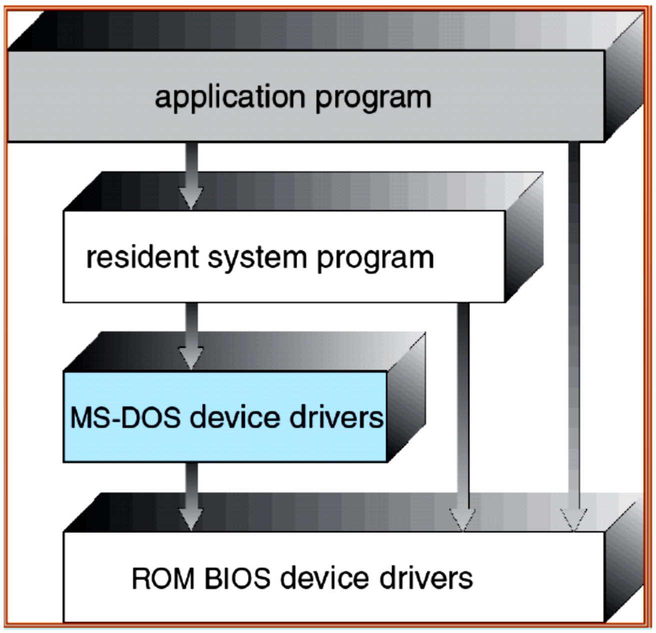
# Chapter 2
# *Operating-System Structures*

This chapter will discuss the following concepts:

2.1 Operating System Services
2.2 User Operating System Interface
2.3 System Calls
2.4 System Programs
2.5 Operating System Design and Implementation
2.6 Operating System Structure
2.7 System Boot



## 2.1 Operating System Services

An operating system provides certain services to **programs** and to the **users** of those programs.

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

- **User interface:** Almost all operating systems have a **user interface** (**UI**). This interface can take several forms. One is a **command-line interface** (**CLI**), which uses text commands. Another is a **batch interface**, in which 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. 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 **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. 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.
- **Communications:** sometimes 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.

- **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). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing.

Another set of operating-system functions exists not for helping the user but rather for ensuring the efficient operation of the **system** itself.

- **Resource allocation:** When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them.
- **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.
- **Protection and security: 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 him or herself to the system, usually by means of a password, to gain access to system resources.

## 2.2 User Operating System Interface

There are two fundamental approaches for users to interface with the operating system. One technique is to provide a **command-line interface** or **command interpreter**. The second approach allows the user to interface with the operating system via a **graphical user interface (GUI).**

### 2.2.1 Command Interpreter

Some operating systems include the command interpreter in the kernel. Others, such as Windows XP and UNIX, treat the command interpreter as a special program that is running when a job is initiated or when a user first logs on. On systems with multiple command interpreters to choose from, the interpreters are known as **shells**. For example, on UNIX and Linux systems, there are several different shells a user may choose from.

### 2.2.2 Graphical User Interfaces

A second strategy for interfacing with the operating system is through a user friendly **graphical user interface** (**GUI**). Rather than having users directly enter commands via a command-line interface, a GUI allows provides a mouse-based window-and-menu system as an interface. A GUI provides a desktop where the mouse is moved to position its pointer on images, or icons, on the screen (the desktop) that represent programs, files, directories—known as a folder—, and system functions.

Graphical user interfaces first appeared due in part to research taking place in the early 1970s at **Xerox PARC** research facility.

## 2.3 System Calls

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

Let's first use an example to illustrate how system calls are used: 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.

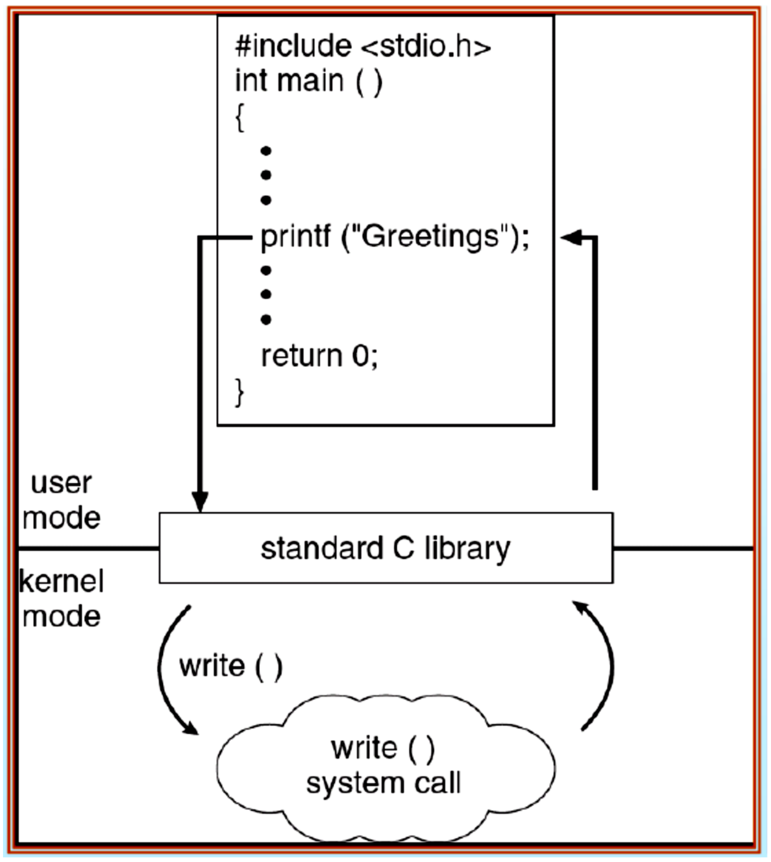
Once the two file names are obtained, the program must open the input file and create the output file. Each of these operations requires system call. Now that both files are set up, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call). Finally, 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.

Most programmers never see this level of detail, however. Typically, application developers design programs according to an **application programming interface** (**API**). The API specifies a set of functions that are available to an application programmer

Three of the most common APIs available to application programmers are the **Win32 API** for Windows systems, the **POSIX API** for POSIX-based systems (which includes virtually all versions of UNIX, Linux, and Mac OS X), and the **Java API** for designing programs that run on the Java virtual machine.

## 2.3.1 Example

C program invoking **printf( )** library call, which calls write( ) system call. As shown in figure 2.1.



```
#include <stdio.h>
int main ( )
{
   •
   •
   •
   printf ("Greetings");
   •
   •
   •
   return 0;
}
```

Figure 2.1 system call example

## 2.3.2 Types of System Calls

System calls can be grouped roughly into five major categories:

- **Process control**
- **file manipulation**
- **device manipulation**
- **information maintenance**
- **Communications**

The following table summarizes the types of system calls normally provided by an operating system.

• **Process control**
o end, abort
o load, execute
*o* create process, terminate process
*o* get process attributes, set process attributes
o wait for time
*o* wait event, signal event
o allocate and free memory

• **File management**
o create file, delete file
o open, close
o read, write, reposition
o get file attributes, set file attributes

• **Device management**
o request device, release device
o read, write, reposition
o get device attributes, set device attributes
o logically attach or detach devices

• **Information maintenance**
o get time or date, set time or date
o get system data, set system data
o get process, file, or device attributes
o set process, file, or device attributes

• **Communications**
o create, delete communication connection
o send, receive messages
o transfer status information
o attach or detach remote devices

## 2.4 System Programs

Another aspect of a modern system is the collection of system programs. Recall Figure 1.1, there are the system programs and the application programs. Some of them are simply user interfaces to system calls. They can be divided into these categories:

- **File management:** These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.
- **Status information:** Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information.
- **File modification:** Several text editors may be available to create and modify the content of files stored on disk or other storage devices.
- **Programming-language support: Compilers, assemblers, debuggers** and **interpreters** for common programming languages (such as C, C++, Java, 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**, **re-locatable loaders**, **linkage editors**, and **overlay loaders**.
- **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.

## 2.5 Operating-System Design and Implementation

The first problem in designing a system is to define **goals** and **specifications**. At the highest level, the design of the system will be affected by the choice of **hardware** and the **type of system**.

Once an operating system is designed, it must be implemented. Traditionally, operating systems have been written in assembly language. Now, however, they are most commonly written in **higher-level languages** such as **C** or **C++.** The first system that was not written in assembly language was probably the **Master Control Program** (**MCP**). MCP was written in a variant of **ALGOL**.

The Linux and Windows XP operating systems are written mostly in **C**, although there are some small sections of assembly code for device drivers and for saving and restoring the state of registers.

The advantages of using a higher-level language for implementing operating systems are: The code can be written **faster**, and is **easier** to understand and debug. Finally, an operating system is far **easier to port**—to move to some other hardware— if it is written in a higher-level language. For example, **MS-DOS** was written in **Intel 8088 assembly language**. Consequently, it is available on only the Intel family of CPUs. The **Linux operating system**, in contrast, is written mostly in **C** and is available on a number of different CPUs, including Intel 80X86, Motorola 680X0, SPARC, and MIPS RXOO0.

The only possible disadvantages of implementing an operating system in a higher-level language are **reduced speed** and **increased storage requirements**. This, however, is no longer a major issue in today's systems.

## 2.6 Operating-System Structure

A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily. A common approach is to partition the task into small components rather than have one monolithic system.

### 2.6.1 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. Figure 2.2 shows its structure.

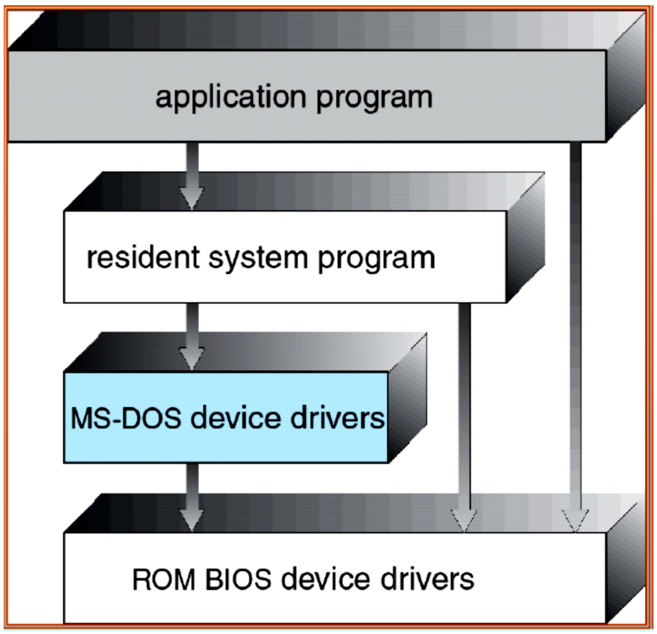


Figure 2.2 MS-DOS layer structures

Another example of limited structuring is the original UNIX operating system. UNIX is another system that initially was limited by hardware functionality. It consists of two separable parts: the **kernel** and the **system programs**. As shown in Figure 2.3.

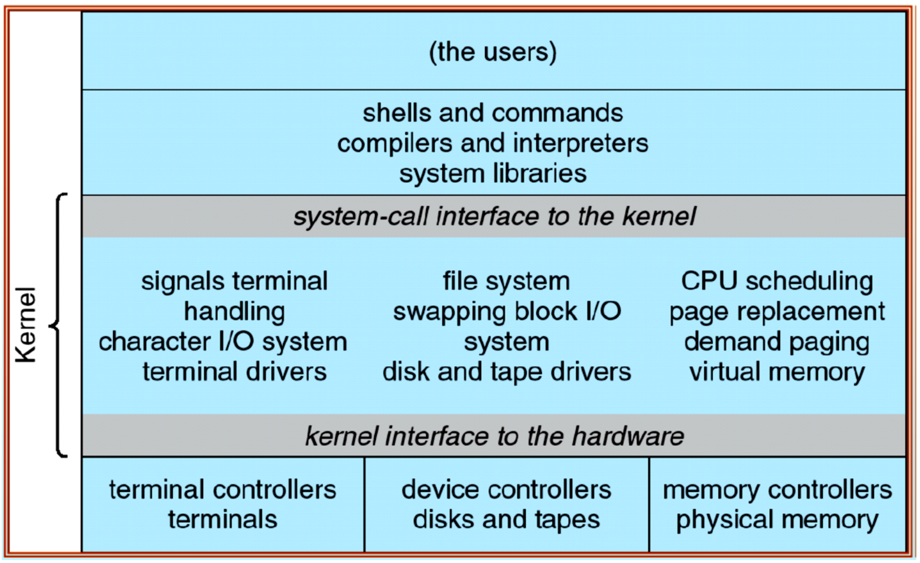


Figure 2.3 UNIX system structure.

## 2.6.2 Layered Approach

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. As shown in Figure 2.4.

The main advantage of the layered approach is **simplicity** of construction and debugging.

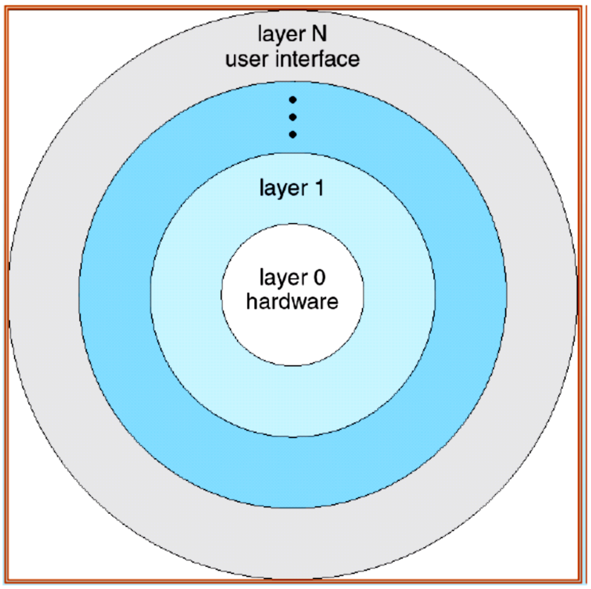


Figure 2.4 A layered operating system.

## 2.6.3 Microkernels

This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs. The result is a **smaller kernel**. One benefit of the microkernel approach is **ease of extending** the operating system. The resulting operating system is **easier to port** from one hardware design to another. The microkernel also provides more **security and reliability**, since most services are running as user processes. If a service fails, the rest of the operating system remains untouched.

## 2.6.4 Modules

Perhaps the best current methodology for operating-system design involves using object oriented programming techniques to create a modular kernel. Here, the kernel has a set of core components and dynamically links in additional services either during boot time or during run time. Such a strategy uses dynamically loadable modules and is common in modern implementations of UNIX, such as **Solaris** and **Linux**. For example, the Solaris operating system structure, shown in Figure 2.5.

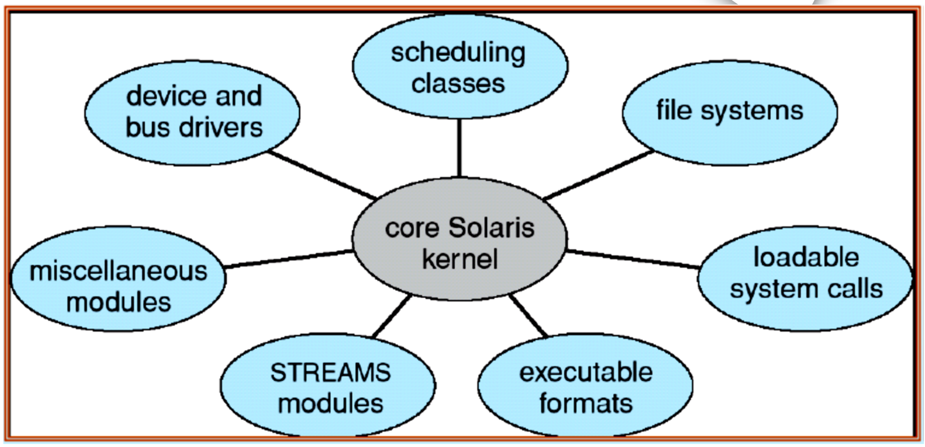


Figure 2.5 Solaris loadable modules.

## 2.7 System Boot

After an operating system is generated, it must be made available for use by the hardware. But how does the hardware know where the kernel is or how to load that kernel? 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** 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.

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.

*End of chapter 2*