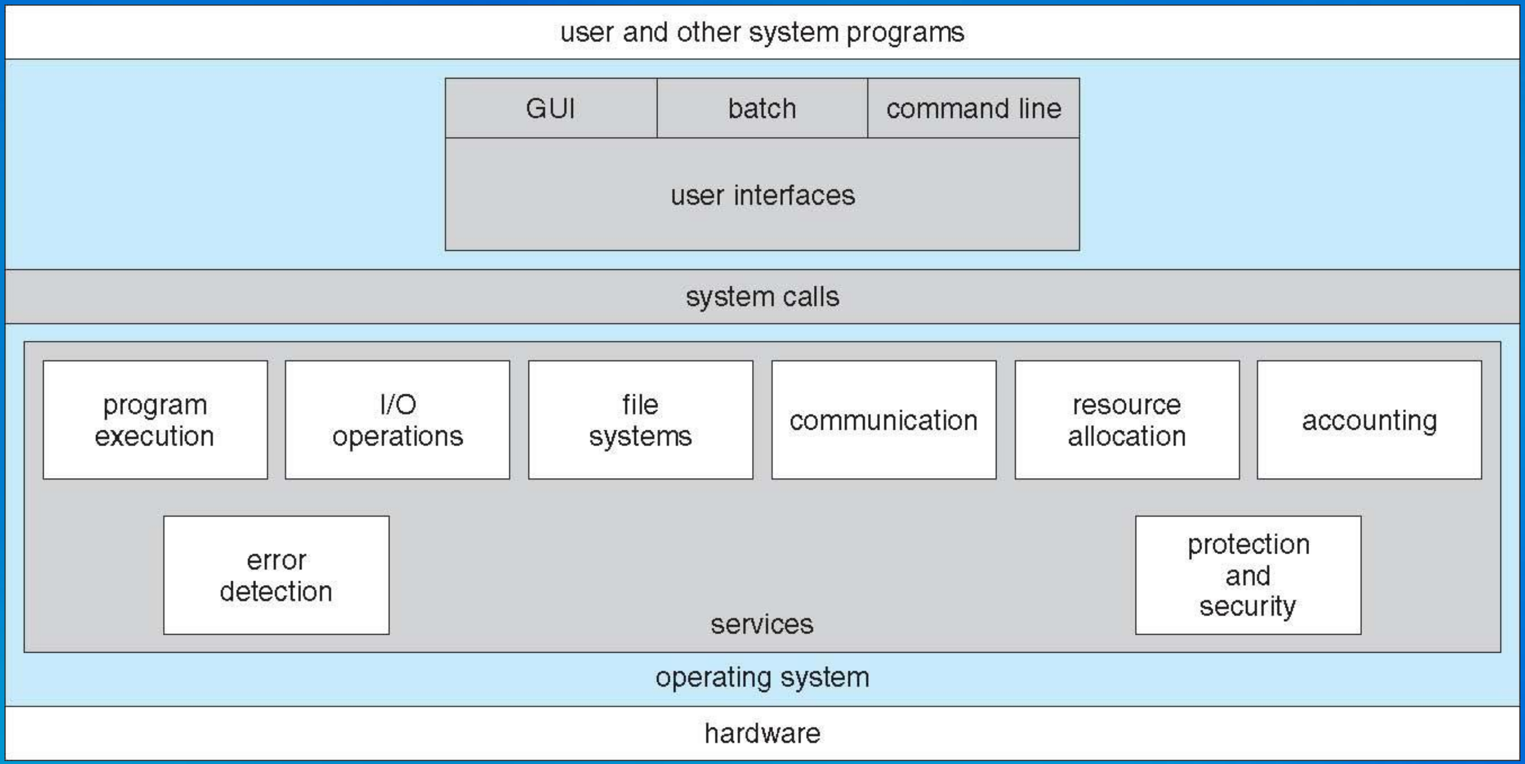
Chapter 2: Operating-System Structures

Operating System Services

- One set of operating-system services provides functions that are helpful to the user:
  - User interface - Almost all operating systems have a user interface (UI)
    - Varies between Command-Line (CLI), Graphics User Interface (GUI), Batch
  - Program execution - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - I/O operations - A running program may require I/O, which may involve a file or an I/O device
  - File-system manipulation - The file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

A View of Operating System Services



Operating System Services (Cont)

- One set of operating-system services provides functions that are helpful to the user (Cont):
  - Communications – Processes may exchange information, on the same computer or between computers over a network
    - Communications may be via shared memory or through message passing (packets moved by the OS)
  - Error detection – OS needs to be constantly aware of possible errors
    - May occur in the CPU and memory hardware, in I/O devices, in user program
    - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
    - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

Operating System Services (Cont)

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code
  - **Accounting -** To keep track of which users use how much and what kinds of computer resources
  - **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
    - **Protection** involves ensuring that all access to system resources is controlled
    - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts
    - If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

User Operating System Interface - CLI

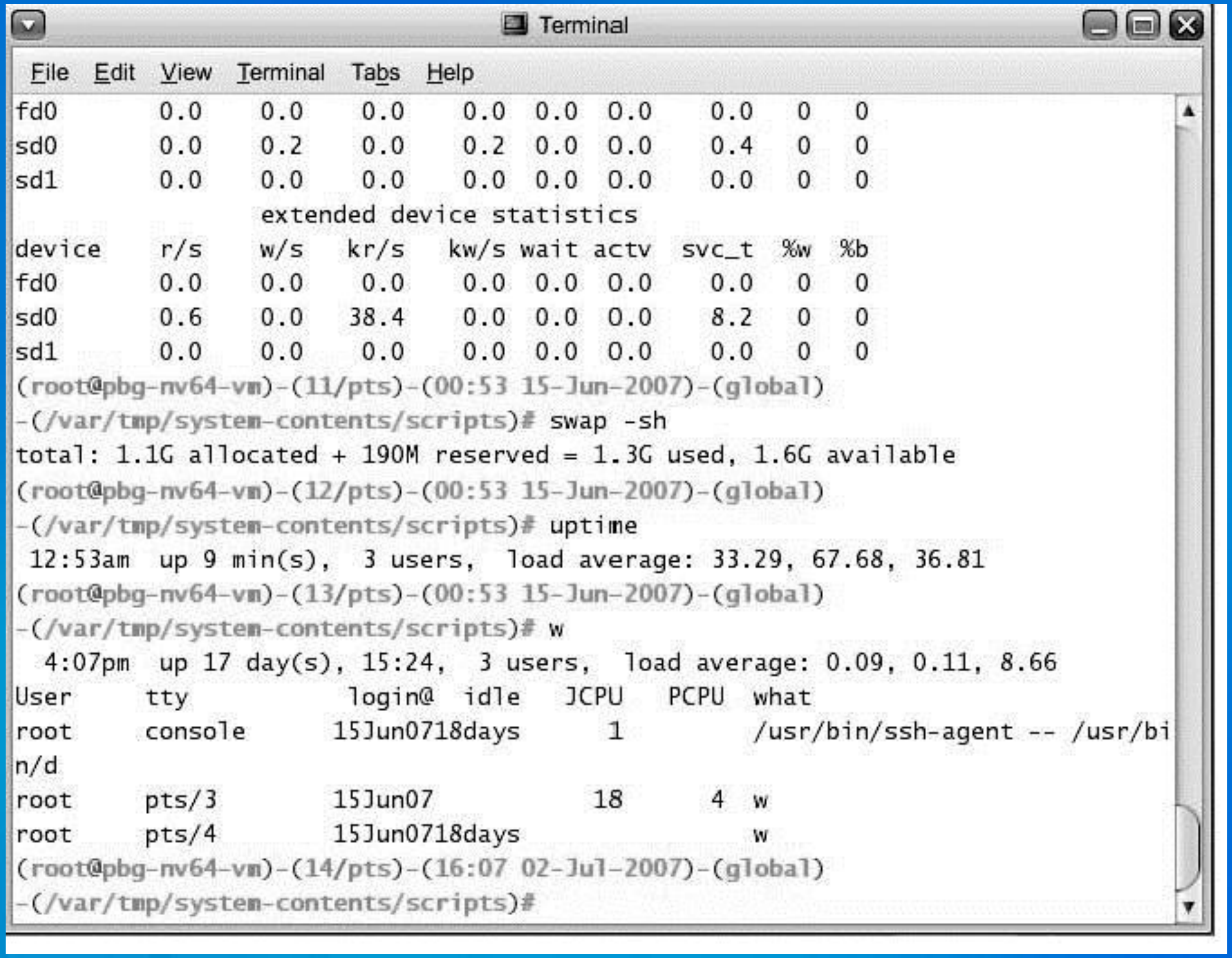
Command Line Interface (CLI) or command interpreter allows direct command entry

- Sometimes implemented in kernel, sometimes by systems program
- Sometimes multiple flavors implemented – shells
- Primarily fetches a command from user and executes it
  - Sometimes commands built-in, sometimes just names of programs
    - If the latter, adding new features doesn’t require shell modification

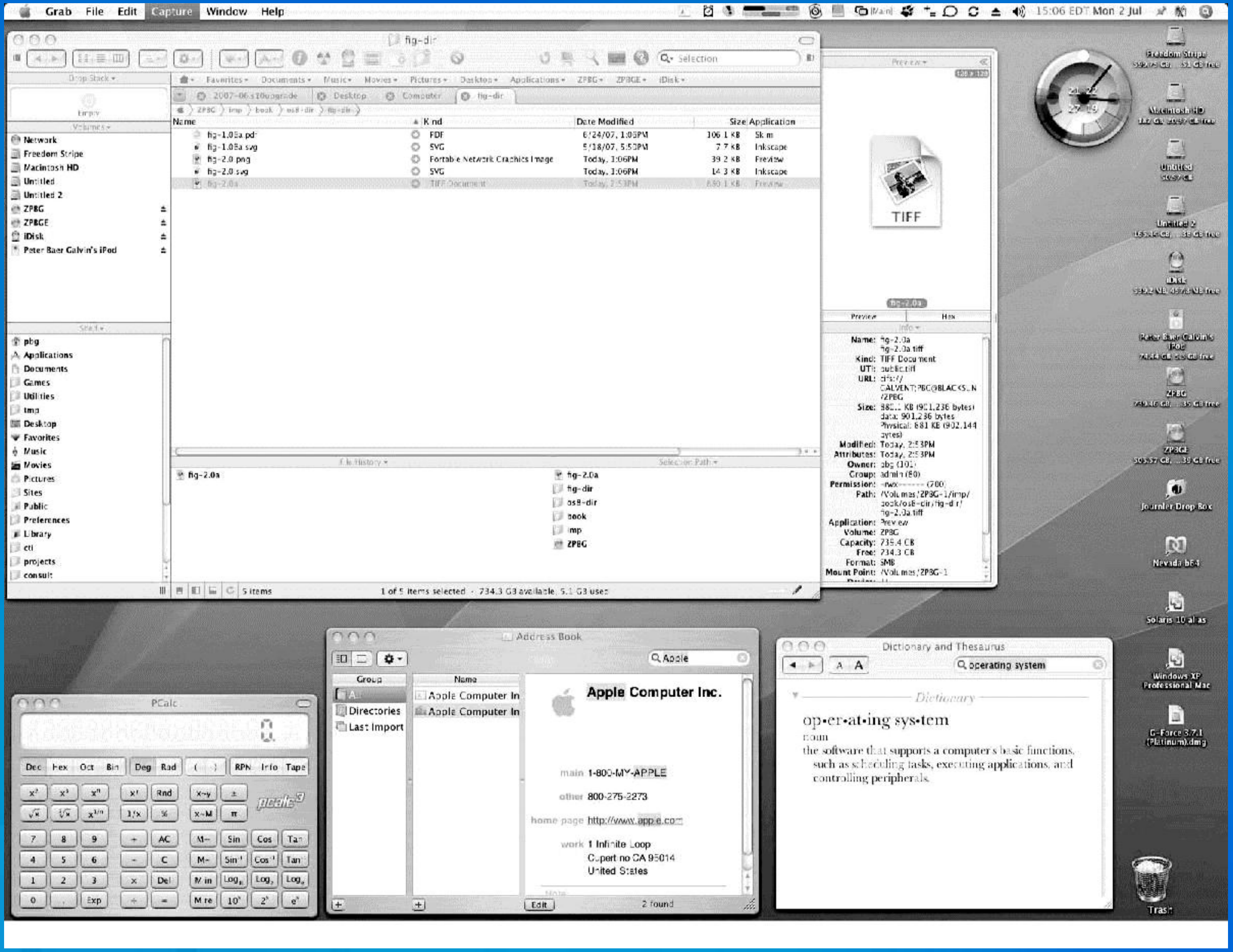
User Operating System Interface - GUI

- User-friendly desktop metaphor interface
  - Usually mouse, keyboard, and monitor
  - Icons represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a folder)
  - Invented at Xerox PARC
- 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)

Bourne Shell Command Interpreter



The Mac OS X GUI



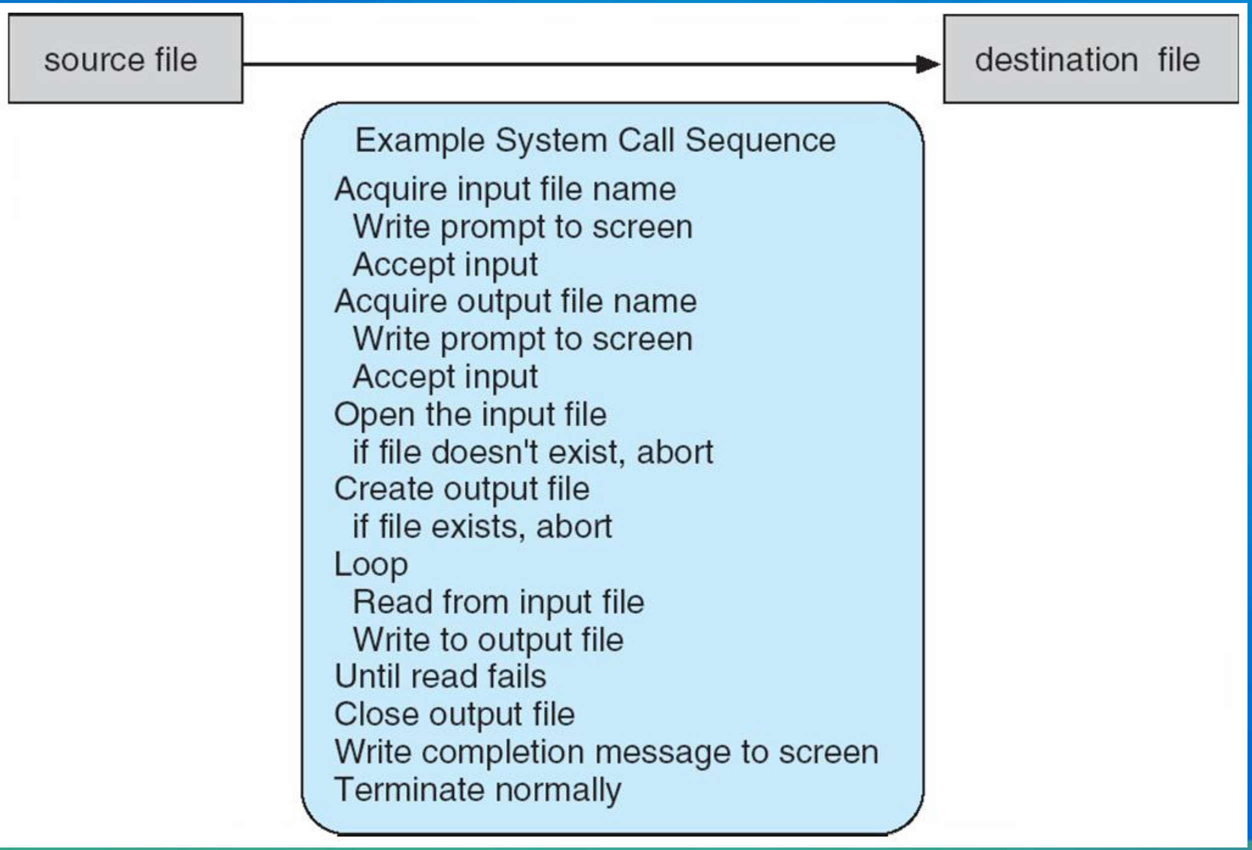
System Calls

- Programming interface to the services provided by the OS
- 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)
  - Why use APIs rather than system calls?
-

(Note that the system-call names used throughout this text are generic)

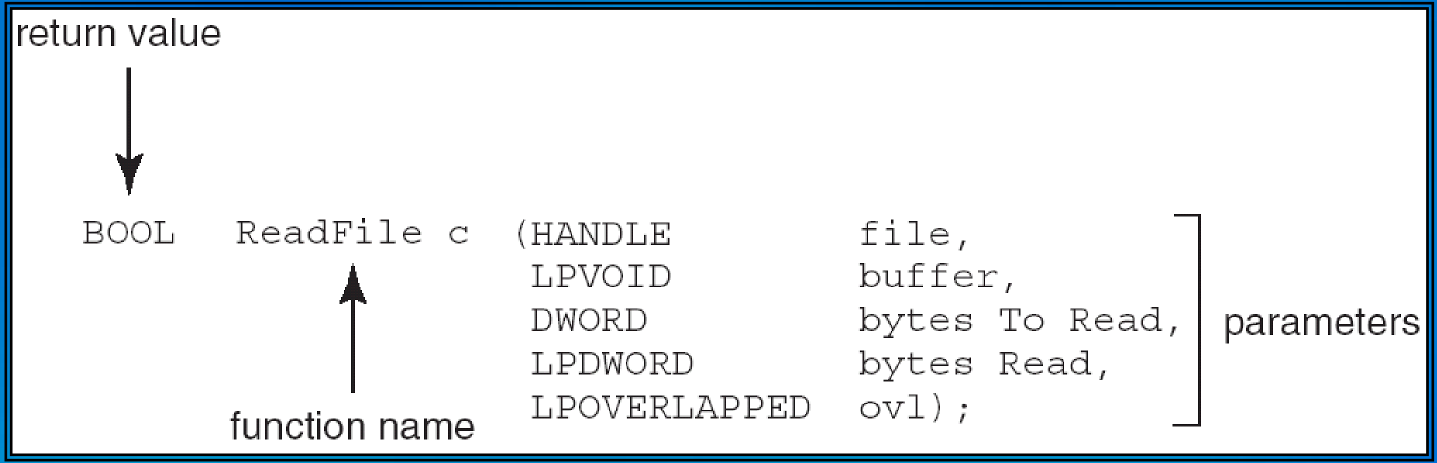
Example of System Calls

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



Example of Standard API

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

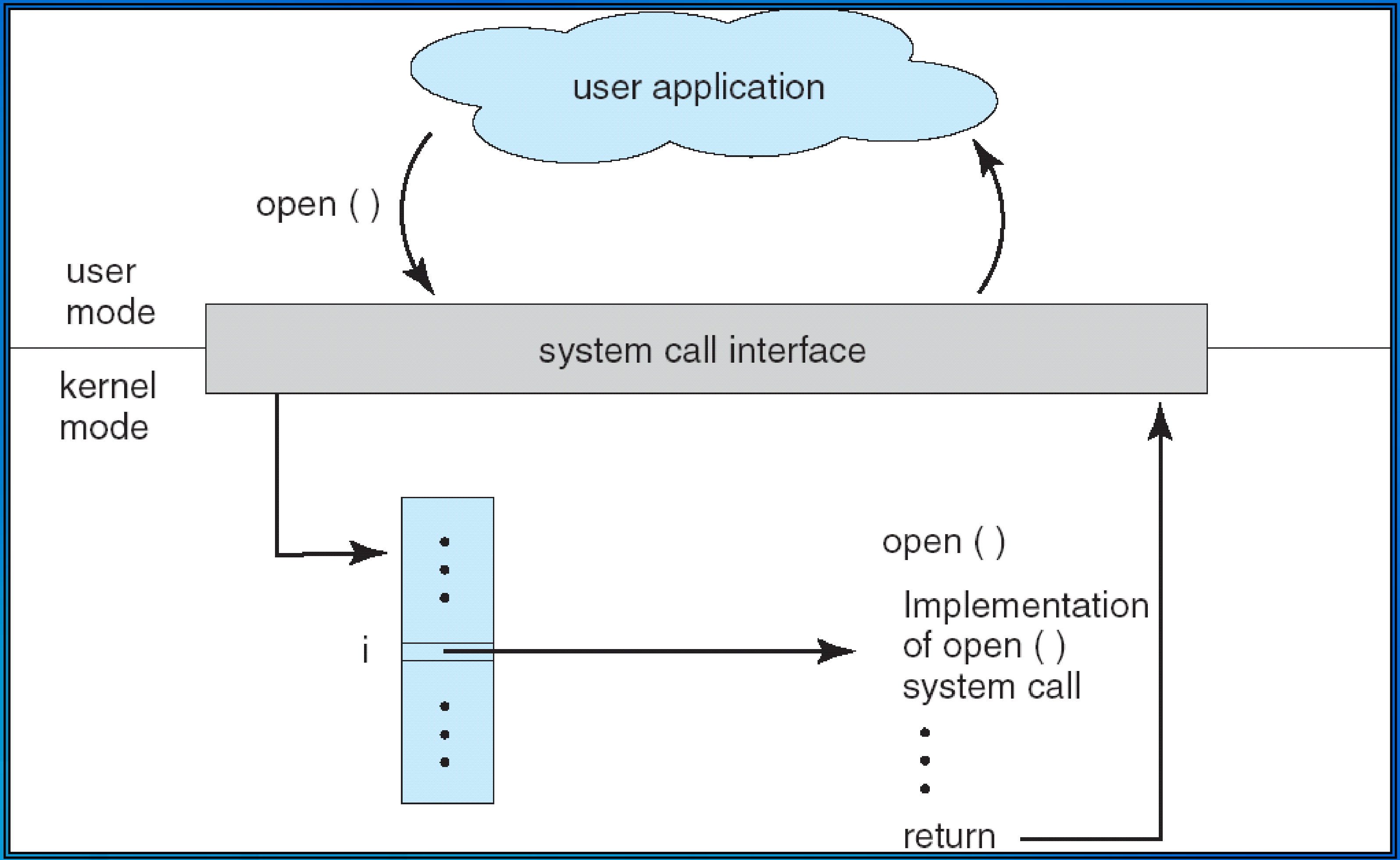


- A description of the parameters passed to ReadFile()
  - HANDLE file—the file to be read
  - LPVOID buffer—a buffer where the data will be read into and written from
  - DWORD bytesToRead—the number of bytes to be read into the buffer
  - LPDWORD bytesRead—the number of bytes read during the last read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

System Call Implementation

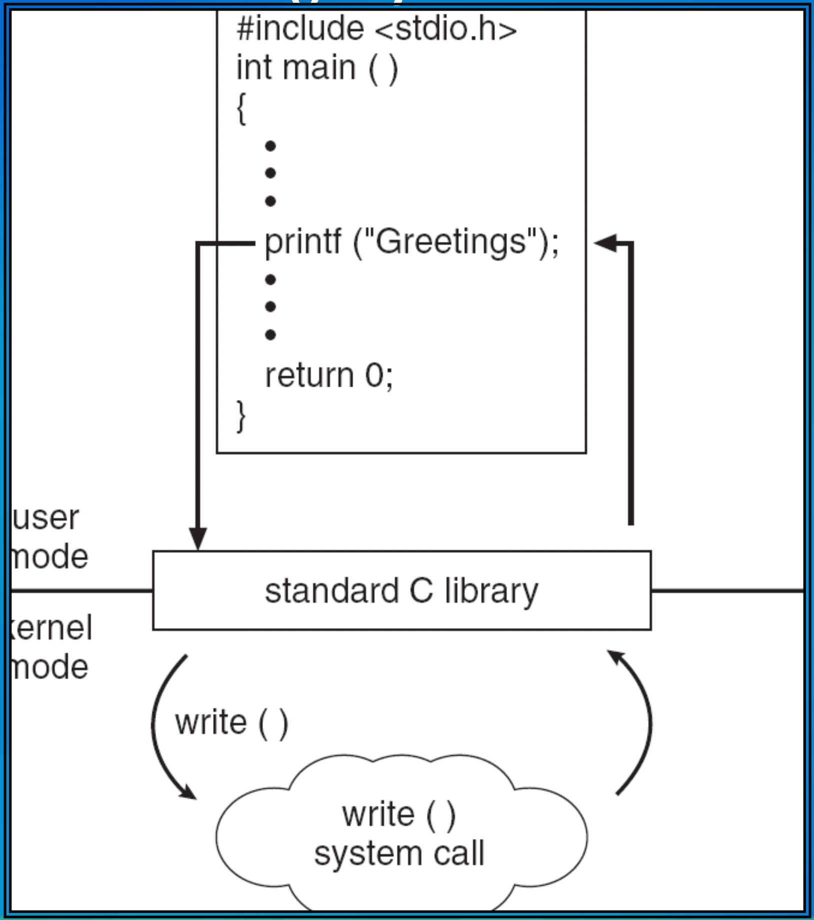
- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about 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)

API – System Call – OS Relationship



Standard C Library Example

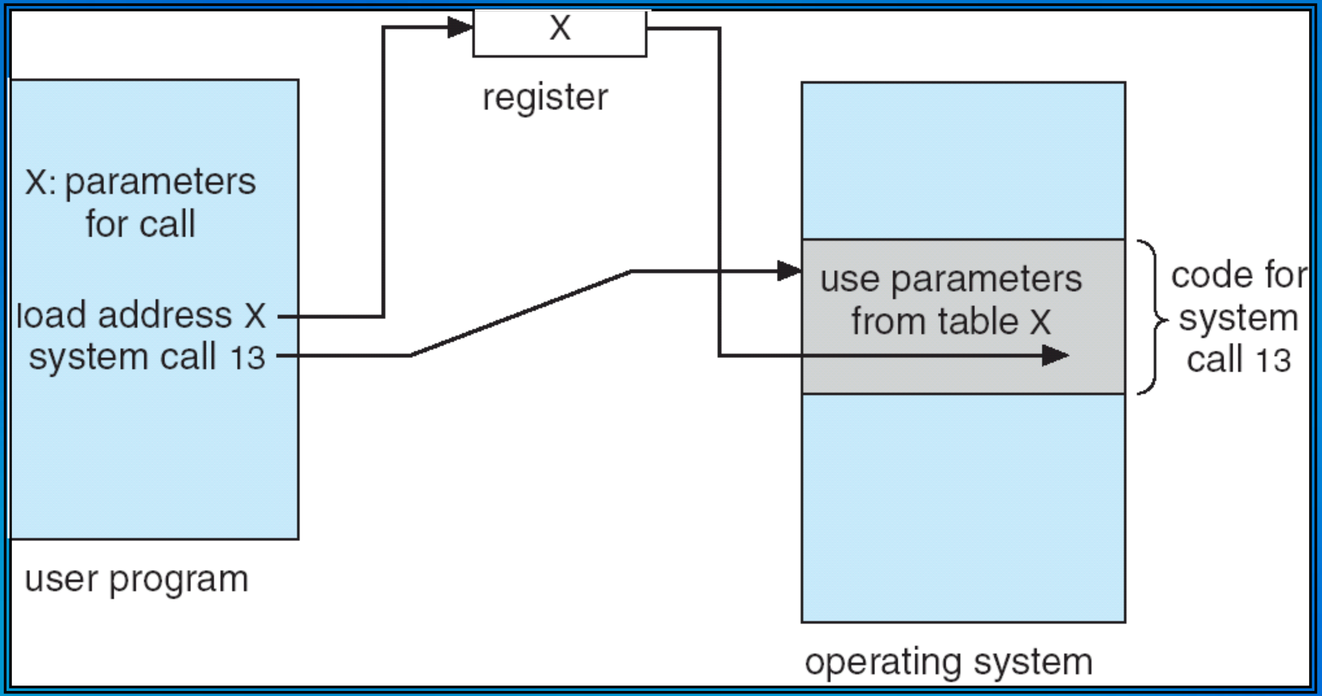
- C program invoking printf() library call, which calls write() system call



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, 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

Parameter Passing via Table



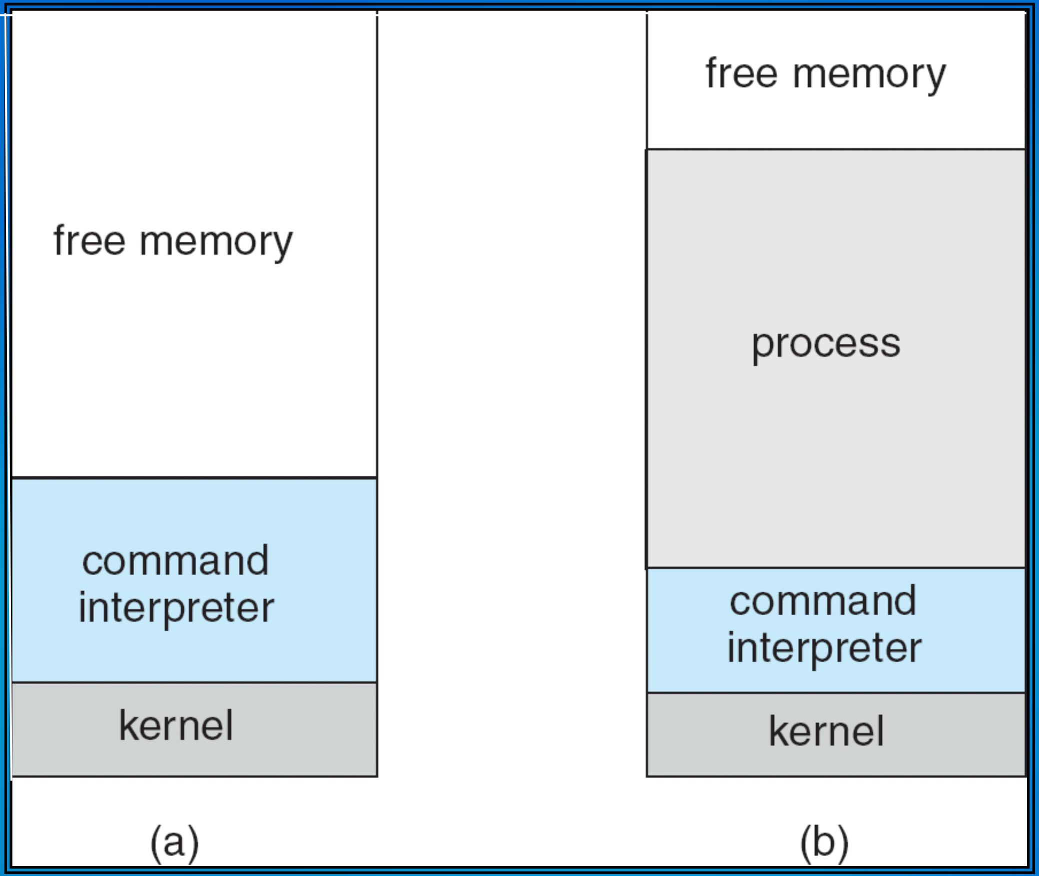
Types of System Calls

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

Examples of Windows and Unix System Calls

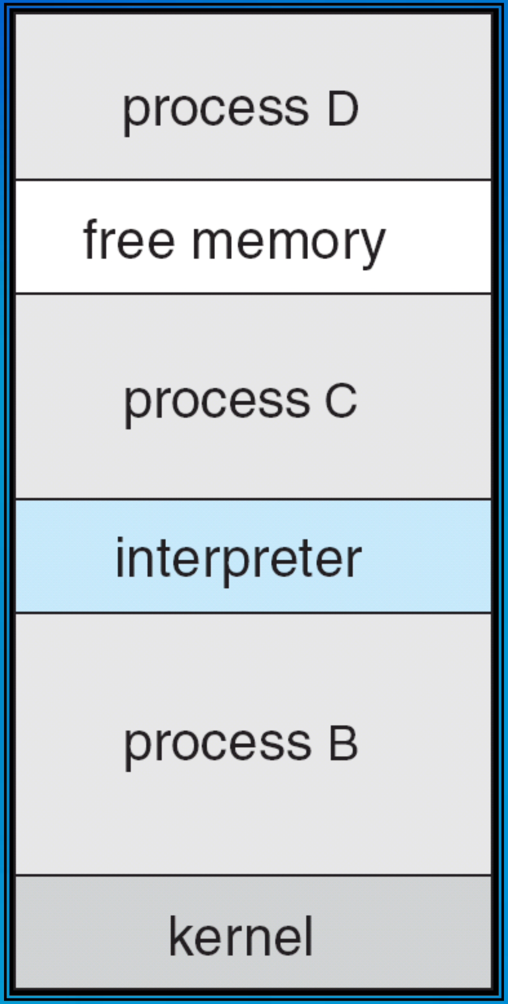
	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information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

MS-DOS execution



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

FreeBSD Running Multiple Programs



System Programs

- System programs provide a convenient environment for program development and execution. The can be divided into:
  - File manipulation
  - Status information
  - File modification
  - Programming language support
  - Program loading and execution
  - Communications
  - Application programs
- Most users' view of the operation system is defined by system programs, not the actual system calls

System Programs

- Provide a convenient environment for program development and execution
  - Some of them are simply user interfaces to system calls; others are considerably more complex
- File management - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- Status information
  - Some ask the system for info - date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information
  - Typically, these programs format and print the output to the terminal or other output devices
  - Some systems implement a registry - used to store and retrieve configuration information

System Programs (cont'd)

- File modification
  - Text editors to create and modify files
  - Special commands to search contents of files or perform transformations of the text
- Programming-language support - Compilers, assemblers, debuggers and interpreters sometimes provided
- Program loading and execution- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- 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

Operating System Design and Implementation

- Design and Implementation of OS not "solvable", but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Start by defining goals and specifications
- Affected by choice of hardware, type of system
- *User* goals and *System* goals
  - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

Operating System Design and Implementation (Cont)

- Important principle to separate

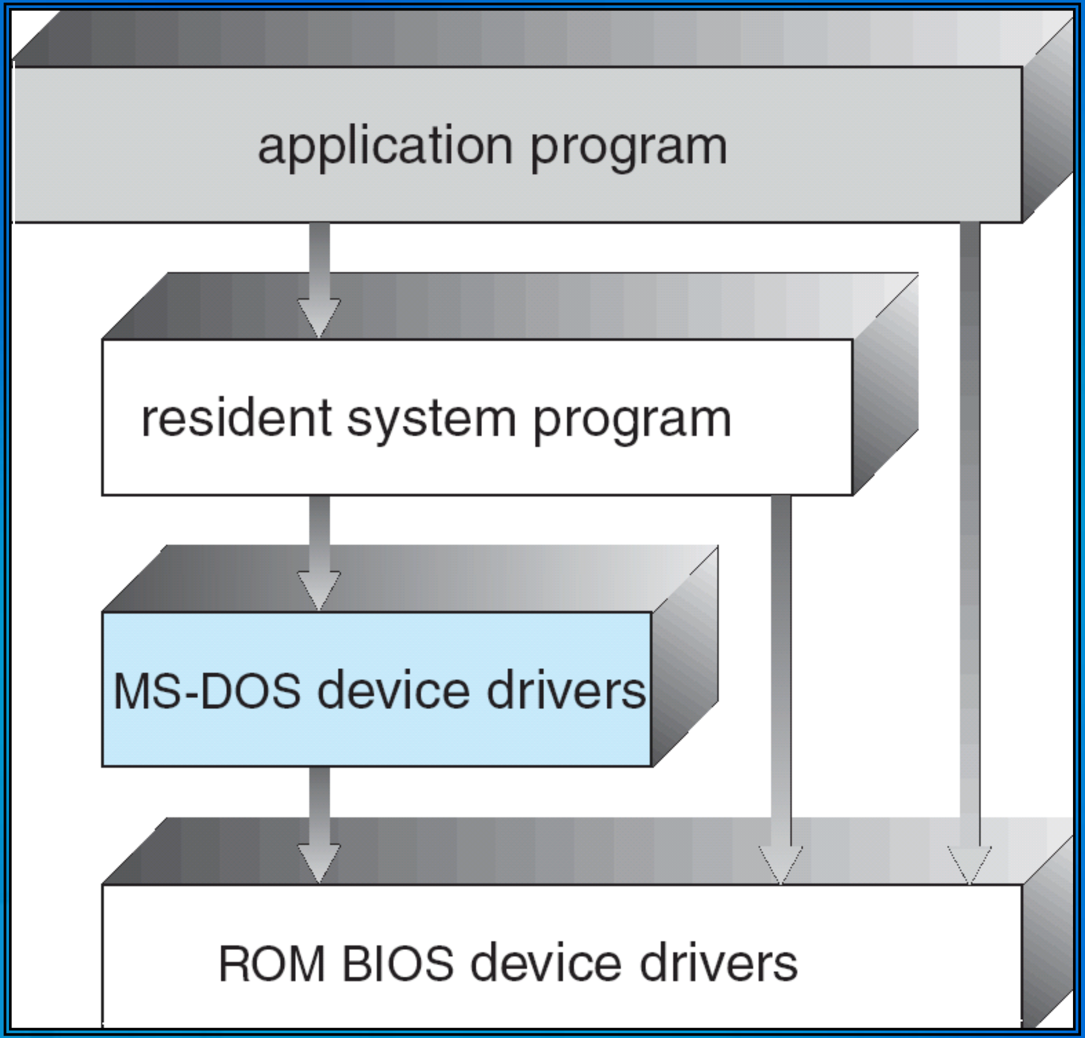
  Policy: What will be done?
  Mechanism: How to do it?

- Mechanisms determine how to do something, policies decide what will be done
  - The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later

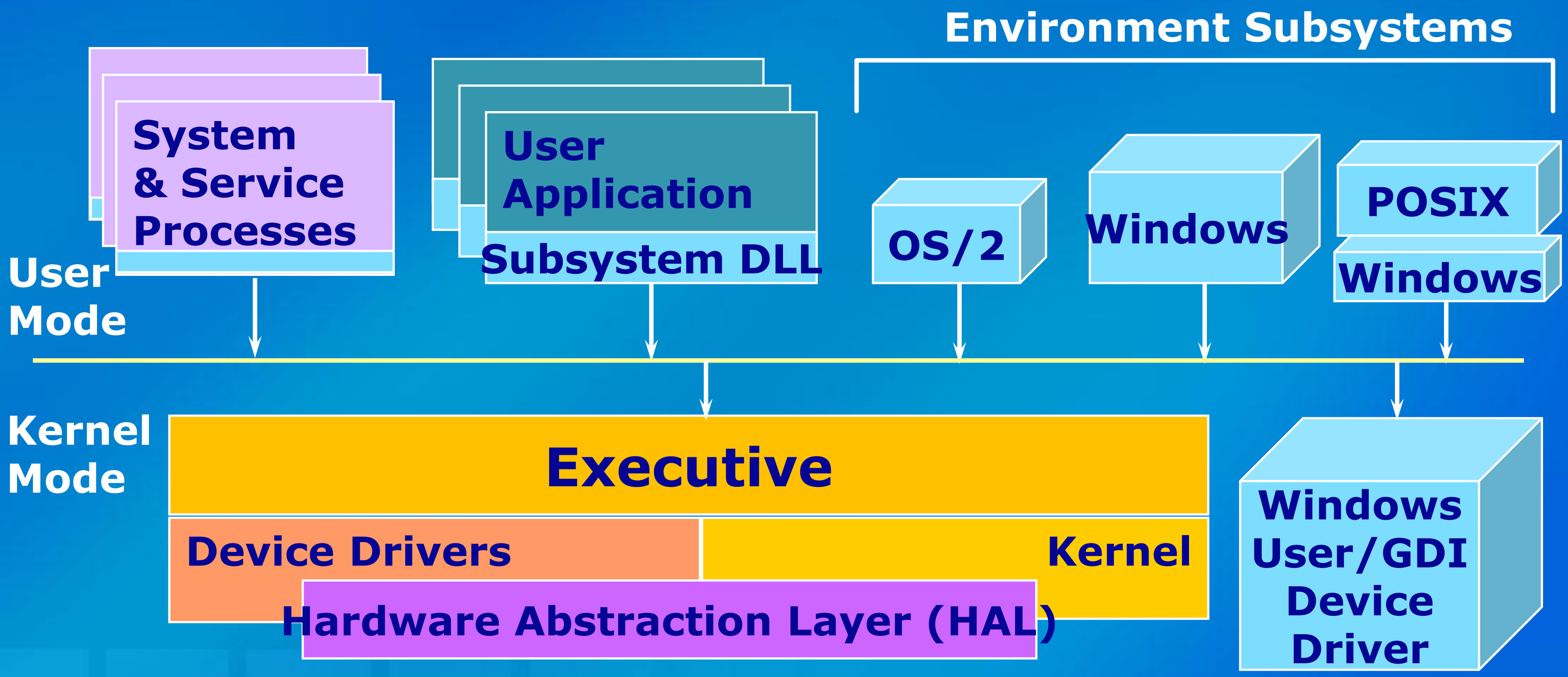
Simple Structure

- MS-DOS – written to provide the most functionality in the least space
  - Not divided into modules
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated

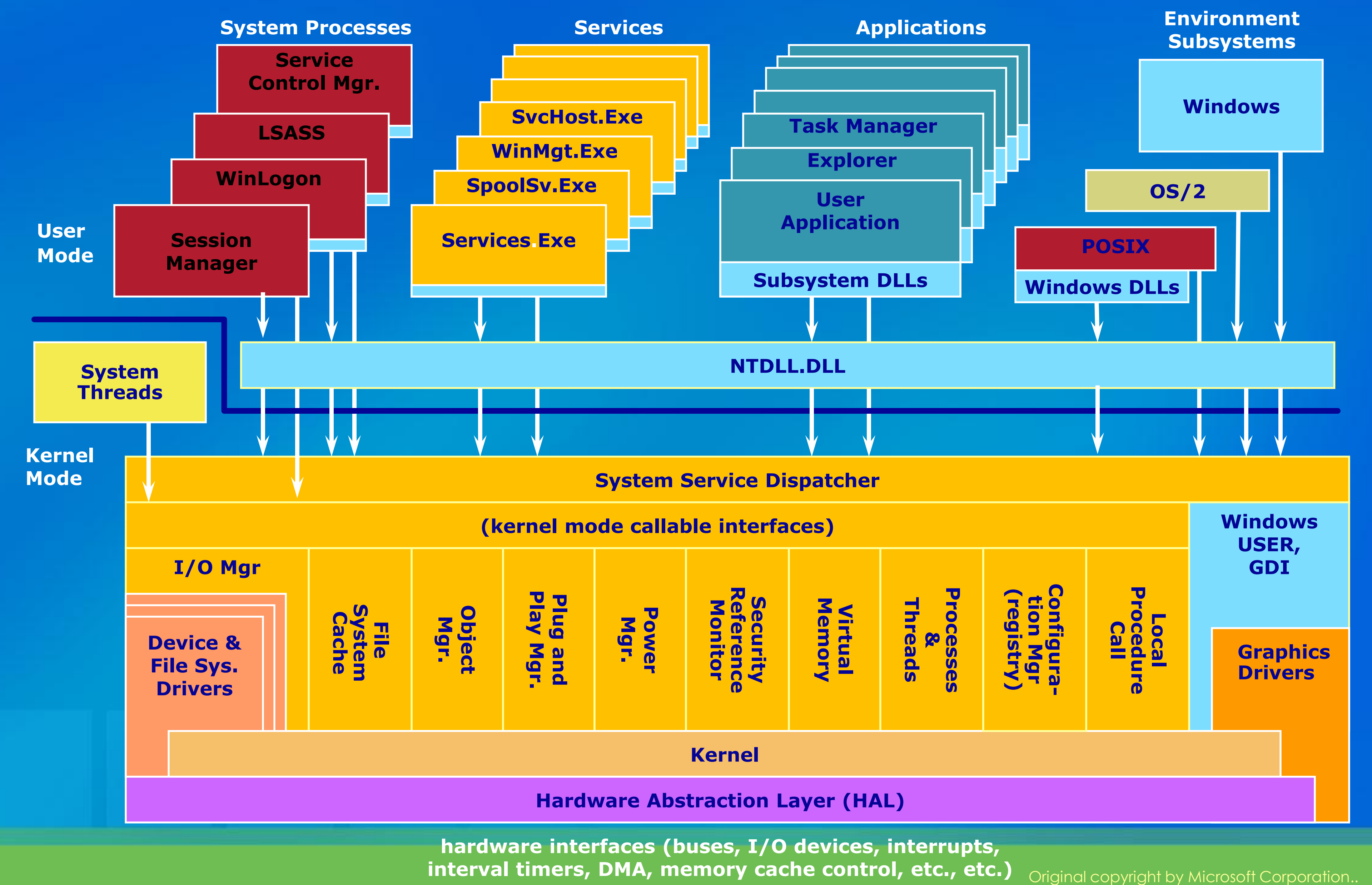
MS-DOS Layer Structure



Key Win NT System Components



Windows Architecture



HAL - Hardware Abstraction Layer

- Responsible for a small part of “hardware abstraction”
  - Components on the motherboard not handled by drivers
    - System timers, Cache coherency, and flushing
    - SMP support, Hardware interrupt priorities
- Subroutine library for the kernel & device drivers
  - Isolates Kernel and Executive from platform-specific details
  - Presents uniform model of I/O hardware interface to drivers
- Reduced role as of Windows 2000
  - Bus support moved to bus drivers
  - Majority of HALs are vendor-independent
- HAL also implements some functions that appear to be in the Executive and Kernel
- Selected at installation time
- HAL kit
  - Special kit only for vendors that must write custom HALs (requires approval from Microsoft)
  - see http://www.microsoft.com/whdc/ddk/HALkit/default.mspx

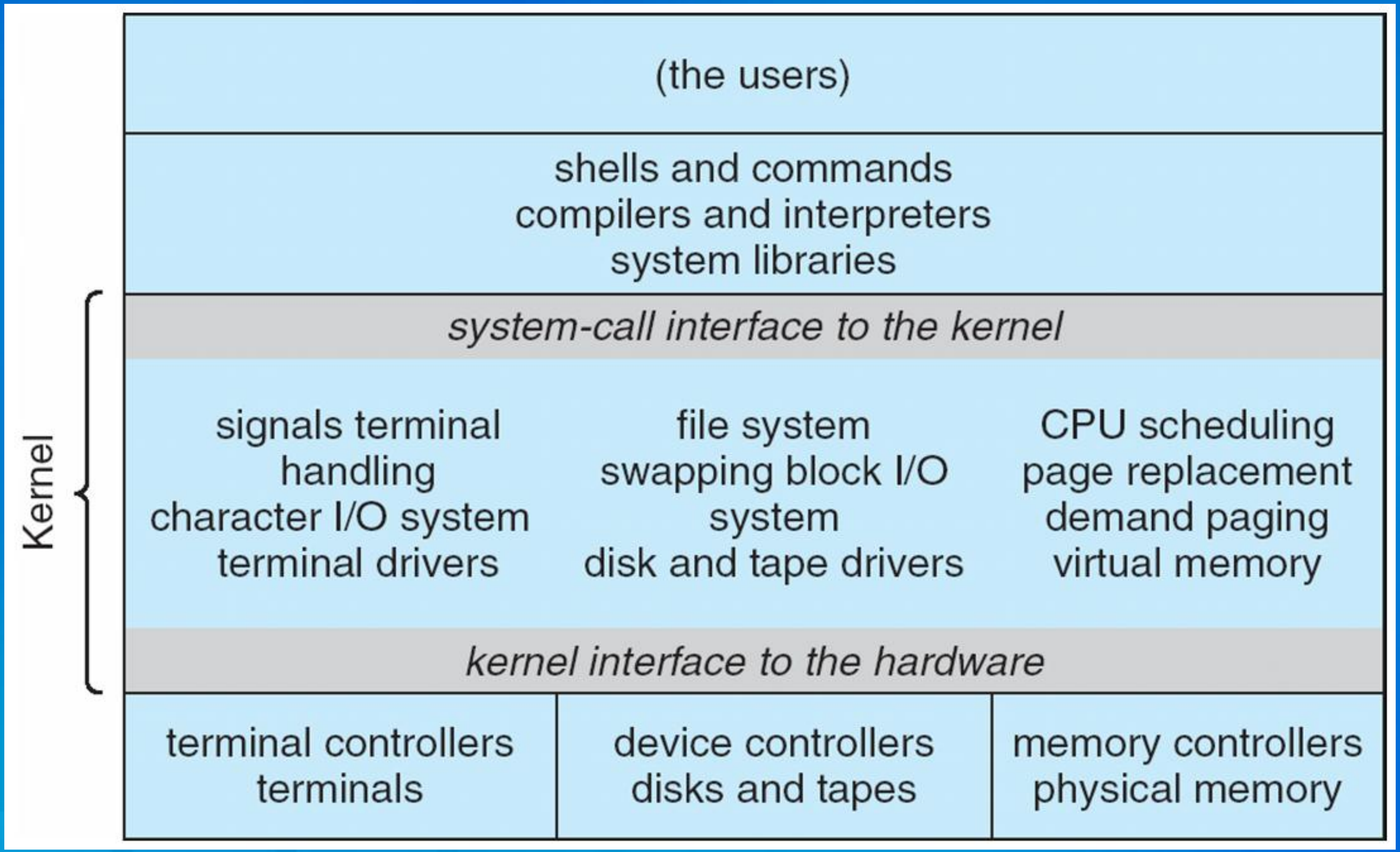
Sample HAL routines:

HalGetInterruptVector
HalGetAdapter
WRITE_PORT_UCHAR

Device Drivers

- *A* device driver is a specific type of computer software developed to allow interaction with hardware devices. Typically this constitutes an interface for communicating with the device, through the specific computer bus or communications subsystem that the hardware is connected to, providing commands to and/or receiving data from the device, and on the other end, the requisite interfaces to the operating system and software applications. It is a specialized hardware-dependent computer program which is also operating system specific that enables another program, typically an operating system or applications software package or computer program running under the operating system kernel, to interact transparently with a hardware device, and usually provides the requisite interrupt handling necessary for any necessary asynchronous time-dependent hardware interfacing needs.
- The key design goal of device drivers is abstraction. Every model of hardware (even within the same class of device) is different. Newer models also are released by manufacturers that provide more reliable or better performance and these newer models are often controlled differently. Computers and their operating systems cannot be expected to know how to control every device, both now and in the future. To solve this problem, operating systems essentially dictate how every type of device should be controlled. The function of the device driver is then to translate these operating system mandated function calls into device specific calls. In theory a new device, which is controlled in a new manner, should function correctly if a suitable driver is available. This new driver will ensure that the device appears to operate as usual from the operating system's point of view.

Traditional UNIX System Structure



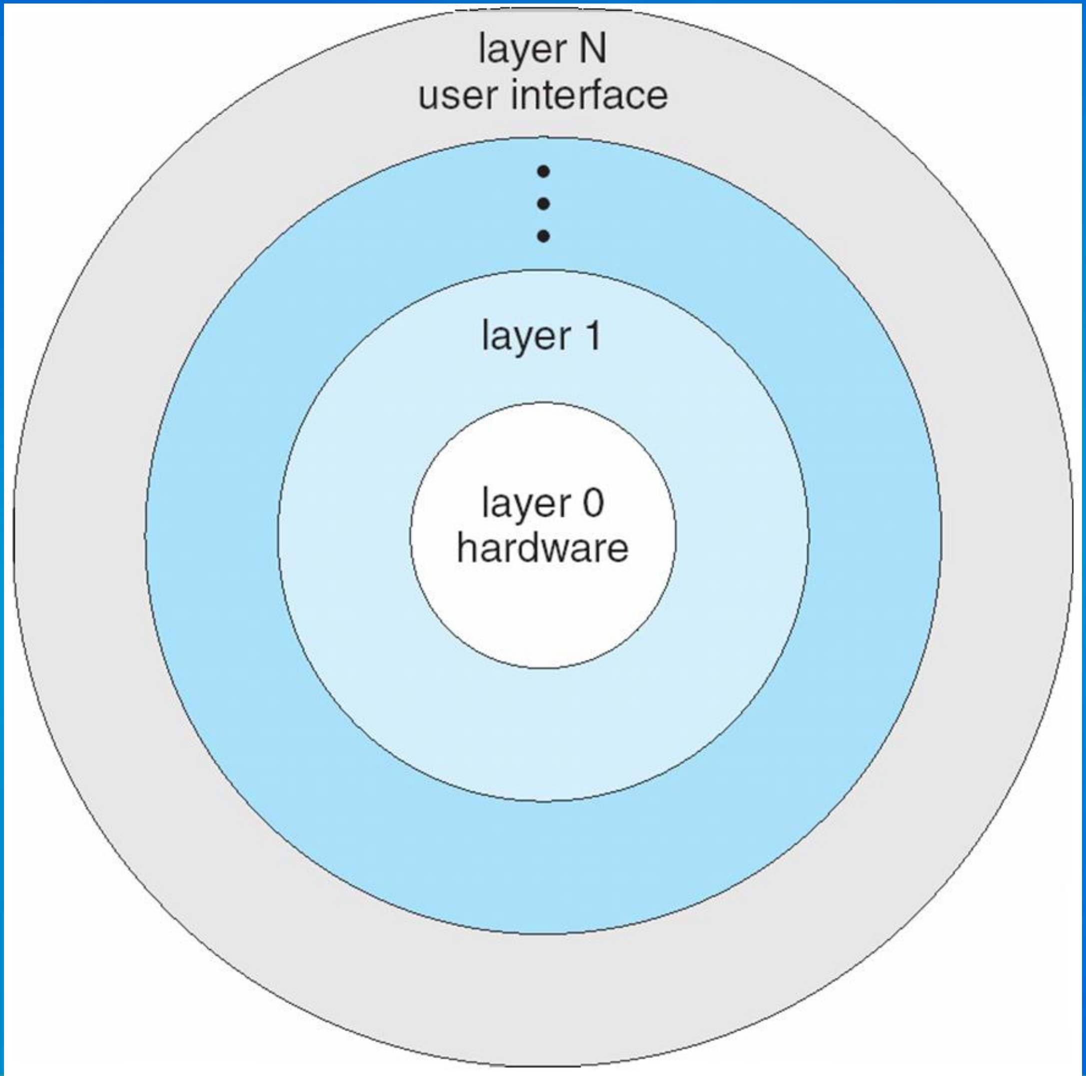
UNIX

- 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

Layered Approach

- 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

Layered Operating System



Microkernel System Structure

- Leave as less functions in the kernel, move rest into "*user*" space
  - Low level memory management, IPC, I/O & Interrupt management
- Communication takes place between user modules using message passing
- Benefits:
  - Easier to extend a microkernel
  - Flexible
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode, modular approach)
  - More secure (uniform Interfaces)
  - Distributed System Support
  - Object Oriented Operating System
- Detriments:
  - Performance overhead of user space to kernel space communication

Microkernel Architecture

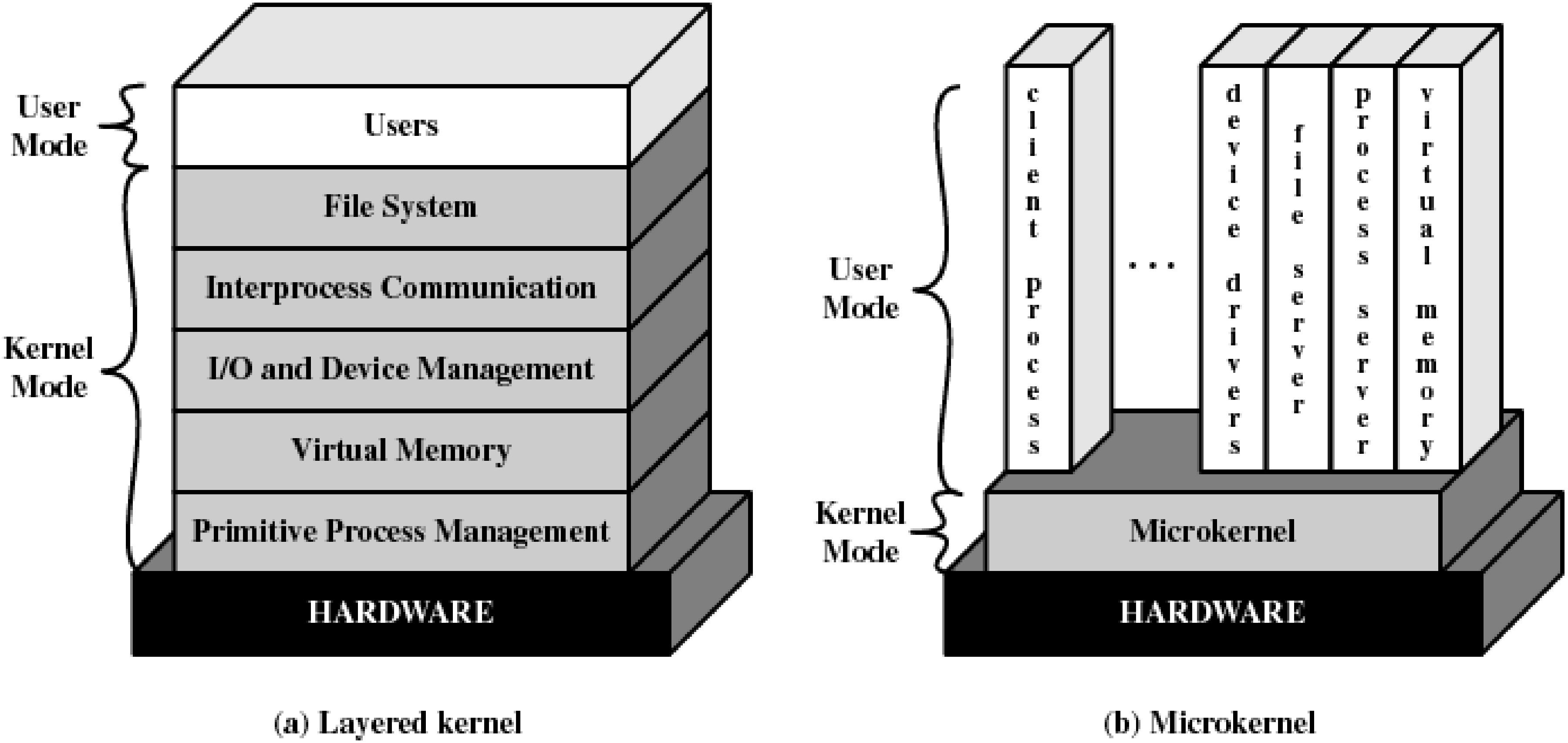
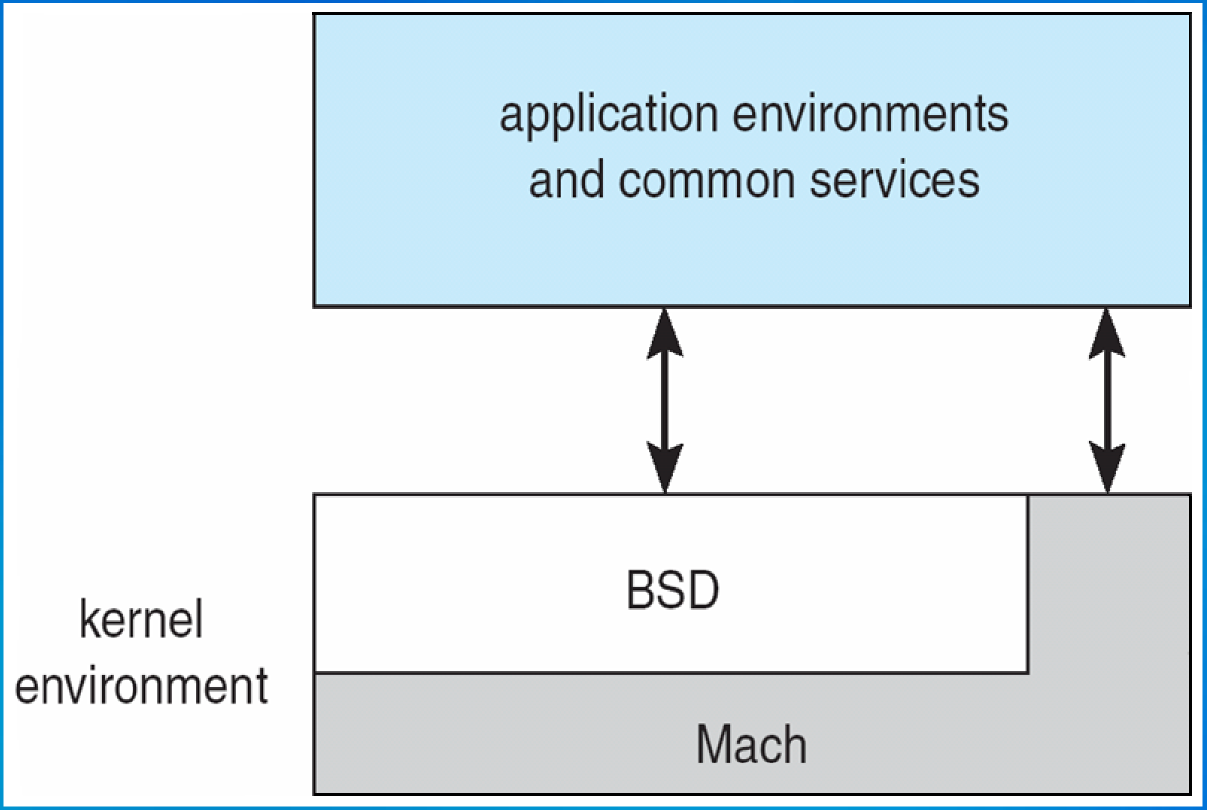


Figure 4.10 Kernel Architecture

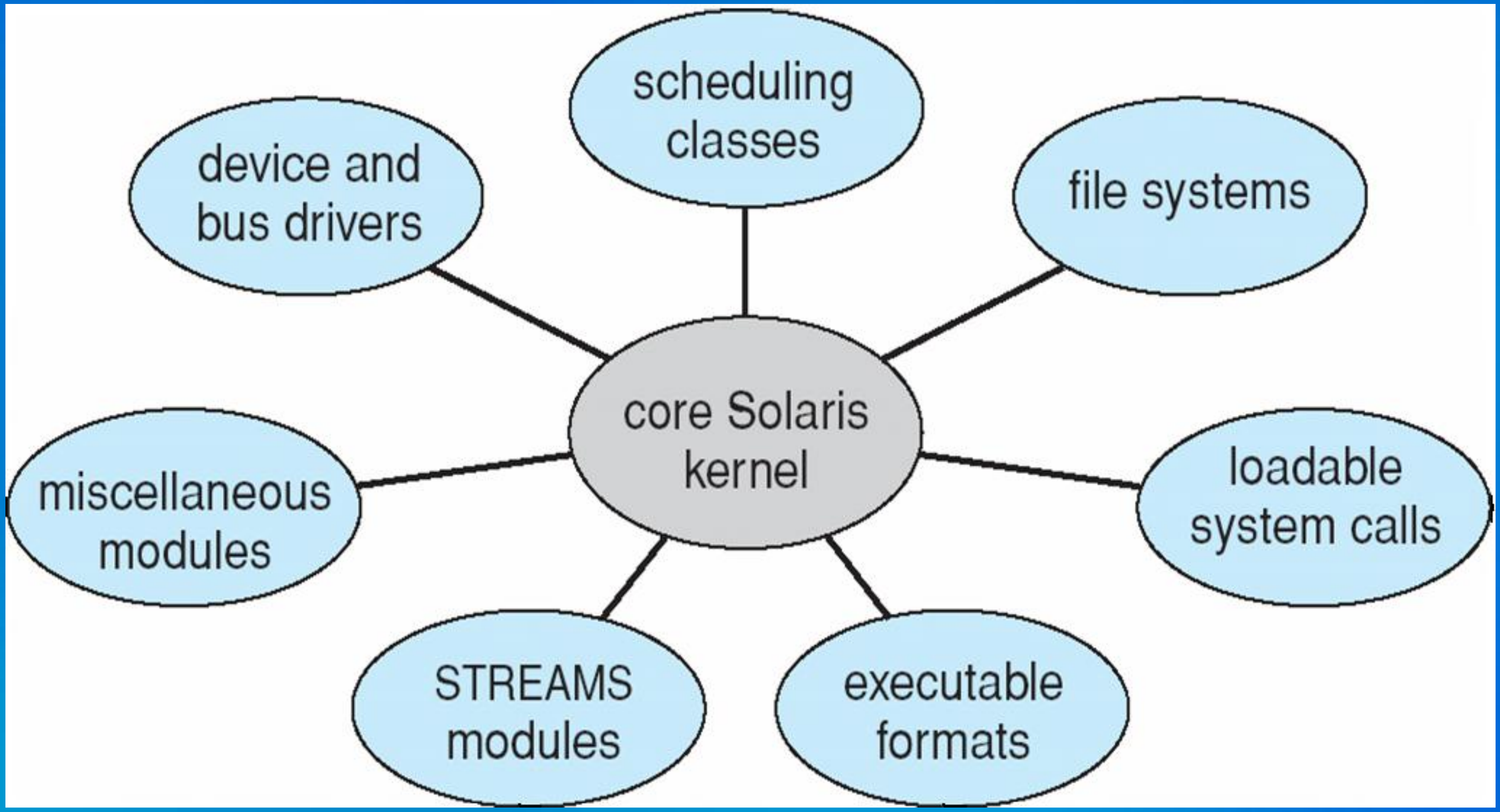
Mac OS X Structure



Modules

- Most modern operating systems implement kernel modules
  - 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

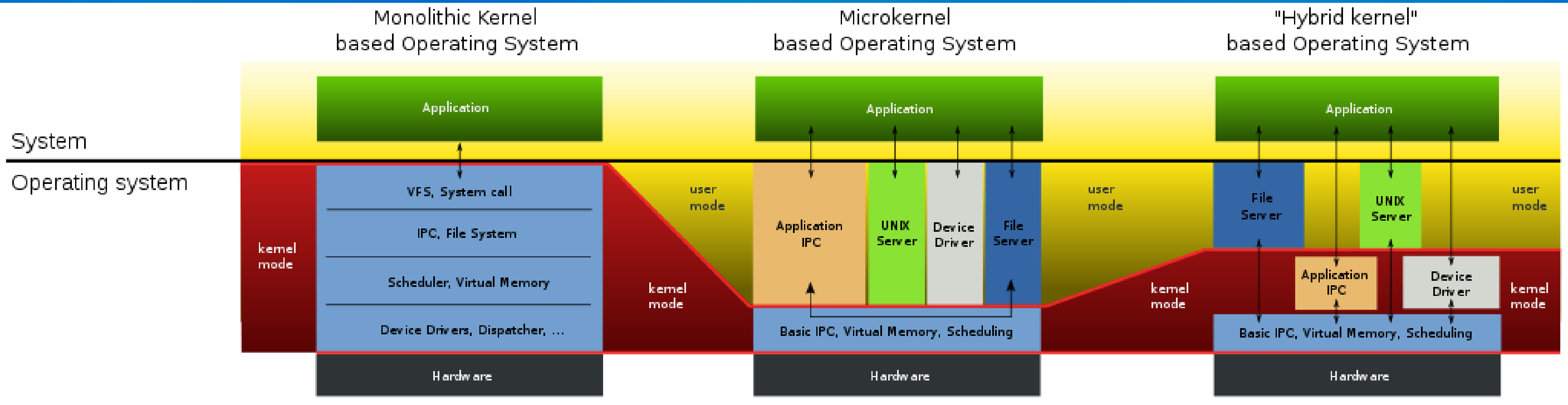
Solaris Modular Approach



Microkernel OS?

- Is Windows a microkernel-based OS?
  - No – not using the academic definition (OS components and drivers run in their own private address spaces, layered on a primitive microkernel)
  - All kernel components live in a common shared address space
    - Therefore no protection between OS and drivers
- Why not pure microkernel?
  - Performance – separate address spaces would mean context switching to call basic OS services
  - Most other commercial OSs (Unix, Linux, VMS etc.) have the same design
- But it does have some attributes of a microkernel OS
  - OS personalities running in user space as separate processes
  - Kernel-mode components don't reach into one another's data structures
    - Use formal interfaces to pass parameters and access and/or modify data structures
  - Therefore the term "modified microkernel"

Kernel Architecture Comparison



Assignment 1:

Q1. What about other OS architecture? Comment.

1. Linux
2. Unix
3. Mac
4. Solaris
5. Minux
6. OS/2
7. Google Chrome
8. Any other OS you know..

Due Date: Before Next Class

Hand Written

Assignment 1: (Cont.)

Q2. What is Exokernel architecture?

Q3. What is Hypervisor Technology? Compare Hypervisor with Virtual machine.

Due Date: Before Next Class

Hand Written

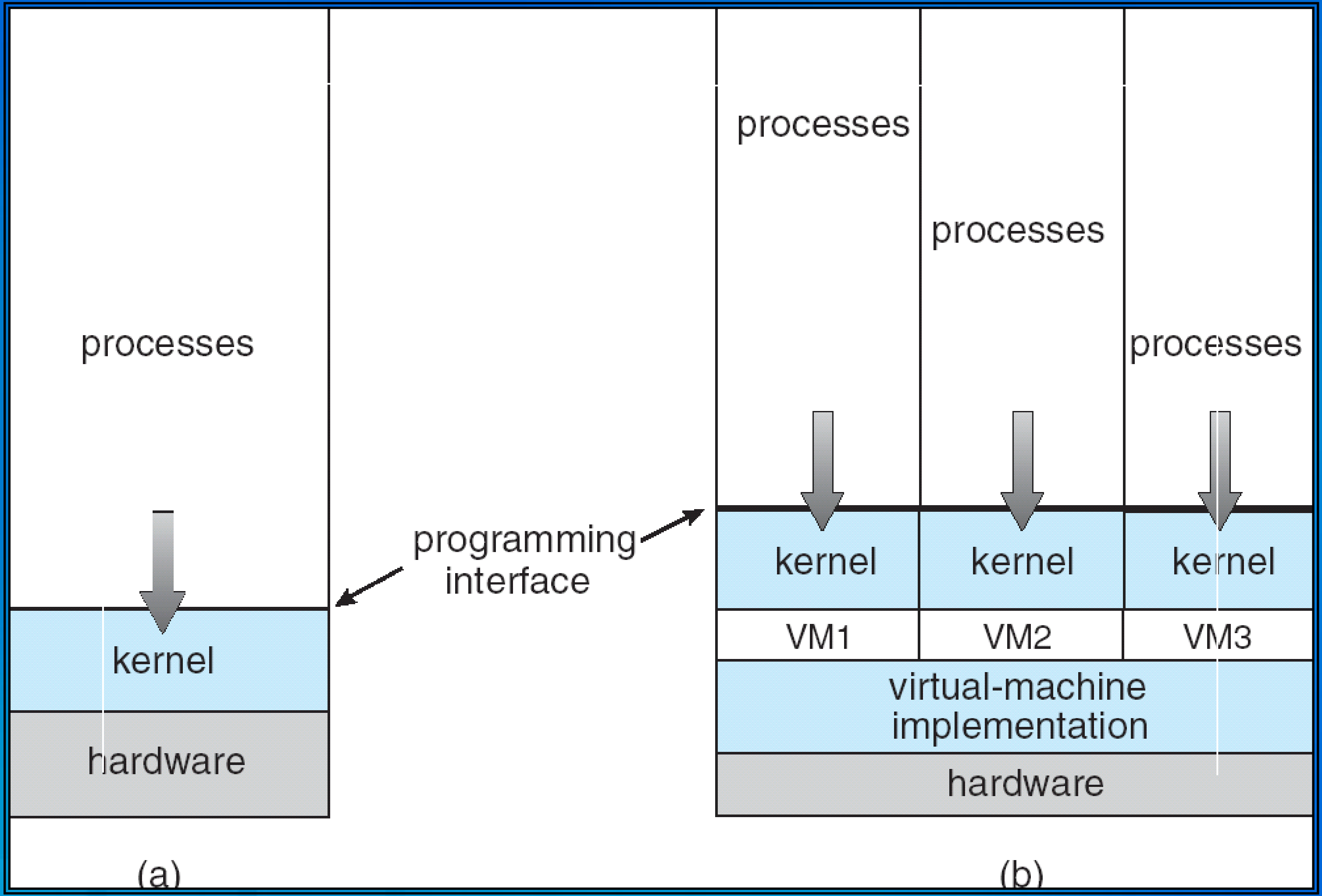
Virtual Machines

- A virtual machine takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware
- 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

Virtual Machines History and Benefits

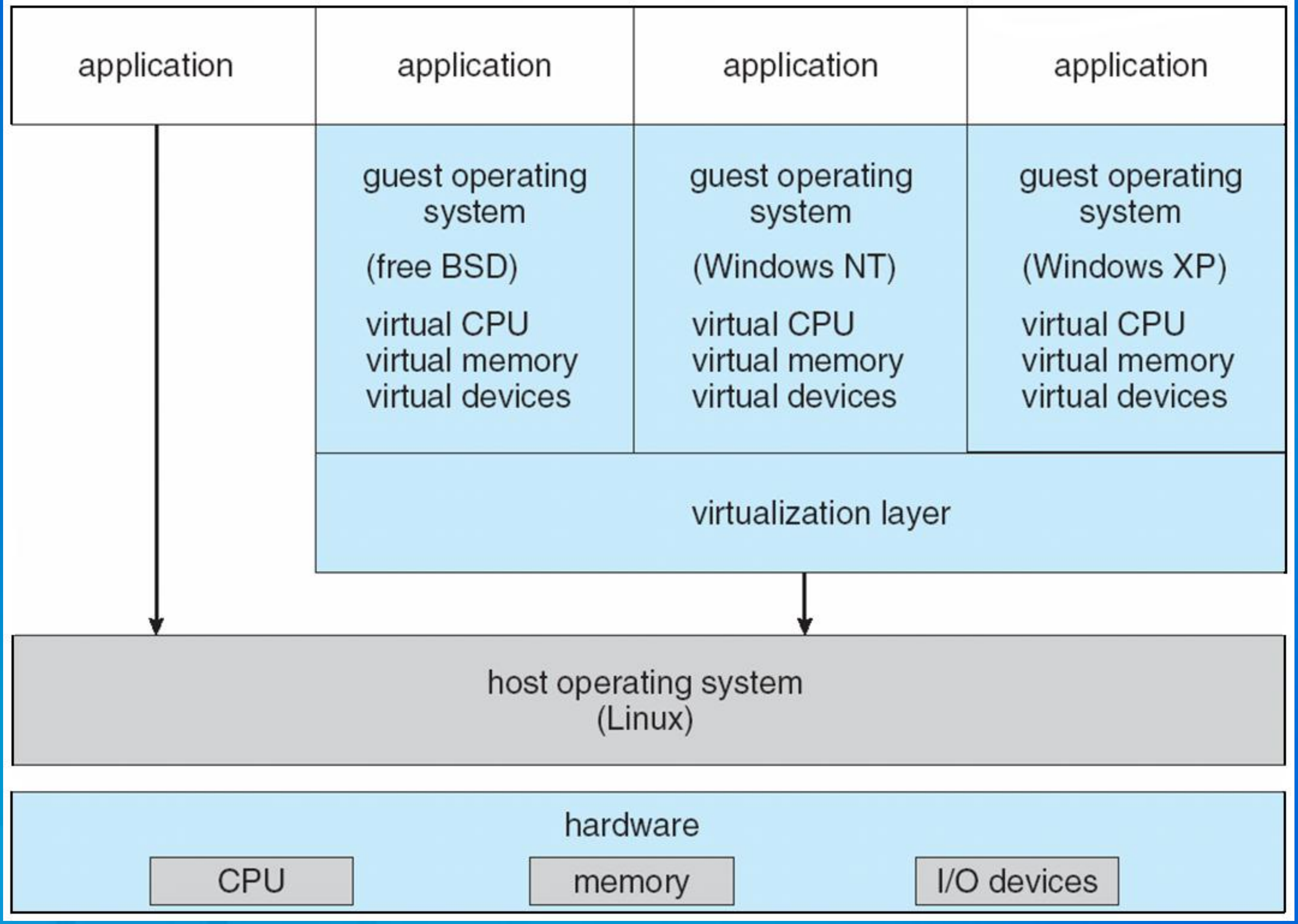
- First appeared commercially in IBM mainframes in 1972
- Fundamentally, multiple execution environments (different operating systems) can share the same hardware
- Protect from each other
- Some sharing of file can be permitted, controlled
- Commutate with each other, other physical systems via networking
- Useful for development, testing
- Consolidation of many low-resource use systems onto fewer busier systems
- “Open Virtual Machine Format”, standard format of virtual machines, allows a VM to run within many different virtual machine (host) platforms

Virtual Machines (Cont)

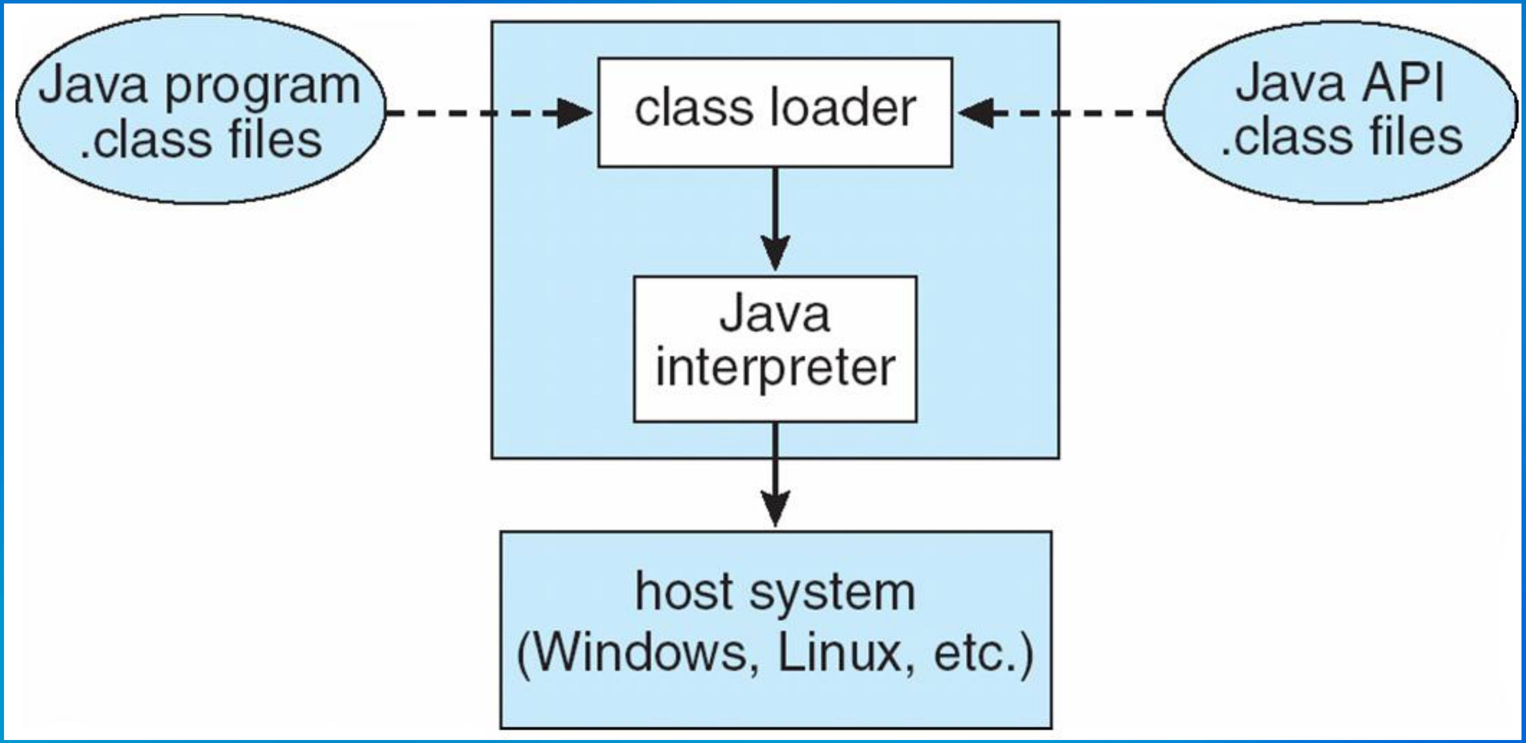


(a) Nonvirtual machine (b) virtual machine

VMware Architecture



The Java Virtual Machine



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
- Beyond crashes, performance tuning can optimize system performance
- Kernighan's Law: "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

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 program obtains information concerning the specific configuration of the hardware system
- *Booting* – starting a computer by loading the kernel
- *Bootstrap program* – code stored in ROM that is able to locate the kernel, load it into memory, and start its execution

System Boot

- Operating system must be made available to hardware so hardware can start it
  - Small piece of code – **bootstrap loader**, locates the kernel, loads it into memory, and starts it
  - Sometimes two-step process where **boot block** at fixed location loads bootstrap loader
  - When power initialized on system, execution starts at a fixed memory location
    - Firmware used to hold initial boot code