

OPERATING SYSTEMS

MODULE-1

Operating Systems Overview: Introduction, Operating system functions, Operating systems operations, Computing environments, Free and Open-Source Operating Systems
System Structures: Operating System Services, User and Operating-System Interface, system calls, Types of System Calls, system programs, Operating system Design and Implementation, Operating system structure, Building and Booting an Operating System, Operating system debugging

INTRODUCTION

An **Operating System** is software that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware. Because an operating system is **large and complex**, it must be created piece by piece. Each of these pieces should be a well-delineated portion of the system, with carefully defined inputs, outputs, and functions.

A computer system can be divided roughly into four components:

- **The Hardware**
- **The Operating System**
- **The Application Programs**
- **A User**

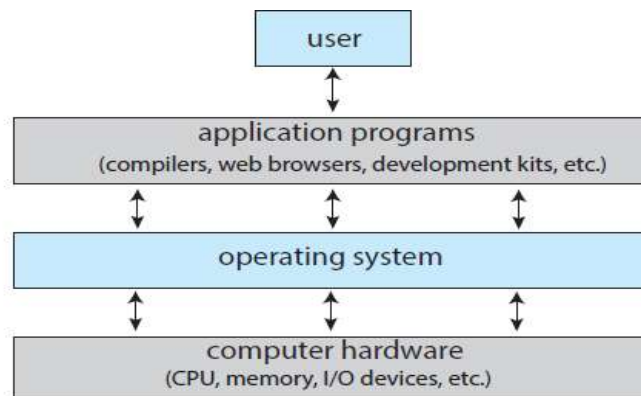


Figure Abstract view of the components of a computer system.

The Hardware

The hardware provides the basic computing resources for the system.

Hardware:

- The central processing unit (CPU),
- The memory,
- The input/output (I/O) devices

The Operating System

The operating system controls the hardware and coordinates its use among the various application programs for the various users.

The Application Programs

The application programs define the ways in which these resources are used to solve users' computing problems.

Application Programs such as

- word processors,
- spreadsheets,
- compilers,
- web browsers

Functions of Operating System

1. Memory Management

It is the management of the main or primary memory. Whatever program is executed, it has to be present in the main memory. Main memory is a quick storage area that may be accessed directly by the CPU. When the program is completed, the memory region is released and can be used by other programs. Therefore, there can be more than one program present at a time. Hence, it is required to manage the memory.

The operating system:

Allocates and deallocates the memory.

- Keeps a record of which part of primary memory is used by whom and how much.
- Distributes the memory while multiprocessing.
- In multiprogramming, the operating system selects which processes acquire memory when and how much memory they get.

2. Processor Management/Scheduling

Every software that runs on a computer, whether in the background or in the frontend, is a process. Processor management is an execution unit in which a program operates. The operating system determines the status of the processor and processes, selects a job and its processor, allocates the processor to the process, and de-allocates the processor after the process is completed.

When more than one process runs on the system the OS decides how and when a process will use the CPU. Hence, the name is also **CPU Scheduling**. The OS:

- Allocates and deallocates processor to the processes.
- Keeps record of CPU status.

Certain algorithms used for CPU scheduling are as follows:

- First Come First Serve (FCFS)
- Shortest Job First (SJF)
- Round-Robin Scheduling
- Priority-based scheduling etc.

Purpose of CPU scheduling

The purpose of CPU scheduling is as follows:

- Proper utilization of CPU. Since the proper utilization of the CPU is necessary. Therefore, the OS makes sure that the CPU should be as busy as possible.
- Since every device should get a chance to use the processor. Hence, the OS makes sure that the devices get fair processor time.
- Increasing the efficiency of the system.

3. Device Management

An operating system regulates device connection using drivers. The processes may require devices for their use. This management is done by the OS. The OS:

- Allocates and deallocates devices to different processes.
- Keeps records of the devices.

- Decides which process can use which device for how much time.

4. File Management

The operating system manages resource allocation and de-allocation. It specifies which process receives the file and for how long. It also keeps track of information, location, uses, status, and so on. These groupings of resources are referred to as file systems. The files on a system are stored in different directories. The OS:

- Keeps records of the status and locations of files.
- Allocates and deallocates resources.
- Decides who gets the resources.

5. Storage Management

Storage management is a procedure that allows users to maximize the utilization of storage devices while also protecting data integrity on whatever media on which it lives. Network virtualization, replication, mirroring, security, compression, deduplication, traffic analysis, process automation, storage provisioning, and memory management are some of the features that may be included. The operating system is in charge of storing and accessing files. The creation of files, the creation of directories, the reading and writing of data from files and directories, as well as the copying of the contents of files and directories from one location to another are all included in storage management.

The OS uses storage management for:

- Improving the performance of the data storage resources.
- It optimizes the use of various storage devices.
- Assists businesses in storing more data on existing hardware, speeding up the data retrieval process, preventing data loss, meeting data retention regulations, and lowering IT costs

What are the functions of Operating System

- **Security** – For security, modern operating systems employ a firewall. A firewall is a type of security system that monitors all computer activity and blocks it if it detects a threat.
- **Job Accounting** – As the operating system keeps track of all the functions of a computer system. Hence, it makes a record of all the activities taking place on the system. It has an account of all the information about the memory, resources, errors, etc. Therefore, this information can be used as and when required.

- **Control over system performance** – The operating system will collect consumption statistics for various resources and monitor performance indicators such as reaction time, which is the time between requesting a service and receiving a response from the system.
- **Error detecting aids** – While a computer system is running, a variety of errors might occur. Error detection guarantees that data is delivered reliably across susceptible networks. The operating system continuously monitors the system to locate or recognize problems and protects the system from them.
- **Coordination between other software and users** – The operating system (OS) allows hardware components to be coordinated and directs and allocates assemblers, interpreters, compilers, and other software to different users of the computer system.

Booting process – The process of starting or restarting a computer is referred to as Booting. Cold booting occurs when a computer is totally turned off and then turned back on. Warm booting occurs when the computer is restarted. The operating system (OS) is in charge of booting the computer.

OPERATING SYSTEM OPERATIONS

For a computer to start running, for instance, when it is powered up or rebooted, it needs to have an initial program to run. This initial program or bootstrap program initializes all aspects of the system, from CPU registers to device controllers to memory contents. The bootstrap program must locate the operating-system kernel and load it into memory. Once the kernel is loaded and executing, it can start providing services to the system and its users.

Some of the main operations that are performed by Operating Systems are

- **Multiprogramming And Multitasking**
- **Dual Mode And Multimode Operation**
- **Timer**

MULTIPROGRAMMING AND MULTITASKING

One of the most important aspects of operating systems is the ability to run multiple programs, as a single program cannot, in general, keep either the CPU or the I/O devices busy at all times. **Multiprogramming** increases CPU utilization, as well as keeping users satisfied, by organizing programs so that the CPU always has one to execute.

In a multiprogrammed system, a program in execution is termed a **process**. The operating system keeps several processes in memory simultaneously. The operating system picks and begins to execute one of these processes.



Figure : Memory layout for a multiprogramming system.

Multitasking is a logical extension of multiprogramming. In multitasking systems, the CPU executes multiple processes by switching among them, but the switches occur frequently, providing the user with a fast **response time**. Consider that when a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O.

DUAL MODE AND MULTIMODE OPERATION

The operating system and its users share the hardware and software resources of the computer system. A properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs or the operating system itself to execute incorrectly. In order to ensure the proper execution of the system, we must be able to distinguish between the execution of operating-system code and user-defined code.

We need two separate modes of operation:

- **user mode**
- **kernel mode** (also called supervisor mode, system mode, or privileged mode).

A bit, called the mode bit, is added to the hardware of the computer to indicate the current mode: **kernel (0) or user (1)**.

When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfill the request.

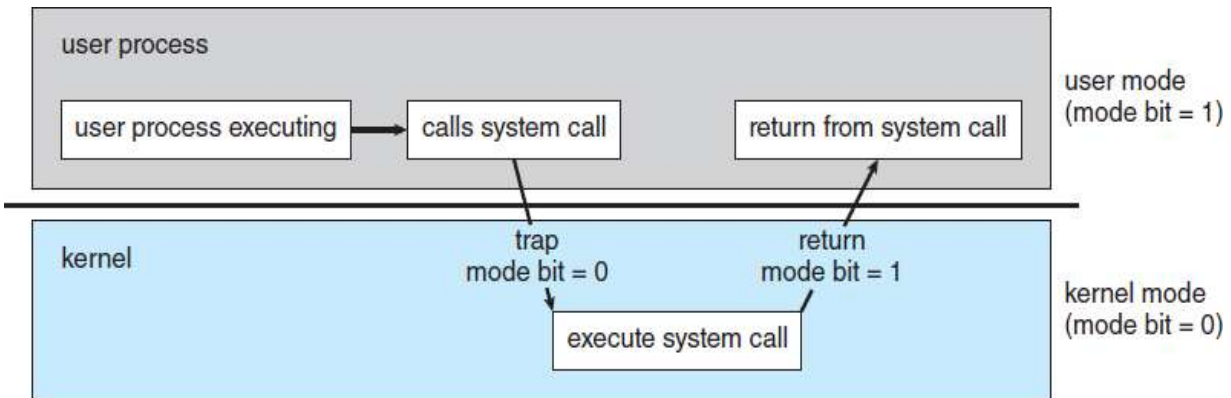


Figure Transition from user to kernel mode.

Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode.

The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0).

Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program. The concept of modes can be extended beyond two modes.

For example, **Intel processors** have four separate **protection rings**, where **ring 0** is **kernel mode** and **ring 3** is **usermode**. Although rings 1 and 2 could be used for various operating-system services, in practice they are rarely used. **ARMv8** systems have seven modes.

TIMER

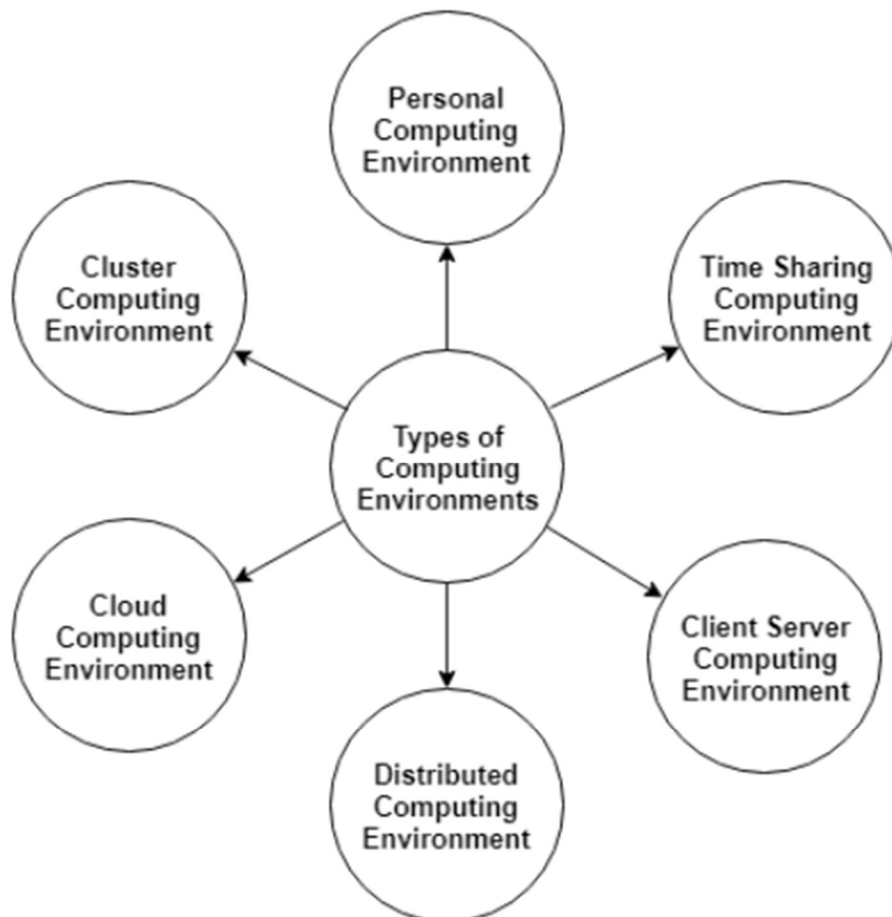
We must ensure that the operating system maintains control over the CPU. We cannot allow a user program to get stuck in an infinite loop or to fail to call system services and never return control to the operating system. To accomplish this goal, we can use a **timer**.

A timer can be set to interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second). A variable timer is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs. Before turning over control to the user, the operating system ensures that the timer is set to interrupt. If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or may give the program more time.

Computing Environments

A computer system uses many devices, arranged in different ways to solve many problems. This constitutes a computing environment where many computers are used to process and exchange information to handle multiple issues.

The different types of Computing Environments are –



Let us begin with Personal Computing Environment –

Personal Computing Environment

In the personal computing environment, there is a single computer system. All the system processes are available on the computer and executed there. The different devices that constitute a personal computing environment are laptops, mobiles, printers, computer systems, scanners etc.

Time Sharing Computing Environment

The time sharing computing environment allows multiple users to share the system simultaneously. Each user is provided a time slice and the processor switches rapidly among the users according to it. Because of this, each user believes that they are the only ones using the system.

Client Server Computing Environment

In client server computing, the client requests a resource and the server provides that resource. A server may serve multiple clients at the same time while a client is in contact with only one server. Both the client and server usually communicate via a computer network but sometimes they may reside in the same system.

Distributed Computing Environment

A distributed computing environment contains multiple nodes that are physically separate but linked together using the network. All the nodes in this system communicate with each other and handle processes in tandem. Each of these nodes contains a small part of the distributed operating system software.

Cloud Computing Environment

The computing is moved away from individual computer systems to a cloud of computers in cloud computing environment. The cloud users only see the service being provided and not the internal details of how the service is provided. This is done by pooling all the computer resources and then managing them using a software.

Cluster Computing Environment

The clustered computing environment is similar to parallel computing environment as

they both have multiple CPUs. However a major difference is that clustered systems are created by two or more individual computer systems merged together which then work parallel to each other.

Free and open-source operating systems (OS):

1. Linux (Various Distributions)

Overview: Linux is the most widely known free and open-source operating system. It is based on the Linux kernel, and there are many distributions (distros) created by different organizations and communities.

- **Popular Distros:**

- **Ubuntu:** Known for its user-friendly interface and ease of installation, making it ideal for beginners.
- **Debian:** A stable and flexible distro, known for its large software repository.
- **Fedora:** Focuses on cutting-edge technology and innovations.
- **Arch Linux:** A minimalist, rolling-release distribution for advanced users.
- **Linux Mint:** Based on Ubuntu, Mint is designed to be more familiar and easy for users coming from other OSes.
- **Manjaro:** A user-friendly distro based on Arch Linux, with a more accessible installation process.

2. FreeBSD

- **Overview:** FreeBSD is a Unix-like OS that is known for its performance and advanced networking features. It is often used in server environments but can also be used for desktop computing.
- **Key Features:** Excellent security, performance, and scalability, with an extensive ports collection for software management.

3. OpenBSD

- **Overview:** OpenBSD focuses on security, correctness, and code simplicity. It is often used in security-sensitive applications like firewalls and VPNs.
- **Key Features:** Strong security features, regular audits, and a secure default configuration.

4. NetBSD

- **Overview:** Known for its portability across many different hardware platforms, NetBSD is one of the oldest open-source operating systems.

- **Key Features:** Works on a wide range of architectures, from embedded systems to large servers.

5. Haiku

- **Overview:** Haiku is an open-source operating system inspired by BeOS. It aims to be simple, fast, and efficient, with an emphasis on user experience.
- **Key Features:** A focus on multimedia, simplicity, and a modern desktop environment.

6. ReactOS

- **Overview:** ReactOS is an open-source operating system that is binary-compatible with Windows. Its goal is to provide an open-source alternative to Windows with a similar interface and support for Windows applications.
- **Key Features:** Compatible with Windows software and drivers, still under development but usable for some tasks.

7. GNU Hurd

- **Overview:** GNU Hurd is the kernel of the GNU operating system, designed to be a free and open-source alternative to proprietary kernels like Linux. It's still in development and is not as widely used as Linux.
- **Key Features:** Uses a microkernel architecture with a focus on flexibility and security.

8. Alpine Linux

- **Overview:** Alpine Linux is a security-oriented, lightweight Linux distribution that is commonly used in Docker containers.
- **Key Features:** Small footprint, security features like PaX and grsecurity, often used for minimal server setups.

9. PureOS

- **Overview:** PureOS is a privacy-focused operating system based on Debian. It is designed to protect your privacy, freedom, and security.
- **Key Features:** Uses privacy-focused apps, such as DuckDuckGo for search and Tor for browsing.

10. Solus

- **Overview:** Solus is an independent, rolling release Linux distribution designed for home computing.
- **Key Features:** Focused on simplicity and ease of use, comes with the Budgie desktop environment, and has its own software center.

11. Tails

- **Overview:** Tails (The Amnesic Incognito Live System) is a privacy-focused OS designed to run from a USB stick. It is often used for anonymous browsing and communication.
- **Key Features:** Runs entirely from RAM, leaves no trace on the computer, uses Tor for anonymity.

12. Ubuntu Server

- **Overview:** A version of Ubuntu specifically designed for server environments, with a strong focus on cloud computing and high-performance computing.
- **Key Features:** Easy installation, wide hardware support, and extensive documentation for web hosting, cloud, and enterprise solutions.

13. Xubuntu

- **Overview:** A lightweight variant of Ubuntu, Xubuntu uses the XFCE desktop environment, which is optimized for speed and low resource usage.
- **Key Features:** Efficient and lightweight, making it ideal for older hardware or users seeking a simple desktop environment.

14. Zorin OS

- **Overview:** Zorin OS is a beginner-friendly Linux distribution that is visually similar to Windows, making it an easy transition for those switching from Microsoft's OS.
- **Key Features:** Offers a "Windows-like" interface, includes built-in software, and provides compatibility with Windows applications.

Each of these operating systems is free to use, and their open-source nature allows users to inspect, modify, and distribute the software. Depending on your needs—whether for desktop, server, security, or privacy—there's likely a suitable option among these.

OPERATING SYSTEM SERVICES

An operating system provides an environment for the execution of programs. It makes certain services available to programs and to the users of those programs. The specific services provided differ from one operating system to another, but we can identify common classes.

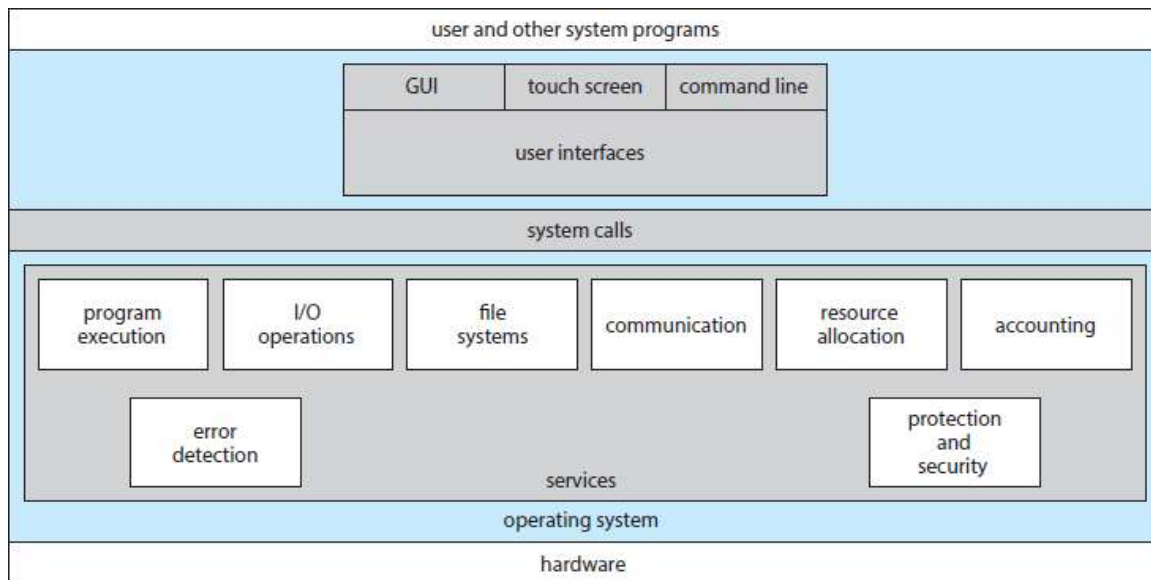


Figure A view of operating system services.

Figure shows one view of the various operating-system services and how they interrelate. These services also make the programming task easier for the programmer. One set of operating system services provides functions that are helpful to the user.

The services provided by operating systems are

- **User Interface**
- **Program Execution**
- **I/O Operations**
- **File system manipulation**
- **Communications**
- **Error**

detection

USER

INTERFACE

Almost all operating systems have a User Interface (UI). This interface can take several forms. Most commonly, a **Graphical User Interface (GUI)** is used. Here, the interface is a window system with a mouse that serves as a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text.

Mobile systems such as phones and tablets provide a **Touch-screen Interface**, enabling users to slide their fingers across the screen or press buttons on the screen to select choices.

Another option is a **Command-Line Interface (CLI)**, which uses text commands and a method for entering them. Some systems provide two or all three of these variations.

PROGRAM EXECUTION

The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).

I/O OPERATIONS

A running program may require I/O, which may involve a file or an I/O device. For specific devices, special functions may be desired (such as reading from a network interface or writing to a file system). 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

Obviously, programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information. Finally, some operating systems include permissions management to allow or deny access to files or directories based on file ownership. Many operating systems provide a variety of file systems, sometimes to allow personal choice and sometimes to provide specific features or performance characteristics.

COMMUNICATIONS

There are many circumstances in which one process needs to exchange information with another process.

Such communication may occur between

- processes that are executing on the same computer
- processes that are executing on different computer systems tied together by a network.

Communications may be implemented

- via **shared memory**, in which two or more processes read and write to a shared section of memory,
- **message passing**, in which packets of information in predefined formats are moved between processes by the operating system.

ERROR DETECTION

The operating system needs to be detecting and correcting errors constantly. Errors may occur in

- **The CPU and Memory Hardware** such as a memory error or a power failure etc.,
- **I/O devices** such as a parity error on disk, a connection failure on a network, or lack of paper in the printer
- **The user program** such as an arithmetic overflow or 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. Sometimes, it has no choice but to halt the system. At other times, it might terminate an error-causing process or return an error code to a process for the process to detect and possibly correct.

USER AND OPERATING SYSTEM INTERFACE

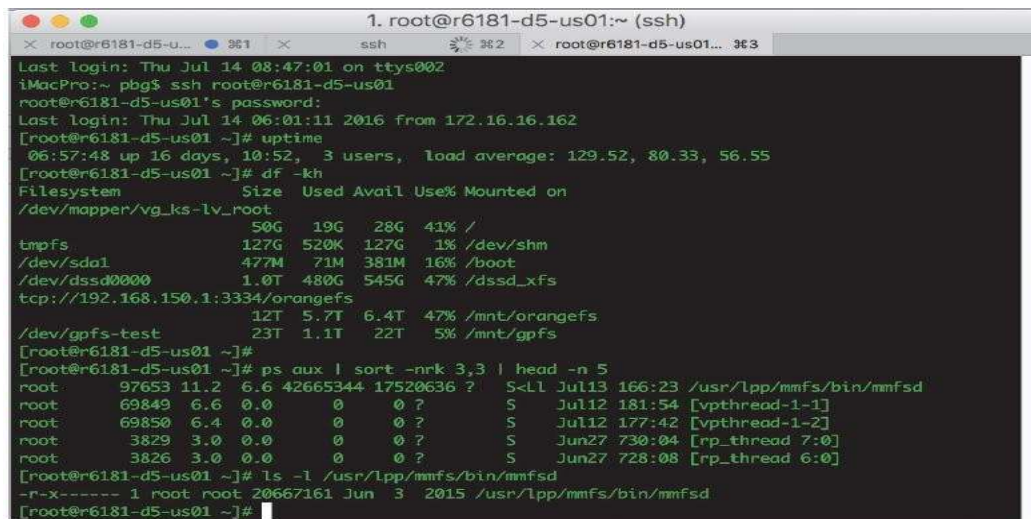
There are several ways for users to interface with the operating system. The three fundamental approaches are

- **COMMAND INTERPRETERS**
- **GRAPHICAL USER INTERFACE**
- **TOUCH SCREEN INTERFACE**

COMMAND INTERPRETERS

Command-line interface, or command interpreter, allows users to directly enter commands to be performed by the operating system. Most operating systems, including Linux, UNIX, and Windows, treat the command interpreter as a special program that is running when a process 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**, a user may choose among several different shells, including the **C shell**, **Bourne-Again shell**, **Korn shell**, and others. Figure shows the Bourne-Again (or bash) shell command interpreter being used on macOS.



```
1. root@r6181-d5-us01:~ (ssh)
Last login: Thu Jul 14 08:47:01 on ttys002
iMacPro:~ pbg$ ssh root@r6181-d5-us01
root@r6181-d5-us01's password:
Last login: Thu Jul 14 06:01:11 2016 from 172.16.16.162
[root@r6181-d5-us01 ~]# uptime
 06:57:48 up 16 days, 10:52,  3 users,  load average: 129.52, 80.33, 56.55
[root@r6181-d5-us01 ~]# df -kh
Filesystem      Size  Used Avail Use% Mounted on
/dev/mapper/vg_ks-lv_root
50G      19G   28G   41% /
tmpfs           127G  520K  127G   1% /dev/shm
/dev/sda1       477M   71M  381M  16% /boot
/dev/dssd00000  1.0T  480G  545G  47% /dssd_xfs
tcp://192.168.150.1:3334/orangefs
12T   5.7T   6.4T   47% /mnt/orangefs
/dev/gpfs-test  23T   1.1T   22T    5% /mnt/gpfs
[root@r6181-d5-us01 ~]#
[root@r6181-d5-us01 ~]# ps aux | sort -nrk 3,3 | head -n 5
root    97653 11.2  6.6 42665344 17520636 ?    S<ll  Jul13 166:23 /usr/lpp/mmfs/bin/mmfdsd
root    69849  6.6  0.0      0      0 ?        S    Jul12 181:54 [vpthread-1-1]
root    69850  6.4  0.0      0      0 ?        S    Jul12 177:42 [vpthread-1-2]
root    3829   3.0  0.0      0      0 ?        S    Jun27 730:04 [rp_thread 7:0]
root    3826   3.0  0.0      0      0 ?        S    Jun27 728:08 [rp_thread 6:0]
[root@r6181-d5-us01 ~]# ls -l /usr/lpp/mmfs/bin/mmfdsd
-r-x----- 1 root root 20667161 Jun  3  2015 /usr/lpp/mmfs/bin/mmfdsd
[root@r6181-d5-us01 ~]#
```

Figure The bash shell command interpreter in macOS.

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 various shells available on UNIX systems operate in this way. These commands can be implemented in two general ways.

In one approach, the **command interpreter itself contains the code to execute the command**. For example, a command to delete a file may cause the command interpreter to jump to a section of its code that sets up the parameters and makes the appropriate system call. In this case, the number of commands that can be given determines the size of the command interpreter, since each command requires its own implementing code.

An alternative approach used by UNIX, among other operating systems, implements most commands through system programs. In this case, the command interpreter does not understand the command in any way; it merely uses the command to identify a file to be loaded into memory and executed. In this way, programmers can add

new commands to the system easily by creating new files with the proper program logic. The command-interpreter program, which can be small, does not have to be changed for new commands to be added.

GRAPHICAL USER INTERFACE

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, users employ a mouse-based window and menu system characterized by a **desktop**.

The user moves the mouse to position its pointer on images, or icons, on the screen (the desktop) that represent programs, files, directories, and system functions. Depending on the mouse pointer's location, clicking a button on the mouse can invoke a program, select a file or directory known as a folder or pull down a menu that contains commands.

The first GUI appeared on the Xerox Alto computer in 1973. However, graphical interfaces became more widespread with the advent of Apple Macintosh computers in the 1980s. The user interface for the Macintosh operating system has undergone various changes over the years, the most significant being the adoption of the **Aqua interface** that appeared with macOS.

TOUCH SCREEN INTERFACE

Because a either a command-line interface or a mouse-and-keyboard system is impractical for most mobile systems. **Smartphones** and handheld **Tablet Computers** typically use a touch-screen interface. Here, users interact by making **gestures on the touch screen**, for example, pressing and swiping fingers across the screen.

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. In contrast, most Windows users are happy to use the Windows GUI environment and almost never use the shell interface.

SYSTEM CALLS

A system call is a programmatic way in which a computer program requests a service from the operating system's kernel. These services might include file manipulation, process control, inter-process communication, or managing devices like printers and disks.

TYPES OF 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 (for example, tasks where hardware must be accessed directly) may have to be written using assembly-language instructions.

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.

cp in.txt out.txt → UNIX cp command

This command copies the input file in.txt to the output file out.txt.

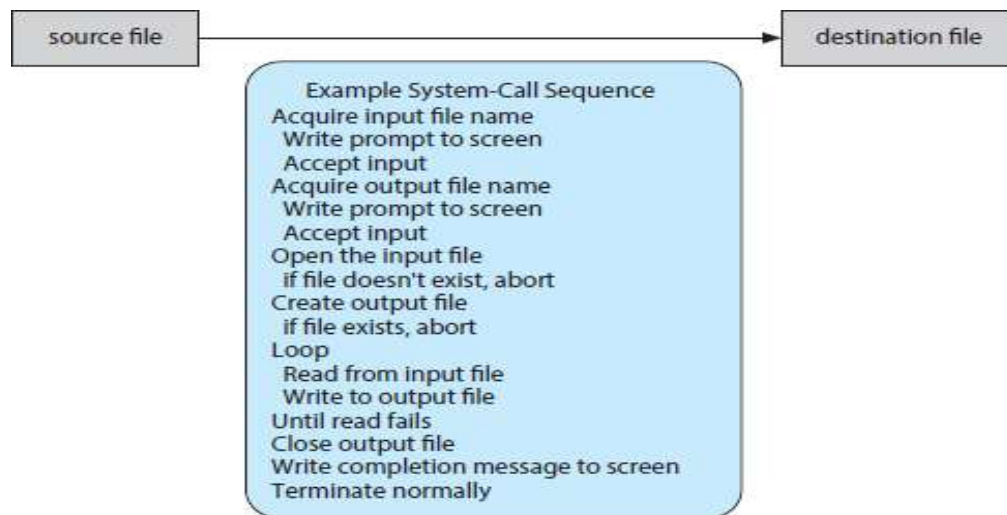


Figure Example of how system calls are used.

System calls can be grouped roughly into six major categories:

- **Process control**
- **File management**
- **Device management**
- **Information maintenance**
- **Communications**
- **Protection**

Process control

Process control is the system call that is used to direct the processes. Some process control examples include creating, load, abort, end, execute, process, terminate the process, etc.

Process control functions

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

File Management

File management system calls handle file manipulation jobs like creating a file, reading, and writing, etc.

File management functions

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

Device Management

Device management does the job of device manipulation like reading from device buffers, writing into device buffers, etc.

Device management functions

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

Information Maintenance

It handles information and its transfer between the OS and the user program.

Information Maintenance functions

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

Communications

These types of system calls are specially used for inter-process communications.

Communications functions

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

Protection

Protection provides a mechanism for controlling access to the resources provide by a computer system.

Protection functions

- get file permissions
- set file permissions

SYSTEM PROGRAMS

In the logical computer hierarchy, at the lowest level is hardware. Next is the operating system, then the system services, and finally the application programs. System services, also known as system utilities, provide a convenient environment for program development and execution.

System programs can be divided into categories:

- **File Management**
- **Status Information**
- **File Modification**
- **Programming Language support**
- **Program loading and execution**
- **Communication**
- **Background Service**

FILE MANAGEMENT

These programs create, delete, copy, rename, print, list, and generally access and manipulate files and directories.

STATUS INFORMATION

Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Others are more complex, providing detailed performance, logging, and debugging information.

Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI. Some systems also support a registry, which is used to store and retrieve configuration information.

FILE MODIFICATIONS

Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.

PROGRAMMING-LANGUAGE SUPPORT

Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, and Python) are often provided with the operating system or available as a separate download.

PROGRAM LOADING AND EXECUTION

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

COMMUNICATIONS

These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another's screens, to browse web pages, to send e-mail messages, to log in remotely, or to 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, while others continue to run until the system is halted. Constantly running system-program processes are known as **services, subsystems, or daemons**.

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.

OPERATING SYSTEM DESIGN AND IMPLEMENTATION

There are problems faced in designing and implementing an operating system. But there are, of course, no complete solutions to such problems, but there are approaches that have proved successful.

DESIGN GOALS

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** like traditional desktop/laptop, mobile, distributed, or real time. 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**
- **System goals**

Users want certain obvious properties in a system. The system should be convenient to use, easy to learn and to use, reliable, safe, and fast. Of course, these specifications are not particularly useful in the system design, since there is no general agreement on how to achieve them.

A similar set of requirements can be defined by the developers who must design, create, maintain, and operate the system. The system should be easy to design, implement, and maintain and it should be flexible, reliable, error free, and efficient. Again, these requirements are vague and may be interpreted in various ways.

There is, in short, no unique solution to the problem of defining the requirements for an operating system. The wide range of systems in existence shows that different requirements can result in a large variety of solutions for different environments. Specifying and designing an operating system is a highly creative task.

MECHANISMS AND POLICIES

One important principle is the separation of policy from mechanism. **Mechanisms** determine how to do something; **Policies** determine what will be done. For example, the timer construct is a mechanism for ensuring CPU protection, but deciding how long the timer is to be set for a particular user is a policy decision.

The separation of policy and mechanism is important for flexibility. Policies are likely to change across places or over time. In the worst case, each change in policy would require a change in the underlying mechanism.

Microkernel-based operating systems take the separation of mechanism and policy to one extreme by implementing a basic set of primitive building blocks. These blocks are almost policy free, allowing more advanced mechanisms and policies to be added via user-created kernel modules or user programs themselves.

In contrast, consider **Windows**, Microsoft has closely encoded both mechanism and policy into the system to enforce a global look and feel across all devices that run the Windows operating system.

IMPLEMENTATION

Once an operating system is designed, it must be implemented. Because operating systems are **collections of many programs**, written by many people over a long period of time, it is difficult to make general statements about how they are implemented.

Early operating systems were written in assembly language. Now, most are written in higher-level languages such as C or C++, with small amounts of the system written in assembly language. The lowest levels of the kernel might be written in assembly language and C.

Higher-level routines might be written in C and C++, and system libraries might be written in C++ or even higher-level languages. Android provides a nice example: its **kernel** is written mostly in C with some **assembly language**.

Most Android **system libraries** are written in **C or C++**, and its **application frameworks** which provide the developer interface to the system are written mostly in **Java**. The **advantages** of using a higher-level language, or at least a systems implementation language, for implementing operating systems 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.

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, or modules, rather than have one single system. Each of these modules should be a well-defined portion of the system, with carefully defined interfaces and functions.

These modules can then be arranged in various architectural configurations. They are

- **Monolithic Structure**
- **Layered Approach**
- **Microkernels**
- **Hybrid Systems**
- **macOS and iOS**
- **Android**

MONOLITHIC STRUCTURE

The simplest structure for organizing an operating system is no structure at all. That is, place all of the functionality of the kernel into a single, static binary file that runs in a single address space. This approach is known as a **Monolithic Structure** and is a common technique for designing operating systems.

An example of such limited structuring is the original **UNIX operating system**, which consists of two separable parts:

- **The Kernel**
- **The System Programs**

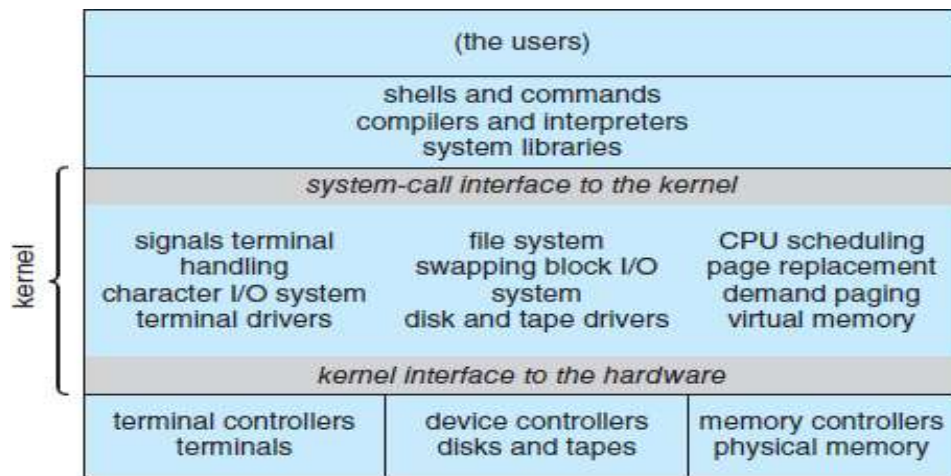


Figure Traditional UNIX system structure.

The kernel is further separated into a series of interfaces and device drivers. Everything below the system-call interface and above the physical hardware is the kernel. The kernel provides the **file system**, **CPU scheduling**, **memory management**, and other **operating system functions** through **system calls**. Taken in sum, that is an enormous amount of functionality to be combined into one single address space.

The **Linux Operating System** is based on UNIX and is structured similarly as shown in the figure. Applications typically use the **glibc standard C library** when communicating with the system call interface to the kernel. The Linux kernel is monolithic in that it runs entirely in kernel mode in a single address space.

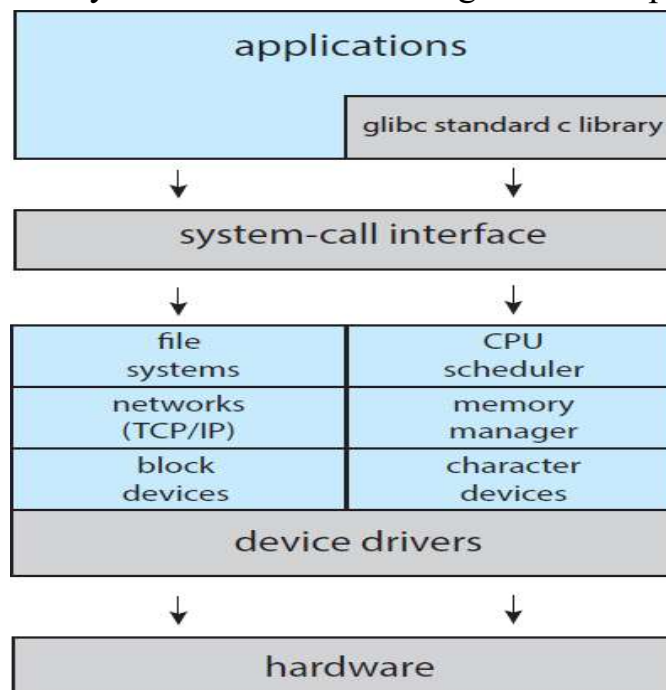


Figure Linux system structure.

Despite the apparent simplicity of monolithic kernels, they are difficult to implement and extend. Monolithic kernels do have a distinct performance advantage, i.e., there is very little overhead in the system-call interface, and communication within the kernel is fast. Therefore, despite the drawbacks of monolithic kernels, their speed and efficiency explains why we still see evidence of this structure in the UNIX, Linux, and Windows operating systems.

LAYERED APPROACH

The **Monolithic Approach** is often known as a **tightly coupled system** because changes to one part of the system can have wide-ranging effects on other parts. Alternatively, we could design a **loosely coupled system**. Such a system is divided into separate, smaller components that have specific and limited functionality. A system can be made modular in many ways. One method is the **Layered Approach**.

In Layered approach the operating system is broken into a number of layers (levels).

- **The bottom layer (layer 0) is the Hardware**
- **The highest layer (layer N) is the User interface**

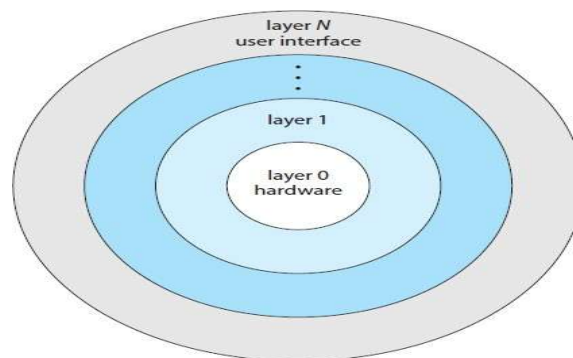


Figure A layered operating system.

An operating-system layer is an **implementation** of an abstract object made up of data and the operations that can manipulate those data. A typical operating-system layer for example say, layer M consists of data structures and a set of functions that can be invoked by higher- level layers. Layer M, in turn, can invoke operations on lower-level layers.

The main advantage of the layered approach is simplicity of construction and debugging. The overall performance of such systems is poor due to the overhead of requiring a user program to traverse through multiple layers to obtain an operating-system service. Layered systems have been successfully used in computer networks and web applications.

MICRO KERNELS

As UNIX expanded, the kernel became large and difficult to manage. Researchers at Carnegie Mellon University developed an operating system called Mach that modularized the kernel using the microkernel approach.

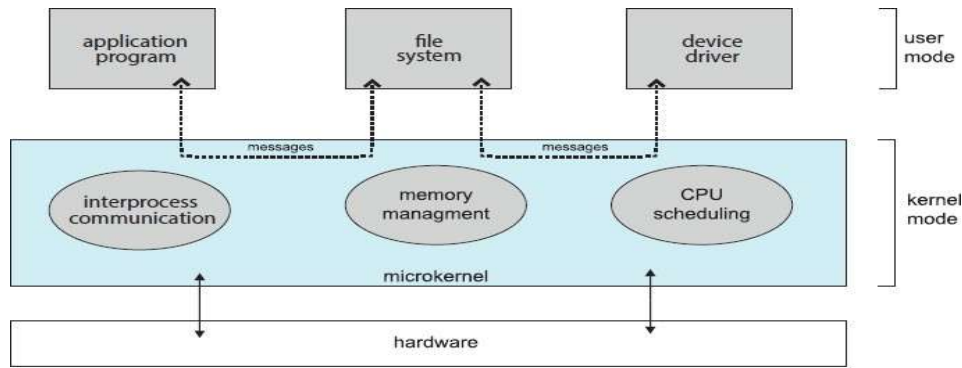


Figure Architecture of a typical microkernel.

This method structures the operating system by removing all nonessential components from the kernel and implementing them as user level programs that reside in separate address spaces. Micro kernels provide minimal process and memory management, in addition to a communication facility.

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. For example, if the client program wishes to access a file, it must interact with the file server. The client program and service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel.

The performance of micro kernels can suffer due to increased system-function overhead. When two user-level services must communicate, messages must be copied between the services, which reside in separate address spaces.

HYBRID SYSTEMS

In practice, very few operating systems adopt a single, strictly defined structure. Instead, they combine different structures, resulting in hybrid systems that address **performance, security, and usability issues**. For example, **Linux** is monolithic, because having the operating system in a single address space provides very efficient performance. However, it also modular so that new functionality can be dynamically added to the kernel. **Windows** is largely monolithic as well but it retains some behavior typical of microkernel systems.

macOS and iOS

Apple's **macOS** operating system is designed to run primarily on **desktop and laptop** computer systems. Whereas **iOS** is a mobile operating system designed for the **iPhone Smartphone** and **iPad tablet** computer.

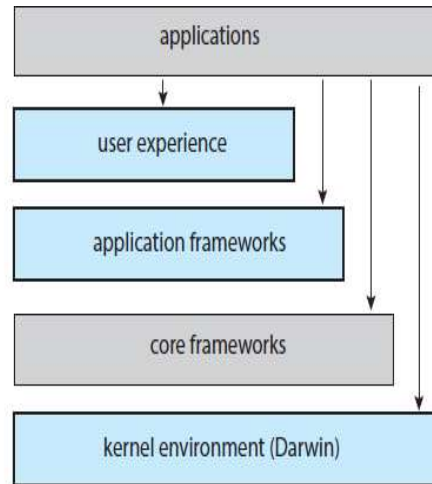


Figure Architecture of Apple's macOS and iOS operating systems.

Darwin is a layered system that consists primarily of the Mach microkernel and the BSD UNIX kernel.

Darwin provides two system-call interfaces:

- **Mach system calls (known as traps)**
- **BSD system calls (provide POSIX functionality)**

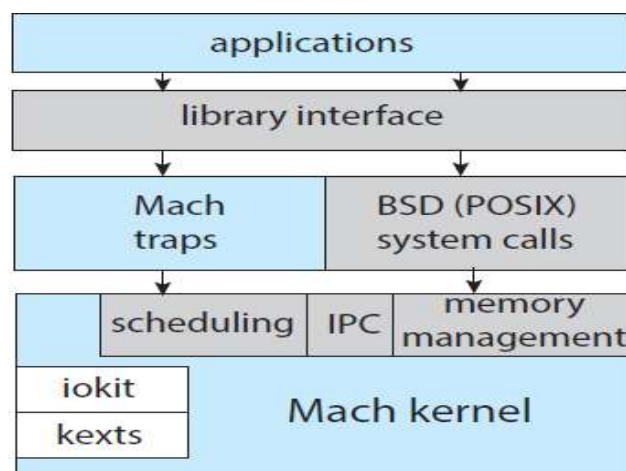


Figure The structure of Darwin.

ANDROID

The Android operating system was designed by the **Open Handset Alliance** (led primarily by Google) and was developed for Android smartphones and tablet computers. Android is similar to iOS in that it is a layered stack of software that provides a rich set of frameworks supporting graphics, audio, and hardware features.

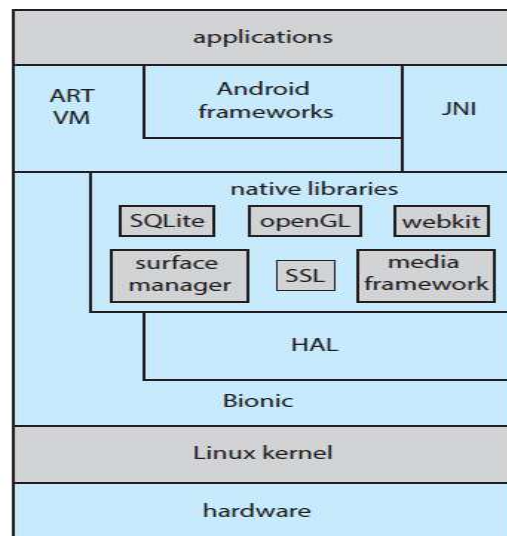


Figure Architecture of Google's Android.

Google has designed a separate Android API for Java development. Java applications are compiled into a form that can execute on the **Android RunTime ART**. Because Android can run on an almost unlimited number of hardware devices, Google has chosen to abstract the physical hardware through the **Hardware Abstraction Layer HAL**. Google has modified the Linux kernel used in Android in a variety of areas to support the special needs of mobile systems, such as power management.

Building and Booting an Operating System

Building and booting an operating system (OS) is a complex and fascinating process. It involves several stages, from designing the OS's components to creating a bootloader that initializes the system. Below is an overview of the key steps involved in building and booting an operating system:

1. Designing the Operating System

Before writing any code, it's essential to plan and design your OS. Consider the following:

- **Target Architecture:** Decide on the CPU architecture (e.g., x86, ARM) you want your OS to run on.

- **Purpose:** Is it for general computing, embedded systems, real-time applications, or something else?
- **Kernel Type:** Decide if you want a monolithic kernel (everything in one) or a microkernel (small, modular).
- **File System:** Choose or design a file system (e.g., ext4, FAT, or a custom one).
- **Device Drivers:** Consider what hardware your OS will support (e.g., storage devices, networking, graphics).
- **User Interface:** Will your OS have a graphical user interface (GUI) or will it be command-line based?

2. Setting Up a Development Environment

To build an OS, you'll need a suitable development environment:

- **Cross-Compiler:** When building an OS, you may need a cross-compiler to generate code that can run on your target architecture (for example, if you're developing for x86 but working on a different platform).
- Example: **GCC** (GNU Compiler Collection) or **Clang** can be configured for cross-compilation.
- **Assembler:** You'll need an assembler to write low-level assembly code that interacts with the hardware.
- Example: **NASM** (Netwide Assembler) or **GAS** (GNU Assembler).
- **Linker:** This tool will combine your object files into an executable binary.
- Example: **LD** (GNU Linker).
- **Emulator/Virtual Machine:** Before booting on real hardware, you'll need an emulator or virtual machine (VM) to test your OS.
- Examples: **QEMU**, **Bochs**, or **VirtualBox**.

3. Writing the Kernel

The **kernel** is the core of the OS. It manages hardware resources, handles system calls, and provides an interface for user applications. The steps for creating the kernel include:

- **Bootloader:** The first step in booting an OS is the bootloader, which loads the kernel into memory. Popular bootloaders include **GRUB** (Grand Unified Bootloader) and **LILLO** (Linux Loader).
- **Kernel Initialization:** The kernel is responsible for initializing hardware, setting up memory management, and starting processes.
 - **Memory Management:** Set up paging or segmentation for managing memory.
 - **Interrupt Handling:** Handle hardware interrupts (e.g., keyboard input, timer).
 - **Device Drivers:** Develop drivers for devices such as storage, network, and display adapters.

- **System Calls:** Provide the interface for programs to interact with the kernel (e.g., file I/O, process creation).

4. Bootloader and Boot Sequence

A **bootloader** is a program that is executed when the system is powered on. It's responsible for loading the OS kernel into memory and transferring control to it.

Here's a simple boot sequence:

1. **BIOS/UEFI:** On most systems, the **BIOS** (or **UEFI** in modern systems) performs initial hardware checks (POST) and looks for a bootable device (e.g., hard drive, USB).
2. **Bootloader:** Once the BIOS/UEFI finds a bootable device, it loads the bootloader. The bootloader's primary task is to load the OS kernel into memory. It can also load the kernel in a specific location or provide a boot menu.
3. **Kernel Loading:** The bootloader loads the OS kernel into memory and passes control to it. The kernel initializes the system (hardware, memory, and process management).
4. **System Initialization:** The kernel will initialize essential subsystems (e.g., device drivers, file systems) and start user processes.

5. Writing a Bootloader

To write a bootloader, you'll need to:

- Write code that can read the kernel image from the boot device (e.g., hard drive or USB stick).
- Load the kernel into memory (usually in protected mode) and transfer control to it.

A simple bootloader can be written in **assembly language**. Here's an outline of what a basic bootloader does:

- Set up a 16-bit real mode environment (to interact with BIOS).
- Load the kernel into memory.
- Switch to protected mode (32-bit or 64-bit) and jump to the kernel code.

You can also use a pre-made bootloader like **GRUB**, which makes things easier by providing features like multi-OS booting and compatibility with modern filesystems.

6. Building the Kernel and Creating a Bootable Image

Once the kernel and bootloader are written:

- **Compile the Kernel:** Use a cross-compiler or native compiler to compile the kernel code.
- **Link the Kernel:** Link the compiled kernel code to create a bootable binary.

- **Create a Bootable Disk:** Use tools like **dd** to write the kernel and bootloader to a USB stick or a hard drive.

For example:

- **Creating a bootable USB:** You can use commands like **dd** in Linux to write the OS image to a USB stick:

bash

Copy code

```
dd if=your-kernel.img of=/dev/sdX bs=512 seek=4
```

7. Booting the OS

Once everything is set up:

- Insert the bootable device (USB stick, disk image) into your system and reboot.
- If using **GRUB**, you can create a configuration file (e.g., `grub.cfg`) to specify the kernel image location.
- The bootloader will load the kernel, and the OS should start running. The kernel will begin initializing hardware and system resources.

8. Testing and Debugging

At this stage, you should test your OS:

- Use an **emulator** (like **QEMU**) to test without needing physical hardware.
- Check for kernel panics, boot failures, or missing device drivers.
- Add more functionality incrementally, such as process scheduling, file systems, and networking.

Additional Resources:

- **Books:**
 - "Operating Systems: Design and Implementation" by Andrew Tanenbaum.
 - "The Linux Programming Interface" by Michael Kerrisk.
- **Online tutorials:**
 - OSDev Wiki: A popular resource for OS development.
 - Writing a Simple Operating System from Scratch: A tutorial to help you create a simple OS.

Key Takeaways:

1. **Bootloader:** Starts the boot process by loading the kernel into memory.
2. **Kernel:** Manages system resources and provides essential services.
3. **Testing:** Use emulators like QEMU or Bochs for testing before running on real hardware.

Building and booting your own OS is a long-term, iterative process. Each step builds on the previous one, and the more you understand low-level systems programming, the better your OS will perform.

Operating System Debugging

Debugging is the process of finding the problems in a computer system and solving them. There are many different ways in which operating systems perform debugging. Some of these are –

Log Files

The log files record all the events that occur in an operating system. This is done by writing all the messages into a log file. There are different types of log files. Some of these are given as follows –

Event Logs

These stores the records of all the events that occur in the execution of a system. This is done so that the activities of all the events can be understood to diagnose problems.

Transaction Logs

The transaction logs store the changes to the data so that the system can recover from crashes and other errors. These logs are readable by a human.

Message Logs

These logs store both the public and private messages between the users. They are mostly plain text files, but in some cases they may be HTML files.

Core Dump Files

The core dump files contain the memory address space of a process that terminates unexpectedly. The creation of the core dump is triggered in response to program crashes by the kernel. The core dump files are used by the developers to find the program's state at the time of its termination so that they can find out why the termination occurred.

The automatic creation of the core dump files can be disabled by the users. This may be done to improve performance, clear disk space or increase security.

Crash Dump Files

In the event of a total system failure, the information about the state of the operating system is captured in crash dump files. There are three types of dump that can be captured when a system crashes. These are –

Complete Memory Dump

The whole contents of the physical memory at the time of the system crash are captured in the complete memory dump. This is the default setting on the Windows Server System.

Kernel Memory Dump

Only the kernel mode read and write pages that are present in the main memory at the time of the system crash are stored in the kernel memory dump.

Small Memory Dump

This memory dump contains the list of device drivers, stop code, process and thread information, kernel stack etc.



Trace Listings

The trace listing record information about a program execution using logging. This information is used by programmers for debugging. System administrators and technical personnel can use the trace listings to find the common problems with software using software monitoring tools.

Profiling

This is a type of program analysis that measures various parameters in a program such as space and time complexity, frequency and duration of function calls, usage of specific instructions etc. Profiling is done by monitoring the source code of the required system program using a code profiler.



END OF MODULE 1