e-PG Pathshala

Subject: Computer Science

Paper: Operating Systems

Module 3: System Calls and OS Structures

Module No: CS/OS/3

Quadrant 1 — e-text

3.1 Introduction

In the previous modules we learnt about the definition of an operating system, the evolution of operating systems, the different components of an operating system and the functions of the different components of an operating system. Now, there should be some way in which users can make use of the functionalities of the operating system without knowing the inner details of the operating system. This is made possible using system calls. This module describes what system calls are, explains about the different types of system calls and briefs about the different types of operating system structures.

3.2 System calls

System calls provide the interface between a running program and the operating system. This helps the user by not wanting them to know the intricacies of the operating system. System calls may be available as assembly language instructions or even as high-level language calls in some systems. They resemble pre-defined function calls. In UNIX, system calls can be invoked directly from a C or C++ program. The system calls for Windows platforms are part of the Win32 API.

3.2.1 Example of System Calls

The use of system calls can be understood using the example shown in Figure 3.1. The example shows the sequence of system calls that will be invoked when the contents of one file is copied to another. Prompting to the screen and asking the user to enter the name of a file is a system call. Getting the name of a source file as input is a system call. Again, asking the user for the name of the destination file, reading as input the destination file name, opening the source file, creating the destination file, reading from the source file, writing to the destination file etc. are all system calls. So, any program written by any user will have a number of system calls in it. However, most users never see this level of detail.

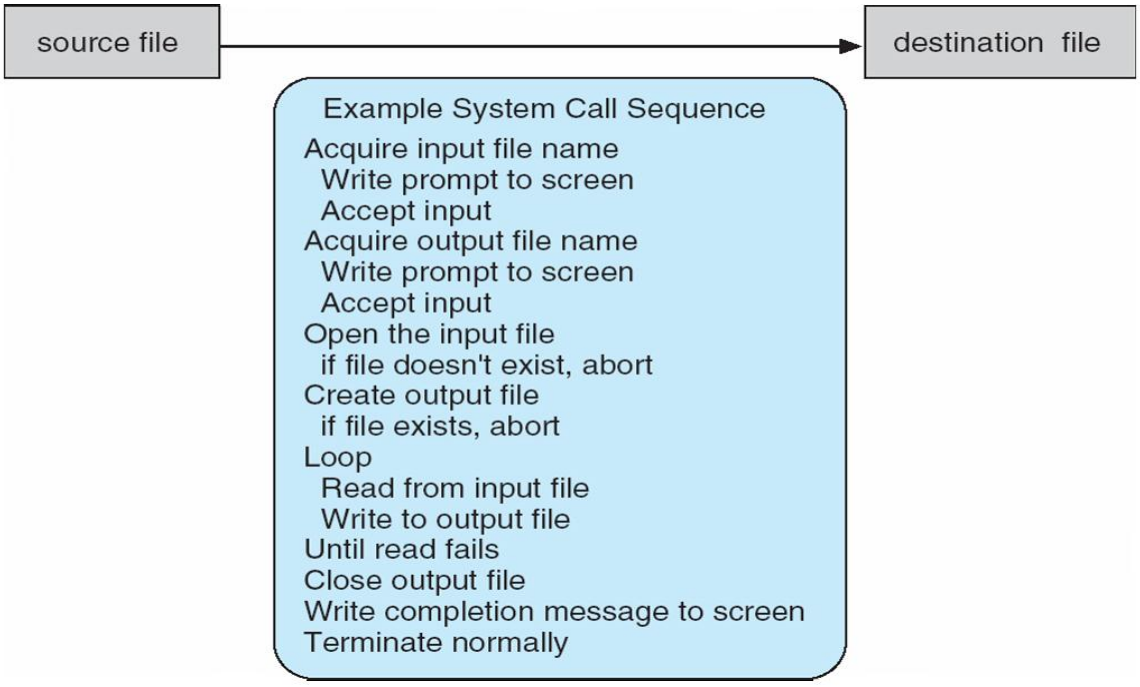

Fig 3.1 System call sequence to copy the contents of one file to another

3.3 System Call Implementation

Typically, a system call number is associated with each system call. A system-call interface maintains a table indexed according to these numbers. This system call table has the address of the routine where the system call is implemented in the operating system kernel. The system call interface invokes the intended system call in the OS kernel and returns the status of the system call and any return values. The caller/user need not know anything about how the system call is implemented. The user just needs to obey the Application Program Interface (API) and understand what the OS will do as a result of the call. Most details of the system call are hidden from the programmer by the API. The system calls are managed by a run-time support library (set of functions built into libraries included with compiler)

3.3.1 API – System Call – OS Relationship

Figure 3.2 shows a user application using the open () call to open a file. The open () call invokes the system call interface. There is a system call number corresponding to the open () call. The system call interface uses this number as an index into the system call table to find the address of the open () routine in the kernel. Using this address, the open () routine in the kernel is now invoked. The open routine in the kernel does whatever is necessary for opening a file and returns a file handle. The file handle (return value) is returned to the user.

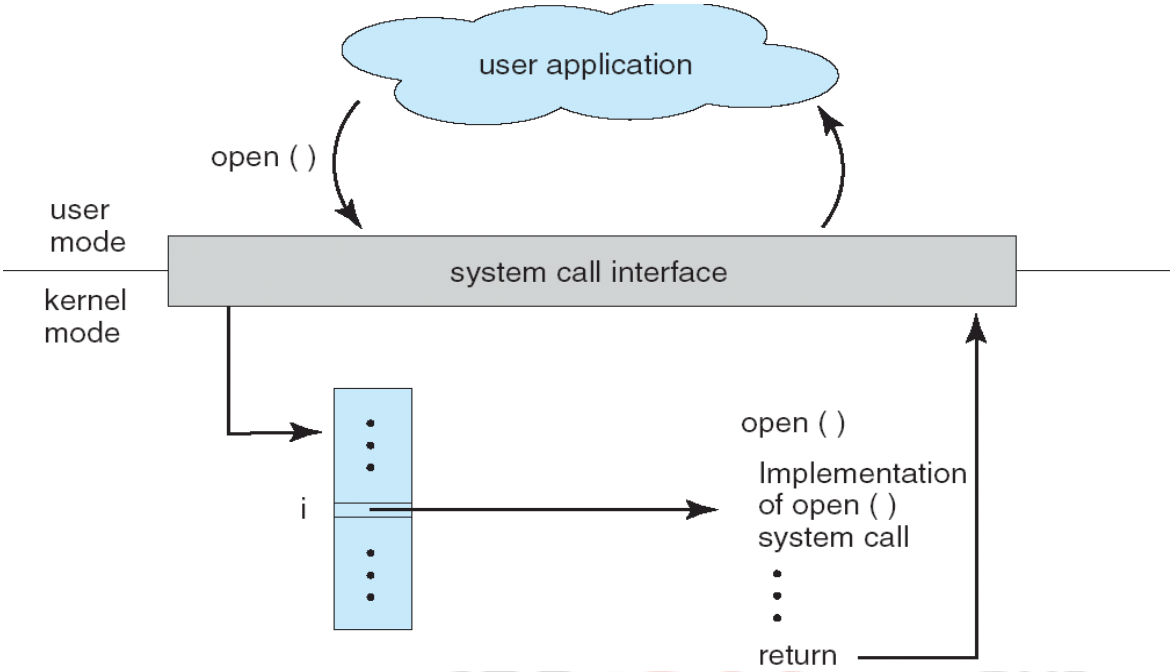

Fig 3.2 Relationship between API, system call and OS

Standard C Library Example

Figure 3.3 shows an example of a C program in which the printf statement is invoked, which is a C library call. This library function in turn calls the *write* system call. The write system call is executed and the result is returned to the user program.

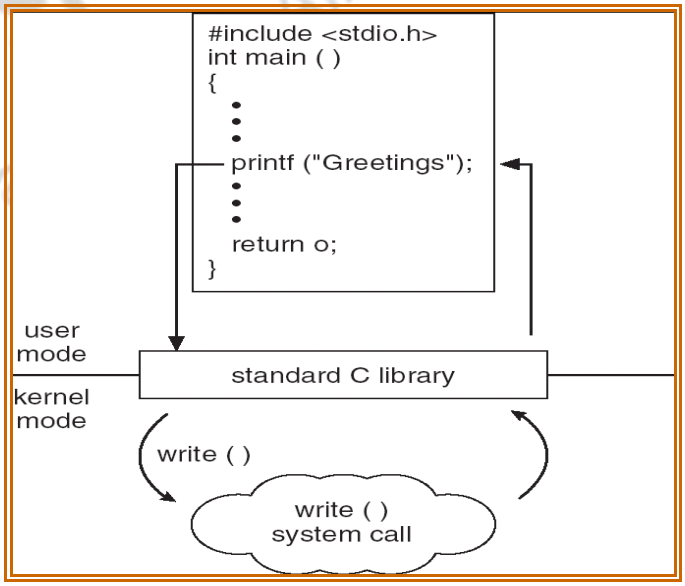

Fig. 3.3 Example of system calls being used in C library

3.4 System Call Parameter Passing

Each system call is like a function call and may or may not need parameters to be passed to them. In the example seen in the previous section, a parameter “Greetings” is passed to the printf statement. This parameter should be passed to the system call write () which is a kernel routine.

There are three general methods used to pass parameters to the OS. The simplest is to pass the parameters in registers. This is fine when there is less number of parameters. In some cases, there may be more parameters than registers. Then the parameters can either be stored in a block or table, in memory, and the address of the block can be passed as a parameter in a register. This approach is taken by Linux and Solaris. Another way of passing parameters is to place or push the parameters onto the stack by the program and the operating system can pop off the parameters from the stack. Both these block and stack methods do not limit the number or length of parameters being passed.

Figure 3.4 shows the passing of parameters of a system call using a table. X is the address in memory where the table with the parameters is stored. The address X is stored in a register. The operating system gets the address X from the register, accesses the memory location X to get the table with the parameters. The system call number of the system call is 13.

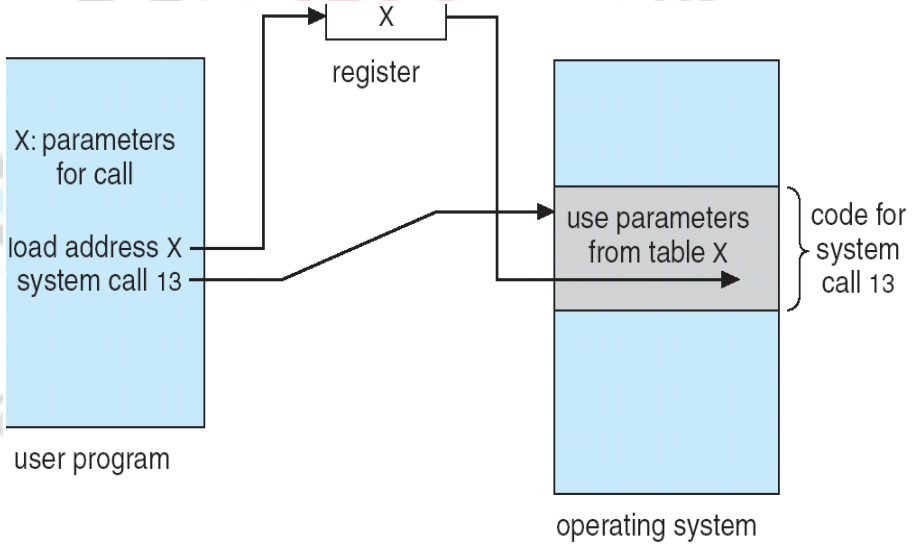

Fig. 3.4 Parameters to system call passed using a table

3.5 Types of System Calls

There are a number of system calls available for process control, file management, device management, information maintenance, communications etc.

3.5.1 Process Control

Any process in a system is created by another process except for the first process in the system. The process which creates the new process is called the parent process and the newly created process is called the child process. Each newly created process is identified by a unique process id in the system. There is a system call for creating a process. Similarly when a process exits naturally or is forced to exit, a system call for termination is called. There are system calls for a program to end normally (end) or abnormally (abort). A process while executing may want to create another new process (create), load a program to the newly created process (load) and execute (execute) that process. There are a number of attributes for a process like the id of the process, state of the process etc. There are system calls available to get the values of these attributes and to set the values. There are system calls available to allocate memory for a process in the main memory as well as swap space. Memory space is freed when the process exits. Processes may have to wait for some amount of time or wait for some event to happen before continuing execution. All these functions have corresponding system calls.

- Create, terminate
- Load, execute
- End, abort
- Get process attributes, set process attributes
- Allocate and free memory
- Wait for time, event

3.5.2 File management

Creation and deletion of files are possible in any system. The system call used to create files needs the name of the file and other attributes. The system call for deletion needs the name of the file as a parameter. Once a file is created, the user may have to open the file, read from the file, write into the file, reposition the file pointer and close the file. All these have corresponding system calls. A file also can have a number of attributes such as the name of the file, size of the file, owner of the file etc. There are system calls to get and set the attributes of a file respectively.

- Create file, delete file
- Open, close
- Read, write, reposition
- Get file attributes, set file attributes

3.5.3 Device management

There are many devices connected to a system like the printer, mouse, keyboard, hard disk, network interface etc. System calls are needed for requesting a device, reading from a device, writing to the device, reposition the location from where the current read or write can take place and finally releasing the device. Similar to

getting and setting file attributes, there needs to be system calls for getting and setting device attributes. System calls are also needed for mounting and unmounting external devices to the system.

- Request device, release device
- Read, write, reposition
- Get device attributes, set device attributes
- Logically attach or detach devices

3.4.4 Information maintenance

There are system calls to get or set the time or date of a system, get or set system data, get or set process, file or device attributes.

- 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

3.4.5 Communications

There are system calls for creating a communication connection with a remote machine and to delete or terminate the connection. There are system calls for sending and receiving messages over a communication connection.

- Create, delete communication connection
- Send, receive messages

3.4.6 Protection

There are system calls to change the owner of a file, change the group that can access a file, change the access permissions of a file (read permissions, write permissions etc.).

3.4.7 Examples of Windows and UNIX System Calls

Table 3.5 shows some of the system calls used in Windows and UNIX for different purposes. For example, for creating a new process, CreateProcess() is the system call used in Windows and fork() is the system call used in UNIX.

Table 3.5 Windows and UNIX system calls

	Windows	**UNIX**
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()

	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	creat()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessId()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapVIewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

3.5 System Structure

A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily. A common approach is to partition the task into small components, rather than have one monolithic system. Each of these modules should be a well-defined portion of the system, with carefully defined inputs, outputs and function. In this section, the way in which the components of an operating system are interconnected and melded into a kernel is discussed.

3.5.1 MS-DOS System Structure

MS-DOS was the operating system written to provide the most functionality in the least space. The operating system was not divided into modules. The kernel provides it support functions called as system functions to application programs in a hardware-independent manner and, in turn, is isolated from the hardware characteristics by relying on the driver routines in the MS-DOS BIOS to perform physical input and output operations. Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated. It is possible to access the device drivers directly even from the application program. This allows malicious users to change the device drivers and cause havoc. Figure 3.6 shows the MS-DOS layer structure.

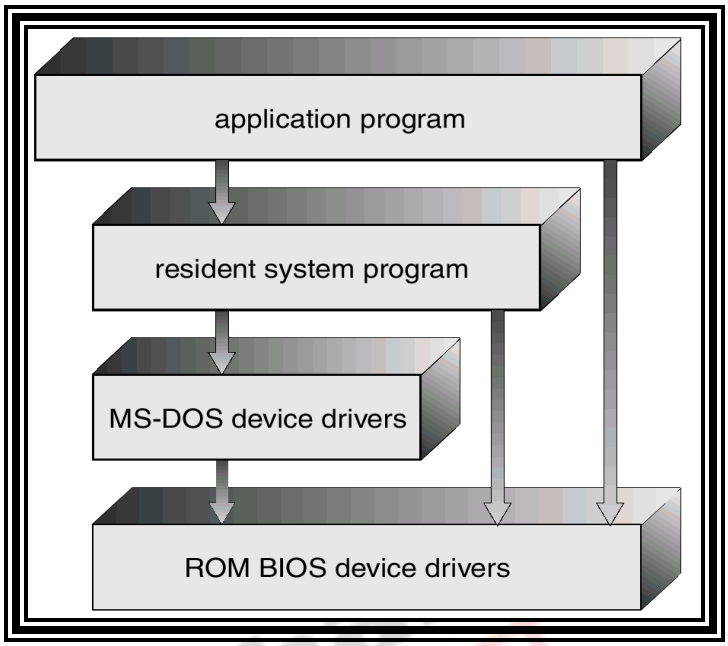

Fig 3.6 MS-DOS Layer Structure

3.5.2 UNIX System Structure

UNIX is an operating system that was initially limited by hardware functionality. It 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 were added and expanded over the years as UNIX evolved. The structure of the UNIX operating system is shown in Figure 3.7. Everything below the system-call interface and above the hardware interface is the kernel. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls. Thus, a lot of functionality has been combined into one level. This makes UNIX difficult to enhance, as changes in one section could adversely affect other areas.

System calls define the API to UNIX; the set of system programs commonly available defines the user interface. New versions of UNIX are designed to use more advanced hardware. Given proper hardware support, operating systems may be broken into pieces that are smaller and more appropriate than are those allowed by the original MS-DOS or UNIX systems. The operating system can then retain much greater control over the computer and over the applications that make use of that computer. Implementers have more freedom to make changes to the inner workings of the system and in the creation of modular operating systems.

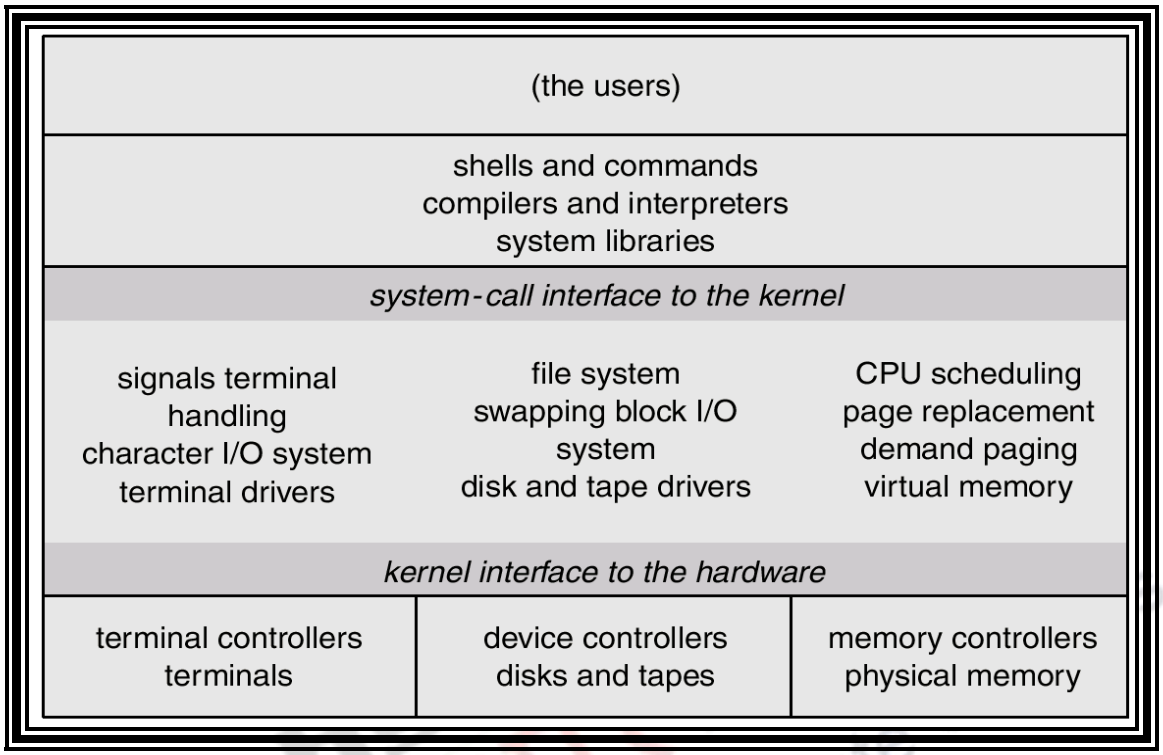

Fig 3.7 UNIX system structure

3.5.3 Layered Approach

One way of modularization is the layered approach in which the operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.

A typical operating-system layer, say layer M, is depicted in Figure 3.8. It consists of data structures and a set of routines that can be invoked by higher-level layers. Layer M, in turn, can invoke operations on lower-level layers. The advantage of having a layered approach is modularity. Layers are selected such that each layer uses the functions (operations) and services of only lower-level layers. This makes the system easy to debug. The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed to be correct) to implement its functions. Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system are simplified when the system is broken down into layers.

Thus, each layer is implemented only by the operations provided by the lower layers. So a layer need not know how these operations are implemented; it needs to know only what these operations do. Hence, each layer hides the existence of certain

data structures, operations and higher-level layers. The difficulty in a layered approach is the need of the careful definition of the layers. This is because a layer can use only those layers below it. For example, the device driver for the disk space used by virtual-memory algorithms must be at a level lower than that of the memory-management routines, because memory management requires the ability to use the disk space.

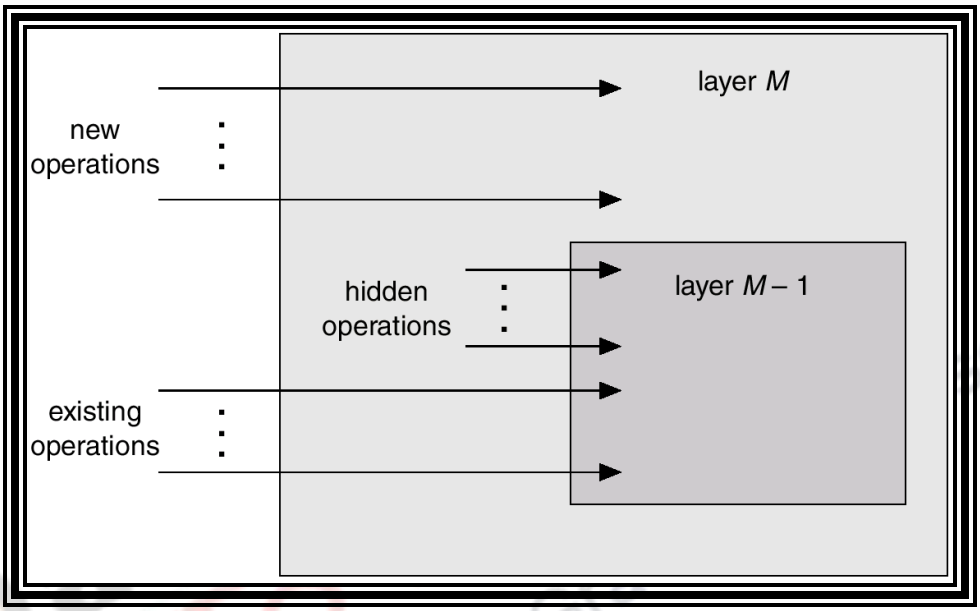

Fig. 3.8 An Operating System Layer

The layered approach was also found to be less efficient than the other types of structures. For instance, when a user program executes an I/O operation, it executes a system call that is trapped to the I/O layer, which calls the memory-management layer, which in turn calls the CPU-scheduling layer, which is then passed to the hardware. At each layer, the parameters may be modified; data may need to be passed, and so on. Each layer adds overhead to the system call; the net result is a system call that takes longer than does one on a non-layered system.

Hence systems were designed with less number of layers. Thus the concept of modularity was brought in with less problems of layer interaction. OS/2 Layer Structure is an example of a layered operating system. Figure 3.9 shows the layer structure of OS/2. OS/2 is a descendant of MSDOS that adds multitasking and dual-mode operation, as well as other new features. Because of this added complexity and the more powerful hardware for which OS/2 was designed, the system was implemented in a more layered fashion.

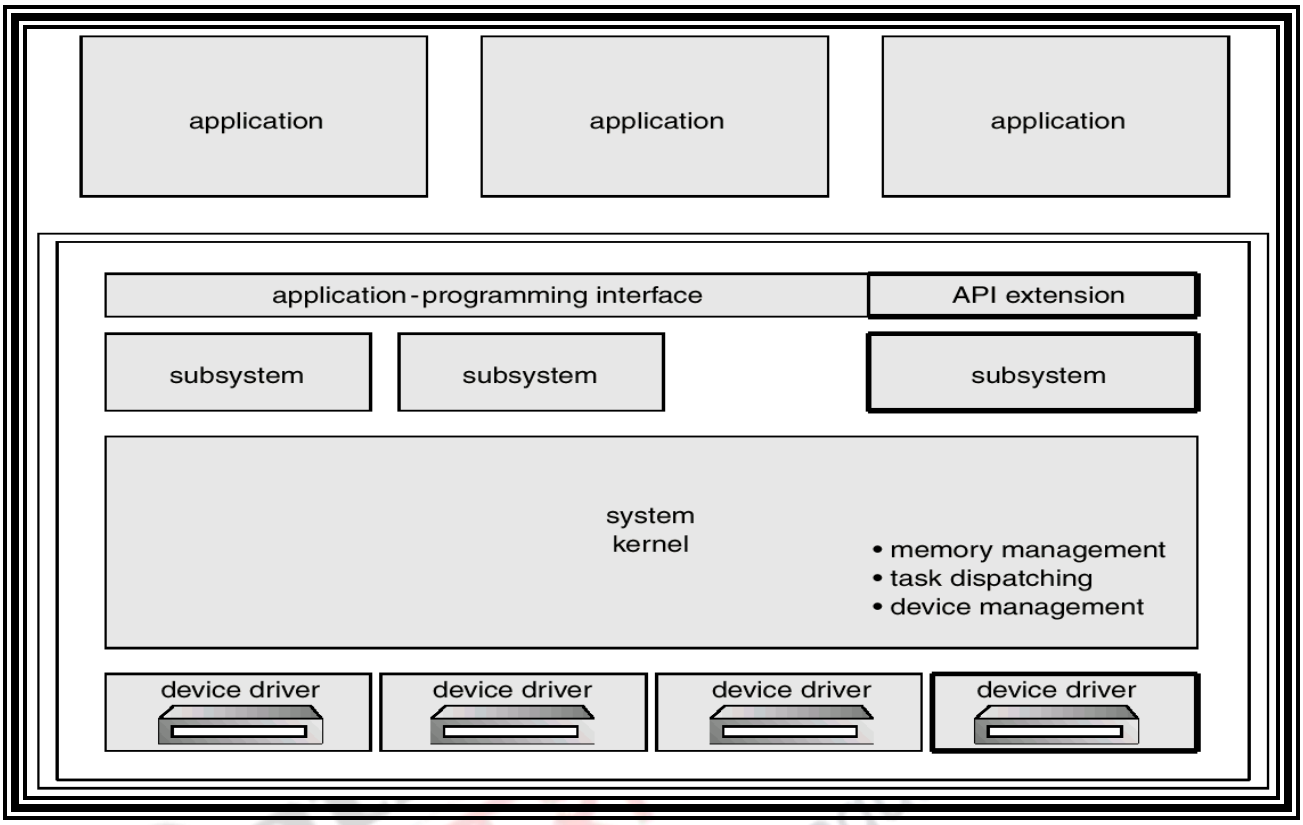

Fig. 3.9 OS/2 Layer Structure

3.6 Summary

Thus, in this module we learnt the need for system calls, the way in which system calls are called and the details of different types of system calls. We also learnt the different operating system structures.

References

1. Abraham Silberschatz, Peter B. Galvin, Greg Gagne, “Operating System Concepts”, Ninth Edition, John Wiley & Sons Inc., 2012.
2. Thomas Anderson, Michael Dahlin, “Operating Systems: Principles and Practice”, Recursive Books, 2012.