Unit 2 Operating System Architecture

Structure:

2.1 Introduction

In the previous unit, you studied about an overview of Operating System. In this unit, let us explore various operating system approaches.

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 component 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 functions. In this unit, we discuss how various components of an operating system are interconnected and melded into a kernel.

Objectives:

After studying this unit, you should be able to:

- explain what is kernel and monolithic kernel architecture
- describe the layered architecture of the operating system
- discuss the microkernel architecture of the operating system
- list the operating system components
- explain operating system services

2.2 Operating System as an Extended Machine

We can think of an operating system as an *Extended Machine* standing between our programs and the bare hardware.

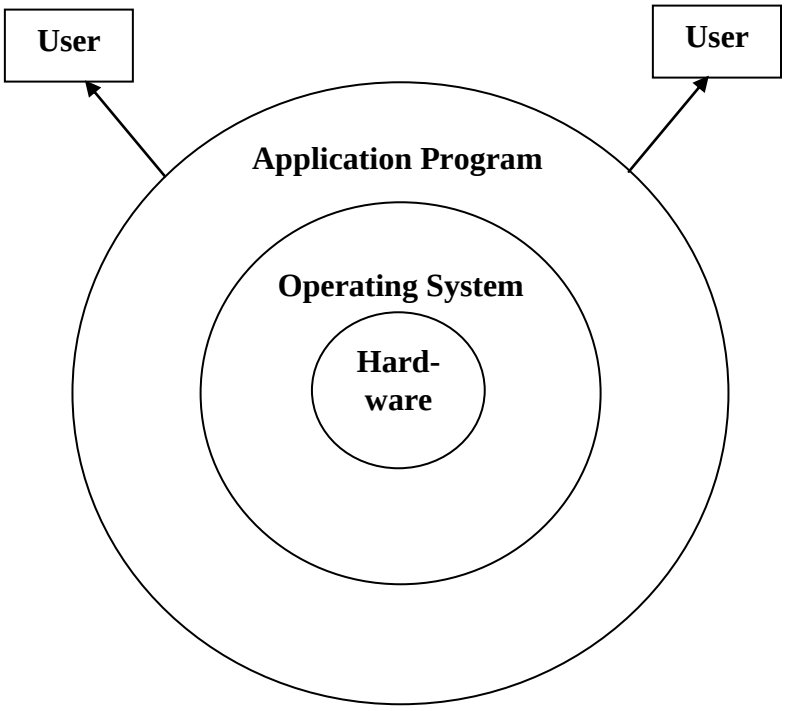

Fig. 2.1: OS as an Extended Machine

As shown in above figure 2.1, the operating system interacts with the hardware hiding it from the application program, and user. Thus it acts as interface between user programs and hardware.

You know that many commercial systems do not have well-defined structures. Frequently, such operating systems started as small, simple, and limited systems and then grew beyond their original scope. MS-DOS is an example of such a system. It was originally designed and implemented by a few people who had no idea that it would become so popular. It was written to provide the most functionality in the least space, so it was not divided into modules carefully. In MS-DOS, the interfaces and levels of functionality are not well separated. For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such

freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail. Of course, MS-DOS was also limited by the hardware of its era. Because the Intel 8088 for which it was written provides no dual mode and no hardware protection, the designers of MS-DOS had no choice but to leave the base hardware accessible.

Another example of limited structuring is the original UNIX operating system. UNIX is another system that initially was 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 have been added and expanded over the years as UNIX has evolved. We can view the traditional UNIX operating system as being layered. 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 level. This monolithic structure was difficult to implement and maintain.

Self Assessment Questions

1. Operating System acts as an interface between user programs and hardware. (True / False)
2. _____ is an example for an Operating System that doesn't have well-defined structure.
3. _____ is a part of UNIX OS.
   a) Kernel
   b) Module
   c) Core
   d) Software

2.3 Layered Approach

With proper hardware support, operating systems can be broken into pieces that are smaller and more appropriate than 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 in changing the inner workings of the system and in creating modular operating systems. Under the top-down approach, the overall functionality and features are determined and the separated into components. Information hiding is also important, because it leaves programmers free to implement the low-level routines as they see fit, provided that the external interface of the routine stays unchanged and that the routine itself performs the advertised task.

A system can be made modular in many ways. One method is the **layered approach,** in which the operating system is broken up into a number of layers (levels). The bottom layer (layer 0) id the hardware; the highest (layer N) is the user interface. Figure 2.2 represents the layer architecture of the operating system.

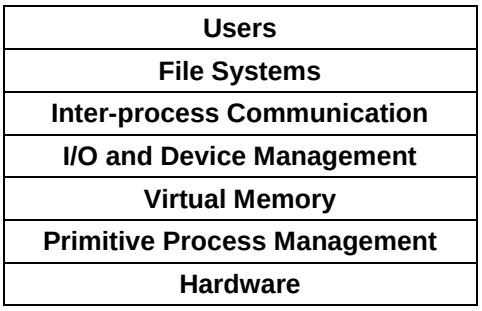

Fig. 2.2: Layered Architecture

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-say, layer M-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 main advantage of the layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions (operations) and services of only lower-level layers. This approach simplifies debugging and system verification. 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 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 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 is simplified.

Each layer is implemented with only those operations provided by lower-level layers. A layer does not need to 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 hardware from higher-level layers. The major difficulty with the layered approach involves appropriately defining the various layers. Because layer can use only lower-level layers, careful planning is necessary. For example, the device driver for the backing store (disk space used by virtual-memory algorithms) must be at a lower level than the memory-management routines, because memory management requires the ability to use the backing store.

Other requirement may not be so obvious. The backing-store driver would normally be above the CPU scheduler, because the driver may need to wait for I/O and the CPU can be rescheduled during this time. However, on a larger system, the CPU scheduler may have more information about all the active processes than can fit in memory. Therefore, this information may need to be swapped in and out of memory, requiring the backing-store driver routine to be below the CPU scheduler.

A final problem with layered implementations is that they tend to be less efficient than other types. 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. These limitations have caused a small backlash against layering in recent years. Fewer layers with more functionality are being designed, providing most of the advantages of modularized code while avoiding the difficult problems of layer definition and interaction.

2.4 Micro-Kernels

We have already seen that as UNIX expanded, the kernel became large and difficult to manage. In the mid-1980s, researches at Carnegie Mellon University developed an operating system called **Mach** that modularized the kernel using the microkernel approach. This method structures the operating system by removing all nonessential components from the kernel and implementing then as system and user-level programs. The result is a smaller kernel. There is little consensus regarding which services should remain in the kernel and which should be implemented in user space. Typically, however, micro-kernels provide minimal process and memory management, in addition to a communication facility. Figure 2.3 shows the microkernel architecture of the operating system.

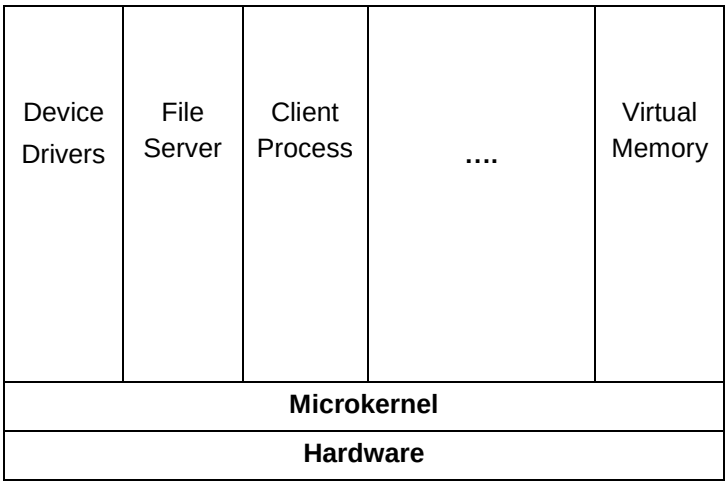

Fig. 2.3: Microkernel Architecture

The main function of the microkernel is to provide a communication facility between the client program and the various services that are also running in user space. Communication is provided by *message passing*. For example, if the client program and service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel.

On benefit of the microkernel approach is ease of extending the operating system. All new services are added to user space and consequently do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a

smaller kernel. The resulting operating system is easier to port from one hardware design to another. The microkernel also provided more security and reliability, since most services are running as user – rather than kernel – processes, if a service fails the rest of the operating system remains untouched.

Several contemporary operating systems have used the microkernel approach. Tru64 UNIX (formerly Digital UNIX provides a UNIX interface to the user, but it is implemented with a March kernel. The March kernel maps UNIX system calls into messages to the appropriate user-level services.

The following figure shows the UNIX operating system architecture. At the center is hardware, covered by kernel. Above that are the UNIX utilities, and command interface, such as shell (sh), etc. Figure 2.4 represents the UNIX architecture.

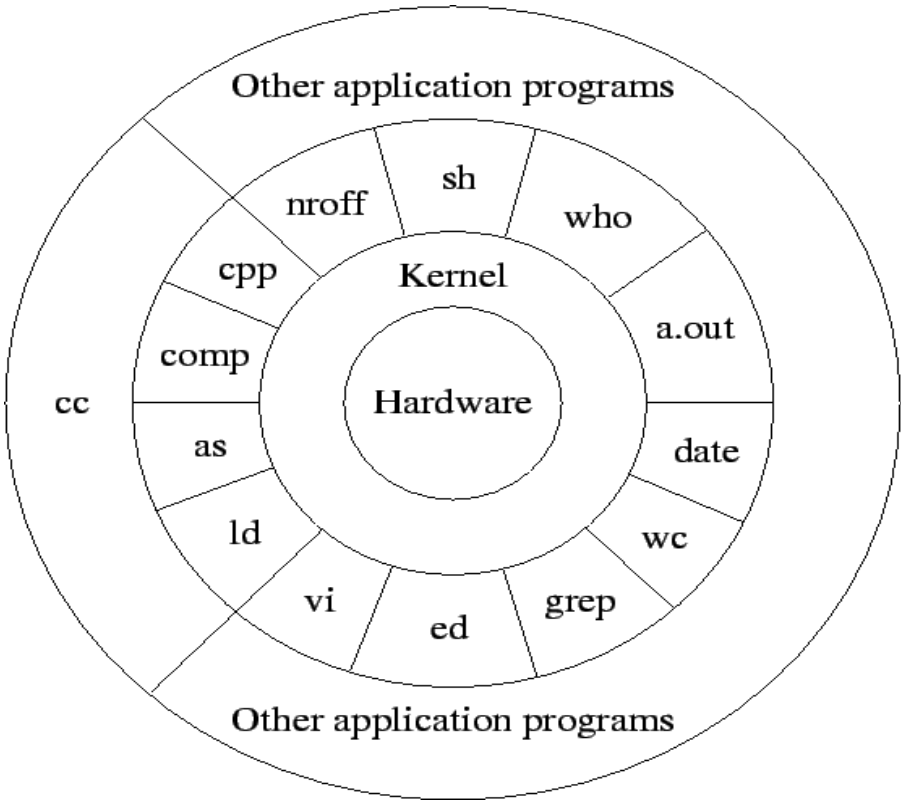

Fig. 2.4: UNIX Architecture

Self Assessment Questions

4. In the bottom-up approach, the overall functionality and features are determined and are separated into components. (True / False)
5. In the Layered Approach, the bottom layer id the _____ ; the highest is the _____.
6. In the mid-1980s, researches at Carnegie Mellon University developed an operating system called _____ that modularized the kernel using the microkernel approach.
   a) DOS
   b) Mac
   c) Unix
   d) Linux

2.5 UNIX Kernel Components

The UNIX kernel s divided in to three modes: user mode, kernel mode, and hardware. The user mode contains user programs which can access the services of the kernel components using system call interface.

The kernel mode has four major components: system calls, file subsystem, process control subsystem, and hardware control. The system calls are interface between user programs and file and process control subsystems. The file subsystem is responsible for file and I/O management through device drivers.

The process control subsystem contains scheduler, Inter-process communication and memory management. Finally the hardware control is the interface between these two subsystems and hardware. Figure 2.6 shows UNIX kernel components.

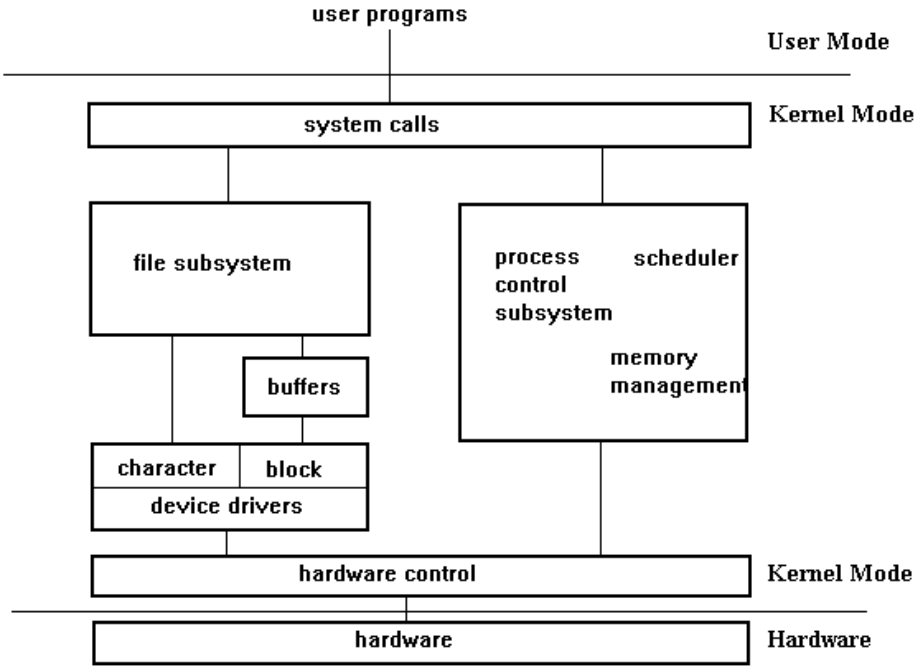

Fig. 2.5: UNIX kernel components

Another example is QNX. QNX is a real-time operating system that is also based on the microkernel design. The QNX microkernel provides services for message passing and process scheduling. It also handled low-level network communication and hardware interrupts. All other services in QNX are provided by standard processes that run outside the kernel in user mode.

Unfortunately, microkernels can suffer from performance decreases due to increased system function overhead. Consider the history of Windows NT. The first release had a layered microkernels organization. However, this version delivered low performance compared with that of Windows 95. Windows NT 4.0 partially redressed the performance problem by moving layers from user space to kernel space and integrating them more closely. By the time Windows XP was designed, its architecture was more monolithic than microkernel.

2.6 Modules

Perhaps the best current methodology for operating-system design involves using object-oriented programming techniques to create a modular kernel. Here, the kernel has a set of core components and dynamically links in additional services either during boot time or during run time. Such a strategy uses dynamically loadable modules and is common in modern implementations of UNIX, such as Solaris, Linux and MacOS. For example, the Solaris operating system structure is organized around a core kernel with seven types of loadable kernel modules:

1. Scheduling classes
2. File systems
3. Loadable system calls
4. Executable formats
5. STREAMS formats
6. Miscellaneous
7. Device and bus drivers

Such a design allow the kernel to provide core services yet also allows certain features to be implemented dynamically. For example device and bus drivers for specific hardware can be added to the kernel, and support for different file systems can be added as loadable modules. The overall result resembles a layered system in that each kernel section has defined, protected interfaces; but it is more flexible than a layered system in that any module can call any other module. Furthermore, the approach is like the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient, because modules do not need to invoke message passing in order to communicate.

2.7 Introduction to Virtual Machines

The layered approach of operating systems is taken to its logical conclusion in the concept of virtual machine. The fundamental idea behind a virtual machine is to abstract the hardware of a single computer (the CPU, Memory, Disk drives, Network Interface Cards, and so forth) into several different execution environments and thereby creating the illusion that each separate execution environment is running its own private computer. By using CPU Scheduling and Virtual Memory techniques, an operating system

can create the illusion that a process has its own processor with its own (virtual) memory. Normally a process has additional features, such as system calls and a file system, which are not provided by the hardware. The Virtual machine approach does not provide any such additional functionality but rather an interface that is identical to the underlying bare hardware. Each process is provided with a (virtual) copy of the underlying computer.

Hardware Virtual Machine

The original meaning of **virtual machine**, sometimes called a **hardware virtual machine**, is that of a number of discrete identical execution environments on a single computer, each of which runs an operating system (OS). This can allow applications written for one OS to be executed on a machine which runs a different OS, or provide execution "sandboxes" which provide a greater level of isolation between processes than is achieved when running multiple processes on the same instance of an OS. One use is to provide multiple users the illusion of having an entire computer, one that is their "private" machine, isolated from other users, all on a single physical machine. Another advantage is that booting and restarting a virtual machine can be much faster than with a physical machine, since it may be possible to skip tasks such as hardware initialization.

Such software is now often referred to with the terms virtualization and virtual servers. The host software which provides this capability is often referred to as a **virtual machine monitor** or **hypervisor**.

Software virtualization can be done in three major ways:

- Emulation, full system simulation, or "full virtualization with dynamic recompilation"– the virtual machine simulates the complete hardware, allowing an unmodified OS for a completely different CPU to be run.
- Para-virtualization – the virtual machine does not simulate hardware but instead offers a special API that requires OS modifications. An example of this is XenSource's XenEnterprise (www.xensource.com)
- Native virtualization and "full virtualization"– the virtual machine only partially simulates enough hardware to allow an unmodified OS to be run in isolation, but the guest OS must be designed for the same type of CPU. The term *native virtualization* is also sometimes used to designate that hardware assistance through Virtualization Technology is used.

Application Virtual Machine

Another meaning of **virtual machine** is a piece of computer software that isolates the application being used by the user from the computer. Because versions of the virtual machine are written for various computer platforms, any application written for the virtual machine can be operated on any of the platforms, instead of having to produce separate versions of the application for each computer and operating system. The application is run on the computer using an interpreter or Just In Time compilation. One of the best known examples of an application virtual machine is Sun Microsystem's Java Virtual Machine.

Self-Assessment Questions

7. The UNIX kernel is divided in to three modes: user mode, kernel mode, and hardware. (True / False)
8. In the UNIX kernel, _____ subsystem contains scheduler, Inter-process communication and memory management.
9. By using _____ techniques, an operating system can create the illusion that a process has its own processor with its own memory.
   a) CPU Scheduling
   b) Virtual Memory
   c) Both A and B
   d) None of the above

2.8 Virtual Environment & Machine Aggregation

A **virtual environment** (otherwise referred to as Virtual private server) is another kind of a virtual machine. In fact, it is a virtualized environment for running user-level programs (i.e. not the operating system kernel and drivers, but applications). Virtual environments are created using the software implementing operating system-level virtualization approach, such as Virtuoso, FreeBSD Jails, Linux-VServer, Solaris Containers, cheroot jail and OpenVZ.

A less common use of the term is to refer to a computer cluster consisting of many computers that have been aggregated together as a larger and more powerful "virtual" machine. In this case, the software allows a single environment to be created spanning multiple computers, so that the end user appears to be using only one computer rather than several.

PVM (Parallel Virtual Machine) and MPI (Message Passing Interface) are two common software packages that permit a heterogeneous collection of networked UNIX and/or Windows computers to be used as a single, large, parallel computer. Thus large computational problems can be solved more cost effectively by using the aggregate power and memory of many computers than with a traditional supercomputer. The Plan9 Operating System from Bell Labs uses this approach.

Boston Circuits had released the gCore (grid-on-chip) Central Processing Unit (CPU) with 16 ARC 750D cores and a Time-machine hardware module to provide a virtual machine that uses this approach.

2.9 Implementation Techniques

Let’s see various implementation techniques.

Emulation of the underlying raw hardware (native execution)

This approach is described as full virtualization of the hardware, and can be implemented using a Type 1 or Type 2 hypervisor. (A Type 1 hypervisor runs directly on the hardware; a Type 2 hypervisor runs on another operating system, such as Linux.) Each virtual machine can run any operating system supported by the underlying hardware. Users can thus run two or more different "guest" operating systems simultaneously, in separate "private" virtual computers.

The pioneer system using this concept was IBM's CP-40, the first (1967) version of IBM's CP/CMS (1967-1972) and the precursor to IBM's VM family (1972-present). With the VM architecture, most users run a relatively simple interactive computing single-user operating system, CMS, as a "guest" on top of the VM control program (VM-CP). This approach kept the CMS design simple, as if it were running alone; the control program quietly provides multitasking and resource management services "behind the scenes". In addition to CMS, VM users can run any of the other IBM operating systems, such as MVS or z/OS. z/VM is the current version of VM, and is used to support hundreds or thousands of virtual machines on a given mainframe. Some installations use Linux for zSeries to run Web servers, where Linux runs as the operating system within many virtual machines.

Full virtualization is particularly helpful in operating system development, when experimental new code can be run at the same time as older, more stable, versions, each in separate virtual machines. (The process can even be recursive: IBM debugged new versions of its virtual machine operating system, VM, in a virtual machine running under an older version of VM, and even used this technique to simulate new hardware.)

The x86 processor architecture as used in modern PCs does not actually meet the Popek and Goldberg virtualization requirements. Notably, there is no execution mode where all sensitive machine instructions always trap, which would allow per-instruction virtualization.

Despite these limitations, several software packages have managed to provide virtualization on the x86 architecture, even though dynamic recompilation of privileged code, as first implemented by VMware, incurs some performance overhead as compared to a VM running on a natively virtualizable architecture such as the IBM System/370 or Motorola MC68020. By now, several other software packages such as Virtual PC, VirtualBox, Parallels Workstation and Virtual Iron manage to implement virtualization on x86 hardware.

On the other hand, plex86 can run only Linux under Linux using a specific patched kernel. It does not emulate a processor, but uses bochs for emulation of motherboard devices. Intel and AMD have introduced features to their x86 processors to enable virtualization in hardware.

Emulation of a non-native system

Virtual machines can also perform the role of an emulator, allowing software applications and operating systems written for computer processor architecture to be run.

Some virtual machines emulate hardware that only exists as a detailed specification. For example:

- One of the first was the p-code machine specification, which allowed programmers to write Pascal programs that would run on any computer running virtual machine software that correctly implemented the specification.
- The specification of the Java virtual machine.

- The Common Language Infrastructure virtual machine at the heart of the Microsoft .NET initiative.
- Open Firmware allows plug-in hardware to include boot-time diagnostics, configuration code, and device drivers that will run on any kind of CPU.

This technique allows diverse computers to run any software written to that specification; only the virtual machine software itself must be written separately for each type of computer on which it runs.

Operating system-level virtualization

Operating System-level Virtualization is a server virtualization technology which virtualizes servers on an operating system (kernel) layer. It can be thought of as partitioning: a single physical server is sliced into multiple small partitions (otherwise called virtual environments (VE), virtual private servers (VPS), guests, zones etc.); each such partition looks and feels like a real server, from the point of view of its users.

The operating system level architecture has low overhead that helps to maximize efficient use of server resources. The virtualization introduces only a negligible overhead and allows running hundreds of virtual private servers on a single physical server. In contrast, approaches such as virtualization (like VMware) and para-virtualization (like Xen or UML) cannot achieve such level of density, due to overhead of running multiple kernels. From the other side, operating system-level virtualization does not allow running different operating systems (i.e. different kernels), although different libraries, distributions etc. are possible

Self Assessment Questions

10. Virtual environment is also referred to as virtual private server. (True / False)
11. _____ and _____ are two common software packages that permit a heterogeneous collection of networked UNIX and/or Windows computers to be used as a single, large, parallel computer.
12. _____ allows plug-in hardware to include boot-time diagnostics, configuration code, and device drivers that will run on any kind of CPU.
    a) p-code
    b) Java Virtual Machine (JVM)
    c) Common Language Runtime (CLR)
    d) Open Firmware

2.10 Summary

Let's summarize the important points:

- The virtual machine concept has several advantages. In this environment, there is complete protection of the various system resources. Each virtual machine is completely isolated from all other virtual machines, so there are no protection problems. At the same time, however, there is no direct sharing of resources. Two approaches to provide sharing have been implemented. A virtual machine is a perfect vehicle for operating systems research and development.
- Operating system as extended machine acts as interface between hardware and user application programs. The kernel is the essential center of a computer operating system, i.e. the core that provides basic services for all other parts of the operating system. It includes interrupts handler, scheduler, operating system address space manager, etc.
- In the layered type architecture of operating systems, the components of kernel are built as layers on one another, and each layer can interact with its neighbor through interface. Whereas in micro-kernel architecture, most of these components are not part of kernel but acts as another layer to the kernel, and the kernel comprises of essential and basic components.

2.11 Terminal Questions

1. Explain operating system as extended machine.
2. What is a kernel? What are the main components of a kernel?
3. What is a micro-kernel? Describe its architecture.
4. Describe UNIX kernel components in brief.
5. Explain Operating System Virtualization in detail.

2.12 Answers

Self Assessment Questions

1. True
2. MS-DOS
3. a
4. False
5. Hardware, User Interface
6. b

7. True
8. Process Control
9. c
10. True
11. Parallel Virtual Machine (PVM), Message Passing Interface (MPI)
12. d

Terminal Questions

1. The operating system interacts with the hardware hiding it from the application program, and user. Thus it acts as interface between user programs and hardware. (Refer Section 2.2)
2. In the UNIX OS, 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. (Refer Section 2.3)
3. The micro-kernel approach structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs. The result is a smaller kernel. Typically, however, micro-kernels provide minimal process and memory management, in addition to a communication facility. (Refer Section 2.4)
4. The UNIX kernel mode has four major components: system calls, file subsystem, process control subsystem, and hardware control. (Refer Section 2.5)
5. A virtual environment (otherwise referred to as virtual private server) is another kind of virtual machine. It is a virtualized environment for running user-level programs. Virtual environments are created using the software implementing operating system-level virtualization approach. (Refer Section 2.8)