Introduction to How Operating Systems Work

If you have a computer, then you have heard about operating systems. Any desktop or laptop PC that you buy normally comes pre-loaded with *Windows XP*. Macintosh computers come pre-loaded with *OS X*. Many corporate servers use the *Linux* or *UNIX* operating systems. The operating system (OS) is the first thing loaded onto the computer -- without the operating system, a computer is useless.

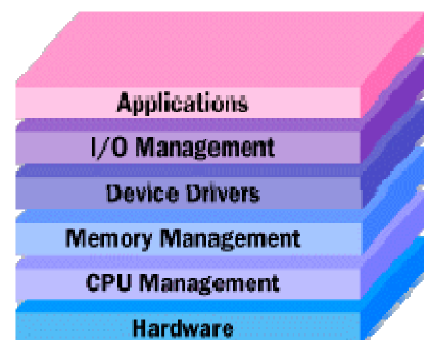
More recently, operating systems have started to pop up in smaller computers as well. If you like to tinker with electronic devices, you are probably pleased that operating systems can now be found on many of the devices we use every day, from cell phones to wireless access points. The computers used in these little devices have gotten so powerful that they can now actually run an operating system and applications. The computer in a typical modern cell phone is now more powerful than a desktop computer from 20 years ago, so this progression makes sense and is a natural development. In any device that has an operating system, there's usually a way to make changes to how the device works. This is far from a happy accident; one of the reasons operating systems are made out of portable code rather than permanent physical circuits is so that they can be changed or modified without having to scrap the whole device.

For a desktop computer user, this means you can add a new security update, system patch, new application or often even a new operating system entirely rather than junk your computer and start again with a new one when you need to make a change. As long as you understand how an operating system works and know how to get at it, you can in many cases change some of the ways it behaves. And, it's as true of your cell phone as it is of your computer.

The purpose of an operating system is to organize and control hardware and software so that the device it lives in behaves in a flexible but predictable way. In this article, we'll tell you what a piece of software must do to be called an operating system, show you how the operating system in your desktop computer works and give you some examples of how to take control of the other operating systems around you.

The Bare Bones

Not all computers have operating systems. The computer that controls the microwave oven in your kitchen, for example, doesn't need an operating system. It has one set of tasks to perform, very straightforward input to expect (a numbered keypad and a few pre-set buttons) and simple, never-changing hardware to control. For a computer like this, an operating system would be unnecessary baggage, driving up the development and manufacturing costs significantly and adding complexity where none is required. Instead, the computer in a microwave oven simply runs a single hard-wired program all the time.



For other devices, an operating system creates the ability to:

- serve a variety of purposes
- interact with users in more complicated ways
- keep up with needs that change over time

All desktop computers have operating systems. The most common are the Windows family of operating systems developed by Microsoft, the Macintosh operating systems developed by Apple and the UNIX family of operating systems (which have been developed by a whole history of individuals, corporations and collaborators). There are hundreds of other operating systems available for special-purpose applications, including specializations for mainframes, robotics, manufacturing, real-time control systems and so on.

What Does It Do?

At the simplest level, an operating system does two things:

1. It manages the hardware and software resources of the system. In a desktop computer, these resources include such things as the processor, memory, disk space, etc. (On a cell phone, they include the keypad, the screen, the address book, the phone dialer, the battery and the network connection.)
2. It provides a stable, consistent way for applications to deal with the hardware without having to know all the details of the hardware.

The first task, managing the hardware and software resources, is very important, as various programs and input methods compete for the attention of the **central processing unit** (CPU) and demand memory, storage and input/output (I/O) bandwidth for their own purposes. In this capacity, the operating system plays the role of the good parent, making sure that each application gets the necessary resources while playing nicely with all the other applications, as well as husbanding the limited capacity of the system to the greatest good of all the users and applications.

The second task, providing a consistent application interface, is especially important if there is to be more than one of a particular type of computer using the operating system, or if the hardware making up the computer is ever open to change. A consistent **application program interface** (API) allows a software developer to write an application on one computer and have a high level of confidence that it will run on another computer of the same type, even if the amount of memory or the quantity of storage is different on the two machines.

Even if a particular computer is unique, an operating system can ensure that applications continue to run when hardware upgrades and updates occur. This is because the operating system and not the application is charged with managing the hardware and the distribution of its resources. One of the challenges facing developers is keeping their operating systems flexible enough to run hardware from the thousands of vendors manufacturing computer equipment. Today's systems can accommodate thousands of different printers, disk drives and special peripherals in any possible combination.

What Kinds Are There?

Within the broad family of operating systems, there are generally four types, categorized based on the types of computers they control and the sort of applications they support. The broad categories are:

- **Real-time operating system** (RTOS) - Real-time operating systems are used to control machinery, scientific instruments and industrial systems. An RTOS typically has very little user-interface capability, and no end-user utilities, since the system will be a "sealed box" when delivered for use. A very important part of an RTOS is managing the resources of the computer so that a particular operation executes in precisely the same amount of time every time it occurs. In a complex machine, having a part move more quickly just because system resources are available may be just as catastrophic as having it not move at all because the system is busy.
- **Single-user, single task** - As the name implies, this operating system is designed to manage the computer so that one user can effectively do one thing at a time. The *Palm OS* for Palm handheld computers is a good example of a modern single-user, single-task operating system.
- **Single-user, multi-tasking** - This is the type of operating system most people use on their desktop and laptop computers today. Microsoft's *Windows* and Apple's *MacOS* platforms are both examples of operating systems that will let a single user have several programs in operation at the same time. For example, it's entirely possible for a Windows user to be writing a note in a word processor while downloading a file from the Internet while printing the text of an e-mail message.
- **Multi-user** - A multi-user operating system allows many different users to take advantage of the computer's resources simultaneously. The operating system must make sure that the requirements of the various users are balanced, and that each of the programs they are using has sufficient and separate resources so that a problem with one user doesn't affect the entire community of users. Unix, VMS and mainframe operating systems, such as *MVS*, are examples of multi-user operating systems.

It's important to differentiate here between multi-user operating systems and single-user operating systems that support networking. *Windows 2000* and *Novell Netware* can each support hundreds or thousands of networked users, but the operating systems themselves aren't true multi-user operating systems. The **system administrator** is the only "user" for *Windows 2000* or *Netware*. The network support and all of the remote user logins the network enables are, in the overall plan of the operating system, a program being run by the administrative user.

With the different types of operating systems in mind, it's time to look at the basic functions provided by an operating system.

Wake-Up Call

When you turn on the power to a computer, the first program that runs is usually a set of instructions kept in the computer's read-only memory (ROM). This code examines the system hardware to make sure everything is functioning properly. This **power-on self test** (POST)

checks the CPU, memory, and basic input-output systems (BIOS) for errors and stores the result in a special memory location. Once the POST has successfully completed, the software loaded in ROM (sometimes called the BIOS or **firmware**) will begin to activate the computer's disk drives. In most modern computers, when the computer activates the hard disk drive, it finds the first piece of the operating system: the **bootstrap loader**.

The bootstrap loader is a small program that has a single function: It loads the operating system into memory and allows it to begin operation. In the most basic form, the bootstrap loader sets up the small driver programs that interface with and control the various hardware subsystems of the computer. It sets up the divisions of memory that hold the operating system, user information and applications. It establishes the data structures that will hold the myriad signals, flags and semaphores that are used to communicate within and between the subsystems and applications of the computer. Then it turns control of the computer over to the operating system.

The operating system's tasks, in the most general sense, fall into six categories:

- Processor management
- Memory management
- Device management
- Storage management
- Application interface
- User interface

While there are some who argue that an operating system should do more than these six tasks, and some operating-system vendors do build many more utility programs and auxiliary functions into their operating systems, these six tasks define the core of nearly all operating systems. Let's look at the tools the operating system uses to perform each of these functions.

Processor Management

The heart of managing the processor comes down to two related issues:

- Ensuring that each process and application receives enough of the processor's time to function properly.
- Using as many processor cycles for real work as is possible.

The basic unit of software that the operating system deals with in scheduling the work done by the processor is either a **process** or a **thread**, depending on the operating system.

It's tempting to think of a process as an application, but that gives an incomplete picture of how processes relate to the operating system and hardware. The application you see (word processor or spreadsheet or game) is, indeed, a process, but that application may cause several other processes to begin, for tasks like communications with other devices or other computers. There are also numerous processes that run without giving you direct evidence that they ever exist. For example, *Windows XP* and *UNIX* can have dozens of background processes running to handle the network, memory management, disk management, virus checking and so on.

A process, then, is software that performs some action and can be controlled -- by a user, by other applications or by the operating system.

It is processes, rather than applications, that the operating system controls and schedules for execution by the CPU. In a single-tasking system, the schedule is straightforward. The operating system allows the application to begin running, suspending the execution only long enough to deal with **interrupts** and user input.

Interrupts are special signals sent by hardware or software to the CPU. It's as if some part of the computer suddenly raised its hand to ask for the CPU's attention in a lively meeting. Sometimes the operating system will schedule the priority of processes so that interrupts are **masked** -- that is, the operating system will ignore the interrupts from some sources so that a particular job can be finished as quickly as possible. There are some interrupts (such as those from error conditions or problems with memory) that are so important that they can't be ignored. These **non-maskable interrupts** (NMIs) must be dealt with immediately, regardless of the other tasks at hand.

While interrupts add some complication to the execution of processes in a single-tasking system, the job of the operating system becomes much more complicated in a multi-tasking system. Now, the operating system must arrange the execution of applications so that you believe that there are several things happening at once. This is complicated because the CPU can only do one thing at a time. In order to give the appearance of lots of things happening at the same time, the operating system has to switch between different processes thousands of times a second. Here's how it happens:

- A process occupies a certain amount of RAM. It also makes use of registers, stacks and queues within the CPU and operating-system memory space.
- When two processes are multi-tasking, the operating system allots a certain number of CPU execution cycles to one program.
- After that number of cycles, the operating system makes copies of all the registers, stacks and queues used by the processes, and notes the point at which the process paused in its execution.
- It then loads all the registers, stacks and queues used by the second process and allows it a certain number of CPU cycles.
- When those are complete, it makes copies of all the registers, stacks and queues used by the second program, and loads the first program.

All of the information needed to keep track of a process when switching is kept in a data package called a process control block. The process control block typically contains:

- An ID number that identifies the process
- Pointers to the locations in the program and its data where processing last occurred
- Register contents
- States of various flags and switches
- Pointers to the upper and lower bounds of the memory required for the process
- A list of files opened by the process
- The priority of the process

- The status of all I/O devices needed by the process

Each process has a status associated with it. Many processes consume no CPU time until they get some sort of input. For example, a process might be waiting on a keystroke from the user. While it is waiting for the keystroke, it uses no CPU time. While it is waiting, it is "**suspended**". When the keystroke arrives, the OS changes its status. When the status of the process changes, from pending to active, for example, or from suspended to running, the information in the process control block must be used like the data in any other program to direct execution of the task-switching portion of the operating system.

This process swapping happens without direct user interference, and each process gets enough CPU cycles to accomplish its task in a reasonable amount of time. Trouble can come, though, if the user tries to have too many processes functioning at the same time. The operating system itself requires some CPU cycles to perform the saving and swapping of all the registers, queues and stacks of the application processes. If enough processes are started, and if the operating system hasn't been carefully designed, the system can begin to use the vast majority of its available CPU cycles to swap between processes rather than run processes. When this happens, it's called **thrashing**, and it usually requires some sort of direct user intervention to stop processes and bring order back to the system.

One way that operating-system designers reduce the chance of thrashing is by reducing the need for new processes to perform various tasks. Some operating systems allow for a "**process-lite**," called a **thread**, that can deal with all the CPU-intensive work of a normal process, but generally does not deal with the various types of I/O and does not establish structures requiring the extensive process control block of a regular process. A process may start many threads or other processes, but a thread cannot start a process.

So far, all the scheduling we've discussed has concerned a single CPU. In a system with two or more CPUs, the operating system must divide the workload among the CPUs, trying to balance the demands of the required processes with the available cycles on the different CPUs. **Asymmetric operating systems** use one CPU for their own needs and divide application processes among the remaining CPUs. **Symmetric operating systems** divide themselves among the various CPUs, balancing demand versus CPU availability even when the operating system itself is all that's running.

Even if the operating system is the only software with execution needs, the CPU is not the only resource to be scheduled. Memory management is the next crucial step in making sure that all processes run smoothly.

Memory Storage and Management

When an operating system manages the computer's memory, there are two broad tasks to be accomplished:

1. Each process must have enough memory in which to execute, and it can neither run into the memory space of another process nor be run into by another process.

2. The different types of memory in the system must be used properly so that each process can run most effectively.

The first task requires the operating system to set up memory boundaries for types of software and for individual applications.

As an example, let's look at an imaginary small system with 1 megabyte (1,000 kilobytes) of RAM. During the boot process, the operating system of our imaginary computer is designed to go to the top of available memory and then "back up" far enough to meet the needs of the operating system itself. Let's say that the operating system needs 300 kilobytes to run. Now, the operating system goes to the bottom of the pool of RAM and starts building up with the various driver software required to control the hardware subsystems of the computer. In our imaginary computer, the drivers take up 200 kilobytes. So after getting the operating system completely loaded, there are 500 kilobytes remaining for application processes.

When applications begin to be loaded into memory, they are loaded in block sizes determined by the operating system. If the block size is 2 kilobytes, then every process that is loaded will be given a chunk of memory that is a multiple of 2 kilobytes in size. Applications will be loaded in these fixed block sizes, with the blocks starting and ending on boundaries established by words of 4 or 8 bytes. These blocks and boundaries help to ensure that applications won't be loaded on top of one another's space by a poorly calculated bit or two. With that ensured, the larger question is what to do when the 500-kilobyte application space is filled.

In most computers, it's possible to add memory beyond the original capacity. For example, you might expand RAM from 1 to 2 megabytes. This works fine, but tends to be relatively expensive. It also ignores a fundamental fact of computing -- most of the information that an application stores in memory is not being used at any given moment. A processor can only access memory one location at a time, so the vast majority of RAM is unused at any moment. Since disk space is cheap compared to RAM, then moving information in RAM to hard disk can greatly expand RAM space at no cost. This technique is called **virtual memory management**.

Disk storage is only one of the memory types that must be managed by the operating system, and is the slowest. Ranked in order of speed, the types of memory in a computer system are:

- **High-speed cache** - This is fast, relatively small amounts of memory that are available to the CPU through the fastest connections. Cache controllers predict which pieces of data the CPU will need next and pull it from main memory into high-speed cache to speed up system performance.
- **Main memory** - This is the RAM that you see measured in megabytes when you buy a computer.
- **Secondary memory** - This is most often some sort of rotating magnetic storage that keeps applications and data available to be used, and serves as virtual RAM under the control of the operating system.

The operating system must balance the needs of the various processes with the availability of the different types of memory, moving data in blocks (called **pages**) between available memory as the schedule of processes dictates.

Device Management

The path between the operating system and virtually all hardware not on the computer's motherboard goes through a special program called a driver. Much of a driver's function is to be the translator between the electrical signals of the hardware subsystems and the high-level programming languages of the operating system and application programs. Drivers take data that the operating system has defined as a file and translate them into streams of bits placed in specific locations on storage devices, or a series of laser pulses in a printer.

Because there are such wide differences in the hardware controlled through drivers, there are differences in the way that the driver programs function, but most are run when the device is required, and function much the same as any other process. The operating system will frequently assign high-priority blocks to drivers so that the hardware resource can be released and readied for further use as quickly as possible.

One reason that drivers are separate from the operating system is so that new functions can be added to the driver -- and thus to the hardware subsystems -- without requiring the operating system itself to be modified, recompiled and redistributed. Through the development of new hardware device drivers, development often performed or paid for by the manufacturer of the subsystems rather than the publisher of the operating system, input/output capabilities of the overall system can be greatly enhanced.

Managing input and output is largely a matter of managing **queues** and **buffers**, special storage facilities that take a stream of bits from a device, perhaps a keyboard or a serial port, hold those bits, and release them to the CPU at a rate slow enough for the CPU to cope with. This function is especially important when a number of processes are running and taking up processor time. The operating system will instruct a buffer to continue taking input from the device, but to stop sending data to the CPU while the process using the input is suspended. Then, when the process needing input is made active once again, the operating system will command the buffer to send data. This process allows a keyboard or a modem to deal with external users or computers at a high speed even though there are times when the CPU can't use input from those sources.

Managing all the resources of the computer system is a large part of the operating system's function and, in the case of real-time operating systems, may be virtually all the functionality required. For other operating systems, though, providing a relatively simple, consistent way for applications and humans to use the power of the hardware is a crucial part of their reason for existing.

Interface to the World

Application Interface

Just as drivers provide a way for applications to make use of hardware subsystems without

having to know every detail of the hardware's operation, **application program interfaces** (APIs) let application programmers use functions of the computer and operating system without having to directly keep track of all the details in the CPU's operation. Let's look at the example of creating a hard disk file for holding data to see why this can be important.

A programmer writing an application to record data from a scientific instrument might want to allow the scientist to specify the name of the file created. The operating system might provide an API function named **MakeFile** for creating files. When writing the program, the programmer would insert a line that looks like this:

MakeFile [1, %Name, 2]
In this example, the instruction tells the operating system to create a file that will allow random access to its data (signified by the 1 -- the other option might be 0 for a serial file), will have a name typed in by the user (%Name) and will be a size that varies depending on how much data is stored in the file (signified by the 2 -- other options might be zero for a fixed size, and 1 for a file that grows as data is added but does not shrink when data is removed). Now, let's look at what the operating system does to turn the instruction into action.

The operating system sends a query to the disk drive to get the location of the first available free storage location.

With that information, the operating system creates an entry in the file system showing the beginning and ending locations of the file, the name of the file, the file type, whether the file has been archived, which users have permission to look at or modify the file, and the date and time of the file's creation.

The operating system writes information at the beginning of the file that identifies the file, sets up the type of access possible and includes other information that ties the file to the application. In all of this information, the queries to the disk drive and addresses of the beginning and ending point of the file are in formats heavily dependent on the manufacturer and model of the disk drive.

Because the programmer has written the program to use the API for disk storage, the programmer doesn't have to keep up with the instruction codes, data types and response codes for every possible hard disk and tape drive. The operating system, connected to drivers for the various hardware subsystems, deals with the changing details of the hardware -- the programmer must simply write code for the API and trust the operating system to do the rest.

APIs have become one of the most hotly contested areas of the computer industry in recent years. Companies realize that programmers using their API will ultimately translate this into the ability to control and profit from a particular part of the industry. This is one of the reasons that so many companies have been willing to provide applications like readers or viewers to the public at no charge. They know consumers will request that programs take advantage of the free readers, and application companies will be ready to pay royalties to allow their software to provide the functions requested by the consumers.

User Interface

Just as the API provides a consistent way for applications to use the resources of the computer system, a **user interface** (UI) brings structure to the interaction between a user and the computer. In the last decade, almost all development in user interfaces has been in the area of the **graphical user interface** (GUI), with two models, Apple's *Macintosh* and Microsoft's *Windows*, receiving most of the attention and gaining most of the market share. The popular, open-source Linux operating system also supports a graphical user interface.

There are other user interfaces, some graphical and some not, for other operating systems.

Unix, for example, has user interfaces called **shells** that present a user interface more flexible and powerful than the standard operating system text-based interface. Programs such as the *Korn Shell* and the *C Shell* are text-based interfaces that add important utilities, but their main purpose is to make it easier for the user to manipulate the functions of the operating system. There are also graphical user interfaces, such as *X-Windows* and *Gnome*, that make Unix and Linux more like Windows and Macintosh computers from the user's point of view.

It's important to remember that in all of these examples, the user interface is a program or set of programs that sits as a layer above the operating system itself. The same thing is true, with somewhat different mechanisms, of both Windows and Macintosh operating systems. The core operating-system functions -- the management of the computer system -- lie in the kernel of the operating system. The **display manager** is separate, though it may be tied tightly to the **kernel** beneath. The ties between the operating-system kernel and the user interface, utilities and other software define many of the differences in operating systems today, and will further define them in the future.

What's New

The Growing Importance of Networks

For desktop systems, access to a LAN or the Internet has become such an expected feature that in many ways it's hard to discuss an operating system without making reference to its connections to other computers and servers. Operating system developers have made the Internet the standard method for delivering crucial operating system updates and bug fixes. Although it is possible to receive these updates via CD, it is becoming increasingly less common. In fact, some entire operating systems themselves are only available through distribution over the Internet.

Further, a process called **NetBooting** has streamlined the capability to move the working operating system of a standard consumer desktop computer - kernel, user interface and all - off of the machine it controls. This was previously only possible for experienced power-users on multi-user platforms like UNIX and with a suite of specialized applications. NetBooting allows the operating system for one computer to be served over a network connection, by a remote computer connected anywhere in the network. One NetBoot server can serve operating systems to several dozen client computers simultaneously, and to the user sitting in front of each client computer the experience is just like they are using their familiar desktop operating system like *Windows* or *MacOS*.

Open Source

One question concerning the future of operating systems revolves around the ability of a particular philosophy of software distribution to create an operating system useable by corporations and consumers together.

Linux, the operating system created and distributed according to the principles of **open source**, has had a significant impact on the operating system in general. Most operating systems, drivers and utility programs are written by commercial organizations that distribute **executable versions** of their software -- versions that can't be studied or altered. Open source requires the distribution of original source materials that can be studied, altered and built upon, with the results once again freely distributed. In the desktop computer realm, this has led to the development and distribution of countless useful and cost-free applications like the image manipulation program GIMP and the popular web server Apache. In the consumer device realm, the use of Linux has paved the way for individual users to have greater control over how their devices behave.