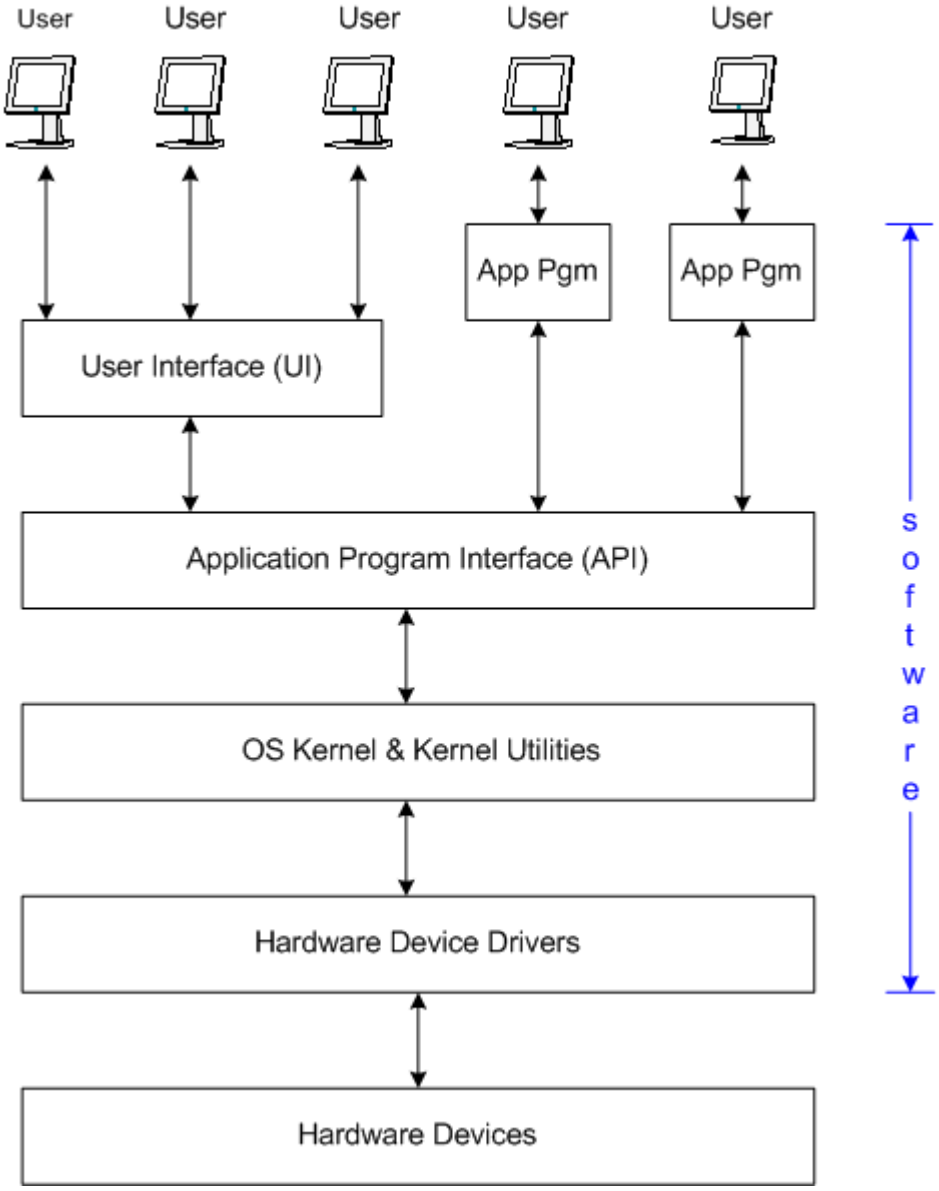
Introduction to OS Organization

Operating Systems today generally consist of many distinct pieces or components. We can simplify our description of an OS by viewing it as many layers of related components. A generic OS layered diagram is pictured below. Note the component at the top of the diagram is the user, and the component at the bottom of the diagram is the physical hardware. Management (control) of all tasks in between these two physical components is the responsibility of the OS.



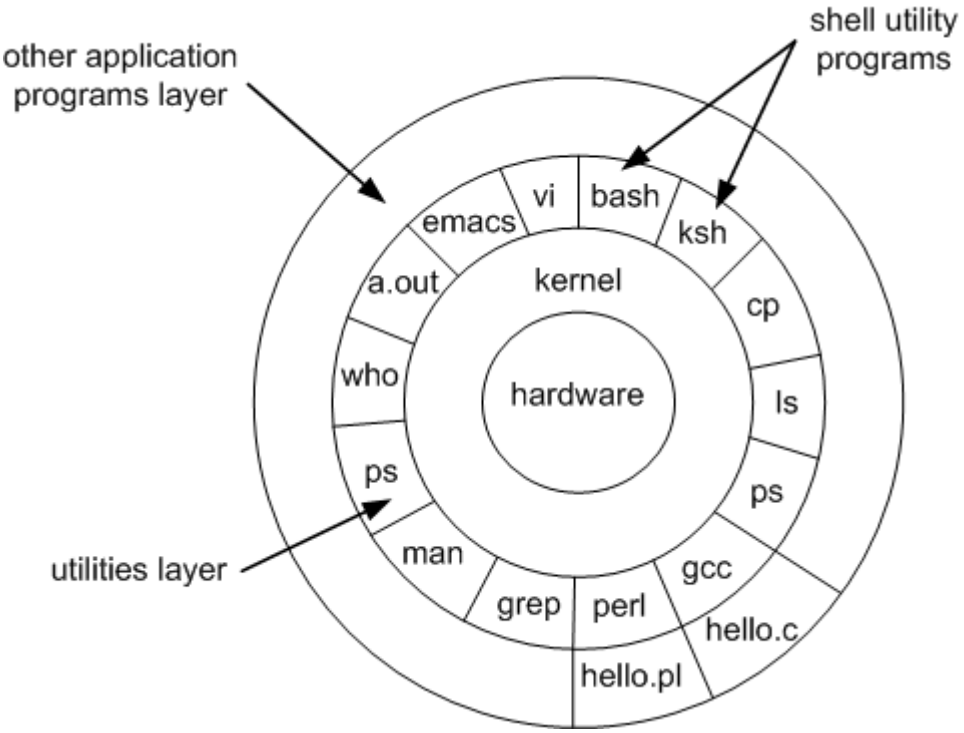
Unix Operating System Organization

As with most modern operating systems, the Unix OS is also madeup of many different components. In a very general sense, Unix is divided into two main components, the kernel component and the utilities. The kernel, which is critical to the operation of the OS, is loaded into Random Access Memory (RAM) by the boot loader, where it remains memory resident for as long as the machine remains powered on.

The utilities are programs which (typically) reside on a disk device (e.g. a harddrive). Individual utillities are loaded into RAM as needed or requested and are discarded from RAM upon completion.

Perhaps the most "important" or well known of the Unix utilitites is known as the Unix shell. The shell is the mechanism which allows users to enter commands to run other utility programs. There are several popular Unix shell programs.

The diagram below provides visual representation of the organization of the Unix OS. At the core of the OS is the hardware, which is managed by the surrounding outer layer, the kernel. In the next outer layer come the utilities. Many of the utilities are system commands, but these can also be user written programs as shown by the a.out program. Finally in the outermost layer are other application programs which can be built on top of lower layer programs.



The Kernel

The kernel communicates to hardware both directly and through drivers.

Just as the kernel abstracts the hardware to user programs, drivers abstract hardware to the kernel. For example there are many different types of graphic card, each one with slightly different features. As long as the kernel exports an API, people who have access to the specifications for the hardware can write drivers to implement that API. This way the kernel can access many different types of hardware.

Monolithic v Microkernels

The monolithic approach is the most common, as taken by most common Unixes (such as Linux). In this model the core privileged kernel is large, containing hardware drivers, file system accesses controls, permissions checking and services such as Network File System (NFS).

Since the kernel is always privileged, if any part of it crashes the whole system has the potential to comes to a halt. If one driver has a bug it can overwrite any memory in the system with no problems, ultimately causing the system to crash.

A microkernel architecture tries to minimise this possibility by making the privileged part of the kernel as small as possible. This means that most of the system runs as unprivileged programs, limiting the harm

that any one crashing component can influence. For example, drivers for hardware can run in separate processes, so if one goes astray it cannot overwrite any memory but that allocated to it.

Whilst this sounds like the most obvious idea, the problem comes back two main issues

1. Performance is decreased. Talking between many different components can decrease performance.

2. It is slightly more difficult for the programmer.

Both of these criticisms come because to keep separation between components most microkernels are implemented with a message passing based system, commonly referred to as inter-process communication or IPC. Communicating between individual components happens via discrete messages which must be bundled up, sent to the other component, unbundled, operated upon, re-bundled up and sent back, and then unbundled again to get the result.

This is a lot of steps for what might be a fairly simple request from a foreign component. Obviously one request might make the other component do more requests of even more components, and the problem can multiply. Slow message passing implementations were largely responsible for the poor performance of early microkernel systems, and the concepts of passing messages are slightly harder for programmers to program for. The enhanced protection from having components run separately was not sufficient to overcome these hurdles in early microkernel systems, so they fell out of fashion.

In a monolithic kernel calls between components are simple function calls, as all programmers are familiar with.

SHELL

A **Shell** provides you with an interface to the Unix system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

Shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of a shell, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

Shell Prompt

The prompt, **$**, which is called the **command prompt**, is issued by the shell. While the prompt is displayed, you can type a command.

Shell reads your input after you press **Enter**. It determines the command you want executed by looking at the first word of your input. A word is an unbroken set of characters. Spaces and tabs separate words.

User mode and kernel mode

A processor in a computer running Windows has two different modes: user mode and kernel mode. The processor switches between the two modes depending on what type of code is running on the processor. Applications run in user mode, and core operating system components run in kernel mode. While many drivers run in kernel mode, some drivers may run in user mode.

When you start a user-mode application, Windows creates a process for the application. The process provides the application with a private virtual address space and a private handle table. Because an application's virtual address space is private, one application cannot alter data that belongs to another application. Each application runs in isolation, and if an application crashes, the crash is limited to that one application. Other applications and the operating system are not affected by the crash.

In addition to being private, the virtual address space of a user-mode application is limited. A processor running in user mode cannot access virtual addresses that are reserved for the operating system. Limiting

the virtual address space of a user-mode application prevents the application from altering, and possibly damaging, critical operating system data.

All code that runs in kernel mode shares a single virtual address space. This means that a kernel-mode driver is not isolated from other drivers and the operating system itself. If a kernel-mode driver accidentally writes to the wrong virtual address, data that belongs to the operating system or another driver could be compromised. If a kernel-mode driver crashes, the entire operating system crashes.

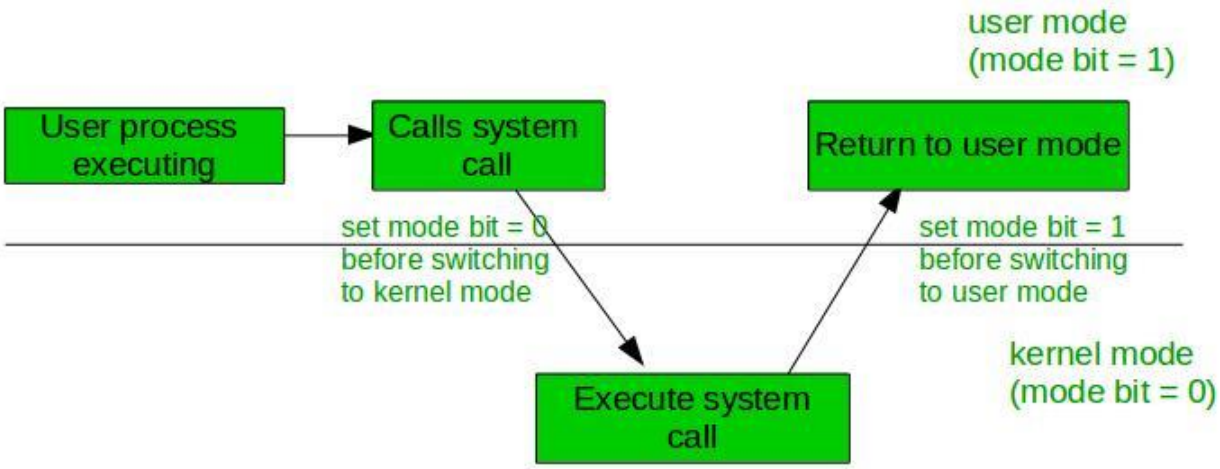
If a process stuck in infinite loop then this infinite loop could affect correct operation of other processes. So to ensure the proper execution of the operating system, there are two modes of operation:

User mode –

When the computer system run user applications like creating a text document or using any application program, then the system is in the user mode. When the user application requests for a service from the operating system or an interrupt occurs or system call, then there will be a transition from user to kernel mode to fulfill the requests.

Note: To switch from kernel mode to user mode, mode bit should be 1.

Given below image describes what happen when an interrupt occurs:



Kernel Mode –

When the system boots, hardware starts in kernel mode and when operating system is loaded, it start user application in user mode. To provide protection to the hardware, we have privileged instructions which execute only in kernel mode. If user attempt to run privileged instruction in user mode then it will treat instruction as illegal and traps to OS. Some of the privileged instructions are:

- Handling Interrupts
- To switch from user mode to kernel mode.
- Input-Output management.

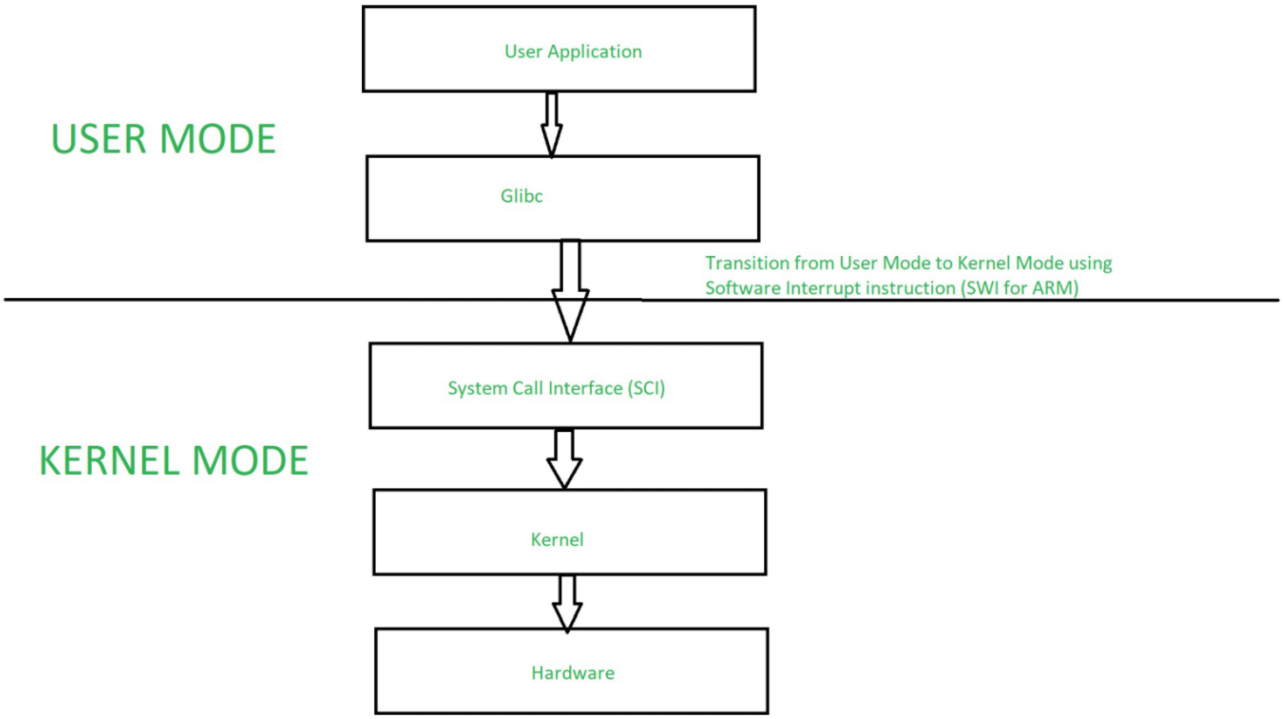
User mode and Kernel mode Switching

In it's life span a process executes in user mode and kernel mode. The User mode is normal mode where the process has limited access. While the Kernel mode is the privileged mode where the process has unrestricted access to system resources like hardware, memory, etc. A process can access I/O Hardware registers to program it, can execute OS kernel code and access kernel data in Kernel mode. Anything

related to Process management, IO hardware management, and Memory management requires process to execute in Kernel mode.

This is important to know that a process in Kernel mode get power to access any device and memory, and same time any crash in kernel mode brings down the whole system. But any crash in user mode brings down the faulty process only.

The kernel provides System Call Interface (SCI), which are the entry points for kernel. System Calls are the only way through which a process can go into kernel mode from user mode. Below diagram explains user mode to kernel mode transition in detail.



To go into Kernel mode, an application process.

- Calls the Glibc library function.
- Glibc library knows the proper way of calling System Call for different architectures. It setup passing arguments as per architecture's Application Binary Interface (ABI) to prepare for System Call entry.
- Now Glibc calls SWI instruction (Software Interrupt instruction for ARM), which puts processor into Supervisor mode by updating Mode bits of CPSR register and jumps to vector address 0x08.
- Till now process execution was in User mode. After SWI instruction execution, the process is allowed to execute kernel code. Memory Management Unit (MMU) will now allow kernel Virtual memory access and execution, for this process.
- From Vector address 0x08, process execution loads and jumps to SW Interrupt handler routine, which is vector_swi() for ARM.
- In vector_swi(), System Call Number (SCNO) is extracted from SWI instruction and execution jumps to system call function using SCNO as index in system call table sys_call_table.
- After System Call execution, in return path, user space registers are restored before starting execution in User Mode.

To support kernel mode and user mode, processor must have hardware support for different privilege modes. For example ARM processor supports seven different modes.

System call

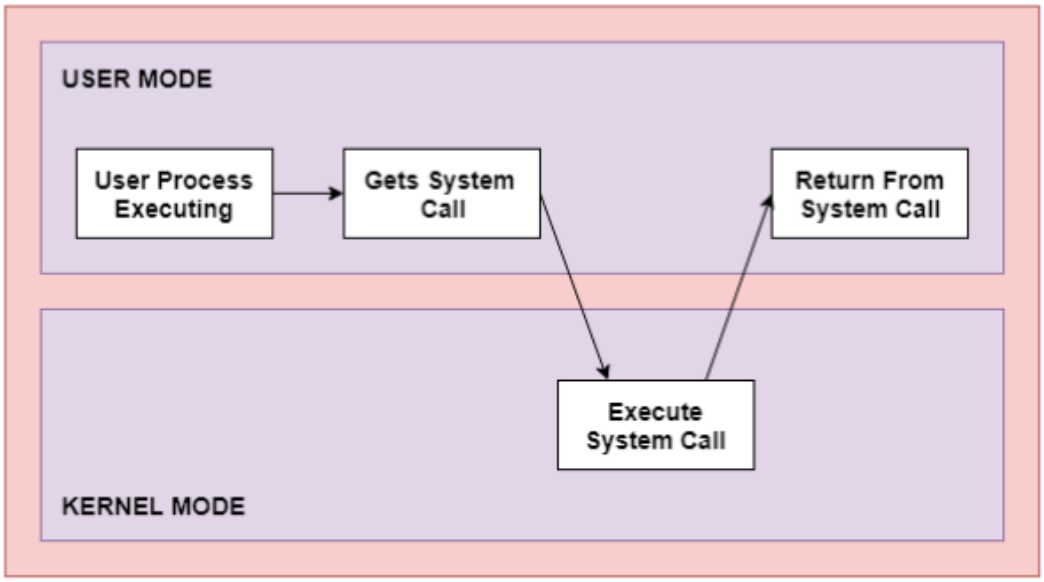
In computing, a system call (commonly abbreviated to syscall) is the programmatic way in which a computer program requests a service from the kernel of the operating system on which it is executed. A system call is a mechanism that provides the interface between a process and the operating system. It is a programmatic method in which a computer program requests a service from the kernel of the OS. This may include hardware-related services (for example, accessing a hard disk drive), creation and execution of new processes, and communication with integral kernel services such as process scheduling. System calls provide an essential interface between a process and the operating system.

System call offers the services of the operating system to the user programs via API (Application Programming Interface). System calls are the only entry points for the kernel system.

Services Provided by System Calls :

- Process creation and management
- Main memory management
- File Access, Directory and File system management
- Device handling(I/O)
- Protection
- Networking, etc.

Here are steps for System Call:



Step 1) The processes executed in the user mode till the time a system call interrupts it.

Step 2) After that, the system call is executed in the kernel-mode on a priority basis.

Step 3) Once system call execution is over, control returns to the user mode.,

Step 4) The execution of user processes resumed in Kernel mode.

In general, system calls are required in the following situations –

- If a file system requires the creation or deletion of files. Reading and writing from files also require a system call.

- Creation and management of new processes.
- Network connections also require system calls. This includes sending and receiving packets.
- Access to a hardware devices such as a printer, scanner etc. requires a system call.

Types of System calls

Here are the five types of system calls used in OS:

- Process Control
- File Management
- Device Management
- Information Maintenance
- Communications
- Protection

Process Control
This system calls perform the task of process creation, process termination, etc.

Functions:

- End and Abort
- Load and Execute
- Create Process and Terminate Process
- Wait and Signed Event
- Allocate and free memory

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

Functions:

- Create a file
- Delete file
- Open and close file
- Read, write, and reposition
- Get and set file attributes

Device Management

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

Functions

- Request and release device
- Logically attach/ detach devices
- Get and Set device attributes

Information Maintenance

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

Functions:

- Get or set time and date
- Get process and device attributes

Communication:

These types of system calls are specially used for interprocess communications.

Functions:

- Create, delete communications connections
- Send, receive message
- Help OS to transfer status information
- Attach or detach remote devices

Protection

It handles the protection and authorization of files, process etc.

Function:

- get/set file permissions

Here are general common rules for passing parameters to the System Call:

- Parameters should be pushed on or popped off the stack by the operating system.
- Parameters can be passed in registers.
- When there are more parameters than registers, it should be stored in a block, and the block address should be passed as a parameter to a register.

Important System Calls Used in OS

wait()

In some systems, a process needs to wait for another process to complete its execution. This type of situation occurs when a parent process creates a child process, and the execution of the parent process remains suspended until its child process executes.

The suspension of the parent process automatically occurs with a wait() system call. When the child process ends execution, the control moves back to the parent process.

fork()

Processes use this system call to create processes that are a copy of themselves. With the help of this system Call parent process creates a child process, and the execution of the parent process will be suspended till the child process executes.

exec()

This system call runs when an executable file in the context of an already running process that replaces the older executable file. However, the original process identifier remains as a new process is not built, but stack, data, head, data, etc. are replaced by the new process.

kill():

The kill() system call is used by OS to send a termination signal to a process that urges the process to exit. However, a kill system call does not necessarily mean killing the process and can have various meanings.

exit():

The exit() system call is used to terminate program execution. Specially in the multi-threaded environment, this call defines that the thread execution is complete. The OS reclaims resources that were used by the process after the use of exit() system call.

Categories	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
Device manipulation	SetConsoleMode() ReadConsole() WriteConsole()	loctl() read() write()
File manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	Open() Read() write() close!)
Information maintanence	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	Pipe() shm_open() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup ()	Chmod() Umask() Chown()

System Programs

The system program serves as a part of the operating system. According to Computer Hierarchy, one which comes at last is Hardware. Then it is Operating System, System Programs, and finally Application Programs. Program Development and Execution can be done conveniently in System Programs. Some of System Programs are simply user interfaces, others are complex. It traditionally lies between user interface and system calls.

So here, user can only view up-to-the System Programs he can't see System Calls.

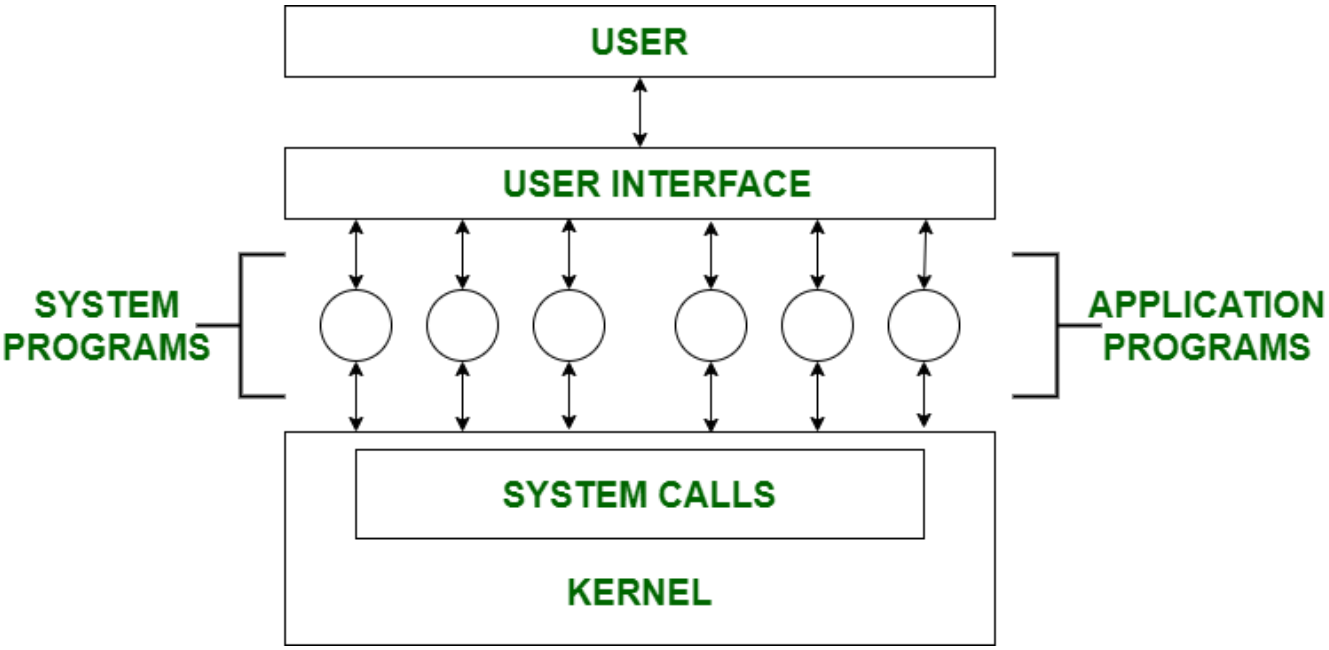
System Programs can be divided into these categories :

File Management –

A file is a collection of specific information stored in memory of computer system. File management is defined as process of manipulating files in computer system, it management includes process of creating, modifying and deleting files.

- It helps to create new files in computer system and placing them at specific locations.
- It helps in easily and quickly locating these files in computer system.
- It makes process of sharing of files among different users very easy and user friendly.
- It helps to stores files in separate folders known as directories.

- These directories help users to search file quickly or to manage files according to their types or uses.
- It helps user to modify data of files or to modify he name of file in directories.



Status Information –

Information like date, time amount of available memory, or disk space is asked by some of users. Others providing detailed performance, logging and debugging information which is more complex. All this information is formatted and displayed on output devices or printed. Terminal or other output devices or files or a window of GUI is used for showing output of programs.

File Modification –

For modifying contents of files we use this. For Files stored on disks or other storage devices we used different types of editors. For searching contents of files or perform transformations of files we use special commands.

Programming-Language support –

For common programming languages we use Compilers, Assemblers, Debuggers and interpreters which are already provided to user. It provides all support to users. We can run any programming languages. All languages of importance are already provided.

Program Loading and Execution –

When program is ready after Assembling and compilation, it must be loaded into memory for execution. A loader is part of an operating system that is responsible for loading programs and libraries. It is one of essential stages for starting a program. Loaders, relocatable loaders, linkage editors and Overlay loaders are provided by system.

Communications –

Virtual connections among processes, users and computer systems are provided by programs. User can send messages to other user on their screen, User can send e-mail, browsing on web pages, remote login, transformation of files from one user to another.

In addition to systems programs, most operating systems are supplied with programs that are useful in solving common problems or performing common operations. Such application programs include web browser, word processors and text formatters, spreadsheets, database systems, compilers, Plotting and statistical analysis packages, and games.