Chapter 2 – Operating System Structures

Operating system provides an environment within which programs are executed. Internally operating systems vary greatly in their makeup to serve different computing environments. The design of a new OS is a major task.

In this chapter we discuss what **services** an OS provides, how they are provided and debugged, what are the various methodologies for designing them, how the OS are created and how a computer starts its OS.

Contents

2.9 Building and Booting an Operating System

2.9.1 Operating System Generation

2.9.2 System Boot

2.10 Operating System Debugging

2.10.1 Failure Analysis

2.10.2 Performance Monitoring and Tuning

2.10.2.1 Counters

2.10.3 Tracing

2.10.4 BCC

Chapter Objectives

- Identify services provided by an operating system.
- Understand how system calls are used to provide OS services.
- Compare and contrast different design strategies of operating systems.
- Illustrate the process for booting an operating system.
- Discuss tools for monitoring operating system performance.

2.1 Operating-System Services

To make environment for the execution of programs, the OS provides certain services to the programs and their users. The specific services differ from one OS to another.

Some of the OS services provide following functions for the **users:**

- **User interface.** Almost all OS provide a user interface (UI) to enable users interact with the system. There are **graphical user interfaces (GUI)**, **touch-screen interfaces** and **command-line interfaces (CLI)** etc. depending on user preferences and the system environment.
- **Program execution.** OS enables the users to load a program into memory and to run that program. The program must be able to end its execution normally or by reporting error.
- **I/O operations.** A running program may require I/O from a file, device or a network interface. The users usually cannot control I/O devices directly, so the OS provides a means to do it.
- **File-system manipulation.** Programs and users need to read, write, create, delete, search and list the files and directories securely. OS provides a variety of file system management services.
- **Communications.** Often one process needs to exchange information with another process in the same system or over the network. OS provides means of communication by implementing **shared memory** (two or more processes read/write to a shared section of memory) or **message passing** (packets of information in predefined formats are moved between processes).
- **Error detection.** Errors may occur in the CPU and memory hardware, in I/O devices and in the user programs. For each type of error, the OS should detect and take proper action to ensure correct and consistent computing.

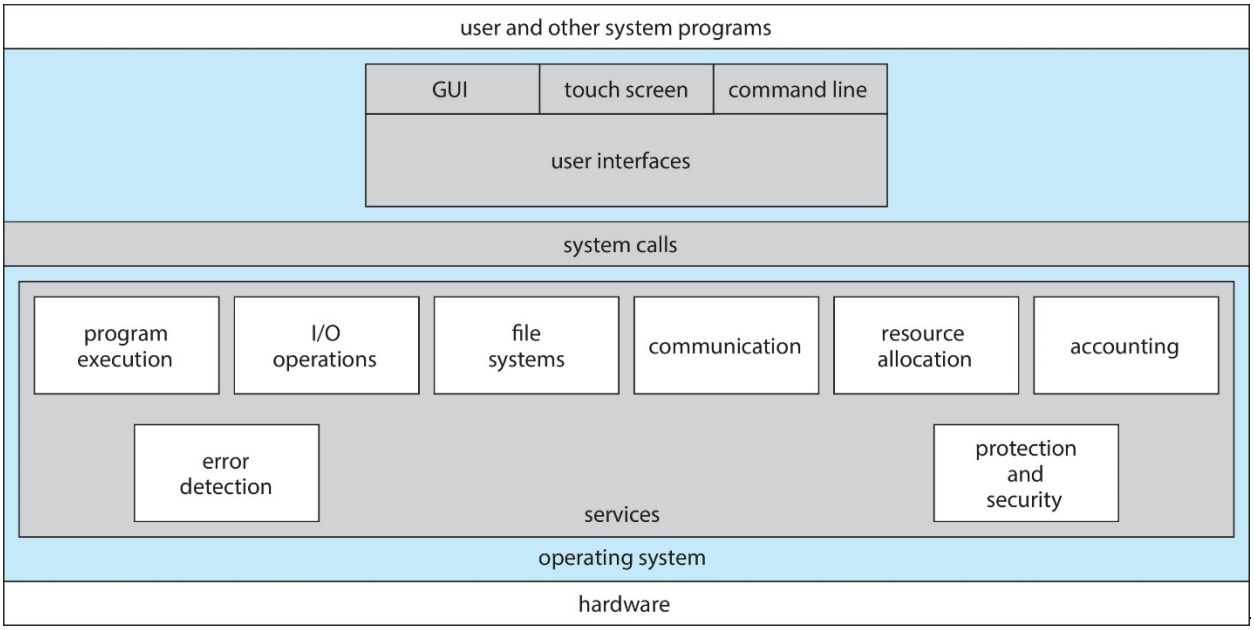

Figure 1 *A view of operating system services*

Some other OS services ensure the efficient operation of the **system** itself:

- **Resource allocation.** When there are multiple processes running at the same time, the OS manages and systematically distributes different types of resources (CPU, memory, storage and I/O devices etc.) among the processes.
- **Logging.** To keep track of which programs use how much and what kinds of computer resources, OS keeps record, which can be used for accounting or for accumulating usage statistics.
- **Protection and security.** Controlling the use of information in a multiuser or networked system is must. When several separate processes execute concurrently, one process should not be able to interfere with the others or with the OS. OS should provide protection to ensure that all access to system resources is controlled; and measures of security to keep the system safe.

2.2 User and Operating-System Interface

There are several ways for users to interface with the OS. Here we discuss **three** different approaches.

2.2.1 Command Interpreters

In modern operating systems (e.g. Linux, UNIX and Windows), **Command line interface (CLI)** or **command interpreter** (commonly known as a **shell**) is a special program that runs when a process is initiated or when a user first logs on and allows users to directly enter commands to be performed. C shell, Bourne-Again shell, Korn shell are a few examples of this.

The main function of the command interpreter is to get and execute the user specified command. The commands can be executed in **two** general ways:

- The command interpreter itself contains the code to execute the command and each command has its own code implementation. The size of the command interpreter depends on the number of commands available.

- Most commands are implemented through system programs (used by Unix). The command interpreter gets the file name from the command, loads into memory and executes it. This way new commands can be easily added to the system by adding new program files.

2.2.2 Graphical User Interface

Here users employ a mouse based window-and-menu system characterized by a desktop metaphor. Clicking a button on the mouse after positioning on desktop screen can invoke a program, select a file or directory, or pull down a menu that contains commands.

2.2.3 Touch-Screen Interface

Smartphone and handheld devices typically use a touch-screen interface, where users interact with the system by making gestures on the touch screen.

2.3 System Calls

System calls provide an interface to the services made available by an operating system. These calls are generally available as functions written in C and C++. Certain low-level tasks where accessing hardware directly is needed, are written using assembly language instructions.

2.3.1 Example

Suppose we want to read data from one file and copy to another file. One way of doing it is to use the Unix cp command: `cp in.txt out.txt.`

Another approach for the program is to ask the file names to the user. In an interactive system, it makes a series of system calls, shown in the following figure.

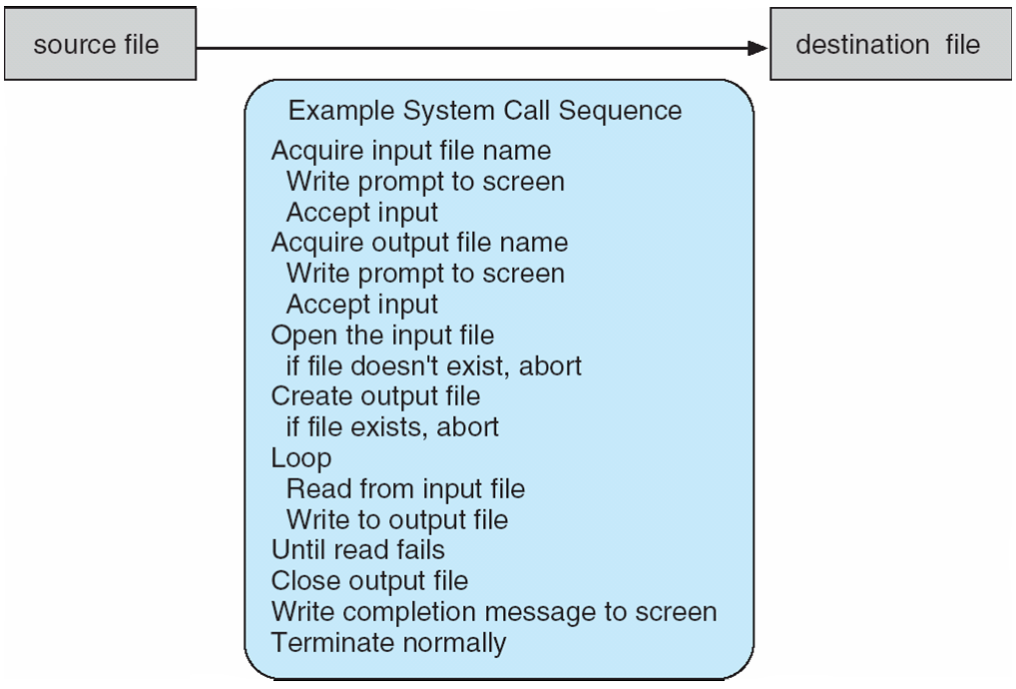

Figure 2 *Example of how system calls are used*

Each of the above actions requires system call.

2.3.2 Application Programming Interface

As we can see, even simple programs may make heavy use of the OS. Frequently 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** including the parameters to pass and their return values.

Windows API for Windows systems, the **POSIX API** for POSIX based systems (which include virtually all versions of UNIX, Linux, and macOS), and the **Java API** for programs that run on the Java virtual machine are most common APIs. A programmer accesses an API via a library provided by the OS (In UNIX and Linux, the library for programs written in the C language is called *libc*). Each OS has its own name for each system call. Behind the scenes the functions in the API invoke the actual system calls.

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t    read(int fd, void *buf, size_t count)
```

return value — function name — parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

Figure 3 *Example of a standard API*

For an application programmer, there are several benefits of using and API instead of invoking the actual system call. It increases the **program portability**. Typically, a program written using an API can be

compiled and run on other systems that support the same API. Furthermore, actual system calls can often be more detailed and difficult to work with than the API. Most of the details of the OS interface are hidden from the programmer by the API and are managed by the Runtime Environment (RTE- the full suite of software needed to execute applications written in a given programming language).

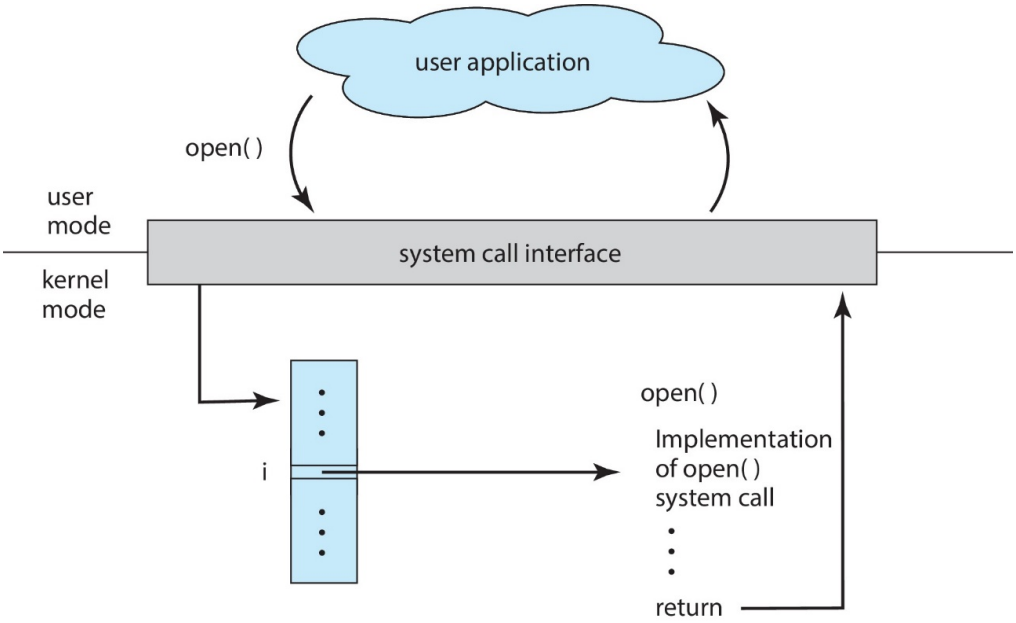

Figure 4 *The handling of a user application invoking the open() system call*

System calls occur in different ways and usually require some information. There are **three** general ways to pass the parameters to the operating system:

- The simplest approach is to pass the parameters in registers.
- When there are more parameters than the registers, these are generally stored in a block, or table, in memory, and the address of the block is paired as a parameter in a register.
- Parameters also can be placed, or pushed, onto a stack by the program and popped off the stack by the operating system.

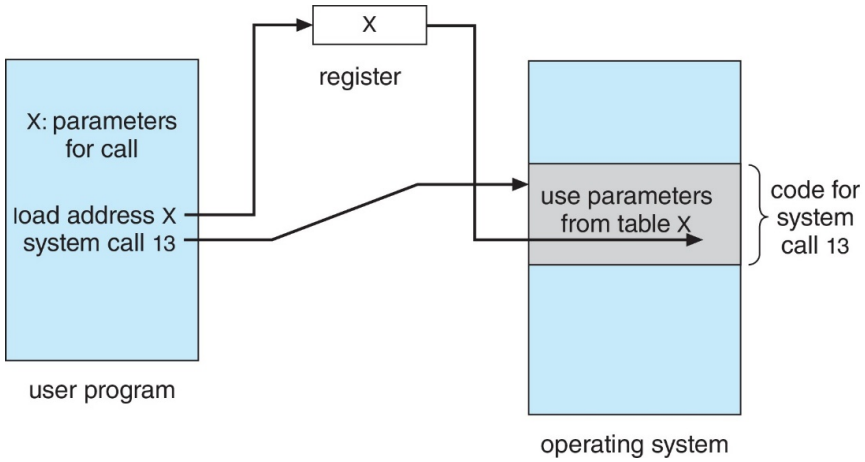

Figure 5 *Passing of parameters as a table*

2.3.3 Types of System Calls

System calls provided by the OS can be grouped roughly into following **six** major categories:

2.3.3.1 Process Control

- create/terminate process
- load, execute
- get/set process attributes
- wait event, signal event
- allocate and free memory

2.3.3.2 File Management

- create file, delete file
- open, close
- read, write, reposition
- get/set file attributes

2.3.3.3 Device Management

- request device, release device
- read, write, reposition
- get device attributes
- set device attributes
- logically attach/detach devices

2.3.3.4 Information Maintenance

- get time/date, set time/date
- get/set system data
- get process, file, or device attributes
- set process, file, or device attributes

2.3.3.5 Communication

- create, delete communication connection
- send, receive messages
- transfer status information
- attach/detach remote devices

2.3.3.6 Protection

- get file permissions
- set file permissions

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communications	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Figure 6 *Examples of Windows and Unix System Calls*

THE STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:

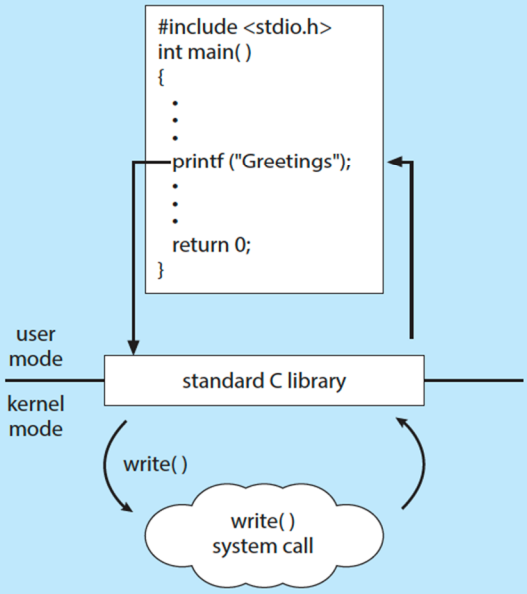

Figure 7 *C program invoking printf() library call, which calls write() system call*

2.4 System Services

Another aspect of a modem OS is its collection of system services (also known as system utilities). Some of them are simply user interfaces to system calls, others are more complex. These provide a convenient environment for program development and execution, and can be divided into these categories:

- **File management.** These programs help create, delete, copy, rename, print, list and generally access and manipulate files and directories.
- **Status information.** These programs collect status information such as date, time, amount of available memory or disk space, number of users etc. from the system. Some more complex programs provide detailed information on performance, logging, debugging and configuration.
- **File modification.** These are editors to create and modify the content of files stored on disk or other storage devices.
- **Programming-language support.** These are compilers, assemblers, debuggers, and interpreters provided with the OS for common programming languages.
- **Program loading and execution.** Once a program is assembled or compiled, it must be loaded into memory to be executed. The OS provides **absolute loaders**, **relocatable loaders**, linkage editors and overlay loaders for that.

- **Communications.** These programs provide the mechanism for creating virtual connections among processes, users, and systems. These allow users to send messages to one another, to browse web pages, send e-mails, log in remotely or transfer files from one machine to another.
- **Background services.** All general purpose systems have methods for launching certain system-program processes at boot time. Some of these processes terminate after completing their tasks, others keep running until the system is halted. Constantly running system-program processes are known as **services**, **subsystems**, or **daemons**.

Along with system programs, most OS have **application programs** for solving common problems or performing common operations such as web browsers, word processors, compilers, games etc.

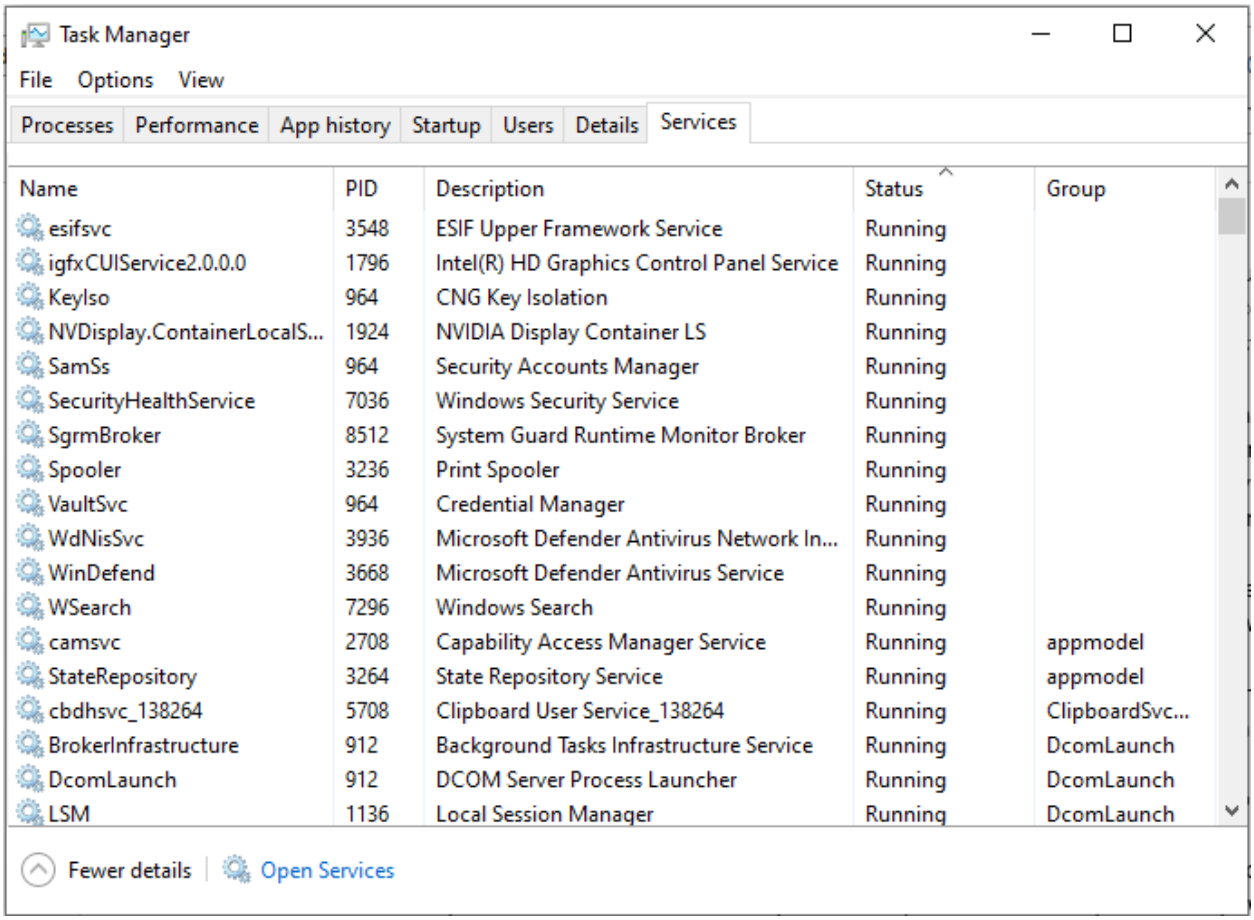

Figure 8 *Some Windows10 Services*

2.5 Linkers and Loaders

Usually a program is a binary executable file stored on the disk; for example, *a.out* or *prog.exe*. To run on a CPU, the program must be loaded into memory and placed in the context of a process. In this section, we describe the steps in this procedure, from compiling a program to placing it in memory, where it can execute its operations.

Source files are compiled into object files that are designed to be loaded into any physical memory location, a format known as a **relocatable** object file. Next, the linker combines these relocatable object

files into a single binary executable file. During the linking phase, other object files or libraries may be included as well, such as the standard C or math library.

A **loader** is used to load the binary executable file into memory, where it is eligible to run on a CPU core. An activity associated with linking and loading is **relocation**, which assigns final addresses to the program parts and adjusts the code and data accordingly so that the code can call library functions and access its variables as it executes.

Most OS (such as Windows) allow a program to **dynamically link libraries (DLLs)** as it is loaded. It allows the libraries to be conditionally linked and loaded only if it is required during runtime.

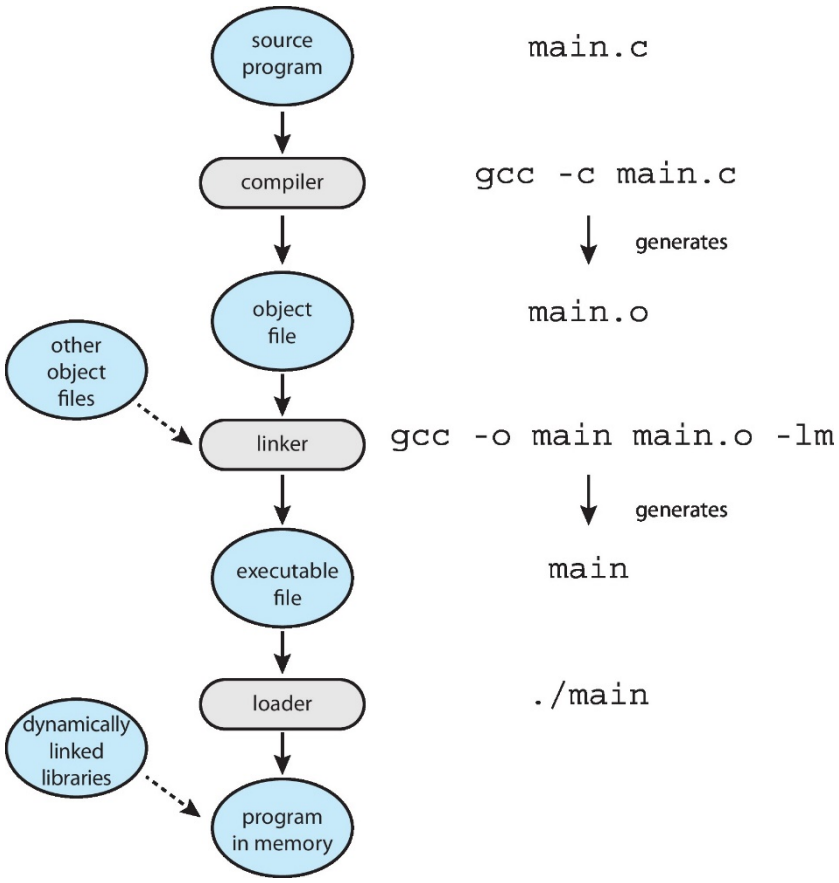

Figure 9 *The role of the linker and loader*

2.6 Why Applications Are Operating-System Specific

Fundamentally applications compiled on one OS are not executable in other OS; the key reasons are:

- Each OS provides a unique set of system calls, which are part of the services provided by the OS that help the applications to execute.
- Each OS has its own binary format for applications. So the layout of the header, instructions, and variables in the executable differs from OS to OS.
- CPUs have varying instruction sets, and only applications containing the appropriate instructions can execute correctly.

An application can be made available (not completely) to run on multiple OS in one of these ways:

- The application can be written in an interpreted language (such as Python or Ruby) that has an interpreter available for multiple OS. Interpreter reads each line of the source program, executes equivalent instructions on the native instruction set, and calls native system calls.
- The application can be written in a language that includes a virtual machine containing the running application. The VM is part of the language's full RTE. For example, Java has a **runtime environment (RTE)** that includes a loader, byte-code verifier, and other components that load the Java application into the Java virtual machine. In theory, any Java app can run within the RTE wherever it is available.
- The application developer can use a standard language or **API** in which the compiler generates binaries in a machine and operating system specific language.

At the architectural level, an **application binary interface (ABI)** is used to define how different components of binary code can interface for a given OS on a given architecture. An ABI specifies low level details such as methods of passing parameters to system calls, organization of the run-time stack, format of system libraries, size of data types etc. So unless an interpreter, RTE or binary executable file is written for and compiled on a specific OS on a specific CPU type, the application will fail to run.

2.7 Operating System Design and Implementation

2.7.1 Design Goals

Specifying and designing an OS is a major task and it needs rigorous software engineering. The first step in designing a system is to define goals and specification. At the highest level, the design depends on the hardware and the type of the system: traditional desktop/laptop, mobile, distributed, or real time. Then the requirements can be divided into two basic groups**: user goals** and s**ystem goals**.

The users want a system convenient to use, easy to learn, reliable, safe and fast. The developers who design, create, maintain and operate the system, want the system to be easy to design, implement and maintain. They want to make it flexible, reliable, error free, and efficient.

2.7.2 Mechanisms and Policies

Mechanisms determine how to do something. Policies determine what will be done. The separation is important for flexibility. Policy decisions are important for all kind of resource allocation.

2.7.3 Implementation

Operating systems are collections of many programs, written by many people over a long period of time. Early OS were written in assembly language. Now higher level routines are mostly written in higher-level languages such as C or C++, with lowest level of the kernel written in assembly language.

2.8 Operating-System Structure

A system as large as a modern OS must be engineered with care to be properly functioning and easily modifiable. A common approach is to partition the task into small components or modules. Each of these modules should be a well-defined portion of the system with carefully designed interfaces and functions. We briefly discussed the common components of an OS in chapter 1 and here we discuss how these components are interconnected and melded into a kernel.

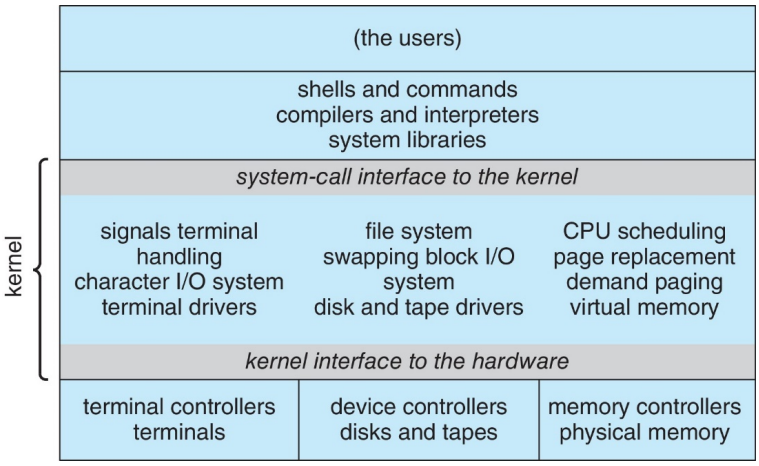

Figure 10 *Traditional UNIX system structure*

2.8.1 Monolithic Structure

This is the simplest structure of an OS where all of the functionality of the kernel are put into a single, static binary file that runs in a single address space. Original Unix is an example of this, which consists of two separable parts: the **kernel** and the **system programs**. The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years. The kernel provides the file system, CPU scheduling, memory management, and other OS functions through system calls. It gives good performance, as there is very little overhead in the system call interface and communication within the kernel is fast. The drawback is that these systems are difficult to implement and extend. Despite that, this is widely used for its simplicity and distinct performance advantage.

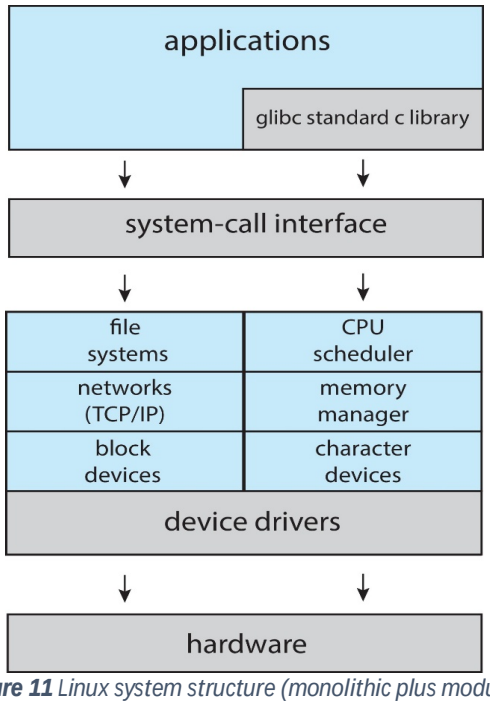

Figure 11 *Linux system structure (monolithic plus modular)*

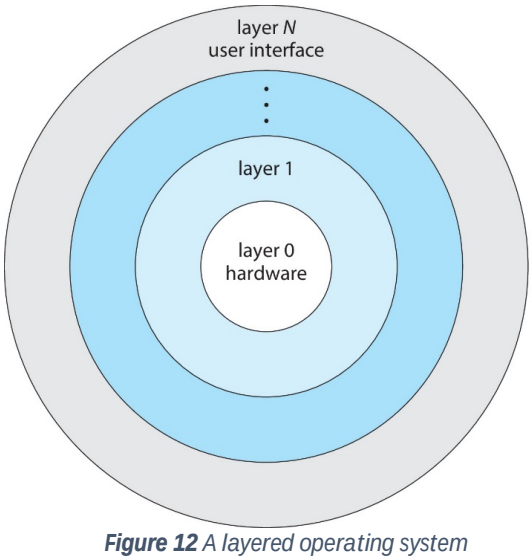

Figure 12 *A layered operating system*

2.8.2 Layered Approach

In monolithic approach, changes in one part of the system significantly effects the other parts. Alternatively a kernel can be divided into separate, smaller components that have specific and limited functionality. The advantage of this modular approach is that changes in one component affect only that component, and no others.

A system can be made modular in many ways, one of them is the **layered** approach, where the OS is divided into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface. A typical layer of the OS consists of data structures and a set of functions. Each layer can use functions of only lower level layers.
Layered systems are used in computer networks (such as TCP/IP) and web applications. Its main advantage is simplicity of construction and debugging. A pure layered approach is not commonly used as appropriately defining the functionality of each layer is challenging. Also, its overall performance is poor as the user programs have the overhead of traversing through multiple layers to obtain an OS service.

2.8.3 Microkernels

In this approach all nonessential components are removed from the kernel and implemented as user-level programs that reside in separate address spaces. As a result the microkernels are smaller which provide minimal process and memory management with a communication facility. Communication between the client program and various services is provided through message passing. For example, if a client program needs to access a file, it communicate indirectly with the file server by exchanging messages with the microkernel.

Perhaps the best-known illustration of a microkernel OS is **Darwin**, the kernel component of the macOS and iOS. Microkernels make the OS easy to extend. All new services are added to user space without modifying the kernel. The resulting OS is easier to port from one hardware design to another. As most services run as user processes, it is more secure and reliable. Unfortunately, the performance of microkernels can suffer for increased system-function overhead due to frequent copying of messages and switching between processes.

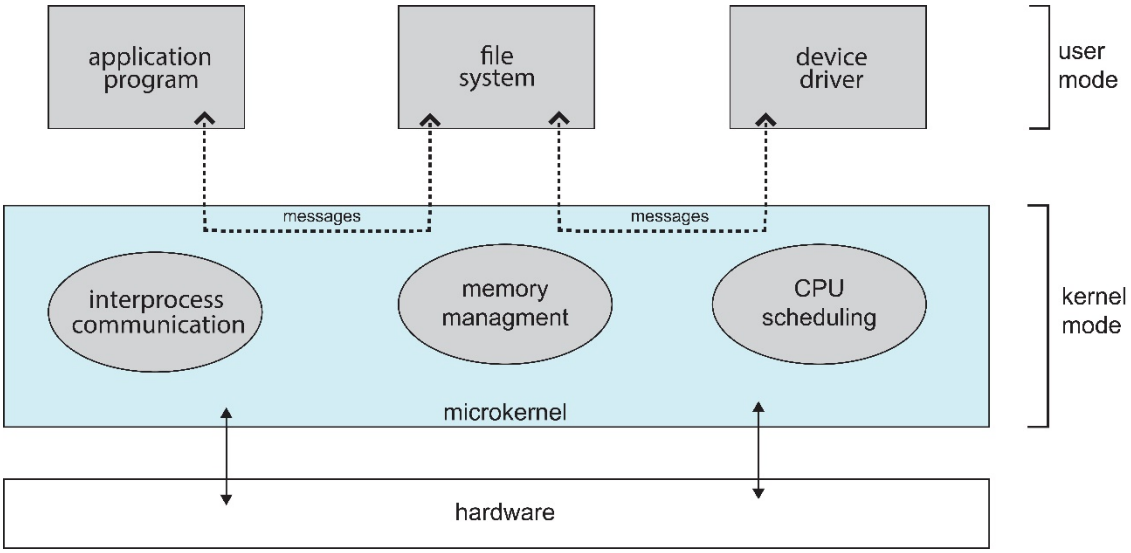

Figure 13 *Architecture of a typical microkernel*

2.8.4 Modules

Perhaps the best current methodology for OS design involves using **loadable kernel modules (LKMs).** Here, the kernel has a set of core components and can link in additional services via modules, either at boot time or during run time. The kernel is designed to provide core services, while other services are implemented dynamically, as the kernel is running. Linking services dynamically is preferable to adding new features directly to the kernel, which would require recompiling the kernel every time a change was made. The overall result resembles a layered system in that each kernel section has defined, protected interfaces; but it is more flexible than a layered system, because any module can call any other module. The approach is also similar to the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient, because modules do not need to invoke message passing in order to communicate. Linux uses loadable kernel modules, primarily for supporting device drivers and file systems. LKMs can be removed from the kernel during run time as well. For Linux, LKMs allow a dynamic and modular kernel, while maintaining the performance benefits of a monolithic system.

2.8.5 Hybrid Systems

In practice, operating systems combine different structures, resulting in hybrid systems that address performance, security, and usability issues. For example, Linux is monolithic because having the OS in a single address space provides very efficient performance. It is also modular, so that new functionality can be dynamically added to the kernel. Windows is largely monolithic but it retains some features of microkernels by providing support for separate sub systems.

2.9 Building and Booting and Operating System

OS are generally designed to run on a class of systems with variety of peripheral configurations.

2.9.1 Operating System Generation

Commonly, OS are already installed on purchased computers. However, to generate (build and install) an OS from scratch, following steps are needed:

- Write/Obtain the operating system source code.
- Configure the operating system for the system on which it will run.
- Compile the operating system.
- Install the operating system.
- Boot the computer and its new operating system.

2.9.2 System Boot

After generating an OS, the process of starting a computer by loading the kernel is known as booting the system. On most systems, the boot process proceeds as follows:

- A small piece of code known as the **bootstrap program** or **boot loader** locates the kernel.
- The kernel is loaded into memory and started.
- The kernel initializes hardware.
- The root file system is mounted.

2.10 Operating System Debugging

Broadly, debugging is the activity of finding and fixing errors in a system. In this section we mention the debugging processes of kernel errors and performance problems.

2.10.1 Failure Analysis

If a process or the kernel fails, most OS write the error information to a log file, capture the memory state of the process/kernel and store them. These files can be analyzed later.

2.10.2 Performance Monitoring and Tuning

To observe the system behavior, the OS must provide tools that can measure per-process or system-wide behaviors. For making the observations, these tools use **counters** or **tracing**.

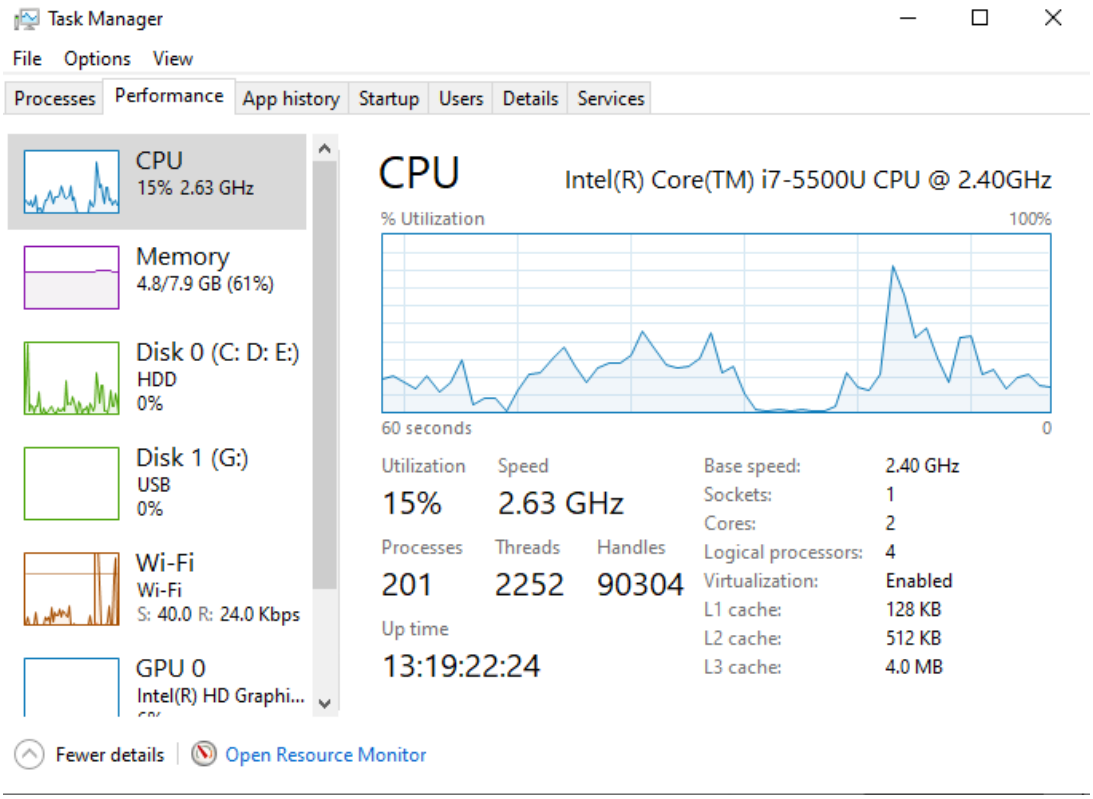

Figure 14 *The Windows 10 task manager*

2.10.2.1 Counters

Operating systems keep track of system activity through a series of counters such the number of system calls made, or the number of operations performed to a network device.

ps, top, vmstat, netstat, iostat etc. are examples of Linux tools that use counters.

2.10.3 Tracing

Counter-based tools simply inquire on the current value of certain statistics maintained by the kernel, where tracing tools collect data for a specific event – such as the steps involved in a system call invocation. *strace, gdb, perf, tcpdump* etc. are examples of Linux tools that use trace events.

2.10.4 BCC

BCC (BPF Compiler Collection) is a rich toolkit that provides tracing features for Linux systems.