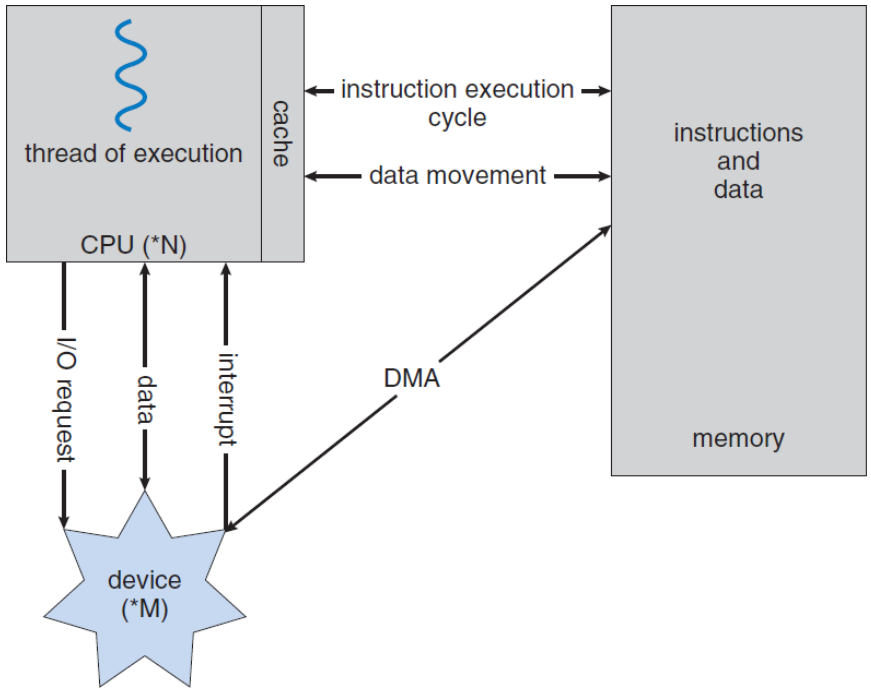


Figure 1.5. How a modern computer system works.

1.3.1 Single-Processor Systems

Until recently, most computer systems used a single processor. On a singleprocessor system, there is one main CPU capable of executing a general-purpose instruction set,

including instructions from user processes. PCs contain amicroprocessor in the keyboard to convert the keystrokes into codes to be sent to the CPU. In other systems or circumstances, special-purpose processors are low-level components built into the hardware. The operating system cannot communicate with these processors; they do their jobs autonomously. The use of special-purpose microprocessors is common and does not turn a single-processor system into a multiprocessor. If there is only one general-purpose CPU, then the system is a single-processor system.

1.3.2 Multiprocessor Systems

Within the past several years, multiprocessor systems (also known as parallel systems or multicore systems) have begun to dominate the landscape of computing. Such systems have two or more processors in close communication, sharing the computer bus and some times the clock, memory, and peripheral devices. Multiprocessor systems first appeared prominently in servers and have since migrated to desktop and laptop systems. Recently, multiple processors have appeared on mobile devices such as smartphones and tablet computers. Multiprocessor systems have three main advantages:

1. **Increased throughput**. By increasing the number of processors,we expect to getmore work done in less time. If the CPU has an integrated memory controller, then adding CPUs can also increase the amount of memory addressable in the system.
2. **Economy of scale**. Multiprocessor systems can cost less than equivalent multiple single-processor systems, because they can share peripherals, mass storage, and power supplies.
3. **Increased reliability**. If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down.

The ability to continue providing service proportional to the level of surviving hardware is called **graceful degradation**. Some systems go beyond graceful degradation and are called **fault tolerant**, because they can suffer a failure of any single component and still continue operation. Fault tolerance requires a mechanism to allow the failure to be detected, diagnosed, and, if possible, corrected.

The multiple-processor systems in use today are of two types. Some systems use **asymmetric multiprocessing**, in which, to each processor is assigned a specific task. A boss processor controls the system by scheduling and allocating work to the worker processors.

The most common systems use **symmetric multiprocessing (SMP)**, in which each processor performs all tasks, including operating system functions and user processes.

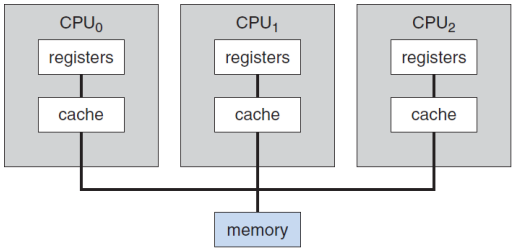


Figure 1.6. Symmetric multiprocessing architecture.

A recent trend in CPU design is to include multiple computing **cores** on a single chip. Such multiprocessor systems are termed **multicore**. They can be more efficient than multiple chips with single cores because on-chip communication is faster than between-chip communication. In addition, one chip with multiple cores uses significantly less power than multiple single-core chips.

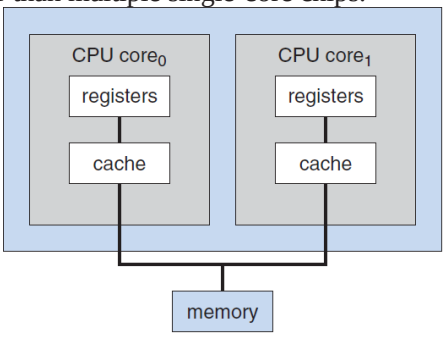


Figure 1.7. A dual-core design with two cores placed on the same chip.

1.3.3 Clustered Systems

A **clustered system** is a multiprocessor system which gathers together multiple CPUs. Clustered systems differ from the multiprocessor systems in that they are composed of two or more individual systems or nodes joined together. Each node may be a single processor system or a multicore system. Clustered
computers share storage and are closely linked via a local-area network LAN or a faster interconnect, such as InfiniBand.

Clustering is usually used to provide **high-availability** service—that is, service will continue even if one or more systems in the cluster fail. Generally, we obtain high availability by adding a level of redundancy in the system. For this purpose, clustering can be structured asymmetrically or symmetrically. In **asymmetric**
clustering, one machine is in **hot-standby mode** while the other is running the applications. In **symmetric clustering**, two or more hosts are running applications and are monitoring each other. This structure is obviously more efficient, as it uses all of the available hardware.

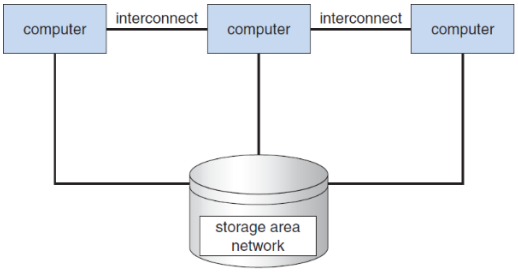


Figure 1.8 General structure of a clustered system.

1.4 Operating-System Structure

Now that we have discussed basic computer-system organization and architecture, we are ready to talk about operating systems. An operating system provides the

environment within which programs are executed. Internally, operating systems vary greatly in their makeup, since they are organized along many different lines.

One of the most important aspects of operating systems is the ability to multiprogram. **Multiprogramming** increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute.

The operating system keeps several jobs in memory simultaneously. Since, main memory is generally small to accommodate all jobs, the jobs are kept initially on the disk in the **job pool**. This pool consists of all processes residing on disk awaiting allocation of main memory.

Time sharing (or **multitasking**) is a logical extension of multiprogramming. In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running. Time sharing requires an **interactive** computer system, which provides direct communication between the user and the system. A time-shared operating system allows many users to share the computer simultaneously. As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to his use, even though it is being shared among many users.

1.5 Operating-System Operations

A **trap** (or an **exception**) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed.

1.5.1 Dual-Mode and Multimode Operation

We must be able to distinguish between the execution of operating-system code and user defined code.

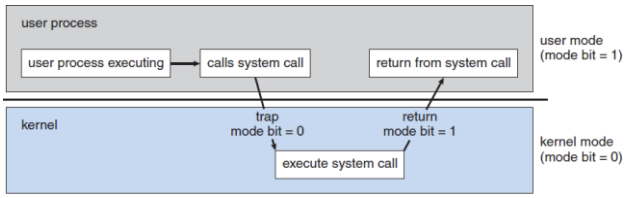


Figure 1.9. Transition from user to kernel mode.

At the very least, we need two separate ***modes*** of operation: **user mode** and **kernel mode** (also called **supervisor mode**, **system mode**, or **privileged mode**). A bit, called the **mode bit**, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode. Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode before passing control to a user program.

1.5.2 Timer

In order to avoid a user program to get stuck in an infinite loop or to fail to call system services and never return control to the operating system, one can use a **timer**. A timer can be set to interrupt the computer after a specified fix or variable period. The timer can also be used to prevent a user program from running too long.

1.6 Process Management

A process can be defined as a time-shared user program such as a compiler,a word-processing program being run by an individual user on a PC, a system task, such as sending output to a printer, etc. Globally, it is a job or a time-shared program.

The operating system is responsible for the following activities in connection with process management:

- Scheduling processes and threads on the CPUs,
- Creating and deleting both user and system processes,
- Suspending and resuming processes,
- Providing mechanisms for process synchronization,
- Providing mechanisms for process communication.

1.7 Memory Management

The operating system is responsible for the following activities in connection with memory management:

- Keeping track of which parts of memory are currently being used and who is using them
- Deciding which processes (or parts of processes) and data to move into and out of memory
- Allocating and deallocating memory space as needed

1.8 Storage Management

1.8.1 File-System Management

The operating system is responsible for the following activities in connection with file management:

- Creating and deleting files,
- Creating and deleting directories to organize files,
- Supporting primitives for manipulating files and directories,
- Mapping files onto secondary storage,
- Backing up files on stable (nonvolatile) storage media.

1.8.2 Mass-Storage Management

The operating system is responsible for the following activities in connection with disk management:

- Free-space management,
- Storage allocation,
- Disk scheduling.

1.8.3 Caching

Caching is an important principle of computer systems. When we need a particular piece of information, we first check whether it is in the cache. If it is, we use the information directly from the cache. If it is not, we use the information from the source, putting a copy in the cache under the assumption that we will need it again soon.

Main memory can be viewed as a fast cache for secondary storage, since data in secondary storage must be copied into main memory for use and data must be in main memory before being moved to secondary storage for safekeeping. The following figure present the performance of some levels of storage.

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	hard disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	hard disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Figure 1.10. Performance of various levels of storage.

1.9 Protection and Security

Protection, then, is any mechanism for controlling the access of processes or users to the resources defined by a computer system. This mechanism must provide means to specify the controls to be imposed and to enforce the controls.

Protection and security require the system to be able to distinguish among all its users. Most operating systems maintain a list of user names and associated **user identifiers (user IDs)**.

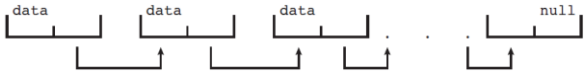
1.10 Kernel Data Structures

The way data are structured in the system is a topic central to operating-system implementation.

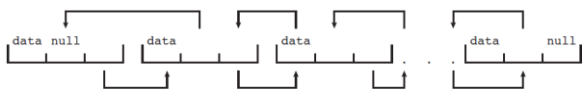
1.10.1 Lists, Stacks, and Queues

An array is a simple data structure in which each element can be accessed directly. For example, main memory is constructed as an array. Linked lists are of several types:

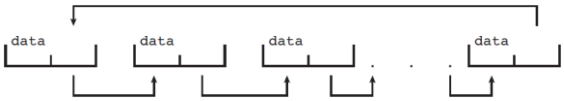
- In a ***singly linked list,*** each item points to its successor.



- In a ***doubly linked list,*** a given item can refer either to its predecessor or to its successor.



- In a ***circularly linked list,*** the last element in the list refers to the first element, rather than to null.



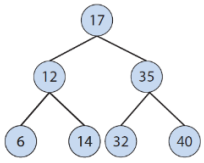
Linked lists accommodate items of varying sizes and allow easy insertion and deletion of items. Lists are sometimes used directly by kernel algorithms. Frequently, though, they are used for constructing more powerful data structures, such as stacks and queues.

A **stack** is a sequentially ordered data structure that uses the last in, first out **(LIFO)** principle for adding and removing items, meaning that the last item placed onto a stack is the first item removed. An operating system often uses a stack when invoking function calls. Parameters, local variables, and the return address are pushed onto the stack when a function is called; returning from the function call pops those items off the stack.

A **queue**, in contrast, is a sequentially ordered data structure that uses the first in, first out **(FIFO)** principle: items are removed from a queue in the order in which they were inserted. Queues are also quite common in operating systems. For example, jobs that are sent to a printer are typically printed in the order in which they were submitted. Also, tasks that are waiting to be run on an available CPU are often organized in queues.

1.10.2 Trees

values in a tree structure are linked through parent–child relationships. In a **general tree**, a parent may have an unlimited number of children. In a **binary search tree**, a parent may have at most two children termed the ***left child*** and the ***right child***, with the condition *left child* <= *right child*. Linux uses a balanced binary search tree as part its CPU-scheduling algorithm.



1.10.3 Hash Functions and Maps

A **hash function** takes data as its input, performs a numeric operation on this data, and returns a numeric value. This numeric value can then be used as an index into a table (typically an array) to quickly retrieve the data. Hash functions are used extensively in operating systems.

1.11 Computing Environments

1.11.1 Traditional Computing

As computing has matured, the lines separating many of the traditional computing environments have blurred. Time-sharing systems used a timer and scheduling algorithms to cycle processes rapidly through the CPU, giving each user a share of the resources. Today, traditional time-sharing systems are uncommon. The same scheduling technique is still in use on desktop computers, laptops, servers, and even mobile computers, but frequently all the processes are owned by the same user.

1.11.2 Mobile Computing

Mobile computing refers to computing on handheld smartphones and tablet computers. Two operating systems currently dominate mobile computing: **Apple iOS** and **Google Android**. iOS was designed to run on Apple iPhone and iPad mobile devices. Android powers smartphones and tablet computers available from many manufacturers.

1.11.3 Distributed Systems

A distributed system is a collection of physically separate, possibly heterogeneous, computer systems that are networked to provide users with access to the various resources that the system maintains. A **network**, in the simplest terms, is a communication path between two or more systems. Distributed systems depend on networking for their functionality. Networks vary by the protocols used, the distances between nodes, and the transport media.

A **network operating system** is an operating system that provides features such as file sharing across the network, along with a communication scheme that allows different processes on different computers to exchange messages. Adistributed operating system provides a less autonomous environment. The different computers communicate closely enough to provide the illusion that only a single operating system controls the network.

1.11.4 Client–Server Computing

Server systems can be broadly categorized as compute servers and file servers:

- The **compute-server system** provides an interface to which a client can send a request to perform an action (for example, read data). In response, the server executes the action and sends the results to the client. A server running a database that responds to client requests for data is an example of such a system.
- The **file-server system** provides a file-system interface where clients can create, update, read, and delete files. An example of such a system is a web server that delivers files to clients running web browsers.

1.11.5 Virtualization

Virtualization is a technology that allows operating systems to run as applications within other operating systems. Broadly speaking, virtualization is one member of a class of software that also includes emulation. **Emulation** is used when the source CPU type is different from the target CPU type.

For example, when Apple switched from the IBM Power CPU to the Intel x86 CPU for its desktop and laptop computers, it included an emulation facility called "Rosetta," which allowed applications compiled for the IBM CPU to run on the Intel CPU.

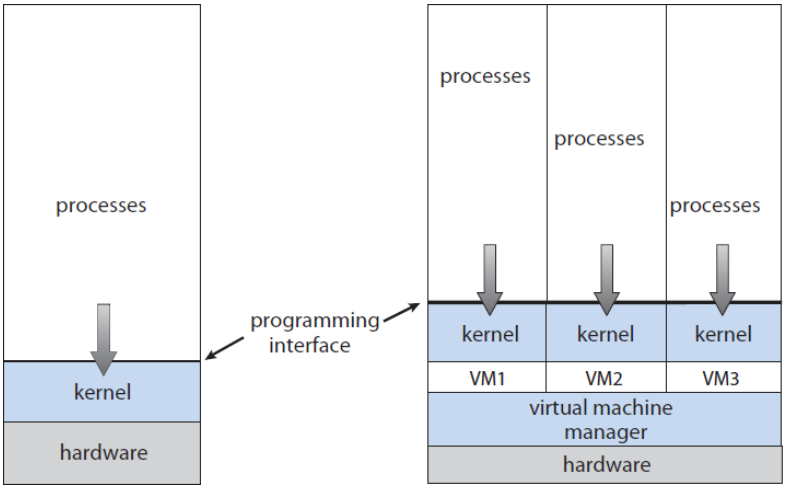


Figure1.11. VMware

1.11.6 Cloud Computing

Cloud computing is a type of computing that delivers computing, storage, and even applications as a service across a network. In some ways, it's a logical extension of virtualization, because it uses virtualization as a base for its functionality. There are actually many types of cloud computing, including the following:

- **Public cloud**—a cloud available via the Internet to anyone willing to pay for the services
- **Private cloud**—a cloud run by a company for that company's own use
- **Hybrid cloud**—a cloud that includes both public and private cloud components
- Software as a service **(SaaS)**—one or more applications (such as word processors or spreadsheets) available via the Internet
- Platform as a service **(PaaS)**—a software stack ready for application use via the Internet (for example, a database server)
- Infrastructure as a service **(IaaS)**—servers or storage available over the Internet (for example, storage available for making backup copies of production data).

Certainly, there are traditional operating systems within many of the types of cloud infrastructure. Beyond those are the VMMs that manage the virtual machines in which the user processes run. At a higher level, the VMMs themselves are managed by cloud management tools, such as Vware vCloud Director and the open-source Eucalyptus toolset. These tools manage the resourceswithin a given cloud and provide interfaces to the cloud components, making a good argument for considering them a new type of operating system. The following figure illustrates a public cloud providing IaaS.

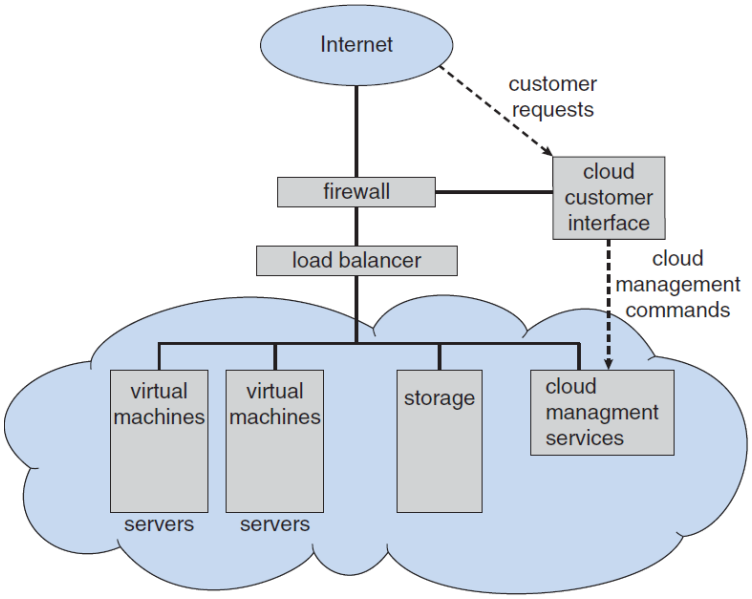


Figure 1.12. Cloud computing.

1.11.7 Real-Time Embedded Systems

Embedded computers are the most prevalent form of computers in existence. These devices are found everywhere, from car engines and manufacturing robots to DVDs and microwave ovens. These embedded systems vary considerably. Some are general-purpose computers, running standard operating systems. Others are hardware devices with a special-purpose embedded operating system providing just the functionality desired.Yet others are hardware deviceswith applicationspecific integrated circuits (**ASICs**) that perform their tasks without an operating system. Embedded systems almost always run **real-time operating systems**.

1.12 Open-Source Operating Systems

Open-source operating systems are those available in source-code format rather than as compiled binary code. Linux is the most famous opensource operating system, while Microsoft Windows is a well-known example of the opposite **closed-source** approach. Apple's Mac OS X and iOS operating systems comprise a hybrid approach. They contain an open-source kernel named Darwin yet include proprietary, closed-source components as well.

Starting with the source code allows the programmer to produce binary code that can be executed on a system. Learning operating systems by examining the source code has other benefits as well. With the source code in hand, a student can modify the operating system and then compile and run the code to try out those changes, which is an excellent learning tool. There are many benefits to open-source operating systems, including a community of interested (and usually unpaid) programmers who contribute to the code by helping to debug it, analyze it, provide support, and suggest changes. Arguably, open-source code is more secure than closed-source code becausemanymore eyes are viewing the code. Certainly, open-source code has bugs, but open-source advocates argue that

bugs tend to be found and fixed faster owing to the number of people using and viewing the code.

1.12.1 History

In the early days of modern computing (that is, the 1950s), a great deal of software was available in open-source format. Computer and software companies eventually sought to limit the use of their software to authorized computers and paying customers. Releasing only the binary files compiled from the source code, rather than the source code itself, helped them to achieve this goal, aswell as protecting their code and their ideas from their competitors. Operating systems and other programs can limit the ability to play back movies and music or display electronic books to authorized computers. Such **copy protection** or **digital rights management** **(DRM)** would not be effective if the source code that implemented these limits were published.

Richard Stallman in 1983 started the GNU project to create a free, open-source, UNIX compatible operating system. In 1985, he published the GNU Manifesto, which argues that all software should be free and open-sourced. He also formed the **Free Software Foundation** **(FSF)** with the goal of encouraging the free exchange of software source code and the free use of that software. Rather than copyright its software, the FSF "copylefts" the software to encourage sharing and improvement. The **GNU General Public License** **(GPL)** codifies copylefting and is a common license underwhich free software is released. Fundamentally, GPL requires that the source code be distributed with any binaries and that any changes made to the source code be released under the same GPL license.

1.12.2 Linux

As an example of an open-source operating system, consider **GNU/Linux**. The GNU project produced many UNIX-compatible tools, including compilers, editors, and utilities, but never released a kernel.

In 1991, a student in Finland, Linus Torvalds, released a rudimentary UNIX-like kernel using the GNU compilers and tools and invited contributions worldwide. The advent of the Internet meant that anyone interested could download the source code, modify it, and submit changes to Torvalds.

The resulting GNU/Linux operating system has spawned hundreds of unique **distributions**, or custom builds, of the system. Major distributions include Red Hat, SUSE, Fedora, Debian, Slackware, and Ubuntu. Distributions vary in function, utility, installed applications, hardware support, user interface, and purpose.

You can run Linux on a Windows system using the following simple, free approach:

1. Download and install the free "VMware Player" tool on your system, from the following URL: http://www.vmware.com/download/player/
2. Choose a Linux version from among the hundreds of "appliances," or virtual machine images, available from VMware at http://www.vmware.com/appliances/
3. Boot the virtual machine within VMware Player.

1.12.3 BSD UNIX

BSD UNIX started in 1978 as a derivative of AT&T's UNIX. It was not opensource because a license from AT&T was required. A fully functional, open-source version, 4.4 BSD-lite, was released in 1994.

Just as with Linux, there are many distributions of BSD UNIX, including FreeBSD, NetBSD, OpenBSD, and DragonflyBSD. To explore the source code of FreeBSD, simply download the virtual machine image of the version of interest and boot it within VMware, as described above for Linux.

The source code comes with the distribution and is stored in /usr/src/. The kernel source code is in /usr/src/sys. For example, to examine the virtual memory implementation code in the FreeBSD kernel, see the files in /usr/src/sys/vm.

Darwin, the core kernel component of Mac OS X, is based on BSD UNIX and is open-sourced as well. That source code is available from http://www.opensource.apple.com/. Every Mac OS X release has its opensource components posted at that site. The name of the package that contains the kernel begins with "xnu." Apple also provides extensive developer tools, documentation, and support at http://connect.apple.com.

1.12.4 Solaris

Solaris is the commercial UNIX-based operating system of Sun Microsystems. Originally, Sun's **SunOS** operating system was based on BSD UNIX. Sun moved to AT&T's System V UNIX as its base in 1991. In 2005, Sun open-sourced most of the Solaris code as the OpenSolaris project. The source code as it was in 2005 is still available via a source code browser and for download at http://src.opensolaris.org/source.

Several groups interested in using OpenSolaris have started from that base and expanded its features. Their working set is Project Illumos, which has expanded from the OpenSolaris base to include more features and to be the basis for several products. Illumos is available at http://wiki.illumos.org.

1.12.5 Open-Source Systems as Learning Tools

The free software movement is driving legions of programmers to create thousands of open-source projects, including operating systems. Sites like http://freshmeat.net/ and http://distrowatch.com/ provide portals to many of these projects. Open-source projects enable students to use source code as a learning tool. They can modify programs and test them, help find and fix bugs, and otherwise explore mature, full-featured operating systems, compilers, tools, user interfaces, and other types of programs. The availability of source code for historic projects, such as Multics, can help students to understand those projects and to build knowledge that will help in the implementation of new projects.

GNU/Linux and BSD UNIX are all open-source operating systems, but each has its own goals, utility, licensing, and purpose. The advantages of free software and open sourcing are likely to increase the number and quality of open-source projects, leading to an increase in the number of individuals and companies that use these projects.

1.13 Summary

An operating system is a software that manages the computer hardware, as well as providing an environment for application programs to run. Perhaps the most visible aspect of an operating system is the interface to the computer system it provides to the human user.

For a computer to do its job of executing programs, the programs must be in main memory. Main memory is the only large storage area that the processor can access directly. It is an array of bytes, ranging in size from millions to billions. Each byte in memory has its own address. The main memory is usually a volatile storage device that loses its contents when power is turned off or lost. Most computer systems provide secondary storage as an extension of main memory. Secondary storage provides a form of nonvolatile storage that is capable of holding large quantities of data permanently. The most common secondary-storage device is a hard disk, which provides storage of both programs and data.

Thewide variety of storage systems in a computer system can be organized in a hierarchy according to speed and cost. The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases.

There are several different strategies for designing a computer system. Single-processor systems have only one processor, while multiprocessor systems contain two or more processors that share physical memory and peripheral devices. The most common multiprocessor design is symmetric multiprocessing (or SMP), where all processors are considered peers and run independently of one another. Clustered systems are a specialized form of multiprocessor systems and consist of multiple computer systems connected by a local-area network.

To best utilize the CPU, modern operating systems employ multiprogramming, which allows several jobs to be in memory at the same time, thus ensuring that the CPU always has a job to execute. Time-sharing systems are an extension of multiprogramming where in CPU scheduling algorithms rapidly switch between jobs, thus providing the illusion that each job is running concurrently.

The operating system must ensure correct operation of the computer system. To prevent user programs from interfering with the proper operation of the system, the hardware has two modes: **user mode** and **kernel mode**. Various instructions (such as I/O instructions and halt instructions) are **privileged** and can be executed only in kernel mode. The memory in which the operating system resides must also be protected from modification by the user.

A timer prevents infinite loops. These facilities (dual mode, privileged instructions, memory protection, and timer interrupt) are basic building blocks used by operating systems to achieve correct operation.

A process (or job) is the fundamental unit of work in an operating system. Process management includes creating and deleting processes and providing mechanisms for processes to communicate and synchronize with each other.

An operating system manages memory by keeping track of what parts of memory are being used and by whom. The operating system is also responsible for dynamically allocating and freeing memory space. Storage space is also managed by the operating system; this includes providing file systems for representing files and directories and managing space on mass-storage devices.

Operating systems must also be concerned with protecting and securing the operating system and users. Protection measures control the access of processes or users to the resources made available by the computer system. Security measures are responsible for defending a computer system from external or internal attacks.

Several data structures that are fundamental to computer science are widely used in operating systems, including lists, stacks, queues, trees, hash functions, maps, and bitmaps.

Computing takes place in a variety of environments. Traditional computing involves desktop and laptop PCs, usually connected to a computer network. Mobile computing refers to computing on handheld smartphones and tablet computers, which offer several unique features. Distributed systems allow users to share resources on geographically dispersed hosts connected via a computer network. Services may be provided through either the client–server model or the peer-to-peer model. Virtualization involves abstracting a computer's hardware into several different execution environments. Cloud computing uses a distributed system to abstract services into a "cloud," where users may access the services from remote locations.

Real-time operating systems are designed for embedded environments, such as consumer devices, automobiles, and robotics.

The free software movement has created thousands of open-source projects, including operating systems. Because of these projects, students are able to use source code as a learning tool. They can modify programs and test them, help find and fix bugs, and otherwise explore mature, full-featured operating systems, compilers, tools, user interfaces, and other types of programs.

GNU/Linux and BSD UNIX are open-source operating systems. The advantages of free software and open sourcing are likely to increase the number and quality of open-source projects, leading to an increase in the number of individuals and companies that use these projects.

1.14 Practice Exercises

1.1 What are the three main purposes of an operating system?

Answer:

The three main puropses are:

- *To provide an environment for a computer user to execute programs on computer hardware in a convenient and efficient manner.*
- *To allocate the separate resources of the computer as needed to solve the problem given. The allocation process should be as fair and efficient as possible.*

- *As a control program it serves two major functions: (1) supervision of the execution of user programs to prevent errors and improper use of the computer, and (2) management of the operation and controlof I/O devices.*

1.2 We have stressed the need for an operating system to make efficient use of the computing hardware. When is it appropriate for the operating system to forsake this principle and to "waste" resources? Why is such a system not really wasteful?

Answer:

Single-user systems should maximize use of the system for the user. A GUI might "waste" CPU cycles, but it optimizes the user's interaction with the system.

1.3 What is the main difficulty that a programmer must overcome in writing an operating system for a real-time environment?

Answer:

Themain difficulty is keeping the operating system within the fixed time constraints of a real-time system. If the system does not complete a task in a certain time frame, it may cause a breakdown of the entire system it is running. Therefore when writing an operating system for a real-time system, the writer must be sure that his scheduling schemes don't allow response time to exceed the time constraint.

1.4 Keeping in mind the various definitions of *operating system*, consider whether the operating system should include applications such as web browsers and mail programs. Argue both that it should and that it should not, and support your answers.

Answer:

An argument in favor of including popular applications with the operating system is that if the application is embedded within the operating system, it is likely to be better able to take advantage of features in the kernel and therefore have performance advantages over an application that runs outside of the kernel. Arguments against embedding applications within the operating system typically dominate however: (1) the applications are applications - and not part of an operating system, (2) any performance benefits of running within the kernel are offset by security vulnerabilities, (3) it leads to a bloated operating system.

1.5 How does the distinction between kernel mode and usermode function as a rudimentary form of protection (security) system?

Answer:

The distinction between kernel mode and user mode provides a rudimentary form of protection in the following manner. Certain instructions could be executed only when the CPU is in kernel mode. Similarly, hardware devices could be accessed only when the program is executing in kernel mode. Control overwhen interrupts could be enabled or disabled is also possible only when the CPU is in kernel mode. Consequently, the CPU has very limited capability when executing in user mode, thereby enforcing protection of critical resources.

1.6 Which of the following instructions should be privileged?

a. Set value of timer.
b. Read the clock.
c. Clear memory.
d. Issue a trap instruction.
e. Turn off interrupts.
f. Modify entries in device-status table.

g. Switch from user to kernel mode.
h. Access I/O device.

Answer:

The following operations need to be privileged: Set value of timer, clear memory, turn off interrupts, modify entries in device-status table, access I/O device. The rest can be performed in user mode.

1.7 Some early computers protected the operating system by placing it in a memory partition that could not be modified by either the user job or the operating system itself. Describe two difficulties that you think could arise with such a scheme.

Answer:

The data required by the operating system (passwords, access controls, accounting information, and so on) would have to be stored in or passed through unprotected memory and thus be accessible to unauthorized users.

1.8 Some CPUs provide for more than two modes of operation. What are two possible uses of these multiple modes?

Answer:

Although most systems only distinguish between user and kernel modes, some CPUs have supported multiple modes. Multiple modes could be used to provide a finer-grained security policy. For example, rather than distinguishing between just user and kernel mode, you could distinguish between different types of user mode. Perhaps users belonging to the same group could execute each other's code. The machine would go into a specified mode when one of these users was running code. When the machine was in this mode, a member of the group could run code belonging to anyone else in the group. Another possibility would be to provide different distinctions within kernel code. For example, a specific mode could allow USB device drivers to run. This would mean that USB devices could be serviced without having to switch to kernel mode, thereby essentially allowing USB device drivers to run in a quasi-user/kernel mode.

1.9 Timers could be used to compute the current time. Provide a short description of how this could be accomplished.

Answer:

A program could use the following approach to compute the current time using timer interrupts. The program could set a timer for some time in the future and go to sleep.When it is awakened by the interrupt, it could update its local state, which it is using to keep track of the number of interrupts it has received thus far. It could then repeat this process of continually setting timer interrupts and updating its local state when the interrupts are actually raised.

1.10 Give two reasons why caches are useful.What problems do they solve? What problems do they cause? If a cache can be made as large as the device for which it is caching (for instance, a cache as large as a disk), why not make it that large and eliminate the device?

Answer:

Caches are useful when two or more components need to exchange data, and the components perform transfers at differing speeds. Caches solve the transfer problem by providing a buffer of intermediate speed between the components. If the fast device finds the data it needs in the cache, it need not wait for the slower device. The data in the cache must be kept consistent with the data in the components. If a component has a data value change, and the datum is also in

the cache, the cache must also be updated. This is especially a problem on multiprocessor systems where more than one process may be accessing a datum. A component may be eliminated by an equal-sized cache, but only if: (a) the cache and the component have equivalent state-saving capacity (that is, if the component retains its data when electricity is removed, the cache must retain data as well), and (b) the cache is affordable, because faster storage tends to be more expensive.

1.11 Distinguish between the client–server and peer-to-peer models of distributed systems.

Answer:

The client-server model firmly distinguishes the roles of the client and server. Under this model, the client requests services that are provided by the server. The peer-to-peer model doesn't have such strict roles. In fact, all nodes in the system are considered peers and thus may act as either clients or servers—or both. A node may request a service from another peer, or the node may in fact provide such a service to other peers in the system.

For example, let's consider a system of nodes that share cooking recipes. Under the client-server model, all recipes are stored with the server. If a clientwishes to access a recipe, it must request the recipe from the specified server.Using the peer-to-peer model, a peer node could ask other peer nodes for the specified recipe. The node (or perhaps nodes) with the requested recipe could provide it to the requesting node. Notice how each peer may act as both a client (it may request recipes) and as a server (it may provide recipes).

1.12 In a multiprogramming and time-sharing environment, several users share the system simultaneously. This situation can result in various security problems.

- What are two such problems?
- Can we ensure the same degree of security in a time-shared machine as in a dedicated machine?

Explain your answer.

1.13 The issue of resource utilization shows up in different forms in different types of operating systems. List what resources must be managed carefully in the following settings:

- Mainframe or minicomputer systems
- Workstations connected to servers
- Mobile computers

1.14 Under what circumstances would a user be better off using a timesharing system than a PC or a single-user workstation?

1.15 Describe the differences between symmetric and asymmetric multiprocessing. What are three advantages and one disadvantage of multiprocessor systems?

1.16 How do clustered systems differ from multiprocessor systems? What is required for two machines belonging to a cluster to cooperate to provide a highly available service?

1.17 Consider a computing cluster consisting of two nodes running a database. Describe two ways in which the cluster software can manage access to the data on the disk. Discuss the benefits and disadvantages of each.

1.18 How are network computers different from traditional personal computers? Describe some usage scenarios in which it is advantageous to use network computers.

1.19 What is the purpose of interrupts? How does an interrupt differ from a trap? Can traps be generated intentionally by a user program? If so, for what purpose?

1.20 Direct memory access is used for high-speed I/O devices in order to avoid increasing the CPU's execution load.

- How does the CPU interface with the device to coordinate the transfer?
- How does the CPU know when the memory operations are complete?
- The CPU is allowed to execute other programs while the DMA controller is transferring data. Does this process interfere with the execution of the user programs? If so, describe what forms of interference are caused.

1.21 Some computer systems do not provide a privileged mode of operation in hardware. Is it possible to construct a secure operating system for these computer systems? Give arguments both that it is and that it is not possible.

1.22 Many SMP systems have different levels of caches; one level is local to each processing core, and another level is shared among all processing cores. Why are caching systems designed this way?

1.23 Consider an SMP system similar to the one shown in Figure 1.6. Illustrate with an example how data residing in memory could in fact have a different value in each of the local caches.

1.24 Discuss, with examples, how the problem of maintaining coherence of cached data manifests itself in the following processing environments:

a. Single-processor systems
b. Multiprocessor systems
c. Distributed systems

1.25 Describe a mechanism for enforcing memory protection in order to prevent a program from modifying the memory associated with other programs.

1.26 Which network configuration—LAN or WAN—would best suit the following environments?

a. A campus student union
b. Several campus locations across a statewide university system
c. A neighborhood

1.27 Describe some of the challenges of designing operating systems for mobile devices compared with designing operating systems for traditional PCs.

1.28 What are some advantages of peer-to-peer systems over client-server systems?

1.29 Describe some distributed applications that would be appropriate for a peer-to-peer system.

1.30 Identify several advantages and several disadvantages of open-source operating systems. Include the types of people who would find each aspect to be an advantage or a disadvantage.

CHAPTER 2. OPERATING –SYSTEM STRUCTURES

2.1 Introduction

An operating system provides the environment within which programs are executed. Internally, operating systems vary greatly in their makeup, since they are organized along many different lines. The design of a new operating system is a major task. It is important that the goals of the system be well defined before the design begins. These goals form the basis for choices among various algorithms and strategies.

We can view an operating system from several vantage points. One view focuses on the services that the system provides; another, on the interface that it makes available to users and programmers; a third, on its components and their interconnections. In this chapter, we explore all three aspects of operating systems, showing the viewpoints of users, programmers, and operating system designers. We consider what services an operating system provides, how they are provided, how they are debugged, and what the various methodologies are for designing such systems. Finally, we describe how operating systems are created and how a computer starts its operating system. A the end of the chapter, each student must be able to:

- describe the services an operating system provides to users, processes, and other systems.
- discuss the various ways of structuring an operating system.
- explain how operating systems are installed and customized and how they boot.

2.2 Operating-System Services

An operating system provides an environment for the execution of programs. It provides certain services to programs and to the users of those programs. The specific services provided, of course, differ from one operating system to another, but we can identify common classes. These operating system services are provided for the convenience of theprogrammer, tomake theprogramming task easier. The following Figure 2.1 shows one view of the various operating-system services and how they interrelate.

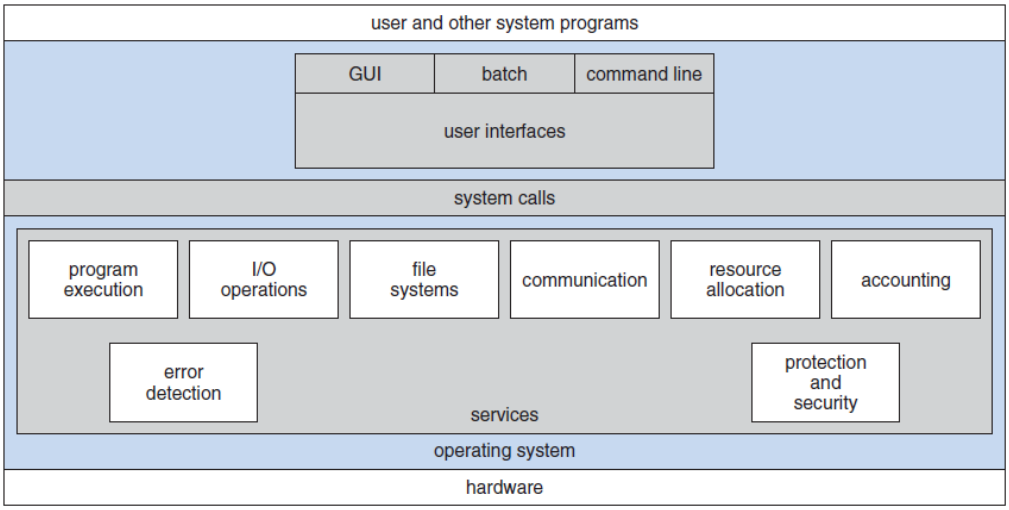


Figure 2.1 A view of operating system services.

One set of operating system services provides functions that are helpful to the user.

❖ **User interface**. Almost all operating systems have a **user interface (UI)**. This interface can take several forms. One is a **command-line interface (CLI)**, which uses text commands and a method for entering them. Another is a **batch interface**, in which commands and control directives are entered into files that are executed. Most commonly, a **graphical user interface (GUI)** is used. Some systems provide two or all three of these variations.

❖ **Program execution**. The system must be able to run a program loaded into memory. The program must be able to end its execution, either normally or abnormally (indicating error).

❖ **I/O operations**. A running program may require I/O, which may involve a file or an I/O device. For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.

❖ **File-system manipulation**. The file system is of particular interest. Obviously, programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information. Many operating systems provide a variety of file systems.

❖ **Communications**. There are many circumstances in which one process needs to exchange information with another process. Communications may be implemented via **shared memory**, in which two or more processes read and write to a shared section of memory, or **message passing**, in which packets of information in predefined formats are moved between processes by the operating system.

❖ **Error detection**. The operating system needs to be detecting and correcting errors constantly. Errors may occur in the CPU and memory hardware, in I/O devices, and in the user program.

❖ **Resource allocation**. When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. The operating system manages many different types of resources, including CPU cycles, main memory, and file storage.

❖ **Accounting**. We want to keep track of which users use how much and what kinds of computer resources. This record keeping may be used for accounting or simply for accumulating usage statistics.

❖ **Protection and security**. The owners of information stored in a multiuser or networked computer system may want to control the use of their information. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important. Such security starts with requiring each user to authenticate.

2.3 User and Operating-System Interface

In tyhis section, we discuss two fundamental approaches. One provides a command-line interface, or **command interpreter**, that allows users to directly enter commands to be performed by the operating system. The other allows users to interface with the operating system via a **graphical user interface, or GUI**.

2.3.1 Command Interpreters

The main function of the command interpreter is to get and execute the next user-specified command. Many of the commands given at this level manipulate files: create, delete, list, print, copy, execute, etc. The MS-DOS and UNIX shells operate in this way. These commands can be implemented in two general ways:

- **Direct interpretation approach:** the command interpreter it self contains the code to execute the command. For example, a command to delete a file may cause the command interpreter to jump to a section of its code that sets up the parameters and makes the appropriate system call.
- **Indirect interpretation approach:** the command interpreter does not understand the command in any way; it merely uses the command to identify a file to be loaded into memory and executed. Thus, the UNIX command to delete a file rm file.txt would search for a file called rm, load the file into memory, and execute it with the parameter file.txt. In this way, programmers can add new commands to the system easily by creating new files with the proper names. The command-interpreter program, which can be small, does not have to be changed for new commands to be added.

2.3.2 Graphical User Interfaces

Rather than entering commands directly via a command-line interface, users employ a mouse-based window and menu system characterized by a desktop metaphor.

2.3.3 Choice of Interface

The choice of whether to use a command-line or GUI interface is mostly one of personal preference. A few group of users, notably **System administrators** who manage computers and **power users** who have deep knowledge of a system frequently use the command-line interface. Other users prefer the GUI interface.

2.4 System Calls

System calls provide an interface to the servicesmade available by an operating system. These calls are generally available as routines written in C and C++, although certain low-level tasks *(for example, tasks where hardware must be accessed directly)* may have to be written using assembly-language instructions. Frequently, systems execute thousands of system calls per second.

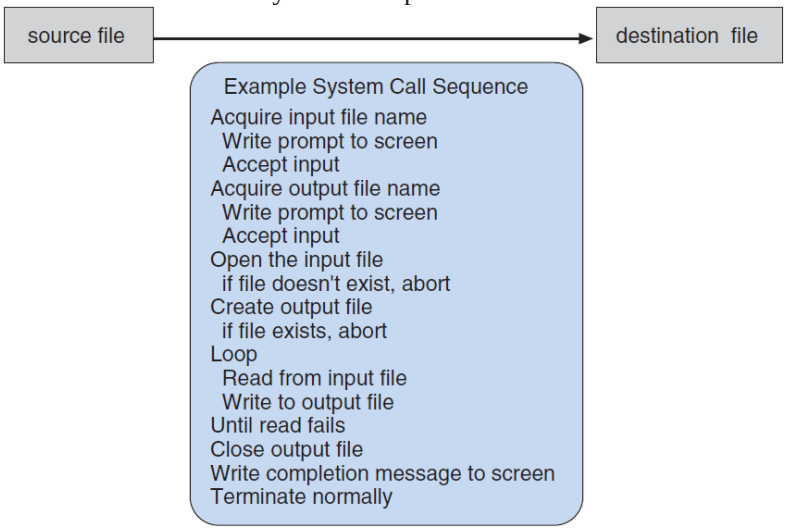


Figure 2.2 Example of how system calls are used.

EXAMPLE OF STANDARD API

As an example of a standard API, consider the *read()* function that is available in UNIX and Linux systems. The API for this function is obtained from the *man* page by invoking the command *man read* on the command line. A description of this API is the following:

```
#include <unistd.h>

ssize_t     read(int fd, void *buf, size_t count)
```

return value — function name — parameters

A program that uses the *read()* function must include the *unistd.h* header file, as this file defines the *ssize t* and *size t* data types (among other things). The parameters passed to *read()* are as follows:

- int fd—the file descriptor to be read
- void *buf—a buffer where the data will be read into
- size t count—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, *read()* returns −1.

For most programming languages, the run-time support system (*a set of functions built into libraries included with a compiler*) provides a **system call interface** that serves as the link to system calls made available by the operating system. The system-call interface intercepts function calls in the API and invokes the necessary system calls within the operating system.

The following figure illustrates the relationship between an API, the system-call interface and the operating system. It shows how the operating system handles a user application invoking the *open()* system call.

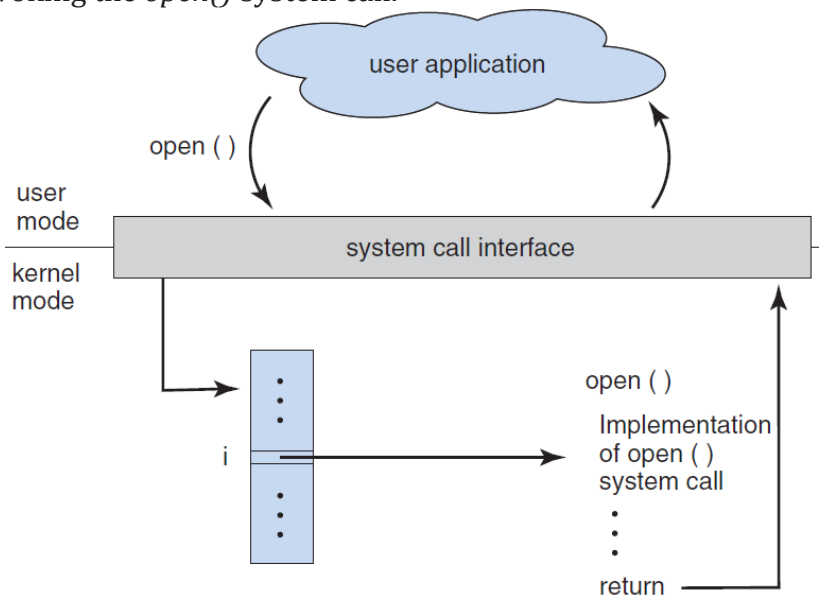


Figure 2.3 The handling of a user application invoking the open() system call.

Three general methods are used to pass parameters to the operating system: registers, block or table in memory, and stack. Some operating systems prefer the block or stack

method because those approaches do not limit the number or length of parameters being passed.

2.5 Types of System Calls

System calls can be grouped roughly into six major categories: **process control**, **file manipulation**, **device manipulation**, **information maintenance**, **communications**, and **protection**.

2.5.1 Process control

A running program needs to be able to halt its execution either ***normally (end())*** or ***abnormally (abort())***.Undereither normal or abnormal circumstances, the operating system must transfer control to the invoking command interpreter. Following are some system calls related to process control.

- end, abort
- load, execute
- create process, terminate process
- get process attributes, set process attributes
- wait for time
- wait event, signal event
- allocate and free memory

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()

2.5.2 File management

There are several common system calls dealing with files. The basic system calls are create() and delete() files. Once the file is created, one can open() it, read(), write(), or reposition(), and finally close() the file, indicating that it is no longer used. Following are some system calls related to File management.

- create file, delete file
- open, close
- read, write, reposition
- get file attributes, set file attributes

	Windows	Unix
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()

2.5.3 Device Management

A process may need several resources to execute—main memory, disk drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user process. Otherwise, the process will have to wait until sufficient resources are available. Following are system calls related to Device management

- request device, release device
- read, write, reposition
- get device attributes, set device attributes

- logically attach or detach devices

	Windows	Unix
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()

2.5.4 Information Maintenance

Many system calls exist simply for the purpose of transferring information between the user program and the operating system. For example, most systems have a system call to return the current *time()* and *date()*. Other system calls may return information about the system, such as the number of current users, the version number of the operating system, the amount of free memory or disk space, etc. the operating system keeps information about all its processes, and system calls are used to access this information. Generally, calls are also used to reset the process information (*get process attributes()* and *set process attributes()*). Following are system calls related to Information maintenance.

- get time or date, set time or date
- get system data, set system data
- get process, file, or device attributes
- set process, file, or device attributes

	Windows	Unix
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()

2.5.5 Communication

There are two common models of interprocess communication: the messagepassing model and the shared-memory model.

In the **message-passing model**, the communicating processes exchange messages with one another to transfer information.

In the **shared-memory model**, processes use shared memory create() and shared memory attach() system calls to create and gain access to regions of memory owned by other processes. Following are system calls related to Communications.

- create, delete communication connection
- send, receive messages
- transfer status information
- attach or detach remote devices

	Windows	Unix
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shm_open()
	MapViewOfFile()	mmap()

2.5.6 Protection

Protection provides a mechanism for controlling access to the resources provided by a computer system. Typically, system calls providing protection include set permission() and get permission(), which manipulate the permission settings of resources such as

files and disks. The allow user() and deny user() system calls specify whether particular users can—or cannot—be allowed access to certain resources. Following are system calls related to Protection.

- Deny permission to a resource;
- Give permission to a resource (file, disk).

	Windows	Unix
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

2.6 System Programs

System programs, also known as **system utilities**, provide a convenient environment for program development and execution.
Some of them are simply user interfaces to system calls. Others are considerably more complex. They can be divided into these categories:

- **File management**. These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.
- **Status information**. Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information.
- **File modification**. Several text editors may be available to create and modify the content of files stored on disk or other storage devices.
- **Programming-language support**. Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, and PERL) are often provided with the operating system or available as a separate download.
- **Program loading and execution**. Once a program is assembled or compiled, it must be loaded into memory to be executed.
- **Communications**. These programs provide the mechanism for creating virtual connections among processes, users, and computer systems.
- **Background services**. Some of processes lunched at boot time terminate after completing their tasks, while others, called **services**, **subsystems**, or **daemons** continue to run until the system is halted.

2.7 Operating-System Design and Implementation

In this section, we discuss some problems faced in designing and implementing an operating system. There are, of course, no complete solutions to such problems, but there are approaches that have proved successful

2.7.1 Design Goals

The first problem in designing a system is to define goals and specifications. At the highest level, the design of the system will be affected by the choice of hardware and the type of system: batch, time sharing, single user, multiuser, distributed, real time, or general purpose. Beyond this highest design level, the requirements may be much harder to specify. The requirements can, however, be divided into two basic groups: **user goals** and **system goals**.

Users want the system to be convenient to use, easy to learn and to use, reliable, safe, and fast. Those who design, create, maintain, and operate the system want the system to be easy to design, implement, and maintain; and it should be flexible, reliable, error free, and efficient.

2.7.2 Implementation

Once an operating system is designed, it must be implemented. Because operating systems are collections of many programs, written by many people over a long period of time, it is difficult to make general statements about how they are implemented.

Early operating systemswerewritten in assembly language.Now, although some operating systems are still written in assembly language, most are written in a higher-level language such as C or an even higher-level language such as C++. Actually, an operating system can be written in more than one language. The lowest levels of the kernel might be assembly language. Higher-level routines might be in C, and system programs might be in C or C++, in interpreted scripting languages like PERL or Python, or in shell scripts. In fact, a given Linux distribution probably includes programs written in all of those languages.

The Linux and Windows operating system kernels are written mostly in C, although there are some small sections of assembly code for device drivers and for saving and restoring the state of registers. The advantages of using a higher-level language, or at least a systems implementation language, for implementing operating systems are the same as those gained when the language is used for application programs: the code can be written faster, is more compact, and is easier to understand and debug.

In addition, improvements in compiler technology will improve the generated code for the entire operating system by simple recompilation. Finally, an operating system is far easier to **port**—*to move to some other hardware*— if it is written in a higher-level language. For example, MS-DOS was written in Intel 8088 assembly language. Consequently, it runs natively only on the Intel x86 family of CPUs. (Note that although MS-DOS runs natively only on Intel x86, emulators of the x86 instruction set allow the operating system to run on other CPUs—but more slowly, and with higher resource use).

The Linux operating system, in contrast, is written mostly in C and is available natively on a number of different CPUs, including Intel x86, Oracle SPARC, and IBM PowerPC. Although operating systems are large, only a small amount of the code is critical to high performance: the interrupt handler, I/O manager, memory manager, and CPU scheduler are probably the most critical routines. After the system is written and is working correctly, bottleneck routines can be identified and can be replaced with assembly-language equivalents.

2.8 Operating-System Structure

2.8.1 Simple Structure

Many operating systems do not have well-defined structures. Frequently, such systems started as small, simple, and limited. MS-DOS is an example of such a system. It was

written to provide the most functionality in the least space, so it was not carefully divided into modules. Figure 2.11 shows its
structure.
In MS-DOS, the interfaces and levels of functionality are not well separated. For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail. Of course, MS-DOS was also limited by the hardware of its era. Because the Intel 8088 for which it was written provides no dual mode and no hardware protection, the designers of MS-DOS had no choice but to leave the base hardware accessible.
Another example of limited structuring is the original UNIX operating system. Like MS-DOS, UNIX initially was limited by hardware functionality. It consists of two separable parts: the kernel and the system programs. The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved.We can view the traditional UNIX operating system as being layered to some extent, as shown in Figure 2.12. Everything below the system-call interface and above the physical hardware is the kernel. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls. Taken in sum, that is an enormous amount of functionality to be combined into one level. This monolithic structure was difficult to implement and maintain. It had a distinct performance advantage, however: there is very little overhead in the system call interface or in communication within the kernel. We still see evidence of this simple, monolithic structure in the UNIX, Linux, and Windows operating systems.