CHAPTER 1

Introduction

1.1 THE ROLE OF OPERATING SYSTEMS

We begin by examining the gap between the requirements and expectations placed on computer systems by the user community and the low-level capabilities of existing hardware. This gap is bridged by the operating system (OS) and other utility and support programs. We then outline the overall organization of OS, including interfaces to the hardware, the application programs, and the user. The remainder of the chapter traces the evolution of key OS concepts in the context of changing technology and the increasing diversity and sophistication of the user community.

1.1.1 Bridging the Hardware/Application Gap

Most computer systems today are based on the principles of a “stored-program computer” formulated by mathematician John von Neumann and others in the late 1940s. The basic components of a computer and their interconnections are shown schematically in Figure 1-1 in the form of a high-level block diagram. At the heart of this system is the computational engine consisting of a central processing unit (CPU) and executable main memory. The memory is a linear sequence of directly addressable cells; it holds programs (lists of executable machine instructions) and data. The CPU continuously repeats the following basic hardware cycle:

- Fetch the instruction pointed to by a special register called the program counter.
- Increment the program counter.
- Decode the current instruction, held in a special instruction register, to determine what must be done.
- Fetch any operands referenced by the instruction.
- Execute the instruction.

This cycle forms the basis of all computations on present-day computers.

For this computational scheme to be of any practical value, two fundamental components must be included. The first is a set of communication devices to allow data and commands to be exchanged between the user and the machine or between one computer
and another. It consists of input/output (I/O) devices (e.g., a keyboard and display terminal) and network interface devices. The second component is secondary storage to hold programs and data that are not currently loaded in main memory or that are only partially or temporarily loaded. This storage is needed because the system’s main memory is volatile and thus loses its contents when power is turned off, and because it is also much smaller in size than secondary storage.

Sometimes the distinction between communication and storage devices is clear cut. For example, a CD-ROM drive is strictly an input device, whereas a hard disk is clearly a storage device. However, there are also common cases where such a distinction cannot be made easily. For example, a removable diskette can be viewed as storage, but it also can be used as an I/O device when moving information between different systems. From an operating system’s perspective, CD-ROM, hard disk, diskette, and other devices are similar in nature, and many of the same techniques are employed to service them. We will refer to secondary storage and communication devices jointly as I/O devices.

Another degree of complexity is added when the computer system consists of more than one CPU. This can take several different forms, depending on the sharing level of the system’s hardware components. Figure 1-2 shows three possible architectures that extend the basic single-CPU architecture of Figure 1-1 in different ways. In the first case (Fig. 1-2a), the two CPUs share a common main memory. The secondary storage and communication devices are typically shared. The presence of multiple CPUs poses new challenges for the OS. One of these is caching. If each CPU maintains its own local memory cache, the system must ensure that two caches do not contain a different values for the same memory element. With a shared memory, this problem, referred to as cache coherence, is handled by the hardware and is transparent to the OS. Another important problem is the scheduling of processes. With a single CPU, scheduling is a matter of controlling the order in which processes execute. With multiple CPUs, the OS (or the application) also must decide on which CPU a given task should run. Synchronization and communication among processes running on different CPUs is performed through the shared memory; the approaches are similar to those for coordinating processes on a single CPU.

Figure 1-2b shows an architecture where each CPU has its own main memory. The secondary storage and other devices could still be shared. However, the communication subsystem must include a interconnection network that allows the CPUs to interact with each other, since no shared memory is available. There is a broad range of interconnection networks, ranging from a simple shared bus to dedicated connections arranged in a
Section 1.1 The Role of Operating Systems

variety of topologies. In the absence of shared memory, both scheduling and process coordination become more complicated. Scheduling involves not only the assignment of processes to different CPUs but also the assignment of data, some of which may be needed by multiple processes, to the disjoint local memory modules. Given that such
Chapter 1 Introduction

architectures are intended primarily for high-performance scientific computing, many scheduling issues are delegated to the applications rather than handled transparently by the OS. However, the system must provide the primitives necessary to support process synchronization and communication. These are based on messages sent between the different CPUs through the interconnection network. Depending on the sophistication of the network, the system also may have to be involved in solving the cache coherence problem.

The architectures of both Figure 1-2a and b are generally referred to as multi-processors. They provide multiple CPUs and possibly multiple main memory modules, but other hardware components are shared. Thus the system can still be viewed as a single computer. Figure 1-2c shows a different type of architecture, generally referred to as a multicomputer, where each CPU has not only its own local memory but also its own set of storage and communication devices. Among the communication devices are special network controllers, which allow each computer to send and receive data to/from other computers through the network. Thus, a multicomputer is viewed as a collection of complete autonomous computers, interacting with each other via the network.

There are many network types, distinguished by their size, topology, speed of data transmission, reliability, and other attributes. One popular way for two computers to communicate is through already existing telephone lines. These lines have been designed to carry analog voice data. To transmit digital data on a telephone line, each of the communicating computers uses a modem— a device designed to transform digital data into analog at the sender site, and analog back into digital at the receiver site. The modem is connected to an I/O bus just like any other device controller.

Faster and more reliable connections can be achieved with communication lines that are specifically designed to interconnect two or more computers. For a small number of computers, collocated in the same room or building, a local area network (LAN) is typically used. Figure 1-3 illustrates the two most common topologies of LANs: the ring network and the broadcast bus. The ring connects each computer to two neighbors. Data is passed along the ring in one direction from the sender to the receiver via the intermediate nodes. The broadcast bus is similar to the internal system buses. The most common type is the Ethernet bus. Each computer connects to the bus via a specialized network controller, an Ethernet card. All such controllers “listen to the bus,” i.e., monitor it for any data placed on it. A computer wishing to send a message to another computer places the message on the bus, preceded by the receiver’s address. All nodes listening to the bus examine the address, and the one whose address it matches reads the body

![Figure 1-3. LAN network topologies. (a) ring network (b) broadcast bus.](image-url)
Section 1.1 The Role of Operating Systems

of the message. The controllers also implement a protocol for resolving collisions—the simultaneous placement of data on the bus by two or more computers.

Local area networks are limited by the number of computers they can support and their physical distance. To interconnect large numbers of computers over long distances, a wide area network (WAN) is used. This employs a variety of hardware devices that repeat and amplify the signals as they travel across long distances. With WANs, it is no longer practical to establish point-to-point communication channels among all communicating computers. Instead, a message is broken into multiple packets, each of which is routed through switches and nodes of the WAN, and reassembled at the destination. This requires sophisticated and complex communication protocols that govern all aspects of the transmissions, so that large numbers of heterogeneous computers and local area subnetworks can cooperate and compete. The ultimate example of a very large heterogeneous network is the Internet, which interconnected millions of computers worldwide.

Unfortunately, there is still a long way to go from the architectural schemes depicted in Figures 1-1 and 1-2 to a practical computer system. The main problem is that the hardware operates at a very low level of abstraction relative to the problems that must be solved at the user level. Consequently, a significant amount of support software must be provided before a user application is able to take advantage of the machine’s hardware. Let us consider the individual components in turn:

- The machine instructions executed by a CPU perform only very simple tasks: read or write a value from/to a register or a main memory location; perform an arithmetic or logical operation on binary numbers; compare two bit-strings; or set the program counter to alter the sequence of instruction execution. The programmer developing an application, on the other hand, thinks in terms of high-level data structures (e.g., arrays) and the corresponding operations to manipulate these (e.g., matrix multiplication or Fast Fourier Transform). The resulting semantic gap must be bridged by software.

- The main memory is a linear sequence of simple cells that hold executable programs and their data. The programmer, in contrast, manages a diverse collection of entities of various types, sizes, or shapes, including executable programs, modules, procedures, or functions; and data structures such as arrays, trees, lists, stacks, or queues, each accessed by different operations. The semantic gap between the programmer’s heterogeneous collections of entities, some of which may be created and destroyed at runtime, and their machine representations in the simple sequential hardware main memory, must be bridged by software.

- The most common secondary storage devices are magnetic disks. These are two-dimensional structures that organize data in concentric rings, called tracks, each consisting of multiple blocks. To access a block of data requires a physical movement of the read/write head (a seek operation), waiting for the desired block to pass under the read/write head, and finally transferring the block to/from memory. Programmers who use the disk for storage of programs or data do not care about the disk’s internal organization and operation. They wish to treat an entire program or data collection as a single abstract entity—a file—that can be created, destroyed, copied, and transferred between different locations using only a small set of high-level commands. These high-level abstractions must be created by software to close this semantic divide.
Chapter 1 Introduction

- The hardware interface to communication and I/O devices generally consists of registers that must be written and read using specific instruction sequences to cause the device to perform its functions. These sequences tend to be very complex, requiring a deep understanding of the specific device’s operation. In contrast, most programmers would like to have simple, uniform interfaces and high-level operations that do not require a detailed understanding of the devices. Similarly, users would like to send messages to processes on other computers without worrying about the underlying network topologies, transmission errors, or network congestion problems. This semantic gap must again be bridged by software in the form of specialized device drivers and communication protocols.

In summary, it is the responsibility of the system’s software to reduce the large divide between the capabilities of a computer’s hardware components and the needs and expectations of programmers. Assemblers, compilers, and interpreters address the problem of the low-level instruction set. They permit programmers to develop code and data structures using high-level languages, which are then automatically translated into equivalent programs in the low-level machine language. Other system software, such as linkers, editors, or debuggers, contribute indirectly to reducing the semantic gap by facilitating the program development process and providing various support functions.

The remaining gaps between the hardware’s capabilities and the programmers’ needs are bridged by the OS.

1.1.2 Three Views of Operating Systems

Three key concepts permeate the design and organization of an OS. Each of these results in a different view of an OS and its role:

1. **Abstraction** is applied to handle complexity. It allows us to hide the complexity and low-level features of the hardware by building a layer of software on top of it. Abstraction also is used to define and construct multiple layers of software, each hiding the complexity of the level below.

2. **Virtualization** helps us to support sharing of computer resources among different users or applications. It also permits resource sharing within a single application or user. Virtualization allows us to create the illusion of multiple dedicated resources that do not need to be shared.

3. **Resource management** addresses the problem of efficiency. It aims at maximizing overall performance of the system while addressing the needs of individual users and their applications.

These three ideas give rise to three views of the nature of an OS: extended machines, virtual machines, and resource managers.

**Operating Systems as Extended Machines**

Imagine for a moment using a computer system without any I/O support. To read a data item from disk, the programmer must first face the challenging task of understanding the disk’s low-level interface. Instead of issuing a single read operation to get the desired data block, he or she must write dozens of low-level instructions that move the read/write
Section 1.1 The Role of Operating Systems

head into position, supervise the transfer of the appropriate data block, and handle a mul-
titude of possible errors. The problem is aggravated further because no widely accepted
standards for the operation of I/O devices exist; different sequences of instructions are
usually required not only for each major device class (e.g., laser printers) but also for
each individual device model within that class.

To liberate the programmer from the low-level details and complexity of I/O
processing, the OS applies the concept of abstraction—one of the main techniques in
computer science that handle complexity. The basic idea is to package sequences of
lower-level instructions into functions that can be invoked as single higher-level opera-
tions. The collection of such functions extends the underlying instruction set, thus creating
the illusion of a more powerful extended machine.

Operating systems make extensive use of abstraction at many levels. Generally, all
I/O operations (e.g., reading or writing a block of data) are implemented as functions
provided by the OS. Figure 1-4 illustrates the abstraction idea with a specific example. It
shows a high-level read operation that specifies the file to be accessed (f1), the logical
block number within the file (lblk), and the main memory offset at which the data block
should be deposited (mm). The underlying I/O routines implement the high-level read by
performing the necessary lower-level operations, such as moving the read/write arm to
the appropriate position on the disk and checking for possible errors. These lower-level
operations are, in turn, abstractions of the actual machine-level sequences that interact
directly with the device by reading and writing specific registers of the device controllers.
The programmer can treat the high-level read instruction as a single primitive operation,
similar to a machine instruction, without worrying about the specific sequence of any
lower-level instructions executed.

Similarly, operations to manipulate files are viewed as primitives, without knowing
anything about the thousands of machine instructions executed to implement each high-
level file operation. The file operations may, in turn, use the abstract I/O operations to
perform their function. Thus, abstraction can be applied repeatedly to create multiple
levels of extended machines.

Another example of abstraction is the command language through which users
control the system’s operation. A command, for example, to load and run a program,

\[
\text{Read (f1, lblk, mm)}
\]

I/O system

Compute position of lblk on disk
Move read/write a run to corresponding disk track
Check for seek errors
Read physical block
Check for read errors
Copy block to mm

FIGURE 1-4. Principle of abstraction.
or to retrieve information about the system status, typically involves many lower-level operations by the system. From the user’s point of view, however, the OS appears as an extended machine with the set of commands as its “machine” language.

Operating Systems as Virtual Machines

Operating systems manage and control the sharing of hardware resources by multiple users. Although some applications must cooperate with one another, most simply compete for the available resources. Ideally, each application would like the entire computer system for its own use, without handling presence of other processes using the system concurrently. The concept of virtualization permits this sharing. It creates the illusion of having multiple copies of the same physical resource or a more powerful form of a given physical resource.

Similar to abstraction, virtualization is applied heavily by an OS at many levels, including CPU management, memory management, and device management. In a time-sharing system, the CPU time is sliced into small time intervals. Each process is allowed to run for only a short time interval, at which time the CPU is taken away from it and assigned to another process. The switching is done completely transparently to the processes, giving each process the illusion of having the CPU for itself. Thus, all processes continue making progress concurrently, albeit at a slower rate than if each had the actual hardware CPU entirely for itself. The time-slicing of the physical CPU implements a virtual CPU for each process.

The prime example of applying virtualization to create a more powerful resource than the physically available one is virtual memory. Whenever a given program exceeds the available memory space, it must somehow be partitioned into smaller segments that are then loaded into memory when needed. If no virtualization is used, i.e., the physical memory is visible at the program level, the programmer is responsible for determining which parts of the program are to reside in main memory at any given time. Modern OSs have liberated the programmer from this burden by implementing virtual memory, which supports the illusion of large contiguous storage regions. The OS accomplishes this by automatically transferring those program segments relevant to the current computation between main memory and secondary storage.

Virtualization also is applied to I/O devices. For example, rather than interacting with an actual hardware printer, an application writes its output to a virtual printer, implemented as a file. When the output is complete, it is sent to the actual printer without having to block the process. This spooling technique allows multiple processes to “print” concurrently, because each has its own set of virtual devices.

Figure 1-5 illustrates the principles of virtualization. The system hardware consists of a CPU, main memory, and a printer. Each hardware component is replicated into multiple virtual components by the OS and allocated to an application. Thus, each application runs as if it had its own dedicated CPU (created by transparent context switching), its own private memory (created by transparent memory management routines), and its own printer (created by spooling).

1The acronym “spool” stands for “simultaneous peripheral operations on-line.” The idea originated in the late 1950s on the IBM 7070 computer system, where it was first recognized that I/O-intensive jobs involving slow peripheral devices, such as printers, could be performed indirectly through secondary storage and in parallel with computational tasks. The concept was an important precursor to multiprogramming, introduced later in this chapter.
IBM pushed virtualization to its limit in the 1970s with the VM/370 OS. It applied the concept uniformly at the lowest possible level by creating multiple virtual machines that are exact replicas of an entire physical machine. Figure 1-6 shows the basic organization. The VM/370 runs on bare hardware. Rather than selectively virtualizing the CPU, the main memory, or the devices for different applications, it creates the illusion of multiple copies of a complete hardware system. Each of these complete virtual machines can then run its own OS. This system only needs to support a single user, who then has the entire (virtual) machine to himself, and thus can be considerably simpler than a general-purpose, time-sharing system.

Operating Systems as Resource Managers

A typical application executes code stored in main memory, reads data from some input device (e.g., a keyboard or a disk), and outputs results to a printer or disk. The performance of the application, even when run in isolation, depends greatly on the efficient use of various devices. In addition, it is affected by the presence of other applications that are running on or occupying the system simultaneously. One of the main tasks of an OS is to optimize the use of all its computational resources to ensure good overall application performance, while satisfying the needs and constraints imposed by specific applications, such as guaranteeing acceptable response time for interactive processes or meeting specific deadlines of time-critical tasks.

To ensure that a given application makes good progress, the OS must grant it sufficient resources. In particular, the OS must implement scheduling policies such that each process gets a fair share of the CPU. A process also requires an adequate amount of main memory to run efficiently. This is especially critical in systems with virtual memory, where processes frequently wait for the relevant portions of their program or
data space to be loaded from disk by the OS. The overhead of moving information between main memory and disk can be substantial and must be minimized. Finally, the OS should try to reduce the time that a process is waiting for its input to be ready or output to be processed. In the case of output, providing virtual devices can solve the problem of waiting. To speed up input, the OS may implement read-ahead strategies that get some data into memory in anticipation of read operations.

The overall performance of all applications can be improved by exploiting the potential parallelism opportunities in the system. Specifically, the OS should strive to keep the CPU, the main memory, and all its storage and I/O devices busy at all times by overlapping their operation whenever possible. Given the potentially large number of different devices, and the great discrepancies in speeds and capabilities, this task is a major challenge for any OS. To gain a better appreciation of the problem, briefly consider the major characteristics of different hardware components.

The CPU is normally the fastest component of a computer. Its speed is measured in CPU cycles, which typically range in hundreds of megahertz (MHz). The actual number of instructions executed per unit of time, however, is highly dependent on the CPU’s architecture, including the degree of pipelining, parallelism, and the amount of on-chip memory.

The speed of instruction execution is bounded by the speed of main memory; it takes approximately 10 nanoseconds or less to perform one read or write operation. Thus, main memory is an order of magnitude slower than the CPU. Since the discrepancy occurs at very low-level instruction execution, it cannot be resolved by any software mechanisms; rather, additional hardware in the form of faster (but smaller and more expensive) memory is necessary. Fast registers hold the current instruction, the program counter, intermediate data values, and various control and status values; caches hold the most recently accessed blocks of instructions and data, with the expectation that these will be accessed again in the future. Both registers and caches have short access times of approximately 1 nanosecond or less.

For nonvolatile secondary storage, most systems provide magnetic disks (hard disks or removable floppies) or tapes. The most important characteristics that differentiate between different device types are their capacity (i.e., the amount of data they can hold), their block size (i.e., the granularity at which information may be accessed), and their access time, which determines how long it takes (on average) to access a data item. Typical hard disk capacities are in the dozens of gigabytes, whereas tapes can hold several times as much. Block sizes for both range in the hundreds of bytes, but the access time for the disk is significantly shorter, typically 10 milliseconds or less. Furthermore, tapes only can be accessed sequentially, which makes them unsuitable for many applications.

Common I/O devices to allow users to communicate with their applications and the system include keyboards, display terminals, pointing devices (e.g., mouse, light pen), and a variety of printers. The most important quantitative characteristic of the different I/O devices is their speed, i.e., the rate at which they can transfer data to or from the system.

The different components of a computer system are interconnected via buses. These are sets of parallel communication links that allow the components to exchange data with each other. In the simplest case, a single bus may interconnect all components. Figure 1-7 shows a more typical architecture, where multiple buses with different speeds of data transmission and different protocols form a hierarchy. The memory bus allows the CPU
Section 1.2 Organization of Operating Systems

FIGURE 1-7. Hardware organization.

to access main memory. A hardware bridge connects the memory bus with the I/O bus; this allows the CPU to communicate with devices such as disks, printers, and terminals. These are attached to the bus via device controllers—specialized hardware components that provide a register-based interface to the CPU. Since the number of slots or places where a device controller may be attached to a bus is limited, multiple devices may share a single controller or have their own low-level bus. In the figure, two disks share a single controller. The slowest, character-oriented devices, such as the keyboard or the mouse, also are connected via a single controller that manages its own serial bus.

1.2 ORGANIZATION OF OPERATING SYSTEMS

Operating systems are large complex collections of software. To be maintainable, extendible, and comprehensible, they must be organized coherently according to general principles. We examine the organization of an OS from several points of view. The first is its structure, which describes how the different functionalities are grouped together and interact with each other. The second aspect, closely related to the system’s internal structure, is the set of interfaces that the OS presents to the outside world. The third view is the system’s runtime organization, which defines the types of entities that exist during execution.

1.2.1 Structural Organization

Any large software system is designed as a collection of functions that call each other. Those functions visible to the outside constitute the system’s programming and user interface; they form the extension of the underlying hardware by exporting a set of high-level abstract operations that applications may invoke.

If we impose no further restrictions on the internal organization of various functions, we end up with a monolithic structure—a single-level extension of bare hardware that provides a set of new functionalities. Any modification to such a structure requires a reccompilation and reinstallation of the complete system.
Monolithic systems are not only very difficult to maintain from the software engineering point of view, but also are very limited in their extensibility and adaptability to different situations. Thus, most modern OSs are designed as subsystem collections, each consisting of sets of functions but each having its own internal interface that facilitates a modular composition of the system. Furthermore, the functions are organized into multiple layers, such that functions at one level of the hierarchy may invoke functions at lower levels to perform their tasks. Thus, each layer may be viewed as a further extension of the ones below, i.e., a different abstract machine.

Figure 1-8 shows the simplest but most common organization of OSs, where the OS functions are split into two main layers. The lower layer, usually called the OS kernel, contains the most fundamental functions necessary to manage the system's resources. These are extended by a layer of system library routines that provide a greatly expanded suite of functionalities to meet application demands. The kernel and library functions are compiled into native code, i.e., instructions that execute directly on the machine hardware—the lowest layer of any computer system.

The application layer takes advantage of all three underlying layers. Most of an application program is compiled directly into machine instructions. Any library functions needed or invoked by the program are linked to the application so that the relevant (compiled) functions execute as part of the application. The application also may take advantage of the OS services by issuing calls directly to the system’s kernel.

1.2.2 The Hardware Interface

The lowest-level interface of the OS is between the hardware and the software; it consists of the machine instructions provided by the CPU. The OS is compiled into these machine instructions and runs entirely on the bare machine hardware. The system’s libraries and applications are compiled into machine instructions and thus run on the machine hardware, but the transitions between the user code and the system code must be carried
Section 1.2 Organization of Operating Systems  

out in a controlled manner that requires hardware support. In addition, the OS must respond quickly to events occurring asynchronously as the system executes. Modern CPUs support two hardware mechanisms that are essential for the above tasks, and thus allow the OS to fulfill its duties as a resource manager and supervisor. These are (1) interrupts and traps, and (2) multiple modes of instruction execution. Let us examine these in turn.

Interrupts and Traps

An important responsibility of an OS is the management of all the storage and communication devices. Efficient operation implies the ability to respond quickly to any change in the state of these devices. Such fast responses are generally accomplished by means of interrupt-driven processing.

An interrupt is a hardware signal issued to the CPU from some external device. Whenever an interrupt occurs, the CPU completes the current machine instruction, but, instead of fetching the next instruction specified by the program counter, it transfers control to a predetermined location. A special routine of the OS, the interrupt handler, is stored at this location. The interrupt handler analyzes the cause of the interrupt and takes appropriate actions. In the simplest case, control is then returned to the interrupted application. Figure 1-9a illustrates the basic principle. It shows how the normal flow of instruction execution is interrupted by the arrival of the external signal and temporarily diverted to the OS. The interrupt is transparent to the application.

There are two main applications of the interrupt mechanism:

- **Process management.** When multiple applications use the system concurrently, the CPU is switched between them according to scheduling policies. To allow the OS to periodically seize control of the CPU, a time-out interrupt can be generated...
14 Chapter 1  Introduction

by a timer device. The OS then invokes a scheduler, which reevaluates the overall situation and decides which tasks to run next.

- **Device management.** Most I/O devices are very slow compared with the CPU. Hence, the CPU can execute other programs while devices are moving data between external sources, main memory, or buffer registers. Whenever a device completes its operation, it generates an interrupt. This immediately alerts the OS which can then assign the next task to the device.

There are other situations that require the OS’s immediate attention, yet are not caused by external devices. Such events are handled through internal interrupt mechanisms called *traps*. Similar to an external interrupt, a trap also sets a bit internal to the CPU, which is automatically tested as part of every machine instruction. When it is set, the CPU suspends the execution of the current program and temporarily diverts control to the OS. The trap idea is sketched in Figure 1-9b. Both traps and interrupts transfer control to the OS—its interrupt or trap handler. The main difference is the source of the interruption: for interrupts, the event that triggers the transfer is external to the CPU, whereas for traps, it is internal.

A typical example of an event causing a trap is the overflow or underflow of an arithmetic operation. The current application cannot proceed beyond this point. Its execution is suspended (or terminated) and the OS is invoked to handle the situation. Another example occurs as part of main memory management. Large programs are sometimes partitioned into smaller chunks called pages or segments. An OS does not always keep all pages of an application resident in memory. Instead, it loads them only when they are actually needed, i.e., on demand. The loading of a page is handled by traps. When the application references a page that is currently not in main memory, the current instruction automatically sets a trap bit, and control is transferred to the OS to load the missing page.

Traps that are triggered by arithmetic errors or memory management are implicit. But traps also are used to cause a voluntary, explicit transfer to the OS. These are necessary whenever an application requests a service, such as an I/O operation, from the OS. The transfer is accomplished by a special machine instruction, a *supervisor call (SVC)*, that sets a trap and forces control to be immediately transferred to the OS. A parameter supplied with each SVC identifies the particular service that the OS must perform when the SVC is issued. SVCs are used to implement all kernel calls and form the basic interface between the OS kernel and the rest of the software.

**Modes of Instruction Execution**

There are certain machine instructions that should not be executed by ordinary applications. These include instructions that control I/O devices, set system registers or flags in the CPU, manipulate system tables, or modify external timers. Misuse of such instructions could jeopardize the security of the system, cause serious malfunction, or simply impede overall progress. To ensure that only trusted system programs execute such instructions, the instruction set is partitioned into multiple classes. In the simplest case, the instructions are divided into privileged and nonprivileged classes. To execute a privileged instruction, the CPU must be in a privileged mode, represented by a special
bit within the CPU; if this bit is not set, an attempt to execute a privileged instruction causes a trap, invoking the OS to handle the violation. The privileged mode bit can be set implicitly as part of interrupts or traps, or explicitly by issuing an SVC instruction. Because the SVC instruction immediately transfers control to the OS, no user program can switch to the privileged mode without also transferring to the OS at the same time.

1.2.3 The Programming Interface

As shown in Figure 1-8, a set of libraries extends the OS kernel, which, in turn, provides the basic functionalities of the system. From the application point of view, the difference between kernel and library functions is not very significant. In fact, some OSs, such as MS Windows, do not even disclose the exact list of existing kernel calls. Instead, the application program interface (API) defines only library functions, which invoke the appropriate kernel functions. The advantage of hiding the kernel from the application is that the kernel may be modified without affecting the correctness of any existing applications. In contrast, the UNIX philosophy is to publicize both the library functions and the available kernel calls. Jointly, the two layers define an extended machine, consisting of the nonprivileged instruction set and a set of functions that application programs may invoke as primitive operations.

From the implementation point of view, however, the two function sets are quite different. Kernel functions execute in the privileged mode, which gives them access to instructions and resources that ordinary user programs are not allowed to use. By restricting itself to only sensitive tasks, the kernel can be relatively small, and thus easier to test, verify, and maintain. All other functions are implemented as library functions that run in the nonprivileged user mode.

Figure 1-10 traces calls on both library and kernel functions. Figure 1-10a shows the control flow when an applications invokes a library function, \textit{lib\_func}(). For example, all mathematical or string-manipulating functions fall into this category. These are handled entirely through the normal function call mechanisms. The application code pushes the parameters on the calling stack and transfers control to the compiled function body. The body carries out the desired service and returns to the caller. After popping the calling stack, the application continues with the code following the function call.

Figure 1-10b traces the control flow when an application uses a kernel service that must execute in privileged mode. For example, most I/O operations fall into this class. The application issues a call to a function corresponding to the desired service, \textit{kern\_func}(). This is a library function just like \textit{lib\_func}() of Figure 1-10a. However, it does not perform the service directly; it only serves as an intermediary between the application and the kernel. Its main role is to set up the necessary parameters for an SVC call in specific registers and to issue the SVC. The actual service is carried out within the kernel in privileged mode. When this is accomplished, the nonprivileged mode is restored and control returns to the library function. The latter, in turn, returns to the application, which pops the stack pointer and continues executing.

The specific function sets provided by a given OS kernel and its libraries vary greatly between different OSs. The following classes of system calls (kernel or library) are generally available; each includes a broad range of functions to support the needs of different applications.
Chapter 1 Introduction

<table>
<thead>
<tr>
<th>application issues call to lib_func(params)</th>
<th>body of lib_func()</th>
</tr>
</thead>
<tbody>
<tr>
<td>push params on stack branch to lib_func() body</td>
<td>perform service return from function</td>
</tr>
<tr>
<td>pop stack pointer</td>
<td>(a)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>application issues call to kern_func(params)</th>
<th>body of kern_func()</th>
<th>kernel code</th>
</tr>
</thead>
<tbody>
<tr>
<td>push params on stack branch to kern_func() body</td>
<td>set up regs for SVC SVC</td>
<td>perform service set nonprivileged mode return from SVC</td>
</tr>
<tr>
<td>pop stack pointer</td>
<td>return from function</td>
<td>(b)</td>
</tr>
</tbody>
</table>

FIGURE 1-10. Invoking system services (a) nonprivileged library service and (b) privileged kernel service.

- Process management:
  - create/destroy, suspend/activate process
  - send signal or message to process
  - wait for event or signal

- Memory management:
  - allocate/deallocate region of main memory
  - increase/decrease size of allocated region

- File management:
  - create/destroy, open/close, move, rename file
  - append data to file or concatenate files
  - display content of file or directory
  - display/change file attributes

- Input/output:
  - read/write character or block of data from a file or device
  - control/check status of a device
• Miscellaneous services:
  – get current date and time
  – schedule an alarm to occur
  – apply mathematical or string-manipulation functions
  – generate various system diagnostics

1.2.4 The User Interface

The application layer (Fig. 1-8) contains both system and user programs. The distinction between the two is not strict; applications that aid in the development and use of other applications are generally considered system programs. These include compilers, assemblers, interpreters (e.g., the java virtual machine), text processors, Internet browsers, and other utilities typically provided by a computer facility.

Each application has its own interface through which the user may interact with it; for example, by supplying parameters and obtaining results of the computation. However, the system must provide one common interface that is ready when the system is initially booted up and which remains available at all times so that users can launch, use, monitor, and manage applications and files. This top-level user interface can have one of two forms: a text-based interface, generally referred to as the shell, or a graphics-based interface, commonly called a GUI (“graphics user interface”).

The Shell

A shell is a command interpreter that accepts and interprets statements (commands) issued by the user through the keyboard, mouse, or other pointing devices. A typical multiuser system would provide commands to perform the following tasks:

• Execute and manage programs
• Manipulate files
• Obtain system information

Not surprisingly, many shell commands are similar to the functions that make up the programming interface discussed earlier. For example, creating/destroying a file or spawning a new task can be done from within a running program or from a user terminal. The format used by a shell, however, is not that of a function call. Instead, the user types in the command name followed by a list of arguments. The command is sent to the shell when the user presses the ENTER or RETURN key.

Let us examine commands in each of the above categories in turn. The primary task of any computer is to develop or execute various application programs. Thus any command language contains statements directing the OS to load and execute programs, including system applications such as assemblers, compilers, interpreters, editors, debuggers, and other utility programs. Typical commands in this domain are those to link separately compiled modules together, load executable programs into memory, and start their execution. Each such command usually includes a wide range of optional parameters, for example, to control the format of the output or the amount of feedback generated, or to support testing and debugging.
The second category of commands relates to file management. These include various commands to create, delete, move, copy, append, print, or rename files and directories (folders), and to change their protection status.

The last class of commands are inquiries into the current state of the system and the state of the user’s applications. Information regarding the system that may be of interest to a particular user includes the current date and time of day, the number of users presently logged in, the system load, memory and device utilization, the system configuration, description of individual devices, and resource availability. Examples of information pertaining to the user’s process are the CPU time used so far, the connect time, files currently open, devices assigned to the process, and other runtime statistics.

In addition to user commands, the shell also may define a number of constructs that make it a complete programming language. This enables the user to write programs, called shell scripts, that tie together different commands and execute them, without the user’s help. The constructs and features supported by a shell fall into several categories:

- **Local variables.** The shell maintains a state during its execution, consisting of system-defined and user-defined variables, usually of type integer and string.

- **Iterative statements.** Constructs such as while or until-do allow the creation of loops.

- **Conditionals.** if-then-else statements permit different commands to be invoked based on runtime information.

- **Arithmetic and logical operators.** Variables, strings, or constants may be combined or compared in different ways.

- **Wildcards.** Names can be compared for partial matches; for example, in one standard notation, sect* matches any string beginning with sect, such as section; sect? matches any string beginning with sect followed by exactly one more character, such as sect9 but not sect10; and [sS]ect* matches any string beginning with sect or Sect, followed by any other characters.

- **Connectors.** Multiple commands can be combined to execute sequentially or in parallel; their inputs or outputs may be connected to files, the user terminal, or to each other.

- **Automatic name completion.** Command and file names need not be typed completely; instead, the user types only the first few leading characters and the system completes the rest as soon as the name becomes unambiguous.

### CASE STUDY: THE UNIX SHELL

The idea of turning the command language into a powerful programming language—the shell—was pioneered in the UNIX operating system. When a user logs into the system, the shell is started and a prompt is displayed. Statements of the following form are then expected:

\[
\text{command}_1 \ \text{arg}_1 \ldots \text{arg}_k; \ldots; \text{command}_n \ \text{arg}_n \ldots \ \text{arg}_n.
\]
The semicolon functions as a RETURN or ENTER key: it allows multiple commands to be issued using a single command line. Proceeding from left to right, the shell interprets each \textit{command} as a program to be executed. It searches the file system directory and, if a file with the name \textit{command} is found, it creates a new process that then executes the program with its arguments \textit{arg1} \ldots When all commands in the sequence have been processed, control returns to the shell, which is ready to interpret the next sequence. Examples of simple commands supported by the shell include:

\begin{table}[h]
\centering
\begin{tabular}{l|l}
\hline
\textbf{command} & \textbf{effect} \\
\hline
\texttt{who} & Generate a list of all users currently logged in. \\
\texttt{pwd} & Print the name of the current working directory. \\
\texttt{cd f} & Set the current working directory to be the file \textit{f}. \\
\texttt{ls} & List in alphabetical order all files recorded in the current directory; several arguments may be specified to modify the basic \texttt{ls} command; for example, \texttt{ls -t} lists the files according to their last modification date. \\
\texttt{du} & Return information about the current disk usage; an argument may be specified to determine the disk usage for only a certain subtree of the current file hierarchy. \\
\texttt{cat f1 f2} & Concatenate the two files \textit{f1} and \textit{f2} and type them on the user terminal. \\
\texttt{lpr f} & Send the file named \textit{f} to the printer; additional parameters may be supplied to specify the desired printer, number of copies to print, page layout, handling of special characters, and other options. \\
\texttt{wc f} & Count the number of lines, words, and characters in the file \textit{f}.
\texttt{man c} & Print on-line manual pages pertaining to the command \textit{c}. \\
\hline
\end{tabular}
\caption{Examples of simple commands}
\end{table}

Multiple commands can be submitted simultaneously; the semicolon indicates sequential processing of commands. The shell also permits several tasks, each started by a different command, to execute concurrently. This is accomplished by including the character “\&” as the last argument of a command. In this case, the shell does not wait for the termination of the corresponding process but immediately proceeds with the next command. For example, the sequence:

\begin{verbatim}
command1 \ldots \&; command2 \ldots
\end{verbatim}

starts two concurrent processes, executing the files \textit{command1} and \textit{command2}. When both terminate, the shell returns to the user with a prompt to accept the next line of commands.

In comparison, the sequence:

\begin{verbatim}
command1 \ldots \&; command2 \ldots \&
\end{verbatim}
also starts two processes concurrently; however, it does not wait for their termination but returns to the user immediately. In this way, background processes may be started and continue executing while the user is free to further use the shell.

Shell facilities also are available for directing the I/O of programs to various devices, files, and other programs. Each time a shell is started for a user, two files, named standard input and standard output, are automatically opened. Any user input statement receives data from the standard input, whereas any output is sent to the standard output. Normally, these standard I/O files are associated with the user terminal, thus permitting the user to communicate directly with the currently running program. However, the shell also permits the standard input and standard output to be reassigned to other files or devices. This is accomplished by specifying one or both of the arguments:

\[ <\text{in\_file}, \>\text{out\_file} \]

where in\_file and out\_file become the I/O files, respectively. The symbols ‘<’ and ‘>’ indicate the direction of the data flow, i.e., ‘from’ in\_file and ‘to’ out\_file. The assignment is valid for the duration of the command. For example, the following command line uses a temporary file to pass the output of one program (command\_1) to be used as the input to another program (command\_2):

\[ \text{command\_1} > \text{temp\_file}; \text{command\_2} < \text{temp\_file} \]

The shell offers a more elegant solution to passing temporary data between programs; it permits the standard output of a program to be directed into the standard input of another program without the use of a temporary file. The two programs are connected using a construct called the pipe, for example:

\[ \text{command\_1} \mid \text{command\_2} \]

Unlike the previous solution, both processes are started concurrently, with the standard output from command\_1 becoming the standard input to command\_2. They may be viewed as a producer and a consumer, respectively, with the pipe serving as a virtually unbounded buffer. The necessary synchronization is performed internally by the system.

The following code is a simple shell script illustrating the use of shell variables and control statements.

```
COUNT='who|wc -l'
if test $COUNT -lt 8
    then echo $COUNT users logged in
    else echo too many users
fi
```

When this code is placed into an executable file, e.g., check\_users, then typing the command check\_users on the command line will display the number of users currently on the system. The script works as follows. The system utility who generates a line
Section 1.2 Organization of Operating Systems

for each user currently on the system. This is piped into the `wc` utility, which, given the flag `-l`, counts the number of lines produced by `who`. The result is stored in the new shell variable `COUNT`. The next line checks the value of `COUNT`; the `$`-sign refers to the variable’s content, rather than its name. If `COUNT` is less than 8, the script displays (using the `echo` command) the message “N users logged in,” where $N$ is the actual number of users; if `COUNT` is greater than or equal to 8, the script displays the message “too many users” instead.

A Graphics User Interface

Using the system through a text-oriented shell requires considerable expertise. To cater to the increasing numbers of casual users, Apple introduced a graphics-based user interface (GUI) in the mid-1980s which was based on seminal work done earlier at the Xerox PARC research laboratory. Instead of typing previously memorized complex command and file names, users click on graphical representations of commands, called icons, to invoke applications. Since then, many systems have adopted the graphics-oriented approach to their user interfaces. Figure 1-11 contains several commonly used icons. Each corresponds to an application that the user can start by simply double-clicking on the corresponding icon. Some of these applications are specific to a given OS. For example, the *Windows Explorer* icon starts the File Manager of the Microsoft (MS) Windows OS. Other icons, such as *America Online* or *Netscape Communicator*, represent independently developed products.

When an application is started, a new area or window of the display is created for its execution. The user can manipulate windows by changing their size, shape, screen position, and visibility relative to other windows. The user also can control the actions taken when the mouse cursor is placed on a particular portion of a window or when a mouse button is pressed. Typically, pressing the right mouse button displays a drop-down menu (a small window listing a number of commands that can be performed by selecting them with the mouse cursor). For example, in MS Windows—an OS that took its name from the windowing concept—pressing the right mouse button on the initial start-up screen or “desktop” displays a drop-down menu with commands that allow the user to arrange the icons on the screen in different ways, create new folders and objects, or find out the properties of the display.

![Commonly used icons](image-url)
Chapter 1  Introduction

An important feature of a GUI is the ability to move objects around by dragging them, i.e., by clicking on an object and then moving the mouse while depressing the button. For example, with this feature, icons can be arranged on the desktop manually, and files can be moved between different folders. A file or an icon also can be easily deleted by dragging it over a special icon; for example, in MS Windows, the icon is named the Recycle Bin.

The main attraction of an icon and menu-oriented GUI over a text-based shell is its visual approach that makes many common tasks more intuitive and thus easier to perform. By establishing simple overall conventions, such as starting all applications by double-clicking a mouse button or displaying drop-down menus with the right mouse button, the GUI enables a user to navigate through the system without prior knowledge of any command and file names. Not surprisingly, GUIs are supported by virtually all OSs today. However, text-based interfaces have not become obsolete. Expert users and programmers require greater control over the system’s resources and must be closer to the system code than is possible with an icon- and menu-based interface. Consequently, UNIX, Linux, and other systems continue to provide a shell, but even these are generally used via a graphics interface such as X Windows. Such a system offers the flexibility of multiple windows on the display, but allows the use of a text-based shell via typed commands and file names within any window.

CASE STUDY: WINDOWS EXPLORER VS. THE UNIX SHELL

To illustrate the difference between a text-based and a graphics-based user interface, we contrast the GUI of the MS Windows OS with the UNIX shell. In both cases, we consider the task of managing files.

The file manager of MS Windows, Windows Explorer, has the GUI of Figure 1-12. It is divided into two panels containing file information and a number of buttons and drop-down menus through which the files are managed. The left panel shows the directory structure, starting with the highest file directory (folder), the Desktop. The latter consists of the directory My Computer that lists the available disk drives, such as a 3 1/2 Floppy (A:), an internal hard disk (C:), and a CD-ROM drive (D:). Each directory is preceded by a box containing a plus or minus sign. Clicking on a minus sign expands the directory to show the next level. In the figure, the hard-disk directory C: is expanded, showing a number of subdirectories: Adobeapp, Aladdin, America Online 5.0, Documents and Settings, and so on. The directory America Online 5.0 is expanded further to show its subdirectories, Aoltemp, backup, and so on.

The right panel shows the contents of one directory in the structure on the left—the one selected with a mouse click or a cursor key. In the figure, the directory backup is selected (highlighted in the structure); it contains a subdirectory Temp and two files, Global.org and main.idx.

Browsing files and directories. Windows Explorer has a very convenient way to browse through existing directories. Using the plus/minus signs and the sliding bars of the panels, the user can quickly examine the directory structure and obtain important information about existing files, their sizes, types, and other properties. In contrast, a UNIX user must be familiar with several commands. The pwd (print working directory) command is used to determine the current directory (the one corresponding to the highlighted one in the Windows Explorer). The ls (list) command lists the contents of
the current directory; this corresponds to the content of the right panel in the Windows Explorer. The directory hierarchy can be navigated with the `cd` (change directory) command; Windows Explorer accomplishes this by browsing and selection as above.

Creating and deleting files and directories. These operations are menu driven in Windows Explorer. The user selects the File menu, which lists commands to Create, Delete, or Rename files and directories. In UNIX, the user must be familiar with the available commands, such as `rm` to remove a file or `mkdir` to create a new directory.

Moving files and directories. With Windows Explorer, one simply selects the appropriate file or directory with the mouse and drags it to the new destination directory. UNIX users use the `mv` (move) command, specifying as parameters the file or directory to be moved and the new directory.

Displaying file contents. Each file in MS Windows may be associated with an application that handles the specific file type. To view text files (ASCII characters), the user invokes a text editor, such as `Notepad`, designed to handle text files without inserting any formatting information. In contrast, UNIX provides a single powerful command to handle text files, denoted `cat` (for concatenate). Typing `cat` followed by a file name displays the file’s content on the user terminal. But `cat` also can be used to create new files, to append text to existing ones, or to combine (concatenate) existing files. For example:

\[ \text{cat } f1, f2, f3 > fnew \]

combines the contents of the three files \( f1 \) through \( f3 \) and writes their contents into the newly created file \( fnew \).
Chapter 1  Introduction

CASE STUDY: WINDOWS EXPLORER VS. THE UNIX SHELL  (continued)

The above comparisons between common operations under MS Windows and UNIX demonstrate how a graphics-oriented user interface offers a more convenient, intuitive way to manage files and directories. Thus most systems, especially those aimed at nonprogrammers and casual users, have a GUI as their high-level OS interface. A text-based shell, however, is still essential to satisfy the needs of many programmers and expert users. Commands can be combined into complex scripts, allowing the efficient and convenient invocation of frequently used sequences. A text-based interface is also more flexible and expressive because each command can specify a wide range of parameters and control flags. For example, the \texttt{rmdir} (remove directory) command of UNIX lets a user recursively delete all files and subdirectories of a given directory; while doing so, the user may choose an interactive mode, requiring a confirmation of certain file deletions. The user also may control the level of diagnostic feedback displayed during the operation. Other commands, such as \texttt{ls} (list), have dozens of optional flags and parameters that can control which file types to display, the format of the display, the level and extent of the attributes shown, and other options. A menu-driven interface is just not powerful enough to capture all the combinations and nuances needed by expert users.

1.2.5 Runtime Organization

Structurally, any OS is organized as a collection of functions as described in Section 1.2.1. At runtime, we have two fundamental choices of how to invoke any given function. To illustrate the differences, consider for example a simple service function, \texttt{time()} that returns the current time of day to its caller. This function can be implemented as a library function or a kernel call if privileged execution was necessary (Section 1.2.3). Any application needing the service can simply call the function, which then executes as part of the invoking process. This choice is sketched in Figure 1-13a. Any OS service that

![FIGURE 1-13. Runtime organization of an operating system (a) service as a subroutine and (b) service as a process.](image-url)
Section 1.3 Operating System Evolution and Concepts

is called explicitly by an application can be handled in this manner, i.e., as subroutines of the user processes. Only services invoked by hardware interrupts are handled by a special OS process.

The second option is to provide OS services as separate autonomous system processes. For the time-of-day service, a timer-server process can be part of the OS. Its task is to accept requests from other processes in the form of messages, execute the \textit{time()} function for each such request, and return the result to the caller, again in the form of a message. Under this implementation, shown graphically in Figure 1-13b, the process providing the service is generally referred to as a \textbf{server} and the calling process termed a \textbf{client}.

The disadvantage of the client-server implementation of system services is that the OS must maintain a number of permanent server processes that listen and respond to various requests. But the client-server approach has important benefits:

- It is suitable for a \textit{distributed system} where a collection of separate machines is connected through a network, and integrated with a single system and set of user interfaces. Here, a given service may be implemented on a separate dedicated machine and invoked remotely by different clients. A network file server is a typical example of such an organization.

- The organization is convenient for many different types of services, not only for those provided by the OS directly. The various services offered over the Internet are examples.

- It has a greater degree of fault tolerance than an organization based on function calls. When a server process (or its machine) crashes, other services of the OS may continue functioning. In contrast, when services are implemented as subroutines invoked as part of the calling processes, a crash generally brings down the entire OS, which must be restarted.

- A client-server organization enforces a strict segregation of functionalities, which makes them easier to understand, debug, and maintain, than large homogeneous collections of functions calling each other.

Most modern OSs use a combination of the two approaches to runtime organization. Services that maintain a global view of resources and that must be constantly available are normally implemented as autonomous concurrent processes, whereas other services are constructed as subroutines invoked as part of existing processes. For example, the UNIX scheduler, which selects the next process to run on the CPU, runs as a separate process, whereas many other UNIX services execute as subroutines.

1.3 OPERATING SYSTEM EVOLUTION AND CONCEPTS

Operating systems have been evolving steadily since their first appearance in the 1950s, driven in part by advances in technology and by growing demands of the user community. Some early developments have become obsolete, but many are still present. Here, we introduce the most important concepts that make up the field and examine their evolution by identifying the main driving forces and the solutions that have emerged in response to the new challenges.
1.3.1 Early Systems

From about 1949, when the first stored-program digital computer actually began executing instructions, until the mid 1950s, the basic organization and mode of operation for computers remained relatively constant. The classical von Neumann architecture implied strictly sequential instruction execution, including I/O operations. To run a program, the users would work at a dedicated systems terminal, called the console, which was generally the only direct (online) connection to the machine. Users would interact with their computations, one at a time on the machine and at the lowest machine level, by manually examining and setting machine registers, stepping through instructions, and looking at contents of memory locations. Programs were written in absolute machine language (using a decimal or octal notation) and loaded into the machine by an absolute loader, which required the user to specify the exact physical memory locations for all code and data.

In these early years of computing, programming aids were either nonexistent or minimal; the most advanced installations featured assemblers to liberate the user from the lowest-level machine language programming, and a few rudimentary library routines to handle the most tedious and repetitive tasks. As the importance of symbolic programming was recognized and assembly systems came into more widespread use, a standard operating procedure evolved: A loader would read an assembler into main memory and transfer control to it. The assembler would read the user source program and library routines, generally from punched cards or paper tape, and then assemble these into executable machine code. Because main memory was very small, the executable code would be written to a magnetic tape or punched cards as it was being produced by the assembler. A loader would again be invoked to read the complete assembled program into main memory, which would then be executed. Any output generated by the program was either written to tape or cards, or printed by a line printer.

A difficulty faced by even the earliest systems was initial startup. The CPU could only execute instructions in main memory, but main memory was volatile—it lost all information when power was turned off. (Older core memories were not volatile, but were small and continually being overwritten.) Thus, the problem was how to get a program, for example, the initial loader capable of loading other programs, into main memory when power was turned on. To accomplish that task, a special hardware mechanism was used to transfer a minimal amount of code into memory and cause it to be executed. This initial program then loaded larger programs into memory, until the entire OS was resident and ready to control the system.

The above approach to solve the initial system startup problem has been described by a metaphor—a person attempting to pull himself up by his own shoe laces or boot straps. Thus, initializing a system after power is turned on was widely termed bootstrapping and later simply booting.

Bootstrapping in older systems was typically done from disk. Turning on power would cause a special hardware circuitry to copy one data record from a predefined disk location into a fixed set of main memory locations, and set the program counter so that the CPU would begin executing this initial program. This program would then incrementally load the complete OS from the disk. In present-day computers, the bootstrapping problem has greatly been simplified by the availability of nonvolatile executable memory, typically in the form of read-only memory (ROM) that can hold the initial startup software. For example, most personal computers (PCs) contain a set of basic I/O functions, called the
basic input output system (BIOS), in nonvolatile memory. It starts automatically when the system is turned on or when the reset button is pressed. BIOS examines the existing hardware configuration and then attempts to “boot” the system, i.e., to load the actual OS. It first tries to read it from a diskette drive, giving the user the opportunity to boot the system from a chosen diskette. If no diskette is present, BIOS attempts to boot from the CD-ROM drive or from the hard disk.

1.3.2 Batch Operating Systems

The first generation of OSs, built in the late 1950s, was motivated by the inefficiencies of the early systems. Because the latter required manual assistance from an operator at each of the steps of loading, assembly, and execution of a program, they consumed a great deal of time and resulted in the inefficient use of the very expensive hardware.

The solution was to automate the standard load/translate/load/execute sequence with a central control program that would find and load required system programs—assemblers, compilers, linkers, or library subroutines—and handle the job-to-job transitions automatically. This permitted submitting multiple jobs to the system for processing at the same time, typically in batches of punched cards. The control program became known as a batch operating system.

The development of these systems was greatly accelerated by new hardware technologies. First, vacuum tubes were replaced by transistors, which made machines considerably smaller, more reliable, and cheaper. I/O processing also was improved significantly. One of the most important hardware innovations was the I/O processor or I/O channel, a processor similar to the CPU but with a much smaller instruction set specialized for I/O processing. An I/O processor frees the main processor—the CPU—from frequent low-level interactions with I/O devices. On receiving a high-level I/O request from a CPU, an I/O processor executes a specialized program to carry out the high-level operation. The CPU is involved only when the operation has been completed. The I/O processor made it possible to efficiently overlap the operation of the CPU and many I/O devices, thus improving overall system performance through parallelism.

Initially, the CPU would periodically interrogate the status of the I/O processor to determine if it was still busy or had encountered some error. But it soon became clear that the system could operate more efficiently if one could find a mechanism that permitted the I/O processor to inform the CPU as soon as it needed its attention, either to report a problem or to accept a new command. The solution was the introduction of interrupts—hardware signals that the I/O processor could send to the CPU to get its immediate attention. Interrupts were later expanded to allow the CPU to respond quickly to a variety of different conditions, such as a division by zero, invalid opcode, memory protection violation, and a host of others.

To further streamline the operation of batch systems, compilers were written to produce relocatable rather than absolute code. Linking loaders were then provided so that separately compiled programs, including library routines, could be combined without recompilation. As a result, services from the human user were needed only to manage the I/O batches, to set up nonstandard jobs, and to take corrective action in case of system failure. The programmer was gradually banished from the machine room, which became the exclusive domain of a new specialist, the computer operator.

The term batch is still widely used today but its meaning has evolved to refer to noninteractive computing. A batch job is one that does not require any user interaction; it
is submitted to the system with the understanding that it will be executed at a time chosen by the system as the most convenient. For example, the system will generally execute a batch job at a lower priority than interactive jobs. It may also choose to postpone its execution until it has spare capacity or until it has accumulated a suitable collections of jobs to run. Alternatively, the user may schedule the job’s execution to commence at a later time (e.g., at night).

1.3.3 Multiprogramming Systems

The introduction of interrupts, I/O processors, and controllers with direct access to memory enabled the CPU to keep many devices operating simultaneously and, at the same time, perform other computations. A single program, however, is rarely able to exploit this potential. Rather, a typical program alternates between phases of input, computation, and output, and limits itself to using just a few of the existing devices.

Consider, for example, a simple program which reads a file, performs some computation on the data, and outputs the results. Even with the most sophisticated interrupt and I/O processing hardware, the overlap between CPU and devices is limited to that of Figure 1-14. Clearly, the CPU is drastically underutilized during the I/O phase; the system is said to be I/O bound during this phase. Conversely, all devices are idle during the compute phase; in this interval, the system is said to be compute bound. Furthermore, even when the CPU and the I/O devices are kept busy most of the time, the use of other computer resources is often poor; for example, if the program is small, much of main memory remains unused.

A solution is to run several programs or jobs in an apparently simultaneous manner by storing them in main memory at the same time and interleaving their executions. This is the basic idea underlying multiprogramming; the OS controlling this kind of virtual parallelism is termed a multiprogramming system. This strategy not only decreases the amount of idle resources, but also improves the system throughput and turnaround time for individual programs.

A particularly good choice of programs to run simultaneously is an I/O-bound program, such as one updating a large data file, and a compute-bound program, such as one solving a set of partial differential equations. While the I/O-bound program is waiting for I/O, the compute-bound program can compute. On I/O completion, control is switched back to the I/O-bound program. Figure 1-15 illustrates the execution of two such programs in both a single-programmed and a multiprogrammed environment. The time saved by multiprogramming is evident in this example. The total time to execute both programs is reduced by eliminating some of the idle times of both the CPU and the I/O device.

![Figure 1-14. CPU and I/O overlap in a single program.](image-url)
Section 1.3 Operating System Evolution and Concepts

(a)

(b)

FIGURE 1-15. CPU and I/O overlap under (a) sequential execution and (b) multiprogramming.

Note that multiprogramming involves the **sharing of time** on processors and the **sharing of space** in main memory, and the potential sharing of other resources. An additional space-saving benefit accrues from the ability to **share code** resident in main memory. If several programs require the same language translator or request the same system service, a single copy of these routines may be kept in memory and used by all. Sharing hardware and software resources through multiprogramming is beneficial to users in another important way—it makes it economically possible to provide a wider variety of services.

To support multiprogramming, the underlying computer hardware should include several key components, and the OS must be extended to efficiently and transparently manage the system resources in this complex environment. A multiprogramming system attempts to hide the fact that resources are shared among different processes or users. Except for the slower speed at which services are performed, each user should have the illusion of having the system entirely for himself. With this kind of environment, three new and important issues must be addressed:

**Protection**

Since the system is shared by many different users, an OS must ensure its own integrity and the integrity of user processes from accidental or malicious damage. In addition, privacy should be guaranteed, i.e., information must be protected against snooping by
unauthorized user programs. Finally, ownership of data, programs, or services also must be respected, which means that the OS prevents unauthorized execution of programs or invocation of services.

Process Synchronization and Communication
In a multiprogramming system, different processes may be executing concurrently. In many situations, two or more processes may compete for a common resource that they need to read or modify. A shared file is a good example: to preserve the integrity of the file’s content, it may be necessary to impose an order in which the file may be accessed. Similarly, processes may wish to cooperate by exchanging information at various stages of their execution. The OS provides synchronization and communication primitives that facilitate correct programming for such competition and cooperation.

Dynamic Memory Management
When a program waits for a lengthy event, such as the completion of an I/O operation, it may be desirable to remove it temporarily from main memory so that other programs can use the otherwise wasted space. As a practical matter, when swapping programs in and out of main memory, it should be possible to load a given program into different locations each time it is returned to main memory. Furthermore, a program should be permitted to only partially reside in main memory. This not only improves memory utilization but also permits execution of programs whose total size exceeds the size of main memory. Thus, effective management of main memory under multiprogramming is a major challenge for the OS.

As a consequence of all of the software and hardware requirements for multiprogramming, computers of the 1960s grew considerably in size, frequently filling entire rooms. The descendents of such computers or mainframes are still in use today, mainly employed as central servers of very large databases, libraries, or Internet servers. Significantly, the techniques and issues of multiprogramming still underlie all contemporary OSs.

1.3.4 Interactive Operating Systems
The main focus of early OSs was performance, especially as measured by job throughput. Given the high cost of the underlying hardware and software, the overriding concern was maximizing the number of jobs processed per unit of time. By using interrupt mechanisms, I/O processors, and devices with direct memory access, batch systems were able to free the CPU from most of the low-level I/O processing while permitting different I/O devices to operate concurrently. The objective of multiprogramming is to generate sufficient workload so that much of the available CPU and device time could be exploited. The success in meeting this objective is certainly one of its great benefits.

A second major benefit is the ability to simultaneously share a machine’s resources so that more than one online user may interact with the system simultaneously. However, to make this approach practical, it is necessary to guarantee some acceptable response time to each user at his terminal. This implies that even programs that display an ideal behavior in terms of a balanced resource utilization must not be permitted to run for extensive periods of time if other programs are waiting. Rather, all active processes must receive a fair share of the available resources, especially the CPU time. One common
Section 1.3 Operating System Evolution and Concepts

System policy is to serve all active processes cyclically by assigning to each a small portion or quantum of the available CPU time, for example, $1/n$th of a second every second for $n$ users. This time-sharing concept employs the basic principles of multiprogramming to create the illusion of a separate virtual machine for each user. Conceptually, each of these virtual machines is an exact image of the actual machine, except for its speed which is divided by the number of currently active users.

One of the first successful time-sharing systems was Compatible time-sharing system (CTSS) developed at MIT in the early 1960s. This was followed by the pioneering Multiplexed information and computing service project (Multics), which yielded many techniques in memory and process management that are still in use today.

Although mainframes continued to be popular as large-scale interactive systems, a new line of smaller computers, known as minicomputers, emerged in the late 1960s and began to dominate the market in the 1970s. These computers were many times more affordable than mainframes, making it possible for smaller organizations and units within organizations to acquire their own computers. The smaller size of minicomputers led to scaled-down versions of the large time-sharing OSs. One prominent example is the UNIX system. Since its creation in the late 1960s, it has gone through many variations and extensions, and remains a favorite in its modern incarnations.

1.3.5 Personal Computer and Workstation Operating Systems

During the 1970s and early 1980s, technology continued to develop at a rapid pace. Large-scale integration (LSI) and very large-scale integration (VLSI) made it possible to manufacture large quantities of powerful hardware components at very low cost. This caused a new major trend in the computer marketplace: the emergence of personal computers (PCs) and workstations. LSI made it possible to manufacture such computers as powerful as many of the earlier mainframe computers but at a fraction of their cost, and thus affordable to individual users.

Because of their relatively simple hardware and single-user mode of operation, no major innovations in OS technology were necessary; rather, most of the principles governing OS for earlier large computers could still be applied. The main significant innovation was the development of GUIs (Section 1.2.4). These interfaces make computers more accessible, especially for the average consumer and professional user, where the main goal is to run existing applications, such as word processors, Internet browsers, or computer games (in the case of consumers), and specific applications for professionals in engineering and research organizations.

Early OSs for PCs, such as DOS, were not only single-user systems but also single-program systems, similar to those of early-generation mainframes. Some were later extended to support multiprogramming, but only in a greatly simplified form where the user is responsible for manually switching between the active tasks. This form of explicit multiprogramming does not improve performance by overlapping CPU and I/O execution, but it greatly improves the system’s usability by allowing the user to work with multiple applications concurrently. For example, while querying a database or debugging a program, the user may wish to keep a word processor open to record notes or to save important pieces of information for later use. The switching between the concurrent tasks is not initiated by the OS, but explicitly by the user (e.g., by clicking the mouse on the window he or she wishes to work with).
More recent PC OSs (e.g., MS Windows 3.x) support *implicit multiprogramming* (called multitasking) where the switching among concurrent tasks happens automatically and transparently to the user. But the tasks must all cooperate in this effort by periodically releasing the CPU so that another task could get its turn; a renegade task that refused to yield would block all other tasks from making any progress. Hence, this form of multiprogramming is referred to as *cooperative multiprogramming*.

True **preemptive multiprogramming** requires that any computation be periodically interrupted and the control given to the OS, which is then able to assign the CPU to different tasks to preserve fairness. This form of multiprogramming, supported by IBM’s OS/2 operating system and MS Windows 95 and later versions, is transparent not only to the interactive user but to the applications themselves.

In the workstation world, more sophisticated OSs were necessary and available from the start, commonly based on UNIX. The systems for the Sun and DEC workstations are good examples. Modern PC and workstation OSs are equally sophisticated and essentially indistinguishable.

In addition to rapidly declining hardware costs, another major driving force responsible for the proliferation of PCs and workstations was the development of networking technologies. LANs made it possible to share costly resources by multiple users and to support efficient communication among them. They also made it possible to dedicate individual machines to specific functions, such as maintaining file systems for other machines on the network. Such functional specialization leads to better utilization of resources, increased flexibility in configuring the system, and scalability. A related result was the development of distributed OSs, introduced in Section 1.3.7.

Wide area networks, notably the Internet, allow the sharing and dissemination of information and services on an unprecedented scale, providing even more motivation for many users to acquire computers. From the OS point of view, one major challenge in such highly distributed environments is to cope with rapid change. This forces them to be highly modular and extensible to rapidly incorporate new functionalities, at the same time guaranteeing adequate protection vis-a-vis an increasingly unpredictable and open-ended environment.

### 1.3.6 Real-Time Operating Systems

A large class of computer systems is devoted to **real-time** applications where the purpose is to monitor, respond to, or control some external process or environment (Shaw 2001). Examples include complete stand-alone systems, such as air traffic control, medical patient monitoring, power plant control, and military command and control systems, and **embedded** systems that are a component of some larger system, such as an automobile control system or a cell phone. Real-time systems have existed since the first days of computers, and their operating software has evolved in tandem with that for general-purpose systems.

However, there are some important differences between the requirements of real-time and conventional software. First, real-time systems must satisfy timing constraints that are imposed upon them by the external world to which they are attached, most commonly in the form of deterministic **deadlines**. Failure to meet a constraint generally means that the system fails. **Concurrency** is another distinguishing feature; external signals arrive asynchronously and sometimes simultaneously, and must be handled in parallel to meet constraints. Many applications have strict **fault tolerance** and
Section 1.3 Operating System Evolution and Concepts

reliability requirements; failures can cause mission, money, property, and even human life losses. These systems also are exceptionally difficult to test and certify because of their complexity and the high cost of failure; live debugging usually is not an option.

Most of the technology for conventional general-purpose OSs has been adapted and extended for the real-time domain. For example, there are many variations of real-time UNIX, and Linux has been modified for real-time use. The three ideas of abstraction, virtualization, and resource management still dominate real-time OS design, as they do for standard OSs. But there are several critical distinctions, related to the unique requirements of real-time processing described above.

Most of the difference follows from the need for predictability. OS functions, while supporting high-level abstractions for the application’s programmer, should have bounded and known execution times. The allocation or scheduling of CPUs to various OS and applications processes must ultimately be under the control of the user, and usually based on time so that deadlines are met. Main memory management, even with virtual memories, cannot involve asynchronous and unpredictable swapping of pages or segments. Instead, the allocation should be preplanned. Predictable and usually contiguous placement of file contents on secondary storage also is necessary to avoid lengthy and essentially nondeterministic file operations, such as seek operations to dynamically determined disk cylinders. Finally, a real-time OS provides user handles for inevitable errors, such as missed deadlines, which are guaranteed to occur in any practical system. Particularly important system services are accurate and controllable clocks and timers.

1.3.7 Distributed Operating Systems

Probably the most significant recent trend in OSs is distributed systems that run over multiple machines connected through one or more networks (Tanenbaum 1995). Virtually all computers are attached or attachable to a communications network. The complete system—machines, network, and software—may be loosely coupled or tightly coupled. Loosely coupled systems are those where each node is relatively independent, communicates at a level not much higher than message passing, and may be connected over long distances, e.g., over a WAN. The other extreme, a very tightly coupled system, is much more integrated, typically in a LAN or multicomputer configuration, and often supports higher-level communication on top of message passing, usually remote-procedure call. In both cases, the OS on each machine or cluster of machines is responsible for handling communication, in much the same way that it supports other I/O.

A client-server software architecture works well for loosely coupled systems. Servers (e.g., file servers, printer servers, special-purpose computational servers, or timer servers) can be implemented on separate nodes of a network with a separate and independent OS kernel in each node. A common communications framework (e.g., transmission protocol and data coding) will ensure coherent client-server interactions. However, the system complexity increases dramatically when the servers themselves are distributed over several nodes. Tightly coupled environments offer a more transparent interface to the user. Ideally, the user should not even be aware of the distributed nature of the underlying system, which still typically uses a client-server approach.

Generally, several new problems arise when designing distributed OSs. One important issue is time. It is a critical part of many applications; for example, time stamps may identify the most recent version of a file or determine the sending or arrival order of messages. To use time correctly, it is imperative that clocks on different machines
be synchronized, accurate, and monotonic increasing—characteristics that are extremely
difficult to achieve and maintain in a distributed system. Networks are normally much
less reliable than single systems, and individual machines occasionally crash. Fault tol-
erance considerations lead to protocols and higher-level software that includes extensive
checking, reconfiguring, checkpointing, and restarting or retransmitting.

Distributed file systems can be especially interesting. Files and directories may be
located on more than one machine and throughout the network. At any time, several
copies of a file could exist, each located at a different machine and each undergoing
various update or read operations. There is the thorny problem of ensuring that each user
views the same coherent version of a file and the directory system. Additionally, there
is the difficulty in creating a file-naming scheme that is location independent.

Starting in the late 1970s and early 1980s, many distributed OSs were built. Exam-
ples that emphasize small kernels and efficient message passing include the V distributed
system (Cheriton 1988) and Mach (Accetta et al. 1986). The Amoeba system (Tanenbaum
et al. 1990) is an object-oriented OS that presents complete distribution transparency to
the user. All three systems, V, Mach, and Amoeba, offer UNIX-like interfaces or directly
emulate UNIX.

THE END OF THE BEGINNING

This ends our introduction and overview. We now start from the “bottom-up” and delve
into the details of OS design, analysis, and construction. Part I, Chapters 2, 3, 4, 5, and
6, are concerned with the management and coordination of processes: the fundamental
computational units that are controlled by an OS. The problems and techniques of main
memory management are presented next in Part II, consisting of Chapters 7, 8, and 9.
File and I/O Management then follow in Part III, Chapters 10 and 11. The last part of the
book, Part IV (Chapters 12 and 13), studies protection and security, two related issues
with important social and technical components.

CONCEPTS, TERMS, AND ABBREVIATIONS

The following concepts have been introduced in this chapter. Test yourself by defining
and discussing each keyword or phrase.

<table>
<thead>
<tr>
<th>Batch processing</th>
<th>Multiprogramming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bootstrap</td>
<td>Network</td>
</tr>
<tr>
<td>Channel</td>
<td>Operating system (OS)</td>
</tr>
<tr>
<td>Client-server</td>
<td>Personal computer (PC)</td>
</tr>
<tr>
<td>Central processing unit (CPU)</td>
<td>Real-time system</td>
</tr>
<tr>
<td>Distributed system</td>
<td>Shell</td>
</tr>
<tr>
<td>Execution mode</td>
<td>Spooling</td>
</tr>
<tr>
<td>Graphics user interface (GUI)</td>
<td>Time-sharing</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Trap</td>
</tr>
<tr>
<td>Input-output (I/O)</td>
<td>Virtual machine</td>
</tr>
<tr>
<td>Kernel</td>
<td>Wide area network (WAN)</td>
</tr>
<tr>
<td>Local area network (LAN)</td>
<td></td>
</tr>
</tbody>
</table>
EXERCISES

1. Give examples of semantic gaps that must be bridged by software, including the OSs at the following levels: CPU (instruction) level, main memory level, secondary storage level, I/O device level.
2. What do abstraction and virtualization have in common? How are they different? Give an example of each.
3. What do traps and interrupts have in common? How are they different? Give an example of each.
4. Display a list of all commands supported by your UNIX system. Estimate what fraction of these are related to file processing. (Hint: Executable commands usually are in the system directory `/bin`.)
5. Compare the convenience and expressiveness of the online help provided by a text-based interface, e.g., using the `man` command of UNIX, with the online help of a graphics-based interface, e.g., the Help button of the Windows OS. Assume you are not very familiar with these systems and attempt to find out the following:
   (a) How to find all files containing the strings “unix,” “UNIX,” but not “Unix.”
   (b) How to make a file read-only for other system users.
   (c) How to open and display a file on the screen (a) from the keyboard and (b) from within a running program (application).
   (d) How to format a floppy disk.
   (e) How much disk space is being used by your files.
6. Determine how many key strokes, mouse clicks, and mouse moves are required for each of the following tasks using (a) the UNIX shell, (b) the Windows GUI:
   (a) Move a file from one directory (folder) to another
   (b) Delete a file
   (c) Locate and display a file
   (d) Create a new directory
   (e) Start a file editor
7. Simplify the shell script on page 20 such that it always displays the number of users currently on the system. Do you still need to use the variable `COUNT`?
8. What do multiprogramming and time-sharing have in common? How are they different? Is multiprogramming possible without interrupts? Is time-sharing possible without interrupts?