Operating system

Introduction

An operating system is a program that manages the computer hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware. A computer system can be divided roughly into four components: Hardware, Operating system, Application programs, and Users.

The hardware, the central processing unit (CPU), the memory, and the input/output (I/O) devices provide the basic computing resources for the system. The application programs such as word processors, spreadsheets, compilers, and web browsers define the ways in which these resources are used to solve users' computing problems. The operating system controls and coordinates the use of the hardware among the various application programs for the various users. We can also view a computer system as consisting of hardware, software, and data. The operating system provides the means for proper use of these resources in the operation of the computer system.

User View

The user's view of the computer varies according to the interface being used. Most computer users sit in front of a PC, consisting of a monitor, keyboard, mouse, and system unit. Such a system is designed for one user to monopolize its resources. The goal is to maximize the work (or play) that the user is performing. In this case, the operating system is designed mostly for ease of use.
System View
From the computer's point of view, the operating system is the program most intimately involved with the hardware. In this context, we can view an operating system as a resource allocator. A computer system has many resources that may be required to solve a problem: CPU time, memory space, file-storage space, I/O devices, and so on. The operating system acts as the manager of these resources. Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them to specific programs and users so that it can operate the computer system efficiently and fairly.

Operating-System Operations
As mentioned earlier, modern operating systems are interrupt driven. If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt or a trap. A trap (or an exception) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed. The interrupt-driven nature of an operating system defines that system's general structure. For each type of interrupt, separate segments of code in the operating system determine what action should be taken. An interrupt service routine is provided that is responsible for dealing with the interrupt. Since the operating system and the users share the hardware and software resources of the computer system, we need to make sure that an error in a user program could cause problems only for the one program that was running. With sharing, many processes could be adversely affected by a bug in one program. For example, if a process gets stuck in an infinite loop, this loop could prevent the correct operation of many other processes. More subtle errors can occur in a multiprogramming system, where one erroneous program might modify another program, the data of another program, or even the operating system itself. Without protection against these sorts of errors either the computer must execute only one process at a time or all output must be suspect. A properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs to execute incorrectly.

Dual-Mode Operation
In order to ensure the proper execution of the operating system, we must be able to distinguish between the execution of operating-system code and user defined code. The approach taken by most computer systems is to provide hardware support that allows us to differentiate among various modes of execution. At the very least, we need two separate modes of operation: user mode and kernel mode (also called supervisor mode, system mode, or privileged mode). A bit, called the mode bit, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). With the mode bit, we are able to distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user. When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), it must transition from user to kernel mode to fulfill the request. This architectural enhancement is useful for many other aspects of system operation as well. At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap
or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program. The dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another.

**Timer**

We must ensure that the operating system maintains control over the CPU. We must prevent a user program from getting stuck in an infinite loop or not calling system services and never returning control to the operating system. To accomplish this goal, we can use a timer. A timer can be set to interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second). A variable timer is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs. For instance, a 10-bit counter with a 1-millisecond clock allows interrupts at intervals from 1 millisecond to 1,024 milliseconds, in steps of 1 millisecond. Before turning over control to the user, the operating system ensures that the timer is set to interrupt. If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or may give the program more time. Clearly, instructions that modify the content of the timer are privileged.

**Operating-System Structure**

An operating system provides the environment within which programs are executed. One of the most important aspects of operating systems is the ability to multiprogramming. A single user cannot, in general, keep either the CPU or the I/O devices busy at all times. **Multiprogramming** increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute. The idea is as follows: The operating system keeps several jobs in memory simultaneously. This set of jobs can be a subset of the jobs kept in the job pool which contains all jobs that enter the system since the number of jobs that can be kept simultaneously in memory is usually smaller than the number of jobs that can be kept in the job pool. The operating system picks and begins to execute one of the jobs in memory. Eventually, the job may have to wait for some task, such as an I/O operation, to complete. In a non-multiprogramming system, the CPU would sit idle. In a multiprogramming system, the operating system simply switches to, and executes, another job. When that job needs to wait, the CPU is switched to another job, and so on. As long as at least one job needs to execute, the CPU is never idle.

```
<table>
<thead>
<tr>
<th>0</th>
<th>Operating System</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Job 1</td>
</tr>
<tr>
<td></td>
<td>Job 2</td>
</tr>
<tr>
<td></td>
<td>Job 3</td>
</tr>
<tr>
<td>512 M</td>
<td>Job 4</td>
</tr>
</tbody>
</table>
```

Memory layout for multiprogramming system
**Time sharing** (or **multitasking**) is a logical extension of multiprogramming. In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running. A time-shared operating system allows many users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user. As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to his use, even though it is being shared among many users. A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer. Each user has at least one separate program in memory.

**Simple Structure**

Many commercial systems do not have well-defined structures. Frequently, such operating systems started as small, simple, and limited systems and then grew beyond their original scope. MS-DOS is an example of such a system. It was originally designed and implemented by a few people who had no idea that it would become so popular. It was written to provide the most functionality in the least space, so it was not divided into modules carefully.

**MS-DOS layer structure**

**Layered Approach**

With proper hardware support, operating systems can be broken into pieces that are smaller and more appropriate than those allowed by the original MS-DOS or UNIX systems. The operating system can then retain much greater control over the computer and over the applications that make use of that computer. Implementers have more freedom in changing the inner workings of the system and in creating modular operating systems. Under the top-down approach, the overall functionality and features are determined and are separated into components. Information hiding
is also important, because it leaves programmers free to implement the low-level routines as they see fit, provided that the external interface of the routine stays unchanged and that the routine itself performs the advertised task. A system can be made modular in many ways. One method is the **layered approach**, in which the operating system is broken up into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface. This layering structure is depicted in Figure below.

![Layered approach diagram](image)

**Operating-System Services**

An operating system provides an environment for the execution of programs. It provides certain services to programs and to the users of those programs. The specific services provided, of course, differ from one operating system to another, but we can identify common classes. These operating-system services are provided for the convenience of the programmer, to make the programming task easier.

- **User interface.** Almost all operating systems have a user interface (UI). This interface can take several forms. One is a **command-line interface (CLI)**, which uses text commands and a method for entering them (say, a program to allow entering and editing of commands). Another is a **batch interface**, in which commands and directives to control those commands are entered into files, and those files are executed. Most commonly/ a **graphical user interface (GUI)** is used. Here, the interface is a window system with a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text. Some systems provide two or all three of these variations.

- **Program execution.** The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).
• **I/O operations.** A running program may require I/O, which may involve a file or an I/O device. For specific devices, special functions may be desired (such as recording to a CD or DVD drive or blanking a CRT screen). For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.

• **File-system manipulation.** The file system is of particular interest. Obviously, programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information. Finally, some programs include permissions management to allow or deny access to files or directories based on file ownership.

• **Communications.** There are many circumstances in which one process needs to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a computer network. Communications may be implemented via *shared memory* or through *message passing*, in which packets of information are moved between processes by the operating system.

• **Error detection.** The operating system needs to be constantly aware of possible errors. Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a parity error on tape, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow, an attempt to access an illegal memory location, or a too-great use of CPU time). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing. Debugging facilities can greatly enhance the user's and programmer's abilities to use the system efficiently.

• **Resource allocation.** When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. Many different types of resources are managed by the operating system. Some (such as CPU cycles, main memory, and file storage) may have special allocation code, whereas others (such as I/O devices) may have much more general request and release code. For instance, in determining how best to use the CPU, operating systems have CPU-scheduling routines that take into account the speed of the CPU, the jobs that must be executed, the number of registers available, and other factors. There may also be routines to allocate printers, modems, USB storage drives, and other peripheral devices.

• **Accounting.** We want to keep track of which users use how much and what kinds of computer resources. This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics. Usage statistics may be a valuable tool for researchers who wish to reconfigure the system to improve computing services.

• **Protection and security.** The owners of information stored in a multiuser or networked computer system may want to control use of that information. When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the operating system itself. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important. Such security starts with requiring each user to authenticate himself or herself to the system, usually by means of a password, to gain access to system resources. It extends to defending external I/O devices,
including modems and network adapters, from invalid access attempts and to recording all such
connections for detection of break-ins. If a system is to be protected and secure, precautions must
be instituted throughout it. A chain is only as strong as its weakest link.

System Calls
System calls provide an interface to the services made available by an operating system. These
calls are generally available as routines, although certain low-level tasks (for example, tasks
where hardware must be accessed directly), may need to be written using assembly-language
instructions.

writing a simple program to read data from one file and copy them to another file. The first input
that the program will need is the names of the two files: the input file and the output file. These
names can be specified in many ways, depending on the operating-system design. One approach
is for the program to ask the user for the names of the two files. In an interactive system, this
approach will require a sequence of system calls, first to write a prompting message on the
screen and then to read from the keyboard the characters that define the two files. On mouse-
based and icon-based systems, a menu of file names is usually displayed in a window. The user
can then use the mouse to select the source name, and a window can be opened for the
destination name to be specified.

This sequence requires many I/O system calls. Once the two file names are obtained, the
program must open the input file and create the output file. Each of these operations requires
another system call. There are also possible error conditions for each operation. When the
program tries to open the input file, it may find that there is no file of that name or that the file is
protected against access. In these cases, the program should print a message on the console
(another sequence of system calls) and then terminate abnormally (another system call). If the
input file exists, then we must create a new output file. We may find that there is already an
output file with the same name. This situation may cause the program to abort (a system call), or
we may delete the existing file (another system call) and create a new one (another system call).
Another option, in an interactive system, is to ask the user (via a sequence of system calls to
output the prompting message and to read the response from the terminal) whether to replace the
existing file or to abort the program.

Now that both files are set up, we enter a loop that reads from the input file (a system call) and
writes to the output file (another system call). Each read and write must return status information
regarding various possible error conditions. On input, the program may find that the end of the
file has been reached or that there was a hardware failure in the read (such as a parity error).
The write operation may encounter various errors, depending on the output device (no more disk
space, printer out of paper, and so on). Finally, after the entire file is copied, the program may
close both files (another system call), write a message to the console or window (more system
calls), and finally terminate normally (the final system call). As we can see, even simple
programs may make heavy use of the operating system. Frequently, systems execute thousands
of system calls per second. This system call sequence is shown in Figure below.
The types of System Calls: System calls can be grouped roughly into five major categories: process control, file manipulation, device manipulation, information maintenance, and communications.

System Programs
Another aspect of a modern system is the collection of system programs. System programs provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls; others are considerably more complex. They can be divided into these categories:

- **File management.** These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.

- **Status information.** Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Others are more complex, providing detailed performance, logging, and debugging information. Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI. Some systems also support a registry, which is used to store and retrieve configuration information.
• **File modification.** Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.

• **Programming-language support.** Compilers, assemblers, debuggers and interpreters for common programming languages (such as C, C++, Java, Visual Basic, and PERL) are often provided to the user with the operating system.

• **Program loading and execution.** Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or machine language are needed as well.

• **Communications.** These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another's screens, to browse web pages, to send electronic-mail messages, to log in remotely, or to transfer files from one machine to another.

**System Boot**

The procedure of starting a computer by loading the kernel is known as **booting** the system. On most computer systems, a small piece of code known as the **bootstrap program** or **bootstrap loader** locates the kernel, loads it into main memory, and starts its execution. Some computer systems, such as PCs, use a two-step process in which a simple bootstrap loader fetches a more complex boot program from disk, which in turn loads the kernel. When a CPU receives a reset event for instance, when it is powered up or rebooted, the instruction register is loaded with a predefined memory location, and execution starts there. At that location is the initial bootstrap program. This program is in the form of **read-only memory (ROM)**, because the RAM is in an unknown state at system startup. ROM is convenient because it needs no initialization and cannot be infected by a computer virus. The bootstrap program can perform a variety of tasks. Usually, one task is to run diagnostics to determine the state of the machine. If the diagnostics pass, the program can continue with the booting steps. It can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory. Sooner or later, it starts the operating system. Some systems such as cellular phones, PDAs, and game consoles store the entire operating system in ROM. Storing the operating system in ROM is suitable for small operating systems, simple supporting hardware, and rugged operation. A problem with this approach is that changing the bootstrap code requires changing the ROM hardware chips. Some systems resolve this problem by using **erasable programmable read-only memory (EPROM)**, which is read only except when explicitly given a command to become writable. All forms of ROM are also known as **firmware**, since their characteristics fall somewhere between those of hardware and those of software. A problem with firmware in general is that executing code there is slower than executing code in RAM. Some systems store the operating system in firmware and copy it to RAM for fast execution. A final issue with firmware is that it is relatively expensive, so usually only small amounts are available. For large operating systems (including most general-purpose operating systems like Windows, Mac OS X, and UNIX) or for systems that change frequently, the bootstrap loader are stored in firmware and the operating system is on disk. In this case, the bootstrap runs diagnostics and has a bit of code
that can read a single block at a fixed location (block zero) from disk into memory and execute the code from that **boot block**. The program stored in the boot block may be sophisticated enough to load the entire operating system into memory and begin its execution. More typically, it is simple code (as it fits in a single disk block) and only knows the address on disk and length of the remainder of the bootstrap program. All of the disk-bound bootstrap, and the operating system itself, can be easily changed by writing new versions to disk. A disk that has a boot partition is called a **boot disk** or system disk. Now that the full bootstrap program has been loaded, it can traverse the file system to find the operating system kernel, load it into memory, and start its execution. It is only at this point that the system is said to be **running**.

**Processes**

Early computer systems allowed only one program to be executed at a time. This program had complete control of the system and had access to all the system's resources. In contrast, current-day computer systems allow multiple programs to be loaded into memory and executed concurrently. This evolution required firmer control and more compartmentalization of the various programs; and these needs resulted in the notion of a process, which is a program in execution. A process is the unit of work in a modern time-sharing system. The more complex the operating system is, the more it is expected to do on behalf of its users. Although its main concern is the execution of user programs, it also needs to take care of various system tasks that are better left outside the kernel itself. A system therefore consists of a collection of processes: operating system processes executing system code and user processes executing user code. Potentially, all these processes can execute concurrently, with the CPU (or CPUs) multiplexed among them. By switching the CPU between processes, the operating system can make the computer more productive a process is a program in execution. A process is more than the program code, which is sometimes known as the **text section**. It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers. A process generally also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables), and a **data section**, which contains global variables. A process may also include a **heap**, which is memory that is dynamically allocated during process run time. The structure of a process in memory is shown in figure below.

![Process in memory diagram](image-url)
Process State
As a process executes, it changes state. The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states:

- **New.** The process is being created.
- **Running.** Instructions are being executed.
- **Waiting.** The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready.** The process is waiting to be assigned to a processor.
- **Terminated.** The process has finished execution.

These names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems also more finely delineate process states. It is important to realize that only one process can be running on any processor at any instant. Many processes may be ready and limiting, however. The state diagram corresponding to these states is presented in Figure below.

![Diagram of process state.](image_url)

**Process Control Block**
Each process is represented in the operating system by a process control block (PCB) also called a task control block. A PCB is shown in Figure below. It contains many pieces of information associated with a specific process, including these:
• **Process state.** The state may be new, ready, running, waiting, halted, and so on.

• **Program counter.** The counter indicates the address of the next instruction to be executed for this process.

• **CPU registers.** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

• **CPU-scheduling information.** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

• **Memory-management information.** This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.

• **Accounting information.** This information includes the amount of CPU and real time used, time limits, account members, job or process numbers, and so on.

• **I/O status information.** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

In brief, the PCB simply serves as the repository for any information that may vary from process to process.

<table>
<thead>
<tr>
<th>Process state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process number</td>
</tr>
<tr>
<td>Process counter</td>
</tr>
<tr>
<td>Registers</td>
</tr>
<tr>
<td>Memory limits</td>
</tr>
<tr>
<td>List of open files</td>
</tr>
<tr>
<td>…</td>
</tr>
</tbody>
</table>

**Process controls block (PCB).**