Operating-System Structures

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- Operating System Debugging
- System Boot

Objectives

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

2.1 Operating System Services

Operating systems provide an environment for execution of programs and services to programs and users.

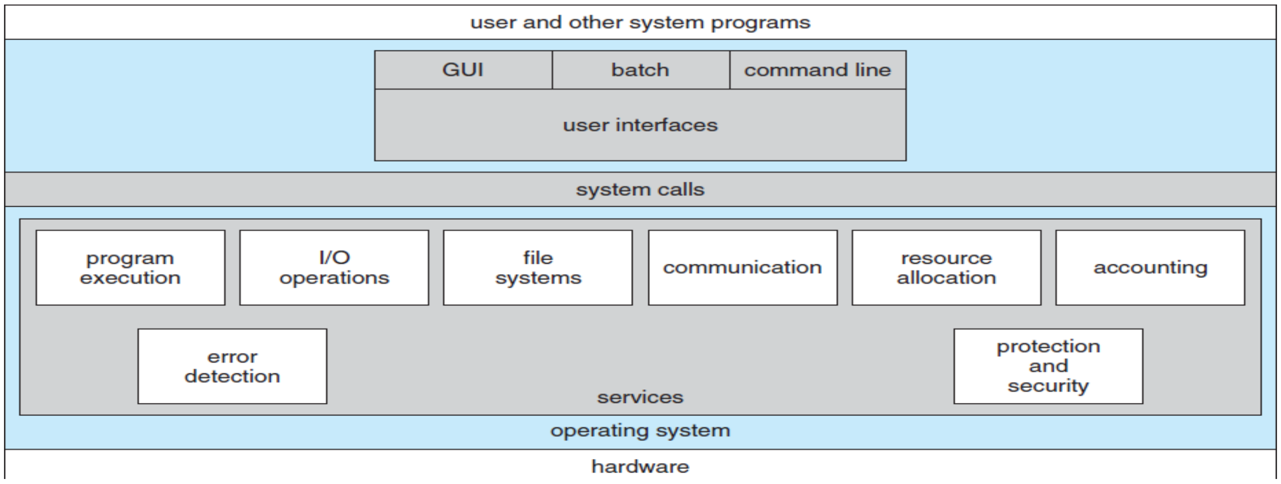


Figure 2.1 A view of operating system services.

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

- **User interface** - Almost all operating systems have a user interface (**UI**).
  - Varies between **Command-Line (CLI), Graphics User Interface (GUI), Batch interface.**
- **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating an 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 display screen).
- **File-system manipulation** - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
- **Communications** – Processes may exchange information, on the same computer or between computers over a network
  - Communications may be via **shared memory** or through **message passing** (packets moved by the OS)
- **Error detection –** The operating system needs to be detecting and correcting errors constantly (باستمرار)
  - May occur in the CPU and memory hardware, in I/O devices, in user program
  - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
  - Debugging facilities can greatly enhance the user’s and programmer’s abilities to efficiently use the system.

■Another set of operating system functions exists not for helping the user, but rather for ensuring the efficient operation of the system itself. Systems with multiple users can gain efficiency by sharing the computer resources among the users

- **Resource allocation -** When there are multiple users or multiple jobs running concurrently (في وقت واحد), resources must be allocated to each of them
  - Many types of resources - CPU cycles, main memory, file storage, I/O devices.

  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.

- **Accounting -** To keep track of which users use how much and what kinds of computer resources
- **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control the 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.
  - **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.

2.2 User Operating System Interface

We mentioned earlier that there are several ways for users to interface with the operating system. Here, we discuss two fundamental approaches. One provides a command-line interface, or command interpreter, that allows users to directly enter commands to be performed by the operating system. The other allows users to interface with the operating system via a graphical user interface, or GUI.

2.2.1 CLI or command interpreter

Some operating systems include the command interpreter in the kernel. Others, such as Windows 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 interactive systems). For example, on UNIX and Linux systems, a user may choose among several different ways.

The main function of the command interpreter is to get and execute the next user-specified command. Many of the commands given at this level manipulate files: create, delete, list, print, copy, execute, and so on. The MS-DOS and UNIX shells operate in this way.

2.2.2 Graphical User Interfaces GUI

A second strategy for interfacing with the operating system is through a user-friendly graphical user interface, or GUI. Here, rather than entering commands directly via a command-line interface

- User-friendly desktop metaphor (استعاره)interface
  - Usually mouse, keyboard, and monitor
  - Icons represent files, programs, actions, etc.
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a folder).

<u>Touchscreen Interfaces</u>

Because a mouse is impractical for most mobile systems, smartphones and handheld tablet computers typically use a touchscreen interface. Here, users interact by making gestures(ايماءات) on the touchscreen—for example, pressing and swiping fingers across the screen. Figure 2.2 illustrates the touchscreen of the Apple iPad. Whereas earlier smartphones included a physical keyboard, most smartphones now simulate a keyboard on the touchscreen.

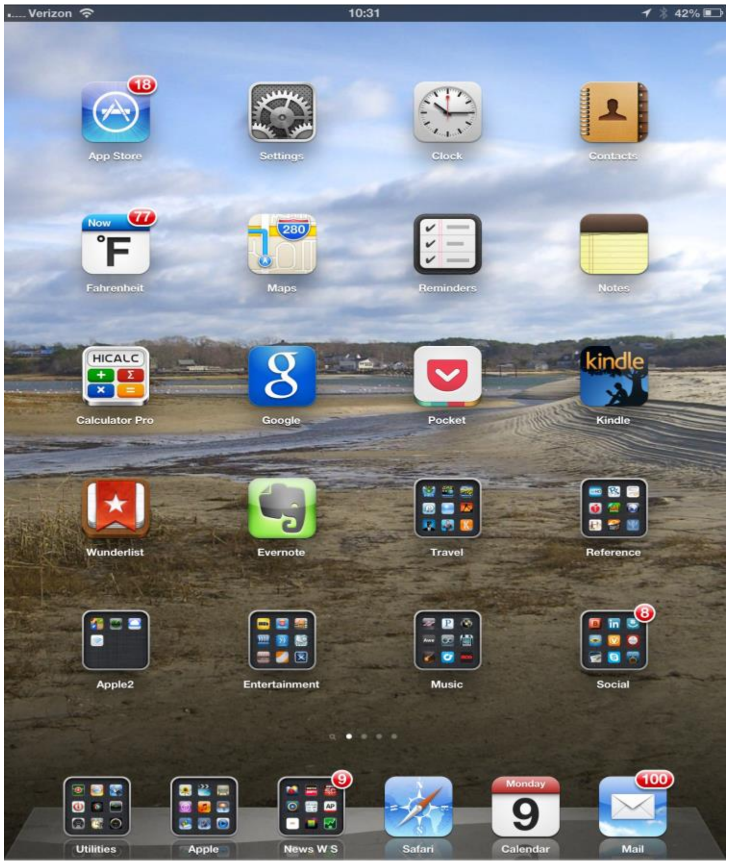


Figure 2.2 The iPad touchscreen.

2.2.3 Choice of Interface

The choice of whether to use a command-line or GUI interface is mostly one of personal preference. System administrators who manage computers and power users who have deep knowledge of a system frequently use the command-line interface. For them, it is more efficient, giving them faster access to the activities they need to perform

- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI "command" shell

- Apple Mac OS X is “Aqua” GUI interface with UNIX kernel underneath (تحت)and shells available
- Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME).

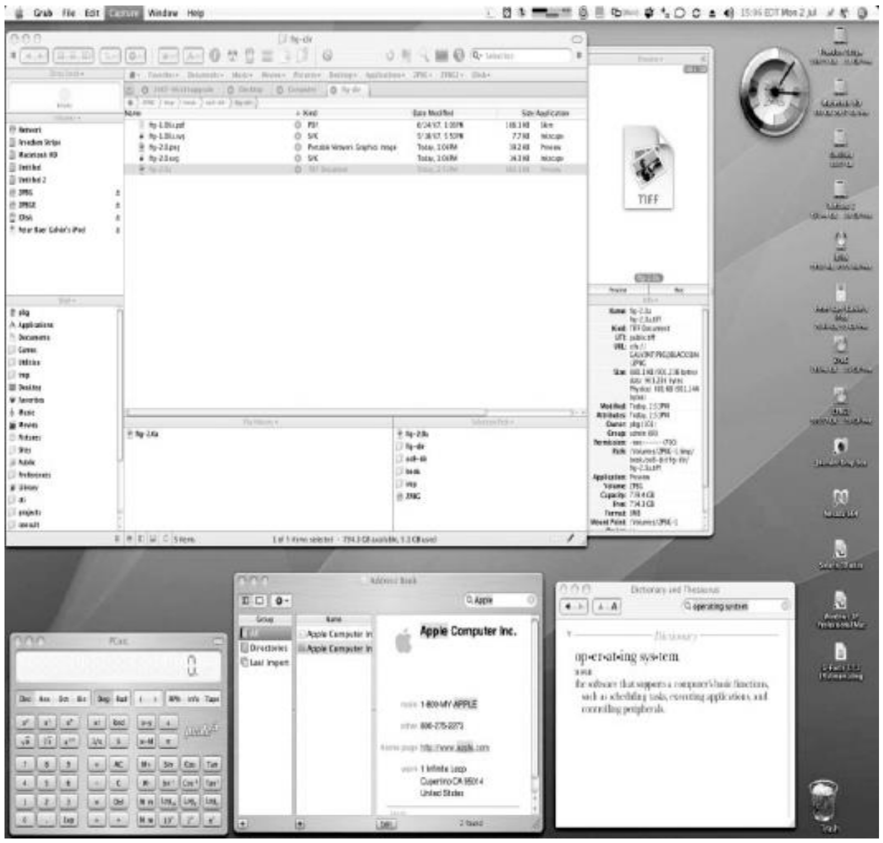


Figure 2.3 The Mac OS X GUI.

2.3 System Call

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 have to be written using assembly-language instructions.

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. **One approach** is for the program to ask the user for the names. 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 have been obtained, the program must open the input file and create the output file.

Each of these operations requires another **system call**. Possible error conditions for each operation can require additional system calls. When the program tries to open the input file, for example, 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 (yet 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.

- When 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 (for example, no more disk space). 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**). This system-call sequence is shown in Figure 2.4.

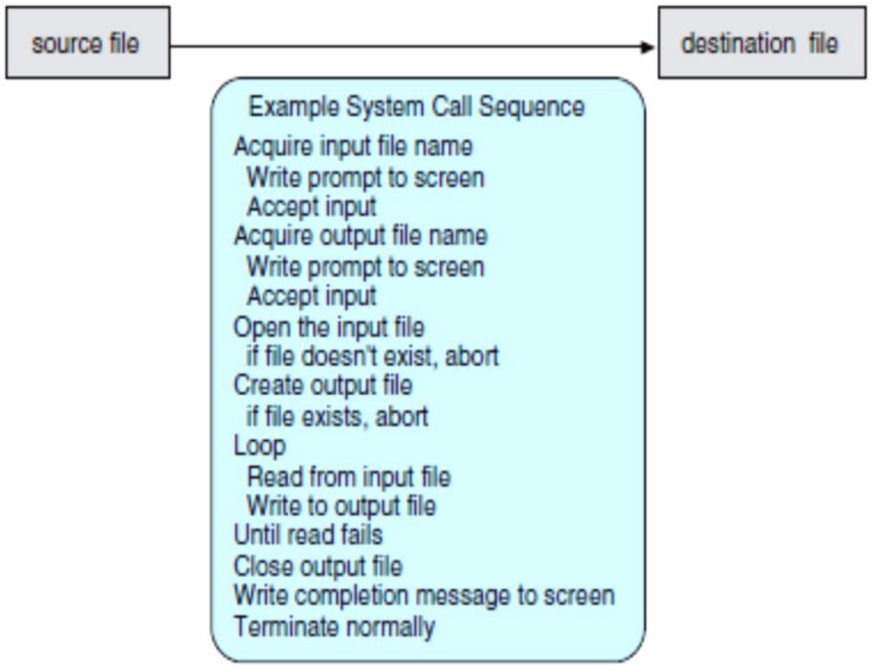


Figure 2.4 Example of how system calls are used.

From the example you can see:

- Even simple programs may make heavy use of the operating system.
- Systems execute thousands of system calls per second.

Most programmers never see this level of detail. 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, including the parameters that are passed to each function and the return values the programmer can expect.

Three of the most common APIs available to application programmers are:

- The Windows API for Windows systems.

- The POSIX API for POSIX-based systems (which include virtually all versions of UNIX, Linux, and Mac OSX), and
- The Java API for programs that run on the Java virtual machine.

2.4 Types of System Calls

System calls can be grouped roughly into six major categories:

1. **Process control**, A running program need to be able to:
   - Create a process, terminate process
   - End, abort
   - Load, execute
   - Get process attributes, set process attributes
   - Wait for a time
   - Wait event, signal event
   - Allocate and free memory
2. **File management,** We can identify several common system calls dealing with files.
   - Create a file, delete file
   - Open, close file
   - Read, write, reposition
   - Get and set file attributes
3. **Device management,** A process may need several resources to execute—main memory, disk drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user process. Otherwise, the process will have to wait until sufficient resources are available.
4. **Information maintenance,** Many system calls exist simply for the purpose of transferring information between the user program and the operating system.
   - Get time or date, set time or date
   - Get system data, set system data
   - Get and set process, file, or device attributes

5. **Communications**

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

- Create, delete communication connection
- Send, receive messages
- Transfer status information
- Attach and detach remote devices

6. **Protection**

The protection provides a mechanism for controlling access to the resources provided by a computer system. Historically, protection was a concern only on multiprogrammed computer systems with several users. However, with the advent(مجيء) of networking and the Internet, all computer systems, from servers to mobile handheld devices, must be concerned with the protection.

- Control access to resources
- Get and set permissions
- Allow and deny user access

2.5 System Programs

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

- **File management:** These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.
- **Status information:** Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information.
- **File modification:** Several text editors may be available to create and modify the content of files stored on disk or other storage devices.

- **Programming-language support:** Compilers, assemblers, debuggers and interpreters for common programming languages (such as C, C++, Java, 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.

Along with system programs, most operating systems are supplied with programs that are useful in solving common problems or performing common operations. Such application programs include Web browsers, word processors and text formatters, spreadsheets, database systems, compilers, plotting and statistical-analysis packages, and games.

2.6 Operating-System Design and Implementation

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

2.6.1 Design Goals

Specifying and designing an operating system is a highly creative(ابداعي)task.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(تتأثر) by the choice of hardware and the type of system: batch, time sharing, single user, multiuser, distributed, real time, or general purpose. Beyond this highest design level, the requirements may be much harder to specify. The requirements can, however, be divided into two basic groups:

- **User goals:** operating system should be convenient to use, easy to learn, reliable, safe, and fast
- **System goals**: operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient.

2.6.2 Implementation

Operating systems are collections of many programs, written by many people over a long period of time, because of this, it is difficult to make general statements about how they are implemented.

- Early operating systems were written in assembly language.
- Then system programming languages like, Algol, PL/1.
- Now, although some operating systems are still written in assembly language, most are written in a higher-level language such as **C** or C++.

Actually, an operating system can be written in more than one language.

- The lowest levels of the kernel might be assembly language.
- Main body in assembly.
- System programs in high level languages like, C, C++, Scripting languages like, PERL, Python.

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

The advantages of using a higher-level language for implementing operating systems are the same as those gained when the language is used for application programs: the code can be **written faster**, is **more compact**, and is **easier to understand** and **debug.**

The only possible disadvantages of implementing an operating system in a higher-level language are **reduced speed** and **increased storage requirements**.

2.7 Operating-System Structure

A system as **large** and **complex,** a modern operating system must be engineered carefully to function properly(بشكل صحيح) and be modified easily. A common approach is to **partition the task** into small components, or modules, rather than have one monolithic system.

2.7.1 Simple Structure

Many operating systems do not have well-defined structures, such 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 2.5 shows its structure.

- It was written to provide the most functionality in the least space.
- Not divided into modules.
- Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated

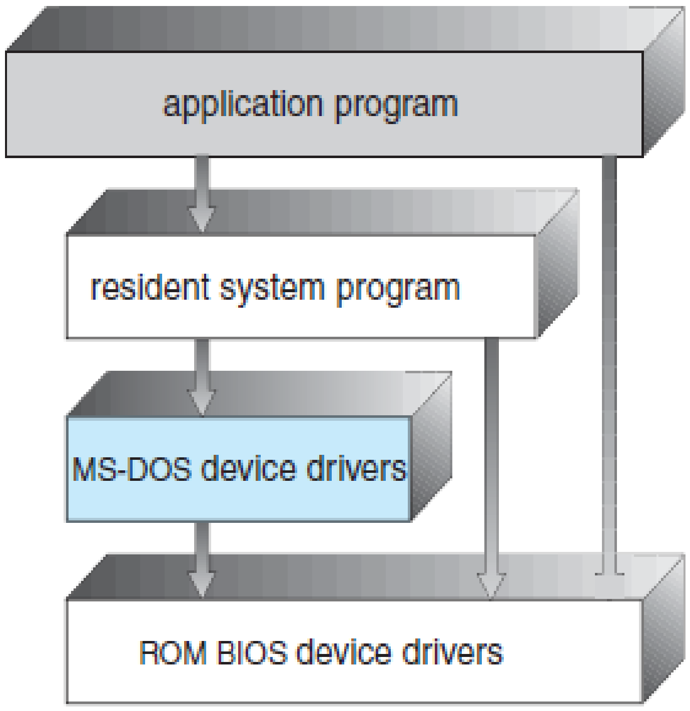


Figure 2.5 MS-DOS layer structure.

Another example of limited structuring is the original **UNIX** operating system. As shown in Figure 2.6.

- UNIX is limited by hardware functionality.
- The UNIX OS had limited structuring It consists of two separable parts:
    - The **system programs.**
    - The **kernel**.
        - Consists of everything below the system-call interface and above the physical hardware.
        - Provides the file system, CPU scheduling, memory management, and other operating-system functions.

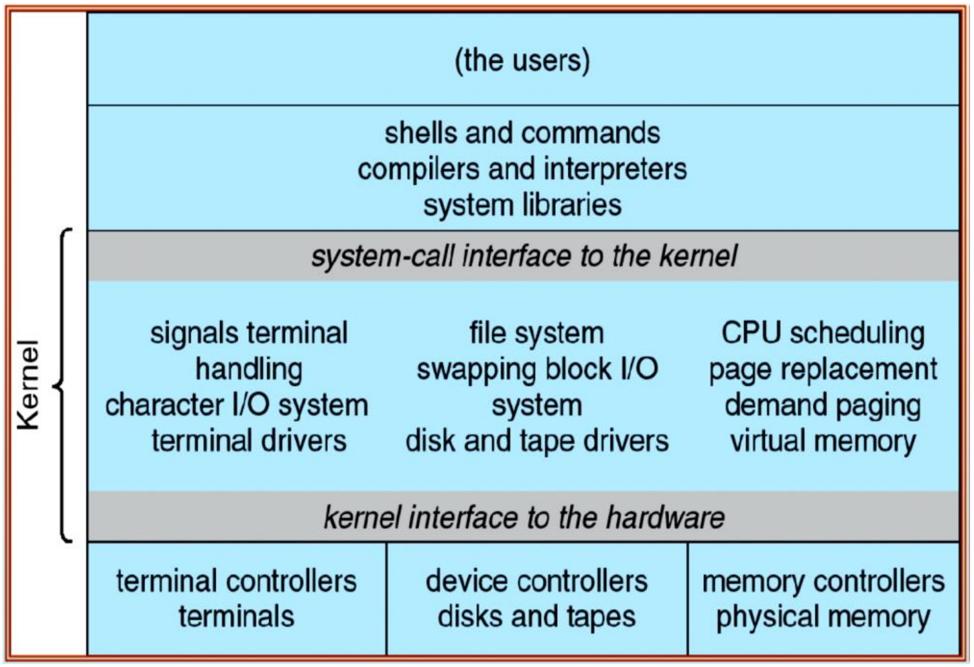


Figure 2.6 UNIX system structure.

2.7.2 Layered Approach

Operating systems can be broken into pieces that are smaller and more appropriate than those allowed by the original MS-DOS and 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.

A system can be made modular in many ways. One method is the **layered approach**, in which the operating system is broken into a number of layers (levels). Figure 2.7.

- The bottom layer (layer 0) is the hardware;
- The highest (layer N) is the user interface.
- With modularity, layers are selected so that each uses functions (operations) and services of only lower-level layers.

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

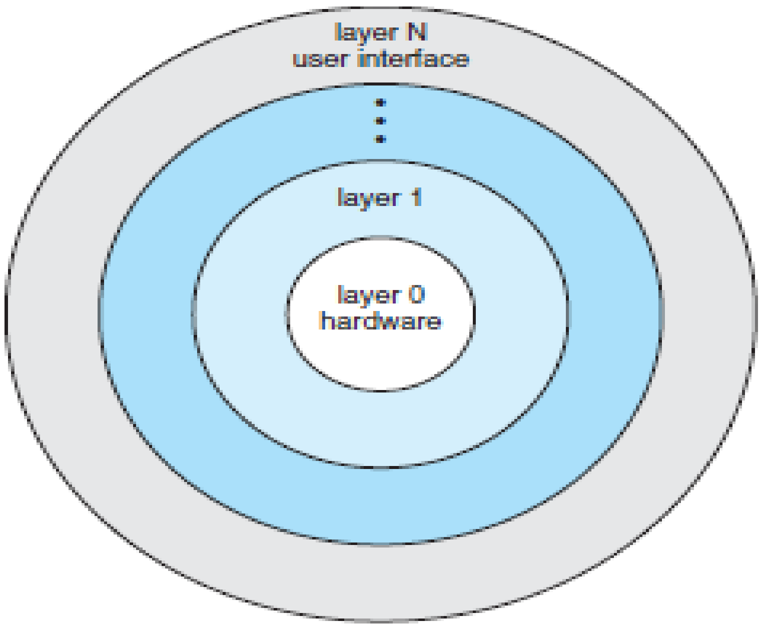


Figure 2.7 A layered operating system.

2.7.3 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.
- Microkernels provide minimal process and memory management, in addition to a communication facility. Figure 2.8 illustrates the architecture of a typical microkernel.

- The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space.
- Communication takes place between user modules using **message passing**.
- Benefits:
  - Easier to extend a microkernel.
  - Easier to put the operating system to new architectures.
  - More reliable (less code is running in kernel mode)
  - More secure

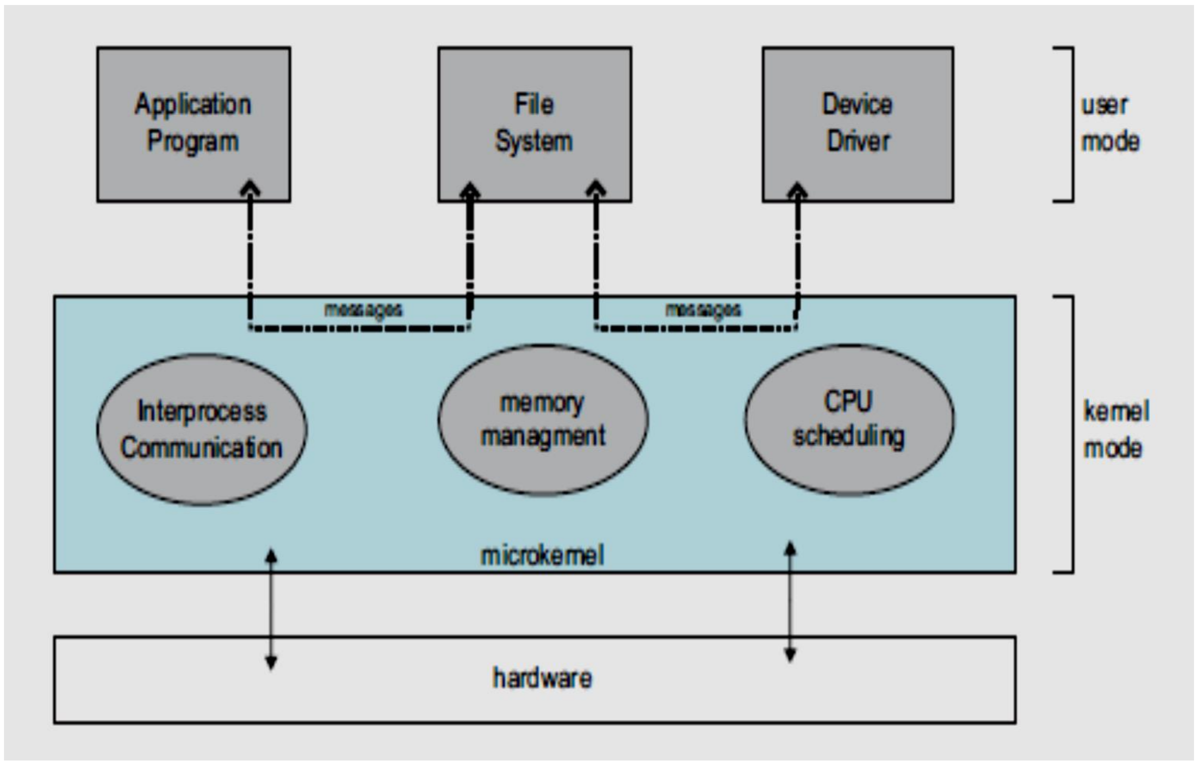


Figure 2.8 Architecture of a typical Microkernel.

2.7.4 Modules

- Many modern operating systems implement **loadable kernel modules.** Figure 2.9.
- Uses the object oriented programming techniques to create a modular kernel.
- Each core component is separate Here, the kernel has a set of core components and dynamically links in additional services either during boot time or during run time.

- Overall, similar to layers, but with more flexible.
- An example of such OS, Linux, Solaris, etc.

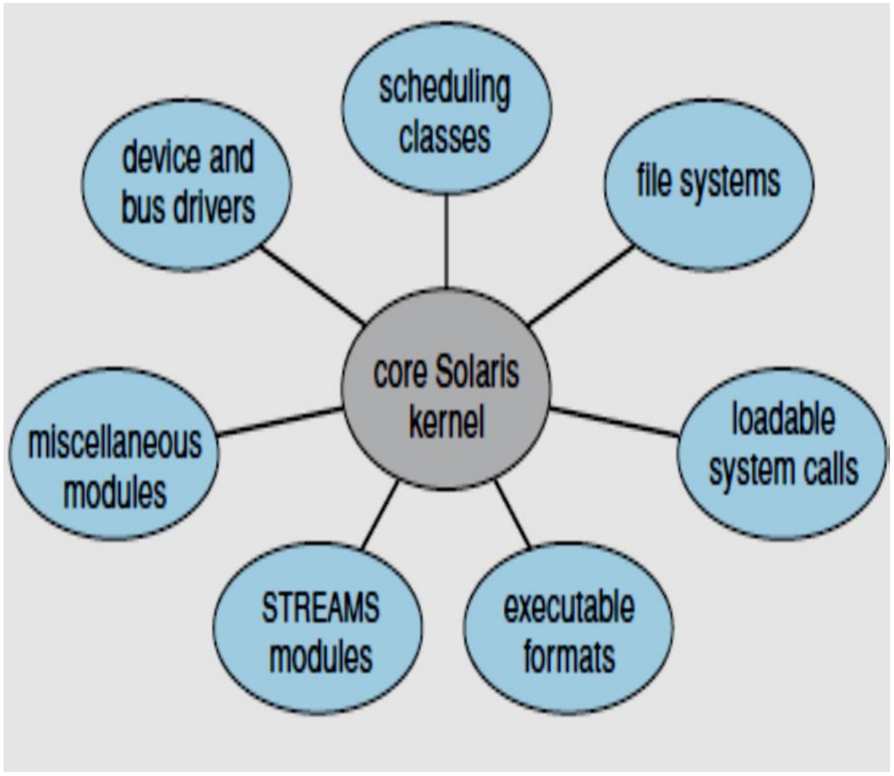


Figure 2.9 Solaris loadable modules

2.7.5 Hybrid Systems

Most modern operating systems are actually not one pure model

- Hybrid combines multiple approaches to address performance, security, usability needs.
- Windows mostly monolithic(متراصة), plus microkernel for different subsystem personalities.
- An examples of Hybird systems:
  - **Apple Mac OS X,** and two prominent mobile operating system
  - **iOS**
  - **Android**

2.8 Operating-System Debugging

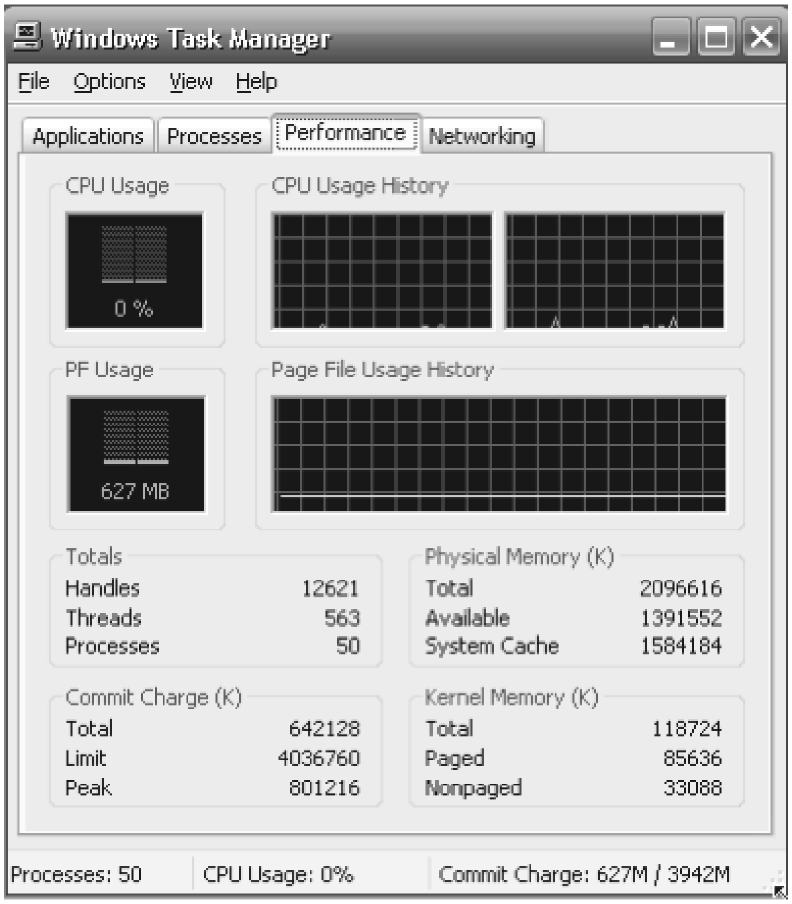
Debugging is the activity of finding and fixing errors or bugs in a system, both in hardware and in software

2.8.1 Failure Analysis

- If a process fails, OS generate a **log file** containing error information to alert system operators or users that the problem occurred.
- The operating system can also take a **core dump(نفاية) -** a capture of the memory of the process - and store it in a file for later analysis. (Memory was referred to as the "core" in the early days of computing.)
- A failure in the kernel is called a **crash**. When a crash occurs, the error information is saved to a log file, and the memory state is saved to a **crash dump**.
- Operating-system debugging and process debugging frequently use different tools and techniques due to the very different nature of these two tasks.

2.8.2 Performance Tuning

- The **performance tuning (ضبط الاداء)** seeks to improve performance by removing processing bottlenecks. To identify bottlenecks, we must be able to monitor system performance.
- The operating system must have some means of computing and displaying measures of system behavior. In a number of systems, the operating system does this by producing trace **listings of system behavior**.
- Another approach to performance tuning uses **single-purpose interactive tools** that allow users and administrators to question the state of various system components to look for bottlenecks.
  - The Windows Task Manager is a similar tool for Windows systems. The task manager includes information for current applications as well as processes, CPU and memory usage, and networking statistics. A screen shot of the task manager appears in Figure 2.10.



Totals
Handles 12621
Threads 563
Processes 50

Physical Memory (K)
Total 2096616
Available 1391552
System Cache 1584184

Commit Charge (K)
Total 642128
Limit 4036760
Peak 801216

Kernel Memory (K)
Total 118724
Paged 85636
Nonpaged 33088

Processes: 50 CPU Usage: 0% Commit Charge: 627M / 3942M

Figure 2.10 The Windows task manager

2.9 System boot

When power initialized on system, the OS 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.
- The operating system must be made available to hardware, so hardware can start it
- On most computer systems, a small piece of code known as the **bootstrap program** or **bootstrap loader** stored in ROM or EEPROM locates the kernel, loads it into main memory, and starts its execution.

- 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(مناسب) because it needs no initialization and cannot be infected by a computer virus.
- 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.
- 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.