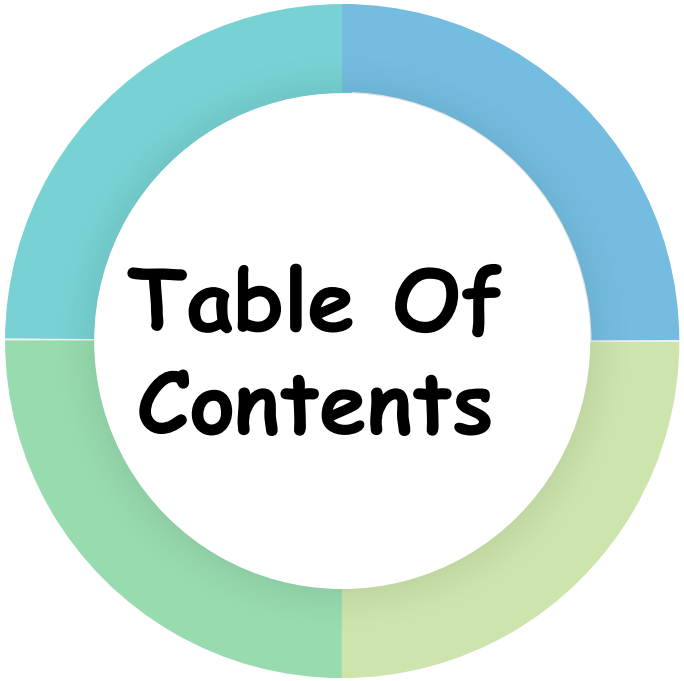
OPERATING SYSTEMS

Module-1(Part2): Operating System Services

Anuradha Pai
Assistant Prof
BTI College of Engineering



1. Objectives

- To describe the services an operating system provides to users, processes, and other systems.
- To discuss the various ways of structuring an operating system
- To explain how operating systems are installed and customized and how they boot

2. Operating System Services

- Operating systems provides an environment for execution of programs and offers services to both programs and users
- One set of operating-system services provides functions that are helpful to the user:

1. **User interface** - User Interfaces allow users to interact with and issue commands to the system. Depending on the operating system, these interfaces could be:
   a) **Command-Line (CLI) -** commands are given to the system.
   b) **Batch** - a series of commands or instructions can be written into a file, often called a script or batch file. When the file is executed, the operating system processes and runs the commands listed in it sequentially.

2. Operating System Services

- For example:
  - In Linux/Unix, these are called shell scripts (with extensions like .sh).
  - In Windows, they can be batch files (with extensions like .bat or .cmd).

  By executing the file, the OS runs all the commands without needing the user to manually input each one, making it a more efficient way to automate tasks or control the system.

c) **Graphics User Interface (GUI)** - where users interact with visual elements using a pointing device (e.g., mouse) and a keyboard for text input.

2. **Program Execution** - The OS must be able to load a program into RAM, run the program, and terminate the program, either normally or abnormally.
3. **I/O Operations** - The OS is responsible for transferring data to and from I/O devices, including keyboards, terminals, printers, and files. For specific devices, special functions are provided (device drivers) by OS.

2. Operating System Services

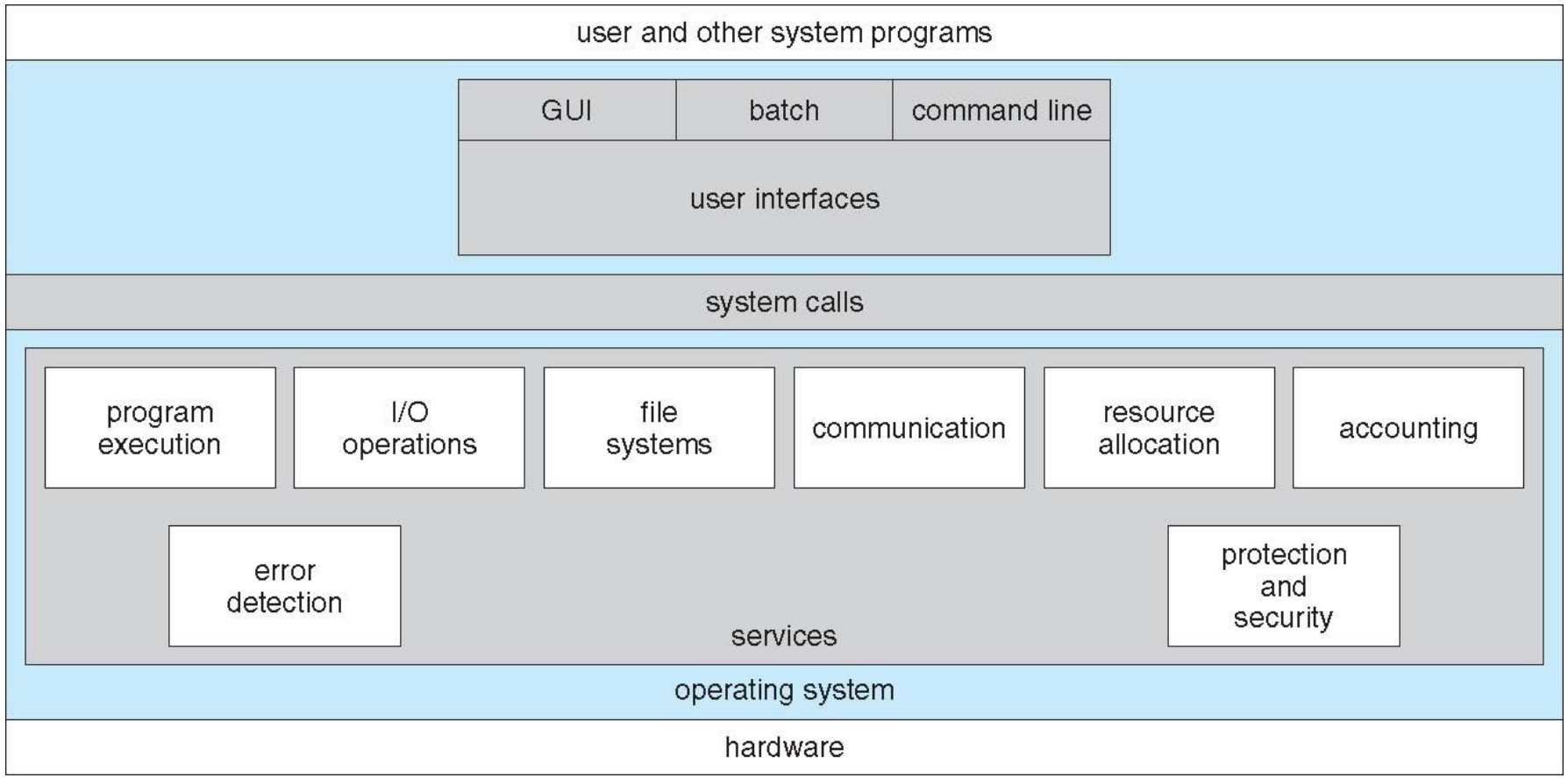
4. **File-System Manipulation** – Programs need to read and write files or directories. The services required to create or delete files, search for a file, list the contents of a file and change the file permissions are provided by OS.
5. **Communications** - Inter-process communications, IPC, either between processes running on the same processor, or between processes running on separate processors or separate machines. May be implemented by using the service of OS- like shared memory or message passing.
6. **Error Detection** - Both hardware and software errors must be detected and handled appropriately by the OS. Errors may occur in
   - The CPU and memory hardware (such as power failure and memory error),
   - In I/O devices (such as a parity error on tape, a connection failure on a network, or lack of paper in the printer)
   - In the user program (such as an arithmetic overflow, an attempt to access an illegal memory location).

2. Operating System Services

Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing

7. **Resource allocation** - Resources like CPU cycles, main memory, storage space, and I/O devices must be allocated to multiple users and multiple jobs at the same time.
8. **Accounting -** The OS keeps track of which users use how much and what kinds of computer resources
9. **Protection and security -** The owners of information(file) stored in a multiuser or networked computer system may want to control use of that information
    - **Protection** involves ensuring that all access to system resources is controlled
    - **Security** of the system from outsiders requires user authentication

2. A view of Operating System Services



3. User – Operating System Interface

There are several ways for users to interface with the operating system.

i. Command-line interface, or command interpreter, allows users to directly enter commands to be performed by the operating system.

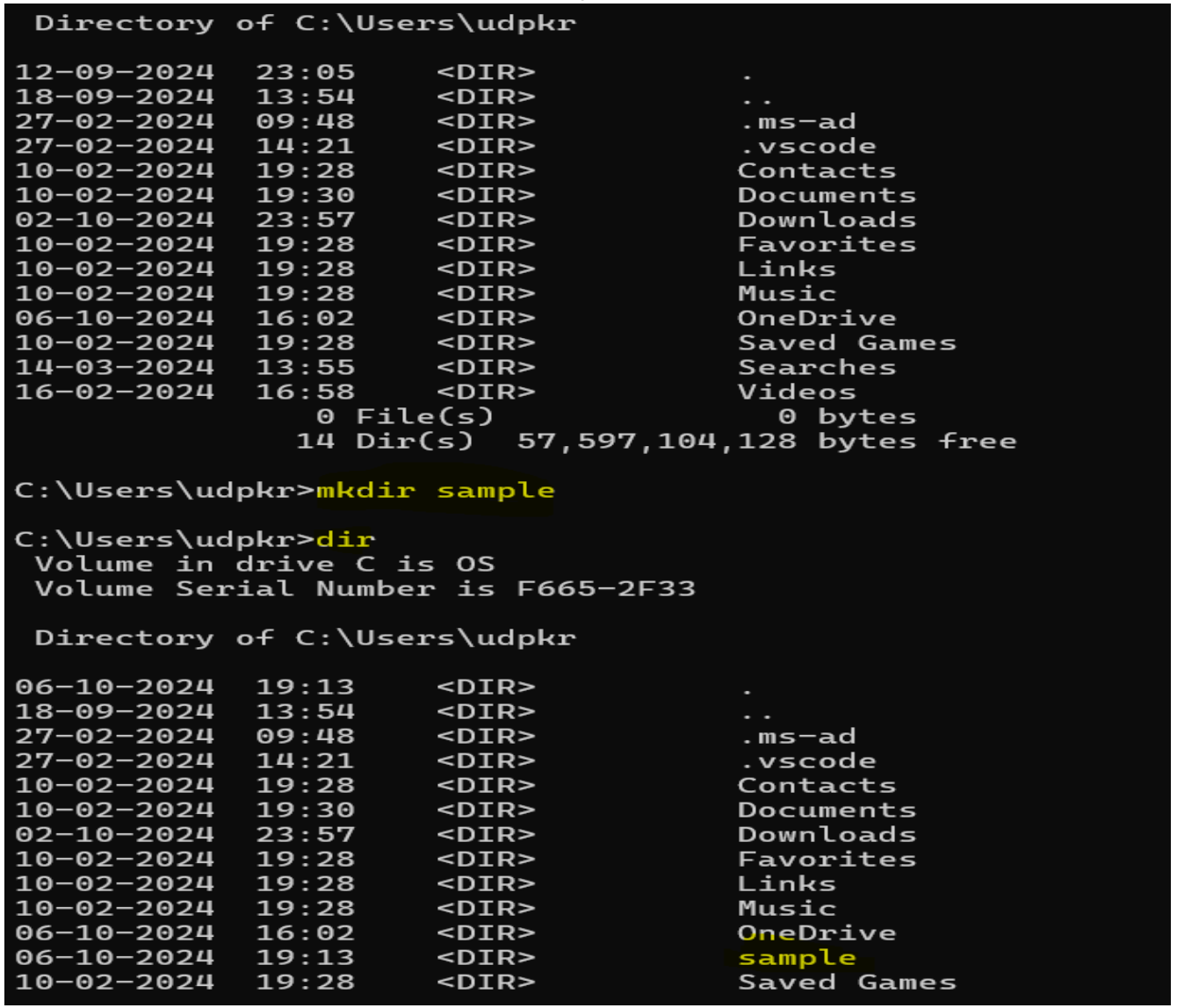
ii. Graphical user interface (GUI), allows users to interface with the operating system using pointer device and menu system.

3.1 Command Interpreter

- Command Interpreters are used to give commands to the OS. There are multiple command interpreters known as shells. In UNIX and Linux systems, there are several different shells, like the *Bourne shell, C shell, Bourne-Again shell, Korn shell,* and others.
- The main function of the command interpreter is to get and execute the user-specified command. Many of the commands manipulate files: create, delete, list, print, copy, execute, and so on.
- The commands can be implemented in two general ways-
    i. 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 particular section of its code that sets up the parameters and makes the appropriate system call.
    ii. The code to implement the command is in a function in a separate file. The interpreter searches for the file and loads it into the memory and executes it by passing the parameter.

Thus by adding new functions new commands can be added easily to the interpreter without disturbing it.

3.1 Command Interpreter



3.2 Graphical User Interface

- A second strategy for interfacing with the operating system is through a user friendly graphical user interface or GUI. Rather than having users directly enter commands via a command-line interface, a GUI allows provides a mouse-based window-and-menu system as an interface.
- A GUI provides a **desktop** metaphor where the mouse is moved 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.

3.2 Graphical User Interface

- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI “command” shell
  - Apple Mac OS X as “Aqua” GUI interface with UNIX kernel underneath and shells available
  - Solaris is CLI with optional GUI interfaces (Java Desktop, KDE)

4. System Calls

- Interface to the services provided of the Operating System
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Program Interface (API)** rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

4.1 Why use APIs rather than system calls?

Without an API (Direct System Calls):

c Copy code

```c
// Linux-specific code
int fd = open("file.txt", O_RDONLY); // Direct system call
char buffer[100];
int bytesRead = read(fd, buffer, 100); // Direct system call
close(fd); // Direct system call
```

This code will only work on Linux because it uses Linux-specific system calls (`open` , `read` , `close`).

With an API:

c Copy code

```c
#include <stdio.h>

int main() {
    FILE *file = fopen("file.txt", "r"); // API function
    char buffer[100];
    fread(buffer, sizeof(char), 100, file); // API function
    fclose(file); // API function
    return 0;
}
```

With API

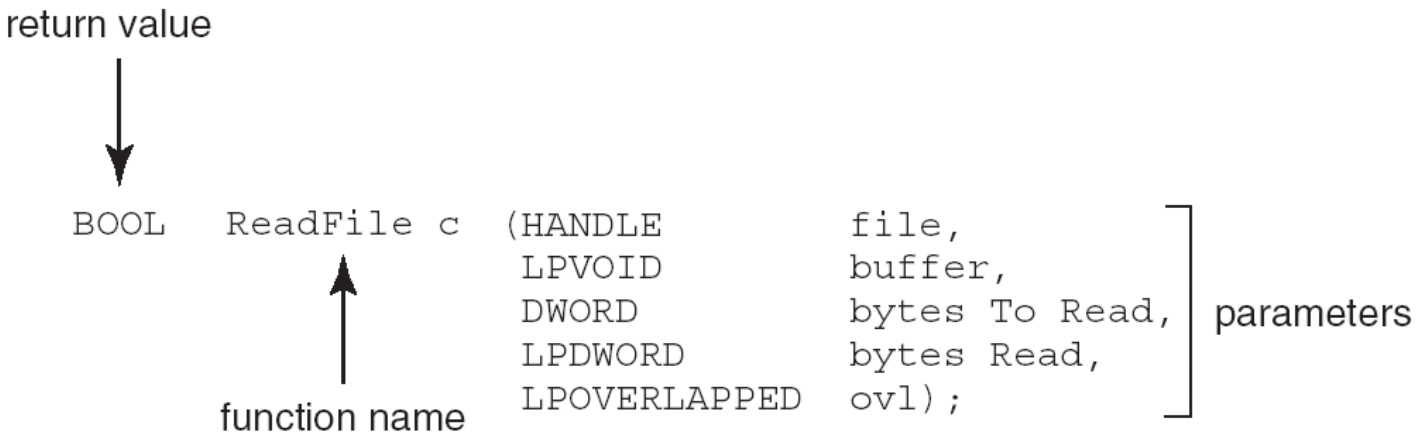
- The program uses the standard I/O library (fopen, fread, fclose), which serves as the API.
- When fopen is called, the API handles the specifics of opening a file, translating it into the appropriate system call for the operating system it is running on (e.g., open for Linux or CreateFile for Windows).

4.1 Why use APIs rather than system calls?

- **Portability**: The same code can run on different operating systems without modification because the API handles the differences. The API translates requests into the appropriate system calls via the system call interface, utilizing a system call table to access specific numbered system calls
- **Ease of Use**: Developers do not need to memorize the specific system calls for each OS, making the development process more straightforward and efficient.

4.2 Example of Standard API

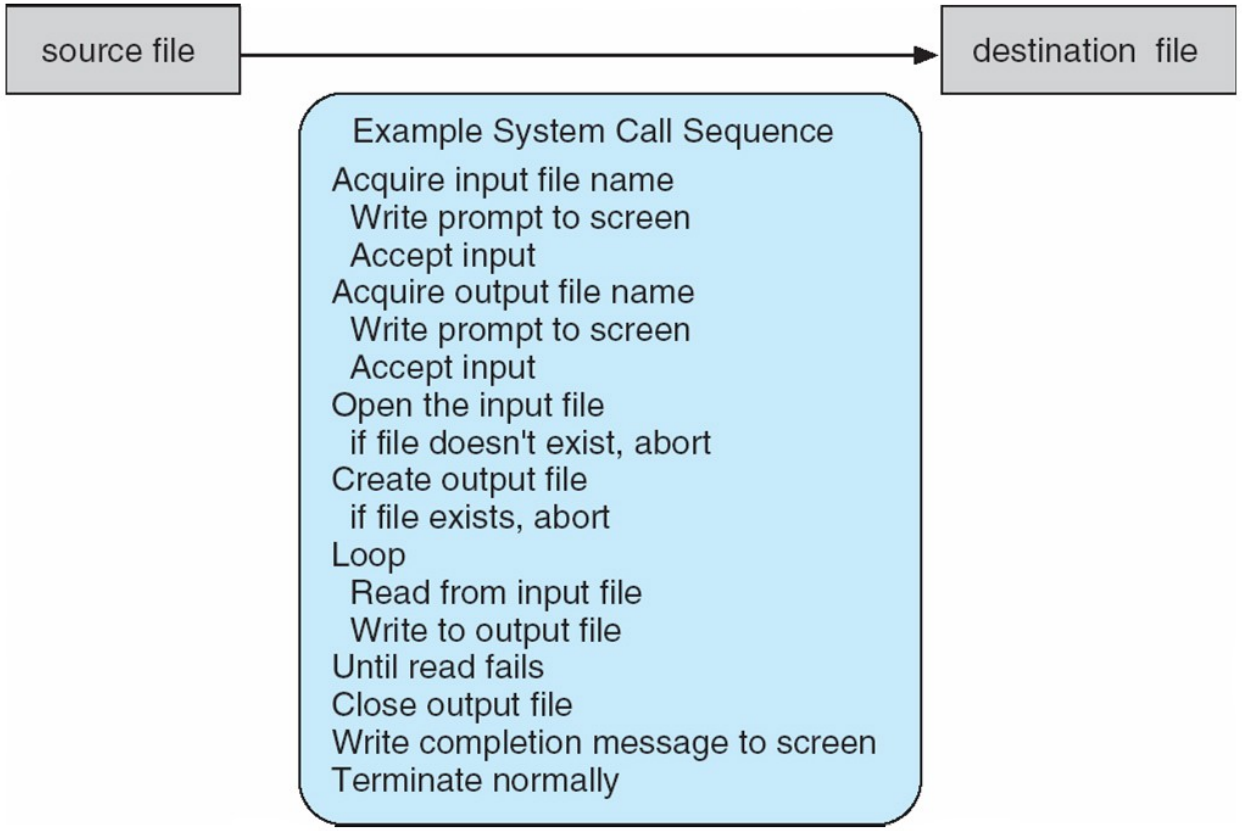
- Consider the ReadFile() function in the
- Win32 API—a function for reading from a file



- A description of the parameters passed to ReadFile()
  - HANDLE file—the file to be read
  - LPVOID buffer—a buffer where the data will be read into and written from
  - DWORD bytesToRead—the number of bytes to be read into the buffer
  - LPDWORD bytesRead—the number of bytes read during the last read
  - LPOVERLAPPED ovl—indicates if overlapped I/O is being used

4.2 Example of System Call

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



4.2 Example of System Calls

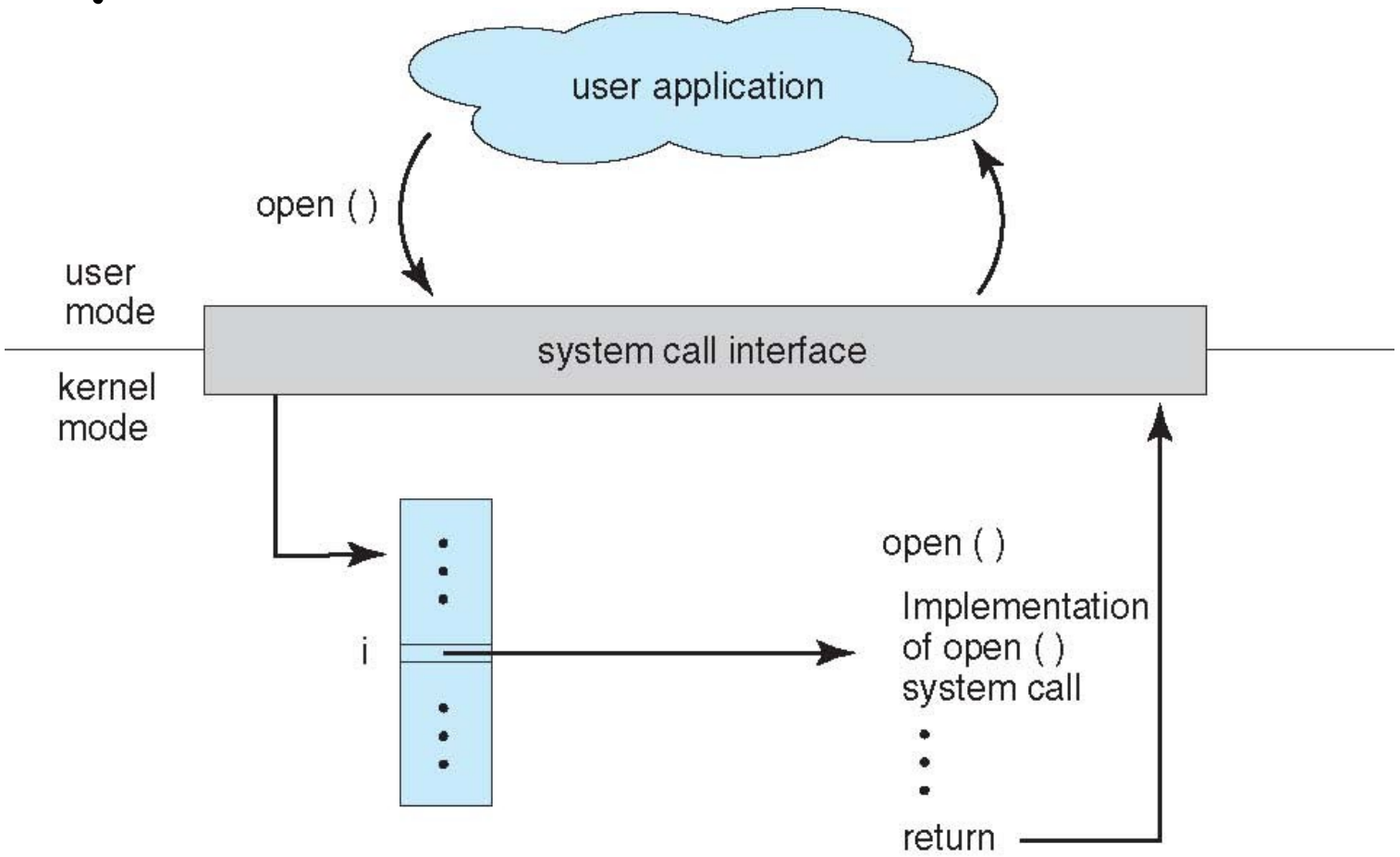
Several system calls are involved in this process.

1. The first call displays a message on the screen (monitor), followed by a prompt to enter the input filename. Then, another system call writes a message on the screen and prompts for the output filename.
2. When the program attempts to open the input file, it may discover that the file does not exist or is protected against access. In such cases, the program should display a message on the console (using another system call), terminate abnormally (through another system call), and then create a new file (yet another system call).
3. Once both files are opened, the program enters a loop to read from the input file (via another system call) and write to the output file (again, another system call).
4. Finally, after copying the entire file, the program should close both files (using another system call), display a message on the console or window (system call), and then terminate normally (the final system call).

4.3 System Call Implementation

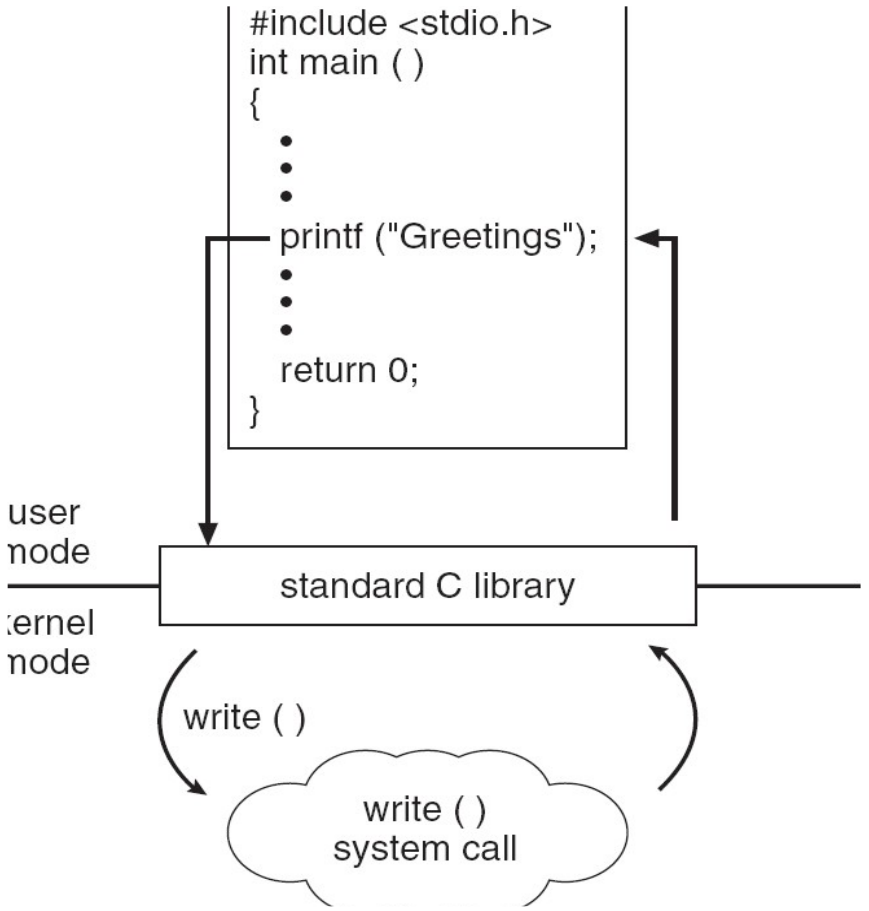
- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

4.3 System Call Implementation
API - System Call - OS relationship



4.4 Standard C Library Example

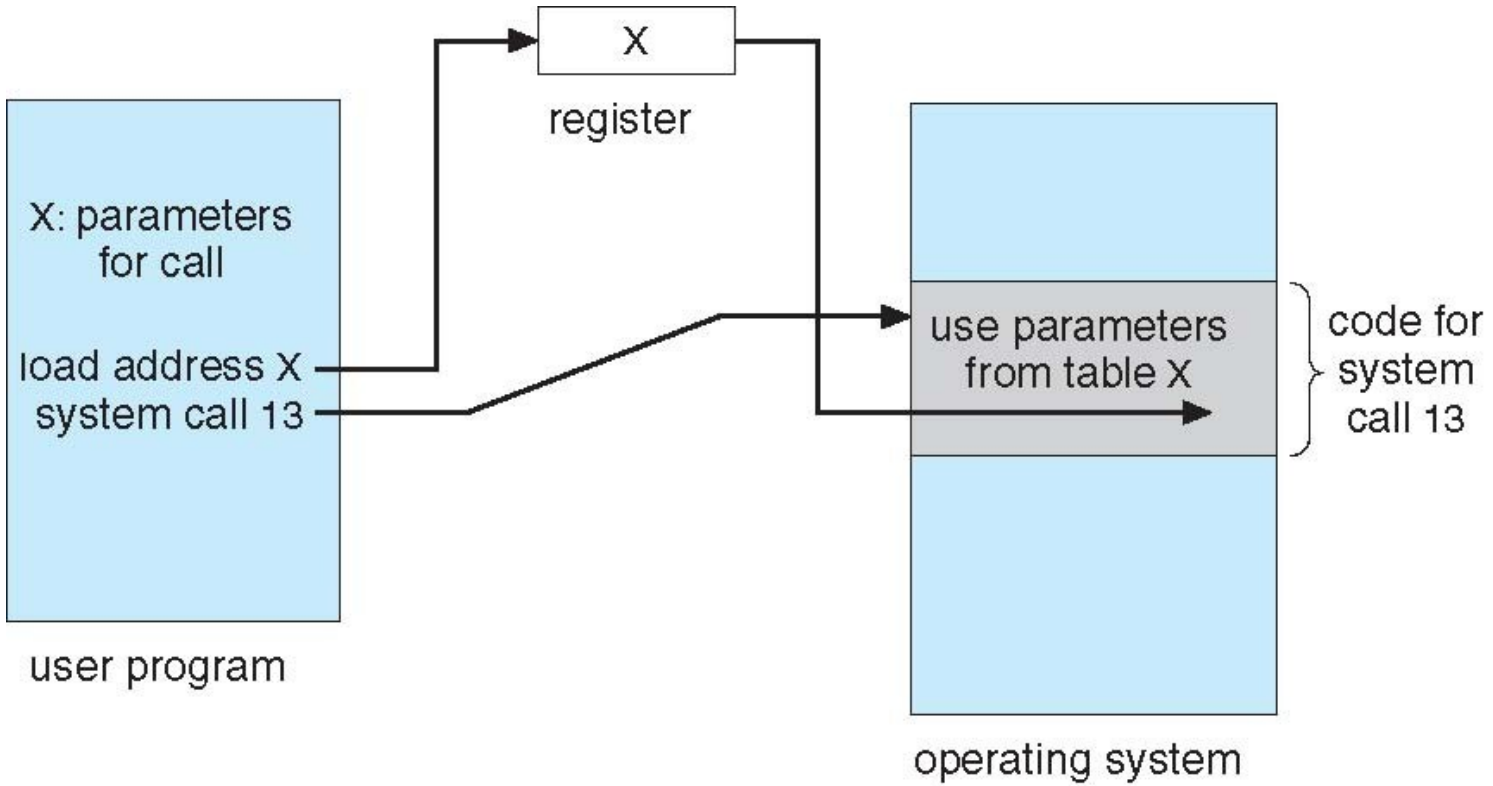
C program invoking printf() library call, which calls write() system call



4.5 System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in *registers*
    - In some cases, there may be more parameters than registers
  - Parameters stored in a *block,* or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or *pushed,* onto the *stack* by the program and *popped* off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

4.5 System Call: Parameter Passing via Table



4.6 Types of System Calls

The system calls can be categorized into six major categories:

1. Process Control
2. File management
3. Device management
4. Information management
5. Communications
6. Protection

4.6.1 Process Control System Calls

- **end, abort**
- **load, execute**
- **create process, terminate process**
- **get process attributes, set process attributes**
- **wait for time**
- **wait event, signal event**
- **allocate and free memory**

- Process control system calls include end, abort, load, execute, create process, terminate process, get/set process attributes, wait for time or event, signal event, and allocate and free memory.
- Processes must be created, launched, monitored, paused, resumed, and eventually stopped.
- When one process pauses or stops, then another must be launched or resumed
- Process attributes like process priority, max. allowable execution time etc. are set and retrieved by OS.
- After creating the new process, the parent process may have to wait (wait time), or wait for an event to occur (wait event). The process sends back a signal when the event has occurred (signal event)

4.6.2 File Management System Calls

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

- File management system calls include create file, delete file, open, close, read, write, reposition, get file attributes, and set file attributes.
- After **creating** a file, the file is **opened**. Data is **read** or **written** to a file.
- The file pointer may need to be **repositioned** to a point.
- The file **attributes** like filename, file type, permissions, etc. are set and retrieved using system calls.
- These operations may also be supported for directories as well as ordinary files.

4.6.3 Device Management System Call

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

- Device management system calls include **request device**, **release** device, **read**, **write**, **reposition**, **get/set** device attributes, and logically **attach** or **detach** devices.
- When a process needs a resource, a request for resource is done. Then the control is granted to the process. If requested resource is already attached to some other process, the requesting process has to wait.
- In multiprogramming systems, after a process uses the device, it has to be returned to OS, so that another process can use the device.
- Devices may be physical (e.g. disk drives), or virtual / abstract (e.g. files, partitions, and RAM disks).

4.6.4 Information maintenance System Calls

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

- Information maintenance system calls include calls to get/set the time, date, system data, and process, file, or device attributes.
- These system calls care used to transfer the information between user and the OS. Information like current time & date, no. of current users,

4.6.5 Communication System Calls

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

- Communication system calls create/delete communication connection, send/receive messages, transfer status information, and attach/detach remote devices.
- The **message passing** model must support calls to:
  - Identify a remote process and/or host with which to communicate.
  - Establish a connection between the two processes.
  - Open and close the connection as needed.
  - Transmit messages along the connection.
  - Wait for incoming messages, in either a blocking or non-blocking state.
  - Delete the connection when no longer needed.

4.6.5 Communication System Call

- The **shared memory** model must support calls to:
  - Create and access memory that is shared amongst processes (and threads.)
  - Free up shared memory and/or dynamically allocate it as needed.
- Message passing is simpler and easier, (particularly for inter-computer communications), and is generally appropriate for small amounts of data. It is easy to implement, but there are system calls for each read and write process.
- Shared memory is faster, and is generally the better approach where large amounts of data are to be shared. This model is difficult to implement, and it consists of only few system calls.

4.6.6 Protection System Call

- **Set permission**
- **Get Permission**

- Protection provides mechanisms for controlling which users / processes have access to which system resources.
- System calls allow the access mechanisms to be adjusted as needed, and for non- privileged users to be granted elevated access permissions under carefully controlled temporary circumstances

4.7 Examples of Windows and Unix System Calls

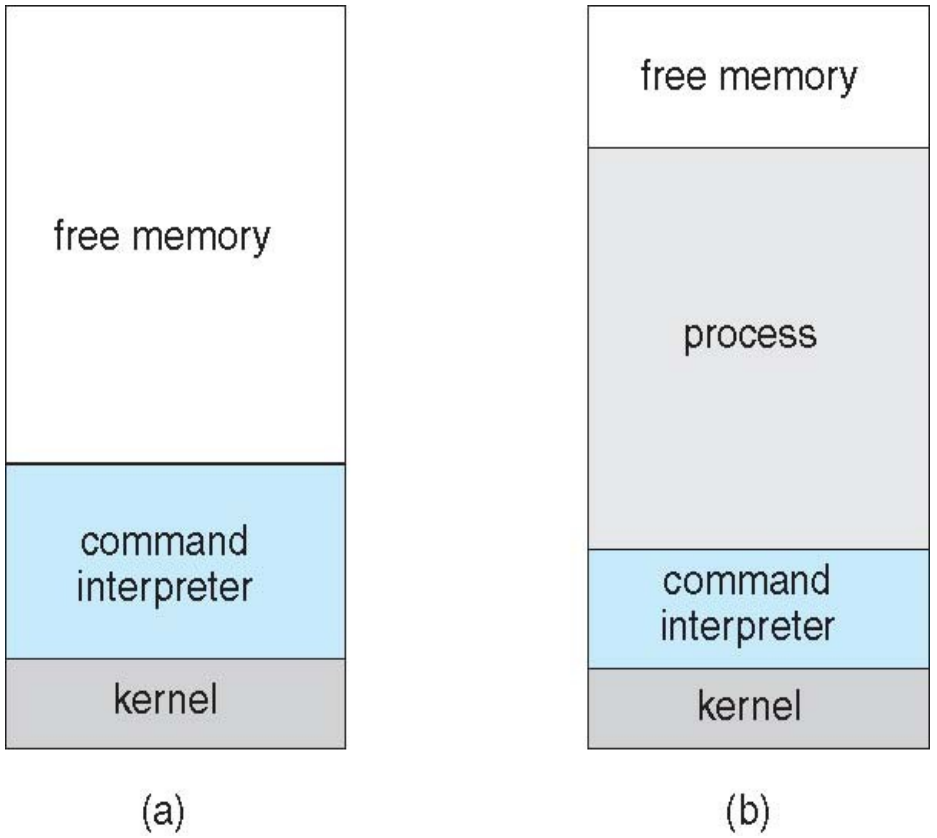
	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information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

4.7.1 Example: MS DOS

- MS-DOS (Microsoft Disk Operating System) is a command-line-based operating system developed by Microsoft, widely used in the 1980s and early 1990s.
- It allows users to manage files, run programs, and control hardware using text-based commands.
- MS-DOS was a foundational OS for early PCs before being replaced by graphical operating systems like Windows.

- Single-tasking
- Shell invoked when system booted
- Simple method to run program
  - No process created
- Single memory space
- Loads program into memory, overwriting all but the kernel
- Program exit -> shell reloaded

4.7.1 MS-DOS Execution



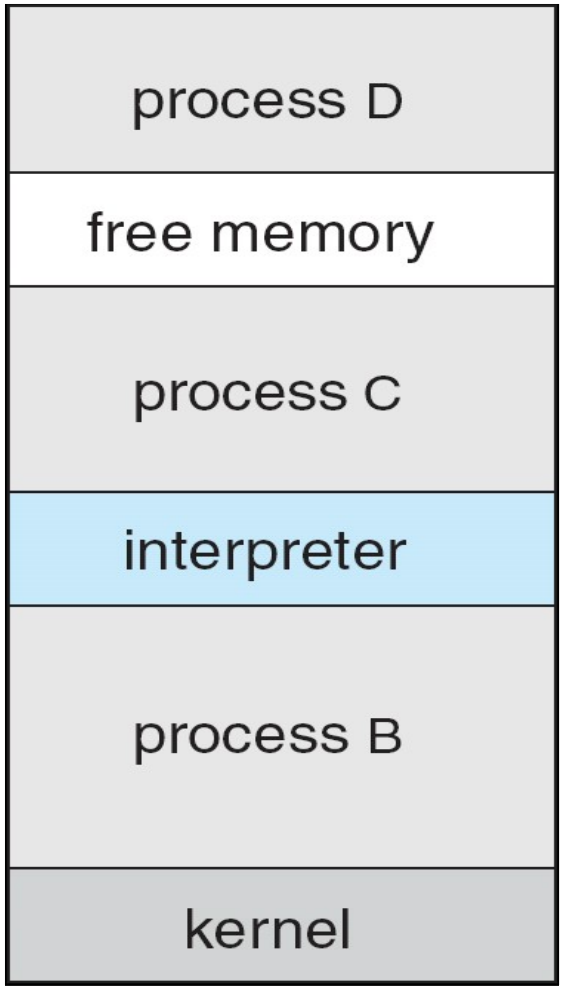
(a) At system startup (b) running a program

4.7.2 Example: FreeBSD

FreeBSD is a free, open-source Unix-like operating system derived from the Berkeley Software Distribution (BSD), a version of Unix developed at the University of California, Berkeley..

- Unix variant
- Multitasking
- User login -> invoke user's choice of shell
- Shell executes fork() system call to create process
  - Executes exec() to load program into process
  - Shell waits for process to terminate or continues with user commands
- Process exits with code of 0 – no error or > 0 – error code

4.7.3 Example: FreeBSD running multiple programs



5. System Programs

- A collection of programs that provide a convenient environment for program development and execution (other than OS) are called system programs or system utilities.
- System programs run in user space.
- Most users perceive the operating system through system programs, rather than through the underlying system calls.

5. System Programs

System Programs Categories:

1. **File management** - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
2. **Status information -** Utilities to check on the date, time, number of users, processes running, data logging, etc.
3. **File modification**
   - Text editors to create and modify files
   - Special commands to search contents of files or perform transformations of the text

5. System Programs

4. **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided
5. **Program loading and execution**- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
6. **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
   - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another

6. Operating System Design and Implementation

- The first step in designing a operating system is to define its goals and specifications. At the highest level, the system's design is influenced by the choice of hardware and the type of system, whether it’s batch, time-shared, single-user, multi-user, distributed, real-time, or general-purpose.
- Beyond this broad level, specifying detailed requirements becomes more challenging. These requirements can be categorized into two main groups:
  1. User goals (user requirements)
  2. System goals (system requirements)
- User requirements focus on features important to the user, such as convenience, ease of use, reliability, safety, and speed.
- System requirements are written for developers, the ones designing the OS. These include ease of design, implementation, and maintenance, as well as flexibility, reliability, error-free operation, and efficiency

6.1 Operating System Design and Implementation

"Policies define what needs to be accomplished, while mechanisms specify how it will be carried out."

- Example: In a timer, the counter and its decrementing function represent the mechanism, while deciding the time duration to be set is part of the policy.
- Policies can evolve over time. In the worst-case scenario, every policy change could necessitate modifications to the underlying mechanism.
- However, when policies and mechanisms are properly separated, policy changes can be easily managed by adjusting parameters or updating data/configuration files without rewriting the code."

6.2 Operating System Design and Implementation

Implementation

- Traditionally OS were written in assembly language.
- In recent years, OS are written in C, or C++. Critical sections of code are still written in assembly language.
- The first OS that was not written in assembly language, it was the Master Control Program (MCP).
- The advantages of using a higher-level language for implementing operating systems are: The code can be written faster, more compact, easy to port to other systems and is easier to understand and debug.
- The only disadvantages of implementing an operating system in a higher-level language are reduced speed and increased storage requirements.

7. Operating System Structure

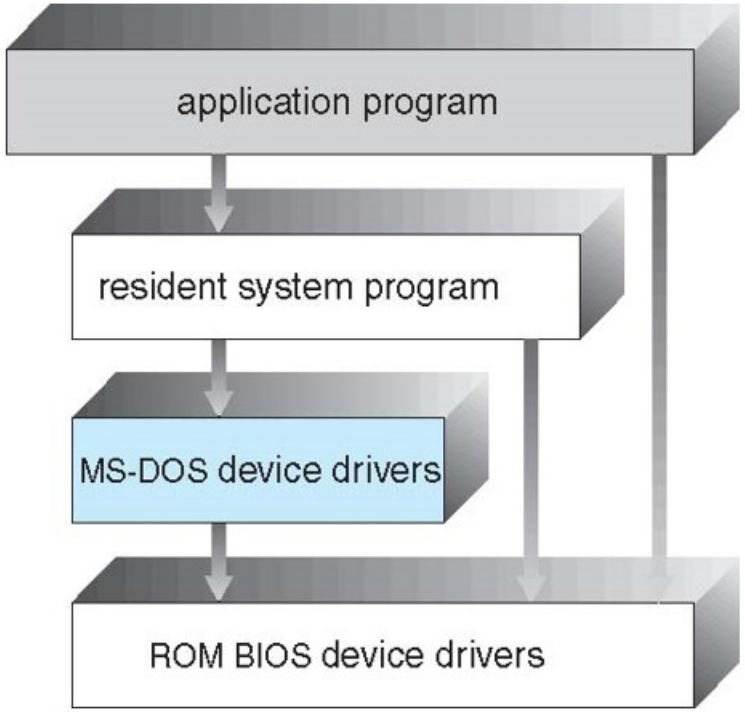
Four main approaches to structure the Operating System

1. Simple Structure
2. Layered Approach
3. Microkernel
4. Modules

7.1 Simple Structure

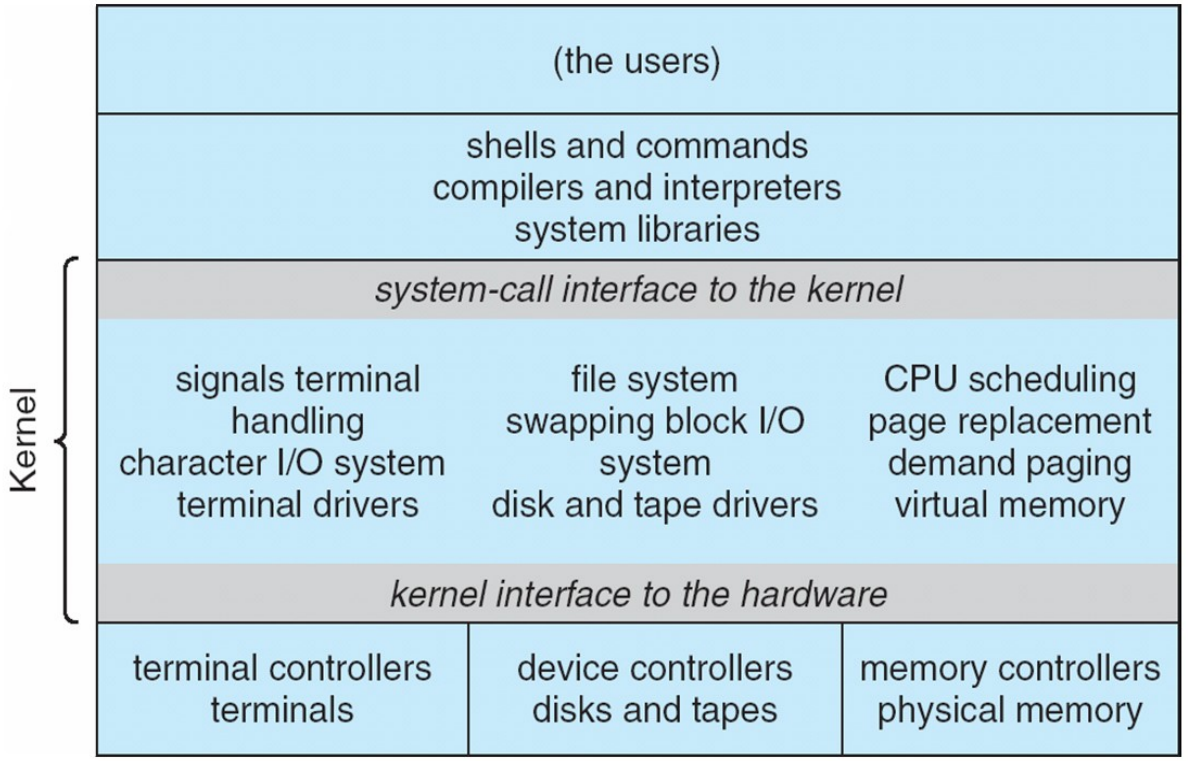
Many operating systems do not have well-defined structures. They started as small, simple, and limited systems and then grew beyond their original scope. Eg: MS-DOS, original UNIX OS

7.1.1 Simple Structure: MS-DOS



- In MS-DOS, the interfaces and levels of functionality are not well separated.
- Application programs can access basic I/O routines to write directly to the display and disk drives.
- Such freedom leaves MS-DOS in bad state and the entire system can crash down when user programs fail.

7.1.2 Simple Structure: UNIX



UNIX - limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts

- **Systems programs**
- **The kernel**
  - Consists of everything below the system-call interface and above the physical hardware
  - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level

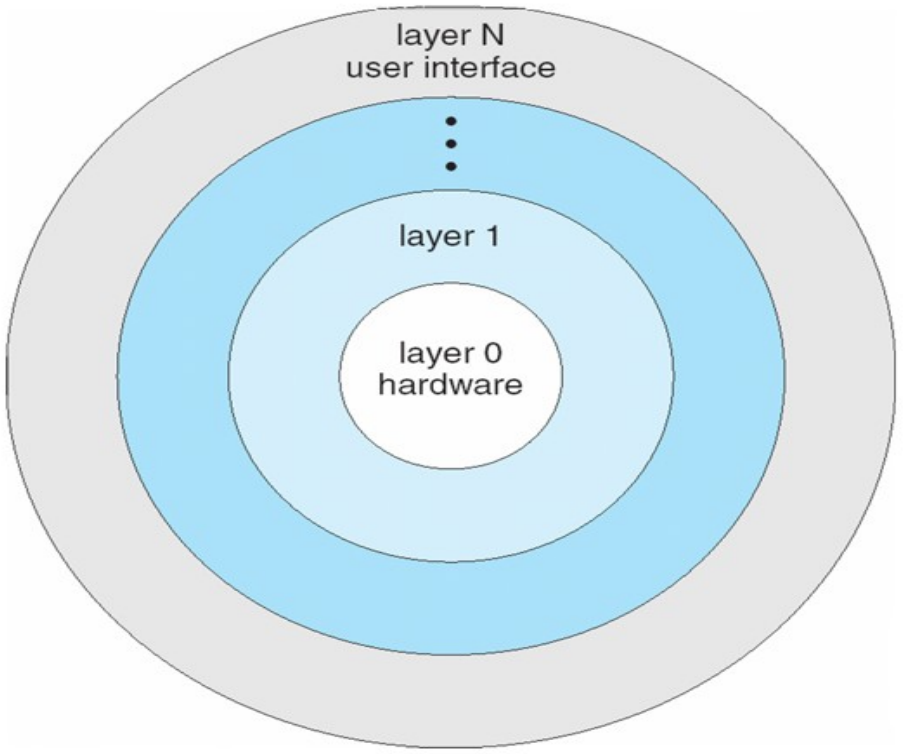
7.1.3 Simple Structure: UNIX

- A **monolithic structure** refers to a design where the entire operating system-including the kernel and all services like device drivers, file system management, memory management, and network management - runs in a single large block of code in **kernel mode**.
- **Example of Monolithic Kernels:**
  - **Linux Kernel:** Although Linux has been modularized to some extent (allowing loadable kernel modules), it is fundamentally monolithic.
  - **Unix:** The traditional Unix operating system uses a monolithic kernel structure.
- **Lack of Modularity:** A monolithic kernel can become very large and complex, making it harder to maintain, extend, or debug. Any error in a single component (such as a driver) can potentially crash the entire system.

7.2 Layered Approach

- With proper hardware support, operating systems can be divided into smaller, more suitable components than those permitted by MS-DOS and UNIX. A system can be made modular in many ways. One method is the layered approach, in which the operating system is broken into layers.
- 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.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers

7.2 Layered Approach

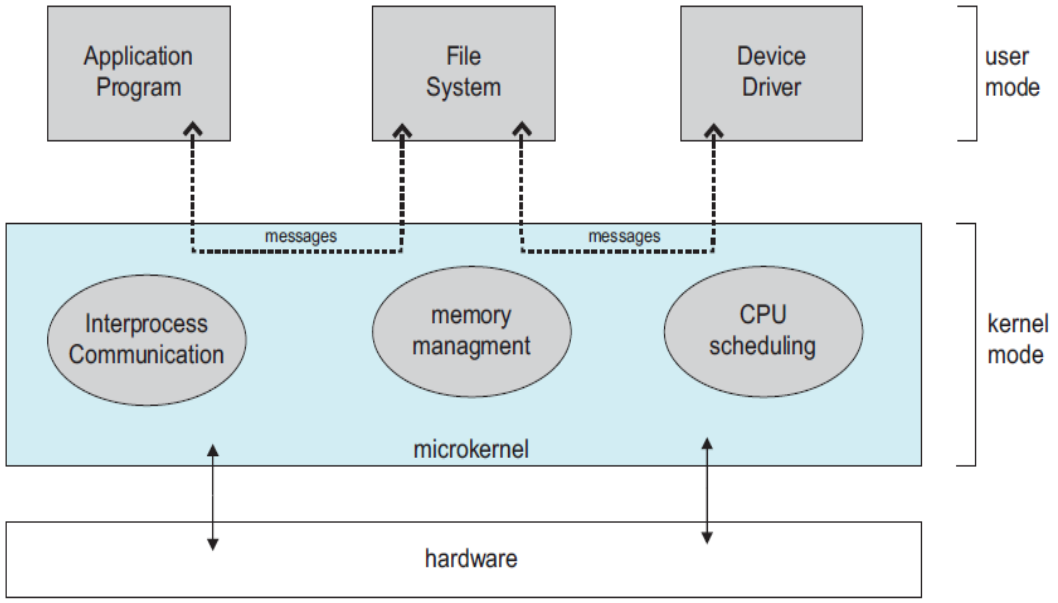


- Advantage of layered approach is simplicity of construction and debugging.
- Disadvantages of layered approach:
  - The various layers must be appropriately defined, as a layer can use only lower-level layers.
  - Less efficient than other types, because any interaction with layer 0 required from top layer. The system call should pass through all the layers and finally to layer 0. This is an overhead.

7.3 Microkernel

- This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs.
- The microkernel only handles essential tasks such as process scheduling, memory management, and IPC, while moving other services (like device drivers and file systems) to user space.
- 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 through message passing.

7.3 Architecture of Microkernel

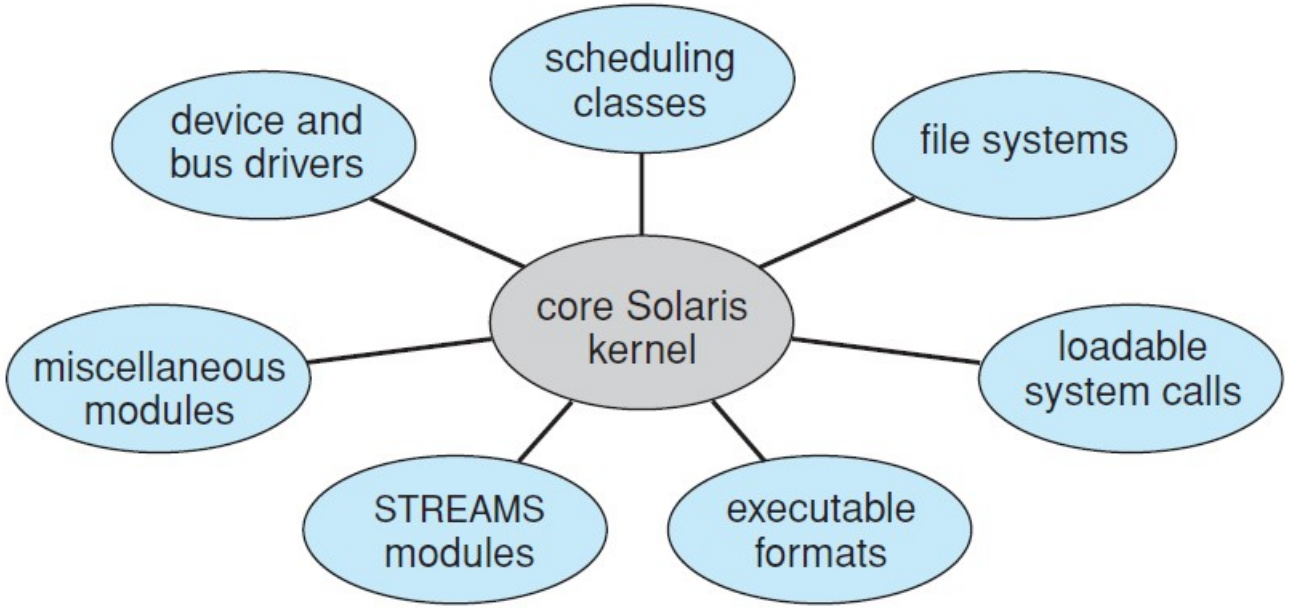


- For example, if the client program wants to access a file, it must interact with the file server.
- The client program and service never interact directly. They communicate indirectly by exchanging messages with the microkernel.

7.4 Modules

- The best method for designing operating systems today is using loadable kernel modules. In this approach, the kernel has a basic set of components, and extra services can be added through modules, either when the system starts or while it’s running.
- This design is common in modern versions of UNIX, like Solaris, Linux, and macOS, as well as in Windows.
- Features
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible

7.4 Solaris Loadable Modules



8. Virtual Machines

- Virtual machines create a software-based imitation of physical hardware, allowing multiple operating systems to run on a single physical machine simultaneously.
- A virtual machine provides an interface *identical* to the underlying bare hardware.
- The operating system **host** creates the illusion that a process has its own processor and (virtual memory).
- Each **guest** provided with a (virtual) copy of underlying computer.

8.1 Virtual Machine History

- Early Development (1960s):
  - The concept of virtual machines originated in the 1960s, with IBM's CP/CMS operating system for mainframes. It allowed multiple users to share the same physical machine by running separate instances of an operating system.
- Commercialization (1970s):
  - IBM continued to develop virtual machine technology, making it available on its mainframe computers in the 1970s. This era marked the first widespread commercial use of VMs for multitasking and resource sharing.
- Decline and Revival (1980s-1990s):
  - Interest in VMs declined with the rise of inexpensive personal computers. However, in the late 1990s, VM technology saw a resurgence with the introduction of VMware, which brought virtualization to x86-based architectures, making it accessible for smaller servers and PCs.
- Modern Expansion (2000s-Present):
  - With the growth of cloud computing, virtual machines became essential for providing scalable, on-demand computing resources. Companies like Amazon (AWS) and Microsoft (Azure) use VMs as the backbone of cloud infrastructure services.

8.2 Virtualisation and Implementation

- Difficult to implement - must provide an exact duplicate of underlying machine
- Typically runs in user mode, creates virtual user mode and virtual kernel mode
- Timing can be an issue - slower than real machine
- A **hypervisor** manages virtual machines by providing the virtualization layer that allows multiple operating systems to share the physical hardware without interfering with one another.
  - **Type 1 (Bare-metal)**: Runs directly on hardware without needing an operating system (e.g., VMware ESXi, Microsoft Hyper-V).
  - **Type 2 (Hosted)**: Runs on top of an existing operating system (e.g., VirtualBox, VMware Workstation).

8.2 Virtualisation and Implementation

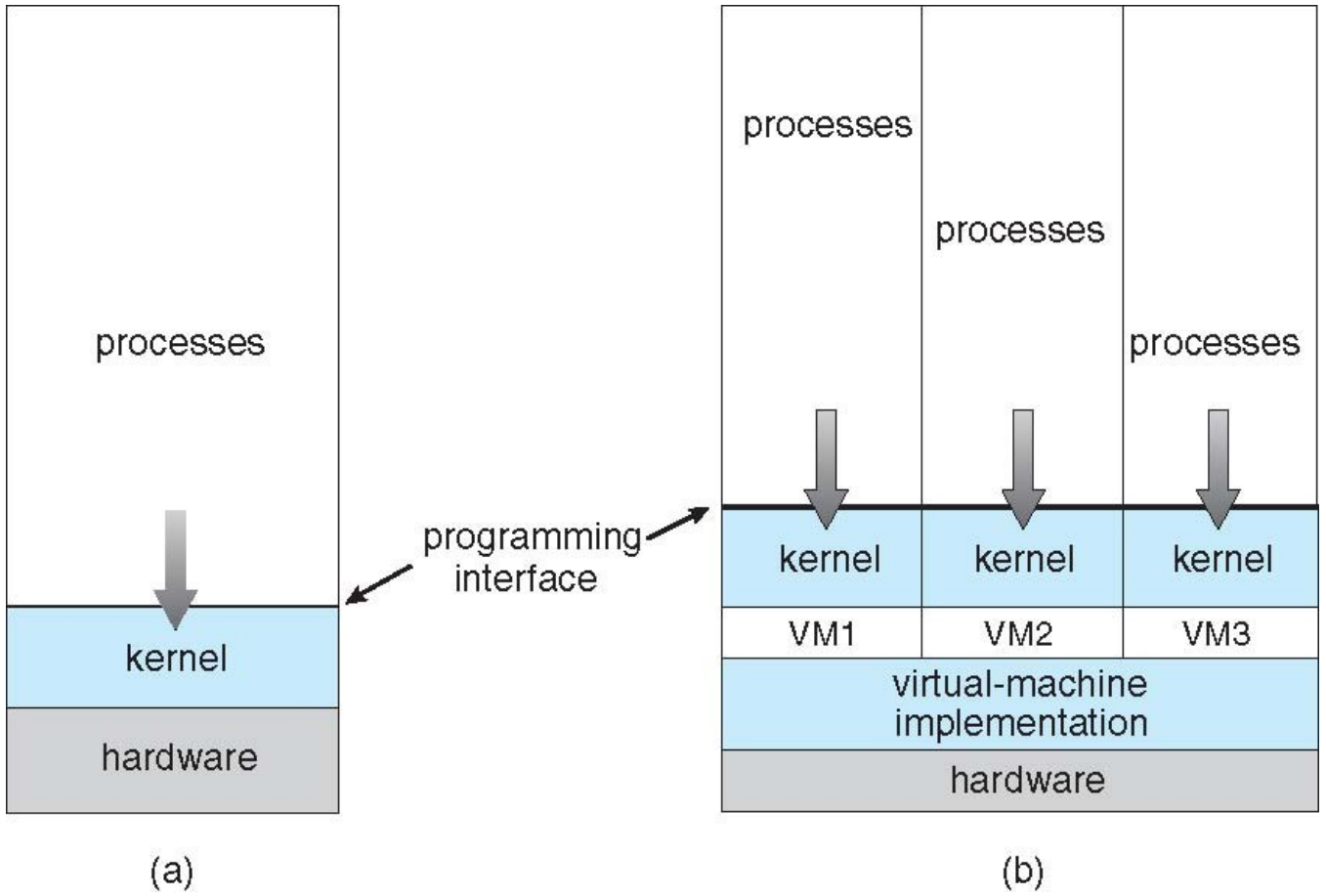


Fig: System Models (a) Nonvirtual Machine (b) Virtual Machine

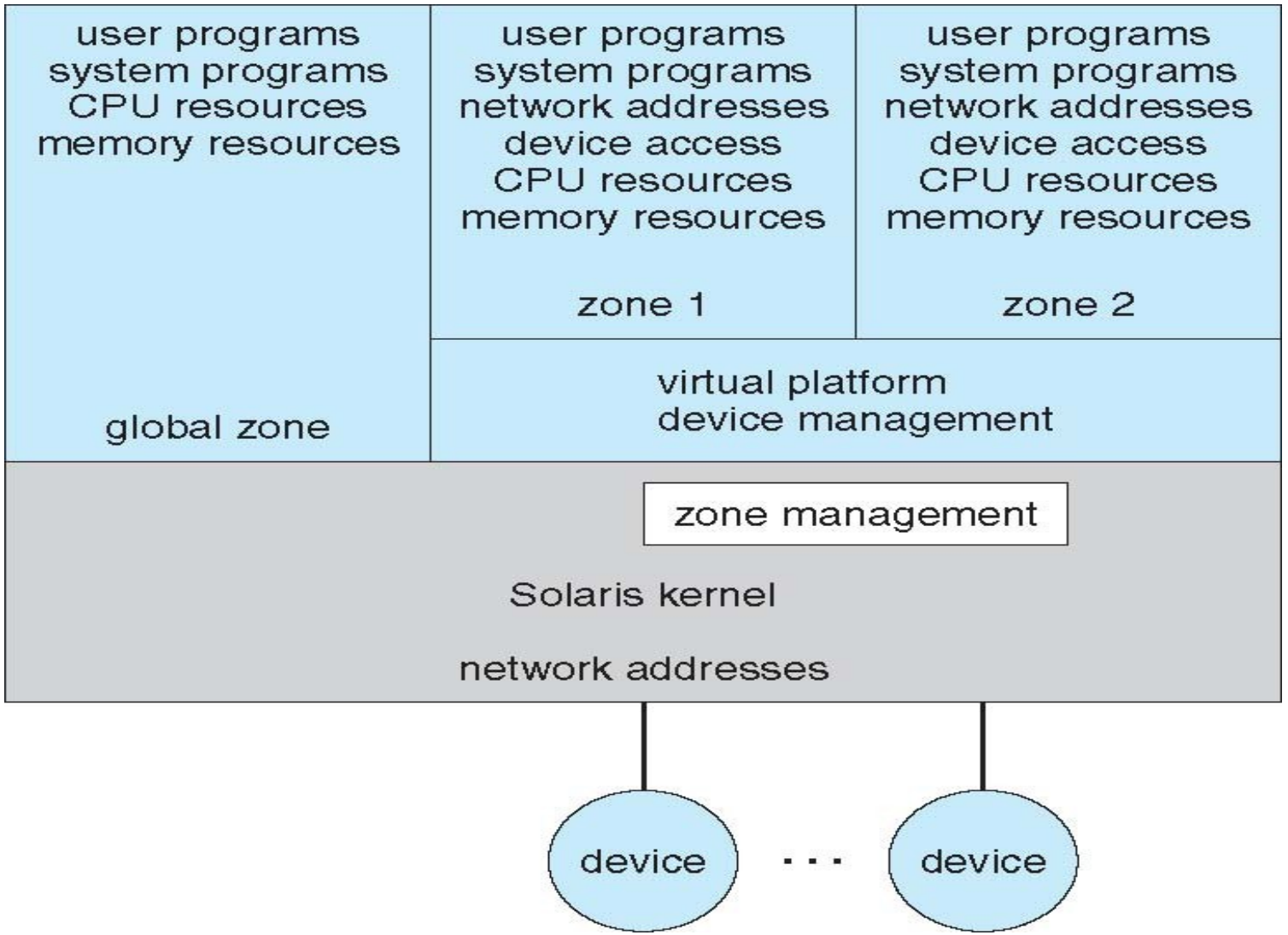
8.3 Virtual Machine Benefits

- **Cost Savings:** Virtual machines let you run multiple systems on one physical computer, reducing hardware costs.
- **Efficient Resource Use:** VMs make better use of CPU, memory, and storage by sharing resources among several systems.
- **Flexibility:** You can easily create, move, or delete VMs without affecting the main computer.
- **Isolation:** Each VM is separate, so if one crashes, it won't affect others or the host system.

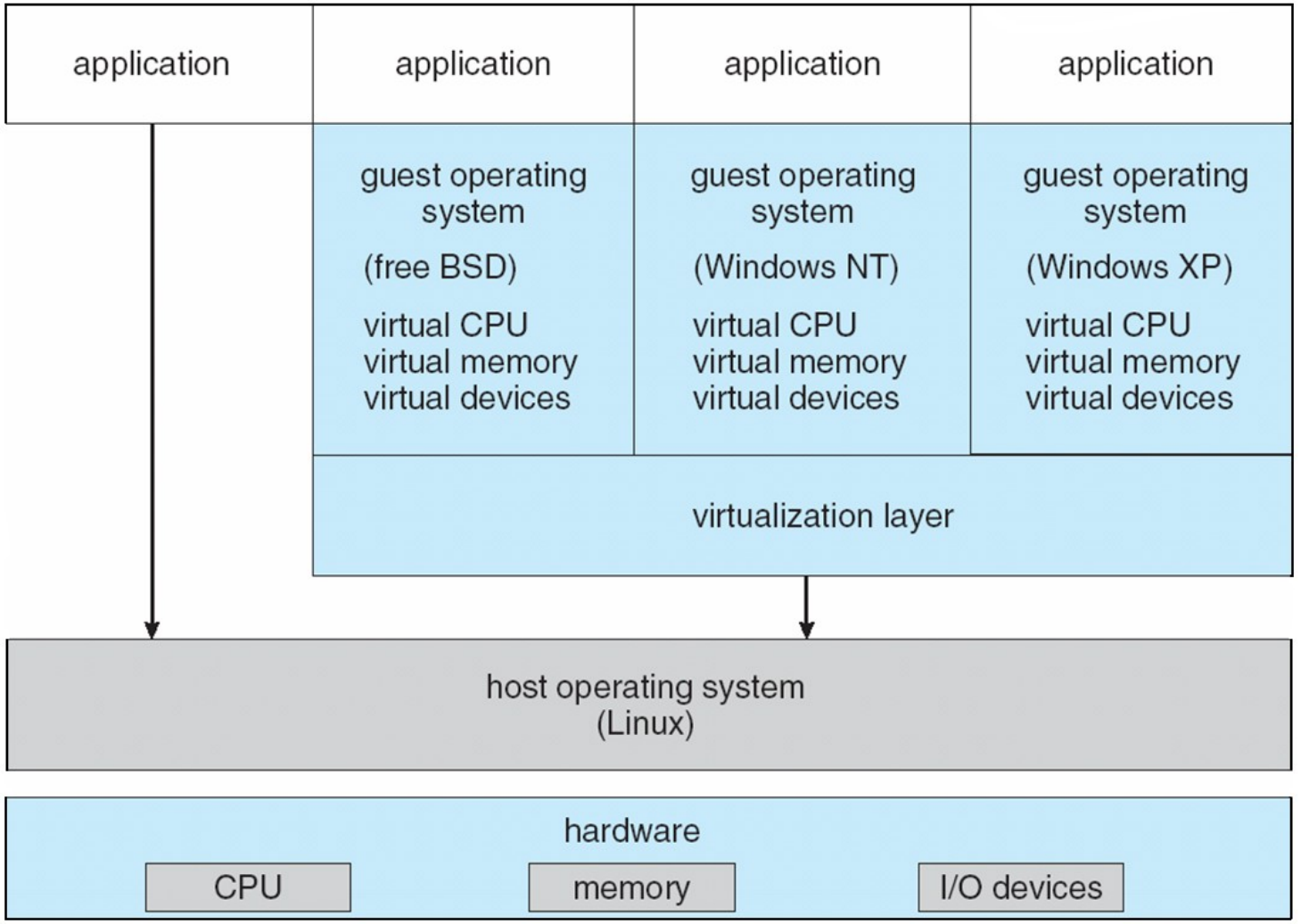
8.4 Para Virtualization

- **Paravirtualization** is a virtualization technique where the guest operating system is aware that it is running in a virtualized environment.
- Unlike traditional full virtualization, where the guest OS runs without any modification, in paravirtualization,
  - the guest OS is modified to interact directly with the hypervisor for improved performance.
- Guest can be an OS, or in the case of Solaris 10 applications running in **containers**

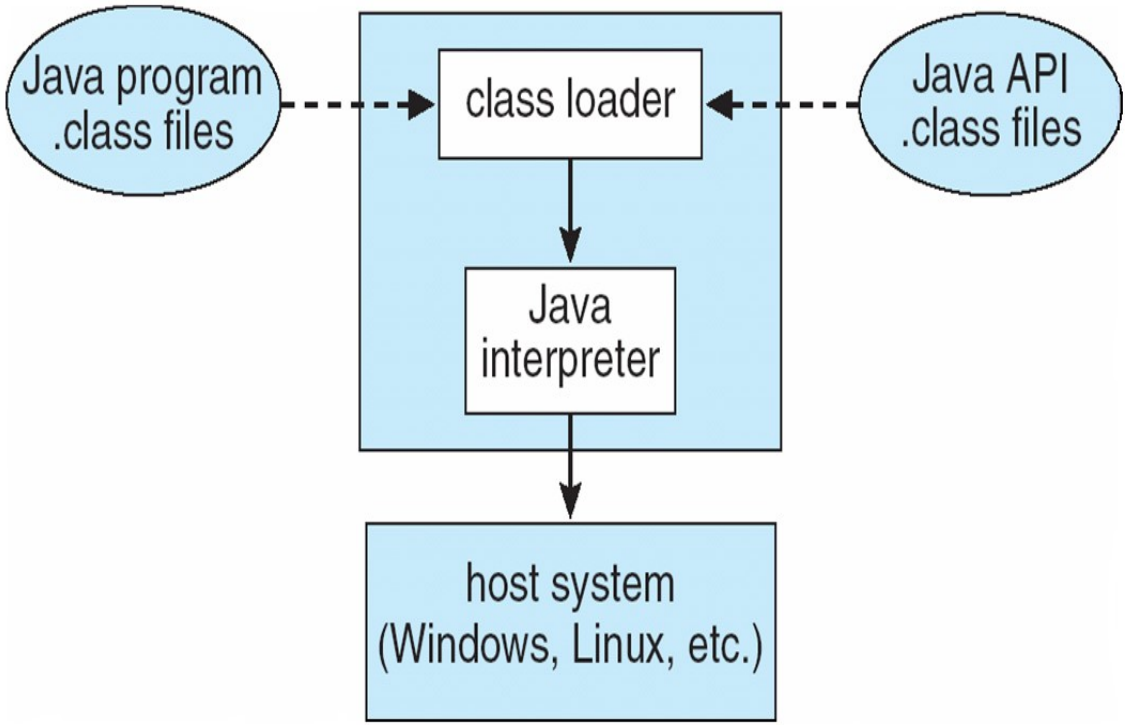
8.5 Solaris 10 with Containers



8.6 VMware Architecture



8.7 Java Virtual Machines



- **Platform Independence:** JVM allows Java programs to run on any OS by translating bytecode to machine-specific code.
- **Memory Management:** Automatic garbage collection handles memory allocation and deallocation.
- **Security:** JVM ensures a secure execution environment through bytecode verification and access controls.
- **Performance Optimization:** JIT compilation improves runtime performance by converting bytecode to native code.

9. Operating System Debugging

- Debugging is finding and fixing errors, or bugs
- OSes generate log files containing error information
- Failure of an application can generate **core dump** file capturing memory of the process
- Operating system failure can generate **crash dump** file containing kernel memory

10. Operating System Generation

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site.
- SYSGEN prepares and customizes the operating system to work efficiently on specific hardware.

11. System Boot

1. **Booting the System:** The process of starting a computer by loading the operating system kernel is known as booting.
2. **Bootstrap Program:** A small piece of code, called the bootstrap loader, is responsible for locating and loading the operating system kernel into memory.
3. **Two-Step Boot Process:** On some systems, the bootstrap loader fetches a more complex boot program from the disk, which then loads the kernel.
4. **Stored in ROM:** The initial bootstrap program is often stored in read-only memory (ROM), since RAM is in an unknown state at startup.
5. **Diagnostic Tasks:** The bootstrap program can run diagnostics to check the state of the hardware before proceeding with loading the operating system.

11. System Boot

6. OS Booting

- Systems like cell phones and tablets often store the entire operating system in ROM. Storing the operating system in ROM is suitable for small operating systems. But changing the bootstrap code requires changing the ROM hardware chips.
  - Some systems use erasable programmable read-only memory (EPROM) to make it possible to update the firmware(**Firmware** is software that is built into a device to control how it operates.) without changing the hardware.
- For larger or frequently updated operating systems (e.g., Windows, Mac OS, Linux), the bootstrap is stored in firmware, and the OS is loaded from disk.

11. System Boot

7. **Boot Block**: The bootstrap program reads a block (like block zero) from the disk to start loading the operating system into memory.
   - A **boot block** is a specific section of a storage device (like a hard drive or SSD) that contains the code needed to start the booting process of an operating system

Smile

IT INCREASES

Your

FACE Value ☺